Increasing CVA6 frequency on FPGA

5th National RISC-V Student Contest 2024–2025

Team RISC Takers – IMT Atlantique / Lab-STICC

Gweltaz Poussin*, Thibaud Quentric*, Pedro Piovesan*, Salomón Ojeda*

*(gweltaz.poussin, thibaud.quentric, pedro-henrique.figueiredo-piovesan, salomon.ojeda)@imt-atlantique.net

***Abstract*—This paper presents the solution proposed by the RISC Takers group to increase the clock frequency of the CV32A6 processor. The results described were obtained during the 5th National RISC-V Student Contest 2024–2025, organized by Thales, the GDR SOC² and the CNFM. We validated our modifications through functional simulation and timing analysis, ensuring compatibility with CoreMark and MNIST benchmarks. As a result, we successfully increased the maximum operating frequency of the processor from 40 MHz to 64,495 MHz while maintaining software compatibility and minimal performance degradation.**

***Index Terms*—CV32A6 processor, RISC-V, performance, clock frequency, simulation.**

I. Introduction

The RISC Takers team is participating in the fifth edition of the RISC-V Student Contest 2024-2025, a national competition organized by Thales, the GDR SOC² and the CNFM, whose main objective is to improve the performance of the CV32A6 processor implemented on an FPGA. The objective is to increase the operating frequency of the processor without compromising its functional compatibility and overall performance.

To measure the performance of the modified processor, the CoreMark and MNIST benchmark applications have been used [1].

This project was developed between October 2024 and May 2025 by a team of four students of the Heterogeneous Embedded Systems (SEH) specialization at IMT Atlantique, Brest campus. Throughout this time, we have worked on the processor optimization and validation on an FPGA platform, applying and consolidating our knowledge in digital design and embedded systems.

The CV32A6 is a compact and optimized version of a 32-bit processor based on the open-source RISC-V instruction set architecture, which continues to gain adoption in embedded systems. In the following, we analyze the internal architecture of the core, identify critical bottlenecks, and propose architectural improvements to enhance its execution speed. These changes are implemented while preserving the software environment and ensuring the correct functionality of the modified design on the Digilent Zybo Z7-20 board. [1].

Modifications are validated through functional simulations and timing analysis, guaranteeing the stability and efficiency of the system. This paper presents the technical strategy of the RISC Takers team, the design decisions adopted throughout the project, and the results obtained by implementing and validating the solution on hardware. The remainder of this article is structured as follows:

- Section II - Tools and Ressources
- Section III - Discussion
- Section IV - Results and Analysis
- Section V - Acknowledgements

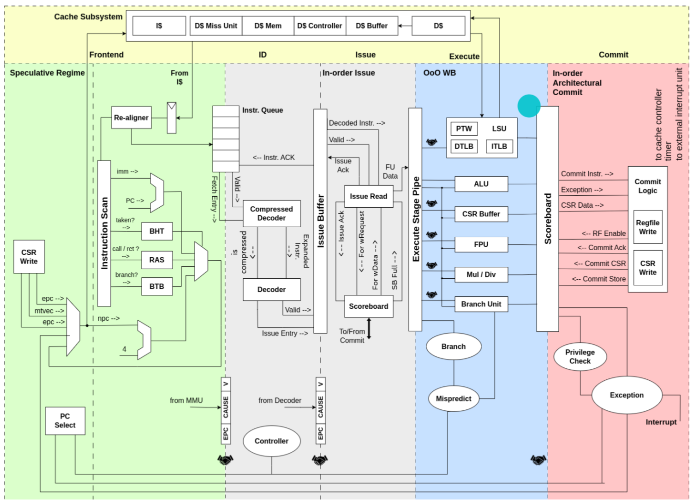


Fig. 1. CVA632 architecture

II. Tools and Ressources

To support the development of the project, the contest organizers – Thales, GDR SOC² and CNFM – made available a complete set of resources, including:

- The base core parameters and synthesis scripts.
- A BSP (Board Support Package) and a reference design for the Digilent Zybo Z7-20 FPGA board.
- Benchmark applications: CoreMark and MNIST, running in bare-metal mode on the processor.
- Access to a Discord server for communication, support and official announcements [1].

- A public GitHub repository containing the official codebase and documentation for the contest[1].

In addition to the provided kit, we used tools such as Siemens Questa for digital simulation and AMD Xilinx Vivado for synthesis and Route on FPGA. The development setup was mainly done in a Linux Ubuntu 20.04 environment, following the recommendations of the organizers. These resources significantly facilitated the development process, allowing us to concentrate on the architectural improvements of the CV32A6 core. Each tool is explained in detail below.

A. *Zybo Z7-20 FPGA board*

The Zybo Z7-20 is a development board based on the Xilinx Zynq-7000 SoC, which combines a dual-core ARM Cortex-A9 processor with a programmable FPGA [2].

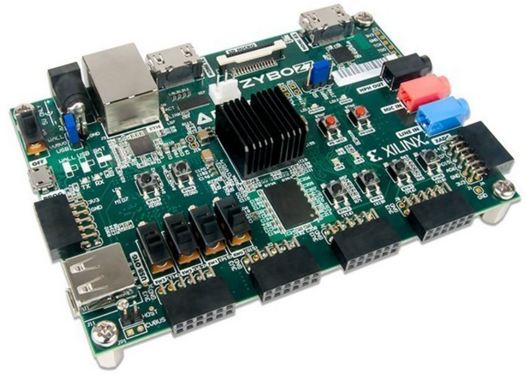


Fig. 2. Digilent Zybo Z7-20 FPGA board

For the implementation and validation of the CV32A6 processor, we used the Zybo Z7-20 along with two modules: the JTAG-HS2 Programming Cable, which enables bitstream loading and debugging via the JTAG protocol, and the Pmod USBUART, a USB to UART converter used for serial communication and display of console output from test applications such as CoreMark and MNIST.

B. *Questa*

Questa is a digital simulator developed by Siemens EDA, widely used for verification of digital design at the Register Transfer Level (RTL). It supports standards such as SystemVerilog, VHDL and Verilog, and allows functional simulations, formal verification and testbench coverage. In our project, Questa was mainly used to verify the correct behavior of the CV32A6 processor after applying architectural modifications, ensuring that no functionality errors were introduced before moving to the physical implementation [3].

C. *Vivado*

Vivado Design Suite, developed by AMD Xilinx, is a comprehensive design, synthesis, and implementation EDA tool for developing digital circuits on FPGAs. In this project, we used Vivado to synthesize the modified CV32A6 core, generate the bitstream for FPGA programming, and perform static timing analysis (STA) to determine the new maximum achievable frequency [4].

Timing analysis tools enabled us not only to know the maximum frequency achievable with our modified design but also to visualize the critical path and all other failing paths.

The main workflow we followed was as follows: We start by increasing the frequency in order to have at least one failing path. This could be achieved either by modifying the tool command language file from the ”xlnx_clk_gen” directory or by using the ”Clocking Wizard” IP within Vivado. Once this is done, we can launch the implementation. This step has two possible outcomes: either the requested frequency can be achieved, in which case we can test and validate the modified processor, or we must reduce the critical path.

We then use the timing analysis to visualize the failing paths. There are 2 ways to do that; we can visualize it on the implemented device with on the generated schematic:

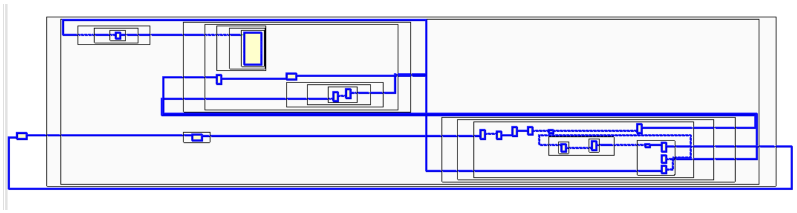
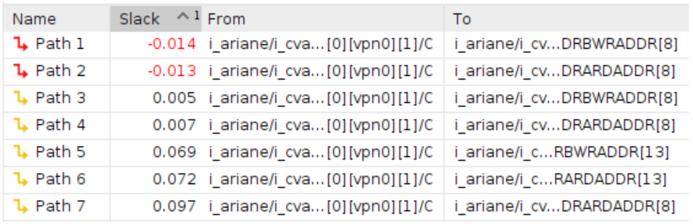


Fig. 3. Critical path visualization by schematic

or simply by looking at the different data paths:



Name	Slack ^ 1	From	To
Path 1	-0.014	i_ariane/i_cva...[0][vpn0][1]/C	i_ariane/i_cv...DRBWRADDR[8]
Path 2	-0.013	i_ariane/i_cva...[0][vpn0][1]/C	i_ariane/i_cv...DRARDADDR[8]
Path 3	0.005	i_ariane/i_cva...[0][vpn0][1]/C	i_ariane/i_cv...DRBWRADDR[8]
Path 4	0.007	i_ariane/i_cva...[0][vpn0][1]/C	i_ariane/i_cv...DRARDADDR[8]
Path 5	0.069	i_ariane/i_cva...[0][vpn0][1]/C	i_ariane/i_c...RBWRADDR[13]
Path 6	0.072	i_ariane/i_cva...[0][vpn0][1]/C	i_ariane/i_c...RARDADDR[13]
Path 7	0.097	i_ariane/i_cva...[0][vpn0][1]/C	i_ariane/i_cv...DRARDADDR[8]

Fig. 4. Vivado Timing analysis : Failing paths in red

The last two are the most useful because the schematic gives us an easy summary of which parts of the processor are part of the path, and the data path gives us more specific timing information. Now, we can simply find the signal responsible for the timing failure and add some delay or even completely remove it.

Finally, Vivado can restart the implementation and see whether or not the timing is now respected.

D. *MNIST Application*

The MNIST application is based on a database of handwritten digit images created by the National Institute of Standards and Technology (NIST). Adapted for bare-metal execution, this application runs directly on the CV32A6 processor without operating system. In the contest, it is used as a functional non-regression test: the modified processor must correctly

[1]https://github.com/ThalesGroup/cva6-softcore-contest

execute the digit inference, thus validating that the frequency improvements did not affect the stability or logic behavior of the core. Furthermore, the MNIST application provides us with two pieces of information. The number of instructions required to execute the program as well as the number of clock cycles. These numbers will help us determine the changes in the performance of the software. Indeed, the modifications to be made are very likely to increase the number of clock cycles needed to complete the program. For every change, it is important to ensure that the number of cycles required doesn't rise too much (exact specification in II-E) [5].

E. CoreMark

CoreMark is a specific benchmark for evaluating the performance of embedded processors, designed by the Embedded Microprocessor Benchmark Consortium (EEMBC). Unlike more general benchmarks, CoreMark measures key operations such as linked list handling, array manipulation and state control by simulating a typical set of tasks of a real embedded system. As part of this contest, CoreMark serves as the primary metric to ensure that modifications increase the frequency of the CV32A6 without significantly degrading its relative performance in critical operations. More specifically, at all times, we must make sure that the "CoreMark/MHz" performance never drops more than 10% [6].

F. PuTTY

Putty is a terminal simulator that can be used to connect to embedded linux systems through SSH or serial connection. It can also be used to transfer files from one OS to another, but we be using it in this way. In this project, PuTTY will only be used with the two test applications as an output window [7].

III. Discussion

In this section, we detail the modifications made to the CV32A6 processor to achieve higher operating frequencies. For each change to the processor, we will describe the specific part of the processor, the reasoning behind each change, and their observed impacts. Our optimizations were applied incrementally, validating each step through simulation and timing analysis to ensure functional correctness and frequency improvements.

A. Pipelining the MMU Exception Signal

In the first version of the provided processor, the only negative slack in the Vivado design occurred on the Execute Stage of the pipeline. The Memory Management Unit (MMU) is responsible for linking the virtual address used by the software running on the CVA-6 to the physical one used by the memory.

The first optimization focused on the MMU. We observed that the exception signal generated by the MMU introduced a timing bottleneck.

However, we noticed that the critical path that was blocking a higher frequency was only linked to the exception signal and, therefore, only involved when a problem emerges. In addition to that, as described on the following sketch, this exception signal was exiting the MMU by going through a FIFO list, meaning that assuming that 2 exceptions were raised simultaneously, one of them would be delayed without any issue.

To reduce the slack from this path, we inserted a pipeline stage into the output of the exception signal by adding a shift register for the exceptions.

This modification effectively reduced the combinatorial path length, allowing timing closure at higher frequencies with minimal impact on functionality. Functional simulations confirmed that the exception handling behavior remained correct after pipelining.

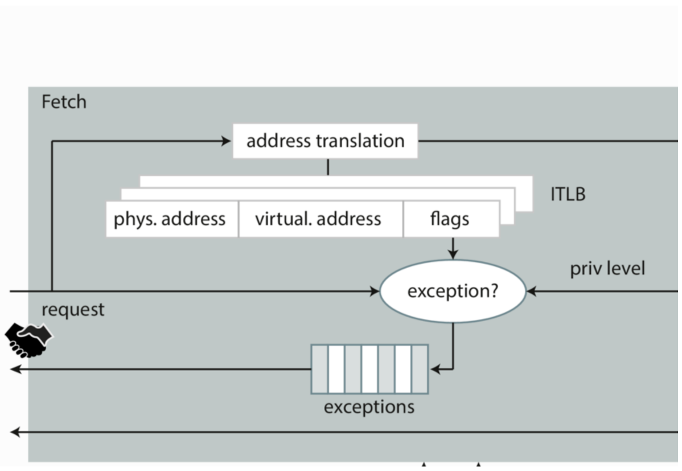
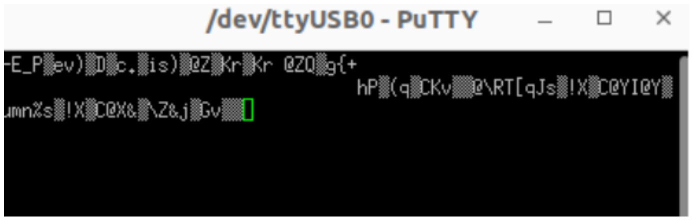


Fig. 5. Memory Managment Unit [8]

In the context of the CV32A6 project, the MMU was primarily responsible for memory access checking. We optimized the MMU by pipelining its exception signal, reducing its timing impact without compromising functionality.

B. Fixing the UART generated frequency

From this point on, our software works faster than the original. However, we are still havingaving some issues with regard to testing our design. Indeed, this final step requires that the coremark and Mnist applications run without any trouble. However, even though the software loads these applications, the results are not satisfactory because random characters are printed on our putty terminal:



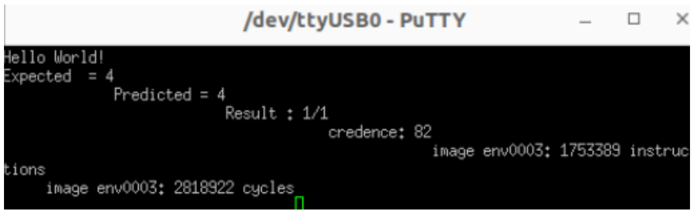
```
-E_P▒ev)▒D▒c.▒is)▒@Z▒Kr▒Kr @ZQ▒g{+
                                        hP▒(q▒CKv▒▒@\RT[qJs▒!X▒C@YI@Y▒
umn%s▒!X▒C@X&▒\Z&j▒Gv▒▒
```

Fig. 6. Wrong output from PuTTY

Additional testing reveals, that the issue is caused by the action of changing the frequency. Indeed, in order to communicate through UART, the processor must send out data at a fixed frequency. This frequency being smaller than the one from the global clock, the UART clock is mainly the global one after a clock divider module. However, by increasing the global clock to 50 MHz, we also changed the UART frequency.

The fix requires an increase in the dividing ration proportionally to the clock. At 40MHz this corresponds to a ratio of 8 meaning that we must set it to $RATIO = 8 + \frac{NewFrequency - 40}{5}$.

After implementing this change, we once again could have the correct output from our testbench :



/dev/ttyUSB0 - PuTTY

```
Hello World!
Expected  = 4
                Predicted = 4
                                Result : 1/1
                                        credence: 82
                                                image env0003: 1753389 instruc
tions
    image env0003: 2818922 cycles
```

Fig. 7. correct output from PuTTY

C. Placement and Routing Optimization

After describing our optimizing of the MMU, we address in this section optimizations to the placement and routing. We fine-tuned Vivado's placement and routing (P&R) strategies, adjusting constraints and allowing more aggressive optimizations. This step primarily targets critical paths that are not easily addressable by architectural changes. Although it does not change the processor's logical behavior, it significantly improves timing closure by reducing routing congestion and shortening critical nets. More specifically, by modifiying two parameters on the "run_cva6_fpga.tcl" file : the property for the place_design step is switch from "RuntimeOptimized" to "ExtraTimingOpt" and for "route_design" from "RuntimeOptimized" to "AggressiveExplore". These modifications significantly improve the performances of the design produced by Vivado and allow us to move from a maximum frequency of 50 MHz to 55MHz.

D. Modification of CSR Register File

For the next milestone of 60 MHz, the different failing paths at this frequency were all going in and out of the scoreboard. This part has many roles including the control of which register is being written into during the Issue stage. This component receives and sends information to many others. Since the scoreboard is implemented as a large FIFO, delaying one signal would have had an impact on all of them. Therefore, we decided to make our modification to the Control Status Registers (CSR) file. This part mainly controls wether or not status registers can load new data. we decided to delay a signal that allowed these registers to receive the data for the next instruction (RF Enable, fig 8). Since the processor is already waiting for this signal, adding some delay will not interfere with the correct function of the processor.

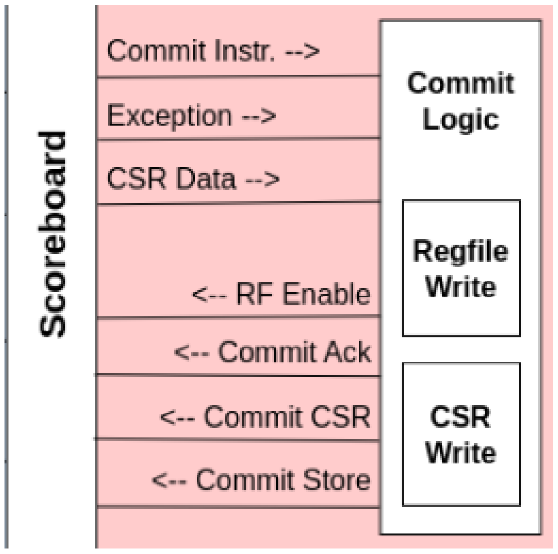


Fig. 8. CSR and scoreboard communication [8]

E. Pipelining Performance counter

The last part of the processor that was limiting the frequency increase was the performance counter. The only usage of this part of the processor was, as its name suggests, to precisely measure internal processor activity. For example, in the CVA-6, it is used to count the number of cache misses if the processor is stalling or if the scoreboard is full. Of all the signals that go through the performance counter, four of them fail the timing constraint: l1 cache misses (for instruction and data), stall issue, and the output for reading the data. Since the counter's primary role is statistical, a one-cycle delay in signal input does not compromise the accuracy or utility of the collected performance metrics.

IV. Results and Analysis

A. Maximum frequency achieve

After adding all of the changes described, the maximum achievable frequency of the processor is significantly higher. However, the situation is unclear, indeed, the modifications added to the processor reduce the critical path enough to reach **65 MHz**. At this frequency, we have a correct implementation by Vivado and no timing error. Furthermore, the MNIST application runs perfectly and gives us the expected output.

"MNIST output at 65 MHz "

```
Expected  = 4
Predicted  = 4
Result  :  1/1
credence:  82
image  env0003:  1760387  instructions
image  env0003:  2890793  cycles
```

However, a problem occurs when we try to run the COREMARK application:

”COREMARK output at 65 MHz”

```
unknown exception
```

This indicates a problem with our modifications. On the other hand, when the same solution is implemented at 64 MHz, both MNIST and COREMARK return correct results:

”COREMARK output at 64 MHz”

```
2K performance run parameters for coremark.
Coremark Size     : 666
Total ticks       : 1189661
Total time (secs): 1.189661
Iterations/Sec    : 2.521727
Iterations        : 3
Compiler version  : GCC13.1.0
Compiler flags    :
Memory location   : BRAM
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0x2e87
Correct operation validated.
See README.md for run and reporting rules.
Coremark 1.0 : 2.521727 / GCC13.1.0
/ BRAM
```

The MNIST result is the same as before (which is expected since the software is the same).

These elements would lead us to think that our maximum working frequency is **64.495 MHz** (our implementation at 64 MHz has a slack of 0.12 ns).

Our current theory is that the issue with the coremark does not come from the software itself but rather from the UART. Unfortunately, we did not have the time to investigate in detail.

B. *Comparison with 40MHz*

Resource	40 MHz Utilization	65MHz Utilization	Difference (%)
LUT	24,475	24,319	-0.64%
LUTRAM	902	902	0.00%
FF	14,568	14,638	+0.48%
BRAM	66	66	0.00%
DSP	4	4	0.00%
IO	9	9	0.00%
BUFG	5	5	0.00%
MMCM	1	1	0.00%

TABLE I
COMPARISON OF RESOURCE UTILIZATION AT 40 MHZ AND 65 MHZ

As we see in Table I, the difference in resources between the original version of the CVA-6 synthesized with 40MHz constraints and the modified CVA-6 with 65MHz constraints is negligible. Adding pipelining at different levels of the processor slightly increases the number of flip-flops used, but this increase is marginal.

Table II shows how little impact our modification has on the processor performance. For the MNIST test, the instructions count is still the same because we didn’t alter the ISA of the CVA-6 core by adding or removing instructions. Therefore, the processor runs the exact same instruction for the MNIST application both at 40 and 65MHz. One anticipated side effect of introducing pipelining was a potential increase in the number of clock cycles required to execute certain operations. This expectation is confirmed by a slight increase in the number of clock cycles needed to complete the MNIST test. However, the impact remains negligible, with the increase being less than 1%. For the CoreMark test, our modified processor has slightly worse performances than the original at 40MHz. At 40 MHz, the processor achieved a CoreMark score of 2.530480 CM/MHz (2.604 is the simulation score and not the implementation), whereas at 65 MHz, the CoreMark score is 2.521737 CM/MHz. This penalty (0.35%) remains well within the 10% allowed under the contest rules.

Parameter	40 MHz	65/64 MHz	Difference (%)
MNIST			
Instructions	1,760,387	1,760,387	0.00%
Cycles	2,818,904	2,890,793	+2.49%
CoreMark			
Score (CM/MHz)	2.604	2.521727	-3.15%

TABLE II
COMPARISON OF MNIST AND COREMARK RESULTS AT 40 MHZ AND 65 MHZ

V. ACKNOWLEDGEMENTS

We would like to thank the team of teachers from IMT Atlantique and the 2AI team from the Lab-STICC laboratory who supported us throughout this competition: Stefan Weithoffer, Amer Baghdadi, Yehya Nasser, Catherine Douillard and Ali Al Ghouwayel. Their support and advice were crucial to the success of this project. We also wish to extend our thanks to the contest organizers for giving us the opportunity to participate in this project.

REFERENCES

[1] GDR SOC² Thales and CNFM. 5th national risc-v student contest 2024–2025: Increasing cva6 frequency on fpga, April 2024. Version 2.11.
[2] Digilent Inc. Jtag-hs2 programming cable datasheet, 2013. Accessed on May 4, 2025.
[3] Siemens EDA. Questa advanced simulator, 2024. Accessed on May 4, 2025.
[4] AMD. Amd vivado™ design suite, 2025. Accessed on May 4, 2025.
[5] Ultralytics. Mnist dataset - ultralytics, 2023. Accessed on May 4, 2025.
[6] Shay Gal-On and EEMBC. Coremark benchmark. https://www.eembc.org/coremark/, 2009. Accessed on May 4, 2025.
[7] Simon Tatham. Putty: A free ssh and telnet client, 2025. Accessed on May 4, 2025.
[8] OpenHW. Cva6 design document, 2021. Accessed on May 4, 2025.