## 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 6.1 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 6.2 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 6.3 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 6.4 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 6.5 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 6.6 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. A simple fix was employed, however, where the frequency of the design was increased from 6MHz to 12MHz, as the base design was already capable of this.

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. 6.7, 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

I successfully implemented an enhanced branch prediction algorithm, surpassing the original implementation. The initial design relied on a 2-bit saturating counter (refer to Fig. 6.8 in the appendix) that lacked consideration for the specific branch instruction. Consequently, this predictor failed to effectively capture the distinct behaviour exhibited by different branch instructions within the program. However, given the relatively simple nature of programs executed on this CPU, limited by the FPGA's resources that restrict the size of instruction and data memory, the original predictor proved reasonably effective.

To enhance branch prediction capabilities, I first implemented a local branch predictor mechanism. This involved constructing a table where the index corresponded to the last 5 bits of a branch instruction's address, and the entry comprised a 2-bit saturating counter that facilitated the prediction. The use of a 2-bit saturating counter offered hysteresis, oscillating between strongly taken, weakly taken, weakly not taken, and strongly not taken states. This ensured that a single incorrect prediction did not drastically alter the predictor's overall behaviour. Ideally, the predictor required at least two negative outcomes before changing its prediction, a characteristic exhibited in this implementation. Consequently, the predictor took longer to "warm-up" (i.e., populate the prediction table with the most probable states for each branch). However, from a broader perspective, this was not a concern, as programs likely to benefit significantly from branch prediction (thus improving performance) would inherently be long enough.

The local branch history table (BHT) employed 5 index bits to index the table (BHT), which could potentially introduce aliasing issues in large programs. However, in this CPU design, this was not a problem since the expectation was to run relatively small programs. Utilizing 5 index bits resulted in a shallow table of only 32 words, striking a balance between resource usage and performance. Increasing the table size for small programs would not outweigh the drawbacks associated with heightened resource utilization, preventing the creation of a Pareto-optimal design. Similar reasoning applies to the utilization of 2-bit saturating counters, which were sufficiently large to be effective without unnecessary resource wastage.

In addition to the local predictor, I incorporated a global branch predictor to detect correlations among branch instructions. The global history register, implemented as a shift register, stored the last 5 branch outcomes, forming a 5-bit word used to index the pattern history table (PHT). The PHT stored predictions for each global history instance, enabling the establishment of correlations among branch instructions located at different addresses—an achievement unattainable by the local predictor alone. Combining the global history register with the lower 5 bits of the current program counter, using XOR operation, facilitated further correlation with the program's present state. A longer global history buffer would have been able to detect (and therefore predict) longer patters, but would require exponentially more resources as its history table would be larger. It would also be wasteful, as most of the history table would be unused. That is why more advanced prediction mechanisms do not resort to larger history tables, but find smarter ways to retain and use the branch history. I did not implement a branch target buffer due to time constraints, but this would have been a useful addition to the CPU, as memory access is often one of the slowest operations in a CPU.

Detailed diagrams illustrating the structure of the global and local branch predictors can be found in Figures 6.9, and the Verilog code encompassing their implementation is provided in Figures 6.10.

However, the challenge emerged when determining the most suitable predictor between the two. To address this, I implemented a straightforward tournament prediction scheme, incorporating another 32-word depth table (indexed by the lower 5 bits of branch instructions) to determine the currently more accurate predictor—the local or global predictor. Admittedly, this aspect of my design is relatively weak and fails to leverage the presence of two distinct predictors. The current scheme merely selects a single predictor rather than effectively combining the information provided by both. Advanced designs such as gselect or gshare present more robust methods of predictor combination but come at the cost of significantly higher resource utilization. Consequently, such designs prove more applicable to larger, high-performance CPUs, particularly those implemented on ASICs, where resource usage is typically less constrained, as opposed to smaller experimental cores on FPGAs.

The implementation of the new branch predictors encountered minimal difficulties, with the main challenge lying in finding sufficiently complex programs to assess the predictors' performance. During the initial stages, I faced a minor obstacle while researching specific branch predictor types to employ. Due to the simplicity of many existing predictors for modern processors and the proprietary nature of CPU intellectual property, locating comprehensive sources listing individual predictors along with their advantages and disadvantages proved arduous. Moreover, imprecise definitions of predictors in various sources hindered targeted searches by name. Consequently, I opted to conduct independent evaluations of each predictor based on their respective merits, attempting to ascertain their behaviour under different branch pattern scenarios. Although time-consuming, this approach significantly enhanced my understanding of branch predictor dynamics.

# Results & Test Procedure

This section contains the results of the benchmark tests used to evaluate the performance of the improved CPU design. Some comments will be made about resource usage and power consumption to place the data into perspective, but a focus will be kept on performance metrics.

### Performance (DSP):

There were multiple test procedures needed for the DSP’s implementation. Firstly, a test was needed to ensure that the DSP was functioning correctly, and outputting all of the required results. This test is included in the appendix as Fig. 6.11. A variety of basic addition and subtraction operations are carried out in this program. The first few tests check the addition and subtraction of the lower and upper sections separately. The DSP is able to do 32-bit addition and subtraction, but needs the inputs to be separated into 16-bit words. After this, the program tests the addition of numbers that involve a carry operation from the lower 16-bits to the upper 16-bits. The same test is carried out with subtraction, but that involves a borrow instead of a carry. The LED pulse waveform informs us if the program has run successfully, by flashing differently depending on the outcome of the result equality tests. This functional verification was successful, and proves that the DSP design works robustly. The time the LED is on/off can be measured accurately with an USB oscilloscope.

The main benefit of using the DSP is that it results in a reduced critical path delay, which can be observed in Fig 6.12, from (74.14ns) 13.49MHz to (70.34 ns) 14.22MHz. However, this reduced critical delay will only result in improved performance if the frequency of the clock is able to increase. The high-frequency oscillator used to generate the clock signal in the FPGA can only do 6MHz, 12MHz, 24MHz, and 48MHz. Therefore, a small increase in the max clock rate will not be reflected in the performance of the design, leading to identical performance, which can be seen in the results table, Fig. 3.1. Our team attempted to instantiate a PLL to control the clock frequency with greater precision, but as the PLL has a minimum frequency of 16Mhz, it was not useful for our design. Bubblesort and another program that tests mathematical operations were run to test performance, which provided the expected results. We can observe that the runtimes are slightly different, but well within the expected margin of error. Multiple pulses were measured and average to increase experimental accuracy.

|  |  |  |
| --- | --- | --- |
| Processor design | Original design | Modified design with DSP |
| Critical delay (ns) | 74.14 | 70.34 |
| Critical frequency (MHz) | 13.49 | 14.22 |
| Frequency of clock during runtime (MHz) | 12 | 12 |
| Bubblesort runtime (s) | 2.09 | 2.08 |
| ALU test (s) | 0.645 | 0.639 |
| Logic cell count | 3128 | 3085 |

Figure 3.1: A table comparing the runtime of the original and modified designs for bubblesort

Furthermore, it can be observed that the modified design (with DSP) has a lower resource usage than the original design, as is expected – 32-bit full adders will a relatively large number of gates to carry out their function. This is why using DSPs makes the device more Pareto optimal.

### Performance (Updated branch predictor):

The branch predictor is more complex to test. In particular, a local and global history branch predictor require tests with different attributes, as each predictor works in a different way. The local predictor will need a test that has consistent (predictable) behaviour for a particular branch, to show the improvenment in runtime that it can cause, whilst the global predictor will need a test that has correlation between branch instructions, to show its potential improvement.

I therefore created a “branch\_test.c” program that includes, and therefore tests, both of these traits. The “testGlobalBranchPredictor” function tests the global predictor, as it contains a fixed branch history pattern (that is under 5 bits long). Branches occur for every multiple of 2, 3 and 5 up to 10,000, which causes a consistent branch history pattern that is 5 bits long (and can therefore be predicted). The “testLocalBranchPredictor” is a function that tests the local history predictor for each of the branching statements in the function, but also global predictor a little, as there are branched “if” statements that have some correlation. Overall, “branch\_test.c” is a useful program to test the different branch prediction mechanisms, as it has an improved running time for the improved branch prediction mechanism.

Bubblesort instructions = 20825022, Branch\_test instructions = 17559912

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Branch Predictor | Algorithm | Runtime | Logic Cell | CPI |
| Original | Bubblesort | 2.115 | 3128 | 1.219 |
| Local | Bubblesort | 2.108 | 3245 | 1.215 |
| Global | Bubblesort | 2.023 | 3256 | 1.166 |
| Tournament | Bubblesort | 1.994 | 3602 | 1.149 |
| Original | Branch\_test | 1.781 | 3445 | 1.217 |
| Local | Branch\_test | 1.784 | 3602 | 1.219 |
| Global | Branch\_test | 1.722 | 3615 | 1.177 |
| Tournament | Branch\_test | 1.692 | 3961 | 1.156 |

Figure 3.2: Branch Prediction test results

From Fig. 3.2 we can observe that for both algorithms the tournament predictor has the highest prediction accuracy (due to the lowest program runtime, which means we can infer that there were fewer pipelines flushes due to misprediction). The local history predictor is not significantly faster than the original predictor, likely because both predictors are very similar, with the only difference that the local predictor has a saturating counter for each branch instruction. Aliasing in the local history table’s index (between differing branch instructions with the same lower bits) is unlikely, as there are a lot of possible branch instruction addresses and very few share the same lower 5 bits, as up-to 32 bits can vary.

The global predictor is fundamentally different to the original predictor, and has improved results in comparison. This is because both programs have global branching patterns (as described earlier), that can be predicted accurately by the global predictor. The tournament predictor is the best predictor, as it selects between the local and global predictors depending on which one has been more accurate. In this way, we reduce the number of cases where a misprediction occurs, as it is unlikely that both predictors are wrong at the same time. It should be noted that as the modified prediction mechanisms contain 32-deep tables, they are resource-intensive. As branch prediction mechanisms are integral to maintaining high performance in pipelined processors, especially for longer programs, it makes sense to sacrifice resource usage for increased performance.

We can also observe that improving the branch predictor improved the CPI for bubblesort from 1.219 to 1.166, and for Branch\_test from 1.217 to 1.156. This is a result of improving the pipeline performance, as stated above. This is a good result, and proves the hypothesis presented earlier (in section 2) that improvements to branch prediction will have a different impact on the CPU’s performance.

# Conclusions

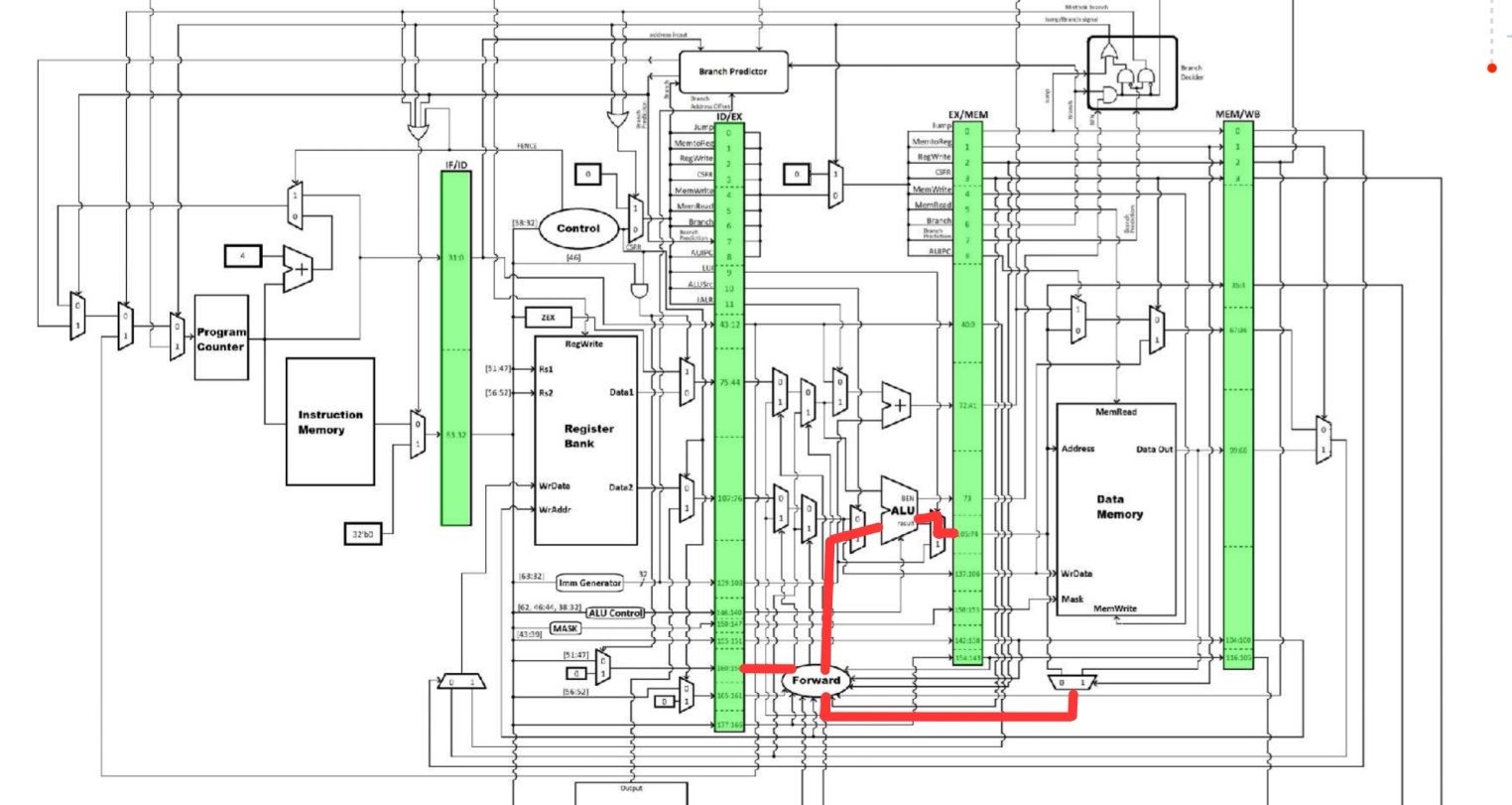
Firstly, we have shown that the improvements made to the processor design have had directly improved the CPU’s performance. This is shown by the improved maximum clock rate from 13.49MHz to 14.22MHz due to the DSP implementation. Furthermore, the CPI of the processor has been improved by 4.3% and 5.0% respectively for the bubblesort and Branch\_test programs. Both of these contributions have pushed the design towards being Pareto-optimal. However, they have resulted in increased resource usage. In particular, the relatively large prediction tables used in the tournament predictor (the most effective predictor), resulted in ~400 extra logic cells being used. This is a trade-off but is necessary as this predictor is the natural next step in progression from the original branch predictor, and will decrease the runtime significantly for programs with lots of branching instructions.

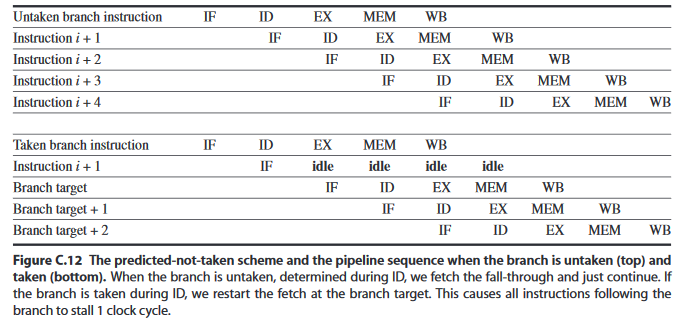
There are several improvements that I would make to the processers design in the future. First, I would implement all comparison operations in the ALU with the DSPs, as I did not have time to do this during the project. It is a simple method to reduce the critical delay path significantly. In addition, I would re-arrange the CPU’s layout slightly so that more pipelining registers could be added. I believe that this would greatly increase performance despite the hit to CPI. As the resource usage is relatively low compared to the FPGA’s maximum, the place-and-route tool is able to generate short routes between components. As such, I think the critical path could be brought down to less than 15ns. If it was 15ns, the critical frequency would be ~65MHz. This would exceed the capacity of the FPGA’s high-frequency oscillator, so a PLL would have to be used instead. However, I would not change the branch predictor in the future. This is because more advanced branch prediction mechanisms such as gshare/gselect will result in much greater resource usage, and are therefore not suitable for simple CPU designs implemented on hobbyist-grade FPGAs. Furthermore, I would implement the use of BRAM (block RAM) for storing the instruction and program memory, as they are currently stored in logic cells. Storing large amounts of data in logic cells is inefficient and very resource-intensive in FPGAs.

I believe that our team co-ordinated well with each other, communicating what we were working on and sharing results promptly. We also gave each other good suggestions on how to proceed when one of was stuck with a particular problem. A weakness in our co-ordination was that we did not have a centrally co-ordinated development plan, but rather each member created a plan for themselves. It would have been beneficial to create such a central plan at the beginning, so that each member was aware of the amount of work required and which components were being worked on at any time. However, I still believe that overall, our team worked very well together.

# Appendix





Fig. 6.1: Output from place-and-route tool for bubblesort critical path for the original processor

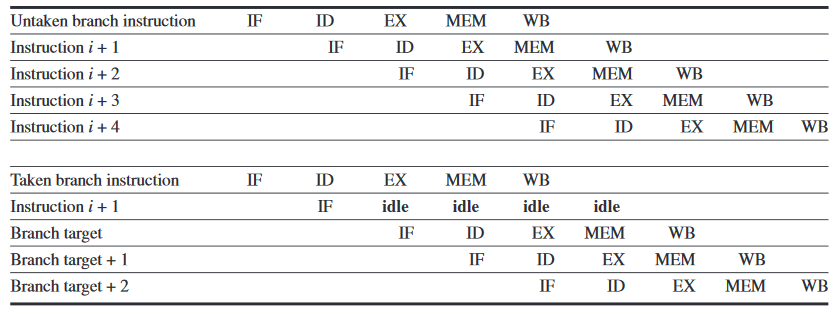


Fig 6.2: Branch hazard in a pipeline – when not-taken is predicted, and it is taken/not-taken in reality (see top and bottom of figure respectively). Source: Computer Architecture by Patterson and Hennessey

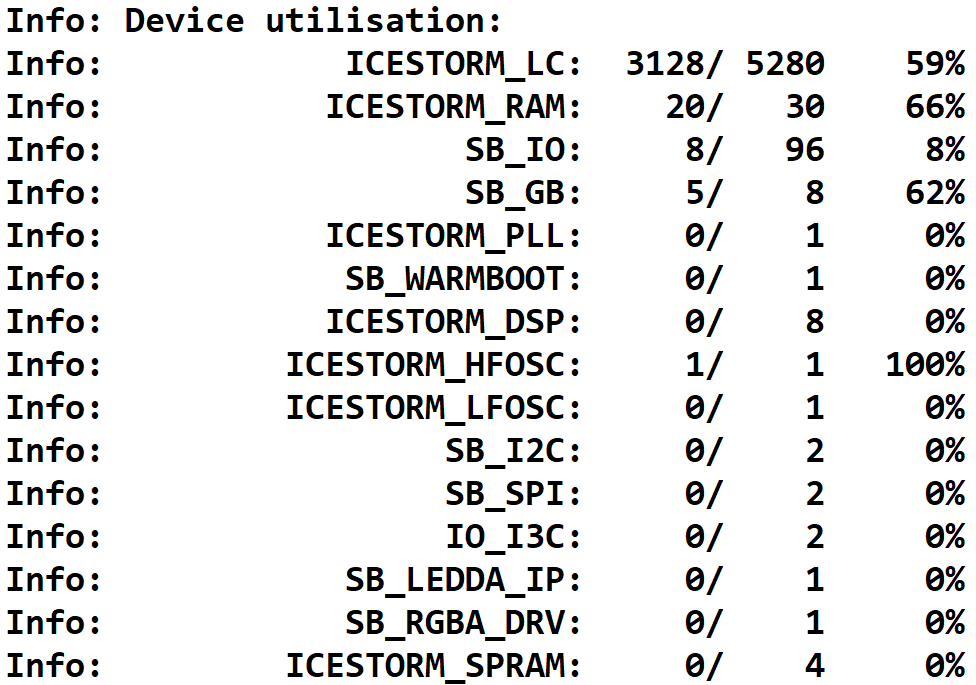
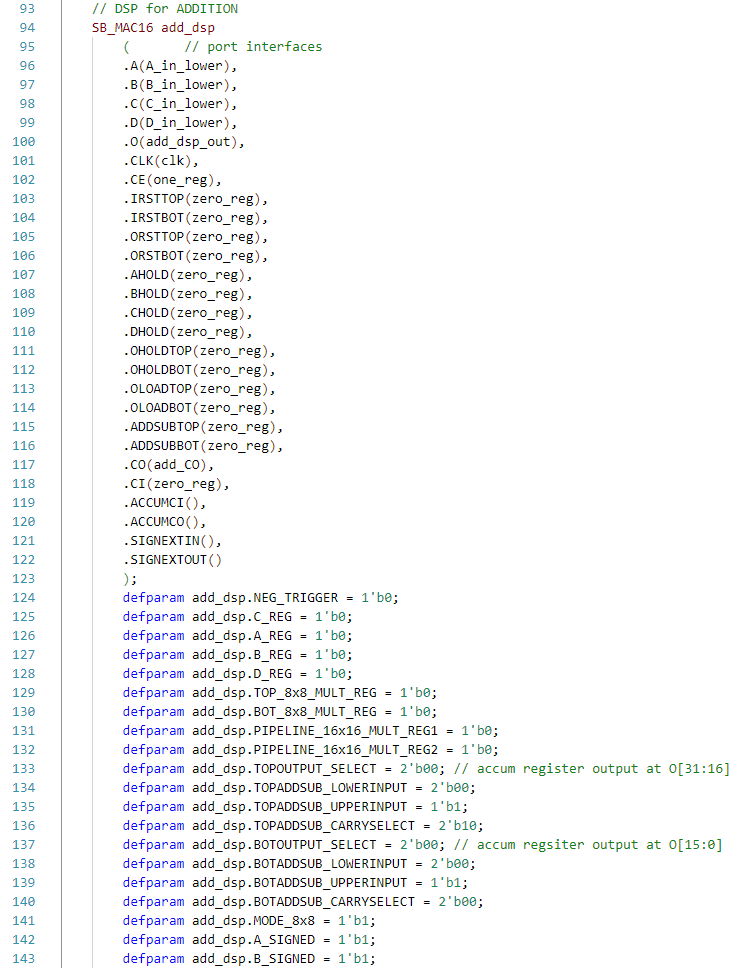


Fig. 6.3: Device utilisation during synthesis of CPU, without using DSP blocks

Fig. 6.4: Explicit Verilog instantiation of DSP block for addition, with all of the parameters defined. Same was done for the subtraction DSP block.

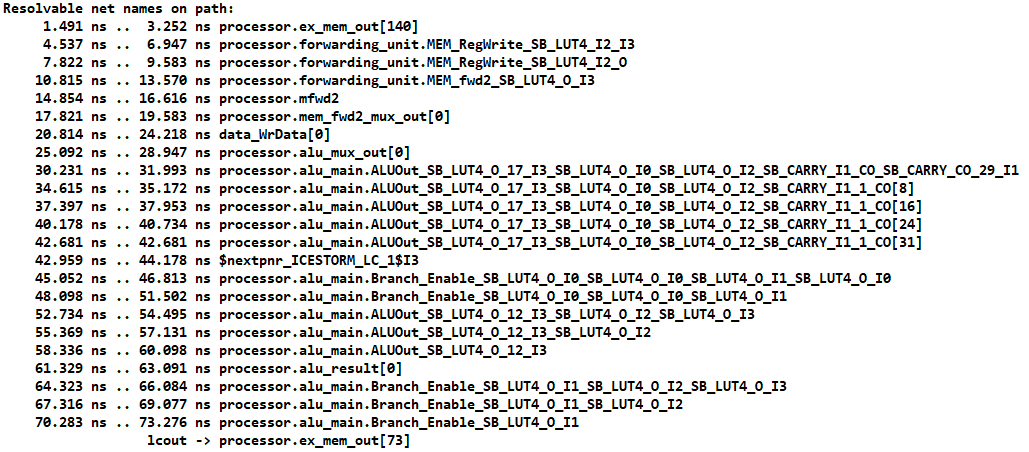
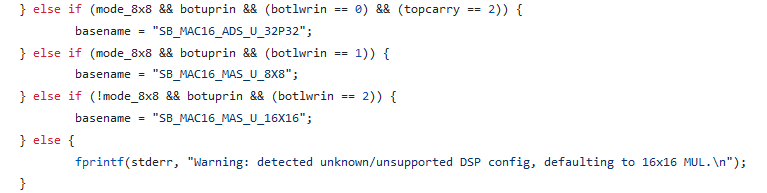
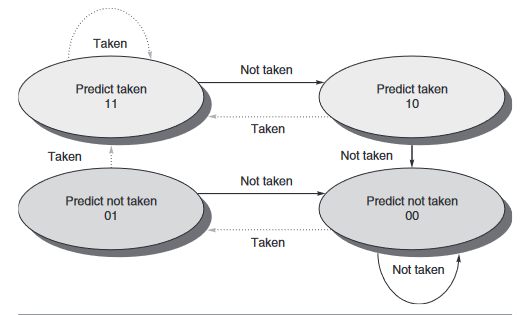
Fig. 6.5: Bubblesort critical path

Fig. 6.6: Error message when DSP was configured wrong

Fig. 6.7: 5-stage pipeline, Source: Computer Architecture by Patterson and Hennessey

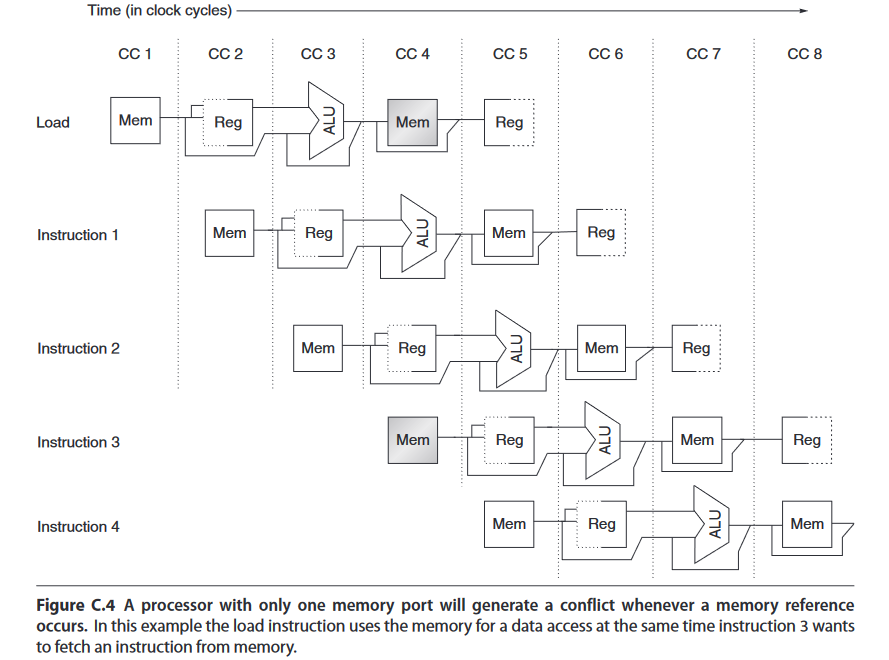
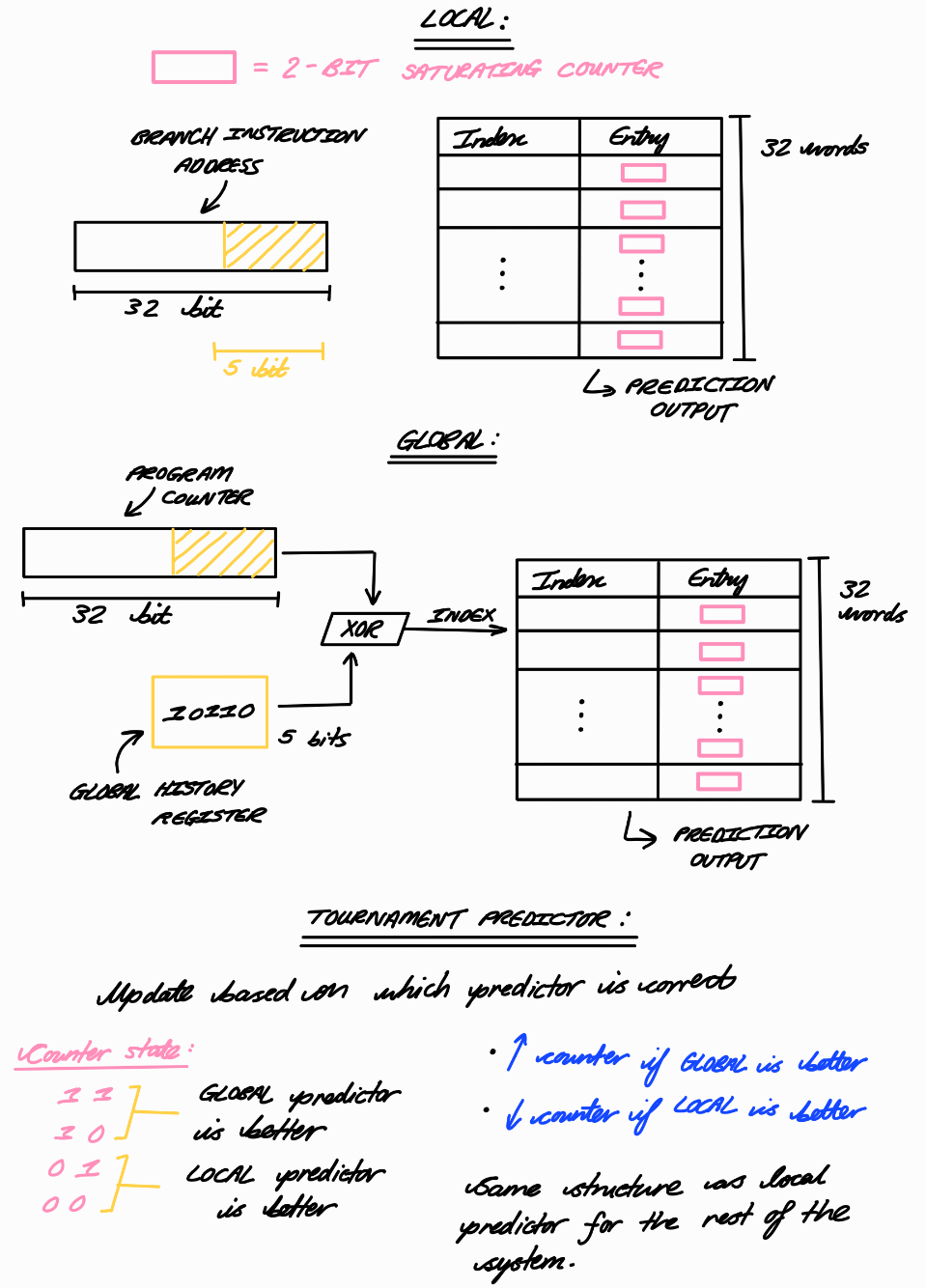


Fig. 6.8: 2-bit saturating counter

Figure 6.9: Local, Global and tournament predictor structures

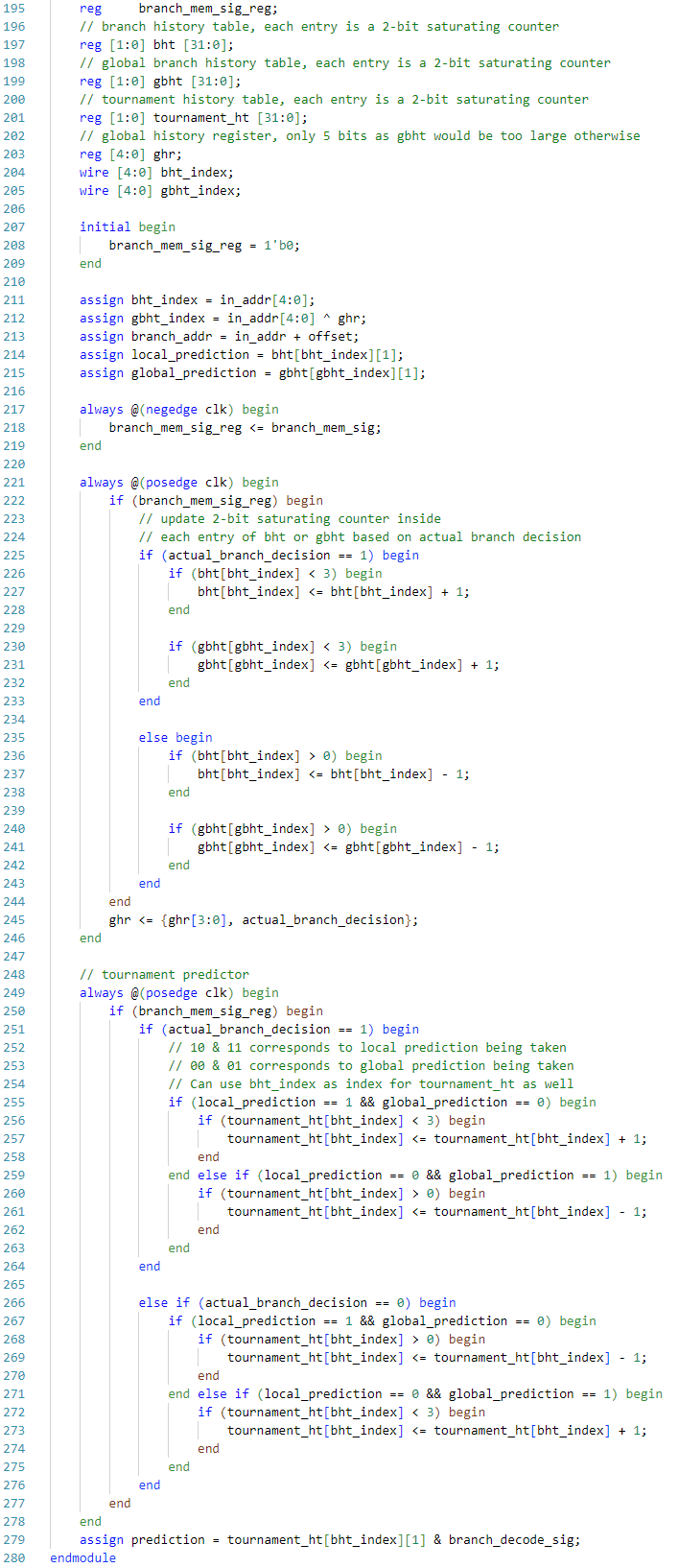


Figure 6.10: Local and global branch predictors with a tournament predictor that combines them

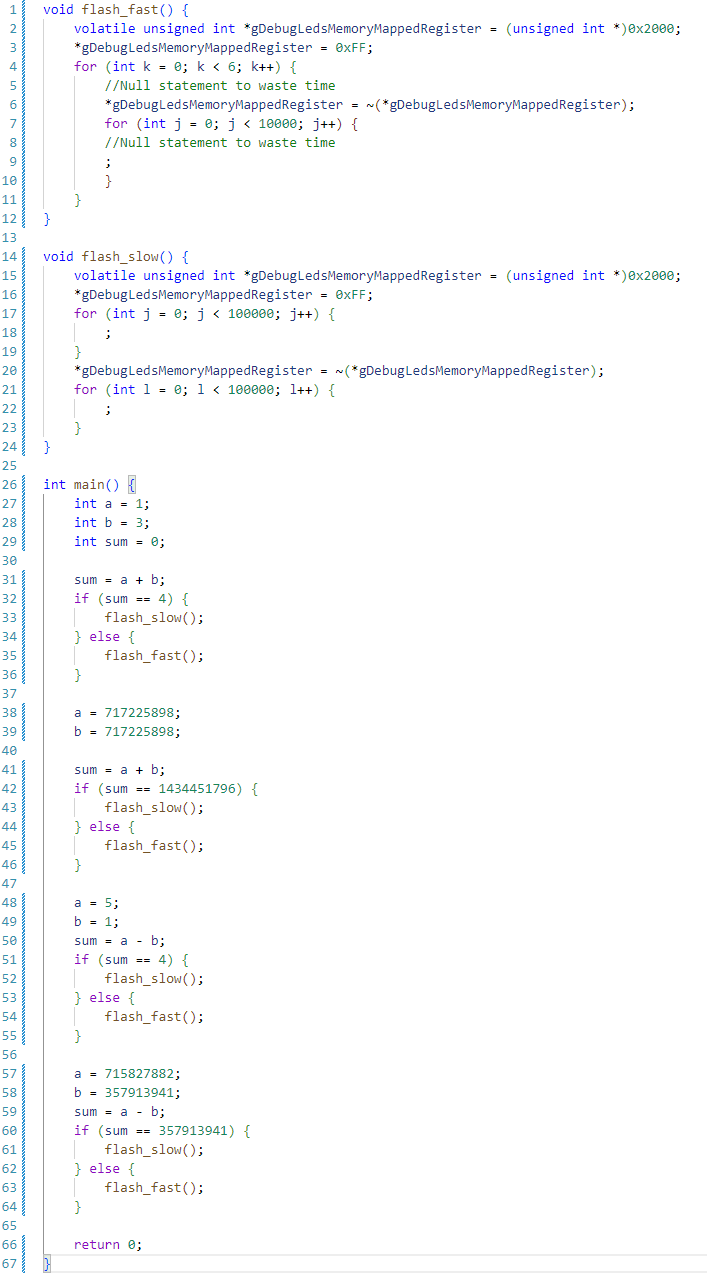


Figure 6.11: modified\_alu\_test.c