## 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 was 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 proposed 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.

Overall, our group was able to achieve the following results (see Fig 1.1), where it can be observed that a trade-off was made for increasing performance (with an increase in CPI and decrease in runtime) at the expense of increased resource usage. These results were found using bubblesort (with minor modifications), as it is a comprehensive test of all of the CPU’s components. Further results are included in section 4.

|  |  |  |
| --- | --- | --- |
| Attribute | Original | Improved design |
| Resource usage (logic cells) | 3128 | 3961 |
| Bubblesort runtime (seconds) | 4.192 | 1.994 |
| Clock frequency | 6 MHz | 12 MHz |
| CPI | 1.21 | 1.14 |
| Power consumption (mA) | Not meaningfully different across any of the 6 measurement points | |

Figure 1.1: Comparison of original and final improved design

# 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 enhance the CPU's clock cycle, efforts were made to reduce the critical path length, which represents the longest combinational path between clock cycles. The critical path includes a critical propagation delay that determines the minimum clock period and limits the maximum clock frequency. Examination of the bubble sort algorithm on the original processor design unveiled a critical path consisting of 50 logic levels and a path delay of 74.30 nanoseconds, as depicted in Figure [-] in the appendix. This path traverses the forwarding unit and encounters delays caused by carry signal propagation in the ALU, as indicated by the hierarchical signal net names in the terminal output of the place-and-route tool. Multiple strategies were employed to diminish the critical delay. Initially, increasing the pipeline depth by synchronising the multiplexers and forwarding units was attempted. While this approach extends the clock cycles required for each instruction to complete, it raises the potential for clock cycle improvement. Section 3 will conduct calculations to determine the maximum viable pipeline depth and evaluate whether it results in enhanced performance despite a lower CPI (cycles per instruction). Additionally, the ALU's addition and subtraction operations were implemented using the FPGA's DSP blocks, as these blocks are optimized for rapid execution of these operations.

Furthermore, attention was given to improving the CPU's branch prediction mechanisms. This is significant as it positively impacts the processor's CPI (cycles per instruction), enabling faster program completion without substantial increases in power consumption and bringing us closer to a Pareto-optimal design. Control hazards in the CPU's pipeline occur when incorrect predictions are made regarding preceding branch instructions, leading to unnecessary loading of instructions. Consequently, the CPU experiences stalls, pipeline flushing, and restarting from a point prior to the hazard. This can result in significant increases in program runtime, as illustrated in Figure [-] in the appendix. By designing a more effective branch predictor, we anticipate a higher accuracy in predicting the outcomes of branching instructions, leading to fewer pipeline stalls. Various types of branch predictors exist, but the focus was placed on implementing local, global, and tournament branch history predictors. Given more time, a more advanced algorithm like gselect/gshare (which combines local and global history predictors optimally) could have been implemented. However, such an implementation would have significantly increased resource usage (as outlined in section 3) and proved complex to accomplish within the limited timeframe. Sections 3 and 4 provide more detailed information on the chosen branch predictor and rationale behind the specific configuration employed.

# Design Description & Problems Encountered

In this section, I will outline the designs that I attempted to implement, outlining any problems that I faced.

### 3.1 Implementing DSP blocks

In the pursuit of enhancing the performance of the Arithmetic Logic Unit (ALU), the initial exploration involved investigating the replacement of the addition and subtraction circuitry with Field-Programmable Gate Arrays (FPGA) Digital Signal Processing (DSP) blocks. Two approaches were considered: explicit instantiation of FPGA blocks by defining them in the Verilog code, or allowing the synthesis program to automatically infer the suitable blocks. However, the initial synthesis results revealed an underutilization of the FPGA's DSP blocks, as depicted in Figure [-] in the appendix. To address this, explicit definition of the DSP blocks as 32-bit adder/subtractors within the ALU's Verilog code was implemented, as shown in Figure [-] in the appendix. 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 to enable their implementation. To streamline the DSP implementation, the utilization of a separate module to handle the switch between addition and subtraction was foregone. While this approach may not have been optimal in terms of resource usage, it could be addressed in subsequent iterations of the CPU design. However, due to the project's time constraints, the utilization of a single DSP was avoided. Furthermore, investigation was conducted to explore the potential optimization of other ALU operations using DSP blocks, particularly the branch-enable operations involving number comparisons, which were identified as the critical path, as confirmed by the output terminal in Figure [-] in the appendix.

One potential improvement considered was performing comparisons between signed numbers using DSP subtraction, which is highly efficient, and subsequently evaluating the Most Significant Bit (MSB) to determine the order of the numbers, thereby ascertaining whether the result was positive or negative. Unfortunately, due to time limitations, the implementation of this approach could not be accomplished. In retrospect, it would have been a relatively straightforward enhancement, and had the unsuccessful attempts been foreseen, prioritization of this approach would have been considered earlier in the project. Challenges were encountered during the process of instantiating the DSP blocks. Initially, difficulties were faced in correctly configuring the DSP by determining the appropriate parameters, as the available documentation occasionally lacked clarity. As a result, the synthesis tool failed to recognize the configuration. To overcome this obstacle, an examination of the source code of the open-source synthesis program was conducted to identify the cause of the error message, illustrated in Figure [-] in the appendix. After identifying the relevant parameters in the source file, successful instantiation of the DSPs was achieved. To verify the correct configuration, a program was developed to assess the accuracy of the ALU and indicate successful execution through a specific LED pattern. Further details regarding this testing methodology can be found in section 4.

### 3.2: Sequential multiplexers and forwarding units

Following the implementation of the DSP blocks, significant efforts were made to minimize the critical path of the processor. The rationale behind this approach stemmed from the understanding that increasing the clock cycle could lead to substantial improvements in the CPU's performance, thus overshadowing other minor enhancements. Synchronising every multiplexer and incorporating sequential elements into the forwarding unit, ALU, and branch predictor were pursued as strategies to achieve this objective. The underlying motivation was rooted in the fact that the CPU's maximum clock frequency is determined by the critical signal propagation delay, and reducing it would greatly enhance the CPU's frequency potential. While this would resemble pipelining, the resulting CPU frequency would be significantly higher. However, this approach entailed a drawback in the form of a drastic decrease in the processor's CPI. The decision to synchronize all of the CPU's components hinged upon a trade-off analysis based on the typical path lengths, critical path length, and number of combinational components between existing pipeline registers. When the critical path length greatly exceeded the typical path length, it was reasonable to synchronize the components within that path to eliminate the extended critical delay. This was because the remaining paths would possess minimal propagation time, thereby enabling a substantial increase in clock rate. Conversely, if the critical path length was comparable to other path lengths in the circuit, synchronizing the combinational components of the CPU to remove the critical path would yield negligible benefits outweighed by a significantly worsened CPI.

After carefully considering the merits and drawbacks of this strategy, it was chosen to be pursued. The inability to accurately determine the average, mode, or median path length necessitated an exploration of potential improvements by implementing these changes and observing their impact on the CPU's performance. Regrettably, successful application of these modifications was not achieved, thus preventing a definitive assessment of the viability of this strategy. Despite synthesizing a design in which every multiplexer was synchronized and achieving a critical frequency of over 55MHz, surpassing initial expectations, the design itself was non-functional. Consequently, this result had to be disregarded in the overall report analysis.

Upon careful reflection, I have identified the potential issues associated with the technique employed. The canonical 5-stage pipeline, encompassing fetch, decode, execute, memory access, and writeback stages (as depicted in Fig. [-], appendix), is common in most CPUs. This standardized pipeline ensures that, assuming proper processor design, there are relatively few inherent timing issues, provided pipeline hazards are infrequent. However, by synchronizing every combinational component in the circuit, including multiplexers, a scenario arises where each instruction executes through pipelines of varying lengths. This is contingent on the number of multiplexers the signal traverses and whether the forwarding unit is utilized for that specific instruction. It is this deviation from the uniform pipeline structure that likely led to the encountered challenges during the implementation of this modified CPU design, rendering unsuccessful program execution. I also experimented with synchronizing only one component at a time, yet encountered similar issues as previously mentioned.

In future attempts at such design changes, a recommended course of action would involve re-architecting the CPU's components. This restructuring would facilitate the determination of the number of components (i.e., pipeline depth) that each instruction's action must traverse. Subsequently, the pipeline could be further segmented using lengthy cascaded registers, similar to the current design. Although this approach would introduce increased complexity and potentially higher resource usage (due to the utilization of numerous registers, many of which exceed 150 bits in length), it would result in a deeper pipeline. Notably, modern CPUs, such as the A-series ARM processors (with pipeline depths ranging from 15 to 20 stages) and high-performance desktop processors (exceeding 30 stages), follow a similar approach to achieve enhanced performance.

### 3.3 Branch Prediction

Finally, I implemented a more advanced branch prediction algorithm than the original implementation. The original design used a 2-bit saturating counter, see Fig. [-], that did not take the specific branch instruction into account. This meant that the differing behaviour of different branch instructions in the program could not be effectively captured by this predictor. In practice it was effective, partially due to the simple nature of programs run on this CPU – they are limited strictly by the FPGA’s resources that limit the size of instruction and data memory. The first type of branch predictor that I implemented was a local branch predictor. In this prediction mechanism, a table is constructed, with the index being the last 5 bits of a branch instructions address, and the entry being a 2-bit saturating counter that provides the prediction. A 2-bit saturating counter is used as it provides hysteresis – it oscillates between strongly taken, weakly taken, weakly not taken and strongly not taken – and ensures that one wrong prediction does not completely change what the predictor predicts, see Fig. [-], appendix. Ideally, we want at least 2 negative results before the predictor has to change its prediction, which is what occurs here. This means that the predictor takes longer to “warm-up” (which refers to populating the prediction table with the most likely states for each branch), but from a wider perspective this is not a concern as programs likely to be affected (have their performance improved) by branch prediction to a noticeable degree will be long enough regardless. 5 index bits are used to index the branch history table (BHT). For very large programs, this could lead to aliasing issues, where there are enough branch instructions that more than one instruction shares the same 5 bit ending. However, this is not a problem in this CPU design as we do not expect very large programs to be run. The advantage of using 5 index bits is that the table is only 32 words deep, which allows for a trade-off between resource usage and performance. The performance increase for small programs with a larger table would not outweigh the drawbacks of increased resource usage – stopping us from creating a Pareto-optimal design. The same argument can be applied to the 2-bit saturating counters, which are large enough to be useful but without wasted resource usage.

The global branch predictor was used as it can detect correlations between branch instructions. The global history register (a shift register) holds the last 5 branch outcomes, a 5-bit word that is later used to index the pattern history table (PHT), which stores the predictions for what we will occur for each global history instance. In this way, we are able to create correlations between branch instructions stored at different addresses, something that the local predictor is unable to achieve. The benefits of a global history buffer can be particularly noticed when dealing with nested loops that contain branching, or programs that cause the local history table to oscillate between weakly not taken and weakly taken. Such patterns (which occur between branching instructions) can only be detected by the global predictor. The global history register is combined with the current program counters’ lower 5 bits, using a XOR, so that a further correlation with the programs current state is created.

However, we are now faced with the problem of having to decide between two predictors. I therefore implemented a simple tournament prediction scheme, where another 32-word deep table is used (with the branch instructions lower 5 bits as an index) is again used to determine whether the local or global predictor is currently providing the more accurate prediction at this time. This is the weakest part of my design, and I do not believe that it currently takes advantage of the fact that there are two different predictors that could be used. This is because it is an elementary scheme that does not combine the information provided by the predictors, instead it just selects a single predictor to use. More advanced designs such as gselect or gshare can better combine the predictors, but have a far greater resource usage. This means that they are useful for larger high-performance CPU’s, particularly those implemented on ASICs – where resource usage is not as high of a concern typically, rather than small experimental cores on FPGAs.

I did not face many problems when implementing the new branch predictors, apart from finding programs that are complex enough to test the predictors performance. A small problem initially arose when I was researching specific branch predictor types to use. As many of these predictors are now too simple for complex modern processers, and CPU IP is typically proprietary, it was hard to find any sources that had a list of particular predictors and their individual advantages and drawbacks. Furthermore, many sources had imprecise definitions of predictors, meaning it was hard to search for them by name. I therefore decided to test each predictor on its merits myself, and attempted to work out how it would behave for different patterns of branches. This was a lengthy process, but helped improve my understanding of the behaviour of branch predictors.

# Results & Test Procedure

### Power consumption:

### Resource Usage:

### Performance (DSP):

### Performance (Updated branch predictor):

# Conclusions

# Appendix



