Hardware coprocessor integration with NEORV32: characterization for efficient implementation of RISC-V-based AI SoCs

Unai Sainz-Estebanez*, Unai Martinez-Corral†, and Koldo Basterretxea‡
Grupo de Diseño en Electrónica Digital (GDED) †‡Dept. Electronics Technology
University of the Basque Country (UPV/EHU)
Bilbao, Basque Country, Spain
*usainz003@ehu.eus †unai.martinezcorral@ehu.es ‡koldo.basterretxea@ehu.es

***Abstract*—Performing AI inference ubiquitously requires energy-efficient, small footprint and highly reliable processing devices. Heterogeneous processing architectures combining customized CPUs with domain specific coprocessors can provide a good trade-off between computational efficiency and application flexibility for edge AI deployments while shortening development times compared to full custom application-specific processor designs. Following the impulse for the European sovereignty in the microelectronics field, in this work we propose the use of a RISC-V based open-source hardware platform and Free/Libre and/or Open Source (FLOS) Electronic Design Automation (EDA) tools to evaluate the performance of different coprocessor integration options in a System-on-Chip (SoC) prototyped on FPGA. We tested four integration options (XBUS, Stream, CFS and CFU) to obtain precise data that will allow making the correct design decisions for the future development of integrated devices for high-performance AI at the edge.**

***Index Terms*—RISC-V, FPGA, NEORV32, FLOS tools, AI at the edge**

I. Introduction

Fully performing the growing number of AI inferences in the cloud is not only undesirable, it is unsustainable [1]. The future of AI is hybrid and distributed, and this future demands the development of highly efficient devices capable of offloading AI inferences to the edge. Integrating custom designed AI coprocessors with CPU cores in heterogeneous processing SoCs specifically tailored to target applications is a strong driver in the semiconductor market for AI today [2].

Last century, development of custom CPU microarchitectures and customisation of instruction sets was hindered by most available Instruction Set Architectures (ISAs) being paywalled. Hence, achieving a royalty-free design involved not only designing the microarchitecture but also building the software tooling for compilation to some custom ISA and for verification of the whole ecosystem. As a result of such effort, in the early 2000s very few production-ready open source CPUs existed, some of the most notable being the OpenSPARC [3] LEON [4] designed by the European Space Agency (ESA) and Gaisler Research, and the OpenRISC [5] ecosystem by OpenCores [6].

Since mid 2000s to 2010s multiple ISAs were made available royalty-free, such as, Power [7], SuperH [8], or RISC-V [9]. In some cases, original patents expired and/or copyright holders made existing specifications open. Conversely, RISC-V was conceived to be an open standard ISA since the project began in 2010, designed from scratch based on established reduced instruction set computer (RISC) principles.

Availability of open and royalty-free ISAs eased the development of microarchitectures to fit the requirements of application-specific tasks, allowing software tooling development and maintenance to be done collaboratively. Still, integrating custom hardware coprocessors with a CPU is beyond having a working microarchitecture and the ISA related software support. In the RISC-V ecosystem, communications are not limited to memory-mapped and stream interfaces. In 2019, Google [10] proposed the CFU concept [11] and integrated it into VexRiscv [12] CPU. Thereinafter, some others CPU microarchitectures have added this feature, such as NEORV32 [13]. Based on this work, there is a draft to specify Custom Function/Extension Units (CFUs/CXUs) [14] for tight software-hardware integration.

In this paper we present the results obtained from a set of experiments to accurately evaluate the processing performance of a RISC-V based SoC prototyped on FPGA by integrating simple coprocessing cores attached through different data communication modes, including memory-mapped and stream interfaces and custom ISA extensions. The whole design and verification tooling setup, from design entry to implementation, from functional simulation to physical data processing was performed using FLOS EDA tools. Obtained results will be used to propose efficient SoC designs for edge AI execution using custom coprocessing cores.

In section II the open source microarchitecture ecosystem is presented and the selection of the target CPU design is explained. In section III tools used for compilation, simulation,

This work was partially supported by Union Europea-NextGenerationEU through the Cátedras Chip program SOC4SENSING TSI-069100-2023-0004, by the Basque Government under grant KK-2023/00090, and by the Spanish Ministry of Science and Innovation under grant PID2020-115375RB-I00.
979-8-3503-6439-2/24/$31.00 ©2024 IEEE

synthesis and communication are explained. In section IV each of the four communication mechanisms is throroughly characterized. In section V results and future work are summarized.

II. RISC-V ECOSYSTEM AND SOC DESIGN

The use of an open standard ISA offers the possibility of describing soft-core CPUs and microcontroller-like SoC based on this architecture and share these designs freely and openly with the community, if the designer so wishes. In this way, users can contribute to the project, for example finding and fixing bugs and according to the features offered by a version control platform, such as GitHub [15] or GitLab [16], keeping the project active and subject to continuous improvement. In addition to this, one of the great advantages of using an open standard ISA is the possibility of implementing on an ASIC without the need to pay any type of royalties, contrary to what would happen with other closed architectures. There are multiple examples of open standard ISA based CPU projects, such as Microwatt [17] and Chiselwatt [18] which are based on Open POWER ISA. However, in this section we will focus on RISC-V based projects. At the moment, there are several RISC-V based CPU projects on GitHub, from microcontroller-sized to linux capable. These projects are described in scala, verilog, system verilog, VHDL and others HDLs. (e.g. Scala: Rocket Chip [19] and VexRiscv [12]; Verilog: PicoRV32 [20], Hummingbirdv2 E203 [21], DarkRISCV [22], Jasonlin316-RISC-V-CPU [23], RISCV-Atom [24] and RISC-V Steel [25]; VHDL: NEORV32 [13], ORCA RISC-V [26], Potato [27], RPU [28] and ReonV [29]).

Principally, we have two requirements. On the one hand, we need a description in VHDL, since our accelerators are in this language. On the other hand, we need the microcontroller to be provided with CFU, CFS and with a top of memory-mapped and stream. Of all these RISC-V projects, the one that meets our requirements is *NEORV32*. Additionally, it is equipped with an official Open Source RISC-V ID, 19 [30]. For these reasons, we have selected this project for the current work. The NEORV32 Processor [13] [31] [32] is an open source customizable microcontroller-like system on chip (SoC) built around the NEORV32 RISC-V CPU, described in platform-independent VHDL.

Figure 1 illustrates a Custom SoC Design composed of NEORV32 and different accelerators connected through various modes:

- SLINK: Stream Link (AXI4-Stream)
- XBUS: External Bus Interface (Wishbone/AXI4-Lite)
- Custom Functions Subsystem (CFS)
- Custom Functions Unit (CFU/CXU)

Stream Link is an interface to perform a stream transmission compatible with a subset of AXI4-Stream [33]. It provides independent unidirectional RX and TX channels for sending and receiving stream data. Each channel features a configurable internal FIFO to buffer stream data. XBUS is a general bus interface for attaching memory-mapped accelerators compatible with Wishbone [34] and AXI4-Lite [35]. An optional cache module *X-CACHE* can be enabled to improve memory

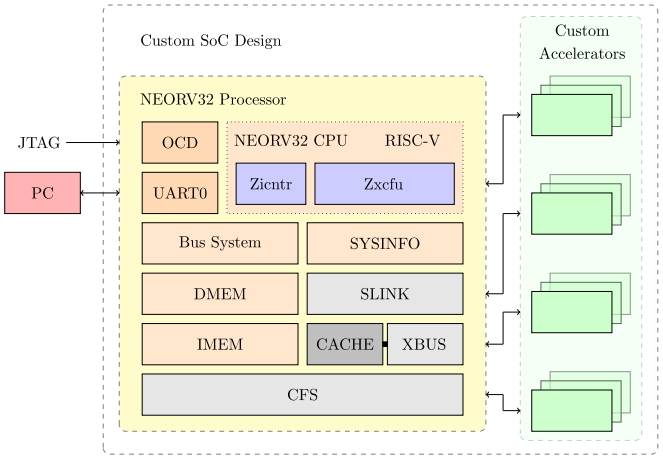

Fig. 1: Scheme of the Custom SoC Design.

access latency. Custom Functions Subsystem is an *empty* template for a memory-mapped, processor-internal module. It provides sixty-four 32-bit memory-mapped read/write registers. It should be noted that CFS does not have direct access to memory, all data (and control instruction) have to be send by the CPU. Custom Functions Unit is a functional unit that is integrated right into the CPU's pipeline. This was added by the NEORV32 developer based on CFU/CXU specification draft. It allows to implement custom RISC-V instructions. The instruction formats supported by NEORV32 are *R3-Type*, *R4-Type* and *R5-Type*. The first two types are a RISC-V standard and the last one is exclusive to NEORV32. Specifically, the first and the second type allows addressing two and three 32-bit input registers respectively. Besides, the function is selected through *funct7* and/or *funct3* in the first case and through *funct3* in the second case. The third type allows addressing four 32-bit input registers. Since it does not have the field *funct3* and/or *funct7* only two custom functions can be performed through this type, A format and B format. Figure 2 shows the 32-bit custom instruction types supported through Zxcfu NEORV32-specific ISA extension.

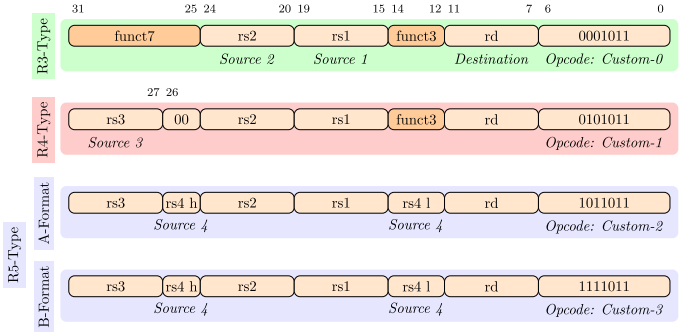

Fig. 2: CFU Custom Instruction Types.

TABLE I: Latency (L) and throughput (T) measurements performed by simulating: *VC*, the accelerator standalone along with Verification Components; and *Complex*, the whole SoC including NEORV32, the accelerator and software execution.

Accelerator		SLINK		XBUS		CFU	CFS
		VC	Complex	VC	Complex	Complex	Complex
Buffered	Not pipelined	Both	Both	Both	Both	L	Both
	Pipelined	Both	Both	Both	Both	L	Both
Unbuffered		L	Both	L	L	L	L

III. Workflow

FLOS and proprietary/commercial tools are not isolated ecosystems, on the contrary, in recent years, we have seen collaborative projects of open source tools integrated in vendor tools, such as RapidWright [36]. In addition to this, we have also seen a contest [37] sponsored by AMD [38] with the goal of promoting and demonstrating the FPGA Interchange Format [39] as an efficient and robust intermediate representation for working on backend FPGA problems, even at industrial scales. In the same line, Siemens [40] has observed a healthy growth between the Open Source VHDL Verification Methodology (OSVVM) [41] and the Universal VHDL Verification Methodology (UVVM) [42] since 2018, which in his own words "is encouraging" [43]. Therefore, in view of these events, we can affirm that traditional vendors are starting to facilitate the use of parts or all of FLOS allowing a hybrid future in the FPGA tools ecosystem.

Figure 3 illustrates the workflows for build, simulation, synthesis, place and root and generate bitstream through FLOS tools and proprietary tools. In this way, the implementation has been successfully tested on Arty A7 35t/100t FPGAs for all accelerators. For this purpose, the bitstream is generated, one the one hand, using *Vivado* [44] and on the other hand, using the following container [45]. This container is built and pushed in continuous integration and contains *GHDL* [46], *yosys* [47], *nextpnr-xilinx* [48] and *prjxray* [49]. It should be noted that the simulation framework allows the use of *ModelSim/QuestaSim* [50].

IV. Performance characterization

The performance characterization has been carried out using *VUnit* verification framework [51]. Since custom accelerators can differ in their internal characteristics, we have tested three designs with different latency and throughput ratios. The defining characteristics of each accelerator are as follows:

- Buffered not pipelined
- Buffered pipelined
- Unbuffered

The tests that have been performed are summarized in Table I and can be divided into two stages. In the first stage each accelerator has been independently tested in simulation with *VUnit Verification Components (VC)* for the Slink and XBUS connection modes. In the second stage the integrating design test bench including the whole SoC (NEORV32, accelerator and software execution) has been tested in simulation for the four selected modes of connection.

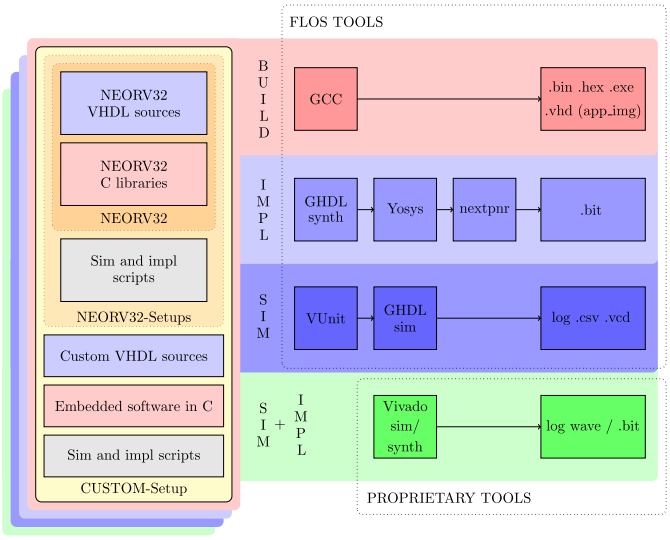

Fig. 3: Workflow of the Custom-Setup.

A. Measurement methodology

To perform a transmission with NEORV32, each connection mode is associated with one or more CPU instructions. Is the running time involved in applying each of these instructions that is measured. For this purpose, four transmission are performed in each test.

In the case of latency measurement, the operations of sending from NEORV32 to the accelerator and receiving from the accelerator to NEORV32 are executed consecutively four times and the running time of each send/receive operation is measured in system clock cycles. In the case of throughput measurement, data are sent from NEORV32 to the accelerator. Then, the receive operation is executed four times and the running time of how many data are received is measured in data per system clock cycles. For this reason, to perform the throughput measurement the accelerator or the mode of connection must have a buffer to store the inputs data from NEORV32. Therefore, for the case of the unbuffered accelerator only the throughput measurement can be performed for the SLINK mode, since this mode has associated transmitter/receiver FIFOs. For the case of CFU mode only latency measurement can be performed because according to the internal characteristics of the custom instruction the sent/received operation is performed in a single step.

The methodology followed to realize the measurements with the whole SoC has been generalized for all tests and the process is graphically summarized in Figure 4. When

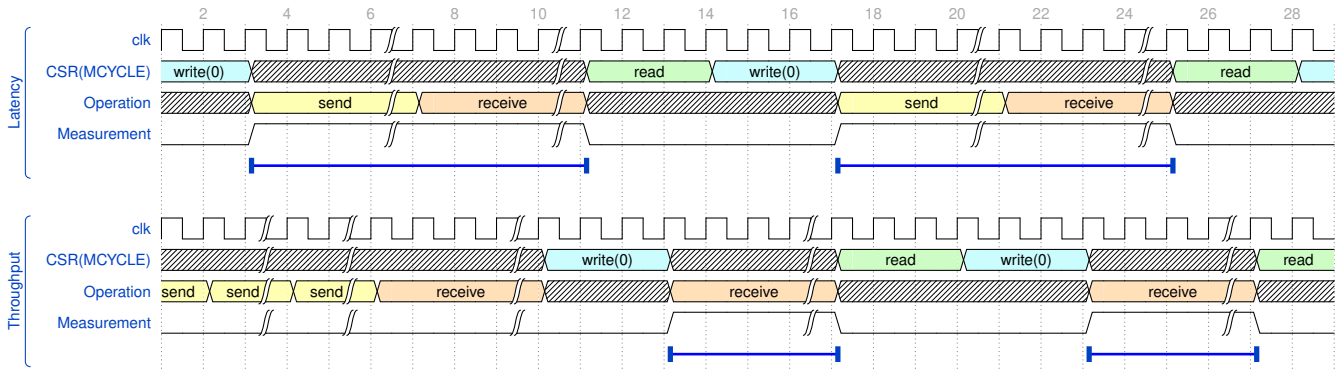

Fig. 4: Latency and throughput measurement process.

```
for x in 0 to test_items-1 loop
  wait until rising_edge(clk) and csr_we = '0' and
  csr_valid = '1' and csr_addr = x"B00" and
  csr_rdata_o /= x"00000000"; -- MCYCLE ADDR 0xB00
  info(logger, "Data " & to_string(x+1) & "/" &
  to_string(test_items) & " latency is " &
  to_string(to_integer(unsigned(csr_rdata_o))-1) &
  " cycles");
  wait until rising_edge(clk);
end loop;
```

Listing 1: VHDL code to extract *CSR(mcycle)* value.

the accelerators are attached to NEORV32 through the four selected modes several test benches are obtained. Each of them is associated with a C program that is compiled and loaded into the NEORV32 instruction memory. This program executes the instructions associated with each connection mode. To measure the execution time of these instructions the *control status register (CSR) mcycle* is used. In this way, whatever we want to measure is placed between `neorv32_cpu_csr_write(CSR_MCYCLE, 0)` instruction, which sets the register to 0 indicating the start of the measurement and `neorv32_cpu_csr_read(CSR_MCYCLE)` instruction, which reads the result indicating the end of the measurement.

The value of the *CSR(mcycle)* is extracted in simulation through the *VUnit info()* function adding Listing 1 to the test bench. Thus, we have automated the latency/throughput measurement every time the simulation of the whole SoC is launched visualizing the results of the measurements at the end of the simulation. It should be noted that the measurement result can be retried e.g. in CSV format using the *VUnit* logger for further processing.

As is shown in Listing 1, when *CSR(mcycle)* value is extracted, 1 is subtracted. This is due to the fact that internally `neorv32_cpu_csr_read(CSR_MCYCLE)` instruction adds one extra cycle to the measurement.

B. Measurement results

The latency and throughput measurement results are summarized in Table II. The entire workflow can be reproduced locally, since all the code and simulation/implementation scripts are available in GitHub. Henceforth, we will discuss the obtained results for each connection mode.

First, for the SLINK Complex test denotes the high latency compared to the SLINK (AXI-Stream) Verification Components test. To perform a send/receive operation with NEORV32 the functions `neorv32_slink_put` and `neorv32_slink_get` are used, respectively. These functions involve moving data through the associated TX/RX FIFOs, which slows down the transmission.

Second, for the XBUS (Wishbone) Verification Component test it should be clarified that it is measured from send acknowledge to receive acknowledge. Normally, XBUS communication takes two cycles between setting the strobe and receiving the acknowledge. In this context, the accelerator starts to operate when the strobe is received, masking one cycle in the measurement compared to SLINK (AXI-Stream) Verification Component test. To perform a send/receive operation with NEORV32 the functions `neorv32_cpu_store_unsigned_word` and `neorv32_cpu_load_unsigned_word` are used, respectively. In the latency case, the first transmission takes two cycles more than the rest because the compiler moves the XBUS address to an immediate register and since the address does not change, this move operation is skipped for all other transmissions. The most restrictive measurement is taken into account in the presentation of the results. Figure 5 illustrates graphically the transmission through XBUS for the buffered pipelined accelerator throughput test.

Third, for the CFU test three custom instructions *R3-type* are defined, one for each accelerator. `funct3=000` for buffered not pipelined, `funct3=001` for buffered pipelined and `funct3=010` for unbuffered. To perform a send/receive operation the `neorv32_cfu_r3_instr(funct7, funct3, rs1, rs2)` function is used. In this function the two 16-bit input operators are contained in *rs1* and the 32-bit result is stored in *rd*. When the *R3-type* custom instruction is executed, the operation of sending the content of *rs1* to

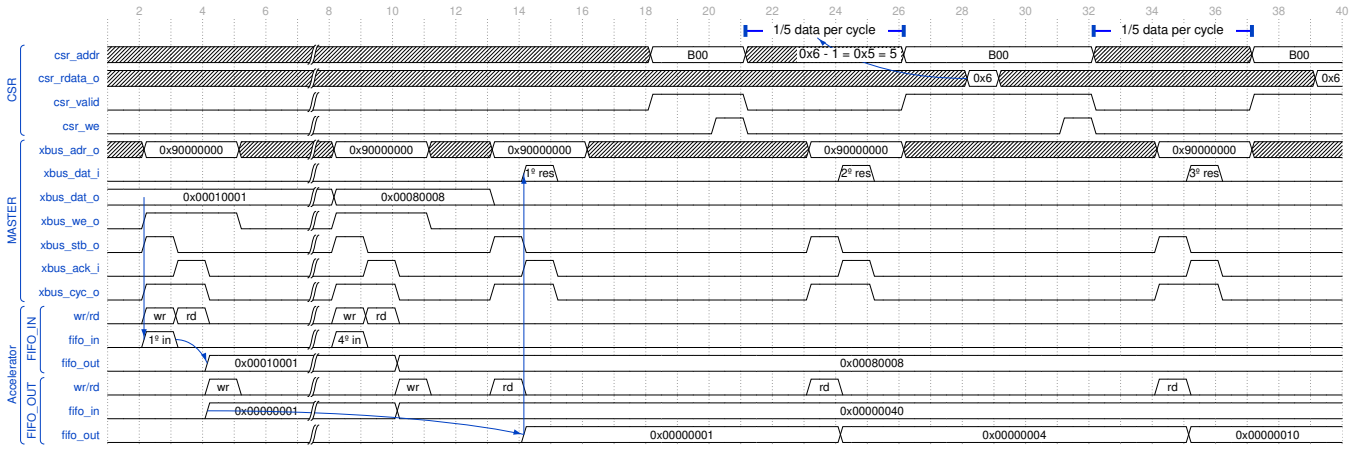

Fig. 5: Waveform of XBUS throughput for buffered pipelined accelerator.

the accelerator and receiving from the accelerator storing the result in *rd* is done consecutively in a single step. Figure 6 illustrates graphically the transmission through CFU for the buffered not pipelined accelerator latency test.

Finally, for the CFS test two situations can be distinguished. In the case of buffered accelerators, two of the sixty-four memory-mapped registers associated with CFS are used. One register to write/read the input/output data and another one to control the accelerator write/read signals. Therefore, to perform a send operation four write operations are needed: write the accelerator input data, write the accelerator write signal, write the accelerator read signal and clean the control register writing a zero in it. To perform a receive operation just one read operation is needed. In the case of unbuffered accelerator, only one memory-mapped register is used to write/read the input/output data. Therefore, to perform a send/receive operation just one write/read operation is needed. This is why the difference between buffered accelerators and unbuffered accelerator is so significant. Also, as in the case of the XBUS mode, the compiler adds extra instructions in the first transmission in order to move data to immediate registers. Again, the most restrictive measurement is taken into account in the presentation of the results.

V. Conclusion

In view of these results we can conclude that CFU is the connection mode that offers the lowest latency, between eight and thirteen system clock cycles depending on the accelerator type. In addition to this, the lowest throughput is obtained with the XBUS mode, one-fifth data per system clock cycle for both accelerators. It is relevant to note that depending on the connection mode, the internal architecture of the accelerator does or does not affect the latency/throughput measurement. This fact can be decisive when the connection mode is selected because according to the internal characteristics of the custom accelerator the transmission performance may be affected. Therefore, for coprocessors with low internal latency, the efficiency is improved through the CFU connection mode, but once the internal latency of the coprocessor is increased, the performance of this connection mode is closer to other modes. Furthermore, if the coprocessor is buffered and we are interested in receiving the highest possible data throughput, the most interesting connection mode would be through XBUS.

The results obtained in this work have allowed to establish which connection mode is the most efficient to attach coprocessors to RISC-V based processors, specifically for NEORV32 case. This information is relevant to us because we are working on optimal integration of different accelerators for efficient execution of AI models, for example, accelerating activation function calculations.

Acknowledgement

Thanks to NEORV32 author Stephan Nolting for his work in Open Source hardware development and documentation, as well as his dedication to keeping the project in constant evolution and up-to-date. His explanations and clarifications have been very useful for the elaboration of this work.

TABLE II: Measurement results: latency (*L*, system clock cycles) and throughput (*T*, data per system clock cycle).

	Accelerator		SLINK		XBUS		CFU	CFS
			VC	Complex	VC	Complex	Complex	Complex
L	Buffered	Not pipelined	6	45	5	16	13	37
		Pipelined	4	45	3	16	11	37
	Unbuffered		1	45	2	16	8	18
T	Buffered	Not pipelined	1/4	1/20	1/2	1/5	X	1/15
		Pipelined	1	1/20	1/2	1/5	X	1/15
	Unbuffered		X	1/20	X	X	X	X

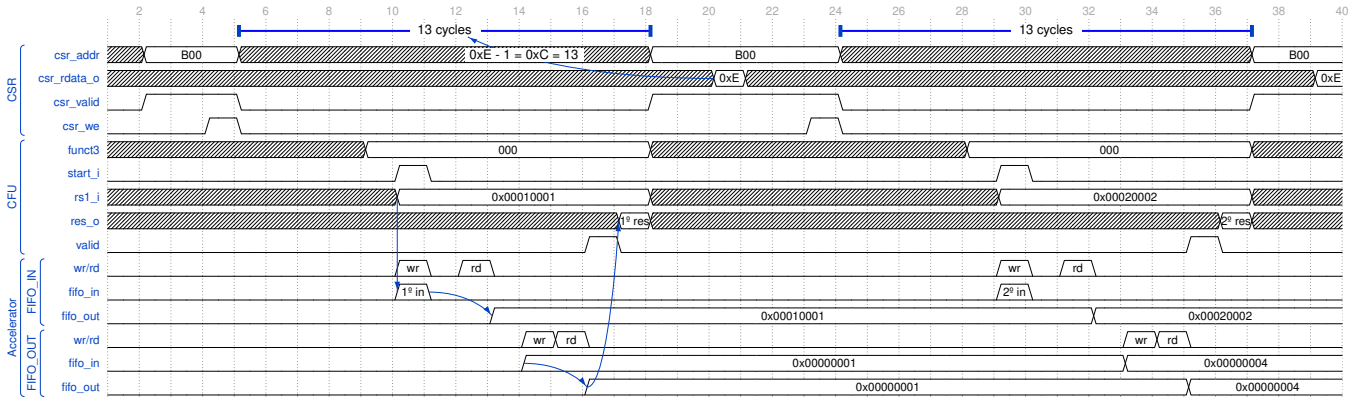

Fig. 6: Waveform of CFU latency for buffered not pipelined accelerator.

REFERENCES

[1] M. M. H. Shuvo, S. K. Islam, J. Cheng, and B. I. Morshed, "Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review," *Proceedings of the IEEE*, vol. 111, no. 1, pp. 42–91, 2023.

[2] S. Duan, D. Wang, J. Ren, F. Lyu, Y. Zhang, H. Wu, and X. Shen, "Distributed Artificial Intelligence Empowered by End-Edge-Cloud Computing: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 591–624, 2023.

[3] Sun Microsystems, Inc., "OpenSPARC." oracle.com/servers/technologies/opensparc-overview, 2005.

[4] European Space Research and Technology Centre, "LEON." esa.int/LEON_the_space_chip_that_Europe_built, 1997.

[5] D. Lampret and the OpenRISC Community, "OpenRISC." gh:openrisc, 2000.

[6] OpenCores Community, "OpenCores." opencores.org, 1999.

[7] OpenPOWER Foundation, "Power ISA." openpower.foundation, 2006.

[8] Hitachi/Renesas, "SuperH." renesas.com/us/en/products/microcontrollers-microprocessors/other-mcus-mpus/superh-risc-engine-family-mcus, 1992.

[9] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. Patterson, and K. Asanovic, "The RISC-V instruction set," in *2013 IEEE Hot Chips 25 Symposium (HCS)*, pp. 1–1, 2013.

[10] Google, "Google." about.google, 1998.

[11] S. Prakash, T. Callahan, J. Bushagour, C. Banbury, A. V. Green, P. Warden, T. Ansell, and V. J. Reddi, "CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (TinyML) Acceleration on FPGAs," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, Apr. 2023.

[12] C. Papon and Contributors, "VexRiscv - A FPGA friendly 32 bit RISC-V CPU." gh:SpinalHDL/VexRiscv, 2016.

[13] S. Nolting and Contributors, "NEORV32 - A tiny, customizable and extensible MCU-class 32-bit RISC-V soft-core CPU and microcontroller-like SoC." gh:stnolting/neorv32, 2024.

[14] J. Gray, "Draft Proposed RISC-V Composable Custom Extensions Specification." raw.githubusercontent.com/grayresearch/CFU/main/spec/spec.pdf, 2019.

[15] Microsoft Corporation, "GitHub." github.com, 2008.

[16] GitLab Inc., "GitLab." gitlab.com, 2011.

[17] A. Blanchard and Contributors, "Microwatt - A tiny Open POWER ISA softcore." gh:antonblanchard/microwatt, 2019.

[18] A. Blanchard and Contributors, "Chiselwatt - A tiny POWER Open ISA soft processor." gh:antonblanchard/chiselwatt, 2019.

[19] A. Waterman and Contributors, "Rocket Chip." gh:chipsalliance/rocket-chip, 2014.

[20] C. Xenia Wolf and Contributors, "PicoRV32 - A Size-Optimized RISC-V CPU." gh:YosysHQ/picorv32, 2021.

[21] Nuclei System Technology, "Hummingbirdv2 E203 Core and SoC." gh:riscv-mcu/e203_hbirdv2, 2020.

[22] M. Samsoniuk, "DarkRISCV." gh:darklife/darkriscv, 2018.

[23] Y.-C. Lin, "RISC-V CPU (Tape-Out with U18 Technology)." gh:jasonlin316/RISC-V-CPU, 2019.

[24] S. Singh, "RISCV-Atom soft-core processor." gh:saursin/riscv-atom, 2021.

[25] R. Calcada and contributors, "RISC-V Steel." gh:riscv-steel/riscv-steel, 2024.

[26] J. Vandergriendt, "ORCA RISC-V RV32IM core." gh:kammoh/ORCA-risc-v, 2015.

[27] K. Klomsten Skordal, "The Potato Processor." gh:skordal/potato, 2015.

[28] C. Riley, "RPU - Basic RISC-V CPU." gh:Domipheus/RPU, 2020.

[29] L. Castro, "ReonV RISC-V." gh:lcbcFoo/ReonV, 2018.

[30] RISC-V Community, "Open-Source RISC-V Architecture IDs." gh:riscv/riscv-isa-manual/blob/main/marchid.md, 2024.

[31] S. Nolting and Contributors, "The NEORV32 RISC-V Processor Datasheet." stnolting.github.io/neorv32/, 2020.

[32] S. Nolting and Contributors, "The NEORV32 RISC-V Processor User Guide." stnolting.github.io/neorv32/ug/, 2020.

[33] ARM, "AXI4-Stream." developer.arm.com/documentation/ihi0051/latest/, 2010.

[34] OpenCores, "Wishbone Bus." cdn.opencores.org/downloads/wbspec_b4.pdf, 2010.

[35] ARM, "AXI4-Lite." developer.arm.com/documentation/ihi0022/e/, 2010.

[36] AMD Research and Advanced Development, "RapidWright." gh:Xilinx/RapidWright, 2018.

[37] AMD, "Runtime-First FPGA Interchange Routing Contes." xilinx.github.io/fpga24_routing_contest/index, 2024.

[38] AMD, "Advanced Micro Devices, Inc.." amd.com, 1969.

[39] CHIPS Alliance, "FPGA Interchange Format." rapidwright.io/docs/FPGA_Interchange_Format, 2020.

[40] Siemens, "Siemens." siemens.com/global, 1847.

[41] J. Lewis and Contributors, "Open Source VHDL Verification Methodology." osvvm.org, 2013.

[42] E. Tallaksen and Contributors, "Universal VHDL Verification Methodology ." uvvm.org, 2013.

[43] Siemens, "The 2022 Wilson Research Group Functional Verification Study." blogs.sw.siemens.com/verificationhorizons/2022/11/21/part-6-the-2022-wilson-research-group-functional-verification-study/, 2022.

[44] Xilinx, "Vivado Design Suite." xilinx.com/products/design-tools/vivado, 2012.

[45] U. Sainz-Estebanez, "Ghdl + yosys + ghdl yosys plugin + nextpnr-xilinx + prjxray container." ghcr.io/unike267/containers/impl-arty:latest, 2024.

[46] T. Gingold and Contributors, "GHDL - VHDL 2008/93/87 simulator." gh:ghdl/ghdl, 2024.

[47] C. Xenia Wolf and Contributors, "YOSYS - Yosys Open SYnthesis Suite." gh:YosysHQ/yosys, 2020.

[48] D. Shah and Contributors, "Nextpnr-Xilinx - Experimental flows using nextpnr for Xilinx devices." gh:gatecat/nextpnr-xilinx, 2020.

[49] Project X-Ray Contributors, "Documenting the Xilinx 7-series bitstream format." gh:f4pga/prjxray, 2020.

[50] Siemens, "Questa advanced simulator." eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/, 2011.

[51] L. Asplund, O. Kraigher, and Contributors, "VUnit - Testing framework for VHDL/SystemVerilog." gh:VUnit/vunit, 2024.