NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Using scripting languages for hardware/software co-design

Evangelos Logaras

ATHENS

DECEMBER 2015

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Μέθοδοι συσχεδίασης υλικού/λογισμικού με χρήση scripting γλωσσών

Ευάγγελος Λογαράς

ΑΘΗΝΑ

ΔΕΚΕΜΒΡΙΟΣ 2015

PhD THESIS

Using scripting languages for hardware/software co-design

Evangelos Logaras

Examination date: 4/12/2015

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Μέθοδοι συσχεδίασης υλικού/λογισμικού με χρήση scripting γλωσσών

Ευάγγελος Λογαράς

Ημερομηνία εξέτασης: 4/12/2015

Abstract

Multiprocessor embedded System on Chips (SoCs) include at least one programmable processor Intellectual Property (IP) core and several other hardware blocks, attached as domain-specific co-processors or peripheral units to the processor's data and control bus. Such a complex system architecture can take advantage of the reconfigurable, parallel processing and low power consumption features of modern FPGA devices. However these benefits cannot be fully delivered due to the lack of mature hardware/software co-design and rapid prototyping tools of embedded multi-processor SoCs.

In this doctoral dissertation we present a new methodology for the hw/sw co-design of multiprocessor embedded SoCs developed by exploiting the strengths of the popular Python scripting language. We exploit the features of Python to rapidly prototype and validate processor-centric SoC designs for Field Programmable Gate Arrays (FPGA). Specifically we developed methods to: (a) describe hardware blocks in Python and automatically generate synthesizable VHDL code, (b) describe in Python and simulate embedded systems both at the algorithmic/functional level as well as at the Register Transfer level, and automatically generate digital waveforms recording the results of the system's simulation that remains cycle-accurate and bit-true, (c) integrate into the design flow software development tools for programming in C the microprocessor core, and (d) generate scripts (Tcl) to ease integration of a complete hw/sw design with FPGA implementation tools that use logic synthesis to construct a physical implementation of the multi-processor SoC.

The above described hw/sw co-design and verification functionalities were implemented in the developed System Python (SysPy) tool that targets the prototyping of processor-centric embedded SoCs for FPGAs. In addition we have developed an Application Programming Interface (API) which enables easy data transfer between a SoC design, running in an FPGA device, and a host PC. The user can utilize Python software running on the PC and interact with C software running on the processor IP core of the SoC and in this way control data processing and storage in the FPGA.

Three sophisticated SoC's have been designed and implemented and are presented in detail as SoC design cases to demonstrate and assess the new co-design and co-simulation features of SysPy along with the supported design methodology. All three designs use a processor IP core as the main programmable system controller along with domain-specific hardware accelerator units designed for: a) image processing (edge detection), b) audio processing (music genre classification), and c) stochastic simulation of large-scale biochemical reaction networks (systems biology). These multiprocessor SoC design cases demonstrate the evolution of the design methodology and each one of them highlights different features of it.

We believe that with our methodology, developed using Python, we contribute towards the development of mature tools for the hw/sw co-design and rapid prototyping of FPGA-based embedded multiprocessor SoCs. To get useful feedback from end users community and contribute to the hw/sw co-design efforts we provide SysPy as an open source tool through GitHub, which is the largest online code repository.
Subject Area: *Digital Design, Embedded Systems*

Keywords: *Python, Processor-centric SoCs, hw/sw co-design, VHDL, FPGA, SysPy.*

Περίληψη

Τα Ενσωματωμένα Συστήματα σε Ψηφίδα υλικού (embedded Systems on Chip - SoC) περιέχουν τουλάχιστον έναν προγραμματιζόμενο επεξεργαστή αλλά και διάφορες μονάδες (IP cores) που διασυνδέονται στους διαύλους ελέγχου και δεδομένων του ως περιφερειακά ή συνεπεξεργαστές ειδικού σκοπού. Ένας τέτοιος τύπος σύνθετης ψηφιακής αρχιτεκτονικής μπορεί να αξιοποιήσει τις δυνατότητες επαναπρογραμματισμού (reconfiguration) των μονάδων FPGA για να επιτύχει υψηλές επιδόσεις και χαμηλή κατανάλωση ενέργειας. Τις προοπτικές αυτές όμως περιορίζει η έλλειψη εργαλείων συσχεδίασης υλικού/λογισμικού για τη γρήγορη πρωτοτυποποίηση (rapid prototyping) ενσωματωμένων πολυεπεξεργαστικών SoCs.

Στην παρούσα διδακτορική διατριβή παρουσιάζουμε μεθοδολογία συσχεδίασης υλικού/λογισμικού για ενσωματωμένα πολυεπεξεργαστικά SoCs που υλοποιείται με τη χρήση της δημοφιλούς scripting γλώσσας προγραμματισμού Python. Αναδεικνύουμε εκείνα τα χαρακτηριστικά της γλώσσας Python που διευκολύνουν τη σχεδίαση ενσωματωμένων SoC με προγραμματιζόμενο επεξεργαστή (processor-centric) και την υλοποίηση τους σε μονάδες FPGA. Συγκεκριμένα αναπτύξαμε μεθόδους για: (α) υποστήριξη περιγραφών στοιχείων υλικού σε Python και αυτόματη μετατροπή τους σε VHDL, (β) χρήση περιγραφών Python για την προσομοίωση ενσωματωμένου συστήματος τόσο σε αλγοριθμικό επίπεδο λειτουργικότητας όσο και σε επίπεδο αρχιτεκτονικής RTL (Register Transfer level) και αυτόματη παραγωγή αρχείων ψηφιακών κυματομορφών με τα αποτελέσματα της ακριβούς προσομοίωσης (cycle-accurate και bit-true) του συστήματος. (γ) Υποστήριξη των απαραίτητων λειτουργιών για τον προγραμματισμό του επεξεργαστή σε γλώσσα C και (δ) παραγωγή αρχείων script (Tcl) για την εύκολη συνεργασία με υπάρχοντα εργαλεία λογικής σύνθεσης για τη φυσική υλοποίηση του συστήματος σε FPGA.

Οι παραπάνω λειτουργίες συσχεδίασης και προσομοίωσης υλικού/λογισμικού εντάχθηκαν στο εργαλείο System Python (SysPy) που στοχεύει στην αποδοτική πρωτοτυποποίηση ενσωματωμένων processor-centric SoCs για FPGAs. Επιπλέον αναπτύξαμε διεπαφή προγράμματος (API: Application Programming Interface) για την εύκολη ανταλλαγή δεδομένων μεταξύ του ενσωματωμένου επεξεργαστή στο FPGA και διασυνδεδεμένων Η/Υ. Με αυτή ο χρήστης μπορεί να αλληλεπιδρά με αντίστοιχο πρόγραμμα σε C που εκτελεί ο ενσωματωμένος επεξεργαστής, προκειμένου να ελέγχει προγραμματιστικά την επεξεργασία και καταχώρηση δεδομένων στο υλικό.

Για τον έλεγχο των δυνατοτήτων της μεθοδολογίας σχεδιάσαμε και υλοποιήσαμε με τη χρήση του SysPy τρία ενσωματωμένα πολυεπεξεργαστικά SoCs, τα οποία αναδεικνύουν τις νέες δυνατότητες συσχεδίασης και προσομοίωσης. Και τα τρία αυτά SoCs χρησιμοποιούν πυρήνα μικροεπεξεργαστή ως κύριο ελεγκτή του συστήματος αλλά και ειδικές μονάδες υλικού που σχεδιάστηκαν για την: α) επεξεργασία εικόνων, β) επεξεργασία αρχείων ήχου και γ) στοχαστική προσομοίωση βιολογικών δικτύων. Η διαδικασία υλοποίησης των τριών πολυεπεξεργαστικών SoCs έγινε στα πλαίσια της εξέλιξης και βελτιστοποίησης του ίδιου του εργαλείου, ενώ κάθε σχέδιο χρησιμοποιεί και αναδεικνύει συγκεκριμένα του χαρακτηριστικά.

Πιστεύουμε ότι η μεθοδολογία σχεδίασης που αναπτύχθηκε με χρήση της Python συνεισφέρει σημαντικά προς την κατεύθυνση της συσχεδίασης υλικού/λογισμικού και πρωτοτυποποίησης για ενσωματωμένα συστήματα σε ψηφίδα υλικού, τομέα όπου σήμερα δεν υπάρχουν ώριμα διαθέσιμα εργαλεία. Επιπλέον το πρωτότυπο εργαλείο SysPy, αποτέλεσμα αυτής της έρευνας, παρέχεται ελεύθερα μέσω του GitHub που αποτελεί την μεγαλύτερη διαδικτυακή πλατφόρμα παροχής λογισμικού ανοιχτού κώδικα (open source) προκειμένου να λάβουμε χρήσιμες πληροφορίες από τους τελικούς χρήστες για την βελτίωση των λειτουργιών και της χρηστικότητάς του.

Θεματική περιοχή: *Ψηφιακή Σχεδίαση, Ενσωματωμένα Συστήματα*

Keywords: *Python, πυρήνες επεξεργαστών, συσχεδίαση υλικού/λογισμικού, VHDL, FPGA, SysPy.*

Acknowledgements

First of all I would like to express my gratitude to my advisor Associate Professor Elias Manolakos for his support during my studies. I am really grateful that he gave me the opportunity to pursue my Ph.D. thesis by becoming part of his group. His deep knowledge in the field of embedded systems design gave me the opportunity to improve my technical skills and explore new ways and methods in digital hardware design. His advice helped me take important decisions in my career and under his guidance I learned how to use existing knowledge to expand and promote my ideas and in general how to interact in an academic environment.

I would also like to thank Professor Antonios Paschalis and Professor Angela Arapoyanni for their constant support and advice during my studies. I am also grateful to Professor George Tzanakos for his guidance and for the way he taught me to conduct research in his laboratory. His ideas and way of thinking will follow me in my professional career.

I also want to thank all my colleagues, officemates and friends who supported me during my studies. I express my gratitude to my good officemates and friends Symeon Chouvardas, Dimitrios Manatakis and Ioannis Stamoulias for the time we spent in the office working together and talking about science and the way to move on with our lives. I want to thank my colleague Orsalia Hazapis for working together, helping me improve my ideas and spending hours over electronic boards. I am grateful to Panagiotis Stamoulis for his support and friendship and for the countless hours we spent together discussing and solving all kinds of problems.

Finally I want thank my parents Elias and Christina, my sister Anna and my wife Rodica for their patience all these years, although sometimes I really feel that they do not have a clue about my research interests and my profession.

List of Publications

Journals:

1. E. Logaras, O. G. Hazapis, and E. S. Manolakos. "Python to accelerate embedded SoC design: A case study for systems biology". ACM Transactions on Embedded Computing Systems, 13(4):84:1-84:25, March 2014.

Conferences:

1. E. Logaras and E. S. Manolakos. "SysPy: using Python for processor-centric SoC design". In Proc. IEEE International Conference on Electronics, Circuits and Systems (ICECS), pages 762-765, 2010.

2. O.G. Hazapis, E. Logaras, and E.S. Manolakos. "A soft IP core generating socs for the efficient stochastic simulation of large Biomolecular Networks using FPGAs". In Proc. IEEE International Conference on Electronics, Circuits and Systems (ICECS), pages 77-80, 2012.

3. E. Logaras, E. Koutsouradis and E.S. Manolakos. "Python facilitates the rapid prototyping and hw/sw verification of processor centric SoCs for FPGAs". submitted to the IEEE International Symposium on Circuits and Systems (ISCAS), October 2015.

Συνοπτική παρουσίαση της Διδακτορικής Διατριβής

1.1 Εισαγωγή

Η δραματική αύξηση της χωρητικότητας των μονάδων προγραμματιζόμενης λογικής Field Programmable Gate Array (FPGA) τα τελευταία χρόνια έχει αυξήσει σημαντικά και την πολυπλοκότητα των συστημάτων που μπορούν να υλοποιηθούν με τη χρήση τους. Η χρήση έτοιμων πυρήνων επεξεργαστών σε ένα σύστημα μπορεί να επιταχύνει σημαντικά τον απαιτούμενο χρόνο σχεδίασης, αλλά πρέπει επίσης η μεθοδολογία σχεδίασης να μπορεί να υποστηρίξει μία τέτοια αρχιτεκτονική. Σε αυτό το κεφάλαιο αναφέρονται τα κίνητρα τα οποία μας οδήγησαν στην ανάπτυξη του SysPy και επίσης η συνεισφορά του εργαλείου και της υποστηριζόμενης μεθοδολογίας στη σχεδίαση System on Chip με πυρήνα επεξεργαστή και χρήση FPGA. Στο τέλος του κεφαλαίου αναφέρουμε συνοπτικά το περιεχόμενο των κεφαλαίων που ακολουθούν.

1.1.1 Βασικοί στόχοι της διατριβής

Ο κύριος στόχος της έρευνας ήταν η ανάπτυξη ενός εργαλείου ψηφιακής σχεδίασης και προσομοίωσης κάνοντας χρήση περιγραφών υψηλού επιπέδου σε γλώσσα Python. Το SysPy στοχεύει ειδικότερα τη σχεδίαση ενσωματωμένων συστημάτων με πυρήνα επεξεργαστή διευκολύνοντας τη συσχεδίαση υλικού/λογισμικού. Τα μοναδικά χαρακτηριστικά της Python σε επίπεδο ανάπτυξης προγραμμάτων script σε περιβάλλον Linux σε συνδυασμό με τις δυνατότητες αντικειμενοστραφούς προγραμματισμού (OOP: Object Oriented Programming) που επίσης παρέχει, μας βοήθησαν να αναπτύξουμε μεθόδους για την:

- δημιουργία μοντέλων υψηλού επιπέδου, π.χ. αριθμητικών μονάδων, στοιχείων μνήμης και λογικών μονάδων, διασύνδεση τους με τη χρήση περιγραφών Python και αυτόματη μετατροπή τους σε συνθέσιμες περιγραφές VHDL συμβατές με υλοποιήσεις σε FPGA.

- ανάπτυξη ενός μέσου το οποίο ελέγχει όλα τα υπόλοιπα εργαλεία που απαιτούνται για την ανάπτυξη υλικού και λογισμικού κατά της διαδικασία σχεδίασης ενός SoC με πυρήνα επεξεργαστή.

- επεξεργασία μεγάλου αριθμών αρχείων που παράγονται κατά τη διαδικασία ψηφιακής σχεδίασης, π.χ. αρχεία αναφορών, αρχεία ψηφιακών κυματομορφών κ.α.

Αντιλαμβανόμαστε ότι η σχεδίαση ενός πολύπλοκου ψηφιακού ενσωματωμένου συστήματος δεν μπορεί να αυτοματοποιηθεί στο σύνολο της, προδιαγράψαμε όμως και αναπτύξαμε ένα λογισμικό σχεδίασης και προσομοίωσης το οποίο διασυνδέει τα περισσότερα εργαλεία σχεδίασης που απαιτούνται για την υλοποίηση ενός τέτοιους συστήματος. Με τη βοήθεια της Python και εργαλείων λεξικής ανάλυσης, καταφέραμε να υποστηρίξουμε περιγραφές υλικού και λογισμικού σε υψηλό επίπεδο, οι οποίες όμως υποστηρίζουν και στοιχεία από τις ήδη υπάρχουσες γλώσσες περιγραφής υλικού, όπως είναι η VHDL και Verilog.

1.1.2 Συνεισφορά

Η συνεισφορά της έρευνας που διεξήχθει επικεντρώνεται στο να αποδείξει ότι μία υψηλού επιπέδου γλώσσα όπως η Python μπορεί να χρησιμοποιηθεί για να σχεδιαστεί, να προσομοιωθεί και να υλοποιηθεί ένα ψηφιακό σύστημα. Οι δυνατότητες αυτές κρίνονται πολύ χρήσιμες, ειδικά στα αρχικά στάδια υλοποίησης, όταν πολλές μονάδες ενός συστήματος δεν έχουν ακόμα προδιαγραφεί πλήρως και επίσης δεν είναι γνωστός ο καταμερισμός των βασικών λειτουργιών ανάμεσα σε υλικό και λογισμικό. Όπως θα δούμε η Python είμαι μια ιδανική γλώσσα για να διαχειριστεί το πλήθος των εργαλείων που απαιτούνται για τη σχεδίαση ενός SoC. Δείξαμε ακόμα τον τρόπο με τον οποίο τα αντικειμενοστραφή και scripting χαρακτηριστικά της γλώσσας μπορούν εύκολα να διαχειριστούν την παραγωγή και επεξεργασία ASCII αρχείων, όπως VHDL, Tcl scripts, αρχεία XML κ.α. με στόχο τη γρήγορη προσομοίωση και διερεύνηση διαφορετικών αρχιτεκτονικών για την υλοποίηση ενός SoC.

Μελετώντας και συγκρίνοντας με τα εργαλεία σχεδίασης που ήδη υπάρχουν, θεωρούμε ότι μόνο το SysPy:

1. υποστηρίζει τη συσχεδίαση υλικού/λογισμικού με την δυνατότητα συνπροσομοίωσης μοντέλων υλικού σε Python και αλγορίθμων λογισμικού σε Python ή C.

2. κάνει χρήση μιας δημοφιλούς scripting γλώσσας για την περιγραφή της αρχιτεκτονικής ενός ψηφιακού συστήματος σε υψηλό επίπεδο (ADL: Architectural Description Language) [68]

και επιπλέον υποστηρίζει την προσομοίωση σε χαμηλό επίπεδο περιγραφών RTL αλλά και σε αλγοριθμικό επίπεδο συμπεριφοράς και αυτόματη παραγωγή αρχείων κυματομορφών Value Change Dump (VCD) για χρήση με δημοφιλή εργαλεία προσομοίωσης, όπως το ModelSim από την Mentor Graphics.

3. υποστηρίζει την *αυτόματη παραγωγή* συνθέσιμου κώδικα VHDL με χρήση παραμετροποιήσιμων συναρτήσεων Python.

4. υποστηρίζει σχεδίαση SoC με ενσωμάτωση δωρεάν διαθέσιμων πυρήνων επεξεργαστών, όπως ο Leon και ο OpenRISC.

5. διευκολύνει τη χρήση των εργαλείων σχεδίασης FPGA με την αυτόματη παραγωγή αρχείων script Tcl για την οδήγηση των εργαλείων λογικής σύνθεσης και φυσικής σχεδίασης και εκτέλεση τους σε περιβάλλον Linux για την παραγωγή αρχείων προγραμματισμού μονάδων FPGA της εταιρείας Xilinx.

Με αυτόν τον τρόπο το εργαλείο που σχεδιάσαμε κάνει χρήση των καλύτερων χαρακτηριστικών της Python για να περιγράψει την αρχιτεκτονική ενός SoC σε υψηλό επίπεδο, αλλά και για να αυτοματοποιήσει τα περισσότερα από τα βήματα σχεδίασης που απαιτούνται για την υλοποίηση του συστήματος στο υλικό.

1.1.3 Σύνοψη κεφαλαίων

Στην παρούσα διατριβή περιγράφουμε τη μεθοδολογία αλλά και τις δυνατότητες του εργαλείου που σχεδιάστηκε και επίσης παραθέτουμε ικανό αριθμό παραδειγμάτων σχεδίασης ώστε να γίνουν κατανοητά τα χαρακτηριστικά του, αλλά και να παρακινήσουμε τον μελλοντικό αναγνώστη να κάνει χρήση του εργαλείου που διατίθεται ελεύθερο. Για τους επεξεργαστές που χρησιμοποιούμε στα παραδείγματα σχεδίασης κάνουμε χρήση των περιγραφών τους για να τους υλοποιήσουμε (soft IP cores), σε αντίθεση με επεξεργαστές που είναι ήδη υλοποιημένοι στο υλικό μέσα σε μια μονάδα FPGA (hard-wired cores). Οι επεξεργαστές που υποστηρίζονται στην τρέχουσα έκδοση του SysPy είναι α) ο μικροελεγκτής 8-bit AVR ATmega128 ATmega128 [76], ο οποίος παρέχεται μέσα από την κοινότητα του OpenCores [73] για τη σχεδίαση ενός συστήματος επεξεργασίας εικόνων και β) ο επεξεργαστής 32-bit Leon3 [11], από την Aeroflex Gaisler, για ένα σύστημα επεξεργασίας ήχου και ένα σύστημα επεξεργασίας βιολογικών δεδομένων. Δοκιμές σχεδίασης επίσης πραγματοποιήσαμε και με τη χρήση του επεξεργαστή 32-bit OpenRISC [29], επίσης από το OpenCores.

Το ελληνικό κείμενο αποτελεί μια ευρεία περίληψη του αγγλικού κειμένου της διατριβής. Η οργάνωση των κεφαλαίων της διατριβής στο ελληνικό κείμενο έχει ως εξής:

- στο Κεφάλαιο 2 γίνεται ανάλυση της σχετικής βιβλιογραφίας και συγκρίνονται οι δυνατότητες του SysPy με άλλα σχετικά εργαλεία ψηφιακής σχεδίασης. Γίνεται αναφορά επίσης στους βασικούς λόγους για τους οποίους κάναμε χρήση της Python.
- στο Κεφάλαιο 3 παρουσιάζονται τα κύρια χαρακτηριστικά και οι λειτουργίες του εργαλείου.
- στο Κεφάλαιο 4 αναλύεται η μεθοδολογία υλοποίησης των τριών παραδειγμάτων σχεδίασης και ο τρόπος με τον οποίο κάναμε χρήση των δυνατοτήτων που περιέχει σε κάθε περίπτωση. Αναφέρονται επίσης τα αποτελέσματα της υλοποίησης στη μονάδα FPGA που είχαμε στη διάθεση μας. Γίνεται αναφορά επίσης στη χρήση της μεθόδου BDTi για μία αρχική ποσοτική αξιολόγηση των χαρακτηριστικών και της χρηστικότητας του SysPy.
- στο Κεφάλαιο 5 συνοψίζουμε τα αποτελέσματα της διατριβής καθώς και πιθανές μελλοντικές βελτιώσεις και επεκτάσεις στις ήδη υπάρχουσες δυνατότητες.

1.2 Σχετική βιβλιογραφία

Για την ανάπτυξη μιας εφαρμογής πρέπει κάθε φορά να επιλέγεται το κατάλληλο προγραμματιστικό περιβάλλον. Στην περίπτωση μας, η Python χρησιμοποιήθηκε για την ανάπτυξη του εργαλείου αλλά και σαν γλώσσα περιγραφής για την μοντελοποίηση στοιχείων υλικού και λογισμικού. Σε αυτό το κεφάλαιο αιτιολογούμε τη χρήση της Python για την ανάπτυξη ενός εργαλείου που στοχεύει τη σχεδίαση ψηφιακών συστημάτων. Επίσης κάνουμε αναφορά στον τρόπο με τον οποίο οι σύγχρονες διατάξεις FPGA υποστηρίζουν τη σχεδίαση με ενσωματωμένους πυρήνες επεξεργαστών. Επιπλέον γίνεται αναφορά σε εφαρμογές στις οποίες έχει χρησιμοποιηθεί η Python, έτσι ώστε να βοηθήσουμε τον αναγνώστη να εκτιμήσει τις δυνατότητες της γλώσσας, ενώ ειδική αναφορά γίνεται σε εργαλεία ψηφιακής σχεδίασης στα οποία χρησιμοποιείται η Python και γίνεται σύγκριση των χαρακτηριστικών τους με τα αντίστοιχα του SysPy.

1.2.1 Ενσωματωμένα συστήματα με πυρήνες επεξεργαστών

Στα περισσότερα ενσωματωμένα συστήματα που υλοποιούνται σε μονάδες προγραμματιζόμενης λογικής χρησιμοποιείται πλέον τουλάχιστον ένας πυρήνας προγραμματιζόμενου επεξεργαστή. Ο επεξεργαστής λειτουργεί σαν ελεγκτής των διαφόρων πρωτοκόλλων επικοινωνίας που διασυνδέουν το σύστημα με εξωτερικές μονάδες. Το λογισμικό που εκτελεί ο επεξεργαστής υλοποιεί τα υψηλά επίπεδα των πρωτοκόλλων επικοινωνίας, όπως π.χ. στην περίπτωση του Ethernet τον διαχωρισμό των δεδομένων σε πακέτα δεδομένων, η διαχείριση των οποίων είναι αρκετά δύσκολη αν πρέπει να γίνει στο υλικό με χρήση μηχανών καταστάσεων. Ο επεξεργαστής επίσης διαχειρίζεται και την ροή δεδομένων προς άλλους επεξεργαστές ειδικού σκοπού που συνδέονται σαν περιφερειακές μονάδες.

Οι πλέον πρόσφατες "οικογένειες" μονάδων FPGA, όπως η σειρά Virtex-7 από την Xilinx και η σειρά Arria-V από την Altera, ενσωματώνουν διπύρηνους επεξεργαστές ARM απευθείας στο υλικό (hardwire cores), οι οποίοι επικοινωνούν με το υπόλοιπα στοιχεία προγραμματιζόμενης λογικής στο FPGA μέσω ειδικών διαύλων ελέγχου και δεδομένων. Ο νέος αυτός τύπος FPGA αναφέρεται ως προγραμματιζόμενο SoC, όπου μπορούν να επαναπρογραμματιστούν ταυτόχρονα στοιχεία λογισμικού και υλικού. Επίσης οι απαραίτητες τροποποιήσεις έχουν γίνει και στο παρεχόμενο λογισμικό σχεδίασης από τις εταιρείες FPGA, ώστε να υποστηρίζει νέες μεθοδολογίες σχεδίασης βασισμένες στις νέες δυνατότητες των FPGAs. Το νέο λογισμικό σχεδίασης υποστηρίζει μεθόδους σχεδίασης χρησιμοποιώντας πληθώρα έτοιμων μονάδων (block oriented design) που είναι συμβατές και μπορούν να συνδεθούν εύκολα σαν περιφερειακές μονάδες του επεξεργαστή. Αυτό ο τρόπος σχεδίασης επιταχύνει σημαντικά τον χρόνο υλοποίησης ενός processor-centric SoC. Αν λάβουμε επίσης υπόψιν και τη μειωμένη κατανάλωση ισχύος που έχουν οι νέες μονάδες FPGA, μπορούμε πλέον να συγκρίνουμε μια υλοποίηση FPGA με την αντίστοιχη σε μονάδα Application Specific Integrated Circuit (ASIC). Αν βέβαια απαιτείται μαζική παραγωγή ενός ολοκληρωμένου κυκλώματος, τότε από άποψη κόστους η λύση του ASIC παραμένει η μόνη επιλογή.

Τα προγραμματιζόμενα SoC θα γίνονται όλο και πιο δημοφιλή, σε σχεδιαστές υλικού αλλά και λογισμικού, όσο το πλήθος των έτοιμων λογικών μονάδων που είναι διαθέσιμες στα εργαλεία σχεδίασης θα αυξάνεται. Εφαρμογές που απαιτούν μεγάλη επεξεργαστική ισχύ, όπως εφαρμογές επεξεργασίας ήχου, εικόνας ή ελεγκτές δικτύου δεδομένων έχουν ήδη υλοποιηθεί σε μονάδες FPGA, κάνοντας χρήση του μεγάλου πλήθους έτοιμων αριθμητικών μονάδων στο υλικό που χρησιμοποιούνται για να εκτελέσουν τους απαραίτητους υπολογισμούς. Βασικός παράγοντας προς αυτή την κατεύθυνση είναι η ευκολία που θα παρέχουν τα διαθέσιμα εργαλεία σχεδίασης στην σύνδεση του προγραμματιζόμενου επεξεργαστή με άλλους επεξεργαστές ειδικού σκοπού και τις περιφερειακές μονάδες που διαχειρίζονται τα πρωτόκολλα επικοινωνίας του SoC.

1.2.2 Σχεδίαση υλικού με χρήση της Python

Η γλώσσα Python [2] είναι μια γλώσσα ανοιχτού λογισμικού η οποία αναπτύχθηκε από τον Guido van Rossum. Είναι μια γλώσσα υψηλού επιπέδου που χαρακτηρίζεται από εύκολη αναγνωσιμότητα του κώδικα λόγω της εύκολης σύνταξης που έχουν οι εντολές της. Υποστηρίζει όλες τις γνωστές μεθόδους προγραμματισμού, όπως τη δημιουργία ακολουθιακών προγραμμάτων (όπως η C), προγραμμάτων script και αντικειμενοστραφή προγραμματισμό. Ένα πρόγραμμα Python μπορεί να εκτελεστεί σαν script σε περιβάλλον λειτουργικού συστήματος Linux ή ακόμα και να συνδυαστεί με την εκτέλεση προγραμμάτων C/C++. Η Python είναι εξαιρετικά δημοφιλής στους σχεδιαστές λογισμικού, κυρίως λόγω της απλής σύνταξης που υποστηρίζει και είναι διαθέσιμη σε όλες τις εκδόσεις του λειτουργικού συστήματος Linux. Παρέχει επίσης μέσω των βιβλιοθηκών της μεγάλο αριθμό έτοιμων μεθόδων για διάφορες λειτουργίες. Μερικές από τις πιο γνωστές βιβλιοθήκες της γλώσσας είναι: α) το NumPy και το SciPy [62] με τη χρήση των οποίων μπορεί να αναπτυχθεί κώδικας σε Python, με σύνταξη παρόμοια με της γλώσσας Matlab, για την εκτέλεση υπολογισμών διανυσματικής και γραμμικής άλγεβρας, β) το matplotlib [43] το οποίο επιτρέπει τη δημιουργία γραφικών παραστάσεων με τη βοήθεια του SciPy και γ) το Scrapy το οποίο παρέχει μεθόδους για την ανεύρεση και επεξεργασία πληροφοριών στο διαδίκτυο.

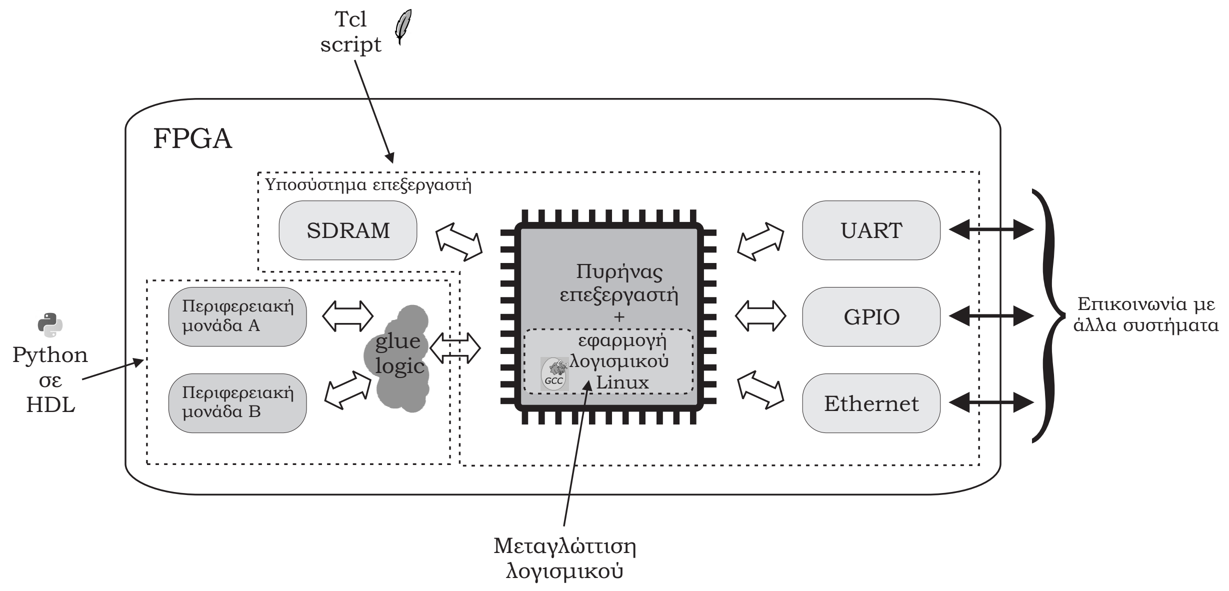
Η Python έχει χρησιμοποιεί στο παρελθόν για την ανάπτυξη δημοφιλών εργαλείων σχεδίασης λογισμικού και υλικού, ειδικότερα όπου απαιτείται παραγωγή και επεξεργασία αρχείων τύπου ASCII. Τα PyCells [22] είναι δομές γραμμένες σε Python που εκφράζουν τη λειτουργία ψηφιακών και αναλογικών μονάδων και χρησιμοποιούνται για σχεδίαση ASIC. Τα PyCells χρησιμοποιούνται ήδη σε εργαλεία σχεδίασης ολοκληρωμένων κυκλωμάτων από τις εταιρίες Cadence και Synopsys. Η Python επίσης χρησιμοποιήθηκε και στην ανάπτυξη της πλατφόρμας VIPER [90] που χρησιμοποιείται για τον προγραμματισμό ενσωματωμένων συστημάτων, όπως του συστήματος Arduino [12] [18], που χρησιμοποιείται για την υλοποίηση εφαρμογών ελέγχου.

Εφόσον ο τρόπος σύνταξης της Python ήταν ήδη δημοφιλής και αποδεκτός από ένα ευρύ κοινό, προσπαθήσαμε κατά την ανάπτυξη του SysPy να κάνουμε χρήση των πιο κοινά αποδεκτών και συμβατών χαρακτηριστικών της γλώσσας και να αποφύγουμε τρόπους περιγραφής και προγραμματισμού ασύμβατους με τη γλώσσα και τα κοινά αποδεκτά πρότυπα της. Τα δύο πιο σημαντικά χαρακτηριστικά της γλώσσας που μας ώθησαν στη χρήση της για την ανάπτυξη ενός εργαλείου ψηφιακής σχεδίασης είναι τα ακόλουθα:

- Ενσωμάτωση και χρήση μέσα από ένα κοινό προγραμματιστικό περιβάλλον όλων των διαφορετικών εργαλείων και προγραμμάτων που απαιτούνται για τη συσχεδίαση υλικού/λογισμικού κατά τη διαδικασία σχεδίασης ενός SoC.

- Υποστήριξη τρόπου σύνταξης με τον οποίο μπορεί να περιγραφεί η λειτουργικότητα μονάδων υλικού σε χαμηλό επίπεδο Register Transfer Level (RTL), αλλά και σε υψηλό επίπεδο με χρήση μεθόδων στην Python οι οποίες παράγουν αυτόματα τις απαιτούμενε περιγραφές υλικού σε γλώσσα VHDL.

Στο Σχήμα Σ1.1 παρουσιάζουμε τον τρόπο με τον οποίο χρησιμοποιούμε την Python για να συνδέσουμε έτοιμους πυρήνες επεξεργαστών με άλλες μονάδες. Με χρήση μεθόδων στην Python παράγεται αυτόματα κώδικας VHDL που περιγράφει τη λειτουργικότητα επεξεργαστών ειδικού σκοπού καθώς και τις απαραίτητες μονάδες διασύνδεσης με τον προγραμματιζόμενο επεξεργαστή. Το SysPy επίσης παράγει αυτόματα τα απαραίτητα Tcl scripts τα οποία εισάγουν όλα τα απαραίτητα αρχεία περιγραφής υλικού (HDL: Hardware Description Language) του επεξεργαστή και εκτελούν σε γραμμή εντολών τις διαδικασίες λογικής σύνθεση και φυσικής σχεδίασης με χρήση των κατάλληλων εργαλείων FPGA. Τα παραγόμενα script είναι συμβατά με το εργαλείο ISE της Xilinx για σχεδίαση υλικού με χρήση μονάδων FPGA. Επίσης εκτελείται αυτόματα η μεταγλώττιση των προγραμμάτων C που απαιτούνται για τον προγραμματισμό του επεξεργαστή με τη χρήση κατάλληλων GCC [1] μεταγλωττιστών, που είναι συμβατοί με τις περισσότερες διαθέσιμες αρχιτεκτονικές επεξεργαστών.



Σχήμα Σ1.1: Δομή συστήματος SoC με προγραμματιζόμενο επεξεργαστή.

1.2.3 Σύγκριση με άλλα εργαλεία ψηφιακής σχεδίασης

Αν και υπάρχουν ήδη διαθέσιμα εργαλεία που υποστηρίζουν σχεδίαση ψηφιακών συστημάτων σε υψηλό επίπεδο, τα περισσότερα από αυτά δεν υποστηρίζουν τη συνύπαρξη σε ένα σχέδιο μονάδων υλικού και πυρήνων προγραμματιζόμενων επεξεργαστών. Μία σωστή μεθοδολογία σχεδίασης πρέπει αν υποστηρίζει τη χρήση εργαλείων ανάπτυξης λογισμικού καθώς επίσης και τη σωστή διασύνδεση επεξεργαστών ειδικού σκοπού με τον προγραμματιζόμενο επεξεργαστή, που αποτελεί τον κεντρικό ελεγκτή του συστήματος. Τα περισσότερα εργαλεία σχεδίασης για FPGA [94], [87] αδυνατούν να διαχειριστούν με ενιαίο τρόπο την σχεδίαση υλικού και λογισμικού που απαιτείται για την υλοποίηση ενός SoC. Επίσης οι εταιρίες FPGA υποστηρίζουν την ενσωμάτωση μόνο εμπορικών πυρήνων επεξεργαστών συμβατών μόνο με τις μονάδες FPGA που παράγουν.

Στη διεθνή βιβλιογραφία υπάρχουν επίσης αναφορές και σε άλλα μη εμπορικά εργαλεία που κάνουν χρήση της Python για τη σχεδίαση ψηφιακών συστημάτων. Το εργαλείο *PyHDL* [39] υποστηρίζει τη σχεδίαση ενός συστήματος με χρήση περιγραφών δομής, όπου δομές σε C++ χρησιμοποιούνται για να περιγράψουν τη λειτουργία ψηφιακών μονάδων. Το *PHDL* [63] επίσης υποστηρίζει περιγραφές δομής σε Python και παράγει κώδικα RTL για υλοποίηση ενός συστήματος σε FPGA. Μειονέκτημα αποτελεί η χρήση έτοιμων βιβλιοθηκών σχετικά απλών λογικών μονάδων π.χ. λογικές πύλες, πολυπλέκτες, καταχωρητές κ.α.

Το *PyMTL* [58] χρησιμοποιεί την Python για να περιγράψει ψηφιακές λειτουργίες σε χαμηλό επίπεδο περιγραφών RTL αλλά και πιο υψηλό αφηρημένο επίπεδο. Αν και υποστηρίζει την μετατροπή των περιγραφών σε γλώσσα Verilog, το *PyMTL* εστιάζει περισσότερο στην προσομοίωση ψηφιακών συστημάτων, όπου η χρήση μεταγλωττιστών C++ επιταχύνει κατά πολύ τον χρόνο προσομοίωσης μια περιγραφής υψηλού επιπέδου Python. Οι παραγόμενες περιγραφές υλικού σε Verilog δεν είναι συμβατές για υλοποίηση σε FPGA ή ASIC. Επίσης τα μοντέλα προσομοίωσης που χρησιμοποιούνται δεν μοντελοποιούν την καθυστέρηση διέλευσης μέσα από τις λογικές μονάδες που παρατηρείται στο υλικό, όπως υποστηρίζεται στα μοντέλα του SysPy.

Αλλα εργαλεία όπως το *MyHDL* [23] [71] και το *PDSDL* [97] υποστηρίζουν λογικές περιγραφές δομής και περιγραφές συμπεριφοράς σε Python και μετατροπή σε γλώσσα HDL. Το *MyHDL* επίσης υποστηρίζει την προσομοίωση ψηφιακών μονάδων και παρέχει αποτελέσματα σε μορφή κειμένου και όχι με χρήση αρχείων κυματομορφών VCD όπως κάνουμε στο SysPy. Επίσης στο *MyHDL* δεν υποστηρίζεται η σχεδίαση με χρήση έτοιμων μονάδων σε VHDL ή Verilog όπως υποστηρίζεται στο SysPy, και όλες οι λογικές λειτουργίες πρέπει να περιγραφούν με τη χρήση της Python.

Συνοψίζοντας τις δυνατότητες του εργαλείου που σχεδιάσαμε σε σύγκριση με τα προαναφερόμενα

	Υποστηριζόμενες δυνατότητες						
Εργαλεία	**Python σε RTL**	**Παραγωγή συνθέσιμου κώδικα για FPGA**	**Δυνατότητα προσομοίωσης**	**Συσχεδίαση υλικού-λογισμικού**	**Χρήση πυρήνων επεξεργαστή**	**Χρήση εργαλείων λογικής σύνθεσης**	Αναφορές
PyHDL	X	-	-	-	-	-	[39]
PHDL	X	X	-	-	-	-	[63]
MyHDL	X	-	X	-	-	-	[23]
PyMTL	X	X	X	-	-	-	[58]
PDSDL	X	-	-	-	-	-	[97]
SysPy	X	X	X	X	X	X	[61]

Πίνακας Π1.1: Σύγκριση χαρακτηριστικών των εργαλείων σχεδίασης.

εργαλεία, μόνο το SysPy υποστηρίζει:

1. τη σχεδίαση processor-centric SoC και διαχείριση των απαραίτητων εργαλείων ανάπτυξης λογισμικού.
2. χρήση παραμετροποιήσιμων μεθόδων Python για αυτόματη μετατροπή τους σε περιγραφές VHDL.
3. προσομοίωση υψηλού επιπέδου με χρήση μεθόδων και κλάσεων Python για την περιγραφή της λειτουργία μονάδων του υλικού.
4. ταυτόχρονη προσομοίωση περιγραφών υλικού υψηλού επιπέδου και λογισμικού σε γλώσσα C
5. καταχώρηση κυματομορφών προσομοίωσης σε αρχεία τύπου VCD.
6. αυτόματη παραγωγή και χρήση Tcl scripts για τη διαχείριση όλων των απαραίτητων εργαλείων για τη συσχεδίαση υλικού/λογισμικού σε μονάδες FPGA.

Στον Πίνακα Π1.1 παρουσιάζουμε μία σύγκριση των προαναφερόμενων εργαλείων. Όπως φαίνεται όλα τα εργαλεία μπορούν να παράγουν κώδικα RTL από περιγραφές Python, αλλά μόνο το SysPy και το *PHDL* υποστηρίζουν την παραγωγή κώδικα συμβατό με υλοποιήσεις FPGA. Επίσης ενώ το *MyHDL* έχει δυνατότητες προσομοιώσεις ενός συστήματος στο επίπεδο μιας περιγραφής Python, μόνο το SysPy υποστηρίζει την παρουσίαση των αποτελεσμάτων προσομοίωσης μοντέλων Python με χρήση αρχείων VCD, που αποτελούν και τον πιο διαδεδομένο τρόπο καταχώρησης αρχείων ψηφιακών κυματομορφών και είναι συμβατά με όλα τα εργαλεία ψηφιακής προσομοίωσης. Τέλος μόνο το SysPy υποστηρίζει τη συσχεδίαση και συνπροσομοίωση στοιχείων υλικού και λογισμικού σε υψηλό επίπεδο, δίνοντας τη δυνατότητα παράλληλης μοντελοποίησης του λογισμικού ενός ενσωματωμένου επεξεργαστή μαζί με τις διασυνδεδεμένες μονάδες υλικού.

1.3 Μεθοδολογία σχεδίασης

Σε αυτό το κεφάλαιο περιγράφεται η μεθοδολογία σχεδίασης που ακολουθείται στο SysPy. Περιγράφονται επίσης οι δυνατότητες για την προσομοίωση συστημάτων SoC με πυρήνα επεξεργαστή. Αναφέρονται επιπλέον τα χαρακτηριστικά εκείνα που υποστηρίζουν την προσομοίωση μονάδων υλικού και λογισμικού και τον τρόπο με τον οποίο συνδυάζονται τα αποτελέσματα της προσομοίωσης σε κοινά μοντέλα περιγραφής. Παρουσιάζεται επίσης και η δυνατότητα παραγωγής αριθμητικών και ψηφιακών κυματομορφών, οι οποίες χρησιμοποιούνται για τον καθορισμό βασικών παραμέτρων ενός συστήματος.

1.3.1 Μεθοδολογίας σχεδίασης

Η μεθοδολογία σχεδίασης παρουσιάζεται στο Σχήμα Σ1.2. Οι δυνατότητες της καλύπτουν έξι κύρια χαρακτηριστικά που θεωρούμε ότι είναι απαραίτητα για τη σχεδίαση ενός processor-centric SoC:

1. Περιγραφή σε HDL μονάδων υλικού, που συνδέονται σαν περιφερειακές μονάδες του πυρήνα επεξεργαστή.

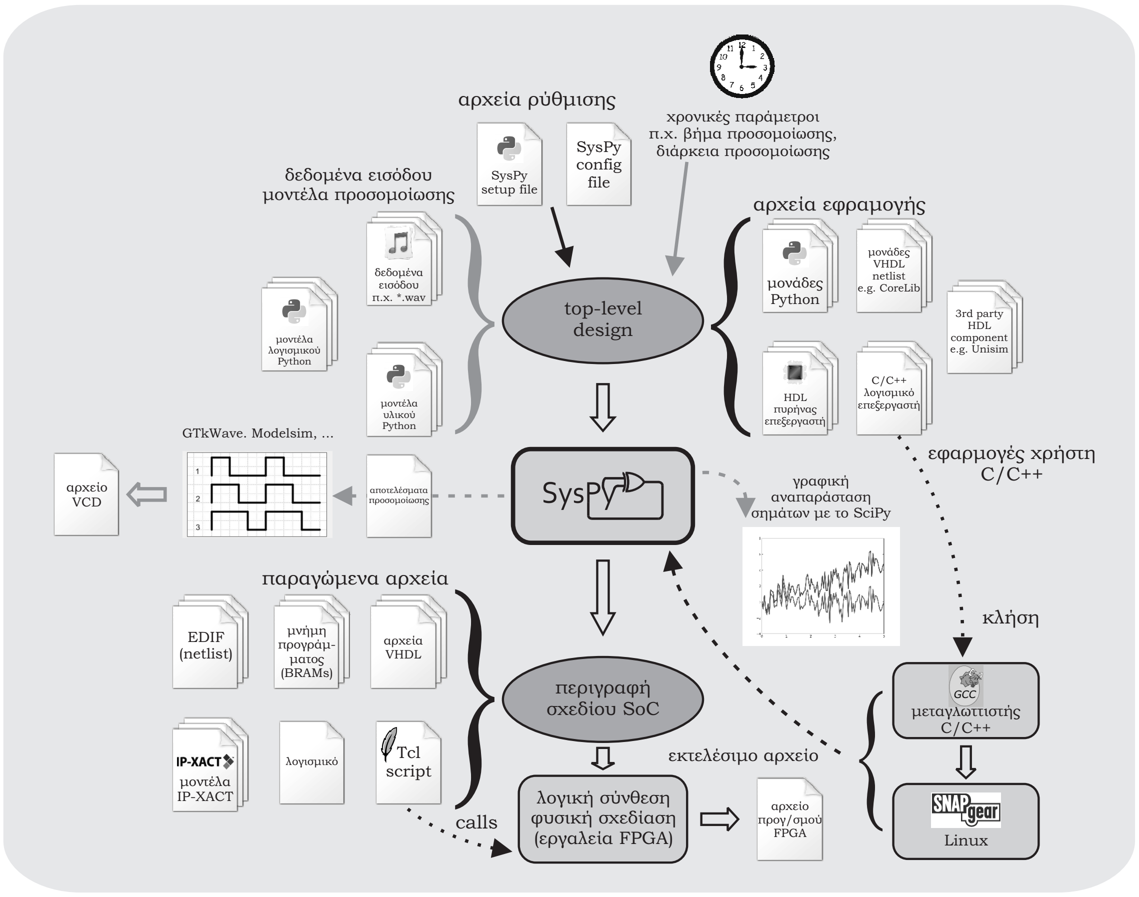
2. Χρήση σε ένα σχέδιο έτοιμων μονάδων υλικού από σχετικές βιβλιοθήκες ή υλοποιημένες σε γλώσσα Verilog ή VHDL.

3. Προσομοίωση περιγραφών Python που προδιαγράφουν τη λειτουργικότητα μονάδων υλικού και λογισμικού που εκτελείται από τον επεξεργαστή.

4. Αυτόματη παραγωγή προγραμμάτων script για την κλήση εργαλείων ανάπτυξη λογισμικού, π.χ. κλήση μεταγλωττιστών C, αρχικοποίηση προγράμματος του επεξεργαστή σε μονάδες μνήμης (BRAM: Block RAM) στο FPGA.

5. Αυτόματη παραγωγή και εκτέλεση προγραμμάτων Tcl script για την οδήγηση των εργαλείων λογικής σύνθεσης και φυσικής σχεδίασης σε FPGA.

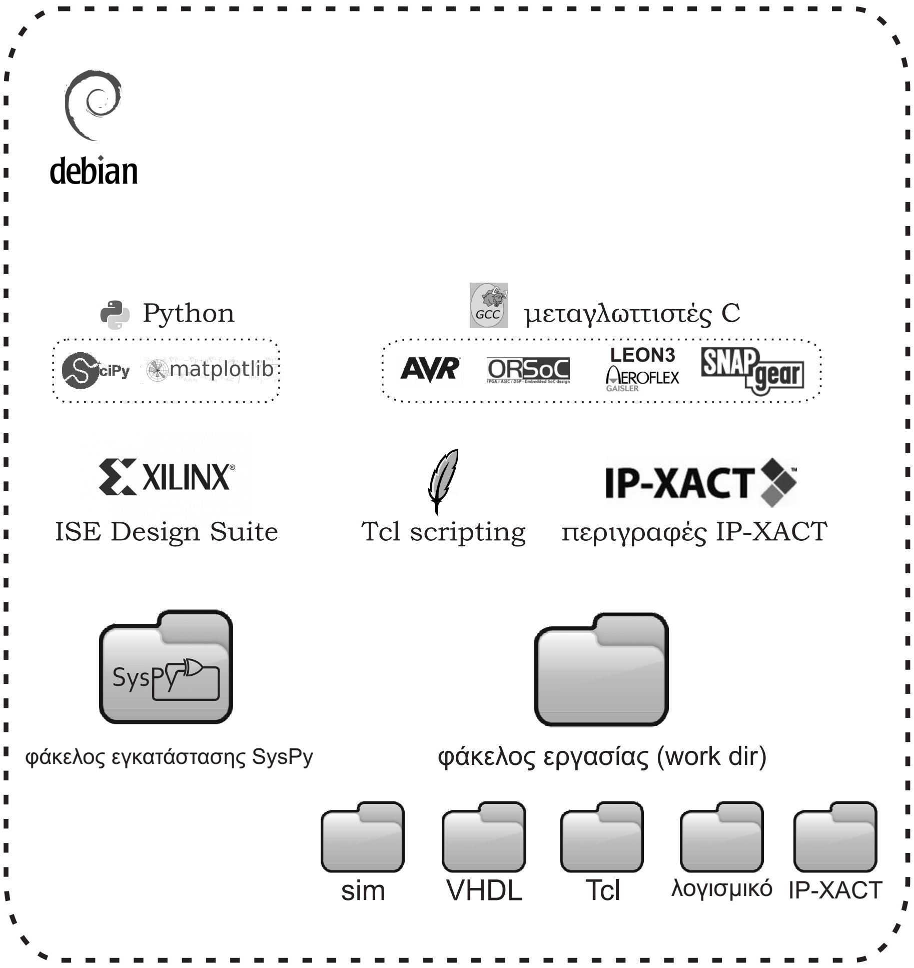
6. Παραγωγή μοντέλων XML, για την περιγραφή μονάδων υλικού, συμβατά με το πρότυπο IP-XACT [15].



Σχήμα Σ1.2: Μεθοδολογία σχεδίασης SoC με χρήση πυρήνων επεξεργαστών.

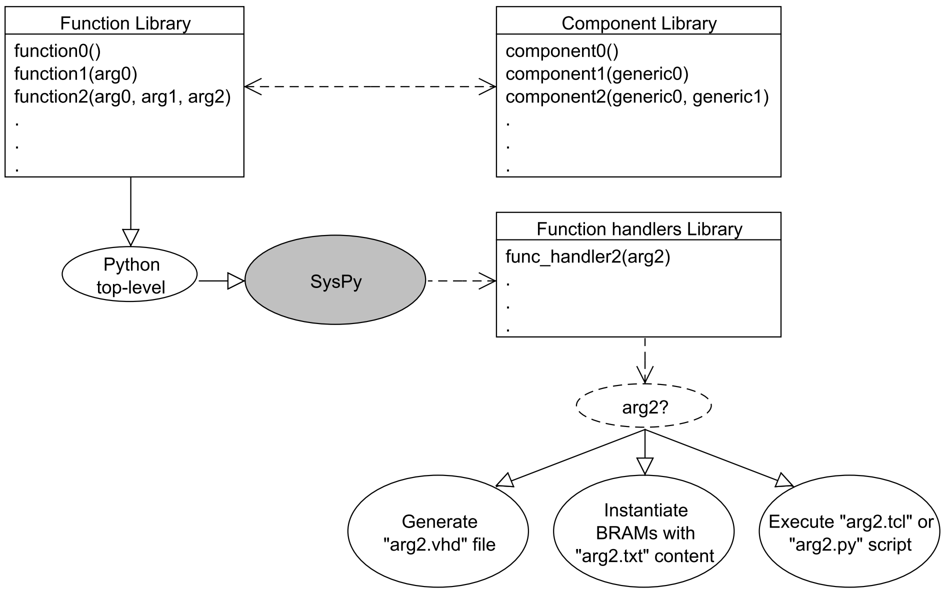
Σύμφωνα με την προτεινόμενη μεθοδολογία, η σχεδίαση ενός SoC ξεκινάει με τη δημιουργία μοντέλων περιγραφής του συστήματος. Τα μοντέλα περιγράφουν σε Python, με τη μορφή ψευδοκώδικα. τη λειτουργία του λογισμικού του επεξεργαστή, ενώ μπορεί να περιγράφουν και τη λειτουργία υλικού, για τα οποία ακόμα δεν είναι αρχικά διαθέσιμη η περιγραφή τους σε επίπεδο RTL. Το SysPy όμως υποστηρίζει και περιγραφές RTL σε επίπεδο Python, στις οποίες ο χρήστης μπορεί να μοντελοποιήσει και τις πιθανές χρονικές καθυστερήσεις που θα παρουσιάζουν μονάδες συνδυαστικής λογικής, π.χ. αριθμητικές μονάδες, πολυπλέκτες κτλ. Όσες μονάδες του υλικού περιγραφούν σε επίπεδο RTL, το εργαλείο μπορεί να μεταφράσει αυτόματα σε VHDL. Τα απαραίτητα σήματα χρονισμού επίσης μπορούν να μοντελοποιηθούν στο περιβάλλον προσομοίωσης του SysPy, π.χ. ένα σήμα ρολογιού 20MHz, δίνοντας την δυνατότητα να περιγραφούν συστήματα ακολουθιακής λογικής με σωλήνωση (pipelined datapath) όπου η ροή των δεδομένων ελέγχεται από το λογισμικό του επεξεργαστή ή από μηχανές καταστάσεων στο υλικό. Μία τέτοια περιγραφή ενός μοντέλου προσομοίωσης αντικατοπτρίζει τη λειτουργία της πλειονότητας των ψηφιακών συστημάτων. Στο Σχήμα Σ1.2 τα γκρίζα βέλη δείχνουν τα βήματα της διαδικασίας προσομοίωσης, ενώ τα μαύρα βέλη δείχνουν τη διαδικασία υλοποίησης του συστήματος στη μονάδα FPGA.

Η δυνατότητα περιγραφής προσομοίωσης υψηλού επιπέδου, σε συνδυασμό με τη δυνατότητα προσομοίωσης συμβατή με τον χρονισμό ενός συστήματος σε επίπεδο περιγραφής RTL (cycle-accurate simulation) δίνει τη δυνατότητα στον χρήστη να πάρει αποφάσεις για κρίσιμες παραμέτρους ενός SoC, πριν ξεκινήσει τη διαδικασία υλοποίησης. Οι παραγόμενες κυματομορφές προσομοίωσης δίνουν τη δυνατότητα για την επίλυση προβλημάτων χρονισμού που μπορεί να παρουσιαστούν, ειδικά στη διασύνδεση του προγραμματιζόμενου επεξεργαστή με τις περιφερειακές του μονάδες και τους επεξεργαστές ειδικού σκοπού. Στις παραγράφους που ακολουθούν περιγράφονται αυτές οι δυνατότητες με μεγαλύτερη λεπτομέρεια.



Σχήμα Σ1.3: Εγκατάσταση και χρήση του SysPy σε περιβάλλον λειτουργικού συστήματος Linux.

Το SysPy αναπτύχθηκε και η λειτουργία του δοκιμάστηκε με χρήση του λειτουργικού συστήματος Debian Linux. Αρχεία εγκατάστασης (configuration files) χρησιμοποιούνται για να ορίσει ο χρήστης όλους τους απαραίτητους φακέλους που έχει γίνει η εγκατάσταση του εργαλείου και του απαραίτητου λογισμικού και των φακέλων όπου καταχωρούνται τα αρχεία σχεδίασης (working directory). Όλα τα εργαλεία που καλούνται μέσω του SysPy καθώς και η δομή των φακέλων που δημιουργεί το εργαλείο κατά τη διαδικασία σχεδίασης, παρουσιάζονται στο Σχήμα Σ1.3.


Function Library
function0()
function1(arg0)
function2(arg0, arg1, arg2)
Component Library
component0()
component1(generic0)
component2(generic0, generic1)
Python top-level
SysPy
Function handlers Library
func_handler2(arg2)
arg2?
Generate "arg2.vhd" file
Instantiate BRAMs with "arg2.txt" content
Execute "arg2.tcl" or "arg2.py" script

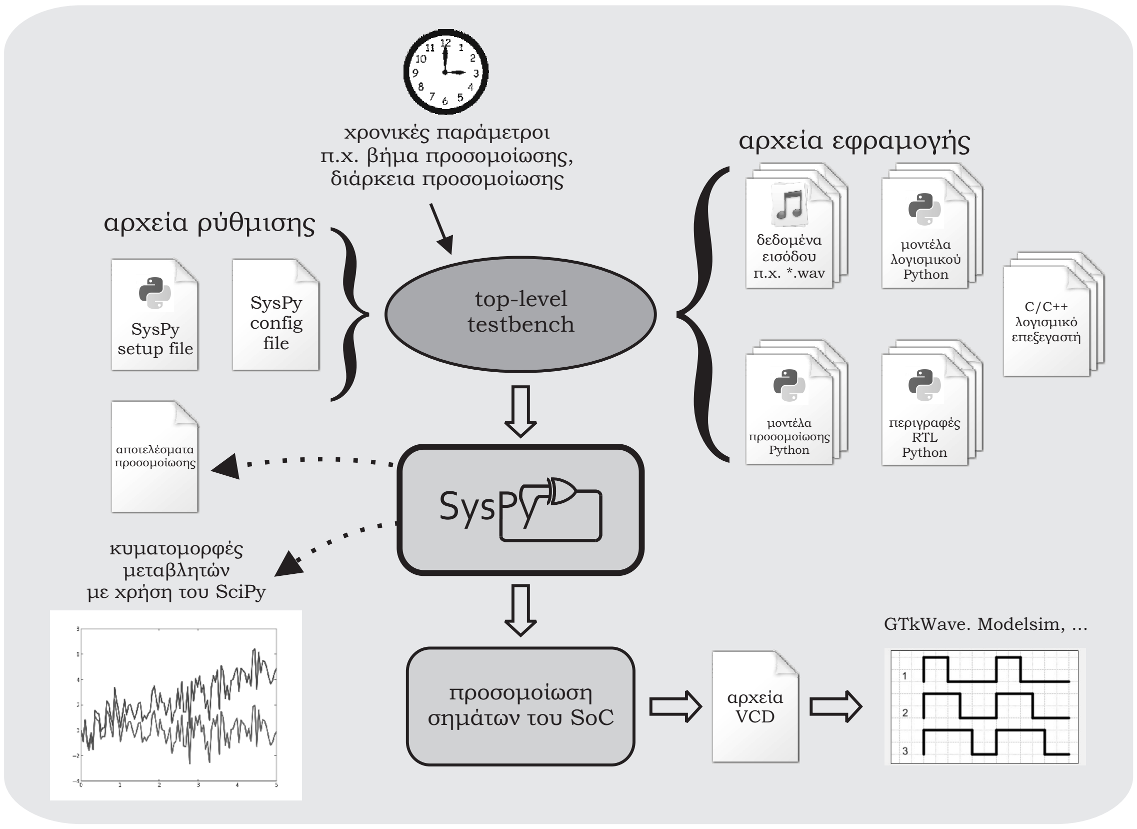
Σχήμα Σ1.4: Διάγραμμα UML για την απεικόνιση της χρήσης μεθόδων Python (function library και function handlers library) για την αυτόματη αρχικοποίηση μονάδων από τις βιβλιοθήκες του SysPy (component library).

Στο Σχήμα Σ1.4 απεικονίζεται μία αλληλουχία σε γλώσσα UML για την παρουσίαση της χρήσης των βιβλιοθηκών του SysPy και της αυτόματης παραμετροποίησης μονάδων υλικού με χρήση μεθόδων Python. Η χρήση της υψηλού επιπέδου παραμέτρου "arg2" στη συνάρτηση "function2" ενεργοποιεί την κλήση της αντίστοιχης μεθόδου "func_handler2()", όπου η παράμετρος μπορεί να αντιπροσωπεύει την ονομασία ενός αρχείου ASCII (VHDL, text, Tcl script) το οποίο επεξεργάζεται ή καλείται αυτόματα για να αρχικοποιήσει τη σχετική μονάδα υλικού. Η χρήση των μεθόδων στην αρχικοποίηση μονάδων υλικού σε περιγραφές δομής καθιστά πιο σύντομες και συμπαγής αυτού του είδους τις περιγραφές σε γλώσσα Python. Η βιβλιοθήκη "function_library" περιέχει δηλώσεις συναρτήσεων οι οποίες αρχικοποιούν αυτόματα τις μονάδες που περιέχονται στη βιβλιοθήκη "component_library".

1.3.2 Δυνατότητες προσομοίωσης ενός SoC

Το SysPy μπορεί να χρησιμοποιηθεί για να προσομοιώσει ψηφιακές μονάδες μέσω περιγραφών RTL σε γλώσσα Python. Το πιο ενδιαφέρον χαρακτηριστικό όμως είναι η δυνατότητα προσομοίωσης αλγοριθμικών περιγραφών συμπεριφοράς υλικού και λογισμικού, ειδικότερα για μονάδες ενός SoC για τις οποίες πρέπει να καθοριστεί αν είναι απαραίτητη η υλοποίηση τους είτε στο υλικό είτε με λογισμικό που εκτελείται από τον επεξεργαστή του συστήματος. Τα κύρια χαρακτηριστικά του μηχανισμού προσομοίωσης συνοψίζονται στα ακόλουθα σημεία:

- προσομοίωση περιγραφών Python, μονάδων υλικού σε επίπεδο RTL.
- προσομοίωση μονάδων υλικού και λογισμικού, π.χ. αριθμητικές μονάδες, μονάδες ελεγκτών επικοινωνίας κ.α., για τις οποίες δεν έχει υπάρχει πλήρης περιγραφή της λειτουργικότητας στα αρχικά στάδιο σχεδίασης.
- ανάπτυξη λογισμικού για τον επεξεργαστή ενός συστήματος σε γλώσσα C και προσομοίωση του σε επίπεδο συστήματος μαζί με άλλες περιγραφές Python υλικού/λογισμικού.
- καταχώρηση των αποτελεσμάτων της προσομοίωσης σε αρχεία ψηφιακών κυματομορφών τύπου VCD και χρήση τους με δημοφιλή προγράμματα προσομοίωσης, όπως το ModelSim και το GTkWave.

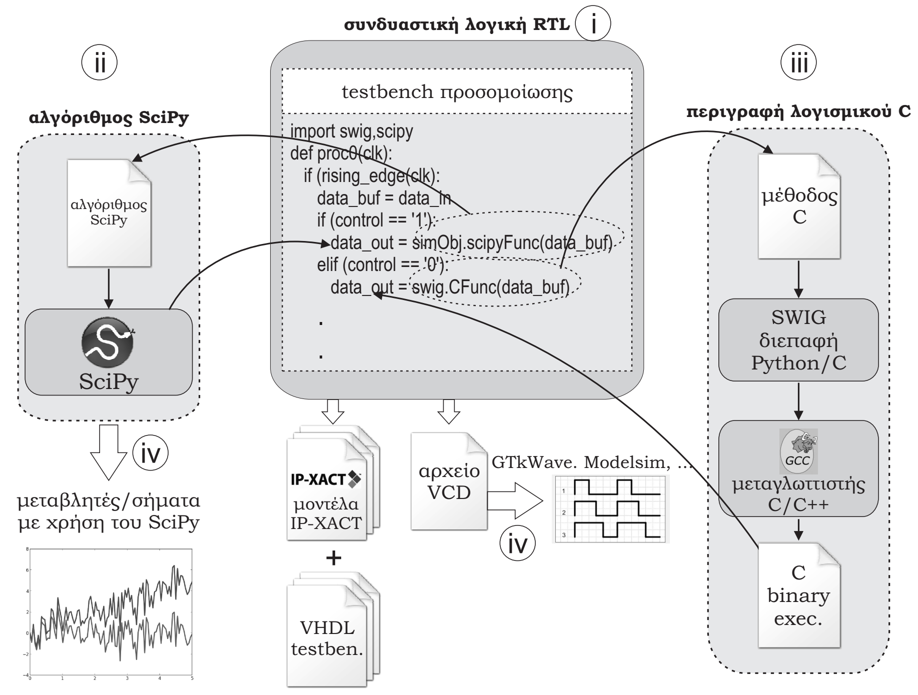


Σχήμα Σ1.5: Λειτουργικότητα του μηχανισμού προσομοίωσης.

Ειδικά για την μοντελοποίηση σύνθετων αριθμητικών μονάδων, γίνεται χρήση της αριθμητικής βιβλιοθήκης SciPy στην Python. Αριθμητικοί αλγόριθμοι μπορούν να προσομοιωθούν κάνοντας χρήση έτοιμων μεθόδων, τα αποτελέσματα των οποίων μπορούν να επεξεργαστούν περαιτέρω από άλλες μονάδες υλικού, μέσω της διασύνδεσης που είναι εφικτή στο SysPy, μεταξύ μοντέλων υλικού RTL και μοντέλων συμπεριφοράς υψηλού επιπέδου. Με τη χρήση επίσης του SciPy είναι δυνατή η παραγωγή αριθμητικών κυματομορφών κατά την προσομοίωση ενός αλγορίθμου σε υψηλό επίπεδο, π.χ. επεξεργασία δεδομένων από ψηφιακά φίλτρα. Οι κυματομορφές αυτές μπορούν να χρησιμοποιηθούν για την ρύθμιση των παραμέτρων υλοποίησης ενός αλγορίθμου, ενώ αργότερα μετά την δημιουργία μοντέλων σε επίπεδο RTL, οι ψηφιακές κυματομορφές των σημάτων μιας μονάδας πιστοποιούν την σωστή λειτουργία του αλγορίθμου στο υλικό και την αριθμητική ακρίβεια των υπολογισμών. Στο Σχήμα Σ1.5 παρουσιάζονται τα βήματα που υποστηρίζονται κατά τη διαδικασία προσομοίωσης ενός SoC.

1.3.2.1 Συνπροσομοίωση υλικού/λογισμικού με περιγραφές υψηλού επιπέδου

Ένας από τους βασικούς λόγους της επιλογής της Python ήταν και η δυνατότητα της γλώσσας να καλεί μεθόδους υλοποιημένες σε άλλες γλώσσες λογισμικού, όπως η C/C++. Η κλήση μεθόδων C μέσα από μοντέλα προσομοίωσης σε Python μαζί με τη χρήση του SciPy, μας δίνει τη δυνατότητα ταυτόχρονης προσομοίωσης αλγορίθμων λογισμικού με μονάδες υλικού, όπου η ανάπτυξη και προσομοίωση του λογισμικού μπορεί να γίνει με χρήση: α) αλγοριθμικών περιγραφών υψηλού επιπέδου με περιγραφές (Matlab-like) στο SciPy και β) με περιγραφές σε γλώσσα C, οι οποίες μπορούν εύκολα μετά να χρησιμοποιηθούν για τον προγραμματισμό του επεξεργαστή ενός συστήματος. Σύμφωνα με τη μελέτη της υπάρχουσας βιβλιογραφίας αυτό αποτελεί ένα από τα πιο καινοτόμα χαρακτηριστικά του SysPy.



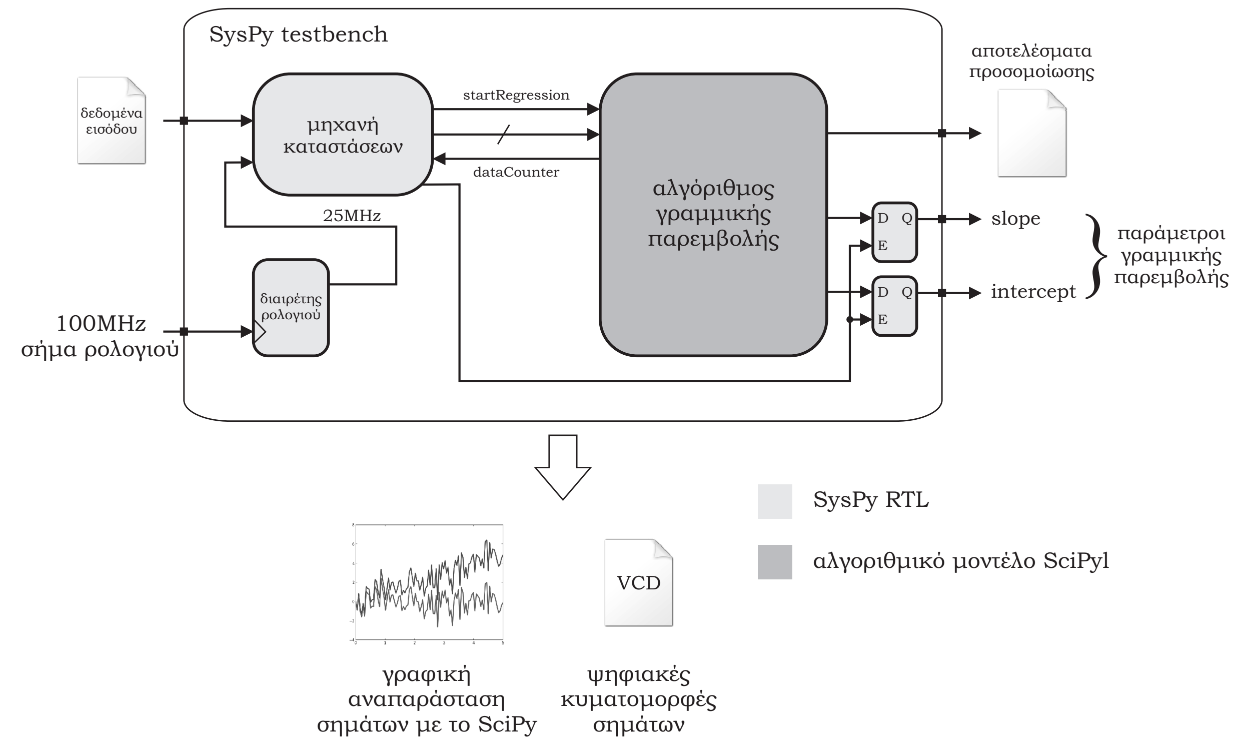
```
import swig,scipy
def proc0(clk):
  if (rising_edge(clk):
    data_buf = data_in
    if (control == '1'):
      data_out = simObj.scipyFunc(data_buf)
    elif (control == '0'):
      data_out = swig.CFunc(data_buf)
    .
    .
```

Σχήμα Σ1.6: Μεθοδολογία προσομοίωσης με χρήση περιγραφών υλικού RTL και αλγοριθμικών μοντέλων.

Στο Σχήμα Σ1.6 παρουσιάζομαι ένα τυπικό μοντέλο προσομοίωσης, όπου i) γίνεται χρήση της Python για την περιγραφή ενός διαύλου δεδομένων (pipelined datapath) ii) περιγραφές Python με τη χρήση του αριθμητικού πακέτου SciPy χρησιμοποιούνται για αλγοριθμικές περιγραφές μονάδων ενός συστήματος για τις οποίες δεν υπάρχει ακόμα υλοποίηση σε υλικό (HDL) ή λογισμικό (C/C++), iii) περιγραφές λογισμικού σε C χρησιμοποιούνται για την υλοποίηση αλγορίθμων των οποίων την εκτέλεση τους θα αναλάβει ο επεξεργαστής του συστήματος, ενώ iv) κατά τη διάρκεια της προσομοίωσης παράγονται κυματομορφές των ψηφιακών σημάτων του συστήματος μέσω του SciPy αλλά και σε μορφή αρχείων τύπου VCD, συμβατή με άλλα εργαλεία προσομοίωσης. Μέσω της χρήσης του μηχανισμού προσομοίωσης του SysPy, οι προγραμματιστές λογισμικού μπορούν α) να αναπτύξουν αλγοριθμικά μοντέλα με τη χρήση του SciPy και να ελέγξουν τη σωστή λειτουργικότητα τους μέσω της συνπροσομοίωσης με τις μονάδες του υλικού. β) Μπορούν επίσης να μετατρέψουν τις περιγραφές λογισμικού από Python σε C και εκ νέου να ελέγξουν τη λειτουργικότητα τους μέσω του SysPy, διευκολύνοντας έτσι και επιταχύνοντας τη διαδικασία ανάπτυξης του λογισμικού σε υψηλό επίπεδο.

1.3.3 Παράδειγμα προσομοίωσης

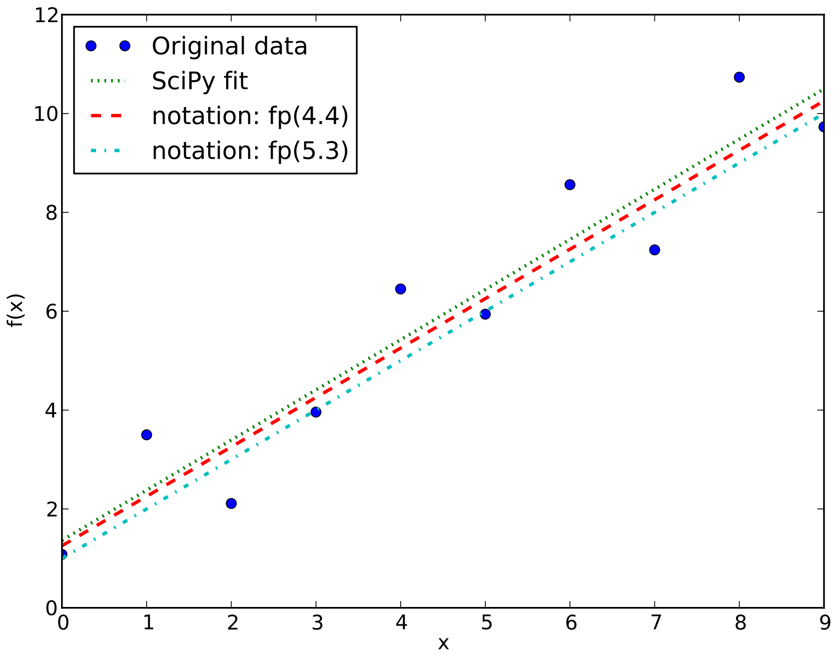
Για την καλύτερη κατανόηση των δυνατοτήτων προσομοίωσης που παρέχει το SysPy παρουσιάζουμε ένα παράδειγμα προσομοίωσης ενός αριθμητικού μοντέλου για τον υπολογισμό του αλγορίθμου εύρεσης πολυωνύμου γραμμικής παρεμβολής. Ο αλγόριθμος εφαρμόζεται πάνω σε δεδομένα εισόδου που παρέχονται σε μορφή αρχείων ASCII και υπολογίζει τις παραμέτρους μιας γραμμικής εξίσωσης που εκφράζει τα εισερχόμενα δεδομένα. Προδιαγράφοντας τη σύνθεση του συστήματος, χωρίσαμε τις λειτουργίες του στις ακόλουθες τρεις μονάδες: α) την μονάδα που εκτελεί τους αριθμητικούς υπολογισμούς του αλγορίθμου, β) μονάδες μνήμης για την καταχώρηση των αρχικών δεδομένων και των αποτελεσμάτων του αλγορίθμου και γ) μονάδες για την λήψη και την μετάδοση δεδομένων από και προς την αριθμητική μονάδα. Όλες οι αριθμητικές λειτουργίες υλοποιούνται σε μονάδες υλικού με στόχο την γρήγορη εκτέλεση του αλγορίθμου. Μία μηχανή καταστάσεων στο υλικό αναλαμβάνει τη διαχείριση διακίνησης των δεδομένων μέσα στο σύστημα, κάτω από τον έλεγχο του λογισμικού, το οποίο επίσης αναλαμβάνει την καταχώρηση των αρχικών δεδομένων και των αποτελεσμάτων επεξεργασίας σε μορφή αρχείων. Η δομή του συστήματος παρουσιάζεται στο Σχήμα Σ1.7.


SysPy testbench
δεδομένα εισόδου
μηχανή καταστάσεων
startRegression
dataCounter
25MHz
διαιρέτης ρολογιού
100MHz σήμα ρολογιού
αλγόριθμος γραμμικής παρεμβολής
D Q
E
αποτελέσματα προσομοίωσης
slope
intercept
παράμετροι γραμμικής παρεμβολής
SysPy RTL
αλγοριθμικό μοντέλο SciPyl
VCD
γραφική αναπαράσταση σημάτων με το SciPy
ψηφιακές κυματομορφές σημάτων

Σχήμα Σ1.7: Σχηματικό διάγραμμα του μοντέλου προσομοίωσης του αλγορίθμου γραμμικής παρεμβολής.

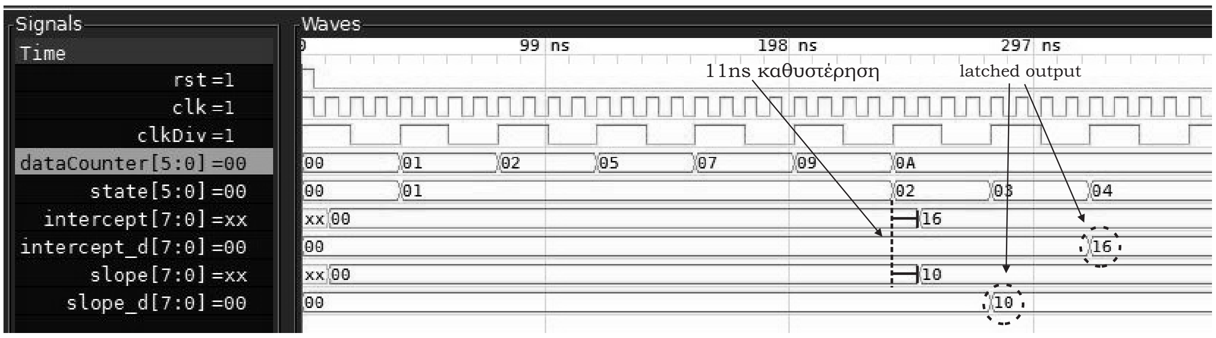
Για λόγους οικονομίας χώρου και έκτασης του κειμένου, παραθέτουμε αναφορές σε παραδείγματα κώδικα, όπως αυτά παρουσιάζονται στο Αγγλικό κείμενο. Στο Code Example 3.1 δηλώνονται όλα τα σήματα εισόδου/εξόδου του συστήματος. Στις γραμμές 4-9 παράγεται η κυματομορφή του βασικού συστήματος ρολογιού (100MHz, 50% duty cycle, διάρκεια προσομοίωσης 15us), ενώ στις γραμμές 19-21 ανατίθενται οι τιμές των σημάτων εισόδου κατά τη διάρκεια της προσομοίωσης, π.χ. στη γραμμή 19 ορίζεται ότι το σήμα `rst` ενεργοποιείται για 5ns στην αρχή της προσομοίωσης. Για τη μηχανή καταστάσεων γίνεται χρήση περιγραφών RTL σε Python, όπως παρουσιάζεται στα Code Examples 3.2 και 3.3. Οι απαραίτητες βιβλιοθήκες και η κλάση του μοντέλου προσομοίωσης καλούνται στις γραμμές 1-4. Η συνάρτηση στη γραμμή 6 περιγράφει συνδυαστική λογική με τη χρήση των σημάτων ασύγχρονου reset και ρολογιού που χρησιμοποιούνται για την οδήγηση όλων των μονάδων του συστήματος. Το αρχείο που περιέχει τα δεδομένα εισόδου δηλώνεται στη γραμμή 19 και στη γραμμή 20 δηλώνεται η μορφή αναπαράστασης των αριθμών τύπου fixed-point (5 ακέραια ψηφία και 3 ψηφία στο δεκαδικό μέρος που θα χρησιμοποιηθούν στους υπολογισμούς. Η λειτουργία της μηχανής καταστάσεων βασίζεται στις ανερχόμενες ακμές του σήματος ρολογιού (γραμμή 26). Εφόσον γίνει η ανάγνωση των δεδομένων εισόδου (κατάσταση 1, γραμμή 38), εκτελείται η κατάσταση 2 όπου ενεργοποιείται η εκτέλεση του αλγορίθμου στην αντίστοιχη μονάδα, ενώ στην κατάσταση 3 (γραμμή

13) τα αποτελέσματα του αλγορίθμου (παράμετροι ‘α’: `slope` και ‘β’: `intercept`) παρουσιάζονται στις αντίστοιχες εξόδου του συστήματος. Στην κατάσταση 4 ενεργοποιείται στο SciPy η απεικόνιση των δεδομένων εισόδου και ο τρόπος με τον οποίο η γραμμική εξίσωση που υπολογίστηκε, όπως παρουσιάζεται στο Σχήμα Σ1.8, χρησιμοποιώντας διαφορετικές μορφές αναπαράστασης για τους δεκαδικούς αριθμούς fixed-point. Ως “original data” χαρακτηρίζονται τα δεδομένα εισόδου από το μουσικό αρχείο.



Σχήμα Σ1.8: Γραμμική παρεμβολή δεδομένων εισόδου, με χρήση διαφορετικών αναπαραστάσεων fixed-point.

Κατά τη διάρκεια της προσομοίωσης, οι κυματομορφές των σημάτων εισόδου/εξόδου του συστήματος καταχωρήθηκαν στο παραγόμενο αρχείο VCD και το πρόγραμμα GTKWave [38] χρησιμοποιήθηκε για την αναπαράσταση των κυματομορφών. Στις κυματομορφές παρουσιάζεται η συμπεριφορά των ψηφιακών σημάτων σε σχέση με το σήμα ρολογιού και επίσης λαμβάνονται υπόψιν οι λογικές καθυστερήσεις που παρουσιάζονται στα σήματα εξόδου, σύμφωνα με τον τρόπο που αυτές μοντελοποιούνται στο Code Example 3.1 (γραμμή 16, παράμετρος “del”, ορισμός 11ns λογική καθυστέρηση για τα σήματα εξόδου “slope” και “intercept”). Στο Σχήμα Σ1.9 παρουσιάζεται το σήμα ρολογιού 100MHz (`clk`), το οποίο διαιρείται μέσω της κατάλληλης λογικής στα 25MHz (`clkDiv`). Το σύστημα επεξεργάζεται δύο τιμές δεδομένων εισόδου και παράγει τις παραμέτρους `slope` και `intercept` της



Σχήμα Σ1.9: Ψηφιακές κυματομορφές σημάτων εισόδου/εξόδου του συστήματος εφαρμογής του αλγορίθμου γραμμικής παρεμβολής.

γραμμικής εξίσωσης που παρεμβάλει τα δεδομένα.

Με εύκολο τρόπο μπορεί ο χρήστης να τροποποιήσει τις συχνότητες των σημάτων ρολογιού, τις τιμές των σημάτων εισόδου και τον χρόνο των λογικών καθυστερήσεων, με στόχο τη διερεύνηση της σωστής λειτουργίας ενός ψηφιακού συστήματος. Με τον τρόπο που προσεγγίζουμε τη σχεδίαση και προσομοίωση ενός ψηφιακού SoC, κάθε λογική μονάδα αντιμετωπίζεται ως ένα μαύρο κουτί στο οποίο ο χρήστης αλλάζοντας τις τιμές των σημάτων εισόδου και τις χρονικές παραμέτρους μπορεί να διερευνήσει τη σωστή λειτουργικότητα ενός συστήματος σε αλγοριθμικό επίπεδο συμπεριφοράς αλλά και σε λογικό επίπεδο RTL της ψηφιακής υλοποίησης. Μέσω του SysPy επίσης είναι δυνατή η χρήση των αλγοριθμικών μοντέλων που αναπτύσσει ο χρήστης με τη χρήση της βιβλιοθήκης SciPy και η μετατροπή τους σε συναρτήσεις σε γλώσσα C, η λειτουργικότητα των οποίων μπορεί να προσομοιωθεί στο επίπεδο των περιγραφών Python. Η δυνατότητα αυτή κρίνεται ιδιαίτερα χρήσιμη εφόσον τα υψηλού επιπέδου μοντέλα του λογισμικού μπορούν εύκολα να μετατραπούν από Python σε γλώσσα C και να χρησιμοποιηθούν για την προσομοίωση του λογισμικού και τον προγραμματισμό του επεξεργαστικού πυρήνα ενός συστήματος.

1.4 Παραδείγματα σχεδίασης SoCs

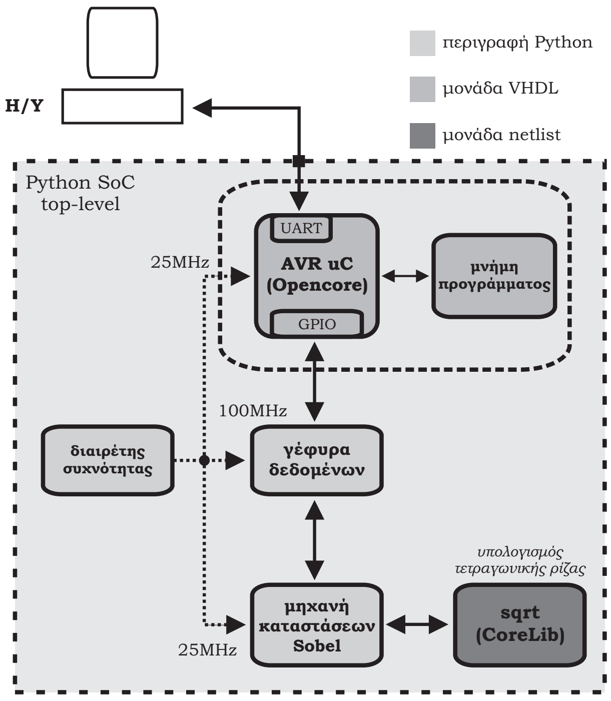
Το SysPy χρησιμοποιήθηκε για τη σχεδίαση τριών σύνθετων σχεδίων SoC με προγραμματιζόμενο επεξεργαστή. Τα παραδείγματα σχεδίασης υλοποιούν SoC: α) επεξεργασίας εικόνων, β) επεξεργασίας βιολογικών δεδομένων και γ) επεξεργασίας ήχου. Μέσω της υλοποίησης των τριών αυτών σχεδίων προσπαθήσαμε να βελτιώσουμε τα χαρακτηριστικά του SysPy και ειδικότερα τις μεθοδολογίες ανάπτυξης και περιγραφής σε υψηλό επίπεδο, χρησιμοποιώντας την Python για την περιγραφή της λειτουργικότητας ενός ψηφιακού συστήματος. Επίσης κατά τη διαδικασία ανάπτυξης κάθε σχεδίου κάναμε χρήση των δυνατοτήτων του SysPy που ήταν κάθε φορά διαθέσιμες, εφόσον η εξέλιξη του εργαλείου ήταν μια παράλληλη διαδικασία με την ανάπτυξη των τριών συστημάτων που χρησιμοποιήθηκαν ως παραδειγμάτων σχεδίασης. Στόχος επίσης ήταν να εκτιμήσουμε την αξία των εργαλείων που παρέχονται μέσω του SysPy, κυρίως ως προς την μείωση του χρόνου σχεδίασης και τη δυνατότητα χρήσης περιγραφών υψηλού επιπέδου για την υλοποίηση ψηφιακών συστημάτων με πυρήνες προγραμματιζόμενων επεξεργαστών σε μονάδες FPGA.

1.4.1 SoC επεξεργασίας εικόνων

Το πρώτο παράδειγμα που υλοποιήσαμε για να εξακριβώσουμε την ορθή λειτουργία του SysPy και να εκτιμήσουμε τις δυνατότητες του εργαλείου ήταν ένα σύστημα επεξεργασίας εικόνων [60], βασισμένο στον μικροελεγκτή 8-bit AVR ATmega128 [45]. Συγκεκριμένα χρησιμοποιήσαμε τον πυρήνα του επεξεργαστή σε περιγραφή VHDL, διαθέσιμο από την ιστοσελίδα του OpenCores [76]. Για τη σχεδίαση του συστήματος κάναμε χρήση της μεθοδολογίας που υποστηρίζετε στο SysPy, ξεκινώντας από την περιγραφή του συστήματος με χρήση της Python, μέχρι τη χρήση αρχείων Tcl για την παραγωγή του αρχείου προγραμματισμού του FPGA. Το SysPy χρησιμοποιήθηκε για να παράγει όλα τα απαιτούμενα, συμβατά με τα εργαλεία λογικής σύνθεσης για FPGA, αρχεία VHDL. Τα αρχεία περιγράφουν το σύστημα καθώς και τη διασύνδεση του επεξεργαστή ειδικού σκοπού που χρησιμοποιήθηκε με τον επεξεργαστή AVR, ενώ το SysPy διαχειρίστηκε επίσης την μεταγλώττιση του απαραίτητου λογισμικού του επεξεργαστή και την αρχικοποίηση της διαθέσιμης μνήμης προγράμματος.

Το σύστημα που υλοποιήσαμε εφαρμόζει τον αλγόριθμο Sobel [37] για την ανεύρεση ακμών σε μία ασπρόμαυρη εικόνα. Ο αλγόριθμος παράγει για κάθε ασπρόμαυρη εικόνα δύο νέες εικόνας όπου ανιχνεύει ακμές σε οριζόντια και κάθετη κατεύθυνση. Υπολογίζοντας την Ευκλείδεια απόσταση μεταξύ των αντίστοιχων εικονοστοιχείων (pixel) σε κάθε μία από τις δύο νέες εικόνας παράγεται μία τρίτη εικόνα που αποτελεί και την τελική επεξεργασμένη εικόνα όπου εντοπίζονται όλες οι ακμές της αρχικής. Η ανίχνευση ακμών πολλές φορές αποτελεί το αρχικό στάδιο επεξεργασίας μίας εικόνας σε άλλους αλγορίθμους που χρησιμοποιούνται για την ανίχνευση αντικειμένων ή προσώπων και επίσης για συμπίεση εικόνων.

Η επιλογή του ATmega128, ως πρώτου πυρήνα επεξεργαστή που ενσωματώθηκε στο SysPy έγινε επειδή ο εν λόγω πυρήνας είναι σχετικά απλός και διέθετε αρκετή μνήμη προγράμματος (128kb) και πόρτες εισόδου/εξόδου γενικού σκοπού που χρησιμοποιήσαμε για τη διασύνδεση του μέσα στο FPGA. Ο επεξεργαστής ενσωματώνει μονάδα σειριακής επικοινωνίας Universal Asynchronous Receiver Transmitter (UART), μέσω της οποίας συνδέεται με ένα H/Y, ο οποίος αποστέλλει τις προς επεξεργασία εικόνας στο FPGA της οικογένειας VIrtex-5 της Xilinx το οποίο χρησιμοποιήσαμε. Επίσης η αρχιτεκτονική AVR είναι μία από τις γρηγορότερες αρχιτεκτονικές 8-bit, όπου οι περισσότερες εντολές του επεξεργαστή απαιτούν ένα κύκλο ρολογιού για την εκτέλεση τους.



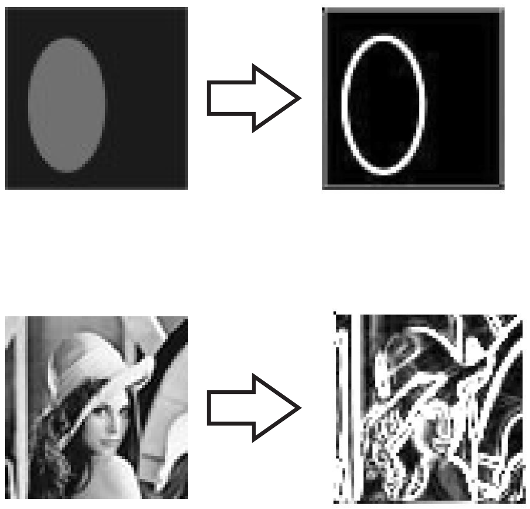
Σχήμα Σ1.10: Διάγραμμα του SoC επεξεργασίας εικόνας.

Στο Σχήμα Σ1.10 δείχνουμε τις μονάδες που διασυνδέονται για την υλοποίηση του συστήματος επεξεργασίας εικόνας. Στο σύστημα περιλαμβάνονται ο επεξεργαστής AVR και ο επεξεργαστής ειδικού σκοπού που αποτελείται από μία μηχανή καταστάσεων που υλοποιεί τα βήματα επεξεργασίας του αλγορίθμου και μία αριθμητική μονάδα που εκτελεί τον υπολογισμό της τετραγωνική ρίζας για την εύρεση της Ευκλείδειας απόστασης. Ο αλγόριθμος του Sobel εφαρμόζει δύο φίλτρα σε μορφή πινάκων μεγέθους 3x3 για την ανίχνευση των οριζόντιων και κάθετων ακμών μιας εικόνας. Για τον υπολογισμό των εικονοστοιχείων της τελικής επεξεργασμένης εικόνας, γίνεται χρήση του αλγορίθμου COordinate Rotation DIgital Computer (CORDIC) [91], [92], για τον υπολογισμό της τετραγωνικής ρίζας και την εύρεση της Ευκλείδειας απόστασης μεταξύ των αντίστοιχων εικονοστοιχείων των εικόνων που δημιουργήθηκαν με τη χρήση των δύο εφαρμοζόμενων φίλτρων. Η τελική εικόνα αποστέλλεται στον H/Y μέσω της σειριακής σύνδεσης.

Η υλοποίηση του αλγορίθμου CORDIC γίνεται με τη χρήση έτοιμης μονάδας από την βιβλιοθήκη CoreLib της Xilinx [48]. Όπως φαίνεται και στο Σχήμα Σ1.10 η μονάδα του αλγορίθμου CORDIC μαζί με τη μηχανή καταστάσεων που εκτελεί τον αλγόριθμο του Sobel επικοινωνούν με τον επεξεργαστή AVR μέσω μίας γέφυρας δεδομένων. Η μονάδα CORDIC χρησιμοποιείται σε μορφή αρχείου netlist από τη βιβλιοθήκη της Xilinx και αρχικοποιείται μέσω της ενσωμάτωσης της μονάδας στις σχετικές βιβλιοθήκες του SysPy.

Στα Code Examples 6.1, 6.2 και 6.3 παρουσιάζεται η περιγραφή σε Python του συστήματος επεξεργασίας εικόνας. Ο επεξεργαστής AVR συνδέεται με τις υπόλοιπες μονάδες κάνοντας χρήση τριών εισόδων/εξόδων γενικού σκοπού (GPIO: General Purpose Input Output), χρησιμοποιώντας την `PORTE` σαν δίαυλο ελέγχου και τις `PORTA` και `PORTB` σαν διαύλους δεδομένων. Για την υλοποίηση του συστήματος κάναμε χρήση της πλακέτας FPGA ML509 [25], της Digilent, η οποία διαθέτει την μεσαία μεγέθους μονάδα FPGA Virtex-5 XC5VLX110T-1. Μία μονάδα διαιρέτη ρολογιού χρησιμοποιείται για να παράγει το βασικό σήμα ρολογιού των 25MHz, από το σήμα των 100MHz που παρέχει η γεννήτρια ρολογιού στην πλακέτα. Το σήμα ρολογιού χρησιμοποιείται για τον χρονισμό του επεξεργαστή AVR και της μονάδας επεξεργαστή ειδικού σκοπού του αλγορίθμου Sobel.

Στην περιγραφή Python αναφέρονται επίσης τα αρχεία C που χρησιμοποιούνται για τον προγραμματισμό του επεξεργαστή. Με χρήση του εργαλείου avr-gcc, που καλείται αυτόματα από το SysPy, γίνεται η μεταγλώττιση του προγράμματος και το SysPy αντιγράφει τον δεκαεξαδικό (hex code) του προγράμματος στις σχετικές μνήμης BRAM που υλοποιούν τη μνήμη προγράμματος του πυρήνα AVR. Όλα τα αρχεία Python που περιγράφουν το σύστημα, καθώς και τα παραγόμενα VHDL αρχεία, υπάρχουν στην ιστοσελίδα [88] που υπάρχει για την περιγραφή των δυνατοτήτων και χαρακτηριστικών του SysPy.



Σχήμα Σ1.11: Επεξεργασμένες εικόνες μεγέθους 64x64 εικονοστοιχείων.

Κατά τη χρονική περίοδο σχεδίασης του SoC δεν ήταν ακόμα διαθέσιμα οι δυνατότητες προσομοίωσης υψηλού επιπέδου που παρέχει το SysPy και συνεπώς η προσομοίωση των λειτουργιών του SoC, πριν την υλοποίηση του στο FPGA, έγινε μόνο με χρήση των προσομοιωτών Modelsim [66] και Xilinx ISE (ISim) [50]. Ο έλεγχος της σωστής λειτουργίας έγινε και στο επίπεδο υλοποίησης του συστήματος στη πλακέτα του FPGA, με χρήση εικόνων μεγέθους 64x64 εικονοστοιχείων. Η αποστολή των εικόνων και η λήψη των επεξεργασμένων αποτελεσμάτων έγινε με χρήση προγράμματος διεπαφής που αναπτύχθηκε με τη γλώσσα Matlab και εκτελείται στον Η/Υ που συνδέεται με την πλακέτα FPGA. Στο Σχήμα Σ1.11 παρουσιάζονται τα αποτελέσματα επεξεργασίας εικόνων μεγέθους 64x64 εικονοστοιχείων, όπου έχει γίνει ανίχνευση των ακμών με χρήση του SoC.

Τα αποτελέσματα δέσμευσης των πόρων του FPGA από το σύστημα επεξεργασίας εικόνων παρουσιάζονται στον Πίνακα Π1.2. ΓΙα την εκτέλεση του αλγορίθμου CORDIC καταλαμβάνονται οχτώ DSP48 μονάδες, η οποία περιλαμβάνει υλοποιήσεις πολλαπλασιαστών. Μονάδες μνήμης BRAM καταλαμβάνονται για την υλοποίηση των μνημών προγράμματος και δεδομένων του επεξεργαστή. Συνολικά για την υλοποίηση όλου του συστήματος χρησιμοποιήθηκαν 367 μονάδες προγραμματιζόμενη λογικής (CLB: Configurable Logic Block). Κάθε CLB περιέχει οχτώ μονάδες Look Up Table (LUT) των έξι εισόδων, ενώ η μονάδες BRAM έχουν μέγεθος 36kbit. Σύμφωνα με τις αναφορές σχεδίασης μετά τη φυσική σχεδίαση του συστήματος (Placement and Routing), το σύστημα μπορεί να χρονιστεί σε συχνότητα 190MHz. Σύμφωνα επίσης με μετρήσεις στο FPGA με τη βοήθεια της μονάδα logic analyzer ChipScope Pro [52] της Xilinx, το σύστημα μπορεί να επεξεργαστεί 10 εικόνες/sec. μεγέθους 64x64 εικονοστοιχείων, όπου ο χρόνος επεξεργασίας μετρήθηκε από τη στιγμή που ξεκινάει η αποστολή της εικόνας από τον Η/Υ μέχρι να καταχωρηθεί επίσης στον Η/Υ η τελική επεξεργασμένη εικόνα.

Components	**CLBs**	**BRAMs**	**DSP48**
Sobel accelerator + sqrt	47	0	8
AVR Processor soft core	267	48	0
Bridge	3	0	0
Clock divider	5	0	0

Πίνακας Π1.2: Αποτελέσματα δέσμευσης λογικών πόρων στη μονάδα FPGA Virtex-5 LX110T (CLB: δύο slices, Slice: τέσσερα 6- εισόδων LUTs, BRAM: 36Kb, DSP48: 25x18-bit).

Μέσω της σχεδίασης του συστήματος επεξεργασίας σήματος δοκιμάσαμε τις βασικές λειτουργίες σχεδίασης του SysPy. Πιο συγκεκριμένα ελέγξαμε την ορθή λειτουργία των μηχανισμών: α) μετατροπής περιγραφών Python σε VHDL σε επίπεδο περιγραφής RTL, β) χρήση πυρήνων επεξεργαστών και διασύνδεση τους με λογικές μονάδες ειδικού σκοπού, γ) χρήση ψηφιακών μονάδων πολλαπλών μορφών (Python, VHDL, netlist) και δ) χρήση μέσω του SysPy εργαλείων ανάπτυξης λογισμικού. Μετά την επιτυχή σχεδίαση του συστήματος επεξεργασίας εικόνας, όσον αφορά την ανάπτυξη του SysPy, προχωρήσαμε στη ενσωμάτωση στο εργαλείο της δυνατότητας χρήσης επεξεργαστών 32-bit, με απώτερο στόχο τη σχεδίαση πολυπλοκότερων συστημάτων. Προδιαγράψαμε επίσης τις δυνατότητες προσομοίωσης που θέλαμε να υποστηρίζει το SysPy και τον τρόπο υλοποίησης τους με χρήση της Python.

1.4.2 SoC προσομοίωσης βιολογικών δικτύων

Η πρώτη προσθήκη στο SysPy, μετά την ανάπτυξη του SoC επεξεργασίας εικόνας, ήταν η χρήση του πυρήνα επεξεργαστή 32-bit Leon3 και των εργαλείων ανάπτυξης λογισμικού που υποστηρίζει. Με τη χρήση του Leon3 ήταν δυνατή η επίτευξη ταχύτερης μεταφοράς δεδομένων μεταξύ του επεξεργαστή και άλλων λογικών μονάδων στο FPGA. Επίσης μέσω κατάλληλων προγραμμάτων οδήγησης (drivers) κατέστη εφικτή α) η διασύνδεση του επεξεργαστή με μονάδες μνήμης τύπου SDRAM διαθέσιμες στην πλακέτα που χρησιμοποιήσαμε και β) η επικοινωνίας του επεξεργαστή με τον Η/Υ μέσω δικτύου Ethernet με χρήση του κατάλληλου ελεγκτή δικτύου στην πλακέτα του FPGA. Για την επίδειξη των νέων δυνατοτήτων σχεδίασης υλοποιήθηκε ένα SoC που προσομοιώνει στο υλικό μοντέλα δικτύων βιοχημικών αντιδράσεων (biochemical reaction networks: BioModels [57], [17]). Η επιτάχυνση της προσομοίωσης τέτοιων μοντέλων στο υλικό είναι πολύ χρήσιμη στον τομέα της υπολογιστικής βιολογίας (computational biology), όπου έως τώρα η προσομοίωση μοντέλων γίνεται σχεδόν αποκλειστικά με τη χρήση λογισμικού.

Ο Leon3 είναι ένας αρκετά δημοφιλής πυρήνας επεξεργαστή 32-bit και παρέχεται σε περιγραφή VHDL από την εταιρεία Aeroflex Gaisler. Ο επεξεργαστής υποστηρίζει την αρχιτεκτονική SPARC V8, η υλοποίηση του σε VHDL υποστηρίζει μεγάλο βαθμό παραμετροποίησης ενώ παρέχεται μαζί με εργαλεία προσομοίωσης και αποσφαλμάτωσης (debugger) λογισμικού. Ο επεξεργαστής αποτελεί πυρήνα της βιβλιοθήκης GRLIB IP [33], στην οποία συγκαταλέγεται πλήθος παραμετροποιήσιμων πυρήνων, που μπορούν να συνδεθούν στον Leon σαν περιφερειακές μονάδες, κάνοντας χρήση του πρωτοκόλλου επικοινωνίας Advanced Microcontroller Bus Architecture (AMBA) [13] που υποστηρίζει ο επεξεργαστής. Ο Leon3 έχει υλοποιηθεί σε αρκετές οικογένειες FPGA (μεταξύ αυτών σε FPGA των εταιρειών Altera και Xilinx) καθώς και σε ολοκληρωμένα κυκλώματα τύπου ASIC. Μία τυπική υλοποίηση του επεξεργαστή απαιτεί περίπου 25k-30k λογικές πύλες. Τα κύρια χαρακτηριστικά του επεξεργαστή αναφέρονται στη λίστα που ακολουθεί:

- Σωλήνωση διαύλου δεδομένων 7-σταδίων.
- Ενσωμάτωση μονάδων διαιρέτη και πολλαπλασιαστή, με χρήση αντίστοιχων εντολών.
- Υποστήριξη αρχιτεκτονικής Harvard.
- Χρονισμός λειτουργίας στα 125MHz και στα 400MHz σε υλοποίηση FPGA και ASIC 0.13μm αντίστοιχα.
- Συμβατότητα με το πρωτόκολλο AMBA-2.0 AHB για τη σύνδεση περιφερειακών μονάδων.
- Διάθεση πλήθους εργαλείων για την ανάπτυξη λογισμικού στον επεξεργαστή: μεταγλωττιστές, προσομοιωτές και εργαλεία για την αποσφαλμάτωση λογισμικού.

Το απαραίτητο λογισμικό για τον Leon αναπτύχθηκε σε γλώσσα C, με χρήση του μεταγλωττιστή sparc-elf-gcc. Για την υλοποίηση του Soc έγινε χρήση της αναπτυξιακής πλακέτας ML509 της Digilent. Για τη μνήμη προγράμματος και τη μνήμη δεδομένων έγινε χρήση της διαθέσιμης στην πλακέτα 256MB SDRAM DDR2 μνήμης. Επίσης χρησιμοποιήθηκε ο σχετικός ελεγκτής στην πλακέτα για τη σύνδεση του επεξεργαστή με Η/Υ μέσω δικτύου Ethernet με ταχύτητα μετάδοσης δεδομένων στα 100Mbps. Η υποστήριξη από τον επεξεργαστή Leon δικτύωσης Ethernet και πρόσβασης σε μεγάλου μεγέθους μνήμες τύπου SDRAM, μας έδωσε τη δυνατότητα σχεδίασης ενός SoC το οποίο είναι σε θέση να συνδυάσει ένα επεξεργαστή γενικού και έναν επεξεργαστή ειδικού σκοπού με στόχο την γρήγορη επεξεργασία μεγάλου όγκου δεδομένων.

Συστήματα επεξεργασίας υψηλών επιδόσεων έχουν χρησιμοποιηθεί την τελευταία δεκαετία σε πολλά επιστημονικά πεδία, ειδικά για την επιτάχυνση της προσομοίωσης πολύπλοκων φυσικών φαινομένων π.χ. μετεωρολογικά και βιολογικά μοντέλα. Η υπολογιστική και συστημική βιολογία είναι επιστημονικά πεδία που επωφελήθηκαν από τη χρήση των νέων υπολογιστικών τεχνικών υψηλών επιδόσεων για την επεξεργασία δεδομένων από βιολογικές βάσεις δεδομένων. Η συστημική βιολογία ερευνά τη δυναμική των βιολογικών συστημάτων σαν ένα σύνολο επιμέρους συστημάτων, εντοπίζοντας έτσι τις αλληλεπιδράσεις μεταξύ των επιμέρους μονάδων και πώς αυτές καθορίζουν τα χαρακτηριστικά ολόκληρου του συστήματος. Προσομοιώνοντας τις αλληλεπιδράσεις διαφορετικών μοριακών ειδών, μπορεί να μελετηθεί η συμπεριφορά τους *in silico* (με τη χρήση υπολογιστών) και να εξαχθούν συμπεράσματα για την συμπεριφορά ενός κυττάρου ή ενός συνόλου κυττάρων. Η στοχαστική προσομοίωση βιοχημικών δικτύων αντιδράσεων που ονομάζονται βιολογικά μοντέλα ή βιομοντέλα (BioModels) [57], [17], μπορεί να οδηγήσει σε συμπεράσματα για τις ιδιότητες ενός βιολογικού συστήματος. Για την αναπαράσταση των μοντέλων γίνεται χρήση της γλώσσας XML και του προτύπου Systems Biology Markup Language (SBML) [42].

Η εξέλιξη ενός βιολογικού συστήματος μπορεί να προσομοιωθεί χρησιμοποιώντας συνήθης διαφορικές εξισώσεις. Η λύση όμως διαφορικών εξισώσεων δεν είναι ο ενδεδειγμένος τρόπος, ειδικά όταν οι ποσότητες των μοριακών ειδών που εμπλέκονται είναι μικρές [69]. Σε αυτή την περίπτωση είναι πολύ δύσκολο να προσομοιωθεί η συμπεριφορά του συστήματος και η χρονική εξέλιξη των χημικών αντιδράσεων. Για την αντιμετώπιση αυτών των προβλημάτων, αντί για διαφορικές εξισώσεις, μπορεί να γίνει χρήση στοχαστικών μοντέλων για την προσομοίωση δικτύων βιοχημικών αντιδράσεων. Η προσομοίωση με χρήση στοχαστικών μοντέλων μπορεί να περιγραφεί ως μίας διαδικασία Markov [36], όπου η επόμενη κατάσταση ενός συστήματος εξαρτάται μόνο από την αμέσως προηγούμενη κατάσταση. Ο D. T. Gillespie πρότεινε έναν βελτιωμένο αλγόριθμο για τη στοχαστική προσομοίωση βιολογικών μοντέλων, ο οποίος ονομάζεται First Reaction Method (FRM) [35].

Το σύστημα το οποίο σχεδιάσαμε δέχεται αρχεία βιολογικών μοντέλων σαν δεδομένα εισόδου και χρησιμοποιεί τον αλγόριθμο FRM για την προσομοίωση των χημικών αντιδράσεων που περιγράφονται στα μοντέλα. Το SoC υλοποιεί την πρωταρχική υλοποίηση του αλγορίθμου FRM, όπως αυτή διατυπώθηκε από τον Gillespie, χωρίς να γίνεται χρήση αριθμητικών προσεγγίσεων στο υλικό, κάτι που συνηθίζεται στις υλοποιήσεις του αλγορίθμου σε λογισμικό και μειώνει την ακρίβεια των αποτελεσμάτων της προσομοίωσης. Έτσι επιταχύνεται σημαντικά η εκτέλεση στοχαστικής προσομοίωσης στο υλικό, με τη χρήση μεθόδων παράλληλης επεξεργασίας, χωρίς να μειώνεται η ακρίβεια των αριθμητικών αποτελεσμάτων.

Με τη χρήση του SysPy για τη σχεδίαση του συστήματος επεξεργασίας βιολογικών δικτύων προσπαθήσαμε να εκτιμήσουμε τις δυνατότητες του εργαλείου και να εμπλουτίσουμε τις ακόλουθες δυνατότητες σχεδίασης που παρέχει:

- αυτόματη παραμετροποίηση και σύνδεση μονάδων υλικού (επεξεργαστής ειδικού σκοπού για την υλοποίηση του αλγορίθμου FRM).
- χρήση των δυνατοτήτων της Python για την επεξεργασία του περιεχομένου αρχείων κειμένου τύπου ASCII (επεξεργασία αρχείων βιομοντέλων XML).
- χρήση της Python για την ανάπτυξη λογισμικού διεπαφής Η/Υ με την υλοποίηση SoC σε μονάδα FPGA (ανάπτυξη λογισμικού HAL: Hardware Abstraction Layer).

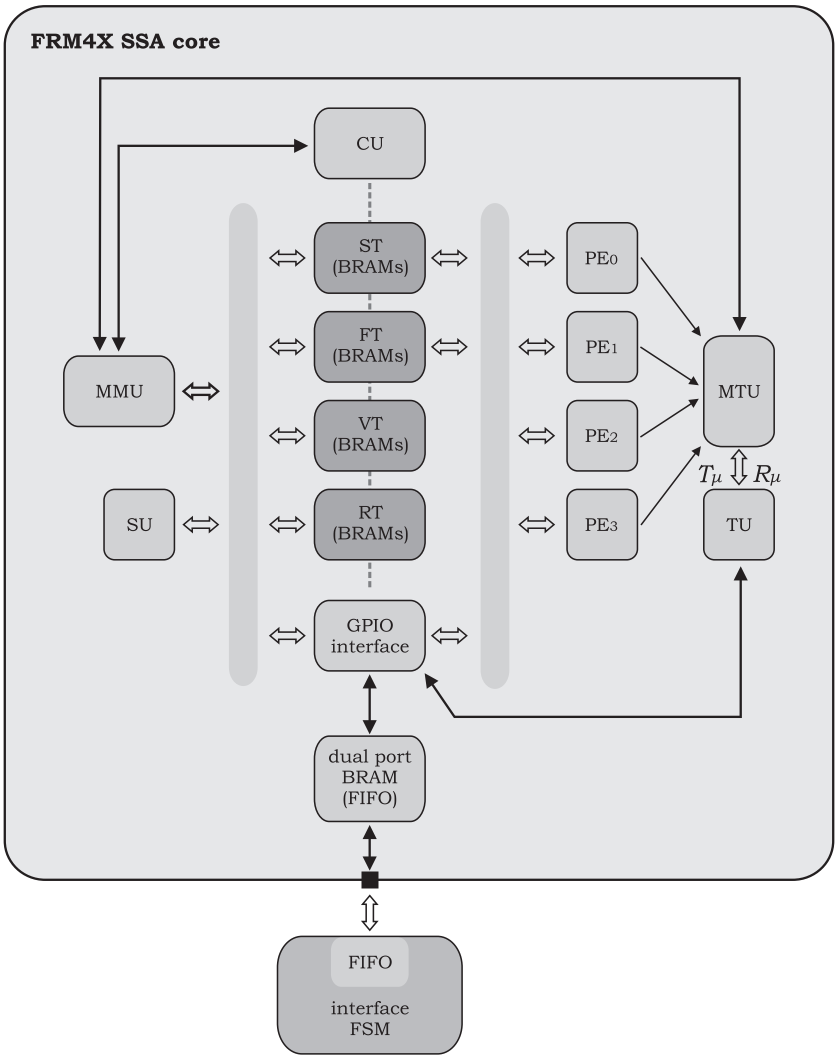
Χρησιμοποιώντας τις δυνατότητες του SyPy υλοποιήσαμε ένα σχέδιο SoC [61] το οποίο συνδυάζει τον πυρήνα του επεξεργαστή Leon3 με τον επεξεργαστή ειδικού σκοπού που σχεδιάστηκε από την ερευνητική μας ομάδα μας [40] και υλοποιεί τον αλγόριθμο FRM. Οι δυνατότητες του SysPy ήταν ιδιαίτερα χρήσιμες στην αρχικοποίηση και σύνδεση του επεξεργαστή ειδικού σκοπού, ειδικά στην επεξεργασία των αρχείων XML των βιομοντέλων σε μορφή SBML και στην αρχικοποίηση του μοντέλου στις διαθέσιμες μονάδες μνήμης του SoC.

Η λίστα των διαθέσιμων παραμέτρων του επεξεργαστή FRM παρουσιάζεται στον Πίνακα Π1.3. Υλοποιήσαμε τρεις διαφορετικές εκδόσεις του επεξεργαστή, με έναν (FRM1X), δύο (FRM2X) και τέσσερις (FRM4X) επεξεργαστές (PE: Processing Element) που λειτουργούν παράλληλα (N=1 ή 2 ή 4). Ο αριθμός των χημικών αντιδράσεων m και το πλήθος των χημικών ειδών n εξάγονται αυτόματα από το αρχείο του βιομοντέλου, ενώ οι υπόλοιπος παράμετροι που αναφέρονται στον πίνακα παρέχονται από τον χρήστη. Για κάθε επεξεργαστή FRM απαιτείται η αρχικοποίηση της τυχαίας γεννήτριας αριθμού (r_j) που περιέχουν και χρησιμοποιείται από τον αλγόριθμο FRM. Το σύστημα μπορεί να επεξεργαστεί βιομοντέλα με δύο διαφορετικούς τρόπους: α) προσομοίωση m/N από κάθε επεξεργαστή ειδικού σκοπού (SSIP: Single Simulation In Parallel) και β) προσομοίωση όλων των αντιδράσεων m παράλληλα από όλους του επεξεργαστές N (MSIP: Multiple Simulations In Parallel).

Parameter	**Name**	**Range**
m	Πλήθος αντιδράσεων	$2^e, e \in [0, 12]$
n	Πλήθος χημικών ειδών	$2^e, e \in [0, 12]$
q	Πλήθος αντιδρώντων (βαθμός αντίδρασης)	$[1 - 3]$
N_{rep}	Πλήθος επαναλήψεων προσομοίωσης	
$RNGseed$	Αρχικές τιμές της γεννήτριας τυχαίων αριθμών	$[0 - 255]$
K	Τρόπος λειτουργίας	$[0 = SSIP, 1 = MSIP]$
T_{sim}	Διάρκεια προσομοίωσης (sec.)	

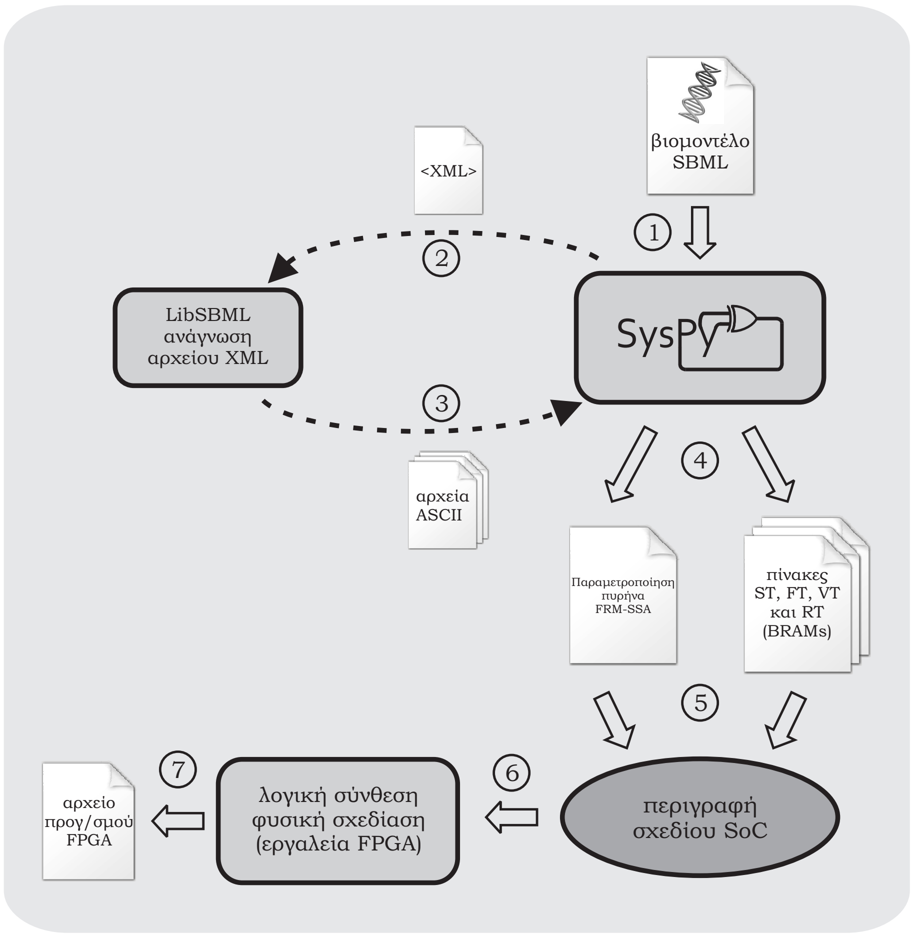
Πίνακας Π1.3: Παράμετροι του επεξεργαστή ειδικού σκοπού υλοποίησης του αλγορίθμου FRM.

Στο Σχήμα Σ1.12 παρουσιάζουμε τη δομή του επεξεργαστή ειδικού σκοπού FRM με τέσσερις επεξεργαστές (FRM4X). Τα βήματα επεξεργασίας του αλγορίθμου υλοποιούνται από την μονάδα ελέγχου (CU: Control Unit). Σε μία σειρά από μονάδες μνήμης καταχωρούνται οι παράμετροι του δικτύου χημικών αντιδράσεων, όπως αυτό περιγράφεται στο αρχείο του βιομοντέλου. Οι μονάδες μνήμης είναι ο πίνακας αντιδράσεων (RT: Reaction Table), ο πίνακας στοιχειομετρίας (VT: Stoichiometry Table), ο πίνακας χημικών ειδών (ST: Species Table) και ο πίνακας χρονοπρογραμματισμού που περιέχει τις τιμές διαφόρων σημάτων ελέγχου (FT: Flags Table). Η μονάδα διαχείρισης μνήμης (MMU: Memory Management Unit) διαχειρίζεται και συγχρονίζει την πρόσβαση σε όλες τις μονάδες μνήμης.



Σχήμα Σ1.12: Δομή του επεξεργαστή ειδικού σκοπού FRM με τέσσερα επεξεργαστικά στοιχεία.

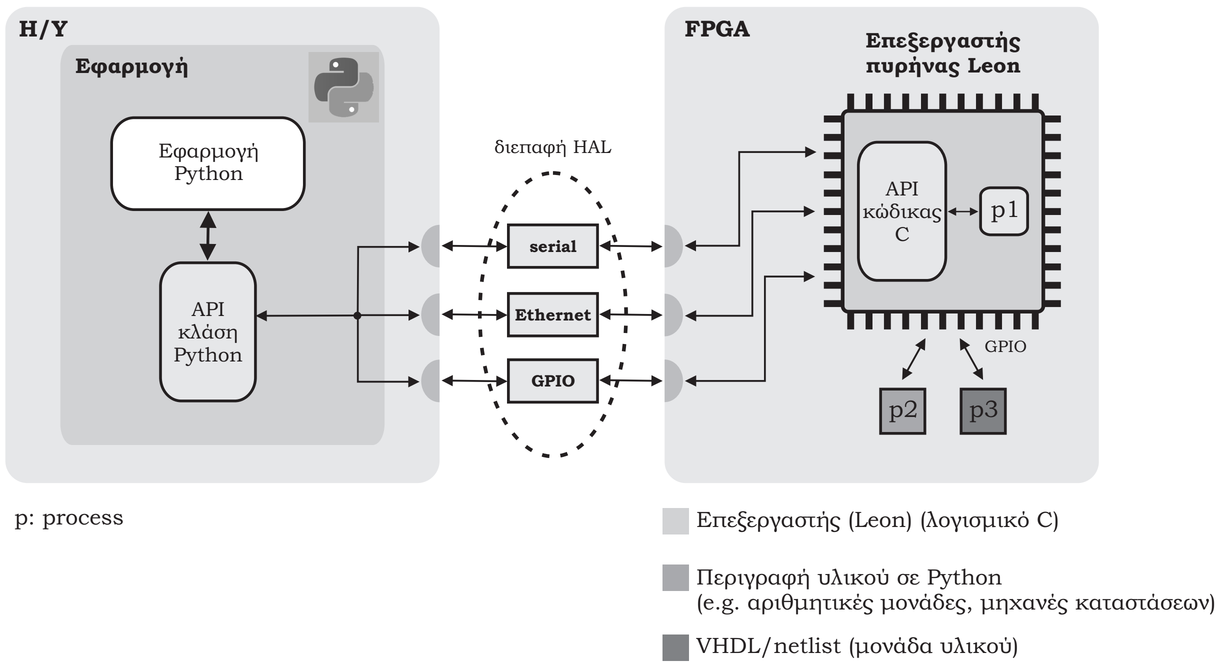
Η επίδοση της επεξεργαστικής ισχύς του επεξεργαστή FRM μετράται σε κύκλους προσομοίωσης ανά δευτερόλεπτο (RC: Reaction Cycles/sec.). Σε κάθε κύκλο προσομοίωσης επεξεργάζονται $10x32-bit = 320-bit$, ενώ υποστηρίζονται βιομοντέλα με μέγιστο αριθμό τριών αντιδρώντων στοιχείων και πέντε παραγώγων αντίδρασης. Με τη χρήση της Python γίνεται δυνατή η επεξεργασία των βιομοντέλων σε μορφή αρχείων XML και η χρήση των δεδομένων που περιέχουν για την αρχικοποίηση μονάδων υλικού, όπως αυτή περιγράφεται και στο Σχήμα Σ1.4. Η διαδικασία εξαγωγής πληροφοριών από τα μοντέλα XML παρουσιάζεται και στο Σχήμα Σ1.13.



βιομοντέλο SBML
<XML>
1
2
LibSBML ανάγνωση αρχείου XML
SysPy
3
αρχεία ASCII
4
Παραμετροποίηση πυρήνα FRM-SSA
πίνακες ST, FT, VT και RT (BRAMs)
5
7
6
αρχείο προγ/σμού FPGA
λογική σύνθεση φυσική σχεδίαση (εργαλεία FPGA)
περιγραφή σχεδίου SoC

Σχήμα Σ1.13: Επεξεργασία βιομοντέλων SBML με στόχο την εξαγωγή πληροφοριών για την αρχικοποίηση του πυρήνα υλοποίησης του αλγορίθμου FRM.

Η δημιουργία της διεπαφής HAL, με λογισμικό C να εκτελείται στον επεξεργαστή Leon στο SoC το οποίο επικοινωνεί μέσω σύνδεσης Ethernet με λογισμικό Python που εκτελείται στον H/Y, ήταν απαραίτητη για: α) να διαχειρίζεται τη ροή των δεδομένων από τον H/Y στο SoC και αντίστροφα και β) να ελέγχει την πρόσβαση του SoC στη μνήμη SDRAM στην πλακέτα του FPGA. Το λογισμικό HAL διαχειρίζεται την ανταλλαγή πληροφοριών στα κανάλια δεδομένων Ethernet (H/Y - SoC), GPIO (Leon - επεξεργαστής FRM ειδικού σκοπού) και στο σειριακό κανάλι ελέγχου μέσω του οποίου χρονίζεται η εκτέλεση του λογισμικού HAL στον Leon και στον H/Y.


H/Y
Εφαρμογή
Εφαρμογή Python
API κλάση Python
διεπαφή HAL
serial
Ethernet
GPIO
FPGA
Επεξεργαστής πυρήνας Leon
API κώδικας C
p1
GPIO
p2
p3
p: process
Επεξεργαστής (Leon) (λογισμικό C)
Περιγραφή υλικού σε Python (e.g. αριθμητικές μονάδες, μηχανές καταστάσεων)
VHDL/netlist (μονάδα υλικού)

Σχήμα Σ1.14: Σύνδεση του H/Y με την πλακέτα FPGA και το SoC επεξεργασία βιολογικών δεδομένων, μέσω της χρήσης του λογισμικού HAL.

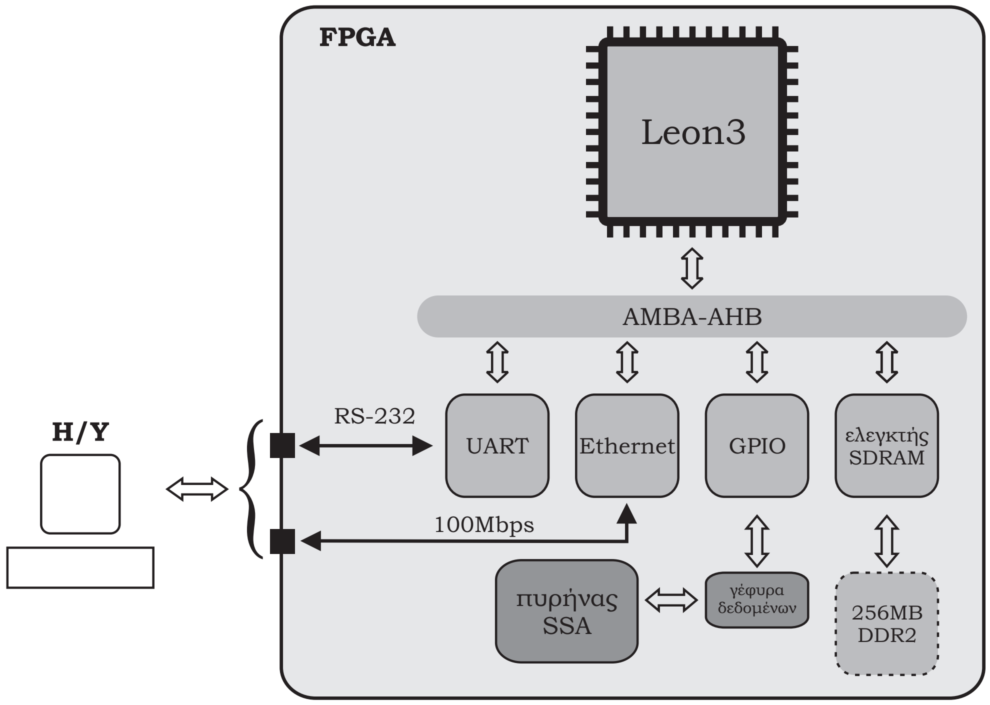
Στο Σχήμα Σ1.14 παρουσιάζεται η σύνδεση του H/Y με την πλακέτα FPGA και το SoC επεξεργασία βιολογικών δεδομένων, μέσω της χρήσης του λογισμικού HAL. Στο σχήμα παρουσιάζεται ο διαχωρισμός του HAL στην υλοποίηση Python στον H/Y και στην υλοποίηση C που εκτελεί ο επεξεργαστής Leon. Η υλοποίηση του HAL παρέχει μεθόδους για την ανταλλαγή δεδομένων και στα τρία διαθέσιμα κανάλια επικοινωνίας του SoC (Ethernet, GPIO, serial connection). Η εκτέλεση των διαθέσιμων μεθόδων στην Python, έχει ως αποτέλεσμα την εκτέλεση των αντίστοιχων μεθόδων C στον επεξεργαστή. Η ανταλλαγή των δεδομένων στο κανάλι Ethernet γίνεται με χρήση πακέτων δεδομένων τύπου MAC με ταχύτητα 100Mbps, ενώ με συχνότητα λειτουργίας του επεξεργαστή στα 160MHz, η ανταλλαγή δεδομένων στο κανάλι GPIO γίνεται με ταχύτητα 25Mbps.

Με την ανάπτυξη του λογισμικού HAL παρουσιάσαμε μία ολοκληρωμένη μεθοδολογία χρήσης μιας αντικειμενοστραφούς γλώσσας όπως η Python για την διαχείριση επεξεργασίας δεδομένων από ένα ενσωματωμένο σύστημα. Με τη χρήση κλάσεων και μεθόδων γίνεται εφικτή η δέσμευση επεξεργαστικών στοιχείων και η δέσμευση καναλιών επικοινωνίας στην πλακέτα του FPGA. Προσπάθεια χρήσης αντικειμενοστραφούς γλώσσας για δέσμευση πόρων σε ενσωματωμένο σύστημα έχει γίνει και στο παρελθόν με τη χρήση της γλώσσας Java [82]. Η ανάπτυξη του HAL για τις ανάγκες χρήσης του SysPy, είναι η πρώτη προσπάθεια να χρησιμοποιηθεί μια ευέλικτη και δημοφιλής γλώσσα όπως η Python για την διαχείριση της επεξεργασίας δεδομένων σε ένα ενσωματωμένο SoC. Η χρήση της

	Leon3	Leon+FRM1X	Leon+FRM2X	Leon+FRM4X
Slices	5,436 (31%)	9,244 (53%)	13,214 (76%)	16,594 (96%)
BRAMs	17 (11%)	56 (37%)	78 (52%)	132 (89%)
MULs	0 (0%)	16 (25%)	26 (41%)	48 (75%)
Power (W)	0.6	4.1	4.8	5.9

Πίνακας Π1.4: Αποτελέσματα δέσμευσης λογικών πόρων στη μονάδα FPGA Virtex-5 LX110T (CLB: δύο slices, Slice: τέσσερα 6- εισόδων LUTs, BRAM: 36Kb, DSP48: 25x18-bit)

Python δίνει ευελιξία στην διαχείριση των λειτουργιών ενός SoC και επίσης δίνει δυνατότητα για προεπεξεργασία των δεδομένων που αποστέλλονται στο ενσωματωμένο σύστημα και χρονισμό των βημάτων επεξεργασίας τους στην πλακέτα του FPGA, με χρήση προγραμμάτων Python script στον Η/Υ.



Σχήμα Σ1.15: Σύνδεση του επεξεργαστή Leon με τον επεξεργαστή FRM ειδικού σκοπού.

Στο Σχήμα Σ1.15 παρουσιάζεται η τοπολογία του SoC με χρήση της πλακέτα FPGA ML509. Ο ελεγκτής Ethernet και η μνήμη SDRAM χρονίζονταν στα 190MHz, ενώ για τον επεξεργαστή Leon έγινε χρήση σήματος ρολογιού στα 160MHz. Τα σήματα ρολογιού παρήχθησαν με χρήση του διαθέσιμου ταλαντωτή των 100MHz στην πλακέτα και μονάδας διαχείρισης σημάτων ρολογιού (DCM: Digital Clock Manager) στο FPGA. Η σύνδεση του Η/Υ και της πλακέτας FPGA έγινε μέσω σειριακού καλωδίου RS-232 και καλωδίου δικτύου Ethernet. Για τον έλεγχο της λειτουργίας του συστήματος έγινε χρήση ενός πολύπλοκου βιομοντέλου [78], το οποίο περιγράφει ένα βιοχημικό δίκτυο με $n = 93$ είδη και $m = 136$ αντιδράσεις. Το συγκεκριμένο μοντέλο περιγράφει την συμπεριφορά της πρωτεΐνης Α-συνουκλεΐνης, η οποία σχετίζεται με τη νόσο του Πάρκινσον. Τα αποτελέσματα της δέσμευσης πόρων στο FPGA μετά την υλοποίηση του SoC με ένα (FRM1X), δύο (FRM2X) και τέσσερα (FRM4X) επεξεργαστικά στοιχεία, παρουσιάζονται στον Πίνακα Π1.4. Όσο αυξάνεται η πολυπλοκότητα ενός βιομοντέλου (μεγαλύτερο πλήθος στοιχείων και αντιδράσεων) τόσο αυξάνονται και οι απαιτήσεις στο υλικό, ειδικά σε σχέση με τον αριθμό των διαθέσιμων μονάδων BRAM στο FPGA. Παρατηρούμε επίσης ότι ο επεξεργαστής Leon δεσμεύει έναν σχετικά μικρό αριθμό λογικών πόρων (περίπου το 1/3 των λογικών μονάδων και το 11% των μονάδων μνήμης). Επίσης ο επεξεργαστής καταναλώνει μόλις το 10% της συνολικής ισχύος του κυκλώματος, ενώ την μεγαλύτερη ισχύ καταναλώνουν οι αριθμητικές μονάδες και οι μονάδες μνήμης. Συνεπώς η χρήση του επεξεργαστή, βοηθάει πάρα πολύ στη δέσμευση και χρήση μονάδων επικοινωνίας (Ethernet) και μονάδων μνήμης (SDRMA), ενώ ταυτόχρονα καταλαμβάνει μικρό μέρος των διαθέσιμων πόρων και καταναλώνει επίσης μικρό μέρος της ισχύος ενός FPGA.

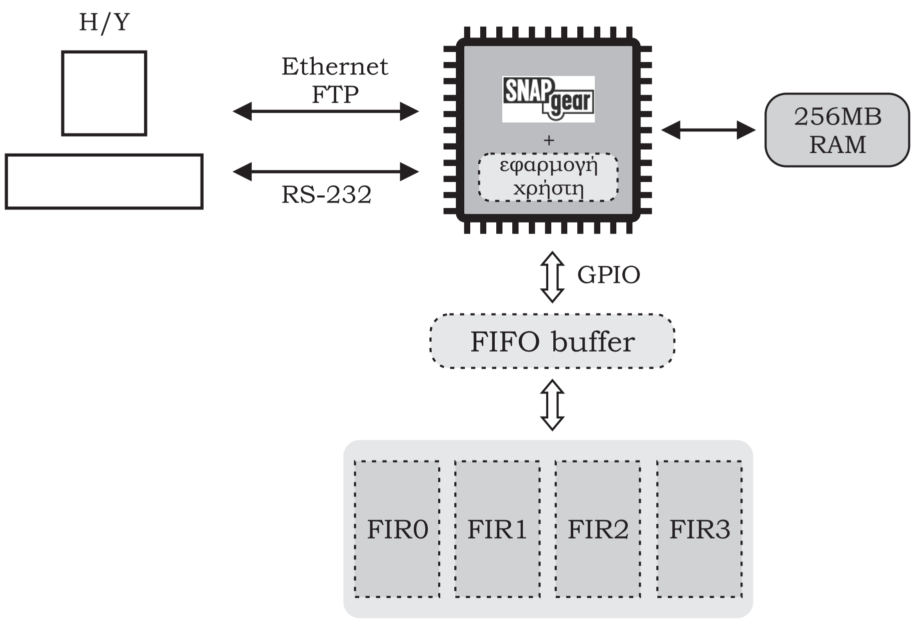
Για να εκτιμήσουμε τις επεξεργαστικές δυνατότητες του SoC, συγκρίναμε το ρυθμό επεξεργασίας και προσομοίωσης δεδομένων που καταγράψαμε στο υλικό με αυτόν που αντίστοιχα καταγράψαμε κάνοντας χρήση δημοφιλών εργαλείων λογισμικού προσομοίωσης δικτύων βιοχημικών αντιδράσεων. Τα εργαλεία λογισμικού που χρησιμοποιήσαμε είναι το iBioSim [70] και το StochPy [85] και εκτελέσαμε προσομοιώσεις με χρήση σύγχρονων υπολογιστικών μονάδων (64-bit PC, 6GB RAM, Intel i7, 2.6GHz, quad-core CPU). Εκτελώντας πλήθος προσομοιώσεων καταγράψαμε $0.35 MReactions/sec.$ ρυθμός προσομοίωσης στο υλικό, απόδοση που είναι περίπου 50 φορές μεγαλύτερη από την απόδοση των iBioSim και StochPy. Εκτός του ότι η υλοποίηση στο FPGA επιταχύνει δραματικά την προσομοίωση των βιομοντέλων, παρέχει επίσης μια πλατφόρμα προσομοίωσης πολύ μικρότερου μεγέθους και με πολύ χαμηλή κατανάλωση ισχύος, σε σύγκριση με υλοποιήσεις λογισμικού που απαιτούν τη χρήση μεγάλων υπολογιστικών μονάδων για να επιτύχουν συγκρίσιμους ρυθμούς επεξεργασίας δεδομένων.

1.4.3 SoC επεξεργασίας ήχου

Με τη χρήση του SysPy και του επεξεργαστή Leon καταφέραμε να σχεδιάσουμε ένα αρκετά πολύπλοκο SoC και να αναδείξουμε τις δυνατότητες του εργαλείου στη χρήση σε ένα ενσωματωμένο σύστημα ενός επεξεργαστή 32-bit σαν πύλη επικοινωνίας με άλλες μονάδες στην πλακέτα του FPGA αλλά και εκτός αυτής. Ειδικότερα η χρήση του Leon μας επέτρεψε να συνδέσουμε το SoC και την πλακέτα FPGA σε δίκτυο Ethernet όπου λειτουργούσε σαν συνεπεξεργαστής του Η/Υ που επικοινωνούσε μέσω δικτύου. Στα πλαίσια της σχεδίασης ενός SoC επεξεργασία ήχου, δύο σημαντικές προσθήκες έγιναν στο SysPy. Αναπτύχθηκε ο μηχανισμός προσομοίωσης που δίνει τη δυνατότητα προσομοίωσης ενός SoC με χρήση περιγραφών υψηλού επιπέδου. Επίσης ενσωματώθηκε η χρήση λειτουργικού συστήματος Linux για την ανάπτυξη εφαρμογών σε γλώσσα C στον επεξεργαστή Leon. Η χρήση του λειτουργικού συστήματος δίνει τη δυνατότητα επικοινωνίας του FPGA με τον Η/Υ και ανταλλαγής δεδομένων σε επίπεδο αρχείων, μέσω του πρωτοκόλλου μεταφοράς δεδομένων File Transfer Protocol (FTP). Επίσης δίνεται η δυνατότητα σύνδεσης του FPGA στο δικτύου Ethernet με χρήση διεύθυνσης IP, δίνοντας πρόσβαση στη μονάδα του FPGA σε πλήθος εφαρμογών που χρησιμοποιούν τη συγκεκριμένη μέθοδο διευθυνσιοδότησης.

Το διάγραμμα του νέου σχεδίου SoC παρουσιάζεται στο Σχήμα Σ1.16. Ο επεξεργαστής Leon λειτουργεί ως FTP client, όπου εκτελείται επίσης η εφαρμογή του χρήστη που επεξεργάζεται τα μουσικά αρχεία που λαμβάνονται από τον Η/Υ. Με τη χρήση του λειτουργικού συστήματος οι εφαρμογές του χρήστη μεταγλωττίζονται μαζί με τον πυρήνα του λειτουργικού Snapgear Linux. Ο στόχος της σχεδίασης του SoC είναι η ταξινόμηση των μουσικών αρχείων σε διαφορετικές κατηγορίες, σύμφωνα με τον τον τύπο της μουσικής που περιέχουν (π.χ. ηλεκτρονική, ροκ ή κλασσική μουσική). Η ταξινόμηση γίνεται σύμφωνα με το συχνοτικό περιεχόμενο των αρχείων, το οποίο εκτιμάται μέσω τεσσάρων ζωνοπερατών (bandpass) φίλτρων τύπου Finite Impulse Response (FIR), τα οποία είναι συνδεδεμένα σαν περιφερειακές μονάδες του επεξεργαστή. Τα αρχεία καταχωρούνται στη μνήμη SDRAM στην πλακέτα του FPGA και από εκεί προωθούνται προς τις μονάδες των φίλτρων οι οποίες συνδέονται μέσω θυρών GPIO με τον επεξεργαστή. Σειριακή σύνδεση μεταξύ του Η/Υ και της πλακέτας FPGA χρησιμοποιείται για να παρέχει πρόσβαση στη γραμμή εντολών του λειτουργικού συστήματος Linux, από όπου ο χρήστης ελέγχει την εκτέλεση της εφαρμογής του και την ανταλλαγή δεδομένων μέσω του πρωτοκόλλου FTP.

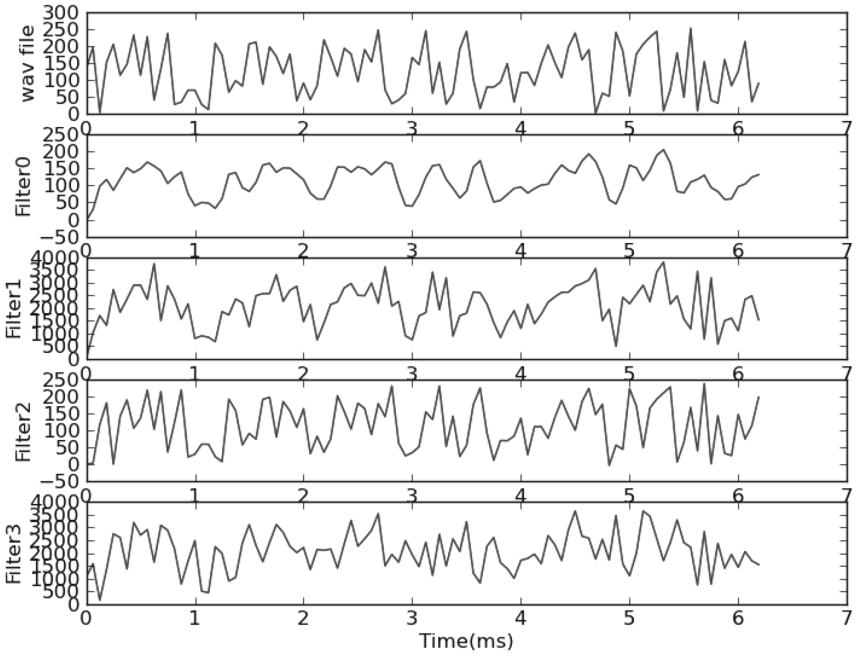
Ξεκινήσαμε τη σχεδίαση του συστήματος με την προδιαγραφή μοντέλων για την προσομοίωση του συστήματος, συμβατά με τον μηχανισμό προσομοίωσης που αναπτύχθηκε στο SysPy και τα χαρακτηριστικά του περιγράφονται στο Κεφάλαιο . Στο Σχήμα Σ1.16 παρουσιάζονται με διακεκομμένες γραμμές τα στοιχεία εκείνα του λογισμικού και του υλικού για τα οποία δημιουργήσαμε μοντέλα προσομοίωσης στο SysPy.Τα αποτελέσματα της προσομοίωσης σε υψηλό επίπεδο με τη χρήση μοντέλων του συστήματος σε γλώσσα Python, βοήθησαν στο να πάρουμε σημαντικές αποφάσεις σχετικά με την αρχιτεκτονική του συστήματος όπως: α) τις παραμέτρους των φίλτρων, π.χ. βαθμός φίλτρων,



Σχήμα Σ1.16: Διάγραμμα μονάδων του SoC επεξεργασίας ήχου.

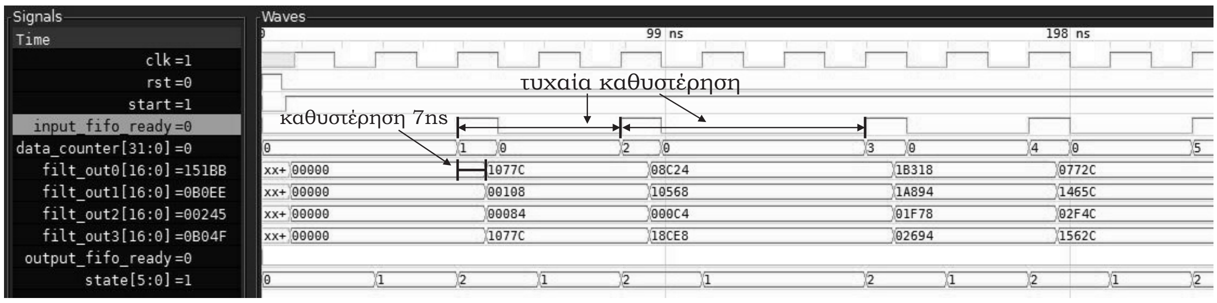
μορφή αναπαράστασης αριθμών κτλ., β) το μέγεθος των μνημών FIFO (First In First Out) προσωρινής καταχώρησης δεδομένων (data buffers) μεταξύ του επεξεργαστή και των φίλτρων και γ) την ανάπτυξη του λογισμικού ελέγχουν που εκτελείται στον επεξεργαστή.

Τα μουσικά αρχεία είναι καταχωρημένα στον H/Y που χρησιμοποιείται σαν file server. O FTP client στο Leon επιλεγεί τα αρχεία που θα σταλούν από τον H/Y στο FPGA και τα καταχωρεί στη μνήμη SDRAM. Μετά αναλαμβάνει την μετάδοση των μουσικών τιμών των αρχείων στα φίλτρα, μέσω των θυρών GPIO, για τις οποίες αναπτύξαμε ειδικό λογισμικό οδήγησης σε γλώσσα C, ώστε το λειτουργικό σύστημα να τις διαχειρίζεται σαν περιφερειακές μονάδες προσβάσιμες από συγκεκριμένες διευθύνσεις μνήμης (memory mapped). Για την ανάλυση του συχνοτικού περιεχομένου των αρχείων υλοποιήσαμε τέσσερα ζωνοπερατά (bandpass) φίλτρα FIR 30 παραμέτρων (30-taps) με ζώνες διέλευσης 0-1KHz, 1-3KHz, 3-5KHz και 5-8KHz που καλύπτουν το μεγαλύτερο μέρος του ακουστικού φάσματος συχνοτήτων. Οι φιλτραρισμένες τιμές καταχωρούνται πάλι στη μνήμη RAM, όπου ο επεξεργαστής τις αναλύει και αποφαίνεται για τον τύπου του μουσικού αρχείου (rock, pop, κλασσική ή ηλεκτρονική μουσική). Οι ζώνες διέλευσης των φίλτρων παρουσιάζονται στο Figure 8.3. Για τον υπολογισμό των παραμέτρων των φίλτρων κάναμε χρήση των διαθέσιμων μεθόδων στο SciPy, όπως επίσης κάναμε και χρήση των σχετικών μεθόδων για την γραφική αναπαράσταση της επεξεργασίας μουσικών αρχείων με τη χρήση των φίλτρων. Στο Σχήμα Σ1.17 παρουσιάζουμε τη χρονική απόκριση (time response) των τεσσάρων φίλτρων, όπως αυτή παράγεται αυτόματα από το SciPy με την εκτέλεση των διαθέσιμων μοντέλων στο SysPy. Σαν είσοδο του συστήματος χρησιμοποιήσαμε ένα αρχείο με 100 δείγματα ήχου και συχνότητα δειγματοληψίας 16kHz (sampling frequency).

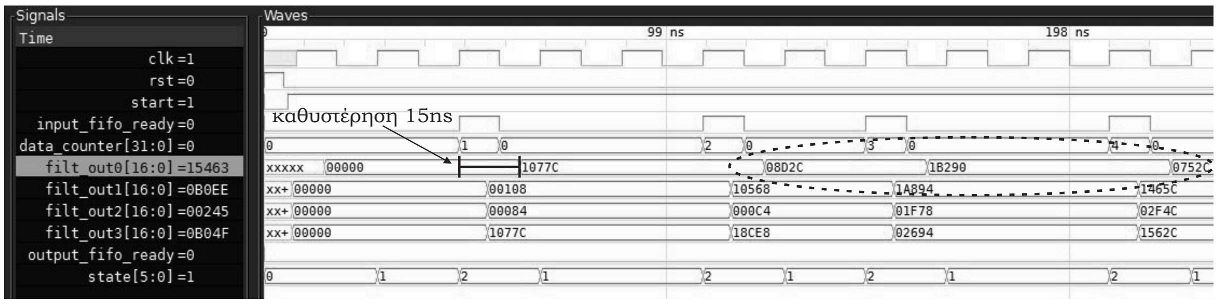


Σχήμα Σ1.17: Χρονική απόκριση των μοντέλων των τεσσάρων φίλτρων, με χρήση του SciPy.

Μέσω του SysPy μπορέσαμε επίσης με τη χρήση ψευδοαλγορίθμων Python να προδιαγράψουμε τη λειτουργικότητα του λογισμικού του Leon που ελέγχει τη ροή των δεδομένων στο SoC. Στο Σχήμα Σ1.18 παρουσιάζεται η συμπεριφορά των σημάτων εισόδου/εξόδου του SoC. Με τη χρήση περιγραφών υψηλού επιπέδου Python μπορέσαμε να περιγράψουμε τη λειτουργία του λογισμικού του Leon, όπου σε σύγκριση με μία υλοποίηση στο υλικό, μπορεί να υπάρχουν καθυστερήσεις στην εκτέλεση ενός προγράμματος. Έτσι στο Σχήμα Σ1.18α η ενεργοποίηση του σήματος `input_fifo_ready` γίνεται με τυχαίο τρόπο, προσομοιώνοντας τον τρόπο με τον οποίο ο επεξεργαστής ελέγχει για την ύπαρξη νέων δεδομένων στις μνήμες FIFO, όπου καταχωρούνται τα αποτελέσματα επεξεργασίας των τεσσάρων φίλτρων. Επίσης με τη χρήση των μοντέλων προσομοίωσης ήταν εύκολο να αλλάζουμε τις παραμέτρους των φίλτρων και να παρατηρούμε την αλλαγή στη συμπεριφορά του συστήματος. Στο Σχήμα Σ1.18β αλλάζουμε την ζώνη διέλευσης του πρώτου φίλτρου, από 0-1KHz σε 0-100Hz και παρατηρούμε την αλλαγή στις φιλτραρισμένες τιμές εξόδου.



(α)



(β)

Σχήμα Σ1.18: Ψηφιακές κυματομορφές σημάτων εισόδου/εξόδου του SoC επεξεργασίας ήχου.

Μέσω της χρήσης μοντέλων προσομοίωσης θα πρέπει να παρέχεται μέθοδος για την μετατροπή τους σε κατάλληλες περιγραφές για υλοποίηση στο υλικό. Οποιαδήποτε περιγραφή υλικού σε επίπεδο RTL σε Python μετατρέπεται από το SysPy σε αντίστοιχη περιγραφή VHDL, ενώ μέθοδοι Python χρησιμοποιούνται για την παραγωγή του κώδικα VHDL των φίλτρων FIR. Στο Παράδειγμα Κώδικα **??** η μέθοδος `func_fir_filt_s` χρησιμοποιείται για να παράγει τον κώδικα VHDL ενός φίλτρου FIR, σύμφωνα με τις παραμέτρους που υπολογίστηκαν κατά την προσομοίωση. Στις γραμμές 5-6 παρέχονται οι παράμετροι του φίλτρου και η αριθμητική μορφή αναπαράστασης fixed-point των παραμέτρων. Οι παράμετροι μετατρέπονται σε κατάλληλη δυαδική μορφή και οι τιμές τους αρχικοποιούνται στην περιγραφή VHDL που παράγει το SysPy. Τα σήματα εισόδου/εξόδου του φίλτρου περιγράφονται στις γραμμές 17-19.

Με την υλοποίηση του SoC στο FPGA επεξεργαστήκαμε ένα μεγάλο αριθμό μουσικών αρχείων για να ελέγξουμε τη συμπεριφορά του συστήματος και να εξακριβώσουμε το βαθμό ταύτισης των αποτελεσμάτων της επεξεργασίας σε σχέση με τα αποτελέσματα της προσομοίωσης. Ο επεξεργαστής Leon χρονίστηκε στα 160MHz, ενώ για τα φίλτρα και τις μνήμες FIFO χρησιμοποιήθηκε σήμα ρολογιού με συχνότητα 100MHz. Για να μετρήσουμε τις επιδόσεις του συστήματος χρησιμοποιήσαμε ένα μουσικό αρχείο μεγέθους 3,924,170x8-bit. Με τη χρήση του πρωτοκόλλου FPT ο ρυθμός μετάδοσης του αρχείου μέσω της σύνδεσης Ethernet από τον Η/Υ στην πλακέτα του FPGA μετρήθηκε στα

21.9Mbps. Η απόδοση όλου του συστήματος, όσον αφορά την επεξεργασία των μουσικών δειγμάτων από τα φίλτρα ήταν 15MMAC/sec. (MAC: Multiply Accumulate), ενώ η ροή των δεδομένων στα τέσσερα φίλτρα που λειτουργούσαν παράλληλα μετρήθηκε στα 119.6Mbps. Τα αποτελέσματα των επιδόσεων του συστήματος παρουσιάζονται στον Πίνακα Π1.5. Οι επιδόσεις του SoC συγκρίνονται με τα αποτελέσματα υλοποίησης σε λογισμικό στον επεξεργαστή Leon (χωρίς την χρήση μονάδων υλικού), όπου φαίνεται ότι το SoC επιτυγχάνει τέσσερις φορές ταχύτερη επεξεργασία των μουσικών αρχείων.

	SoC	**Leon (υλοποίηση στο λογισμικό)**
Χρόνος μετάδοσης FTP (sec.)	1.4 (21.9Mbps)	-
Χρόνος επεξεργασίας φίλτρων (sec.)	31.5 (15.0 MMACs/sec)	132.3 (3.6 MMACs/sec.)
Ρυθμός μετάδοσης δεδομένων (Mbps)	119.6	-

Πίνακας Π1.5: Αποτελέσματα υλοποίησης κατά τη διάρκεια της επεξεργασίας μουσικού αρχείου (3,924,170 samples x 8-bit).

Με τη σχεδίαση και υλοποίηση του SoC επεξεργασίας ήχου παρουσιάσαμε τις δυνατότητες προσομοίωσης που παρέχει το SysPy και πώς αυτές χρησιμοποιούνται για την προδιαγραφή των παραμέτρων ενός συστήματος πριν ξεκινήσει η σχεδίαση ενός SoC. Μέσω της χρήσης του SciPy εστιάσαμε ειδικότερα στην προδιαγραφή και προσομοίωση αριθμητικών αλγορίθμων και στην χρήση των αποτελεσμάτων προσομοίωσης στην υλοποίηση του αλγορίθμου στο υλικό. Δείξαμε επίσης τη χρησιμότητα ενός πυρήνα λειτουργικού συστήματος Linux στον έλεγχο και τη διασύνδεση ενός SoC και στην μεταφορά και επεξεργασία δεδομένων σε μορφή αρχείων, κάτι ιδιαιτέρως χρήσιμο σε περιπτώσεις που ένα SoC επικοινωνεί με άλλα υπολογιστικά συστήματα που διαθέτουν επίσης λειτουργικό σύστημα π.χ. Η/Υ με σύστημα Windows ή Linux. Παρουσιάσαμε επίσης τη δυνατότητα του SysPy να παραμετροποιεί αυτόματα και να μεταγλωττίζει τις εφαρμογές του χρήστη παράλληλα με τον πυρήνα του λειτουργικού συστήματος.

1.4.4 Εκτίμηση χρηστικότητας του SysPy

Με στόχο να έχουμε μία ποσοτική εκτίμηση των δυνατοτήτων του SysPy όσον αφορά τη σχεδίαση ενός SoC, κάναμε χρήση της πλατφόρμας αξιολόγησης BDTi [46]. Η μεθοδολογία αξιολόγησης απευθύνεται σε εργαλεία σχεδίασης υλικού σε υψηλό επίπεδο και αναπτύχθηκε από το πανεπιστήμιο του Berkeley. Η μεθοδολογία BDTi χρησιμοποιείται και ως πρότυπο για την εκτίμηση των επιδόσεων ενσωματωμένων επεξεργαστών και επεξεργαστών DSP. Η ανάπτυξη της μεθοδολογίας για την εκτίμηση εργαλείων σχεδίασης SoC σε υψηλό επίπεδο έγινε με στόχο την περαιτέρω ανάπτυξη εργαλείων τέτοιου τύπου που αυτοματοποιούν τη σχεδίαση πολύπλοκων SoC και επειδή οι ερευνητές στο Berkeley αναγνώρισαν ότι τα εργαλεία σχεδίασης που υπάρχουν δεν συμβαδίζουν με τις δυνατότητες που παρέχουν τα σύγχρονα συστήματα επεξεργασίας, όπως οι μονάδες FPGA.

Μέσω της χρήσης της μεθοδολογίας BDTi κάναμε χρήση των διαθέσιμων παραμέτρων και εκτιμήσαμε τη χρηστικότητα και λειτουργικότητα του SysPy ακολουθώντας τη μεθοδολογία σχεδίασης που υποστηρίζουμε. Για τη διαδικασία αξιολόγησης ζητήσαμε από τρεις χρήστες του SysPy που συμμετείχαν στη σχεδίασης των SoCs επεξεργασίας ήχου (Κεφάλαιο) και επεξεργασίας βιολογικών δεδομένων (Κεφάλαιο) να βαθμολογήσουν το SysPy σύμφωνα με τις παραμέτρους αξιολόγησης που παρέχονται στο BDTi. Τα αποτελέσματα των παραμέτρων που αξιολογήθηκαν από τους τρεις σχεδιαστές παρουσιάζονται στον Πίνακα Π1.6. Κάθε παράμετρος του πίνακα αξιολογήθηκε με μία από τις ακόλουθες βαθμολογίες: "Εξαιρετικά", "Πολύ καλά", "Καλά", "Μέτρια", "Χαμηλά". Στην ακόλουθη λίστα παρέχεται μία σύντομη περιγραφή των παραμέτρων αξιολόγησης:

- Out-of-Box Experience: Ευκολία εγκατάστασης και ρύθμισης του εργαλείου σε περιβάλλον Linux.
- Ease of Use: Ευκολία χρήσης του εργαλείου.
- Completeness of Capabilities: Εκτίμηση της επάρκειας των δυνατοτήτων του εργαλείου στη σχεδίαση SoC με πυρήνα επεξεργαστή.
- Quality of Documentation and Support: Εκτίμηση των οδηγιών χρήσης που παρέχονται με το SysPy.
- Learning to Use the Tool: Εκτίμηση της ευκολίας εκμάθησης χρήσης του εργαλείου.
- First Compiling Version: Εκτίμηση της προσπάθειας που απαιτείται για την υλοποίηση του πρώτου λειτουργικού σχεδίου (initial functional design) ενός συστήματος.
- Final Optimized Version: Εκτίμηση της προσπάθειας που απαιτείται για την υλοποίηση του τελικού λειτουργικού σχεδίου ενός συστήματος.
- Platform Infrastructure Development: Εκτίμηση της ευκολίας χρήσης του SysPy παράλληλα με άλλα εργαλεία που διευκολύνουν την τελική υλοποίηση ενός συστήματος (π.χ. εργαλεία

Out-of-Box Experience	**Ease of Use**	**Completeness of Capabilities**	**Quality of Documentation and Support**	**Learning to Use the Tool**	**First Compiling Version**	**Final Optimized Version**	**Platform Infrastructure Development**
Καλά	Χαμηλά	Μέτρια	Χαμηλά	Μέτρια	Καλά	Καλά	Πολύ καλά
Μέτρια	Καλά	Καλά	Χαμηλά	Μέτρια	Καλά	Μέτρια	Μέτρια
Μέτρια	Καλά	Καλά	Μέτρια	Μέτρια	Μέτρια	Μέτρια	Καλά

Πίνακας Π1.6: Παράμετροι αξιολόγησης του SysPy, σύμφωνα με το πρότυπο BDTi, από τρεις διαφορετικούς χρήστες.

φυσικής σχεδίασης FPGA).

Σύμφωνα με τα αποτελέσματα του Πίνακα Π1.6 οι δυνατότητες που παρέχει το SysPy κρίνονται επαρκείς ώστε ένας σχεδιαστής να προδιαγράψει την πρώτη λειτουργική έκδοση ενός SoC με επεξεργαστή σε εύλογο χρονικό διάστημα, να την προσομοιώσει και να επιτύχει την ορθή λειτουργία του σε μονάδα FPGA. Η παράμετρος "Platform Infrastructure Development" αξιολογήθηκε θετικά, εφόσον μέσω του SysPy παράγονται μία σειρά από προγράμματα scripts τα οποία διευκολύνουν τη διαχείριση των εργαλείων φυσικής σχεδίασης στο FPGA, ενώ ειδικά για τα SoCs επεξεργασίας βιολογικών δεδομένων και ήχου παρέχονται η διεπαφή HAL και η διεπαφή με το λειτουργικό σύστημα Linux, τα οποία βοηθούν στην χρήση των SoCs σε πραγματικές εφαρμογές εφόσον μπορούν εύκολα να ανταλλάξουν δεδομένα με άλλες διασυνδεδεμένες μονάδες με την πλακέτα του FPGA. Επίσης οι σχεδιαστές έκανα χρήση των διαθέσιμων παραδειγμάτων ώστε σε επίπεδο προσομοίωσης να συνδυάσουν εύκολα περιγραφές RTL με υψηλού επιπέδου μοντέλα προσομοίωσης σε Python. Οι χρήστες έδωσαν χαμηλή βαθμολογία στην κατηγορία "Ease of Use" κυρίως λόγω της έλλειψης πλήρους εγχειριδίου χρήσης κατά τη χρονική περίοδο της αξιολόγησης.

Σύμφωνα με τα αποτελέσματα της αξιολόγησης βελτιώσαμε τον τρόπο σύνταξης των μοντέλων προσομοίωσης, κυρίως τον τρόπο δήλωσης των σημάτων εισόδου και των αντίστοιχων χρόνων καθυστέρησης (input signal delay). Επίσης βελτιώσαμε τη μεθοδολογία διασύνδεσης περιγραφών RTL με μοντέλα SciPy διατηρώντας πάντα την αρχή ότι ο τρόπος σύνταξης των περιγραφών να είναι συμβατός με τα πιο κοινά αποδεκτά πρότυπα σύνταξης σε Python, ώστε το SysPy να μπορεί εύκολα να χρησιμοποιηθεί για την περιγραφή και προσομοίωση μονάδων υλικού ακόμα και από μηχανικούς λογισμικού με μικρή εμπειρία στην ψηφιακή σχεδίαση.

1.5 Συμπεράσματα

Μέσω των αποτελεσμάτων της διατριβής δείξαμε τους τρόπους με τους οποίους μια δημοφιλής και εύχρηστη γλώσσα προγραμματισμού υψηλού επιπέδου όπως η Python, που απευθύνεται σχεδόν αποκλειστικά στην ανάπτυξη λογισμικού, μπορεί να υποστηρίξει τη μεθοδολογία συσχεδίασης ενός ενσωματωμένου SoC με πυρήνα επεξεργαστή σε μονάδα προγραμματιζόμενης λογικής FPGA. Με τη χρήση επίσης τριών ολοκληρωμένων παραδειγμάτων σχεδίασης SoC με επεξεργαστή δείξαμε τον τρόπο με τον οποίο το SysPy αποδοτικά και αποτελεσματικά υποστηρίζει όλα τα βήματα που απαιτούνται στη συσχεδίαση υλικού/λογισμικού, ακόμα και για πολύπλοκα συστήματα όπου απαιτείται η χρήση λειτουργικού συστήματος Linux για τον έλεγχο της επεξεργασίας των δεδομένων.

1.5.1 Συνεισφορά

Με την παρουσίαση του SysPy αναπτύξαμε τη μεθοδολογία που προδιαγράψαμε ώστε να υποστηρίξουμε την υψηλού επιπέδου αρχιτεκτονική περιγραφή και προσομοίωση ενός ψηφιακού συστήματος με πυρήνα επεξεργαστή και την ορθή υλοποίηση του σε μονάδα προγραμματιζόμενης λογικής FPGA.

Παρά το γεγονός ότι υπάρχουν διαθέσιμα εργαλεία μοντελοποίησης και ψηφιακής προσομοίωσης σε υψηλό επίπεδο εδώ και πολλά χρόνια, όπως η SystemC, δεν υπάρχει διαθέσιμο ένα ολοκληρωμένο περιβάλλον το οποίο μπορεί να χρησιμοποιηθεί: α) για μοντελοποίηση και συνπροσομοίωση υλικού-/λογισμικού συστημάτων με πυρήνα επεξεργαστή και β) για την υποστήριξη όλων των βημάτων που απαιτούνται για την υλοποίηση ενός συστήματος σε μονάδα FPGA. Επίσης όλα τα εργαλεία που υπάρχουν στην βιβλιογραφία και κάνουν χρήση της Python για ψηφιακή σχεδίαση υποστηρίζουν μόνο συγκεκριμένα βήματα όσον αφορά τη σχεδίαση και υλοποίηση ενός συστήματος σε FPGA. Πιστεύουμε, σύμφωνα και με τα στοιχεία που παρουσιάσαμε στον Πίνακα Π1.1, ότι μόνο η μεθοδολογία που αναπτύξαμε και υλοποιήσαμε με βάση το SysPy υποστηρίζει όλα τα βήματα σχεδίασης και υλοποίησης σε μονάδα FPGA ενός SoC με πυρήνα επεξεργαστή (προσομοίωση σε επίπεδο RTL ή σε υψηλό αλγοριθμικό επίπεδο - σχεδίαση και υλοποίηση σε HDL - ανάπτυξη λογισμικού - διαχείριση εργαλείων λογικής σύνθεσης και φυσικής σχεδίασης σε FPGA - επικοινωνία και ανταλλαγή δεδομένων με το SoC μετά την υλοποίηση σε FPGA). Αφού μελετήσαμε τις μεθόδους και τα υπάρχοντα εργαλεία πιστεύουμε ότι απαιτείται πιο εντατική έρευνα στον τομέα της ανάπτυξης εργαλείων που κάνουν χρήση δωρεάν διαθέσιμων πυρήνων επεξεργαστών και παρέχουν τις κατάλληλες μεθόδους για τη χρήση τους και τη σχεδίαση ψηφιακών συστημάτων με χρήση FPGA. Τα εργαλεία αυτά θα πρέπει να υποστηρίζουν μεθόδους περιγραφής σε υψηλό επίπεδο, ειδικά για τις μονάδες υλικού και λογισμικού για τις οποίες δεν υπάρχει ακόμα υλοποίηση στα αρχικά στάδια σχεδίασης. Με τη χρήση των μεθόδων περιγραφής ο σχεδιαστής θα είναι σε θέση να αλλάζει εύκολα τις παραμέτρους των μοντέλων και να εκτελεί προσομοιώσεις σε επίπεδο συστήματος ώστε να πάρει τις σωστές αποφάσεις σχετικά με τη λειτουργικότητα και τις επεξεργαστικές επιδόσεις του συστήματος.

Με τη χρήση της Python καταφέραμε να αναπτύξουμε ένα εργαλείο που έχει ως στόχο την χρήση περιγραφών υψηλού επιπέδου σε γλώσσα Python για την προσομοίωση, περιγραφή και υλοποίηση σε FPGA ενσωματωμένων SoC. Όλα τα εργαλεία που παρέχει το SysPy κάνουν χρήση περιγραφών Python συμβατών με τα κοινά αποδεκτά συντακτικά και προγραμματιστικά χαρακτηριστικά της γλώσσας, ώστε η χρήση του να είναι προσιτή σε άτομα με ελάχιστη ή και καθόλου εμπειρία στη σχεδίαση ψηφιακών συστημάτων με γλώσσες περιγραφής υλικού. Αλλωστε αυτός ήταν και εξ αρχής ο βασικός στόχος ανάπτυξης του SysPy, δηλαδή η χρήση του για την ανάπτυξη υλικού από μηχανικούς και ερευνητές άλλων επιστημονικών πεδίων για την επιτάχυνση της εκτέλεσης πολύπλοκων αλγορίθμων επεξεργασίας δεδομένων στο υλικό. Τα κύρια και πρωτοποριακά χαρακτηριστικά του SysPy συνοψίζονται ακολούθως:

- Σχεδίαση ενσωματωμένων SoC με χρήση περιγραφών δομής Python (block-oriented design) με χρήση πυρήνων σε μορφή RTL ή σε μορφή netlist και υλοποίηση συνδυαστικής και ακολουθιακής λογικής για τη διασύνδεση τους.
- Προσομοίωση ψηφιακών συστημάτων με χρήση μοντέλων Python υψηλού επιπέδου ή περιγραφών C για την περιγραφή δομών υλικού ή λογισμικού που εκτελείται από τον επεξεργαστή ενός SoC.
- Αυτόματη παραγωγή συνθέσιμων περιγραφών VHDL
- Παροχή εργαλείων και λογισμικού διεπαφής για την επικοινωνία του SoC με άλλες ψηφιακές μονάδες π.χ. Η/Υ, άλλες μονάδες FPGA κτλ.
- Αυτόματη παραγωγή αρχείων script για τη διευκόλυνση χρήσης του κατάλληλου λογισμικού λογικής σύνθεσης και φυσικής σχεδίασης σε μονάδα FPGA.

Μέσω του SysPy γίνεται διαχείριση και χρήση μίας σειράς άλλων εργαλείων που απαιτούνται για την σχεδίαση ενός SoC. Όλα τα εργαλεία, όπως αρχεία script Tcl, αρχεία κυματομορφών VCD, μεταγλωττιστές λογισμικού GCC κ.α. αποτελούν ευρέως διαδεδομένα εργαλεία στο χώρο της ψηφιακής σχεδίασης. Με αυτόν τον τρόπο μέσω του SysPy κάνουμε χρήση πρακτικών και εργαλείων σχεδίασης που είναι αποδεκτά από την κοινότητα των μηχανικών ψηφιακής σχεδίασης.

Τα πρωτοποριακά χαρακτηριστικά του SysPy δοκιμάστηκαν επίσης και κατά τη σχεδίαση των παραδειγμάτων SoC που παραθέσαμε. Πιο συγκεκριμένα, τα ακόλουθα πρωτοποριακά χαρακτηριστικά του SysPy χρησιμοποιήθηκαν στη σχεδίαση κάθε SoC:

- SoC επεξεργασίας εικόνων
  - Αυτόματη μεταγλώττιση λογισμικού C.
  - Χρήση μονάδων υλικού σε μορφή netlist για περιγραφές δομής.
  - Χρήση μεθόδων Python για την αυτόματη παραμετροποίηση και σύνδεση μονάδων υλικού σε περιγραφές δομής.
- SoC στοχαστικής προσομοίωσης βιολογικών δικτύων
  - Χρήση διαύλου επικοινωνίας Ethernet για την επικοινωνία της μονάδας FPGA με H/Y, ώστε η μονάδα FPGA να καθίσταται ένας συνεπεξεργαστής που καταχωρεί και διαβάζει δεδομένα από και προς τον H/Y.
  - Χρήση μεγάλων εξωτερικών μονάδων μνήμης τύπου SDRAM εκτός του FPGA για την γρήγορη καταχώρηση και ανάκληση δεδομένων.
  - Χρήση δεδομένων από εξωτερικά αρχεία π.χ. αρχεία για την παραμετροποίηση ψηφιακών μονάδων.
  - Χρήση διεπαφής Python/C (HAL: Hardware Abstraction Layer) για την επικοινωνία H/Y και FPGA.
- SoC επεξεργασίας ήχου
  - Ανάπτυξη εφαρμογών λογισμικού με χρήση ενσωματωμένου λειτουργικού συστήματος Linux.
  - Ανάπτυξη μηχανισμού προσομοίωσης σε επίπεδο περιγραφών RTL αλλά και σε αλγοριθμικό επίπεδο για τον προσδιορισμό των παραμέτρων ενός συστήματος πριν την διαδικασία σχεδίασης.
  - Συνπροσομοίωση υλικού/λογισμικού με παράλληλη χρήση αλγοριθμικών περιγραφών (Matlab-like) και περιγραφών C μαζί με περιγραφές υλικού RTL.

– Παρουσίαση των αποτελεσμάτων της προσομοίωσης σε μορφή αρχείων VCD, συμβατή και με άλλα εργαλεία προσομοίωσης (π.χ. Modelsim).

Με στόχο τη βελτίωση του εργαλείου και τη διάθεση του σε άλλους χρήστες, παρέχουμε ελεύθερα όλον τον κώδικα Python μαζί με παραδείγματα σχεδίασης [89] μέσω του GitHub που αποτελεί την μεγαλύτερη ιστοσελίδα παροχής δωρεάν λογισμικού στο διαδίκτυο. Μέσω του GitHub το SysPy έχει ήδη αναφορές σε άλλα αντίστοιχα εργαλεία που κάνουν χρήση της Python για σχεδίαση και προσομοίωση μονάδων υλικού, όπως το PyMTL [58], [59] που αναπτύχθηκε στο πανεπιστήμιο Cornell. Αναφορά στο SysPy και στις μεθόδους που χρησιμοποιούμαι γίνεται και σε μία βελτιωμένη έκδοση του εργαλείου MyHDL [54]. Επίσης κώδικας του SysPy χρησιμοποιείται στον επεξεργαστή MinSoC [29] (έκδοση του OpenRISC για πλατφόρμες FPGA) για την αρχικοποίηση της μνήμης προγράμματος, ενώ η ομάδα μας υπήρξε η πρώτη που υλοποίησε τον συγκεκριμένο επεξεργαστή σε μονάδα FPGA Virtex-5. Χρήση του SysPy γίνεται επίσης και για τη σχεδίαση του συστήματος Aura SoC [14].

1.5.2 Προτεινόμενες βελτιώσεις

Σύμφωνα με τα αποτελέσματα των τριών παραδειγμάτων σχεδίασης που παρουσιάσαμε, μπορούμε να συνοψίσουμε σε τρεις βασικούς άξονες τα πεδία στα οποία υπάρχει περιθώριο βελτίωσης όσον αφορά τις δυνατότητες του εργαλείου: α) προσθήκη νέων δυνατοτήτων σχεδίασης και προσομοίωσης ενός SoC, β) προσθήκη περισσότερων έτοιμων μονάδων προς χρήση στις βιβλιοθήκες του εργαλείου και γ) υλοποίηση περισσότερων παραδειγμάτων σχεδίασης που θα λειτουργήσουν και ως κίνητρο για τη χρήση του εργαλείου από άλλους χρήστες.

Σημαντική προσθήκη θα ήταν η παραγωγή κώδικα VHDL και η υποστήριξη εργαλείων σχεδίασης ASIC και όχι μόνο εργαλείων σχεδίασης με μονάδες FPGA. Η προσθήκη περισσότερων έτοιμων μονάδων, ειδικά σε επίπεδο περιγραφών netlist κρίνεται απαραίτητη, στις βιβλιοθήκες του SysPy, αλλά και η ανάπτυξη των σχετικών μεθόδων Python που θα αρχικοποιούν και θα συνδέουν αυτόματα τις νέες μονάδες σε περιγραφές δομής. Θα πρέπει επίσης να ενσωματωθούν περισσότεροι πυρήνες επεξεργαστών οι οποίοι υποστηρίζονται και σε μονάδες FPGA άλλων εταιρειών, εκτός της εταιρείας Xilinx, όπως π.χ. ο πυρήνας ανοιχτού κώδικα LatticeMico32 από την εταιρεία κατασκευής FPGA Lattice. Επιθυμητή κρίνεται η υποστήριξη λειτουργικού συστήματος Linux στους νέους πυρήνες επεξεργαστών ώστε να είναι εύκολη η υποστήριξη προγραμμάτων οδήγησης περιφερειακών μονάδων π.χ. ελεγκτών μνήμης κ.α. καθώς και διαφόρων πρωτοκόλλων επικοινωνίας π.χ. USB, Ethernet κ.α.

Απαραίτητη είναι και η διάθεση περισσότερων παραδειγμάτων σχεδίασης με τη χρήση του SysPy. Ειδικότερα παραδείγματα σχεδίασης που απαιτούν αριθμητική επεξεργασία δεδομένων, όπως επεξεργασία βίντεο ή επεξεργασία πακέτων σε δίκτυο δεδομένων, θα αναδείξουν τις δυνατότητες του εργαλείου στη σχεδίαση SoC με χρήση πυρήνων επεξεργαστών σε συνδυασμό με τη σύνδεση αριθμητικών μονάδων ως επεξεργαστών ειδικού σκοπού.

Μέσω της ιστοσελίδας του SysPy στην πλατφόρμα GitHub [89] παρέχονται πληροφορίες για τη χρήση αλλά και την εγκατάσταση του εργαλείου σε περιβάλλον λειτουργικού συστήματος Linux. Πληροφορίες για το SysPy υπάρχουν και στην σχετική ιστοσελίδα που περιγράφει τις λειτουργίες του εργαλείου και την μεθοδολογία ανάπτυξης του [88]. Στόχος της διάθεσης του SysPy ως εργαλείο ανοιχτού λογισμικού, είναι η διαρκής βελτίωση των χαρακτηριστικών του και η χρήση του από έναν αυξανόμενο αριθμό χρηστών που θα εκτιμήσει και θα αξιοποιήσει τις δυνατότητες που παρέχει για σχεδίαση ενός SoC σε υψηλό επίπεδο περιγραφής με τη χρήση της Python.

Contents

List of Figures

List of Tables

List of Algorithms

Preface

This work has been supported by the Greek State Scholarships Foundation (IKY) under grant 2008-5530.

Chapter 1

Introduction

Modern Field Programmable Gate Array (FPGA) devices can host very complex digital designs. Most of the implemented System on Chips (SoCs) incorporate at least one programmable microprocessor (uP) unit. The processor's Intellectual Property (IP) core is key elements for the rapid prototyping of new digital systems, but on the other hand its usage raises a lot of design challenges that have to be addressed in the design flow. In this chapter we state the goals that we wanted to achieve by developing methods and tools for designing processor-centric systems on chip implemented on FPGAs.

1.1 Goals and vision

The main goal of this dissertation was the development of methods and a design tool targeting the hardware/software co-design and verification, using hight-level abstract descriptions, of processor-centric SoCs implemented using FPGAs. For the needs of the research, we evaluated Python's programming features and especially the combination of scripting capabilities in a Linux shell, combined with Object Oriented Programming (OOP) features. These supported features could be used to:

- Implement high-level abstract models of blocks, e.g. arithmetic, memory and logic blocks, connect them using structural Python descriptions and translate them automatically to FPGA synthesizable Very high speed integrated circuit Hardware Description Language (VHDL), or use them to perform Register Transfer Level (RTL) bit-true simulation of a system. The integration of the SciPy library in Python provides a large number of functions which can be used for modeling arithmetic blocks.

- Build a framework and a design tool that implements the end-to-end design flow of a processor-centric system-on-chip, which invokes/calls other hardware and software related tools, e.g. logic synthesizers, software compilers, simulation tools etc.

- Process the large number of text-based files generated during a hardware design flow. Information extracted from generated text files is used many times as an input for the next design step or can be transformed/parsed to a different format.

While designing a complex digital system cannot be done automatically at a press of a button, we envisioned a design tool that would integrate the majority of the tools needed for an FPGA implementation of an embedded SoC. The first and most difficult task was to build the Python-to-VHDL parser. For this task we needed to define our supported coding style/syntax for the hardware descriptions in Python. The syntax should support a level of abstraction but on the other hand support features that are used in well established HDL languages, like VHDL and Verilog. A lexical analyzer also needed [31] to recognize and track, in the user supplied Python descriptions, the supported syntax and parse these parts of Python code that later on would be mapped and translated to synthesizable VHDL.

Python also provides the data structures to easily handle the large number of parameters/constraints required to design modular SoCs and to constrain the synthesis and implementation process. Some of these parameters either should be defined directly in the design files, e.g. size of a data bus in bits, time resolution of a simulation testbench, or they can be defined in text files read/parsed by SysPy during translating a design to VHDL or simulating a design, e.g. timing constraints, placement constraints during floorplanning in the FPGA, list of files to be synthesized etc. All these parameters could be stored efficiently using Python key/value hash table structures, called dictionaries. Using dictionaries it becomes easy to store and recall information using user-defined keywords. Dictionaries have been used not only to store design constraints, but also to store information about a given Python description design, e.g. all the I/O signal names and their properties, clock and reset signal names, sequential block names etc.

The use of a good lexical analysis tool, combined with Python's unique features to store and manipulate text files used within our project as a very solid programming environment for scanning and manipulating textual data. Using these features we could create the necessary Python - to - VHDL parser and also generate all the Tool Control Language (TCL) scripts needed for an FPGA implementation by the synthesis tools. Python in our tool also handles all the calls to external software tools, like the C compilers used to develop the software executed by the processor. The existence also of a good Matlab-like tool, like SciPy, embedded into the Python environment triggered our interest on building a simulation tool, especially targeting arithmetic hardware blocks. Algorithmic descriptions could be used to describe numerical operations in fixed or floating point format. In this way a processing datapath (arithmetic blocks plus registers) controlled by a state machine could be easily simulated in an abstract, but bit-true format, while the Python code of the arithmetic and control blocks (FSM) could be later translated to synthesizable VHDL.

1.2 Contributions

The main contribution of this dissertation is to show that a modern programming language like Python can be used to design, simulate and implement processor-centric embedded SoCs, using high-level, abstract descriptions. This is very useful especially early in the design flow when control and processing logic of a system must be partitioned among software and hardware implementation. Our research work also shows convincingly that Python is a good candidate language to handle the large number of design tools needed to capture and implement a SoC in an FPGA device, in terms of hardware and software development. It also demonstrates Python's capabilities to process text files and string variables, which is very useful for text conversion and parsing and for auto-generating code, e.g. VHDL, Tcl scripts, XML. SysPy facilitates architectural exploration and simulation. Python functions and objects are treated like digital modules during simulation, while SysPy provides the timing mechanism needed to perform a bit-true and cycle-accurate simulation. After simulation, Python descriptions are parsed to synthesizable VHDL code for FPGA implementation and the tool also generates all the necessary scripts to drive and support the Xilinx FPGA synthesis tools.

Using scripting languages for hardware/software co-design

To our knowledge only SysPy:

1. supports hw/sw co-design and the use of high-level software models in Python or C along with RLT-like hardware descriptions also in Python for hw/sw co-simulation.
2. uses a popular software language like Python as an Architectural Description Language (ADL) [68] to support abstract simulation of SoC designs and generation of Value Change Dump (VCD) files for top-level I/O signal visualization.
3. supports the use of parameterized Python functions to automatically generate synthesizable VHDL code.
4. supports the design of processor-centric SoCs implemented in FPGA devices using freely available processor cores, like Leon and OpenRISC.
5. integrates with FPGA implementation tools and supports generation of Tcl scripts for the Xilinx design tools to ease the required design steps down to the generation of the bitstream files used for FPGA programming.

In this way we take advantage of Python's best features and deliver a design tool that can be used to describe the architecture of a SoC in an abstract or algorithmic way, especially when arithmetic operations are involved and also supports most of the steps required in an FPGA design flow to implement the design in silicon.

1.3 Thesis outline

In the thesis we tried to describe in enough detail the innovative features of SysPy and also provide design examples to motivate the reader about the usefulness of the tool. We present three large processor-centric design examples along with experimental implementation/performance results. In all design examples that we implemented the FPGA board was connected to a host PC using a serial or/and Ethernet interface, in order to exchange data and provide processing results to the PC for further processing and analysis. The FPGA board in this way was used as an attached data co-processor, where the PC provided data pre-processing functionality, e.g. split data in different files or in different network packets, while the performance demanding processing tasks were accomplished by the FPGA device.

The implementation of the design examples reflects the evolution of SysPy during time. The utilized processors for the examples are the freely available, through OpenCores [73], 8-bit soft core of the widely used AVR ATmega128 [76] and the popular 32-bit Leon3 soft IP architecture [11] by Aeroflex Gaisler. The first design example is an image processing SoC [60] built around the 8-bit AVR core. The second design is a high performance embedded computing SoC used to accelerate stochastic simulations of large-scale biochemical reaction networks for systems biology [61], [40], [41]. The third design is an audio signal processing SoC implemented using the Leon3 core as the main system controller, where the software executed by the processor is compiled along with a Linux Operating System (O/S) kernel which handles efficiently all the I/O and memory communication in the SoC. Test examples were also performed using the OpenRISC 32-bit core [29] also provided by OpenCores.

The chapters of the thesis are organized as follows:

- Chapter 2 describes in detail Python main features and how they are utilized by popular hardware and software related projects. A description is provided of the architecture of modern FPGA devices and the way they favor processor-centric designs. Related work of academic/research projects using Python for digital hardware design is also presented along with the tool's contribution to embedded SoC design.
- Chapter 3 presents the design flow developed and also how the simulation features of SysPy can be used for embedded SoC verification, early in the design phase.
- Chapter 4 presents methods to describe digital logic in behavioral or structural format in Python and how SysPy translates the description to FPGA synthesizable VHDL code.
- Chapter 5 presents in detail how SysPy can be used to instantiate a processor soft core in a design and also the way it supports software development flow for bare-metal and O/S-centric applications.
- Chapters 6, 7 and 8 provide details and description of the implemented SoC design cases. The architectural specification of each design and the flow used to design and verify the SoCs in SysPy are provided. The data processing algorithms ported in hardware in each design are described as well as the process of sw/hw partitioning.

Implementation results in terms of FPGA resources utilization and timing results are provided.

Also for all the designs information is provided on the type of processed data, e.g. image/audio data, biomolecular reactions XML-data. The host-PC interface used in every design is also described as well as the procedure of exchanging data between the FPGA board and the host PC. Chapter 8 also provides the results of an evaluation method that we used to assess the usability of the features of our tool, where feedback for the evaluation has been provided by users of SysPy.

- Chapter 9 presents the conclusions of SysPy's development and of the design of the various examples used to showcase the features of the tool. A number of future improvements and enhancements is also suggested.

- Appendix A provides tables with the synthesis parameters that can be used in Python descriptions along with compilation parameters for the supported Snapgear Linux kernel. A short installation guide of SysPy under Debian Linux is also provided.

- Appendix B presents extended code examples used to demonstrate the various design and verification features of SysPy.

Chapter 2

Background and related work

Choosing the correct programming language for the job at hand is not an easy task. In our case the chosen language is used for developing a design tool but also for describing hardware and software running in an embedded processor. In this chapter we elaborate on these features that Python has and make it suitable for developing a digital hardware design tool. A brief introduction to the architecture of modern FPGA devices is presented. We also provide details about other hardware and software related projects where Python has been used, so the reader can have a better understanding about the applications that Python can be used for. Emphasis is placed on digital hardware related tools developed using Python, especially related work from academic/research projects.

2.1 Importance of processor-centric Systems-on-Chip

Most of the complex SoC designs implemented nowadays using FPGAs are processor-centric, where the uP is used as a gateway, implementing in software different communication protocols needed to exchange data with other logic devices on the same board or with a connected PC/server. Software executed by the processor is also used to control and exchange data with other custom blocks implemented in the FPGA fabric. Complex control and data processing logic can now be easily partitioned between software and hardware inside the same FPGA device.

The latest devices from FPGA vendors, such as the Virtex-7 device from Xilinx or the Arria-V device from Altera, include fast embedded dual-core ARM processors which can communicate with the rest of the FPGA fabric using a very fast data and control bus. This type of new devices are mentioned as programmable SoCs, where software executed in the processor is tightly coupled with hardware-based processing and hardwired bus interface protocols are used to exchange data and control signals between the processor and blocks implemented in the FPGA fabric. The digital architecture adopted in the latest FPGA devices introduces a block oriented way design flow of a SoC, where the designer connects new peripheral devices to the processor, according to the data processing requirements of the application. New designs can be efficiently prototyped within a few days, while the performance and power consumption of a SoC in an FPGA device can be now compared with that of an ASIC implementation. Of course for mass IC production the ASIC device remains the only option, especially in terms of cost reduction.

The more ready-to-use logic will be “squeezed” inside an FPGA chip, the more popular and attractive these devices will become, not only to hardware but also to software developers, which seek ways to accelerate software algorithmic implementations. Applications such as audio, image and network data processing have already been mapped into FPGA devices, taking advantage of the large number of hardwired arithmetic blocks, used to perform parallel fixed/floating point arithmetic operations. The key to success remains the efficiency of the hardware/software co-design tools to combine, connect and program all the necessary digital blocks inside an FPGA.

2.1.1 New capabilities of modern FPGA devices

Programmable Logic Devices (PLD) is a family of digital Integrated Circuits (IC) that has seen tremendous growth over the past decade. Silicon complexity has increased dramatically, compared to the devices that were available a decade ago. A large number of available resources is now available inside an FPGA, ranging from Configurable Logic Blocks (CLBs) and Block RAMs (BRAMs) to arithmetic/multiplier blocks and hardwired embedded processor cores. Large devices can now be used to replace several digital Application Specific Integrated Circuits (ASIC), minimizing in this way the size of Printed Circuit Boards (PCBs).

Incorporation of processor cores, in the form of hardwired or processor soft IP cores, brought a new era to logic reconfigurability using PLDs. FPGA design tools must now efficiently handle processor IP cores and their software ecosystem as well as the integration of the necessary glue logic around the processor. Also more complex design tools needed for multiprocessor systems, where in many cases an O/S is needed to distribute the processing tasks among the available processors.

One big challenge in latest devices is to efficiently utilize a fast interface between an implemented soft or hardwired processor core and units implemented on the FPGA fabric. Devices utilizing hardwired ARM processor cores contain also hardwired implementations of such interface blocks. In case of softcore processors, the designer is responsible for building this interconnection interface block on the FPGA fabric. The design of such a block is a challenging task since FIFO memories must be implemented to ensure that data can be exchanged in a safe way between the two different clock domains and also that the operation of the faster domain is not stalled due to the data exchange with the slower domain.

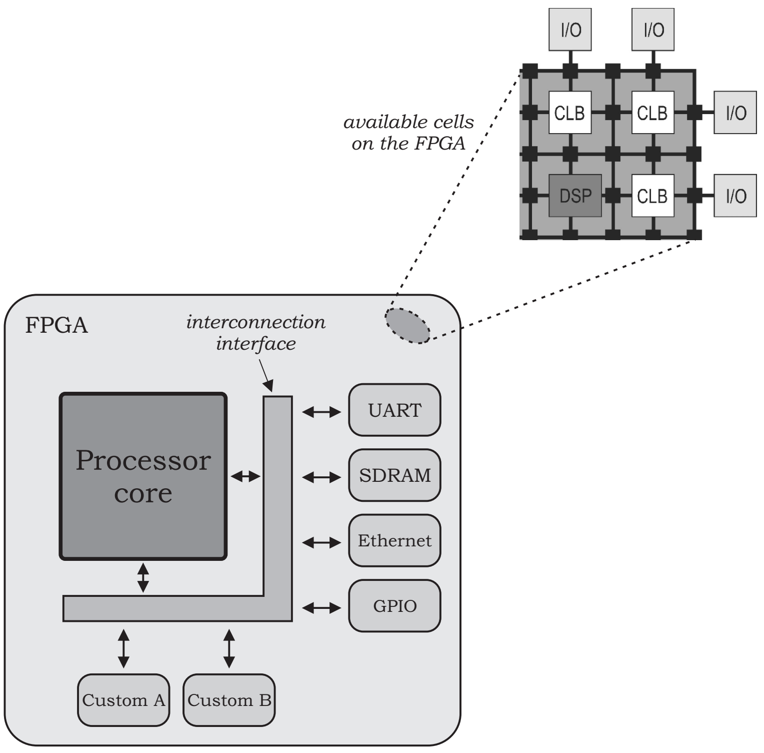


Figure 2.1: Interconnection interface between a processor core and various peripheral units in the FPGA fabric.

Among the latest features of modern FPGA devices is the existence on the FPGA fabric of complex arithmetic units, which can implement in hardware a number of arithmetic operations ofter required during execution of signal processing algorithms. Synthesis tools automatically configure and connect the required number of arithmetic blocks to appropriate CLBs. The arithmetic blocks are placed in columns inside the FPGA (just like BRAMs) and a large number of blocks can be automatically cascaded to handle arithmetic operands of large bit size. In this way the existence of a processor core in conjunction with the arithmetic blocks in an FPGA design can easily outperform the processing power of traditional DSP processors. The main drawback for a good implementation remains the ability of an engineer to combine software and hardware design skills to implement a good and well timed digital design for his/her application. A typical diagram of the blocks found in a modern FPGA devices is presented in Figure 2.1.

The latest trend on FPGA design is to use devices where the main block is not anymore the programmable logic fabric, but one or more hardwired processors. Fast hardwired embedded dual-core ARM processors along with a large number of hardwired MULs, which are included in the latest devices from FPGA vendors (e.g Xilinx's Virtex-7 and Zynq and Altera's Arria-V) can be configured to perform several fixed and floating point operations commonly used in DSP applications e.g. FIR/IIR filters, FFT, CORDIC etc. Such kind of devices are often called as All Programmable System-on-Chip (APSoC), since they favor software development in conjunction with the development of tightly coupled custom hardware blocks to handle the process demanding parts of an algorithm.

2.2 Choosing Python for hardware design

Python [2] is a freely available, high-level, interpreted language developed by Guido van Rossum. Python's standard library is extensive and well-documented, while code documentation and reusability are core principles of the language. Python gives to the user rapid development and flexibility and can be combined with C/C++ which is the industry standard for fast algorithmic implementations. Python has become extremely popular in the software community during the last decade mainly because of its clear syntax and of the reduced time required to develop complex software. Python code with good and clear syntax has close resemblance to the pseudocode, most of the times used to describe algorithms before their development. Also C code can easily be called within a Python program, in cases where access to C libraries is required. Python supports object oriented programming, where classes and objects can be used to encapsulate data and the related processing functions/methods, required to process them, in the same programming structure.

Python is installed by default on all the major Linux distributions and in many cases it is used as a replacement for shell scripting. Also a large number of ready-to-use libraries exist for Python, targeting a wide range of programming applications. Some of the most important freely available libraries are the following: i) *NumPy* and *SciPy* [62] are Matlab-like libraries where Python is used for linear algebra and vector computations used in Digital Signal Processing (DSP). ii) *matplotlib* [43] is a numerical plotting library, which can be easily combined with SciPy. iii) *Scrapy* is a very useful tool in web programming for extracting information and performing data mining on the web.

Keeping in mind the great programming features supported by Python and the popularity of the language in the software community, we tried to support Python structures/syntax across all design and simulation steps in our flow to describe the datapath of the SoC and the related simulation models and avoid custom-defined syntax.

2.2.1 Python's features exploited for digital design

For describing digital systems, choosing the proper language to develop the tool was of great importance. The special features that are required in a design flow targeting processor-centric designs and hw/sw co-design are:

- Integrate under the same scripting environment all the required tools for hardware/-software co-development and co-simulation.

- Support clear and powerful syntax that could be used to describe hardware digital designs using Hardware Description Language (HDL) like syntax or in an abstract way, using functions to auto-generate HDL code.

Most hardware design tools are based on scripting tools, such as Perl and Tcl [21]. Although Perl is a powerful dynamic programming language, it is severely lacking in reusability. Python's standard library is more flexible and well-documented compared to Perl [30], while code reusability is core principle of the language. Python is also rapidly gaining popularity in scientific computing, where a lot of engineers already use SciPy for algorithmic coding.

Using scripting languages for hardware/software co-design

Python's high level structures can be used to support abstraction in building a SoC in a block oriented way. In most cases custom peripherals are described in HDL and connected to the main data and control bus of the processor. Moreover, they facilitate the description of glue logic among these blocks. The processor acts as a gateway in the FPGA board and advanced communication features are available if an O/S is running in the processor.

Since a processor IP core is a fairly complex block, Tcl scripts have to be used in synthesis tools to import all the necessary design and constraint files. Makefile scripts are also required for compiling C software for the processors. Python's text processing features are used in SysPy to generate and modify the required synthesis and compilation scripts. The scripts are generated according to user input parameters defined at the top level module of his/her Python description in SysPy. These parameters define hardware related information, .e.g FPGA device family, main clock frequency etc., and also software related information, e.g. C file names, O/S name and type etc. Python functions are also used to generate all the text-based simulation files containing the values of the VHDL signals during a simulation, along with the related scripts used to start the simulation.

Figure 2.2 shows how we have used Python to handle the integration of ready-to-use processor IP-core in a SoC design. FPGA synthesizable VHDL code is generated from Python description for custom blocks along with data/control bus interface/glue logic. SysPy generates Tcl scripts for each one of the supported processor IP cores, which executes along with the FPGA synthesis tools in a command line, in order to incorporate the processor subsystem in the design. The tool also compiles, using the GNU Compiler Collection (GCC) [1] C tools, the processor's control software along with any existing O/S kernel. FPGA bitstream file along with the compiled software binary files are generated, since SysPy makes all the necessary external tool calls (synthesizer, compiler), and can be used for FPGA implementation.

We have also developed methods to simulate the processor's software in a bit-true way expressed in Python, along with the Python hardware descriptions of the connected peripheral blocks. VCD files are generated in simulation mode that can be used to generate VHDL signals' waveforms. In this way we use Python to handle the complete tool chain required to verify/debug and implement a processor-centric SoC for an FPGA device.

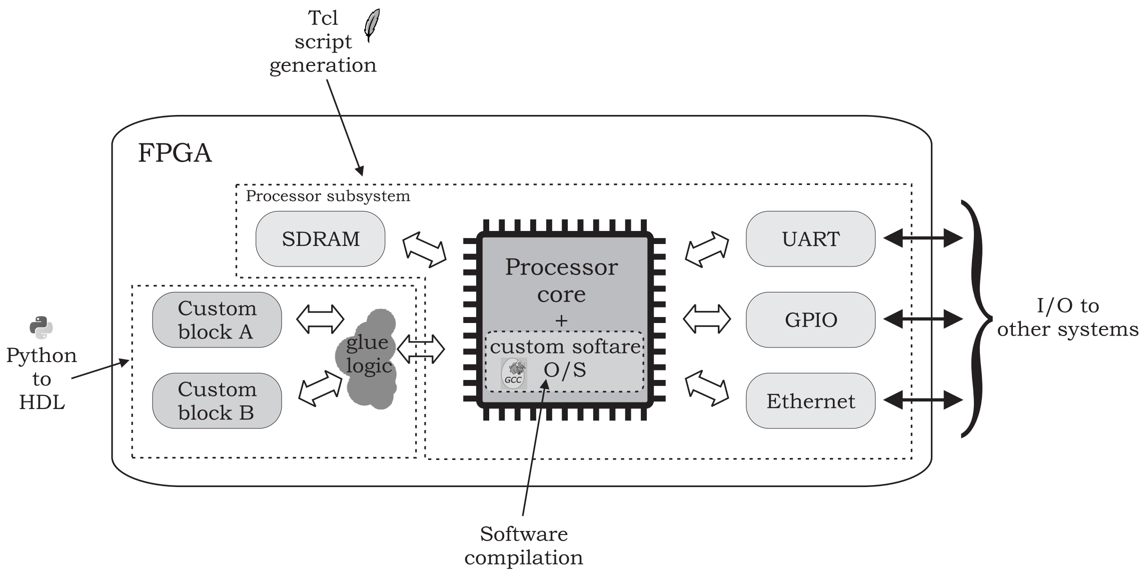


Figure 2.2: Generic processor-centric SoC diagram.

One important tool based on the NumPy Python library is Biopython [20], which is used for biological computations. The tool is used to parse and process a large number of different file types used in bioinformatics, containing biological related data. Scrapy [8] is another tool that uses Python text processing features to create internet crawlers and extract information from webpages. By using Scrapy web-based data mining and monitoring tools can be easily created to process the huge amount of information found today on the internet. Pandas [64] is a Python project that performs data statistical analysis. Pandas can be used along with NumPy to perform data analysis and extract information from large data sets. Pandas can easily parse data from a large number of different file types, e.g. HTML, Excel, SQL etc. and store them in Python arrays and dictionaries for further processing. OpenStack [83] is another great tool developed using Python that is used as a framework deployed on top of an O/S to control and distribute large numbers of storage and computing elements in a cloud computing environment. It is this data parsing and raw text processing features of Python, used in Pandas, that software developers like most about the language and the way Python can be used to unify different programming and computing environments, as utilized in OpenStack.

2.3 Related work

2.3.1 Popular Python tools for hardware and embedded design

Python has been used for the development of commercial and widely used complex hardware and especially software development tool projects, where the text manipulation and generation features of the language are exercised in the best way. *PyCells* [22] are structures written in Python resembling parameterized analog or digital cells for ASIC design. The tool can be used for creating Universal OpenAccess PCells, which physically represents analog circuits in a transistor level design. PyCell uses Python object-oriented features to define the geometry of complex analog physical designs with more compact and smaller descriptions, compared to the SKILL language used in PCells. The PyCell architecture has already been adopted by leading EDA companies, like Cadence and Synopsys. Python language has also been used in the VIPER project [90] for the development of a programming environment targeting embedded system application. Several microcontroller (uC) board are supported, like the Arduino boards [12], [18], for developing internet and cloud based applications.

2.3.2 Comparison to other existing tools

While there is a large repository of tools targeting hardware design, most tools cannot facilitate the integration of both dedicated hardware components and programmable uPs coexisting in the same processor-centric SoC. The inclusion of processor cores requires the usage of software development tools, while special-purpose hardware components (accelerators etc.) must exchange data with the processor using dedicated data and control buses.

Most FPGA synthesizers [94] or high-level design tools [87], [93] lack the ability to handle the software and the hardware aspects of a design in a unified manner. Processor cores are handled by separated tool chains, while their connection and programming occurs during the last design steps. FPGA vendor embedded processor tools, such as Embedded Development Kit (EDK) by Xilinx, support processor-centric designs. EDK, however, handles hardware and software as two parallel threads with no interaction between them and supports only Xilinx proprietary processor cores, such as MicroBlaze and PowerPC.

The related hardware design efforts found in academia are the following:

PyHDL [39] is a tool for structural descriptions which tried to simplify system design with the use of optimized hardware objects described in C++. *PyHDL* extends the functionality of *PAM-Blox* [65] by creating a Python/C Application Programming Interface (API) so that a designer can have access to the C++ libraries by using Python scripts. *PAM-Blox* has been developed in times when the capacity of FPGA devices was very constrained, so it did not find many practical uses.

PyMTL [58] is Python related tool targeting digital hardware design by unifying functional, cycle-accurate and Register Transfer Level (RTL) hardware descriptions. *PyMTL* can be used for behavioral and structural descriptions and also to verify the functionality of a design. The tool also provides HDL generation features, by translating Python descriptions to Verilog. *PyMTL* uses a Just-In-Time (JIT) compiler to convert Python testbenches to C++ in order to significantly accelerate simulation execution in Python. As we do in SysPy, *PyMTL* takes advantage of Python clear syntax and popularity and also of ready-to-use numerical libraries to create abstract models of hardware. The tool generates from abstract and RTL-like descriptions ready-to-synthesized Verilog code, targeting hardware implementation, but without optimizing the generated files for ASIC or FPGA implementations. Emphasis has been placed during the tool's development on accelerating the functional and RTL simulation by using the JIT tool. On the other hand the tool does not generate any files, e.g. Tcl scripts, folder hierarchy, to support implementation process by synthesis tools, as we do in SysPy for FPGA implementation and also there is not support for hw/sw co-design and processor-centric systems. Furthermore in the simulation models there is no support to include timing and logic latency information, as we support in SysPy, where the generated VCD files include all the logic and timing information of a digital block, as described by the user in Python.

PyGen [77] is an add-on tool for Matlab/Simulink for hardware design. *PyGen* maps Simulink's hardware building blocks to Python classes. Simulink handles hardware DSP blocks through System Generator [93], an add-on tool provided by Xilinx. A designer can develop a structural description by instantiating objects from a Python library. *PyGen* cannot be used to design custom logic, but provides a Python API that handles at a high-level Xilinx proprietary cores. It can also be used to perform power and performance estimation of a designed system.

Using scripting languages for hardware/software co-design

PHDL [63] uses Python to provide a higher abstraction level for hardware design. A designer can structurally describe a system using components from a Python library supplied with *PHDL*. Parameters' selection can alter the size of a module, e.g. the bus width or a global parameter can control the synthesis process of a design, e.g. FPGA device selection. While *PHDL* supports FPGA implementations, the tool's libraries provide access only to basic RTL blocks e.g. registers, MUXs, adders etc.

Other tools use Python directly as an HDL to describe hardware. *MyHDL* [23], [71] supports, as SysPy, the three basic design capture methods, i.e. behavioral, dataflow and structural and it also has a behavioral simulation function, presenting text based simulation results of a given design. *MyHDL* supports parsing Python descriptions to both VHDL and Verilog. Unlike *SysPy*, *MyHDL* has no support using pre-designed HDL blocks, so all the components of a system must be described in Python.

PDSDL [97] supports, as *MyHDL*, the translation of behavioral Python descriptions to HDL. The tool also supports a simulation engine for behavioral descriptions. The supported Python syntax for signal declaration and assignment is rather complex. Furthermore, the authors do not provide sufficient design examples demonstrating the quality of the translated HDL code.

In summary, all previously described efforts try to use Python's high-level structures to perform abstractions in hardware description. *PyGen* and *PHDL* map ready-to-use HDL blocks and netlist files to Python functions and classes. In this way a Python script resembles a structural HDL description, where function and class arguments act as generic parameters. *MyHDL* and *PDSDL* use Python as a high-level HDL and support structural and behavioral descriptions. *MyHDL* is suitable for designing and debugging a system at the Python level, but there is no provision for handling processor IP cores and complete SoC systems.

Although *MyHDL* never evolved to a commercial tool, the way it uses Python's data structures to define HDL-like digital hardware descriptions inspired us in the way we could take advantage of Python's unique features in this work. In all Python related tools that we found in the literature we identified the following major weaknesses: a) lack of support of processing-centric systems and hw/sw co-design flow and b) lack of ability to handle the complete design flow of a digital system, starting from a high level description down to the generation of a bitstream file and FPGA programming.

SysPy can also be compared with other SoC customization tools, like the Target IP Designer [32] from Synopsys. The tool can be used to specify the architecture of a custom processor and also to provide code optimization for algorithms in software by the processor. While Target Compiler is a well tested commercial tool, it uses a vendor proprietary language/syntax (nML) to describe a system's architecture and also it is not possible to simulate/verify the design's datapath in the nML abstract description level, as it is the case in SysPy. Introduction of a new language and style for hardware design was something that we tried to avoid and on the other hand we wanted to stay consistent, from a high-level point of view, to the typical design flow of a digital system. We used Python to express the functionality of a SoC in an abstract way and connect all the required logic to build a processor-centric system and also to automate the design and verification steps in the adopted flow. Python is also used to "glue" together typical hardware and software design tools, e.g. RTL synthesizer, C compilers, Tcl scripting etc. to ease FPGA implementation of a SoC.

2.3.2.1 Comparing verification features of SysPy to SystemC

SystemC is an extension of the popular C++ object oriented language. The language standard [44] [9] provides a set of C++ classes and data types that support the development of event-driven models and testbenches of digital systems.

We believe that with the addition of the event-driven mechanism in the simulation environment of SysPy, Python syntax can be really suitable to model and simulate a digital system. Especially with the use of SciPy, arithmetic models and signal plots can easily be used for system modeling and verification. The supported syntax and format of simulation models in SysPy allow a testbench to easily merge with already existing RTL-like descriptions, something which is not easy in SystemC, since the language requires some effort to merge a model along with RTL descriptions. Also the native SystemC environment does not support generation of VCD files for digital signal visualization. However tools supporting the SystemC verification environment, like irun [19] simulator from Cadence extend the features of the language by supporting VCD data dumping.

In Figure 2.3 we summarize the main features and differences between the supported modeling environment in SystemC and SysPy. Also the main advantages in system verifica-

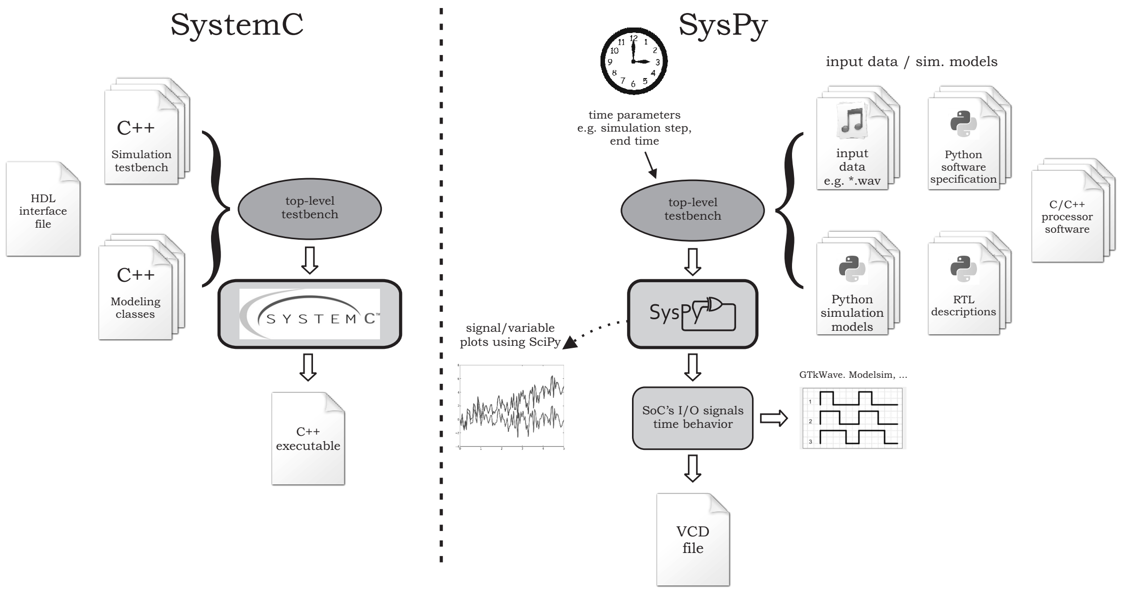


Figure 2.3: Comparison between SystemC and SysPy verification features.

tion of SysPy are summarized in the following list:

- Integration into a single model of event-driven, RTL behavior along with abstract algorithmic descriptions.
- Use of the Python integrated SciPy library to describe in Matlab-like style the behavior of arithmetic blocks.
- Use of SciPy functions to plot arithmetic variable and signals during simulation.
- Automatic dumping in VCD format of digital signals for I/O signal visualization using standard simulation tools, like ModelSim and GTKWave.

In Figure 2.3 it is shown that in order to interface a SystemC testbench with already existing RTL code, an external HDL file is required that defines the I/O signals of the Unit Under Test (UUT). In SysPy we adopted a more integrated solution, since the I/O is described within a testbench. Also a single class is enough to describe the abstract models that define digital behavior or even software sequences that will be executed by a programmable processor. Also it is easy to merge the models' description with RTL-like descriptions and express the functionality of a system in a bit-true and cycle-accurate manner.

Although SystemC is a well-defined and popular digital verification methodology, we believe that our approach is more unified, since it provides, using a single testbench file, more options for system debugging and also a more clear syntax for expressing a system's behavior. In addition our tool supports the silicon implementation of processor-centric systems using FPGA devices and ease the use of synthesis and software development tools. As described also in Section 3.3.2, in SysPy we support the feature of developing and co-simulating C code that can be later on ported and executed by the programmable processor of the designed system. So the high-level algorithmic descriptions in SciPy can be used to easily express and verify the functionality of an algorithm, while the code can be easily ported to C code and used in the silicon implementation.

2.3.3 SysPy's contribution to SoC FPGA design

In Table 2.1 a comparison of the Python related tools targeting digital hardware design is presented. All of the tools support Python code translation to RTL code, but only SysPy and *PHDL* support generation of FPGA synthesizable RTL code. Except from SysPy, only *MyHDL* supports behavioral simulation of a designed system at the Python level, but no Value Change Dump (VCD) file generation is supported, which is the industry's standard for storing simulation results. Except SysPy, none of the other tools support:

(a) processor-centric designs and related C compiler tools handling for software development.

(b) RTL code generation using parameterized Python functions.

(c) abstract simulation of a design by mapping hardware functionality to Python function and classes.

(d) hw/sw co-design using high-level hardware descriptions along with software expressed using C code

(e) simulation and generation of VCD files for top-level I/O signal visualization.

(f) automatic generation of Tcl scripts for FPGA synthesis tools.

Automatic generation of FPGA synthesizable HDL code in SysPy, combined with high-level simulation capabilities of the tool provides all the means for the verification, description

	Support for						
Tools	Python to RTL	FPGA synthesizable code	Behavioral simulation	Hw/Sw co-simulation	Processor-centric design	Synthesis tools integration	References
PyHDL	X	-	-	-	-	-	[39]
PHDL	X	X	-	-	-	-	[63]
MyHDL	X	-	X	-	-	-	[23]
PyMTL	X	X	X	-	-	-	[58]
PDSDL	X	-	-	-	-	-	[97]
SysPy	X	X	X	X	X	X	[61]

Table 2.1: Python digital hardware related tools comparison.

and implementation in FPGA device of a SoC. With the approach that we used to develop SysPy, most of the steps of a SoC FPGA design flow are supported. Combining RTL descriptions with object oriented structures and arithmetic models gives the designer the ability to: a) verify proper functionality of a hardware implemented algorithm and b) use complex data form as simulation input e.g. audio/image files, XML and Comma Separated Values (CSV) files. Taking into account the available simulation tools and the increasing complexity of embedded systems (32-bit CPUs, arithmetic units, large data/program memories) high-level simulation of a SoC is an innovative feature of SysPy and gives flexibility to decide about key features of a system early in the design phase.

By developing SysPy we tried to demonstrate how a popular language like Python, can be used with an abundance of freely available packages, to describe a SoC system in a high level of abstraction, verify it and deliver RTL FPGA synthesizable code. Python's key features are also demonstrated through the way the language is used in SysPy:

(a) Easy to understand and read syntax by people that are not closely related to hardware design e.g. software developers.

(b) Use of Python's core libraries (functions, classes, lists, dictionaries), and not any custom tool-related code, so that it is easy for a Python programmer to understand and use the supported design/verification flow.

(c) Use of a good lexical analysis tool, combined with Python's unique features, to store and manipulate text files, provides a very solid programming environment for scanning and manipulate textual data.

(d) The existence also of a good and freely available Matlab-like tool, such as SciPy, embedded into the Python environment, triggered our interest on building a simulation tool, especially targeting arithmetic hardware blocks, so that the designer can quickly acquire bit-true numerical results, using Python-level abstract model descriptions, that match his/her design specifications.

In all design and simulation steps of our design flow we use Python structures/syntax, and not any custom-defined syntax, to describe the datapath of the SoC and the related simulation models. All supported hardware description and simulation programming structures are compatible with the basic coding style used by the majority of Python programmers. In this way we tried to ease modeling and implementation of a processor-centric SoC even by software programmers with limited experience in hardware design.

Chapter 3

High-level design/verification methods

The design and verification flow we have developed is presented in this chapter. A detailed description is given for all the verification features targeting the simulation of a processor-centric System-on-Chips. Details are also provided on how to build high-level algorithmic simulation models, using the SciPy library and also on how generation of digital signal and arithmetic plots are used for defining key parameters of a SoC. Using also external C compiling tools, C functions can be developed and called, interchangeably with the SciPy functions, to implement hw/sw co-simulation, where the developed C code can be used to program the Leon3 processor during silicon implementation.

3.1 The design flow

Design flow of Figure 3.1 covers six major tasks related to the design of a processor-centric SoC:

1. Functional simulation of Python code describing the behavior of the hardware blocks and of the software executed by the processor in a cycle accurate manner.

2. Description (using Python or VHDL) of hardware components (modules) that are going to be connected with a processor soft core.

3. Incorporation in a SoC design of ready-to-use components (Python, VHDL, netlist format) and connection to a processor soft core.

4. Automatic generation of scripts facilitating the software development flow for the processor core, e.g. automated calls to C compiler tools, initialization of the processor's program memory in BRAMs etc.

5. Generation and execution of scripts to automate the processes involved in a SoC's design flow, e.g. Tcl scripts for FPGA synthesis tools etc.

6. Generation of meta-data XML description , compatible with the IP-XACT standard [15] and VHDL testbench templates.

In SysPy, Python acts as the backbone of a set of tools for hardware description as well as to incorporate other software tools. A normal design cycle starts by providing the simulation models of the desired system and also the timing information, e.g. main clock frequency, duration of the simulation, input data etc. The hardware description can have an HDL-like syntax or a more abstract algorithmic-style syntax. The first syntax style can later be translated automatically to synthesizable VHDL code, while the later one cannot be directly translated to VHDL, but can help a designer to easily verify the functionality of a SoC. Python classes can be used to simulate software modules of a system. The timing mechanism can then be used to provide timing information for a software or hardware component, treating in this way such a component as combinational logic that needs a specified execution time to provide results on its outputs. Placing registers among the blocks of combinational logic forms a pipelined synchronous datapath design, which is controlled either by software running on the processor or by hardware implemented Finite State Machines (FSM). All simulation tasks in Figure 3.1 are highlighted using arrows in gray color, while for hw/sw design tasks black color arrows are used.

The supported level of abstraction during simulation, combined with the provision of cycle-accurate digital waveform results, gives the ability to the designer to make decisions for critical parameters of the system early in the design phase. Synchronization and timing problems, especially found on the boundary of the processor/custom hardware interface, can be observed during simulation, using Python modules to model software and hardware blocks. Python can also be used to efficiently parse almost any kind of data format and use them as input to the simulation. In this way music, image or raw text ASCII files can be stored in Python arrays and processed during simulation.

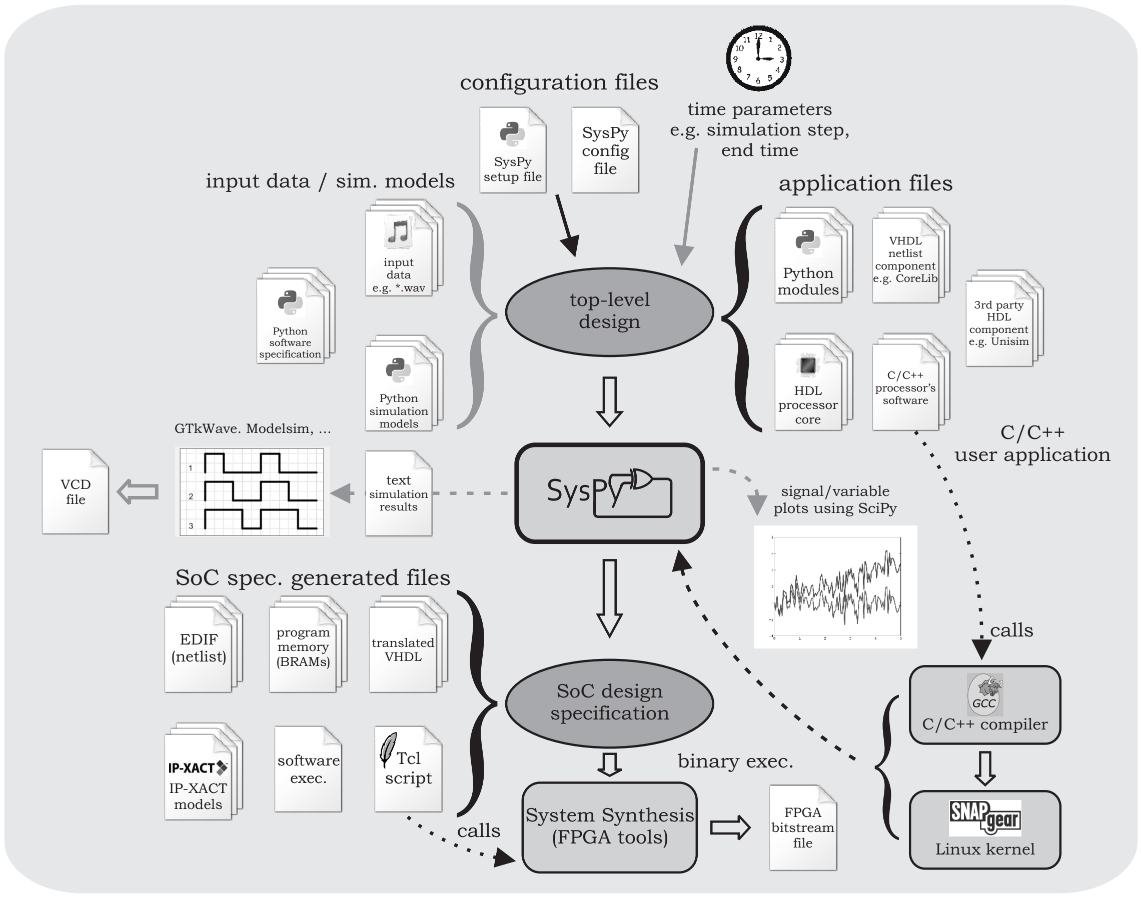


Figure 3.1: Processor-centric SoC design flow using SysPy.

All Python described hardware blocks used during simulation, e.g. combinational logic, FSMs, arithmetic blocks etc. can be directly translated to synthesizable VHDL. The Python abstract algorithmic-style hardware models, e.g. a complex arithmetic algorithm, must be manually ported by the user either to a Python HDL or directly to a VHDL description. Other components from a commercial library, such as Xilinx's Unisim [47] or CoreLib [53], can also be used along with components in pre-synthesized netlist format. All the ready-to-use blocks are supported in SysPy's libraries. The Python described processor software must be ported to a C description before it could be executed. However, SysPy makes a call to C compiler tools to automatically compile the software and generate the binary executable files for the processor.

Hardware synthesis tools require a number of scripts to automate the required steps needed to convert an RTL HDL description to a placed and routed design on an FPGA device. Scripts are needed to combine the large number of design files with the required timing and placement constraints files. A set of constraints is required especially for designs which implement ready-to-use blocks, like the processor core, where its implementation must meet several timing constraints. Interface of the processor with communication and memory related IC controllers on the FPGA board, e.g. SDRAM, Ethernet, USB et.c, requires the definition of a number of timing and placement constraints, otherwise it is not possible to achieve the required processing and communication performance. A set of constraints already exists for the supported processors in SysPy and according to the design files, the tool automatically includes the required script files and executes in command line all the synthesis and implementation steps using Xilinx's tools.

3.2 Hw/sw co-simulation features

SysPy can be used as an Architectural Description Language (ADL) [68] to verify the functionality of a digital system using abstract algorithmic descriptions. This is very useful in cases where a designer has to apply hw/sw partitioning and decide whether software implementation is sufficient for an algorithm or its function has to be mapped to dedicated hardware blocks. SysPy supports simulation models using combinational and sequential module descriptions, expressed in Python. The supported Python syntax for simulation is the same as the one used for RTL Python descriptions automatically translated by the tool to VHDL (see Chapter 4). The most challenging feature that we implemented in SysPy's simulation mechanism is the ability to simulate, along with the HDL-like descriptions, Python code that describes in an abstract algorithmic way hardware blocks that are not yet implemented at the RTL level.

The developed simulation mechanism can:

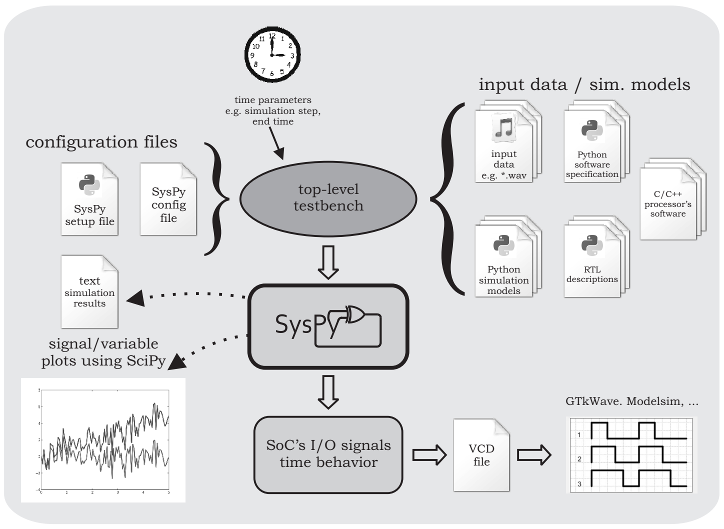
- simulate models of complex logic blocks, e.g. arithmetic units, communication controllers etc., for which there is no available RTL description
- support development of processor software directly in C and execute it in Python to prove its functionality in a system-level simulation
- simulate HDL-like Python RTL descriptions, e.g. logic gates, MUXs, state machines etc.

- save simulation results in VCD format and visualize them using already available simulation software, e.g. ModelSim, GTKWave etc.

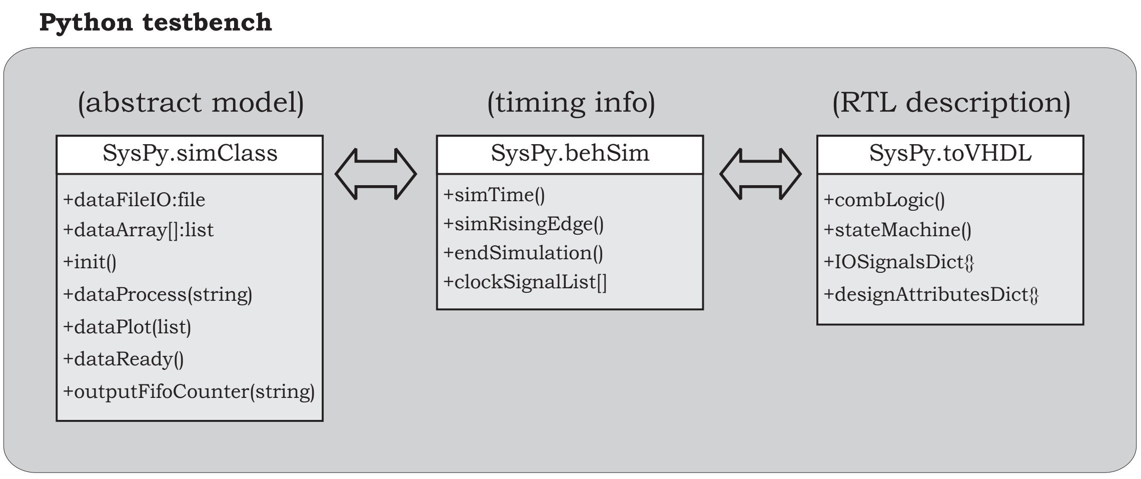
By combining RTL and abstract algorithmic model descriptions, all blocks of a system can be easily combined, while the models correspond to either hardware or software functionality. Especially for modeling complex arithmetic operations, the SciPy library can be used through SysPy. Ready-to-use arithmetic functions are available to simulate a numerical algorithm, where the results can be further processed and stored by already described RTL blocks. The arithmetic operation accuracy, in terms of the number of representation bits used in a system, can be easily explored, since the results are bit accurate. The timing performance and synchronization of a system can also be explored since the simulation results are also cycle-accurate. The combination of asynchronous and synchronous signals can be simulated along with the impact of delays introduced by combinational logic.

The VCD stored output data represent cycle and bit-accurate results. Inputs and outputs of abstractly defined digital blocks can be synchronized/registered using the developed SysPy timing mechanism and in this way simulate a digital pipelined datapath. The complexity of not yet defined blocks is “hidden” inside Python classes which provide functions to control the models’ functionality. Functions are used within classes to:

- Provide input data to a model
- Store processed data in arrays, files etc.
- Visualize processed data signals



(a)



(b)

Figure 3.2: (a) SoC simulation flow, (b) Simulation models - RTL descriptions interface.

In Figure 3.2a the flow for using a simulation testbench to verify the functionality of SoC is presented. A mixed system description is provided in the top-level testbench, including RTL descriptions combined with simulation models along with any related data files need to be processed during simulation. The models can describe in an abstract/algorithmic level either software or hardware functionality. Especially for Python algorithmic models that will be later on mapped to software functionality running in a programmable processor, SysPy supports the feature of executing software algorithms expressed in C within a Python testbench. In this way Python objects or C software functions can be called interchangeably to simulate software algorithms, while during implementation steps, the C code can, in an effortless way, be compiled and used directly to program the processor of the system. More information about developing, simulating and debugging C code in SysPy can be found in Section 3.3.2.

Timing information is added to the testbench in the form of clock signal definition and simulation step and stop time. Simulation results can be: a) text log files containing either information about the simulated calculations, or about the value of each I/O and internal signal during simulation, b) plots, using SciPy, of numerical results and c) VCD files where all the I/O signals' activity has been dumped during simulation. The feature of reporting numerical results in a text or plot format is very useful, since digital binary or hex values can be converted to corresponding fixed or floating-point format, where the accuracy of numerical calculations can be accessed.

In Figure 3.2b the interface in Python testbench between the abstract models and HDL-like RTL descriptions is shown. The models are provided by the user in the form of a class, containing functions that control processing of data objects by the model. In the Figure a typical diagram of a class named `simClass` is presented that contains functions for: initializing (`init()`) the class object, processing and plotting the available data (`dataProcess()` and `dataPlot()`) and for signaling the end of a processing cycle (dataReady()). Data structures are also included in the class, like the name of a file to be processed (`dataFileIO`) and an array that is used to store intermediate processing results (`dataArray[]`). The testbench timing information is provided by SysPy's `behSim` library, where functions included for: controlling timing of sequential elements (`simRisingEdge()`), reporting/logging the simulation time `simTime()`) and stopping the simulation at a desired time (`endSimulation()`). The pattern of a clock signal (frequency, duty cycle) can be define using a provided by the library list (`clockSignalList`). The required RTL logic can be described by using elements from SysPy's `toVHDL()` library.

3.3 Software simulation in a high-level verification model

To support the role of SysPy as hw/sw co-verification environment, we provide the tools to support: a) development of bit-true and cycle-accurate algorithmic models in SciPy compatible syntax, integrated with other digital hardware models and b) co-simulation of hardware models along with C software routines, that later on are compiled by SysPy and used to be executed by the embedded processor of a system. In the following two subsections these two innovative features of our tool are described and analyzed.

3.3.1 SciPy for algorithmic software development

An example of a testbench for a SoC design and the generated simulation results are presented in this section to provide an understanding of SysPy's verification capabilities. The system accepts data in the form of a text file and tries to perform a linear regression to calculate the linear equation that fits better the data. The system can be divided into three main functional blocks: a) the arithmetic co-processor that implements the regression algorithm, b) parsing of the text data file and storing the numerical values to a dedicated memory and c) data I/O to and from the arithmetic co-processor. A state machine described in Python is implemented in hardware, while parsing the data file can be easily handled in software. The arithmetic block is better to be fully implemented in hardware, so that the regression algorithm will be executed as fast as possible. A schematic of the SoC is presented in Figure 3.3.

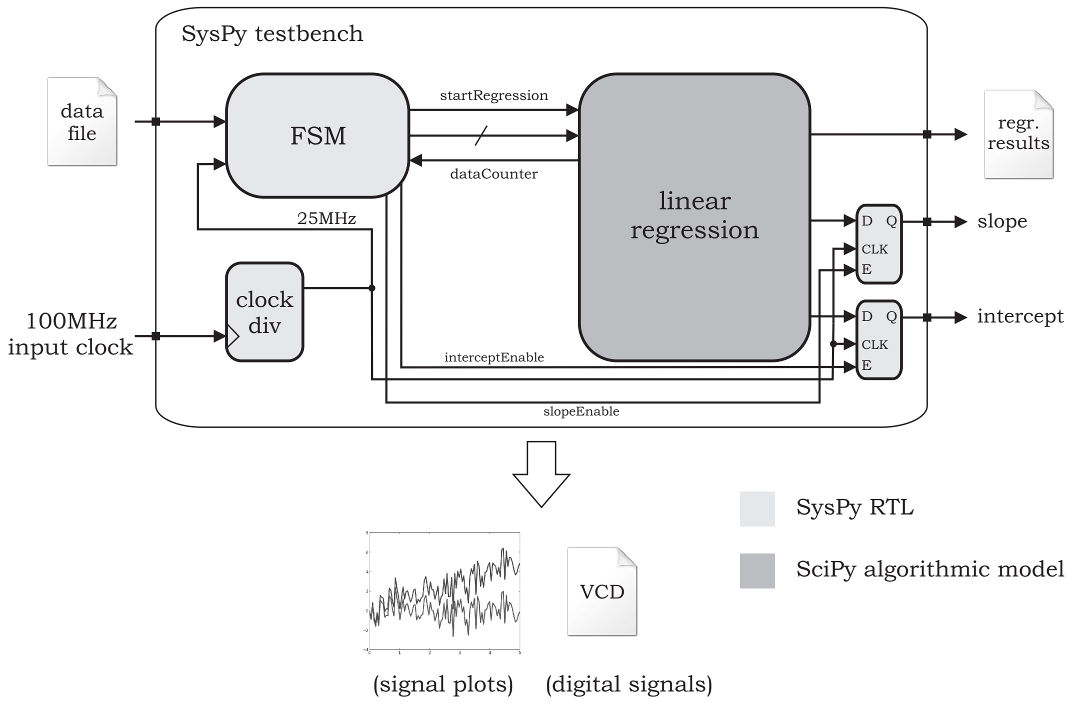


Figure 3.3: Top-level schematic of the linear regression SoC testbench example.

According to this partition we developed the testbench of the system in SysPy, where: a) the arithmetic co-processor is described in an algorithmic way using functions from SciPy's libraries, b) parsing of the data file is done using Python's core library functions and c) the state machine is expressed using Python RTL description supported in SysPy. The main clock input of the system is 100MHz, but a clock divider is used to reduce the frequency to 25MHz. The state machine exchanges data with the arithmetic block in an asynchronous way. A data vector with a predefined number of elements is transmitted from the FSM to the regression block, where the parameters of the linear equation fitting the data are calculated. The FSM is the controller of the process and signals the beginning and the end of the calculation procedure. The Python top-level module in SysPy is the wrapper of the testbench and handles all the file I/O activities. The calculated parameters of the linear equation are stored in a text file, while the `slope` and the `intercept` parameters of the equation are also presented as registered output values of the system.

The I/O interface of the SoC is presented in Code Example 3.1. Using associative arrays data structures in Python, called dictionaries, every signal or variable within a hardware RTL description can be fully characterized by specifying its name, size, type, direction and possible initialization value. A signal's dictionary, e.g. `i_sigs0` (line 12), defines using the related dictionary keys, concisely a group of signals of the same direction ('D': input), type ('T': binary) and size in bits ('L': 1-bit). In this example two internal signals are used as data counter (`dataCounter`) and as counter for the clock divider (`divCounter`). An internal signal is also used to store the states of the FSM. Extra keys are used to define parameters of the I/O signals of a testbench.The main clock sequence is generated using a for loop (line 4) and then assigned as initialization value to the clock input (line 20). The `start` input is used to trigger the processing procedure, after applying the global reset signal. The delay for output signals, modeling the delay of a combinational logic path, can be defined using the "del" key. An 11ns delay is used for the `slope` and the `intercept` output signals (line 16). These output signals are also registered using a control signal coming from the FSM a control signal of the arithmetic block. Input signal value sequences can be defined using the 'V' key, using pairs of time and assigned value. In this way the `rst` signal is applied for 5ns (line 19), while the clock sequence with the desired frequency is assigned to the clock input (line 20). Using this set of five dictionary keys supported in SysPy ('N':Name, 'D': Direction, 'T':

Type, 'L': Length, 'del': delay and 'V': Value) a signal can be fully characterized in terms of RTL description and simulation.

The description of the state machine is provided in Code Examples 3.2 and 3.3. The required SysPy libraries, `behSim` and `toVHDL` are imported along with the user-provided `linearRegressionSimFunctions` class of the simulation models used in this testbench (lines 1-4). A process is defined in (line 6) as a Python function, sensitive to the main system reset and clock signals. All the related counters, state machine and output signals get a reset value after the asynchronous `rst` signals has been triggered. Parameters of the simulation model are also initialized during system reset. The name of the data file to be processed is defined and also the fixed-point notation that is used for the signals (lines 19-20). A state machine is triggered by a rising edged of the main system clock (line 26). Data processing is enabled using the `start` signal (line 27) and when the signal `divCounter` has a value of three (line 30). This counter implements the clock divider to generate the required 25MHz clock. In state 1 (line 38) data elements are transmitted to the regression core, while the state machine is reading an internal counter of the model (line 42) to identify the number of data elements already transmitted. After ten elements have been transmitted the state machine moves to state 2, in Code Example 3.3, where the regression algorithm is executed. In state 3 the `slope` result (line 12) of the linear equation is presented on the related output signal. In state 4 the `intercept` result (line 17) is assigned to the output and also a plot of the linear curve fit and a file containing the regression's results are generated (line 23).

The class model (`linearRegressionSimFunctions`) of the linear regression arithmetic block is provided in Code Examples B.1, B.2 and B.3.

Using this testbench we were able to check the accuracy of the regression algorithm, by applying different fixed-point accuracy notation for the operands. The standard error of the linear fit was used as a criterion to choose the proper notation for the operands. In Figure 3.4 fit plots are presented using different fixed-point notations. Three fitting lines are presented using SciPy floating point calculations, converting the operand to 8-bit fixed-point numbers using an fp(4.4) notation and also using an fp(5.3) notation. It is clear from the plot that the results are better when we used 4-bits for the decimal part of the fixed-point numbers. The arithmetic results for each line fit are the following:

- SciPy fit - slope: 1.0147, interception: 1.3643, error: 0.9681

Code Example 3.1: I/O interface of the SoC performing the linear regression algorithm.

```
# Create a 100MHz clock sequence for 15us, 50% duty cycle
clk_seq = []
clk = 0
for i in range(0, 15000, 10):
  clk = not clk
  if (clk == True):
    clk_seq.append([str(i), '1'])
  else:
    clk_seq.append([str(i), '0'])

# I/O and internal signal declaration
i_sigs0 = {'D': 'i', 'T': 'b', 'L': 1, 'N': ["rst", "clk", "start", "clkDiv"]}
o_sigs0 = {'D': 'intr', 'del': 12, 'T': 'b', 'L': [0, 5], 'N': ["state", "dataCounter",
                                                              "divCounter"]}
o_sigs1 = {'D': 'o', 'del': 0, 'T': 'b', 'L': [0, 7], 'N': ["slope_d", "intercept_d"]}
intr_sigs0 = {'D': 'intr', 'del': 11, 'T': 'b', 'L': [0, 7], 'N': ["slope", "intercept"]}

# Define values for the testbench input signals
sim_sigs0 = {'D': 'sim', 'T': 'b', 'L': 1, 'N': "rst", 'V': [['0', '1'], ['5', '0']]}
sim_sigs1 = {'D': 'sim', 'T': 'b', 'L': 1, 'N': "clk", 'V': clk_seq}
sim_sigs2 = {'D': 'sim', 'T': 'b', 'L': [0, 7], 'N': ["slope", "intercept"]}
```

Code Example 3.2: FSM description for controlling the linear regression arithmetic block (part 1).

```
import SysPy_setup
import _toVHDL
import funcs._behSim
from linearRegressionSimFunctions import *

def proc_1(clk):
  # Signal's reset
  if (rst == 1):
    state = 0
    dataCounter = 0
    divClk = 0
    divCounter = 0
    slope = 0
    intercept = 0
    slope_d = 0
    intercept_d = 0

    # Define the data filename and fp notation
    SimObj.dataFileName = "data_file.txt"
    SimObj.fpNotation = "5.3"

    # Initialize the regression arithmetic model
    SimObj.init()

  # State machine description
  if (funcs._beh_sim.rising_edge("clk") == True):
    if (start == 1):

      # Clock divider counter (100MHz / 4 = 25MHz)
      if (divCounter == 3):
        divCounter = 0

        divClk = not divClk

        if (state == 0):
              state = 1

        elif (state == 1):

          # Write data to the model
          SimObj.writeData()
          dataCounter = SimObj.dataCounter
```

Code Example 3.3: FSM description for controlling the linear regression arithmetic block (part 2).

```
            # Provide a vector with 10 data elements
            if (SimObj.dataCounter == 10):
              state = 2
            else:
              state = 1

          elif (state == 2):
            # After loading the data start the regression algorithm
            SimObj.startRegression()
            state = 3

          elif (state == 3):
            # Slope output ready
            slope = SimObj.returnSlope()
            state = 4

          elif (state == 4):
            # Intercept output ready
            intercept = SimObj.returnIntercept()
            state = 4

            # Plot the fitted line over the data
            SimObj.plotRegressionResults()

            # Terminate the simulation
            funcs._beh_sim.endSimulation()

        else:
          divCounter = divCounter + 1
```

- fp(4.4) - slope: 1.0, interception: 1.375, error: 0.9706
- fp(5.3) - slope: 1.0, interception: 1.0, error: 1.0604

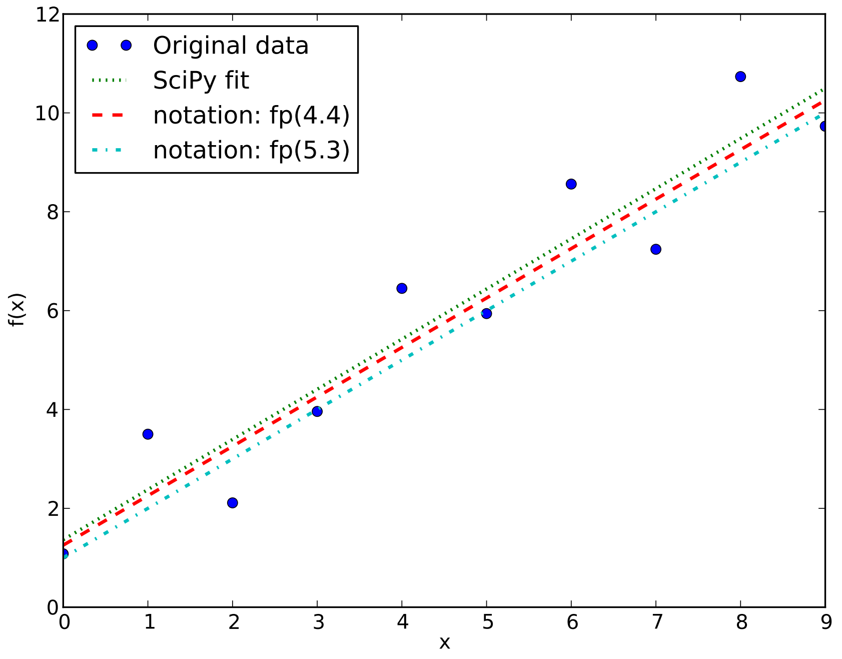
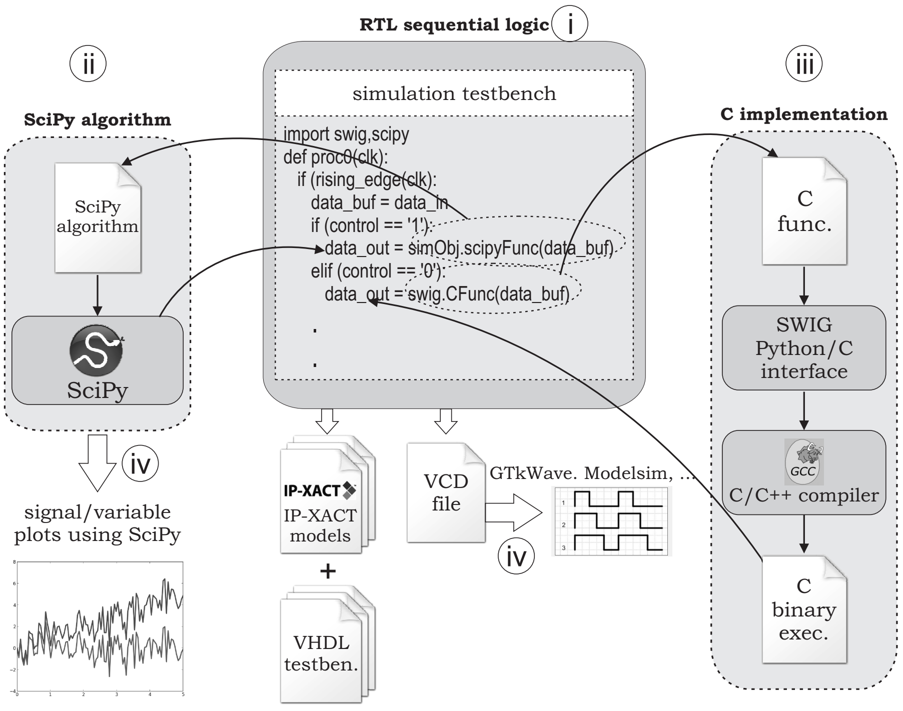


Figure 3.4: Linear regression fit plots using different fixed-point notations.

3.3.2 Using C tools in Python for hw/sw co-simulation

One of the reasons that we chose Python for the development of SysPy, except the language's clear syntax and powerful text processing features, is that Python scripts can be easily combined with and call C/C++ functions. Developing and using C code within a Python testbench in SysPy, provides a great advantage in terms of hw/sw co-simulation, since the software can be directly used to program a processor in silicon. To the best of our knowledge, SysPy is the only high-level digital verification tool that combines in the same simulation model: a) software algorithmic development using Matlbab-like syntax (in SciPy) and b) C functions that interact with the rest of the Python testbench (accept input arguments and return data results) and simulate algorithms which with a minimal effort can be later on mapped and executed by the embedded processor.



```
import swig,scipy
def proc0(clk):
  if (rising_edge(clk)):
    data_buf = data_in
    if (control == '1'):
      data_out = simObj.scipyFunc(data_buf)
    elif (control == '0'):
      data_out = swig.CFunc(data_buf)
    .
    .
```

Figure 3.5: Simulation flow in SysPy using RTL and algorithmic models.

Our goal is to use in a testbench, either Python, SciPy compatible, or C code in a transparent way to express algorithmic behavior. The suggested simulation flow, starting from a high level description in Python, would use SciPy syntax to quickly express software functionality, as in Code Examples B.1, B.2 and B.3, and interface with hardware descriptions in Python. For the software descriptions in SciPy to be useful and support hw/sw co-simulation in SysPy, we also provide the mechanism for expressing the software algorithms in C and call these C functions within the Python testbench.

A detailed flow of the C functions supported feature is presented in Figure 3.5. In the presented code example `control` signal select the execution of the proper function that handles processing of data contained in the `data_buf` signal. If `control='0'` then the SciPy implemented function is called, while if `control='1'` the C implemented function is called. Both function may accept as arguments digital I/O or internal signals (`data_buf`), defined in the top-level testbench using SysPy supported format (as shown in Code Example B.1) and also return data in compatible format to other testbench variables (`data_out`). The Simplified Wrapper and Interface Generator (SWIG) [86] is used within SysPy, to automatically compile the C code, using GCC compiler in Linux and to generate the Python interface function (`import swig`). A detailed example is provided in Code Example 8.1, on the way a SciPy function or a C function can be used interchangeably first to simulate and algorithm in Python and then to develop and co-simulate the application software in C. The interface also of the C function (input arguments and return variables) can be the same as the simulation models in SciPy, so it can be exchanged in place in the Python testbench with a minimal effort.

In Figure 3.5, a typical testbench i) describes, using HDL-like syntax, the main elements of a pipelined datapath. ii) Python code in SciPy is used to describe in an algorithmic way functionality of hardware blocks not yet defined in HDL. iii) The same functionality can be expressed using C code, in case the required SoC functions need to be ported to software executed by a processor core. iv) Signals plots are generated during simulation in SciPy to observe signals behavior and also SysPy generates VCD files to represent in binary format the systems I/O signals. We tried to promote the concept how a scripting language can be used to combine the algorithmic description features of SciPy, with the embedded software development features of C. Calling different functions within the Python testbench the user can: a) easily simulate the expected algorithmic behavior, using SciPy syntax, where the results are synchronized (datapath) with sequential logic and b) also simulate most algorithmic parts of the C implementation (using the SWIG Python/C interface and GCC in Linux) of his/her algorithm in case a software implementation is decided. In this way decisions about the timing performance can be made early before RTL implementation, while the software developers can also test and debug their embedded C code in a cycle-accurate high-level testbench, along with hardware behavior expressed in Python.

3.4 I/O signal visualization

Timing results of the SoC can be observed using the digital waveforms dumped in the automatically generated VCD file and visualized using the GTKWave [38] digital waveform viewer. Other simulation tools like ModelSim [66] can also be used to visualize the content of VCD files.

Using the provided clock signal information and also the assigned delays to specific output and internal signals, the timing concept of a SoC can be captured in the top-level Python description. Accurate timing simulation models can be acquired only when the physical design of a system has been accomplished, but during the initial design or specification steps it is very important to have description models where the timing information can be expressed.

In Figure 3.6 the main 100MHz (`clk`) and the divided 25MHz signal (`clkDiv`) are presented. Two data values are transmitted to the regression module at the rising edge of the divided clock and the value of the `dataCounter` is provided in the waveform along with the `state` information of the FSM. New values are assigned to the `slope` and `intercept` output after an 11ns delay during `state` 2, while the results are buffered to the SoC's output during `state` 3 and 4.

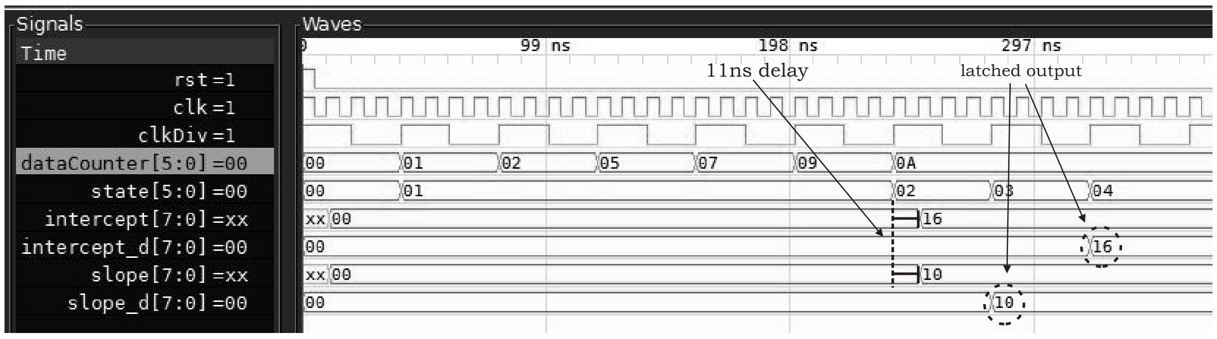


Figure 3.6: Digital I/O signal waveforms of the linear regression SoC.

The testbench format supported in SysPy provides all the required parameters that need to be explored from the designer. Simulation results can be taken into account to define the system's clocking frequency, the arithmetic calculation accuracy, the design of the state machine and also the architecture of the arithmetic block. If a higher clocking frequency is required then pipelining must be applied to the computational stages of the arithmetic block, which is considered as a combinational path with a fixed delay in our simulation. The Python environment also gives great flexibility in the way to define values patterns for the input signals, like the way the for loop was used in Code Example 3.1 (line 4). In this way the Design Under Test (DUT) is considered as a black box, where input data patterns are applied to it and arithmetic and digital plots are generated to assess the computational and timing accuracy of a system.

Chapter 4

RTL descriptions using SysPy

In this chapter we discuss the HDL generation features of SysPy and the features provided to ease the description of processor-centric systems. Behavioral and structural descriptions are supported to define interface and glue logic around an instantiated processor core. Use of Python functions to automatically instantiate hardware modules ease system integration activities and support the use of a more compact structural description coding style. Automatic generation also of VHDL testbenches and XML-based IP-XACT models for every module in a design support code re-use of RTL descriptions among a complete set of tools developed by different vendors, targeting digital design and verification.

4.1 Python to HDL translator

The translation mechanism of Python descriptions to VHDL is the basis of the SysPy design environment. It was very important right from the beginning to specify the subset of the Python data and control structures that we would use to describe a digital system a) in an HDL-like way especially for combinational glue and sequential control logic and also b) in an abstract way for parameterizing and instantiating ready-to-use blocks, e.g. processor blocks and also use Python algorithmic descriptions to auto-generate HDL code for arithmetic blocks. Except from describing or instantiating digital blocks, a way to define the I/O interface and block related parameters, e.g. size of a bus, was also required.

A lexical analyzer [28] is used to track the structures in the Python code that describe a digital system. Process structures with sensitivity lists are supported in behavioral descriptions, targeting the design of sequential circuits, e.g. Finite State Machine (FSM). All the basic arithmetic and logical operations for binary, integer and array operands are supported. It is also very important that user-defined Python components can be incorporated into the same SoC design along with pre-existing user-defined or 3rd party components that can be expressed in VHDL or in pre-synthesized netlists format. With the usage of Python dictionaries every signal or variable can be fully characterized by specifying its name, size, type, direction and possible initialization value. Every signal, variable, or attribute is checked against its declaration and the appropriate error messages are asserted as needed. Dictionaries in Python are lists indexed by keys and they can be seen as associative arrays. Dictionaries can be very useful in storing design or signal parameters, thus creating easy-to-read hardware descriptions.

4.1.1 Behavioral descriptions

SysPy supports behavioral, dataflow and structural descriptions of hardware modules in Python. In behavioral description, process structures with sensitivity lists are supported targeting the design of sequential circuits (FSMs). Arrays of binary or integer signals are modeled as RAM memories, implemented with Blocks RAMs (BRAMs) on FPGAs. Full functional BRAMs can be used separately to form larger memory systems. All signals and variables are declared as Python dictionary structures, while processes and port map assignments are declared using function statements.

In Code Example 4.1 a typical FSM description in SysPy is provided. A set of dictionaries is used to define the desired design attributes, generic parameters and signals. In this example, four elements are included in the `attributes` dictionary (lines 29-30): The `sign` element is used to specify that the design will handle signed signals. The `FSM_STYLE` and the `MULT_STYLE` elements are attributes related to the Xilinx Synthesis Technology (XST) tools. The former specifies the implementation style of FSMs (to be based on lookup tables) while the latter specifies whether DSP48 [49] blocks (Virtex family) or block multipliers (Spartan-3 family) will be used for the multiplication operations. The `FPGA_DEV` element indicates the targeted FPGA family and is used for the correct instantiation of components that are FPGA family (line 30) dependent, e.g. CoreLib netlists, BRAMs etc. A signal's dictionary, e.g. `iosigs0`, defines concisely a group of signals of the same direction (D), type (T) and size in bits (L). The bit size of signals may depend on a generic parameter (n in this example). All signal names are provided after the N declaration and signals can be initialized to a specific value using the V declaration. For example, `iosigs1` defines an output signal of type binary and length n named `PORTA` that is represented as binary array with elements indexed `(n-1) downto 0`.

The state machine is described with one process block (line 5), which has the reset and the state transition logic. The value of the `buf` signal is defined in every state according to the value of the `ctl` control signal. Combinational assignments are handled outside the process, which is driven by a gated clock signal. Clock gating is controlled by the "clkCtl" signal (line 25) and the output results is presented in the "PORTA" signal, using a combinational assignments (line 26). The "toVHDL()" function is called at the end (line 45) to generate the VHDL design files. The function accepts as arguments the name of the top-level design description file and the dictionaries of all the I/O and internal signals, as well as the generics and the attributes dictionaries.

4.1.2 Structural descriptions

While behavioral descriptions can be used to define glue logic among the main digital blocks of a design, SysPy also supports structural descriptions, where ready-to-use blocks can be utilized. The blocks can be described in Python or HDL syntax or they can be in the form of a synthesized netlist. Especially the netlist description format can be very useful, since many FPGA design tools provide ready-to-use complex arithmetic and communication blocks using this format. All components used in a top-level SoC description have to be declared in a Python library file, called the component library. For every block used in a structural description, the I/O information must be provided, along with any existing generic parameters related to the instantiation of the block.

Code example 4.2 presents the declaration of a user supplied custom core in the component library of SysPy. The I/O signals of the core are shown in Figure 4.1. The core is declared as a Python parameterized function in the component library, where the supplied argument ("bus_size"), declared in the "generics" list, is a generic parameter of the core. The

Code Example 4.1: Typical FSM description in Python, using the supported HDL-like description syntax in SysPy.

```
from inspect import *
import SysPy_ver._toVHDL
def demo_FSM():
  # Behavioral code (process)
  def proc_0(gatedClk, rst):
      if rst == '1':
        buf = "00000000"
        state_hrb = s0
      elif rising_edge(gatedClk):
        # FSM declaration
        if state == s0:
          if ctl == '0':
            buf = "00000001"
          else:
            buf = "00000010"
          state = s1
        elif state == s1:
          buf = buf * "00000011"
          state = s1
      else:
        buf = "00000000"
        state = s0

  # combinational assignments
  gatedClk = clk & clkCtl
  PORTA = buf

# Design atrributes
attributes = {"sign": '+', "FSM_STYLE":"lut",
              "MULT_STYLE": "block", "FPGA_DEV": "Virtex5"}

# Generic parameters
generics = {'n': 8}

# I/O signals
iosigs0 = {'D': 'i', 'T': 'b', 'L': 1, 'N': ["rst", "clk", "clkCtl", "ctl"]}
iosigs1 = {'D': 'o', 'T': 'b', 'L': ["(n-1)", 0], 'N': "PORTA"}

# Internal signals
intrsigs0 = {'D': 'intr', 'T': 's', 'L': 1, 'N': "state", 'V': ["s0", "s1"]}
intrsigs1 = {'D': 'intr', 'T': 'b', 'L': ["(n-1)", 0], 'N': "buf"}
intrsigs2 = {'D': 'intr', 'T': 'b', 'L': 1, 'N': "gatedClk"}

# Calling "toVHDL" converter function
SysPy_ver._toVHDL.toVHDL("demo_FSM", attributes, generics, iosigs0, iosigs1,
                         intrsigs0, intrsigs1, intrsigs2)
```

Code Example 4.2: Custom block declaration in SysPy's component library.

```
#Component library declaration of the ‘‘demo_FSM’’ block
def demo_FSM(bus_size):
   CompLib=‘‘custom’’
   generics=[True, ‘‘bus_size’’]
   signals=[{‘D’: ‘i’, ‘T’: ‘b’, ‘L’: 1, ‘N’: [‘‘rst’’, ‘‘clk’’,‘‘clkCtl’’,
            ‘‘ctl’’]},
            {‘D’: ‘o’, ‘T’: ‘b’, ‘L’: [‘‘(bus_size-1)’’,0], ‘N’: [‘‘PORTA’’,
             ‘‘PORTB’’]},
             [CompLib, generics, ‘‘demo_FSM’’]]

   return signals

#-----------------------------------------------------------------------------------------

#Function library declaration of the ‘‘demo_FSM’’ function associated with the
#‘‘demo_FSM’’ component
def demo_FSM():
         #Dictionary for the component association
        func_info = {‘‘Comp_name’’: ‘‘demo_FSM’’, ‘‘numGPIOPorts’’: 1}

        return func_info
```

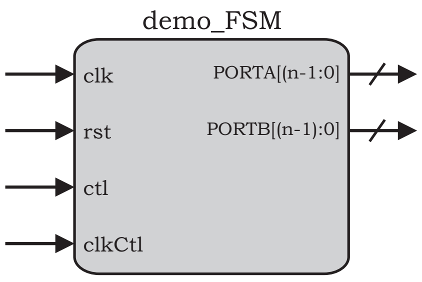


Figure 4.1: Demo FSM pinout.

"True" flag (line 4) in the "generics" is used to define the existence of generic parameters in the design. All the I/O signals are declared using Python dictionaries, in the same way as it was described in In Code Example 4.1. All the signals' dictionaries are merged together in the "signals" list (line 5). The library of the core is declared as "custom" ("CompLib" variable, line 3), since the core has been provided by the user in the form of a Python or VHDL description. Other blocks in an RTL or netlist form from commercial libraries can also be used, such as Xilinx's Unisim or CoreLib library. Every block in the component library is defined using a function and after the instantiation of the block in the top-module, the function will return the signals, the generics and the CompLib lists which contain all the required information for the block. Every time a block is connected in a structural description, its instantiation is checked against the information provided in the component library and the appropriate errors are asserted if needed. A tool has been developed as part of the SysPy project that parses component entity declarations in a VHDL package and generates component declarations in Python syntax, compatible with the corresponding component library format.

Since a module is defined as a function, it is possible in SysPy to hide some of the low-level details, wherever it is possible and connect a module in a more abstract way. In this way each module in the structural library can be associated with a Python function, declared in SysPy's function library using the format shown in Code example 4.2 (line 17). The function's body contains a dictionary ("func_info") with the name of the associated component (`demo_FSM`) and any other special attribute required by the function to instantiate the associated block. A special argument is used in this example (`numGPIOPorts`) (line 19) to define the number of I/O ports of the state machine. A function handler combines the information of the modules' declaration along with data provided in the associated function.

In this case the number of GPIO ports can have the value of 1 or 2, so either only `PORTA` will be implemented or `PORTA` along with `PORTB`. The description of this function handler is presented in Code Example 4.3. The function (`demo_FSM_ports()`) accepts as arguments the number of the required GPIO ports and the signals' dictionary list, as it was defined in the components library. An if-else structure (line 10) is used to remove the declaration of `PORTB` (line 14-15) if only one data ports is required. The usage of handlers to modify a blocks' connection and declaration in a structural description, shows how Python's compact and easy to read coding style can be used to write abstract hardware descriptions. A more detailed and complex example is presented in Chapter 7, where a function is used to initialize a block's memory arrays with the content of an XML file (see Figure 7.6).

Code Example 4.3: Function handler for automatic instantiation of a predefined block.

```
 1 def demo_FSM_ports(IOPorts, numPorts):
 2   """
 3     FUNCTION: demo_FSM_ports(a{}, b int)
 4     a: dictionary containing function's IO ports
 5     b: integer number of implemented data ports
 6
 7     - Function handler for "demo_FSM()".
 8   """
 9
10   if (numPort == 1):
11     ## One data port implemented (PORTA)
12     for i in IOPorts:
13       ## Remove PORTB
14       if (i['N'] == "PORTB"):
15         IOPorts.remove(i)
16   else:
17     ## In any other case two ports are implemented
18     pass
19
20   return IOPorts
```

In Figure 4.2, a Unified Modeling Language (UML) sequence diagram shows the way we combine the information about a hardware block from the function and the component library, along with the way a block must be instantiated, described in the related function handler. In Figure 4.2 a special argument such as "arg2" used in "function2". While "arg0" and “arg1” represent the values of generic parameters of “component2”, “arg2” triggers the execution of the handler, where “arg2” may represent the name of: a) a VHDL file that will be generated, b) a text file that contains initialization values for BRAMs, c) a Tcl or Python script file that can be used e.g. to generate design files or folders for other FPGA design tools. The usage of functions favors compact abstract Python descriptions, while port-map assignments give complete flexibility in terms of connectivity of a component, e.g. the user can define internal or I/O signals to connect to the ports of a component. Every function in the function library has a) a related block in the component library and can also have b) a function handler which will process any special arguments related to a block, like the “numPort” argument in Code Example 4.3.

Special arguments cannot be handled directly by an HDL language, like processing the content of a text file (see Figure 7.6) and make decisions about the structure of a block, e.g. increase the size of memory blocks or change the number of I/O ports). Using function handlers in SysPy adds more flexibility in Python RTL descriptions, where Python is used to auto-generate in a smart way the required RTL description, based on block information defined in the component library.

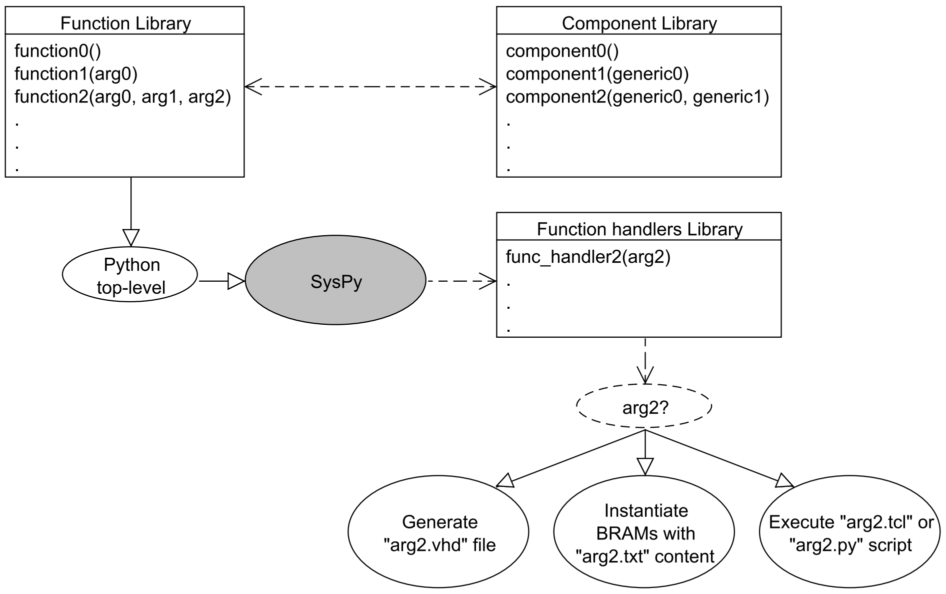

Function Library
function0()
function1(arg0)
function2(arg0, arg1, arg2)
Component Library
component0()
component1(generic0)
component2(generic0, generic1)
Python top-level
SysPy
Function handlers Library
func_handler2(arg2)
arg2?
Generate "arg2.vhd" file
Instantiate BRAMs with "arg2.txt" content
Execute "arg2.tcl" or "arg2.py" script

Figure 4.2: UML sequence diagram depicting the usage of function handlers in Python structural descriptions and the interaction among different modules of SysPy’s libraries.

While we support Python RTL-like descriptions we also provide the required level of abstraction using Python's programming and data structures. Functions in SysPy are used in such a way to automate the insertion of ready-to-use digital blocks, while through the use of dictionaries a structured way is also provided to declare I/O and parameterization information of a digital module.

4.2 RTL verification models

Except the generated VHDL descriptions, SysPy automatically generates different models and files of a Python described design. Python is used to extract the required information from the user's descriptions and generate different views of a design useful in the verification flow of a system. VHDL testbench and IP-XACT model files are generated for each Python described module, which can be used by third-party verification tools (Xilinx ISE [50], Mentor Graphics ModelSim [66]) to simulate the generated RTL code. In this way Python support not only the use of high-level verification models expressed in Python (see Chapter 3), but also RTL functional simulations, especially useful for block-level verification.

4.2.1 VHDL testbench template

To ease verification procedure of automatically generated VHDL code, SysPy also generates VHDL testbench templates for each Python hardware module. Within the testbench description the module-Unit Under Test (UUT) is instantiated, according to the I/O information provided in the Python description. A clock process is also provided, where the use has to define the frequency of the main system clock. The input stimuli can then be defined by the user in respect to any existing clock signals in the design. The testbench files are included along with the design VHDL files in the design project generated for the Xilinx design tools. In Figure 4.4 the auto-generated testbench template of the "demo_FSM" block in Figure 4.1 is presented.

In the beginning of a VHDL testbench some default libraries need to be used (lines 1-4), while in the architecture block of the I/O signals are defined (lines 9-18). Definition of any existing clock driver is assigned according to the proper clock frequency (line 21) and all the I/O signals are instantiated in testbench file (lines 23-30). Duty cycle is defined as a 50

Code Example 4.4: VHDL testbench template of the "demo_FSM" block in Figure 4.1.

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
-- Entity declaration
entity demo_FSM_tb is
end demo_FSM_tb;

architecture behavior of demo_FSM_tb is
        -- Component Declaration for the Unit Under Test (UUT)
        component demo_FSM
        port (
                clk: in std_logic;
                rst: in std_logic;
                clkCtl: in std_logic;
                ctl: in std_logic;
                PORTA: out std_logic_vector(7 downto 0);
        end component;

        -- Clock period definition
        constant clock_period: time := 1ns;
begin
        uut: demo_FSM port map (
                clk => clk,
                rst => rst,
                dimem => dimem,
                clkCtl => clkCtl,
                ctl => ctl,
                PORTA => PORTA
                );

        -- Clock process with 50% duty cycle
        clock_process: process
        begin
                <clock_name> <= '0'
                wait for clock_period / 2;
                <clock_name> <= '1'
                wait for clock_period / 2;
        end process;

        -- Stimulus process
        stimulus_process: process
        begin
                -- Enter your testbench stimulus here
        end process;
end;
```

4.2.2 IP-XACT models

Generation of RTL code from Python descriptions in SysPy is compatible with the IP-XACT XML schema. Using meta-data information in XML format, the IP-XACT standard [15] tries to ease, integration and IP-reuse of IP blocks among many different digital design and verification tools. Using keys in XML format several properties of an IP core can be described using the IP-XACT standard and different tools can parse information about a block without reading the associated RTL description. This feature provides information about the properties of a digital block in a higher, meta-data level. The IP-XACT XML schema is an IEEE standard, is managed by the SPIRIT consortium and is supported by major companies in the semiconductor industry like Intel, ARM, AMD, NXP, Texas Instruments and Synopsys.

SysPy automatically generates the IP-XACT descriptions during a Python to VHDL translation. The description contains filepath information about the generated VHDL files along with I/O information of the associated IP block. In Code Example 4.5 we present the IP-XACT representation of the "demo_FSM" block in Figure 4.1. XML tags can be used to provide general information about the block (lines 2-13) along with the I/O signal properties (lines 14-46).

The IP-XACT generation feature in SysPy proves that Python can easily generate and process many different views of an IP core in different syntax and languages [55]. The IP-XACT standard promotes reuse of existing cores and standardizes the way information about IP cores can be catalogued and also describes it in a format that can be processed by system integration and verification tools. IP-XACT keys can also be used to describe memory mapping of peripheral blocks and their interconnection properties on a data bus. An example on how useful the standard can be in a system-level, Universal Verification Methodology (UVM) [81] compatible, verification environment is the way CPU registers can be easily mapped and converted to SystemVerilog models, by using the meta-data information provided for the registers in IP-XACT format [26]. While SysPy already provides tools for building and simulating a SoC, with the addition of the IP-XACT generation feature it proves its ability to be compatible with all the industry standards related to digital hardware design and verification.

Code Example 4.5: XML description of the "demo_FSM" block in Figure 4.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:name>demo_FSM</spirit:name>
<spirit:description>Generated by SysPy, Author: Evangelos Logaras</spirit:description>
<spirit:fileSets>
  <spirit:fileSet>
    <spirit:name>fs-vhdlwrapper</spirit:name>
    <spirit:file>
      <spirit:name>/home/sim_test/SysPy/work/demo_FSM.vhd</spirit:name>
      <spirit:fileType>vhdlSource</spirit:fileType>
      <spirit:logicalName>demo_FSM_lib</spirit:logicalName>
    </spirit:file>
  </spirit:fileSet>
</spirit:fileSets>
<spirit:model>
  <spirit:views>
    <spirit:view>
      <spirit:name>vhdlwrapper</spirit:name>
      <spirit:envIdentifier>:modelsim.mentor.com:</spirit:envIdentifier>
      <spirit:envIdentifier>:vcs.synopsys.com:</spirit:envIdentifier>
      <spirit:envIdentifier>:designcompiler.synopsys.com:</spirit:envIdentifier>
      <spirit:language>VHDL</spirit:language>
      <spirit:modelName>demo_FSM</spirit:modelName>
      <spirit:fileSetRef>
        <spirit:localName>fs-vhdlwrapper</spirit:localName>
      </spirit:fileSetRef>
    </spirit:view>
  </spirit:views>
  <spirit:ports>
    <spirit:port>
      <spirit:name>clk</spirit:name>
      <spirit:wire>
        <spirit:direction>in</spirit:direction>
        <spirit:wireTypeDefs>
          <spirit:wireTypeDef/>
        </spirit:wireTypeDefs>
      </spirit:wire>
    </spirit:port>
    <spirit:port>
      <spirit:name>ce2int</spirit:name>
      <spirit:wire>
        <spirit:direction>in</spirit:direction>
        <spirit:wireTypeDefs>
          <spirit:wireTypeDef/>
        </spirit:wireTypeDefs>
      </spirit:wire>
    </spirit:port>
```

Chapter 5

Processor-centric SoC designs

Utilizing a processor soft IP core in a SoC design is a challenging task in terms of: a) handling implementation of the core by logic synthesis and physical design (placement and layout) tools, b) handling software development flow and related tools and c) connecting the processor with the rest of the design in a way that the processor does not decrease computing performance of other blocks. The processor acts as a communication and control gateway, since it is much easier to implement in software the layers of data transmission protocols and control algorithms. In this chapter we present how the supported processor cores are instantiated, using the supported syntax, in a Python top-level description. We also present how the tool is used to ease development of bare metal and O/S-centric applications. Software development for the processor core is also supported along with Tcl scripts to create design projects and execute in an automated way the related Xilinx FPGA design tools.

5.1 Processor core instantiation

Structural and behavioral RTL descriptions in SysPy have been supported in such a way to favor the instantiation of processor-cores in a design. Several parameters have to be set related to the available resources that the core will utilize in the FPGA. Also dedicated constraint files must be used during synthesis for every processor core, so that the design will meet the desired timing and optimal placement requirements.

SysPy generates synthesizable VHDL code of the RTL description of the SoC and imports in the design folders the appropriate timing and placement constraints of the utilized processor core. The included constraint files in SysPy cover the usage of all three supported processor cores for implementation in one of the three Xilinx FPGA families that we used: Spartan3, Virtex2Pro and Virtex5. Constraints files for other FPGA families can be easily added to SysPy's libraries.

The steps of the implementation flow used by the FPGA tools, like XST from Xilinx, is presented in Figure 5.1. The constraints files along with the RTL files generated by SysPy or fetched from the tool's libraries (ready-to-use blocks) are provided as input to the synthesis tools. Synthesis tool transform the design into a flat netlist, where nets are used to connect the main building blocks of a design, e.g. memories, adders, multiplexers, gates etc. As a next step, the synthesizer maps all the building blocks and any associated glue logic into technology-specific implementation. In the case of FPGAs, the synthesizer infers and connects hardwired macro blocks, such as BRAMs, arithmetic units, clock and I/O buffers etc. According to the provided timing constraints, in the case of a sequential design, the synthesis tool selects the appropriate blocks to build the clock tree structure to propagate the main clock signals through the design with the minimum latency. In the case of an FPGA synthesizer, the generated netlist also contains the Look Up Tables (LUTs) to program the inferred logic blocks (CLBs), which usually implement glue or control logic around the utilized macro blocks. For example, in the case of an arithmetic unit design, the arithmetic operations will be mapped in a hardwired macro adder/multiplier block, while the data flow and data buffering control will be realized with CLBs implementing an FSM and a number of shift registers.

In case of a processor core implementation, the clock tree must support the required frequency that the processor must be clocked at. Also special memory structures, like the register files and the cache memories are realized using BRAMs. Special high performance peripheral units of the processor, like data, image or audio controllers, also have a set of timing constraints that defines the relation of their clock signals with the clock signals of the Central Processing Unit (CPU). The designer must manually edit the processor constraints files already imported by SysPy, in case new timing constraints have to be added, reflecting the behavior of added custom logic, e.g. new or modified peripheral controllers.

After all the technology-specific logic has been inferred by the synthesizer, the related netlist file is generated and then the tool performs the floorplanning and the Place and Route

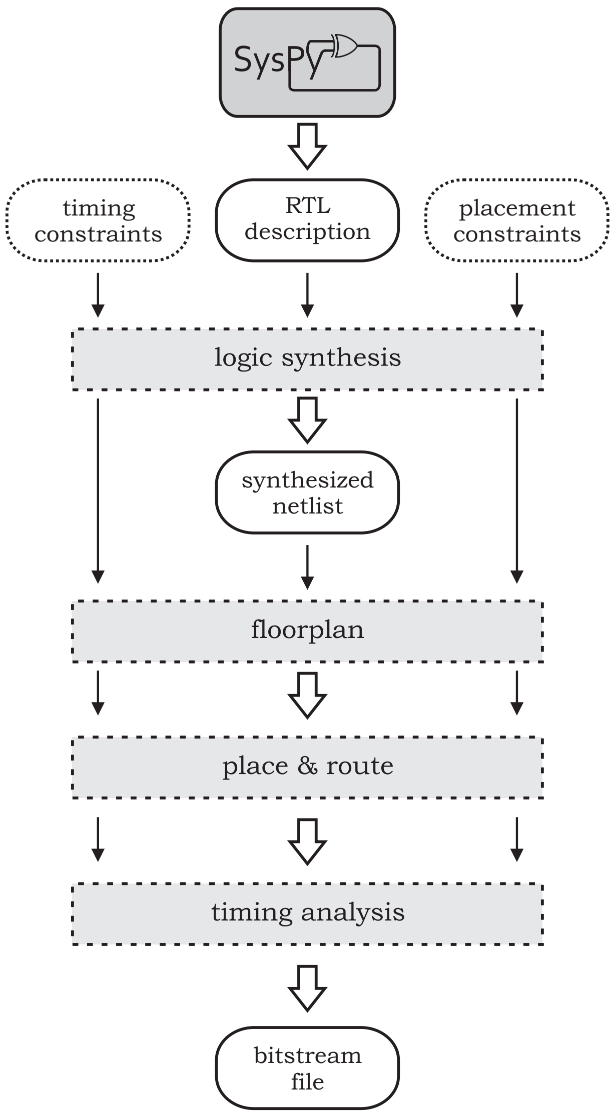


Figure 5.1: Design flow adopted by FPGA tools, from RTL design down to the generation of the FPGA bistream file.

(PR) steps. During these steps the tool does the physical design of the SoC, where it selects the location of the blocks that are going to be utilized on the FPGA device and also the paths through the available programmable switches to connect them. The location and the paths are selected in such a way to map similar logic functions close together and reduce the length of routing paths. Placement constraints can be used to lock the location of specific utilized blocks and also to define the connection of the design to the FPGA I/O blocks and pins, where other ICs are connected. In this case high speed I/O blocks can be used when a fast connection is required, e.g. connection to an external memory IC, or slower, power optimized blocks can be used when power consumption is the critical factor, e.g. connection with a slow serial bus controller.

Finally all the information regarding the physical placement and routing of a design is converted into a bitstream executable file which is downloaded to the memory of the FPGA device. The content of the bitstream programs the LUTs in the CLBs and also the programmable switches to form the desired signal paths.

5.2 Processor interface

One of the most challenging design task when adding a programmable controller, like a processor, in hardware design, is the interface logic on the software/hardware boundary. The specifications of this interface block defines the overall system performance. The processor can easily handle in software all the interface tasks of a system, especially when data transfer protocols must be used to communicate with other ICs on the FPGA board or even ICs on different boards or a host PC connected to the FPGA board. On the other hand the custom hardware blocks must be able to perform data processing operating at a high speed frequency and exchange data with the processor in a time efficient way, so that the operation of these blocks is not stalled by the slower operating frequency of the processor.

Fast local memory blocks have to be included in the design of the interface module to store intermediate processing results. In order to have fast access, memory blocks that support dual port access were designed, so that the processor and the custom hardware block could perform read operations in parallel. The memories were used in a dual First In-First Out (FIFO) configuration, while the size of the memories and the number of available data ports were implemented as configurable generic parameters in the HDL description. The interface block was described in VHDL and added to SysPy's component and function library, so that it can be instantiated in a design using a function call.

Two different types of interface blocks were implemented: one for the AVR core and another one for the Leon3 and the OpenRisc processors. The first one was designed using BRAMs connected as a FIFO memories and an FSM was used to control data flow through the memory. For the second implementation, a dedicated FIFO design was parameterized and implemented using Xilinx Core Generator [53], which is a Graphical User Interface

(GUI) for instantiating ready-to-use blocks from the Xilinx CoreLib library. More details about the design of these interface blocks are provided in Chapters 6, 7 and 8, where the design examples, where the blocks have been instantiated, are analyzed.

5.3 Embedded software flow

5.3.1 Using compilers for different processor architectures

Although emphasis of this work is not in software development, we wanted to be able to have a single description in Python that could support the related hardware and also define the software files executed by the processor. In this way the top-level description of a processor-centric system is self-contained since a designer can easily figure out, by examining the description, the processor core that is used, the surrounding glue logic and any custom peripheral blocks and also the software executed by the processor.

In Figure 5.2 the envisioned software development process is presented. All three currently supported processor cores are supplied under a GPL license, while a C compiler under the GCC project is also available for each one of them. The design flow automatically invokes the required C compiler tools to compile software for the utilized processor core. SysPy makes a system call to the appropriate compiler, which generates the binary executable file from the user's C/C++ code file. In the case of OpenRisc and AVR processors, VHDL files are generated containing BRAMs instantiations, initialized with the hexadecimal code of the executable file. In the case of Leon processor only the executable file is generated that can be used to program the processor's external memory with the GRMON [10] debug tool provided with the processor core. Especially for the openly available Leon3 processor IP core, software development using an embedded 32-bit Linux O/S is supported, as SysPy automatically compiles the user's C application along with the Linux kernel. The supported Linux distribution is the Snapgear Linux kernel [84]. In order to prepare the O/S kernel for compilation, SysPy copies the user provided custom application files to the O/S folders and registers the new application in the related configuration files. The tool also configures O/S's parameters, e.g. program/data memory size etc., related to the hardware implementation of the processor. These parameters are provided by the user in the top-level description of the system. After compilation, the O/S executable file contains the user's custom software as system application, which can be executed by the serial terminal, provided through an RS-232 connection.

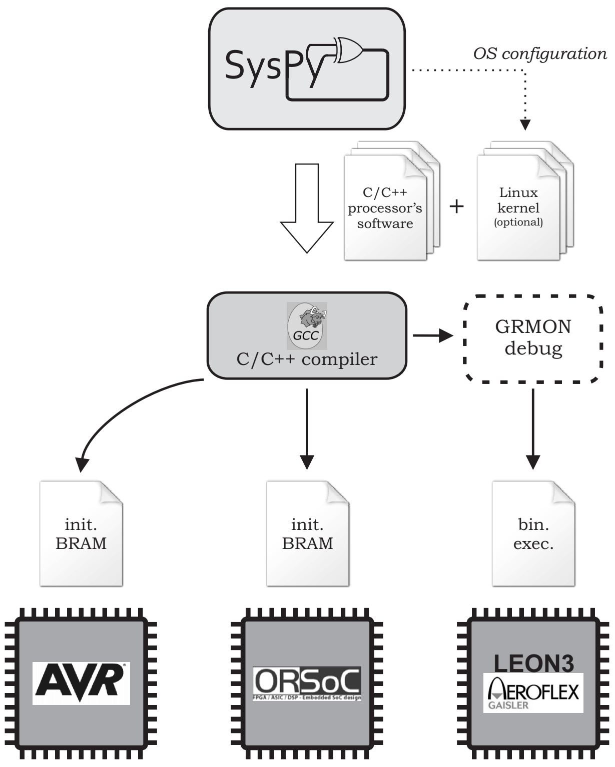


Figure 5.2: Software compilation flow for the three supported processor architectures.

In Code Example 5.1 the Leon processor core (`struct_leon3mp`) is instantiated in the design and connected to another custom block (`struct_FSM`). The attributes dictionary contains the `SYS_FREQ` and the `PROC_FREQ` attributes which define respectively the system's main clock frequency and the processor's frequency in MHz. This frequency configuration is valid only for Leon's implementation, since there is a parameter in its VHDL description that affects the operation of a Digital Clock Manager (DCM) block connected to Leon. The processor's frequency can be defined using a 10MHz step (160MHz maximum frequency). The attributes dictionary also contains the names of the C files (`linux_kernel`, `usr_app_classification`) that are automatically compiled using the "Sparc GCC" compiler to create the binary executable file for the Leon processor and the family of the FPGA device that will be used for implementation. A Tcl script is automatically generated and executed, to prepare the complete hierarchy of files needed to form a project for Xilinx synthesis tools. Linux kernel parameters can be specified in a configuration file, e.g. `RAM_SIZE, ETHERNET_EN, UART_EN` etc. Using a configuration file, SysPy can be used as a Python package-tool. In this file (*.ini), the user can define all the appropriate system paths (e.g. SysPy directory, gcc compiler directory etc.).

Code Example 5.1: Processor core instantiation in the top-level description.

```
import SysPy_ver._toVHDL

 # Connecting the Leon3 core
 def struct_leon3mp():
    sys_rst_in = sys_rst_in
    clk_out = clk_int
    clk_100 = clk_100
    clk_200_p = clk_200_p
    clk_200_n = clk_200_n
    clk_33  = clk_33
    sram_flash_addr = sram_flash_addr
    sram_flash_data = sram_flash_data
    PORTA_out = inputA
    PORTB_out = inputB
    .
    .
    .

 # Connecting the a custom block to the Leon3 processor
 def struct_FSM(``32''):
    clk = clk_int
    ce2int = ce2int
    PORTA_in = inputA
    PORTB_in = inputB
    .
    .
    .

# Leon's software C file names
attributes = {``SYS_FREQ:'' 100, ``PROC_FREQ'': 160,
               ``PROC_SW'': [``linux_kernel'',
               ``usr_app_classification'',
               ``FPGA_DEV'': ``Virtex5''}
```

5.4 FPGA design tools scripting

Hundreds or some times thousands of files are required to be generated or edited during the design cycle of a SoC on an FPGA device. In order to efficiently handle all the design steps and also be able to rapidly "respin" and optimize the design, scripting tools are required to automate as many design and implementation steps as possible.

A Tcl script ensures portability, since a design will be synthesized in the same way independently of the host computer. The script also captures the desired design setup, so that a design work can be shared among many designers. This feature can significantly ease maintenance of a design and of the associated file hierarchy. Scripts can also be executed in a command line mode, where the design steps are executed much faster than using a GUI interface or entering the commands manually.

Except handling the large number of files related to the design, a Tcl script must also control the synthesis and floorplanning activities. FPGA synthesis process is a fairly complex task where many parameters have to be defined by the user. Especially for the implementation of processor soft cores, as mentioned in Section 5.1, many timing and placement constraints have to be used.

SysPy automatically generates a Tcl script that can be used to run all the implementation steps on the XST design tools, according to the processor used in a design and also to the parameters defined by the user in the top-level Python description. The script can be executed in command line and run all the required steps, from logic synthesis down to the generation of the bitstream file to be used to program the FPGA. The top-level design file name is also declared in the Tcl file, along with other HDL design file names generated from Python descriptions or the names of blocks used for structural descriptions from the tool's libraries. In Figure 5.3 the file hierarchy used for a design project in XST is presented.

The synthesis options that can be set by the user in a Python description are the following:

- FPGA_DEV: defines the FPGA family used for implementation.
- FSM_STYLE: controls the way state machines are implemented, using BRAMs or CLBs to store the states' encoding.
- FSM_ENCODING: defines the state encoding style used by a state machine, e.g. one

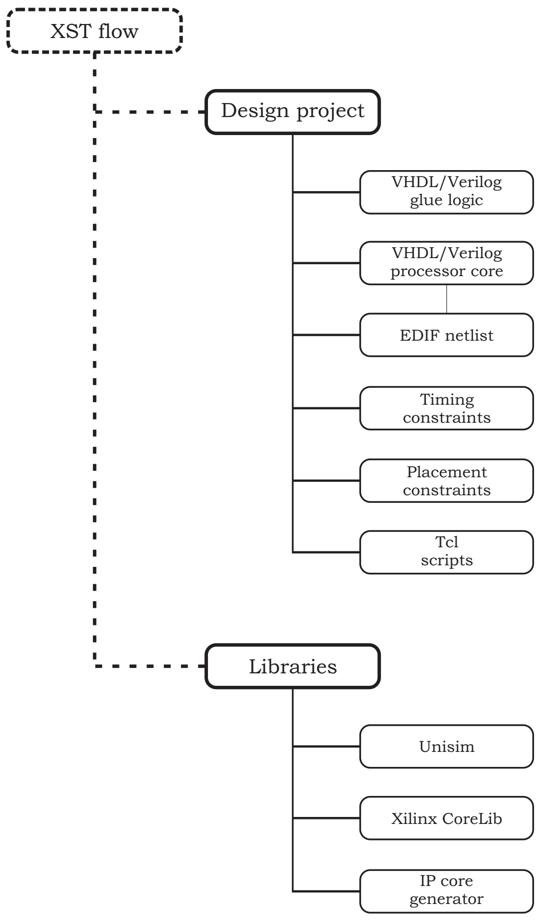


Figure 5.3: File hierarchy used in XST design project.

hot, gray, johnson etc.

- MULT_STYLE: controls the way multiplier blocks are implemented, using hardwired multiplier blocks or CLBs.
- RAM_STYLE: controls the way memory blocks are implemented, using hardwired BRAMs or CLBs.
- RESOURCE_SHARING: defines if multiple arithmetic operations will share the same logic block, if multiplication operations can be implemented in a pipelined way instead of parallel execution.

All the values of the synthesis parameters defined by the user are parsed to the Tcl script. Also SysPy generates the folders hierarchy required by XST, so that upon execution of the Tcl script this hierarchy is recognized as an already existing project, ready to be synthesized by the tools. The full set of possible values for the synthesis options can be found in Table 1.1.

According to the diagram presented in Figure 5.3, SysPy delivers a ready to be synthesized processor-centric design to the FPGA tools. Code reuse is exercised in SysPy, since digital blocks in RTL or netlist description format can be utilized by the designer and combined with custom defined logic. All the required hardware and software information is provided along with the required Tcl scripts that "glue" all hw/sw design, logic synthesis and physical implementation tools together.

Chapter 6

Image processing SoC design case

The first big design that we used to verify the processor-centric features of SysPy was an image processing system built around the AVR 8-bit microcontroller (uC) core. It is the first example that used the complete design flow, from top-level Python description down to the generation of the binary FPGA programming file. The tool was used in his example to generate all the VHDL synthesizable code for the image processing block and its connection to the uC and also insert the uC core in the design and compile the required software developed in C and assembly language.

6.1 Design features for image processing

With the design described in this chapter we proved that a full, real system could be designed using Python abstract descriptions in SysPy and implemented using an FPGA device. While custom logic can be described directly in Python and translated by the tool to VHDL, other modules are utilized as ready-to-use blocks in RTL or synthesized netlist description. As a summary, the following three basic design concepts adopted in SysPy were used and tested during the development of the image processing SoC:

(a) Python as an HDL: Capability to describe hardware components in Python that the tool automatically translates to VHDL.

(b) Modular SoC design - Components Reuse: Use of Python to build descriptions of SoCs based on hardware components that are defined in Python, but may have a Python or

VHDL or pre-synthesized netlist implementation in a library.

(c) Processor-centric SoC design: Support for IP cores of programmable processors. The tool takes as input the user's C code and implements all the steps necessary to produce a synthesizable VHDL description of the corresponding program memory file for the targeted FPGA device.

Some features of the system also triggered the development of new tool features during design. Most important are the automatic generation of Tcl scripts and the use of modules in a synthesized netlist format from the Xilinx CoreLib library. In SysPy we wanted to minimize the effort of connecting a processor core in a design. A large number of HDL files is used to describe even a small 8-bit processor core, like the AVR block. The automatic generation of a Tcl script that could be used along with the FPGA synthesis tools, was required to automate the processor instantiation in the design. We also wanted to take advantage of the large number of complex IP cores that could be automatically generated using the Xilinx Core Generator tool. The cores are provided in the form of a synthesized netlist, so their HDL description could not be parameterized. We managed to synthesize a number of useful arithmetic cores and store them in SysPy's libraries. According to user provided attributes in the top level Python description, the proper synthesized block was selected and instantiated in the design.

All the previous features were developed and tested using the implemented image processing core. We especially tested the way to connect together the processor core and the custom connected arithmetic blocks to speed up the image processing task. A serial connection was used to interface the design through an interface software developed with Matlab and trigger data transfers between a host PC and the Virtex-5 FPGA board that we had available.

6.2 AVR core and features

The processor core that we used in this design is a compatible VHDL description of the 8-bit AVR ATmega128 uC [45] from Atmel. The project for the AVR core is uploaded in the OpenCores website [76]. The core has the following basic features:

- 32x8-bit general purpose registers
- supports up to 128kb of program and 64kb of data memory
- programmable UART block
- Two 8-bit parallel ports
- Eight external interrupt sources

We selected the ATmega128 as the first processor core to use within the tool because it is a small core with sufficient program memory that would let us prove the concept of processor-centric design using SysPy. We used its UART module to communicate with a host PC and the two available GPIO ports to interface custom peripheral logic in the FPGA. GCC compilers have been ported to all the popular processor architectures, so we used the available avr-gcc C compiler [3] to develop and compile the software for our application. A block diagram of the AVR architecture is presented in Figure 6.1.

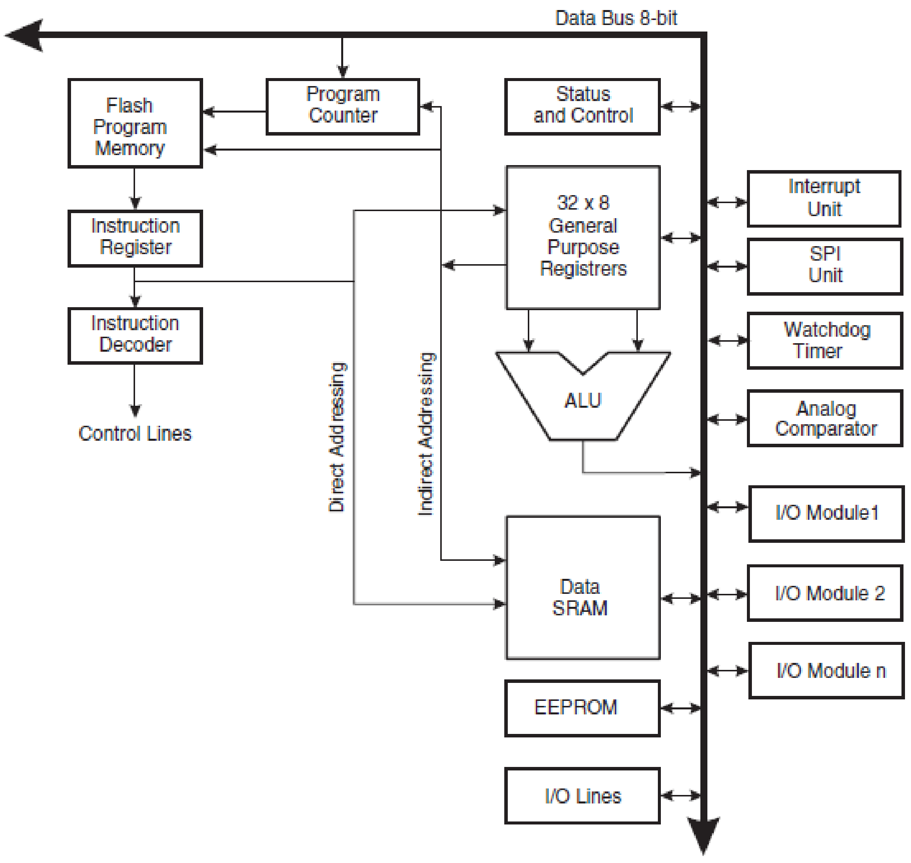


Figure 6.1: Block diagram of the AVR architecture (source: www.atmel.com).

The AVR is one of the fastest 8-bit architectures available where most instructions require just one clock cycle to be executed. The ATmega128 core was one of the few free available and well tested 8-bit processor cores, also compatible with GCC software development tools. While the core is not a fast, high end processor, it was used in SysPy to prove the supported automated flow of inserting a processor core in a design and connecting it to custom blocks. For the first processor core that we used we wanted to support at least serial connectivity with the host PC and provide data ports for interfacing custom peripheral devices. We also wanted to prove the ability to have a self contained Python description file, where the connection of a processor core along with the software to be executed are defined and the related compilation tools (C compilers, RTL synthesizers) are invoked automatically.

All memories of the uC are mapped to BRAMs, e.g. register file, data and program memory. Two GPIO ports are supported for connecting the uC to other blocks in the FPGA. The size of the implemented ROM memory was enough to hold the software that we implemented, while the size of the RAM memory was also enough to hold the images transmitted from the host PC to the FPGA. Using the two GPIO ports we implemented the data and control interface between the processor and the custom blocks created to perform the required arithmetic operation of the image processing algorithm. Any analog related blocks shown in Figure 6.1, like the analog comparator, are obviously not implemented in the IP core that we used and are part of the ATmega128 ASIC implementation.

The avr-gcc library, also used within SysPy, is a stable and well tested environment for software development for the AVR architecture. It supports a subset of the standard C library. It also provides ready-to-use code for several common tasks used in the AVR architecture related to memory and peripheral device accesses. The library also supports almost all of the available AVR devices. In general we can say that the ATmega128 core along with its software development environment was a good candidate to initially test and debug the processor-centric design flow that we implemented in SysPy.

6.3 Image processing SoC design

To demonstrate the capabilities of our tool, we have used SysPy to design a processor-centric SoC system that applies Sobel edge detection [37] to grayscale images. The Sobel algorithm detects vertical and horizontal edges of an image. The result of the filtering process is two images representing the gradient of the original image in the horizontal and vertical direction. By calculating the Euclidean distance between corresponding pixels in these two images, a third image is produced representing the magnitude of the gradient. Finding the edge of various objects in an image can reveal important object properties which later can be used for applying more complex image processing algorithms, like object detection, image transformation/editing etc.

6.3.1 Sobel's algorithm

The Sobel algorithm uses two convolution kernels to identify vertical and horizontal contrast changes on an image.

$$N(x,y) = \sum_{k=-1}^{1} \sum_{j=-1}^{1} K(j,k)p(x-j,y-k) \tag{6.1}$$

Equation 6.1 describes the convolution between a K group of pixels of the image and one of the two kernels used by the Sobel operator. Two new images are generated after applying the two convolution kernels showing the gradient approximation for horizontal and vertical contrast changes respectively. Equations 6.2 and 6.3 describe the convolution between the two 3x3 kernels and the original image.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * p \tag{6.2}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * p \tag{6.3}$$

By calculating the Euclidean distance between corresponding pixels in these two images, a third image is produced representing the magnitude of the gradient. The G_x and G_y magnitude vectors are the x and y components of the combined G vector. The magnitude and direction of the combined G vector can be calculated using Equations 6.4 and 6.5 respectively.

$$G = \sqrt{G_x^2 + G_y^2} \tag{6.4}$$

$$\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{6.5}$$

The magnitude of vector G represents the final processed image where the detected edges information is included. By combining the G_x and the G_y using the Euclidean distance computation (Equation 6.4) vectors it becomes easy for the algorithm to track steep line curves of an image and also detect edges among many complex objects that exist in an image. Other formulas can be used to calculate the magnitude vectors to avoid the square root computation in the Euclidean distance formula like the absolute squared distance or the Manhattan distance.

6.3.2 Using SysPy to glue the AVR uC and the custom peripherals

In Figure 6.2 we show the type of components involved and their connections. The SoC incorporates the AVR processor IP core, a special purpose processor which implements Sobel's operator and an arithmetic component from the CoreLib library used to implement the square root function as needed for the Euclidean distance calculation. Sobel's algorithm apply two, horizontal and vertical, 3x3 kernels on the original image. So the image has to be partitioned into blocks of 3x3 pixels every time one of the kernels is to be applied. A 64x64 image is transmitted through a serial connection from a PC to the AVR controller and stored in its data memory. This FSM applies the two Sobel kernels to the received block and two pixels are produced. Their values are squared and then added. The result of the addition is sent to the CoreLib component that calculates its square root by applying the COordinate Rotation DIgital Computer (CORDIC) algorithm [91], [92]. The result is sent back to the AVR and then returned to the PC through the serial connection.

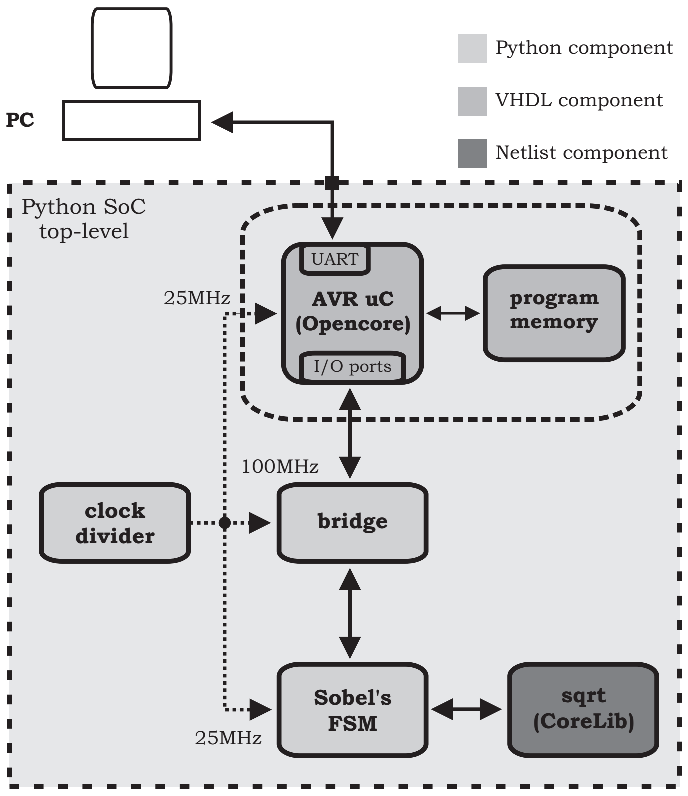


Figure 6.2: Diagram of the Sobel edge detection SoC. The shading indicates the type of each component.

For the computation of nontrivial algebraic functions, like square root computation, trigonometric functions etc. the Taylor series can be used to approximate the desired result. The required function is then calculated as a series of multiply and add operations. The CORDIC algorithm provides a more efficient implementation for solving algebraic functions which also favors hardware implementation. The algorithm is used in many applications like calculators, adaptive filters, FFT blocks, etc. The first implementation of the CORDIC algorithm in FPGAs was introduced by Meyer-Base [67], realizing a multiplier free CORDIC block for fast and accurate computation of trigonometric functions.

In our design the CORDIC block from the Xilinx CoreLib library [48] has been used to implement the required square root computation of the Sobel algorithm. As presented in In Figure 6.2 the CORDIC block along with an FSM to handle data flow forms a Sobel accelerator block attached through a data bridge to the processor's I/O ports. Sobel's algorithm functions have been partitioned to the AVR which handles all memory management and data manipulation and to the Sobel accelerator, along with the *sqrt* arithmetic component which handles all the numerical calculations. The two processors (AVR and Sobel) communicate through a bridge that manages the interaction and data exchange between them, so that alternative processor implementations can be plugged in and out as long as they respect the defined protocol of interactions. The Sobel accelerator processes data faster than the AVR controller, so the bridge acts as a buffer and also produces all the control signals informing the two components when they can send or receive data. The Sobel accelerator, the bridge and a clock divider (needed to produce all the clock signals for the system) have been described in Python. The bridge is a parameterized component, since its data buses can be easily resized with the use of generic parameters declared in its Python description. The *sqrt* CORDIC component is a pre-synthesized EDIF netlist taken from the CoreLib library. The AVR processor third party soft core is provided in VHDL. The VHDL entity for its program memory is produced by SysPy automatically based on the provided user C code.

In Code Examples 6.1, 6.2 and 6.3 the top-level description of the image processing SoC is presented. In Code Example 6.1 in line 10 the AVR core is instantiated and connected to the rest of the design. `PORTE` is used as a control port to exchange control signals with the data bridge, while `PORTA` and `PORTB` are used as data ports. In Code Example 6.2 in line 2 the Sobel accelerator is instantiated, interfacing with the rest of the design using the appropriate reset, clock and control logic and the required data I/O ports. The accelerator block contains the FSM block and the CORDIC block for the square root computation. The data bridge is instantiated in line 13, where an argument is passed to the related function (`struct_bridge`) defining the width of its data ports (`n = 8`). A clock divider is instantiated in line 29 to generate the 25MHz clock signal (used to clock the AVR and the Sobel accelerator block). In line 35 design attributes like the FSM implementation style, the FPGA device type and a C file (`sobel.c`) representing the software executed by the processor, are defined in the `attributes[]` dictionary. All the I/O signals are defined in line 39-41. In Code Example 6.3 all the internal signals are declared in lines 2-13, while the `toVHDL()` function is called to generate the VHDL top-level description.

During the translation of the Python to VHDL description, the avr-gcc compiler is automatically invoked to compile the available C software, while SysPy initializes AVR's program memory with the generated hexadecimal code. The bridge and the Sobel accelerator blocks are described in separate Python modules and they are translated to VHDL along with the top-level description. After translation a new directory is created containing all the required VHDL and software files ready to be used by FPGA synthesis tools for implementation. All the Python description files as well as the resulting VHDL files produced by SysPy and used for the FPGA implementation can be found in [88].

Code Example 6.1: Top-level Python description of the image processing SoC (part 1).

```
from inspect import *
import SysPy_ver._toVHDL

def sobel_wrapper():
  gn = '0'
  gnb = others('0')
  rst_int = ~ rst_buf

  # AVR core instantiation
  def struct_avr_core():
    nrst = rst_int
    clk = clk_div_25MHz_int
    porta = porta_avr
    portb = portb_avr
    portc = portc_avr
    portd = portd_avr
    porte[0] = porte_avr[0]
    porte[1] = porte_avr[1]
    porte[3] = pine_avr[1]
    porte[2] = pine_avr[0]
    porte[7] = porte_avr_unus[7]
    porte[6] = porte_avr_unus[6]
    porte[5] = porte_avr_unus[5]
    porte[4] = porte_avr_unus[4]
    ddrareg_out = ddrareg_out_int
    ddrbreg_out = ddrbreg_out_int
    ddrcreg_out = ddrcreg_out_int
    ddrdreg_out = ddrdreg_out_int
    ddrereg_out = ddrereg_out_int
    rxd = rxd_buf
    txd = txd_buf
    INTx = gnb
    TMS = gn
    TCK = gn
    TDI = gn
    TDO = "open"
    TRSTn = rst_int
    man_rst = rst_int
```

Code Example 6.2: Top-level Python description of the image processing SoC (part 2).

```
# Sobel accelerator instantiation
def struct_sobel():
  rst = rst_buf
  clk = clk_div_25MHz_int
  t_cts = t_cts_int
  t_dpr = t_dpr_int
  t_write = t_write_int
  t_read = t_read_int
  data_in = t_data_out_int
  data_out = t_data_in_int

# Data bridge instantiation with 8-bit I/O
def struct_bridge(n = 8):
  rst = rst_buf
  clk = clk_buf
  h_write = porte_int[0]
  t_write = t_write_int
  h_read = porte_int[1]
  t_read = t_read_int
  h_cts = pine_int[0]
  t_cts = t_cts_int
  h_dpr = pine_int[1]
  t_dpr = t_dpr_int
  h_data_in = porta_int
  t_data_in = t_data_in_int
  h_data_out = portb_int
  t_data_out = t_data_out_int

def struct_clk_div():
  clk = clk_buf
  rst = rst_buf
  clk_div_25MHz = clk_div_25MHz_int

# Design atrributes
attributes = {"sign": '+', "FSM_STYLE": "lut", "MUX_EXTRACT": "yes",
              "FPGA_DEV": "Virtex5", "PROC_SW": ["sobel.c"]}

# I/O signals
iosigs0 = {'D': 'i', 'T': 'b', 'L': 1, 'N': ["clk", "rst", "rxd"]}
iosigs1 = {'D': 'o', 'T': 'b', 'L': 1, 'N': "txd"}
iosigs2 = {'D': 'o', 'T': 'b', 'L': [7, 0], 'N': "portc"}
```

Code Example 6.3: Top-level Python description of the image processing SoC (part 3).

```
# Internal signals
intrsigs0 = {'D': 'intr', 'T': 'b', 'L': 1, 'N': ["clk_buf", "clk_div_25MHz_int",
             "clk_div_50MHz_int", "clk_div_100Hz_int", "rst_int", "gn"]}
intrsigs1 = {'D': 'intr', 'T': 'b', 'L': 1, 'N': ["t_cts_int", "t_dpr_int",
             "t_write_int", "t_read_int", "rst_buf", "txd_buf", "rxd_buf"]}
intrsigs2 = {'D': 'intr', 'T': 'b', 'L': [7, 0], 'N': ["porta_int", "portb_int",
             "portc_int", "portd_int", "porta_avr", "portb_avr", "portc_avr"]}
intrsigs3 = {'D': 'intr', 'T': 'b', 'L': [7, 0], 'N': ["portd_avr", "porte_avr_unus",
             "ddrareg_out_int", "ddrbreg_out_int", "ddrcreg_out_int"]}
intrsigs4 = {'D': 'intr', 'T': 'b', 'L': [7, 0], 'N': ["ddrdreg_out_int",
             "ddrereg_out_int","t_data_in_int", "t_data_out_int", "portc_buf", "gnb"]}
intrsigs5 = {'D': 'intr', 'T': 'b', 'L': [1, 0], 'N': ["porte_int", "pine_int",
             "porte_avr", "pine_avr"]}

# Calling the "to_VHDL()" function to generate VHDL code
_toVHDL.toVHDL("sobel_wrapper", attributes, generics, iosigs0, iosigs1, iosigs2,
               intrsigs0, intrsigs1, intrsigs2, intrsigs3, intrsigs4, intrsigs5)
```

6.4 Performance and implementation results

For the SoC implementation we used the ML509 FPGA board from Digilent [25] equipped with the medium size Virtex-5 XC5VLX110T-1 FPGA device. For this initial SoC implementation using our tool the only requirements for the board were to a) have an FPGA device equipped with DSP48 [49] blocks utilized by the CORDIC block and b) to have a serial connection with a host PC, used to transmit JPEG images to the FPGA.

Correct functionality of the SoC, in terms of proper execution of the Sobel algorithm, has been verified using the Modelsim [66] and the Xilinx ISE (ISim) [50] RTL functional simulators before the FPGA implementation. After implementation we validated the design in silicon by processing several images and comparing the results that we got in terms of proper edge detection. In Figure 6.3 we present two processed 64x64 pixel processed images, where the detected edges are the combination of the two applied kernels of the Sobel algorithm. The initial image was transmitted to the FPGA board through a serial interface developed in Matlab and executed on the host PC and the processed images are also transmitted back to the PC and stored.

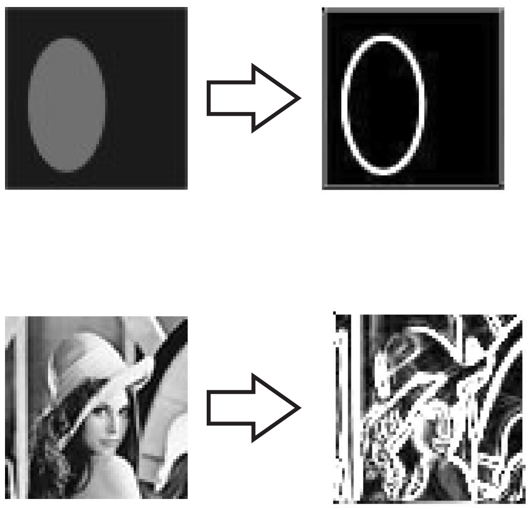


Figure 6.3: Processed 64x64 pixel images by the implemented SoC.

In Table 6.1 we summarize the synthesis results for each component used for the implementation of Sobel's algorithm. Sobel's FSM uses the *sqrt* component for the computation of the Sobel gradient's magnitude. The *sqrt* utilizes eight DSP48 slices for the multiplication operations. BRAMs are utilized only by the AVR processor for its data and program memory implementation. Each CLB in the Virtex-5 family contains eight 6-input LUTs, while the BRAMs' block size is 36Kbit. According to place and route results, the system can be clocked as high as 190MHz and it utilizes 367 CLBs, covering 6% of the available CLBs and 32% of the available BRAMs of the device. With the current clock frequency setting we measured that ten 64x64 images/second can be processed by the system . Timing measurements have been performed in silicon, after system's implementation, using the provided by Xilinx ChipScope Pro [52] logic analyzer. The analyzer is configured as an IP core in Xilinx ISE and connected to user defined signals in the design. After capturing the desired signal values, data are transmitted to the host PC using the Joint Test Action Group (JTAG) connection, used for FPGA system programming and debugging.

With the image processing SoC design we verified correct functionality of SysPy's basic

Components	**CLBs**	**BRAMs**	**DSP48**
Sobel accelerator + sqrt	47	0	8
AVR Processor soft core	267	48	0
Bridge	3	0	0
Clock divider	5	0	0

Table 6.1: Synthesis results for the Sobel system. Utilized resources for the Virtex-5 LX110T device are presented (CLB: two slices, Slice: four 6- input LUTs, BRAM: 36Kb, DSP48: 25x18-bit).

features. During system design we verified proper: a) Python to VHDL translation, b) use and insertion of processor soft cores and c) use of digital blocks in different description format (Python, VHDL, netlist) and d) use of software developing tools. After successful implementation of this design we moved on with the development of SysPy and planned the use of 32-bit processor cores to test and use the tool with more elaborate applications. We also specified the development of the hw/sw co-simulation mechanism to ease verification of the designed SoCs.

Chapter 7

Biomolecular interaction networks simulation SoC design case

Based on the main tool features presented on Chapter 6 we extended SysPy features to support the Leon3 32-bit processor soft core and its software development environment. By using Leon3 we were able to implement a fast communication interface between the FPGA board and the connected host PC and also have access to a large memory space on the board. To support SoC design using the new processor core we also developed a new interface software to ease data transfers between the FPGA board and the PC. To demonstrate SysPy's improved design flow and functionalities we designed and implemented a processor-centric embedded SoC for computational systems biology. The SoC combines high performance computing features of FPGAs along with the flexibility of a programmable processor core to simulate efficiently the stochastic behavior of large size biomolecular reaction networks.

7.1 Selection of a 32-bit processor soft core

Moving forward with the development of SysPy and using the results we got from the development of the image processing SoC presented in Chapter 6, we searched for new 32-bit processor cores that we could add to the tool. As a minimum set of features for the new cores we defined that the processor must be able to establish a fast connection (the transfer rate should be in the order of Mbps) with a host PC and also have access to a large memory space (in the order of hundreds of MB). Also general I/O ports should be available to interface the processor with custom processing blocks in the FPGA. To meet these required set of communication and memory features, the cores should support at least an 100Mbps Ethernet controller and an SDRAM DDR memory controller. The physical layer for these two controllers was already present on the ML509 board that we were using and only the higher protocol layers had to be implemented by the processor peripherals. The required peripheral controllers for the Ethernet and the SDRAM DDR memory connection were available for both the Leon3 and the OpenRISC core, so we decided to include them both to the SysPy processor repository.

7.1.1 OpenRISC core and features

Initial trials for the use of a 32-bit processor core in SysPy have been done using the OpenRISC 32-bit processor soft core [56] and more specifically using the Minimal OpenRISC System on Chip (MinSoC) [29] customized implementation of the processor, while the processor model is provided as a Verilog description. The processor supports the Wishbone [74] protocol which is a general purpose communication interface for data exchange between IP core modules. Several IP cores that exist in the OpenCores repository are compatible with the Wishbone specification and can be used as peripheral devices of OpenRISC. The basic features of the processor are summarized in the following list:

- All major characteristics of the core can be set by the user.
- High performance of 300 Dhrystone 2.1 MIPS at 300 MHz using 0.18u process.
- WISHBONE SoC Interconnection Rev. B compliant interface.
- Support of a CPU/DSP central block.
- Debug unit and development interface.

The block diagram of the OpenRISC architecture is presented in FIgure 7.1.

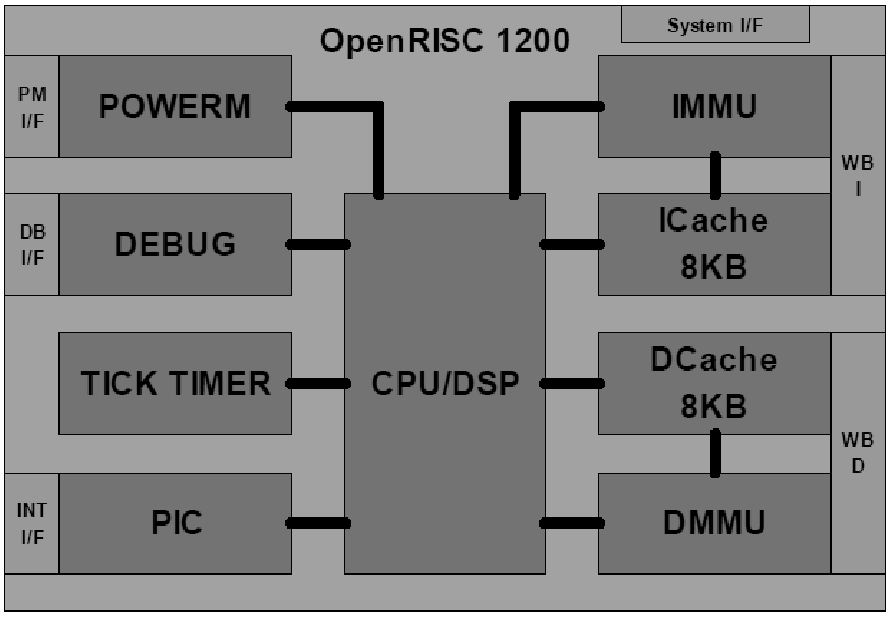


Figure 7.1: OpenRISC1200 block diagram (source: www.opencores.org).

Most integer instructions in OpenRISC can execute in one cycle, while a Multiply Accumulate (MAC) unit in the CPU executes DSP MAC operations. The implemented debug unit also assists software developers to debug in real time code executed by the processor. The MinSoC is a ready-to-use SoC that implements the OpenRISC1200 connected with various peripheral devices, like memory controllers, Ethernet, SPI and UART controller and JTAG debug interface. The SoC also included specific code description for the memory, the JTAG and clock management DCM blocks, for Xilinx and Altera FPGA devices, for easy implementation using FPGA tools. The data and program memory is also easy resizable using generic parameters in the HDL description. An overview of the MinSoC configuration is presented in Figure 7.2.

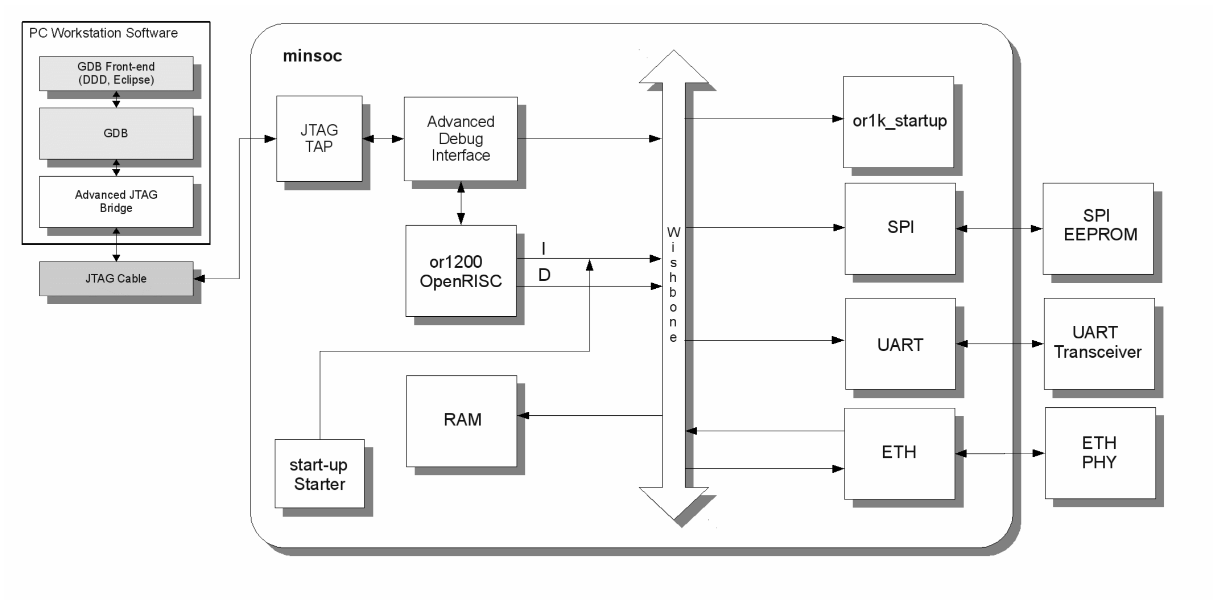


Figure 7.2: MinSoC block diagram (source: www.opencores.org).

Several FPGA boards are supported for implementing the MinSoC design and our team has been the first to port the SoC to the ML509 Digilent board, as mentioned in the MinSoC project related webpage [29] in the OpenCores website. Also Python scripts that have been developed under SysPy for program memory initialization of the MinSoC core are publicly available and have been included to the projects' repository [75].

Although MinSoC's architecture and performance is almost equivalent to the Leon3 core, while testing the core on our ML509 board we did not manage to get the Ethernet connection working between the processor in the FPGA and the host PC. While we managed to get the memory DDR controller working, the lack of network connectivity was a major step back on the use of MinSoC in SysPy. Although the MinSoC core was integrated and added to SysPy's component libraries, it was never used during the design flow of a complex system.

7.1.2 Leon3 core and features

Leon3 [11] is a popular, open source, 32-bit processor provided as a synthesizable VHDL description by Aeroflex Gaisler. The core is compatible with the SPARC V8 architecture and is highly configurable and distributed along with configuration, debugging and simulation software. The Leon3 core is also part of the GRLIB IP library [33]. The library contains a large number of configurable IP core peripheral devices that can be connected to Leon. The library has many data and memory controller devices which can be attached to Leon using the supported Advanced Microcontroller Bus Architecture (AMBA) [13] peripheral interface bus. The processor has been ported in all major FPGA platforms (Xilinx and Altera) and also can be synthesized using ASIC synthesis tools (Cadence, Synopsys). Typical synthesis results show that the core occupies 25k-30k gates when implemented.

The main features of the Leon3 processor core are the following:

- Advanced 7-stage pipeline
- Hardware multiply, divide and MAC units
- Separate instruction and data cache (Harvard architecture)
- Up to 125 MHz in FPGA and 400 MHz on 0.13 um ASIC technologies
- AMBA-2.0 AHB bus interface
- Large range of software tools: compilers, kernels, simulators and debug monitors

The structure of the processor along with the main peripheral units and features is presented in Figure 7.3.

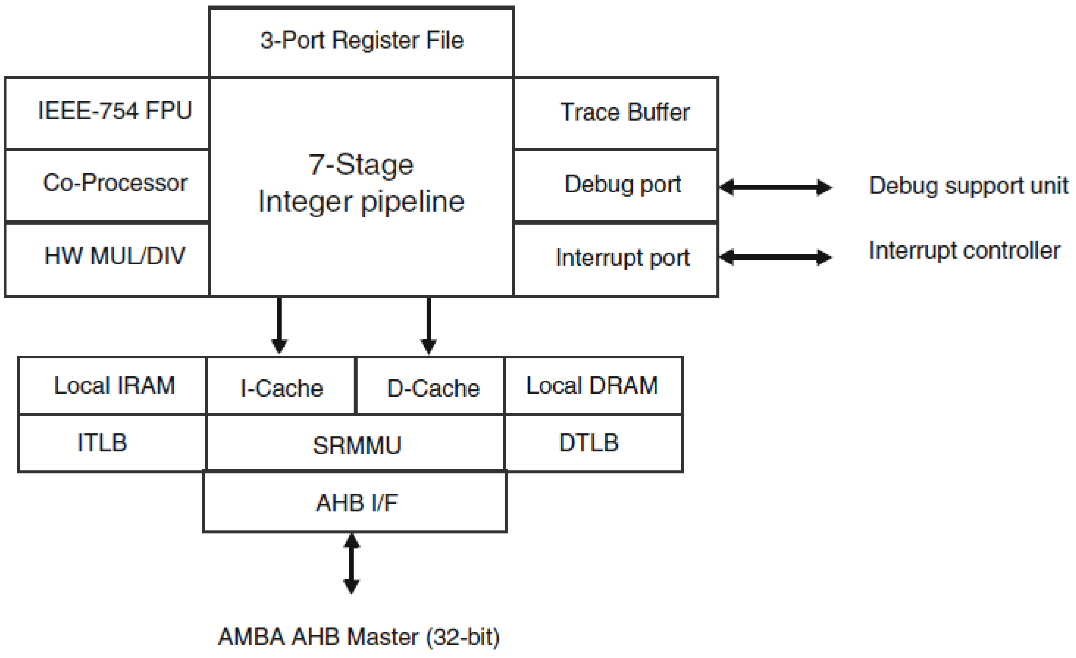


Figure 7.3: Structure of the Leon3 processor (source: www.gaisler.com).

All the tests with Leon have been accomplished using the integrated debug unit, connected as a peripheral device to the processor. Using the GRMON debug tool [10], we were able to use the provided debug interface, connect to the processor via Ethernet connection and download the required software. Software debugging was also performed on silicon, since using GRMON we were able to read/write to the entire memory space. Leon3 is highly configurable using the provided GUI interface or by modifying manually the VHDL description of the processor. A functional simulator, called Tsim, exists that can be used for software and hardware debugging, before FPGA implementation. Using GRMON we could also inspect the processor configuration and mapping of peripheral devices in silicon, to check if it matches the way the core was configured.

The sparc-elf-gcc was used to compile the C software developed for Leon and the generated binary file was used to program the processor. In the case of Leon, the code is downloaded directly to the connected 256MB SDRAM DDR2 memory available on the ML509 board, which is used for mapping both the program and the data memory of the system. The Ethernet PHY chip and the serial connection are also utilized on the FPGA board by Leon. A large number of other peripheral devices, e.g. USB controllers, audio and video drivers, hard disk controllers etc. also exist for Leon, that are compatible with the processor's AMBA interface. The data transmission of the Ethernet connection was 100Mbps, while a commercial version of Leon3 also supports Gigabit Ethernet connection. The Leon3 core was a good candidate for the 32-bit core that we wanted to integrate to SysPy, since all the I/O connectivity (Ethernet, GPIO ports) and memory requirements (SDRAM controller) were met according to the core's supported features. The Leon3 core has been tested, using SysPy to configure it, on the ML509 board as well as on the older Virtex-II Pro board [24] from Digilent which we had at our disposal. Depending on the selection of the proper FPGA device (`FPGA_DEV` design attribute, see Section 4.1.1 and Table 1.1), proper constraints files are selected by the tool and used during logic synthesis.

7.2 BioModel files

Advanced high performance computational techniques have been used the last decade in several scientific domains to ease and accelerate simulation of complex physical phenomena.

Computational and systems biology is a rather new scientific field that takes advantage of computing techniques to describe and simulate complex biological phenomena and extract knowledge from biological databases. Systems biology studies the dynamics of cellular processes, rather than the characteristics of their isolated parts and is an emerging field that benefits from advanced simulation techniques. By simulating the chemical kinetics of the different molecular species interacting in a cell, or in a population of cells, emergent properties of a biological system can be studied *in silico* (i.e. with the use of computers). Stochastic simulations of biochemical reaction networks, called BioModels [57], [17] can be performed to study the properties of these biological systems. A large number of BioModels, can be found online in the BioModel's database [27], [57]. The database remains the largest online repository of dynamic models of biological processes and most models are validated and also linked to external publications and original scientific data which used to create the model.

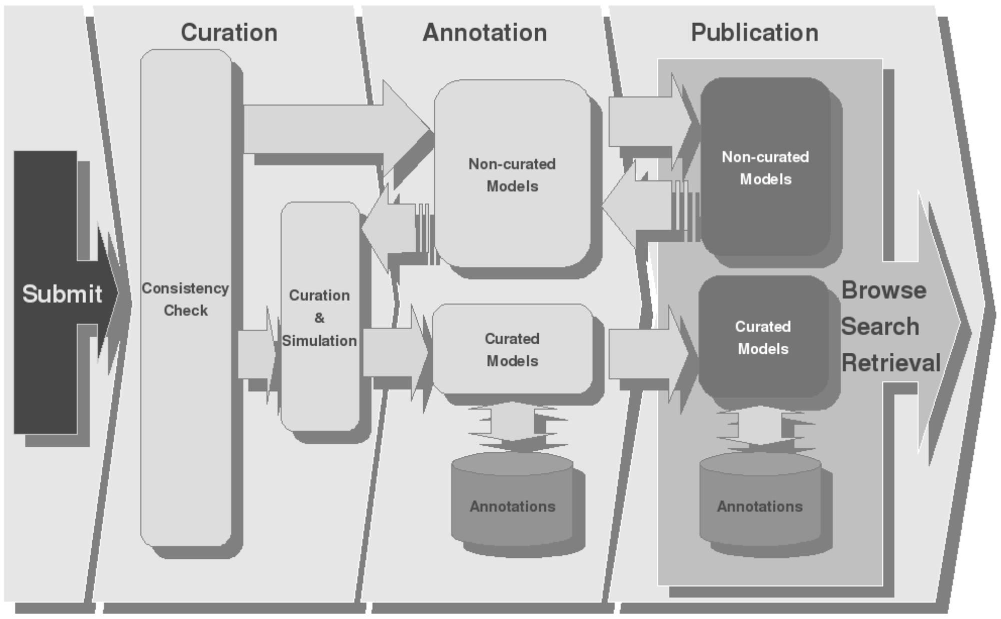


Figure 7.4: Submission flow to the BioModels database (source: www.ebi.ac.uk/biomodels/).

All models in the database are compliant with the Minimum Information Required In The Annotation of Models (MIRIAM) standard [72]. Files in the database are divided to curated and to non-curated models, where curated models have been checked and validated that they have the expected behavior during simulation, as it is described by the creators of the model. Models can be stored in the database in SBML format [42] and can be later downloaded in other formats too, like BioPAX, Octave, SciLab and PDF. The flow used for submission and validation of models to the database is presented in Figure 7.4.

7.3 Gillespie's stochastic simulation algorithm

In order to apply computational techniques in system biology, models should be used that describe the chemical dynamics of a cell or of a set of cells. These chemical reactions can be expressed using coupled Ordinary Differential Equations (ODE). Using ODEs to describe a chemical reaction is not always the best option, especially if there is no prior knowledge about the initial conditions of the molecular species in the system [69]. Then it becomes very difficult to track the occurrence time of each reaction. Also it is very difficult to detect stochastic behavior of a small number of reactant species in a chemical reaction. Instead of ODEs, a Stochastic Chemical Kinetic (SCK) model can be used to describe a chemical process. A specific rate constant, c_j, is assigned to every reaction in the system. This constant expresses the probability that a combination of reactant molecules of R_j interact in the next time interval $[t, t + dt]$. Multiplying c_j by the number of possible combinations of reactant molecules for R_j in state $\mathbf{X}$ yields the propensity function, $\alpha_j(\mathbf{X})$, of reaction channel R_j.

A jump Markov process [36] is used within SCK models to describe evolution in time of a reacting system. In such a process the next state of the model is only dependent on the present state. An equation can be used to describe the evolution of reactions in time, using a dt time step. Using a small dt step size to advance time and solve this equation, where on each time step: a) the system performs a reactions and its state is updated or b) nothing happens and the system state remains unchanged. If the step is too small, then in most cases there will be no reaction in the system. Gillespie proposed a new method to solve these kind of equations, which is a Stochastic Simulation Algorithm (SSA) and called the Direct Method (DM). This method improves the chemical reactions simulation's efficiency by performing only the necessary time steps.

D. T. Gillespie also proposed another method known as the First Reaction Method (FRM) [35]. In this approach every reaction is described as a “reaction channel” R_j and a putative next reaction time is calculated for every channel. The algorithm detects the reaction channel with the smallest reaction time and this reaction is “fired”. According to these statements, Gillespie described the FRM algorithm as follows:

1. Initialize $t = t_0$ and $\mathbf{X} = \mathbf{X}_0$.
2. For every reaction channel R_j evaluate propensity function $\alpha_j(\mathbf{X})$.
3. In addition determine the putative time τ_j for each reaction channel R_j:
$$\tau_j = \frac{1}{\alpha_j(\mathbf{x})} \ln\left(\frac{1}{r_j}\right) \tag{7.1}$$
where r_j is a unit uniform random number.
4. Let R_μ be the winner reaction channel whose τ_j is the smallest.
5. Determine the new state after firing reaction R_μ: $t' := t + \tau_\mu$ and $\mathbf{X} := \mathbf{X} + \mathbf{v}_\mu$, where $\mathbf{v}_\mu$ is the stoichiometry change information.
6. If t' is greater than the desired simulation time T_{sim} then halt.
7. Record $(\mathbf{X}, t)$ and go to step 2 to start a new reactions cycle.

The main idea of the design of our SoC was to accept as an input BioModel files and simulate the chemical reactions of the described system using the FRM algorithm. We designed our SoC in a way that implements the exact Gillespie FRM SSA which simulates all reaction events and not anyone of its many approximations (such as tau leaping [80] etc.). Approximations are used during simulation to avoid high computational cost, but on the other hand they reduce results' and calculations' accuracy. Only hardware based SSA implementations, such as our SoC, can be used to apply parallel computation techniques and in this way avoid approximations introduced by the majority of software implementations.

By choosing to design a SoC using SysPy to speed up the simulation of biochemical reaction networks, we wanted to explore the capabilities of SysPy and use Python to extend and improve the features of our tools in order to:

- automatically parameterize and instantiate the custom core, implementing the FRM algorithm.
- use Python text processing features to parse BioModel files content and use it to instantiate memory blocks in the SoC.
- use Python to develop a Hardware Abstraction Layer (HAL) for data exchange via Ethernet connection between the SoC and the host PC.

Using this biomolecular SoC as a design example for SysPy we got feedback and new ways to improve the tool. We also created a useful SoC which engineers from a different scientific field and with no knowledge in digital hardware design could easily modify, using the supported design scripts, and use to perform their experimental studies to analyze fast a number of different BioModels.

7.4 Biomolecular network SoC features

7.4.1 Scalable IP core for stochastic simulation

An IP core (SSA core) implementing the FRM algorithm was already in place and designed in our group before the availability of 32-bit processor cores in SysPy. The development of the SSA core was done in parallel to the SysPy project and when it reached the desired maturity as a standalone IP core we decided to include it to the SysPy component library. First version of the SSA core was presented in [40]. The IP core implemented N Processing Elements (PEs) working in parallel, for simulating in reasonable amount of time large-scale biochemical reacting networks. The IP core was described in VHDL and it is parameterized by the number of PEs (N) and the number of reactions in the BioModel (m). Other parameters of the SSA core are also tunable in order to implement in an efficient way a parallelized version of Gillespie's FRM-SSA.

In particular, the DM requires summing up the propensity functions of the individual reactions during the last step of a reaction cycle in order to find the next reaction to fire, R_μ. The FRM-SSA allows parallel execution of the reaction channels R_j, where the inner for loop in steps 2 and 3 of the algorithms description in Section 7.3 computes the reaction propensities and putative times (τ_j. The processing cost in the SoC is distribute among the available PEs of the design. If the FRM-SSA core has N PEs, each PE will compute m/N propensities and putative reaction times (τ_j) and their minimum in a pipelined fashion. A simple binary tree of log_2N comparators which is part of the Minimum Time Unit (MTU) block in the FRM-SSA design finds the global minimum time τ_μ of the next reaction to "fire", R_μ.

Another option was to use the NRM Gibson and Bruck's algorithm [34] which is faster than the FRM-SSA implementation but much more difficult to implement and parallelize it in hardware since it requires maintaining a shared data memory structure. The NRM-SSA has been implemented in hardware using FPGA devices [95], [96] using an interconnection network of MUXes. Implementation results reported in [96] prove that it is difficult to implement the NRM-SSA using more than $N = 4$ PEs and this was one of the main reasons for choosing to implement the FRM-SSA instead, since we were planning to scale our design, using large FPGA devices, and implement up to $N = 32$ parallel PEs.

In this work we focus on how we can harness SysPy's design capabilities to build a flexible multi-processor SoC around the Leon3 processor utilizing an improved version of this SSA IP core component as an accelerator for systems biology simulations. Proper instantiation of the SSA core requires many, BioModel dependent, parameters. We present how using SysPy's design flow we can automatically parse a BioModel's XML file in Systems Biology Markup Language (SBML) format [42], extract automatically these parameters and construct the memory structures and the top-level specification necessary for synthesizing the SoC's FPGA implementation in an effortless way.

7.4.1.1 FRM SoC architecture

Comparing the design of the FRM-SSA design presented in [40] to the later IP core design presented in [61], we improved the core's functionality so that it:

- Provides a high-level mechanism for configuring the SoC's parameters based on the BioModel's parameters.
- Provides an easy to follow design flow for translating the SoC's top-level description into a hardware/software FPGA implementation for Gillespie's FRM SSA.

- Provides an easy-to-use interface allowing a user application running on the host PC to harness the computational power of the system biology specific SoC residing in the FPGA.
- Facilitates the rapid prototyping of SoCs to efficiently simulate any large-size biomolecular reactions network using an appropriate size FPGA.

In [61] we presented a flexible multi-processor SoC around the Leon3 processor using the improved version of the FRM-SSA IP core. In this new design we proved how using SysPy we can automatically parse a BioModel file in SBML format, which uses XML syntax to define a biochemical reaction network. All the model's important parameters are extracted and used to automatically construct the memory structures and the top-level specification necessary for synthesizing the SoC's FPGA design. With this design approach the SoC can be used to process any BioModel of interest (captured in SBML) without any user intervention or required expertise in FPGA design.

The supported list of generic parameters of the FRM-SSA IP core are presented in Table 7.1. Three different versions of the SSA core implementing the FRM algorithm have been designed, using one (FRM1X), two (FRM2X) or four PEs (FRM4X) operating in parallel (N=1 or 2 or 4). The number of reactions m and the number of species n are automatically parsed from the BioModel SBML file, while the rest of the parameters mentioned in Table 7.1 are declared by the user. Each instantiated PE requires a different random seed which is used by a random number generator to provide the unit random number r_j used in step 3 of the FRM algorithm, presented in Section 7.3. Two different modes of operation are supported in the design: each PE can either a) perform m/N reactions of a BioModel in a pipelined fashion (in SSIP: Single Simulation In Parallel mode) or b) perform all m reactions (in MSIP: Multiple Simulations In Parallel mode), while all N PEs work in parallel.

Parameter	**Name**	**Range**
m	Number of reactions	$2^e, e \in [0, 12]$
n	Number of species	$2^e, e \in [0, 12]$
q	Number of reactants (reaction order)	$[1 - 3]$
N_{rep}	Number of simulation repetitions	
$RNGseed$	Initial seed for the number generator	$[0 - 255]$
K	Mode of operation	$[0 = SSIP, 1 = MSIP]$
T_{sim}	Simulation time in seconds	

Table 7.1: Generic parameters of the FRM-SSA core.

A top-level architectural diagram of the FRM-SSA core is presented in Figure 7.5. The processing procedure is handled by the Control Unit (CU) which is the main state machine of the design. A number of memory tables hold information parameters about the reaction network, which are parsed from the SBML file. The memory Tables are the Reactions Table (RT), the Stoichiometry Table (VT), the Species Table (ST) which holds the species counts and the Flags Table (FT) which stores binary flags controlling the operation of the PE's propensity function computation sub-module according to the order of the reaction channel under evaluation. A Memory Management Unit (MMU) is required to address and combine information included in the memory tables.

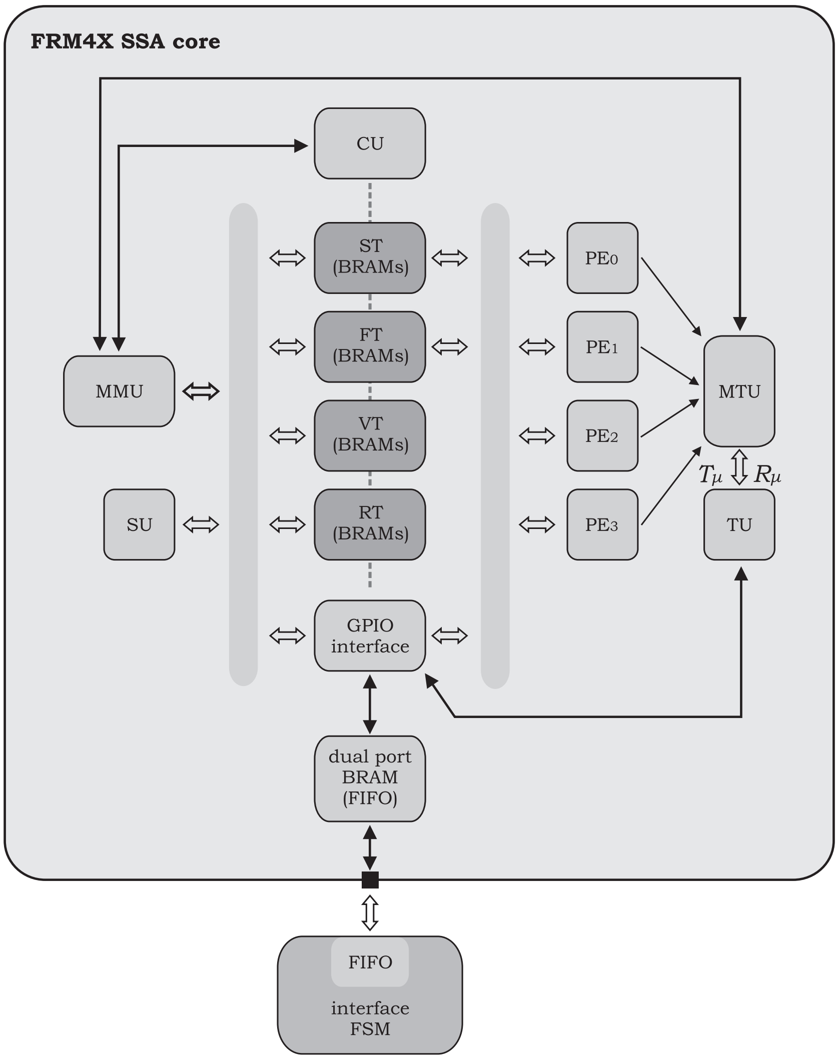


Figure 7.5: FRM-SSA core architecture with four Processing Elements (FRM4X).

A tree of comparators is used to determine the reaction channel R_μ with the minimum reaction time τ_μ, according to step 4 of the FRM algorithm. The comparator is included in the Minimum Time Unit (MTU) and is divided into two stages, where the first stage finds the minimum of the putative times for the reaction channels that are processed by each PE, and the second stage compares these local minima to find the overall minimum τ_μ time of the reaction to fire R_μ.

The steps of the FRM algorithm in section 7.3 can be mapped to individual units on the FRM-SSA IP core schematic in Figure 7.5. The PEs compute the propensity function $\alpha_j(\mathbf{X})$ and determine the putative time τ_j time of each reaction channel (steps 2 and 3). The reaction channel R_μ with the smallest reaction time (step 4) is determined by the MTU,

SSA core	**Reaction Cycle time (us)**	**MReaction Cycles/sec.**
FRM1X	1.356	0.737
FRM2X	0.93	1.075
FRM4X	0.73	1.37

Table 7.2: Throughput of the SSA cores at a clock frequency of 160MHz for a network with $m = 136$ reactions and $n = 93$ molecular species.

while the TU calculates the new simulation time (step 5). The number of new species is calculated by the SU (step 5) and the result is stored in the ST memory block.

The performance of the FRM-SSA IP core is measured in terms of Reaction Cycles executed per second. One reaction cycle is required for the core to process all m reaction channels of a BioModel and in the end the one with the minimum reaction time is selected (steps 2-5). Processed information is presented at the end of a reaction cycle in the form of a reaction packet containing 10x32-bit values. Using IEEE-754 single precision floating point format the first eight values of the reaction packet represent the number of the species involved in the reaction (three reactants and up to five products). The last two 32-bit values represent the simulation time the reaction occurred (t') and a pointer that specifies which reaction of the BioModel has been selected (R_μ). The performance results of the three FRM-SSA core design (N=1 or 2 or 4) are presented in Table 7.2 in terms of time per Reaction Cycle and the number of Reaction Cycles that can be executed per second when the clock frequency of the cores is 160MHz.

7.4.2 Custom core automatic parameterization

The three developed cores presented in Table 7.2 were used as third party IP cores and added to SysPy's component libraries. Python functions that automatically generated and parameterize the HDL code of the FRM cores were developed and added to the tool's function library, as described in Section 4.1.2. Using Python, any text based file can be easily parsed so that its content used to provided special generic parameters for IP core instantiation. Function handlers are used to process the content of XML Biomodel files in SBML format, where the BioModel's file name is used as a special function argument (see Figure 4.2).

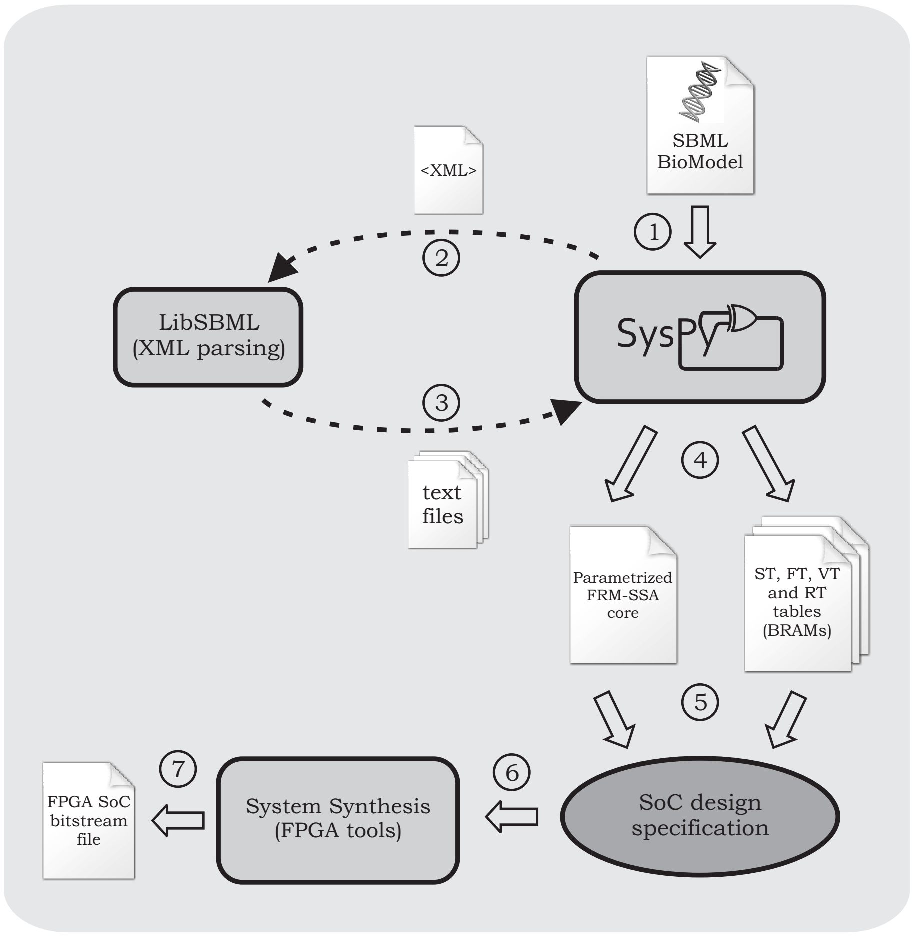


Figure 7.6: Parsing SBML BioModel files using SysPy.

An external C library called LibSMBL [16] is called within SysPy to identify the SBML related XML tags in the BioModel file and generate an intermediate raw text file. This file is then processed further by the associated with the FRM core function handler and used to initialize the memory block in the FRM-SSA core. The LibSBML library provides several functions to manipulate SBML files which were easily embedded in SysPy. As shown in Figure 7.5 all the memory tables, mapped to BRAMs, in the FRM core have to be initialized with data based on information available in the BioModel's file. In Figure 7.6 we show the steps of the parsing procedure of the BioModel file using SysPy. The Python function is also used to initialize the values of the generic parameters of the SSA core. All the values for the generic parameters in Table 7.1 are defined in a Python dictionary, while the values of m and n are parsed from the SBML file. Complexity of a biomolecular reaction network is directly related to these two values.

7.4.3 Leon3 interface and connection to the FRM-SSA core

According to the processors already embedded to SysPy, Leon3 was the best candidate to support a processor-centric SoC and handle communication issues between a fast custom processor and a host PC. The FRM core is connected to the GPIO ports of the processor, while an FSM is used as a data bridge between the GPIO ports and the custom core. The processor handles data transfers on the Ethernet channel as well as data storage on the buffers of the data bridge during data processing and on the attached to the processor SDRAM memory, where initial and processed data sets are stored.

In Code example 7.1 we show the top-level Python description of the SSA SoC, where the SSA core, the interface FSM and the Leon3 processor core are connected. The Python function "Gillespie_FRM4X" (line 5), as described in 7.4.2, is used to instantiate the core. Call of the associated Python handler is done automatically by SysPy for parsing the SBML BioModel file. The FSM block instantiation ("HAL_FSM") is done using a port-map like assignment (line 8) in Python and a generic parameter passed to the associated function defines the size of its data ports. The Leon3 IP core is connected to the design using a port-map like assignment (line 16), so the user can define her/his own internal or I/O signals that are connected to Leon's signals. Internal signals "inputA" and "inputB" (lines 25-26) are used to connect the FSM to Leon. The internal signal "clk_int" is used to clock the SSA core and the FSM at 160MHz (line 30). All internal and I/O signals of the generated VHDL entity, must be declared at the end of the Python description, using dictionary statements. A set of software files must be also provided by the user to program the processor. All the files are automatically compiled using the "Sparc GCC" C compiler. For the biomolecular SoC three C files were used and compiled (lines 31-32) to a single binary executable file: the `greth_api` C library contains functions to access the Ethernet controller of the board while the `Python_intrf_user` and `Python_intrf` files contain the user application which implements the interface with the host PC. The C files must be provided by the user under his/her working directory, as shown in Figure 1.1. All the required arguments are passed in the end to "to_VHDL" function to generate all the HDL files and the folders hierarchy to be

Code Example 7.1: Python description of the SSA SoC top-level design file, using port-map like assignments.

```
import SysPy_ver._toVHDL
def FRM4XplusLeon():
  # Connecting the FRM4X SSA core
  # and passing as argument the name of the SBML file
  func_Gillespie_FRM4X(``Biomodel.xml'')

  # Connecting the FSM as a compoment using port-map like statement
  def struct_HAL_FSM(``32''):
    clk = clk_int
    ce2int = ce2int
    PORTA_in = inputA
    PORTB_in = inputB
    .
    .
  # Connecting the Leon3 core
  def struct_leon3mp():
    sys_rst_in = sys_rst_in
    clk_out = clk_int
    clk_100 = clk_100
    clk_200_p = clk_200_p
    clk_200_n = clk_200_n
    clk_33  = clk_33
    sram_flash_addr = sram_flash_addr
    sram_flash_data = sram_flash_data
    PORTA_out = inputA
    PORTB_out = inputB
                .
                .
# Leon's software C file names
attributes = {``SYS_FREQ:'' 100, ``PROC_FREQ'': 160,
              ``PROC_SW'': [``Python_intrf_user'',``Python_intrf'',
              ``greth_api''], ``FPGA_DEV'': ``Virtex5''}

# Generic argument for Python functions
generics = {``Gillespie_FRM4X'': {`q': 3:, ``Nrep'': 10, ``Tsim'': 50000, `K': 0,
           ``RNGseed1'': 3, ``RNGseed2'': 34, ``RNGseed3'': 100, ``RNGseed4'': 180}}

# I/O and internal signal declaration
i_sigs0 = {`D': `i', `T': `b', `L': 1, `N': [``sys_rst_in'', ``ctrl'', ``clk_100'',
           ``clk_int'', ``clk_200_p'', ``clk_200_n'', ``clk_33'', ``sram_clk_fb'',
           ``phy_tx_clk'']}
i_sigs1 = {`D': `intr', `T': `b', `L': [31, 0], `N': [``inputA'', ``inputB'']}
.
# Calling the "to_VHDL()" function to generate VHDL descriptions
SysPy_ver._toVHDL.toVHDL(``FRM4XplusLeon'', attributes, generics, i_sigs0,
                  i_sigs1,...)
```

used as input of the FPGA synthesis tools, as described in Chapter 3.

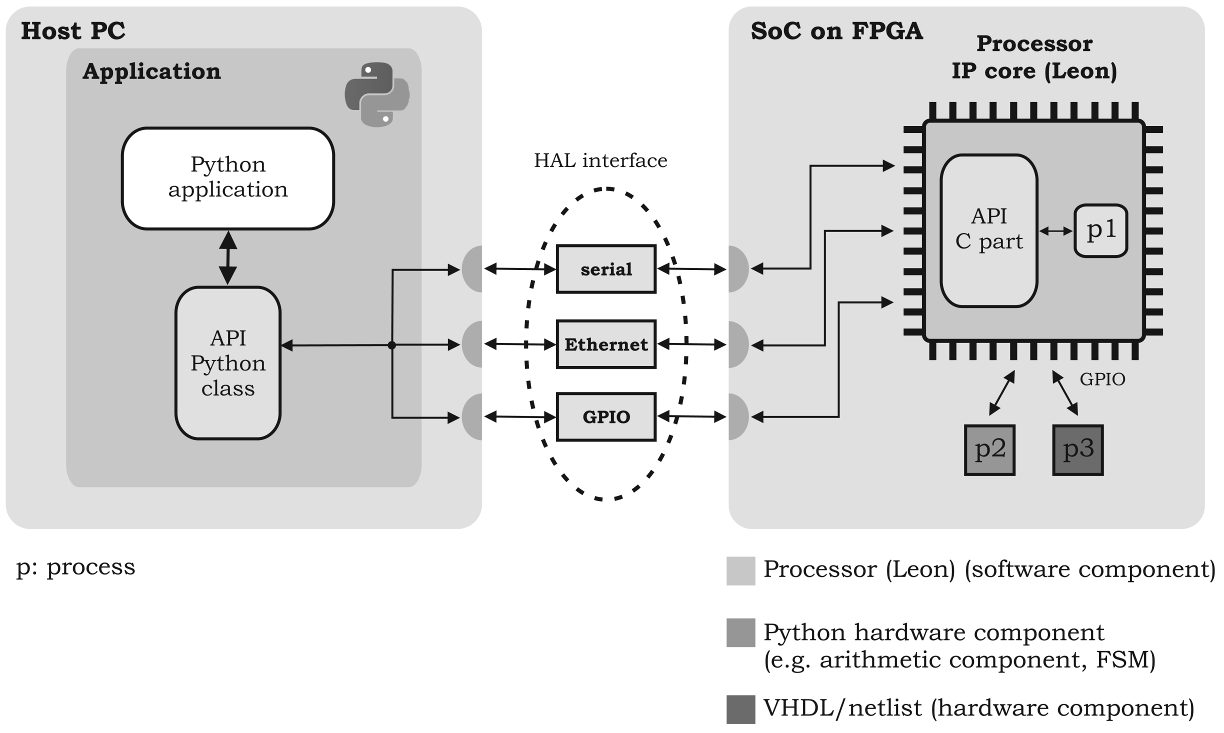


Figure 7.7: Hardware Abstraction Layer API for interfacing a typical processor-centric SoC running on the FPGA.

A piece of software was required on the PC side to handle: a) data flow from the PC down to SSA core through Leon and also b) the memory transactions on the onboard SDRAM memory. This interface software was implemented as a HAL API software for the Leon3 core. The interface supports one control channel and two fast data channels. The control channel is implemented using the serial connection between the PC and the FPGA board. The first data channel is based on the Ethernet 100Mbps connection and targets interfacing of the SoC with other external digital systems. The second data channel is used for intra-chip communication and is based on the GPIO channel connection between Leon and the FRM core. Python was used to develop the HAL on the PC side, so that to create a unified environment where Python is used to design and also interface a complex SoC design after its FPGA implementation. Object oriented capabilities of Python also let us develop the HAL as a class with associated functions for data handling.

In Figure 7.7 the HAL implementation and connection between the PC and the FPGA board is presented. The API is partitioned into two parts: a) a Python part running on a host PC and b) a part developed in C running on Leon in the FPGA. Functions are provided through the HAL to receive and transmit data on all three supported data and control channels (serial, Ethernet and GPIO). Identical functions for data transmission and control have been implemented in Python (Python API part) on the PC side and in C (C API part) on the processor side on the FPGA. Communication parameters e.g. Ethernet MAC address and serial port number, can also be set using the Python API class attributes. Execution of a Python function triggers the execution of the corresponding mirror function on the SoC's processor. Data transmission on the GPIO channel is triggered using control commands over the serial channel and is handled by the implemented interface FSM. With Leon operating at 160MHz, a 25Mbps data rate is realized over the GPIO channel.

In Code example 7.2 we show how the Python HAL interface can be used on the host PC side to create a HAL object and exchange data with the SSA core over the GPIO channel. An object named `Leon` (line 4) inherits all its properties from the `HAL_API` class. Communication attributes are defined (lines 6-13) for the created object, such as the MAC address of the PC and the FPGA board and the serial port number of the PC. Data sent from the SSA core to the PC through the GPIO channel are stored in the `gpio_rx_data[]` list after calling the `gpio_rx() function` (line 31). Data sent from the PC to the SSA core must be first stored to Python lists and then pass them as arguments to the available functions `gpio_tx()` or `eth_tx()` (line 26) functions. The first function can be used to transmit directly to the SSA core or the latter one to sent data to the SDRAM memory of the FPGA board. All data are transmitted using Ethernet MAC long packets of 1024 bytes. The Python and the C HAL API slice data stored in Python or C lists before transmission.

By using an object oriented programming environment for managing the hardware resources of an embedded system, it becomes easier to distinguish data and the control processes for storing and transmitting. A very good feature of Python is the ability to support text processing and scripting features with object oriented capabilities. While object orientation was not used for SysPy's hardware description features, object orientation was very useful and handy for creating the HAL software. Object oriented languages have been used in other projects to control functions of an embedded system. In [82] Java has been used to develop a HAL software running on top of a Java Virtual Machine (JVM) running on the programmable processor of a SoC. Java classes and functions are used for handling I/O

Code Example 7.2: Creating a Python object to transmit/receive data over the GPIO channel.

```
import _HAL_API

# Creating object using the ``HAL_API'' class
Leon = _HAL_API.HAL_API()

# PCs Ethernet card interface name
Leon.ethernet_interface = r'\Device\NPF_{514ED014-A2E9-4E68-8C7D-9AD9FBA598...
# PC's Ethernet MAC address
Leon.dest_MAC = ``0:23:8b:37:8f:81''
# FPGA board's Ethernet MAC address
Leon.source_MAC = ``30:31:32:33:34:35''
# PC's serial port address
Leon.serial_interface = ``COM1''

# Lists' declaration
gpio_rx_data = []
gpio_tx_data = []

# Initializing ``gpio_tx_data'' list
for i in range(2000):
        gpio_tx_data.append(i)

# Transmitting 0x7D0 (2,000) x 32-bit (list "gpio_tx_data")
# data to GPIO port C buffering them in SDRAM memory at address 0x0
# gpio_tx(str port, int gpio_data_tx[], str start_addr, str data_len)
Leon.gpio_tx(`c', gpio_tx_data, ``00000000'', ``000007D0'')

#Receiving 0xC350 (50,000) x 32-bit data from GPIO port C
#buffering them in SDRAM memory at address 0x0
# gpio_rx(str port, str start_addr, str data_len)
gpio_rx_data = Leon.gpio_rx(`c', ``00000000'', ``0000C350'')
```

peripheral devices e.g. serial and Ethernet communication controllers.

Use of Python to create the HAL API software is the first attempt to use a scripting language to control a SoC's data processing tasks. Using a Python script on the host PC side to initiate processing tasks on the FPGA gives flexibility to the user to pre-process data and also schedule the processing and communication tasks in the SoC in a very structured way.

7.5 Performance and implementation results

The ML509 Xilinx board equipped with a Virtex-5 XC5VLX110T-1 FPGA device was used for the implementation of the biomolecular SoC. The available PHY Ethernet chip and the 256MB DDR2 SDRAM module clocked at 190MHz along with the FPGA device were utilized from our design. The board was equipped with an 100MHz crystal oscillator and with the use of the DCM block in the FPGA, we managed to generate all the required clock signals in the design, especially the 160MHz signal used to clock the Leon3 processor and the SSA core. The board was externally connected to the host PC using a serial RS-232 and an Ethernet cable. The serial connection was used to trigger data execution on the board, using a Python script which imports the HAL library, as the one presented in Code Example 7.2. A top level schematic of the SoC along with the connected peripheral devices on the FPGA board is shown in Figure 7.8.

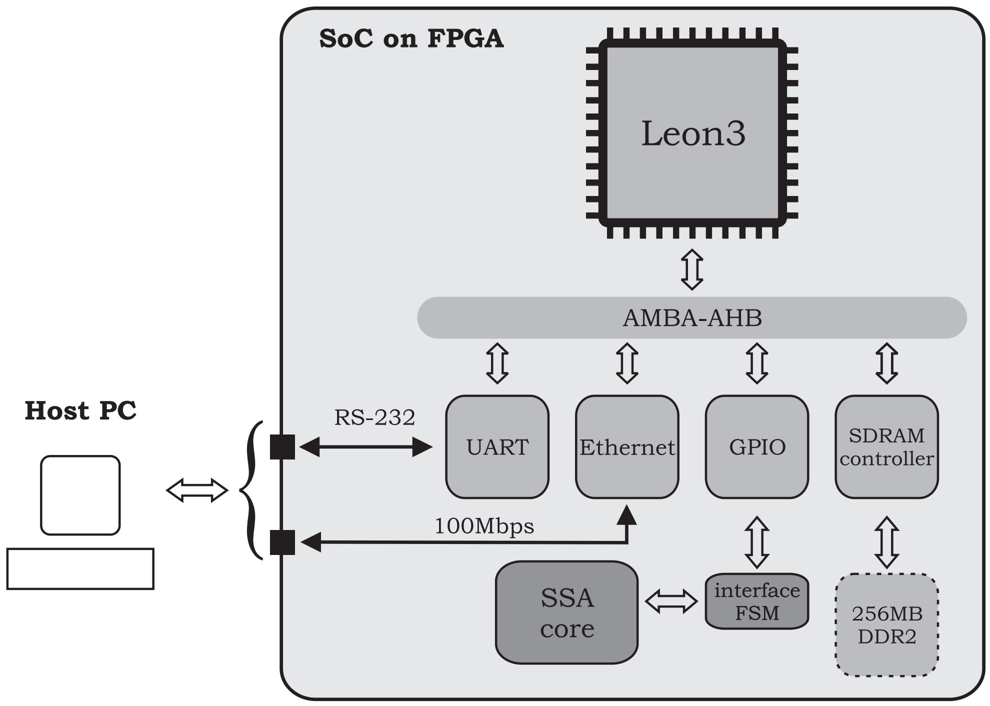


Figure 7.8: SSA SoC. Connection of the SSA core to the Leon processor.

A complex SBML BioModel [78] having $n = 93$ chemical species and $m = 136$ reactions have been used to test the SoC on the FPGA. The memory tables of the FRM core (FT, RT, ST and VT) have been initialized using the content of the BioModel SBML file. The SoC has been synthesized and tested with all three different version of the FRM-SSA core shown in Table 7.2, using Leon connected to an SSA core with either one (FRM1X), two (FRM2X) or four (FRM4X) PEs. FPGA resource utilization and power consumption results for all three implementations and also the Leon3 core alone are presented in Table 7.3. The Virtex-5 device present on the board was able to support up to $N = 4$ PEs working in parallel (FRM4X core). The maximum number of PEs that can be instantiated on a given device depends on the complexity of the BioModel file in terms of m and n. A higher number of species and reactions requires more BRAMs to hold the associated memory tables. Larger SoCs of the same architecture but with $N = 8$ PEs have also been synthesized for larger Virtex-5 LX155T and Virtex-6 LX240T FPGA devices.

The Leon3 core has a small footprint on the FPGA, since it utilizes 1/3 of the available logic blocks and 11% of the BRAMs for the data and instruction cache memories. All the rest of the occupied BRAMs were used for the implementation of the memory tables inside every

	Leon3	Leon+FRM1X	Leon+FRM2X	Leon+FRM4X
Slices	5,436 (31%)	9,244 (53%)	13,214 (76%)	16,594 (96%)
BRAMs	17 (11%)	56 (37%)	78 (52%)	132 (89%)
MULs	0 (0%)	16 (25%)	26 (41%)	48 (75%)
Power (W)	0.6	4.1	4.8	5.9

Table 7.3: Resource utilization for the Virtex-5 XC5VLX110T-1FF1136 FPGA device (Slice: four 6-input LUTs, BRAM: 36Kb, MUL: 25x18-bit).

PE, while the MUL blocks were utilized for the single precision floating point arithmetic calculations of the FRM algorithm. Since FPGA devices are used to parallelize algorithms using the large number of arithmetic and memory blocks, FPGA implementations do not favor low power designs, compared to ASIC implementations. Despite that we can observe that the Leon processor core consumes only 10% of the dissipated power. Power consumption was estimated using Xilinx's XPower Analyzer tool. Most of the power is dissipated on the memory, multiplier and especially I/O blocks, since a large number of external pins are used to interconnect the FPGA with the Ethernet and memory peripheral ICs on the board.

7.5.1 Comparison against software based tools for BioModel stochastic simulation

In order to validate our SoC implementation we performed measurements of critical control and data signals using the Xilinx ChipScope Pro [52] logic analyzer. Several signals were captured and analyzed to prove that all the required computations occur in the correct order and that the computations have the required accuracy. BioModels simulation results were transferred back to the host PC, where data returned from the HAL's `gpio_rx_data[]` (Code Example 7.2) are exported to text files for further processing. These results were compared to the results generated while simulating curated models from the SBML BioModels Database [27] using popular software simulators, like iBioSim [70] and StochPy [85]. We tested the performance of the software simulators using a modern and fast PC configuration where the latest version of the required Windows and Linux O/S were installed (Windows 7-iBioSim/Ubuntu Linux 12.04-StochPy, 64-bit PC, 6GB RAM, Intel i7, 2.6GHz, quad-core CPU). We used a complex BioModel for our tests [78] which forms a biomolecular network with $m = 136$ reactions and $n = 93$ chemical species and models the role of the A-synuclein (ASYN) protein on the homeostasis of dopaminergic neurons. Modeling this protein is of great importance for understanding the pathogenesis of Parkinson's Disease.

The FRM core was implemented with four PEs (FRM4X) working in parallel and processed data in SSIP mode, where each PE simulated 1/4 of the available reactions. Simulation process was initiated using the proper command sequence from the Python HAL interface. Simulation times presented in Table include the following individual processing/data transmission times:

- Command transmissions, using the serial channel, from the host PC to Leon, initiating the simulation on the SSA core.
- Simulation processing time of the FRM4X SSA core.
- Data transmission from the SSA core to Leon using the GPIO channel.
- Buffering data on the SDRAM memory on the FPGA board.
- Data transmission from the FPGA to the PC using the Ethernet channel.
- Saving simulation results on the PC's disk drive.

Six different simulation run were conducted using the FRM SoC, iBioSim and StochPy and the results are presented in Table 7.4. Simulation experiments were performed for three different simulation time values (T_{sim}), namely 50,000, 200,000 and 600,000 seconds (of lab time) and conducting one ($N_{rep} = 1$) or ten repetitions ($N_{rep} = 10$) respectively, while simulation performance was measured in MReactions/sec. The goal was to measure the speed factor that the hardware implementation achieves compare to the software tools. As presented in Table 7.4 the performance of the SoC design exceeds by 50 times or more that of the software simulators. Also the parameter T_{sim}, which is the real lab time required to perform the requested reaction cycles, does not affect the performance of the hardware design and remains constant at about $0.35 MReactions/sec$. When we performed multiple

ASYN BioModel in SSIP mode ($m = 136, n = 93$)								
$N_{rep} = 1$								
T_{sim} (sec.)	**FRM4X**		**StochPy**		Speedup factor	**iBioSim**		Speedup factor
	End-to-end time (sec.)	MReact./sec.	Sim. proc. time (sec.)	MReact./sec.		Sim. proc. time (sec.)	MReact./sec.	
50k	1.06	0.353	58.2	0.0066	54.9	54.5	0.0069	51.4
200k	4.38	0.354	231.0	0.0071	52.7	223.0	0.0071	50.9
600k	12.6	0.354	623.0	0.0075	49.4	584.0	0.0078	46.3
$N_{rep} = 10$								
T_{sim} (sec.)	**FRM4X**		**StochPy**		Speedup factor	**iBioSim**		Speedup factor
	End-to-end time (sec.)	MReact./sec.	Sim. proc. time (sec.)	MReact./sec.		Sim. proc. time (sec.)	MReact./sec.	
50k	10.6	0.353	615.7	0.0064	58.1	792.0	0.0049	74.7
200k	44.4	0.354	2425.0	0.0066	54.6	3952.0	0.0040	89.0
600k	127.0	0.354	7400.0	0.0061	58.3	14545.0	0.0031	114.5

Table 7.4: End-to-end performance comparison of FRM4X SoC to software simulators.

simulation runs ($Nrep = 10$) of we observed a non-linear increase on the required simulation time T_{sim} while using iBioSim. On the contrary the performance of StochPy remained almost constant. When the number of repetitions was greater than two ($Nrep > 2$) we noticed that iBioSim had a poor performance because it was executing a large number of disk accesses.

Comparison was also performed with biochemical reactions performed in a stochastic way, where computer clusters were involved to apply distributed computing techniques. In [79] a 20 nodes cluster (each node is a 2.4GHz AMD Opteron dual-core CPU) was used to simulate BioModels behavior. Two BioModels, the logistic-growth with only two reactions ($m = 2$) and the epidemiological metapopulation model (SIRS) applied to 100 "patches" having overall $m = 4 * 100 = 400$ reactions were simulated in SSIP mode in the cluster. The performance achieved by [79] is 0.132E-3 MReactions/sec for the logistic-growth and 0.029 MReactions/sec for the SIRS model (estimated using the data reported in Table 1 of [79]). The performance of our FPGA implementation is more than an order of magnitude higher than the cluster's performance. Nowadays a large number of PEs can be instantiated and interconnected in a single large FPGA device, so it comes as no surprise that the FPGA implementation outperforms a computer cluster implementation, where a lot of time is wasted on data communication across many different circuit board and ICs interconnected using many different transmission protocols. Furthermore the FPGA implementation offers an order of magnitude more area and power efficient solution.

In this SoC design we demonstrated how SysPy can be used to combine a 32-bit programmable processor core along with a dedicated custom co-processor, targeting applications related to a real world, biologic related application. Using Python we also managed to provide a unified development and testing environment, since Python controls the hardware/software design related flow, but also interface application after the implementation of the SoC in an FPGA device. The design has been tested thoroughly using real biological models and has been developed in a way that is easy to be used and interfaced by users and scientists with little or no experience in digital hardware design.

Chapter 8

Audio processing SoC design case

In the current chapter we demonstrate the verification and O/S based software development flow supported in SysPy, via the design of an audio signal processing SoC, implemented using a Xilinx Virtex-5 FPGA device. We present how SysPy facilitates SoC development early in the design phase where models of hardware components (not yet captured in HDL) and software code to run in the processor core can be co-simulated using Python descriptions. The design is based on the Leon3 core, running an embedded Linux O/S. In this way file-oriented data processing is achieved, while the FPGA board acts as a data co-processor attached as a network node in an Ethernet TCP/IP network.

8.1 Audio SoC features

In this design we use a bank of four FIR bandpass filters, which divide the audible spectrum into four regions. According to the filtered audio signal strength in these four audio bands, the processor in the design can classify a music file into one of four styles (rock, pop, classical, electro). Several features of SysPy were exercised during the design of this SoC, like high-level hw/sw co-verification, automatic generation of RTL code and software development using an embedded Linux O/S kernel.

With this design example we tested SysPy's high-level verification features by creating a Python testbench when the specification of the system was available. With the testbench we were able to define in advance key parameters of the system that would let us to analyze accurately and fast the frequency content of the music files. The required data fixed-point representation was defined using the testbench, as well as the size of the memory buffers between the processor and the filter bank. After defining the parameters of the system, we used high-level Python descriptions for the instantiation of the processor IP core and of the high-order digital filters (accelerators), which SysPy translated to synthesizable VHDL code. The RTL code for the filter bank was auto-generated using Python parameterizable functions.

On this design we wanted to use the Ethernet connection in a different way, compared to the biomolecular SoC design where a bare-metal application was executed by Leon. Using a standalone application it was hard to use all the layers of the Ethernet protocol. That is why the HAL interface for the biomolecular SoC was implemented, exchanging data using MAC packets. In the current design the software of the application is compiled along with the Linux embedded O/S kernel and the application take advantage of the O/S: a) implemented TCP/IP stack and b) installed, ready-to-use application, like an FPT client used to transfer data between the FPGA board and a connected via Ethernet host PC. Decisions were also taken about the control software that the processor would have to execute in order to store the music files, broadcast data to the filters, collect and store the filtered results and finally analyze the results to define the music genre of each file. Linux kernel parameters were also specified using dedicated configuration files, while the kernel and the related C application were compiled through SysPy, using the dedicated C compiler supporting the Leon architecture. Using the top-level Python description, SysPy generates automatically: RTL VHDL code, compiled software for the processor and all the script files necessary to facilitate synthesis and physical implementation of the SoC.

8.2 Audio processing SoC design

8.2.1 SoC verification using SysPy

For the design of the SoC we used SysPy to create a Python testbench of the system early in the design phase. The testbench was used to capture the complete system's functionality using Python and co-simulate its hardware and software. Simulation models were used to describe the functionality of the filters and also Python code to define the control algorithm needed to be executed by the processor. The testbench was built in such a way so that during the design phase it was easy to exchange the hardware simulation models by Python functions that would generate the required RTL code and instantiate the blocks in the top-level Python description.

A top-level schematic of the SoC is shown in Figure 8.1. Leon3 is running an FPT client, along with the user custom application in the Snapgear Linux O/S.The custom application uses the client to establish an FPT connection with an FTP server running in the connected PC. Upon reception of a music file, data is stored in the system's RAM memory and the custom application, implemented in C, copies the file content to an array. The filter bank is connected to the GPIO ports of the processor, through a FIFO buffer, which is required to store and synchronize data transmission between the two different clock domains of the processor and the filter bank. The Universal Asynchronous Receiver Transmitter (UART) serial connection is used to provide command line access to the O/S, where the user can trigger execution of his application.

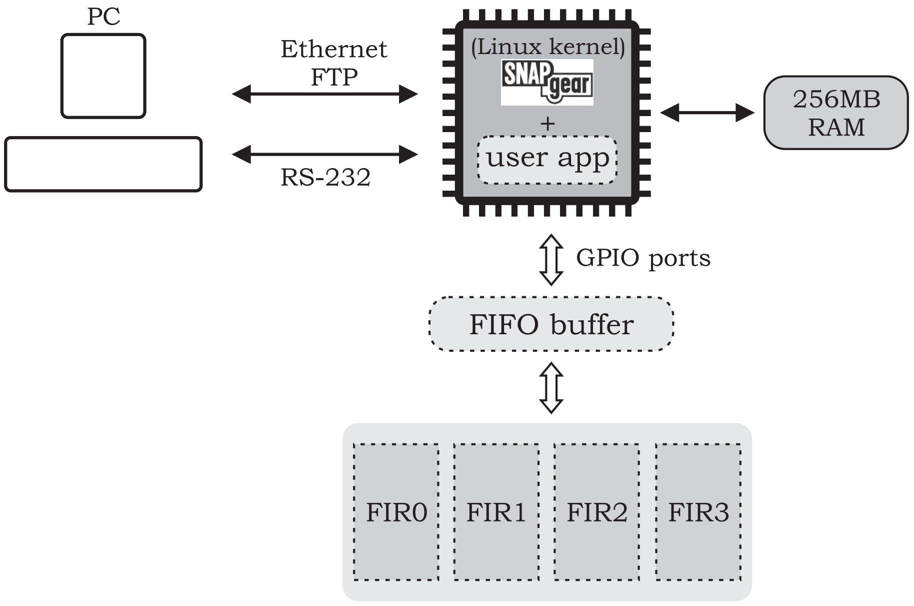


Figure 8.1: Top-level schematic of the audio SoC.

In Figure 8.1 dashed line boxes are used for the hardware and software modules that we simulated in our testbench. By simulating the SoC's functionality at a high-level using Python models, we were able to make judicious decisions regarding the following key aspects of the hardware/software design space: a) filter properties e.g. fixed-point notation, number of taps, value of the taps using SciPy according to the desired cutoff frequencies, b) size of the data buffers and control signals needed to handle data flow between the processor core and the filter bank and c) control software running on the processor, which was later developed on top of the Linux kernel, that allows data to be handled in a file oriented manner.

In Figure 8.2a a more detailed block level schematic of the abstract simulation model of the audio DSP SoC is presented. Software blocks are represented using dashed line boxes inside the processor's block, while the rest of the blocks correspond to hardware functionality.

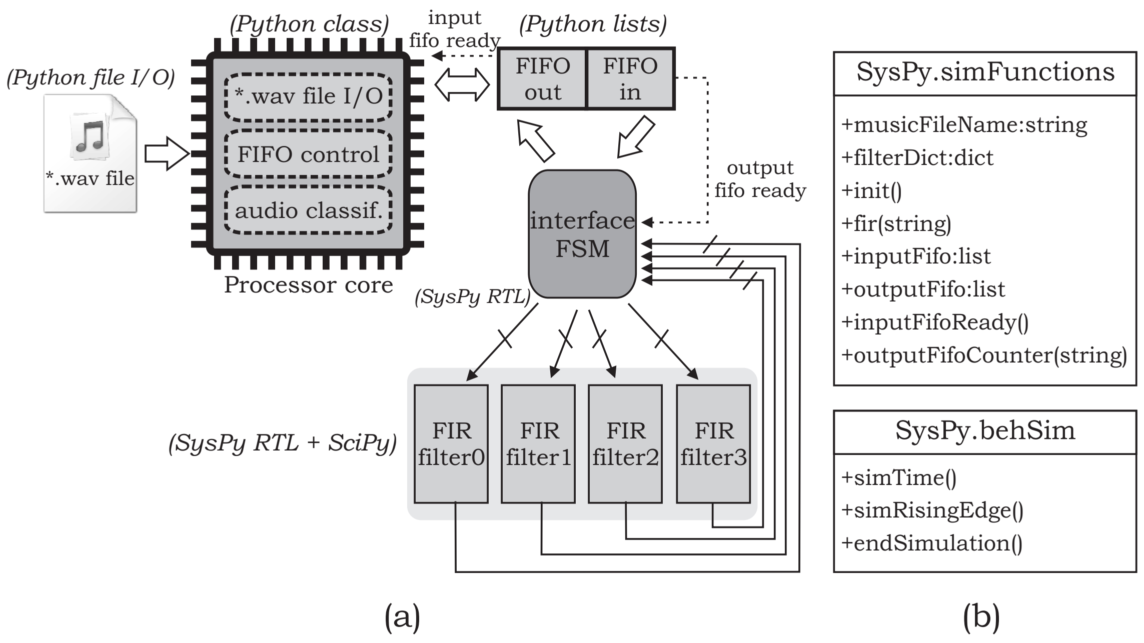


Figure 8.2: a) Abstract modeling of the SoC using SysPy and SciPy, b) diagrams of the Python classes used in the SoC's testbench.

Software on the processor handles music file I/O, transmits the audio samples to the input FIFO and reads back the filtered samples. Finally the processor classifies the audio files by analyzing the filtered samples from the four audio bands. Simulated hardware functionality in the SoC involves the FIFO memories, the interface FSM that handles the data traffic from and to the FIFO memories and the four FIR filters.

The text in parentheses in Figure 8.2a states the type of Python structure used to simulate each block. Functions of the core classes of SysPy support the timing mechanism of the cycle-accurate simulation, while user provided classes model the algorithmic behavior of the SoC, especially of hardware blocks that have not yet been defined at the RTL level. Diagrams of the Python classes used for the audio SoC simulation are shown in Figure

8.2b. The three functions provided by SysPy's `behSim` class handle timing information in the testbench: `simRisingEdge()` to simulate a pipelined synchronous datapath, according to the provided clock source, `simTime()` which makes the simulation time visible to the testbench and `endSimulation()` which terminates the simulation when a desired condition is met.

A number of data structures and functions are defined in the `simFunctions` class provided by the user. The `inputFifo` and `outputFifo` Python lists are used as data buffers and simulate the behavior of the data transactions between the processor and the filter bank. During system reset, function `init()` initializes the data buffers and calculates the filter tap values, according to user supplied specifications (cutoff and sampling frequencies, number of taps) provided for each filter in the `filterDict` dictionary. The processor and the filters exchange data in an asynchronous way using control signals generated by the two implemented buffers. Function `inputFifoReady()` is used to inform the processor that the input buffer is ready to accept data, which the processor reads from the provided music file (`musicFileName`). Function `outputFifoCounter()` returns the number of existing filtered data in the output buffer and the simulation is terminated if the number of data has reached a desired value. Function `fir()` is used to simulate, in a bit-true manner, the registered datapath of the filters. The number of registers is defined by the number of the required taps for each implemented filter. Control and storage functions can be described in an abstract manner, using Python libraries.

The I/O signals among the blocks in the testbench are pipelined using the timing simulation functions provided by SysPy. Signals' values are also stored in VCD file format, so that their behavior can be checked in respect to the clock and reset circuitry of the system.

8.2.2 Filter bank design using SysPy and SciPy

One of the main targets of the developed testbench for the audio SoC was to define and fine-tune the parameters of the four implemented filters. Three different aspects of the filters could be easily explored using the SysPy testbench:

- values of the filter coefficients to tune the filters to the desired frequencies
- number of taps of the filters

- fixed-point notation and number of bits used to express the parameters

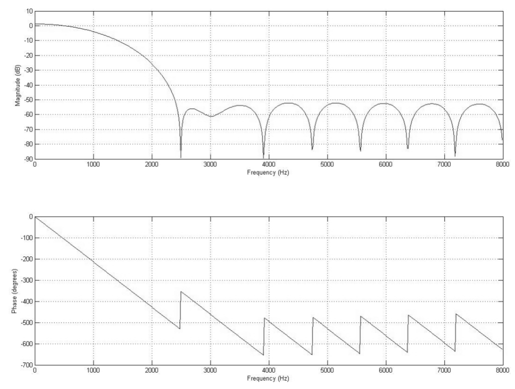
Calculation of the values of the parameters and the number of taps can be explored using SciPy provided functions for digital filter calculations, used within the model developed in SysPy. The accuracy of data processing is heavily affected from the chosen representation, something that was easy to explore using SysPy provided functions for converting fixed-point decimal numbers to binary format and vice versa.

In Code Example 8.1 a code snippet is presented with two functions used to simulate the datapath of the FIR filters. In line 1 the SciPy library is imported, while in line 3 the SWIG interface C function is imported. In line 4 the class `simFuntions` is defined to hold all the required data and functions for modeling the FIR filters. Sampling and cutoff frequencies and number of taps for each filter are stored in dictionaries (lines 8-12) and passed as arguments to the SciPy `firwin()` function which calculates the coefficients. Alternatively a C function can also be used to calculate the filter coefficients by passing the same arguments used with the `firwin()` function. The `hamming_win` in lines 19-20 is implemented as a C function using SWIG and returns back to the testbench the desired set of coefficients according to the input arguments (methodology on how to use C functions in Python is described in Section 3.3.2). The SysPy provided function `fp_sign_to_bin()` in line 26 is used to convert the coefficients to the corresponding fixed-point notation format. The `fir()` function in line 29 is used to implement the register logic of the pipelined datapath in the filters. The registers chain is created using the for loop in lines 30-32 and the Multiply Accumulate (MAC) operation for each tap is implemented using the for loop in lines 38-40. One data element is pushed to the datapath every time the `fir()` function is called. This function is called within the testbench on the occurrence of a rising edge on the main system clock, simulating in this way the clocking process of the datapath (Code Example B.5, lines 30-33). A described also in Chapter 3, the C code implemented using SWIG and co-simulated in SysPy, can be compiled and used directly to program the Leon3 processor.

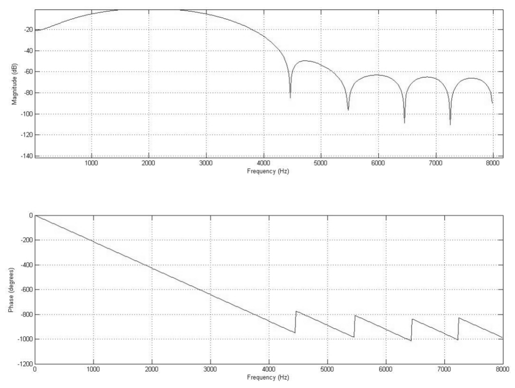
Code Example 8.1: Using SciPy for simulating the filter's datapath and C function for hw/sw co-simulation.

```
from scipy import signal
from _fp_sign_to_bin import *
import swig
class SimFunctions:
  # Filter0
  #------------------------------------------------------------
  # Calculate Filter0 coefficients using SciPy functions
  coeffFilter0 = signal.firwin(self.FilterDict["Filter0"]['N'],
                               [self.FilterDict["Filter0"]["fc0"],
                                self.FilterDict["Filter0"]["fc1"]],
                                nyq = (0.5 * self.FilterDict["Filter0"]["fs"]),
                                window = 'hamming')
  ## OR

  ## Calculate coefficients in C
  print "C implementation"

  for i in range(len(coeffFilter0)):
    coeffFilter0[i] = swig.hamming_win(self.FilterDict["Filter0"]['N'], i,
    self.FilterDict["Filter0"]["fc1"], self.FilterDict["Filter0"]["fs"])
  #------------------------------------------------------------

  # Convert the coefficients to the required binary fixed-point format
  coeffFilter0Fp = []
  for i in coeffFilter0:
    coeffFilter0Fp.append(int(fp_sign_to_bin(i, self.fpNotation), 2))

  # Simulate the FIR pipelined data path
  def Fir(self, SignalIn, FilterName):
    for i in range((len(self.FilterDict[FilterName]["registerQueue"]) - 2), -1, -1):
      self.FilterDict[FilterName]["registerQueue"][i + 1] =
      self.FilterDict[FilterName]["registerQueue"][i]

    self.FilterDict[FilterName]["registerQueue"][0] =
    int(self.MusicFileArray[self.FilterDict[FilterName]["Counter"]])

    acc = 0
    for i in range(len(self.FilterDict[FilterName]["registerQueue"])):
      acc = acc + self.FilterDict[FilterName]["Coefficients"][i] *
            self.FilterDict[FilterName]["registerQueue"][i]

    self.FilterDict[FilterName]["OutPort"] = acc
    self.FilterDict[FilterName]["Counter"] = self.FilterDict[FilterName]["Counter"] + 1

    self.outputFifo[FilterName].append(self.FilterDict[FilterName]["OutPort"])

    return (self.FilterDict[FilterName]["OutPort"])
```

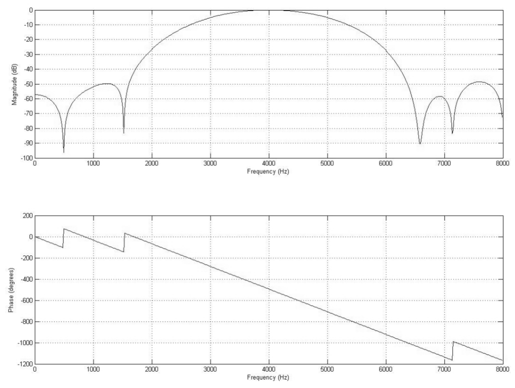
The four designed filters had 30-tap each, covering the audible spectrum and their coefficients were calculated using SciPy (0-1KHz, 1-3KHz, 3-5KHz, 5-8KHz). Filters with fewer taps have also been tested, but the best classification results, also considering the available resources on the FPGA, were obtained using 30-taps. The frequency and phase response of the four implemented filters is presented in Figure 8.3.



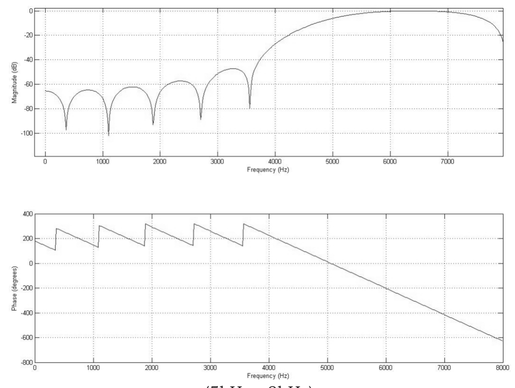
(0kHz - 1kHz)



(1kHz - 3kHz)



(3kHz - 5kHz)



(5kHz - 8kHz)

Figure 8.3: Frequency and phase response of the four implemented filters.

The model of the filters is instantiated at the Python testbench and connected to a state machine which controls data flow in the testbench from and to the processor. Using the testbench it was easy to change on the fly the parameters of the filters, data input or the state machine synchronization. The full testbench where the filters model is instantiated is presented in Code Examples B.4, B.5 and B.6.

8.2.3 Processor interface using SysPy

As presented in Figure 8.1, using the FIFO buffers blocks we were able to partition the design into two main clocking domains: a) the 160MHz domain where the processor is implemented and the 100MHz domain, where the FIFO memories and the filter bank are implemented. The clock frequency used for Leon is the highest that could be achieved on the Virtex-5 device. With this frequency a data rate of 25Mbps was measured on the connected GPIO ports.

While the data rate performance of the processor is not very high, the Leon core add software programmability in the system which is very important for handling I/O and control operations. Without the existence of the processor all the required drivers for the Ethernet, GPIO and UART communication should be hardcoded using huge state machines to implement all the layers of the related communication protocols. In order to decouple the performance of the processor from the high data rate performance of the filter bank, we used the FIFO generator tool provided in Xilinx ISE [51] to build a dual 64kx32-bit FIFO block.

The functional diagram of the FIFO block is presented in 8.4. According to the required FIFO size, the generator cascades the required number of built-in FIFO blocks. These blocks are physically implemented using BRAMs, distributed RAM (CLBs) or shift-registers. Two clocking domains exist in the FIFO design, supporting concurrent read and write operation. The generator tool also implements the FIFO interface logic, providing all the data I/O and control signals.

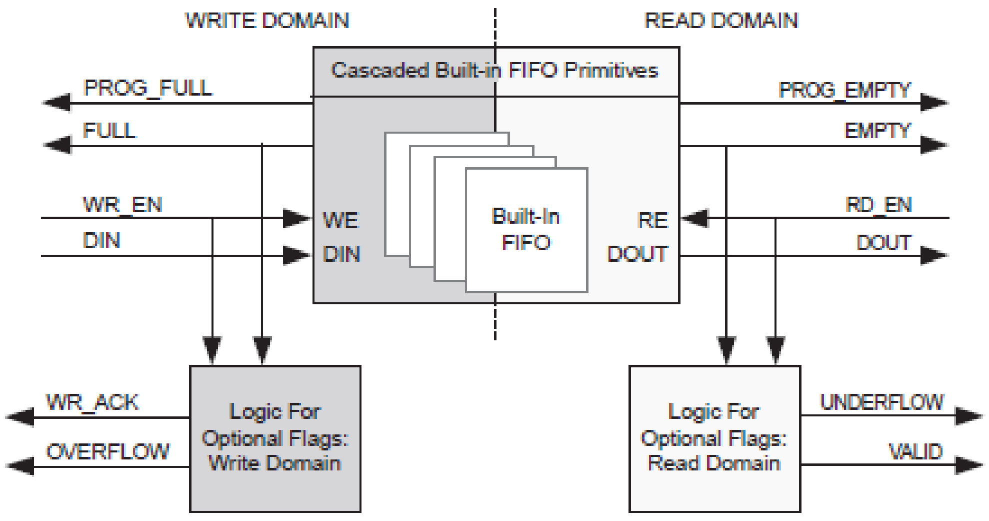


Figure 8.4: Functional FIFO diagram (source: www.xilinx.com).

The block diagram of the interface unit is presented in 8.5. The interface unit consists of two 64kx32-bit FIFO memories and four FSMs. When a new 8-bit data sample is received from the GPIO, it is stored in the Tx FIFO and then broadcasted to the filter bank unit consisting of four parallel filters. The interface unit and the filter bank are connected on the same clock domain, running at 100MHz. A new filtered value appears at the output of each filter after one clock cycle. The Rx FIFO collects the output from each filter (a 32-bit value) and stores the four outputs in four different memory locations. The processor polls the interface unit and fetches filtered results when available.

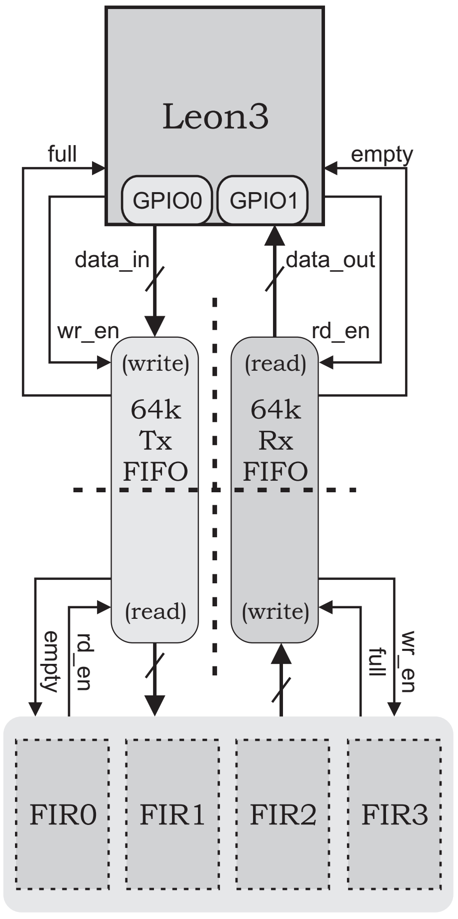


Figure 8.5: FIFO interface architecture in the audio SoC design example.

The FIFO design has been inserted as a block into the structural library of SysPy, so that it can be instantiated at a top-level Python description. The design has been generated and synthesized in the Xilinx ISE design environment and the generated netlist (*.ngc file) has been stored in SysPy's libraries. SysPy automatically creates the ISE compatible folders hierarchy and copies the required netlist files every time the FIFO block is used in a Python description. A smaller FIFO design has also been generated using the FIFO generator tool in SysPy. This design can be instantiated directly in the top-level module, using an HDL-like description and connected to the GPIO ports of the processor. Although the FIFO blocks have been synthesized for the Virtex-5 XC5VLX110T-1FF1136 device that we used, the synthesizer is able to re-synthesize the block for other devices without the user having to re-generate the block using the FIFO generator tool, if the FPGA device has the resources to support the FIFO design, e.g. compatible dual-port BRAMs.

8.2.4 SoC simulation results

A testbench setup was used to verify the SoC functionality using models for the software algorithm needed and for the design of the filter bank. We used SciPy FIR functions to model the filters data path behavior and also plots to observe the filters' response. Combining the filter models with abstract software descriptions in Python and also with the digital plot results generated in VCD format we were able to configure the SoC's arithmetic calculation and timing parameters.

In SciPy we plotted the input music signal waveform and the output waveform from Filters 0, 1, 2 and 3, as presented in Figure 8.6. A *.wav music file was used as an input to the filter with a sampling frequency of 16kHz and the first 100 samples are presented in the plot. Simulation results for the four filters was also stored in text files during simulation for further analysis.

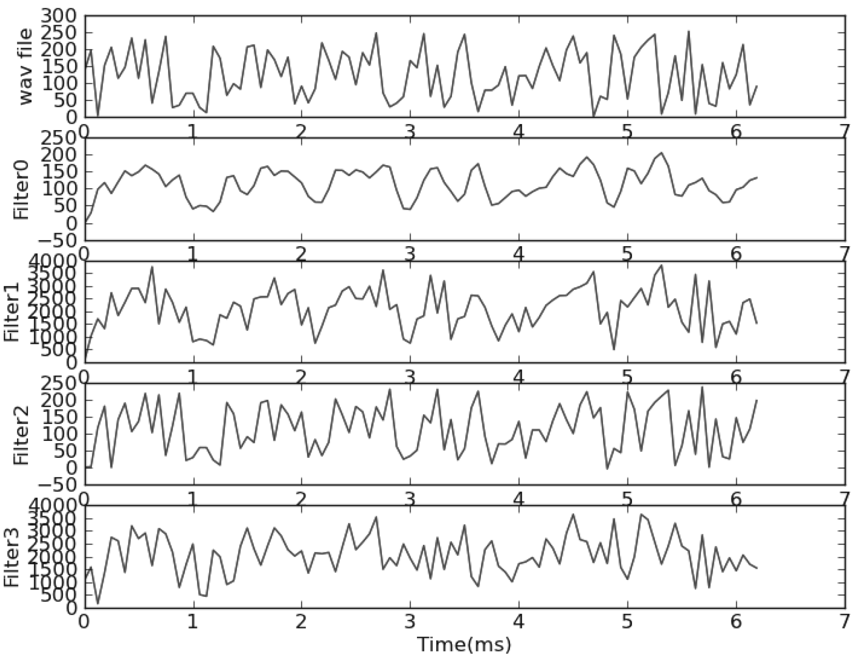


Figure 8.6: Filters time response plotted in SciPy.

Using the generated VCD file during simulation, digital waveforms can be observed using the GTKWave tool. In Figure 8.7 the I/O signals of the SoC, including all the clock, reset, control and data signals, are presented. The filter output results are presented after a delay that is user defined in the testbench (7ns) (Code Example B.6, line 14). In Figure 8.7a, signal `input_fifo_ready` is generated in the testbench in a random way, simulating the way the processor polls the control signals of the FIFO memory to provide new data samples to the filter bank. Using the testbench it was was easy to experiment and fine-tune the various parameters of the SoC. In Figure 8.7b, compared to Figure 8.7a, we changed the cutoff frequency fc_0 from 1kHz to 100Hz and also the output delay for Filter0 (signal `filt_out0`. The filtered values are different in Figure 8.7b and the logic delay has been increased to 15ns.

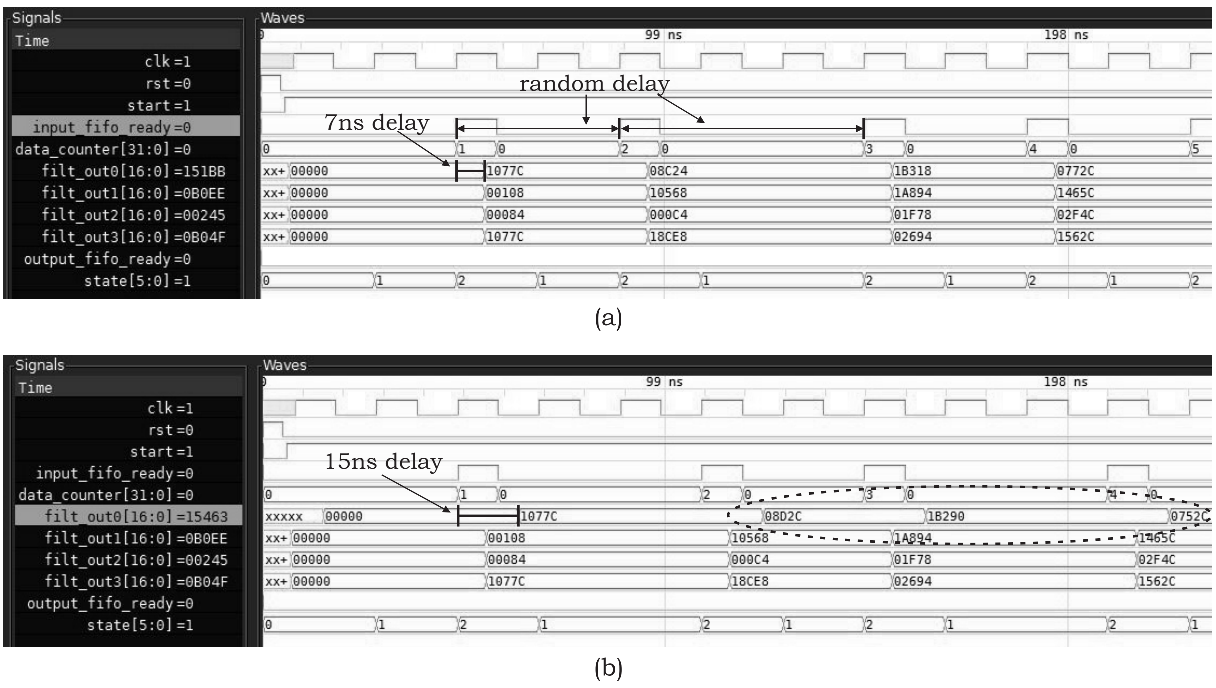


Figure 8.7: Digital I/O signal waveforms of the audio processing SoC.

Using the developed testbench for the audio processing SoC, it was easier for us to define the required set of tap parameters for the filters, the size of the FIFO memories, the clocking frequency of the system and data transfers synchronization between the processor and the filter bank. A prototype of the software executed by the processor has also been defined in Python before starting software development.

8.2.5 Mapping co-simulated design to hardware

It is highly desirable that any kind of models of a digital design must be easily adaptable to synthesizable RTL code. SysPy provides the means to ease hardware and software mapping, from Python hw/sw descriptions and behavioral models to synthesizable RTL and executable software for the processor respectively.

The state machine description in our testbench and the I/O interface of the SoC can be used within SysPy to automatically generate synthesizable RTL code. Moreover, for the implementation of the FIR filters, a parameterized Python function is used to auto-generate their synthesizable RTL description.

An example of the usage of Python functions to instantiate hardware blocks is shown in Code Example 8.2. Function `func_fir_filt_s` is used to instantiate an FIR filter in a VHDL RTL description. The function generates the RTL description of a filter with signed coefficients. In lines 5 and 6 the coefficient of the filters and the required fixed-point notation are provided as arguments in decimal format. The tap parameters of the filter are provided as they have been previously estimated during the SoC's simulation. SysPy will translate and initialize the values of the taps in binary format while instantiating the block in VHDL. In line 14 (8-bit + 1-bit for signed representation, 'n': 9) the number of the input bits is provided in the `generics` dictionary, while the fixed-point representation, the values and the number of taps are function parameters and are handled in an automatic way by SysPy during RTL code generation. I/O signal names are defined in lines 17-19. If predefined names are used (`filt_in, filt_out, clk and rst`) then they are connected automatically to the instantiated filter. The generated VHDL code for the filter can be found in Code Example B.7.

Leon is connected to the FIFO interface unit using its GPIO ports. The Python algorithm tested during simulation was used as a template for the development of the FIFO control and classification C code. The user application for the SoC was compiled and built on top of the Snapgear Linux kernel. The names of the O/S kernel and of the user C application file are defined in the `attributes` dictionary (lines 10-11, `linux_kernel, usr_app_classification`). SysPy makes an external call to the Sparc GCC compiler and one single binary executable (image) file is generated after compilation. Several Linux parameters can be defined in

SysPy's configuration file, e.g. RAM_SIZE, ETHERNET_EN, UART_EN. For the SoC implementation, a set of low level Linux GPIO drivers was developed in the user application. The full set of parameters of the Snapgear O/S kernel that can be configured through SysPy can be found in Table 1.2. The drivers for the ports were accessed by the application by mapping them to the RAM memory of the system. The application also performed processing of the filtered music values to find the genre of each music file. We defined four music styles/groups and assigned to them, according to measurements, a vector with the average values of the output for the four frequency bands of the implemented filters.

By using SysPy to describe the SoC, most of the system parameters, related either to hardware or software implementation are visible and configurable within the Python, self-contained, top-level description. Many system parameters also defined during system simulation using SysPy can be directly used in the system's Python description, like the filters' tap values, order of the filters fixed-point notation, SoC's control logic implemented in software and CPU and the SDRAM clock frequency.

Code Example 8.2: Python function description used to instantiate an FIR filter block.

```
import SysPy_ver._toVHDL
def filt_SoC():
  # 30-taps FIR filter with 1.7 fixed-point notation for
  # parameters's binary representation
  func_fir_filt_s([-0.0019, -0.0042, -0.0080, -0.0100,
                   -0.0028, 0.0648, 0.1203, ..., ], "1.7")

# Leon's software C file names
attributes = {``SYS_FREQ:'' 100, ``PROC_FREQ'': 160,
              ``PROC_SW'': [``linux_kernel'', ``usr_app_classification'',
              ``FPGA_DEV'': ``Virtex5''}

# Generic argument for Python functions
generics = {"fir_filt_s": {'n':9}}

# I/O and internal signal declaration
filt_in = {'D':'i','T':'b','L':[8, 0],'N':"filt_in"}
clk_rst = {'D':'i','T':'b','L':1,'N':["clk","rst"]}
filt_out = {'D':'o','T':'b','L':[16, 0],'N':"filt_out"}

# Calling the "to_VHDL()" function to generate VHDL code
SysPy_ver._toVHDL.toVHDL("filt_SoC", attributes, generics, filt_in, filt_out, clk_rst)
```

8.3 Software development using an embedded Linux kernel

8.3.1 Software debugging

Developing our software application on top of the Linux kernel, gave to the SoC access to fast I/O interfaces and large memory space. All the complex software interface layers needed to map the Ethernet and the SDRAM controllers were already implemented in the kernel. Required software also needed to handle file-oriented operation was also included in the kernel. The user control application used to handle data flow, storage and processing was compiled along with the O/S kernel and utilized all of the ready-to-use features of Linux.

Although software debugging during development was possible in SysPy, since compilation message appeared in Python's command line during parsing of the top-level design, more complex debugging techniques have to be used when compiling an O/S-centric application. To support this requirement Leon3 was implemented using: a) the UART interface, b) the Ethernet interface with TCP/IP support and c) the Debug Support Unit (DSU) interface. Implementation of the DSU block enabled the usage of the GRMON [10] debugging tool, supplied along with the Leon3 distribution.

To debug our software code, we configured GRMON to connect to the DSU through the Ethernet connection. Using the debug tool we were able to:

- download to the processors program memory (SDRAM) the compiled executable image, containing the Linux kernel along with the user applications
- verify hardware configuration of the processor (memory size, available peripheral units, clock frequency etc.)
- read/write memory content

Using the debug tool, we implemented step-by-step the communication sequence between the processor, the interface block and the filter bank. Having access to the entire memory space of Leon gave us the ability to test in real time the implementation of the GPIO low-level drivers and exchange data with the FIFO memories and the filters. Implementation of a programmable complex debug unit is another reason why the existence of a processor in a SoC design is useful not only during functional mode but also during test mode of a design.

8.3.2 Software development flow

The flow that we used during software development, followed the path of data flow in our system. A complete processing sequence in the audio SoC, implements the following steps:

(a) data transfer of an audio file from the host PC to the FPGA board

(b) save the file to the SDRAM memory

(c) open the file in embedded Linux and pre-process the audio samples

(d) send the audio samples to the filter bank

(e) receive and store the filtered samples

(f) analyze filtered samples and define the audio file genre

(g) send the filtered samples back to the PC using raw text file format

First thing was to setup the FTP connection between the host PC and Leon. An FTP server was configured on the PC side, while an FTP client application was activated in the Linux kernel on the FPGA side. A bash script was developed in Linux to setup the network connection (IP address, subnet and gateway) and to start the FTP client. All the scripts that we used were stored in a default directory of Snapgear Linux during development so that they were included in the image of the kernel after compilation.

The compiled C application was implemented in the kernel and registered as an environment variable, so that can be called from any path location of the O/S. The standard output of Linux was the serial terminal connection, where the bash script was executed. After having the audio files stored in the SDRAM memory the processing application was executed to start transmitting the audio samples to the filters and also receiving the filtered samples. The application is also parameterizable and can be used to process only part of an audio file. The audio samples were copied from the files to an array and then moved sequentially to the FIFO memory. New filtered samples were stored in four different arrays, one for each filter.

The software also used a moving time window to evaluate the mean value of the filtered audio samples for each filter. At the end of the filtering process, a vector with four values, representing the mean value of the filtered samples for each of the four frequency bands, was created. According to the distance of the vector from each of the vectors representing the available music groups, the audio file was categorized to one of these groups. The filtered samples were also copied to four different files and sent back to the PC through the FTP connection, while the assigned music group and the required processing time were presented in Linux command line.

By using a compiled C application to control data flow in the SoC and also scripts running in the O/S to handle file I/O operations, it became much easier to take advantage of the memory and I/O peripherals on the FPGA board. Using the O/S in our processor-centric SoC, it was also much easier to debug software and hardware issues. Also many different scripts and application could be loaded at the same time on the processor, merged together with the Linux image kernel. reducing in this way the time needed for software development since different programs could be tested at once, without having to reprogram the processor or the FPGA.

8.4 Implementation results

The ML509 Xilinx board from Digilent [25], equipped with the medium size Virtex-5 XC5VLX 110T-1 FPGA device, was used for system implementation. The board also includes a 256MB SDRAM DDR2 memory, clocked at 190MHz, which used as the main program and data memory for Leon3. An Ethernet PHY and a UART RS-232 chip are also included on the board and connected as peripheral devices to the processor.

During system development we managed to clock the processor at 160MHz, which is the maximum frequency for the Leon3 core. It was very important to get the processor to operate at the highest possible frequency since its performance is the bottleneck in the data processing path. The filter bank was clocked in its own domain at 100MHz. To estimate the processing throughput we had to measure the timing performance of the following individual processing and data transferring tasks taking place in the system:

- data transfer speed over the FTP connection

- communication speed between Leon and the filter bank
- filter data path clocking frequency
- processing of filtered samples

A music file having 3,924,170x8-bit samples was used to perform the timing measurements. The file transfer speed on the FTP connection, over an 100Mbps Ethernet connection, was 21.9Mbps. A 15MMAC/sec. performance has been achieved during the filtering of the music file. The number of filtered samples returned to the processor was four times higher (all the samples were processed in parallel from all filters). The data throughput processing of the filter bank was calculated using the following formula: $3,924,170 samples * 8 bit * 4 filters * 30 taps)/31.5$ *sec.* $= 119.6 Mbps$ have been processed in parallel in the pipelined path of the four FIR filters. This performance is four times higher than an all-software C implementation running on Leon and processing the same data. The same performance was obtained on average for many other music files. Timing performance results of the SoC are presented in Table 8.1.

	SoC	**Leon (C implementation)**
FTP transmission time (sec.)	1.4 (21.9Mbps)	-
Filter processing time (sec.)	31.5 (15.0 MMACs/sec)	132.3 (3.6 MMACs/sec.)
Data throughput (Mbps)	119.6	-

Table 8.1: SoC timing results when processing a music file (3,924,170 samples x 8-bit) compared to a C software implementation of the filtering algorithm running on Leon.

Resource utilization results are presented in Table 8.2 for the processor core only and for the entire SoC design for different number of filter taps. For 30-tap filters 120 multipliers were utilized, using multiplier/DSP slices or logic (LUTs) slices. While the processor core utilized almost 1/3 of the available logic resources, on the other hand through the software implementation of the embedded Linux kernel we managed to utilize the required memory and communication resources available on the board.

	Leon3	**15-taps**	**20-taps**	**30-taps**
Slices	5,436 (31%)	6,235 (36%)	6,880 (40%)	7,190 (42%))
BRAMs	17 (11%)	132 (89%)	133 (90%)	133 (90%)
MULs	0	60	80	120

Table 8.2: FPGA resources utilization used by the SoC's implementation in the Virtex-5 XC5VLX110T-1FF1136 device. (Slice: four 6-input LUTs, BRAM: 36Kb, MUL: 25x18-bit) (implemented using multiplier/DSP slices or logic (LUTs) slices).

Implementing the audio SoC design we managed to use the complete design and verification flow of SysPy. Using the supported hw/sw co-simulation environment it was possible to build simulation models for software and hardware modules of the design and define critical parameters of the system before writing software or RTL code. Automatic compilation of a Linux kernel, along with user-defined application, provided easy-to-use communication solutions, such as the FTP protocol. The embedded O/S also facilitates the usage of system scripts to control data I/O and processing tasks in the design, allowing the user to control the SoC from the Linux command line.

8.5 Usability evaluation of SysPy

In order to establish a quantitative approach and assess how the developed methodology can help a designer during the design flow of a SoC, we adopted the metrics used in the BDTi evaluation methodology [46] for high-level synthesis tools developed by Berkeley Design Technologies. The BDTi has been established as a benchmark to evaluate the performance of embedded and DSP processors. Except the basic benchmark, other flavors exist for evaluating processors for special applications, like video and real-time data processing. Another flavor of the benchmark, that is useful in our case, evaluates the capabilities of high-level synthesis tools. In Berkeley Design Technologies they developed this specific testbench because they recognized that development of design tools has not kept pace with the rapid growth of the capacity of FPGAs. With the development of the testbench the Berkeley team wanted to evaluate any existing FPGA high-level design tools and also trigger the development of more tools that can be used and design complex SoC devices in an effortless way.

From the available BDTi high-level synthesis benchmark we made use of the available metrics and evaluated our tool while using SysPy. We focused on the usability metrics and input was provided by three designers who were involved in the design of the biomolecular and the audio processing SoCs. The metrics results are presented in Table 8.3. Every metric in the testbench is assessed using one of the following scores: "Excellent", "Very Good", "Good", "Fair", "Poor". The following list provides description about the evaluation metrics that we used:

- Out-of-Box Experience: Ease of installing the tool in a Debian Linux O/S.
- Ease of Use: Assessment about how easy was to use the tool in terms of productivity and bug-free operation.
- Completeness of Capabilities: Assessment to check if the features of the tool are adequate to design a processor-centric SoC in FPGA.
- Quality of Documentation and Support: Evaluation of the documentation supplied with SysPy.
- Learning to Use the Tool: Ease of learning to efficiently use the high-level synthesis tool.
- First Compiling Version: Evaluate the effort required to design initial functional design of a SoC.
- Final Optimized Version: Evaluate the effort required to design the final version of the SoC.
- Platform Infrastructure Development: Assess any external tools supported by SysPy (e.g. Xilinx physical integration tools) to ease physical implementation in silicon.

According to the results presented in Table 8.3 the supported features were well evaluated and the designers can get in a descent amount of time an initial version of their design up and running in the FPGA. Also the "Platform Infrastructure Development" was also well evaluated, since a number of auto-generated scripts ease physical development of a SoC and

Out-of-Box Experience	**Ease of Use**	**Completeness of Capabilities**	**Quality of Documentation and Support**
Good Fair Fair	Poor Good Good	Fair Good Good	Poor Poor Fair

Learning to Use the Tool	**First Compiling Version**	**Final Optimized Version**	**Platform Infrastructure Development**
Fair Fair Fair	Good Good Fair	Good Fair Fair	Very Good Fair Good

Table 8.3: Usability metrics, according to the BDTi benchmark, provided by three different designers.

also the supported HAL and Linux O/S based interfaces helped the designers to use their SoCs in real world applications, where data could easily transferred from a host PC to the FPGA board and vice versa. The designers easily learned how to create their own examples and designs with the tool and also appreciated the way high-level descriptions and pure RTL-like coding are used within SysPy, by utilizing and modifying the existing component and function libraries and the function handlers to instantiate already existing blocks in an effortless way (Figure 4.2). SysPy scored poor results in documentation, since at the time when the SoCs were designed, there was no full documentation and code examples in place. This is the reason why the tool also scored poor results in the "Ease of Use" metric, since without full documentation it was some time hard for the designers to understand the syntax and the design flow they should follow.

The feedback that we got from the preliminary usability evaluation was very useful and helped us improve several aspects in the adopted design flow, especially in the simulation flow. We applied improvements on the way an input signal sequence and timing delays are declared. We also put a lot of effort to improve the way abstract algorithmic models are connected in an RTL description (Figure 3.2 and Code Examples B.4, B.5 and B.6). We also applied changes to the HAL interface and the way objects are initialized and used to exchange data with the FPGA board over the Ethernet channel (Code Example 7.2). In general during development we always tried to deliver a tool where the supported Python syntax is compatible with the most common coding styles, so the hardware descriptions in Python can be easily read by people with little experience in hardware design. On the other hand the supported syntax should be powerful enough to used within SysPy to generate RTL and functional testbenches and also deliver synthesizable VHDL RTL code.

The usability metrics results also reveal another aspect of conducting research in the area of high-level hardware design tools. Although new methods have to be introduced in the area of design abstraction so that complex SoCs can be designed in a block-oriented manner and verified in an faster way, these new methods must remain consistent to the strict design rules defined by the already existing ecosystem of digital design tools. Abstraction methods must be introduced step by step and the final outcome must always be pure synthesizable HDL code that can be used as an input to the FPGA logic synthesis and physical design tools. The adopted methodologies must also be consistent to the design rules of synchronous datapath designs and must use a syntax/language to express a design that is familiar to other engineers. The new methodologies must also adopt and use any already existing and well defined flow in digital design, as we do in SysPy, e.g. Tcl scripting, VCD files, Matlab-like algorithmic coding, GCC compilers, IP-XACT models etc.

Chapter 9

Conclusions

In this dissertation we proved the thesis that a popular language, mainly used by the software development community, can be used for the hw/sw co-design and verification of SoCs at an early design phase. The presentation also of three complicated processor-centric design examples show how the developed tool combines all the necessary steps to support a hw/sw co-design flow, even for non-trivial SoCs where a Linux O/S is used to control data processing.

9.1 Summary of contributions

Although some well established platforms exist for high-level digital modeling and verification, like SystemC, there is still not a single integrated environment that can be used: a) to model and co-simulate the hw/sw architecture of a processor-centric SoC and b) perform all the steps needed to implement the design using FPGA devices. Other tools that exploit Python's unique text processing features for digital design have been developed in academia. None of them support the full design flow, from simulating and verifying a design down to providing the means to handle FPGA synthesis tools. Some of them focus only on specific steps of the design flow, like PyMTL [58] which supports a very fast RTL simulation engine. To our knowledge, and according to the comparison performed against other related tools (Table 2.1), SysPy is the first available tool that can handle the complete design flow of an FPGA processor-centric design (RTL/functional or algorithmic/high-level simulation - HDL implementation - software implementation - FPGA tools scripting - interface implemented design in silicon) After reviewing the tools and the methods already available we believe that in the area of processor-centric design and verification flow more research is needed to develop tools that can take advantage of at least freely available processor cores (in the form of soft or hardwired cores) and provide suitable syntax and methods to help software and hardware designers to design and implement a system using modern FPGA devices. The required methods should provide a high-level of abstraction, especially for these software or hardware modules of a system not yet specified, so that a designer can use hw/sw sw models and easily alter system parameters in the hw/sw design space during simulation. This will help a designer to make correct architectural choices regarding functionality and timing performance of a system.

The usability metrics results that we performed for SysPy also revealed another aspect of conducting research in the area of high-level hardware design tools. Although new methods have to be introduced in the area of design abstraction so that complex SoCs can be designed in a block-oriented manner and verified in an faster way, these new methods must remain consistent to the strict design rules defined by the already existing ecosystem of digital design tools. Abstraction methods must be introduced step by step and the final outcome must always be pure synthesizable HDL code that can be used as an input to the FPGA logic synthesis and physical design tools. The adopted methodologies must also be consistent to the design rules of synchronous datapath designs and must use a syntax/language to express a design that is familiar to other engineers. The new methodologies must also adopt and use any already existing and well defined flow in digital design, as we do in SysPy, e.g. Tcl scripting, VCD files, Matlab-like algorithmic coding, GCC compilers, IP-XACT models etc.

In the methodology developed we used a popular and easy to read and write language like Python to address the area of processor-centric design and verification using FPGA devices. Our main focus from the design point of view was to ease the integration of freely available processor cores in a design, by providing all the necessary scripting and HDL generation tools. From the verification point of view we utilized the SciPy powerful arithmetic package available in Python and developed a way to combine arithmetic models along with digital HDL-like descriptions for system-level verification. Across all design and simulation steps in our flow we use Python structures/syntax, and not any custom-defined syntax, to describe the datapath of the SoC and the related simulation models. This is very important since the main target group of a high-level design tool are engineers and scientists who have little or no experience at all in digital hardware design. The goal in this case is to deliver a design tool where a high-level interface can be used to:

- Design a SoC in a block-oriented way, using IP cores in RTL or netlist format and apply minimum effort to include any required digital glue logic between the blocks.
- Support a high-level verification flow, where Python descriptions can be used along with Matlab-like or C descriptions to simulate a digital block either at a functional/algorithmic or in a cycle-accurate Register-Transfer-Level.
- Automated generation of FPGA synthesizable VHDL descriptions
- Provide tools to interface a SoC design after its implementation, in the form of software components running in parallel on the processor-core in the SoC and in the host PC connected to the SoC.
- Ease the use of digital synthesis and physical implementation tools for FPGAs, by auto-generating synthesis and compilation scripts.

All five items in the previous list are critical for modern SoC designs. We believe that the methods developed in this thesis and SysPy provide an integrated environment and utilizes Python best programming practices like object oriented programming, text processing features, associative lists and ready-to-use numerical libraries, in order to design, verify, implement and test a processor-centric SoC. The tool supports the most common and basic Python syntax and also any third-party tool or file format used or called within SysPy is adopted by the EDA industry and the software community tools, like Tcl, VCD, Linux OS, SciPy and gcc compiler. In this way our tool was implemented on top of already existing, popular and standard tools used in a hw/sw co-design flow and we do not introduce any new, custom defined and "exotic" practices that would be uncommon.

Although the basic syntax for describing digital models in Python and all the required lexical analysis tools were developed during the initial development steps of the thesis, many other features have been developed or improved during the implementation of the SoC design case studies that we presented in Chapters 6, 7 and 8. In the following list we summarize the main innovative features of the tool developed and tested in each of the design examples:

Using scripting languages for hardware/software co-design

- Image processing SoC
  - Automatic compilation of C software code.
  - Use of pre-synthesized, netlist block in structural descriptions.
  - Use of Python functions to parameterize, instantiate and connect a block in a structural description.
- Biocomputing SoC
  - Utilize fast data transmission Ethernet channel between the FPGA and the host PC, so that the FPGA is used as a co-processor unit that stores and retrieve data to and from the connected PC.
  - Utilize large SDRAM memory resources connected to the FPGA for fast data storage.
  - Parse and use special arguments during a block's instantiation that parameterize an RTL design, e.g. XML files, using the implemented function handlers.
  - Provide Hardware Abstraction Layer (HAL) to interface the implemented SoC design.
- Audio processing SoC
  - Develop of O/S-centric applications using Linux embedded O/S.
  - Simulation mechanism in RTL and functional/algorithmic level for specifying the architecture of a SoC early in the design phase.
  - Hw/sw co-simulation using algorithmic Matlab-like models and with C functions along with RTL hardware descriptions.
  - Provide simulation results in the form of VCD files, compatible with popular RTL simulation tools (e.g. Modelsim).

Using a 32-bit CPU connected to custom hardwired co-processors, we proved with the results reported in Chapter 7 (Table 7.4) that using the processor-centric design flow adopted in SysPy we easily outperformed the simulation performance of popular software tools used for simulating biomolecular reaction networks. We also presented in Chapter 7 a very structured way for building an object oriented interface environment and easily parse and exchange, between an FPGA board and a host PC, any kind of data files over an Ethernet connection. In addition we demonstrated how contents of a data file can be used to parameterize and instantiate an IP core in a higher-level compared to using only generic parameters in VHDL. Python scripts parse these special parameters and re-arrange the structure of an IP core, e.g. modify the number of memory blocks, change fixed-point number representation etc.

O/S software development features added to SysPy, by configuring and compiling a Linux embedded O/S along with user application files, demonstrated the ability to easily design a system capable to act as a co-processor attached to a host PC, connected in an Ethernet network. Since the existence of the Linux O/S easily adds IP based Ethernet addressing, the FPGA board could also act as a data processing FTP client. Development also of the simulation environment added value to SysPy's FPGA SoC implementation features, since a designer can get a first good timing estimation of a system's data processing capabilities and also easily explore different architectural options.

9.2 Proposed future research

According to the experience that we acquired through the design examples implemented using SysPy and also from the user feedback we can summarize some points towards the improvement and further enhancement of the design tool. We can divide these points into three main categories of improvements: a) add new design and verification features, b) enhance the tool's already existing libraries with new digital blocks and models and c) use the tool to provide more real world processor-centric SoC design examples.

A good design feature improvement would be to support ASIC implementations by generating HDL code compatible with popular ASIC synthesizers, like Cadence RTL Compiler and Synopsys Design Compiler. In this case ready-to-use pre-synthesized netlist components could be structurally connected in a top-level Python description, in the same way we are using FPGA synthesized blocks. Also memory blocks should be mapped to RAM structures design using standard cell libraries and not BRAMs as we use in FPGAs.

The tool's function and component library can also be enhanced with more modules and models. Especially the function library must be enhanced by also adding more function handlers and in this way automate instantiation of complex logic and arithmetic blocks, like the function handler used in Figure 7.6. Good candidates blocks are DSP related modules used in audio and video processing. More processor cores can also be supported in SysPy supporting other than Xilinx FPGA devices, like the LatticeMico32 processor core by Lattice. Although Xilinx is the largest FPGA provider, it would be a very good feature to support FPGA families by different vendors. Candidate processor cores must support Linux or other high-end O/S to ease handling of many different communication protocols and device drivers.

More designs must also be developed to present the capabilities of our methodology to ease the design flow of processor-centric systems. The design of SoCs that require high performance computing resources, like video and network processors, would emphasize on SysPy's features to couple together processor cores and custom arithmetic blocks in the FPGA silicon. The way Python is used to parse any form of data files and use them during simulation or implementation of a SoC can also be exercised in these new design examples. Faster data communication channels can also be supported, like PCI Express or Gigabit Ethernet, so data can be exchanged faster between host PCs and the FPGA board or between different FPGA boards. Also better processor-custom block interface logic should be implemented, to improve data throughput, provided that the processor is able to handle and process these higher data throughputs.

In order to make the tool widely accessible and also to get feedback from users we provide access to the tool's source code through a public Git code repository. The repository is hosted in GitHub [89], which is one of the largest online code repositories. Code examples are also provided in the repository, along with information on how to setup the tool and run the examples. SysPy is delivered as an open source design tool in order to continuously improve its features and also increase the number of people that use it to design processor-centric embedded SoCs for FPGAs based on Python component descriptions.

Appendix A

Installing SysPy

A.1 SysPy setup in Debian Linux

SysPy has been developed and tested under a Debian Linux operating system. In order to resolve path installation issues for the various tools used within SysPy, we decided to use a configuration file, where all the required system paths info is included.

The following tools must be already installed in a system prior to SysPy's usage:

- *Python* v2.6 or v2.7
  - `SciPy` Python package v0.14.0
  - `matplotlib` Python package v1.4.2
- *Tcl* v8.5 or greater
- *avr-gcc* C v1.8 compiler for the AVR architecture
- *sparc-elf-gcc* compiler for the Leon3 architecture
- *or1k-gcc* for the OpenRic architecture
- *Xilinx ISE* v12 or greater

`SciPy` [7] and `matplotlib` [5] are Python packages which can automatically be downloaded and installed in Python. The two packages are required for SoC verification. The *Tcl* compiler is available under all Linux distributions, while the *avr-gcc* [3], the *sparc-elf-gcc*

[4] and the *or1k-gcc* [6] C cross-compilers have to be downloaded and installed in the Linux environment. If an O/S based application has to be developed for the Leon3 architecture the *Snapgear* embedded Linux [84] tool chain has to be installed. The *Xilinx ISE* FPGA tools have to be installed by the user if auto-generated by SysPy Tcl scripts are going to be used in command line to execute the FPGA implementation steps. Otherwise the user can manually create a design project in *ISE*, using the generated HDL and constraints files from the SysPy working directory. All required tools used by SysPy are also presented in Figure 1.1.

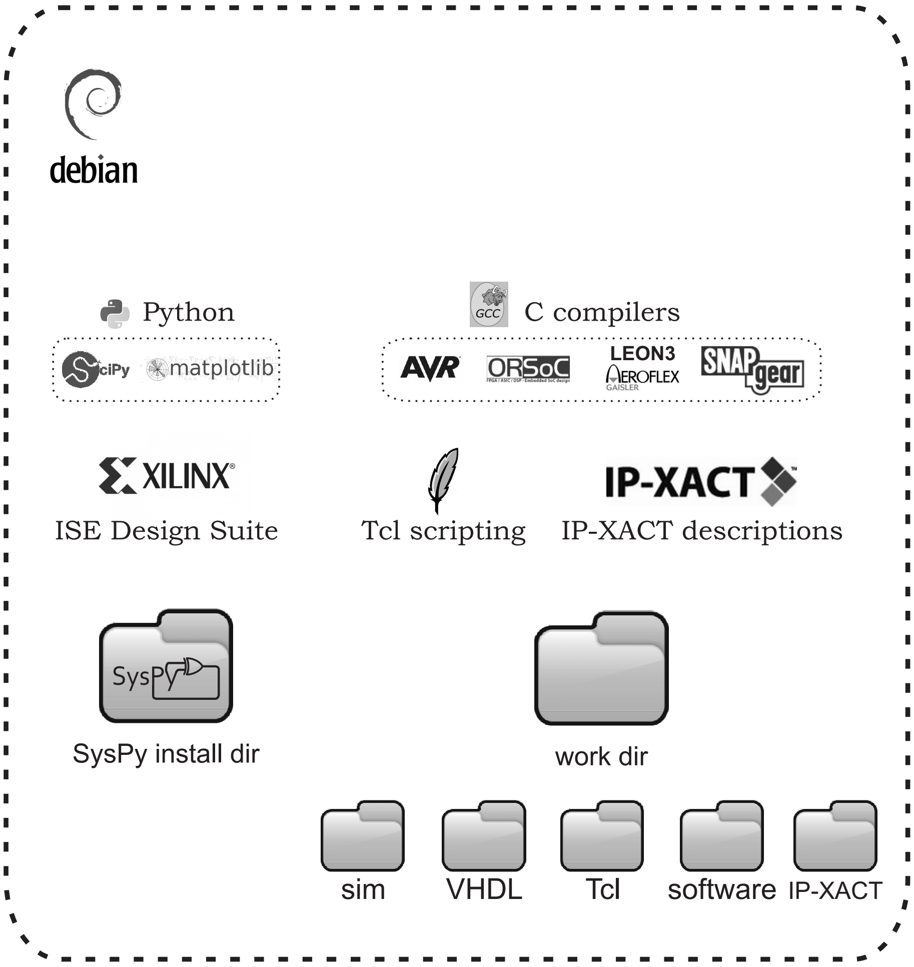


Figure 1.1: Design tools used by SysPy installed under Debian Linux.

The working directory can be placed by the user in the desired path. A setup initialization file (*.init) is used in SysPy, where the user can define the installation paths of the required compilers. A setup script is run automatically by SysPy to resolve these paths and also register the user-provided Python top-module to the main Python path, so that this modules can be directly imported by SysPy. Under the work directory five subfolders are generated, containing a) the VHDL generated files, b) the compiled executable software files, c) the VCD simulation files, d) Tcl script files that can be used along with the ISE design tools and e) IP-XACT [15] description of Python described blocks, which can be used along with the generated VHDL files for easy integration and reuse in other digital design and verification tools. With the use of the initialization file and of the setup script, SysPy is used as a standalone Python package which references all the user-provided tools in order to compile all the required software, simulate a SoC design, generate the related HDL files and execute the required Tcl scripts for FPGA implementation.

A.2 Synthesis options

Synthesis options	**Possible values**
FPGA_DEV	spartan3 — virtex2p — virtex5
FSM_STYLE	lut — bram
FSM_ENCODING	auto — one-hot — compact — sequential — gray — johnson — speed
MULT_STYLE	auto — block — lut — pipe_block
RAM_STYLE	auto — block — distributed
RESOURCE_SHARING	yes — no

Table 1.1: Synthesis options and possible values.

- FPGA_DEV: define one of the supported FPGA device families in SysPy
- FSM_STYLE: logic resources used for the implementation of state machines Faster implementation can be obtained by using BRAMs, when available

- FSM_ENCODING: state machine coding style
- MULT_STYLE: defines the way multipliers block are implemented. DSP48 [49] macro blocks can be used in Virtex-5 devices for fast multiplier implementations
- RAM_STYLE: defines the way RAM memory will be implemented, either by using BRAM blocks or in a distributed way using CLB block
- RESOURCE_SHARING: resource sharing of arithmetic operators

A.3 Snapgear Linux kernel parameters

Kernel compilation parameters	**Default values**
CPU_FREQ	100MHz(default)
ETHERNET_EN	yes — no
FPU_EN	yes — no
MUL_DIV_EN	yes — no
SDRAM_FREQ	100MHz(default)
SDRAM_SIZE	256MB(default)
SERIAL_BAUDRATE	38400bps(default)
TCPIP_EN	yes — no
UART_EN	yes — no

Table 1.2: Snapgear Linux kernel compilation parameters and default values.

- CPU_FREQ: processor's clock frequency in MHz

- ETHERNET_EN: include Ethernet controller driver
- FPU_EN: enable floating-point libraries compilation
- MUL_DIV_EN: MUL and DIV operations hardware support
- SDRAM_FREQ: SDRAM memory clock frequency in MHz
- SDRAM_SIZE: SDRAM memory clock frequency in MB
- SERIAL_BAUDRATE: serial communication baud rate in bps
- TCPIP_EN: enable TCP/IP stack compilation
- UART_EN: include UART serial communication drivers

Appendix B

Extended code examples

B.1 Python examples

B.1.1 Arithmetic simulation model

Code Example B.1: Linear regression class model (part 1).

```python
import math
from _fp_sign_to_bin import *
from scipy import signal, stats
from random import *
import matplotlib.pyplot as plt

class linearRegressionSimFunctions:
  slopeOut = 0
  interceptOut = 0
  stdErrOut = 0
  dataCounter = 0
  # data input buffer
  dataFileArray = []

  # regression variables
  slope = 0
  intercept = 0
  r_value = 0
  p_value = 0
  st_err = 0

  # input FIFO of the arithmetic block
  FIFOSize = 10
  FIFOArray = FIFOSize * [0]
  fpNotation = ''
  fpDecimalSize = 0
  dataFileName = ''
```

- line 2: import the SysPy provided function for converting fixed-point numbers to binary format
- line 3: import the SciPy package
- line 5: import the `matplotlib` package required for creating data plots
- line 7: declare the class used to model the linear regression block
- line 13: create array to hold file data
- lines 16-20: initialize all the model related variables
- line 24: array to model the input FIFO buffer of the arithmetic block

Code Example B.2: Linear regression class model (part 2).

```
def init(self):
  # Open the data file
  dataFile = open(self.dataFileName, 'r')

  # Read file data into an array
  self.dataFileArray = dataFile.read()
  self.dataFileArray = self.dataFileArray[:(len(self.dataFileArray) - 1)]
  self.dataFileArray = self.dataFileArray.split("\n")
  # Extract decimal notation form fp notation
  i = self.fpNotation.find('.')
  self.fpDecimalSize = int(self.fpNotation[(i+1):].replace('"', ''))

def writeFIFO(self):
  self.FIFOArray[self.dataCounter] = int(fp_sign_to_bin(
  self.dataFileArray[self.dataCounter], self.fpNotation), 2)
  # Store data values to the arithmetic block input FIFO
  if (self.FIFOArray[self.dataCounter] < 255):
    # Positive values
    self.FIFOArray[self.dataCounter] = float(self.FIFOArray[self.dataCounter]) /
                                        float(pow(2, self.fpDecimalSize))
  else:
    #Negative values
    self.FIFOArray[self.dataCounter] = -1.0 * float(512 -
    self.FIFOArray[self.dataCounter]) / float(pow(2, self.fpDecimalSize))
  # Increase FIFO counter
  self.dataCounter = self.dataCounter + 1

def startRegression(self):
  xAxe = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  # Calculate the regression algorithm
  self.slope, self.intercept, self.r_value, self.p_value, self.st_err =
  stats.linregress(xAxe, self.FIFOArray)
def plotRegressionResults(self):
  # Open file to store regression results
  simDataFile = open("./sim/simData.txt", 'w')
  # Convert the calculated slope parameter to the provided fp notation
  slope = int(fp_sign_to_bin(self.slope, self.fpNotation), 2)
  if (slope < 255):
    slope = float(slope) / float(pow(2, self.fpDecimalSize))
  else:
    slope = -1.0 * float(512 - self.slope) / float(pow(2, self.fpDecimalSize))
  # Convert the calculated intercept parameter to the provided fp notation
  intercept = int(fp_sign_to_bin(self.intercept, self.fpNotation), 2)
  if (intercept < 255):
    intercept = float(intercept) / float(pow(2, self.fpDecimalSize))
  else:
    intercept = -1.0 * float(512 - self.intercept) / float(pow(2, self.fpDecimalSize))
```

- line 1: class initialization function
- line 3: open data file
- lines 6-8: copy file data into the `dataFileArray[]`
- lines 10-11: extract the number of decimal points from the supported notation
- line 13: function to store data into the FIFO memory of the arithmetic block
- line 14-15: converting decimal values to the corresponding fixed-point form (`fp_sign_to_bin()`)
- lines 17-24: distinguish between positive and negative data values
- line 28: function to perform the linear regression algorithm
- line 33: function to plot regression results
- lines 37-47: convert the linear equation parameters (slope and intercept) to the defined fixed-point notation

Code Example B.3: Linear regression class model (part 3).

```
1       # Convert the calculated standard error parameter to the provided fp notation
2       st_err = int(fp_sign_to_bin(self.st_err, self.fpNotation), 2)
3       if (st_err < 255):
4         st_err = float(st_err) / float(pow(2, self.fpDecimalSize))
5       else:
6         st_err = -1.0 * float(512 - st_err) / float(pow(2, self.fpDecimalSize))
7
8       # Write to file the regression results
9       simDataFile.write("Slope: "+ str(slope) + "\n")
10      simDataFile.write("Interception: "+ str(intercept) + "\n")
11      simDataFile.write("Standard error: "+ str(st_err) + "\n")
12
13      simDataFile.close()
14
15      FIFOArrayFloat = []
16      for i in self.FIFOArray:
17        FIFOArrayFloat.append(float(i))
18
19      FIFOArrayEstimatedHw = []
20      FIFOArrayEstimatedSw = []
21      for i in range(0, 10):
22        FIFOArrayEstimatedHw.append(slope * float(i) + intercept)
23
24      # Plot the data points alogn with the claculated regression linear curve
25      plt.plot(range(0, 10), FIFOArrayFloat)
26      plt.plot(range(0, 10), FIFOArrayEstimatedHw, 'r')
27
28      plt.show()
29
30    # Return slope result to the testbench
31    def returnSlope(self):
32      return int(fp_sign_to_bin(self.slope, self.fpNotation), 2)
33
34    # Return intercept result to the testbench
35    def returnIntercept(self):
36      return int(fp_sign_to_bin(self.intercept, self.fpNotation), 2)
```

- lines 1-36:

B.2 Testbench example for the audio processing SoC

Code Example B.4: Testbench for the audio processing SoC (part 1).

```
import SysPy_setup
import _toVHDL
import funcs._beh_sim
from filterSimfunctions import *

def fir_sim():
  numOfSamples = 100

  def proc_1(clk, rst):
    if (rst == 1):
      state = 0
      sim_time = funcs._beh_sim.simTime()
      output_fifo_ready = 0
      data_counter = 0

      # Read values from *.wav file
      SimObj.musicFileName = "music_file.wav"

      # Define filter parameters (# of taps, sampling
      # and cutoff freq. in Hz) Filter0
      SimObj.FilterDict["Filter0"]['N'] = 30
      SimObj.FilterDict["Filter0"]["fs"] = 16000.0
      SimObj.FilterDict["Filter0"]["fc0"] = 250.0
      SimObj.FilterDict["Filter0"]["fc1"] = 1000.0

      # Define filter parameters (# of taps, sampling
      # and cutoff freq. in Hz) Filter1
      SimObj.FilterDict["Filter1"]['N'] = 30
      SimObj.FilterDict["Filter1"]["fs"] = 16000.0
      SimObj.FilterDict["Filter1"]["fc0"] = 1600.0
      SimObj.FilterDict["Filter1"]["fc1"] = 4800.0

      # Define filter parameters (# of taps, sampling
      # and cutoff freq. in Hz) Filter2
      SimObj.FilterDict["Filter2"]['N'] = 30
      SimObj.FilterDict["Filter2"]["fs"] = 16000.0
      SimObj.FilterDict["Filter2"]["fc0"] = 3000.0
      SimObj.FilterDict["Filter2"]["fc1"] = 5000.0

      # Define filter parameters (# of taps, sampling
      # and cutoff freq. in Hz) Filter3
      SimObj.FilterDict["Filter3"]['N'] = 30
      SimObj.FilterDict["Filter3"]["fs"] = 16000.0
      SimObj.FilterDict["Filter3"]["fc0"] = 5000.0
      SimObj.FilterDict["Filter3"]["fc1"] = 7999.0
```

- lines 3-4: import the filter simulation mode (`filterSimfunctions`) and SysPy's timing simulation library (`_beh_sim`)
- line 7: define the number of audio samples to be processed
- line 9: state transition, sequential part of the state machine
- lines 11-14: initialize state and output signals. `sim_time` is used to observe simulation time
- line 17: open the audio file
- lines 19-45: define number of taps, sampling frequency and cutoff frequencies for the four implemented filters

Code Example B.5: Testbench for the audio processing SoC (part 2).

```
    # Initialize simulation object
    SimObj.init()

    if (funcs._beh_sim.rising_edge2("clk") == True):
      sim_time = funcs._beh_sim.simTime()
      if (start == 1):
        if (state == 0):
          state = 1
        elif (state == 1):
          input_fifo_ready = SimObj.inputFifoReady()
          if (input_fifo_ready == 1):
            state = 2
          else:
            state = 1
        elif (state == 2):
          if (SimObj.outputFifoCounter("Filter0") == numOfSamples):
            state = 3
          else:
            state = 1
        elif (state == 3):
          SimObj.printOutputFifoData()
          SimObj.plotSignalWaveforms()
          state = 3
          funcs._beh_sim.endSimulation()



  def proc_2(state):
    if (state == 2):
      filt_out0 = SimObj.Fir(1, "Filter0")
      filt_out1 = SimObj.Fir(1, "Filter1")
      filt_out2 = SimObj.Fir(1, "Filter2")
      filt_out3 = SimObj.Fir(1, "Filter3")
      data_counter = SimObj.outputFifoCounter("Filter0")
    elif (state == 3):
      output_fifo_ready = 1
    else:
      filt_out0 = SimObj.PreserveState("Filter0")
      filt_out1 = SimObj.PreserveState("Filter1")
      filt_out2 = SimObj.PreserveState("Filter2")
      filt_out3 = SimObj.PreserveState("Filter3")

generics = {}
# Simulation parameters (50ms duration, 1ns time step)
attributes = {"sign": '-', "simulation": [5000000, 1, "ns"], "FPGA_DEV": "Virtex5"}
```

- line 2: initialize a simulation object using the provided class
- line 4: begin state transition section
- line 11: check if there is free space in the connected input FIFO memory. `inputFifoReady()` function models asynchronous communication between the processor and the filter bank
- line 16: function `outputFifoCounter()` is used to provide the number of processed audio samples existing in the output FIFO memory
- line 21: function `printOutputFifoData()` is called to generate a text file with the values of the filtered samples
- line 22: function `plotSignalWaveforms()` is used to plot, using SciPy, the initial and the filtered audio signals
- line 24: function `endSimulation()` is used to terminate the simulation
- line 28: combinational part of the state machine
- lines 30-33: function `Fir()` used for each of the filters to push one audio sample to the filters' datapath, simulating in this way one processing cycle
- line 36: `output_fifo_ready` signal is asserted in state 3 to indicate that the requested number of audio samples have been filtered
- line 45: define simulation step and duration

Code Example B.6: Testbench for the audio processing SoC (part 3).

```
1 # Create a 50MHz clock sequence for 50ms, 50% duty cycle
2 clk_seq = []
3 clk = 0
4 for i in range(8, 50000000, 20):
5   clk = not clk
6   if (clk == True):
7     clk_seq.append([str(i), '1'])
8   else:
9     clk_seq.append([str(i), '0'])
10
11 # I/O and internal signal declaration
12 i_sigs0 = {'D': 'i', 'T': 'b', 'L': 1, 'N': ["rst", "clk", "start"]}
13 o_sigs0 = {'D': 'o', 'del': 0, 'T': 'b', 'L': [0, 5], 'N': "state"}
14 o_sigs1 = {'D': 'o', 'del': 7, 'T': 'b', 'L': [0, 16], 'N': ["filt_out3", "filt_out1",
15                                                            "filt_out2"]}
16 o_sigs2 = {'D': 'o', 'del': 15, 'T': 'b', 'L': [0, 16], 'N': "filt_out0"}
17 o_sigs3 = {'D': 'o', 'del': 0, 'T': 'b', 'L': 1, 'N': "input_fifo_ready"}
18 o_sigs4 = {'D': 'o', 'del': 0, 'T': 'b', 'L': [0, 31], 'N': "sim_time"}
19 o_sigs5 = {'D': 'o', 'del': 0, 'T': 'b', 'L': [0, 31], 'N': "data_counter"}
20 o_sigs6 = {'D': 'o', 'del': 0, 'T': 'b', 'L': 1, 'N': "output_fifo_ready"}
21
22 # Define values for input signals
23 sim_sigs0 = {'D': 'sim', 'T': 'b', 'L': 1, 'N': "rst", 'V': [['0', '1'], ['5', '0']]}
24 sim_sigs1 = {'D': 'sim', 'T': 'b', 'L': 1, 'N': "clk", 'V': clk_seq}
25 sim_sigs2 = {'D': 'sim', 'T': 'b', 'L': 1, 'N': "start", 'V': [['0', '0'], ['6', '1']]}
26
27 code = getsourcelines(fsm_sim)
28
29 _toVHDL.toVHDL("fir_sim", attributes, generics, i_sigs0, o_sigs0, o_sigs1, o_sigs2,
30                o_sigs3, o_sigs4, o_sigs5, o_sigs6, sim_sigs0, sim_sigs1, sim_sigs2, code)
```

- line 4-9: create a 50ms clock sequence with a 50MHz frequency and 50% duty cycle
- lines 12-20: define I/O signals
- line 14: define data path delay to 7ns for filters 1, 2 and 3
- line 16: define data path delay to 15ns for filter 0
- line 23: define `rst` signal input sequence (assert reset for the first 5ns)
- line 24: assign the 50MHz clock sequence to the `clk` signal
- line 25: define `start` signal input sequence (assert the signal after 6ns)

- line 29: call `to_VHDL()` to start simulation

B.3 HDL examples

Code Example B.7: Auto-generated VHDL description for the Python description in Code Example 8.2.

```
-- filt_SoC.vhd
-- Generated by SysPy
-- Mon Nov 17 15:29:42 2014

library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
library work;
entity filt_SoC is
        port (
    clk: in std_logic;
    rst: in std_logic;
    filt_in: in std_logic_vector(8 downto 0);
    filt_out: out std_logic_vector(18 downto 0));
end filt_SoC;

architecture filt_SoC_arch of filt_SoC is

-- Internal signals
--------------------------------------------------------------------------------
signal filt_out_int: std_logic_vector(18 downto 0);
--------------------------------------------------------------------------------

component fir_filt_s_comp
  generic (n ,m ,filt_param: integer);
  Port (
    clk: in std_logic;
    rst: in std_logic;
    filt_in: in std_logic_vector((n - 1) downto 0);
    filt_out: out std_logic_vector((filt_acc_bus(n, m) - 1) downto 0));
end component;

begin

  fir_filt_s_comp_U0: fir_filt_s_comp generic map(filt_param => "111110011000011010",
                                                  m => 2, n => 8)
  port map (
    clk => clk,
    rst => rst,
    filt_in => filt_in,
    filt_out => filt_out);

end filt_SoC_arch;
```

B.4 Tcl example

Code Example B.8: Auto-generated Tcl script used for synthesizing, placing/routing the design and generating the FPGA programming file (part 1).

```
# Define project directory
set compile_directory Leon3SoCTest

# Define all the custom related blocks and the top-level block of the processor
set hdl_files [ list \
  ../../SysPy_ver/Leon3_comps/ahbrom.vhd \
  ../../SysPy_ver/Leon3_comps/config.vhd \
  ../../SysPy_ver/Leon3_comps/leon3mp.vhd \
  ../../SysPy_ver/Leon3_comps/Leon3_wrapper.ucf \
  /home/test/SysPy/work/top_level_wrapper.vhd \
  /home/test/SysPy/work/demo_FSM.vhd \
]

# Timing and placement constraints
set constraints_file ../../SysPy_ver/Leon3_comps/Leon3_wrapper.ucf \

# Create project directory
if {![file isdirectory $compile_directory]} {
  file mkdir $compile_directory
}

# Define project name
project new Leon3SoCTest

# Define FPGA device and design attributes
project set family Virtex5
project set device xc5vlx110t
project set package ff1136
project set speed -1
project set top_level_module_type "HDL"
project set synthesis_tool "XST (VHDL/Verilog)"
project set simulator "ISim (VHDL/Verilog)"
project set "Preferred Language" "VHDL"
project set "Multiplier Style" Block
project set "FSM Style" lut

# Copy design files to the project
foreach filename $hdl_files {
  xfile add ../$filename -copy
  puts "Adding file $filename to the project."
}

# Set top-level module
project set top "rtl" "top_level_wrapper"
```

- lines 6-8: define processor and cache memory cores description files
- lines 10-11: include the top wrapper module and any custom peripheral modules
- lines 10-11: include the top wrapper module and any custom peripheral modules
- line 15: define timing and placement constraints file
- lines 26-28: set FPGA device family and package
- lines 31-35: set synthesis options
- lines 38-40: create design project and copy all the HDL files
- line 44: define top-level module

Code Example B.9: Auto generated Tcl script used for synthesizing, placing/routing the design and generating the FPGA programming file (part 2).

```
 1 # Create processor related libraries
 2 xfile add "../../../SysPy_ver/Leon3_comps/libs/eth/comp/ethcomp.vhd" −lib_vhdl eth
 3 xfile add "../../../SysPy_ver/Leon3_comps/libs/eth/core/grethc.vhd" −lib_vhdl eth
 4 xfile add "../../../SysPy_ver/Leon3_comps/libs/gaisler/jtag/ahbjtag_bsd.vhd" −lib_vhdl gaisler
 5 xfile add "../../../SysPy_ver/Leon3_comps/libs/gaisler/misc/ahbram.vhd" −lib_vhdl gaisler
 6 xfile add "../../../SysPy_ver/Leon3_comps/libs/gaisler/uart/ahbuart.vhd" −lib_vhdl gaisler
 7 xfile add "../../../SysPy_ver/Leon3_comps/libs/gaisler/uart/apbuart.vhd" −lib_vhdl gaisler
 8 .
 9 .
10 .
11
12 # Run synthesis, PR and generate the FPGA programming file
13 process run "Synthesize − XST"
14
15 process run "Generate Programming File"
```

- lines 2-7: add HDL files related to peripheral devices attached to the processor, e.g. UART, Ethernet and memory controllers etc.
- lines 13-15: execute in command line mode synthesis and PR processes and generate FPGA programming file

List of abbreviations

ADL (Architectural Description Language)

API (Application Programming Interface)

ASIC (Application Specific Integrated Circuits)

AMBA (Advanced Microcontroller Bus Architecture)

CLB (Configurable Logic Block)

CORDIC (COordinate Rotation DIgital Computer)

CPU (Central Processing Unit)

CSV (Comma Separated Value)

CU (Control Unit)

DCM (Digital Clock Manager)

DM (Direct Method)

DSP (Digital Signal Processing)

DSU (Debug Support Unit)

DUT (Design Under Test)

EDA (Electronic Design Automation)

FIFO (First In-First Out)

FPGA (Field Programmable Gate Array)

FRM (First Reaction Method)

FSM (Finite State Machine)

FT (Flags Table)

GCC (GNU Compiler Collection)

GUI (Graphical User Interface)

HAL (Hardware Abstraction Layer)

HDL (Hardware Description Language)

HLS (High-Level Synthesis)

IC (Integrated Circuit)

IP (Intellectual Property)

JTAG (Joint Test Action Group)

JVM (Java Virtual Machine)

LUT (Look Up Table)

MAC (Multiply Accumulate)

MB (Mega Bytes)

MIRIAM (Minimum Information Required In The Annotation of Models)

Mbps (Mega bits per second)

MinSoC (Minimal OpenRISC System on Chip)

MMU (Memory Management Unit)

MSIP (Multiple Simulations In Parallel)

MTU (Minimum Time Unit)

ODE (Ordinary Differential Equations)

OOP (Object Oriented Programming)

O/S (Operating System)

PE (Processing Element)

PLD (Programmable Logic Device)

PR (Place and Route)

RT (Reactions Table)

RTL (Register Transfer Level)

SBML (Systems Biology Markup Language)

SSA (Stochastic Simulation Algorithm)

SoC (System on Chip)

SSIP (Single Simulation In Parallel)

ST (Species Table)

SWIG (Simplified Wrapper and Interface Generator)

TCL (Tool Control Language)

UART (Universal Asynchronous Receiver Transmitter)

UML (Unified Modeling Language)

uC (microcontroller)

uP (microprocessor)

UUT (Unit Under Test)

UVM (Universal Verification Methodology)

VCD (Value Change Dump)

VT (Stoichiometry Table)

VHDL (Very high speed integrated circuit Hardware Description Language)

XST (Xilinx Synthesis Technology)

Bibliography

[1] *GCC, the GNU Compiler Collection.* http://gcc.gnu.org/.

[2] *Python Documentation (Python 2.7).* http://www.python.org.

[3] *AVR GCC library.* http://www.nongnu.org/avr-libc/, 2012.

[4] *Bare-C Cross-Compiler System for LEON3.* http://www.gaisler.com/index.php/ downloads/compilers, 2013.

[5] *Matplotlib 1.4.2 User's Guide.* http://matplotlib.org/contents.html, 2014.

[6] *OpenRisc GNU tool chain.* http://opencores.org/or1k/OpenRISC_GNU_tool_chain, 2014.

[7] *SciPy 0.14.0 Reference Guide.* http://docs.scipy.org/doc/, 2014.

[8] Scrapy documentation, 2014. http://scrapy.org.

[9] Accelera System Initiative, http://www.accellera.org/downloads/standards/. *SystemC verification 2.0*, 2014.

[10] GRMON: Debug monitor for Leon, 2012. http://www.gaisler.com.

[11] LEON3 Multiprocessing CPU Core, 2012. http://www.gaisler.com.

[12] Arduino embedded platform, 2014. http://www.arduino.cc.

[13] ARM. *AMBA specification v2.0.* http://www.arm.com, 1999.

[14] Aura SoC, note = https://github.com/aurabindo/aura-soc, year = 2015.

[15] Victor Berman. Standards: the P1685 IP-XACT IP metadata standard. *Design & Test of Computers, IEEE*, 23(4):316–317, 2006.

[16] Benjamin J. Bornstein, Sarah M. Keating, Akiya Jouraku, and Michael Hucka. LibSBML: an API library for SBML. *Bioinformatics*, 24(6):880–881, March 2008.

[17] Rainer Breitling, Robin A Donaldson, David R Gilbert, and Monika Heiner. Biomodel engineering–from structure to behavior. In *Transactions on Computational Systems Biology XII*, pages 1–12. Springer, 2010.

[18] J. Dean Brock, Rebecca F. Bruce, and Susan L. Reiser. Using Arduino for introductory programming courses. *J. Comput. Sci. Coll.*, 25(2):129–130, December 2009.

[19] Cadence Design Systems Inc. *Incisive Enterprise Simulator*, 2012.

[20] Brad Chapman and Jeffrey Chang. Biopython: Python tools for computational biology. *SIGBIO Newsl.*, 20(2):15–19, August 2000.

[21] Pinhong Chen, D.A. Kirkpatrick, and K. Keutzer. Scripting for EDA tools: a case study. In *Quality Electronic Design, 2001 International Symposium on*, pages 87 –93, 2001.

[22] PyCell Studio, 2012. http://www.ciranova.com.

[23] J. Decaluwe. MyHDL: a Python-based Hardware Description Language. *Linux Journal*, 2004(127):5–9, November 2004.

[24] Digilent Inc., http://www.digilent.com. *Xilinx University Program Virtex-II Pro Development System*, 2005.

[25] Digilent Inc., http://www.digilent.com. *Virtex-5 OpenSPARC Evaluation Platform (ML509)*, 2014.

[26] Rolf Drechsler, Christophe Chevallaz, Franco Fummi, Alan J Hu, Ronny Morad, Frank Schirrmeister, and Alex Goryachev. Future SoC verification methodology: UVM evolution or revolution? In *Proceedings of the conference on Design, Automation & Test in Europe*, page 372. European Design and Automation Association, 2014.

[27] Biomodels database - a database of annotated published models, 2012. http://www.ebi.ac.uk/biomodels-main/.

[28] Plex: A Lexical Analysis Module for Python, 2007. http://www.cosc.canterbury.ac.nz/greg.ewing/python/Plex.

[29] Raul Fajardo. *Minimal OpenRISC System on Chip.* http://opencores.org/project,minsoc, 2012.

[30] Hans Fangohr. A comparison of C, MATLAB, and Python as teaching languages in engineering. In Marian Bubak, Geert van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, volume 3039, pages 1210–1217. Springer Berlin, 2004.

[31] Ewing G. Plex: A Lexical Analysis Module for Python, 2007. http://www.cosc.canterbury.ac.nz/greg.ewing/python/Plex.

[32] W. Geirts G. Goossens, D. Lanneer and J. Van Praet. *Designing ASIPs in Multicore SoCs).* Synopsys Inc., 2014.

[33] Aeroflex Gaisler. *GRLIB IP Library Userõs Manual v1.3.7.* http://www.gaisler.com, 2014.

[34] Michael A. Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry A*, 104(9):1876–1889, 2000.

[35] D. T. Gillespie. Stochastic simulation of chemical kinetics. *Annu. Rev. Phys. Chem.*, 58:35–55, 2007.

[36] Daniel T. Gillespie. *Markov processes : an introduction for physical scientists.* Academic Press, Boston, San Diego, New York, 1992.

[37] R. C. Gonzalez and R. E. Woods (2nd Edition). *Digital Image Processing*, chapter Image Enhancement in the Spatial Domain. Prentice-Hall, 2002.

[38] GTKWave waveform viewer, 2012. http://gtkwave.sourceforge.net.

[39] P. Haglund, O. Mencer, W. Luk, and B. Tai. PyHDL: Hardware scripting with Python. In *Proc. International Conference on Field Programmable Logic (FPL)*, pages 1040–1043, 2003.

[40] O. G. Hazapis and E. S. Manolakos. Scalable FRM-SSA SoC design for the simulation of networks with thousands of biochemical reactions in real time. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pages 459–463, 2011.

[41] O.G. Hazapis, E. Logaras, and E.S. Manolakos. A soft IP core generating socs for the efficient stochastic simulation of large Biomolecular Networks using FPGAs. In *Proc. IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 77–80, 2012.

[42] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, and H. Kitano. The Systems Biology Markup Language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[43] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[44] IEEE, https://standards.ieee.org. *1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual*, 2011.

[45] Atmel Inc. *ATmega128 AVR 8-bit microcontroller.* http://www.atmel.com/, 2011.

[46] Berkeley Design Technology Inc. *High-Level Synthesis Tools for Xilinx FPGAs.* http://www.bdti.com, 2010.

[47] Xilinx Inc. *Virtex-5 Libraries Guide for HDL Designs.* http://www.xilinx.com/itp/xilinx9/books/docs/v5ldl/v5ldl.pdf.

[48] Xilinx Inc. *LogiCORE IP CORDIC v4.0.* http://www.xilinx.com, 2011.

[49] Xilinx Inc. *LogiCore IP DSP48 Macro.* http://www.xilinx.com, 2011.

[50] Xilinx Inc. *ISim user guide.* http://www.xilinx.com, 2012.

[51] Xilinx Inc. *LogiCORE IP FIFO Generator v9.3.* http://www.xilinx.com, 2012.

[52] Xilinx Inc. *ChipScope Pro software and cores user guide.* http://www.xilinx.com, 2014.

[53] Xilinx Inc. *Xilinx CORE Generator System.* http://www.xilinx.com/tools/coregen.htm, 2014.

[54] Keerthan Jaic and Melissa C Smith. Enhancing Hardware Design Flows with MyHDL. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 28–31. ACM, 2015.

[55] Wido Kruijtzer, Pieter van der Wolf, Erwin de Kock, Jan Stuyt, Wolfgang Ecker, Albrecht Mayer, Serge Hustin, Christophe Amerijckx, Serge de Paoli, and Emmanuel Vaumorin. Industrial IP integration flows based on IP-XACT standards. In *Design, Automation and Test in Europe, 2008. DATE'08*, pages 32–37. IEEE, 2008.

[56] Damjan Lampret. *OpenRISC1200 IP core specification v0.7.* http://opencores.org, 2001.

[57] Chen Li, Marco Donizelli, Nicolas Rodriguez, Harish Dharuri, Lukas Endler, Vijayalakshmi Chelliah, Lu Li, Enuo He, Arnaud Henry, Melanie I Stefan, et al. BioModels Database: An enhanced, curated and annotated resource for published quantitative kinetic models. *BMC systems biology*, 4(1):92, 2010.

[58] Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 280–292. IEEE, 2014.

[59] Derek Matthew Lockhart. *Constructing Vertically Integrated Hardware Design Methodologies Using Embedded Domain-Specific Languages And Just-In-Time Optimization.* PhD thesis, Cornell University, 2015.

[60] E. Logaras and E. S. Manolakos. SysPy: using Python for processor-centric SoC design. In *Proc. IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 762–765, 2010.

[61] Evangelos Logaras, Orsalia G. Hazapis, and Elias S. Manolakos. Python to accelerate embedded SoC design: A case study for systems biology. *ACM Trans. Embed. Comput. Syst.*, 13(4):84:1–84:25, March 2014.

[62] K. Jarrod M. and M. Aivazis. Python for Scientists and Engineers. *Computing in Science Eng.*, 13(2):9 –12, March 2011.

[63] A. Mashtizadeh. PHDL: A Python hardware design framework. Master's thesis, ECE Dept. MIT, 2007.

[64] Wes McKinney. Data structures for statistical computing in Python. In *Proc. 9th Python Sci. Conf*, pages 51–56, 2010.

[65] O. Mencer, M. Morf, and M. J. Flynn. PAM-Blox: High performance FPGA design for adaptive computing. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, pages 167–174, 1998.

[66] ModelSim HDL simulator, 2012. http://model.com.

[67] U. Meyer-Base, A. Meyer-Base, and W. Hilberg. COordinate Rotation DIgital Computer (CORDIC) synthesis for FPGA. In ReinerW. Hartenstein and MichalZ. Servνıt, editors, *Field-Programmable Logic Architectures, Synthesis and Applications*, volume 849 of *Lecture Notes in Computer Science*, pages 397–408. Springer Berlin Heidelberg, 1994.

[68] Prabhat Mishra and Nikil Dutt. Architecture description languages for programmable embedded systems. *IEE Proceedings-Computers and Digital Techniques*, 152(3):285–297, 2005.

[69] Chris J. Myers. *Engineering Genetic Circuits*. Chapman and Hall/CRC mathematical & computational biology series. CRC Press, 2009.

[70] Chris J. Myers, Nathan Barker, Kevin Jones, Hiroyuki Kuwahara, Curtis Madsen, and Nam-Phuong D. Nguyen. iBioSim. *Bioinformatics*, 25(21):2848–2849, November 2009.

[71] The MyHDL manual, 2015. "http://www.myhdl.org.

[72] Nicolas Le Novere, Andrew Finney, Michael Hucka, Upinder S Bhalla, Fabien Campagne, Julio Collado-Vides, Edmund J Crampin, Matt Halstead, Edda Klipp, Pedro Mendes, et al. Minimum information requested in the annotation of biochemical models (MIRIAM). *Nature biotechnology*, 23(12):1509–1515, 2005.

[73] Open source hardware IP-cores. http://opencores.org.

[74] OpenCores. *WISHBONE B4 System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores.* http://opencores.org/project,minsoc, 2010.

[75] OpenCores. *MinSoC subversion repository, bin2init.py script.* http://opencores.org/websvn,listing?repname=minsoc, 2013.

[76] Atmel Mega128 processor core, 2009. http://opencores.org/project,avr_core.

[77] Jingzhao Ou and Viktor K. Prasanna. PyGen: A Matlab/Simulink based tool for synthesizing parameterized and energy efficient designs using FPGAs. In *Proc. International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 47–56, 2004.

[78] Eleftherios Ouzounoglou, Dimitrios Kalamatianos, Evangelia Emmanouilidou, Maria Xilouri, Leonidas Stefanis, Kostas Vekrellis, and Elias S Manolakos. In silico modeling of the effects of alpha-synuclein oligomerization on dopaminergic neuronal homeostasis. *BMC systems biology*, 8(1):54, 2014.

[79] Mario Pineda-Krch. GillespieSSA: Implementing the Gillespie Stochastic Simulation Algorithm in R. *Journal of Statistical Software*, 25(12):1–18, 4 2008.

[80] Muruhan Rathinam, Linda R. Petzold, Yang Cao, and Daniel T. Gillespie. Stiffness in stochastic chemically reacting systems: The implicit tau-leaping method. *The Journal of Chemical Physics*, 119(24):12784–12794, 2003.

[81] Sharon Rosenberg and Kathleen Meade. *A practical guide to adopting the Universal Verification Methodology (UVM).* Cadence Design Systems, 2013.

[82] Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn. A Hardware Abstraction Layer in Java. *ACM Transactions on Embedded Computing Systems*, 10(4):1–40, November 2011.

[83] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[84] Snapgear Linux for LEON, 2008. http://www.gaisler.com/anonftp/linux/linux-2.6/snapgear/snapgear-manual-1.0.37.pdf.

[85] Stochastic modelling in Python, 2012. http://stompy.sourceforge.net/.

[86] Simplified Wrapper and Interface Generator (SWIG), 2015. https://github.com/swig/swig.

[87] High Level Synthesis with Synphony C compiler, 2012. http://www.synopsys.com.

[88] System Python, 2012. http://cgi.di.uoa.gr/∼ evlog/syspy.html.

[89] SysPy Git code repository, 2015. https://github.com/evlog/SysPy.

[90] Viper: Python embedded real-time development, 2014. http://http://viper.thingsoninternet.biz/.

[91] Jack E. Volder. The CORDIC trigonometric computing technique. *Electronic Computers, IRE Transactions on*, EC-8(3):330–334, Sept 1959.

[92] J. S. Walther. A unified algorithm for elementary functions. In *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*, AFIPS '71 (Spring), pages 379–385, New York, NY, USA, 1971. ACM.

[93] System Generator for DSP, UG640, 2009. http://www.xilinx.com.

[94] Xilinx Synthesis Technology User Guide, UG627, 2009. http://www.xilinx.com.

[95] M. Yoshimi, Y. Iwaoka, Yuri Nishikawa, T. Kojima, Y. Osana, A. Funahashi, N. Hiroi, Y. Shibata, N. Iwanaga, H. Yamada, H. Kitano, and H. Amano. FPGA implementation

of a data-driven stochastic biochemical simulator with the Next Reaction Method. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 254–259, 2007.

[96] M. Yoshimi, Y. Osana, Y. Iwaoka, Y. Nishikawa, T. Kojima, A. Funahashi, N. Hiroi, Y. Shibata, N. Iwanaga, H. Kitano, and H. Amano. An FPGA implementation of high throughput stochastic simulator for large-scale biochemical systems. In *Field Programmable Logic and Applications, 2006. FPL '06.*, pages 1 –6, aug. 2006.

[97] M. Zhang, S.L. Tu, and Z.L. Chai. PDSDL: A dynamic System Description Language. In *IEEE Int'l SoC Design Conference*, volume 1, pages I–204–I–209, November 2008.