Journal of **N**onlinear **A**nalysis and **O**ptimization
Vol. 15, Issue. 1 : 2024
ISSN : **1906-9685**

IMPLEMENTATION OF A RISC-V ARCHITECTURE FOR REAL-TIME SEIZURE DETECTION

G. Laxmi Durga Bhavani, Department of Electronics and Communication Engineering, DVR & Dr.HS MIC College of Technology, Kanchikacherla , Andhra Pradesh.
[1] laxmidurgabhavanigamasu@gmail.com
B.Radhakrishna Singh , Associate Professor, Department of Electronics and Communication Engineering, DVR & Dr.HS MIC College of Technology, Kanchikacherla , Andhra Pradesh.
[2] bondilirk@gmail.com.
S.Bharadwaj , E.Dinesh Reddy and N. Rakesh Department of Electronics and Communication Engineering, DVR & Dr.HS MIC College of Technology, Kanchikacherla , Andhra Pradesh.
[3] bharadwajsudula@gmail.com, [4] edineshreddy2003@gmail.com [5] rakeshnaragani2002@gmail.com

ABSTRACT
This paper centers on crafting and implementing a RISC-V architecture specifically tailored for real-time seizure detection. By leveraging the open-source and adaptable nature of the RISC-V framework, the aim is to create an Instruction Set Architecture (ISA) that offers a versatile and efficient platform for processing biomedical signals, thereby facilitating the prompt identification of seizure episodes. Epilepsy, a neurological condition marked by recurring seizures, demands swift and precise detection for effective management. This endeavor tackles this challenge by devising a specialized RISC-V Architecture optimized for the real-time processing and analysis of electroencephalogram (EEG) signals.
The RISC-V processor in development is fine-tuned for signal processing tasks, supplemented by bespoke hardware accelerators to bolster computational efficiency. Traditional instructions deemed unnecessary are omitted to enhance power efficiency and reduce silicon area usage, ultimately leading to faster operation speeds. Such advancements promise significant benefits across various critical tasks essential for timely seizure detection.
Keywords: RISC-V processor, Electroencephalogram (EEG), Instruction Set Architecture (ISA)

1 INTRODUCTION
The pursuit of timely seizure detection in the realm of epilepsy management stands as a paramount challenge in healthcare. To address this critical need, the implementation of a RISC-V architecture tailored explicitly for real-time seizure detection emerges as a pioneering endeavor. Leveraging the inherent flexibility and adaptability of the RISC-V framework, this paper aims to design an Instruction Set Architecture (ISA) finely tuned for processing biomedical signals Hi Rui (2021), particularly electroencephalogram (EEG) data. By crafting a specialized RISC-V processor optimized for signal processing tasks and supplemented with custom hardware accelerators, this initiative seeks to enhance computational efficiency while minimizing power consumption. Omitting superfluous instructions further amplifies power efficiency and operational speed, culminating in a platform poised to revolutionize the landscape of seizure detection] J. Tranter (2022) through its timely and precise analysis of EEG signals. This introduction sets the stage for a comprehensive exploration of the design and implementation of a RISC-V architecture dedicated to the critical task of real-time seizure detection.

MOTIVATION

NASA aims to send a rover to Venus in the 2030s, but Venus's extreme conditions pose challenges for electronic circuits. New technologies like Silicon Carbide (SiC) and Silicon on Insulator (SoI) CMOS are being explored. SoI CMOS, with its higher bandgap, can withstand elevated temperatures, though not as high as SiC. Despite limitations, SoI CMOS allows for digital circuit manufacturing, albeit with constraints on size and performance. To assess its feasibility, a CPU is being designed as a testbed for Venus's harsh environment.

2 Previous Works

This chapter offers a comprehensive overview of the foundational aspects concerning RISC architecture, juxtaposed with its counterpart, CISC architecture. It delves into the manifold advantages that RISC architecture offers in real-time applications. Furthermore, it introduces the RISC-V architecture, a particular variant of RISC architecture chosen for implementation in this thesis paper. The chapter extensively discusses the specifications of RISC-V, particularly focusing on the RV32IM instruction set employed within the paper. Additionally, the related literature will encompass studies on RISC-V, including the development of products by other researchers, as well as the pivotal role played by RISC-V ISA extensions.

Since the advent of RISC-V O. A. Rusanu, et al (2020), it has developed quickly. More and more scholars and institutions have researched this architecture. Many researchers are working on developing RISC-V products or applying RISC-V to some specified applications. In this section, two RISC processor cores Y. Hsieh, et al. (2022) are introduced. They are RI5CY and Zero-risky. In addition, how the RISC-V processor plays a role in neural networks is also mentioned.

3.METHDOLOGY

RISC-V Processor Structure

This chapter discusses the structure of the RISC-V processor. presents the core architecture of the processor. In addition, each module of the core is described in detail. In the next sections , the structure of RISC-V SoC is discussed. The extra modules of the RISC-V platform, like peripheral units and memory, are also explained.

RISC-V core

In this SoC design, a solitary RISC-V processing core takes center stage. Equipped with its independent instruction fetch unit, it seamlessly executes instructions stored in memory, facilitating data exchange with peripheral devices. Unlike some RISC-V cores supporting multi-threading, this core operates without such capability. While certain cores boast specialized instruction sets and co-processors, this core remains uncomplicated.

Its three-stage pipeline, illustrated in Figure 3.1, enhances efficiency, with each stage operating independently: instruction fetch, decode and retrieval, computation, and data write-back. Comprising ten components, including PC_REG, IFu_IDu, IDu, IDu_EXu, EXu, DIVu, reg_bank, CSR_REG, INTu, and CTRLu, each is detailed in subsequent sections.

3.1.1 PC register and General-purpose registers

Because the base instruction set is RV32I, there are 32 registers in the register bank, which are all 32 bits wide. All these registers can be written synchronously and read asynchronously.

These registers also serve different purposes. Table 3.1 describes the role of each type of register. Register x0 is hardwired to the constant zero. Register x1 (ra) holds the return address

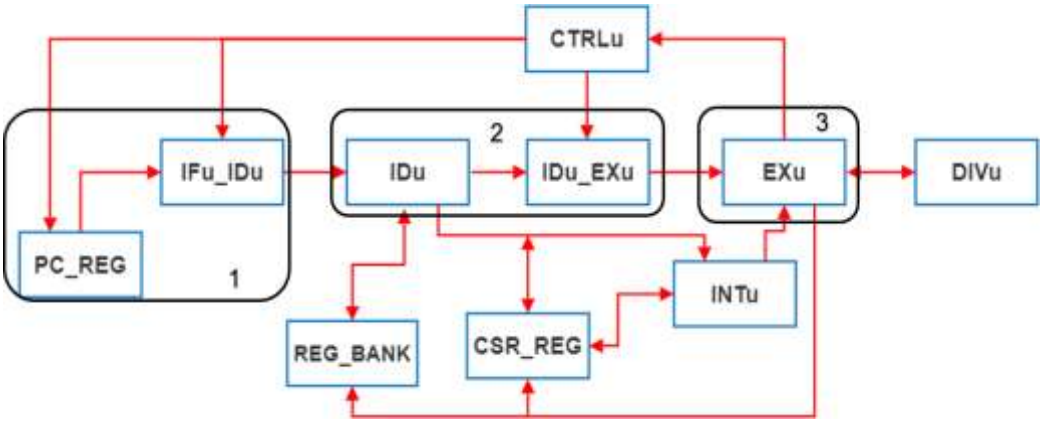
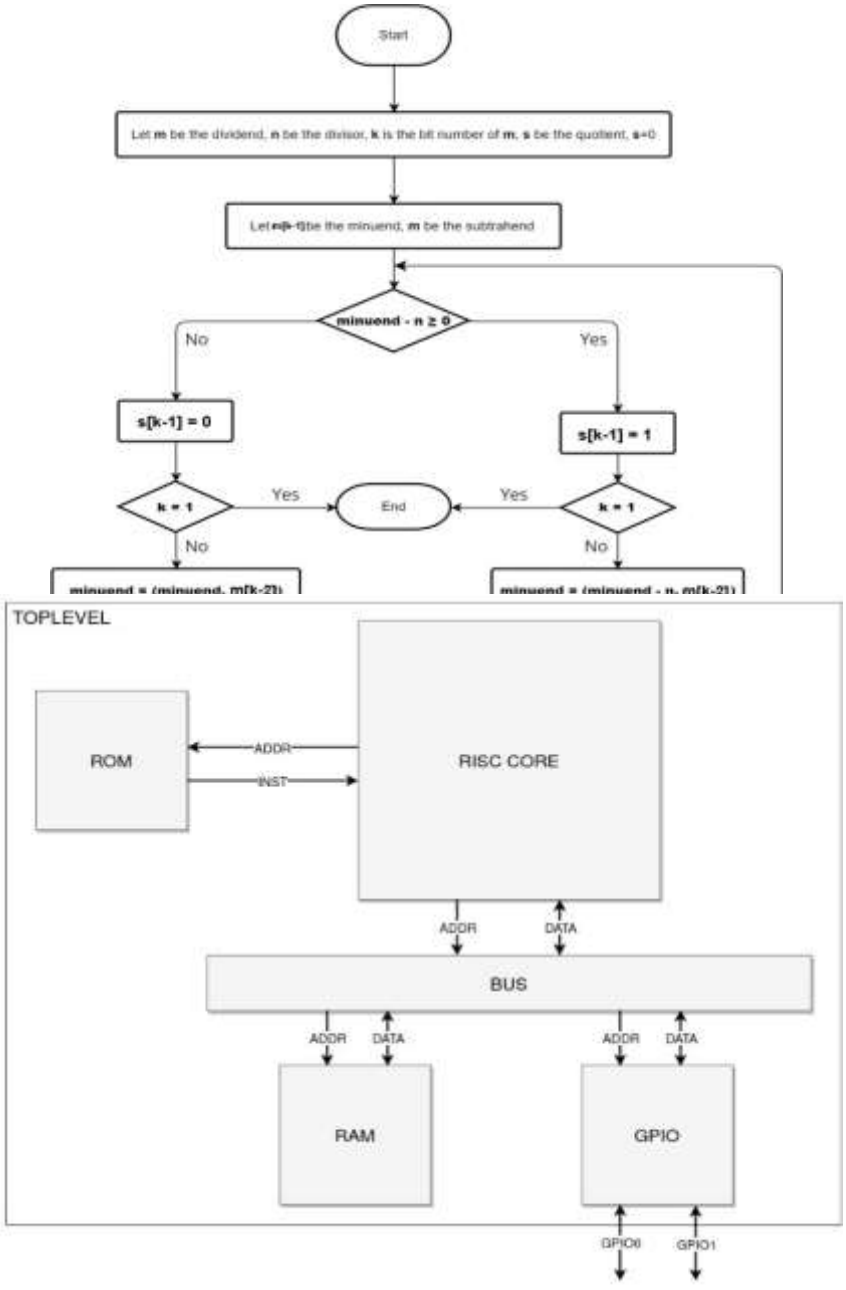


Figure 3.1: RISC-V core structure



4.RISC-V PROCESSOR IMPLEMENTATION (PROPOSED METHODOLOGY)

This chapter delves into the intricacies of implementing the RISC-V processor. During the RTL development phase, extensive modifications were made to the code of numerous modules compared to the previous paper. These alterations are comprehensively discussed in Section 4.1. Additionally, the chapter delves into the intricacies of the place and route processes.

4.1 RTL DEVELOPMENT

In this section, some important codes of each module are explained. Compared to the previous paper there are some changes in certain modules, which are discussed in detail. Two new units, CSR_REG and INTu, are also discussed.

4.1.1 PC_REG

Figure 4.1 shows the codes of the PC_REG unit. As shown in line 15, the value of pc_o can be recovered to the initial value by the active low reset signal. The reset value is set to ‘prcreset which is set as 32’h0 in the define file. In line 17, if the jump flag is asserted, the value of the pc_o would be set to the destination jump address, and then the processor would fetch the instruction from this address. In line 19, the value of ‘pchold is 3’b001. If the hold flag is asserted, the value of the pc_o would be held. This hold flag is also used by the IFu_IDu and IDu_EXu modules.

```
1  `include "define.sv"
2
3  module pc_reg (
4      input  logic                clk      , // Clock
5      input  logic                rst_n    , // Asynchronous reset active low
6
7      input  logic                jump_flag, // jump flag from ctrl
8      input  logic [`instaddrbus] jump_addr, // jump address from ctrl
9      input  logic [`holdflagbus] hold_flag, // hold flag from ctrl
10
11     output logic [`instaddrbus] pc_o       // output pc pointer
12 );
13
14 always_ff @(posedge clk) begin  proc_pc
15     if(rst_n == `rstenable) begin //if reset signal is asserted, reset pc register to 0
16         pc_o <= `prcreset;
17     end else if (jump_flag == `jumpenable) begin //if jump flag is asserted, jump to destination address
18         pc_o <= jump_addr;
19     end else if (hold_flag >= `holdpc) begin //if hold flag is asserted, stay at current flag
20         pc_o <= pc_o;
21     end else begin //if nothing happens, go to next instruction (pc + 4)
22         pc_o <= pc_o + 4'h4;
23     end
24 end
25
26 endmodule  pc_reg
```

If only the PC register is paused, then the IFu_IDu module and the IDu_EXu module could work independently. If the IFu_IDu module is suspended, the PC register would be also held at the same time. If the IDu_EXu module is paused, the whole pipeline would be paused. RISC-V must be aligned on 32-bit boundaries because of fixed-length 32-bit instructions (i.e. at memory locations divisible by 4) [15], if the processor is working normally, the pc_o would be added by four.

```
always_ff @(posedge clk) begin
    if(!rst_n | hold_en) begin
        qout_t <= dfval;
    end else begin
        qout_t <= din;
    end
end
assign qout = qout_r;
```

Figure 4.2: D flip-flop code

```
logic [`instbus] inst;
gen_dff #(32) inst_dff(clk, rst_n, hold_en, `instnop, inst_i, inst);
assign inst_o = inst;

logic [`instaddrbus] inst_addr;
gen_dff #(32) inst_addr_dff(clk, rst_n, hold_en, `zeroword, inst_addr_i, inst_addr);
assign inst_addr_o = inst_addr;

logic [`intbus] int_flag;
gen_dff #(8) int_ff(clk, rst_n, hold_en, `intnone, int_flag_i, int_flag);
assign int_flag_o = int_flag;
```

Figure 4.3: IFu_IDu code

.

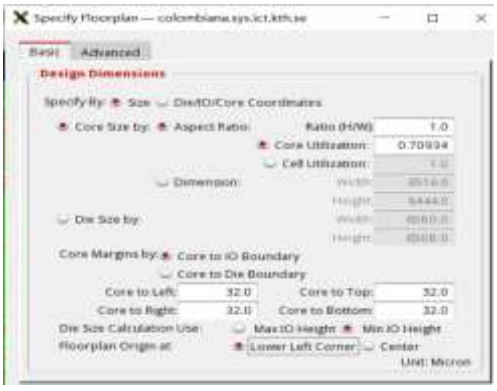
Figure 4.4: IDu_EXu code

Figure 4.5: C2 internal signal connection

4.1.2 GPIO

In the GPIO, there is a 32-bit control signal and a 32-bit data signal. Each I/O is controlled by two bits. There are three modes, input, output, and high impedance state respectively. Hence, at most 16 I/O can be controlled in this module. As shown in figure 4.16, the GPIO can be written. When the write enable signal is asserted and the addr_i[3:0] is 4'b0000, the input data would be written to the control register. If the addr_i[3:0] is 4'b0100, the input data would be written to the data register. When the write enable signal is not asserted and every two bits in the control register is 10, the input value io_pin_i would be written to the corresponding I/O. The reading process is similar to the writing process.

4.2 Place & Route



5.RESULTS AND ANALYSIS

In this chapter, all the necessary results are presented and analysed. In the previous section, the verification is discussed briefly. Because most of the content is like previous paper, only the simulation part is emphasized. The second section records the area estimation and the timing report for both synthesis. The last section illustrates the layout and area report after physical design.

5.1 Verification and Simulation

In this section, the verification and simulation for the processor is explained. Compared to the previous works, the simulation part is newly added. Hence, the simulation part is discussed in detail.

5.1.1 Verification

The verification strategy for each module is the same as the previous paper. Hence, the individual verification of each module is not discussed here. This subsection focuses on the verification of the processor core and the top-level design.

RISC-V core verification strategy

For the RISC-V core verification, the aim of this verification environment is to test the whole data flow from the input to the output of the RISC-V core. The functionality of the core with the bus or the other peripherals like ROM, RAM and GPIO is not verified by this environment. The PC_REG unit needs to modify in a way that it increments in steps of 1 and not steps of 4 because of the temporary ROM in place. This is because the temporary ROM is an array whose width is 32 bits. Since there is no memory which is used to store instructions and then pass them to the IFu_Idu module, a temporary ROM is applied to the verification. In the testbench, the rst_n signal is asserted for 10ns. After de-asserting rst_n, the bus_inst_i input of the Ifu_Idu unit is probed to check if the instructions are fetched according to the increments of the program counter. A monitor is used to know what instruction is being executed at what clock cycle and the value of each general-purpose register is checked to determine whether the instruction is executing correctly or not. In this verification, it cannot randomize instructions to the input port (bus_inst_i). This is because when there is a branch instruction, the core needs to fetch instructions according to the corresponding change in the program counter, but randomizing instructions would hide this effect.

RISC-V top verification strategy

The top-level verification is used to verify the data flow of the RISC-V design, which includes the core integrated with the bus, the ROM, the RAM and the GPIO. The verification plan can be divided into two types. One is a directed test. Another is a randomized test. For the direct test, the clock port is modified with a multiplexer to select between the system clock and the test clock. The test clock would be used to load the instructions into ROM. After loading the instructions, the system clock would be turned on for the core. All the input of the ROM module should be added with a multiplexer, which is applied to select between the

INITIAL_TEST PHASE and POST_INITIAL_TEST PHASE. The former is used to load all instructions into the ROM module, and the latter is where the core takes over the control signals of the ROM. The randomized test covers a wider range of instructions, both in terms of registers and immediate values as well as the type of instructions. The Design Under Test (DUT) clock port and the ROM ports need to be modified in the same way as the directed test. A separate class for every type of instruction should be defined first. Each class of instruction has its own constraints to ensure the instructions are structured according to the ISA. However, in order to ensure the test can be controlled better, the branch (B-type) and the jump (J-type) instructions are not randomized. The other instructions are generated through constrained randomization and then loaded into the ROM in order. An assertion is used to check if randomization passed or not for each class. In addition, there is a monitor class to collect coverage for the source register 1 (rs1), source register 2 (rs2), destination register (rd), immediate values, and the ROM address written into.

RANDOMIZED TEST RESULT

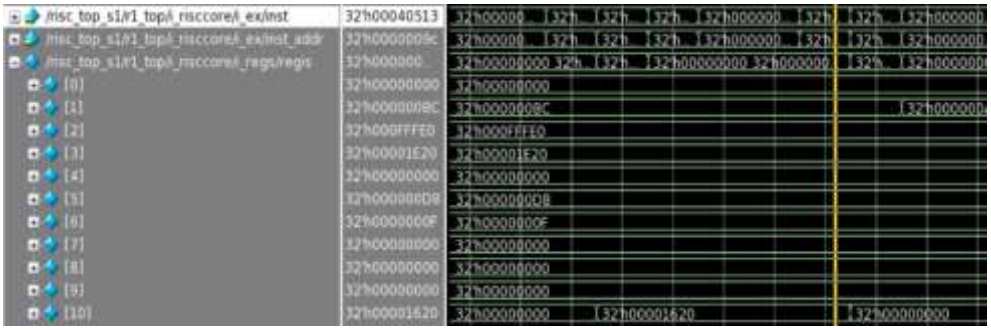
As mentioned before, in order to keep track of all necessary objectives, the cover groups are defined for the source register 1 and 2 (rs1, rs2), the destination register (rd), the ROM addresses, and the immediate values. Table 5.1 shows the hit rate of the bins defined in the cover groups. The expected hit rate of the source and destination register bins was expected to be more than or equal to 50%. This was because the constrained randomization used only sets of registers for generating a few instruction types, such as the R-type and the S-type. On the other hand, the ROM was about 4K bytes, and it was not possible to exercise all the locations for sign-off. Only half of the memory is used for storing instructions.

Table 5.1 Hit rate of the bins

Covergroup	**Metric**	**Goal**
Coverpoint cg::RS1	68.5%	100
Coverpoint cg::RS2	52.2%	100
Coverpoint cg::RD	70.6%	100
Coverpoint cg::ROM_ADDR	50%	100
/risc_top/monitor_risc/cg	60.1%	100

5.1.2 SIMULATION

The main goal is to compare the designed processor with a RISC-V processor produced by the virtual platform tool Imperas OVP to check if the designed processor works correctly.



5.2 Digital Synthesis

In this section, the synthesis part is discussed. The sizes of ROM and RAM are too large, and insufficient resources are allocated. It may cause insufficient permissions for the synthesis tool. The top-level synthesis needs too much time. Hence, only the processor core is synthesized using Genus in this paper.

5.2.1 SOI_STDLIB

The SoI standard library is used as the target library in the synthesis. The main goal is to get the area estimates, check for timing violations, as well as get the Verilog netlists and constraints. Some information is used for physical design. Figure 5.5 indicates how to load the liberty timing files and the System Verilog code. After that, the constraints should be created for the clock, inputs, and outputs, sets loads, etc. The constraints are shown in figure 5.6. The period is set to 100 nanoseconds. Finally, it performs a generic synthesis, maps it to the available

cells in the library, and then runs optimization. The Verilog netlists and constraints are written out, and the area and timing reports are also stored.

5.2.2 Lsi_10k

In the previous paper, the Lsi_10K is used as the target library, which comes with Synopsys DC. Therefore, synthesis with the Lsi_10K library is necessary, as it can be used for comparison. The script for running synthesis with Lsi_10K is quite similar to the above one. In the load sources part, only the path of the library and the library name should be modified. Moreover, the operating condition is set to nominal. For the constraints part, there are no block constraints for the Lsi_10K library. The other setting is the same as the SoI standard library.

5.2.3 Digital Synthesis result

Table 5.3 also illustrates the area and time report about *SoI_STD* library. The cell count is around 23877. In this library, the area unit is defined as μm^2. Therefore, the total area is around 24.177 mm^2. It is less than 49 mm^2 (7mmx7mm), which is the largest area that our in-house SoI technology allows. The design also has a positive slack of about 0.5ns.

Library Name	Cell Count	Cell Area	Slack time
SoI_Lib	23877	25176960 μm^2	548*ps*
Lsi_10K(new)	13292	36077	4327*ps*
Lsi_10K(pre)	9154	24688	8654*ps*

Table 5.3: Synthesis results of each library

5.3 Place & Route

The final layout is shown in figure 5.8. The final core utilization is quite high. The I/O pins are allocated on every side. A more detailed and clearer picture is available on Innovus. Figure 5.9. This is because the optimization requires the insertion of additional cells like buffers.

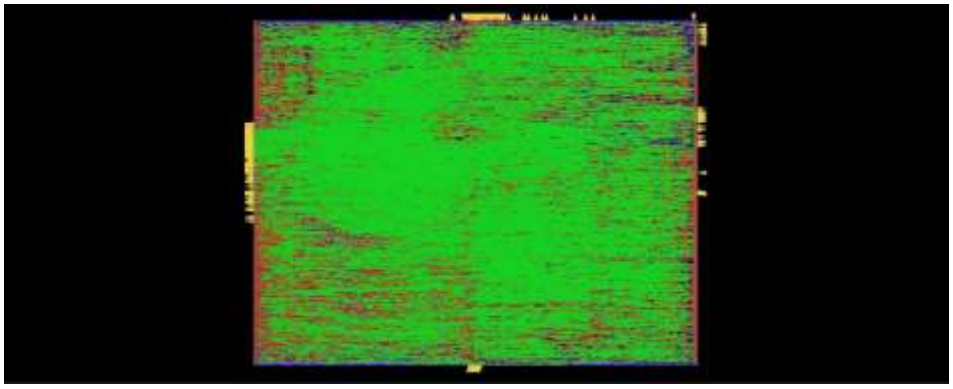


Figure 5.8: Final routed layout

Figure 5.9: Area report of final routed layout

```
innovus 1> report_area
   Hinst Name       Module Name         Inst Count          Total Area
-----------------------------------------------------------------------
risccore                                    26384         28308528.000
 i_clint           clint                    672           902016.000
 i_csr_reg         csr_reg                 1797          2351376.000
 i_div             div                     2042          2486160.000
```

6.1 CONCLUSIONS

This new computer chip is made with a basic design called RISC-V. It has a small processor inside with three main steps it follows when doing tasks. It can understand and do basic math operations like division and getting remainders. Also, some extra parts have been added to help manage certain things.

The chip has a built-in memory where it stores important instructions. This makes it faster because it doesn't have to ask for instructions from somewhere else. But this way of doing things might not be standard and could need changes later.
It can connect to up to four other parts to work together, but if we want it to connect to more, we'll need to make some changes.
We've tested and made sure it works well by trying different tasks on it. It's also been turned into an actual chip that can be used in real devices. When we checked its size and speed, it seems to be okay for now, but we might be able to make it better in the future.

6.2 FUTURE WORK

We can make the chip better by adding more parts, like tools for debugging or extra features such as a timer or communication tools. This would make it more useful.
In the part of the chip that handles interruptions, we can add more ways to deal with different types of interruptions. We can also make tasks run faster, which would save time and make the chip work more efficiently. We can improve how the chip's pipeline works by adding more stages to it, making it faster and able to handle more instructions at once.
Some important parts of the chip, like connections to the outside world and temporary storage, still need work. We should also try to make the chip take up less space by making the building blocks we use more efficient.
When the chip is working on certain tasks, there are moments when it's not doing anything, which slows things down. We can speed this up by checking if the next task needs the result of the previous one.
For tasks like jumps or branches, we can make the chip guess which path to take first. If it guesses wrong, it can try again. Right now, the chip can only connect to four other parts, but we might need it to connect to more. We need to find a way to let it connect to as many parts as we need.

REFERENCES

[1] Design and Implementation of a RISC-V Soc for Real-Time Epilepsy Detection on FPGA. DOI: 10.1109/MSN53354.2021.00037
[2] Design and Implementation of a RISC V Processor on FPGA, 2022. DOI: 10.1109/MSN53354.2021.00037
[3] O. A. Rusanu, et al. " LabVIEW and Android BCI Chat App Controlled By Voluntary Eye – Blinks Using NeuroSky Mindwave Mobile EEG Headset," 2020 International Conference on e- Health and Bioengineering(EHB), 2020, pp. 1 - 4, DOI : 10.1109/EHB50910.2020.9280193.
[4] Y. –Y. Hsieh, et al. -H. Yang, "A 96.2nJ/ class Neural Signal Processor with Adaptable Intelligence for Seizure Prediction," 2022 IEEE International Solid - State Circuits Conference (ISSCC), 2022, pp. 1-3, DOI : 10.1109/ISSCC614.2022.9731759.
[5] Hi Rui, "Design and Implementation of an Extended Instruction Microprocessor Based on RISC-V Architecture [D]." Beijing University of Chemical Technology, 2021. DOI: 10.26939/d.cnki.gbhgu.2021.001318
[6] J. Tranter, "What is RISC-V and Why is it Important?," May 12, 2021. https://www.ics.com/blog/what-risc-v-and-why-itimportant (accessed Jul. 01, 2022).
[7] S. H. Loh, I. M. Tan, and J. J. Sim, "VLSI Design Course with Commercial EDA Tools to Meet Industry Demand – From Logic Synthesis to Physical Design," in 2021 11th IEEE International Conference on Control System, Computing and Engineering (ICCSCE), Penang, Malaysia, Aug. 2021, pp. 55– 60
[8] S. Arif, M. Arif, S. Munawar, Y. Ayaz, M. J. Khan, and N. Naseer, ''EEG spectral comparison between occipital and prefrontal cortices for early detection of driver drowsiness,'' in Proc. Int. Conf. Artif. Intell. Mechatronics Syst. (AIMS), Apr. 2021, pp. 1–6.
[9] Synthacore "SCR1 RISC-V Core" GitHub 2021 https://github.com/syntacore/scr1
[10] IC-Lab-DUTH Repository. (2023) RISCV two-way superscalar processor and random instruction generator. [Online]. Available: https://github.com/ic-lab-duth/DRIM-S.