POLITECNICO DI TORINO

Master's Degree in Embedded Systems

Master's Degree Thesis

Feasibility study of the Software-Based Self Test methodology on a Hardware Accelerator for Neural Networks

Prof. Paolo BERNARDI

Prof. Riccardo CANTORO

Dr. Olivier MONTFORT

Lorenzo ZAIA

December 2020

Summary

The goal of the following thesis is to demonstrate that the Software-Based Self Test can be adopted on dedicated Multi-Cores architectures for the implementation of Neural Networks. The first part of the paper will deal with all the theoretical concepts required for the correct understanding of all the theoretical aspects. Notions will be provided on the topics of Testing and Fault Tolerance, Neural Networks, and Multi-Processor Systems concluding with an accurate description of the RISC-V ISA. The latter design was adopted by the company for the implementation of the VEP Design, an hardware accelerator for DSP and Neural Networks. In the second part, a description of the entire SoC will follow with an explanation regarding the organization of the memories, which will be one of the main aspects to consider for the correct functioning of the test software. Successively, the algorithms and the workflow used by the test programs to reach an acceptable Fault Coverage value will be described, concerning the different strategies adopted by the various EDA Tools to obtain and validate the results. The last part will include an explanation of the entire software architecture implemented to orchestrate the whole test suite. Therefore, concluding with the results obtained and further analysis.

Table of Contents

List of Tables

List of Figures

Acronyms

IP
: intellectual property

ATE
: Automatic Test Equipment

DfT
: Design for Testability

BIST
: Built-In Self-Test

UUT
: Unit Under Test

DUT
: Device Under Test

PIs
: Primary Inputs

POs
: Primary Outputs

TPG
: Test Pattern Generator

ATPG
: Test Pattern Generator

LFSR

Linear Feedback Shift Register

ODE

Output Data Evaluator

FC

Fault Coverage

SBST

Software Based Self-Test

OS

Operative System

EABI

Embedded Application Binary Interface

GPU

Graphic Processor Unit

CPU

Core Processor Unit

GPGPU

General Purpose Graphic Processor Unit

ILP

Instruction Level Parallelism

TLP

Thread Level Parallelism

NN

Neural Network

ISA

Instruction Set Architecture

RISC

Reduced Instruction Set Computer

GCC

GNU C Compiler

FPGA

Field Programmable Gate Array

ASIC

Application Specific Integrated Circuit

DMA

Direct Memory Access

SoC

System on Chip

APB

Amba Peripheral Bus

AXI

Amba eXtensible Interface

FLL

Frequency-Locked Loop

DCO

Digital Controlled Oscillator

DSP

Digital Signal Processing

VCD

Value Change Dump

eVCD

Extended Value Change Dump

Chapter

Introduction and Goal

The following feasibility study aims to demonstrate that the SBST technique can also be successfully applied on complex multi-core systems used for the implementation of neural networks and DSP computations. In this place, all the problems encountered during the experience are highlighted and possible solutions are provided. The results will be demonstrated through the implementation of a first version of the test software, explaining in detail the whole workflow adopted for its development. The study shows how the methodology has heavy limitations due to the demanding resources to simulate large sequential multi-core circuits proposing, a first solution able to give an effective indication of the fault coverage on the targeted modules of the design.

Chapter 1

Testing and Fault Tolerance

1.1 Concept of Dependability

It happened to everyone to go to a store to buy an electronic product and be undecided about which model to take. It has often happened to decide to buy a more expensive version of the latter in the hope that it will last longer in time. In general, it's possible to say that, the durability over time is, therefore, a key parameter to determine the quality of a product.

The **Dependability** is the property characterizing a dependable system. It is defined as "*the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers* ".[1] [2]
Three different aspects allow to determine and influence the degree of dependability of a system:

- **Attributes** - *Reliability, Availability, Safety, Integrity, Confidentiality, Maintainability, Testability*
- **Threats** - *Faults, Errors, Failures*
- **Means** - *Prevention, Tolerance, Removal, Forecasting*

1.1.1 Attributes

The probability of a system to provides its functionality correctly at the end of a defined time is defined as **Reliability**. In general, the metric used to quantify the reliability is commonly defined as **Mean Time to Failure** (MTTF) measured, obviously, in hours. Another metric, widely used, is the **Failure Rate** denoted as number of failures over a period.
Taken a defected system, the probability to fix it within a period of time is called

maintanability. In this case, the metric adopted is the *Mean Time to Repair* (MMTR) defined as the average time required to fix the system.
Similarly to the reliability, the **availability** is the probability of performing the operation at a generic fixed time T and therefore, without considering time periods. The **integrity** defines a system in absence of improper alteration while the **confidentiality** its related to the absence of unauthorised disclosure of information. The **Safety** defines a system capable to behave correctly in presence of a defect and/or interrupt its operations without causing serious damages. To assesses the seriousness of possible misbehaviours it's necessary to perform some preliminary studies, the *Failure Mode and Effect Analysis* (FMEA) and the *Fault Tree Analysis*. The **testability** describes the easiness of a system to be tested to detect possible faults taking into account the possibles effects on Reliability and Availability.

1.1.2 Threats

The **threats** are phenomena that can affect the system and affect the dependability of a systems as:

- **Fault**
- **Errors**
- **Failures**

Figure 1.1: From fault to failure UML

A **Fault** is a defect in the system and its presence can lead or not towards a failure. An **Error** is an internal deviation of the behavior of the system. This is caused by the entering in an illegal state of some modules affected by one or more faults. Since, an error is an internal discrepancy it could be propagated, generating a failure, or not. The **Misbehaviour (Failure)** determines a different function with respect the designed one. A failure can be masked by the presence of fault tolerance techniques and in this case the system can be defined as **Fault Tolerant**.

1.1.3 Means

The industries decided to adopt a standard process for each development phase to ensure dependable systems.
A common example is the **ISO26262** for the automotive industry. This standard

defines a set of clearly guidelines that are required during the whole development flow, involving hardware and software aspects.
The techniques adopted are normally classified in four categories:

- **Fault Prevention**
- **Fault Removal**
- **Fault Tolerance**
- **Fault Forecasting**

All the different approaches are *complementary* of each other.
The **Fault Prevention** can be performed through an accurate development methodology and with a carefully modules integration preventing that the faults will be incorporated in the system.
The **Fault Removal** aims to detect and remove many as possible faults from the system. As said before, this operation is preferable during the development phase to reduce the fixing costs.
The **Fault Tolerance** deals to inject mechanism into the defected system in order to still provide the correct functionalities.
The **Fault Forecasting** is the faults prediction so that their effect can be masked or removed.

1.2 Failures and Faults

1.2.1 Failures Classification

Failures can be classified according to the different misbehaves generated.

- Static or Dynamic
- Permanent or Transient
- Consistent or Inconsistent
- Benign, Serious or Catastrophic

A failure is considered as **static** if the results produced are wrong every time the operation is performed. Inversely, failures that provide correct results but with wrong timings are considered as **dynamic**.
The duration of failure can be **permanent**, if it is significant concerning the mission duration, or **transient**. The third aspect of failure classification regards how they are perceived. In case the experienced effects are the same for all the users, the

misbehaviour is **consistent**, contrarily, as **inconsistent**.
The last perspective permits to identify the failures considering their gravity of effect. In the case of negligible effects, the failure is considered as **benign**. More critical is a **serious** failure that affects the mission of the system even if it can continue the latter. Differently, if the mission is interrupted it is classified as **catastrophic**.

1.2.2 Faults Classification

As said before, a failure is caused by the presence of faults. It's possible to classify them accordingly:

- Fault Origin
  - Location
  - Development phase
  - Cause
- Fault Nature
  - Type
  - Intentionality
  - Duration

A fault can arise by **internal** defects or by some **external** perturbations by functional environment or wrong usages by the user. It can arise during each development phase. Considering the latter, if a fault is detected before the operation phase it is named *creation fault*. The last aspect to determine the origin of a fault concerns the causes that generated it. A common example is a fault caused by technology during the device production phase.

To characterize the nature of a fault is necessary to identify the type. The different fault type are **functional** or **technological**. A functional fault relates to how the device has been designed, specified, or used. This topology of fault is called *systemic*. A technological fault relates to the implementation of the system during the production phase and, it is identified as *disruptive*. Sometimes, a defect is injected in the system to achieve a specific goal, in this case, the fault is **intentional** otherwise it is **accidental**. Also for the faults exist the concept of duration. They can be **temporary** or **permanent**. A temporary fault can be divided into two subclasses, *transient* if it disappears after some time or *intermittent* if it never disappears completely.

1.2.3 Faults Propagation, Lifecycle and Latency

When a fault arises inside in the system it may affect others modules causing errors and/or misbehaviour over the whole system. The analysis of the **propagation path** is a crucial aspect to take into account to assess the dependability of the product.
The **life-cycle** of a fault across three possible phases:

- **Phase I**: it is present in the system but it never produces any effect. In this case, the fault is defined as *dormant* or textitpassive
- **Phase II**: it is *active* and it's possible to detect some error in the system
- **Phase III**: it produces failures so, it is *propagated*

Before to see failures, due to a fault propagation, some time occurs. This time is defined as **latency**. To characterize the latency it's fundamental to evaluate the *module containing the fault*, *the time of occurrence*, the *usage* and the *observation level*. The frequency at which a module is used permits to clarify better which is the latency as it is the usage of its results.

Furthermore, it's reasonable to determine the propagation of a failure for what concern the consequences on the mission, the metric is called **inertia**. Higher is the value of inertia better is visible the fault propagation and so, its detection. This means that it is correlated to a lower value of latency. The worst case is when the latency is high and the inertia is low; in this situation, the fault detection is difficult since it is "hidden" inside the system. The **figure 1.3** explains better the concepts of Latency and Inertia.

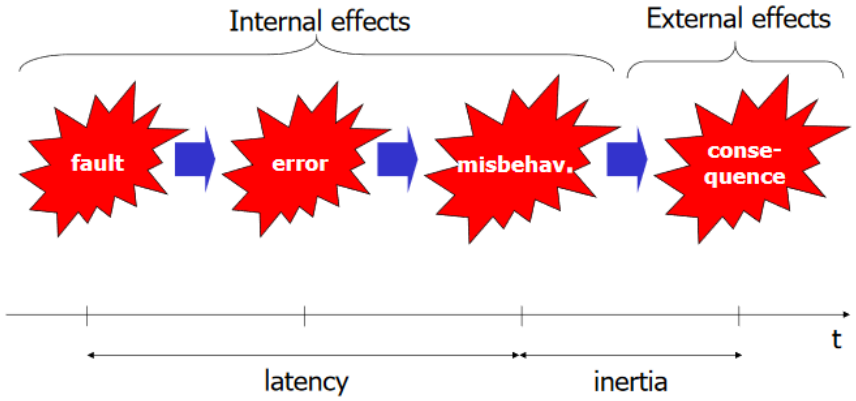

Figure 1.2: Latency and Inertia

1.3 Testing

The **test** is a process aiming to identify as much as possible the defects of a system. This procedure is employed to distinguish *faulty products* between *good products*. An overview of the test application phase is in **figure 1.4**. During this phase, the responses provided by the DUT are compared with those expected from a faulty-free device. The test application is executed in different ways according to

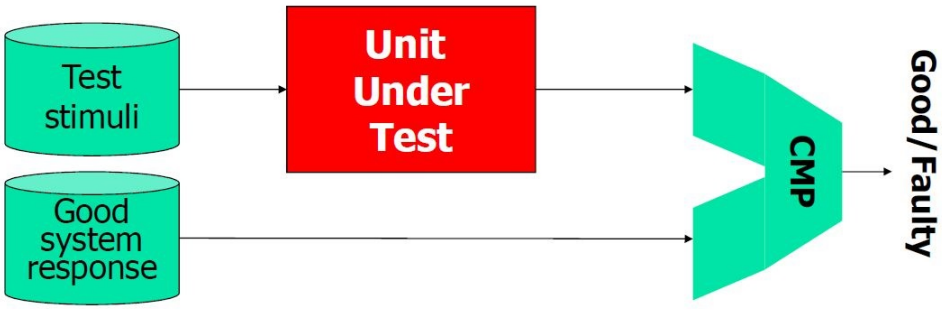

Figure 1.3: Test Application

the development phase. The test is performed through specific inputs aimed at highlighting discrepancies in the results when there is a defect in the system, the latter are specifically identified before the production phase. The metric adopted to assess the quality of a test is the **defect level.** This parameter gives the percentage of the faulty product and it is usually expressed as *part per million (ppm)* or *part per billion (ppb)*. After the manufacturing process, the test application permits the assessment of the product quality. In some cases, faulty products can be labeled as good and, unfortunately, they are delivered to the customers. Contrarily, some good products can be classified as faulty and discarded even if they are good, this issue is defined as **overtesting**. Analytically, the defect level can be expressed by the equation 1.1.

$$DL = 1 - Y^{(1-T)} \tag{1.1}$$

where:

- DL is the defect level
- Y is the manufacturing process Yield
- T is the fault coverage

Identifying the correct **Fault Coverage** (FC) is a key aspect of all the application domains. The yield is the percentage of the products that have passed all the

manufacturing processes. It's important to highlight that the yield can be improved during all the life cycle of the product because the manufacturing process can be tuned and optimized.

1.3.1 Test Types

During the development is possible to apply several type of testing methodologies. Each type of test is aimed at validating each different stage of development. The common techniques widely used are the following:

- Verification Testing
- Production Testing
- Burn-In
- System Level Test
- Incoming Inspection
- On-line Testing

Verification Testing

The verification testing aims to validate the correctness and compliance concerning the device specifications. It is involved to find also the operating limits of the circuit in terms of voltage, temperature maximum frequency. This type of test is performed before the mass production of the system and it is also called *design debug*.

Production Testing

The goal of this kind of test is to guarantee the correct behavior of the produced devices. Normally is composed by two phases, the *parametric test* and *functional test*. The parametric test is performed to the circuit applying different frequencies and current values. Doing this procedure is possible to highlight if some technical parameters fall into unexpected intervals by the presence (or not) of physical defects. It's mandatory to repeat this test every time a new technology is adopted. The functional test is used to check the functionalities of the system applying stimuli that are used during the normal functioning of the system. It can include some patterns used during the verification testing and this procedure is technology independent.

Burn-In

The idea is to stress the circuit under extreme conditions in terms of temperature, voltages, etc. In this context, the circuit ages in a way similar to that produced by time. Often, this test is applied to all newly printed circuits identifying the possible subjects to *infant mortality*. Furthermore, this stress test permits to evaluate the reliability of a restricted number of printed circuits. This practice is applied for those circuits designated for particular applications as space missions and nuclear power plants.

System Level Test

This methodology is performed including all the system modules under operative conditions. In this phase, newly defects can arise and, consequently, generate unexpected behaviours.

Incoming Inspection

This check is performed by the buyer and it aims to evaluate that the circuit works correctly before mounting it on another board. Sometimes, the buyer doesn't know all the internal design since it could be an intellectual property (IP).

On-line Test

This practice is widely adopted when the circuit is used in critical missions and it requires, continuously, testing operations aimed to evaluate the reliability of the circuit. In general, it is done through specific software able to activate dedicated test modules inside the circuit keeping the circuit under its operational environment. The On-line test is also called *in-field test*. The test can be executed *concurrently* during the normal operations or *non-concurrently*. The interesting aspect of this methodology is to decide when it's better to perform the test. In many applications the system cannot be powered off therefore, the idea is to guarantee the Fault Coverage inside a defined time interval evaluating, for instance, possibles idle times of the operative system.

1.4 Test economics

Even thought, the cost per transistor is reduced scaling the technology, its testing costs are rising since that they became more complex and sophisticated.[3] Nowadays, a net percentage of development costs is determined by the presence of different testing methodologies. As it possible to see in **figure 1.4** the testing

phase, of a software application, has the same impact of all the other development phases (excluding maintenance).

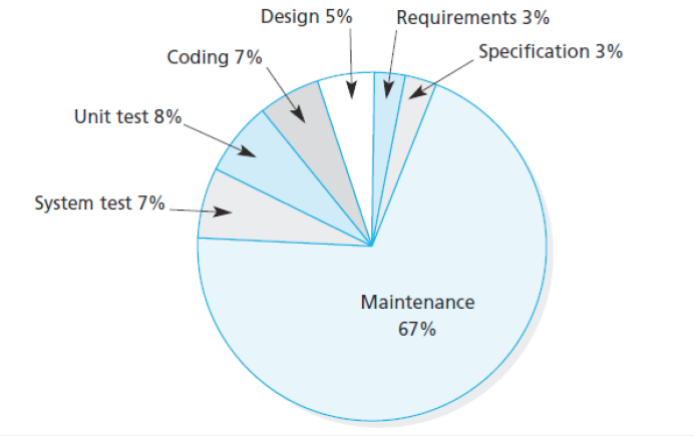

Figure 1.4: Costs of Development stages for software application (2005, Bell)

1.4.1 Effects on Design Flow

During the development of an electronic device it is necessary to pay attention to the testing procedures adopted especially during the very first stages of development.As shown in **Figure 1.5**, the costs of fixing any defects increase exponentially in the subsequent development phases.

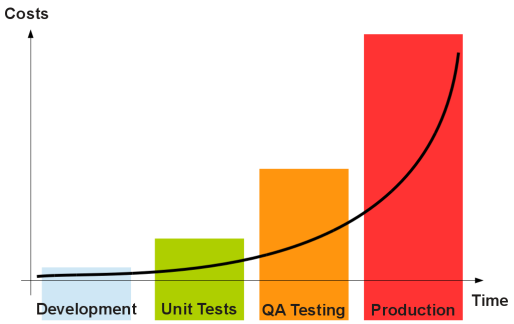

Figure 1.5: Time delay detection / Fix cost

For this reason the development flow has been highly modified introducing validation phases after each development stage. These modifications are shown in **Figure 1.6**.

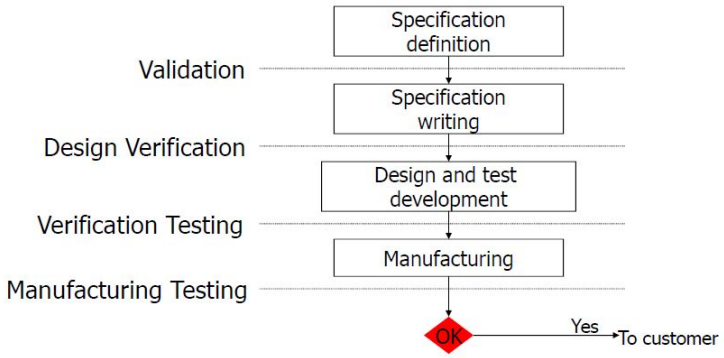

Figure 1.6: Development flow with validation phases

1.4.2 Technology Evolution

As said before, the technology scaling has increased a lot the cost of the testing procedures. The complexity of the test procedures are affected by three main parameters:

- Working Frequency
- Density
- Analog and Digital modules integration

Rising the working frequency of a system implies an increase in the difficulty to perform *at-speed* tests. Moreover, for operative frequency higher than 1 GHz the electromagnetic interference becomes an important effect to take into account since, it can enhance the number of possible defects. A higher density of gates inside the circuit provokes a reduction of accessible testing points while, for the same reason, the possibility to generate defects due to power and temperature dissipation is increased. A complex device that mixes analog and digital devices can be harder to test due to different testing methodologies.
All those aspect are reflected in the cost for the Automatic Test Equipment (ATE). The cost of this kind of device is highly dependent on the working frequency, number of test pins and in the amount of memory required by the testing application.

1.5 Design For Testability

As anticipated in section 1.4.1, sometimes is convenient and necessary to modify the intern structure of the circuit to increase the testability of the circuit. This approach is called **Design for Testability** (DfT) and it is used in practice by all the designers of digital devices. In general, two techniques are widely adopted:

- Scan Test
- Built-in Self-Test (BIST)

1.5.1 Scan Test

The basic idea is to generate a sequence of patterns, this can be easily done for combinational circuits but is impractically impossible for the sequential ones. To solve the issue, the basic idea is to adopt a new type of flip-flop called *scan flip-flop* that adds the possibility to *scan-in* and *scan-out* its value when the test is enable. In this way, all the sequential blocks are separated into small combinational ones. The modification is visible in **Figure 1.7**. The structure of a scan-FF is shown in

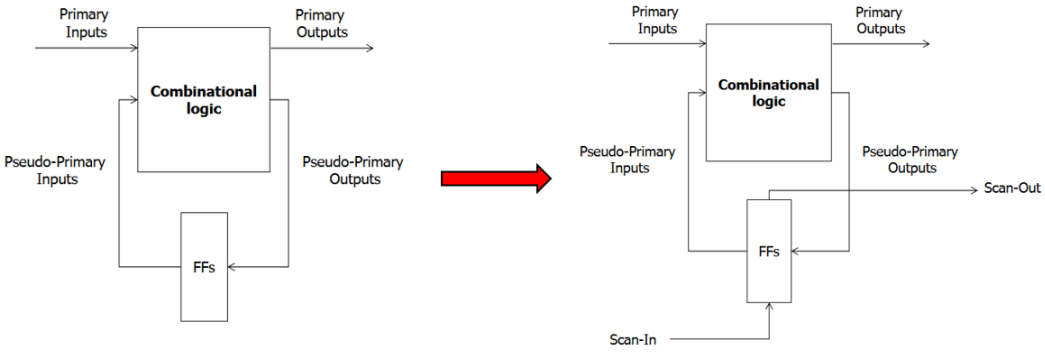

Figure 1.7: Scan Test

Figure 1.8.

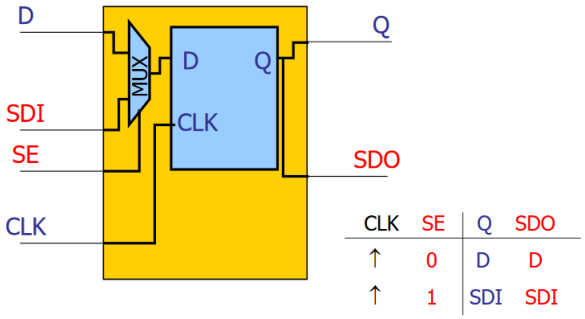

Figure 1.8: Scan Test

Typically, the test operation is performed in four steps:

1. Upload Scan-FF serially with known values
2. Apply the known values to the combinational logic
3. Store the output into the scan-FF
4. Shift out the captured value

In a real system the scan FF are organized in *scan-chains* and their length its a crucial aspect to take into account, since, they directly affect the duration of the scan-in and scan-out phases. For technical reasons, sometimes, the two scan phases are performed at different clock speed with respect to the operational one of the circuit. This means that the handle of the signals overhead could be a problem especially in presence of another clock. This technique can provide optimal results in terms of fault coverage but, due to the application of not functional patterns, it can produce overtesting. Another aspect to consider, it's the necessity to reproduce a safe context during the scan phases because, during those operations, the system is fed with random values that can damage the circuit permanently.
In a real board with many modules developed by different companies the scan signals are integrated leveraging the capabilities provided by the JTAG interface and its TAP Port.

1.5.2 Built-in Self-Test

The BIST technique leverage the concept of built-in ATEs. The goal is to perform at-speed test guaranteeing a good fault coverage giving also the possibility to test modules that are embedded in the circuit. The basic scheme of the BIST circuitry is in **Figure 1.9.**

The basic blocks of the BIST architecture are:

- The **Unit Under Test** (UUT) is the part of the circuitry tested by the BIST. It could be a sequential or a combinational block. Sometimes, the UUT is also memory or an analog circuit. The observable points of the UUT are its respective PIs and POs.
- The **Test Pattern Generator** (TPG) is a dedicate circuit dedicated to generating the test stimuli for the UUT. Those patterns can be generated randomically, through an LFSR, or deterministically.
- The **Multiplexer** is in charge to select the normal UUT inputs with the ones generated by the TPG.
- The **Output Data Evaluator** (ODE) compares the POs responses, obtained in test mode, with the expected ones.

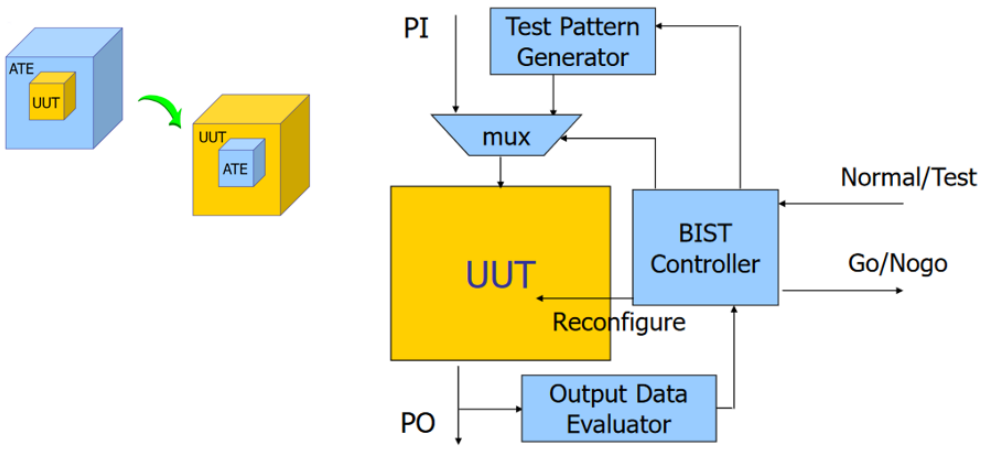

Figure 1.9: BIST Architecture

- The **BIST Controller** is the logic unit that controls the execution of the test managing the TPG, the ODE, the MUX and the UUT. It is controlled by the normal/test signal and drives the go/nogo signal.

The **Normal/Test** signal is in charge to activate enable the test for the UUT. In test mode the **reconfigure** signal is asserted to enable some features, if present, within the UUT to increase its testability. The **Go/Nogo** signal is used to declare the result of the test outside the module.

The test procedure across several phases.

1. The Normal/Test signal is switched on test mode. This action is performed internally, in the same board, by another module or externally, by an ATE or from the Boundary Scan Interface (JTAG).

2. The BIST controller sets the multiplexer and enables the TPG to generate the test stimuli feeding the UUT.

3. The BIST controller activates the ODE to analyze the POs.

4. The responses, provided by the ODE, are compared with respect to the gold machine. At this point, the BIST controller performs some decisions to evaluate the result of the test and asserts the Pass/failure flag.

5. The BIST controller, according to the internal flag, drives the GO/Nogo signal.

From the external point of view, the whole test session is observed using specific protocols, in a way quite similar to a UART protocol. In this case, the signals

that regulate the protocol are Normal/Test and Go/Nogo. Thanks to the use of more advanced protocols it is also possible to identify possible faults on the BIST controller.
In conclusion, this technique is optimal and widely used, especially inside the memories. Some critical aspects arise due to the area overhead and, consequently, by the power consumption that can cause radical changes in the circuit design.

1.6 Faults Models

A product can be affected by a large number of possibly defects of different nature. For this reason, its useful to model all the faults with share the same properties. The commons used models are:

- Stuck-At
- Short (or Bridge) Open circuit
- Transistor Stuck-on/off
- Delay Fault
- Single Event effect

Those models permit to describe, with a high level of accuracy, a large scale of possible physical defects. In this way, the physical defect is translated into logical faults which are more meaningful under the point of view of the system functionality.

1.6.1 Stuck-At

The **stuck-at** model is widely adopted in almost all domains due to its simplicity. In many situations is the first model used to develop testing techniques. It consists to represent a signal to a fixed value at a high o low level. Considering the gate netlist in **figure 1.10**.

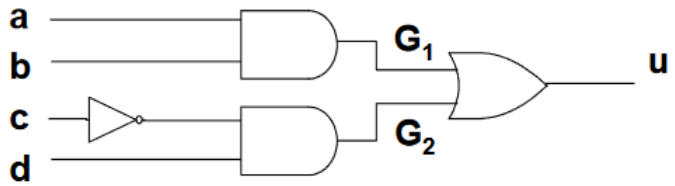

Figure 1.10: Netlist Example

A possible fault, can affect the wire D "stucking" its value at 1 as shown in **figure 1.11**

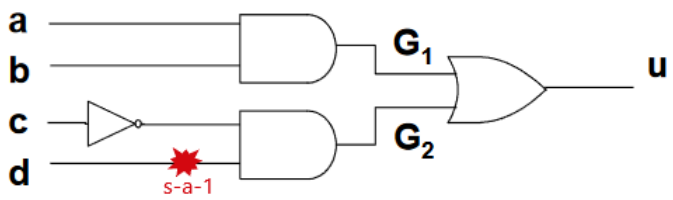

Figure 1.11: Netlist Example

To detect this kind of logic fault it's necessary to find some inputs combination (pattern) able to **excite** and **propagate** the defect to, at least, one output ports. Sometimes, find an input pattern is not trivial and it is highly related to the netlist structure. The presence of *reconvergent fanout* could make it difficult to find the pattern since the fault effect could be masked. A high-level example of a reconvergent network is shown in **figure 1.12**. In this example, the fault effect propagation is masked by the result of the combinational logic C.

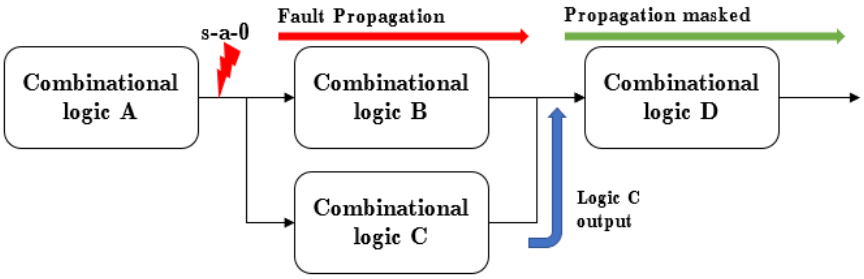

Figure 1.12: Netlist Example

The stuck-at can be used also to detect faults in sequential circuit even if is more complex. For this goal is necessary to apply a sequence of input vectors. The worst situation, in terms of pattern generation is when the memory elements belong to a cycle.
In conclusion, for what concerns the stuck-at model, it's feasible to classify it as a very effective one. It permits to enumerate the number of faults and, thanks to its simplicity, it is supported by almost all testing tools. Unfortunately, some physical defects are not covered by its modelization, this implies to adopt and take in consideration also other faults models.

1.6.2 Short (or Bridge) Open circuit

The **Short (or Bridge) Model** consider the effects of a bad connection between two nodes in the circuit. Often this defect is caused by electromigration phenomena. The electromigration is caused by the currents that flow inside a conductor. The electrons transfer their momentum to the metal atoms moving them. By this phenomena, the metal elementary structure is altered evolving, in the worst case, into short or open circuit. The value assumed by the nodes coupling may be depending by the technology and the circuit generated after the short-open defect. As instance, in case of a dominant 0-logic the behaviour of bridge-defect is equivalent to an AND-gate. Inversely, in a 1-dominant logic the presence of two bridged signals is equivalent to an OR-gate. both the examples are illustrated in **figure 1.13**.

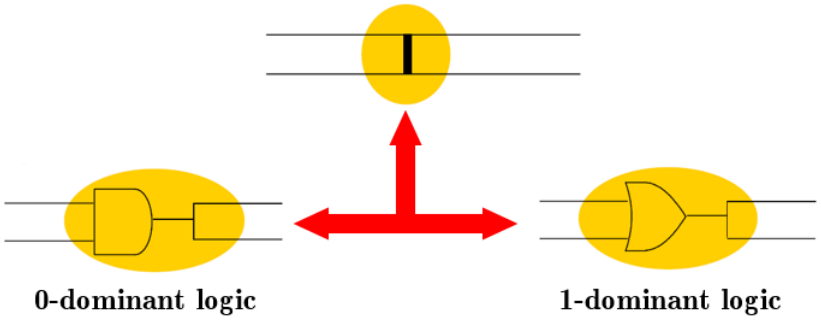

Figure 1.13: Bridge Example

The number of all possible faults in a circuit is given by the following formula:

$$\#Faults = n(n-1)/2 \tag{1.2}$$

where n is the number of nodes. It's straightforward to notify that, for a large circuit handle this fault model is impossible and meaningless. The practical approach is to consider restricted circuit areas where their occurrence probability is higher, evaluating topological design aspects. For these reasons, this fault model is widely adopted to perform circuit analysis at board level.

1.6.3 Transistor Stuck-On/Off

This model operates on the transistor description level of the circuit. This kind of interpretation highlight the possibility to have a transistor continuously in conduction or interdiction independent by the value of the drain-source voltage. An example of stuck-on transistor is in **figure 1.14**

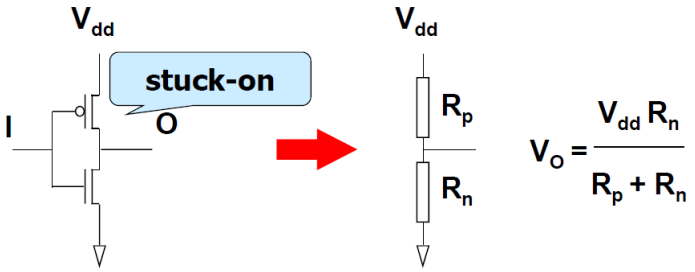

Figure 1.14: Stuck-On transistor Example

This abstraction overcomes some of the limitations of the stuck-at models. Since that it required very low-level circuit description the number of possible faults is, also in this case, too high. Moreover, the access to this kind of circuit description is not trivial especially for big SoCs that count hundreds millions of transistors.

1.6.4 Delay Fault

The **delay fault** model identifies defects that afflict the inputs and outputs transitions of the gates present in the design. The delay that affects the transitions causes an incorrect signal propagation and, consecutively, an incorrect result sample under specific clock constraints. From a technological point of view this kind of faults can be caused by a combination between parasitic capacitance with a wrong wire/interconnection sizing that affects the value of resistivity and so, the circuit timing.

From this assumptions two fault model are commonly used.

- Transition Fault model
- Path Delay Fault model

Transition Fault model

This representation assumes that the delay caused by the faults is large enough to doesn't permit the transition within the observation time. The two possible faults are *slow-to-rise* and *slow-to-fall*. The possible number of fault present in design is given by the following equation:

$$\#Faults = 2n + k \tag{1.3}$$

where n is the number of gates and k the number of inputs.
To detect this faults the following procedure is adopted:

1. Find a pattern to force the output value of the target gate to 0 or 1

2. Find a second pattern to force the gate output transition and propagate this transition to the POs

Thus, this methodology requires a *pair of vectors* applied at circuit speed clock. The **figure 1.15** shows an example of the procedure.

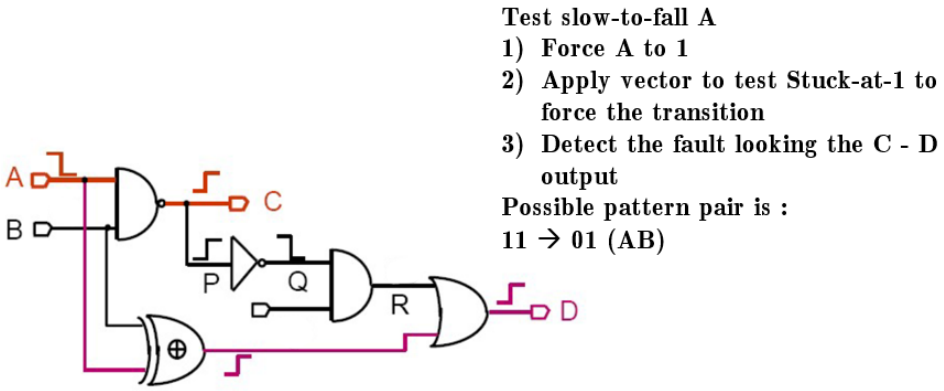

Figure 1.15: Transition fault Example

This technique is widely used by its simplicity being an extension of the stuck-at model. Some limitations are due to the presence of reconvercenge paths that mask the transition. Anyway, its reduced fault list and the possibility to identify a large number of physical defects it makes the assumption of this model very useful when developing test procedures.

Path Delay Fault model

This model considers the differences in the path delay caused by the presence of defects that afflict the slow-to-rise and slow-to-fall transitions. The number of possible faults is given by :

$$\#Faults = 2m \tag{1.4}$$

where m is the number of paths. The strategy to test the path delay between an input A and an output B is the following:

1. Activate the transition on input A

2. Find the input pattern to propagate the transition along all the target path up to B

A major issue is the large number of paths present in a real circuit. For a circuit with **n** gates, the complexity magnitude order is $O(N^2)$. Due to this fact, are taken into consideration only a subset of all the paths. A ranking mechanism, that considers the path length, is adopted to classify and determine the path subset.

The metric to properly identify the paths leverages the concept of *slack*. The slack specifies the time quantity remaining between the clock period and the delay caused by the path. This means that less is the value of slack longer is the path. Therefore, a possible fault can add a quantity of time higher than the value of the slack generating a defect.

The detection of the Delay fault is highly dependent on the structure of the circuit. Some faults can be defined as *structurally untestable* when they are not detectable by the layout of the paths. Sometimes, applying a sequence of specific input patterns is impossible due to functional limitations reducing the list of testable faults. The latter topology of untestable fault is called *functionally untestable* faults. The **figure 1.16** gives an overview of the testability hierarchy for delay faults.

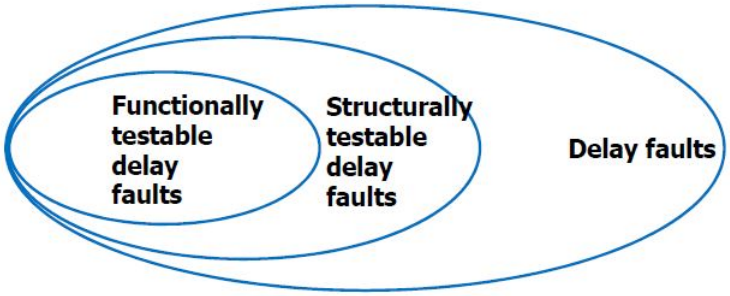

Figure 1.16: Testability Hierarchy

1.7 Test Process

Nowadays, in the market many EDA tools are designed to give a flow to develop the requested testing methodology. The goal of this **Test Process** is to provide a valid test set able to detect the faults inside the DUT, displaying the Fault Coverage score and the undetected fault list. When the required FC is satisfied, the test is ready to be applied, on the real circuit, leveraging ATE and/or DfT techniques.

The test process is summarized in **figure 1.17**.

The first phase is the **Test Generation** and it is organized mainly in three steps:

- Fault Management
- Fault Simulation
- ATPG

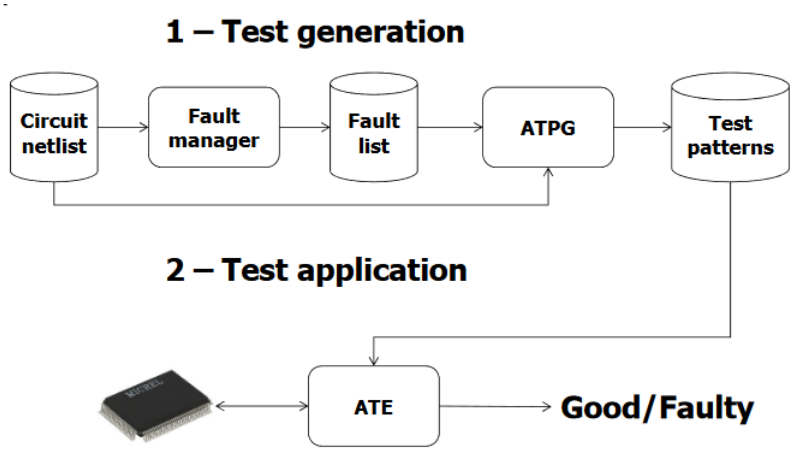

Figure 1.17: Test Process phases

1.7.1 Fault Management

The Fault manager is properly configurated to work with the correctly fault model. Successively, it is involved during the fault management phase analyzing the circuit netlist and providing the fault list. According to the fault equivalence classes the list can be collapsed into a smaller one to improve the effort required by the simulation.

During fault list generation phase, some faults can be marked as untestable due to circuit analyses on the gates that are:

- Not reachable by PIs
- Not connected to the POs
- Connected to lines stucked at fixed values

1.7.2 Fault Simulation

The fault simulation phase permits to reach different purposes in order to evaluate the effectiveness of a test set. In general, this phase aims to perform:

- Testability analysis
- Fault Coverage Computation
- Analysis of Faulty circuit Behaviour

The **testability analysis** permits to identify all the circuit areas that are lack in terms of controllability and testability. In addition, the fault simulation is often

used to verify the behavior of a circuit affected by defects. This procedure allows to classify the possible failures and to assess the risks. These operations can be done using a physical fault injection or using a dedicated software named **Fault Simulator**. The full software environment is summarized in **figure 1.18**

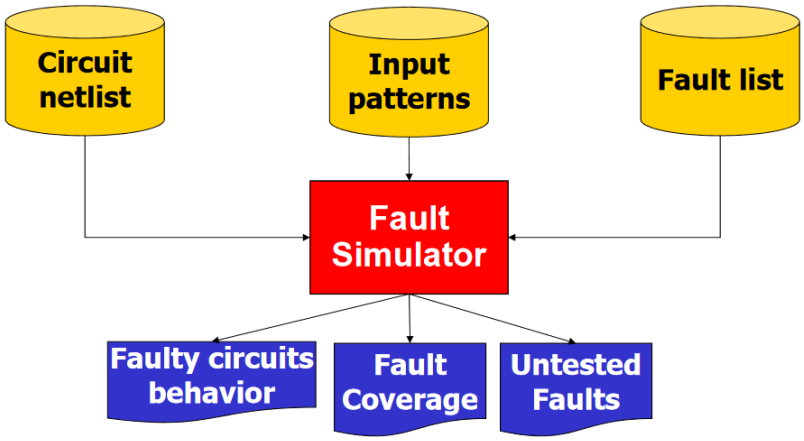

Figure 1.18: Fault Simulation Phase

The fault simulator, normally, is a SW capable to simulate the test set on a circuit netlist analyzing the fault list generated and the previous step. This tool permits to simulate the circuit behaviour injecting the fault into it. By the way, the results obtained from the faulty circuit are compared with the original one and, in case of differences on the POs, the tool marks the fault as tested or not.

1.7.3 ATPG

The **automatic test pattern generator** is a dedicated software module used to create test patterns. Due to a large number of inputs and/or to the high complexity of DUT, in many application is not easy to create the test pattern manually. This tool identifies the fault list and creates the patterns consecutively. The approach, to generate the stimuli, follows some specific algorithms. By the way, a fault can be *ATPG untestable* when the algorithm proves that is not it's not identifiable, due to the presence of inputs constraints and/or output masks. Nowadays, the ATPG constrained approach is a research field aiming about the distinguish between Functional Faults and Non-functional faults, reducing as much as possible the possibility of overtesting. For what concern the sequential circuit the test pattern generation through ATPG is a research field since that the computational cost, to find a pattern sequence with a deep of many clock cycles, is very high. Commonly, the ATPG is used in combination with the DfT techniques where a big sequential circuit is split by scan chains or bist modules. In this applications, the ATPG is

very effective because it deals with reduced combinational regions having, in theory, fewer inputs.

1.8 Software Based Self Test

The SBST is a different testing approach. The basic idea is to exercise the DUT performing a series of coded operations able to excite and propagates the fault inside the internal modules. This method has many advantages compared with classical DfT techniques. It doesn't require any hardware modification avoiding to perform adjustments in the design. An important aspect of this technique is related with the test suite organized in several test programs targeting all the internal modules. This aspects permits to develop a log system which can provide some diagnosis analysis. Since that the test procedure is simply a program, it can be performed at-speed clock and it can be launched autonomously, during the mission, leveraging the free time slots available by the OS or at the power-on/off as shown in **figure 1.19**. This methodology is widely used to identify the functional faults

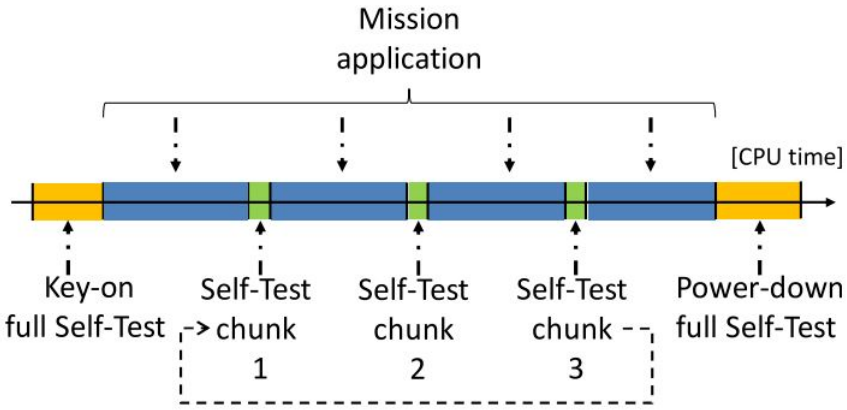

Figure 1.19: Online Test operation example

since the test programs are executed during the online context of the module. The test procedure is organized in several test programs where each of them aims to exercise a specific internal module as ALU, multiplier and so on. Each program is composed of an algorithm that awakes all the internal circuitry doing specific operations using different operands. To find the test algorithms may adopt several strategies:

- ATPG Based
- Manually or Deterministic Approach

The **ATPG Based** strategy permits to leverage the automatic pattern generation feature to find the stimuli for the fault detection. To achieve this goal, it is required an accurate constraints selection in order to generate only functional patterns that are feasible to reconstruct through a software. An alternative strategy is the **Manually Approach**, it aims to identify a dedicated algorithm to exercise the module analyzing the RTL description.

Each test program must be compliant with all the software environment supporting the *Embedded Application Binary Interface* (EABI) used to perform correctly the context switching.
During the development of the Test Suite is a good strategy to target the modules following a specific order based on the architecture of the module. In this way, reach a good Fault Coverage is easier since many faults are detected by side-effect. A possible order is in **Figure 1.20**.

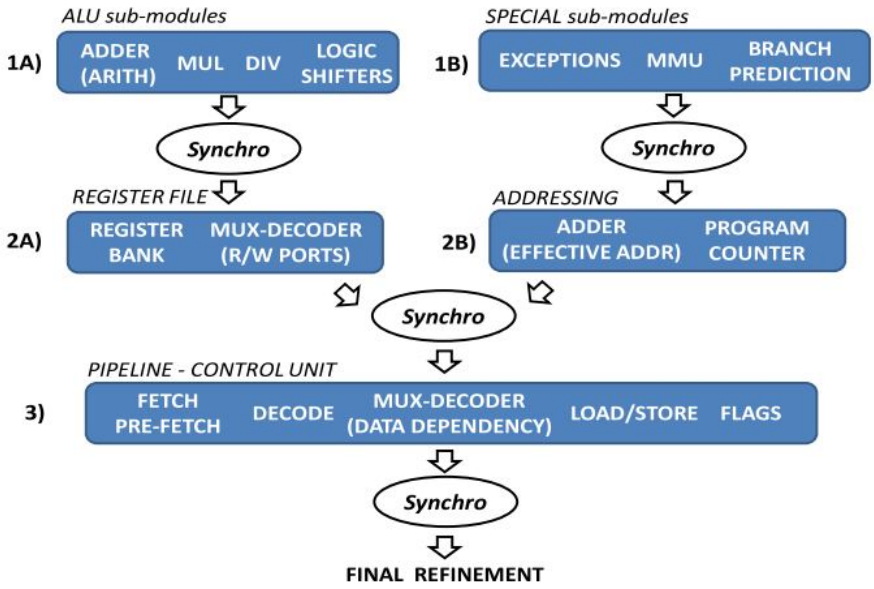

Figure 1.20: SBST Test Suite development order

Each Test program can be evaluated using a dedicated EDA Tool to perform the fault simulation. The effectiveness of a test program is given by looking at the POs involved to store the test result in memory. This is done because each test program will provide a signature mechanism to evaluate by software the presence of a fault inside the module. The idea is to accumulate the fault effect into a variable that will be compared with gold signature stored in memory. In this direction, it's crucial to ensure the signature stability in case of changes in the code size/location during the compilation process. After the development of all the test programs

it's mandatory to develop software to handle the test suite and initialize the test environment. By the way, a classical initialization regards the interrupt routines for the illegal instruction detection and a watchdog timer to measure the length of the test covering the possibility to have a defect in the decoding stage.

The SBST technique is a very effective technique providing fault coverages, for a single core, higher than 80% for Stuck-AT fault model. Even without hardware modifications to the design, the development of the SBST could be expensive in some contexts by the need of large computational efforts to perform the fault simulation of big sequential blocks, requiring the purchase of many licences of the testing tool to run parallel simulations.[4]

Chapter 2

Neural Networks and Multi-Computer architectures

2.1 Multi-Computer Systems

In this section we will introduce the most common multi-computer system architectures useful to allow an efficient implementation of neural networks.

2.1.1 Taxonomy of multi-computers

It is possible to classify a computer architecture according to Flynn's taxonomy. It distinguishes four different types of systems:

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data streams (SIMD)
- Multiple instruction streams, single data stream (MISD)
- Multiple instruction streams, multiple data streams (MIMD)

Starting from the **SISD**, it consists of a single control unit that fetches a single instruction at a time, addressing the processing unit to a single data stream. A common example of this architecture are uniprocessors systems.
The **SIMD** architecture operates a single instruction stream on a multiple data stream. Computer clusters and graphics processors (GPUs), widely used to manipulate neural networks, belong to this type.
The **MISD** architecture uses separated streams of instructions operating on a

single data stream. This type is very particular and is often used for fault tolerance reasons where it is necessary to verify that distinct systems and, therefore, distinct instructions always provide the same result.
The last architecture is **MIMD** and operates on different streams of instructions and data. Superscalar computers fall into this category. The latter can, in turn, be distinguished on the basis of the organization of the memory that can be shared or distributed. [5] All the different architectures are summarized in **figure 2.1**

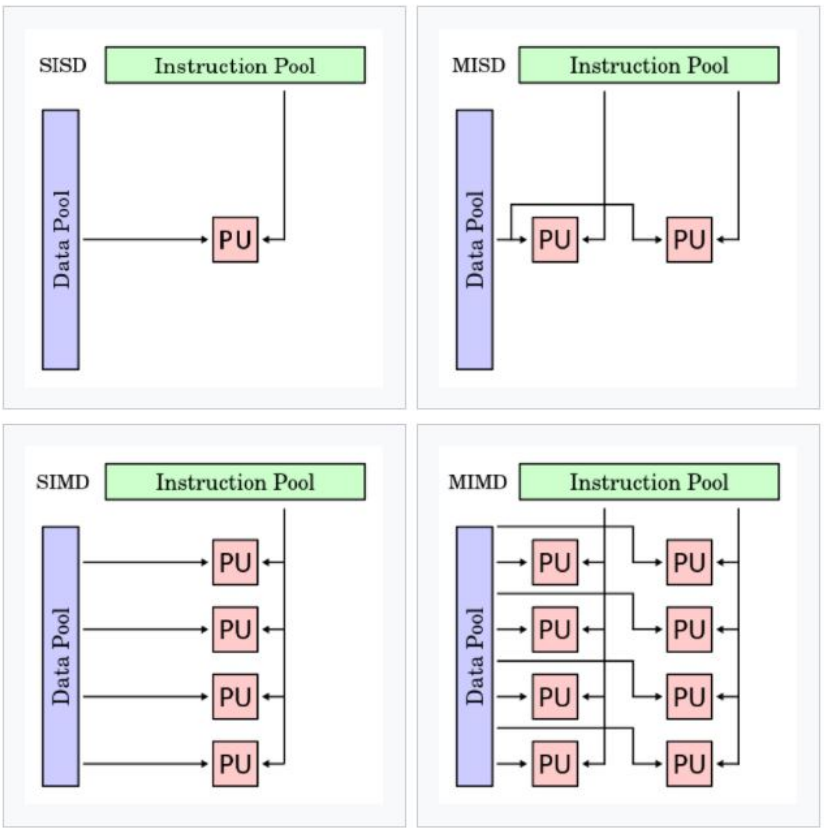

Figure 2.1: Architecture taxonomy

2.1.2 Multi-Core processor

With the reduction of the technological process, the physical limitations of semiconductor based microelectronic have become one of the most problematic aspects of design. These limitations lead to the creation of heat hotspots and data synchronization problems. New methods have been adopted to increase CPU performance as the **Instruction Level Parallelism** (ILP) and **Thread-level Parallelism** (TLP).

Instruction Level Parallelism

The ILP is a measurement, performed by CPUs to quantify the amount of code that can be executed in parallel without generating hazards. In modern CPUs, due to the presence of multiple computing units and the pipeline structure, the identification and exploitation of the instructions executable in parallel allows to significantly increase the performances. An example:

```
1) h = a + b
2) f = c + d
3) g = h * f
```

Instruction 1) and instruction 2) can be executed in parallel since they require data (a, b, c, d) independent of other instructions and therefore are free. Instead, instruction 3, which requires the data h and f, must wait for the first two to be completed to be executed. Assuming there exists, at least two independent ALUs, it is possible to execute instructions 1) and 2) in parallel, while 3) must wait for the other two. At this point, if all the operations are carried out in one clock cycle the code can be executed in just two clock cycles instead of three obtaining an improvement of 33%. In summary, the effectiveness of the ILP is strongly influenced by the following factors.

- Number of functional units
- Data dependency
- Control dependency

Considering having a complex program, respecting and optimizing the code to maximize the benefits of the ILP is very complex and highly infeasible. [6]

Thread-level Parallelism

Given the difficulty in solving the limitations of ILP, it was decided to adopt the TLP strategy adopting multi independent CPUs. In a multiprocessor system, task parallelism is achieved when each processor executes a different process on the same or different data. Each thread can execute the same or different code. In the general case, different execution threads can communicate among them. The communication usually takes place by passing data from one thread to others. An example:

```
program:
...
```

```
if CPU = "a" then
    do task "A"
else if CPU="b" then
    do task "B"
end if
...
end program
```

Both Cores start execution from the same memory address (program:) but the presence of the if statement allows differentiating tasks between them by running distinct programs at the same time.
This approach has made it possible to significantly improve the performances by driving the research towards multi-core systems. [7]

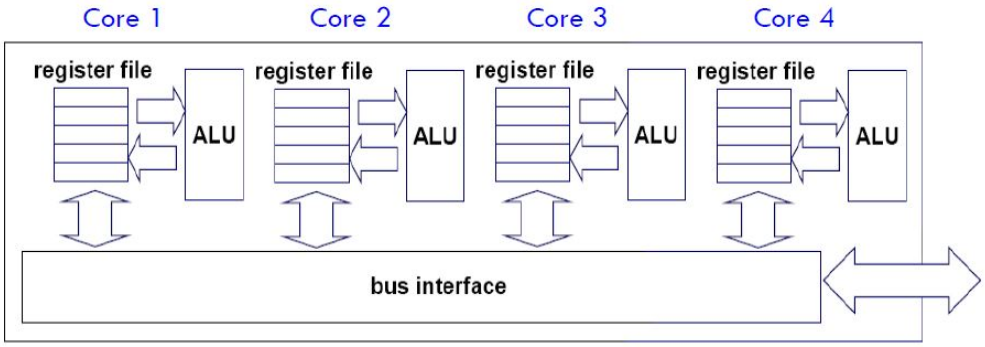

Figure 2.2: Multi-Core Architecture

The general structure of a multi-core is in **figure 2.2**. Each core is provided with all the essential hardware modules and, in general, it communicates with an external memory accessing a dedicated bus. The memory is shared among all the cores and it can be distributed or not. The data consistency is ensured by dedicated policies. [8]
The advantages of a multi-core architecture are not only determined by the performances but also for what concerns the power density. A better distribution of the power permits to dissipate easily the heat avoiding the probability of temperature hotspots all over the chip.

2.1.3 General Purpose Graphical Processor Unit

A General-Purpose Graphics Processing Unit (GPGPU) is a graphics processing unit (GPU) that is programmed for purposes beyond graphics processing, performing operations that are generally performed by normal CPUs. [9] Currently, all GPUs on the market are GPGPUs. The GPU is a system that uses multiple cores to exploit massive parallelism on a large scale. This type of system was mainly used

for visual rendering. The possibility of making these GPUs more general-purpose has made it possible to use these systems also for scientific purposes, as in the case of neural networks. The key advantage of this architecture is the capability to delegate portions of computationally high-cost code to the GPU, leaving the CPU free to perform other sequential operations. This highly parallel approach allows for significant performance improvements. In **figure 2.3** is shown the differences between a CPU and GPU in terms of architecture. [10] In 2007, Nvidia released

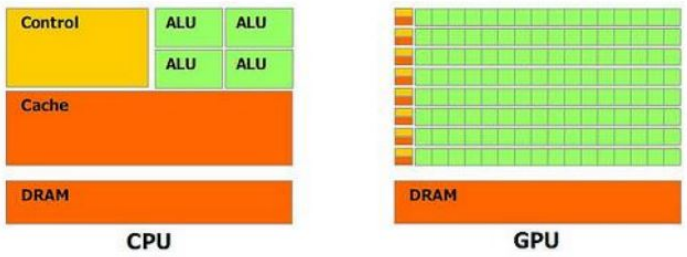

Figure 2.3: CPU and GPU comparison

CUDA, a platform completely dedicated to parallel computing compatible with all subsequent models of GPUs developed by the company. In coupled, dedicated APIs were also made available, compatible with the C, C ++, and Fortran languages, that are currently used in the development of neural networks.[11]

2.1.4 Computer Cluster

The Cluster computer is a series of computers connected to work together one powerful machine. Each connected computer is called a node and, unlike grid computers, each of them performs the same operations. Generally, the connection of the various nodes is carried out through high-speed local area networks. Each node can be identified as a separate computer with a dedicated operating system. From a functional point of view, cluster computers allow for excellent performances, larger storage capacity, better data integrity, greater reliability and a wider availability of resources. Compared to mainframes, cluster computers allow for a more distributed structure and better performance at the same cost. **figure 2.4** shows the structure of the computer cluster. As it is possible to observe, the system includes a central server that performs the scheduling functions by orchestrating all the nodes. [12]

2.2 What is a Neural Networks

In recent years the continuous research on artificial intelligence has made many steps forward trying to replicate and predict the way of thinking of the human

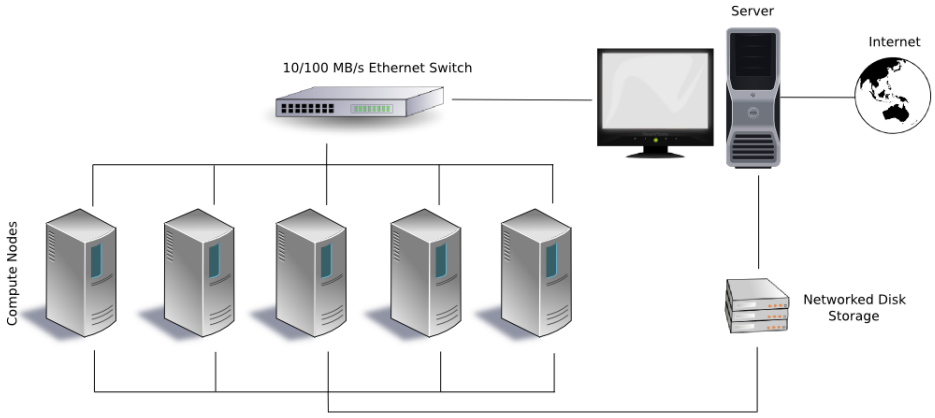

Figure 2.4: Scheme of Cluster Computer

being. This development inspired the search for a mathematical model that draws inspiration from the very structure of the human brain. The main idea is to replicate the neuron-synapse structure through the use of mathematical functions. In a completely analogous way to the human brain, the constituent element is defined as neuron connected with other ones through edges, making a network, as shown in **figure 2.5**. [13] Each artificial neuron receives a signal that is processed

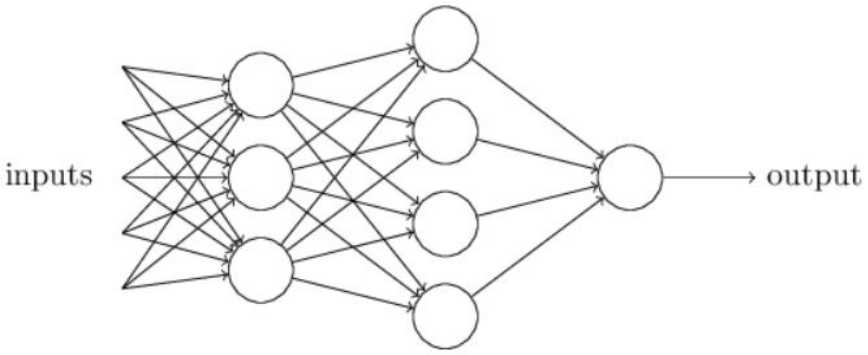

Figure 2.5: Example of neural network

through a non-linear function providing a result which, in turn, will be addressed as input to the following nodes. Each signal is a real number whose value, as we will see later, is determined by some parameters defined according to the operating specifications of the neural network. The purpose of a neural network is to create a computer structure capable of learning and self-regulating to perform prediction functions or, in the most common use, digital recognition. The main concepts of the neural networks will be discussed in the following paragraphs.

2.2.1 Neurons and activation functions

As previously anticipated, a neuron is a structure capable of receiving input parameters and delivering an output value. In **figure 2.6**, the node receives three inputs x_1, x_2, x_3 where, each of them is associated with a weight.

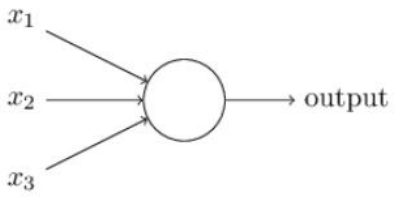

Figure 2.6: Example of node with three inputs

The output is determined by the weighted sum of the inputs compared with a threshold value. Mathematically everything is described by the following equation.

$$output = \begin{cases} 0 & \sum_{i=1}^{n} w_i x_i - b \leq 0 \\ 1 & \sum_{i=1}^{n} w_i x_i - b > 1 \end{cases} \tag{2.1}$$

Where:

- w is the weight
- x is the input
- n is the number of inputs
- b is the threshold named as bias

In short, each input represents an influencing factor on the node's decision while the weight determines the importance of each of them. For example, assuming you have to make the choice to go to a place or not, the factors could be the weather,

the place and the time. These three factors could have a different effect on the final choice, thus associating a weight.
At this point, the goal is to "train" the network applying learning algorithms capable of making slight variations to the weights and biases of each neuron in order to instruct the neural network to perform the desired function. Due to the strong linearity of relation (2.1) a small variation in the bias and weights does not lead to a variation on the output. To solve this problem, the concept of sigmoind neuron is introduced. The peculiarity of this model is given by the following sigmoid function 2.2

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

and substituting z with $\sum_{i=1}^{n} w_i x_i - b$

$$\sigma(z) = \frac{1}{1 + \exp(-\sum_{i=1}^{n} w_i x_i - b)} \tag{2.3}$$

Formula 2.3 allows to obtain inputs and outputs in the range of 0 and 1 and it is defined as **activation function**.

2.2.2 Architecture of the neural network

Structurally, the neural network is organized on different layers. The first layer is defined as the input layer while the output layer is defined as the output layer. In the middle are the hidden layers. The organization is shown in **figure 2.7**. In

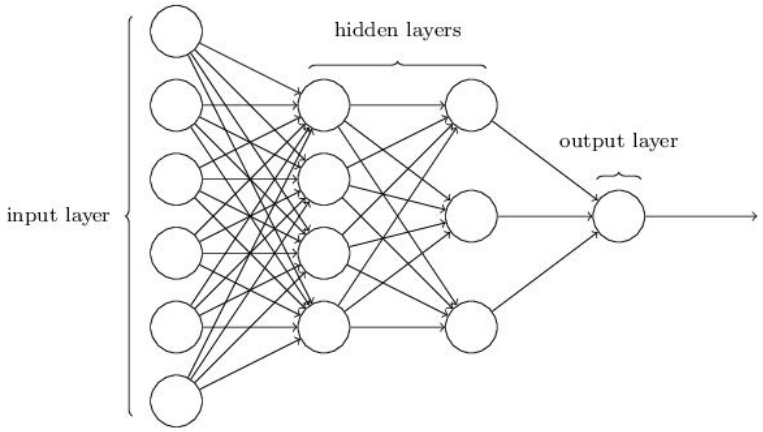

Figure 2.7: Neural network architecture

general, the design of a neural network is often intuitive. For example, supposing

you have an image of 64 by 64 gray scale you can imagine 4096 = 64x64 input neurons with intensity between 0 and 1. Inside this image there is a handwritten number to identify. Therefore, it is possible to have 10 output neurons representing the numbers between 0 and 9. Unlike the hidden layers don't have a thumbs rule and their design is still a field of research. In this direction, some heuristic methods are available to identify the correct design of the internal layers using the time required to train the neurons as constraints. By analyzing the layer outputs it is possible to identify two different types of architectures. It is defined feedforward when the direction of the output is always directed to the following layers. On the contrary, an architecture can contain feedbacks and in this case the neural network is defined as recurrent neural networks.

2.2.3 Learning process of a neural network

The learning process of a neural network is the phase in which the weights and biases of neurons are defined. This process is carried out through numerous iterations based on the idea of "going and return". [14] The **"going"** is a *forwardpropagation* of the result and the **"return"** is a *backpropagation* of the latter. During the forward propagation phase, the system is exposed to specific training data that cross all the layers up to the output nodes. Each neuron applies its own transformations with the starting parameters and sends them to the next layer. Once the output layer is reached, it provides a result that will be analyzed by specific loss functions. At this point, the loss function will provide a result of the comparison between the output obtained and the correct one providing a measure of the error obtained. This difference, in turn, will allow to adjust the parameters of each neuron in order to reduce the error on the next iterate.
The backpropagation phase will provide information on the loss starting from the layers closest to the outputs and then proceeding to adjust the parameters. The innermost layers will then undergo lighter variations based on their contribution to the output layers. Everything is carried out until the desired result is achieved by progressing more and more internally on the layers. The process is summarized in the **figure 2.8** The adjustment of the parameters takes place through the gradient descent technique. The approach is to calculate the derivative of the loss function to evaluate the "descent" of the error towards the global minimum. Obviously, it is necessary to provide a valid dataset in order to make the prediction error convergent at each iteration. Currently there are some specially designed datasets such as the *Modified National Institute of Standards and Technology* database (MINST) useful for recognizing handwritten numbers. An **epoch** is when entire dataset is passed forward and backward through the neural network only once. In case the dataset, constituting an epoch, is too big it is divided in subsets and the training process is performed in parallel as will be explained in the next paragraph.

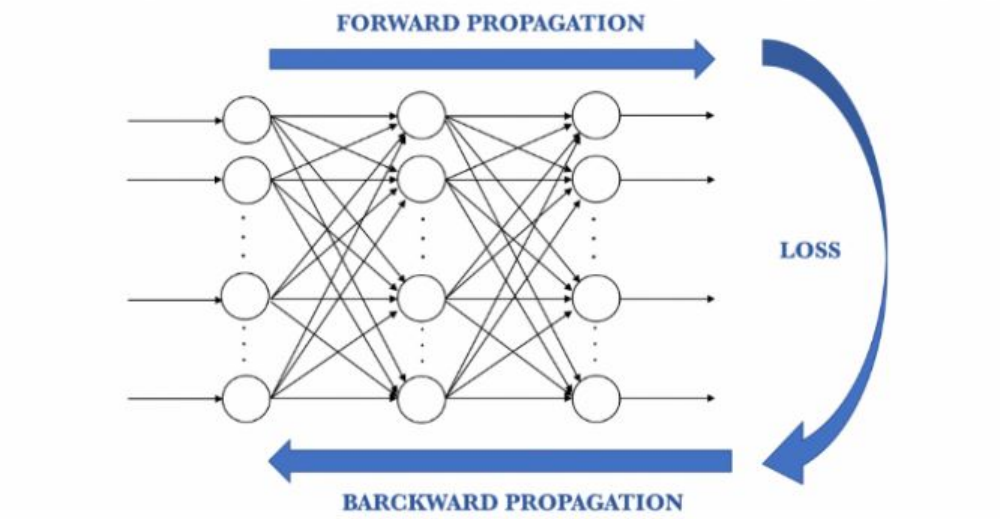

Figure 2.8: Neural network architecture

In summary, the training phase is summarized in **figure 2.9** in the following steps:

1. Begin with values (usually random) for the network weight and bias parameters
2. Pass the dataset to the network and predict the result
3. The prediction are compared with the correct ones and the loss is calculated
4. Through the backpropagation the loss is spread to each node that make up the model of the neural network.
5. Adjust the parameters using the gradient descent approach to rework the precision of the network
6. Continue iteratively until the loss reach the minimum value acceptable

2.2.4 Matrix Representation

From a mathematical point of view, managing neural networks through activation functions is clunky and complicated. For this purpose, a matrix representation is more functional guaranteeing, further, an easier software implementation. Starting again from the weights and the biases represented by 2.4 and the activation function 2.5.

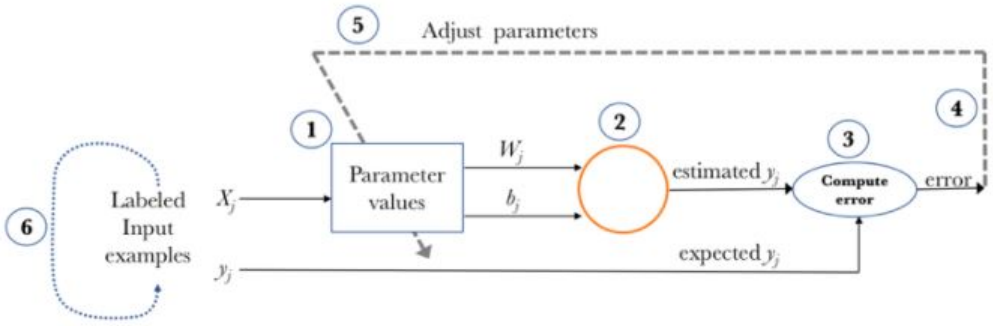

Figure 2.9: Learning algorithm flow

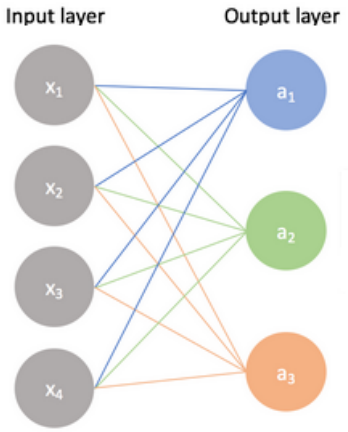

Figure 2.10: Example

$$z(x) = \sum_{i=1}^{n} w_i x_i - b \tag{2.4}$$

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2.5}$$

Considering the network in **figure 2.10**, The matrix representation is 2.6.

$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix} = \begin{bmatrix} w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \end{bmatrix} \xrightarrow{g()} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{2.6}$$

The next step is to write the weight as vector w_i where, i is the output neuron, and similarly for the biases.

$$w_1 = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} \; w_2 = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} \; w_3 = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} \tag{2.7}$$

$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \end{bmatrix} \textit{becomes} \begin{bmatrix} \leftarrow w_1^T \rightarrow \\ \leftarrow w_2^T \rightarrow \\ \leftarrow w_3^T \rightarrow \end{bmatrix} = W \tag{2.8}$$

The benefits of this notation occur when the goal is to calculate the values of the activation functions in presence of a network composed by hidden layers, as in **figure 2.11**. For example, supposing that it is required to compute the activation

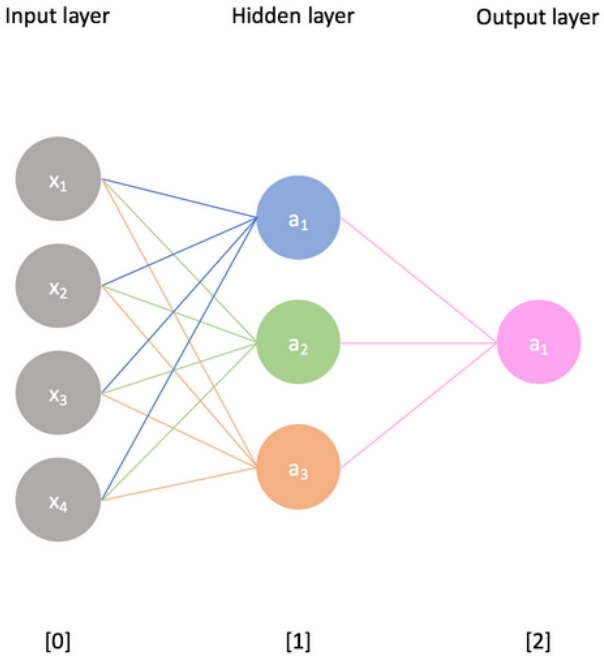

Figure 2.11: Example

function for the second neuron in the first hidden layer. It is given by 2.9 and 2.10.

$$z_2^{[1]} = w_2^{[1]T} a^{[0]} + b_2 \tag{2.9}$$

$$a_2^{[2]} = g(z_2^{[2]}) \tag{2.10}$$

In general, using as subscript $[l]$ that represents the l^th hidden layer and i for the i^th neuron.

$$z_i^{[l]} = w_i + 1^{[l]T} a^{[l-1]} + b_i \tag{2.11}$$

$$a_i^{[l]} = g(z_i^{[l]}) \tag{2.12}$$

Through the weights and biases matrix, it's possible to obtains all the activation functions.

$$Z^{[l]} = W + 1^{[l]T} A^{[l-1]} + b_i \tag{2.13}$$

$$A^{[l]} = g(Z^{[l]}) \tag{2.14}$$

This last equation demonstrates that the neural networks outputs can be evaluated through an efficient matrix representation.

2.2.5 Parallel Computing

The process of training and evaluating each node in the neural network is a procedure that is commonly referred to as embarrassingly parallel. This type of task is strongly aimed at dividing the problem into a number of parallel processes which are all highly independent and without the need to share the results among them. Observing the structure of the neural networks, it is possible to understand how each step of the training phase carries out independent computations in parallel.The need to execute concurrent operations implies the use of multi-computer systems in order to speed up their execution.

There are, therefore, some common strategies specifically designed to parallelize the neural network efficiently.

- Training Session Parallelism
- Exemplar Parallelism
- Node Parallelism

The **Training Session Parallelism** splits the training dataset into a subset and each node of the cluster computer is trained in parallel and, independently, seeking for the best result.
The **Exemplar Parallelism** used a divided subset combining the errors and updating all the weights for each epoch in the cluster. The main difference with the previous parallelism is that the nodes are able to communicate during the process by the combination of their results. The **Node Parallelism** consists to assign each neuron of the network to each node of the cluster and compute their respective activation functions. This approach is not feasible and impractical, for

computer cluster, since the number of neurons is not comparable with the lower number of nodes. [15]

2.2.6 Impact on Instruction Set Architecture

The strong need for data parallelism and the practical matrix representation of neural networks required the addition of new instructions to the classic ISA in order to improve the performance. By the way, multiply and accumulation instructions have been added in almost all the ISA. Furthermore, to speed up the new extensions many CPU architectures have been slightly modified to speed up the data location update related with weight and biases. An example of this will be discussed in chapter 4. [16]

Chapter 3

Risc-V

In this section we will describe the RISC-V Instruction Set, which was used throughout the entire project, showing the design and its principles in terms of ISA and expandability.

3.1 Instruction Set Architecture ISA

The RISC-V is an open-source hardware Instruction Set Architecture based on the principles of reduced instruction set computer (RISC). The project developed at Berkeley by the University of California for research and education purposes currently aims to establish itself as the standard for industrial implementations. One of the advantages of Risc-V is the high degree of extensibility of the standard ISA which in turn implies an increase in the complexity of the hardware architecture. [17] It is therefore possible to implement a CPU that best suits the needs. A standard base integer ISA is defined, on 32 bit and 64 bit marked as "I" extensions. From this starting point, it is possible to extend ISA by adding instructions dedicated to operations between vectors, floating points, multiplications and divisions, and so on. All extensions are compatible with the GCC (GNU C Compiler), furthermore, the RISC-V Linux Kernel is fully supported by the Linux Foundation thus guaranteeing the ability to run Linux increasing the industry interest.

3.1.1 Register File

The Risc-v includes a register file made up of 32 integer registers to which can be added the possibility of being used as floating point register in the presence of the "F" and "Zfinx" extensions. Given the "Load and Store" nature, except for read/write memory instructions, the instructions can only be executed only through the registers or by immediate values. In **table 3.1** there is the complete

list of the registers.

Register name	Symbolic name	Description
32 integer registers		
x0	Zero	Always Zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary
x6-7	t1-t2	Temporary
x8	s0/fp	Saved register Frame pointer
x9	s1	Saved register
x10-11	a0-1	Function argument/ return value
x12-17	a2-7	Function argument
x18-27	s2-11	Saved register
x28-31	t3-6	Temporary
32 floating-point extension registers		
f0-7	ft0-7	Floating-point temporaries
f8-9	fs0-1	Floating-point saved registers
f10-11	fa0-1	Floating-point arguments/ return values
f12-17	fa2-7	Floating-point arguments
f18-27	fs2-11	Floating-point saved registers
f28-31	ft8-11	Floating point temporaries

Table 3.1: Register sets

3.2 Instruction Format

The integer standard ISA is a simple set which comprises less than fifty instructions, capable of satisfying all the requirements required by modern applications. The instructions are structured in four different formats as sown in figure 3.1.

- R-Type
- I-Type
- S-Type

- U-Type

All these formats share the same structure keeping the source registers *RS1,RS2* and the destination register *RD* in the same position. This is done to reduce the complex of the decoding hardware. Similarly, the immediate operands *Imm* start always on the leftmost significant bit and the sign position is always placed in the 31 bit.

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper Immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

Figure 3.1: RISC-V Instruction Format

3.2.1 Common RISC-V Extensions

As previously mentioned, one of the goals of the RISC-V design is to keep ISA as simple as possible. Some extensions have been standardized and will not change in the future.

- **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
- **C** : Standard Extension for Compressed Instructions

3.3 PULP RI5CY

The RI5CY is a 4-stage in-order 32-bit RISC-V processor core, that implements the RV32-IMC extended with additional instructions as hardware loops, post-increment load and store instructions and additional ALU instructions. Optionally, the RI5CY can include support for single-floating-point operations and SIMD instructions. The general architecture is in **figure 3.2**.

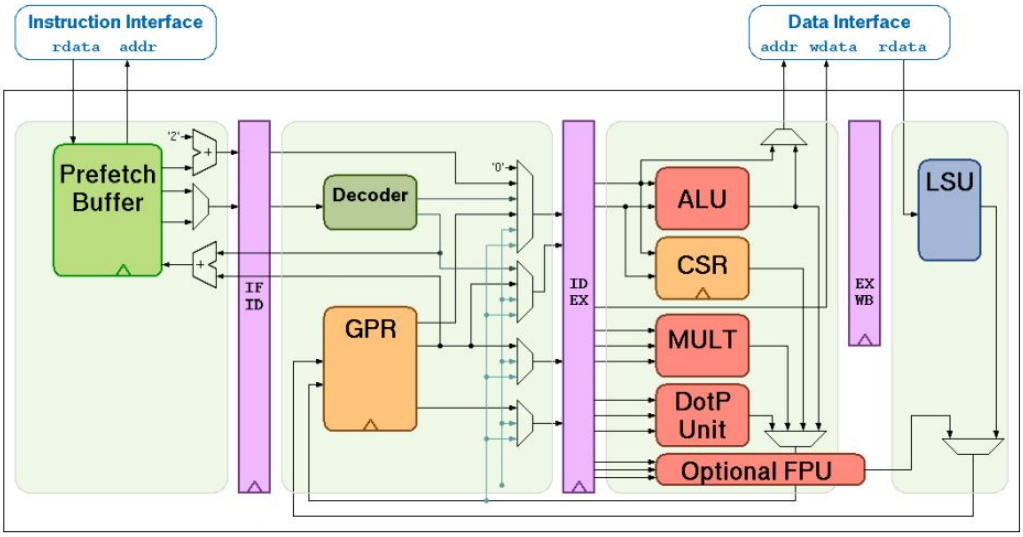

Figure 3.2: RI5CY Architecture

3.3.1 Instruction Fetch Stage

The Instruction Fetcher is capable of delivering one instruction for each clock cycle. The instructions are half word aligned as compressed instructions are also supported thanks to the "C" extension. Due to performance and timing reasons the IF stage is composed by a prefetcher that is available in two distinct implementations.

- **32 Bit Word Prefetcher** - It stores three instructions in a FIFO memory
- **128 Bit Cache line Prefetcher** - It stores the entire cache line.

3.3.2 Decode Stage

The Decode stage is composed by the Register File and several controllers in charge to decode the instruction. This stage is able to handle jumps and exceptions. The register file is fully compliant with the RISC-V standards having 32 registers and it is available in two different flavours, Flip-Flops and Latches based. The first is recommended for FPGA synthesis while the second one for ASIC.

3.3.3 Execute Stage

The Execute Stage is composed by many elements dedicated to compute the operation. The common elements are:

- Algebraic Logic Unit ALU

- Multiply and Accumulate Unit MULT
- Control and Status Registers CSR
- Optional Floating Point Unit FPU
- Hardware Loop Controller HWLOOP

Algebraic Logic Unit ALU

The ALU is one of the main building blocks of the Core. The unit covers a wide range of operations not only used for pure mathematical calculations. It is involved in:

- General ALU operations
- Bit Manipulation
- Jump Addresses computations
- Branching Decisions

This module is compatible with the full standard "I" extension.

Multiply and Accumulate Unit MULT

The MULT unit is a 32bit unit dedicated to perform multiplication, division and accumulation operations. This module is highly involved during the training process of the neural networks. While the multiplication is performed in single clock cycle, the division can take between 2 to 32 cycles. The unit supports all the instructions of the "M" extension.

Control and Status Registers CSR

The RI5CY implementation does not include all the registers specified in the RISC-V privileged manual but only the ones required to properly configure the system. [18] This is done to reduce as many as possible the footprint of the Core once synthetized. The list of the CSR is in **figure 3.3**.

Floating Point Unit FPU

The FPU is capable of performing all the instruction included inside the "F" extension. It is implementation is completely optional. Obviously, the latency of the instructions are different according to the complexity. The FPU is organized in three parts:

CSR Address				Hex	Name	Acc.	Description
11:10	9:8	7:6	5:0				
00	11	00	000000	0x300	MSTATUS	R/W	Machine Status
00	11	00	000101	0x305	MTVEC	R	Machine Trap-Vector Base Address
00	11	01	000001	0x341	MEPC	R/W	Machine Exception Program Counter
00	11	01	000010	0x342	MCAUSE	R/W	Machine Trap Cause
01	11	00	0xxxxx	0x780-0x79F	PCCRs	R/W	Performance Counter Counter Registers
01	11	10	100000	0x7A0	PCER	R/W	Performance Counter Enable
01	11	10	100001	0x7A1	PCMR	R/W	Performance Counter Mode
01	11	10	110xxx	0x7B0-0x7B7	HWLP	R/W	Hardware Loop Registers
11	00	00	010000	0xC10	PRIVLV	R	Privilege Level
00	00	00	010100	0x014	UHARTID	R	Hardware Thread ID
11	11	00	010100	0xF14	MHARTID	R	Hardware Thread ID
01	11	10	110000	0x7B0	DCSR	R/W	Debug Control and Status
01	11	10	110001	0x7B1	DPC	R/W	Debug PC
01	11	10	110010	0x7B2	DSCRATCH0	R/W	Debug Scratch Register 0
01	11	10	110011	0x7B3	DSCRATCH1	R/W	Debug Scratch Register 1

Figure 3.3: Control and Status Registers List

- A simple FPU which computes FP-ADD, FP-SUB and FP-casts
- An iterative FP-DIV/SQRT unit which in charge to perform FP-DIV/SQRT operations
- An FP-FMA unit which takes care of all combined operations

Adding the FPU, there are accessible also some dedicated CSR used to handle the rounding mode, precision, exception and to check the status of the unit. As said before, the register file can be implemented in a separate manner or shared with the integer one.

Hardware Loop Controller HWLOOP

The HWLOOP controller is a dedicated module in charge to handle the loops guaranteeing a lower code density. This module permits to repeat a section of code without updating counter and using branch instructions. It involves zero stall cycles for perform the jump. The controller is configurated through some dedicated CSRs that define the start and end addresses plus the number of loop repetitions. Using two configuration levels there is the possibility to configure nested loops. [18]

3.3.4 Load and Store Unit

The LSU allows to communicate with the memory through dedicated load and store operations. Accesses in memory can also be performed in a misaligned way through specific instructions available in the ISA. In this stage there is a Memory Protection Unit in charge of blocking some accesses in memory according to the Privileged Modes. The last important aspect of this module is the capability to Post-Increment the Memory Pointer reducing the length of code required for this operations. [18]

Chapter 4

Case of Study - VEP Design

In the next chapter will be introduced the hardware accelerator on which the whole experience has taken place describing, in part, its characteristics.

4.1 About Dolphin Design

Dolphin Design is a company headquartered at Meylan near Grenoble, France, since 1985.

They participate in the design industry for microelectronics, leveraging the dynamics of the semiconductor industry. Focusing on CMOS Virtual Components of Silicon IP, the company aims at enabling low-consumption System-on-Chip (SoC) based on digital libraries of standard cells and memories. Another research field is focused on the development of digital-to-analog converters for audio applications with their power regulators. [19]

4.2 Overview

The VEP is designed to demonstrate the effectiveness of the entire Dolphin GF22FDX Platform in a real application. The GF22FDX Platform is a set of pre-configured and validated custom IPs aiming to provide smart solutions to reduce the energy efficiency guaranteeing an improved design time cycle and implementation. [19] The targets of the design cover both application and power consumption aspects. The main applications are tasks executable through the implementation of Neural Networks as:

- Speech recognition
- Natural Language Processing

- Object detection
- Image classification

From the point of view of the Power consumption the design ensures the following specifications:

- Ultra-Low power Deep Sleep Power < 20uW
- Best Class Energy Efficiency with 0.5V operating Mode (>15 TOPs/W)

[20]

The SoC is composed of several Clusters connected to each other through a low latency interconnection and orchestrated by the main core named **Fabric Controller (FC-CORE)**. Each Cluster is composed of several dedicated computing units in order to parallelize as much as possible the operations. The power side is managed through Always On Safe-Domain containing several IPs developed by Dolphin Design. The scheme is shown in **figure 4.1**. The SoC is organized in

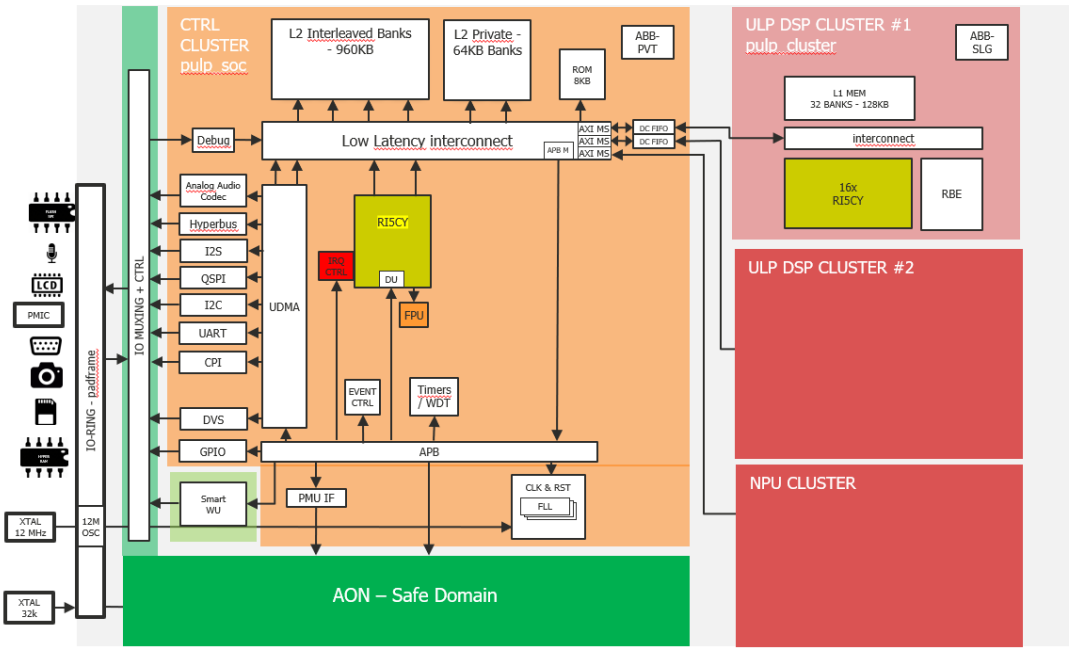

Figure 4.1: VEP Design

distinct operational regions:

- **Control Cluster**
- **DSP Cluster**

- **NPU Cluster**

With reference to chapter two, the structure of this SoC can be classified as a multicore system classified as SIMD. From the programming point of view, the code executed by the different cores of each task is executed and orchestrated through the TLP model explained in section 2.1.2. *In this experience, the Control Cluster and only one of the DSP-Clusters have been considered.*

4.3 Control Cluster

The Control Cluster allows to manage all the interfaces of the chip. Therefore, it allows to orchestrate all clusters and configure the power domain through the PMU. The main features are the following:

- Fabric Controller
- System Interconnect
- L2 Memory
- uDMA subsystem with interface peripherals
- Clock Management
- Event Unit

4.3.1 Fabric Controller

The Fabric Controller implements a PULP RI5CY core optimized for DSP pre-processing. It supports the RV32IMFC ISA with additional extensions to improve digital signal processing performances. The implementation include optimized settings for:

- Vectorial Instructions
- Fixed Point Instructions
- Complex Number Operations

In addition, it's present an LP Timer configurable to generate interrupts or work as System Clock.

4.3.2 System Interconnect

The interconnection is organized through three different types of BUS. The APB is used to communicate with the uDMA and the AON Domain. The AXI bus with a data size equal to 64bit communicates with the different Clusters presents inside the SoC. Successively, there is a multi master/slave low-latency crossbar (XBAR) which communicates with the internal memory L2, the FC-CORE and the remaining internal modules such as HWPE, JTAG Debug unit and the TX and RX ports of uDMA. Inside the XBAR, it's present a dedicated Arbitration Tree to handle the policy adopted for the multi master/slave communications.

4.3.3 L2 Memory

Memory is structured according to two different types: shared and private. The larger shared memory is divided into four different blocks all of them accessible by the cluster regions albeit, with a lower priority with respect to the control cluster . To ensure the simultaneous access each block has dedicated ports communicating with the XBAR. The shared memory is implemented using SRAM. The private memory is composed by two blocks dedicated for Control Cluster and implemented with SRAM and SCM. Standard cell memories (SCMs) are becoming a common alternative to SRAM IPs due to their design flexibility, ease of implementation, and robust operation at low supply voltages. Exclusively composed of standard cells, these memory arrays are implemented as part of the standard digital design flow.From the functional point of view the L2 memory contains the code of the program and it is fetched by the DSP Cluster Instruction Cache.

4.3.4 uDMA with Interface Peripherals

The uDMA subsystem is dedicated to move the data from the L2 memory toward the external peripheral modules and viceversa through the XBAR. It ensure an high-bandwidth of data with a low-power consumption. The data transfer is triggered by the event unit. The external interfaces are:

- Serial interfaces (UART, I2C, SPI)
- Memory interfaces (QSPI/Hyperbus)
- Camera interfaces (CPI/DVS)
- Audio interfaces (ADC, I2S)

4.3.5 Clock Management

The clock Management is handle by four FLL fed by two oscillators operating at 32 KHz and 12 Mhz. The FLL modules generate the clock for all the regions of the SoC.

- Cluster FLL : generates the clock of the DSP cluster
- NPU FLL: generates the clock of the NPU cluster
- SOC FLL: generates the clock of the control cluster
- Peripheral FLL: generates the clock of interface peripherals

All the FLL can operate in two modes Normal and Standalone. In **normal mode** the frequency is determined by configuring two specific registers used to set the multiplication factor M and the division factor D_{out}. The output frequency is given by 4.1.

$$F_{out,normal} = F_{ref} \cdot \frac{M}{D_{out}} \tag{4.1}$$

In **Standalone mode** the control loop is unregulated and not operational. This permits to generate the frequency by the input word of the DCO without requiring any reference clock.

4.3.6 Event Management Unit

The Event Management Unit is in charge to collect all the event coming from the SoC and dispatch them towards one of the three destinations the FC-Core, uDMA and DSP Cluster. The latter is provided of a dedicated Event Unit directly connected with the main one. The events generated can be handled by a priority mechanism.

4.4 DSP Cluster

The DSP cluster has been designed to accelerate and parallelize operations as much as possible with the aim of facilitating the implementation of neural networks. The schematic is summarized in **figure 4.2**.

The cluster is composed by several blocks.

- 16 x PULP RI5CY with Neural network extension
- Dedicated L1 Data Memory
- Shared FPU

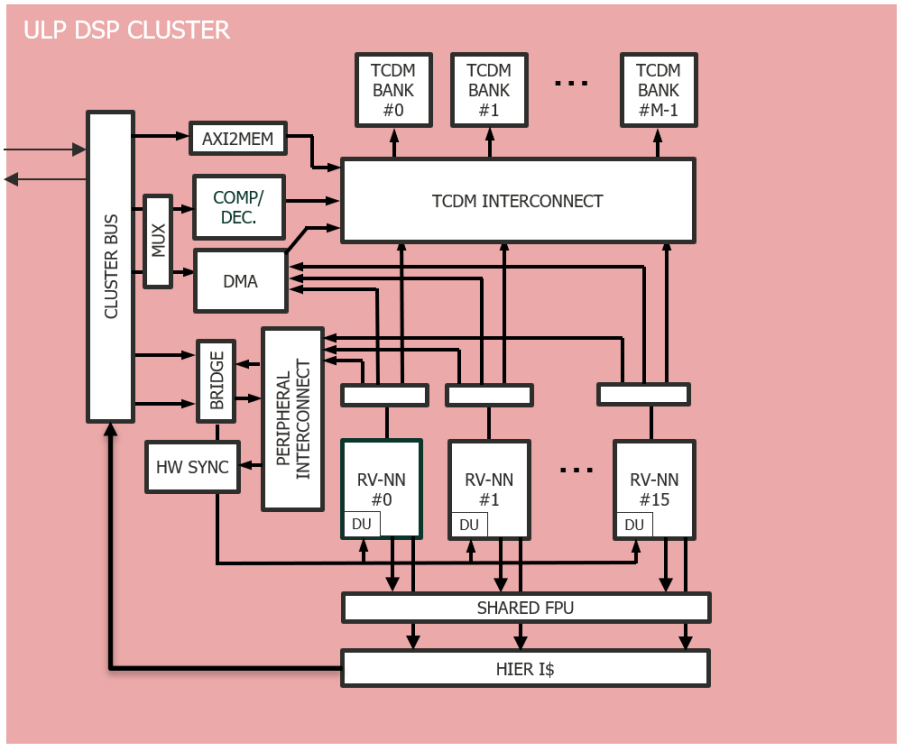

Figure 4.2: DSP Cluster Scheme

- Instruction Cache
- DMA
- Cluster Bus.

4.4.1 RISC-V NN ISA

The NN extension includes a new instructions library for parallel ultra-low-power tightly coupled cluster of RISC-V processors. The key innovation in PULP-NN is a set of dedicated operations for quantized neural network inference, targeting byte and sub-byte data types, tuned for the recent trend toward aggressive quantization in deep neural network inference. The proposed library leverages both the digital signal processing extensions available in the PULP RISC-V processors and the cluster's parallelism improving performance by up to 63× with respect to a sequential implementation on a single RISC-V core using the baseline RV32IMC ISA. [16] An example of new instruction is the pl.sdotsp.h.0 rd,rA,rB and pl.sdotsp.h.1rd,rA,rB . Each of these two VLIW instructions combine the post increment load word (lw!)

and the pl.sdotsp.h operation improving, further, the code density.

From the Hardware point of view, some Special Purpose Registers have been added in the decode stage in order to avoid pipeline stalls caused by the new VLIW instructions. The hardware modifications are shown in **figure 4.3**.

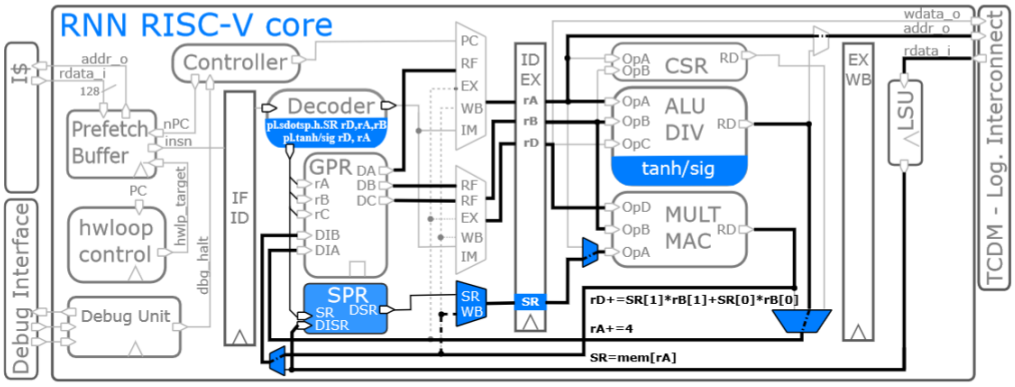

Figure 4.3: PULP RISC-NN Scheme

4.4.2 L1 Data Memory

The DSP Cluster includes 128KB of Data memory organized in 32 Banks. the whole memory is shared between all the cores accessible by the crossbar. Each bank has its interface dedicated with the crossbar, this is done to ensure concurrent accesses to the memory. The crossbar supports up to 32 slaves (all memory banks).

4.4.3 Shared FPU

Inside the DSP cluster the FPU is shared among all the cores. This approach permits to save area and costs since the FPU operations are less frequent with respect the integer ones. The shared FPU is made up by:

- Multiplexer logic to route the operands towards the targeted FPU unit
- 8x FPU unit dedicated for additions, multiplications, comparisons and conversions. Each unit is shared between two cores
- One FPU dedicated for divisions and root square operations

4.4.4 Instruction Cache

The solution adopted for the instruction cache exploits the benefits brought by SCMs by combining a two-level architecture capable of providing a larger "virtual" cache capability. The L1 level is private for each core, while the L 1.5 level includes a larger and shared low latency cache (one clock cycle). From a functional point of view, the assumption of a shared cache is especially advantageous for the execution of parallel code as it avoids replicating large portions of code in each private cache.[21] The general structure is shown in **Figure 4.4**. The two caches are

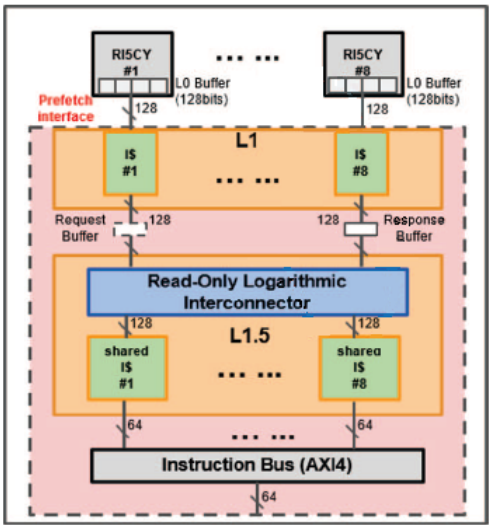

Figure 4.4: Instruction Cache Architecture

connected together with a low latency interconnect. The shared cache is organized in several blocks each of them is accessible by the core. The functioning of the cache is summarized in the flow chart in **figure 4.5**

A Round-Robin policy regulates the contention when two or more cores are accessing to the same shared cache block. Furthermore, the presence of a buffer between the L1 and L 1.5 ensures a shorter critical path improving the architecture scalability towards high-end clusters.

4.4.5 DMA and Bus

The DMA module is in charge to move the data from the L1 Memory towards the L2. The operation is performed through the dedicated cluster bus which leveraging an AXI interfaces. The Data Bus is 64-bit wide.

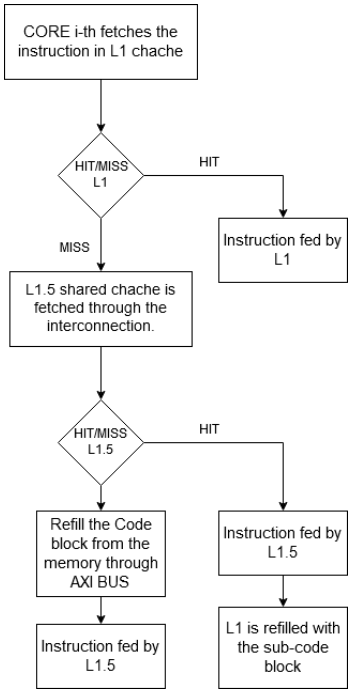

Figure 4.5: Instruction Cache Functioning

4.5 Testing Aspects

The main goal of the developed software is to detected as much as possible the faults related with the computational block of the DSP-Cluster providing in addition a safe environment to perform in field test. In this section, will be shown the hierarchy organization of the faults and which of them have been considered during the experience. The **table 4.1** shows the complete fault hierarchy list for the Stuck-At model.

The software that will be explained targets the SA faults inside all the core regions and the shared FPU. Other side-effect faults can be detected even without an ad-hoc test program.

11688826	**Top Module**
51040	/Cluster Bus
28836	/AXI to Memory Bus
13188	/AXI to Peripheral Bus
566	/Peripheral Demux
105172	/Peripheral to AXI Bus
1077114	/Cluster Interconnect Bus
477070	/DMA
506974	/Cluster Peripherals
226924	/Core 0 Region
226124	/Core 1 Region
226732	/Core 2 Region
226192	/Core 3 Region
225392	/Core 4 Region
225346	/Core 5 Region
225636	/Core 6 Region
226008	/Core 7 Region
225400	/Core 8 Region
225968	/Core 9 Region
226732	/Core 10 Region
225226	/Core 11 Region
225888	/Core 12 Region
226226	/Core 13 Region
225952	/Core 14 Region
226184	/Core 15 Region
1447620	/Shared FPU Cluster
2774690	/Hardware Peripheral Subsystem
1361710	/Instruction Cache
103684	/L1 Memory Banks
118890	/Other Modules

Table 4.1: Fault summary

Chapter 5

Testing Software

The following chapter will provide a detailed overview of the workflow used to develop the SBST, starting from the development of the several test programs up to the software interface for their execution. *This is an explanation about a possible solution and it is not definitive but, it tries to give a starting point for a more complex and efficient implementation.*
Another software designer could implement different flows, tools and perform others considerations. The software architecture is structured on three layers as shown in **figure 5.1**.

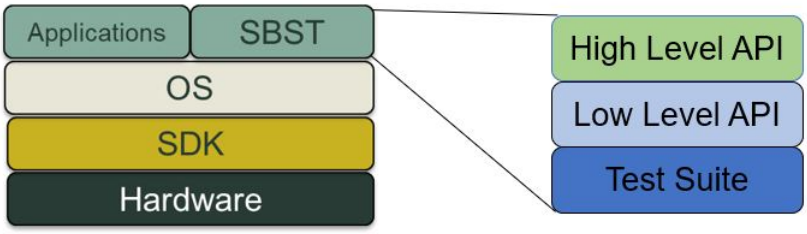

Figure 5.1: Framework

At the bottom, there are all the test programs ordered within a **Test Suite**. Each program allows exercising a specific functionality of the DSP Cluster. In the middle, there is a layer of functions (**Low Level API**) to initialize the system for testing. The higher level (**High Level API**) of functions allows organizing the test programs by test sets taking into account, further, some aspects concerning the multi-core architecture of the system.

The development starts after the completion of a stable and functioning simulation environment providing the possibility to write programs, in C or Assembler,

which can be executed on one or more cores. Once a program is written, it is possible to follow its flow through an RTL simulation of the circuit using QUESTA SIM. This feature, combined with the access of the complete System Verilog description, permits to understand the functionalities of the hardware involved in the operations.

5.1 Test Suite Development

Each test program is coded through a C function containing within it the code written in Assembler leveraging the possibilities provided by the ASM Extended library.

The development of each program follows a well-defined workflow summarized in **figure 5.2**.

The flow is composed by the following step list that will be explained in the next sections.

- Analysis and Simulation
- Pattern Generation
- Fault simulation without output Masks
- Fault list managing
- Signature implementation
- Context Saving
- Fault Simulation including output Masks

5.1.1 Analysis and Simulation

As previously mentioned, the first step is characterized by an in-depth study of the target module. This process requires careful reading of the RTL description and running some "hello world" programs to identify the behaviour of the different control signals. As instance, in the case of the multiplier, all the distinct assembler functions provided by the ISA are tested and analyzed, identifying all the signals commutations to distinguish the type of multiplications. This procedure allows to decide which is the best approach to develop the test program patterns.

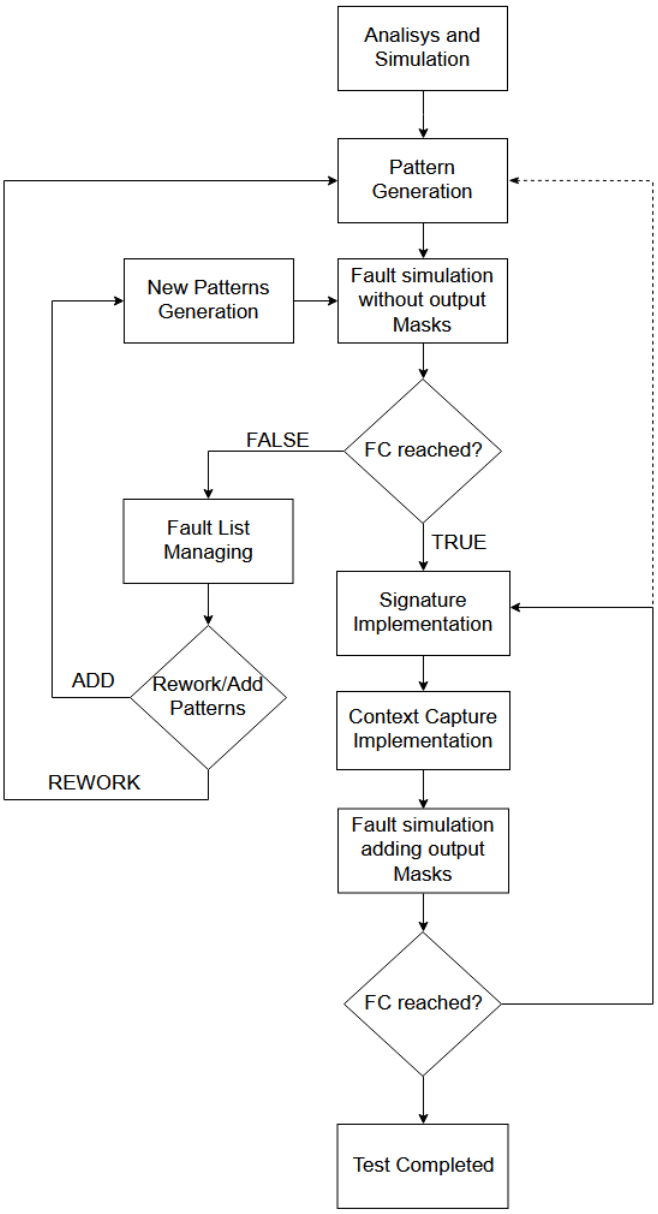

Figure 5.2: SBST Pattern Generation Flow

5.1.2 Pattern Implementation

The implementation phase of the patterns is the main part of the development allowing to reach the targeted fault coverage value. Once, understanding the functioning of the hardware in the first step, it is possible to start defining which strategy is more convenient to use. As anticipated in section 1.1.8, the designing of

the patterns can be performed through a Pseudo-Random approach or by exploiting the patterns created by the ATPG.

Pseudo-Random Approach

The pseudo deterministic approach is based on the remarkable effectiveness of "chessboard" patterns. This type of pattern is especially effective for the computational units of the core because the switching of the operand bits greatly stresses the internal circuitry. An example of a very effective and widely used set pattern in this experience is the (5.1).

$$set\ patterns = \{0x00000000, 0x11111111, 0x22222222, ..., 0xFFFFFFFF\} \tag{5.1}$$

By the way, an algorithm frequently used for the computational blocks is to switch the operands using all the possible combination of all the patterns above on all the available instructions leveraging loop mechanisms. Even though this approach is very effective, it has a considerable impact on the test duration length, since, the number of operation can be estimated by the (5.2) equation.

$$\#Operations = S_1 \cdot S_2 \cdot I \tag{5.2}$$

Where, S_1 and S_2 are the number of patterns of the set while I is the number of all possible operations for the target block.

A possible implementation of the basic algorithm is shown in the following pseudo-code:

```
stim_A[] ={0x00000000,0x11111111,0x22222222, ... , 0xFFFFFFFF }
stim_B[] ={0x00000000,0x11111111,0x22222222, ... , 0xFFFFFFFF }

for(i=0; i<size(stim_A); i++){
    for(k=0; k<size(stim_B); k++){
        result = stim_A[i] op1 stim_B[k];
        store(result);
        result = stim_A[i] op2 stim_B[k];
        store(result);
        ...
        so on
        ...
    }
}
```

Furthermore, the application of the test set in (5.2) can find important test applications on all the memory based modules. In general, for this type of circuit the application of a **Memory Test** algorithm is a good approach to achieve high

values in terms of fault coverage. The classical March algorithms exploit the potentialities given by this pattern set, since, it grants the detection of possible faults present inside the address logic of the memories as well as in the memory cells. [2] An example pseudo-code of a basic Memory Test algorithm that provide a good fault coverage for Stuck-AT is:

```
stim_A[] ={0x00000000,0x11111111,0x22222222, ... , 0xFFFFFFFF }

for(k=0; k<size(stim_A); k++){
    for(l; l<size(memory); l++){

        store(stim_A,address[l]);
        result = read(address[l]);
        if (result != stim_A)
            Error();

        c_result = complement(result)
        store(c_result);
        result= read(address[l]);
        if (result != c_stim_A)
            Error();
    }
    ...
    so on
    ...
}
```

To apply further stress to the module and finalize some test programs it's reasonable to use, besides, the instructions containing immediate fields. Flipping all the bits of those fields it's possible to exercise all the circuitry involved to decode and compute this type of operations. For the control blocks, the random approach is not always applicable since there are restrictions for the input patterns. In general, many control blocks are tested by side effects due to the applications of the other test programs as mentioned in section 1.18. Anyway, to improve the coverage of the control blocks it's a good strategy to apply all the possible control instructions available by the ISA for that block. Fortunately, the control blocks have a less quantity of faults concerning the computational blocks placing them in background.

ATPG Based Approach

The ATPG based approach needs a very deep knowledge regarding the functional aspects of the module taken into account. The idea is to constrain the ATPG to

create only functional patterns and reproducible via software. A very interesting aspect of the ATPG approach is the ability to generate sequential patterns with variable depth, allowing the fault identification through specific sequences of patterns. Once the test vectors have been generated, it will be necessary to "translate" them into assembler instructions through a text parsing script. The flow is shown in **figure 5.3**.

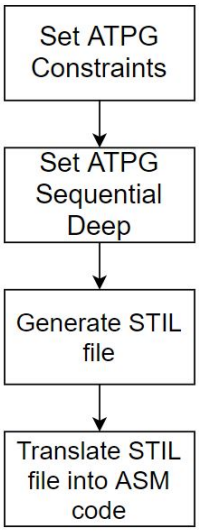

Figure 5.3: ATPG Pattern Generation Flow

The STIL format is an IEEE approved standard for the Test Vector format. This format mainly comprises four distinct sections:

- **Signals Block** - Defines all the pins of the module
- **Timing Block and Waveform Table** - Defines the waveform formatting and signal timings
- **DCLevels Block** - Defines the DC Signal Levels applied to and expected from the device under test
- **Pattern Block** - Defines the Test Vectors

[22]

The most complex part of the procedure is the translation of the different patterns into instructions, requiring complex parsing algorithms in situations of a large number of instructions and control signals,. Each pattern requires the use of multiple ASM instructions to be properly translated as shown below. This example could be the basic translation for a pattern generated for the ALU where, before to perform the operation, it is needed to load the two operands

```
/////  asm.s /////
li t0, (operandA);
li t1, (operandB);

add t2,t0,t1;  //operator defined by control signal
sw t2,4(sp);   //show the result on the output port

}
```

The ATPG approach is very effective but difficult to apply. It is convenient to use it in a complementary way to the pseudo-random by taking into account reduced fault lists and a considerable number of constraints to minimize the complexity of the parser algorithm. The main advantages of the technique concern the speed of program execution (w.r.t the loop structure example of pseudo-random) and the identification of hardly observable faults. If everything is done correctly, the coverage obtained by the test program is close to that obtained by the ATPG. In case of a high number of patterns, it is necessary to structure the program within loops by saving the patterns in dedicated pools, thus allowing to obtain a reduced code length and so, save the memory occupation.

Fault simulation without output Masks

Once the input patterns have been created, it is possible to proceed with the Fault Simulation step to verify their effectiveness. The methodologies adopted depend on the capabilities of the simulation tool. In this experience, it was adopted the method based on the registration of the eVCD file. The **Extended Value Change Dump (eVCD)** is a type of file that records all the input and output signals through a well-defined format compatible with a large number of verification tools. This file can be used as input to carry out circuit simulations or it can be recorded through a simulation performed with another testbench. By running the RTL simulation of the implemented patterns, it is possible to obtain the eVCD file and use it as a testbench for the Fault Simulator, thus obtaining the fault-free "Gold Machine" simulation. At this point, the Fault Simulator proceeds with the fault injection comparing the result with the Gold machine. If the result shows discrepancies, this means that the fault has been identified. In a completely automatic way, this procedure is repeated fault by fault, providing the final Fault Coverage value. The steps are summarized in **figure 5.4**.

This first fault simulation aims to roughly evaluate the effectiveness of the test program by observing the results on all the POs including all the control signals of the module. As explained below, the results obtained in this step cannot be considered valid but indicate the methodology used for creating the patterns.

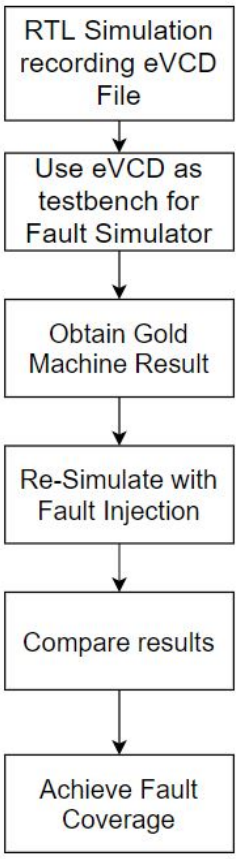

Figure 5.4: Fault Simulation Flow

Fault list managing

If the algorithms adopted to create the patterns are not sufficient to satisfy the required Fault Coverage requirement, it is necessary to proceed with managing the fault list. This step is very important to avoid re-simulating faults already identified by other test programs in case it is necessary to add new ones. The reduction in the number of tested faults has a significant impact on the time needed to complete the Fault Simulation. In real circuits, performing the fault simulation from the beginning can take a long time even more than a week. Obviously, if the patterns do not give satisfactory results, it is necessary to rework (or delete!) the latter, simulating all the faults in the initial fault list.

Signature implementation

As will be explained later. The test software makes a comparison of an "accumulated" result from the various operations included in the test programs. In order to make

as few comparisons as possible it is necessary to develop a signature related to the results of all the operations carried out. For this purpose, dedicated procedures must be introduced within the different tests. Starting from the pseudo example code:

```
test_program(){
    stim_A[] ={0x00000000,0x11111111,0x22222222, ... , 0xFFFFFFFF }
    stim_B[] ={0x00000000,0x11111111,0x22222222, ... , 0xFFFFFFFF }

    for(i=0; i<size(stim_A); i++){
        for(k=0; k<size(stim_B); k++){
            result = stim_A[i] op1 stim_B[k];
            store(result);
            result = stim_A[i] op2 stim_B[k];
            store(result);
            ...
            so on
            ...
        }
    }
}
```

Instead of carrying out the store operations after each instruction, it is necessary to "jump" into a procedure capable of accumulating a signature, as shown below.

```
test_program(){
    stim_A[] ={0x00000000,0x11111111,0x22222222, ... , 0xFFFFFFFF }
    stim_B[] ={0x00000000,0x11111111,0x22222222, ... , 0xFFFFFFFF }
    for(i=0; i<size(stim_A); i++){
        for(k=0; k<size(stim_B); k++){
            result = stim_A[i] op1 stim_B[k];
            final_result = signature_fun(result);
            result = stim_A[i] op2 stim_B[k];
            final_result = signature_fun(result);
            ...
            so on
            ...
        }
    }
    return final_result;
}

signature_fun(result){
    mask_1=0xFFFFFFFF,
    mask_2=0x5EED5EED;
    mask_3=0xCAFECAFE,
```

```
    result = result XOR mask_1;
    result = result XOR mask_2;
    result = result XOR mask_3;
}
return result;
```

Using this strategy, only one result that keeps track of all the previous ones will be compared. Given that, in most cases, the length is maintained between the intermediate and final results, the implementation of a signature algorithm is not too complex. It is recommended to use "transparent" operations (XOR) to bits changing in order to avoid the presence of aliasing and therefore the loss of Fault Coverage. At this point, each test programs will provide a unique signature which will be saved in a reserved memory space. Therefore, the presence of a fault will cause a discrepancy between the expected signature and the one calculated by notifying the identification of the latter.

Context Saving

Since test programs are run through a function call it is very important to consider the possible loss of specific return addresses saved in the register file by the ABI interface. The latter ensures that the function calls are executed in a "conventional" way using a strategy aimed at correctly saving the context. The procedure is completely automated by the C compiler but the latter is "unaware" of the presence of an ASM code that could overwrite the special registers that contain, for example, the stack pointer and return addresses. With reference to table 3.1 it is possible to identify some of these registers used by the ABI. Temporary registers can be modified at will and are best suited for manipulation by test instructions. Sometimes in order to obtain an acceptable coverage value, it is necessary to have a number of registers higher than the temporary ones, overwriting some registers used by the ABI. In this situation there is a risk of corrupting the expected flow of instructions, creating stalls in the program. To solve the problem it is recommended to introduce a "prologue" and an "epilogue" to each test program, capable of capturing the initial state of the register file and restoring it at the end of the test procedure. The solution, if applied at the right time, can be used to save the context even in entry and exit from the test routines, thus ensuring the operating system an exact restore of the initial conditions. [3]

Fault Simulation including output Masks

The final step of the implementation of the test program takes place through a fault simulation by placing masks on all outputs where the signature is not visible to evaluate the presence of aliasing caused by the following reasons:

- Signature Algorithm
- Faults that affect the control signals

A bad implementation of the signature algorithm may affect the trace-ability of the fault detected masking its effects on the signature. A fault that is propagated to an output control signal is not trivial to see via software since it could not creates differences in the signature. As will be explained in the next sections, the software needs to be initialized in order to activate some specific functionalities to support the possible "failures" caused by this type of fault. In some cases, this fault simulation will reduce the score of the Fault Coverage but permits an evaluation of the effectiveness of the test program from the signature comparison point of view.

5.2 Low-Level API

The first level of functions provides the system with support to successfully run test programs. Previously the fault detection was performed through a dedicated tool (Fault simulator) that directly observed the values scoping the POs. In this phase, however, the software must be able to cover the role of the tool by detecting inconsistencies in the signature but, at the same time, ensuring the program flow compatible with the presence of any OS.

It mainly takes into account the following aspects.

- Test flags
- Failures caused by test programs
- Test programs interface
- Signature comparison

5.2.1 Test Flags

For correct coordination at multi-core level, the software needs to set some flags relating to the execution of the test. These flags can be saved in a dedicated memory area and must be unique for each core. Mainly, flags must take into account the following aspects:

- **Running Flag** - Notify if the core is in test mode
- **Illegal Instruction Detection** - Check if all the instructions have been decoded correctly
- **Test Result** - Analyze the test outcome

Obviously the flags shown are only indicative, for more advanced implementations the number of the latter may be higher in order to cover the needs of software designers.

5.2.2 Test Programs Failures Detection

As previously mentioned, the observation of the signature is not able to cover the effects caused by all possible faults. During the execution of the test some faults could create problems with the correct flow of the program, generating issues in the signature computation. For example, a possible fault inside the Decode Stage could trigger an exception that would lead to the execution of different instructions rather than those expected. For this purpose, it is necessary to think about all the possible failures generated by the faults reflected on the system control logic. Some solutions could be the following:

- Modified Illegal Instruction Exception Routines
- Add Watchdog Timer

The first solution aims to solve possible faults in the IF-ID stages providing a modified handler for the illegal instruction exception which set the flag in the correct memory location. Furthermore, the handler needs to support a correct return to the program "skipping" the faulty instruction in order to conclude the test. If well implemented the exception handler can use a set of variables able to keep trace of the faulty instructions in order to have diagnosis features. The second solution take into account the occurrence of some failures which can cause unexpected looping behaviour causing infinite loop of execution time out of the fixed boundaries. This can be implemented through a watchdog timer properly configured in the initialization phase of the test.

5.2.3 Test programs interface

A possible solution for interfacing, between the Low-Level API and the test program, is proposed by exploiting the ASM Extended libraries. This library allows you to use C variables as input and output within the ASM code snippet. With this solution, it is easy to save and restore the Register File backup using a pointer to the memory location. Furthermore, the same approach facilitates the return of the signature by allowing saving in an easy-to-use C variable. The use of the ASM Extended is shown by the following pseudo-code:

```
int test_program_i( int *reg_file_pointer){
    int signature = 0;
```

```
    asm extended(

        /// PROLOGUE ///////////////////////
        sw r0,%[rf_backup_loc]
        sw r1,(%[rf_backup_loc] + 4)
        .
        .
        .
        sw r31,(%[rf_backup_loc] + 4*32)

        ////////////////////////
        // Test program HERE //
        ////////////////////////

        /// EPILOGUE ///////////////////////

        lw r0,%[rf_backup_loc]
        lw r1,(%[rf_backup_loc] + 4)
        .
        . //    skip the load in the register which contains the
    signature
        .
        lw r31,(%[rf_backup_loc] + 4*32)

        /// STORE SIGNATURE ////////////////
        sw r5, %[address_mem_sign]


    :[address_mem_sign] "=m" (signature)
    :[rf_backup_loc] "m" (reg_file_pointer)

    );
    }
    return signature;
}
```

The library has numerous features that must be used with care because, during the compilation phase, the program could assume an unexpected flow.

5.2.4 Signature comparison

This step is very intuitive and involves the comparison between the calculated signature and the "gold" signature. The software will need a pool of pre-calculated signatures saved in an appropriate memory location that can easily be Read-Only. For some test programs, whose signature stabilization is complicated due to the strong dependence on the context in which the test program is launched, it might

be useful to adopt a "dynamic" signature system thus allowing access to the pool. The result of the comparison will be reflected on the result flag allowing other cores to evaluate the test outcome.

5.3 High Level API

The purpose of the high level API is to allow the correct execution of one or more test programs on the different cores. Mainly its functions are:

- Test Sets Creations
- Multi-Core Addressing
- Test Program Launching
- Test Result Evaluation

5.3.1 Test Sets Creations

The possibility to decide which and how many test programs can be executed in sequence allows to optimize the use of the time intervals left free by the OS. For this reason, it is necessary to provide an interface for creating test sets. A possible solution is to create dedicated data structures containing enable flags that allow to orchestrate the different test programs.

5.3.2 Multi-Core Addressing

Once the test set is ready it is possible to launch it on the different cores using the solution mentioned in section 3.1.2 (TLP). The methodology to carry out this task is highly dependent on the availability in terms of Kernel functions, therefore different approaches may be required. The test software must ensure that it is possible to run simultaneously on multiple cores test sets without creating conflicts. For this reason, it is recommended to allocate memory spaces reserved for each core, in which, store the different test flags seen in the previous section. Similarly, it is suggested to allocate a dedicated location, for each core, to backup the register file. A careful analysis of the linker file is therefore crucial to understand exactly in which memory sections to save the information as well as the correct addressing of the latter through C structures.

5.3.3 Test Program Launching

The high-level APIs must allow the correct execution of the tests selected within the set. The software will be in charge of performing all the initialization functions

implemented in the lower level. Once the system is ready it will be possible to execute each test program in sequence. A solution proposed leverages the use of loops where the index points to a specific function pointer.

5.3.4 Test Result Evaluation

Once all the cores reach the synchronization barrier, the test result evaluation can be performed simply reading all the memory area dedicated to the performance flags.

5.4 The study-case software

The SBST developed for the VEP design was developed following the guidelines shown in the previous section.
As previously mentioned, due to the high complexity of the VEP design, only the DSP Cluster was taken into consideration. In turn, it was not possible to develop a program covering all modules. The modules tested and analyzed are the RISC-V Cores and the Shared FPU unit with the implementation of the software interface required for concurrent multi-core testing.

5.4.1 Test Suite

The following tables show all the test programs divided according to the type of approach used during development.

- Pseudo-Random (Table 5.1)
- Pseudo-Random + ATPG (Table 5.2)
- Memory Algorithm (Table 5.3)

Table 5.1: Test programs adopting Pseudo-Random.

Test Program Label	**Main Module Targeted**
TEST_ALU_ADD_CMP	ALU
TEST_ALU_DIV	ALU and DIVISOR
TEST_ALU_IMM_VAL	ALU
TEST_ALU_NN_PV	ALU
TES_CMPLX_OPS	ALU and Multiplier
TEST_MUL_OPS	Multiplier

TEST_MUL_IMM_VALUES	Multiplier
TEST_MUL_NN_PV	Multiplier
TEST_MUL_RNN	Multiplier
TEST_CMPRSS_DEC_A	Decode stage
TEST_CMPRSS_DEC_B	Decode stage
TEST_LOAD_STORE	LS Unit
TEST_BRANCH_UNIT	Branch Unit
TEST_HWLOOP	HW Loop Controller
TEST_FPU_LITE	Decode stage and FPU
TEST_HWLOOP	HW Loop Controller

Table 5.2: Test programs adopting Pseudo-Random and ATPG .

Test Program Label	**Main Module Targeted**
TEST_FPU_FMA	HW Loop Controller
TEST_FPU_FMASP	FPU FMA
TEST_FPU_FMA	FPU FMA
TEST_FPU_FDIVSQRT	FPU DIV
TEST_FPU_FCMP	FPU CMP
TEST_FPU_FCVT	FPU CVT
TEST_FPU_FCVT_SP	FPU CMP
TEST_FPU_FCVT	FPU CVT

Table 5.3: Test programs adopting memory algorithm.

Test Program Label	**Main Module Targeted**
TEST_REG_FILE	Decode stage and FPU
TEST_TCDM_XBAR_CTRL	TCDM interconnection
TEST_TCDM_XBAR_DATA	TCDM interconnection

As it is possible to see, all the programs were implemented using the pseudo-random approach because it guaranteed good results in terms of fault coverage. The ATPG approach was used as a "refinement" by applying it on reduced fault lists and with a high number of input constraints in order to reduce as much as possible

the difficulty of translating the vector tests generated in the STIL file. Furthermore, the ATPG has been effective identifying some "special" patterns dedicated to the FPU. Each test program interface as been implemented as the example in **section 6.2.3**.

5.4.2 LLD and HLD Libraries Implementation

The LLD and HLD C libraries have been implemented just after a first version of the test suite. The two libraries comprise all the considerations explained in **section 6.2 and 6.3**.

It should be noted that, during the development, some problems were encountered by launching some test programs in sequence. As previously said, the implementation adopted in this context is far from a real version dedicated to business purposes. According to some analyzes, the use of the ASM Extended library and the test program interface must be reworked in order to ensure a correct and stable version of the software. However, the results that will be shown in the next chapter have been obtained in multiple merges of mutually compatible test sets.

L2 Memory Organization

The L2 memory contains the performance flags and the signatures pool. The performance flags location takes 192 bytes accordingly with the following equation:

$$Size = 32bit \cdot N\_CORES \cdot N\_FLAGS \tag{5.3}$$

The choice to allocate the space inside the L2 memory is due to the fact that it is a memory "closer" to the FC Core which is in charge to checks the test results at the end of the test procedure. Furthermore, this memory is not involved during the DSP Cluster operations as data memory, thus, it is fetched only to load the instructions into the DSP Instruction Cache.

The signature pool has also been allocated in L2 memory within a Read-Only space using the Constant directive. The reserved space is 100 Bytes calculated through the following equation:

$$Size = 32bit \cdot N\_TEST \tag{5.4}$$

For both, the space has been allocated in the .data memory section.

L1 Memory Organization

In VEP design the L1 memory is inside the DSP cluster and it is in charge to save the data of the operations performed on all the sixteen cores in their respective stack

locations. The software developed reserves dedicated spaces for the backup of the register files in the specific section .data. With this solution all computations remain inside the DSP Cluster in order to minimize contentions to use the external memory L2. The Cores will then use the L2 memory exclusively to load performance flags and gold signatures, minimizing communications and therefore possibles contentions and stalls.

Illegal Instruction Handler Modifications

The Illegal Instruction Handler has been modified to support the writing of the appropriate flag for the identification of illegal opcodes caused by possible faults within the modules involved in fetching and decoding instructions. The illegal instruction flag is set only when the Running Flag is active which means that the test mode is activated.

5.4.3 Software Functioning

The following steps summarize the operations perfomed by the software. In **figure 6.5** it's possible to identify how the software is structured on the hardware resources.

1. Initialize Illegal instruction Detection and set running flag
   (a) Call Init Function
   (b) Set running flag and initialize illegal flag to initial value 0
2. Run Test and compute signature
3. Compare signatures and check presence of illegal instructions
   (a) Check Illegal instruction flag
   (b) Load Signature
   (c) Compare signature
4. Write test Result
5. FC CORE checks the results

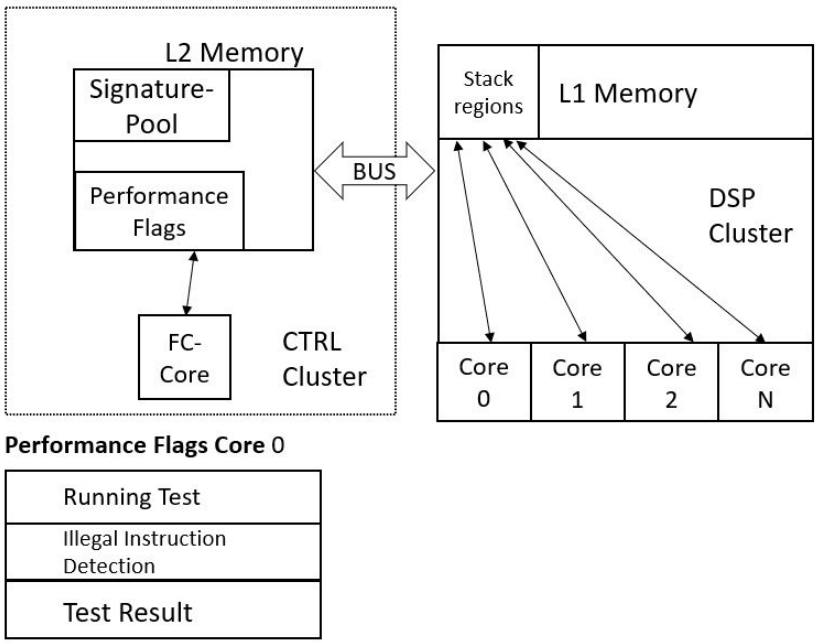

Figure 5.5: SBST Functioning on VEP Design

Chapter 6

Results and Analysis

In the following conclusive chapter are analyzed all the results obtained through the SBST software highlighting the limitations and problems faced during the development. Furthermore, showing the results, many considerations are collected to expand and finalize the test suite leading towards the conclusions regarding the effectiveness of the method.

6.1 Results on VEP design

The following metrics were used to evaluate the feasibility of the method obtained on VEP design:

- Fault Coverage
- Simulation Time
- Fault Simulation Time
- Absolute Time

6.1.1 Fault Coverage

As mentioned above, the effectiveness of the test programs has been evaluated through local simulations of the different modules of the DSP Cluster by observing only the POs involved in the signature propagation. Initially, the idea was to simulate the entire cluster running the entire software recording the PIs and POs. Due to factors inherent to computational resources, module complexity and some parts still under development, this simulation turned out to be too ambitious. In order to obviate to this problem it has been tried to replace the "sleeping" modules with the respective "stubs". The solution for a first moment has been

found effective for the RTL simulations, while, numerous discrepancies have been found in the Gate-Level simulations. Moreover, the clock-gating manages the generations of simulation events on all the circuitry through the Clock Enable signal. This consideration led to abandoning the idea of stubs because it would not bring tangible benefits in terms of simulation time since the not used circuitry is not evaluated. Additionally, the develop of a fully functional stub can take weeks of development. Reasoning on how to improve the times required by the fault simulation the only feasible solution was to use described circuits mixing Gate-Level components with RTL components. Unfortunately, the supplied testing tool did not support the description of the components in SystemVerilog and therefore it was not possible to verify the effectiveness of the solution. The choice of the masks on the POs allows to obtain an estimate of the Fault Coverage able to better approach the "ideal" one of the complete cluster but, at the same time, to stay away from the overestimation of the fault simulation obtained considering also the faults propagated on the control output signals. The results in **figure 6.1** are reproducible by running all the test suite on all the cores concurrently.

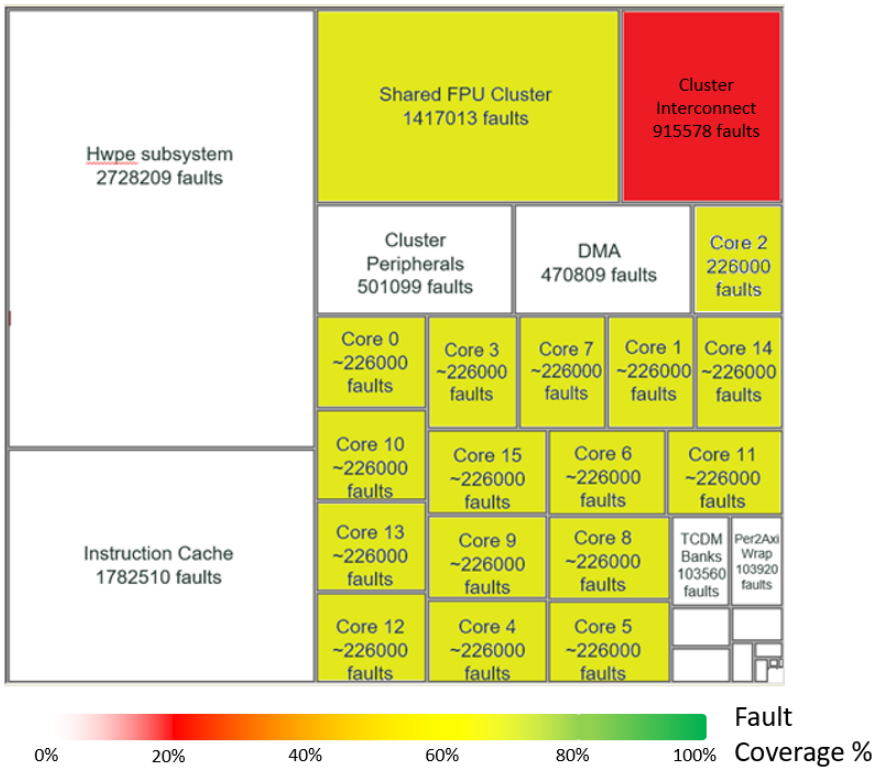

Figure 6.1: Result on DSP Cluster

The white boxes, in the right corner of figure 6.1, without a label are written in boldface in **table 6.1**.

11688826	**Top Module**	Fault Coverage %
51040	**/Cluster Bus**	N.C.
28836	**/AXI to Memory Bus**	N.C.
13188	**/AXI to Peripheral Bus**	N.C.
566	**/Peripheral Demux**	N.C.
105172	/Peripheral to AXI Bus	N.C.
1077114	/Cluster Interconnect Bus	19%
477070	/DMA	N.C.
506974	/Cluster Peripherals	N.C.
226924	/Core 0 Region	73.75%
226124	/Core 1 Region	73.75%
226732	/Core 2 Region	73.75%
226192	/Core 3 Region	73.75%
225392	/Core 4 Region	73.75%
225346	/Core 5 Region	73.75%
225636	/Core 6 Region	73.75%
226008	/Core 7 Region	73.75%
225400	/Core 8 Region	73.75%
225968	/Core 9 Region	73.75%
226732	/Core 10 Region	73.75%
225226	/Core 11 Region	73.75%
225888	/Core 12 Region	73.75%
226226	/Core 13 Region	73.75%
225952	/Core 14 Region	73.75%
226184	/Core 15 Region	73.75%
1447620	/Shared FPU Cluster	75%
2774690	/Hardware Peripheral Subsystem	N.C.
1361710	/Instruction Cache	N.C.
103684	/L1 Memory Banks	N.C.
118890	**/Other Modules**	N.C.

Table 6.1: DSP Cluster fault coverage

RISC-V NN Core

RISC-V has been covered with a percentage of 73.75%. The maximum coverage achieved was 88% without stabilization of signatures and use of masks on POs. The overview of the coverage is in **table 6.2** .

#faults	Instance name	Fault Coverage %
221710	/top_Module	73.75
18212	/if_stage_i/	64.06
13188	/if_stage_i/prefetch_128_buffer_i	65.86
1576	/if_stage_i/hwloop_controller_i	56.85
1616	/if_stage_i/compressed_decoder_i	85.27
84750	/id_stage_i	72.41
41910	/id_stage_i/register_file	77.55
6670	/id_stage_i/decoder_i	55.02
2268	/id_stage_i/controller_i	37.80
184	/id_stage_i/int_controller_i	0.00
4648	/id_stage_i/hwloop_regs_i	78.73
96184	/ex_stage_i	85.17
38824	/ex_stage_i/alu_i	84.43
48676	/ex_stage_i/mult_i	95.73
1740	/ex_stage_i/apu_disp_i	19.71
6290	/load_store_unit_i	82.90
14922	/cs_registers_i	16.67

Table 6.2: RISC-V NN Core fault coverage

One of the main sources of aliasing was caused by the lack of dedicated programs for the Control and Status Registers due to difficulties in the signature stabilization, which, guaranteed an increase of 3-4% of the RISC-V NN coverage. The results obtained exclude the addition of the program dedicated to the MAC operations circuitry for neural networks due to some recent changes to the ISA of the RNN extension. With the previous version of ISA the execution of this test program guaranteed an increase in the order of 3-4% of the total. Additionally, test programs for the FPU that could detect faults in the APU, Fetch and Decode stages were omitted with a 1-2% reduction in global coverage. It is therefore clear that with the necessary optimizations and some additions it is very feasible to obtain a coverage between 80 and 85%, which is an acceptable result for in-field testing practices in many application domains.

Shared FPU Cluster

The shared FPU is one of the macroblocks that make up the DSP Cluster. The test programs have allowed to obtain a good total coverage value of 75%. The **figure 6.1** shows a general view of the results obtained divided by regions. In **table 6.3**, the summary of the coverages.

The **FPU_Unit** was tested by dedicated test programs in charge to exercise

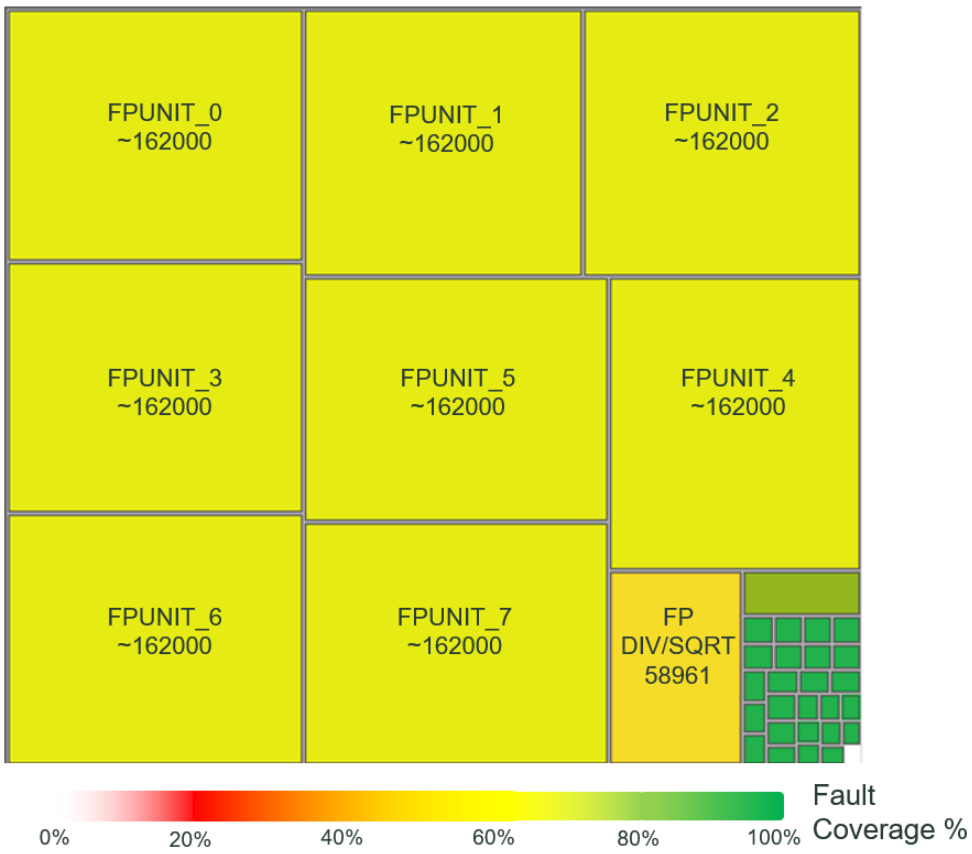

Figure 6.2: Result on DSP Cluster

each operational group. The results are summarized in **table 6.4**.

The use of the pseudo-random approach has proved to be less effective on this type of circuit because, the format of the IEEE 754 operands for floating point operations requires a more in-depth study on the latter. A choice of more "functional" patterns would guarantee to better exercise all the lines within each operational group. This type of approach requires an in-depth study of the architecture but could improve overall coverage by 8-10%, with significant improvements in execution time. Some of the tests used have been developed using the ATPG-based approach in a complementary way, ensuring significant improvements especially to group 2, involved in the floating comparison operations bringing the coverage from around 30% to the actual value of 64.93% using only not sequential patterns.

The **FP_DIVSQRT Unit** follows the same speech made previously, although the choice of patterns could be even more complex due to the high sequentiality of the module. Even with an in-depth study it is feasible to improve its coverage to a

#faults	Instance name	Fault Coverage %
~1417000	/top_module	~75.00
~162000	FPunit_0	72.88
~162000	FPunit_1	72.88
~162000	FPunit_2	72.88
~162000	FPunit_3	72.88
~162000	FPunit_4	72.88
~162000	FPunit_5	72.88
~162000	FPunit_6	72.88
~162000	FPunit_7	72.88
58961	FPDIV/SQRT	72.88
~55000	Mux_logic	90.00

Table 6.3: Shared FPU fault coverage

#faults	Instance name	Fault Coverage %
171314	/top_module	72.88
89962	/gen_operation_groups_0_i_opgroup_block	81.45
1338	/gen_operation_groups_1_i_opgroup_block	9.46
30208	/gen_operation_groups_2_i_opgroup_block	64.93
47282	/gen_operation_groups_3_i_opgroup_block	62.23
1846	/i_arbiter	54.90

Table 6.4: FPU fault coverage

higher value than the current 49%.

The **MUX logic** has been effectively covered simply by launching test programs on all the cores simultaneously. Almost all multiplexers achieved a coverage of between 87% and 92%.

Summarizing, obtaining a global coverage of 85% on the whole block dedicated to floating point operations is a feasible operation provided that accurate circuit analysis is carried out in order to identify the correct patterns. From the point of view of signature stability no noteworthy difficulties have been encountered.

Cluster Interconnection

The Cluster Interconnection is a TCDM XBAR which orchestrates the communication among all the internal modules of the cluster. During the experience some simulations were made with simple programs involving communications between the cores and memory banks. Through two simple programs a coverage of about 20% was obtained. The result, although very low, may indicate that to detect a

good number of faults in the XBAR it is necessary to involve all the modules that it interfaces. One hypothesis is that the module can be tested for side-effect once dedicated test programs have been developed for all modules in the cluster.

Instruction Cache

The instruction cache is a very complex module to test with the SBST technique because of several reasons as:

- Generate a stable signature
- Difficult to handle its content (TAG and DATA) since it contains only instructions
- Not easy implement an algortithm able to exercise the control logic related with the memory address.

During the experience it was not possible to implement a valid strategy in order to derive a value due to limitations to perform a fault simulation that could give significant results. However, some strategies have been designed taking advantage of the possibility to target not-cacheable memory areas. [23]

DMA

DMA is a block dedicated to transferring entire areas of memory from outside or inside the DSP cluster completely independently of the cores. Although it was not possible to implement a dedicated module for reasons of time, it is reasonable to test it using the SBST technique. A possible idea involves the transfer of entire areas of memory preloaded with specific patterns. It might be more problematic to effectively stimulate the control part concerning address management as it might not be feasible to use DMA on all available memory due to design limitations. Some techniques are currently under development and research.

Hwpe subsystem, Cluster Peripherals and other modules

Due to time issues and limitations due to the simulative capabilities of the test software it was not possible to consider the remaining blocks in the DSP Cluster. For this purpose, the idea is to return to these modules once you have obtained the possibility to simulate the entire DSP cluster in a reasonable time.

6.1.2 Simulation Time

During the development it was necessary to simulate the entire suite several times through RTL and Gate-Level simulations of the programs in order to verify their

operation. In this section, the idea is to give an indication on the times required to simulate the waveforms of the entire test suite with the simulation tool. The analysis is in **table 6.5**

Module Simulated	Test Programs	Level	Simulation Time (min)
DSP CLUSTER	RISC-V SET+FPU LITE	RTL	5 min
DSP CLUSTER	RISC-V SET+FPU LITE	GATE	20-25 min
DSP CLUSTER	All FPU Tests	RTL	7-8 min
DSP CLUSTER	All FPU Tests	GATE	25 min

Table 6.5: RTL Waveform Simulation Time

As is possible to notify, The time of simulation demanded from the Gate level turns out multiplied of a factor equal to 4-5 times regarding the simulation RTL.

6.1.3 Fault Simulation Time

Below, in **table 6.6** are reported the times required to perform the fault simulation of the blocks examined with the respective test sets considering the complete fault lists. To fault simulate the shared FPU in a reasonable period of time it is

Module Simulated	Test Programs	Simulation Time (hours)
DSP CLUSTER	RISC-V SET+FPU LITE	Not Defined
RISC-V	RISC-V SET+FPU LITE	16-20
Shared FPU	All FPU Tests on all the cores	20-24
FP Unit	All FPU Tests	2

Table 6.6: Fault Simulation Time

mandatory remove from the list all the faults coming from the different FP units. In this way the complete fault list counts around 150k faults and it is possible to test them in less than one day. The complete simulation of the cluster beyond to demand fault simulation times too much long, it has shown some issues during the logical simulation performed by the testing tool due to the presence of some hardware aspects still in phase of development.

6.1.4 Absolute Time

In this paragraph are summarized in the table the times required by the test programs to be performed by SoC. The results have been obtained using the value obtained at the end of the RTL simulation.

Module Simulated	Test Programs	Simulation Time
DSP CLUSTER	RISC-V SET+FPU LITE single core	10 ms
DSP CLUSTER	FPU Test Set on single Core	12 ms
DSP CLUSTER	FPU Test Set on all cores	16 ms

Table 6.7: Absolute Time

It was not possible to measure the time of each of the test programs because it would have required an excessive number of simulations and since these versions were not definitive it would have had less meaning. However, it is clear that the duration of the test program is "relatively low" demonstrating the effectiveness of the method for in-field testing purposes. Obviously it is necessary to evaluate the application domain and all the limits that it involves. Comparing with DfT techniques such as BIST or Scan-Chains it is clear that, the time for testing operations can be on average one order of magnitude shorter because the SBST does not require Shift-In and Shift-Out phases.

6.2 Conclusive analysis and considerations

The SBST method has proven once again effective for testing all computational parts of the SoC. As expected, the big limitations concern the high computation required to perform sequential fault simulations of very large circuits that count millions of faults. To solve the problem the solutions, as anticipated, concerning the use of hybrid circuits between the Gate and RTL level. Anyway, in order to make this kind of description completely effective, it could be necessary to spend a lot of time to solve possible discrepancies. Another alternative to solve the problem is to use multiple licenses of the testing tool that allow running the fault simulation in parallel and on multiple cores. Obviously, this solution requires a considerable economic effort. From the point of view of fault coverage, it could be very useful to develop an algorithm capable of "capturing" the state of the SoC allowing later to calculate "dynamic" signatures. This addition could guarantee a substantial increase in coverage for control units and CS registers. The technique can be defined feasible on multi-core systems providing software able to correctly orchestrate the testing operations avoiding possible conflicts, mainly, due to shared memory areas. Considering this last aspect the software will have to reduce the possible contentions between the data buses in order to reduce possible "stalls" slowing down the testing procedure. With regard to the circuitry "around" the cores, it is necessary to carry out a further study once the environment is ready to perform big fault simulations. This preliminary study did not reveal any other problems that do not allow the use of the SBST technique on multi-core circuits.

Bibliography

[1] J.C. Laprie. *Dependability: Basic Concepts and Terminology*. SpringerVerlag, 1992 (cit. on p. 2).

[2] Matteo Sonza Reorda, Paolo Bernardi, Michelangelo Grosso, and Ernesto Sanchez. *Slide course: Testing and Fault Tolerance*. Politecnico di Torino. 2020 (cit. on pp. 2, 61).

[3] Seifedine Kadry. «A New Proposed Technique to Improve Software Regression Testing Cost». In: International Journal of Security and its Applications (Nov. 2011) (cit. on pp. 9, 66).

[4] Paolo Bernardi, Riccardo Cantoro, Sergio De Luca, Ernesto Sànchez, and Alessandro Sansonetti. «Development Flow for On-Line Core Self-Test of Automotive Microcontrollers». In: TRANSACTIONS ON COMPUTERS n65 (Mar. 2016) (cit. on p. 25).

[5] Wikipedia contributors. *Flynn's taxonomy — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-December-2020]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Flynn%5C%27s_taxonomy&oldid=992186057` (cit. on p. 27).

[6] Wikipedia. *Instruction level parallelism — Wikipedia, L'enciclopedia libera*. [Online; in data 6-dicembre-2020]. 2019. URL: `http://it.wikipedia.org/w/index.php?title=Instruction_level_parallelism&oldid=109183493` (cit. on p. 28).

[7] Wikipedia contributors. *Task parallelism — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-December-2020]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Task_parallelism&oldid=960925335` (cit. on p. 29).

[8] Wikipedia contributors. *Multi-core processor — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-December-2020]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Multi-core_processor&oldid=989274864` (cit. on p. 29).

[9] OmniSci. *GPGPU Definition*. Available on line. URL: https://www.omnisci.com/technical-glossary/gpgpu (cit. on p. 29).

[10] Mingzhe Wang, Bo Wang, Qiu He, Xiuxiu Liu, and Kunshuai Zhu. «Analysis of GPU Parallel Computing based on Matlab». In: (2020) (cit. on p. 30).

[11] Wikipedia. *CUDA — Wikipedia, L'enciclopedia libera*. [Online; in data 6-dicembre-2020]. 2020. URL: http://it.wikipedia.org/w/index.php?title=CUDA&oldid=116820339 (cit. on p. 30).

[12] SUSE. *Definition Computer Cluster*. Available on line. URL: https://susedefines.suse.com/definition/computer-cluster/ (cit. on p. 30).

[13] Michael Nielsen. *Using neural nets to recognize handwritten digits*. Available on line. Dec 2019. URL: http://neuralnetworksanddeeplearning.com/chap1.html (cit. on p. 31).

[14] Jordi Torres.ai. *Learning process of a neural network*. Available on line. Sept. 2018. URL: https://towardsdatascience.com/how-do-artificial-neural-networks-learn-773e46399fc7 (cit. on p. 34).

[15] Md. Haidar Sharif and Osman Gursoy. «Parallel Computing for Artificial Neural Network Training using Java Native Socket Programming». In: 6.1 (Feb. 2020) (cit. on p. 39).

[16] R. Andri, T. Henriksson, and L. Benini. «Extending the RISC-V ISA for Efficient RNN-based 5G Radio Resource Management». In: (2020), pp. 1–6. DOI: 10.1109/DAC18072.2020.9218496 (cit. on pp. 39, 52).

[17] Andrew Waterman and Krste Asanovi. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. Available on line. Dec 2019 (cit. on p. 40).

[18] Andreas Traber, Michael Gautschi, and Pasquale Davide Schiavone. *Ri5CY User Manual, Rev.4*. Available on line. 2019. URL: https://pulp-platform.org/docs/ri5cy_user_manual.pdf (cit. on pp. 44–46).

[19] Dolphin Design. *platform-solutions*. Available on line. URL: https://www.dolphin-design.fr/ (cit. on p. 47).

[20] Dolphin Design. *VEP specifications*. Reserved. URL: https://www.dolphin-design.fr/ (cit. on p. 48).

[21] C. Jie, I. Loi, L. Benini, and D. Rossi. «Energy-Efficient Two-level Instruction Cache Design for an Ultra-Low-Power Multi-core Cluster». In: (2020), pp. 1734–1739. DOI: 10.23919/DATE48585.2020.9116212 (cit. on p. 54).

[22] Dean Cracknell and Micross Components. *STIL Language Test Vector Format (Simplified)*. Available on line. URL: https://shop.micross.com/pdf/Micross_Technical_Paper-STIL_Language_Test_Vector_Format_Simplified.pdf (cit. on p. 62).

[23] G. Theodorou, N. Kranitis, A. Paschalis, and D. Gizopoulos. «A software-based self-test methodology for in-system testing of processor cache tag arrays». In: *2010 IEEE 16th International On-Line Testing Symposium*. 2010, pp. 159–164. DOI: 10.1109/IOLTS.2010.5560214 (cit. on p. 82).