Hardware Design Automation with Python and Verilog HDL

Yassen Gorbounov
Department of Informatics
New Bulgarian University, University of Mining and Geology
Sofia, Bulgaria
ORCID 0000-0002-2936-951X

Abstract— **This article discusses an approach to facilitate the design of digital devices of increasing complexity by using the Python language. In this way, both synthesizable designs and non-synthesizable testbench modules can be created. Some practical examples used in the training of students are considered, including the design of a series of frames for control of matrix indicator, a binary multiplier, implemented both as a memory and by using a recursive algorithm. The capabilities of programming languages are increasingly used to create verification environments, as they provide convenience in describing the behavior in the submission of test vectors and making comparisons with the expected response despite the requirement for parallelism in the design. Also, in the verification process, the need for low-level hardware knowledge is not as great as when creating synthesizable designs. In this sense, the software approach, including the concept of object-oriented programming, turns out to be very appropriate. This makes working with complex designs more accessible, particularly for university students, saving them time and allowing them to concentrate on tasks with a higher level of difficulty.**

Keywords— Field Programmable Gate Arrays, Electronic Design Automation, Python, Verilog, Digital Design

I. Introduction

Since the invention of the transistor in 1947, the world of electronics has undergone extremely rapid development. The design of modern digital electronic devices is a long and complex process. Each integrated circuit (IC) can be represented by different levels of abstraction, given graphically by the Gajski-Kuhn Y-diagram [2][3] (with addition) shown in Fig. 1.

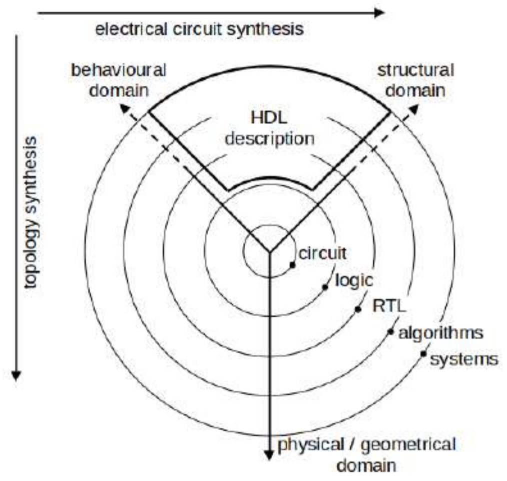


Fig. 1. Gajski-Kuhn Y-diagram

In the figure there are three axes arranged in the form of the letter Y, hence the name. The outer ring represents the highest level of abstraction which is the system view of the design while the inner rings correspond to the finer description of the lower levels of design topology down to transistor level. The physical domain axis relates to the topological design of integrated circuit. Going from the inner to the outer ring it is comprised of transistor/polygon, cell and module layouts, floorplans, and physical partitions of the chip. This axis reflects the physical nature of the materials and the production process. The two remaining axes relate to the digital design of the logic circuit description, the computation and the algorithmic structures. The structural domain axis is encompassing transistors (inner ring), gates, flip-flops, arithmetic logic units, multiplexers, memories, microarchitecture, instruction set architecture and processor descriptions, at the highest level of abstraction. The behavioral axis serves the purpose of describing the digital design in terms of transfer functions, logic, register transfer level (RTL), algorithms and systems point of view. The solid lines in the figure denote the automated process stages while the dashed lines point out there is still a large amount of work that have to be done by the hardware designer manually.

Modern ICs contain billions of transistors. The first commercially produced microprocessor 4004 [6] contained 2300 transistors and has been manufactured by less than one hundred people. That is about 23 transistors per employee. Considering the Moore's law [1], if that ratio stayed the same, Intel would need about 65 million employees to produce the Core i7 processor. This simple observation imposes two conclusions: a) it is not possible to design a device by drawing a schematic, hardware description languages (HDL) such as Verilog and VHDL are being used instead, and b) complex circuits require large teams of engineers often distributed around the world. This puts demands on the level of automation of the design process. At the same time, to ensure the required quality of the final product, the increasing complexity of the design increases the volume of work that is needed for its simulation and verification. In addition, the increased degree of circuits integration leads to an increasing need to hide the low level of description and the use of languages with higher-level of abstraction. The increasing attempts to utilize object-oriented programming languages such as C++ or Python in parallel with HDL languages makes the elaboration of complex designs more accessible by bridging the gap between computer engineers and software developers. The same goes for university students, saving them time and allowing them to concentrate on tasks with a higher level of difficulty under the condition of constrained time.

The rest of this article discusses several examples of design automation using the Python programming language. This approach has been tested in real conditions with students from different years of study and shows a very good level of acceptance.

II. Design Automation with Python

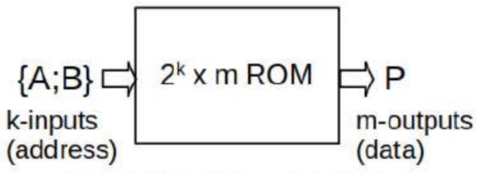
Field Programmable Gate Arrays (FPGAs) are increasingly used in the design of digital electronic devices as a prototype platform, both for training and in industry. Typically they require a serious knowledge not only of the logic element base, but also of the computational structures, the internal organization of the FPGA and problems such as clock-domain-crossing (CDC), synchronous and asynchronous interfaces, transaction-level communication, etc., which impact on the overall complexity of the project. The manufacturers of programmable logic circuits offer proprietary high-level synthesis (HLS) tools and this becomes an increasingly popular approach in the electronic design automation (EDA) that raises the abstraction level for designing digital circuits. In addition, higher-level programming languages, such as C, C ++ and Python, are more and more aggressively penetrating the field of automating the different stages of the design process. Despite that Python is an interpretive language, thanks to the concepts of the object oriented programming, it allows for the synthesis of complex structures with code reuse. In this regard there exist publicly available tools in the open source domain. PyEDA [7] is a Python library that provides high level Boolean functions interface and optimized C extensions for high-performance fundamental algorithms. It is aimed at symbolic Boolean algebra, logic expressions, truth tables, reduced, ordered binary decision diagrams (ROBDDs) and satisfiability (SAT) solvers. MyHDL [8] is a Python-based hardware description language (HDL) featuring the ability to generate VHDL and Verilog code from a MyHDL design and also the ability to generate a testbench with test vectors that are based on complex computations. It is also capable of doing co-simulation with Verilog. Cocotb (Coroutine cosimulation testbench) [9] is a library for digital logic verification in Python. It provides Python interface to control standard RTL simulators such as Cadence, Questa, VCS, etc., and offers an alternative to using Verilog, SystemVerilog or VHDL framework for verification. Unfortunately, although these products are mostly free, they are too complex as they aim at creating a complete toolchain and methodology, which goal is to fully replace some of the popular hardware description languages. Therefore they are characterized by a steep learning curve. The pure Python language does not require special skills in the field of the hardware. That is why this approach is adopted in this paper.

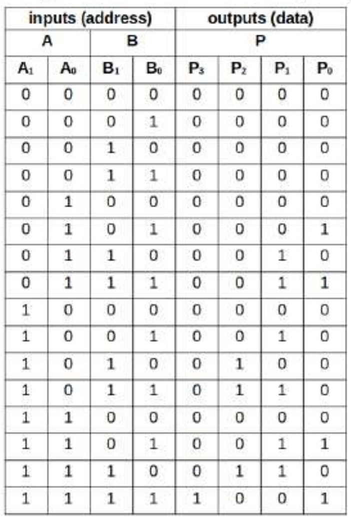
Below are demonstrated several hardware modules implemented with the aid of the Python language that were used in the process of teaching students.

A. *Binary multiplier*

Modern hardware description languages allow the behavioral description of arithmetic modules with the aid of standard mathematical operators, leaving the task of synthesizing or finding a suitable library module or structure to the compiler. This is due to the fact that in programmable logic circuits there exist fast (even single cycle) embedded multipliers like the DSP48A1 slice in the Xilinx Spartan 6 FPGA [13]. However, there are many cases where the generated design is not optimal and needs to be done by the designer [12]. In addition, for training purposes, an understanding of how the arithmetic modules work is mandatory. For this purpose, an implementation of a multiplication device is considered below.

The simplest method for implementing a multiplier device is by using a read only memory (ROM) which is in fact a lookup table (LUT). The approach is depicted in Fig. 2.





inputs (address)				outputs (data)			
A		B		P			
A_1	A_0	B_1	B_0	P_3	P_2	P_1	P_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Fig. 2. LUT-based binary multiplier

In the figure A and B are the operands and P is the product. The concatenation of A and B corresponds to the address bus and the memory word is the result from the multiplication. The great benefit of this method is that it allows to implement other functions as well. During the labs students deal with multipliers of different widths. Since the number of rows in the table equals 2^k, where k is the sum of A and B bit widths, it becomes obvious that for a multiplier of 4 by 4 bits the row count will increase to 256 and it will become even higher for larger bit widths.

The essence of the ROM data generator written in Python is shown in Fig. 3.

```
filename = "rom_{0}x{1}.dat"
a = int(input('Enter "a" width = '))
b = int(input('Enter "b" width = '))
w = a + b

name = filename.format(f'{2**w}', f'{w}')
fileobject = open(name, 'w')

for x in range(2 ** a):
    for y in range(2 ** b):
        fileobject.write(f'{bin(x * y)[2:].zfill(w)}\n')

fileobject.close()
```

Fig. 3. Excerpt of the Python ROM data generator

If the bit width of both the operands is 2 this would result in a file named "rom_16x4.dat" that can be read later using the $readmemb Verilog task. The Verilog module for solving the task is given in Fig. 4.

```
module multrom #(parameter AW = 2, BW = 2) (
            input [(AW+BW) - 1:0] address,
            input read_en,
            input ce,
            output [(AW+BW) - 1:0] data);

            reg [(AW+BW)-1:0] mem [0:2**(AW+BW)-1];

            assign data = (ce && read_en) ? mem[address] : 0;

            initial begin
            $readmemb("rom_16x4.dat", mem);
            end
endmodule
```

Fig. 4. Synthesizable Verilog module that accepts a ROM memory file

As it can be seen the file name and the word width are parameterized and this is the only adjustment that is to be made. The register transfer level (RTL) view of the generated module is shown in Fig. 5.

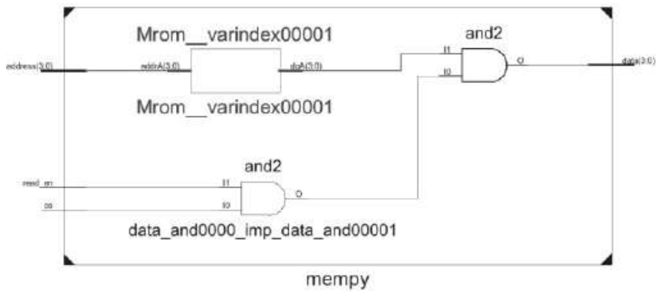


Fig. 5. RTL view of the synthesized LUT-based multiplier design

There exist many other types of multiplying algorithms such as Dadda multiplier, Wallace tree, parallel multipliers [14], Karatsuba algorithm [15] and its generalization the Toom–Cook (Toom-3) [16] or Fourier transform and other methods that are capable of multiplying complex numbers. A design approach for building parallel binary multiplier circuit by using partial products is shown in Fig. 6.

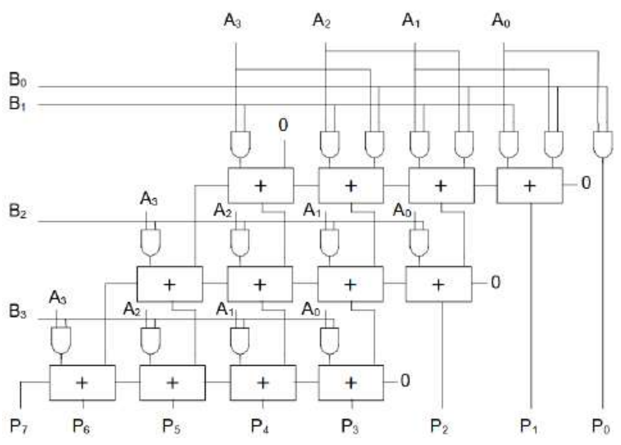


Fig. 6. Parallel binary multiplier using partial products

It can be seen from the figure that this design is highly modular, therefore it can be easily described by using a high-level language such as Python. It can be further exported as a Verilog HDL description and then implemented on a real FPGA chip.

B. LED matrix indicator frames

As a more advanced example it can be pointed out the 8x8 light emitting diode (LED) display. It can be driven with the aid of a serially interfaced display driver like the MAX7219 integrated circuit as shown in Fig. 7.

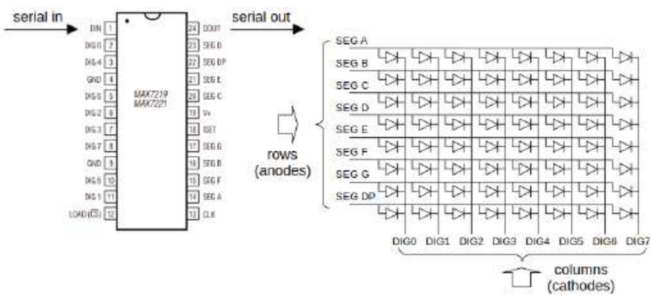


Fig. 7. Driving a matrix display using the MAX7219 shift register

The driver can be seen as a serial shift register driven by standard serial peripheral interface (SPI) protocol. That makes it easy to sequentially shift data from a pre-generated array of data. In order to visualize the LEDs configuration (the image) a Python application has been built and its graphical frontend is shown in Fig. 8.

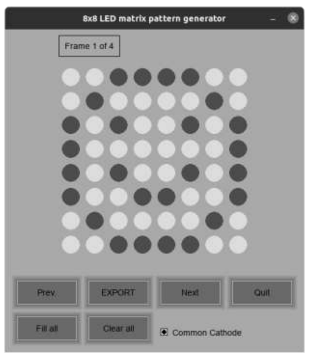


Fig. 8. Python application for exporting frames

This application is built around the cross-platform set of Python modules originally designed for writing video games – Pygame [17]. Pygame provides tools and utilities for generating buttons (the LEDs dots are buttons as well), checkboxes, and other graphical components. The application is capable of generating unlimited number of image frames, and it can export data for either common anode (CA) or common cathode (CC) displays. Using this tool single image or a series of frames can be generated. Later these frames can be animated on a single matrix display or spread over several serially connected matrix displays. Each frame can be exported as a ROM file, as in the multiplier example, or as Verilog data array which is shown in Fig. 9.

```
reg [7:0] data [7:0];
data[0] = 8'b00111100
data[1] = 8'b01000010
data[2] = 8'b10100101
data[3] = 8'b10000001
data[4] = 8'b10100101
data[5] = 8'b10011001
data[6] = 8'b01000010
data[7] = 8'b00111100
```

Fig. 9. Verilog HDL data array, exported by the application

This approach can save a lot of efforts to students providing them sufficient time to concentrate on the main task namely the digital design of the communication controller and the verification of the project.

C. Digital design verification

The design verification is the second most important aspect of the product development process which may consume as much as 80% of the total product development time [18]. The design verification aims to ensure that the design works correctly not only in functional logic sense but also that it is free from signal race conditions, glitches etc., i.e. it is time-accurate. Therefore they could be distinguished the functional verification, where functional models are verified against the behavioral specification, and formal verification, where a parameter checking against presumed properties is done, such as preventing a finite-state machine (FSM) from entering disallowed or unreachable states. It is obvious that this is a long and possibly quite complicated process. That is why besides the classic testbenches (the term is a single word), common to HDL languages, growing popularity gain high-level programming languages and object oriented technologies such as the Universal Verification Methodology (UVM). In fact UVM is not a language itself but a library that works as part of the SystemVerilog standard.

The concept for the static testbench approach is shown in Fig. 10.

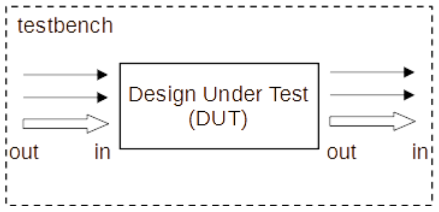


Fig. 10. A verification environment – a testbench

The testbench is an HDL file without input and output ports. It is used to provide a test environment in which a predefined input stimuli are applied on the input ports of the design under test (DUT), which is the module to be verified. In the same time the testbench expects some reaction that appears on the output ports of the DUT. Usually the result is expressed graphically. A sample waveform resulting from the simulation of the multiplier device from point II.A, obtained by the aid of a Verilog testbench is shown in Fig. 11.

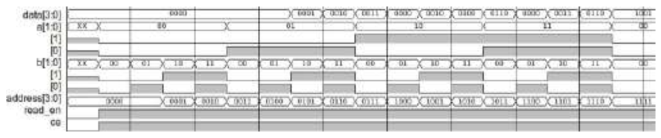


Fig. 11. Simulation result proving the working of the LUT-based multiplier

The described process is very straightforward but quite limited in functionality. It is suitable for small to medium size digital designs but it is not very convenient for complex interfaces such as the advanced extensible interface (AXI) bus or the peripheral component interconnect express (PCIe) bus. Such interfaces embed complex state machines and sometimes include a handshaking process that may require testing with not only known in advance static data but also with randomized data streams. That is why there exist other techniques like the UVM. A sample basic overview of the UVM concept is shown in Fig. 12.

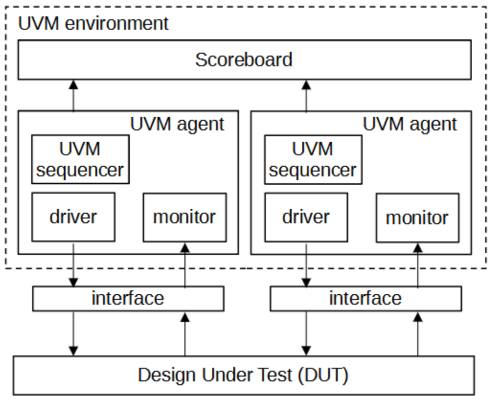


Fig. 12. Verification environment - testbench

This oversimplified diagram shows one UVM environment. The DUT interacts with the UVM agents via public and virtual interfaces which provide medium for transferring stimuli from the drivers to the DUT and responses from DUT to the monitors. Each driver can accept a series of sequences – scenarios. The results are compared by the scoreboard and also with the aid of a function coverage unit. The UVM environment is extremely powerful but is suffers two major drawbacks – (1) it is too complex and it has very steep learning curve, and (2) it is supported mainly by proprietary and expensive design tools or has limited coverage in very few freely available environments such as EDA Playground [20] and Xilinx Vivado [21]. This strongly restricts the usage of this technology in the university or by enthusiasts.

The above problems make it appropriate to use the Python language to automate the verification process. Although programming languages, unlike hardware description languages, do not have constructions for parallel execution of tasks, they can still be used to automatically generate re-running functional and non-functional tests, known as regression testing. They can also be used for generating complex parallel or serial randomized data or pass/fail verification with different scenarios. A simple example of the later is shown in Fig. 13.

```
    filename = "{0}_{1}.txt"
    name = filename.format(f'{"Pass"}', f'{"Fail"}')

    f = open(name, 'w')
    f1 = open("rom_16x4.dat", "r")
    f2 = open("results.dat", "r")
    i = 0
    for line1 in f1:
        i += 1
        for line2 in f2:
            if line1 == line2:
                print(" Line ", i, ": PASS")
                f.write("PASS\n")
            else:
                print(" Line ", i, ":", "\tFAIL")
                print("\tFile 1:", line1, end='')
                print("\tFile 2:", line2, end='')
                f.write(f"Difference found in Line {i}\n " )
                f.write(f"Line {i} in File1)--> {line1} ")
                f.write(f"Line {i} in File2)--> {line2}")
            break
    f1.close()
    f2.close()
f.close()
```

Fig. 13. Scoreboard-like pass/fail test

In this example a text mode verification of the multiplier from point II.A is considered. The script opens two files, the input "rom_16x4.dat" and the generated by the Verilog module "results.dat". Next it compares the stimuli with the reaction and generates a pass/fail report in an output file. This way it becomes easy to distinguish the correct from the incorrect results. Additional benefits of this approach are the increased speed of simulation and the possibility to run many verification scenarios in a sequential manner. Another possibility is the use of assertions but this method is applicable to larger and more complex designs.

III. Future Work

There is a large field for improvement of the given examples. One future task is to extend the Python application for the 8x8 LED matrix indicator to support displays with different resolutions and to simulate the sequential frame change before generating the Verilog file. Another natural continuation of the present work includes the elaboration of a finite state machine framework written in the Python language. This would greatly facilitate the mastery of the theory of FSM and would allow for easier design of extremely complex control algorithms inherent to modern computing machines. Also the integration with the free Octave and Scilab mathematical products would be very useful as it would allow for building much more versatile abstract models using a high-level simulation environment.

IV. Conclusion

In the present work, some ways to introduce the Python programming language to support the hardware synthesis of digital devices were presented. This is a long-term trend that aims to bridge the gap between computer engineers and software developers. The need for this is dictated by the growing complexity of digital circuits, which requires extremely in-depth knowledge of digital electronics, while the development time is getting shorter. In the present work, an attempt is made to avoid expensive and complex company development tools by proposing the use of classical constructions of the Python language. This makes working with complex designs more accessible, particularly for university students, saving them time and allowing them to concentrate on tasks with a higher level of difficulty.

References

[1] C. Mody, The Long Arm of Moore's law: Microelectronics and American Science, MIT Press, ISBN 978-0262035491, 2016

[2] D. Gajski, R. Kuhn, New VLSI Tools, Computer Magazine, Vol. 16, pp. 11-14, DOI:10.1109/MC.1983.1654264, 1983

[3] W. Meeus, et al., An overview of today’s high-level synthesis tools, Design Automation for Embedded Systems, Springer, Vol. 16, Issue 3, pp.31–51, DOI:10.1007/s10617-012-9096-8, 2012

[4] J. Villar, et al., Python as a hardware description language: A case study, VII Southern Conference on Programmable Logic (SPL), pp.117-122, ISBN 978-1-4244-8847-6, DOI:10.1109/SPL.2011.5782635, 2011

[5] S. Ravi, M. Joseph, Open source HLS tools: A stepping stone for modern electronic CAD, IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), pp. 1-8, ISBN 978-1-5090-0613-7, DOI:10.1109/ICCIC.2016.7919615, 2016

[6] Intel's First Microprocessor, Its invention, introduction, and lasting influence, Intel Corporation, 2021, https://www.intel.co.uk/content/www/uk/en/history/museum-story-of-intel-4004.html

[7] C. Drake, PyEDA: Data Structures and Algorithms for Electronic Design Automation, 14th Python in Science Conference, Austin, Texas, ISSN 2575-9752, 2015

[8] J. Decaluwe, MyHDL: a python-based hardware description language, Linux Journal, Vol. 2004, No. 127, doi:10.5555/1029015.1029020, 2004

[9] B. Rosser, Cocotb: a Python-based digital logic verification framework, The ATLAS Experiment, CERN Microelectronics, University of Pennsylvania, 2018

[10] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, T. Sherwood, A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation, 27th International Conference on Field Programmable Logic and Applications (FPL), pp. 1-7, ISBN 978-1-5386-2040-3

[11] S. Yamazaki, Pyverilog-A Python-Based Hardware Design Processing Toolkit for Verilog HDL, ARC 2015, Lecture Notes in Computer Science, Vol. 9040, Springer, Cham, pp. 451-460, ISBN 978-3-319-16213-3, 2015

[12] Zh. Zhelyazkov, Y. Gorbounov, Configurable signed adder, built using a programmable logic device, 14th National Youth Scientific and Practical Conference, pp. 1-6, ISSN 1314-8931, 2016

[13] Xilinx Spartan-6 FPGA DSP48A1 Slice User Guide UG389 (v1.2), 2014

[14] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, ISBN 978-8178672663, 2012

[15] A. Karatsuba, The Complexity of Computations, Proceedings of the Steklov Institute of Mathematics, Vol. 211, pp. 169–183, 1995

[16] M. Bodrato, Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0, Springer-Verlag, International Workshop on the Arithmetic of Finite Fields WAIFI 2007 proceedings, Madrid, Spain, pp. 116–133, LNCS 4547, 2007

[17] L. Lindstrom, R. Dudfield, et al., Pygame v. 2.0.1, 2020, https://www.pygame.org

[18] L. Wang, Y. Chang, K. Cheng (editors), Electronic Design Automation, Morgan Kaufmann, Elsevier, 2009, ISBN 978-0-12-374364-0

[19] Accellera Systems Initiative, CA 95624, Universal Verification Methodology, https://www.accellera.org/downloads/standards/uvm

[20] Doulos Ltd., UK, EDA Playground, https://www.edaplayground.com/

[21] Xilinx Inc., San Jose, California, USA, https://www.xilinx.com/products/design-tools/vivado.html