GB3: Final Report

# Introduction:

RISC-V processors are built upon the RISC-V instruction set, which is a contemporary, open-source architecture aimed at fostering open innovation and facilitating the development of open-source CPU designs. The objective of this project is to optimize the design of a RISC-V processor, with a focus on enhancing performance, reducing power consumption, and minimizing resource utilization, ultimately leading to a Pareto optimal design. To achieve the Pareto optimal design, we propose improvements to the CPU's architecture after conducting a thorough analysis of its design and measuring its runtime performance. The motivation behind pursuing a Pareto-optimal design is that prioritizing one characteristic of the CPU at the expense of others would result in an inadequate general-purpose CPU. For instance, favouring performance improvements might necessitate a larger number of components, leading to increased costs, resource usage, and power consumption. In this project, soft-core CPU designs are executed directly on the FPGA, enabling the incorporation of the FPGA's existing built-in logic and function blocks into the CPU's design. For instance, we utilized the FPGA's DSP to decrease the ALU (arithmetic logic unit) runtime, resulting in improved performance (refer to sections 2 and 3).

To begin, we implemented a RISC-V processor on a Lattice ICE40 UltraPlus FPGA board and utilized it as a soft-core. This approach allowed for rapid design modifications and testing. Furthermore, it facilitated I/O interfacing with the CPU, enabling measurement of the CPU program runtime (detailed in section 4 - test procedure). To approximate the number of instructions executed by each program, we employed the Sunflower CPU simulation suite. Combining these parameters, as explained in Section 4, we determined program runtime, total number of instructions executed, clock rate, and cycles per instruction (CPI) – a crucial metric we aimed to improve. Additionally, custom test programs were written in C to specifically emphasize various aspects of the CPU, such as the branch predictor. These specialized tests aided in better analysing the benefits of the modifications (further details provided in Section 4). Within our team, I focused on enhancing the CPU's performance. CPI improvements were pursued through architectural enhancements, particularly targeting pipeline stalls and hazards, while the clock rate was enhanced by reducing the critical path. The design process involved an iterative approach, with each improvement being tested by analysing the output of synthesis and place-and-route tools, as well as evaluating the performance of the modified CPU designs when executed on the FPGA. In addition, the aforementioned custom C tests were used to verify the performance and correctness of the modified design. For example, when the ALU was replaced with a DSP block, the correctness of the implementation was checked to ensure that the DSP provided the same numerical results as the original ALU block.

[FINAL PARAGRAPH STATING CLEAR NUMBERS THAT INDICATE IMRPOVEMENT IN CPU PERFORMANCE]

# Design Strategy

Multiple strategies were employed to enhance the processor's performance, with a primary focus on augmenting the CPU's clock cycle. The clock cycle is crucial as it synchronizes the sequential components of the CPU, enabling them to respond at the positive edge of the clock signal. By increasing the clock cycle, the processor's runtime is directly reduced. This inverse correlation between the program's runtime and the CPU's clock frequency suggests that elevating the clock frequency decreases the overall cycle count required for program execution. For example, if the CPU requires 10 cycles to execute a single instruction and a program consists of 1000 instructions, the program will be completed in 10,000 cycles. This example clearly illustrates the advantages derived from higher CPU clock frequencies (cycles per second). However, it is essential to acknowledge that an increased clock rate can potentially amplify power consumption in the CPU. Power usage is predominantly attributed to the switching of CMOS logic within the FPGA. Consequently, a higher clock rate results in more frequent logic switches per unit time, leading to escalated power consumption. Nevertheless, when the program's runtime is significantly shorter, the power draw occurs over a shorter duration. Thus, striking a delicate equilibrium between performance and power efficiency depends on the intended purpose of the CPU. Despite these considerations, prioritizing performance improvement is favoured since the CPU can be deactivated during periods of inactivity. It is worth noting that employing clock gating in a more advanced design could mitigate static power consumption. However, due to time and subject knowledge limitations, this approach was not pursued in this project.

To increase the CPU's clock cycle, the critical path length, which represents the maximum combinatorial path between clock cycles, was reduced. The critical path has a critical propagation delay, which determines the minimum clock period and limits the maximum clock frequency. Analysing the bubble sort algorithm on the original processor design revealed a critical path comprising 50 logic levels and a path delay of 74.30 nanoseconds, as shown in Figure \_ & \_ (see appendix). This path traverses the forwarding unit and experiences delays due to carry signal propagation in the ALU, as evidenced by the hierarchical signal net names in the place-and-route tool's terminal output. To reduce the critical delay, several strategies were employed. Initially, increasing the pipeline depth by making the multiplexers and forwarding units sequential was attempted. This approach has drawbacks, as each instruction takes more clock cycles to complete, but it increases the potential clock cycle. Section 3 will conduct calculations to determine the maximum feasible pipeline depth and assess whether it translates into improved performance despite the lower CPI (cycles per instruction). I also implemented addition and subtraction in the ALU using the FPGA’s DSP blocks, as they are designed to carry out addition and subtraction as fast as possible.

In addition, I worked on improving the branch prediction mechanisms of the CPU. This is important as it improves the CPI of the processor (cycles per instruction). This means that for the same clock rate, we can expect the program to complete faster. This is great as it does not increase power consumption meaningfully and thus leads us closer to a Pareto-optimal design. Control hazards in a CPU’s pipeline are caused when the pipeline makes an incorrect prediction about a previous branch instruction, and loads instructions that are no longer need. This means that the CPU needs to stall, flush the pipeline, and restart from before the hazard. This can take hundreds of clock cycles, and therefore leads to increased program runtime. If we design a better branch predictor, we expect the pipeline to correctly predict the outcome of branching instructions more often, leading to fewer pipelines stalls. There are many types of branch predictors, but I focused on implementing a local branch history predictor, where the outcome of each branch instruction is stored separately. Given more time, I would have implemented a global history predictor, or a more advanced algorithm such as gselect/gshare (which are optimum combinations of a local and global history predictors). However, this would have resulted in much greater resource usage (see section 3), and was complex to implement in the short period of time available. Sections 3 and 4 contain more specific information regarding the particular choice of branch predictor, and why a specific configuration of the predictor was used.

# Design Description & Problems Encountered

Plan: Instantiate DSP’s, branch prediction, attempt to make multiplexers sequential

In an effort to enhance the performance of the Arithmetic Logic Unit (ALU), I initially explored replacing the circuitry responsible for addition and subtraction with the Field-Programmable Gate Arrays (FPGA) Digital Signal Processing (DSP) blocks. This process involved two approaches: instantiating FPGA blocks explicitly by defining them in the Verilog code, or allowing the synthesis program to infer the appropriate blocks automatically. However, the initial synthesis results revealed that the FPGA's DSP blocks were not utilized. To address this, I explicitly defined the DSP blocks as 32-bit adder/subtractors within the ALU's Verilog code. Since the DSPs operate on a clocked component, the global clock signal needed to be passed to them, even if the inputs and outputs were not registered. Wiring the 10+ input signals of the DSP blocks was necessary for their implementation. To expedite the DSP implementation, I opted against using a separate module to handle the switch between addition and subtraction. Such an approach would have required modifying the input parameters before providing input signals, necessitating detailed timing analysis. While this approach may not have been ideal in terms of resource usage, it could have been addressed in subsequent iterations of the CPU design. However, due to the project's limited timeframe, I decided to avoid utilizing a single DSP. Additionally, I investigated the potential for using the DSP blocks to optimize other ALU operations, particularly the branch-enable operations that involved comparisons between numbers. These comparisons proved to be the critical path, as confirmed by the output terminal.

One potential improvement involved performing comparisons between signed numbers using DSP subtraction, which is highly efficient, and subsequently examining the Most Significant Bit (MSB) to determine whether the result was positive or negative, thus establishing the order of the numbers. Unfortunately, due to time constraints, I was unable to implement this approach. In retrospect, it would have been a relatively straightforward enhancement, and had I known that my other attempts would be unsuccessful, I would have prioritized this approach earlier in the project. During the process of instantiating the DSP blocks, I encountered several challenges. Initially, I faced difficulties in correctly configuring the DSP by determining the appropriate parameters, as the available documentation was occasionally unclear. Consequently, the synthesis tool failed to recognize my configuration. To overcome this hurdle, I delved into the source code of the open-source synthesis program to identify the cause of the error message, which is depicted in Figure [-]. After locating the relevant parameters in the source file, I successfully instantiated the DSPs. To verify their correct configuration, I developed a program that tested the ALU's accuracy and indicated successful execution through a specific LED pattern. Further information on this testing methodology is given in section 4.

After implementing the DSP blocks, I attempted to greatly reduce the critical path of the processor. I reasoned that, as an increase in the clock cycle resulted in vast improvements in the CPU’s performance, most changes to improve the CPU’s max clock rate would outweigh any other minor improvements that I could create. I attempted to do this by making every multiplexer sequential (rather than combinational), and making the forwarding unit, ALU and branch predictor sequential as well. The motivation behind this is that the CPU’s clock max frequency is limited by the maximum (critical) signal propagation delay, and as such, reducing it will greatly increase the CPU’s frequency. This would have an effect that is ultimately similar to pipelining, but with a much larger CPU frequency possible. However, this does have a downside, as the CPI of the processor would drastically decrease. The decision on whether it is worth clocking all of the CPU’s components is dependent on a trade-off based on the typical path lengths, the critical path length and number of combinational components between existing pipeline registers. If the critical path length is much larger than the typical path length, it makes sense to remove the large critical path length by clocking the components in that path. This is because the rest of the paths will have a small propagation time, allowing the clock rate to be increased greatly. If the critical path is similar to the other path lengths in the circuit, it does not make sense to remove this critical path by synchronising the combinational components of the CPU, as there will be a small benefit outweighed by a much worse CPI.

After considering the benefits and drawbacks of this strategy, I decided to pursue it. This is because it was not possible for to tell what the average/mode/median path length accurately, so I could not tell if the second longest critical path would be much shorter than the current critical path. I therefore decided to attempt to implement these changes, and observe the effects that they would have the CPU’s performance. Unfortunately, I was unable to apply these changes successfully, meaning that I was never able to determine if this was a wise strategy to pursue. After synthesising a design where every multiplexer was synchronised, I reached a critical frequency of 55MHz+, which was far higher than I expected. However, the design was not functional, so it was necessary for me to discount this result from the rest of my report.

After some consideration, I believe I have worked out what the issues with this technique were. The canonical 5-stage pipeline in most CPUs (fetch, decode, execute, memory access, writeback) occurs for all instructions, so there are relatively few direct timing issues (once the processer is designed correctly), assuming that pipeline hazards do not happen often. However, by clocking (synchronising) every combinational component in the circuit, including multiplexers, I am essentially creating pipelines of different lengths for each instruction that is carried out, depending on how many multiplexers the signal has to pass through and if the forwarding unit is used for that instruction. This is why I believe I faced issues when implementing this design, and was unable to get this modified CPU design to run programs successfully. I also attempted to synchronise only one component at a time, but that failed for the same reasons stated above. In the future, if I was to reattempt such a design change, I would first re-architect the CPU’s components, such that it was easier to determine the number of components (i.e., depth of pipeline) that each instruction’s action had to pass through. I would then break-up the pipeline further using long cascaded registers, as the design currently does. This would be more complex, and likely require much higher resource usage (due to an increased number of registers being used, most of which are over 150 bits long), but would result in a deeper pipeline. This is the approach used by modern CPUs – the A-series ARM processers have a pipeline depth of 15-20 stages, while high-performance desktop processers can exceed 30 stages in their pipelines.

# Results & Test Procedure

# Conclusions

# Appendix



