Exploratory Research in HW based FPUs, it's linkage with CPU, Exploration of reconfiguration and it's Approaches.

02211 - Group 5

Mirza Jafar Ali Baig - s242545
Dept. of Applied Mathematics and Computer Science
Technical University of Denmark

René Antonio Hjort - s203880
Dept. of Applied Mathematics and Computer Science
Technical University of Denmark

Mathieu Astagneau - s243237
Dept. of Applied Mathematics and Computer Science
Technical University of Denmark

Jiayi Lu - s242648
Dept. of Applied Mathematics and Computer Science
Technical University of Denmark

Abstract* - *The use of floating-point units (FPUs) is increasingly important in low-power applications to enhance computational capabilities without significantly impacting energy consumption. Specifically, FPUs are becoming critical in areas such as machine learning inference on embedded devices, real-time signal processing, autonomous systems, and edge computing platforms. This paper presents the design and implementation of a reconfigurable floating-point unit (FPU) capable of dynamic reconfiguration, highlighting its advantages in terms of power, performance, and flexibility. The goal is to enable adaptable floating-point computation by allowing precision and functionality changes without full system reprogramming. This minimizes dead silicon compared to the alternative of having multiple dedicated FPUs. In this paper, the FPU is integrated into the Wildcat core architecture [1] and evaluated in testing environments to ensure robustness. Performance and resource utilization are then analyzed to assess trade-offs, with applications in reconfigurable computing platforms and embedded systems where flexibility and efficiency are critical.

***Index Terms*—FPGA, DSP, Reconfiguration, FPU, PPA assessment.**

I. Introduction

The increasing demand for high-performance yet power-efficient embedded and edge AI systems has renewed interest in floating-point computation. While FPUs offer flexibility and numerical range, their power and area cost often makes them impractical for constrained environments—leading designers to fall back on fixed-point arithmetic or slow software emulation. However, many modern workloads, such as neural network inference or signal processing, still require floating-point precision. To address this trade-off, we explore a HW based floating-point unit (FPU) interface with a CPU, made reconfigurable and implemented on FPGA hardware. This approach enables switching between FPU configurations (e.g., adders, multipliers, dividers) depending on application needs. This will improve power rating, performance, and area utilization due to the reconfiguration. These metrics are commonly termed Power, Performance and Area (PPA).The adaptive FPU is integrated into the Wildcat RISC-V CPU core [1], and the design is evaluated in terms of functionality, resource usage, and reconfiguration overhead.

Figure 1 provides a high-level overview of the system architecture expanded in this work. The focus is on the reconfigurable FPU and the combinational logic block (CL), which manages communication and switching between the reconfigurable FPU and data memory (DMEM).

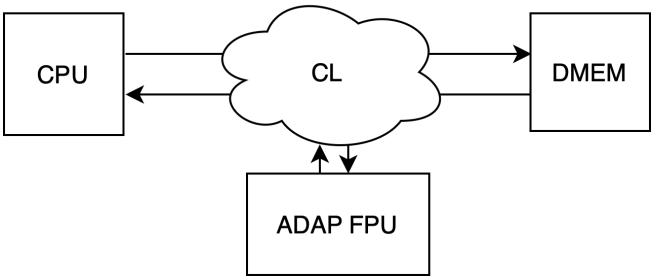

Fig. 1: Generic block diagram of the system proposed in this paper.

In this paper, we explore the design, integration, and evaluation of a HW based floating-point unit (FPU) within a RISC-V CPU architecture on an FPGA, making it reconfigurable and exploring approaches to make the reconfiguration dynamic. In other words, our focus is to explore runtime flexibility by allowing different floating-point operations to be swapped dynamically, aiming to improve power, performance, and area (PPA) efficiency. We further investigate the system's behavior, implementation challenges, and trade-offs, particularly in relation to reconfiguration overhead and future scalability.

A. Problem Definition

To realize a reconfigurable floating-point system, this paper focuses on exploring the following:

- Integrating a FPU with the Wildcat RISC-V CPU core.
- Designing an interface between the CPU, FPU, and a memory arbiter to manage operand and result flow without extending the instruction set.
- Setting up an automated environment for testing and Verilog generation from Chisel.
- Implementing partial reconfiguration to reduce dead time.
- Managing partial bit-streams and memory configurations for efficient reconfiguration.
- Exploring reconfiguration to enable flexible FPU functionality at runtime
- PPA trade-offs for different system configurations.

Components such as the CPU core, FPU implementation, and FPGA toolchain must be carefully selected to fit the project scope. The following section provides background information and related work that guided these choices.

II. Background and Related Work

Integrating an FPU requires a CPU core to manage data flow and coordinate operations. Since the scope of this project is limited, the CPU core used will be the publicly available Wildcat RISC-V core [1]. The FPU core, developed by Jon Dawson, is sourced from a publicly available repository [2]. The Vivado Suite is used to enable partial reconfiguration on the FPGA. Hence the FPGA types are limited to Xilinx (AMD) devices. Specifically Artix-7 devices are used.

In addition, several prior works influenced both the architecture and reconfiguration strategies adopted in this project and also highlight areas for future development:

- **AdaptivFloat** [3]: Explores the use of flexible floating-point representations to improve the resilience and efficiency of deep learning inference. This inspired the idea of adapting FPU functionality at runtime to match precision needs. The large number of configurations makes this use case well-suited for reducing dead silicon and improving power efficiency.
- **Wang and Wu** [4]: Discusses dynamic partial reconfiguration in FPGAs. Here the feasibility and practical considerations for implementing runtime hardware reconfiguration within a proposed adaptive architecture is tested.
- **Pezzarossa** [5]: Utilizes dynamic partial reconfiguration (DPR) in real-time systems. Many of the techniques proposed for managing reconfiguration, such as lightweight controllers and compressed bit-streams, were adopted or adapted for this project.

These works highlight the need for a flexible, efficient floating-point system capable of adaptation. Building on these insights, the following sections describe the specifications and design choices made in this project.

III. Specifications and Design

The objective is to explore a reconfigurable FPU that improves power, performance, and area efficiency by enabling operation switching without full system reprogramming. The design decisions for creating a reconfigurable floating point unit are elaborated upon in the following section.

A. Communication between core and FPU

The communication between the core and FPU could be achieved through several methods. One would be extending the instruction set to support floating-point operations. Specifically, to use proper floating-point instructions compatible with the wildcat core, RV32IF or RV32IMF need to be supported, which requires new instruction decode logic, an FPU register file and pipeline. This is neglected in favor of a more simple memory-arbiter solution, which has a memory-mapped I/O. It utilizes `sw` and `lw` instructions to interface with the FPU, see Figure 2. The implemented

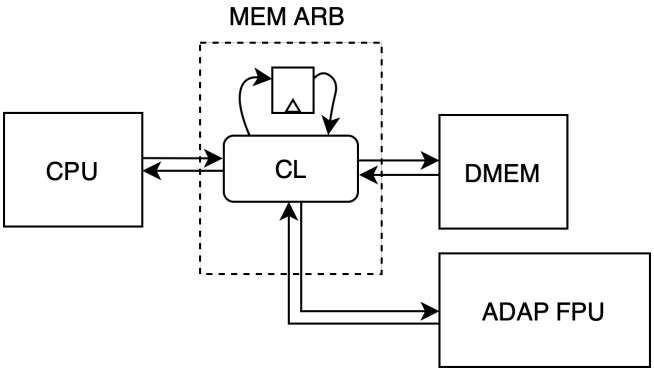

Fig. 2: Block diagram of the interfacing between the memory arbiter, the CPU, the FPU, and the DMEM.

memory arbiter does not introduce additional pipelining registers when communication is not being directed toward the FPU. When communication is directed towards the FPU, the memory arbiter blocks communication between data memory and cpu by deasserting the `cpu_wr_enable`. The arbitration scheme is therefore bypass-based. CPU and FPU communication through the memory arbiter is controlled through a finite state machine (FSM) with the following states:

1) The CPU issues two `sw` (store word) instructions targeting the memory-mapped address range `0x7FFFFxxx`, where the lower 12 bits (`xxx`) are used to encode operation-specific information to determine the FPU operation mode, enabling reconfiguration if necessary.
2) The memory arbiter routes the instruction appropriately to the FPU.
3) The FPU processes the operation and stores the result internally.
4) Once the computation is complete, the CPU can issue a `lw` (load word) instruction at address `0x7FFFFxxx` to retrieve the result through the memory arbiter.
5) The memory arbiter fetches the result from the FPU and returns it to the CPU.

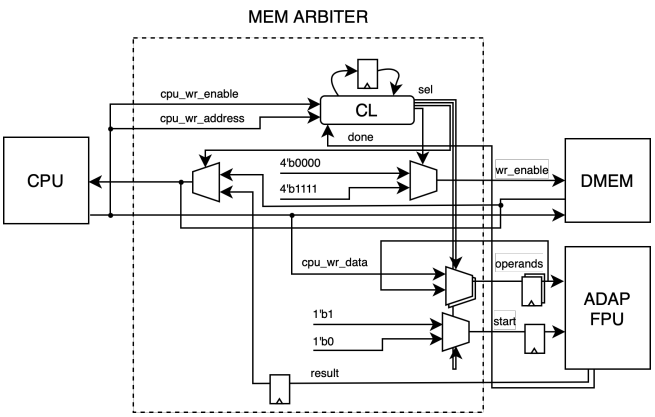

Fig. 3: Block diagram of the contents of the memory arbiter. Only relevant wires have been named.

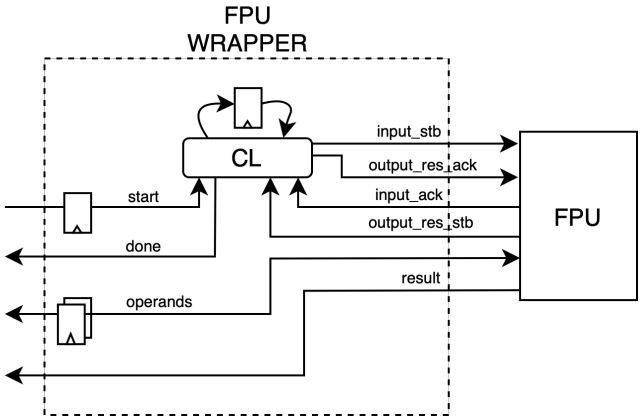

Fig. 4: Block diagram of the contents of the fpu wrapper. Only relevant wires have been named.

Figure 3 shows the overall flow within the memory arbiter. Registers are allocated only as output/input buffers for the FPU to retain operands across multiple clock cycles. Hence it is only combinational logic added between the CPU and DMEM during normal operation.

B. Setting up the adaptive FPU

The FPUs used in the reconfigurable regions have varying interfaces that differ from the memory arbiter's I/O. To standardize communication, a wrapper module was implemented. It includes a small FSM to enforce correct control signal sequencing. While this introduces additional logic in the reconfigurable region, which slightly increasing area and reconfiguration time, it ensures reliable integration. A block diagram of the multiplication FPU wrapper is shown in Figure 4. The sequencing of the FPU wrappers FSM is the following:

- When the `start` bit goes high, the strobe signals `input_a_stb` and `input_b_stb` are asserted. These signals instruct the FPU to load the input operands and begin the operation.
- When `input_ack` is asserted, `input_a_stb` and `input_b_stb` are deasserted to complete the handshaking phase.
- When `output_res_stb` goes high, `output_res_ack` is asserted to acknowledge the result. The `done` flag is then raised, signaling the memory arbiter to read the result.

C. Creating an automated environment

To ensure reproducible testing and code generation, a Docker environment with Verilator and Chisel toolchain dependencies was created, supporting both x86 and amd64 architectures. To compile the projects and run tests, Makefiles are used to automate the build process. This simplifies compilation across different environments, such as Verilator for Verilog simulation and ChiselTest for unit testing.

Additionally, two TCL scripts were made: one for generating the static DCP file and another for producing DCPs for different FPU configurations. This makes certain that the dynamic region always are the same and that certain naming schemes for the reconfigurable parts remain correct.

D. Implementing the Dynamic Partial Reconfiguration

The Dynamic Partial reconfiguration is achieved by selecting a particular portion on the FPGA chip and setting it as configurable/dynamic. When the bit stream is generated, it contains 2 parts: Static logic and dynamic logic. The static logic remains the same through the life cycle of the FPGAs operations but the dynamic portion can be reconfigured during runtime.

E. Storing and changing the bit-streams

There are two main solutions for storing the bit-streams on the FPGA: using the onboard flash memory or the on-chip block RAMs. The bit-stream can be loaded onto the FPGA through interfaces such as JTAG, ICAPE2, Serial, and SelectMAP [6]. However, during runtime, only the FPGA primitive `ICAPE2` can be used for reconfiguration. Both flash memory and block RAM were considered. Flash memory offers larger storage capacity, making it suitable for handling multiple or larger partial bit-streams. However, it introduces additional read latency during runtime access. Conversely, block RAM enables faster access with minimal latency, but is limited in storage size.

F. Reconfiguration

Utilizing the FPGA's logic blocks and its architectural distinction from an ASIC allows us to reconfigure a portion of the FPGA after it has been initially programmed.

The design for reconfiguration can be divided into the following:

1) Demarcation of P-Blocks: The FPGA programmable area is divided into a static region and a dynamic region. The marked dynamic region is referred to as placement Blocks (P-Blocks). Registers are placed on the inputs and outputs of the dynamic logic. Figure 5 demonstrates an example of demarcation of a P-Block within FPGA Architecture. when reconfigured the logic cells in this region will rearrange themselves according the to partial logic/bitstream provided.

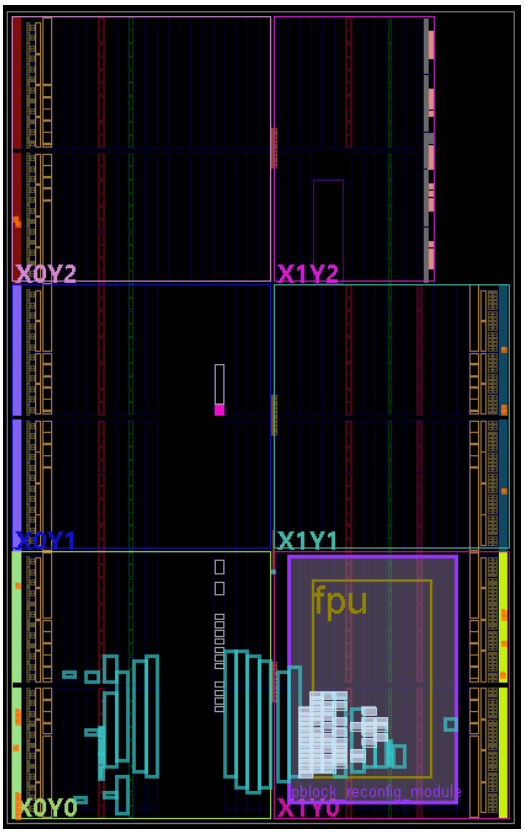

Fig. 5: Demarcation of a P-Block within FPGA Architecture. The highlighted pink square is the reconfigurable region assigned. The currently set asset is a floating point multiplier. The surrounding resource utilization is the Wildcat CPU core + the memory arbiter.

All the blocks/IPs that are intended to be dynamic are synthesized (their ports are kept same) and their area utilization is recorded. The P-blocks are created based on the largest area utilization, so as to ensure all dynamic blocks/IPs fit inside the reconfigurable area.

The area is marked after completion of the design and after synthesizing it. Vivado provides creation of the P-Blocks by 3 methods:

- Declaration in Constraint file
- Through Tcl commands
- Through the Dynamic Function Exchange Wizard

any one of them can be used to create and alter the P-Block definition.

2) Generation and upload of the Bit streams : The final design file that will be programmed on the FPGA is called a Bit Stream. Its file format is `.bit`.

After creation of the P-Blocks the workflow for creating the complete and partial bitstreams, including the Bit streams with a particular partial design implemented by default, is the same.

3) Understanding the bit stream : The bit stream contains a lot of design data and headers. The Bit file contains data that is 32 bit and is itself Big-Endian(data arranged from MSB to LSB) [6].

The few basic structures are given below:

- Padding: It is a `32-bit` dummy word that have the value `0xFFFFFFFF`.
- Sync Word : It is a `32-bit` word that indicates the start of the bit stream and is also used for alignment of the transmitted data. The value of Sync Word is `0xAA995566`.
- DeSync Word: It is a `32-bit` word that indicates the end of the bit stream. The value of DeSync Word is `0x0000000D`.
- Command: This is from the set of special command fed into the FPGA that tell the FPGA where the start address is. The command consists of a byte value `0x03` followed by the starting address of the bit stream.
- NOOP: It tells the FPGA to perform no operation and to not waste a cycle. The value of NO-op Word is `0x2000000D`.
- CRC code : This is used for CRC check and ensure that no wrong configuration is loaded onto the FPGA. Even if the whole bit stream is loaded the FPGA only configures it if the crc is checked.

G. Dynamic Reconfiguration

This is a build up on the partial reconfiguration ability of the FPGA and allows for the reconfiguration to take place at runtime, This means that the reconfiguration can be controlled by the static logic.

In the Artix-7 series of FPGA it is carried out by the use of ICAPE2 FPGA primitive.

ICAPE2: ICAPE2 is an FPGA primitive whose purpose is to program the FPGA and boot up the FPGA with the configuration that is loaded/ programmed at the time of stratup. It sits idle after bootup and can be utilized to program a specific portion of a FPGA or/and be setup to boot an image set at `0x000000` location of the spi flash or load dynamically load a bitstream to placement Blocks [6].

It is particularly important that the ICAPE2 is operated below 70 `MHz` [7]. Also, the fed data and the endian type of the ICAPE2 primitive must match.ICAPe2 is little endian (LSB to MSB) [6]. In the Artix 7 series, the ICAPE2 primitive requires a delay of 3 clock cycles before the first meaningful data can be sent [7]. This is usually accomplished by sending dummy words `0xFFFFFFFF` from the list of preset values [7].

The Block level diagram and ports for the ICAPE2 primitive are given in figure 6 .

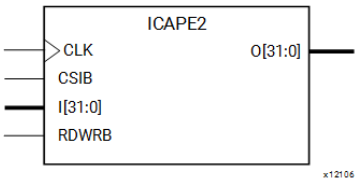

Fig. 6: FPGA primitive ICAPE2.

IV. Implementation

A. Memory Arbiter Implementation and Unit Testing

When integrating the memory-arbiter Some of the blocking statements were migrated from `WildcatTop`

to `MemArbiter`. The `fpu_wrapper` and `fpu` modules were bundled together as part of the reconfigurable pblock. Three configurations of the adaptive FPU were created: multiplier, adder, and divider. These system configuration implementations resource usage can be seen in Table I. This also includes the full FPU implementation, which contains all configurations that are then multiplexed dependent on the operation select value. It highlights the resource savings of each configuration, which scale significantly with more configurations needed. The reconfigurable area's size is determined by the most resource-demanding module, which in this case is the divider configuration, see Table I.

TABLE I: Comparison of resource usage for different FPU configurations and the full FPU. Numbers in parentheses show FPU-specific resources. (*) Reconfiguration controller resources are excluded, as integration was pending during analysis.

Implementation	LUTs	DFFs	BRAMs
Adder config.	1854 (+353)	835 (+282)	2 (*)
Div config.	1918 (+417)	944 (+163)	2 (*)
Mult config.	1747 (+246)	818 (+125)	2 (*)
Full FPU	2233 (+759)	1092 (+573)	2

Unit testing for the memory arbiter was performed with ChiselTest with a `DummyFpu` to check the behavior of the memory arbiter when poked with different sets of operands. The test writes operands through the CPU interface, waits for the operation to complete, and verifies the result by reading the output via memory.

B. Reconfiguration

The reconfiguration module was setup in VHDL and verilog for two different approaches. The implementation was confirmed in both the cases. Each module is used to implement a different approach for dynamic reconfiguration, which is discussed in detail in a later chapter.

The Partial Reconfiguration is implemented by the following method:

- **Static Logic and P-Blocks** The static logic containing the wiring logic, the registers for timing requirements, the reconfiguration definition and constraints file, the SPI flash definition, the logic for the reconfiguration controller, the FPGA Primitives. The reconfiguration control logic and bit stream compression are implemented.// The SPI Flash controller, logic for reconfiguration controller and their accompaying components/devices will be discussed in a subsequent section.
- **Dynamic Bit Stream** Partial bit streams of an adder and substractor are implemented for testing/proof of concept and partial bit streams of different FPU's are implemented for the project.

C. Dynamic Reconfiguration

A VHDL based module for SPI based dynamic reconfiguration and a verilog based module for block ram based dynamic reconfiguration was created to be incorporated with the cpu.
After the partial bit streams have been created, they must be converted into data appropriate to be used in:

- SPI based Flash memory
- Block Ram

1) Generation of Memory Configuration Files: In general, the binary files contains a Start of word, a end of word, a synchronization code, a ID-CODE, etc. It is also important to note that the .bin file created in BPI, SMAP and serial are all bit swapped with respect to ICAPE2 or simply big endian [6].
The memory configuration files(.bin) are created from Vivado. They are then fed to the HxD Hex editor(3rd party open-source software), where they are read and converted to 32-bit hex values. These hex values are converted to binary with a python script for either ram entries to use in user initialize block RAM or in .coe file for Vivado ip generated block RAMs.
Bit stream compression is also utilized to reduce the size of memory entries.
2) Method of memory setup and Reconfiguration: Bit streams to be used in dynamic reconfiguration can be stored using two methods:

- **On Flash Memory** There are two options to use the flash Memory (S25FL128S) [8].
  - SPIx4
  - SPIx1

  SPIx1 is used, and the specific SPI module must be specified in the `.xdc` constraint file.
  After bitstreams are generated, the device must be added in the Hardware Manager. The memory configuration file `.bin` for the dynamic logic (discussed later) must be stored sequentially in SPI flash at specific offsets, with start addresses incorporated into the design. To establish SPI communication, a one-byte read command (`0x03`) is sent, followed by a three-byte address, after which the dynamic logic (partial bitstream) is read back.
  See figure 7 for the ports used to interface with the SPI Flash on the Nexys-7 board.

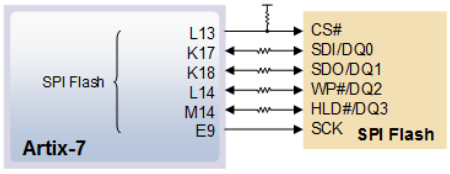

Fig. 7: Connections between SPI Flash and FPGA.

  The ports for SPI communication are given in the Nexys-7 reference manual [8], but the spi clk cannot be directly used, because the FPGA internal configuration logic still own the E9 pin which is used at boot up, so the spi clk is passed through another FPGA primitive "STARTUPE2" [8].
  See figure 8 for the ports used to interface of STARTUPE2 on the Nexys-7 board.

- **On Block RAMs**

  There are two methods to put bit streams on Block RAM in Vivado. In both the methods the incoming

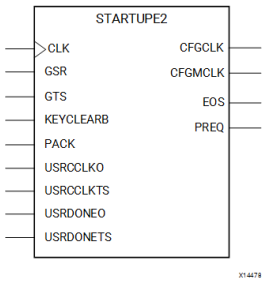

Fig. 8: Connections between STARTUPE2 port on FPGA.

data is sent to the ICAPE2 FPGA primitive by the help of a FSM embedded in the top module. This top module can then be incorporated and controlled by the inputs from the cpu.

- Array Style Block RAM: After generating the bit streams and partial bit streams, the partial bit streams are converted to memory configuration file(.bin), which are stripped of additional header. These files are then converted to hexadecimal values and, ultimately, to 32 bit binary values.
- IP-generated Block RAM: The vivado IP generator can also be used to created an IP core of block RAM, with a fixed data width of 32 bits, whereas, it's depths is dependent on the memory locations that the partial bit stream, to be uploaded to it, will occupy. An important point to note is that the IP generated need to be initialized with data (the bitstreams) in `.coe`(Coefficient) format.

3) Reconfiguration Memory: The spi flash, Vivado IP generator and user defined block memory was implemented and user defined Block RAM is finaly implemented for the design, which is used due to its ease of adaptablity.

V. Results and Evaluation

The system passed unit tests across all configurations, but reconfiguration could not be verified as the module was unfinished. A waveform snippet of the unit testing of the multiplication can be seen in Figure 9.

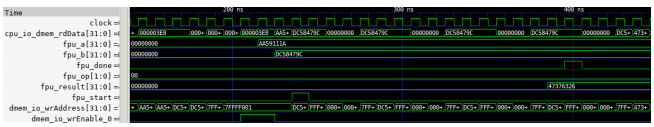

Fig. 9: Waveform showing a multiplication issued by the CPU by writing to `0x7FFFFxxx`. Operands A and B are loaded, computing the correct result `0x4737626`. A `lw` follows, with the result visible in the final captured cycle at `cpu_io_dmem_rdData`.

The logic of the various configurations was implemented on hardware. Due to timing issues, the Wildcat RISC-V core could not meet a 10 ns clock constraint, so the system clock was set to 20 ns instead. Three different setups were analyzed to compare PPA efficiency:

- CPU with assembly based floating point operations.
- CPU with a memory-arbiter and an adaptive FPU.
- CPU with a memory-arbiter and FPUs of all available configurations.

The tables below present the PPA metrics for three implementation types, based on implementation results from Vivado. Importantly, the adaptive FPU and full FPU configurations still only include three operations: An adder, multiplier, and divider.

TABLE II: Comparison of resource utilization for the base, adaptive, and full FPU + CPU implementations of the `WildcatTop` module. (*) For the adaptive FPU implementation, the resource numbers do not include the reconfiguration controller, as its integration was pending at the time of analysis.

Implementation	LUTs	DFFs	BRAMs
CPU (assembly MUL)	1392	418	2
CPU + adaptive FPU	1747 (*)	818 (*)	2 (*)
CPU + full FPU	2233	1092	2

Table III presents the performance in terms of clock cycles, dynamic power consumption, and the resulting energy per multiplication operation. The reported clock cycles for the adaptive and full FPU setups are equal (14 cycles) because both configurations used the same FPU datapath for multiplication, and the adaptive version had the multiplier module loaded prior to execution, which means this estimated time does not include the reconfiguration time required to load different operations. In a completely realized reconfigurable system, this reconfiguration overhead would increase the total latency for the adaptive design. The estimated energy per multiplication clearly demonstrates the superior energy efficiency of the adaptive FPU configuration.

TABLE III: Comparison between base, adaptive and full FPU + CPU implementations of the `WildcatTop` module regarding the amount of cycles and energy required per floating-point multiplication. (*) For the adaptive FPU implementation, the timing numbers do not include the reconfiguration time and the dynamic power does not include the reconfiguration controller as its integration was pending at the time of analysis.

Implementation	Cycles	Power	Energy
CPU (assembly MUL)	420	22 mW	9.2 J
CPU + adaptive FPU (*)	14 (*)	27 mW (*)	0.46 J (*)
CPU + full FPU	14	101 mW	1.41 J

As a proof of concept the reconfiguration of a simple adder and substractor based ALU was created. The bitstream comparision of the two designs with and without partial reconfiguration is given in table III. The table proofs a major reduction in Area.

TABLE IV: Comparison of bit stream of simple adder and substractor based ALU with and without partial reconfiguration.

Bit stream	without Reconfiguration	With Reconfiguration
adder (partial)	-	47.1 KB
substractor (partial)	-	48.1 KB
top module	3.46 MB	428 KB
Total	3.46 MB	523.2 KB

The results show that the adaptive FPU achieves significant area and energy savings compared to a full FPU setup, at the cost of reconfiguration overhead not yet accounted for. Future work should include full integration of the reconfiguration controller to assess this trade-off accurately.

VI. Conclusion

This work presented a reconfigurable FPU integrated into a RISC-V CPU core, managed via a memory-mapped `MemArbiter` interface and encapsulated using a standardized `fpu_wrapper`. These components ensured seamless data flow, control signal coordination, and allowed switching between arithmetic operations without modifying the CPU's instruction set.

While the design incorporating partial reconfiguration (DPR) demonstrates improved PPA metrics, it is important to note that DPR introduces a setup latency proportional to the bit-stream size. This latency can become a significant overhead in scenarios requiring frequent operation switching. In such cases, using dedicated hardware modules or employing multiple reconfigurable regions—where the least recently used FPU module is reconfigured during runtime—could provide a more efficient solution.

Although inspired by AdaptivFloat [3], which proposes adjusting floating-point representations dynamically, our implementation did not yet include varying precision configurations due to time limitations. Future extensions could explore integrating multiple bitwidths into the reconfigurable FPU architecture to better align with the goals of AdaptivFloat.

Building on these insights, future work should be the following:

- Fully integrate a dynamic reconfiguration controller to accurately capture the overhead and dynamic behavior during runtime.
- Expand the range of available FPU configurations, including different precision levels and bitwidths, to align with concepts from AdaptivFloat [3].
- Investigate using multiple reconfigurable regions, combined with an LRU replacement strategy, to hide reconfiguration latency and maintain continuous operation.
- Explore faster reconfiguration schemes, such as parallel block RAM instantiation or segmenting bitstreams, to further reduce downtime during switching.

All source code and testing environments are publicly available at: https://github.com/rene2997/adaptive_fpu.

Author Contributions

Mirza Jafar Ali Baig worked on implementing dynamic partial reconfiguration, managing partial bitstreams, and converting memory configuration files into RAM-compatible formats. He also explored methods for enabling dynamic reconfiguration during runtime.

René Antonio Hjort designed the interface between the CPU core and FPU without modifying the instruction set. He also set up the automated testing framework and Verilog generation environment, and played a leading role in the integration of dynamic partial reconfiguration, including evaluating power, performance, and area (PPA) trade-offs.

Mathieu Astagneau contributed to the early exploration of partial reconfiguration, supported the evaluation of FPU IP cores, and assisted with bitstream extraction from configuration files.

Jiayi Lu mainly focused on exploring runtime reconfiguration strategies, helped coordinate the project workflow, and contributed to discussions and presentations throughout the development process.

References

[1] M. Schoeberl, "Wildcat: Educational risc-v microprocessors," 02 2025.

[2] J. Dawson, "Synthesiseable ieee 754 floating point library in verilog," https://github.com/dawsonjon/fpu, 2021, accessed: 2025-04-21.

[3] T. Tambe, E.-Y. Yang, Z. Wan, Y. Deng, V. J. Reddi, A. Rush, D. Brooks, and G.-Y. Wei, "Adaptivfloat: A floating-point based data type for resilient deep learning inference," 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1909.13271

[4] L. Wang and F. yan Wu, "Dynamic partial reconfiguration in fpgas," *Proceedings of the International Conference on Computer Application and System Modeling (ICCASM)*, 2010, department of Computer Science & Electronic Information, Guangxi University.

[5] L. Pezzarossa, A. T. Kristensen, M. Schoeberl, and J. Sparsø, "Using dynamic partial reconfiguration of fpgas in real-time systems," *Microprocessors and Microsystems*, vol. 61, pp. 198–206, 2018. [Online]. Available: https://doi.org/10.1016/j.micpro.2018.05.017

[6] Xilinx, *7 Series FPGAs Configuration User Guide*, Xilinx.

[7] AMD, *Artix 7 FPGAs Data Sheet: DC and AC Switching Characteristics*, Xilinx.

[8] Diligent, *Nexys A7™ FPGA Board Reference Manual*, Diligent.