Master thesis - Electronic engineering

Design of a fault tolerant RISC-V instruction execute stage for safety critical applications

Supervisors:

Prof. Stefano Di Carlo
Prof. Alessandro Savino
Prof. Maurizio Martina
Prof. Guido Masera

Candidate:

Luca Fiore

April 5, 2021

Abstract

Combining performances, power consumption and fault tolerance in modern integrated circuits is a real challenge. The goal of this work is to present a possible technique to detect and correct both transient and permanent errors in the execution unit of a RISC-V core, without affecting performances. TMR, Standby-Sparing, Alpha Counting and other techniques are mixed together to protect the ALU and the Multiplier against single transient errors and multiple permanent errors. This technique can potentially be applied to any other critical component and its main advantage is that it allows the component affected by permanent fault to be reused in the remaining non-faulty subparts thus maximizing resources exploitation.

To my mum and my dad
who gave me life

Acknowledgements

I would like to thank my supervisors Prof. Stefano Di Carlo, Prof. Alessandro Savino, Prof. Maurizio Martina and Prof. Guido Masera for their professional support and guidance. Furthermore I would like to thank my two teammates Marcello Neri and Elia Ribaldone for their collaborative effort into this project leading to reach a great result. I would also like to acknowledge Politecnico di Torino for the amazing journey I have had the past five years as a student. Thank you "my family", thank you Jessica.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

Electronic devices are currently used in a such a wide variety of fields that leads to the necessity to design devices suitable for really different environments.
For this reason a single component such as a processor can be realized in many different ways because of the necessity to meet multiple different requirements and to be suitable for various scenarios. For example the processor designed for a smart washing machine differs from that of an aircraft on board control system for performance reasons, security reasons and reliability reasons, and this also entails a diversity in terms of cost.

Anyway, whether we talk about critical applications or simple minor applications the customer who invests on a system will require longevity, safety and reliability.
Nowadays the complexity of a machine is so high that the human user can't manage all the aspects or the problems related to the machine functioning, so it is necessary to limit at the maximum the possibility that a system fails. The three main reasons are:

1. economic reasons: a broken device has to be substituted or fixed;
2. time reasons: the repairing time is lost time for the application;
3. safety reasons: a faulty device can potentially be dangerous.

In this work we want to find solutions to make a simple processor able to tolerate errors and possibly to recover from them, in order to obtain a

device suitable for safety critical applications.
About processors, lots of researches and efforts were made in order to make the device secure and reliable (on these terms see 3.1) and many techniques have been discovered to improve processors endurance.
Among the multitude of processors used in embedded world, we will focus on RISC-V, an open source ISA (Instruction set architecture) that has became one the most promising architecture because of its simplicity and open sources licenses.
The basic idea of the RISC-V project at the beginning was exactly to have a large base of contributors who freely could use the ISA in academic and in any hardware or software designs. At the time of writing, lots of RISC-V implementations are used (Western digital SweRV [59] or Alibaba XuanTie [15]), some others are in development (cv32e40p and other cores on PULP platforms [68]) and probably many others will be designed for the reasons we have just explained.

1.2 Thesis organization

This work is organized as follows:

1. Chapter 1: Introduction to get a general idea on the motivations that led us to develop this project.

2. Chapter 2: Overview on the international standards for functional safety that guided us in this project.
   Our goal to provide an fault tolerant HW extension for a processor and we want to make our work suitable for comparison with other researches, hence the need for a standard guideline that standardize both the safety problem and the required process to manage it.

3. Chapter 3: Introduction to fault tolerance processor, terminology and techniques. In this project we will adopt some already known solutions to protect our RISC-V core, but we will try to explore some new approaches to achieve a better fault tolerance.
   Chapter 3 is therefore a detailed description of the state of the art in processor fault tolerance, in particular the two main themes of *error detection* and *error correction* are dealt with.

4. Chapter 4: In this Chapter there is a general description of the RISC-V ISA, the historical milestones and the main characteristics of the ISA.

5. Chapter 5: Description of the cv32e40p core (the RISC-V implementation adopted in this work), with particular attention to the Execution Unit of the core, mainly including the ALU and the Multiplier.

6. Chapter 6: This is the main Chapter of this work because it deal with the particular hardware designed on the cv32e40p to protect its Execution Unit.
   After an introduction to the fault tolerance problem on the cv32e40p we introduce our main ideas to solve the problem and finally there is the detailed description of the hardware extension provided to the cv32e40p.

7. Chapter 7: Shows the results in terms of simulations results. first of all, there is the description of the simulation environment and then we show the simulations done to verify the correct behaviour of the core and to evaluate its fault tolerance level.

8. Chapter 8: Summary of the main conclusions and some considerations on the results obtained.

9. Chapter 9: It presents some ideas for further developments on the core.

The main acronyms are summarized in the Nomenclature section to allow the reading of this thesis by experts and not-experts.

Chapter 2

Standard for functional safety

At the beginning of the 20th century there was an incredibly fast advancement in electronics and computer science which led to the growth of industrial production. For this reason arose the need for standard guidelines for industrial activities and in particular the compatibility of different standards between countries became fundamental. In fact, every country begins to open up to the world, leaving its borders as trade began to involve the whole world in a way never seen before. Nowadays there are many international standardization organizations that actually collaborate to provide documents and guidelines for almost every type of industrial activity and also in other sectors. Regarding electrical and electronic devices, the main organizations are the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) which are responsible for the development, maintenance and promotion of standards in the fields of technology of information and communication (ICT).
In our work we will refer to standards developed by those organization to have some guidelines to orient our reasoning in a proper way.
The main reference we will follow is *IEC* 61508, the international standard currently adopted by companies and industries to ensure functional safety of electrical, electronic and programmable electronic (E/E/PE) safety-related systems. It is divided into seven parts and its two main objectives are to help individual industries develop supplemental standards, tailored specifically to those industries based on the original 61508 standard, and on the other hand to enable the development functional safety systems where specific application sector standards do not already exist [42].
For example IEC61511 for the process industries, IEC 62061 addressing machinery safety and IEC 61513 for the nuclear industry come directly from

IEC 61508 and reference it accordingly.

The first 3 parts of the Standard represent the technical requirements (general, hardware and software requirements) and the remaining 4 parts contain all the supporting informations (definitions and abbreviations, examples, guidelines on the application of Part 2 and 3, and so on); the relationship between these seven parts is shown in Figure 2.0.1.

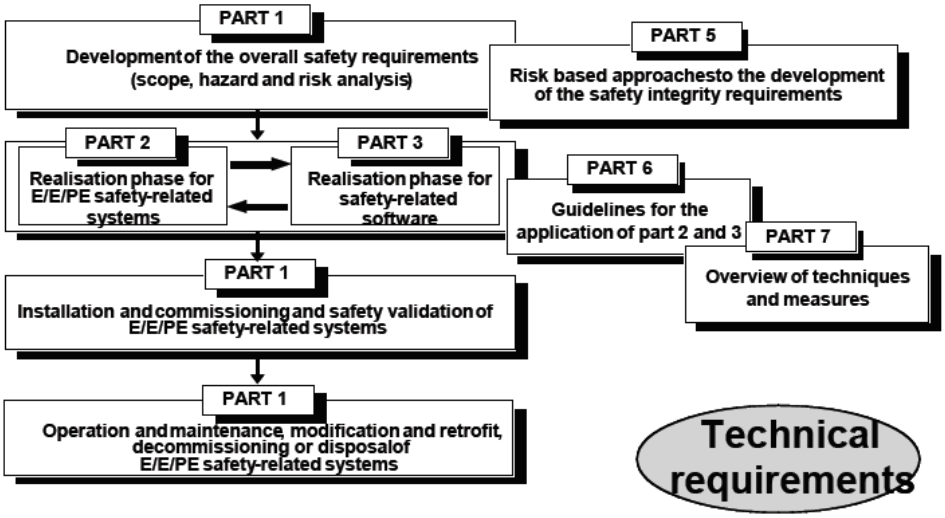

Figure 2.0.1: IEC 61508 - Technical Requirements

IEC 61508 is concerned with the E/E/PE safety-related systems whose failure could affect the safety of persons, animals and/or the environment. The main concept of all documents is *functional safety* defined as the detection of a potentially dangerous condition that triggers the activation of a protective or corrective device or mechanism to prevent the occurrence of dangerous events or that provides mitigation to reduce the consequences of the dangerous event. *functional safety* is the part of safety that depends on the correct functioning of a system or equipment in response to its inputs, while safety is the absence of unacceptable risks of physical injury or damage to the health of persons, directly or indirectly as a result of damage to property or the environment.
From the concepts of *functional safety* come other two fundamental concepts: the *safety life cycle* and the *safety integrity levels*. *Safety life cycle* is the intended as the process necessary to achieve *functional safety* while

safety integrity levels are levels of risk reduction and are four, the higher one is that related to the strongest risk reduction.
The standard is far more complex than described in the previous few lines but the essential is to approach the digital design for fault tolerance in a way that follow the standard guidelines.

The goal of the project described in this report is exactly to extend the architecture of a RISC-V core to reach a good level of fault tolerance; to be compliant to *IEC* 61508 we will follow the safety life cycle as described in Figure 2.0.2.

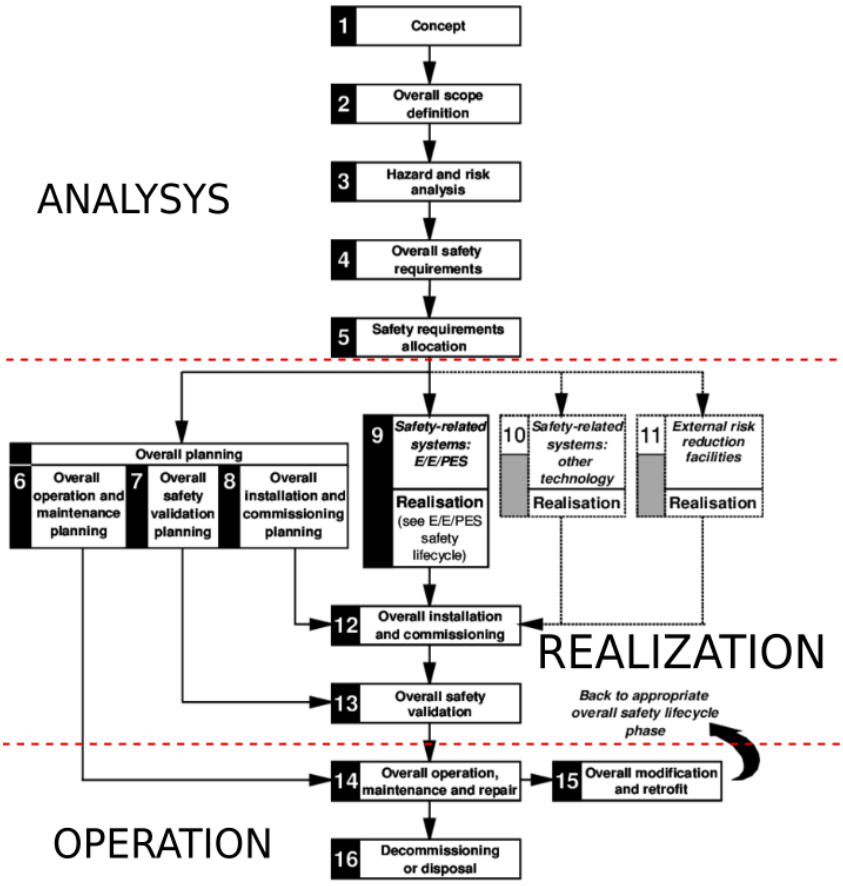

Figure 2.0.2: IEC 61508 - Safety life cycle [42]

The idea behind the safety life cycle is to develop and document a safety

plan, execute that plan, document its execution (to prove that the plan has been met), and continue to follow that safety plan until decommissioned with additional appropriate documentation throughout the life of the system. Changes along the way must similarly follow the pattern of planning, execution, validation, and documentation.

In our project we will not follow all the steps required by a complete Safety Life Cycle because our final product will not be a commercial and physical device, so we will try to take into consideration the Standard and to follow its guidelines but we will certainly have to adapt the Standard to our case of study as any company do for their special projects. Our target is to investigate and design innovative solutions to increase the fault tolerance of our processor but basically we don't know where and for which purposes the core will be used, therefore we can only make some hypotheses.

Chapter 3

Fault tolerant processors

Very often in modern era the real center of human activities is no more the human itself but electronic devices with their brain: processors together with machine learning. Processor are able to execute millions of simple instructions per second and nowadays they are used in almost any kind of electronic device. We are in the so called "smart era" and the future is going to be even smarter as long as the earth allows it. Anyway, apart from ethical and environmental issues, modern human activities are strictly related to computers, smartphone and vehicles that always integrate some computational unit and so we can extend the concept of processor to almost everything around us.
In this complex modern era, humans want to feel helped and supported by electronic devices but in particular they want to feel safe because sometimes using a too technological device, and therefore a device that is not completely and deeply known, can worry the user. For example if we think about the autonomous driving mode of modern vehicles, we understand how difficult can be for some users to completely trust the engineering team that have developed that driving mode.
Therefore, safety is an important requirement for modern electronic device whether we talk about a simple city car or an innovative space aircraft module, and this is valid for both processors or any other part of a system.

3.1 Fault tolerance terminology

In this paper we will focus on a RISC-V processor for safety critical applications such as space mission flights where the dependability is probably the main characteristic that must be preserved. Depending on the special

application and field of use, different emphasis can be put on the word *dependability* and in the following summary we try to analyze all the different meaning.

- *Dependability*: is the property of a computer system for which we can legitimately rely on the service it provides;
- *Reliability*: is the continuity of the service. The reliability of a system at time t is the probability that the system has been operating correctly from time zero until time t.
- *Availability*: is the readiness to use of the service. The availability of a system at time t is the probability that the system is operating correctly at time t;
- *Safety*: is the non occurrence of any dangerous events due to malfunctioning of the system;
- *integrity*: is the non occurrence of undesired corruption of informations.
- *Mean Time To Failure (MTTF)*: Mean time to failure can be a useful metric to have an idea of the expected life of our device. Unfortunately this parameter can't be considered standalone since we need to know also the variance of failures because a device with an high MTTF but with an high variance can potentially be worse than another device with both parameters at a lower level.
- *Mean Time Between Failure (MTBF)*: Mean time between failures is derived from MTTF with the addition of the extra time required to repair the device, the mean time to repair (MTTR). Therefore the device will be available for less than 100% of the time:

$$Availability = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR} \tag{3.1}$$

Therefore a device can deliver its service in many different ways, with discontinuity, with non-perfect output, with possibility to damage the surrounding environment or it can have all the properties mentioned above and so be a good device that can be used with confidence.

3.1.1 Faults, errors and failures [85]

An electronic device such as a processor, as described in this paper will certainly experiment some kind of problems due to time wearing, after one year as well as after one day, and this could cause the fail of the service provided by the device. For example a damaged wire or a broken transistor can produce an incorrect output leading to an unuseful and potentially dangerous processor.

In this perspective, the words *fault*, *error* and *failure* are erroneously used as synonyms but all have their own specific meaning: fault is physical defect for example due to time wearing or external physical damage and it can manifest itself as an error, such as a bit flip, or it can be masked by architectural solutions or by application reasons. In the same way, an error can produce a failure (so a user-visible incorrect behaviour of the system) or not.
According to the duration of faults (and errors) we distinguish between transient, permanent and intermittent faults (errors) which therefore require different fault tolerant techniques.

- transient: is the so called *soft error* or *single event upset* (SEU). This kind of error is temporary and it usually last few clock cycles (1 clock cycle in the extreme case).
  Four are the most important causes of transient faults in a processor or in a general electronic device [94]:

  1. high energy particles produced when cosmic radiations impact the atmosphere
  2. alpha particles produced by natural decay of radioactive materials that are very often the metallic materials of the package itself.
  3. electromagnetic interference: this source of interference can come from outside or from inside, also called cross-talk.
  4. current spikes: it is the so called $\delta I/\delta t$ problem and can produce unexpected behaviour because of unexpected high current draw [67].

  The most common FT techniques in this case include spatial and temporal redundancy (see 3.2).

- permanent: is the so called *hard error* because it last forever until some external operation repairs the damaged component. This kind of error can be due to three main reasons:

1. temporal wear-out: electromigration [16][33], break down of transistor's gate oxide [74][51], mechanical stress, high voltage, high temperature and others [86]. These kind of faults can be limited by voltage and frequency scaling [87], dynamic temperature adjustment [82] or adaptive body biasing [88].
2. chip fabrication defects: this defects can be detected before device shipment through a test, but sometimes the test has not a 100% coverage or sometimes the defect can't be detected if the chip is not in the field.
3. bad design: in this situation the error is permanent because even a perfectly fabricated device won't behave as expected. This type of error should be avoided thanks to simulations during verification but sometimes the same verification environment doesn't provide a full testing for that specific incorrect feature, and so the bug arrives to the final product as it did for some processors in the past [30][43].

   The most common FT technique in the above cases is the use of a fault-free duplicate of the component that substitutes the corrupted one, so mainly spatial redundancy.

- intermittent: is a fault or error that happens repeatedly in the same place due to process variation combined with voltage and temperature fluctuation. There are some techniques developed exactly for this kind of errors [92] but in general the approach is to treat them as permanent or as transient.

Another important classification of errors is based on the symptoms that HW errors can cause in SW at assembler level [76]. It is an error model extended from Forin initial model [34], used in HW independent error detection mechanism, so at SW level, but it is still very interesting:

1. Exchanged operand: A different but valid operand is used.
2. Exchanged operator: A different operator is used, for example, an addition is executed instead of a subtraction. The operands remain the same.
3. Faulty operation: An operator such as addition or subtraction does not work as expected and produces incorrect results despite of correct input values. Every usage of the result produced is influenced by this error.

4. Lost update: A store operation to a register or memory location is omitted. This can result in the usage of out-dated values later on.

5. Modified operand: An operand used by an instruction is modified by a single or a multiple bit-flip. In contrast to a faulty operation, this error only influences one read of a value.

3.2 Fault tolerant techniques: error detection

When working with a specific digital device, let's say a μprocessor, we have to know which will be the application environment of our device to know the types of error we are going to deal with. For this reason the error models described above become very useful into errors classification and diversification. The subsequent step after knowing the problems that our device would encounter is to detect the possible error and then to recover from it if possible. Error detection and error recovery are so the main topics of this work and special attention will put on their meaning into the Execution Unit of a RISC-V core.

Among the multitude of FT techniques examined over years we are going to focus on those one related to the ALU and to execution unit in general. The keyword in FT field is always *redundancy*. It is important to distinguish between spatial redundancy and temporal redundancy even if hybrid solutions are often used. We talk about spatial redundancy when the basic component that we are going to protect against faults is surrounded by extra HW useful to detect and sometimes correct errors, while on the other hand we talk about temporal redundancy when it is required some extra time, with respect to the normal execution, to identify some errors and eventually correct them.

In the following list there is an overview on the main FT techniques developed in the past years and some ideas for our specific application on a RISC-V core; we are going to distinguish between spatial and temporal redundancy techniques even if as expected some techniques are a mixture.

3.2.1 Spatial redundancy techniques

These techniques require always some extra HW to ensure a certain level of security to the device and so the main drawback we can face are related to occupied area and power consumption of the extra components.

Modular redundancy

It consist in inserting some replicas of the unit to protect (UTP) and compare the output coming from the replicas to check if something in the execution went wrong and eventually recover from errors. Transient errors are the main target of this technique but permanent errors can be detected too, but not corrected. In fact, by performing the same operation many times and verifying that the same error has occurred, we can conclude that it is a permanent error (for example the repetition of the same operation can be performed during spare cycles if the UTP is free but for additional informations on this technique see Temporal Redundancy 3.2.2 and BIST, Built In Self Test).

This technique is very efficient in terms of error detection and correction (if the number of replicas is greater than 2 and if it is an odd number, see the TMR presentation below) but can't provide any kind of help against design bugs. Infact even if we use a very high number of replicas, if they are all equal and so if they are all affected by the same design bug, it can not be detected.
An important solution to overcome this problem is the so called *etherogeneous replica* or in general *design diversity* [7] where at least one replica has to be architecturally different from the fundamental one.
In this way there is a chance that if a design bugs occurs into an identical replica of the fundamental component the different replica could be fault free.
An extreme example of this solution is the *watch-dog timer* [57] where this "replica" monitors some hardware for sign of life. For example a processor watch-dog timer (see temporal redundancy below) is a simple coproccessor that tracks memory request on the bus and if the bus is empty for too long time it is interpreted as sign of error and something is done to recover from this situation.

- *DMR - Double Modular Redundancy*: Duplicate the UTP and insert a comparator to check the equality of the outputs of the two replicas as shown in Figure 3.2.1. DMR is very simple technique and very effective to detect single errors except for those affecting the comparator, design

bugs and simultaneous errors in both the modules.

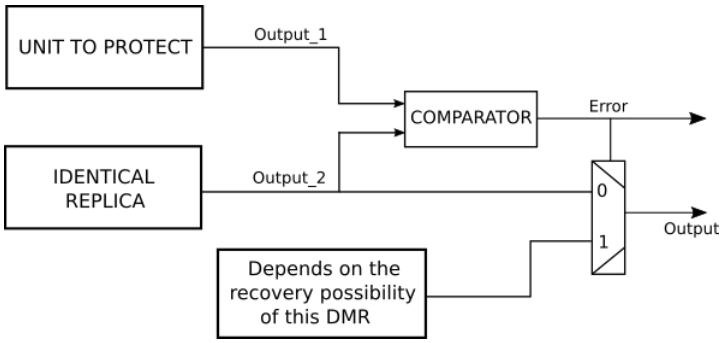

Figure 3.2.1: DMR scheme

In DMR There is no possibility of recovery if the two replicas are identical because neither of them has higher priority over the other. Anyway there are two tricks that can be adopted to add recovery to DMR:

1. self check each of the two modules using spare cycles of the core. If one of the two modules reports to have more errors than the other in the future it will be assigned less priority than the other which in this case will be trusted more.
2. add information redundancy to each of the two replicas obtaining a two level fault tolerant system. The comparator checks the equality of the two outputs and can immediately know if something went wrong but then internally each of the two modules can know if the operation has been performed correctly or not.

This technique will result in more than double power consumption and occupied area (two modules + comparator) while regarding the delay little overhead has to be accepted only due to the comparator.

- *TMR - Triple Modular Redundancy*: Triplicate the UTP and insert a majority voter to check if one of the unit has encountered some problems and eventually discard that output as shown in Figure 3.2.2.
  It is conceptually really close to DMR because only one error can be detected as in DMR, but this time it can be corrected because the two fault free modules will produce the same output that is considered to be the correct one.

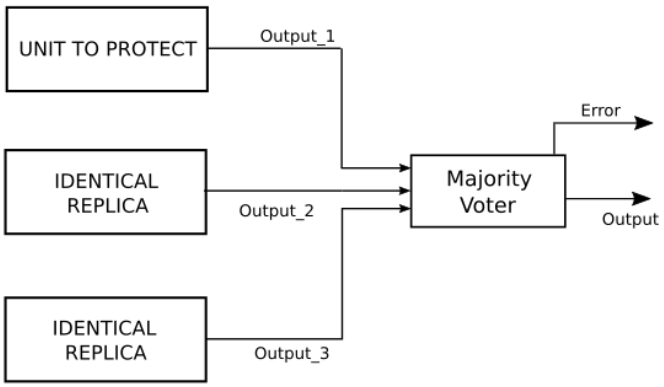

Figure 3.2.2: TMR scheme

One problem related to this technique is that it is unable to prevent errors in the majority voter. Therefor if the voter is faulty the TMR mechanism will be ineffective and even dangerous because can let errors to be propagated to the following stage. To remedy this problem a two stage TMR [55] can be applied as in Figure 3.2.3.

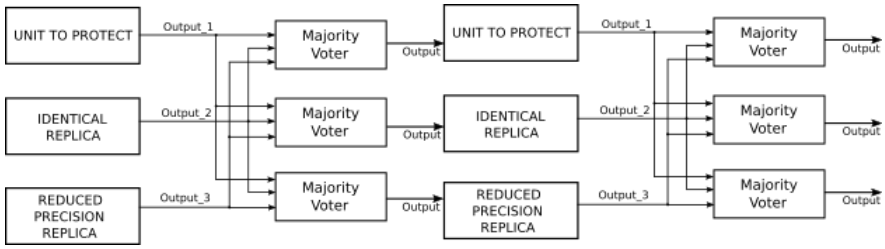

Figure 3.2.3: Two stage TMR scheme

In this way there is redundancy for the voter too and so if we know that we are going to use TMR for two subsequent unit or if we can split our unit into two sub units we can directly leave three voters and solve the problem of faulty voter for that unit.

Another intermediate solution similar to the two stage TMR is something where we use a two level voting as in Figure 3.2.4. This technique hes been little investigated in the past [79] because of extra complexity which means more area and so higher probability of faults and because the final voter can still be subjected to errors. Obviously if we consider the fault occurrence in all the component to be equally probable

there is no advantage from this technique. What we think on the other hand is that the probability that one of the three parallel voters has a fault is lower than the probability that one of the three replicas of the functional unit is wrong. So the overall probability than one input of the final voter is faulty is really low and in the same way there is less probability to fall in an undetectable fault because it far less likely to find that two input of the final voter are wrong. This statement comes from the consideration that one ALU is surely more complex than a voter and therefore the probability an ALU is faulty is greater that the probability a voter is faulty.

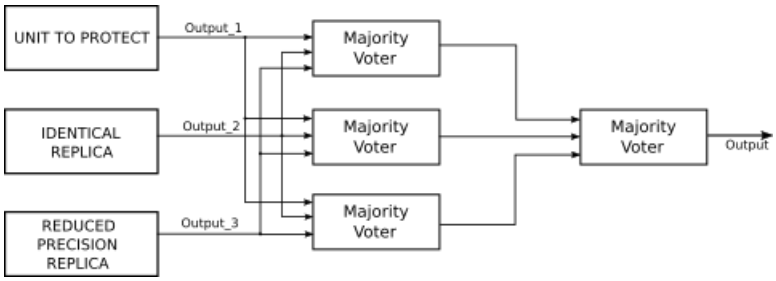

Figure 3.2.4: 2 level voting TMR scheme

Other hybrid solutions can be investigated but still the biggest problem remains that only one error can be detected and corrected. However, as always, if there are multiple errors masking each other or if there is the same error in two modules, the errors cannot be detected and instead there is the serious risk to have to error directly on the output. TMR as the standard DMR with no "diversity", is blind to design bugs and for this reason "heterogeneous" replicas can be exploited (see the following NRMR).

- *NRMR - N Reduced precision Modular Redundancy*: Replicate N times the UTP and insert a voter to detect and eventually correct error(s). If needed some of these replicas can have reduced precision [52] or different microarchitecture with respect to the fundamental one which leads to the so called design diversity. This means to use a replica that will produce an output with different HW effort with respect to the fundamental unit and this is useful especially if the number N of replicas become large (5 or more) in order to limit the Area and Power consumption overhead.
  See Figure 3.2.5 to have a visual idea of this technique.

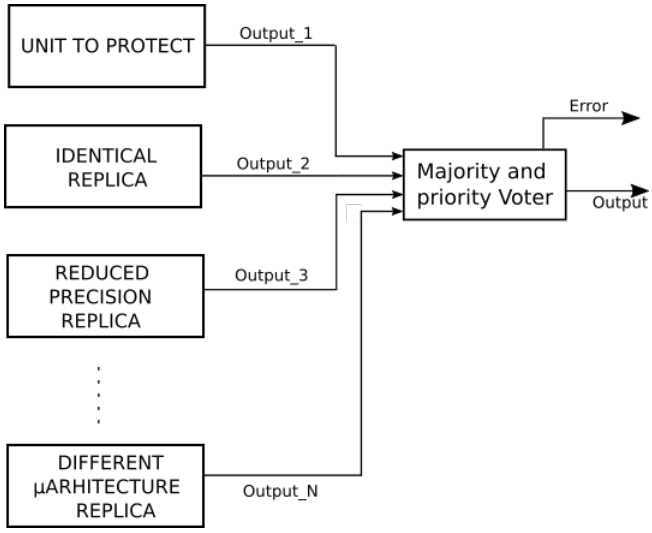

Figure 3.2.5: NRMR scheme

This time, as in TMR the voter strongly contributes to the area, power and delay overheads and for this reason some technique were adopted to contain the voter impact on the performance of the circuit [46].

Important remark about the choice of N: If N is and odd number it means that there is not the possibility of parity conditions, that is when for example with N=4 two modules say something and the other two agree on something different.
On the other hand N has to be large enough to detect the number of errors that we expect to occur but not too large because N times as much hardware is susceptible to N times as many errors, if we assume a constant error rate per unit of hardware [85].

Information redundancy - Arithmetic codes

A special case of spatial redundancy is the so called *information redundancy* and in particular we will focus on arithmetic codes.
Arithmetic codes add redundancy to processed data leading to the creation of a larger "domain" of possible data words that include the smaller subset of valid code words. In a fault free operation the output of such encoded circuit is always a valid code word, while on the other hand, faulty arith-

metic operations do not preserve the code with a high probability, that is, faulty operations most likely result in a non-valid output code word [76] [6]. Arithmetic codes can also detect errors produced by faulty transmission or storage because the transmitted or stored value will be invalid code word with high probability and so it will be detectable by checking the code.

This kind of circuits are also called *self checking* circuits [90] because the circuit itself is able to recognize if an error occurred by means of a checker. In this field we are going to examine four particular coding techniques useful in ALU operations: Berger codes, Residue codes, AN codes and parity codes, and we will use the expression "functional value" to indicate the word we are going to protect and encode, and the expression "redundant part" to indicate the extra bits added to the functional value.

The presented codes are classified according to the relation between the functional value and the redundant part (or check bits): if the code word can be clearly divided into the two parts we said it is a *systematic* code and if the check bits are computed in parallel to the functional value the code is *separate*.

- Berger codes [10]

Berger code is a systematic and separate code. This technique can detect all the unidirectional errors that occur in the same direction and so simple bit flips that move '1' into '0' and '0' into '1' but not mixed transitions. Obviously Berger codes can not correct errors as the majority of information redundancy techniques.
Berger codes consist of the X bit functional value and a word of N bit with N=$\lceil \log_2 X + 1 \rceil$ reporting the binary value of the number of '0' contained in the functional value itself. For example in the 8bit word "01101011" there are 3 zeros and so the final code word will be "01101011|011" where the last three bits give us information on the number of '0s' in the functional value. At this link at the time of writing you can find a good and user-friendly app to explore Berger Codes potentialities.

The main advantage of this technique is that the functional value can be immediately read from the encoded word but there are unfortunately some drawbacks. The first problem is about the kind of errors that Berger Codes can detect as mentioned before, that is, only unidirectional bit-flips. For example imagine that in the previous 8bit word one bit flips form '1' to '0' and another one from '0' to '1', this is the typical condition of *undetectable* error or *silent* error because the final value will still be a valid code word.

Despite this, referring to 3.1.1 this technique is effective for different kind of errors such as faulty operations or modified operands (for example we encode an operand saved in a register and when we are ready for the computation, we read from that register a value that doesn't match with its code that means that the operator will receive a modified operand). Depending on the HW implementation it can detect also exchanged operands and exchanged operators (for example the two operands are exchanged and so the redundant code of the two don't match the functional values, and the same for the exchanged operator because the redundant code of the result doesn't match the functional value of the results because it was obtained with a different operation).

Another important issue of these redundant codes techniques including Berger code, is that the computation of the redundant part has to be performed separately as depicted in Figure 3.2.6 with some extra hardware that has the "same" fault probability of the protected unit.

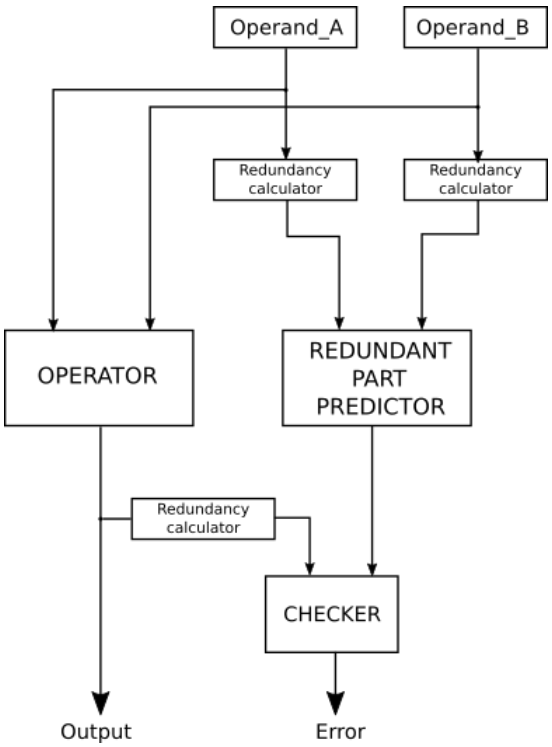

Figure 3.2.6: Berger code scheme

Figure 3.2.6 shows a simple idea behind Berger codes and other redundant code solutions. To have a better idea of a self checking ALU using Berger codes have a look at [53] and [54].

However, there are many situations where the detection of all possible unidirectional errors (provided by Berger codes) is not necessary and this has led to the development of several 'modified' versions that allow to overcome even the limitations of the Berger code: the main ones are Bose-Lin code and Dong's code.

Bose-Lin code is a *systematic* error-detecting code for detecting up to t unidirectional errors and requires fewer check bits and a much simpler checker than a Berger code [13] [39] [28]. If the number of information bits X is greater than $2^N - 1$, Bose-Lin code can be better than Berger code. the main advantage of this technique is that it needs only a fixed number of check bits independent of the number of information bits to detect a defined maximum number of errors.

Then we have Dong's Code. It is less expensive than Berger code (exactly as Bose-Lin code) in terms of the number of check bits and the complexity of checkers[40], but the Dong's code has the advantage that its error detection capability is a function of the number of check bits used, independent of the total number of the information bits; that is the error detection capability of code can be made to be application specific [56].

- Residue codes

Residue code is *systematic* and *separable* code, i.e. the redundant part is separated from the functional value as in Berger codes family that means it requires extra HW and operations to compute the results's check bits for each operation. The code is created by appending the residue of a number to itself [36]: the code word x_c for the functional value x is therefore the tuple of x and its residue to a code specific constant A greater than 1.

$$x_c = (x, x \ mod \ A) = (x, x_A), \ A > 1$$

We can refer to Figure 3.2.6 used to understand Berger approach to also understand Residue codes because they are both redundant code techniques and so the main difference is only on the redundancy computation that now is a residue computation.
In the following table are reported the supported and unsupported operations by residue codes and eventually the expression to compute both functional value and check bits.

encoded operation	implementation	
	functional value	**check bits**
arithmetic operations:		
$z_c = x_c +_c y_c$	$z = x + y$	$z_A = (x_A + y_A) \mod A$
$z_c = x_c -_c y_c$	$z = x - y$	$z_A = (x_A - y_A) \mod A$
$z_c = x_c *_c y_c$	$z = x * y$	$z_A = (x_A * y_A) \mod A$
$z_c = x_c /_c y_c$	$z = x/y$	*not directly encodable*
signed numbers: *supported*		
shift operations:		
$z_c = x_c <<_c y_c$	$z = x << y$	*not directly encodable*
$z_c = x_c >>_c y_c$	$z = x >> y$	*not directly encodable*
logical boolean operations:		
`or:` $z_c = x_c \,\|\|_c\, y_c$	$z = x \,\|\|\, y$	$z_A = x_A + y_A - x_A * y_A$
`and:` $z_c = x_c \,\&\&_c\, y_c$	$z = x \,\&\&\, y$	$z_A = x_A * y_A$
`not:` $z_c = !_c\, x_c$	$z = !\, x$	$z_A = 1 - x_A$
bitwise boolean operations:		
`or:` $z_c = x_c \,\|_c\, y_c$	$z = x \,\|\, y$	*not directly encodable*
`and:` $z_c = x_c \,\&_c\, y_c$	$z = x \,\&\, y$	*not directly encodable*
`not:` $z_c = \sim_c\, x_c$	$z = \sim\, x$	*not directly encodable*
comparisons: *not supported*		

Figure 3.2.7: Implementation of encoded operations for Residue codes [76]

A very unpleasant results from the previous table is that there is no direct encoding for division, shift operations and bit-wise Boolean operations. This means that to support these operations you have two possibilities: use some other technique or use loops of supported operations with their residue codes (for example the division can be emulated expensively using a loop that subtracts the divisor from the dividend until zero is reached).

With this technique we can detect faulty operations and modified operands (because of no matching of output and inputs residues respectively), exchanged operands and exchanged operator (because check bits are computed separately). All this considerations are valid if no masking errors occur (for example one operand is modified and randomly its residue is affected by an error in the same direction) because we could get a a valid tuple (x, x_A) even

if there were some errors.

Two advanced variants of Residue codes are the so called $Multiresidue\ codes$ and $Inverse\ residue\ codes$. The first adoption of Multiresidue approach is described in [70] for the case of Biresidue codes where is highlighted the possibility of error correction. As for the Inverse Residue codes, the check bits are $A - x_A$ instead of x_A and in [6] it is stated that they have significant advantages in fault detection of repeated-use faults.

- AN codes [76]

AN codes are non-separate and non-systematic code word because functional value and check bits are processed together and we can not distinguish them into the code word. The code word x_c for the functional value x is derived from the multiplication of x itself by a constant A greater than 1.

$$x_c = (x \cdot A),\ A > 1$$

To check if the code-word x_c is valid we just need to compute the modulus with A and verify that it is 0:

$$x_c\ mod\ A = 0$$

In the following table are reported the supported and unsupported operations by AN codes and eventually the expression to compute both functional value and check bits.

encoded operation	implementation
arithmetic operations:	
$z_c = x_c +_c y_c$	$z_c = Ax + Ay = A(x + y)$
$z_c = x_c -_c y_c$	$z_c = Ax - Ay = A(x - y)$
$z_c = x_c *_c y_c$	$z_c = (Ax * Ay)/A = A(x * y)$
$z_c = \lfloor x_c /_c y_c \rfloor_c$	$z_c = \lfloor (A * Ax)/\ Ay \rfloor = A \left\lfloor \frac{x}{y} \right\rfloor$
signed numbers: *supported*	
shift operations:	
$z_c = x_c <<_c y_c$	*not directly encodable*
$z_c = x_c >>_c y_c$	*not directly encodable*
logical boolean operations:	
`or:` $z_c = x_c \|\|_c y_c$	$z_c = x_c +_c y_c -_c x_c *_c y_c$
`and:` $z_c = x_c \&\&_c y_c$	$z_c = x_c *_c y_c$
`not:` $z_c = !_c x_c$	$z_c = 1_c -_c x_c = A -_c x_c$
bitwise boolean operations:	
`or:` $z_c = x_c \|_c y_c$	*not directly encodable*
`and:` $z_c = x_c \&_c y_c$	*not directly encodable*
`not:` $z_c = \sim_c x_c$	*not directly encodable*
comparisons: *AN-encoded numbers can be compared directly. However, obtaining a valid encoded result requires an unprotected and, thus, unsafe if-statement.*	

Figure 3.2.8: Implementation of encoded operations for AN code [76]

AN-code can detect with high probability faulty operations and modified operands. Actually it is very unlikely that for example a bit-flip into operands or a wrong result will result in another multiple of A.
The great difference with respect to Residue codes is that now division is supported with a direct encoding, but the two main drawbacks of this technique are:

1. the functional value can't be read directly from the encoded word but it is obtained by an integer division $x = x_c/A$.

2. no detection of exchanged operands and exchanged operator. In fact if the new operand is still a multiple of A nothing is wrong, and the

same for the exchanged operator, if it is executed a subtraction instead of an addition the results will be again a multiple of A.

To overcome the limits of AN codes, several variants have been developed over years:

- error correcting AN Codes: there are many examples of AN Codes able to correct errors as in [58], [70] and others. These solution are really complex and we will not go into detail, just know the idea behind them: choose A so that each correctable error pattern gives a specific modulus for A. An error pattern is composed of the original code word and the number, the position, and the direction of the bits flipped by the error.

  These AN-codes can be used to correct transmission errors and errors related to storage of the encoded value (therefore modified operands are correctable). However, it is not possible to correct errors occurring during the execution of operations. Thus, we can't correct the symptoms faulty operation, exchanged operand, exchanged operator, and lost update.

- Systematic AN Codes: to overcome the problem related to the mixture of functional value and redundant part into the code word, a systematic variant of AN codes has been exploited.
  The code word x_c is obtained using the following expression:

  $$x_c = 2^m \cdot x + [(-2^m \cdot x) \ mod \ A], \qquad 2^{(m-1)} < A < 2^m$$

  The code word is valid if $(-2^m \cdot x) \ mod \ A$ is equal to its check bits.
  In this way we have in the most significant part of the code word x_c the functional value shifted by m position and in the remaining part there is the redundant part (check bits). For this reason the code is systematic and it is also a valid AN code because x_c is again a multiple of A. More details at [60].
  The main drawback of this technique is that the code word of the result of an operation even simple such as an addition, can't always be computed directly from the operands. Actually an addition of two valid codes can result in invalid code or valid but wrong code and this will require some more adjustment to the result to be valid and correct.

- $|\text{gAN}|_{\text{M}}$ Codes: this variant of AN codes is strictly related to systematic AN codes because even $|\text{gAN}|_{\text{M}}$ codes are systematic and non-separate

code with the great difference that the check bits are in the MS position and the functional value is in the LS positions. For more details have a look at [71].

What we just want to highlight in this paragraph is that $|\text{gAN}|_{\text{M}}$ Codes do not support signed numbers and divisions but they are close with respect to addition, subtraction and multiplication in contrast with systematic AN codes.

- ANB Codes: this can be seen as an evolution of standard AN Codes because it try to solve the problem of undetectable exchanged operators and exchanged operand as seen in the section regarding AN Codes.
  P. Forin in [35] extend the idea under AN Codes with *signature* that is an extra parameter B_x in the following expression:

  $$x_c = (x \cdot A) + B_x,\ 0 < B_x < 1$$

  In this way, giving a special B_x to each operand will result in a code word that is not a simple multiple of A but something strictly related to that special x thanks to the B_x value. So the exchanged operators and exchanged operand errors can be easily detected because each word has its special encoding (or better set of encodings). The functional value x is obtained by an integer division $x = x_c/A$ and the correctness of the operand or operation is obtained just by computing $B_x = x_c \ mod\ A$.

  The main drawbacks are that there is no direct division encoding and that many corrections have to be applied to the result of an operation with ANB encoding in order to obtain a valid code word whose signature only depends on the signatures of the operands.

- ANBD Codes: In [35] Forin present a further extension of ANB code with D value. The reason for this extra variable is due to the necessity to detect also the so called *lost update* errors, that is an error caused by an incorrect store of data (see 3.1.1).
  The code word x_c is obtained with:

  $$x_c = (x \cdot A) + B_x + C_x,\ 0 < B_x < 1$$

  Because in addition to signature B_x there is also timestamp D that counts variable updates, this technique requires more corrections than ANB code to obtain a valid output code word, but fortunately this solution support the detection of *lost update* errors.

- Parity codes [63] [78]

Parity prediction circuits are not part of Arithmetic codes but we want to talk about them because of their effectiveness and because they are currently used in commercial microprocessor, such as the Fujitsu SPARC V microprocessor [50].

Parity code is a systematic and separate code. The code generator adds an extra bit, called parity bit, to the data frame in order to have the total number of 1s either odd or even depending upon the type of parity. This technique is good for single bit error detection only.

The two types of parity codes are

- Even Parity: the total number of bits in the message has to be even: if the message has an even number of 1s, the extra bit will be '0'.
- Odd Parity: the total number of bits in the message has to be odd: if the message has an even number of 1s, the extra bit will be '1'.

Parity prediction circuits generate parity bits for both the operands and the result, as shown in Figure 3.2.6. Remember, this technique can only be used for arithmetic operations. At the beginning two operands undergo an arithmetic operator and the parity of the result is computed; then parity bits for each operand are generated and are sent to a logical XOR gate that acts as the "redundant part predictor". Finally, this result is compared to the result of the first computation in the so called checker, to obtain the error signal.

Conclusion on spatial redundancy techniques

We have seen that there are lot of techniques suitable for error detection and correction for a generic ALU: those concerning modular redundancy are surely the most effective and simple in terms of HW implementation while those exploiting information redundancy can be very effective in some case but their implementation can result really complicated. For this reason, very often arithmetic codes are used in redundancy techniques implemented in SW but these are out of the scope of this work.

The main problem of some codes we have studied is the lack of the encoded division support, such as in Residue codes. In the proceeding of this work we will decide which techniques are the most suitable to protect

the EX unit of our specific core, but for the moment we can't exclude any of these solutions a priori.

Surely there are several other ideas developed over years in this field and surely those presented above are described very briefly, so please refer to the cited references to have more informations. We will go in deep only for those techniques that will be adopted in our project of protecting a RISC-V core

3.2.2 Temporal redundancy techniques

This family of protection techniques is based on the main idea of executing the operations more than once in the same or different way in order to detect and sometimes correct various types of errors.

It is clear that for this family of fault tolerant solutions the major issue concerns control signals. In fact, if we think of a fine grained temporal redundancy, where the modules can decide to activate or deactivate the redundancy, we understand that synchronizing the different modules will be a very ambitious job. On the other hand, temporal redundancy requires almost no extra HW in addition to the basic architecture contrary to what is typical in spatial redundancy.

The following are the main known temporal techniques:

- REDO: this is not an acronym as for example the following RESO and RERO, but just for semantic continuity with them we decided to use the name *REDO*. It stands exactly for redo, that is to recompute the same operation on the same HW and in the same way two or more times in order to detect possible errors.
  This approach allow to detect transient errors that vanish between two repetition of the operation but obviously it is ineffective in detecting a permanent error.

- RESO: this technique, together with RERO described in the next point, overcomes the limits of REDO that was able to detect only transient errors because it is able to detect both transient and permanent errors.

  RESO stands for Repeating with Shifted Operands [64], that is redo the same operation on the same HW with shifted operands. The result has to be shifted too in reverse direction in order to obtain a coherent value to compare to the one obtained with no-shifted operands (Figure 3.2.9).

For this reason the register that stores the value of the operand and of the result need to be shift registers and in particular if we decide to use a k-bit shift for n-bit operations, we need (n+k)-bit ALU and (n+k)-bit registers. The main problem of RESO is that the shifting operation and the comparison have to be considered fault free or at last they have to be protected in other ways (parity bits for shift registers and self checking comparator for the checker [4]).

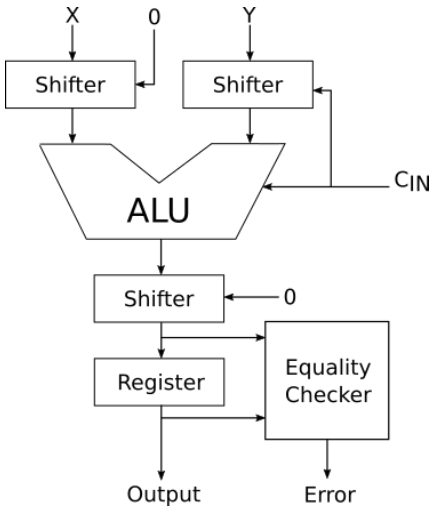

Figure 3.2.9: Implementation of RESO [64]

- RERO [49]: this acronym stands for Repeating with Rotated Operands and it is conceptually really similar to RESO with the only difference that operands are rotated instead of shifted. For this reason there is no more the need of shift registers and of (n+k)-bit ALU but ideally we can use n-bit register and n-bit ALU if operands are n-bit.
  Unfortunately with this approach there can be an incorrect result due to carry propagation from a bit inside the word to the MSB, so an additional bit is required for registers and ALU in order to ensure correct results (Figure 3.2.10) .

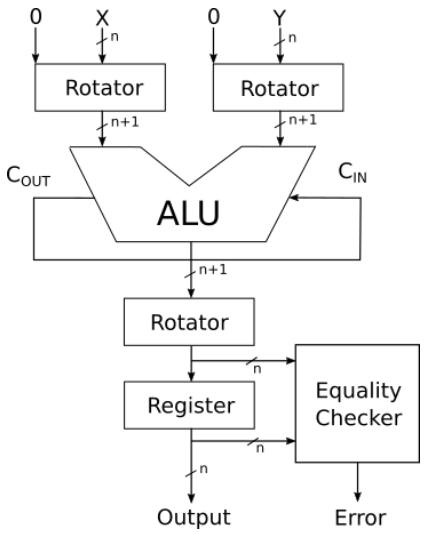

Figure 3.2.10: Implementation of RERO [49]

- BIST: built in self test is a technique that allows to detect both permanent and transient errors at the cost of energy spent in performing extra operations.

  As explained in 3.3.3, taking periodically checkpoints is a good way to ensure a quite recent state of the core that can be resumed if en error occurred in order to restart from a well known condition. During each "computation epoch", thus between taken checkpoints, the core can test a component if free, feeding it with known inputs and checking the result against well known output [81].

  In [84] Smolens et al. introduced a BIST variant called FIRST that performed periodically BIST at different clock frequencies looking at the maximum frequnecy at which the core no longer met timing requirements. Their purpose was to monitor wearing-out before any fault occurred, and in this way they were able to understand when the core would have a permanent fault because of the decrease in the maximum frequency.

  BIST is actually a diagnosis mechanism, necessary if the error is permanent since detection and recovery may not be sufficient in this case.

- Active-stream/Redundant-stream Simultaneous Multithreading (AR SMT): the idea behind this technique is to always fill the core with the maximum number of threads it can manage. In this way if a

core with T thread contexts at a certain point have to execute T'<T threads, the free space can be given to redundant threads. So a thread can be replicated and two identical threads can be executed in "parallel", that is a real parallelism if the processor is multicore. In this work we will focus on single RISC-V core so the interesting point is the multi-threads management with redundant threads on a single core.

This is counted as "Temporal Redundancy Technique" because of the time overhead it causes. In fact redundant threads on a single core will produce a queuing delay for the useful threads because of necessary resource sharing between threads [83]. Lot of works were done on this technique [75], [48], [77] and others, to improve the initial idea of SMT for example avoiding resource sharing between threads to speed up execution, but in this work we are going to focus only on the Execution Unit of a RISC-V core and for this reason SMT will not be considered.

Conclusion on temporal redundancy techniques

As we have seen in the previous paragraph, temporal redundancy techniques could be very effective in detecting both transient and permanent errors avoiding extra HW, at the cost of performances reduction.
Obviously there are lot of other techniques involving temporal redundancy that have been analyzed and adopted over years; sometimes they are a mixture of the above mentioned solutions, for example the Recomputing with Swapped Operand (RESWO) [44] or the Recomputing with Duplication with Comparison (REDWC) [14].

Regarding our fault tolerant RISC-V core we will adopt only spatial redundancy techniques for reasons that will be clearer later, so we will not go in detail on temporal techniques for our work.

3.3 Fault tolerant techniques: error recovery [61]

The purpose of this work is to provide an efficient fault tolerant version of an existent RISC-V core. In particular the main target is to design an HW error detection mechanism capable of signaling to the SW that something went wrong and try to manage the problems.
Very often the error recovery mechanism is left to the SW but sometimes it can be implemented partially in HW thanks to some redundant techniques such as TMR or NRMR as seen in 3.2.1.

In this section we will analyze the main error recovery techniques including those concerning the OS intervention even if they will not be used in this work.
We will distinguish between Forward Error Recovery (FER) techniques and Backward Error Recovery (BER) techniques that differ for the recovery approach adopted. In FER, the execution of the operation can continue after an error is detected while in BER the machine is returned to a previous well known (and fault-free) saved state called checkpoint and the operation is performed again.

3.3.1 Reboot

Reboot is a brute force technique adopted when the machine encounter a serious problem and so it is reverted to its initial state, and execution is restarted.
This is an effective recovery solution for transient errors, if the latency of restarting is not critical, but for permanent faults, however, this mechanism may be ineffective since the system may encounter the same error again.

3.3.2 Forward Error Recovery

Forward error recovery allows a system to continue the operations from its current state once the system detects a fault. In this section four styles of FER schemes are discussed: failover, DMR, TMR, and pair-and-spare systems.

- failover systems: this solution basically consist in duplication of the operating unit in order to obtain a standby unit ready to start operation when on the primary one an error has been detected. In this way the process execution is guaranteed without hard interruption with only a small delay due to switching between the two units.
  This is the very initial FER idea and was used in many cases such as in Tandem Computer Systems [8] or in Marathon $Endurance^{TM}$4000 Server [12] were the two processors are also separated by 1.5km in order to avoid terrorist attacks.

  Today failover systems are obsolete because of the limited errors they can recover from. A failover system can correct a permanent error just once discarding the faulty replica and can recover from transient errors with some limitation. In fact if the transient error corrupt the state of the machine than the execution can't proceed even in the fault free

replica because the state of the machine is wrong and so a reboot is still necessary.

- DMR systems with recovery: as already described in 3.2.1 in the paragraph dedicated to DMR, this technique is capable to detect an error thanks to duplication of the UTP, but alone it cannot provide any possibility of fault recovery. However, if other error detection techniques are used inside the two replicas, the recovery become possible.
  Looking at Figure 3.2.1, the ideal flow is the following:

  1. the same process is executed on the two replicas;
  2. error affect the execution on one replica;
  3. the comparator of the output of the two replicas detects a mismatch and so an error in one of the two;
  4. if no other technique are used inside replicas no further correction can be provided. But if internally, replicas have some error detection mechanism (such as arithmetic codes) the faulty replica will fire an error signal to the outside. In this way the comparator knows which one of the two replica is problematic.
  5. Now the system can copy in the faulty replica the correct state of the non-faulty one and everything will continue correctly.

  Obviously with this technique we can only recover from single errors on just one of the two replicas while we can detect errors on both the two replicas since we assume that the two replicas have some internal error detection mechanism.

- TMR systems: as already described in 3.2.1 in the paragraph dedicated to TMR, this technique is capable to detect an error thanks to triplication of the UTP.
  Having three copies of the same unit is the minimum to ensure a simple error recovery mechanism. In fact if one unit suffers from any faults the voter will provide the correct output from the other two fault-free replicas. Then, as in DMR, the state of the correct replicas have to be copied in the faulty one in order to have again a full TMR. Unfortunately, if the error is permanent, the previous copying approach is ineffective because the faulty unit will get again the same error. In this case the system will continue as a degraded DMR and all the specifications described in previous paragraph about DMR are valid.

Another problem related to TMR (but also to DMR and in general to all the redundant modular techniques) is that during copies of correct state into the faulty unit, the (T)MR system can be unavailable. This delay can be prohibitive and really annoying for a real time system or on the contrary it can be accepted.

In Hewlett-Packard NSAA (NonStop Advanced Architecture [11]) two solution were adopted to avoid this problem: the first was to provide a fast and direct link for reintegration and the second was to let the system continue the workflow during reintegration itself.

- Pair-and-spare systems: This technique is an hybrid between DMR and failover systems. As depicted in Figure 3.3.1 it consist in couples of units that work in parallel and one N_to_2 switch that select the right couple among the available couples. In each moment there is an active couple simply called pair and an auxiliary couple called spare pair.

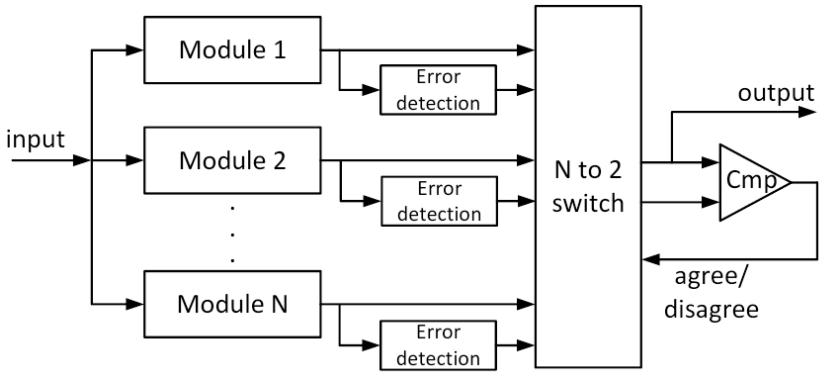

Figure 3.3.1: Implementation of Pair and Spare [45]

Each pair operates as in DMR with a final comparator; if there is an output mismatch the switch commute on the spare pair and the process is continued there. This is a FER technique because the spare pair doesn't start from the beginning of the process but exactly from the moment just before the fault occurrence. In fact the state of the main pair is copied into the spare pair to ensure the context continuity. A real application of this technique is in Stratur XA/R Series 300 [41].

3.3.3 Backward Error Recovery

With BER we can restore a previous state of the system that has been declared safe and stable, and then we can try to go on with the operations as if nothing had happened. This technique can be really effective but surely it may require lot of time and energy, depending on the method adopted to save the state. In other words, there is a specific policy on what state to save, where and when save it, and when to deallocate it. In addition to that, the designer has to manage the restoring mechanism and has to plan what the system has to do after the recovery.

The first problem related to BER is the so called *Output Commit Problem* [31]. It occurs when the detection of the error happens when the error is already out of the recoverable region. It means that if we want to recover from a fault with BER technique we have to detect the error before it goes out of the sphere of recoverability. With the FER technique this was not a problem since errors were detected and corrected simultaneously, but now, with BER there is the possibility of this bad situation. For example, any peripheral such as the screen, the speakers or a printer, are surely out of the sphere of recoverability and therefore errors must be detected before propagating to them in order to apply BER.

If the BER target is the core, as in our case, the error should not propagates to the memory hierarchy, and if the memory is included too the error should not propagates to the peripherals as explained above; the choice of the sphere of recoverability is discussed by Gold et al. [38]

The common approach to the *Output Commit Problem* is to wait for the completion of the error detection mechanism before sending the data out of the sphere of recoverability, but this can surely affect the error-free performances since the delay to detect the error is places on the critical path. Fortunately this degradation occurs only if there are output operations, while, if there is nothing to be send out of the sphere, the waiting time is parallel to the working time for error-free operations.

Now we are going to discuss briefly the choice of the state to save and the choice of the saving mechanism.

To answer the question "What state must be saved for the recovery point?", we have to list the main characteristics of the ideal recovery state:

1. the state has to be error free

2. the state has to be consistent with the current faulty operation

3. the state must allow the resuming of the execution

Therefore, can be really difficult to find a mechanism to save the state and update it according to the three mentioned points, because the actual system configuration can be really complicated (e.g. multi-processor/multi-memory systems).

The typical algorithms used for saving the *recovery point* are checkpointing and logging. The first is the mechanism that periodically saves the state of the machine. The "period" between checkpoints may be fixed or variable in response to certain events and this has impact on the effectiveness of the solution. For example the more frequent we take checkpoints the greater is the performance penalty of checkpointing, but at the same time the lower is the amount of work that must be recomputed after the recovery. With the *logging*, we records the changes of the machine and when an error is detected the log sequence can be unrolled to the recovery point.

At the end, since this two mechanism have both their advantages and disadvantages, sometimes they are used in a hybrid solutions.

3.4 Importance of fault tolerance

The reason why in the last 30 years the fault tolerance has became an important property of a system is because number of faults is increasing. This is due in particular to miniaturization which leads to increasingly sensitive devices. For example, a smaller RAM cell has lower critical charge and an high energy particle from the environment can more easily flip the value of the cell. This is clear from Shivakumar et al. [80] analysis where they showed how transient errors will increase year by years because of scaling of transistors dimensions. But this is true also for permanent faults because of photolithography challenges: for example a smaller device will require more precise fabrication processes and the probability of a mask misalignment may be higher.

As derived from IBM experiments in the early 1980s [93] to measure the particle flux from cosmic rays, particles of lower energy occur far more frequently than particles of higher energy. In particular, a one order of magnitude difference in energy can correspond to a two orders of magnitude larger flux for the lower energy particles. As CMOS device sizes decrease, they are more easily affected by these lower energy particles, potentially leading to a much higher rate of soft errors.

Another important problem is temperature because smaller devices with more transistor will consume more energy and so the power density will increase as well as the temperature of the chip. It's clear that handling higher temperatures in a package is always something problematic because several phenomena are exacerbated with higher temperature [86].
Transistor density, is surely a problem that leads to an increasing in power density but it is also clear that the probability of faults will increase with increasing in transistor density. Infact more transistors per unit of area have a lower chance to function correctly with respect to a lower number of transistors in the same unit of area, because of more interconnections and more complicated designs.

All these considerations just to remark the importance of the effort that is made on fault tolerance for safety critical devices and we could surely say for all those devices that interact in some way with humans.

Chapter 4

The RISC-V ISA and cv32e40p

RISC-V is the center of this project. It is an open source ISA much valued in this years because of its simplicity and effectiveness for both academic and commercial applications. We will briefly describe this ISA and then go into detail on our particular implementation, the cv232e40p core developed by PULP group and OpenHW group.

4.1 The RISC-V ISA

An Instruction Set Architecture (ISA) is the abstract representation of a processor, the set of rules that describe the functioning of the machine. The practical realization, the so called microarchitecture is the implementation of the ISA. This distinction is the reason way two completely different cores can have the same ISA.

RISC-V is an open source ISA developed at the Berkeley University from 2010 with the aim to provide a simple and effective ISA suitable for both academic and industrial purposes. Today it is en emerging ISA because of the possibility to reduce design cost. Infact new companies that decided to move to RISC-V could start from open projects and go on with their custom ISA extension with therefore a minimum cost.
As stated in the *The RISC-V Instruction Set Manual - Volume I: Unprivileged ISA* [19], the ISA avoids "over-architecting" for a particular microarchitecture style (microcoded, in-order, decoupled, out-of-order) or implementation technology (ASIC, FPGA), but allows efficient implementation in any of these. Furthermore, the ISA is separated into a small base integer ISA in both 32 bit and 64 bit versions (RV32(64)I), usable by itself as a base

for customized accelerators or for academic purposes, and optional standard extensions, to support general purpose software development.

The great interest on RISC-V comes from the fact that it is a *free* and *open* standard which allows anyone to easily develop their ideas. Furthermore RISC-V is *modular*, so it is based on a strong and *frozen* set, and all the extensions are optional without the need for continuous update as happens in *incremental* ISAs like the 80x86.

RISC-V International provides the standard specification while the practical hardware implementation is left to external designers together with the the RISC-V software.
For a more detailed description look at the two instruction set manuals at [18] and [19].

4.1.1 History

[72] RISC-V project started in 2010 by Prof. Krste Asanović and graduate students Yunsup Lee and Andrew Waterman as part of the Parallel Computing Laboratory (Par Lab) at University of California Berkeley, directed by Prof. David Patterson.
The Par Lab was a five-year project to advance parallel computing funded by Intel and Microsoft for 10M over 5 years, from 2008 to 2013, but it also received founding from other companies and the State of California. All the projects in the Par Lab were open source using the Berkeley Software Distribution (BSD) license, including RISC-V and Chisel, the hardware description language that was used to design many RISC-V processors. You can find the first publication that describes the RISC-V instruction set in The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA [2] by the Par Lab.
Important milestone in RISC-V life are:

1. first RISC-V chip in 25nm FDSOI donated by ST Microelectronics;
2. publication of a paper on the benefits of open ISA in 2014 [65];
3. first RISC-V Workshop in January 2015;
4. start of RISC-V Foundation in 2015 with 36 Founding Members, with the aim to own, maintain, and publish RISC-V intellectual property;

5. RISC-V Foundation start collaboration with the Linux Foundation in November 2018;

6. RISC-V International Association is incorporated in Switzerland in March 2020.

4.1.2 Base Integer ISA

The basic architecture of RISC-V supports integer operations such us integer arithmetic or memory access and several other instructions that involve operations on registers. The reason of such a simple ISA is the need of a basic starting point, a minimal set of instructions, to provide a reasonable target for toolchains and software environments. Then developers can extend the ISA with almost any kind of additional instruction sets to fully customize their RISC-V implementation.

The two main integer variant of the ISA are the so called RV32I and RV64I that stand for RiscV 32(64)bit Integer and they are very similar to each other with differences only in the width of the registers, the size of the address space and other minor things. There are other two less used variants that are RV128I that is simply an extension of the RV64I in terms of address space and instruction formats and RV32E designed as a reduction of RV32I for embedded systems.

Now we will go into details for the RV32I which is the base ISA of the RISC-V implementation used in this work. RV32I includes 47 instructions that are encoded with 6 possible formats (R,I,S,U,(S)B,(U)J) as shown in Figure 4.1.1.

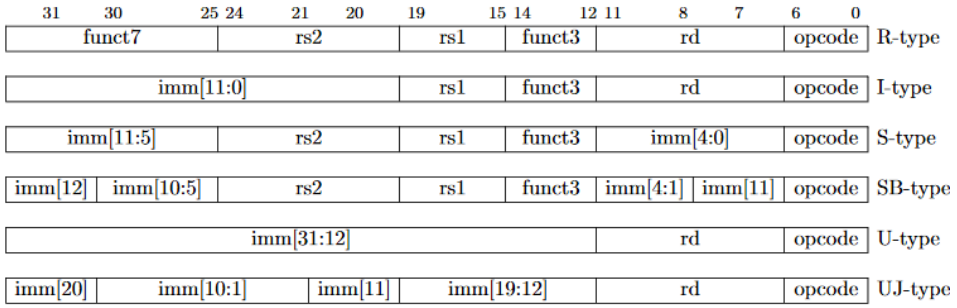

Figure 4.1.1: RISC-V base instruction formats [2]

These formats try to simplify the decoding keeping the source (rs1 and rs2) and destination (rd) registers always at the same positions while immediates are packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always the MSB of the instruction to speed sign-extension circuitry.

Anyway, despite the specific format used in the core we can identify four main subsets of instructions:

1. Integer Computational Instructions:

   They are encoded in R-type format or I-type format respectively if we have a register-register operation or a register-immediate operation. We have addition (ADD, ADDI), subtraction (SUB), shifts (SLL, SLLI, SRL, SRLI, SRA, SRAI), bitwise logical operations (AND, ANDI, OR, ORI, XOR, XORI) and comparison operations (SLT, SLTI, SLTU, SLTIU). Then we can also include two additional instructions: AUIPC (add upper immediate to pc) used to build the pc-relative address and LUI (load upper immediate) used to build 32-bit constants, both encoded with U-type format. There is one last instruction in this set, the no-operation (NOP) actually encoded as ADDI $x0,\ x0,\ 0$.

2. Control Transfer Instructions:

   These instructions are responsible for jumps to other instructions in the range of ± 4 KiB that break the normal flow of the program.
   Jump And Link (JAL) and Jump And Link Integer (JALR) are dedicated to unconditional jumps and are encoded respectively with UJ-type and I-type formats. Then we have all the other instructions responsible for conditional jumps (branches) that are all encoded with SB-type format. These instructions produce a comparison between two registers and an action is performed based on the result.
   Branch if Equal (BEQ), Branch if Lower Than (BLT), Branch if Lower Than Unsigned (BLTU), Branch if Not Equal (BNE), Branch if Greater or Equal (BGE), Branch if Greater or Equal Unsigned (BGEU). Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

3. Load and Store Instructions:

   These instructions are the only few simple instructions responsible for memory access. As RISC-V is a load-store architecture the only memory related operations are loading a value from memory to a register

and storing a value from a register to the memory.
Register to Memory instructions are those of the STORE family (SW, SH, SB) that store respectively a Word (32b), Half a word (16b) and a Byte (8b). The instructions responsible for Memory to Register transfers are those of LOAD family (LW, LH, LHU, LB, LBU), where the *U* stands for Unsigned as the Half (16b) and Quarter (8b) words are zero-extended instead of sign-extended as in the non-*U* version.

4. Control and Status Register Instructions:

   These are all SYSTEM instructions, encoded using I-type format. Control and Status Register atomic Read/Write (CSRRW) is used to atomically swap values in the CSRs and integer registers; atomic Read and Set/Clear bits (CSRRS, CSRRC) are used to read the value of a CSR and then set or clear specific bits of the read word. The corresponding immediate versions of the three previous instructions, CSRRWI, CSRRCI, and CSRRSI, behave like their counterparts but they use an immediate instead of an integer register.

In addition to the previous instructions there are other four particular instructions:

1. FENCE: guarantees any specific ordering between memory operations from different RISC-V threads;

2. FENCE.I: guarantees that stores to instruction memory will be visible to instruction fetches on the same thread;

3. ECALL: makes a request to the supporting execution environment (e.g. the OS);

4. EBREAK: used by the debugger to cause control to be transferred back to a debugging environment.
   Note ECALL and EBREAK are both SYSTEM instructions as those related to CSRs.

4.1.3 Standard extensions

In the previous paragraph we have briefly described the base integer ISA of RISC-V but as known the most important feature of RISC-V is the possibility to easily add other custom instructions to enlarge the ISA and adapt it to specific objectives. Despite these "non-standard" extensions, there are several other standard ISA extensions that have been developed during years

and can be added very easily to our particular implementation making the final RISC-V a very powerful core, and they are:

- "M" Standard Extension for Integer Multiplication and Division;
- "A" Standard Extension for Atomic Instructions;
- "F" Standard Extension for Single-Precision Floating-Point;
- "D" Standard Extension for Double-Precision Floating-Point;
- "Q" Standard Extension for Quad-Precision Floating-Point;
- "L" Standard Extension for Decimal Floating-Point;
- "C" Standard Extension for Compressed Instructions;
- "B" Standard Extension for Bit Manipulation;
- "J" Standard Extension for Dynamically Translated Languages;
- "T" Standard Extension for Transactional Memory;
- "P" Standard Extension for Packed-SIMD Instructions;
- "V" Standard Extension for Vector Operations;
- "N" Standard Extension for User-Level Interrupts;

Among the previous ISA extensions, only "M","A","F","D","Q" and "C" are frozen, that is they are completed and delivered, while the others are still ongoing or are just proposed.

4.2 The cv32e40p core

cv32e40p is the RISC-V implementation that is analyzed in this paper and the fault tolerant optimization will be performed on its Execution Unit (EX unit).
As of today, the core is maintained [20] and documented [22] by OpenHW Group. In this paper the core is described in a less detail with respect the official documentation, and particular focus is put only on the EX Unit because we will try to optimize it from a fault tolerant point of view.
The core was born as part of the PULP project from OpenRISC OR10N CPU then moving to RISC-V (2016) as RI5CY. In February 2020, finally it

has been contributed to OpenHW Group with the name cv32e40p.
The cv32e40p version used in this work is close to the last RI5CY release except for the changes in the verification environment because OpenHW Group has spent lot of effort in building a solid environment for the verification [69] [17].

cv32e40p is a simple 32-bit, in-order RISC-V core with a 4-stage pipeline that implements the RV32IM[F]C ISA, and the Xpulp custom extensions in order to achieve higher code density, performance, and energy efficiency [1] [37] [29]. Infact the PULP project (Parallel Ultra Low Power) was born as a joint effort between ETH University of Zurich and University of Bologna, with the main goal to develop ultra low power platforms suitable for energy-efficient computing.
With the name *RV32IM[F]C ISA* we means that the supported instruction sets are:

1. I: Base Integer Instruction Set, 32-bit;
2. M: Standard Extension for Integer Multiplication and Division;
3. C: Standard Extension for Compressed Instructions.
4. optional F: Standard Extension for Single-Precision Floating-Point. Note that cv32e40p Floating Point unit is maintained by PULP platform in a separate project [24].

The core also supports the following instruction sets although they are not mentioned in the name:

1. Zicount: Performance Counters;
2. Zicsr: Control and Status Register Instructions;
3. Zifencei: Instruction-Fetch Fence.

The core support also XPULP custom extensions from PULP group:

1. Post-incrementing load and store;
2. Multiply-Accumulate (MAC) extensions;
3. ALU extensions;
4. Hardware Loops.

4.2.1 Design

In Figure 4.2.1 is reported the cv32e40p basic scheme:

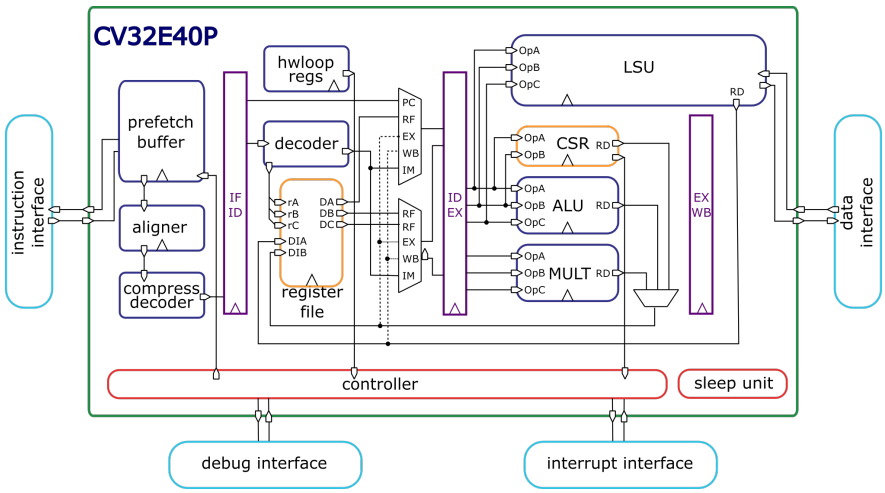

Figure 4.2.1: cv32e40p microarchitectural scheme [25]

In this paper are taken into consideration the main components of the EX Stage: the ALU that contains the divider and the Multiplier that includes MAC unit and Dot Product support (Dotp).
We will focus our efforts to introduce fault tolerance into the core right on its execution unit, but surely, in order to have a full fault tolerant processor also the other parts of the core must be protected, and this is done in parallel to this work by other two teammates of the project.

4.2.1.1 Execution unit

The execution unit is the main computational unit of the architecture. It is made of the ALU, the Multiplier and the additional processing unit (APU) or the Floating Point Unit (FPU).

ALU

The cv32e40p ALU supports standard instructions as well as some extensions made to increase the efficiency of the processor.
Here we will introduce the main features of the extensions provided to the standard RISC-V ISA regarding the ALU operations:

- General ALU instructions:

These instructions try to fuse common used sequences of instructions into a single one and thereby attempting to increase the performance of programs that use those sequences. For example to perform the operation to find the minimum between two numbers there is a specific instructions called (min) that acts like this:

$$min\ r_D, rs_1, rs_2 - - > r_D = rs_1 < rs_2\ ?\ rs_1 : rs_2$$

(Note: Comparison is signed).

Furthermore some new fixed point instructions have also been supported (with all their variants):
- p.add[R]N (addition with round and norm);
- p.sub[R]N (subtraction with round and norm);
- p.clip (check if a number is between two values and saturates the result to a minimum or maximum bound otherwise)

- Packed-SIMD [3]:

  The SIMD instructions (Single Instruction Multiple Data), also called micro SIMD, perform operations on multiple sub-word elements at the same time, in two flavors:

  1. 8-Bit, to perform four simultaneous operations on the 4 bytes inside the 32-bit word (.b);
  2. 16-Bit, to perform two operations on the 2 half-words inside a 32-bit word at the same time (.h).

  Regarding the second operand, we can divide the SIMD instructions into three different sets:

  1. Normal mode, vector-vector operation: operands, from registers rs1 and rs2, are treated as vectors of bytes or half-words.
  2. Scalar replication mode (.sc), vector-scalar operation: operand in register rs1 is treated as a vector, while operand 2 is treated as a scalar and replicated two or four times to form a complete vector.
  3. Immediate scalar replication mode (.sci), vector-scalar operation: operand in register rs1 is treated as vector, while operand 2 is treated as a scalar and comes from an immediate. The immediate is either sign- or zero-extended, depending on the operation. If not specified, the immediate is sign-extended.

All the operations are rounded to the specified bidwidth as for the original RISC-V arithmetic operations.

The supported instructions are the usual instructions for a standard ALU such as *add*, *sub*, *avg* and others, just in the SIMD versions.

- Bit manipulation instructions:

  These instructions are useful to work on single bits or groups of bits within a word because sometimes we need to access a subpart of the word, e.g. when we want to access a configuration bit of a memory mapped register. The main new instructions are: extract/insert (read/write-to a register set of bits), bclr, bset (clear/set a set of bits), cnt (count number of bits that are 1), ff1, fl1 (find index of first/last bit that is 1 in a register), clb (count leading bits in a register), and bitrev (reverse bits in groupings of 1, 2 or 3 bits).

- Iterative Divider:

  The divisions is actually implemented reusing existing comparators, shifters and adders of the ALU through a long integer division algorithm. This approach is surely slower than having a dedicated full parallel divider but saves a lot of area.

Multiplier

The multiplier can perform different kind of multiplications as shown in Figure 4.2.2 [37]:

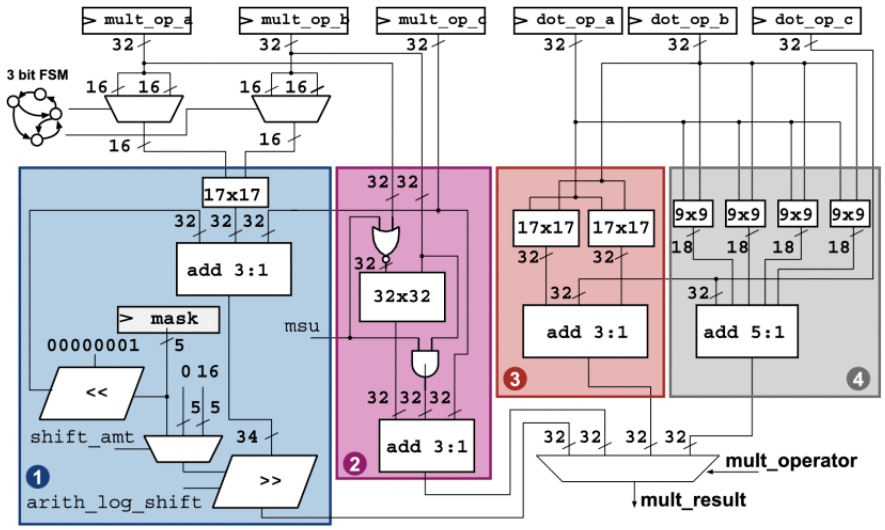

Figure 4.2.2: Multiplier block diagram

- 16bx16b fractional fixed point multiplication and multiply and accumulate: multiply two signed or unsigned numbers extended on 17 bits and eventually add a 32b number (mult_op_c).
  Examples [5]:
  -p.muls: LSP (Least Significant Part) short int by LSP short int into int multiplication.
  -p.mulsRN: LSP short int by LSP short int into int multiplication followed by rounding and then normalization (immediate value);
  -p.mulshh: MSP (Most Significant Part) Short int by MSP short int into int multiplication.
  -p.machhsRN: MSP short int by MSP short int into int and accumulate. Then rounding followed by normalization.

- 32b×32b integer multiplication and multiply-accumulate: is the classical 32bx32b integer multiplication with the additional add of the 32b mult_op_c.

- 16b dot-product (dotp): 16b×16b + 16b×16b: see next description.

- 8b dot-product (dotp): 8b×8b + 8b×8b + 8b×8b + 8b×8b: multiply two vectors and sum a 32b value in one clock cycle. The vector can be made of 2 16b elements or 4 8b elements. To perform signed multiplication the MSB is extended leading to 17b and 9b elements.
  In region 3 and 4 of Figure 4.2.2 there is a simple description of these

two dotp version. There are enough multiplier to perform the complete set of multiplication all at once. Then a compression tree is adopted to obtain the final result. In both cases an additional 32b element (the accumulation register) can be added to the dotp result, and it is done inside the adder compression tree.

The design of the Multiplier with the support for vectorial operations was more involved than the design of the modified ALU. As in Figure 4.2.2 we have seen four different modules than allow to perform any sort of multiplication related operation: a 32bx32b integer multiplier, a fractional multiplier and two dot-product multipliers. Its architecture is surely more complex than the one of the ALU and for this reason we will make different considerations (in terms of fault tolerance) on the Multiplier compared to the ALU.

FPU [23]:

The FPU is maintained into a separate project [24] from that of the cv32e40p core.

In the top-level file cv32e40p_core.sv we can set the parameter FPU enabling the RV32F ISA extension that support single precision floating point operations in the form of IEEE-754. This will extend the cv32e40p decoder accordingly and it will extend the ALU to support the floating-point comparisons and classifications. The Floating Point Unit is instantiated outside the cv32e40p and is accessed via the APU interface. By default a dedicated register file consisting of 32 floating-point registers, f0-f31, is instantiated.

The FPU is divided into three parts:

- A basic FPU of 10kGE (10^3 Gate Equivalent) complexity, which computes FP-ADD, FP-SUB and FP-casts.
- An iterative FP-DIV/SQRT unit of 7 kGE complexity, which computes FP-DIV/SQRT operations.
- An FP-FMA unit which takes care of all fused operations. This unit is currently only supported through a Synopsys Design Ware instantiation, or a Xilinx block for FPGA targets.

We will not go into details for this unit since we will discard it for our fault tolerance reasoning.

Chapter 5

Fault tolerance on cv32e40p

This is the central chapter of our work as we will describe the workflow to get a certain fault tolerance on cv32e40p RISC-V core.
First of all we will introduce the objectives of the fault tolerance analysis of this project, then we will describe the ideas behind the practical realization of the work and at the end the architectural solutions are presented, with main focus on ALU and Multiplier.

5.1 Target

Since there are no particular performance specifications for this work, our aim is just to obtain a working and secure version of the cv32e40p RISC-V core with minimum area overhead trying not to affect performances. At the time of writing this report, cv32e40p does not have a specific fault tolerance extension, so the goal is to investigate different solutions to achieve a good level of fault tolerance.

The classic three-point target power-area-delay, is therefore completely free. In this work we will add a fourth vertex to the triangle: the fault-tolerance vertex as in Figure 5.1.1. We will strive for all the vertex but in particular we will take into consideration those of fault-tolerance and delay.

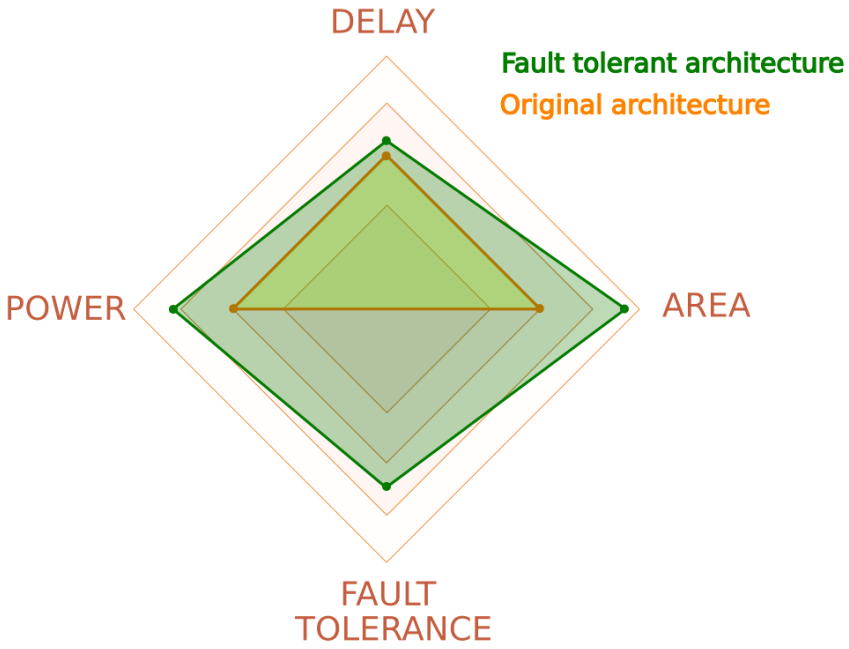

Figure 5.1.1: Diagram of performances

Among the multitude of techniques described in paragraph 3, we will focus on those related to modular redundancy because they are the best in terms of speed and recovery capability. In particular, since we want to achieve good timing performances, all time redundant technique are unacceptable. If for example we use a standard REDO approach in the full pipeline the additional delay will be of a single "clock cycle" instead of the delay of the complete forward path. However, we decided to add the Fault Tolerance at a fine grain in order to reach a finer detection capability and for this reason all the known time redundancy technique are unsuitable. In fact if for example in the EX unit the ALU use a REDO (or RERO or RESO) approach all the pipe needs to be stalled waiting for the completion of the recomputation with an unacceptable loosing of performances.
Another reason to use Modular Redundancy techniques is the forward recovery capability: with Repetition techniques (REDO, RESO, RERO, ..), BIST or even AR-SMT there is always the need of checkpointing, the mechanism of saving current stable state of the machine to restore it if necessary. For this reason we will privilege forward error recovery techniques that avoid checkpointing and backward restoring.

In this work we will focus on EX unit of the pipeline which is the unit dedicated to computations as described in the previous paragraph. EX unit is therefore a very important part of the core because a faulty computation can have catastrophic effects and in terms of cost the computations are in general avid energy thief therefore a great effort must be made to optimize this unit.

5.2 Ideas

The investigated technique for the EX unit of cv32e40p core is strongly related to TMR and in particular we will study standby sparing (similar to pair and spare). The dual level TMR, as described in paragraph 3 in the section related to spatial redundancy, could surely also be applied and would lead to a greater safety level. However we have decided to discard this technique because of the unnecessary extra complexity that would be introduced if the errors we will target are single transient or permanent errors because it is far more unlikely to have an error in one voter than in the ALU, for example.

For these reasons two are the main ideas we will investigate to identify our final approach:

1. Hybrid stand-by sparing: leave an extra unit (full precision replica) in stand-by (guarded evaluation [89] or clock gating) and use it if one of the main three goes down. For transient errors there is no need of the fourth replica but if the error is permanent having an extra module can allow to maintain the TMR architecture.

2. A good idea may be to try to use the faulty replica in the still valid subparts anyway. It means that after the substitution of the faulty replica with the spare one, the faulty replica can be self-tested against well known inputs to understand where it fails, or it can be immediately marked as partially faulty if we are able to know which sub-part gave the error. In this way it can still be used for those operations that are correctly supported. So, if the new TMR will experiment another permanent fault the ex-faulty replica can be a valid substitute for its still supported operations.

The above introduced approach is unuseful for transient faults because in case of a transient problem the faulty unit will again give a correct result

after one cycle, so the fourth replica is unuseful. For permanent errors on the other side it is a very good approach because allow the core to tolerate potentially lots of permanent fault if they are confined to a sub-part of the protected component.

5.3 Criticality levels

Looking at the schematic in Figure 4.2.1 we have a simple idea of the inside of cv32e40p EX unit: the ALU, the Multiplier and even the FP unit (not reported in the figure because it is optional) and the related APU dispatcher for the Additional Processing Unit such as the FPU itself.

In this work we will target the ALU which includes also the Divider, and the MULT which includes the DotP unit for Dot-Product together with the standard multiplication unit (see Figure 4.2.2).
The implemented architecture reflects the ideas presented above, with TMR, standby sparing and other solutions mixed together; in Figure 5.3.1 there is a general scheme of the hardware extensions provided for the ID and EX units of cv32e40p.

It is immediately clear how the ALU and the MULT are treated in different way. This came from the consideration of fault criticality levels. Surely this evaluation of fault criticality is not the most accurate we can imagine but it is based on evidence of great difference on the workload for the two components. In a "standard" program running on the core, for example an "Hello world" with printf() and scanf() calls, the ALU is used almost at any cycle while the MULT can be quiet if not called via an explicit multiplication for instance, and even an explicit multiplication in the source code can be translated in sequence of sums and shift (if at least one of the operand is a constant) to avoid the using of multiplier surely really power expensive. The ALU is indeed used for many purposes: to obtain address via immediate, to obtain the correct value of the program counter in case of branches or simply to compute an addition we find on the C code for example.
From this considerations we decided to use a similar but not identical protection approach for ALU and Multiplier. In both cases the errors we will try to tolerate on a single component are: single transient errors and multiple permanent errors. This means that the ALU and the Multiplier will be no more affected by a transient error occurring in one of the replicas thanks to TMR and will not be affected by permanent errors thanks to TMR itself and thanks to the spare unit for the ALU. The amount of permanent errors this technique can protect from depends on the error extent and on the corrupted

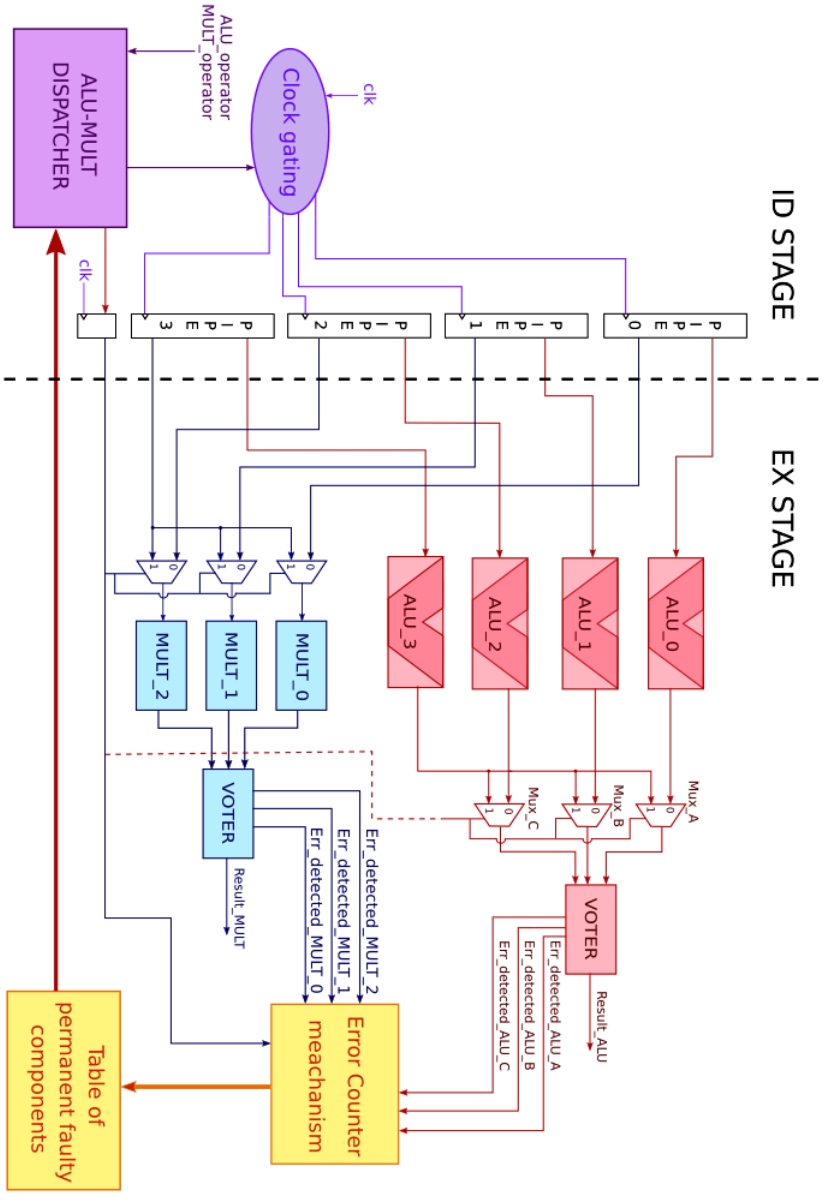

Figure 5.3.1: General schematic of design

part. We will better address this problem in the rest of the discussion, now we will give a general and detailed description of the architecture to have a clear idea of the platform where we will conduct our final tests with faults injection.

5.4 The architecture

In Figure 5.3.1 is represented an high level scheme of the implemented architecture. We will divide the discussion in description of the ALU and description of the Multiplier even if there are common parts because the two units follow a similar approach.

The general idea is to exploit the so called spatial redundancy to achieve TMR, the best trade off in terms of complexity and efficiency for single transient errors. However, time, fabrication defects or external interventions can cause a permanent issue on one of the three replicas used for TMR. Therefore, in order to have a functioning TMR even when a permanent errors occurs, a fourth replicas has been introduced for the ALU, the so called spare unit (while for the MULT another approach has been adopted which consists in exploiting the ALU to perform a multiplication by translating it into a sequence of sums and shifts).
The extra fourth unit (or the translating mechanism for the MULT) can replace the permanent faulty unit if there is a mechanism to detect permanent errors. For this reason each ALU and MULT replica is associated to a set of α-counters which count the number of errors during program execution and mark a component as permanent faulty if the associated counter reach a defined threshold (see subsection 5.4.3). To keep track of the faulty units we use a "table", actually a set of FFs, set to '1' if the corresponding component has been considered "broken".
Even if we have only 4 ALUs, the counters are 36 because each ALU has been divided in 9 sub-parts in terms of common executed operations. This means that among 58 opcodes associated to ALU usage we can identify set of opcodes and so set of operations associated to the same physical component inside the ALU. Therefore, if one of this component became faulty all the associated instructions that would exploit that component may no longer be supported in that ALU. For example imagine that for several contiguous cycles the ALU has to perform a comparison, say the ALU_LTS (lower than unsigned) which obviously uses the comparator mechanism inside the ALU, and that for each computation the results in incorrect. Now we can imagine

that the other operations that involves the comparison mechanism may also be incorrect and for this reason we state that all that operations are no more supported in that ALU. At the same time there is still the possibility to use that ALU for the remaining non faulty subparts.
A similar approach has been adopted for the MULT with only three replicas and with the addition of the above mentioned mechanism to translate a multiplication in a sequence of sum and shifts.
In the next pages we will go inside each high level block in Figure 5.3.1 to discover the specific implementation and the solutions adopted to optimize the system.

An remarkable point is about the customizability level of the HW extension provided to the EX unit of cv32e40p for the fault tolerance. Infact the idea is to provide the possibility to activate or deactivate the fault tolerant extension in order to give the user an additional freedom degree. The user can enable or disable the whole supplied mechanism for the fault tolerance at pre-synthesis time, since the same core can be used in environment with a different criticality level.
If we decide not to use the FT extension we have to write '0' in the SystemVerilog parameter *FT* inside the top level module at $cv32e40p\_core.sv$. In this way we go back to use a single ALU and a single MULT avoiding the compilation of all the provided mechanism to manage these extra units, as we will explain in the rest of the paper.
When *FT* is '0' all the extra signals defined and used for fault tolerance reasons are simply set to '0' to be harmless.

Now we will address the ALU HW extensions and then similar reasoning will be done on the MULT.

5.4.1 ALU dispatcher, pipeline replicas and TMR

Despite the targets of this work are the ALU and the Multiplier and so the EX stage, a lot of work is done inside the ID stage of the core because it is actually the source of most of the signals used in the EX stage.

In Figure 5.4.1 there is a more detailed scheme of the HW extension provided to the ID stage to manage the TMR mechanisms in the EX stage.

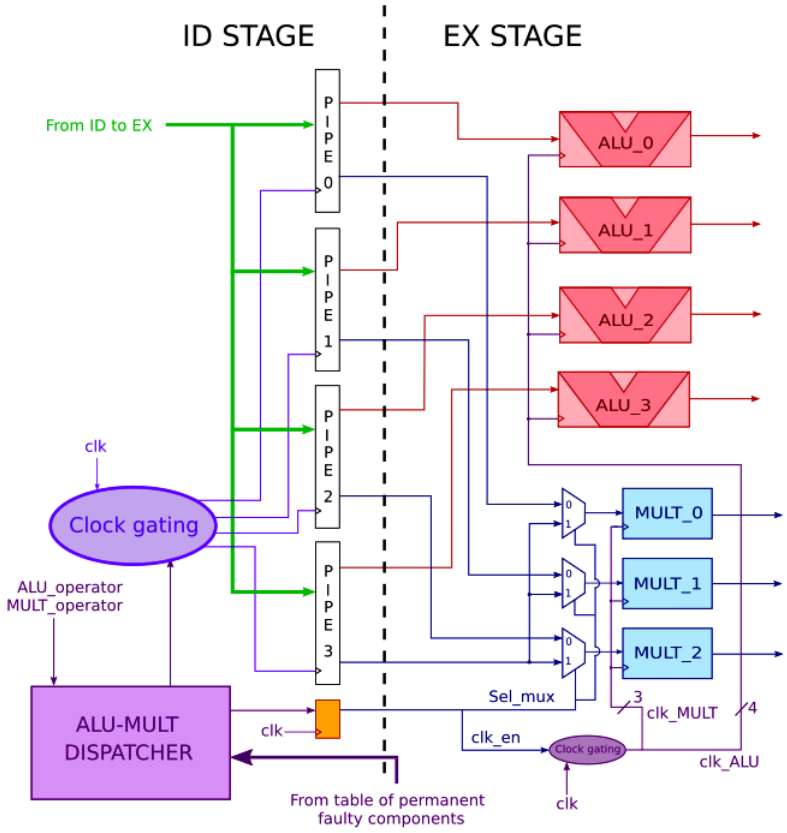

Figure 5.4.1: Zoom on the ID-EX interface

The green rows indicate the signals produced into ID stage that have to be send to the EX stage and therefore those signals coming in the ID-EX pipeline register, interface between the two stages. This big pipeline register is now quadruplicated in order to protect it too from errors as well. Each pipe register receive the same inputs and provide the same outputs with the only difference on the clock they receive because now it comes from a clock gating block controlled by the ALU-MULT dispatcher. If for example the ALU_3 has not to be used for the TMR it will put in stand-by mode through clock gating on the associated pipe register. In this way all inputs to the ALU_3 remains the same thus giving zero switching activity and so zero dynamic power consumption. The Dispatcher has so the role of deciding which of the four ALU has to be the spare one for that clock cycle.
Regarding the Multiplier, even if there is no extra 4th unit if one of the three replicas became permanent faulty it will be put in stand-by mode on

the specific faulty sub unit and the above mentioned translating mechanism will be activated.
There are four pipeline replicas and four ALU replicas so the matching is immediate while for the Multiplier, on the other side, there is the need of an addressing mechanism to choose the right three pipeline registers to feed the three Multipliers. If the ALU and the Multiplier will never be used together into the same cycle this addressing mechanism is unuseful because we can always choose the same set of three pipeline registers to feed the three multipliers. However we can't exclude an ISA extension where ALU and MULT are used together so we decided to provide this mechanism to be compliant with the maximum number of ISA extension.
The Dispatcher receive informations on the status of the ALUs and MULTs and taking into consideration the current opcode the EX stage will receive, decides which ALUs and which MULTs are suitable for that operation. In particular the orange register in Figure 5.4.1 is the one responsible to pass the MUXs selectors to the EX stage and the clock enable used for the clock gating of the pipe registers. The reason why clock enable is pipelined too is because the next cycles after the pipelining the clock gating will be applied to the sequential circuit related to the stand-by unit. This means that if for example the ALU_3 is the spare one in a precise clock cycle, the associated set of counters have to be disabled too. But this will be more clear in 5.4.3 dedicated to the counters.
That clock enable is also used as a redundant protection inside the stand-by ALU or inside the stand-by MULT. In fact, even if these components have already minimized power consumption thanks to clock gating, they contain sequential circuits inside, for example the divider into the ALU. So we decide to do clock gating also on that parts in order to have a second level of protection against unwanted switching in that unit. For example it can happen that some signals after the pipelines switch unexpectedly (because of transient error), so the FSM of the divider inside the ALU can have transitions and lead to undesired outputs. Anyway those possibly undesired output will be stopped by the same mechanism that put that component in stand-by (in fact they will not taken into consideration by the voter), but however the unwanted switching would have already occurred thus consuming power, so the reason of this extra protection.

Regarding the ALUs, since there are operations characterized by the same computations and so by the exploiting of the same physical component inside, they are virtually divided into 9 subparts and the errors will be counted on each of those subparts (for a deeper understanding see 5.4.3).

It means that an ALU can be marked as permanent faulty on one of the 9 subparts it is divided in, thus leaving the possibility to exploit that ALU for the remaining good parts.

Unfortunately the implemented technique is really effective only for single permanent errors or for multiple permanent errors that affect different ALUs not in the same subset of operations. In other words if for example one ALU is permanently faulty for the first set of operations of the 9 it has, and another ALU has the same problem, the dispatcher can't provide a complete set of three ALUs to have a complete TMR for that unlucky operation because only two are suitable for that operation. And the situation become even worse if there are three or even four ALUs permanently wrong.
To manage this situations the dispatcher has to dynamically change the behaviour of the result checking mechanism in the EX stage, i.e. the voter and the counters, to handle all the possible configurations. Therefore this is what the Dispatcher actually do:

- Receive informations about the status of ALUs and MULTs that are stored into the table of permanent faulty components (see 5.4.3) that is actually a set of flops that are set to '1' if the associated components inside one ALU or inside one MULT is faulty.

- Looking at the operation that has to be performed in the EX stage at the next cycle it will decide which ALUs or MULTs are suitable to perform that operation. The dispatcher will try to select set of three ALU and all the three MULTs in order to have a full TMR.

- If there are less than three available replicas suitable for that operations the dispatcher will try to select the higher number of good replicas, thus two or one or zero if there are not suitable replicas.

    1. in case of only two good replicas nothing changes in the counting mechanism because errors will just be counted on the two good replicas. However the voter can no more be used as in the case when there were three supposed correct inputs because now just two of them are supposed to be correct.

       For example suppose that two ALUs are faulty for ADD operation, now we actually choose the other two and one of the faulty ones to have a full TMR. In this case if one of the two has accidentally a fault in the same direction of the chosen faulty unit, the wrong result will pass as correct through the voter and we

will never know about this. To solve this problem, dispatcher provides additional signals to the voters indicating that only two units are non faulty and redirecting their outputs on the first two input of the voter. Thus the voter became a comparator and if it detects a mismatch it will just make the corresponding counters to increment saying that an error has been detected and that it has not been corrected as in the case of three different inputs to the voter.

2. in case of only one good replica the voter can't be used because it will probably say that the output is not valid since it would receive three different inputs and so can't decide which one is prevalent.We have written "probably" because there can be a remote possibility that one faulty unit is faulty in a "correct" way in that cycle.

   Anyway, to solve this problem we added after the voter a 4_to_1 MUX that will provide to the next stage (i.e. the decode stage where is the register file) either the result provided by the voter or directly one of the three inputs to the voter, thus bypassing the voter itself (Figure 5.4.3).

3. in case of no good replica no one of the four replicas of the ID_EX pipeline will be clocked but a special signal will be triggered to report that bad occurrence to the SW (the OS for example).

5.4.2 Voter

The key component of this fault tolerant system is surely the voter which is responsible for comparing the results of the involved computational units in order to provide a correct output. That means the voter receives three inputs and gives the voted output with majority voting approach. In fact the three inputs have all the same confidence and they come from equal replicas of a component so there is no a specific input more reliable than the others. Actually, the developed voter in Figure 5.4.2 has some additional outputs that give extra informations about the status of the three inputs and about the confidence of the output.

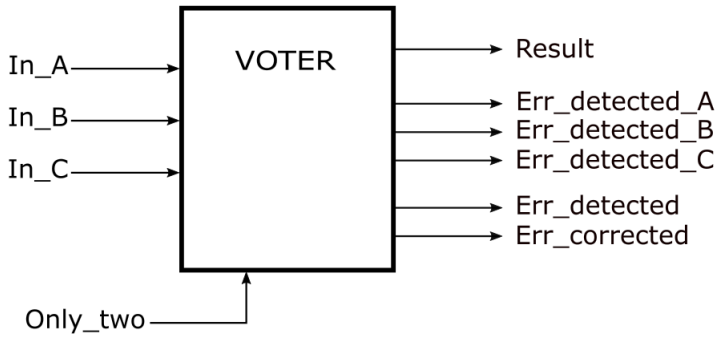

Figure 5.4.2: Majority Voter

In Figure 5.4.2 *Result* is the voted between In_A, In_B and In_C and so it is equal to the majority of inputs or in case all three differ from each other it is set to In_A as a default. Additional information are obtained through $Err\_detected$ and $Err\_corrected$: if at least one error has been detected then $Err\_detected$ is '1' and if only one error has been detected and corrected, $Err\_corrected$ became '1' too. In this way we have all the possible combinations:

1. Zero errors were detected: the output is valid.
   $Err\_detected = 0$;
   $Err\_corrected = 0$;

2. One error was detected and corrected: the output is valid;
   $Err\_detected = 1$;
   $Err\_corrected = 1$;

3. Two errors were detected and not corrected: the output is not valid;
   $Err\_detected = 1$;
   $Err\_corrected = 0$;

4. Three errors were detected and not corrected: the output is not valid;
   $Err\_detected = 1$
   $Err\_corrected = 0$;

The implemented Voter gives other three signals to inform the system about the position of the error among the three inputs: if $Err\_detected\_<i>$ is '1' it means the corresponding i-input is incorrect because it differs from the other two that are equal. It is clear that only one of the three signals can be active at the same time because it can happen that one input is different

from the other two, but in the extreme case when all inputs are different from each other, all the three $Err\_detected\_<i>$ are active at '1'.

In the case of only two available replicas the voter can't be used as a standard voter but became a simple comparator. The dispatcher redirect the output from the two non faulty replicas to the first two input of the voter and the "Only_two" signals inform the voter of the particular situation it had to face.

5.4.3 ALU alpha-counters and table of faulty components

The voter described in the previous paragraph is used to inform the counting mechanism about the occurrence of errors and about their origins; in this way we are able to focus on the component which gives incorrect results monitoring its errors frequency. In Figure 5.4.3 there is a simple scheme of the counting system.
About the ALU, since we have 4 ALUs virtually divided into 9 subsets of operations, we need 36 counters each counting the errors that occur into the associated ALU in the associated subset of operations. The idea is that if one counter reaches a predefined threshold, the associated ALU is marked as permanently faulty for that subset of operations setting to '1' the corresponding flip-flop in the table of permanent faulty components. When the application will require the same operation by the ALU, the dispatcher will select the ALUs suitable for that operations trying to discard the faulty one.
Obviously, as mentioned in the previous paragraph about the dispatcher, if more than one ALU are marked as faulty for that operation (for example say that 2 of the 4 ALUs are faulty), the dispatcher can select only two available ALUs or just one in the extreme case where 3 ALUs are permanent faulty. In these scenarios the provided mechanism is ineffective because we would need higher degree of redundancy to manage these situations. On the other hand, the probability that such a extreme events occur is lower the greater is the number of components we can mark as faulty, or in other words the finer is the granularity at which we apply error detection. We could imagine that if these situations occur it would mean that the processor has encountered serious problems that would therefore require a complete renewal at higher level with respect to the recovery mechanism provided by our fault tolerant architecture.

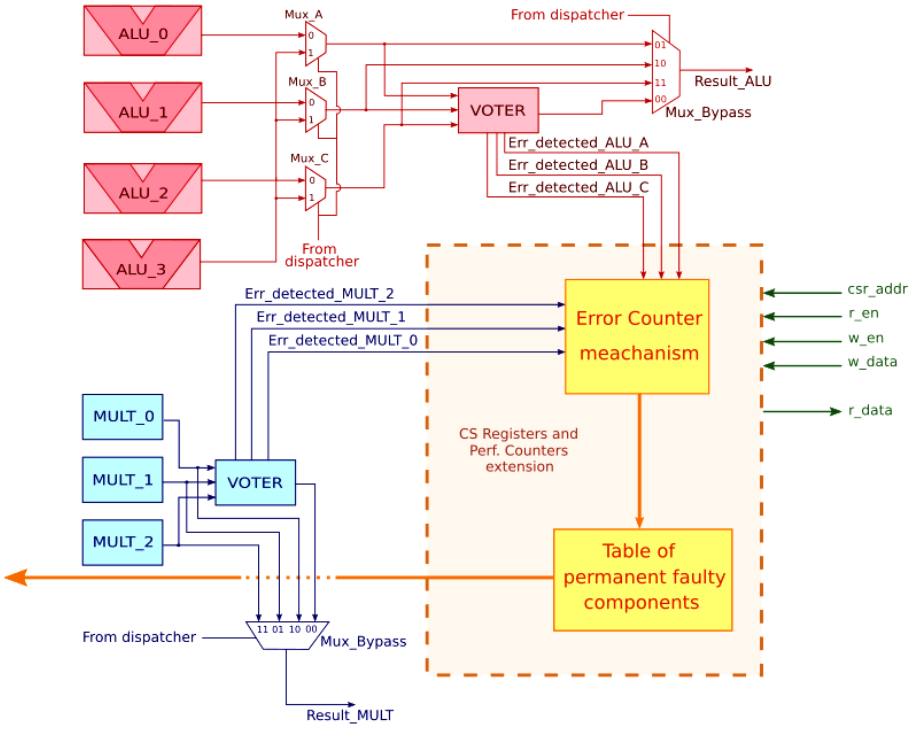

Figure 5.4.3: Zoom on the voting and bypass mechanism

In Figure 5.4.3 in the dashed orange block is depicted the counting mechanism. Actually the counters and the table of permanent faulty components are accessed by EX_STAGE and ID_STAGE but they can be also addressed by the outside with the protocol described by the green signals: csr_addr, r_en, w_en, w_data and r_data.
These signals are managed by a custom extension on Control and Status Registers (CSR) (see 5.5) provided to allow the registers status to be saved when the core is powered off. For example if the core is powered off and then powered on again, we want to maintain the informations on the status of faulty components, so probably we want to reload the counters and the table of faulty components with the old value they had before the shutdown. Therefore the need to save and restore these register treating them as CSRs.

The behaviour of these counters is really simple: if an error is detected by the voter, the counter associated with that component will count up, while if the component provides a correct result, the counter will count down only

if the counter value is greater than 1, otherwise it remains 0.
The up and down counting values are stored in two registers as well as the threshold the counter has to reach to state that the associated component is permanently faulty. The designer can set the values of these three parameters before synthesis updating the package at $cv32e40p\_pkg.sv$.

The choice of these three parameters reflects the hardness of the fault detection mechanism for permanent faults, infact the lower is the threshold the sooner the unit is said to be permanently damaged, or even the higher the ratio between up and down counting values the sooner the counter reaches the threshold. On the other hand, if we want to be sure that a specific unit is really permanently damaged we would like to count lot of errors, so we would set an higher threshold or a lower ratio between up and down counting values. The designer will make his/her choice in order to obtain a more or less aggressive permanent fault detection.

Three issue are related to this counting mechanism:

1. if the operation requires more than one clock cycle, for example a division or a multiplication, the counters must wait for the completion of that operation to increment or decrement. If the counters are timed only with the clock they will try to count at each cycle, thus interpreting the outputs of the voter in the wrong way. Therefore, to solve this problem the counters are actually enable only when $ready\_o$ from ALU or MULT is set. This signal is used to synchronize stages as depicted in Figure 5.4.4

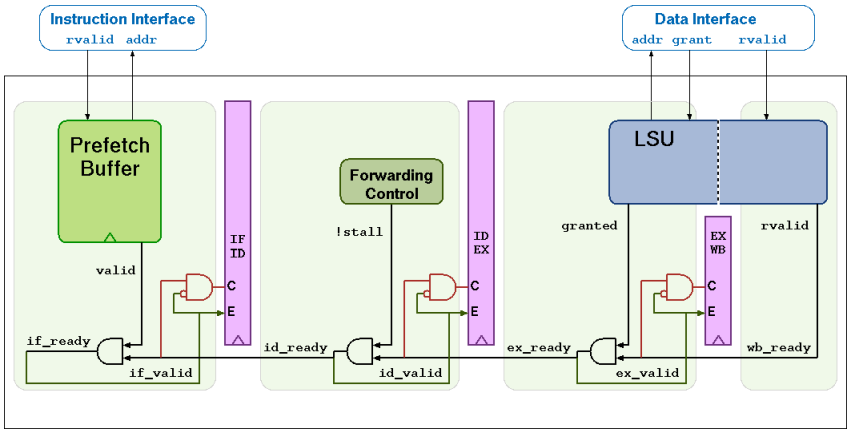

Figure 5.4.4: cv32e40p Pipeline [26]

When *ex_ready* is active the EX stage has completed the operation and so the outputs of the voters are correct and can be read by the counter mechanism in the correct way.

2. the values of the table of faulty components are available exactly two clock cycle after the completion of the operations. Infact when the operation is completed, we have to wait one clock cycle to update the counters and the following one clock cycle to update the set of flip flops in the table based on the value of the counters. However we would like to have the new value of the table as soon as possible, so we output both the value of the input to the FFs of the table the value stored into the table itself. Infact the input to the table is available the clock cycle before it is saved into the table so we are able to save one clock cycle and the dispatcher can do its job one cycle in advance.

3. If there are only two non faulty replicas available for a certain operation or even just one replica, the voter became a simple comparator or it is just bypassed as we have seen, and this implies no error can be detected and therefore counted on those replica. To solve this problem we would like to correct errors also in the unlucky case when TMR become a DMR: a BIST mechanism may be implemented to distinguish between the only two available replicas but this will implies a great overhead in terms of complexity and time consumption (7.1.1).

5.4.4 MULT dispatcher, pipeline replicas and TMR

The MULT protection basically follows the approach used for the ALU but with some differences.
The MULT unit has been triplicated with a final voter as in the ALU solution (Figure 5.4.1). However there is no extra 4^{th} replica because of criticality consideration made in 5.3.
Actually, in an average application, the multiplier is used far less frequently than the ALU and its structural implementation can be really huge. For this reason it may be a bad idea to add another MULT replica if other solution can be applied instead. Infact we want to have the same degree of security and recovery for ALU and MULT but if we don't add the spare MULT unit there will be some unbalanced protection approach between ALU and MULT. Therefore we think about a possible solution that will try to translate a multiplication into a set of sums and shifts. As anticipated in 5.3, gcc compiler does something like this when he translate constant multiplication with add-shift method. Infact it is quite easy for the software to predict the

result if it knows the operands values but when the multiplication operands are variables (so unknown a priori) only the hardware can do something in this direction. See 5.4.6 for further details.

Since there are four pipe replicas and only three MULTs we have to redirect outputs from each replicas to the right MULT. If the ALU is never used together with the MULT we can just assign the first three pipes to the three MULTs for example. Actually we can't exclude further extensions where ALU and MULT units are used together so we decide to provide an addressing mechanism to manage the choice of the right three pipe registers.
The redirecting mechanism is composed by three MUXs (the blue ones in Figure 5.4.1) whose selectors are generated by the dispatcher. These selectors are actually those also used for MUXs after the 4 ALU replicas, just because we have to choose the same three pipe registers both if we have to use only the ALU or if we have to use ALU and MULT together.

As for ALUs, power consumption of each MULT replica is minimized by clock gating. First of all if one MULT is unavailable for a specific operation, the associated pipe register will be clock gated. The second clock gating level is inside each MULT replica itself: the multiplication algorithms sometimes requires more than one clock cycle to complete operation, therefore to check for possible errors through the voter we have to wait until completion of the computation and in the meantime the counters have to be in stand-by mode.

5.4.5 MULT alpha-counters and table of faulty components

The same considerations made for the ALU counters and the ALU table of permanent faulty components are still valid for MULT (see Figure 5.4.3). In particular we have divided the MULT into 4 subparts in order to monitor the internal status of each of them as if they are standalone, so to make the most of each resource. The division into four subparts is clear looking at Figure 4.2.2: 32bx32b integer mult, 16bx16b fractional mult, 16b dot product and 8b dot product.
Since we have divided the MULT into 4 subparts, hence 4 sets of operations, we have 4·3=12 counters, and 12 flip-flops to store the faulty status of each subpart.

As for the ALU, counters and FFs associated to each MULT are mapped to the CSR address space in order to make them readable and writable by instruction in case of processor shutdown and new power-on so that no in-

formation is lost.

Also for the voter of the MULT there is the necessity to manage particularly unlucky situations where just two MULTs are available for the operation or when only one unit is available. In the first case the voter became a comparator with the two good inputs redirected to the first two inputs of the voter, while in the second case the voter is completely discard with the bypass mux, obviously controlled by the same dispatcher.

5.4.6 MULT translated into Add and Shift

In the previous paragraphs we have introduced the idea to decompose a mult into a sequence of sums and shifts when no MULT replica is available.
This approach is also used by the gcc compiler when we have to perform a multiplication by a constant since it can predict the result during compilation, or when the target platform does not have a multiplier. We want to replicate this behaviour for any multiplication that occurs into the core when the multiplier is no more available because it is permanently damaged.
In our core the multiplier architecture is managed by the synthesis tool, for example Synopsys, because the multiplication is described in behavioural way. Anyway we can assume that for example the synthesized multiplier will include shifters and adders because it will use the common shift-add algorithm (actually there are many other architecture that the synthesis tool can choose for the multiplier: fast adder with Dadda [27] or Wallace [91] tree, Baugh-Wooley multiplier [9] and others).

Unfortunately, even assuming that the architecture chosen by the synthesis tool is that of simple sequential multiplication with right shift, if we tried to replicate the same multiplication algorithm using the ALU, we would make an extreme effort and greatly increase the complexity of the circuit, since we will have to design the RTL of the multiplier instead of leaving the burden to the synthesis tool. The problems related to this solution and the great complexity are related to the amount of resources that we will introduce. In fact, we can think of the possibility of using this mechanism as something truly remote, meaningful only when a great damage is experienced on the multiplier. Therefore is better to minimize the resource overhead and to try to exploit in the easiest way what is actually available on the current architecture.
For this reason, it might be good to "simply" decompose the instructions of the *mult* family into a sequence of *add* and $shift$ instructions. Surely the

worst drawback will be the necessity to stall the pipe because of the need to execute possibly a greater amount of instructions instead of a single one. The greatest advantage on the other hand is that we will leave the architecture as it is with the single requirement to create a module that takes care of decomposing the mult instruction.

To explain the architectural idea, we will clarify what is actually this decomposition we have mentioned above. Suppose we have to perform the following multiplication: $x \cdot 14$, we can decompose this simple multiplication by a constant in at least two ways using power of 2 [62]:

1. $x \cdot 14 = x \cdot 16 - x \cdot 2 \rightarrow (x << 4) - (x << 1)$
2. $x \cdot 14 = x \cdot 8 + x \cdot 4 + x \cdot 2 \rightarrow (x << 3) + (x << 2) + (x << 1)$

Multiplying a number for a power of 2 is actually a simple left shift, that is a really simple operation with respect to a multiplication involving the whole multiplier.
One problem may be which of the two possible decompositions to choose. To answer this question we have to look at the binary representation of the previous example: $x \cdot 14 = x \cdot 0..01110$. We can formulate a simple law: looking at the '1' in the higher position, we choose the decomposition with the subtraction if the number of '0's is lower than the number of '1's in the other bits at lower positions. In fact number of '1' is related to the number of shifts we have to perform, so if we have many '1's it means that we have to add as many elements computed with shifts.

A very power saving technique to perform the decomposition might be to share common subexpressions [62] [66]. If for example we have to perform a dot product operation where it is adopted the multiply and accumulate approach, if two operands have some '1's at the same position it means that the decomposition based on powers of 2 will include common subexpressions. Obviously this optimization is effective in DSP-like computations, where there is a sequence of multiplications (and optionally a sequence of sums). To clarify we can think at the following example:

$$z = x[0] \cdot a + x[1] \cdot b$$

The two constants a and b are: a=185=10111001_2 and b=221=11011101_2. Looking at the positions of the '1's we see that the first, fourth, fifth, sixth and eighth '1's are commons, so the corresponding shifts can be shared.

However, from an HW point of view, we will now introduce the approach to follow in order to reach a functioning translation system and to integrate it into the fault tolerance architecture already developed. It is actually a future work but we want now to lay the foundations for a proper development of the idea.

To obtain the previously described system we will mainly need two blocks inside the ID stage:

1. Translation Controller: it is actually a dispatcher extension, it has to recognize the need to activate the translation mechanism looking at the availability of the MULT replicas. It's main job is to activate the translation mechanism once it recognize that there are no more MULTs available for the current instruction and the second job is to stall the pipe in order to let the translator to complete its job.

2. Translator: it is the real block that perform the translation. It receives the operands of the multiplications and perform the translation. This is the component where we can do most of the considerations: we could think to have a very passive block that will always decompose the same input, or we could think of a dynamic components that decide which input to translate in order to do less work as possible (so less shifts and less sums). Other consideration can be made on this process in order to optimize it but anyway the most important thing is about the complexity of this Translator. We could think of a Translator that has to perform like the ID decoder, thus directly providing the commands and the operands to the ALUs to perform the "multiplication" with sums and shifts, or we could have a simpler Translator that is responsible for just acting as a virtual IF stage for instructions that it creates on fly and that it gives the ID decoder. In this second case the Translator creates a sequence of simple add and shift instructions and send it to the decoder that then acts in normal way as nothing is happening.

We will not go into details for this translations mechanism because will not be implemented in this version of the core, but this is a good starting point for future works. Actually in this version of the core, the dispatcher has been updated with the possibility to detect the unlucky case of no available MULTs but much other works has to be done to support such a mechanism.

5.5 CSR extension

Control and Status Register are registers that maintain the working state of a RISC-V machine.
CSRs are defined in the RISC-V privileged specification but cv32e40p implements only the registers that were needed for the PULP system [21]. The reason for this is that we wanted to keep the footprint of the core as low as possible and avoid any overhead that we do not explicitly need.

CSR are addressed by 12-bit word (csr[11:0]) for up to 4,096 CSRs.

By convention, the upper 4 bits of the CSR address (csr[11:8]) are used to encode the read-write accessibility of the CSRs according to privilege level as shown in Figure 5.5.1. The two most significant bits (csr[11:10]) indicate whether the register is read/write (00,01, or10) or read-only (11). The next two bits (csr[9:8]) encode the lowest privilege level that can access the CSR:

- 00: Machine level;
- 01: Supervisor level;
- 10: Hypervisor level;
- 11: User level.

CSR Address			Hex	Use and Accessibility
[11:10]	[9:8]	[7:4]		
User CSRs				
00	00	XXXX	0x000-0x0FF	Standard read/write
01	00	XXXX	0x400-0x4FF	Standard read/write
10	00	XXXX	0x800-0x8FF	Custom read/write
11	00	0XXX	0xC00-0xC7F	Standard read-only
11	00	10XX	0xC80-0xCBF	Standard read-only
11	00	11XX	0xCC0-0xCFF	Custom read-only
Supervisor CSRs				
00	01	XXXX	0x100-0x1FF	Standard read/write
01	01	0XXX	0x500-0x57F	Standard read/write
01	01	10XX	0x580-0x5BF	Standard read/write
01	01	11XX	0x5C0-0x5FF	Custom read/write
10	01	0XXX	0x900-0x97F	Standard read/write
10	01	10XX	0x980-0x9BF	Standard read/write
10	01	11XX	0x9C0-0x9FF	Custom read/write
11	01	0XXX	0xD00-0xD7F	Standard read-only
11	01	10XX	0xD80-0xDBF	Standard read-only
11	01	11XX	0xDC0-0xDFF	Custom read-only
Hypervisor CSRs				
00	10	XXXX	0x200-0x2FF	Standard read/write
01	10	0XXX	0x600-0x67F	Standard read/write
01	10	10XX	0x680-0x6BF	Standard read/write
01	10	11XX	0x6C0-0x6FF	Custom read/write
10	10	0XXX	0xA00-0xA7F	Standard read/write
10	10	10XX	0xA80-0xABF	Standard read/write
10	10	11XX	0xAC0-0xAFF	Custom read/write
11	10	0XXX	0xE00-0xE7F	Standard read-only
11	10	10XX	0xE80-0xEBF	Standard read-only
11	10	11XX	0xEC0-0xEFF	Custom read-only
Machine CSRs				
00	11	XXXX	0x300-0x3FF	Standard read/write
01	11	0XXX	0x700-0x77F	Standard read/write
01	11	100X	0x780-0x79F	Standard read/write
01	11	1010	0x7A0-0x7AF	Standard read/write debug CSRs
01	11	1011	0x7B0-0x7BF	Debug-mode-only CSRs
01	11	11XX	0x7C0-0x7FF	Custom read/write
10	11	0XXX	0xB00-0xB7F	Standard read/write
10	11	10XX	0xB80-0xBBF	Standard read/write
10	11	11XX	0xBC0-0xBFF	Custom read/write
11	11	0XXX	0xF00-0xF7F	Standard read-only
11	11	10XX	0xF80-0xFBF	Standard read-only
11	11	11XX	0xFC0-0xFFF	Custom read-only

Figure 5.5.1: Allocation of RISC-V CSR address ranges.

Attempts to access a non-existent CSR raise an illegal instruction exception. Attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction exceptions. A read-/write register might also contain some bits that are read-only, in which case writes to the read-only bits are ignored. [18]

For a complete description of all RISC-V CSRs refer to [18] in the section "Control and Status Register" where the meaning of any single CSR is explained, and refer to Table 8 of [21] where there is a detailed description of the RISC-V CSR actually implemented in cv32e40p core.

The instructions to access and manipulate CSR are described in Figure 5.5.2:

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

Figure 5.5.2: CSR manipulation instructions [18]

In order to monitor the status of our ALUs and MULTs we will add custom CSRs to the set of supported CSRs in order to count number of errors into the monitored components and to save their health state.
There are already some performance counters supported by RISC-V ISA and they are also mapped to the CSR address space. We will add our custom performance counters and a set of flops as the table of permanent faulty components.
First of all we have to choose the address space we want to reserve for our custom extension. Looking at Figure 5.5.1 we have chosen $0x7C0 - 0x7FF$ and $0x800 - 0x8FF$ address spaces to map our registers. Both regions are *custom read/write* so both satisfy the first requirement of being able to be read and written by software via instructions.
$0x800 - 0x8FF$ addresses are partially occupied by PULP extension registers, in particular $0x800 - 0x807$ are used for Hardware Loop PULP functions.
We decided to map counters for our 36 ALUs and 12 MULTs to $0x808 -$

$0x837$ address space, while we decide to locate tables of permanent faulty components into three registers at addressed $0x7C0 - 0x7C2$.
All register are infact 32 bits registers and in order to save the status of 36+12 components we need 48 bits. However we decided to separate registers dedicated to ALU to those related to MULT because we can not exclude future modifications. The 48 flops distribution is the following:

- $0x7C0$: first 32 ALU bits - CSR_PERM_FAULTY_ALUL_FT
- $0x7C1$: last 4 ALU bits - CSR_PERM_FAULTY_ALUH_FT
- $0x7C2$: 12 MULT bits - CSR_PERM_FAULTY_MULT_FT

The three registers that compose the table of permanent faulty components are in the Machine level CSR address space so they are protected against wrong read/write from the software that runs at an higher level. Unfortunately we can not insert the 48 performance counters inside that space too because it is too limited. Therefore performance counters are in User address space and they are surely accessible by any high privileged level but are less secure because any program running on the core can possibly corrupt them.

All our custom CSRs are accessed via special protocol that extend the one provided in the basic cv32e40p.
The instructions to access those CSRs are correctly decoded in the ID stage and the proper signals are generated into CSR block. We added special signals to manage the reading and writing on our custom CSR. In particular, if the received address is one of those added for custom reasons, it is redirected into the EX stage where there are our physical registers. At the same time, if we have to read a custom register we activate a read enable r_en and redirect the output of the physical register from EX stage to the CSR block. If on the other side we have to write the register, we activate a write enable w_en and redirect the writing value from CSR block to EX stage (look at the Figure 5.4.3).

Chapter 6

Simulation and results

In this paragraph we will address the verification problem. It is a fundamental step to state that the architecture operates in the correct way and to investigate the real fault tolerance it is able to achieve.
We have to distinguish between the classical functional verification and the simulations campaign for the fault tolerance evaluation. The first is done to verify that the hardware extensions provided to protect the core do not affect the normal behaviour of the core in absence of faults, while the second is conducted to evaluate the fault tolerance capabilities of the new version of the core.

We will use QuestaSim software for the SystemVerilog simulation and thanks to the *force* QuestaSim command we will be able to inject faults into the core by forcing a flip in a random bit of the architecture.

6.1 The simulation environment

The simulation environment that we have developed is a complex software composed of several scripts with the main objective of automating the entire simulation and validation process.

Our fault tolerance validation environment (from now on called FT verification environment) is actually based on the one adopted for RI5CY core by PULP group and here reported in Figure 6.1.1.

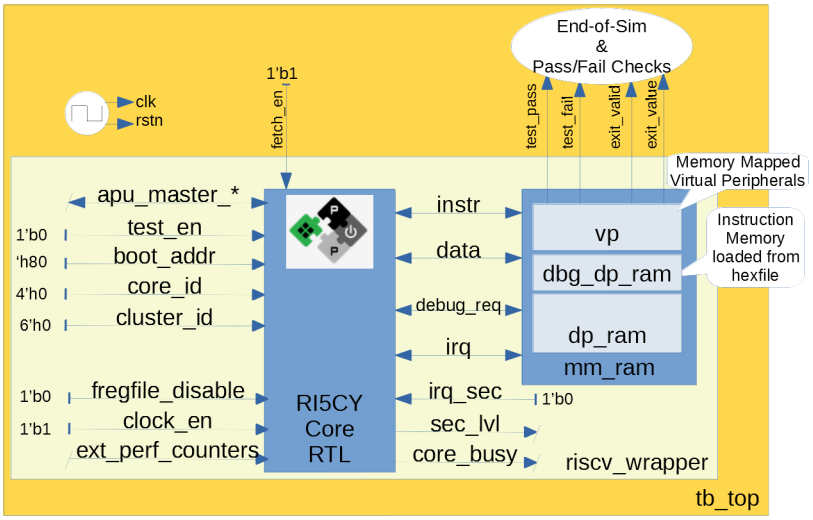

Figure 6.1.1: RI5CY Testbench [69]

The testbench is written entirely in System Verilog: in the top module tb_top there is the wrapper riscv_wrapper that contains the instantiation of the RISC-V core together with a memory model. The mm_ram maps the dp_ram module to the instruction and data ports of the RI5CY processor core and some pseudo peripherals. The most useful of these is a virtual printer, actually a redirection of writes to stdout (when the core writes ASCII data to a specific memory location it is written to stdout). In this way, programs running on the core can write human readable messages to terminals and log files.
Testcases are coded in C and/or RISC-V assembly-language and are all self-checking i.e. the pass/fail determination is made by the testcase itself as the testbench is not able to find errors.

At [69] there is a detailed description of the verification environment adopted for RI5CY core and here adopted in the verification of the fault tolerant version of cv32e40p.

Its main limitations are:

- A significant amount of testcase writing will be required to achieve full core coverage.
- testcases are the same every time they run, so only the stimulus we think about will be run and only the bugs we can imagine will be found.

- Stimulus generation and response checking is 100% manual.
- The performance counters are not verified.
- others at [69]

OpenHW group actually developed a good UVM environment to improve the verification of the core [17], but we could not use it as we lacked the license to use the Imperas Instruction Set Simulator (ISS).

However, apart from the verification of the special HW added to the basic core, we built an automatic simulation process to compare the fault tolerant architecture with the basic one. Every software was firstly simulated on the basic core and then simulated again on the new architecture, feeding it with the same inputs. This approach will be a real advantage for the various simulations that we will perform on the fault tolerant version of the core, when we want to simulate only a small piece of the core (e.g. the ex_stage) with the fault injection. In fact, since we want to save time spent on simulations, we don't want to simulate the whole core but we aim to simulate and inject faults only on the unit of interest. Therefore we would have need of a special testbench for each sub-unit and this may be quite tricky as the unit under test could change.

So we decided to go with the following ideas:

1. simulate the whole core with a specific firmware;
2. save inputs and outputs of a sub-unit;
3. re-simulate only the sub-unit feeding it with the saved inputs;
4. load the previous saved outputs and compare them with the outputs of the current simulation.

A very important aspect is that the previous steps are all completed within QuestaSim environment thanks to built in commands and tcl scripts (QuestaSim language is actually tcl).

We can exploit this technique for two purposes as suggested above:

1. verify that the fault tolerant extension doesn't affect the normal behaviour of the core in absence of faults.
   We perform this verification if the initial simulation is done on the

basic core (called ref) and the second one on the protected version of the core (called ft), in other words we will say ref vs ft with no FI (fault injection).

2. verify that the ft core is protected against faults, if the initial simulation is done on the ft core and the second on the same ft core with now the fault injection. We will say ft vs ft with FI.

6.1.1 Detailed structure of FT verification environment

The main software is called comp_sim.sh and it is the program adopted to manage:

1. the compilation of the firmwares;
2. the simulation of the firmwares on the core;
3. the evaluation of the fault tolerance of the ft version of the core

The first two points are actually managed by the RI5CY verification environment and we just create a system that better organize the compilation and simulation of firmwares in order to obtain a more user friendly interface.

The real challenge was the creation of the fault tolerance verification system and its integration inside the general verification environment.
As introduced above, the main idea was to inject faults only inside a sub-unit of the core in order to save time and to have a finer grain fault tolerance evaluation.

The process as described in section 6.1 is mainly managed by three *tcl* scripts running on QuestaSim:

1. *vsim_save_data_in.tcl*: it is responsible for saving all the input signals value during the first simulation into a .vcd file that will be used again as input for the second simulation.
2. *vsim_save_data_out.tcl*: it is responsible for saving all the output signals value during the first simulation into a .wlf file that will be used again as a comparison with the outputs from the second simulation.
3. *vsim_stage_compare.tcl*: this is the most complex file, responsible for comparing the outputs from the second simulation with those from the previous simulation that have been stored into the .wlf file. As

mentioned before, the comparison can be performed on couples of simulations in order to verify either the functionality of the core (ref vs ft with no fault injection) or the fault tolerance of the core (ft vs ft with fault injection).
This script manages several things to be able to get the final comparison between the two different simulations: first of all it has to find all the signals where to possibly inject the faults, and they are all the inputs signals plus the internal registers. Then the script has to manage the clock creation because clock is a special signals and can't be loaded from the previously saved .vcd file as if it were a standard input. The clock creation is done with the $force$ command, the same we will use to generate the fault. Anyway, the script has to reload the so called "dataset" that is the previous saved simulations (in the .wlf file) and has to start the new simulation and comparison at the same time. In fact we will perform the new simulation by dividing it in several pieces and we will compare the new simulation piece from time to time: in this way, when the first error is encountered, the simulation stops and we can go further, thus saving time.
Actually *vsim_stage_compare.tcl* performs the same comparison for a specific firmware N times, with N equal to the number of cycles we want to repeat the simulation-comparison with the fault injection. Each simulation-comparison will have the fault injected on a different instant or on a different signal, in order not to repeat an already done simulation (see subsection 6.3.1). If the comparison is done without fault injection it is not necessary to repeat the comparison N times, so it is only performed once.

Each firmware that we want to run on the core has to be compiled with riscv-gnu-toolchain to obtain the .hex file ready for the execution on our core.

6.2 Functional verification

The first step to rate the fault tolerance of the protected version of cv32e40p is to verify the functionality of the core itself. In fact the most important characteristic of the fault tolerant extension is that it has to be completely transparent in terms of functionality if no faults occur.
Therefore the core was fed with several example firmwares that had already been used to verify the functionality of the basic core (ref). In addition, some of them were modified to stimulate the new hardware in order to state it

has been correctly designed. For example to verify the managing of the new CSRs added to the basic list of registers, a C software was modified with some *asm* directives in order to directly access those registers via custom assembler code (Listing 6.1). With this simple C program we are also able to verify the mechanism of the spare unit and so the dispatcher job together with the clock gating switching between replicas.

```
1
2  ...
3    //Variables for custom CSR for Fault Tolerance monitoring
4    unsigned int count_shift_0;
5    unsigned int perm_faulty_alu_h, perm_faulty_alu_l;
6    unsigned int perm_faulty_mult;
7  ...
8
9    // Control and Status Register Write
10   // pseudoinstruction: expands to csrrw x0, csr, rs1
11   __asm__ volatile("li a5, 0x0");
12   __asm__ volatile("li a5,0x89ABCDEF");
13   __asm__ volatile("csrw 0x808, a5");
14
15   __asm__ volatile("li a5,0x32");
16   __asm__ volatile("csrw 0x7C0, a5");
17
18   __asm__ volatile("li a5,0xF7");
19   __asm__ volatile("csrw 0x7C1, a5");
20
21   __asm__ volatile("li a5,0xF7");
22   __asm__ volatile("csrw 0x7C2, a5");
23
24   // Control and Status Register Read
25   // pseudoinstruction: expands to csrrs rd, csr, x0
26   __asm__ volatile("csrr %0, 0x808" : "=r"(count_shift_0));
27   __asm__ volatile("csrr %0, 0x7C0" : "=r"(perm_faulty_alu_l));
28   __asm__ volatile("csrr %0, 0x7C1" : "=r"(perm_faulty_alu_h));
29   __asm__ volatile("csrr %0, 0x7C2" : "=r"(perm_faulty_mult));
30
31  ...
32   // print value read from CSRs
33   printf("\tcount_shift_0      = 0x%0x\n", count_shift_0);
34   printf("\tperm_faulty_alu_l  = 0x%0x\n", perm_faulty_alu_l);
35   printf("\tperm_faulty_alu_h  = 0x%0x\n", perm_faulty_alu_h);
36   printf("\tperm_faulty_mult   = 0x%0x\n", perm_faulty_mult);
37 ...
38
```

Listing 6.1: C code to verify custom CSR accessing

The above code can be divided in four pieces: at the beginning we simply declare some variables where will be stored the content of the CSR after the

read instructions. Then with a set of three instructions we write into four custom CSRs some random values (in hexadecimal format). The next step is obviously to read what has been written into those register and finally to print the read value. In this way if the write/read mechanism is correct, we must see on screen exactly what we have decided to write into the registers. In particular we decided to write $0x89ABCDEF$ in the register associated to the first of the nine subparts in which is divided the first ALU, thus the counter of the *logic* operations on the first ALU, which is at address $0x808$. Then we wrote $0x32$, $0xF7$ and $0xF7$ into the three registers which represent the table of permanent faulty components at addresses $0x7C0$, $0x7C1$ and $0x7C2$. The first two store the 36 bits associated to the 36 counters for the four ALUs, while the third stores the 12 bits associated the 12 counters for the three MULTs (see 5.5).

In Figure 6.2.1 there is the output of the previous C program printed on the QuestaSim Transcript window. The printed value for the four registers we have considered is exactly as expected. We decided to write a strange value ($0x89ABCDEF$) on the first register at $0x808$ even if it can only reach 100 that is the maximum threshold we have chosen for the counters, just to verify that no conflict arose inside.

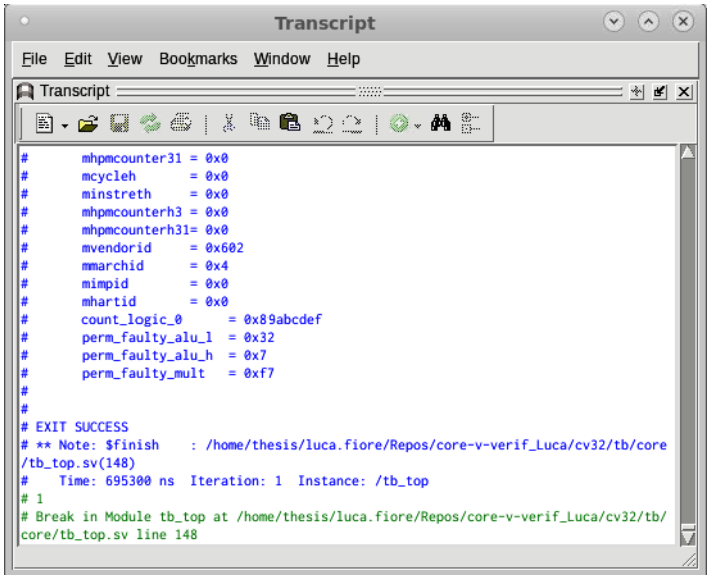

Figure 6.2.1: Custom csr access: output of C code in Listing 6.1 from QuestaSim simulation.

Following the approach used to verify the CSR read/write, we can test the functioning of the fault tolerant mechanism: counters, dispatcher and clock gating for the spare unit.
If for example we write into the counter dedicated to logic operation the same value as the previous example, that is $0x89ABCDEF$, since we will not inject any fault, we expect to see that value to be decremented. In fact, each time that specific ALU has to compute an operation that involve a logic operation, the voter will not detect any error and so the counter will be decremented by the value we have set into the package, thus '1'.
In Figure 6.2.2 we got exactly what we expected: $count\_logic\_n[0]$ (counter of errors on logic operations for the first ALU) is $0x89ABCDEF$ and at 40080 ns, when $alu\_operator\_i$ is 15 that is a logic AND, the counter decrease to $0x89ABCDEE$ because no error can be detected since no fault has been injected. From Figure 6.2.2 we can also see that $count\_logic\_q[0]$ is modified one clock cycle after the update of $count\_logic\_n[0]$ as expected, because q is the output of the register and n is the input.

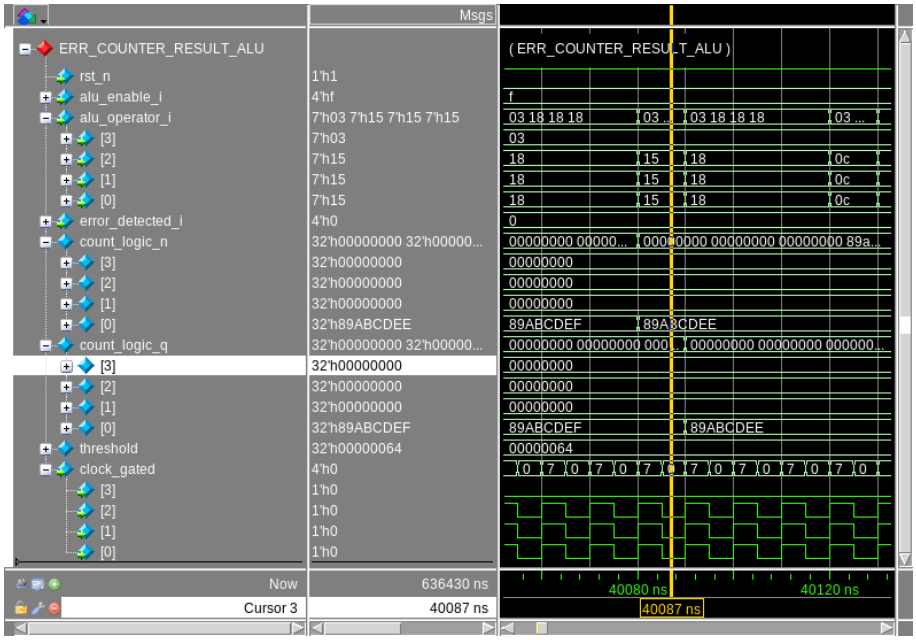

Figure 6.2.2: Counter decreases after writing, because no fault is injected.

Another important thing we can see from Figure 6.2.2 is the clock gating mechanism. The signal $clock\_gated$ is the clock received from the four replicas of the pipe, from the four replicas of the ALU and from the four

sets of nine counters. The clock of the fourth chain (fourth pipe replica, fourth ALU replica and the associated set of counters) is gated while only the first three replicas work. In fact if we look at $alu\_operator\_i$, only the first three signals ([2], [1] and [0]) contain the updated value of the operator, while the fourth one is frozen due to clock gating. This is even more evident in Figure 6.2.3

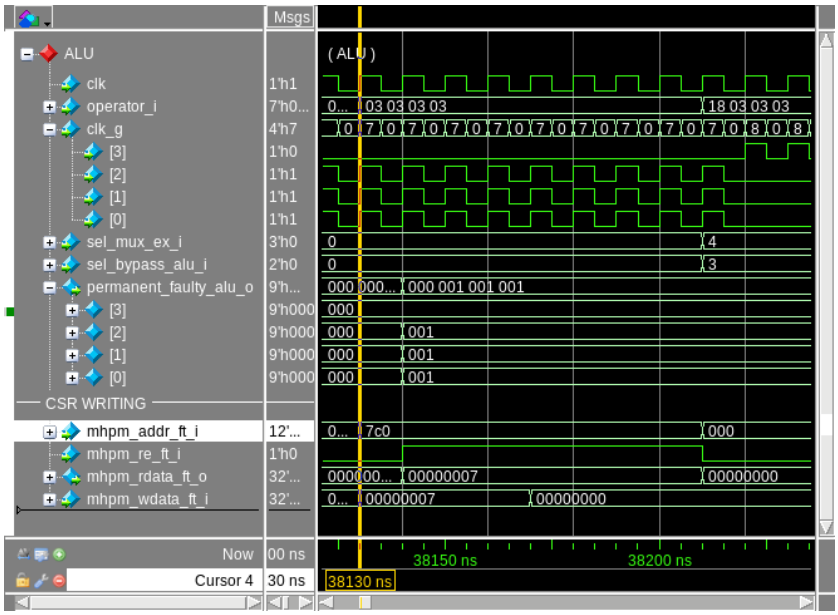

Figure 6.2.3: Custom csr access: Clock gating.

The previous figure shows some waveforms of a simple program where we write a value of $0x7$ inside the table of permanent faulty components for the ALU, in particular we wrote $0x7$ into CSR at address $0x7C0$ (See 5.5). In the section $CSR\ WRITING$ of Figure 6.2.3 there is the standard CSR writing/reading protocol already seen: at 38130ns the writing starts, and after a clock cycle we have the value inside the register.

We wrote a $0x7$ which in binary means 00..0111 into the register at $0x7C0$: the three '1's of the value mean that the first subpart of the first three ALUs are permanently broken, as you can see looking at the signal $permanent\_faulty\_alu\_0$ where the three '1's are divided into the address [0], [1] and [2] of $permanent\_faulty\_alu\_0$.
Now what we expect is to see the first three ALUs frozen when a logic operation has to be performed, because the first subpart in which an ALU is

divided is exactly the one dedicated to the logic operations. This is exactly what we see at time 38210ns (almost at the end of the simulation displayed in Figure 6.2.3) when ALU operator become 18: only the 4th ALU replica is active infact ALU operator ($operator\_i$ in the figure) is updated with the value of 15 only for the 4th ALU replica and the same happens to the clock that is gated for the first three replicas (the faulty ones).

Several other firmwares have been tested on the core and almost every one of them were taken from the database provided by OpenHW group that was adopted to test the basic core too. We just modified some of them to test the added components and added Coremark, an important program to benchmark the whole core. [73] [32].

So, the first series of simulations were done to state that the functioning of the core was not affected by the fault tolerant extension. The results we got for each sw simulated on the core are similar to what is reported in Figure 6.2.4.

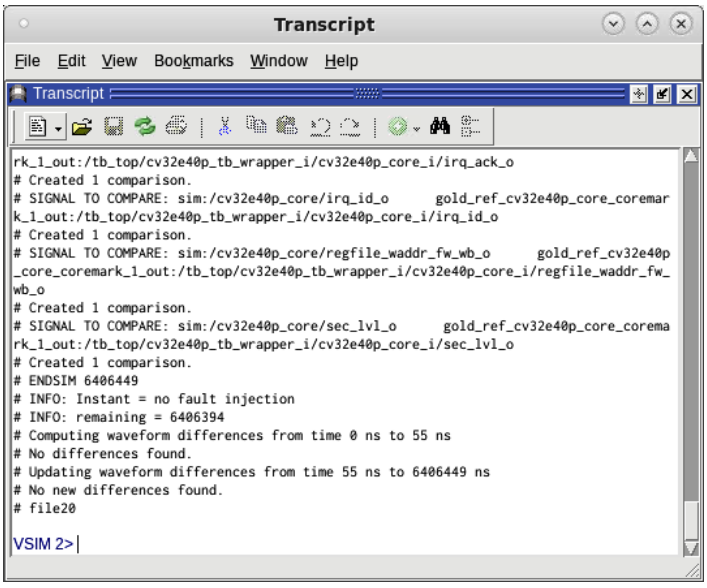

Figure 6.2.4: Wave comparison: ft vs ref architecture.

The previous figure shows the command window of QuestaSim where are reported the functions executed from the *.tcl* script we have written to compare the two simulations. As you can see there is a list of signals that are

compared (the list is actually longer than what is in the figure) and at the end there is a summary of the found differences. In this case obviously, since there is no fault injection, there are no differences and so we can confirm that for this particular SW (Coremark benchmark) the fault tolerant version of the core behaves like the basic one.

6.3 Fault tolerance verification

This section will cover the fault tolerance evaluation of the cv32e40p with and without the HW extension designed in this project.

First of all we have to decide how to perform the simulation campaign, that is, where and when inject faults and in particular how many simulations to get a good coverage and so to have a reliable result on the fault tolerance of our architecture. Then we have to practically perform the simulations and we have to compare the results obtained on the ref architecture with those obtained on the ft architecture with fault injections in both cases.

6.3.1 Choice of the number of simulations for the fault tolerance evaluation

The fault injection campaign we need to run to get the information on the network fault tolerance capability could be incredibly huge in terms of number of simulations.
Just think of the need to inject a fault on every single bit of every possible signal on every clock cycle over the entire simulation time for a specific firmware running on the core. It is easy to predict how large could be the required number of simulations to have a coverage of 100% of all the possibilities. For example if we just want to simulate a single stage of the core without any fault tolerance extension, the ID stage for instance, and if we decide to inject the fault on all the signals coming from the pipe of the previous stage and on each register of the register file, we have an amount of 53 signals and 1560 bits. If we simulate a simple "hello world" code consisting of 16000 clock cycles, we will need $1560 \cdot 1600 = 24960000$ different simulation to cover all the possible fault injections. And if each simulation requires 1min to be done on a computer (actually it could be even worse), we will have to wait almost 550 days to receive a correct result, so an unacceptable amount of time.

For this reason we decide to follow the Statistical Fault Injection (SFI) method [47], a way that allow a great reduction of the number of simulation

required to have a good result, where *good* is related to the confidence level and to the margin of error with respect to the result that would have been obtained with the complete fault injection campaign.
If N is the total population, representing the number of simulations to have a full fault injection, we can compute the number of simulations n required to have a certain coverage with Equation 6.1:

$$n = \frac{N}{1 + e^2 \cdot \frac{N-1}{t^2 \cdot p \cdot (1-p)}} \tag{6.1}$$

- N: initial population size. It is the total amount of faults we can inject;
- n: reduced number of faults to inject. It is the number of fault randomly selected for the injection or in other words the sample size. We consider each individual in the initial population N (a specific fault at a given clock cycle) to have the same probability to be sampled, thus a uniform distribution must be used during sample.
  When N is large, increase of N has little influence on the sample size for a given margin of error and a given confidence.
- p: estimated proportion of individuals in the population having a given characteristic (e.g. the estimated probability of faults resulting in a failure). This parameter defines the standard error.
  The parameter p basically corresponds to an estimate of the true value being searched (e.g. percentage of errors resulting a failure). Since this value is a priori unknown (but between 0 and 1), a conservative approach is to use the value that will maximize the sample size. It has been demonstrated that this is achieved for p=0.5;
- e: margin of error. It means that if we classify the types of failure we have after the fault injection on all the population we will get that the exact probability that individuals have the desired characteristic should be a value between $[X - e; X + e]$, where X is the value of probability that individuals have the desired characteristic obtained during the campaign using the sample.
  The margin of error is the most influential parameter on n because a small reduction of e produce a strong increasing of the required sample size (Figure 6.3.1).

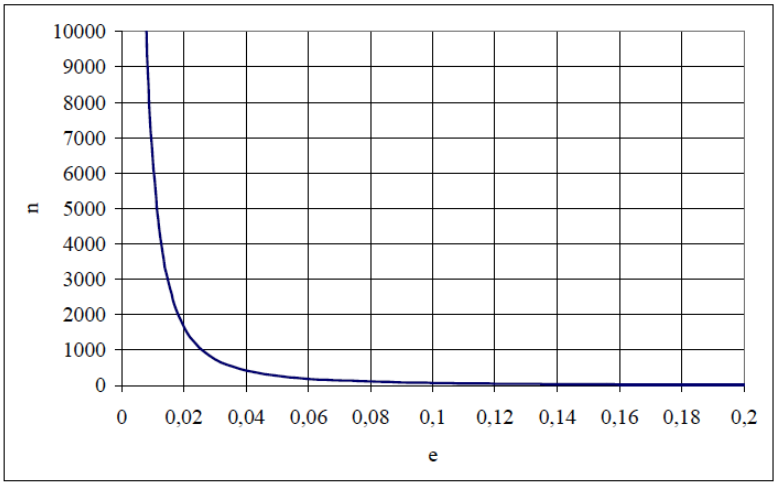

Figure 6.3.1: $n = f(e)$ (p=0.5, N=150000, 90% confidence level). [47]

- t: cut off point corresponding to the confidence level. It is computed with respect to the Normal distribution (quantile table) and is the probability that the exact value is actually within the error interval. A 90%, 95% or 99% confidence level is usually chosen (typically 95%) with corresponding t of 1.96, 2.5758 and 3.0902.
  Increasing the confidence level (and thus the t parameter) has less impact on n than reducing e.

In the initial population, sampling is done without replacement. Otherwise, the population can be considered as infinite as the same individual can be chosen multiple times thus giving a possible bias.

What we have to do:

1. compute the number of bits where we can apply fault injection;
2. compute the number of cycles where a given bit can change;
3. multiply the two previous values obtaining N.
4. set the estimated proportion p as 0.5, choose a confidence level t (between 95% and 99%) but more important choose a margin of error e (between 0.1% and 5%).

For our simulations we have chosen:

- t=2.5758, (99%)
- e=0.05.

and the resulting (rounded) n is 663 for all the firmwares, so for all N.

6.3.2 Results

We performed the same simulations on both ref and ft architecture to have a final comparison on the fault tolerant improvement provided by our work.

The two tables below summarize the main results:

Firmware	Errors	Fault Tolerance
coremark_1	1	99.85%
counters	0	100.0%
csr_instructions	2	99.70%
cv32e40p_csr_access_test	14	97.89%
dhrystone	0	100.0%
fibonacci	0	100.0%
generic_exception_test	0	100.0%
hello_world	1	99.85%
illegal	0	100.0%
interrupt_bootstrap	0	100.0%
interrupt_test	1	99.85%
misalign	0	100.0%
modeled_csr_por	0	100.0%
perf_counters_instructions	0	100.0%
requested_csr_por	0	100.0%
riscv_arithmetic_basic_test_0	10	98.50%
riscv_arithmetic_basic_test_1	11	98.35%
riscv_ebreak_test_0	0	100.0%
mean	≈ 2	99.69%

Table 6.1: Results with transient fault injection on ft architecture

Firmware	Errors	Fault Tolerance
coremark_1	70	89.45%
counters	156	76.48%
csr_instructions	86	87.03%
cv32e40p_csr_access_test	103	84.47%
dhrystone	125	81.15%
fibonacci	155	76.63%
generic_exception_test	95	85.68%
hello_world	77	88.39%
illegal	90	86.43%
interrupt_bootstrap	94	85.83%
interrupt_test	99	85.07%
misalign	136	79.49%
modeled_csr_por	83	87.49%
perf_counters_instructions	150	77.38%
requested_csr_por	67	89.90%
riscv_arithmetic_basic_test_0	121	81.75%
riscv_arithmetic_basic_test_1	88	86.73%
riscv_ebreak_test_0	122	81.60%
mean	≈ 107	83.86%

Table 6.2: Results with transient fault injection on ref architecture

The average fault tolerance of the ref architecture is so 83.86% while the one of the ft architecture is 99.69%. These results are obtained injecting transient faults that is the reason why also the ref architecture is quite good in terms of fault tolerance. In fact it is clear that there is a non-zero possibility that the injected transient fault occurs on a signal that is actually not used in that specific clock cycle thus giving no problems to the simulation.
But what happens if we inject permanent fault? We performed permanent faults injection on simulations of few firmwares, Coremark, hello_world and riscv_arithmetic_basic_test_1. This is due to the limited time as we did not have time to complete the fault injections for all the firmwares.

Firmware	Errors	Fault Tolerance
coremark_1	18	97.22%
hello_world_1	19	97.13%
riscv_arithmetic_basic_test_1	21	96.83%
mean	≈ 19	97.13%

Table 6.3: Results with permanent fault injection on ft architecture

Firmware	Errors	Fault Tolerance
coremark_1	330	50.23%
hello_world	167	74.82%
riscv_arithmetic_basic_test_1	276	58.37%
mean	≈ 193	70.89%

Table 6.4: Results with permanent fault injection on ref architecture

As you can see from the two previous table the fault tolerance of the core decrease because it has to face permanent faults that are surely more critical than transient ones.
The ref architecture goes from 83.86% to 70.89% while the ft architecture goes from 99.69% to 97.13%. This is due to the fact that a transient error may occurs in a clock cycle where the infected signal is not used, while if we apply a permanent fault on that signal there is an higher probability that it will affect the outputs.

The ft architecture can't provide a 100% fault tolerance because there are some secondary processes in the ex_stage.sv file that were not protected because we focused only on the ALU ans on the MULT. Actually it would be enough to apply a simple TMR to these parts as they are really simple and not critical.

Chapter 7

Conclusions

This thesis aims to find an innovative solution to increase the fault tolerance of cv32e40p RISC-V core. We have adopted already know techniques like TMR within a new approach aimed to exploit available resources at the maximum level. We also managed to make the fault tolerant extension to be customizable (to be activated or deactivated, to set the counting and threshold values for the performance counters) and we tried to optimize the new architecture with a view to low power consumption (clock gating on the spare components and on the faulty ones).

As reported in the previous chapter we got very good results in terms of fault tolerance both for transient and permanent errors. What we could do in the future is first of all to increase the number of simulations to get a wider coverage (now we have performed 663 simulations for each firmware) and secondly to complete the permanent faults simulations campaign with all other firmwares (in fact we tested Coremark only).

Another important aspect that has not been properly addressed during this work is the power and speed ratings. In fact we have not synthesised the network and we have not analyzed it from these points of view for reasons of time. In fact all the work done on the ID and EX stage of the machine is really expensive in terms of area as we have added many extra components to the basic architecture leading to a large occupation of area. In partiular we have added three extra ALUs, two extra MULTs, two voters, 36 up/down counters for the 4 ALUs, 12 up/down counters for the three MULTs, two 32b registers as table of faulty components and other minor components such as some muxs and some clock gating blocks. This means that the area is surely

increased and the static power consumption too for the same reason, but we tried to maintain this overhead as low as possible thanks to clock gating over the faulty and spare components. Actually our aim is to get a good level of fault tolerance with less interest on power, area and delay: the main future work can be to improve the architecture with a view to performances.

In the next section just some additional future works.

7.1 Future works

We have mentioned many future works during the discussion, such as the translation of multiplications into an adds-shifts sequence, or the general optimization of the architecture for performances. Here we want to add some good ideas that could be used to improve the core even more and to reach an even better level of fault tolerance.

7.1.1 BIST

There is a big problem related to the previous described architecture. If for example two of the four ALUs are declared defective, the voter that now is a simple comparator can't recover from a detected mismatch of the two received inputs (the usual problem of DMR with respect to TMR). Infact is impossible to choose between two inputs coming from units at the same level, that is, with the same confidence and the same implementation.
In this situation a good solution can be to implement a Built In Self Test (BIST) as described in 3.2.2.
The idea is to provide a quiescent BIST mechanism that will be waked up only when there is the remote and unlucky situation of only two available components for the TMR (now DMR).
When the error detection mechanism detects errors without the possibility to recover them, so when the results of the only two available components are different, BIST support will be enabled. We can think of a BIST done on both the available units to understand which of them is operating correctly. This new error detection mechanism can be applied even before the error happening in order to prevent wasting of time once the error actually occurs. That is, if for example there are spare cycle where nothing has to be done on the ALU because the MULT is involved in those cycles, we can perform BIST on the ALU, and vice versa.

You can think of the real need for a TMR when you could apply a BIST knowing that it is a good way to test the behavior of each single replica without comparing it with the others. Unfortunately BIST is really a time and energy consuming activity and the scheduling of BIST cycles in order to minimize energy losses and waste of time is a real challenging activity because dynamically depends on the core workload, therefore on the actual application.
The real disadvantage of this technique is the complexity it introduce and the low frequency of the situation in which it can actually be applied. Infact the probability that two of the three or four available components are broken is low and sometimes it may never happen before the entire core is decommissioned. So we would have set a very complex BIST mechanism for nothing.

7.1.2 NRMR

Replicate the module in heterogeneous way, with different architectural solutions or simply with different output precision, and compare the outputs; if the difference is greater than a certain threshold then we can say that there is an error. For example, inside the voter, we can check if the output is just 1bit far from the correct result (1bit threshold).
The reduced precision replica can be an extra 4th replica in the TMR approach, used just when an error is detected to ensure that the error is just one and not two identical errors in two different replicas, or it can be useful when there are only two available units for that operation and so when the TMR became a DMR.

In 3.2.1 we have introduced the technique and now we want to summarize the pros and cons of it.
Among the benefit of a NRMR there is the *design diversity* which implies the possibility to distinguish between results coming from different replicas. Infact if two outputs have different precision we can give more confidence to the higher precision one, or even if the only difference is on the architecture of the replica. we can give them different confidence.
This is particularly important when we are in the case of only two available components (see 7.1.1), and also when we want to be sure that there is not an even number of errors going into the voter. In the first case for example we can have a sort of voting even with only two data because they are in some way different and so the DMR can take a decision. In the second case, if an error is detected on one of the three inputs to the voter we can state that

the error is exactly the one detected, and that there is not the possibility to have instead two errors in the other two results which may implies the voter to be wrong, for the same reason of different confidence of the inputs to the voter.

7.1.3 Repetition if unsure

if from TMR there is an uncertainty (only two of the three units agree) still continue but repeate the faulty operation in the next clock cycle to verify if it was a transient fault or something more serious (so spatial redundancy with temporal redundancy also called instruction retry). For example if the ALU detects and corrects one error thanks to TMR, nothing happens except that the pipe will be stalled for the number of clock cycle required to understand if the fault is transient or permanent. If it was transient everything return to normal, but if it is permanent the core continues with a DMR in that section knowing that it will be only able to detect an error and nothing more. This is a faster way to detect permanent errors with respect to our one that is based on counters, but actually the two techniques collapse into just one if for example we decide to fix the threshold to '2' thus counting maximum two consecutive errors.

Bibliography

[1] M. Gautschi P. D. Schiavone A. Traber. *RI5CY: User Manual*. Apr. 2019.

[2] D. A. Patterson A. Waterman Y. Lee and K. Asanović. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. May 2016.

[3] *ALU instruction set extension for SIMD*. URL: https://cv32e40p.readthedocs.io/en/latest/instruction_set_extensions/#simd.

[4] D. Anderson. "Design of self-checking digital networks using coding techniques". In: 1971.

[5] *Arithmetic Operations GCC built-ins for GAP8 RISC-V ISA extensions*. URL: https://greenwaves-technologies.com/manuals/BUILD/PULP-OS/html/group__Arith.html.

[6] A. Avizienis. "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design". In: *IEEE Transactions on Computers* C-20.11 (1971), pp. 1322–1331.

[7] Avizienis and Kelly. "Fault Tolerance by Design Diversity: Concepts and Experiments". In: *Computer* 17.8 (1984), pp. 67–80.

[8] Joel Bartlett, Jim Gray, and Bob Horst. "Fault Tolerance in Tandem Computer Systems". In: *The Evolution of Fault-Tolerant Computing*. Ed. by Algirdas Avižienis, Hermann Kopetz, and Jean-Claude Laprie. Springer Vienna, 1987.

[9] Charles R Baugh and Bruce A Wooley. "A two's complement parallel array multiplication algorithm". In: *IEEE Transactions on computers* 100.12 (1973), pp. 1045–1047.

[10] J.M. Berger. "A note on error detection codes for asymmetric channels". In: *Information and Control* 4.1 (1961), pp. 68–73. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(61)80037-5.

[11] D. Bernick et al. “NonStop/spl reg/ advanced architecture”. In: *2005 International Conference on Dependable Systems and Networks (DSN' 05)*. 2005, pp. 12–21.

[12] T. D. Bissett et al. *Loosely-Coupled, Synchronized Execution*. United States Patent 5.896.523, 1999.

[13] Bose and Der Jei Lin. “Systematic Unidirectional Error Detecting Codes”. In: *IEEE Transactions on Computers* C-34.11 (1985), pp. 1026–1032.

[14] L. Breveglieri, P. Maistri, and I. Koren. “A Note on Error Detection in an RSA Architecture by Means of Residue Codes”. In: *Proceedings of the 12th IEEE International Symposium on On-Line Testing*. IOLTS '06. USA: IEEE Computer Society, 2006, pp. 176–177. ISBN: 0769526209. DOI: 10.1109/IOLTS.2006.8.

[15] C. Chen et al. “Xuantie-910: A Commercial Multi Core 12 Stage Pipeline Out-of-Order 64 bit High Performance RISC-V Processor with Vector Extension : Industrial Product”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 52–64.

[16] J. J. Clement. “Electromigration modeling for integrated circuit interconnect reliability analysis”. In: *IEEE Transactions on Device and Materials Reliability* 1.1 (2001), pp. 33–42.

[17] *CORE-V Verification Environment*. URL: https://core-v-docs-verif-strat.readthedocs.io/en/latest/corev_env.html.

[18] Berkeley CS Division EECS Department University of California. *The RISC-V Instruction Set Manual privileged Architecture*. May 2017.

[19] Berkeley CS Division EECS Department University of California. *The RISC-V Instruction Set Manual Unprivileged ISA*. July 2020.

[20] *cv32e40p*. URL: https://github.com/openhwgroup/cv32e40p.

[21] *cv32e40p CSR*. URL: https://cv32e40p.readthedocs.io/en/latest/control_status_registers/.

[22] *cv32e40p Documentation*. URL: https://github.com/openhwgroup/core-v-docs/tree/master/cores/cv32e40p.

[23] *cv32e40p FPU*. URL: https://cv32e40p.readthedocs.io/en/latest/fpu/.

[24] *cv32e40p fpu*. URL: https://github.com/pulp-platform/fpnew.

[25] *cv32e40p Introduction*. URL: https://cv32e40p.readthedocs.io/en/latest/intro/.

[26] *cv32e40p Pipeline*. URL: https://cv32e40p.readthedocs.io/en/latest/pipeline/.

[27] Luigi Dadda. "Some schemes for parallel multipliers". In: *Alta frequenza* 34 (1965), pp. 349–356.

[28] D. Das and N. A. Touba. "Synthesis of circuits with low-cost concurrent error detection based on Bose-Lin codes". In: *Proceedings. 16th IEEE VLSI Test Symposium (Cat. No.98TB100231)*. 1998, pp. 309–315.

[29] P. Davide Schiavone et al. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications". In: *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2017, pp. 1–8.

[30] Advanced Micro Devices. *Revision Guide for AMD Athlon 64 and AMD Opteron Processors*. July 2009.

[31] E. N. Elnozahy and W. Zwaenepoel. "Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit". In: *IEEE Transactions on Computers* 41.5 (1992), pp. 526–531. DOI: 10.1109/12.142678.

[32] *EMBC Coremark*. URL: https://www.eembc.org/.

[33] A. H. Fischer et al. "Electromigration failure mechanism studies on copper interconnects". In: *Proceedings of the IEEE 2002 International Interconnect Technology Conference (Cat. No.02EX519)*. 2002, pp. 139–141.

[34] P. Forin. "IFA-GCCT". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (1989), pp. 79–84.

[35] Philippe Forin. "Vital coded microprocessor principles and application for various transit systems". In: *IFAC Proceedings Volumes* 23.2 (1990), pp. 79–84.

[36] R. Forsati et al. "A Fault Tolerant Method for Residue Arithmetic Circuits". In: *2009 International Conference on Information Management and Engineering*. 2009, pp. 59–63.

[37] M. Gautschi et al. "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2700–2713.

[38] Brian T Gold et al. *The granularity of soft-error containment in shared-memory multiprocessors*. Tech. rep. 2006.

[39] S. S. Gorshe and B. Bose. "A self-checking ALU design with efficient codes". In: *Proceedings of 14th VLSI Test Symposium*. 1996, pp. 157–161.

[40] Hao Dong. "Modified Berger Codes for Detection of Unidirectional Errors". In: *IEEE Transactions on Computers* C-33.6 (1984), pp. 572–575.

[41] Gavin Holl and Dhiraj Pradhan. "Fault Tolerant Multiprocessor Systems". In: (July 2002).

[42] IEC. *Overview Report*. 2006.

[43] Intel. *Intel Pentium 4 Processor on 90 nm Process Specification Update*. Sept. 2006.

[44] B. W. Johnson. "Fault-Tolerant Microprocessor-Based Systems". In: *IEEE Micro* 4.6 (1984), pp. 6–21.

[45] Barry W. Johnson. *Design & Analysis of Fault Tolerant Digital Systems*. USA: Addison-Wesley Longman Publishing Co., Inc., 1988. ISBN: 0201075709.

[46] E. P. Kim and N. R. Shanbhag. "Soft N-Modular Redundancy". In: *IEEE Transactions on Computers* 61.3 (2012), pp. 323–336.

[47] R. Leveugle et al. "Statistical fault injection: Quantified error and confidence". In: *2009 Design, Automation Test in Europe Conference Exhibition*. 2009, pp. 502–506. DOI: 10.1109/DATE.2009.5090716.

[48] H. M. Levy et al. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor". In: *23rd Annual International Symposium on Computer Architecture (ISCA'96)*. 1996, pp. 191–191.

[49] J. Li and E. E. Swartzlander. "Concurrent error detection in ALUs by recomputing with rotated operands". In: *Proceedings 1992 IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*. 1992, pp. 109–116.

[50] FUJITSU LIMITED. *SPARC64 V Processor For UNIX Server*. 2004.

[51] B. P. Linder et al. "Growth and scaling of oxide conduction after breakdown". In: *2003 IEEE International Reliability Physics Symposium Proceedings, 2003. 41st Annual*. 2003, pp. 402–405.

[52] S. Liu et al. “Reduced Precision Redundancy for Reliable Processing of Data”. In: *IEEE Transactions on Emerging Topics in Computing* (2019), pp. 1–1.

[53] J. -. Lo, S. Thanawastien, and T. R. N. Rao. “Concurrent error detection in arithmetic and logical operations using Berger codes”. In: *Proceedings of 9th Symposium on Computer Arithmetic*. 1989, pp. 233–240.

[54] J. -. Lo et al. “An SFS Berger check prediction ALU and its application to self-checking processor designs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11.4 (1992), pp. 525–540.

[55] R. E. Lyons and W. Vanderkulk. “The Use of Triple-Modular Redundancy to Improve Computer Reliability”. In: *IBM Journal of Research and Development* 6.2 (1962), pp. 200–209.

[56] A. Maamar and G. Russell. “A 32 bit RISC processor with concurrent error detection”. In: *Proceedings. 24th EUROMICRO Conference (Cat. No.98EX204)*. Vol. 1. 1998, 461–467 vol.1.

[57] A. Mahmood and E. J. McCluskey. “Concurrent error detection using watchdog processors-a survey”. In: *IEEE Transactions on Computers* 37.2 (1988), pp. 160–174.

[58] D. Mandelbaum. “Arithmetic codes with large distance”. In: *IEEE Transactions on Information Theory* 13.2 (1967), pp. 237–242.

[59] T. Marena. “RISC-V: high performance embedded SweRV™ core microarchitecture, performance and CHIPS Alliance”. In: 2019.

[60] J. J. Massey. “Survey of residue coding for arithmetic errors”. In: *ICC Bulletin* 3 (1964), pp. 195–209.

[61] “CHAPTER 7 - Hardware Error Recovery”. In: *Architecture Design for Soft Errors*. Ed. by Shubu Mukherjee. Burlington: Morgan Kaufmann, 2008, pp. 253–295.

[62] H. T. Nguyen and A. Chattejee. “Number-splitting with shift-and-add decomposition for power and hardware optimization in linear DSP synthesis”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8.4 (2000), pp. 419–424. DOI: 10.1109/92.863621.

[63] M. Nicolaidis. “Carry checking/parity prediction adders and ALUs”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11.1 (2003), pp. 121–128.

[64] Patel and Fung. "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands". In: *IEEE Transactions on Computers* C-31.7 (1982), pp. 589–595.

[65] D. A. Patterson and K. Asanović. *Instruction Sets Should Be Free: The Case For RISC-V*. Aug. 2014.

[66] M. Potkonjak, M. B. Srivastava, and A. P. Chandrakasan. "Multiple constant multiplications: efficient and versatile framework and algorithms for exploring common subexpression elimination". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 15.2 (1996), pp. 151–165. DOI: 10.1109/43.486662.

[67] M. D. Powell and T. N. Vijaykumar. "Pipeline damping: a microarchitectural technique to reduce inductive noise in supply voltage". In: *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* 2003, pp. 72–83.

[68] *PULP platform*. URL: https://pulp-platform.org.

[69] *PULP-Platform Simulation Verification*. URL: https://core-v-docs-verif-strat.readthedocs.io/en/latest/pulp_verif.html.

[70] T. R. N. Rao. "Biresidue Error-Correcting Codes for Computer Arithmetic". In: *IEEE Transactions on Computers* C-19.5 (1970), pp. 398–402.

[71] Thammavarapu R. N. Rao. *Error Coding for Arithmetic Processors*. USA: Academic Press, Inc., 1974. ISBN: 0125807503.

[72] *RISC-V*. URL: https://riscv.org.

[73] *RISCV BOOM Coremark*. URL: https://github.com/riscv-boom/riscv-coremark.

[74] R. Rodriguez, J. H. Stathis, and B. P. Linder. "Modeling and experimental verification of the effect of gate oxide breakdown on CMOS inverters". In: *2003 IEEE International Reliability Physics Symposium Proceedings, 2003. 41st Annual.* 2003, pp. 11–16.

[75] E. Rotenberg. "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors". In: *Digest of Papers. Twenty Ninth Annual International Symposium on Fault Tolerant Computing (Cat. No. 99CB36352).* 1999, pp. 84–91.

[76] U. Schiffel et al. "Software-Implemented Hardware Error Detection: Costs and Gains". In: *2010 Third International Conference on Dependability.* 2010, pp. 51–57.

[77] E. Schuchman and T. N. Vijaykumar. "BlackJack: Hard Error Detection with Redundant Threads on SMT". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 2007, pp. 327–337.

[78] Frederick F Sellers, Hsiao Mu Yue, and Leroy W Bearnson. "Error detecting logic for digital computers". In: (1968).

[79] K. G. Shin and Hagbae Kim. "A time redundancy approach to TMR failures using fault-state likelihoods". In: *IEEE Transactions on Computers* 43.10 (1994), pp. 1151–1162.

[80] P. Shivakumar et al. "Modeling the effect of technology trends on the soft error rate of combinational logic". In: *Proceedings International Conference on Dependable Systems and Networks*. 2002, pp. 389–398.

[81] Smitha Shyam et al. "Ultra Low-Cost Defect Protection for Microprocessor Pipelines". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2006, pp. 73–82. DOI: 10.1145/1168857.1168868.

[82] K. Skadron et al. "Temperature-aware microarchitecture". In: *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* 2003, pp. 2–13.

[83] J. C. Smolens et al. "Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures". In: *37th International Symposium on Microarchitecture (MICRO-37'04)*. 2004, pp. 257–268.

[84] Jared C. Smolens et al. "Detecting Emerging Wearout Faults". In: 2007.

[85] Daniel Sorin. *Fault Tolerant Computer Architecture*. Morgan Claypool, 2009.

[86] J. Srinivasan et al. "The impact of technology scaling on lifetime reliability". In: *International Conference on Dependable Systems and Networks, 2004*. 2004, pp. 177–186.

[87] D. Sylvester, D. Blaauw, and E. Karl. "ElastIC: An Adaptive Self-Healing Architecture for Unpredictable Silicon". In: *IEEE Design Test of Computers* 23.6 (2006), pp. 484–490.

[88] A. Tiwari and J. Torrellas. "Facelift: Hiding and slowing down aging in multicores". In: *2008 41st IEEE/ACM International Symposium on Microarchitecture*. 2008, pp. 129–140.

[89] V. Tiwari, S. Malik, and P. Ashar. "Guarded evaluation: pushing power management to logic synthesis/design". In: *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* 17.10 (1998), pp. 1051–1060.

[90] J. F. Wakerly. "Partially Self-Checking Circuits and Their Use in Performing Logical Operations". In: *IEEE Transactions on Computers* C-23.7 (1974), pp. 658–666.

[91] C. S. Wallace. "A Suggestion for a Fast Multiplier". In: *IEEE Transactions on Electronic Computers* EC-13.1 (1964), pp. 14–17. DOI: 10.1109/PGEC.1964.263830.

[92] P. M. Wells, K. Chakraborty, and G. S. Sohi. "Adapting to Intermittent Faults in Future Multicore Systems". In: (2007), pp. 431–431.

[93] J. F. Ziegler. "Terrestrial cosmic ray intensities". In: *IBM Journal of Research and Development* 42.1 (1998), pp. 117–140.

[94] J. F. Ziegler et al. "IBM experiments in soft fails in computer electronics (1978–1994)". In: *IBM Journal of Research and Development* 40.1 (1996), pp. 3–18.