Design and Development of an FPGA-Based Real-Time Reconfigurable Computing Platform

Jayaraj U. Kidav and Varun Mohan

Abstract The computational requirements in modern workloads like artificial intelligence (AI), machine learning (ML), etc., demand the necessity of hardware acceleration and partial reconfiguration (PR) on a platform like a field programmable gate array (FPGA). In this work, the extendibility and modularity of reduced instruction set computer (RISC)-V® instruction set architecture (ISA) are combined with the dynamic partial reconfigurable features of contemporary FPGAs, to develop new hardware accelerated computing architecture. The architecture executes the extended instruction as a coprocessor implementation, and the necessary coprocessor logic will be loaded into the PR region in real time. The work also explores the idea of having multiple PR regions and caching frequently used coprocessor logic and evaluates its impact on system performance.

Keywords FPGA · RISC-V® · Partial reconfiguration · Hardware acceleration · Coprocessors

1 Introduction

Hardware acceleration refers to the process by which an application will offload specific workloads to specialized hardware components within the system enabling higher speed and efficiency. It can be utilized to obtain better performance than is possible with software running on general-purpose CPUs alone. Hardware acceleration combines the pliability of CPUs, with the efficiency and performance of fully custom hardware, like FPGAs and ASICs or even GPUs [1, 2]. There are numerous types of specialized hardware acceleration systems. One of the most popular is an FPGA, or field programmable gate array, an IC chip that allows for the majority of the device's logical functionality to be modified even after it has been sent to customers

J. U. Kidav (✉) · V. Mohan
National Institute of Electronics and Information Technology, Calicut, Kerala, India
e-mail: jayaraj@calicut.nielit.in

S. N. Singh et al. (eds.), *Proceedings of the NIELIT's International Conference on Communication, Electronics and Digital Technology*, Lecture Notes in Networks and Systems 676, https://doi.org/10.1007/978-981-99-1699-3_24

and is in use. Design engineers may adapt to new standards or needs with the help of these powerful devices, which can be adapted to speed up important workloads.

Modern FPGAs support a feature called partial reconfiguration (PR). Unlike the usual method of configuring FPGAs where the entire FPGA is programmed every time, partial reconfiguration partitions the FPGA into static and dynamically reconfigurable regions at design time. The partial regions can have their bitstreams generated and can be configured without affecting the working of the static regions. This new feature allows much more flexibility in FPGA-based designs. Heterogeneous architectures involving reconfigurable computing are gaining more and more popularity [1–4]. Recent research works [5–8] suggest that modern workloads like AI/ML, signal processing, etc., can take advantage of partial reconfiguration, specifically real-time partial reconfiguration. Also, it is evident that the presence of reconfigurable systems in the future computing platforms are going to increase further [9, 10].

With ISAs like RISC-V® gaining popularity, researchers are now trying to combine the extensibility of the ISA with the flexibility of partial reconfigurations to design new and intuitive architectures for modern compute workloads [9–11]. Following are the main points identified from research papers that presented similar work, especially heterogeneous architectures involving RISC-V® CPU cores tightly coupled with coprocessors implemented on partially reconfigurable regions. The work presented in [11] describes an idea which in contrast to a fully static design, this work implements a reconfigurable coprocessor coupled to a RISC-V® soft-core processor, enhancing the performance of various encryption/decryption algorithms using fewer FPGA resources. The authors were able to achieve hardware acceleration viz. the speedup of the coprocessors and switching the configurations during execution. But the solution presented is not a generic FPGA solution. It uses Linux with modifications to the kernel and uses workarounds that the authors themselves confirm to be not secure. Also, there is no notion of multiple PR regions and their management or caching of bitstreams. It is not a fully hardware solution with software support implemented in the Linux Kernel. In the conclusions, the authors mention how techniques such as configuration prefetching and reuse could be explored to reduce the reconfiguration time for programs that require more than one coprocessor.

An all-open-source system for integrating FPGAs into RISC-V® CPUs is presented in paper [12]. A custom embedded FPGA (eFPGA) is connected directly to a RISC-V® CPU and allows for run-time instruction swapping by providing partial reconfiguration. The research is based on a custom application specific integrated circuit (ASIC) with their custom eFPGA fabric. So again, it is not a generic solution that could be applied to off the shelf FPGAs. The concentration is more on the custom ASIC design flow for integrating the core and the eFPGA on silicon. Further, delays of the coprocessors are handled in the instructions rather than coprocessor|cpu handshakes. The mentioned work uses multiple PR regions but does not consider caching of bitstreams. In the conclusion, the authors reiterated the performance improvement that can be achieved by the use of reconfigurable custom instructions.

In [13], a framework is presented that makes use of an embedded processor that is tightly connected with the programmable logic found inside a modern multiprocessor system on chip (MPSoC) like the Xilinx® Zynq platform. This concept's

fundamental goal is to implement the software's sequential control flow for applications, while it is possible to use reconfigurable hardware accelerators on demand, to improve the performance of compute intensive work. The presented idea is implemented on the Zynq platform which has ARM® CPUs as hard IP-cores hence not applicable for generic FPGAs and also does not involve ISA extensions and using custom instructions. The coprocessors are interfaced using the advanced extensible interface (AXI) and not connected to the CPU execution unit, which means the identification of the need of the coprocessor and its loading is done via software interfaces. Needless the work [13] covers the need for proper abstraction in such complex designs for easy software programmability. Management of bitstream is identified as a problem area since a bitstream has to be generated per coprocessor per PR region. A method of relocation of bitstreams is used in the work which is not supported by FPGA vendors yet and hence not explored further. In the concluding remarks, the authors reiterate the gain in resource efficiency achieved due to time division multiplexing (partial real-time reconfigurations) of IP-cores (coprocessors).

The architecture described in [14] consists of a soft dual processor system enhanced with a 128-bit single instruction multiple data (SIMD) engine that is partially run-time reconfigurable. Custom SIMD instructions can be incorporated at run-time to speed up time-consuming kernels for media applications, in addition to standard operations like arithmetic or Boolean operations, which the SIMD engine can execute. The paper explores an implementation that only uses a single PR region hence there is no exploration of multiple PR regions and resulting multiple partial bitstreams and their management and caching of bitstreams. But the author was able to get the intended performance boost from the use of partial reconfiguration by implementing custom SIMD instructions, integrated at run-time (PR) to accelerate compute intensive kernels for media processing. Moreover, they suggest the scope of using additional custom vector instructions to accelerate a wider range of applications [14]. Presents an efficient controller for the internal configuration access port (ICAP) interface on Xilinx® Devices. Though parts of it are in hardware, major components of the controller are in software and are mainly targeted for the Xilinx® Zynq platform. The solution does not use RISC-V® or RISC-V® extensions as coprocessors. It has been used as a reference to implement the ICAP controller which is a component of the design presented in this paper.

Most of the current systems based on FPGA do not explore partial or real-time reconfiguration. Of the systems that currently implement such mechanisms, the programming methods and usage require modification to standard OS/Tools and/or hardware knowledge of the underlying hardware accelerator. Also, these solutions do not handle the use case of having multiple partial reconfiguration regions or the caching of bitstreams and loads bitstreams on every access incurring the overhead of FPGA reconfiguration.

2 Methodology

2.1 RISC-V® ISA and PICORV32 Core [15]

RISC-V® is a free and open source instruction set architecture that has been gaining popularity recently. Apart from free and being open to using, the ISA is designed to be very extensible. RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. The ISA is designed to be modular and consists of several other groups of instructions also called Extensions which can be enabled or disabled as needed. The base integer instruction set is called the RV32I and consists of around 40 instructions. The type of instructions generated while compiling a C program depends on flags that are passed to the compiler. The compiler can be directed to generate instructions only from the base Integer instruction set or use one or more of the extensions as needed.

The PicoRV32 is a simple FPGA targeted soft core that implements RISC-V® RV32IMC Instruction Set. The integer multiplication and division extensions are implemented as coprocessors using the unique PCPI interface provided by the core. So strictly speaking, the core only implements the base Integer ISA. When the core encounters any other instruction that is not implemented, the instruction is presented on the PCPI interface, which can then be used to implement the logic for processing that instruction and waits until a result is returned to it. This interface is used as the entry point for the designed platform. Figure 1 explains the instruction execution flow of the PicoRV32 core when executing an instruction from the base integer ISA or one from the many extensions.

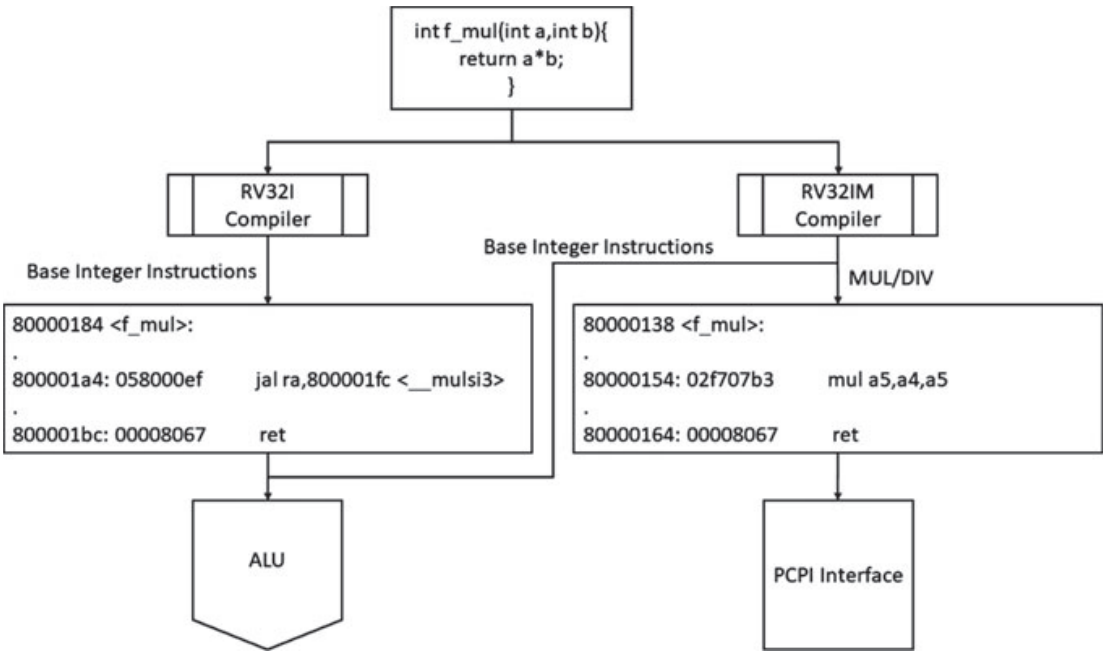

Fig. 1 PicoRV32 instruction execution

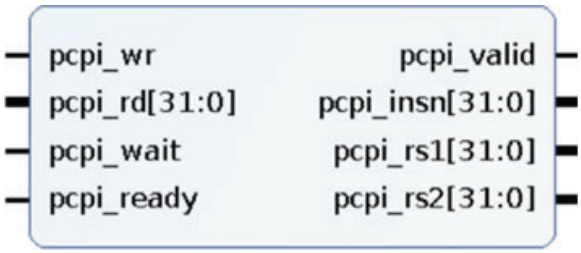

Fig. 2 Core PCPI interface

2.2 PICO Coprocessor Interface (PCPI)

The Pico coprocessor interface (PCPI) (Fig. 2) of the PicoRV32 RISC-V® core can be used to implement nonbranching instructions in external coprocessors. The instruction word itself is output on pcpi_insn when an unsupported instruction is detected. The rs1 and rs2 fields are decoded, and the values in those registers are output on pcpi_rs1 and pcpi_rs2. An external PCPI core can then decode the instruction, execute it, and assert pcpi_ready when execution has finished. The external core can ask the CPU to wait for the result by asserting the pcpi_wait signal.

2.3 XILINX Partial Reconfiguration and ICAP Interface

Xilinx partial reconfiguration or dynamic function exchange (DFX) allows for the reconfiguration of parts within an active functional design. Multiple configurations must be implemented for this flow to work, and as a result, whole bitstreams for each configuration and partial bitstreams for each reconfigurable module are produced (RM). Partial bit files can be downloaded to change the FPGA's reconfigurable areas after a full bit file configures it, protecting the integrity of the designs operating on the device's other sections that aren't being changed.

Since partial bitstreams are entirely self-contained, they are sent to the correct configuration port. Similar to full configuration bitstreams, these bitstreams include all addressing, header, and footer information. The partial bitstreams can be delivered to the FPGA through any external non-master configuration mode, such as JTAG, slave serial, or slave select map as well as the internal configuration access through the ICAP. The select map interface used to configure the FPGA has an internal counterpart called the ICAP. In a user design, this port can be instantiated and used to load any bitstream onto the FPGA. The port also provides information about the success and status of the current configuration process.

2.4 Main Flow Chart

The main flow chart of the controller is presented in Fig. 3. The idea is to run the application in the RISC-V® core with the parts requiring hardware acceleration implemented as coprocessors accessible via the PCPI interface. The designed platform sits between the core's PCPI interface and the PCPI MUX, thus being able to identify requests for coprocessors. The RISC-V® compiler flags are used to generate suitable instruction from the ISA extensions whose functionalities are implemented in the coprocessors. The coprocessors are designed and applied through the partial reconfiguration flow to generate their individual partial bitstreams corresponding to each PR region. These bitstreams then pass through an automated software flow where their binary data is added as byte arrays into a C header file using a python script which also generates an initialization function that can automatically configure the main controller registers with the start address and length of all the individual partial bit streams.

When the core invokes a coprocessor, it asserts the pcpi_ready signal which acts as the start signal for the main controller. It first checks whether the opcode is implemented in a coprocessor. If not it goes to the error state else it checks if the coprocessor that can execute that opcode is already loaded in a PR region. If yes, the controller simply updates the least frequently used (LFU) counters that are used for caching

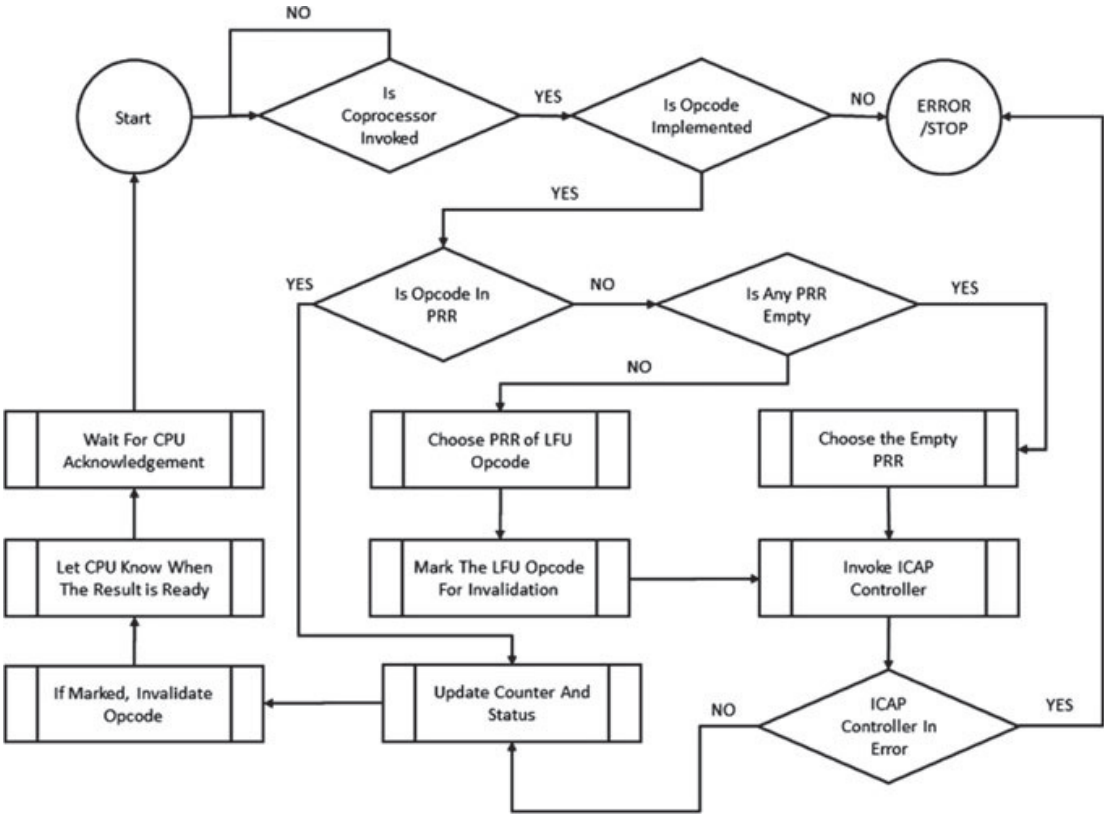

Fig. 3 Controller main flow chart

and lets the CPU know when the coprocessor is ready with the result, via the PCPI interface and then waits for the next cycle.

If not, the controller checks whether any of the PR regions is free. If yes, it chooses that PR region and then the bitstream of the required coprocessor for that PR region. The start address of that bitstream in memory and its length is loaded into the ICAP controller and a command is given to start the partial reconfiguration. The controller then monitors the output of the ICAP controller for failures. If everything is ok the hardware accelerator is now operational, and the controller also updates the LFU counters as well as other status registers for the current opcode. Once the result is ready, the controller can assert it to the core via the PCPI interface. If the ICAP operation was not successful, the system goes into the error state. If there are no PR regions free, the process is similar to the above with one difference that the caching logic is consulted to obtain the PR number for the least frequently used coprocessor currently configured.

2.5 Caching Mechanism

Figure 4 depicts the flowchart of the caching mechanism implemented in the controller. With a fixed number of PR regions, the platform can bring in and out the partial bitstreams as and when necessary. The platform uses LFU caching technique to keep the most frequently used coprocessors always configured so that the reconfiguration overhead can be avoided. If all the PR regions are currently occupied and the platform has to bring in a new coprocessor, it chooses the least frequently used coprocessor among the currently configured ones for replacement.

The caching logic works by maintaining access counters for each implemented opcode. The counters are incremented whenever the platform is requested by the CPU. The cache logic uses these counters to determine the opcode that was used the least number of times (Fig. 4). The output of the process is the current valid opcode, i.e., the opcodes whose implementations are currently loaded in the PR regions that have the least number of accesses. This opcode is then passed to the opcode to the PRR mapping register to obtain the PRR number in which the LFU opcode is located. The main controller can then use this PRR number and the opcode which is currently requested to determine the start address and length of the partial bitstream and then pass it to the ICAP controller for the FPGA reconfiguration.

3 Results

The platform was successfully designed and implemented on the Xilinx® ARTY A7 100 T FPGA evaluation board. The board hosts a 16 MB QSPI Flash which was used to permanently store the partial and static bitstreams as well as the application executable and linkable format (ELF) file. It also features 256 MB DDR3L memory,

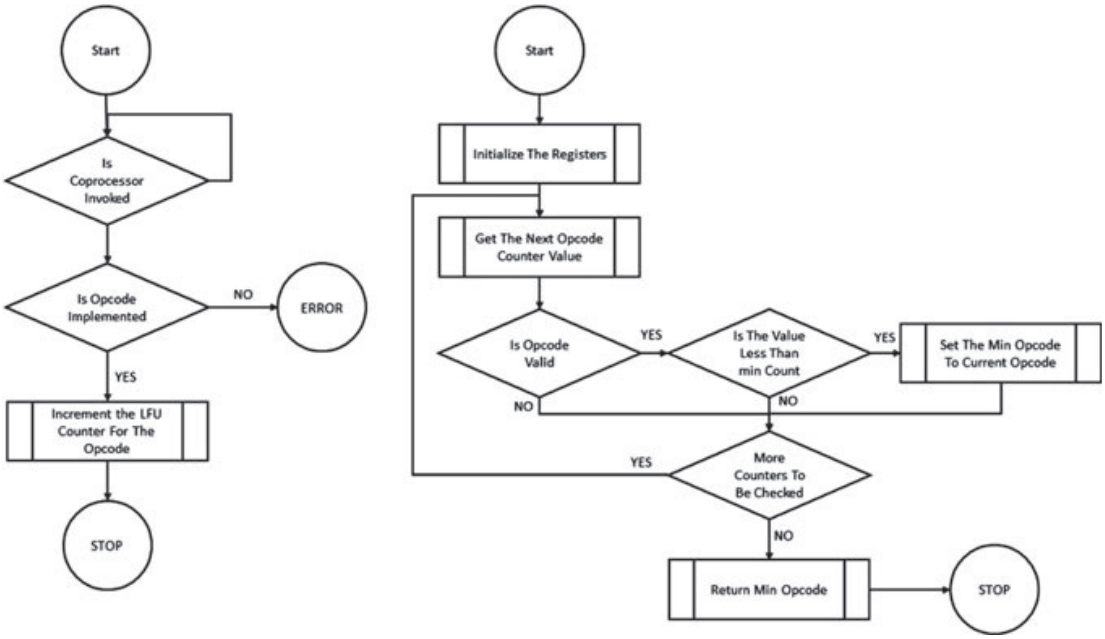

Fig. 4 Caching logic flow chart

which was used to keep the bitstreams and ELF for faster access. A fully automated and integrated flow for generating the partial bitstreams, the corresponding C header files, and the final application ELF was developed, implemented, and tested successfully. The platform is easily programmable using the standard C language and the GNU RISC-V® GNU toolchain.

3.1 Real-Time Reconfigurations

The platform was implemented with two PR regions that can be loaded with coprocessors. It was able to identify the need for a coprocessor on the fly and load it in real time. The platform used LFU cache replacement policy to keep the most used coprocessor always loaded in the PR regions. Actual output via UART from the FPGA implementation is shown below. The outputs of running the tests are shown in Figs. 5 and 6. The integer multiplication and division extension of the RISC-V® ISA was implemented as coprocessors for testing the platform. The result of the actual operation is also printed to show that the platform ensures the functionality of the coprocessors. The ANDN and XNOR instructions are part of the RISC-V® bit manipulation extension and are used here to simulate multiple coprocessors. Their functionality is not implemented and the output is always zero, which can be ignored. The output also dumps the opcode status register. '0' indicates that the opcode is currently not in any of the PR regions, while a '1' indicates that the coprocessor responsible for that opcode is configured in one of the available PR regions. Initially, all the opcode status is '0', indicating that none of the coprocessors are in the PR region. Then, a series of multiplication and XNOR operations (Fig. 5.) were

```
SREC SPI Bootloader
Reading flash ID
Succesfully read flash ID
Loading SREC image from flash @ address: 600000
Executing program starting at address: 80000000

Intializing!!!
Done!!!
MUL => 0 | DIV => 0 | ANDN => 0 | XNOR => 0 | OPSTAT=0

The result of 10*10 is : 100
MUL => 1 | DIV => 0 | ANDN => 0 | XNOR => 0 | OPSTAT=1

The result of 7*30 is : 210
MUL => 1 | DIV => 0 | ANDN => 0 | XNOR => 0 | OPSTAT=1

The result xnor is : 0
MUL => 1 | DIV => 0 | ANDN => 0 | XNOR => 1 | OPSTAT=9

The result xnor is : 0
MUL => 1 | DIV => 0 | ANDN => 0 | XNOR => 1 | OPSTAT=9

The result of 10*10 is : 100
MUL => 1 | DIV => 0 | ANDN => 0 | XNOR => 1 | OPSTAT=9
```

Fig. 5 Real-time reconfiguration output 1

executed. After these operations, the state of the opcode status register reveals that the platform on the fly identified the coprocessors for the MUL and XNOR instructions and loaded them into the PR regions. Moreover, the result of the MUL operation is as expected. It has to be also noted that at this point, the system has executed more MUL instructions compared to XNOR. Next, a new instruction DIV is executed. Here, the platform determines that it has to bring in a new coprocessor, but does not have a free PR region. In this case, the platform chooses the LFU opcode, XNOR for replacement. This is evident from the opcode status in Fig. 6 second line. Likewise, the other outputs can also be interpreted, suggesting the correct operation of real-time reconfiguration with caching.

3.2 Easy User Programmability

Hardware acceleration could be achieved by only changing a compiler flag without any change to the application code. With the program compiled using the compiler flag -march = rv32i, the instructions generated would only contain the RISC-V® base integer ISA, which are executed by the core itself and no coprocessor is invoked as seen in the opcode status dump in Fig. 7. All the statuses are '0', indicating none of them are currently loaded in the PR regions. With the program compiled by just changing the compiler flag to -march = rv32im, the compiler would then

```
The result of 50/2 is : 25
MUL => 1 | DIV => 1 | ANDN => 0 | XNOR => 0 | OPSTAT=3

The result of 22/2 is : 11
MUL => 1 | DIV => 1 | ANDN => 0 | XNOR => 0 | OPSTAT=3

The result xnor is : 0
MUL => 1 | DIV => 0 | ANDN => 0 | XNOR => 1 | OPSTAT=9

The result andn is : 0
MUL => 0 | DIV => 0 | ANDN => 1 | XNOR => 1 | OPSTAT=C

The result of 30*30 is : 900
MUL => 1 | DIV => 0 | ANDN => 0 | XNOR => 1 | OPSTAT=9

The result of 20*15 is : 300
MUL => 1 | DIV => 0 | ANDN => 0 | XNOR => 1 | OPSTAT=9

The result of 10*15 is : 150
MUL => 1 | DIV => 0 | ANDN => 0 | XNOR => 1 | OPSTAT=9

The result of 50/2 is : 25
MUL => 1 | DIV => 1 | ANDN => 0 | XNOR => 0 | OPSTAT=3

done
```

Fig. 6 Real-time reconfiguration output 2

```
Intializing!!!
Done!!!
MUL => 0 | DIV => 0 | ANDN => 0 | XNOR => 0 | OPSTAT=0

The result of 10*10 is : 100
MUL => 0 | DIV => 0 | ANDN => 0 | XNOR => 0 | OPSTAT=0

done
```

Fig. 7 Output with -march = rv32i

generate instructions from the integer multiplication extension if it encounters an integer multiplication. In this case, the platform was able to identify the need of the coprocessor, load it into the PR region and produce the correct results as well which is evident from Fig. 8. Thus, the platform was able to achieve hardware acceleration with absolutely no code change.

3.3 Caching

The platform was able to avoid reconfiguration overhead using caching. LFU policy was used to keep the most frequently used coprocessor in the PR region. Finding

```
Intializing!!!
Done!!!
MUL => 0 | DIV => 0 | ANDN => 0 | XNOR => 0 | OPSTAT=0

The result of 10*10 is : 100
MUL => 1 | DIV => 0 | ANDN => 0 | XNOR => 0 | OPSTAT=1

done
```

Fig. 8 Output with -march = rv32im

from Table 1 is represented graphically in Fig. 9 using logarithmic scale for better visibility. The cycle increase ratios refer to the factor by which the number of CPU cycles increase when the number of multiplications is increased ten folds. From the results, it is evident that for a system with a very low number of multiplications, the performance of the system is not much better than the one without hardware acceleration.

In fact, for a very low number of multiplications, the platform performs badly compared to the implementation without any acceleration. This is due to the overhead of partial reconfiguration out of which the main component now is reading of the partial bitstream binary from the DRAM. However, as the number of multiplications increases, the platform is able to catch up rapidly and cover up the overhead. From the table, it can be seen that when the performance for the platform was poor for 100 multiplication, with just a 10 times increase to 1000, the platform was able to perform 1.3 times that of the application when no hardware acceleration was in use. Finally, it was able to give the full potential acceleration from the hardware integer multiplier unit at around 100,000 multiplications. The threshold at which this happens entirely depends on the reconfiguration overhead. In systems with very high reconfiguration overhead due to power or clock frequency limitations or the bitstream has to be loaded from slow storage like a QSPI flash, the threshold can be extremely high to get any benefit of hardware acceleration from the platform. It is in this situation that the caching was found to be beneficial. With caching, the platform could keep the coprocessor always configured and ready avoiding the overhead of reconfiguration and providing the maximum available speedup provided by the hardware accelerator even for a low number of operations.

4 Conclusions

Successfully designed, implemented, and tested a RISC-V®-based real-time reconfigurable computing platform for hardware acceleration on FPGAs. The platform could identify coprocessors based on the currently fetched RISC-V® instruction and load it on the fly, on a need basis. The platform is programmable using C language and end users can easily take advantage of hardware acceleration using simple compiler

Table 1 Number of multiplications Vs CPU cycles

Number of multiplications	Without HW acceleration (-march=rv32i)		With HW acceleration (-march=rv32im)			With HW acceleration and caching	
	Number of cycles	Cycle increase ratio	Number of cycles	Cycle increase ratio	Speedup	Number of cycles	Speedup
100	456,283		1,761,754		0.258	62,811	7.26
1000	4,561,263	9.996	2,383,366	1.352	1.913	684,423	6.664
10,000	45,611,101	9.999	8,599,361	3.608	5.304	6,900,418	6.609
100,000	456,109,565	9.999	70,759,286	8.228	6.445	69,060,343	6.604
1,000,000	4,561,095,650	10	692,359,368	9.784	6.587	690,660,425	6.603
10,000,000	45,610,956,500	10	6,908,352,632	9.977	6.602	6,906,653,689	6.603

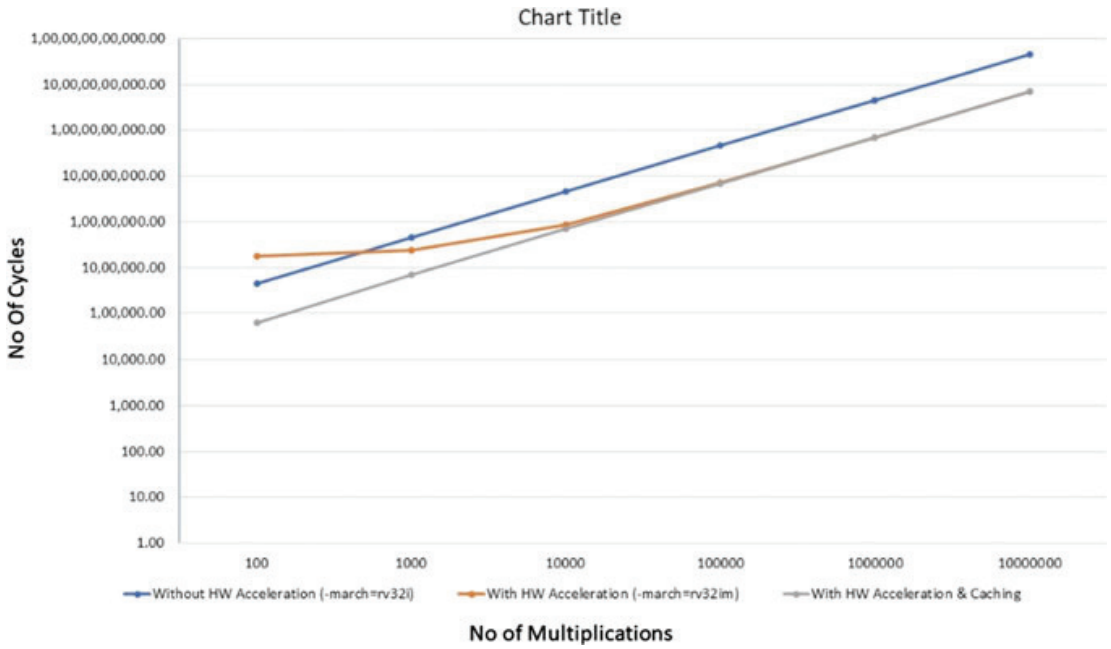

Fig. 9 Number of multiplications versus CPU cycles (log scale)

settings. A fully automated flow for generation of partial bitstreams and its management in flash and DRAM was developed, saving valuable design time and making the PR bitstream management automatic. Multiple PR regions with caching were implemented, and its performance benefits were analyzed. It was found that caching bitstreams is beneficial for avoiding reconfiguration overhead on frequently used hardware accelerators.

Increasing the performance of the ICAP controller can be explored further which at present is the biggest contributor to the reconfiguration overhead. In this work, we have used the LFU cache replacement policy, but the usability and efficiency of other cache replacement policies can also be evaluated. Currently, the system does not implement support for interrupts or exception handling of the CPU which can be designed. Handling error would ideally be done by tapping into the CPUs error handling/exception mechanism. Unfortunately, the RISC-V core used for POC in this paper don't have support for error handling / exception handling so it's beyond the scope of this paper. Also adding multicore CPUs and OS support and behavior of the system in modern pipelined, superscalar CPU designs including speculative execution can be investigated. The system does not handle error conditions at present. The implementation of errors as exceptions to the CPU can be explored.

References

1. Jindal A, Chadha M, Gerndt M, Frielinghaus J, Podolskiy V, Chen P (2021) Poster: function delivery network: extending serverless to heterogeneous computing. In: 2021 IEEE 41st International conference on distributed computing systems (ICDCS), 2021, pp 1128–1129. https://doi.org/10.1109/ICDCS51616.2021.00120

2. C. Hagleitneret al (2021) Heterogeneous computing systems for complex scientific discovery workflows. In: 2021 Design, automation and test in Europe conference and exhibition (DATE), 2021, pp 13–18.https://doi.org/10.23919/DATE51398.2021.9474061
3. Lee J et al (2021) An energy-efficient floating-point DNN processor using heterogeneous computing architecture with exponent-computing-in- memory. In: 2021 IEEE hot chips 33 symposium (HCS), 2021, pp 1–20. https://doi.org/10.1109/HCS52781.2021.9566881
4. D'Agostino D, Cesini D (2021) Editorial: heterogeneous computing for AI and big data in high energy physics. Front Big Data 4:652881. https://doi.org/10.3389/fdata.2021.652881
5. Nurmi J, Perera DG (2021) Intelligent cognitive radio architecture applying machine learning and reconfigurability. In: 2021 IEEE Nordic circuits and systems conference (NorCAS), 2021, pp 1–6.https://doi.org/10.1109/NorCAS53631.2021.9599858
6. Masadeh M, Elderhalli Y, Hasan O, Tahar S (2021) A quality-assured approximate hardware accelerators–based on machine learning and dynamic partial reconfiguration. J Emerg Technol Comput Syst 17(4):19, Article 57. https://doi.org/10.1145/3462329
7. Babu P, Parthasarathy E (2021) Reconfigurable FPGA architectures: a survey and applications. J Inst Eng India Ser B 102:143–156. https://doi.org/10.1007/s40031-020-00508-y
8. Nez N, Vilchez AN, Zohouri HR, Khavin O, Dasgupta S (2021) Dynamic neural accelerator for reconfigurable & energy-efficient neural network inference. In: 2021 IEEE hot chips 33 symposium (HCS), 2021, pp 1–21. https://doi.org/10.1109/HCS52781.2021.9566886
9. Bobda C, Mbongue JM, Chow P, Ewais M, Tarafdar N, Vega CM, Eguro K, Koch D, Handagala S, Leeser M, Herbordt M, Shahzad H, Hofste P, Ringlein B, Szefer J, Sanaullah A, Russell Tessier (2022) The future of FPGA acceleration indatacenters and the cloud. ACM Trans Reconfigurable Technol Syst 15(3):42 Article 34. https://doi.org/10.1145/3506713
10. Abdala Castro JW, Morales-Villanueva A (2021) Exploring dynamic partial reconfiguration in a tightly-coupled coprocessor attached to a RISC-V Soft-processor on a FPGA. In: 2021 IEEE XXVIII international conference on electronics, electrical engineering and computing (INTERCON), 2021, pp 1–4. https://doi.org/10.1109/INTERCON52678.2021.9532810
11. Dao N, Attwood A, Healy B, Koch D (2020) FlexBex: a RISC-V with a reconfigurable instruction extension. In: 2020 International conference on field-programmable technology (ICFPT), 2020, pp 190–195.https://doi.org/10.1109/ICFPT51103.2020.00034
12. Essig M, Ackermann KF (2017) On-demand instantiation of coprocessors on dynamically reconfigurable FPGAs. In: 2017 12th International symposium on reconfigurable communication-centric systems-on-chip (ReCoSoC), 2017, pp 1–8. https://doi.org/10.1109/ReCoSoC.2017.8016153
13. Ordaz JRG, Koch D (2018) A soft dual-processor system with a partially run-time reconfigurable shared 128-bit SIMD engine. In: 2018 IEEE 29th international conference on application-specific systems, architectures and processors (ASAP), 2018, pp 1–8. https://doi.org/10.1109/ASAP.2018.8445115
14. Vipin K, Fahmy SA (2014) ZyCAP: efficient partial reconfiguration management on the Xilinx Zynq. IEEE Embed Syst Lett 6(3):41–44. https://doi.org/10.1109/LES.2014.2314390
15. https:// github.com/YosysHQ/picorv32