Computer Engineering

Electronics and Communication Systems

ADD Functional Unit of a Coarse-Grained Reconfigurable Array with Triple Modular Redundancy Fault Tolerance Technique

Author:
Matteo Biondi

Academic Year: 2022/2023

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem Description and Specifications

Design the Functional Unit(FU) of a Coarse-Grained Reconfigurable Array(CGRA) architecture for performing signed sums in C2. The architecture shall have the following characteristics:

1. Inputs and outputs consisting of 8-bit data and 2-bit flag (carry, overflow, etc.)
2. Ready-valid communication protocol along the entire data chain
3. Application of fault detection and correction Triple Modular Redundancy(TMR) mechanisms at least on the combinatorial part of the architecture.

Below is a simple block diagram:

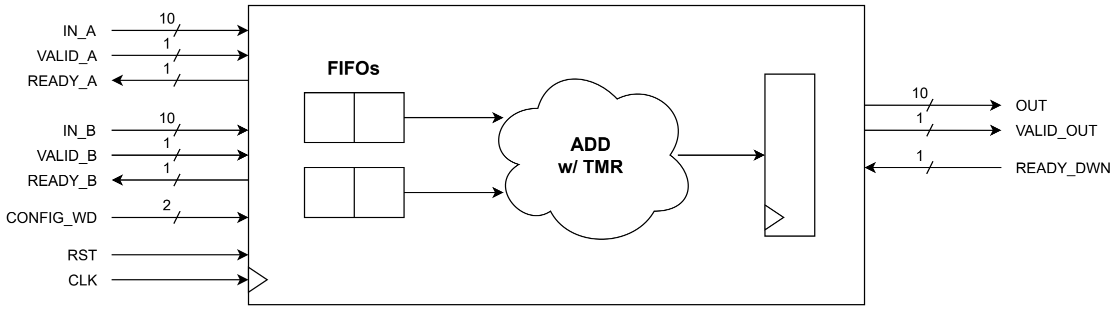

Figure 1: C2 sum FU of a CGRA

In the following sections, a more detailed explanation of the main concepts expressed above is presented.

1.2 Functional Unit of a CGRA

The *Functional Unit (FU) in the Coarse-Grained Reconfigurable Array (CGRA)* architecture plays a central role in executing operations such as signed sums in the C2 format. To fully appreciate its importance, it's essential to understand the broader context of CGRA. CGRA is a specialized hardware architecture designed for the efficient execution of complex computational tasks. It achieves this by combining configurable logic blocks, interconnected data paths, and a versatile control unit, all customized for specific applications. This adaptability makes CGRAs suitable for a wide range of computational workloads.

Among various hardware accelerators, ranging from highly efficient yet inflexible COTS (Commercial Off-The-Shelf) solutions to more versatile FPGA-based alternatives, Coarse-Grained Reconfigurable Array architectures are gaining significance. CGRAs find applications in various domains, including digital signal processing, image and video processing, and cryptography, making them even suitable for space-related applications. They consist of an array of Processing Elements (PEs), which have a complexity level between FPGA logic cells and general-purpose processors, interconnected through a Network on Chip [1].

In recent years, Coarse-Grained Reconfigurable Array (CGRA) architectures have emerged as a promising solution for accelerating compute-intensive data-flow applications. They are designed to exploit parallelism and data locality and consist of 2D structures composed of Processing Elements (PEs) connected by a Network on Chip (NoC) [1]. A simple example of a CGRA scheme with PEs connected via a NoC is depicted in Fig. 2. A 4x4 CGRA is essentially a 2D grid of PEs, and data and instruction memories. Each PE can operate on the results of its neighboring PEs.A PE is essentially an ALU with a local register file [2].

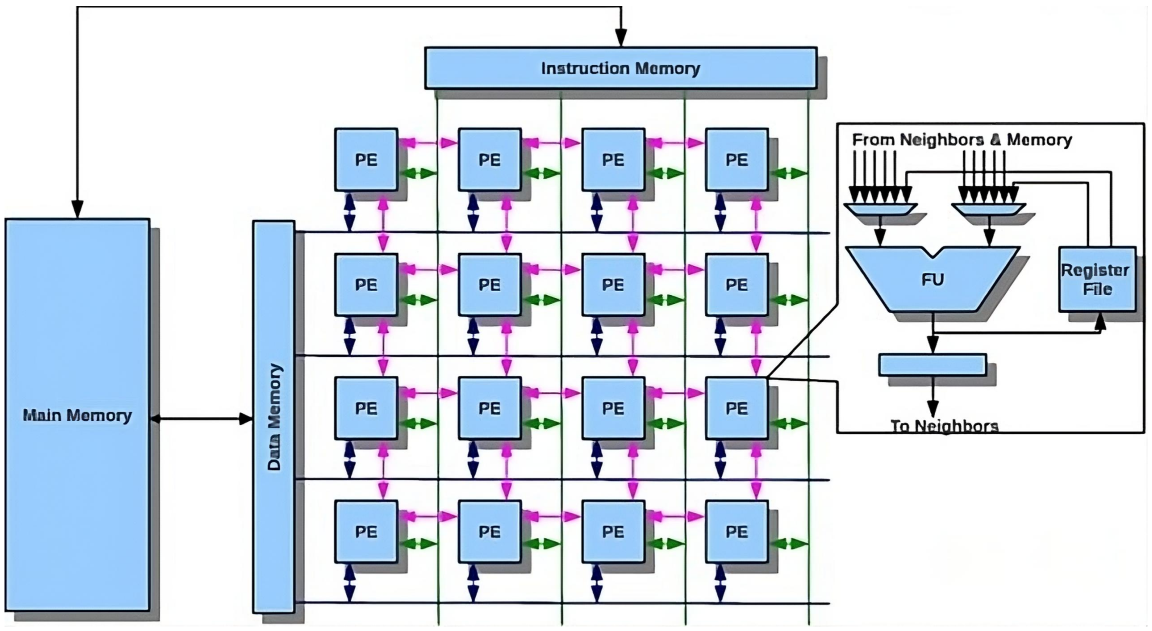

Figure 2: Simple example of a CGRA scheme with PEs connected via a NoC.

Within the CGRA context, in the present case, the Functional Unit is responsible for processing 8-bit data, coupled with 2-bit flags, which communicate important information like overflow occurrences. The FU is meticulously designed for efficient arithmetic operations. This report will delve into the specific design and functionality of the FU within the CGRA architecture, emphasizing its central role in executing signed sums in the C2 format.

1.3 Triple Modular Redundancy - TMR

Triple Modular Redundancy (TMR) is a passive HW fault tolerance technique (fault masking objective). TMR stands as a robust fault tolerance technique utilized in this case in the CGRA architecture to enhance its reliability.

TMR involves creating three independent copies of a critical component, such as a combinatorial logic block, and implementing a majority voting mechanism to determine the correct output. The motivation behind TMR is to detect and mask potential faults, ensuring that the CGRA system remains resilient to errors. The three instances of a component should fail in an independent way (it can be achieved for example by adopting different designs)

In order for the result obtained to be correct, at least two over three of the modules must deliver the correct results. In that case the effects of faults are neutralized without notification of their occurrence. This technique allows the masking of a failure in any one of the three copies at a time. For permanent faults, since the faulty module is not isolated, the fault tolerance decreases. TMR is a good choice as a mitigation technique against transient faults. In some cases two faulty modules are tolerated, such as in the case in which these involve modules of different systems and which do not interfere with each other. It is important to clarify that the voter appears to be a single point of failure and therefore its reliability determines the tolerance level of the entire module. A simple TMR schema with a generic module replicated is represented in Fig. 3 [3].

Given the same input to three independent copies of the same module, these will return three results: A, B, and C. The logic gate representation of the result of the TMR after the voter would be represented in Fig. 3 and described by the equation logic 1.

$$Z = AB + BC + AC \tag{1}$$

By triplicating critical components and comparing their outputs, TMR has a not negligible impact on delay in signal propagation, power consumption, and occupied area. This report will provide a comprehensive overview of the TMR implementation within the CGRA architecture, elucidating its role in fault tolerance and system integrity.

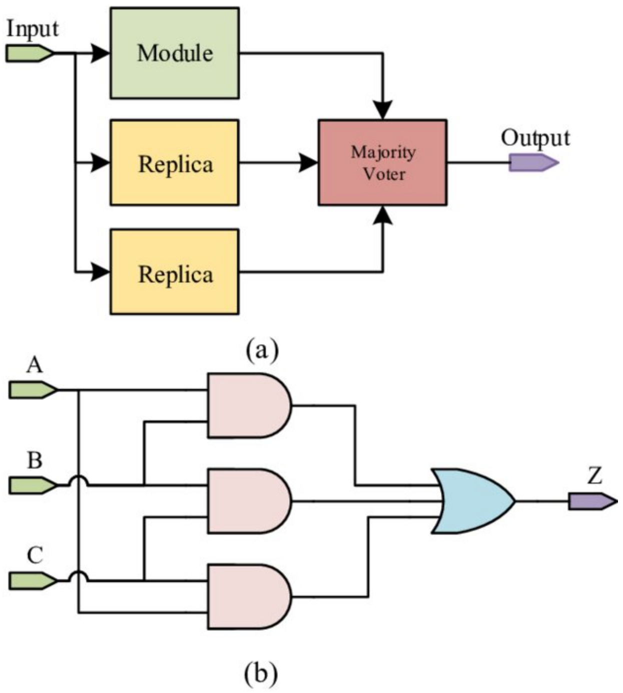

Figure 3: (a) Simple TMR implementation schema; (b) Gate level implementation of Majority Voter.

1.4 First-In-First-Out Buffer - FIFO

The *First-In-First-Out (FIFO) Buffer* represents a fundamental component of the communication protocol employed in the CGRA architecture. Its primary function is to facilitate the orderly transfer of data and flags between different parts of the system. In the context of data processing, ensuring the correct sequencing and alignment with the ready-valid communication protocol (1.5) is paramount.

The FIFO mechanism serves as a data buffer, enabling the system to manage data flow efficiently, preventing data loss, and ensuring that data is processed in the proper sequence. This report will provide a detailed exploration of the FIFO and its pivotal role in maintaining data integrity and a smooth data flow within the CGRA architecture.

1.5 Ready-Valid Handshake Protocol

The *Ready-Valid Handshake Protocol* is a cornerstone of the CGRA architecture's communication system. It establishes a robust framework for the reliable and efficient exchange of data between various components of the system. This protocol incorporates a set of signals that indicate when data is ready to be transmitted and when it is valid to be received, reducing the risk of data loss or miscommunication. Ready-valid handshake is used to decouple all the elements of the architecture

By employing this protocol, the CGRA architecture ensures that data is transferred seamlessly, enabling different components to work in harmony. This report will delve into the principles of the Ready-Valid Handshake Protocol, highlighting its significance in the overall design of the CGRA architecture and the maintenance of data integrity throughout the system.

2 Architecture Design

2.1 Preliminary Phase

The first step was to clearly identify what the interface of the component is, in order to understand how the circuit interacts with the outside world. In this case, the interface was already provided by specifications.

Subsequently, the component was divided into subcomponents by design. The problem decomposition technique allowed the identification of subcomponents (macro and micro subcomponents necessary to create the circuit) and their respective interconnections. Finally, all signals and components involved were sized correctly to meet the initial specifications.

Since nothing else is stated in the specifications, the following encoding was chosen for the 2-bit flags:

- "01": Overflow Flag Coding
- "11": Zero Flag Coding
- "10": Negative Result Flag Coding
- "00": Other Cases

In particular, an order of priority for the flags was established if two conditions overlapped, each with its own coding. In detail, the list above describes this ordering starting from the flag codification with the highest priority. Ensuring higher priority for overflow encoding is essential to avoid considering data as valid even if it comes from an overflow, without being aware of it. Since nothing else is stated in the specifications, the following encoding was chosen for the 2-bit configuration word:

- "01": The output flag will be the one associated with the first operand.
- "10": The output flag will be the one associated with the second operand.
- "11" or "00": The output flag will be the one associated with the sum operation.

Furthermore, the following choices were made during the design phase:

- The circuit is based on a single clock domain, all operations are completed in one clock cycle.

- The architecture is oriented towards ease of design rather than speed and energy/area consumption.
- The architectures presented were designed and created in order to study the impact of TMR on the architecture if it is applied on the entire component or only on some of its subcomponents

2.2 Interface

The interface of the ADD Functional Unit of a Coarse Grained Reconfigurable Array, as shown in Fig. 1, is composed by 8 inputs and 4 outputs. More in details, the inputs are:

- **IN_A**: An 8-bit integer encoded in 2's complement(bits 0 to 7) with a prefix of 2 flag bits(bits 8 and 9). This input represents the first operand of the addition operation.
- **VALID_A**: A single validity bit associated with the first operand. The data will be considered valid when its validity bit is '1'.
- **IN_B**: An 8-bit integer encoded in 2's complement(bits 0 to 7) with a prefix of 2 flag bits(bits 8 and 9). This input represents the second operand of the addition operation.
- **VALID_B**: A single validity bit associated with the second operand. The data will be considered valid when its validity bit is '1'.
- **CONFIG_WD**: 2-bit input used as a configuration word to specify which flag to present at the output.
- **READY_DOWN**: A single "ready" bit associated with the downstream component that is to receive the results of the addition operation. This component will be considered ready when the related "ready" bit is '1'.
- **RST**: High active reset signal, used to initialize all the internal components of the circuit in order to reset any spurious values, e.g. when the circuit is powered on.
- **CLK**: Clock signal used by internal synchronous components, such as memory elements. At each clock cycle, new data is presented as input and the output is updated according to the temporal evolution of the subcomponents.

Regarding the outputs:

- **READY_A**: A single "ready" bit associated with the upstream component that must send the first operand of the addition operation. The component under consideration in this report can be considered ready when its "ready" bit is '1'. Under this condition, the upstream component associated with this signal output can send a new first operand to the component under consideration.

- **READY_B**: A single "ready" bit associated with the upstream component that must send the second operand of the addition operation. The component under consideration in this report can be considered ready when its "ready" bit is '1'. Under this condition, the upstream component associated with this signal output can send a new second operand to the component under consideration.

- **OUT**: An 8-bit integer encoded in 2's complement(bits 0 to 7) with a prefix of 2 flag bits(bits 8 and 9). This output represents the result of the addition operation.

- **VALID_OUT**: A single validity bit associated with the addition result. The data will be considered valid when its validity bit is '1'.

2.3 Components

The architecture of the *ADD Functional Unit of a Coarse Grained Reconfigurable Array* was defined following a hierarchical approach, in such a way as to obtain the final circuit as a composition of several components, each dedicated to a specific function. In a similar manner, each component was designed following the same approach, defining sub-components. The procedure was iterated until sub-components composed only of basic logic gates were obtained. The ADD FU of a CGRA is mainly composed of the following elements:

- **D-Edge Triggered Flip-Flop With Enable**
- **FIFO Buffer**
- **Full Adder**
- **Ripple Carry Adder**
- **Flag Generator**
- **Flag Selector**

Each of these elements will be examined in detail in the following sections. It is important to underline that the following diagrams, especially those relating to the component interfaces, follow the nomenclature used subsequently for the VHDL description. In schemes in which the internal composition of components is analysed, an ad-hoc and different nomenclature is often used in favor of greater interpretability of the signals. However, the match with the VHDL is immediate. The clock and reset signals can be omitted for a more easily readable diagram.

2.3.1 D-Edge Triggered Flip-Flop With Enable

The *DFF with enable* component is used as a standalone (for example for output signals) or as a basic component for the creation of other components (such as the FIFO Buffer). Its design in the basic version does not require dedicated and in-depth study. These are basic circuits that can be easily found in the literature. These components are present in different sizes. For this circuit two different architectures are presented: with and without TMR. The version with TMR has the structure indicated in Fig. 3 where the replicated module is that of a simple positive edge triggered D flip-flop with enable and a low active reset signal (i.e. the basic module without TMR). The interface of this component is composed of:

- **d**: input signal representing the input data to be memorized.
- **en**: one-bit input signal to select the needed behavior: hold('0') or save('1').
- **clk**: clock signal, necessary for the memory elements.
- **async_rst_n**: active low reset signal, used to initialize the internal memory elements to zero.
- **q**: output signal representing the memorised data.

2.3.2 FIFO Buffer

As already mentioned in the paragraph 1.4, the *FIFO* is introduced to allow the non-simultaneous entry of operands for the addition operation. In particular, each stage of the FIFO will contain validity information, flags and data of an operand given input at a time t. The interface of this component is composed of:

- **valid_in**: one-bit input signal that represents operand validity bit.

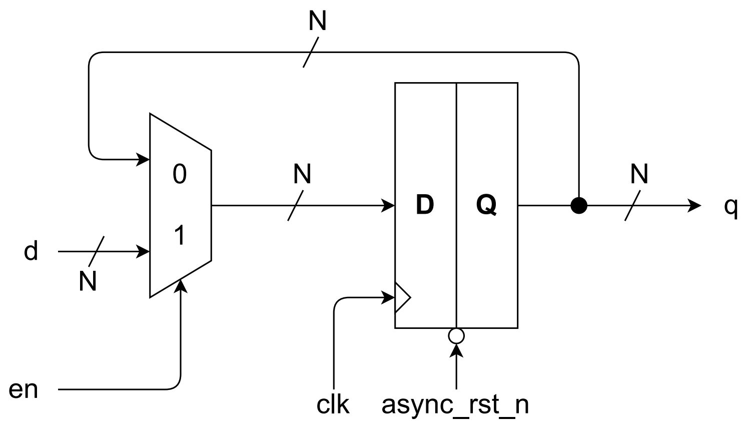

Figure 4: Simple DFF with enable schema

- **data_in**: input signal representing the operand on 10 bits, of which the two most significant are flags and the others are data
- **ready_downstream**: input signal representing the 'ready' state of the downstream device that must receive the sum results.
- **sync_final_state**: one-bit input signal which represents the validity of the data in the final stage of a possible parallel FIFO in order to synchronize the output of the operands to the RCA.
- **clk**: clock signal, necessary for the internal memory elements.
- **async_rst_n**: active low reset signal, used to initialize the output of the circuit and internal memory elements to zero.
- **valid_out**: one-bit output signal representing the validity bit of the operand contained in the final stage of the FIFO (operand passed to the RCA).
- **data_out**: output signal representing the operand passed to the RCA on 10 bits, of which the two most significant are flags and the others are data
- **ready_upstream**: output signal representing the "ready" state of the FIFO to receive and store new operands.

The use of this component for the management of incoming operands also allows operands to be stored in each FIFO for a number equal to its depth and speeds up subsequent addition operations by having amortized interactions with upstream

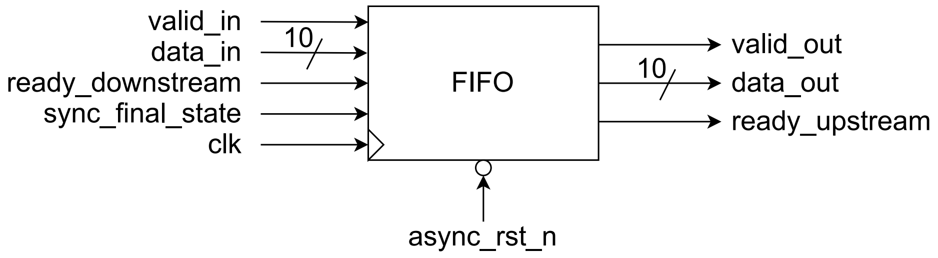

Figure 5: FIFO interface schema

devices. New operands are presented as input every clock cycle if the output-ready signal is active (of value '1'). Each stage of the FIFO is created through a DFF with enable which allows you to maintain the current data or store new data, leading to the shift of the contents of the FIFO. To drive these enablers, Karnaugh maps were created from which the associated logical equations were extracted (refer to Fig. 6).

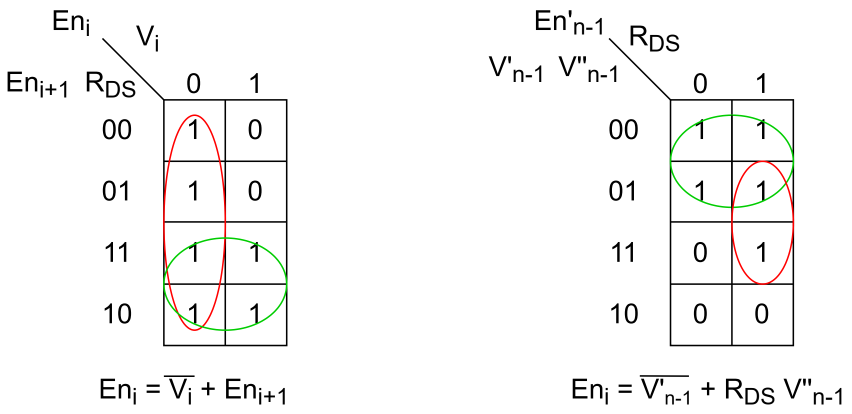

Figure 6: Karnaugh maps for FIFO enable signals

Each stage of the FIFO, except the last, stores new data only if the downstream stage adjacent to it in turn saves new data (data shift) or if the downstream adjacent stage keeps the current data but the current stage has invalid data. This management allows you to eliminate bubbles (invalid data) contained within the FIFO (if this is currently blocked and does not provide new data downstream) and allows you to avoid duplication of operands (an i-th stage maintains its value while the i+1th saves a new data). However, the last of the FIFO stages requires a different enable driving rule. In particular, this stage stores new data only if the current one is invalid or if the downstream device is ready and the possible FIFO of a second operand has valid data in its last stage (refer to Fig. 7). The ready signal returned at the upstream output corresponds to the enable signal of the first stage (if it is possible to write new values then it means that it is also possible to provide new ones). There is both a version with TMR and without TMR of this component. The version with TMR is created by applying one of the simplest techniques, that is, applying the TMR to each of the memory stages (DFF with enable with TMR referred to in paragraph 2.3.1). The enable signals are sent to the various stages with a simple combinatorial network. This could cause the creation of a large path, potentially a critical path. An implementation using registers to interrupt this path would be more adequate to improve performance but would increase the complexity of the circuit. Taking into account the objectives of the project (preferring simplicity over performance), the basic version of the enablers without registers was created.

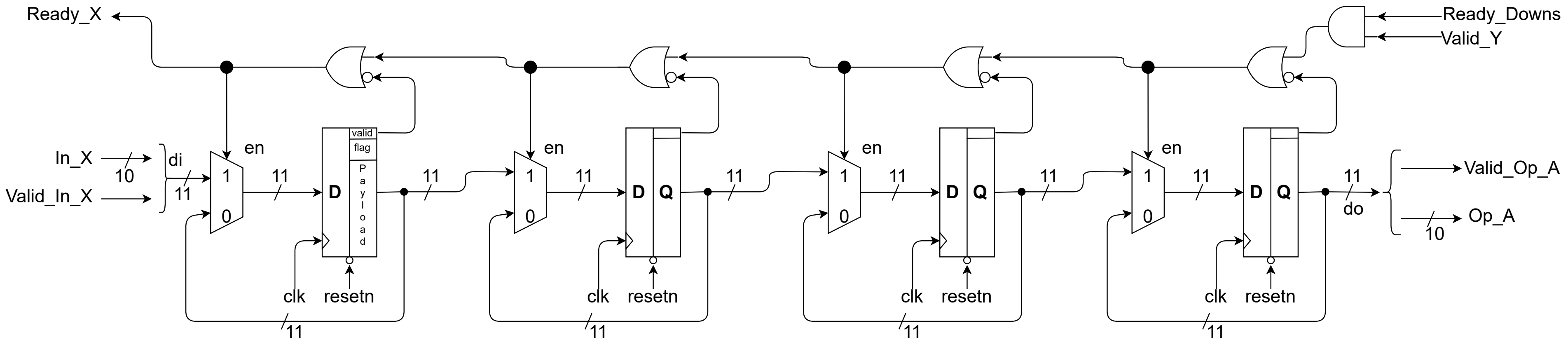

Figure 7: Example scheme for a FIFO of depth 4 with enable signal generation

2.3.3 Full Adder

The *Full Adder (FA)* is a fundamental digital circuit that plays a crucial role in arithmetic operations. It is designed to sum three one-bit inputs: A, B, and an incoming carry (Cin), producing two one-bit outputs: the sum (S) and an outgoing carry (Cout). The Full Adder logic combines the input bits and any incoming carries to determine the binary sum. It is an essential element for more complex arithmetic operations. It is used in the context presented for the creation of a Ripple Carry Adder which performs the sum between operands of 8 bits each, obtained by a Full Adder sequence chain. The Full Adder is a well-known component in the literature. The interface of this basic component is omitted as it can be deduced from Fig. 9.

2.3.4 Ripple Carry Adder

The *Ripple Carry Adder (RCA)* is an essential component in binary arithmetic within digital circuits. This is a type of binary adder that can be used to add two fixed-length binary numbers. Its operation is based on a sequential chain of Full Adders. The RCA interface (Fig. 8 is as follows:

- **A**: input signal that represents the first operand to be added.
- **B**: input signal that represents the second operand to be added.
- **Cin**: input signal that represents the input carry to be added.
- **Sum**: output signal that represents the output addition result.
- **Of**: output signal that represents overflow occurrence in addition operation.

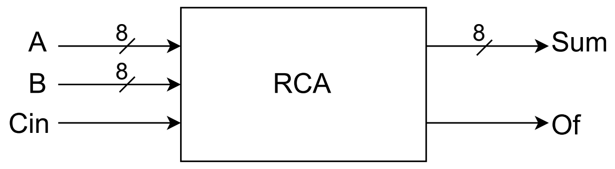

Figure 8: RCA interface schema

In an RCA, the bits of each input are added one at a time, going from one Full Adder to the next along a chain. The least significant bit (LSB) is added first, and if a carry occurs, it is passed to the next Full Adder in the chain. This process continues until all the bits have been added together.

The final result of the addition is obtained from the chain of Full Adders, with the most significant carry-out (Carry-Out) representing the final carry. However, they are limited by latency due to carry propagation across the chain, which can affect performance in high-speed applications. In this project, a basic RCA is used, without applying any type of optimization for the carry chain. The RCA of interest in the project produces at the output, in addition to the sum, also an overflow bit (since integers are added) obtained as an XOR of the last two carries (Fig. 9).

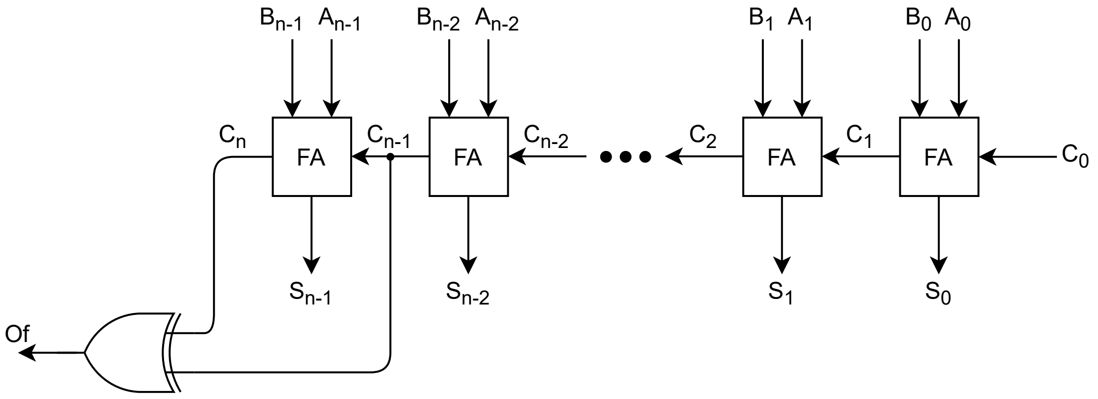

Figure 9: RCA with overflow signal generation

There are two versions of this component, with TMR and without TMR. The version without TMR is shown as an example in Fig. 9. The version with TMR is instead obtained by applying the Majority Voter to the individual bits of three different RCAs, i.e. applying it on the FA outputs in the same position in the three RCA instances (Fig. 10)[4].

2.3.5 Flag Generator

The *Flag Generator* is a digital circuit component responsible for generating encoded flags based on the output of a Ripple Carry Adder (RCA). These encoded flags provide information on the result of the addition operation following the instructions given at the beginning of the chapter. The interface of this component (Fig. 11) is as follows:

- **Sum_res**: input signal that represents the addition result from RCA.
- **Sum_of**: one-bit input signal that represents the overflow flag for addition result from RCA

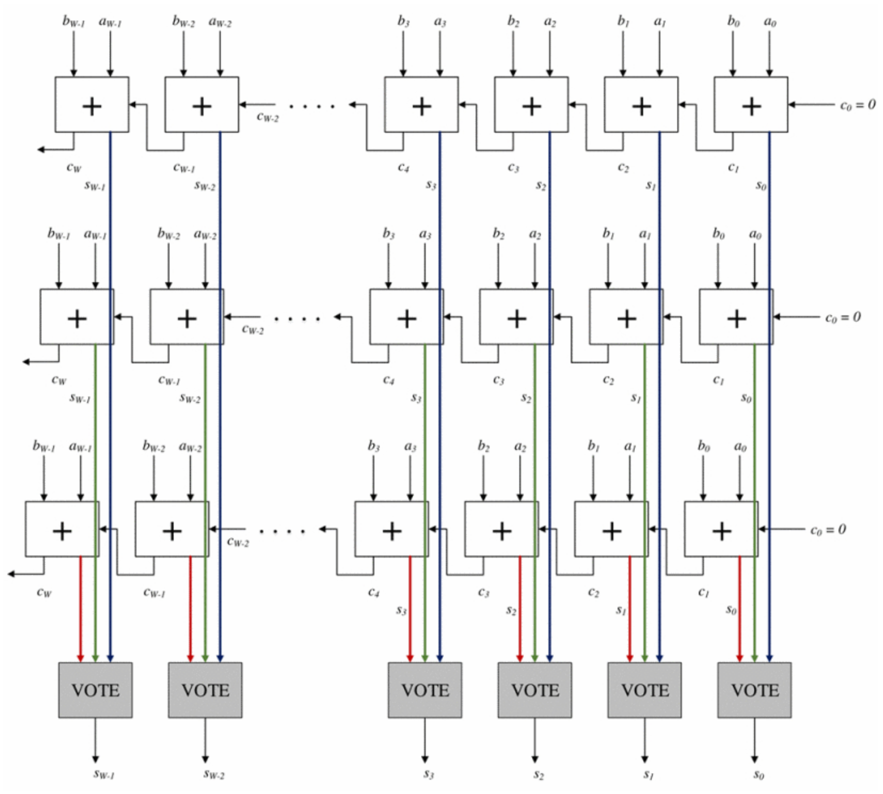

Figure 10: TMR applied to RCA

- **Flag_res**: two-bit encoded flag output signal that represents the occurrence of a particular condition on addition result

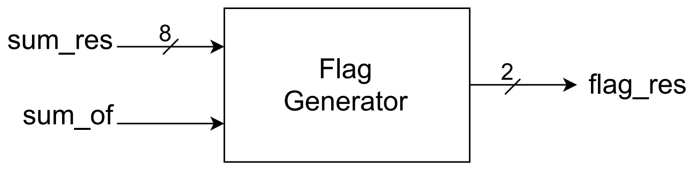

Figure 11: Flag Generator interface schema

The Flag Generator component is crucial in digital systems for signaling important conditions during arithmetic operations, such as overflow, zero results, and negative results. It encodes these conditions into a compact two-bit flag format for further processing or decision-making in the system. A simple schema of the internal composition for the Flag Generator is proposed in Fig. 12.

There are two versions of this component: with and without TMR. The version without TMR is represented in Fig. 12. The version with TMR applies a simple triplication of the simple module with the final Majority Voter to compare the results.

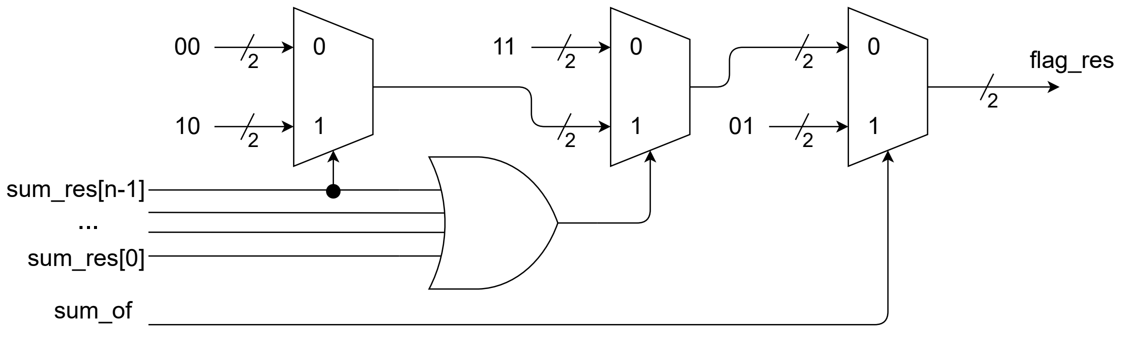

Figure 12: Flag Generator design schema

2.3.6 Flag Selector

The *Flag Selector* is a digital circuit component responsible for selecting encoded flags based on the configuration word signal received as input from the upstream device. The encoded flag outputted from this component provides information on the result of the addition operation or on one of the addition operands following the instructions given at the beginning of the chapter. The interface of this component (Fig. 13) is as follows:

- **flag_res_rca**: two-bit input signal representing the flag coming from RCA, and more precisely from Flag Generator.
- **flag_operand_A**: two-bit input signal representing the flag associated with the first addition operand.
- **flag_operand_B**: two-bit input signal representing the flag associated with the second addition operand.
- **conf_wd**: two-bit input signal representing the configuration word to identify the flag to return as output.
- **flag_res**: two-bit output signal representing the selected flag to return as output.

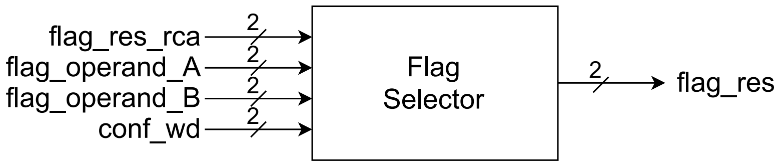

Figure 13: Flag Selector interface schema

The Flag Selector component is crucial for passing downstream the proper flag value. A simple schema of the internal composition for the Flag Selector is proposed in Fig. 14.

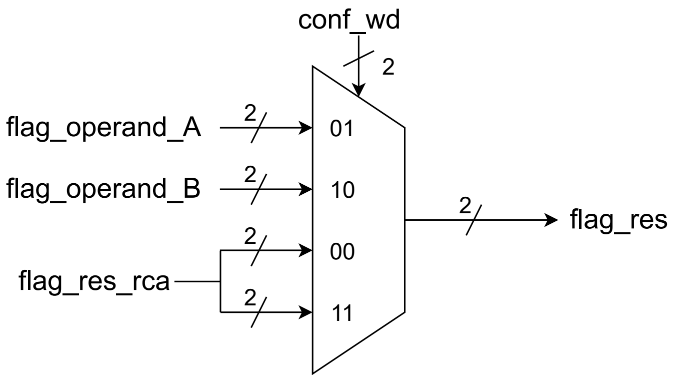

Figure 14: Flag Selector design schema

There are two versions of this component: with and without TMR. The version without TMR is represented in Fig. 14. The version with TMR applies a simple triplication of the simple module with the final Majority Voter to compare the results.

2.3.7 Interconnections

Once all the components have been designed, they are interconnected according to the diagram shown in Fig. 15, obtaining the ADD FU for CGRA. In the schematic diagram Fig. 15, it is possible to see the paths created between input and output. The clock and reset signals are omitted for reasons of simplicity and greater clarity of the diagram represented.

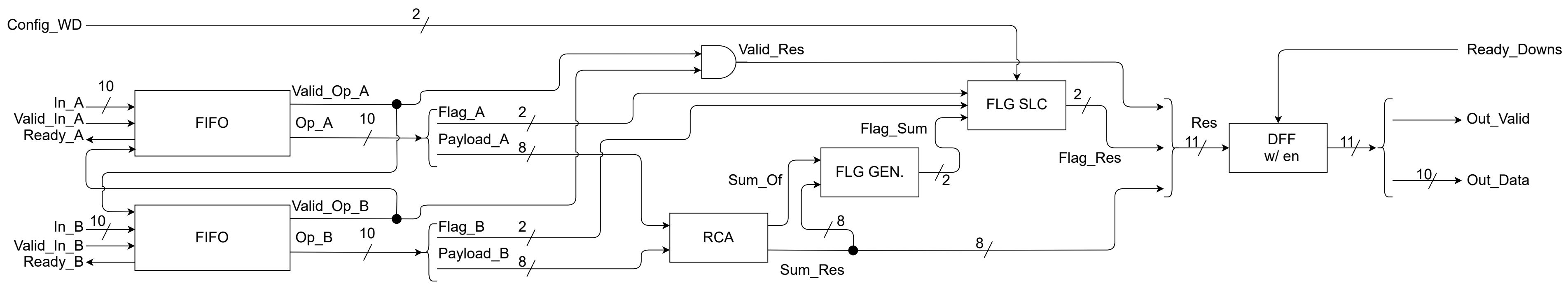

Figure 15: High-level diagram of the various interconnected components for the creation of the ADD FU of the CGRA.

In the creation of the final circuit, four different architectures were created in order to compare them according to different metrics. The following versions of the final circuit were created:

- Version without TMR: Each component is found in its basic version without TMR. This version is created to have a point of comparison with the simplest version of the circuit.
- Version with TMR applied to the combinatorial network containing the RCA, the Flag generator and the Flag selector: This version is created to evaluate the impact of applying the TMR to only the part of the circuit that is responsible for actually implementing the summing functionality.
- Version with TMR applied to the memory elements: This version is created to evaluate the impact of applying the TMR to only the part of the circuit that actually implements the storage functionality (FIFO and registers).
- Version with TMR applied to the memory elements and to the combinatorial network of RCA, Flag generator and Flag selector: This version is created to evaluate the impact that the application of the TMR has both to the memory elements and to the main combinatorial network. This version of the ADD FU is the most reliable in terms of faults (stuck at, etc) compared to the three previous versions in which the mitigation technique was applied only in selected parts of the circuit.

The four versions will be compared with each other in terms of:

- **Power consumption**
- **Utilization area**
- **Timing**

3 Hardware Description and Testing

3.1 Code organization

The digital circuit has been fully described using VHDL language, adopting structural, behavioral and data flow approaches. Rather than producing a description aimed at optimization in terms of area consumption, performance, and energy consumption, the followed approach aims to facilitate the reuse of the code and the obtaining of a clear and simple description, which reflects as much as possible the structures seen so far. In particular, the structural approach was adopted to reflect the design style followed up to now, The behavioral approach was limited as much as possible to keep the description of the circuit at the lowest possible level. In components where the manipulation was sufficiently clear and simple, a data-flow approach was followed. The source code has been organized within a dedicated directory that follows the component structure. The description of the final circuit, in each of the four versions created, is found in a dedicated folder. The components that make up the circuit have been divided into two different folders depending on the version described (with or without TMR). A further folder has been dedicated to basic components (Full Adder). The source directory also has a wrapper subdirectory, containing wrappers useful during the synthesis phase. Inside this folder, there is another subfolder containing the description of components useful only for the creation of the wrapped version of the final circuit. All the testbench codes are found in a dedicated directory within which the same division just described is reflected and each name has a "_tb" suffix.

The VHDL code describes the components quite closely as designed and analyzed so far, so it shouldn't take much effort to understand. However, in order to further facilitate the reading and understanding of the source code, some conventions explained here are adopted:

- Component interface signals begin with a prefix representative of the component to which they belong, e.g. the interface signals for the Flag Selector are in the form flg_slc_*, while those for the FIFO are in the form fifo_*. The same naming policy is also maintained for naming constants in the generic section.
- In the description of the ADD FU for CGRA, all the internal signals used to interconnect the macro-components indicate in the name the source component and, if unique, the recipient component.
- The description of all components provides a generic section that can be used to instantiate the component by customizing some of the component's charac-

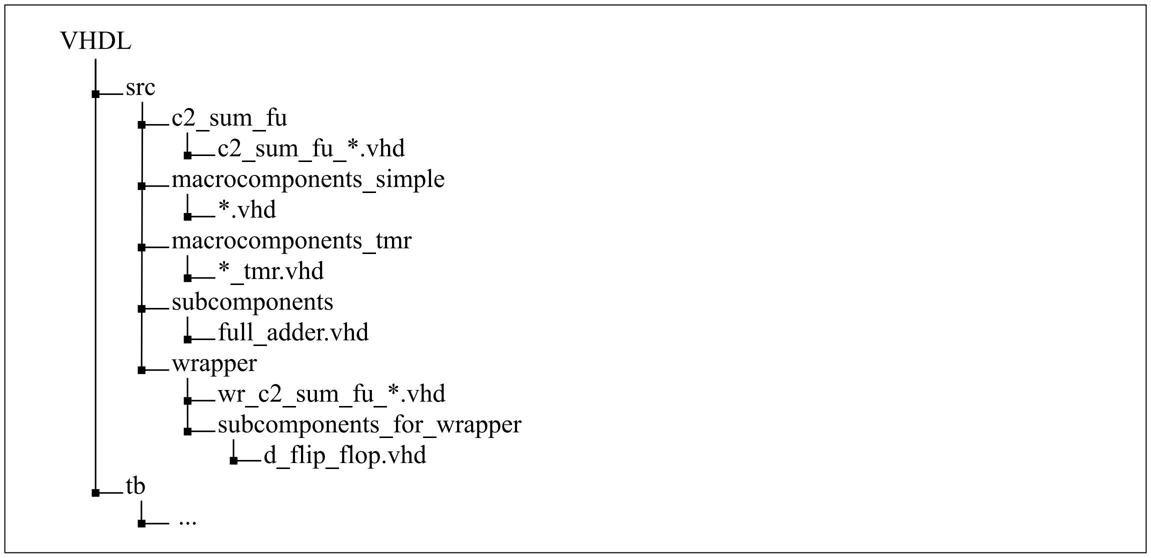

Figure 16: VHDL code organization

teristics (e.g. the number of bits in the case of memory elements, FIFO depth, etc.). This allows for the flexible reuse of components in other component descriptions. If the instance does not specify a generic map, the default value will be used. The value of the number of redundant modules is fixed at three (intrinsic in the definition of TMR).

- The VHDL components descriptions and testbenches are well commented and have a well-structured internal organization of the comments divided into sections. The structure is maintained in each of the documents.

3.2 Testing phase

The components were described following a bottom-up approach, so that each component could be tested during the development of its description, catching any errors as soon as possible. For each component, a test-bench has been realized, in order to test the expected behavior through simulation. The test-bench file, the one that determines the evolution of the simulation, consists of an instance of the device under test (DUT), and some external signals connected to the interface of the DUT to determine the inputs and verify the output produced. For each component the test, although may not be exhaustive, consists in:

- Checks the reset signal, if present.

- Checks the output in the presence of "normal" input values, e.g. checks the result of the sum of two operands.
- Checks the output in the presence of edge input value that may cause overflow or other results that need a special flag codification, e.g. add one to the maximum representable number or add a negative number to zero.
- Checks the behavior of the circuit in the case of a blocked FIFO (no sliding and elimination of bubbles) and a heterogeneous interaction via ready-valid protocol downstream and upstream.
- Checks the behavior of the circuit components in the case of faults in which a signal temporarily or permanently assumes a fixed value and which may not correspond to the exact one (the TMR should mitigate this situation). These types of faults were tested using Modelsim features accessible via a graphical interface.

Due to the simplicity of the implemented functionalities, the results were manually validated by observing the wave drawn by the simulator.

The following sections of the report show the testbenches for the macrocomponents that make up the final circuit. For each of them, the version with TMR is reported so as to highlight, in addition to the actual correct functioning, the effectiveness of the mitigation technique applied. The testbench for the basic components (i.e. Full Adder and one-bit DFF with enable for wrappers) are not reported as they are trivial and of little interest. Finally, the testbench of one of the versions of the final circuit is reported. All four are not reported as the testbenches are identical to each other and produce the same result. In fact, the application of the TMR does not affect the final result but only masks a possible internal fault. The tests on the TMR are not repeated on the final circuit as they have already been tested on the individual components and therefore their operation has already been proven. The purpose of the testbench on the final circuit is to demonstrate the correct operation of the same, with correct interaction between each of the macrocomponents.

3.2.1 Ripple Carry Adder with TMR - Testbench

The testbench reported below demonstrates the correct functioning of the RCA with TMR even in the presence of faults. In this case, module number 2 of the TMR suffers from a stuck-at fault and therefore assumes a fixed value ("-1" in this case) for the duration of the simulation. Despite this, under the assumption that the other two modules are not corrupted, the final result of the RCA with TMR is not affected by this fault, which is therefore masked. From the RCA the following are also tested:

- The correctness of the sum in vanilla cases.
- The correctness of the range of representation of the result.
- The correctness in generating the overflow bit in both directions (i.e. subtracting 1 from the minimum representable integer or adding 1 to the maximum representable integer).

More details can be extrapolated by consulting the code, which is provided with exhaustive comments. In the simulation, the data relating to the operand and result payload were represented in decimal form for ease of reading the results. To do this it was necessary to group them differently from what is reported in the code but the information is easily traceable.

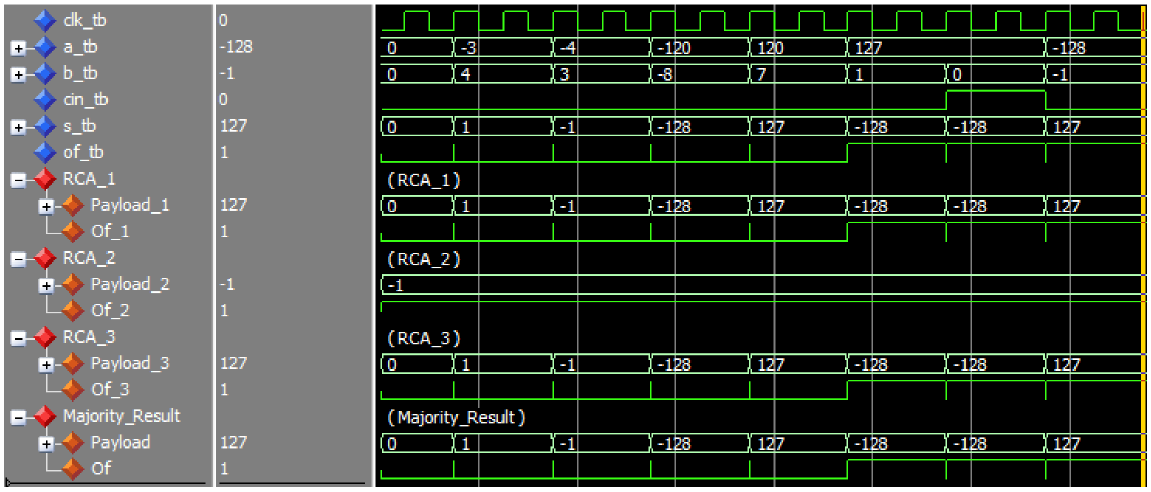

Figure 17: RCA testbench with a TMR's module stuck at a fixed wrong value. The fault is masked by TMR mitigation technique.

3.2.2 N-bit DFF with enable with TMR - Testbench

The testbench reported below demonstrates the correct functioning of the DFF with enable and with TMR even in the presence of faults. In this case, the second module of the TMR suffers from a stuck-at fault, therefore it assumes a fixed value ("00" in this case) for the duration of the simulation. Despite this, under the assumption that the other two modules are not corrupt, the final result of the DFF with enable and with TMR is not affected by this fault, which is therefore masked. The following are also tested from the DFF:

- The correctness of its behavior (it stores new data only with enable value "1" and otherwise maintains the old value).
- The correctness of the reset (which brings everything back to an initial condition with value "0" and writing disabled, i.e. enable with value "0").

More details can be extrapolated by consulting the code, which is provided with exhaustive comments. In the simulation, the data relating to the operand and result payload were represented in hexadecimal form for ease of reading the results.

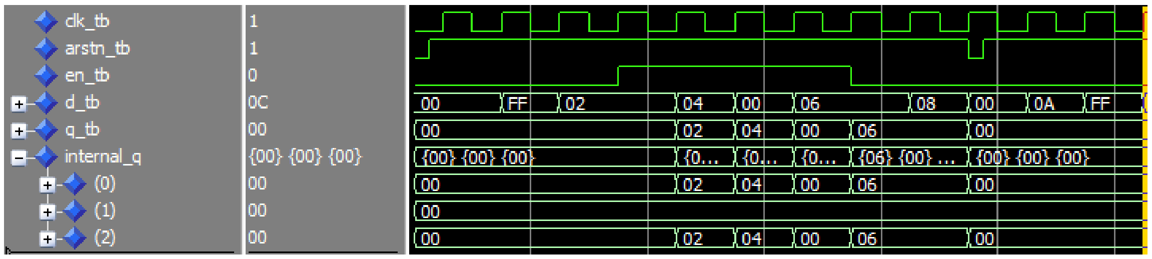

Figure 18: N-Bits DFF with enable testbench with a TMR's module stuck at a fixed wrong value. The fault is masked by TMR mitigation technique.

3.2.3 FIFO with TMR - Testbench

The testbench reported below demonstrates the correct functioning of the FIFO with TMR even in the presence of faults. In this case, the FIFO presents a stuck-at fault in its first stage in the third TMR module. This faulty stage then takes on a fixed value ("-1", with validity "1" and with flag "11" in this case) for the duration of the simulation. Despite this, under the assumption that the other two modules of the same stage are not corrupted, the final result of the FIFO with TMR is not affected by this fault, which is therefore masked. Due to the way the FIFO was designed, it can resist and mask up to a fault of a module (which makes up the TMR) for each stage. Of the FIFO the following are also tested:

- The correctness of its behavior (storage of new data, the correctness of the ready-valid handshake protocol, the correctness of the management of the enable signals, the synchronization with a second FIFO, avoiding data duplication).

- The ability to block all or part of the information contained in the FIFO without proceeding with further data advancement.
- The ability to store new data even when the downstream device is not ready, only in the presence of bubbles (invalid data) in the FIFO, so as to eliminate these values and fill the FIFO with only valid data.
- The correctness of the reset (which brings everything back to an initial condition with value "0" and enable at "1" as the data in the FIFO is invalid and therefore overwritable).

From the simulation reported it is possible to clearly observe these behaviors, also being able to observe the inside of the FIFO and therefore the state of each of its stages. In order to simplify the reading of the simulation data depicted here, the signals representing the operand payloads are represented in decimal form. As for the four enable bits, these are encoded as a single hexadecimal digit. Furthermore, in order to represent the information in a clear and structured form, it was necessary to group the signals differently compared to what is reported in the code but the information is easily traceable. More details can be extrapolated by consulting the code, which is provided with exhaustive comments.

3.2.4 Flag Generator with TMR - Testbench

The testbench reported below demonstrates the correct functioning of the Flag Generator with TMR even in the presence of faults. In this case the Flag Generator has a stuck-at fault in the second module that makes up the TMR. This faulty module then takes on a fixed value ("10", or "Negative Flag", in this case) for the duration of the simulation. Despite this, under the assumption that the other two modules are not corrupted, the final result of the Flag Generator with TMR is not affected by this fault, which is therefore masked. The correctness of its behavior is tested for the Flag Generator, i.e. the ability to generate all four flag codings respecting the priority order that has been established if two conditions overlap, each with its own coding (refer to 2.1 for the order of priority). More details can be extrapolated by consulting the code, which is provided with exhaustive comments. It is important to underline that the base Flag Generator is described within a VHDL 2008 file.

3.2.5 Flag Selector with TMR - Testbench

The testbench reported below demonstrates the correct functioning of the Flag Selector with TMR even in the presence of faults. In this case the Flag Selector has

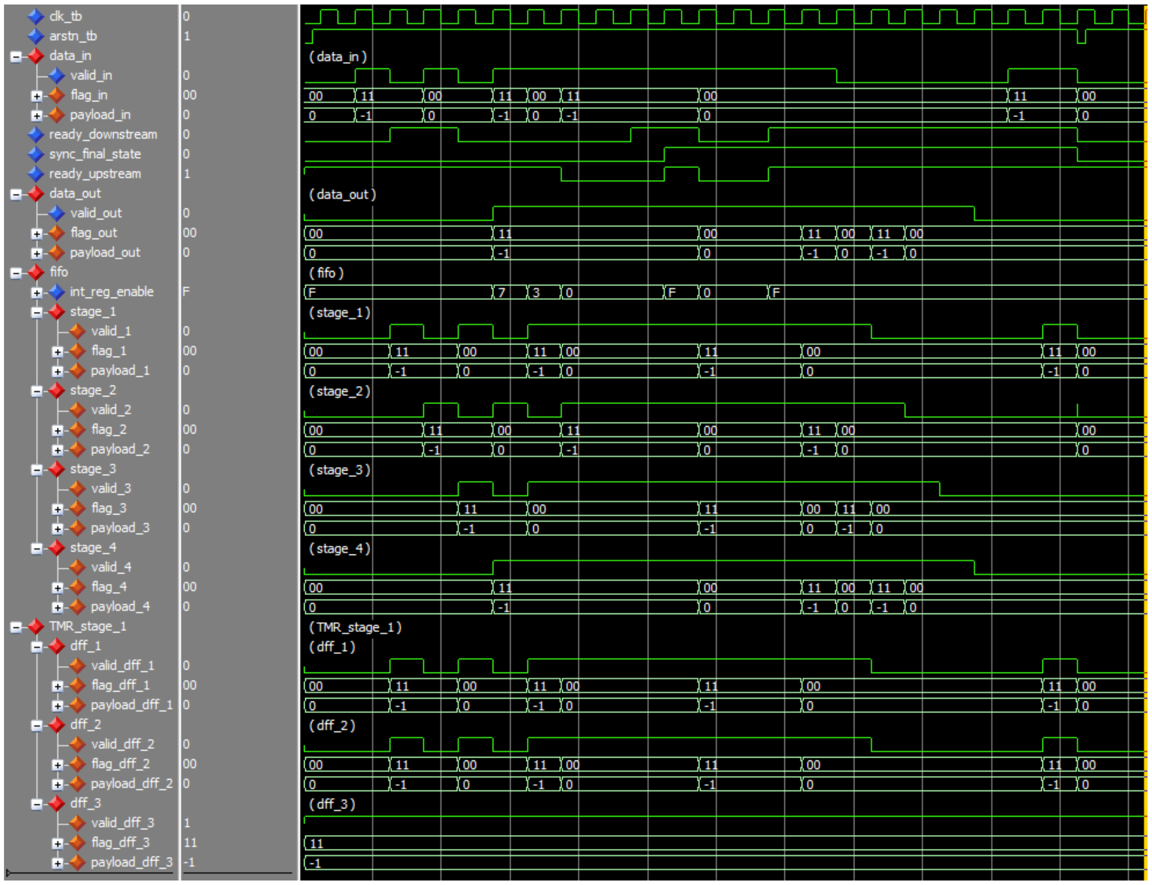

Figure 19: FIFO testbench with a TMR's module stuck at a fixed wrong value. The fault is masked by TMR mitigation technique.

a stuck-at fault in the second module that makes up the TMR. This faulty module then takes on a fixed value ("11", or "Zero Flag", in this case) for the duration of the simulation. Despite this, under the assumption that the other two modules are not corrupted, the final result of the Flag Selector with TMR is not affected by this fault, which is therefore masked. The correctness of its behavior is tested for the Flag Selector, i.e. the ability to select each of the three flag information delivered to the component (the two flags associated with the operands and the one produced by the Flag Generator on the basis of the sum operation performed on the same operands) depending on the value of the configuration word input to the component (refer to 2.1 for the coding of the configuration word and its associated meaning). More details can be extrapolated by consulting the code, which is provided with exhaustive comments.

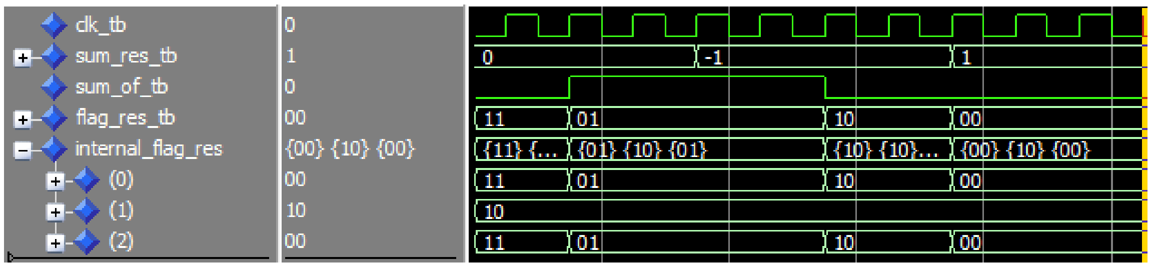

Figure 20: Flag Generator testbench with a TMR's module stuck at a fixed wrong value. The fault is masked by TMR mitigation technique.

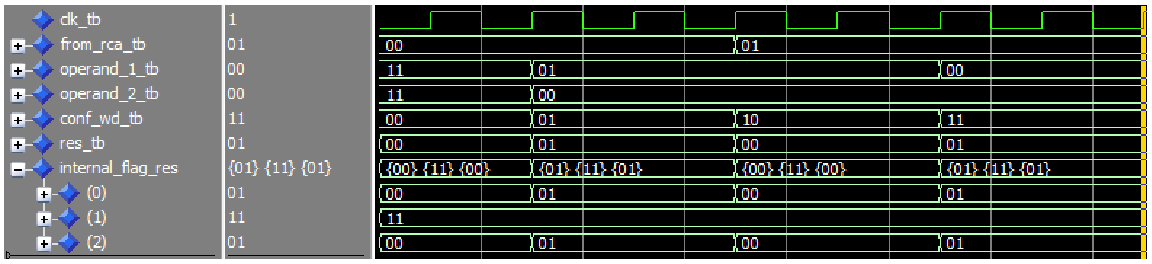

Figure 21: Flag Selector testbench with a TMR's module stuck at a fixed wrong value. The fault is masked by TMR mitigation technique.

3.2.6 ADD FU of the CGRA - Testbench

The testbench below demonstrates the correct operation of the final ADD FU circuit for a CGRA. Having demonstrated the correct functioning of the individual modules to which the TMR can be applied individually, the current focus is to demonstrate the correct interaction of the modules with each other in order to achieve the required sum functionality. In fact, each component presents equivalent interfaces in the versions with and without the TMR and there are no different results when running the testbench (not even delays in the state of the component, given the parallelism with which the TMR works). By demonstrating the theoretical operation of the circuit in the basic version, correct operation can be demonstrated even in the presence of TMR and faults, which will be masked by the individual modules if at least two of the three TMR modules are functioning. The ADD FU for the CGRA manages to mask up to one faulty module for each component to which the TMR is applied, except for the FIFO for which it can tolerate up to one fault for each stage in case of TMR. Of the ADD FU for CGRA is tested:

- The correctness of its behavior, i.e. the ability to calculate the sum of two operands stored within as many FIFOs. These operands can also be supplied to the circuit at different times.

- The correctness of the management of the change of the configuration word for the selection of the final flag.
- The correctness of the circuit behavior in the event of a reset.
- The correct management of interactions with downstream and upstream devices

In order to simplify the reading of the simulation data depicted here, the signals representing the operand payloads are represented in decimal form. Furthermore, in order to represent the information in a clear and structured form, it was necessary to group the signals differently compared to what is reported in the code but the information is easily traceable. More details can be extrapolated by consulting the code, which is provided with exhaustive comments.

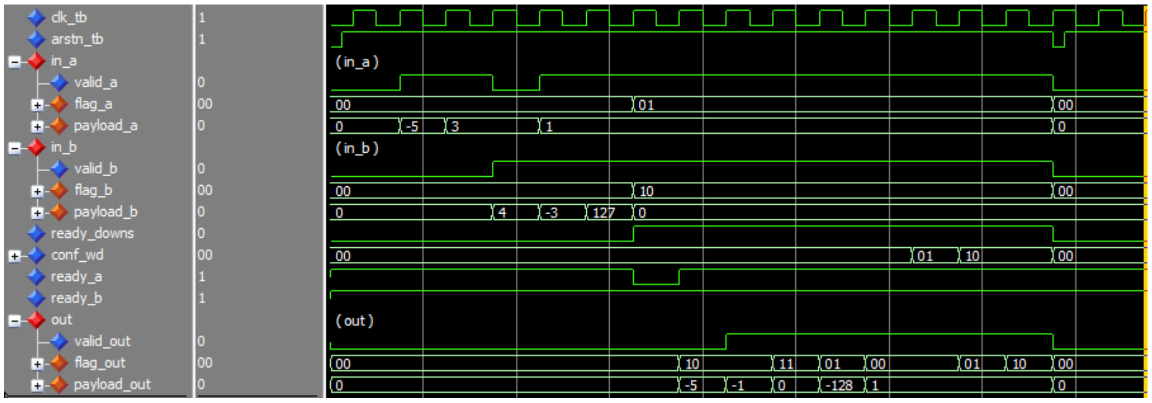

Figure 22: ADD FU of a CGRA testbench

4 Synthesis and Implementation

The final step is to synthesize and implement the VHDL hardware description to evaluate the performance of the designed circuit. The performance is evaluated in terms of clock speed, area utilization, and power consumption, referring to the FPGA identified by the code "xc7z010clg400-1". The FPGA provides many resources, but only the basic ones relevant to ADD FU of a CGRA evaluation are listed below:

- 4,400 logic slices each composed by:
  - 4 6-input LUT
  - 8 flip-flops

The VHDL descriptions provided for the four different architectures represent the same functionality but with different fault tolerant levels. It is interesting to compare the different architectures from the point of view of the metrics indicated above (time, area and power consumption). The architectures all have the same basic scheme but, if present, the modules affected by TMR are replaced with the TMR version of the module itself. Below are the four versions affected by the comparison of the results.

- **ADD FU Simple**: ADD FU for a CGRA with no TMR applied. Lowest fault tolerance level.
- **ADD FU Comb. TMR**: ADD FU for a CGRA with TMR applied only on the combinatorial part (RCA, Flag Generator and Flag Selector).
- **ADD FU Reg. TMR**: ADD FU for a CGRA with TMR applied only on registers and FIFO.
- **ADD FU Full TMR**: ADD FU for a CGRA with TMR applied on both combinatorial part and registers/FIFO components.

All versions were synthesized and implemented within a wrapper, i.e. a register barrier connected to the inputs and output of the circuit. By doing this you have an evaluation of every possible register-logic-register path of the component, in order to get a more precise timing evaluation. The synthesis and implementation tool reported the following warnings after the implementation phase, which must be considered when assessing the confidence of the results to be reported later:

- **Pin Planning**: In order to obtain an out-of-context evaluation, use was made of the out-of-context synthesis mode so as not to be affected by delays due to the pins. This way you don't need to do pin planning. The generation of bitstream and programming is not taken into account, so these warnings have been ignored, even though it may cause a poor evaluation of the implemented design. Warnings pop up to inform the user that, due to the out-of-context mode, some constraints and properties are not set. This reduces the accuracy in the time estimation.

  - *The property HD.CLK_SRC of clock port "clk" is not set. In out-of-context mode, this prevents timing estimation for clock delay/skew*
  - *Clock port "clk" does not have an associated HD.CLK_SRC. Without this constraint, timing analysis may not be accurate and upstream checks cannot be done to ensure correct clock placement.*
  - (For each input port) *Port α does not have an associated HD.β, which will prevent the partial routing of the signal α. Without this partial route, timing analysis to/from this port will not be accurate, and no routing information for this port can be exported.*

- **PS7**: The implementation warns the user to about the PS7 cell, i.e. a component on the Zynq-7000 platform that refers to the ARM core processing system available on the board (other than the FPGA). This warning should be better inspected before going on with the device programming, but as already pointed out this step is not covered and furthermore seems to be platform-specific.

  - *The PS7 cell must be used in this Zynq design in order to enable correct default configuration.*

During synthesis and implementation, it was found that Vivado performs optimizations by default that lead to the cancellation of the redundant modules added for the TMR mitigation technique. To avoid this behavior, the DONT_TOUCH attribute was introduced and set to 'true' for each component that makes up the TMR. The DONT_TOUCH works in the same way as KEEP or KEEP_HIERARCHY attributes; however, unlike KEEP and KEEP_HIERARCHY , DONT_TOUCH is forward-annotated to place and route to prevent logic optimization. The use of the DONT_TOUCH attribute is essential for the TMR to be maintained, a fundamental condition for the entire treatment of the project. The addition of this attribute makes VHDL not technology independent, as it should be.

5 Results and Considerations

The following sections report the post-implementation results obtained for the four architectures.

5.1 Timing

This section reports the results relating to the timing performance obtained following the implementation of the ADD FU for CGRA in its four different architectures. The evaluation metric considered is that of worst hold slack (WHS) and worst negative slack (WNS) which determine the maximum achievable clock frequency. The performance analysis was performed by specifying a 10 ns period constraint for the clock signal, i.e. a clock frequency of 100 MHz. The results show that to achieve maximum

Version	**WHS [ns]**	**WNS [ns]**	**Max. Clock Freq. [MHz]**
ADD FU Simple	0.145	5.542	224.32
ADD FU Comb. TMR	0.098	1.013	111.27
ADD FU Reg. TMR	0.124	4.678	187.90
ADD FU Full TMR	0.110	1.069	111.97

Table 1: Timing results

speed the best architecture is the simple version without TMR (this makes perfect sense and was easily deduced). Among the versions with TMR it is possible to notice how the addition of TMR only on registers and FIFOs does not greatly affect (with respect to other architectures) the maximum achievable frequency compared to the version without TMR. This version appears to be, among those with TMR, the one that guarantees the highest maximum achievable frequency. In terms of speed, the worst architectures are the Full TMR one and the one with TMR only in the combinatorial logic (comparable to each other). The TMR on combinatorial logic is therefore the one with the greatest impact on speed. Considering that the Full TMR version and the one with TMR only in the combinatorial network are comparable, it is possible to assume that Vivado was able to perform a notable optimization in the first of the two. The version with TMR only in the combinatorial logic is an uninteresting architecture from the point of view of the metric in question since for the same performance it is better to prefer the one that with the TMR mitigates faults in the entire circuit. Assuming that TMR is required to be implemented, if the maximum achievable frequency has priority over the fault tolerance level, the best choice would be the architecture with TMR only on registers and FIFOs. If, however, the priority was the level of fault tolerance, given the large difference between the two architectures, the full TMR version would be the most appropriate choice. The critical path, i.e. the one with the longest register-to-register delay, as might be expected, is the logic chain composed by the FIFO2's last stage, Ripple Carry Adder, Flag Generator, Flag Selector and Output register. If the performance of one or all of the architectures is not satisfactory, it is possible to try to improve them using a different synthesis and implementation algorithm/strategy oriented towards performance optimization or by adding a register pipeline at the output and then carrying out the automatic retiming process of Vivado, which reorganizes registers to optimize performance.

5.2 Area Utilization

Another important metric to consider is the area occupied by the circuit, which can be expressed as the amount of resources required to map the design onto the selected FPGA. The results shown by the post-implementation resource utilization report

Version	**LUTs (17600)**	**Flip-Flops (35200)**	**Slices (4400)**
ADD FU Simple	21	137	35
ADD FU Comb. TMR	50	137	42
ADD FU Reg. TMR	135	335	89
ADD FU Full TMR	177	335	92

Table 2: Resource utilization results

are as expected. In fact, an increasing circuit complexity corresponds to a greater number of resources used. The architecture that uses the most resources is the one in which TMR is applied across the entire circuit. The architecture that uses the least number of resources is the one without the application of TMR. Interesting to note how:

- The architecture with TMR only on combinatorial logic differs from the simple one only in the number of LUTs and slices used and not in the number of registers (the memory elements do not change in the two architectures).
- The architecture with TMR only on registers and FIFOs differs from that with TMR only on combinational logic for all three types of resources. The increase in LUTs can be justified by the need to add logic to manage the added memory elements.

- The full TMR architecture differs from the one with TMR only on registers and FIFO only for the number of LUTs and slices (the memory elements do not change in the two architectures).

The number of slices used increases significantly in the second comparison but not too much in the first and third. If the results of one or all of the architectures is not satisfactory, it is possible to try to improve them using a different synthesis and implementation algorithm/strategy oriented towards area consumption optimization.

5.3 Power Consumption

The last metric taken into account is the estimated power consumption for the circuit designed on the selected FPGA. A summary table containing details for each implementation is shown. It contains results produced by the synthesis tool, obtained with medium confidence level of the report. The first table shows the total on-chip power and the static power and dynamic power components with the percentages of the total. All data reported in the following tables are to be considered approximate and are authentic to what is reported by Vivado in the summary of consumption results.

Version	Total [W]	Static [W](%)	Dynamic [W](%)
ADD FU Simple	0.091	≈ 0.090(98%)	≈ 0.002(2%)
ADD FU Comb. TMR	0.092	≈ 0.090(98%)	≈ 0.002(2%)
ADD FU Reg. TMR	0.096	≈ 0.090(94%)	≈ 0.006(6%)
ADD FU Full TMR	0.097	≈ 0.090(93%)	≈ 0.007(7%)

Table 3: Power consumption results

The internal components of dynamic power are analyzed here.

Version	Clocks [W](%)	Signals [W](%)	Logic [W](%)
ADD FU Simple	0.001 (79%)	< 0.001 (14%)	< 0.001 (7%)
ADD FU Comb. TMR	0.001 (62%)	< 0.001 (22%)	< 0.001 (16%)
ADD FU Reg. TMR	0.004 (66%)	0.001 (21%)	0.001 (13%)
ADD FU Full TMR	0.004 (56%)	0.002 (26%)	0.001 (18%)

Table 4: Dynamic power consumption components

As expected, the power consumption of the circuit increases as the complexity of the circuit increases. Among the versions with partial TMR, the one with TMR only on the combinatorial part has lower power consumption. If the results of one or all of the architectures is not satisfactory, it is possible to try to improve them using a different synthesis and implementation algorithm/strategy oriented towards power consumption optimization.

6 Conclusions

The ADD FU of a CGRA designed and analyzed in this report has an architecture based on simplicity rather than optimizations that improve one of the performances (time, area, or power consumption). The designed architectures proved capable of carrying out their task of adding operands while respecting all the specifications and requirements imposed in the trace. The creation of the TMR in the three different versions allows them to be compared with each other according to the metrics under examination. The introduction of the TMR mitigation technique increases the complexity of the circuit and at the same time also increases its fault tolerance. It is therefore necessary to seek a trade-off in order to identify an architecture that best reflects the needs, both in terms of area/power consumption and clock frequency and in terms of fault tolerance. According to the metrics under consideration, as reported in the previous section, the architectures have characteristics that make their application under these conditions more or less attractive. In order to improve performance it is possible to apply the strategies declared in the relevant sections for each metric. As a further possible starting point for the advancement of this project, the following is suggested:

- Perform simulations and tests on the final post-synthesis netlist to be sure that the TMR was not canceled by Vivado during the optimization phase.
- Ensure that the placement occurs in such a way that the TMR modules relating to the same component do not have common resources (they should be in independent areas).
- Identify alternative techniques to the DONT_TOUCH attribute that ensure the TMR is not eliminated in optimization but at the same time allow Vivado to perform more in-depth optimizations.

Identifying the best architecture to adopt to meet the needs is impossible in a unique way. This task falls to those who must apply this architecture as it requires an in-depth analysis of the trade-off with respect to the necessary requirements and the priority of the different metrics analyzed and the level of fault tolerance required. Furthermore, TMR can also be applied in a mixed mode than presented in this report (e.g. only on FIFOs and RCAs but not elsewhere). This requires in-depth and ad-hoc analysis.

7 References

References

[1] Luca Zulberti, Matteo Monopoli, Pietro Nannipieri, and Luca Fanucci. Highly-parameterised cgra architecture for design space exploration of machine learning applications onboard satellites. 09 2023.

[2] Aviral Shrivastava, Jared Pager, Reiley Jeyapaul, Mahdi Hamzeh, and Sarma Vrudhula. Enabling multithreading on cgras. pages 255–264, 09 2011.

[3] Faiq Khalid, Syed Rafay Hasan, Osman Hasan, and Falah Awwad. Low power soft error tolerant macro synchronous micro asynchronous (msma)pipeline. 07 2014.

[4] Rahul Parhi, Chris H. Kim, and Keshab K. Parhi. Fault-tolerant ripple-carry binary adder using partial triple modular redundancy (ptmr). In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 41–44, 2015.