International Scientific Conference 20 UNITECH 26

INTERNATIONAL SCIENTIFIC CONFERENCE
19-21 November 2026
GABROVO

ПОДХОД ЗА ПРОЕКТИРАНЕ И ИНТЕГРАЦИЯ НА FPGA ВЕКТОРНИ УСКОРИТЕЛИ В ОБУЧЕНИЕТО ПО ПРОЕКТИРАНЕ НА ХАРДУЕР С ЕЗИЦИ ОТ ВИСОКО НИВО

Петър Минев[1], Валентина Кукенска[1], Илиан Върбов[1]

[1]Технически университет – Габрово, ул. Х. Димитър 4, Габрово, България
**кореспондиращ автор: pminev@tugab.bg*

AN APPROACH TO DESIGNING AND INTEGRATING FPGA VECTOR ACCELERATORS IN HIGH-LEVEL HARDWARE DESCRIPTION LANGUAGE EDUCATION

Petar Minev[1], Valentina Kukenska[1], Ilian Varbov[1]

[1]Technical University of Gabrovo, 4 H. Dimitar Str., Gabrovo, Bulgaria
**Corresponding author: pminev@tugab.bg*

Abstract
This paper presents a methodical and practical approach to the design, hardware-software integration, and validation of specialized FPGA soft accelerators for vector operations, tailored for implementation in higher education courses covering High-Level Hardware Description Languages (High-Level HDLs / HLS) and digital system design. The proposed methodology spans the entire development cycle: from abstract execution logic modeling using the Python-based PyRTL framework, through automated generation and encapsulation as an AXI4-Lite peripheral IP block in Xilinx Vivado, to high-level runtime control via the PYNQ framework on a Zynq-7000 SoC platform. Key implementation aspects – including AXI address decoding, control signal handshaking, and interface logic adaptation between the generated RTL code and the system bus – are thoroughly analyzed. The approach enables students to bypass tedious low-level Verilog/VHDL syntax complexities, focusing instead on architectural logic, Hardware/Software Co-design, and practical System-on-Chip integration.

Keywords: High-Level Hardware Description Languages, PyRTL, PYNQ, Vector Accelerator, Engineering Education.

ВЪВЕДЕНИЕ

Съвременните компютърни архитектури все повече разчитат на специализирани хардуерни ускорители (като векторни и матрични процесорни ядра) за ефективна обработка на масиви от данни, използвани в сфери като машинното обучение, цифровата обработка на сигнали и компютърното зрение [5]. Запознаването на студентите с принципите на проектиране, синтез и софтуерно управление на подобни ускорители често среща сериозна бариера – ниската ефективност на традиционните езици VHDL/Verilog при описание на комплексен хардуер и сложната настройка на тестовите среди.

За да се преодолее това ограничение, в настоящата разработка се предлага Python-центриран подход за проектиране и верификация на специализиран софтуерен векторен ускорител. В основата на този подход са две основни технологии:

1) PyRTL (Python Register Transfer Level): Специализирана библиотека за Python, която позволява хардуерното проектиране на регистрово ниво да се

Print ISSN: 1313-230X, Online ISSN: 3033-1404

извършва директно чрез Python синтаксис. Вместо да се пише сложен VHDL/Verilog код, векторното изчислително ядро (векторният ускорител) се моделира, симулира и генерира като Verilog RTL код изцяло чрез PyRTL. Това драстично съкращава цикъла за разработка и прави концепции като дървета от сумратори, паралелни умножители и регистрови файлове значително по-лесни за разбиране от незапознатия читател [1].

2) PYNQ (Python Productivity for Zynq): Развойна среда от AMD/Xilinx, базирана на Zynq-7000 System-on-Chip (SoC). Тя обединява двуядрен ARM процесор (управляващ Linux и Jupyter Notebook) и FPGA матрица (в която се зарежда генерираното чрез PyRTL ядро) в обща система [2].

Целта на настоящата работа е да предложи и систематизира методология за пълен развоен цикъл при изграждане на копроцесор (ускорител) за векторно скаларно произведение, интегриращ съвременни инструменти за проектиране на високо ниво (PyRTL), AXI4-Lite шинна интеграция във Vivado и интерактивна програмна валидация през Python/PYNQ.

Този подход дава възможност на студентите или изследователите да взаимодействат с хардуера на най-високо абстрактно ниво (Python), докато отдолу физически се изпълняват реални, паралелни FPGA изчисления.

Основните стъпки, описващи прехода от Python модела до реалната имплементация в FPGA интегралната схема са:

1) Моделиране на ядрото (PyRTL): Векторният ускорител се описва като Python обект с помощта на PyRTL класове (Input, Output, Register, WireLogic).

2) Функционална симулация (PyRTL Fast Simulation): Тестване на математическата коректност на ядрото директно в Python среда (без нужда от външни симулатори като ModelSim).

3) Експорт към Verilog: PyRTL автоматично транслира създадения хардуерен граф в стандартен Verilog файл (vector_core.v).

4) Синтез и AXI интеграция (Vivado): Verilog файлът се свързва към AXI4-Lite шината и се синтезира до битстрийм файл (.bit).

5) Изпълнение и визуализация (PYNQ / Jupyter): Битстриймът се зарежда върху Zynq-7000 платката, а резултатите от изчисленията се извличат интерактивно през Jupyter Notebook.

ИЗЛОЖЕНИЕ

Проектираният ускорител се използва за извършване на векторно скаларно произведение по следната формулата:

$$S = \sum_{i=0}^{7} (a_i \cdot b_i) \qquad (1)$$

На фиг. 1 е представена обобщената архитектура на векторния ускорител. Тя се състои от три подсистеми:

1) Комуникационната подсистема служи като мост между 32-битовата системна шина AXI4-Lite и вътрешната 64-битова архитектура на ускорителя. Тъй като AXI-Lite поддържа 32-битови транзакции, за зареждането на един 8-елементен вектор от 8-битови стойности (8 x 8 = 64 бита) се използва пакетиране/обединяване на данните през двойка последователно свързани 32-ббитови регистри.

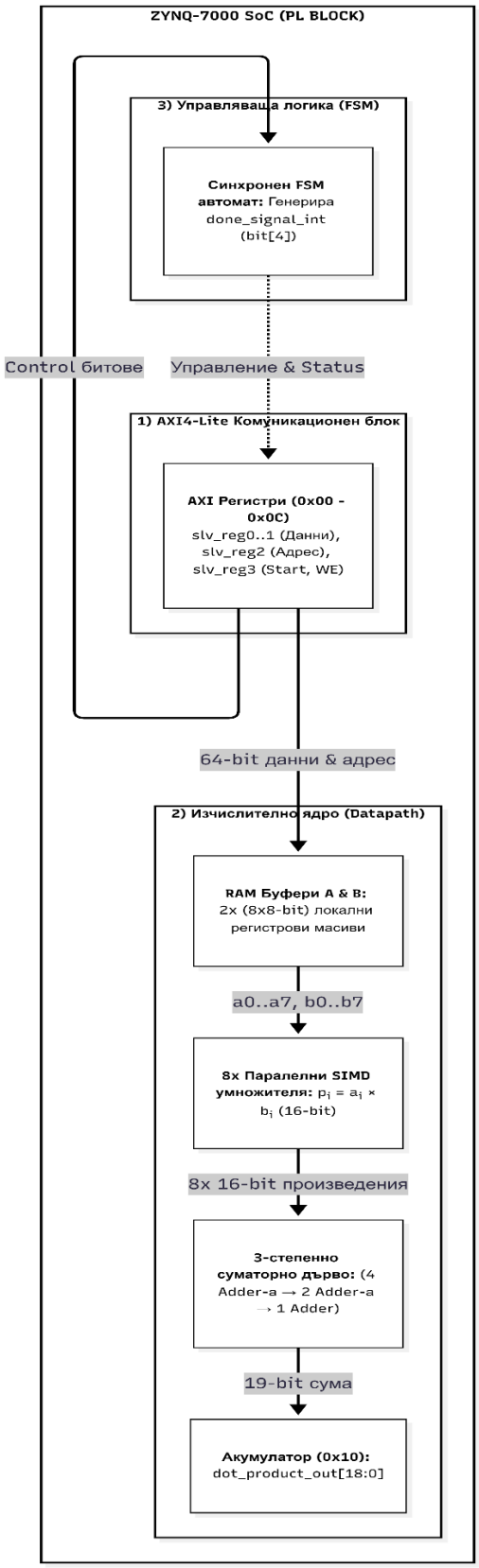
На високо ниво, процесът на обединяване и разделяне се управлява през Python/PYNQ, където 8-байтовият вектор се позиционира чрез побитови изместванията *(<<(i * 8))*, а на ниво хардуерен интерфейс 64-битовият вход *axi_data_in[63:0]* се формира от конкатенацията *{slv_reg1, slv_reg0}*.

2) Изчислителното ядро изпълнява векторно скаларно произведение над два 8-елементни масива. Архитектурата на операционния блок е проектирана за паралелно обработване на данните (SIMD).

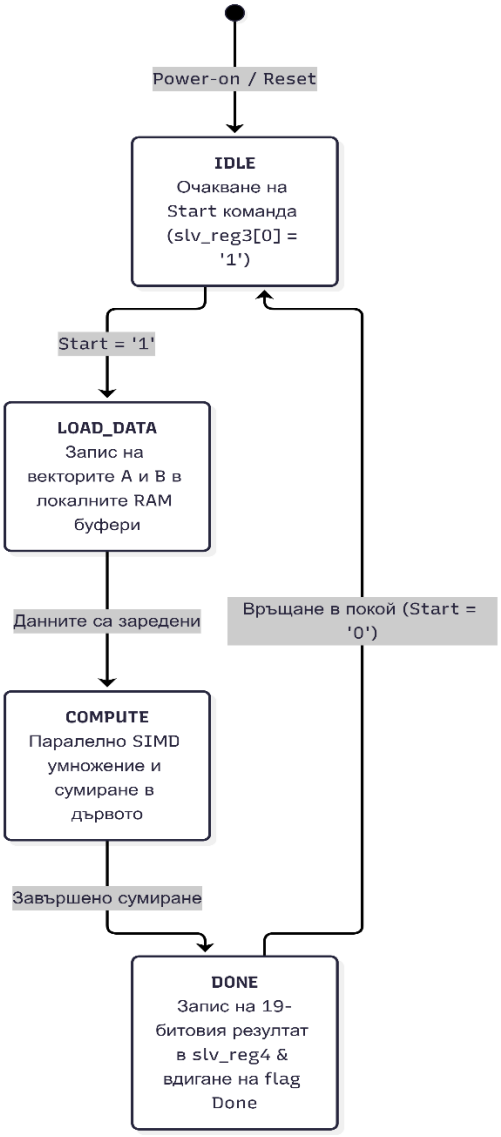
Всички 8 елемента на двата вектора се съхраняват в локални регистрови масиви с директен паралелен достъп до всеки байт. Осем независими 8-битови беззнакови умножителя пресмятат едновременно всички произведения, генерирайки осем частични 16-битови резултата в един тактов цикъл. За да се избегнат дълги комбинационни вериги, сумирането на частичните произведения се извършва чрез 3-степенно дърво от суматори. Резултатът се фиксира в 19-битов изходен регистър *dot_product_out[18:0]*, с което се избягва препълване.

3) Управляващата подсистема съдържа краен автомат (фиг. 2), изпълняващ протокол за синхронизация (handshaking) с процесорната система през управляващия регистър *slv_reg3 (0x0C)*.

За осъществяване на връзката между процесорната система (PS – ARM Cortex-A9) и програмируемата логика (PL – FPGA) се използва стандартният протокол AXI4-Lite [3]. Той осигурява т. нар. “memory-mapped” достъп до вътрешните регистри на ускорителя. В стандартния развоен процес на Xilinx Vivado, грижата за съгласуването на ниско ниво на AXI сигналите (като Handshake сигналите AWVALID, AWREADY, WVALID, WREADY, BRESP и др.) се поема от автоматизиран съветник (Create and Package IP Wizard).



Фиг. 1. *Обща архитектура на векторния ускорител*



Фиг. 2. *Диаграма на състоянията на крайния автомат в управляващата логика*

При създаване на нов потребителски IP блок, Vivado автоматично генерира йерархична структура от два Verilog файла: 1) във файла myip_v1_0.v (Top-level IP Wrapper) е реализиранна външната обвивка на IP блока, която изкарва портовете към системата; 2) модулът myip_v1_0_S00_AXI.v (AXI Slave Logic File) съдържа същинската AXI4-Lite Slave логика, дешифратора на адреси, вътрешните регистри (slv_reg0 до slv_reg7) и крайния автомат за управление на транзакциите по четене и запис. Файлът myip_v1_0_S00_AXI.v съдържа дефиниции на регистри slv_regN за четене/запис. Задачата на студентите в обучението по проектиране на хардуер с езици от високо ниво е да модифицират този файл, интегрирайки вътре синтезираното от PyRTL ядро на векторния ускорител и да пренасочат AXI шината да чете реалните хардуерни сигнали, вместо статичните регистри.

Всички регистри са подравнени по 32-битови (4-байтови) граници (таблица 1).

Табл. 1. *AXI карта на паметта*

Байтов адрес	**AXI Регистър**	**Достъп**	**Функция / Назначение**
0x00	slv_reg0	Write	Младша 32-битова дума от данните (байтове 0..3)
0x04	slv_reg1	Write	Старша 32-битова дума от данните (байтове 4..7)
0x08	slv_reg2	Write	Адрес за запис във вътрешната RAM (slv_reg2[7:0])
0x0C	slv_reg3	Write / Read	Управление/Статус: bit[0]:axi_write_en_a bit[1]:axi_write_en_b bit[2]: start_signal bit[3]: matrix_done bit[4] (Read): done_signal_int
0x10	slv_reg4	Read-only	Изходен резултат (dot_product_out_int [18:0])

Интеграцията на PyRTL ядрото изисква три последователни стъпки на модификация вътре в генерирания от Vivado файл myip_v1_0_S00_AXI.v: 1) В секцията за потребителски декларации се дефинират проводниците, които ще пренасят изходните сигнали от генерирания през PyRTL Verilog модул на ядрото към AXI логиката (фиг. 3); 2) В специално предназначената от Xilinx секция *// Add user logic here* студентите инстанцират изчислителния модул (PyRTL ядрото), като го свързват към AXI регистрите (фиг. 4); По подразбиране, AXI Slave логиката връща съдържанието на вътрешния регистър *slv_reg4* при четене от адрес 0x10. Тъй като *slv_reg4* е пасивен регистър за запис,

прочитането му винаги би върнало 0. За да може AXI шината да прочете актуалния комбинационен изход от PyRTL ускорителя, адресният дешифратор (*case* операторът в *always @(*)*) се коригира, като адреси 3'h3 (0x0C) и 3'h4 (0x10) се обвързват с реалните хардуерни линии (фиг. 5).

```
// --- USER LOGIC SIGNALS START ---
wire done_signal_int; // Изходен флаг за готовност от PyRTL ядрото
wire [18:0] dot_product_out_int; // 19-битов изходен резултат от суматорното дърво
// --- USER LOGIC SIGNALS END ---
```

Фиг. 3. *Деклариране на свързващите проводници*

```
// Add user logic here

toplevel my_vector_proc_inst (
    .clk            (S_AXI_ACLK),                   // Системен AXI тактов сигнал
    .rst            (~S_AXI_ARESETN),               // Инвертиран AXI нулиращ сигнал
    .axi_data_in    ({slv_reg1, slv_reg0}),         // Конкатенация на 2x32-бита до 64-бита вход
    .axi_write_addr (slv_reg2[7:0]),                // Адрес за вътрешната RAM
    .axi_write_en_a (slv_reg3[0]),                  // Control Bit 0: write_en_a
    .axi_write_en_b (slv_reg3[1]),                  // Control Bit 1: write_en_b
    .start_signal   (slv_reg3[2]),                  // Control Bit 2: start
    .matrix_done    (slv_reg3[3]),                  // Control Bit 3: matrix_done
    .done_signal    (done_signal_int),              // Изходен флаг към bit[4] на 0x0C
    .dot_product_out (dot_product_out_int)          // Резултат към адрес 0x10
);

// User logic ends
```

Фиг. 4. *Инстанциране на PyRTL ядрото.*

Чрез тези модификации студентите придобиват практически опит как абстрактният хардуерен код, описан на високо ниво (PyRTL/Python), физически се интегрира, адресира и управлява в реална System-on-Chip (SoC) среда.

За практическата валидация и софтуерното управление на проектирания векторен ускорител се използва платформата PYNQ (Python Productivity for Zynq). Това е отворена развойна среда от AMD/Xilinx, предназначена за висшето образование и индустриалното прототипиране.

```
// Address decoding for reading registers
always @(*)
begin
    case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
        3'h0   : reg_data_out <= slv_reg0;
        3'h1   : reg_data_out <= slv_reg1;
        3'h2   : reg_data_out <= slv_reg2;

        // Адрес 0x0C (3'h3): При четене се сливат записаните контролни битове [3:0]
        // с реалния статус флаг done_signal_int, разположен на bit[4]
        3'h3   : reg_data_out <= {slv_reg3[31:5], done_signal_int, slv_reg3[3:0]};

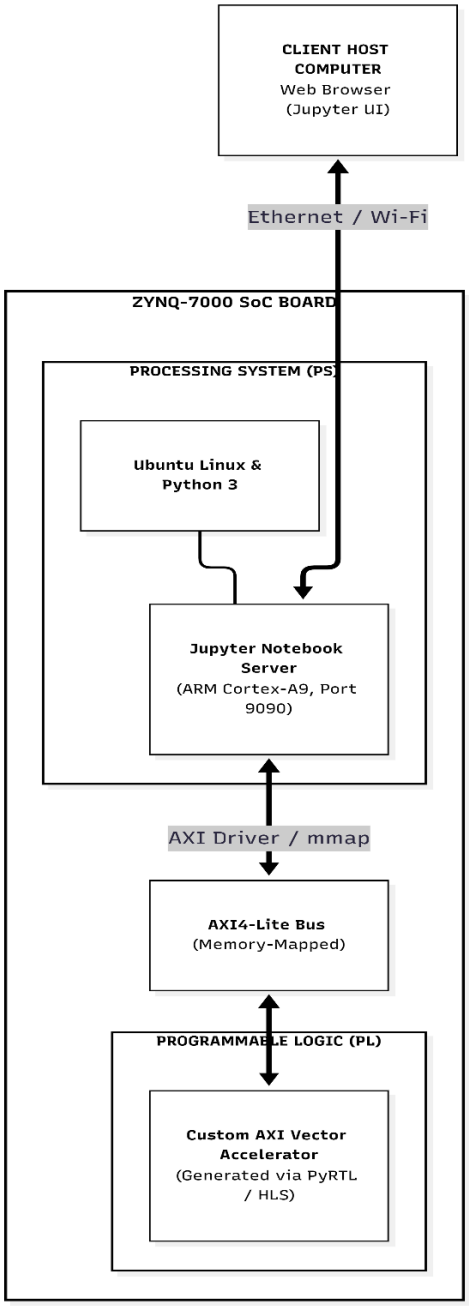
        // Адрес 0x10 (3'h4): Четене ДИРЕКТНО от изчислителното ядро (dot_product_out_int)
        // Допълва се с нули в старшите битове до 32-битова AXI дума
        3'h4   : reg_data_out <= {13'b0, dot_product_out_int};

        3'h5   : reg_data_out <= slv_reg5;
        3'h6   : reg_data_out <= slv_reg6;
        3'h7   : reg_data_out <= slv_reg7;
        default : reg_data_out <= 0;
    endcase
end
```

Фиг. 5. *Корекция на адресния дешифратор за четене*

На фиг. 6 е представена архитектурата на Zynq-7000 SoC [4]. Тя съчетава две основни подсистеми: 1) Processing System (PS): Двуядрен ARM Cortex-A9 процесор, управляващ периферните устройства и системната DDR памет; 2) Programmable Logic (PL): Програмируема FPGA ма-

трица (Artix-7 клас), в която се имплементира разработения векторен ускорител.



Фиг. 6. *Архитектура на изпълнение на Jupyter Notebook средата върху Zynq-7000 SoC и AXI комуникация с FPGA логиката*

За да може Jupyter Notebook да се изпълнява върху развойната платка с Zynq-7000 SoC предварително е направено следното: 1) заредена е операционна система. За целта върху MicroSD картата на развойната платка е инсталиран специализирана Ubuntu Linux дистрибуция, която е оптимизирана за ARM архитектурата на Zynq; 2) стартиран е вграден уеб сървър. При включване на платката, Linux автоматично стартира Jupyter Notebook сървър като системна услуга (daemon), работещ на порт 9090; 3) свързване и конфигуриране на мрежа. Платката се свързва към локалната мрежа на лабораторията през Ethernet интерфейс. Студентите достъпват потребителския интерфейс директно през стандартен уеб браузър на работните си станции (напр. http://local_ip_adress:9090), без да е необходимо инсталиране на сложни IDE среди или специални драйвери на клиентския компютър; 4) управление на хардуера през Python (pynq.Overlay). Python кодът, изпълняван в клетките на Jupyter, използва C-wrapper библиотека (/dev/mem достъп), която позволява динамично зареждане на FPGA битстрийм файла (.bit) през AXI DevC (Device Configurator) и директно четене/запис в AXI адресните пространства (read() / write()).

За да тестват ускорителя, студентите пишат капсулиран Python клас или функция, която реализира протокола за синхронизация (handshake) през AXI регистрите, дефинирани в таблица 1. На фигура 7 е представен капсулираният Python драйвер за валидация на ускорителя.

Студентите изпълняват тестовия скрипт от фиг. 8 в интерактивна клетка на Jupyter Notebook с контролни вектори $A = [1, 2, 3, 4, 5, 6, 7, 8]$ и $B = [1, 1, 1, 1, 1, 1, 1, 1]$. Регистър 0x0C (статус) трябва да върне 0x1C (0b11100), което да потвърди, че *done_signal_int* (bit 4) е активен. Регистър 0x10 трябва да върне стойността 36, с което да се потвърди функционалната коректност на ускорителя.

Внедряването на този подход в лабораторните упражнения по проектиране на хардуер с езици от високо ниво предоставя следните педагогически ползи:

```
from pynq import Overlay
import time

overlay =
Overlay('design_1.bit')
my_ip = overlay.myip_v1_0_0
 # AXI IP софт-ускорител

def
run_vector_dot_product(ip,
vec_a, vec_b, addr=0):
    # Пакетиране на 8 байта
в две 32-битови думи (low,
high)
    pack = lambda v: (sum(b
<< (i*8) for i, b in
enumerate(v[:4])),
                      sum(b
<< (i*8) for i, b in
enumerate(v[4:8])))

    # 1. Запис на Вектор A
(Buffer A) и Вектор B
(Buffer B)
    for vec, en_bit in
[(vec_a, 0x01), (vec_b,
0x02)]:
        low, high =
pack(vec)
        ip.write(0x00,
low);  ip.write(0x04, high)
        ip.write(0x08,
addr); ip.write(0x0C,
en_bit); ip.write(0x0C,
0x00)

    # 2. Стартиране на FSM
(bit[2]) и Handshake
(bit[3])
    ip.write(0x0C, 0x04);
time.sleep(0.001)
    ip.write(0x0C, 0x0C);
time.sleep(0.001)

    # 3. Четене на статус
(0x0C) и 19-бит резултат от
0x10
    status, result =
ip.read(0x0C), ip.read(0x10)
    ip.write(0x0C, 0x00)  #
Сваляне на управляващите
сигнали
    return result, status
```

Фиг. 7. *Python драйвер за векторния ускорител*

```
# Изпълнение на теста в
Jupyter Notebook
vec_a = [1, 2, 3, 4, 5, 6, 7,
8]
vec_b = [1, 1, 1, 1, 1, 1, 1,
1]

hw_result, status =
run_vector_dot_product(my_ip,
vec_a, vec_b)
sw_expected = sum(a * b for
a, b in zip(vec_a, vec_b))

print(f"Статусен регистър
(0x0C): hex={hex(status)},
bin={bin(status)}")
print(f"Резултат от FPGA
ускорителя (0x10):
{hw_result}")
print(f"Очакван софтуерен
резултат (Python):
{sw_expected}")

assert hw_result ==
sw_expected, "ГРЕШКА:
Несъответствие в
изчисленията!"
print(">>> УСПЕШНА ХАРДУЕРНА
ВЕРИФИКАЦИЯ! <<<")
```

Фиг. 8. *Скрипта на Python за функционална верификация в Jupyter Notebook*

1) Студентите виждат физическото съответствие между Python инструкциите write()/read() и AXI транзакциите по шината; 2) Необходимостта от генериране на импулсни управляващи сигнали (write_en, start_signal) спрямо подаването на статични данни; 3) Замяната на бавните HDL симулации с реално изпълнение върху FPGA платка през PYNQ съкращава времето за итерация от часове на минути.

ЗАКЛЮЧЕНИЕ

Представеният подход демонстрира завършен цикъл за проектиране, интеграция и изпитване на софтуерни ускорите-

ли за векторна обработка в FPGA базирани SoC среди. Решени са ключови въпроси по AXI адресирането, дешифрирането и синхронизацията на управляващия автомат. Методологията има практическа стойност и представлява готова основа за лабораторни упражнения и курсови проекти, подготвящи студентите за съвременните предизвикателства в областта на съвместното софтуерно-хардуерно проектиране.

В бъдеще се предвижда разширяване на архитектурата чрез добавяне на поддръжка за операции с плаваща запетая (FP16/FP32) и интегриране на DMA (Direct Memory Access) контролер за директен обмен на големи масиви от данни с DDR паметта [4, 5]. Планира се също така надграждане на обучението с експерименти за паралелно управление на няколко изчислителни ядра, както и проучване на възможностите за автоматизиран транслиращ синтез на по-сложни алгоритми за машинно обучение директно от PyRTL към FPGA битстрийм.

Докладът се публикува във връзка с резултатите от изпълнение на дейностите по проект № НИП2026-21

ЛИТЕРАТУРА

[1] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, 2017, pp. 1-7, doi: 10.23919/FPL.2017.8056860.

[2] AMD / Xilinx, "PYNQ: Python Productivity for Zynq – Official Documentation," Online: pynq.readthedocs.io, 2023.

[3] ARM Limited, "AMBA AXI and ACE Protocol Specification: AXI3, AXI4, and AXI4-Lite," ARM IHI 0022E, 2021.

[4] AMD / Xilinx, "Zynq-7000 SoC Technical Reference Manual," UG585 (v1.13), Feb. 2021.

[5] J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, 6th ed., Morgan Kaufmann / Elsevier, 2017.