POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

AI-Oriented Hardware Accelerators Reliability

Supervisors

Matteo SONZA REORDA

Juan David GUERRERO BALAGUERA

Josie Esteban RODRIGUEZ CONDIA

Candidate

Yanghejian ZHANG

April 2024

Table of Contents

List of Tables

List of Figures

Acronyms

ATPG
Automatic Test Pattern Generation

BIST
Built-in Self-test

CASP
Concurrent Autonomous Chip Self-test Using Stored Test Patterns

CDMA
Convolution Direct Memory Acccess

CMAC
Convolutional Multiply-accumulate

CSA
Carry-save Adder

DFT
Design For Test

DL
Deep Learning

DPPM
Defect Part Per Million

IC
Integrated Circuits

NVDLA
NVIDIA Deep Learning Accelerator

Chapter 1

Introduction

With the rapid advancement of artificial intelligence, AI-oriented hardware accelerators have become a crucial technology for enhancing the performance and efficiency of AI applications. These hardware accelerators include Graphics Processing Units, Tensor Processing Units, specialized ASICs, and various other custom-designed hardware. While they excel in speeding up AI tasks and improving efficiency, their reliability is also a critical concern.

Hardware reliability is paramount for AI applications because training and inference of AI models often require significant computational resources and time. Hardware failures during task execution can lead to data loss, task interruptions, or system crashes, resulting in a substantial impact on productivity and availability. Therefore, the issue of reliability in AI-oriented hardware accelerators is a crucial challenge that demands serious attention.

One of the most important parts of the hardware accelerator, and also the part with the highest number of logic gates, is the convolutional arithmetic part, which is mainly composed of multipliers and accumulators. For this part of the hardware, a simple and efficient test becomes the key to improve the stability of the hardware as well as a quick self-test.

In-field testing plays a crucial role in verifying system functionality, assessing performance and reliability, identifying potential issues, enhancing user experience, and ensuring compliance with legal and industry standards. By simulating real-world usage scenarios, In-field testing ensures that systems operate as designed and exposes potential issues for timely resolution, thereby improving system quality and reliability. Additionally, user feedback and observations obtained through In-field testing help optimize user experience and increase satisfaction. For certain industries, In-field testing is also a necessary step to comply with legal regulations and industry standards, mitigating legal risks and enhancing market competitiveness.

To do so, In this thesis, we present a methodology for DL (Deep Learning) accelerator design aimed at generating high-quality functional tests that can be

deployed during normal phases of real-world operation. Our methodology is effective in detecting various types of permanent failures including early failures, circuit aging, manufacturing defects, and variations[1], which are important reliability concerns. Unlike structural testing, functional testing does not rely on hardware structural testing support, such as CASP(Concurrent Autonomous Chip Self-test Using Stored Test Patterns)[2],[3] or logic BIST(Built-in Self-test)[4], which is still not available in some DL accelerators, despite the growing popularity of structural testing capabilities in self-testing applications covering a wide range of domains from data centers to automotive systems[4],[5],[6],[7]. Therefore, for DL accelerators lacking structural testing support, in-field testing in real-world deployment scenarios remains critical.

We use NVIDIA's open source chip NVDLA, a neural network chip designed to accelerate deep learning, as an example for this thesis.

1.1 Problem To Be Solved

Methods of In-field testing to generate excitation can vary depending on the characteristics and requirements of a specific system, device or chip. The main ones are randomized testing and manual design of test cases, where randomized testing is a simple but effective method for test excitation generation. By randomly generating test vectors, different combinations of inputs to a circuit can be covered and thus tested. Although random testing does not guarantee complete coverage of all possible scenarios, it can be used to quickly evaluate the basic functionality of a system. And manually designing test cases For some special cases or specific functional tests, manually designing test cases may be an effective method. Test engineers design representative test cases according to the characteristics and requirements of the system or chip, and manually generate the corresponding test incentives. However, it usually takes more time and labor cost, therefore, this thesis proposes a method of manually designing test cases considering the single stuck-at model, which combines ATPG and can efficiently generate test cases with corresponding representative test cases.

1.2 Brief Summary

The main works and activities of this thesis could be summarized as follow:

- Study of algorithms and principles about neural networks;
- Be familiar with the architecture of NVDLA, understand the role of each component, and select the typical parts to use in testing;

- Study of how to use Tmax and Z01X for generating test patterns and computing the coverage report for NVDLA;
- Generate test patterns for key computational parts of NVDLA;
- The generated test patterns are used as input to Z01X to derive the final coverage.

1.3 Thesis Structure

This thesis is divided into chapters which contain the following information:

- Chapter 2 It is mainly about the composition of Convolutional Neural Networks, the hardware structure of NVDLA and the main process and reasons for testing and fault tolerance.
- Chapter 3 It focuses on the tools used in this thesis and the methodology implemented in this thesis.
- Chapter 4 It focuses on the specific implementation of this thesis to generate test vectors for single stuck-at model in-field testing specifically for the CMAC portion of NVDLA.
- Chapter 5 The main focus is on the results produced using the methods discussed in this thesis and comparing them with the results of NVDLA's own tests.
- Chapter 6 It is mainly about the applicable scenarios using the methods discussed in this thesis, and the limitations.

Chapter 2

Background

Hardware reliability occupies a crucial position in electronic engineering. Hardware, such as chips as core components of various devices and systems, must maintain performance and stability over long periods of operation. This involves quality control at the manufacturing stage, including component selection, process control and structural testing to minimize the risk of manufacturing defects. During the life of a device, reliability testing helps monitor performance and diagnose potential problems, ensuring stable operation in a variety of environmental conditions.

The rise of deep neural networks has led to the rapid development of artificial intelligence technology. These networks play a key role in natural language processing, computer vision, and automated decision making. However, the reliability of deep neural networks is equally crucial, especially in critical applications such as medical diagnostics, self-driving cars and intelligent assistants. Wrong decisions can have a significant impact on users and society.

In this context, AI acceleration hardware has emerged to improve the computational performance and energy efficiency of deep neural networks. These special hardware are widely used in embedded systems, mobile devices, and data centers to improve the efficiency of deep learning applications. However, the reliability of AI acceleration chips is equally important. They must be able to operate stably in a variety of environments to ensure deep neural network accuracy and system reliability.

2.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a sophisticated deep learning algorithm comprising neurons equipped with adaptable weights and biases. Every neuron carries out a mathematical operation known as a dot product between its input and the weights, which is subsequently subjected to a non-linear activation function.

CNN architecture stands out due to its ability to autonomously acquire features from unprocessed image data and its inherent resilience to various types of inputs. CNNs exhibit outstanding performance in tasks such as image classification, natural language processing, recommender systems, and more. Moreover, CNNs possess a remarkable capability to generate an output expressed as a single differentiable score, which emerges directly from the original input image.
Convolutional neural networks have the following layers:

- Convolution Layer
- Activation Layer
- Pooling Layer
- Fully Connected Layer

2.1.1 Convolution Layer

The Convolution Layer serves the purpose of information extraction from the input image. It achieves this through an element-wise multiplication operation performed between the kernel weights and the input feature maps. This operation involves sliding the kernel across the input, examining each location, and shifting one neuron at a time (using a stride of 1). The partial results of these convolutions are then accumulated to form the respective output feature maps. To simplify the notation, Let us denote the input image as I(i,j), where we consider each pixel as a scalar. The filter is represented as a kernel K(n,m), and the resulting convolved output is expressed as h(i,j). Equation 2.1 and figure 2.1 represent a convolution operation.

$$h(i,j) = \sum_{m=0}^{m} \sum_{n=0}^{n} I(i-m, i-n)K(m,n) \tag{2.1}$$

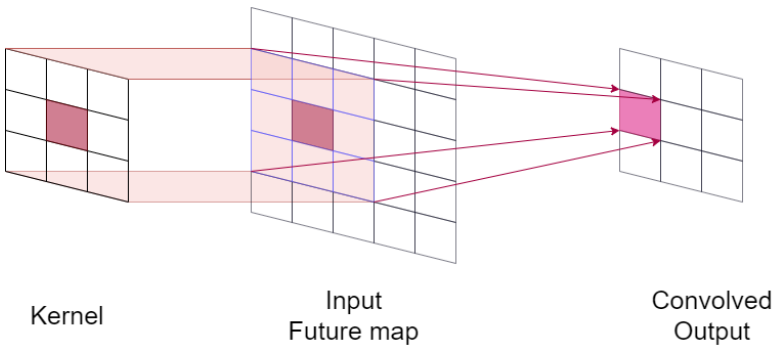


Figure 2.1: 2D Direct Convolution

In the kernel process, all the parameters remain constant during the sliding operations. This property enables the sharing of kernel weights when calculating the weighted sums for different hidden nodes. Beyond the convolutional layer, there are additional computational blocks involved in constructing a complete CNN, and we will discuss these in the following sections.

2.1.2 Activation Layer

Activation Layers play a pivotal role in introducing non-linearity into the system, allowing the network to grasp intricate relationships within the feature maps. The choice of an activation function significantly impacts the model's output, prediction accuracy, and computational efficiency during the training process. These functionalities are achieved through dedicated hardware logic and Lookup Tables in hardware implementations.

Various activation functions are selected based on the specific problem statement. Some common choices include:

- **ReLu**
  The most popular activation function for CNN's are the Rectified Linear Unit[8]. Since they result in sparse activation reducing the network parameters. They are defined as.
  $$f(x) = max(0, x) \tag{2.2}$$
  This activation function produces an output 'x' when 'x' is positive and 0 otherwise. It offers several advantages, such as avoiding issues with vanishing gradients and being computationally more efficient. However, it is not without its drawbacks. One common problem associated with this activation function is the issue of "dying" nodes. Occasionally, ReLU nodes get pushed into regions of inactivity, where they consistently output zero for all inputs. This leads to the creation of "dead neurons" that no longer contribute to the network's learning process.

- **PReLu**
  To address the issue mentioned earlier, a Parametric ReLU (PReLU) is employed [9]. Unlike standard ReLU, which sets the outputs to 0 when the inputs are less than or equal to zero, PReLU maintains a small linear trainable parameter a. This parameter a learns alongside other neural network parameters, effectively mitigating the problem of dead neurons. The comparison between ReLU and PReLU is illustrated in the equation 2.3 and figure 2.2.
  $$f(x) = \begin{cases} x, & x > 0 \\ ax, & x \leq 0 \end{cases} \tag{2.3}$$

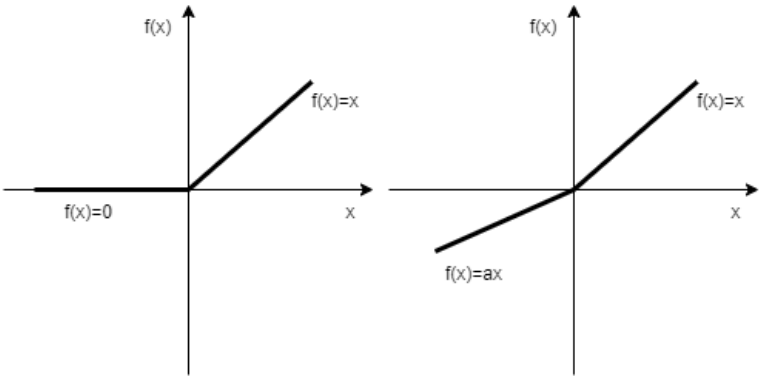


Figure 2.2: ReLu and PReLu

- **Hyperbolic Tangent**
  Hyperbolic Tangent (tanh) is an activation function commonly used in neural networks. It is an S-shaped curve that maps input values to an output range between -1 and 1. Tanh is popular in neural network architectures because it introduces non-linearity into the model. It is particularly useful when dealing with data that has zero-centered features. Tanh squashes the input data to fall within the -1 to 1 range, making it easier for the network to learn and capture complex patterns and relationships in the data.The equation 2.4 and figure 2.3 following activation is presented.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.4}$$

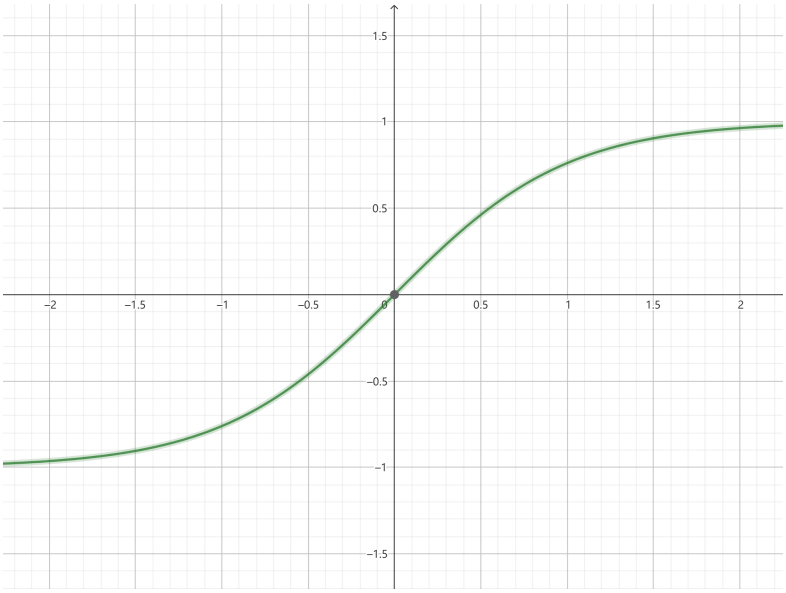


Figure 2.3: Hyperbolic Tangent

- **Sigmoid**
  Sigmoid is an activation function commonly used in neural networks. It is an S-shaped curve that maps input values to an output range between 0 and 1. Sigmoid is useful for binary classification tasks, where the network's output represents the probability of a certain class. It squashes the input data to fall within this 0 to 1 range, making it suitable for modeling probabilities. However, it has some limitations, such as vanishing gradients.The equation 2.5 and figure 2.4 following activation is presented.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

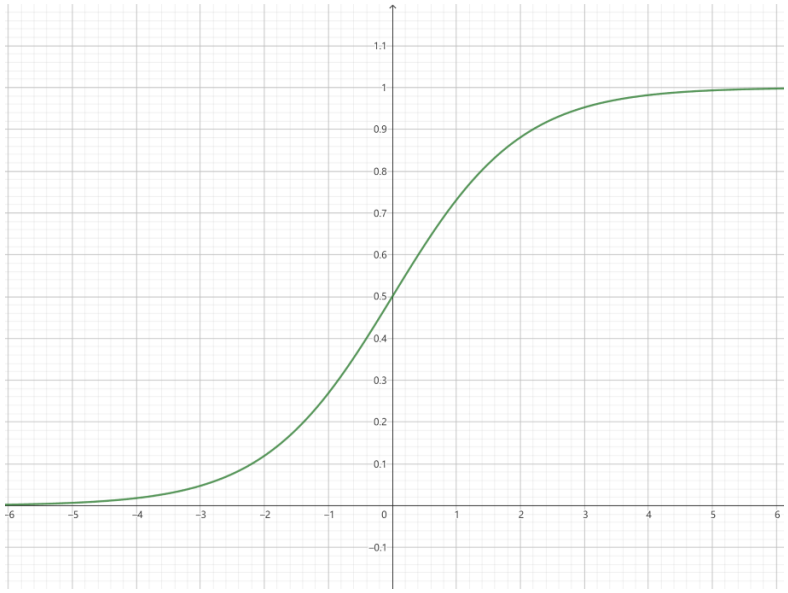


Figure 2.4: Sigmoid

2.1.3 Pooling Layer

Pooling Layer is a component commonly used in neural networks. Its primary purpose is to reduce the spatial dimensions of the input data, which is typically an image or feature map. It operates by down-sampling the data in order to decrease the number of parameters and computational complexity in the network. There are different types of pooling layers

- **Max pooling**
  Max pooling divides the input data into non-overlapping regions, often referred to as "pools" or "windows." For each of these regions, it selects the maximum value. This process effectively downsamples the data, reducing its size by keeping only the highest activation value in each pool. An example is shown in the figure 2.5.

- **Min pooling**
  in this operation the minimum value within the pooling window is selected.

- **average pooling**
  This function returns the average value within the receptive field

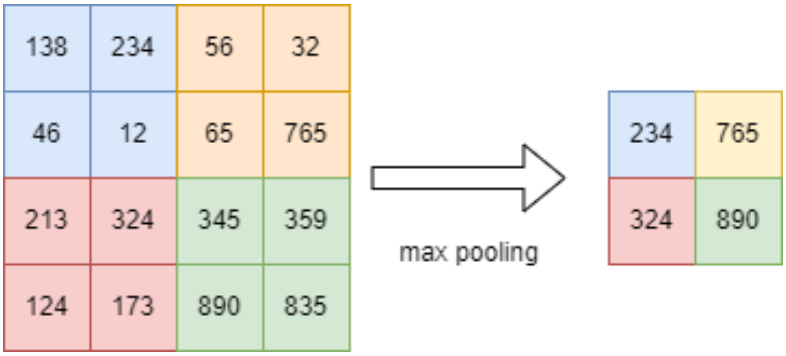


Figure 2.5: maxpooling

Pooling layers help the network become more robust to variations in the input data and reduce overfitting by focusing on the most significant features. They are often placed between convolutional layers in convolutional neural networks (CNNs) and can be used to progressively reduce the spatial dimensions of the data as it flows through the network.

2.1.4 Fully Connected Layer

A Fully Connected (FC) layer can be likened to a multi-layer perceptron model, where each neuron is intricately connected to the neurons in the previous layers. This layer takes as input a 1D vector of numbers, which is a flattened representation of the 3D volumes obtained from the previous convolutional layers. The output of an FC layer is essentially a list of probabilities associated with various class scores. This specific FC layer essentially functions as a classifier.

In this operation, the FC layer examines the high-level feature representations derived from the previous layers (following convolution and activation). It assigns weights to these features to predict their correlation with specific classes. The class score with the highest probability ultimately determines the classifier's decision. For example, if the model is tasked with predicting whether an image depicts a bird, it would assign significant values to the activation maps corresponding to high-level features like wings or beaks. In this way, the classifier establishes a connection between the extracted high-level feature maps and definitive class scores.

2.2 NVDLA

The NVIDIA Deep Learning Accelerator (NVDLA) is a free and open architecture that promotes a standard way to design deep learning inference accelerators. With its modular architecture, NVDLA is scalable, highly configurable, and designed to

simplify integration and portability. The hardware supports a wide range of IoT devices.[10]

2.2.1 Interface

NVDLA is a fixed function accelerator engine which is targeted towards deep learning. The following diagram shows the NVDLA connections to an SoC 2.6

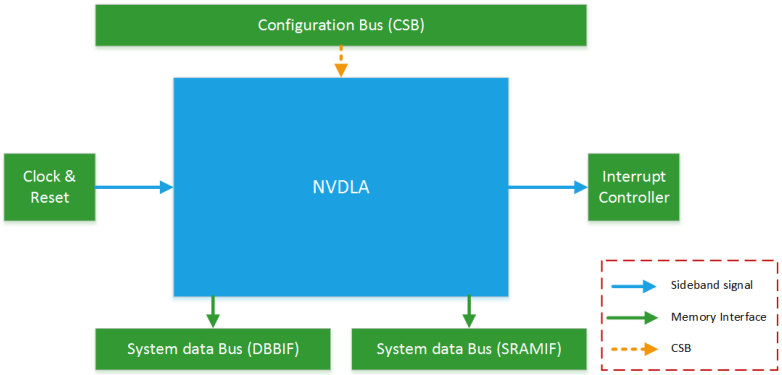


Figure 2.6: NVDLA Interface

2.2.2 Hardware Unit

NVDLA is mainly composed of various types of computing units, the most important of which are the following hardware components.

Convolution Pipeline

The Convolution Pipeline, as one of the pipelines within the NVDLA core logic, serves to accelerate the convolution algorithm. It supports comprehensive programmable parameters to accommodate various convolution sizes. Advanced features such as Winograd and multi-batch processing are implemented within the Convolution Pipeline to enhance performance and increase MAC efficiency.

Below is the diagram of the convolution pipeline.

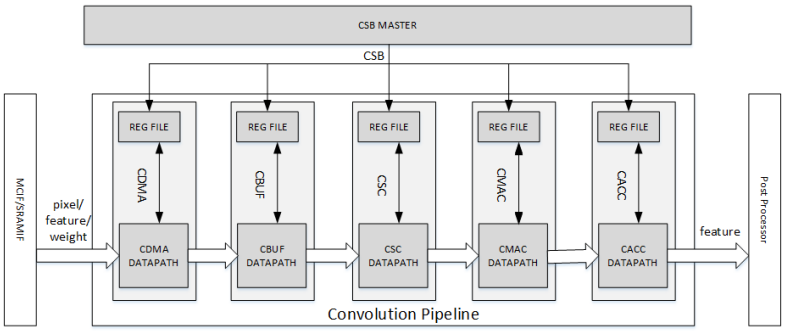


Figure 2.7: NVDLA Convolution pipeline

- **Bridge DMA**
  The NVDLA architecture optimizes its processing capabilities by integrating on-chip SRAM due to the limitations in external DRAM bandwidth and latency. However, the efficient utilization of the MAC (Multiply-Accumulate) arrays is hindered by these external memory constraints. To address this, NVDLA incorporates a secondary memory interface with on-chip SRAM.

  For seamless data movement between external DRAM and on-chip SRAM, the Bridge DMA (BDMA) module is introduced. BDMA facilitates two independent paths: one for copying data from external DRAM to internal SRAM and another for copying data from internal SRAM to external DRAM. However, these directions cannot operate simultaneously. Additionally, BDMA is versatile, supporting data movement between external DRAM to external DRAM and internal SRAM to internal SRAM.

  BDMA is equipped with two DMA interfaces—one connecting to external DRAM and the other to internal SRAM. Both interfaces support read and write requests with a data width of 512 bits and a maximum burst length of 4.

  To efficiently handle three-dimensional data structures, BDMA introduces line repeat functionality, allowing the fetching of multiple lines with address jumps between them, effectively reflecting a surface. Moreover, BDMA supports an additional layer of repeat, enabling the repeated fetching of multiple lines, which reflects multiple surfaces and, ultimately, represents a cube.

  Below is the diagram of the NVDLA BDMA.

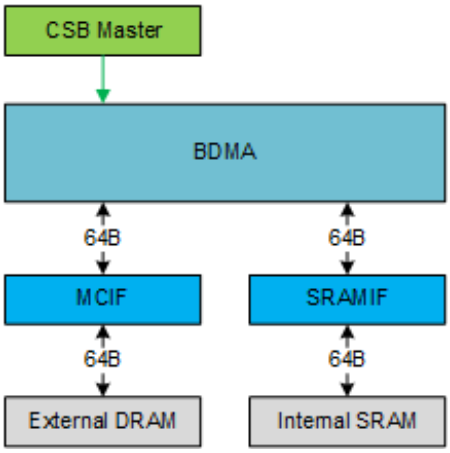


Figure 2.8: NVDLA BDMA

- **Convolution DMA**
  Convolution DMA (CDMA) serves as a pivotal stage within the convolution pipeline. Its primary function is to retrieve data from on-chip SRAM (Static Random-Access Memory) or off-chip DRAM (Dynamic Random-Access Memory) to facilitate the convolution operation. The retrieved data is then efficiently stored in a buffer known as the Convolution Buffer (CBUF) in the specific order required by the convolution engine. This process ensures that the data is prepared and organized appropriately for the subsequent stages of the convolution operation within the pipeline. Below is the diagram of the NVDLA CDMA.

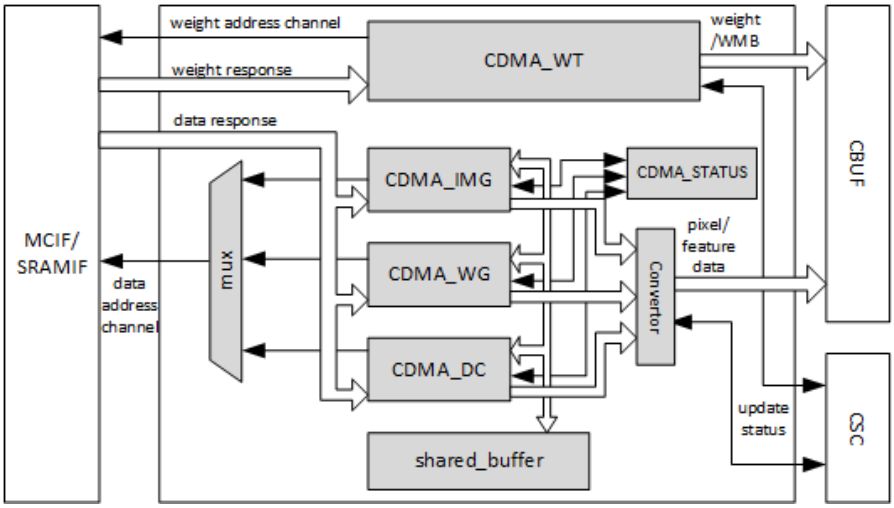


Figure 2.9: NVDLA CDMA

CDMA consists of three sub-modules: CDMA_DC, CDMA_WG, and CDMA_IMG responsible for fetching pixel data or feature data for convolution. While the procedures of these sub-modules are similar, they differ in how they organize the data into the CBUF RAM. Only one of the sub-modules is activated at any given time to fetch pixel/feature data.

- **Convolution Buffer**
  The Convolution Buffer (CBUF) is a stage in convolution pipeline. It contains a total of 512KB of SRAM. The SRAMs cache input pixel data, input feature data, weight data and WMB data from CDMA module, and are read by convolution sequence generator module. CBUF has two write ports and three read ports. CBUF contains of 16 32KB banks. Each bank consists of two 512-bit-wide, 256-entry two-port SRAMs. Below is the diagram of the NVDLA Convolution Buffer.

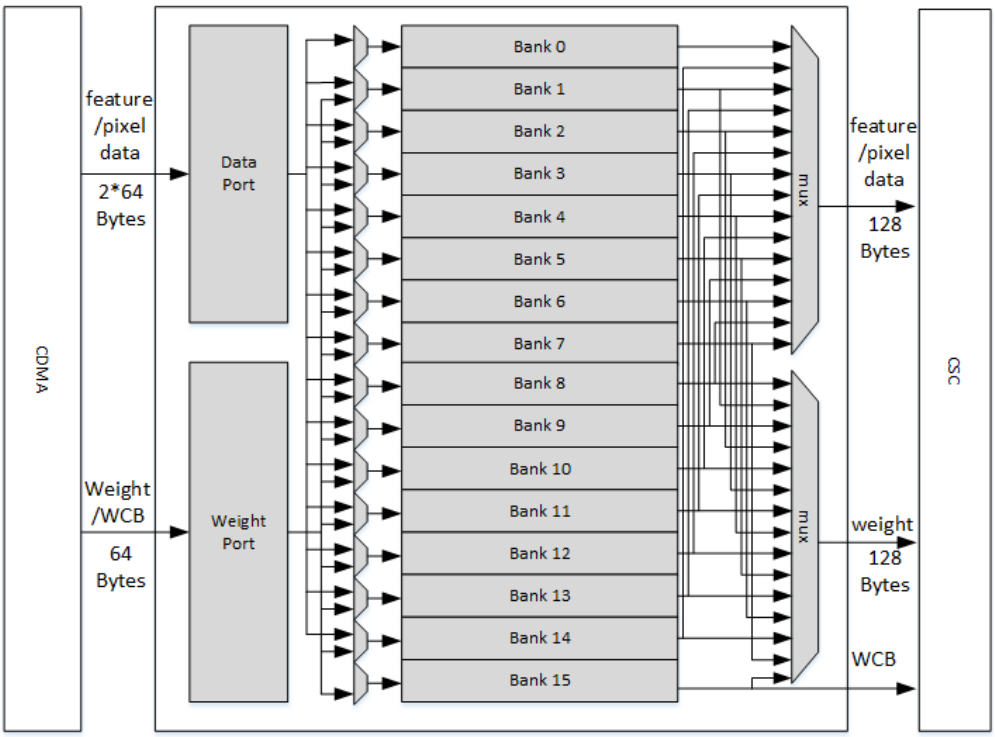


Figure 2.10: NVDLA CBUF

- **Convolution Sequence Controller**
  The Convolution Sequence Controller (CSC) is responsible for loading input feature data, pixel data, and weight data from CBUF and sending it to the Convolution MAC unit. It is the key module for computing and controlling

the convolution sequence in the Convolution Pipeline.The Convolution Sequence Controller (CSC) includes three sub-modules: CSC_SG, CSC_WL and CSC_DL.

CSC_SG is the convolution sequence generator, responsible for generating the sequence to control the convolution operation.

CSC_DL is the convolution data loader, which executes the feature/pixel loading sequence. It receives packages from the sequence generator, loads feature/pixel data from CBUF, and sends them to the Convolution MAC. Additionally, it manages the data buffer status and communicates with CDMA to keep the status up to date. In Winograd mode, it also performs PRA (pre-addition) to transform the input feature data.

CSC_WL for convolution weight loader, executes the weight loading sequence. It receives packages from the sequence generator, loads weights from CBUF, performs necessary decompression, and sends them to the convolution MAC. It manages the weight buffer status and communicates with CDMA_WT to keep the status up to date.

Below is the diagram of the NVDLA Convolution Sequence Controller.

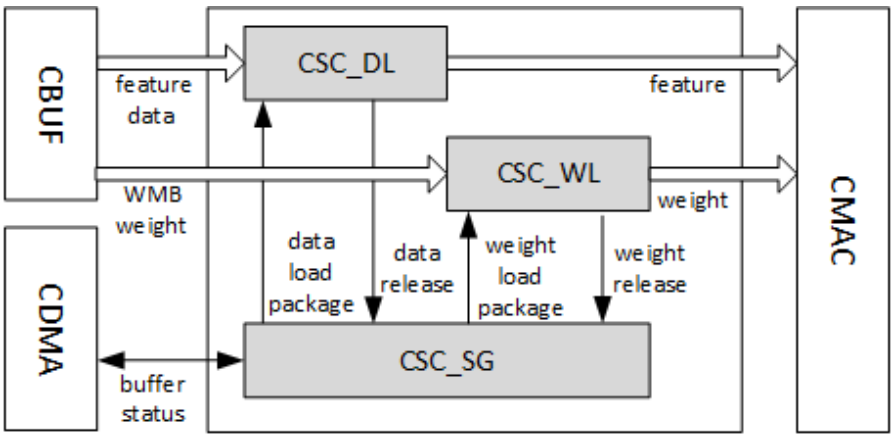


Figure 2.11: NVDLA CSC

- **Convolution MAC**
  The Convolution MAC (CMAC) module is one stage of the convolution pipeline dedicated to the convolution operation. It's the biggest part of the calculation.It receives input data and weights from the convolution sequence controller (CSC), executes multiplication and addition operations, and forwards the result to the convolution accumulator. In Winograd mode, the Convolution MAC conducts POA (post addition) on the output to convert the result back to the standard activation format.

CMAC consists of 16 identical sub-modules known as MAC cells. Each MAC cell comprises 64 16-bit multipliers for int16/fp16 data. Additionally, it contains 72 adders for int16/fp16, specifically designated for Winograd POA. Each multiplier and adder can be split into two calculation units for int8 format data. The throughput of int8 data is twice that of int16 data in any mode. The output result from the MAC cell is referred to as partial sum. The pipeline depth of CMAC is 7 cycles.

CMAC is divided into two parts, CMAC_A and CMAC_B. Each part has its own CSB interface and register file. However, they are treated as a single pipeline stage during operation. Below is the diagram of the NVDLA Convolution MAC.

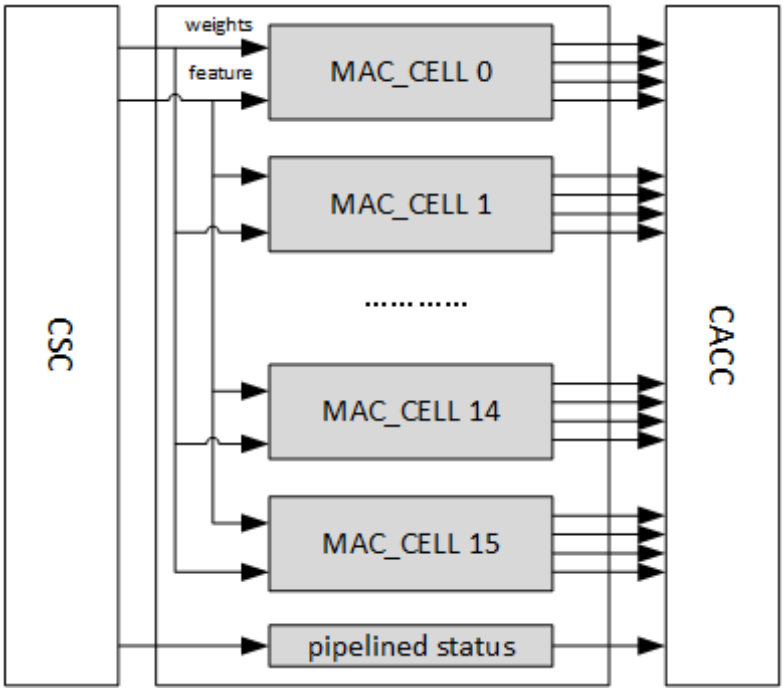


Figure 2.12: NVDLA CMAC

- **Convolution Accumulator**
  The Convolution Accumulator (CACC) is the stage following CMAC in the convolution pipeline. Its primary function is to accumulate partial sums generated by the Convolution MAC, and to round/saturate the result before transmitting it to the SDP (Software Defined Processor). Moreover, the large buffer within the convolution accumulator serves to smooth out the peak throughput of the convolution pipeline. The components in CACC comprise assembly SRAM group, delivery SRAM group, adder array, truncating array,

valid-credit controller, and a checker.
Below is the diagram of the NVDLA Convolution Accumulator.

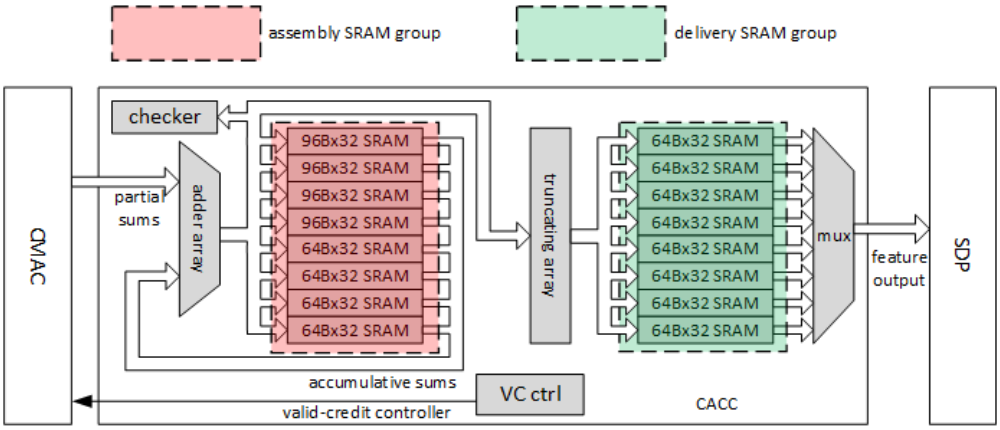


Figure 2.13: NVDLA CACC

- **Single Point Data Processor**
  The Single Point Data Processor (SDP) performs post-processing operations at the single data element level. The Single Point Data processing is designed to accomplish the following operations

  – Bias Addition
  For a convolutional layer, there're always a bias addition after convolution. In NVDLA,implement bias addition in SDP.
  Below is the mathematical formula for bias addition.

$$y = x + bias \tag{2.6}$$

  – Non-Linear Function
  The Non-Linear function hardware in SDP is used to accomplish activation layer operations. Based on current network analysis, there are three activation functions are commonly used

    * ReLU
    * PReLU
    * Sigmoid
    * Hyperbolic tangent

  These activation functions were introduced in 2.1.2.

  – Batch Normalization
  Batch normalization a is widely used layer. It can be descripted by formula

below:

$$x' = (x - \mu)/\theta \tag{2.7}$$

μ is the mean and θ is the standard variance and χ is element of feature data cubes

In addition to the above commonly used mathematical formulas, SDP also supports the following mathematical formulas

- Format conversion
- Comparison

- Planar Data Processor
  The Planar Data Processor (PDP) carries out operations across the plane defined by its width and height dimensions. The PDP is engineered to perform pooling layers, supporting methods such as max, min, and mean pooling. A group of adjacent input elements within a plane undergoes processing through a non-linear function to compute a single output element.

- Cross Channel Data Processor
  The Cross Channel Data Processor (CDP) performs operations in the direction of channels.Channel processing is implemented to handle local response normalization (LRN) layers. LRN conducts a form of lateral inhibition by normalizing across a local input region along the channel direction.

- RUBIK
  RUBIK module is similar to BDMA. It transforms data mapping format without any data calculation. Below is the diagram of the NVDLA Convolution Accumulator.

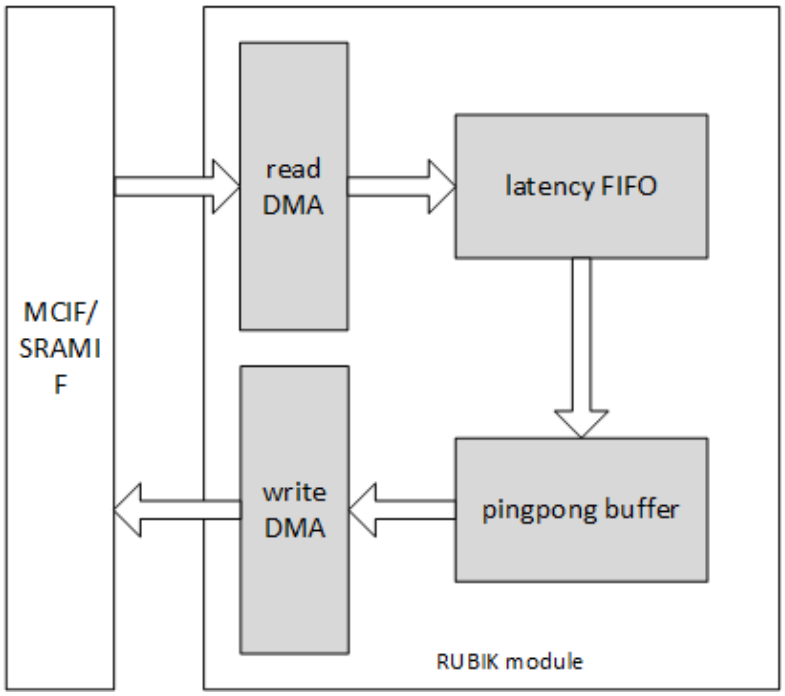


Figure 2.14: NVDLA RUBIK

- MCIF
  MCIF is used to arbitrate requests from several internal sub modules and convert to AXI protocol to connect to external DRAM. MCIF will support both a read and write channels, but some NVDLA sub-module will only have read requirement. Below is the diagram of the NVDLA MCIF.

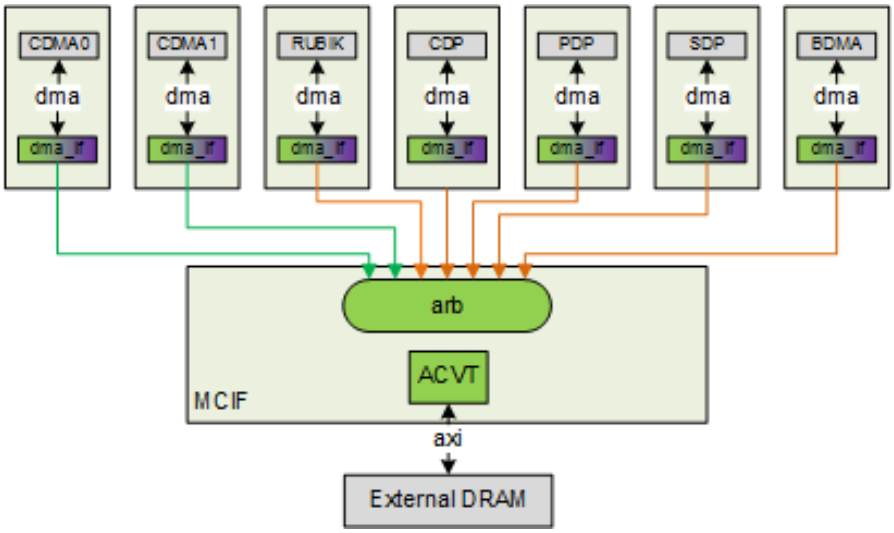


Figure 2.15: NVDLA MCIF

- SRAMIF
  The SRAMIF moduile is used to connect several internal sub-modules to on-chip SRAM. SRAMIF will support both read and write channels, but some NVDLA sub-modules will only have a read requirement. Below is the diagram of the NVDLA SRAMIF.

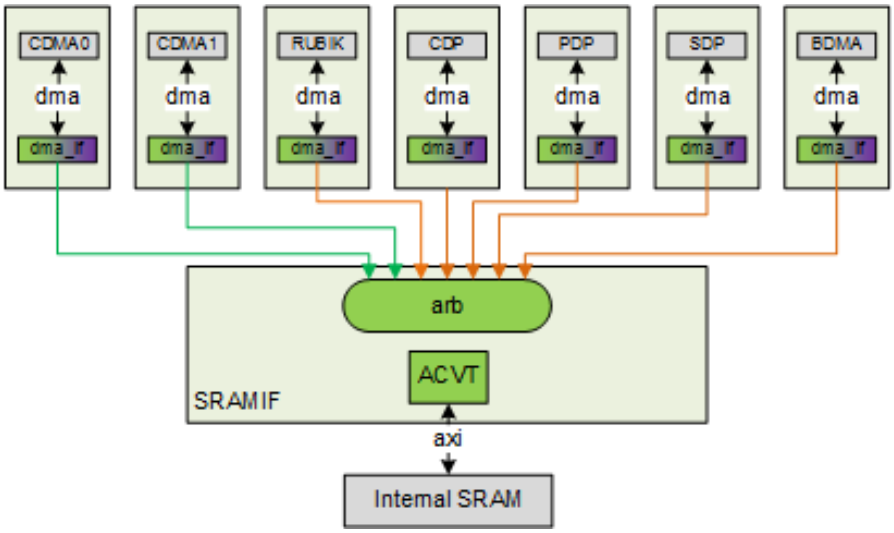


Figure 2.16: NVDLA SRAMIF

2.3 Testing And Fault Tolerance

2.3.1 Generate test vectors

In the current era of booming Artificial Intelligence (AI) technology, research on chip testing has become crucial. With the wide application of AI algorithms, the demand for high-performance and high-efficiency chips continues to grow. This demand drives more in-depth exploration and innovation in chip testing.

First, AI algorithms usually require large-scale datasets for training and optimization. These datasets may contain thousands of samples, making efficient processing and fast computation of the data a critical need. Chips, as the core component of these computations, need to be characterized by high performance and efficiency. Therefore, chips must be tested to ensure that they can operate stably and have sufficient computational and storage capacity to meet the needs of AI algorithms.

Second, the optimization and iteration of AI algorithms usually require a large number of tests and verifications. In this process, the requirement for chip testing is even more urgent, because any potential hardware error or defect may lead to failure or inaccuracy of algorithm training. Therefore, effective chip testing strategies and techniques can help detect potential problems early and ensure the stability and reliability of the chip.

Overall, AI technology poses new challenges and demands for chip testing. Effective testing strategies and techniques will provide important support for the optimization and application of AI algorithms and promote the continuous development and application of AI technology. Therefore, the research and innovation of chip testing is of great significance and will lay a solid foundation for the further development of artificial intelligence technology.

In order to generate test vectors for the chip in a better and more efficient way, the commonly used ways are ATPG, and scan chain, they all have certain advantages and disadvantages, for ATPG, the advantages of ATPG are

- High coverage: ATPG can generate comprehensive test patterns to effectively detect all kinds of faults and defects in the chip, improving the comprehensiveness and accuracy of the test.
- Automation: ATPG technology automates the generation of test patterns, reducing the need for human intervention. This automation feature can significantly reduce the labour and time costs of testing.
- Flexibility: ATPG technology can generate different types of test patterns based on different test requirements and constraints[2]. This flexibility makes ATPG technology applicable to a wide range of different types of chip design and test needs.

The disadvantages of ATPG are

- High computational resource requirements: ATPG techniques usually require a large amount of computational resources to generate test patterns, especially for complex chip designs, which may require long computational time to generate test patterns.
- Excessive number of test vectors: The number of test patterns generated by ATPG is usually huge, which may lead to long test time and high test cost. In practice, the generated test patterns may need to be filtered and optimized to reduce the cost and risk of testing.
- Difficult to cover all faults: Although ATPG can generate a large number of test patterns, it is not guaranteed to cover all possible faults and defects. Some specific types of faults may require additional manually designed test patterns to detect.

For the scan chain,the advantages of scan chain are

- Simple to implement: Scan Chain is a relatively simple DFT technology that is easy to implement and deploy. It makes the application of test patterns straightforward by connecting the memory elements in a chip into a serial link.
- High Test Coverage: Scan Chain can cover all testable nodes in the chip, including logic gates, registers, etc., thus improving test comprehensiveness and accuracy.
- Widely used: Scan Chain has become a common DFT technique in many chip designs and has been widely used and verified.

The disadvantages of scan chain are

- Higher test overhead: Scan Chain technology requires some additional memory components and control logic in the chip, which may increase the area and power consumption of the chip, resulting in a certain test overhead.
- Limited Design Flexibility: Scan Chain wiring and logic needs to be considered during the design process, which may limit the flexibility and optimization of the chip design.
- Not Applicable to All Designs: Due to test overhead and design constraints, Scan Chain technology may not be applicable to some specific types of chip designs, especially those with high area and power requirements.

With the rapid development of AI technology, its application in highly real-time scenarios such as automated control of vehicles is increasing. In these safety-critical application scenarios, the stability and reliability of the chip is critical to safety. Any system failure due to chip malfunction or error may bring serious consequences. In order to ensure the safety and reliability of the system, In-field testing becomes especially important in these scenarios[11].

In-field testing refers to the process of testing and verifying the chip in real-world scenarios after the chip has been manufactured. Unlike traditional laboratory testing, In-field testing is conducted directly in actual application scenarios, which can more realistically simulate the performance and stability of the chip under actual working conditions. In addition, In-field testing can help optimize system performance and efficiency. By testing and debugging the chip in real-world application scenarios, performance bottlenecks and optimization space can be identified and resolved on time, improving system performance and responsiveness.

2.3.2 Fault simulation

Fault simulation is the process of considering all the potential faults in a design and determining how many of them can be detected.

Effective failure simulation will cover the following three main phases of the chip lifecycle:

- Chip development phase:
  In this phase, fault simulation should be used to prove and document the robustness of the design and verification flow. In other words, failure simulation ensures that the implementation tools and flows do not introduce design defects (systematic failures) and that the verification tools and flows accurately report all design defects and are able to fix the vulnerabilities. So, failure simulation also ensures that the design verification methodology is robust enough to provide high confidence in the zero-defect design goal.

- Chip manufacturing phase:
  In this phase, fault simulation can help reduce the defective rate due to random faults (DPPM) by monitoring the functional test vectors of the DFT. Real-world operation phase: Fault simulation proves and documents whether the safety mechanism is functioning properly. Safety mechanisms are triggered in the event of a failure (and should only be triggered in the event of a failure), and they are effective in getting the design into a safe state.

-

Failure target coverage (i.e., diagnostic coverage) for failure simulation is closely related to the degree of safety criticality. The more safety-critical an application is,

the higher the target coverage requirements for fault simulation will be.

Achieving high diagnostic coverage while determining whether all faults in the design can be detected is also a very difficult task for developers. The process requires testing the chip design in various scenarios using a large number of testbenches and stimulating test incentives. However, it is often impossible to determine whether these test sequences are sufficient because there is no clear methodology to evaluate the value of each testbed and stimulus for error coverage.

Moreover, as chip designs become more complex, simulation runs take longer and longer to run. For some SoC designs for safety-critical applications, in order to measure diagnostic coverage, up to millions of faults need to be simulated to ensure functional safety compliance[12].

Fault simulation tasks are computationally challenging. From simple logic circuits to complex integrated circuits (IC) with millions of components, fault simulation requires significant computational resources and advanced methodologies.

At its core, fault simulation is about evaluating the behavior of a system under various fault conditions. This involves injecting faults (e.g., stuck-at faults or bridging faults) into circuit models and analyzing how these faults propagate through the system. However, as circuits become more complex and integrated, the computational burden grows exponentially.

A major challenge stems from the sheer size of modern IC designs. With millions of transistors on a single chip, modeling every possible failure scenario becomes a daunting task. Traditional fault simulation algorithms often struggle to cope with this complexity, resulting in extremely long simulation times.

From a computational point of view, the failure simulation task is a continuous challenge. As electronic systems continue to evolve in complexity and size, addressing these challenges will require continued innovation in algorithms, models, and computational infrastructure. However, overcoming these challenges is critical to ensuring the reliability and resilience of modern electronic systems in the face of faults and failures.

Chapter 3

Method

In-field testing is especially important to solve the stability testing of AI chips in real-time systems. In-field testing needs to be fast and high coverage, fast means that the test vectors need to be as few as possible, but the coverage of the test vectors is as high as possible, and at the same time, because of the problems of the application scenarios, in order to control the power consumption and the cost of the scan chain technology is not usually used. Combining the above reasons, this paper proposes a new method to generate test vectors in single stuck-at model for In-field testing, taking NVDLA as an example, and generating test vectors for the largest computational part of it: CMAC. Firstly, we will introduce the tools used, and secondly, we will introduce the methods used.

3.1 Introduction To TetraMAX And Z01X

3.1.1 TetraMAX

TetraMAX is an advanced Automatic Test Pattern Generation (ATPG) tool developed by Synopsys, designed to help chip designers efficiently generate high-quality test patterns for detecting and diagnosing faults and defects in integrated circuits. The following is an introduction to TetraMAX:

Using advanced algorithms and techniques, TetraMAX is able to automatically generate highly optimized test patterns that cover a wide range of fault types in a chip, including logic faults, interconnect faults and memory faults. It supports a variety of test pattern generation techniques, including scan-chain test, boundary-scan test and random test, etc., which allows you to choose the appropriate test pattern generation strategy according to different design requirements and test objectives.

TETRA MAX can be used to constrain the inputs to the DUT by setting constrain for the DUT to better generate the desired test vectors for the combinational logic circuits.

3.1.2 Z01X

The Z01X is a functional fault simulator developed by synopsys to inject faults into a chip and simulate their effects, emulate faults not covered by the ATPG solution, and ensure that current functional verification environments are able to detect these faults, which can complement the ATPG fault scope coverage and thus improve test coverage.

The generated test vectors can be placed into Z01X and fault simulation can be performed to obtain realistic coverage as well as coverage so that the test vectors can be optimized.

3.2 Proposed Method For Generating Test Patterns

By analyzing the netlist of NVDLA, the computational part with the largest percentage of faults is the CMAC unit, while CMAC has a lot of repetitive structures, but since it is a multi-stage pipeline including multi-precision support design, ATPG does not have a good way to generate the corresponding test vectors.

In this paper, we propose a method, by analyzing the structure of NVDLA, we find that the structure of CMAC is composed of 2 large parts CMAC_A and CMAC_B, each part has 16 MAC_CELLs, and there are 64 int16/fp16 multipliers in each MAC_CELL, as well as a CSA TREE, for the part of MAC_CELL, we can for For the MAC_CELL section, we can generate test vectors for a single multiplier in it, and then use it in all of them, and at the same time generate test vectors separately for the CSA TREE. Due to the repetitive and multi-stage pipelined design of CMACs, we can increase the coverage rate and reduce the size of the test vectors by generating the test vectors in such a separate way.

Chapter 4

Implementation

4.1 Data

NVDLA reads the feature data cube and the weight data cube from a file as a basis for convolution. Below we describe what the data cube used by NVDLA looks like and how it is stored in memory and in a file.

4.1.1 Feature Data Cube

For NVDLA, the feature data it uses is a kind of feature Data Cube, Take an 8*8 feature data cube as an example as shown below

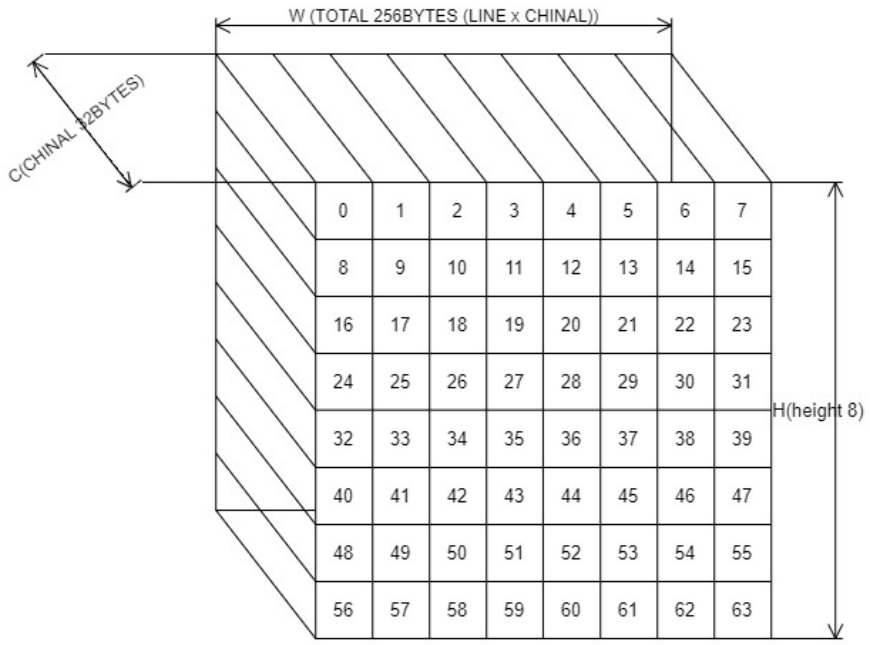


Figure 4.1: NVDLA Feature Data Cube

All elements of feature data for one layer are organized as a 3D data cube. Three dimensions are width (W), height (H) and channel size (C). The memory mapping rules are:

- Adding data into end of channel if the original data is not 32byte aligned in C direction.
- The attached data can be any value except NaN when it's fp16.
- Split the data cube into 1x1x32byte small atom cubes.
- Reordering atom cubes in by progressively scanning the data cube. Scanning order: W (line) -> H (height) -> C (channel).
- Map all atom cubes into memory by scanning sequence.
- All atom cubes in the same line are mapped compactly.
- Atom cube mapping at line boundary and/or surface boundary can be either adjacently or incompactly. But they are always 32-byte aligned.
- In conclusion, mapping in memory follows pitch linear format. The order is C' (32byte) -> W -> H -> C (surfaces). Here C' changes fastest and C changes slowest.

The Feature data cube is stored in memory in two forms, one packed and one unpacked.

Here is a comparison of packaged and unpacked data stored in memory.For the unpacked part, blue is the part with actual values

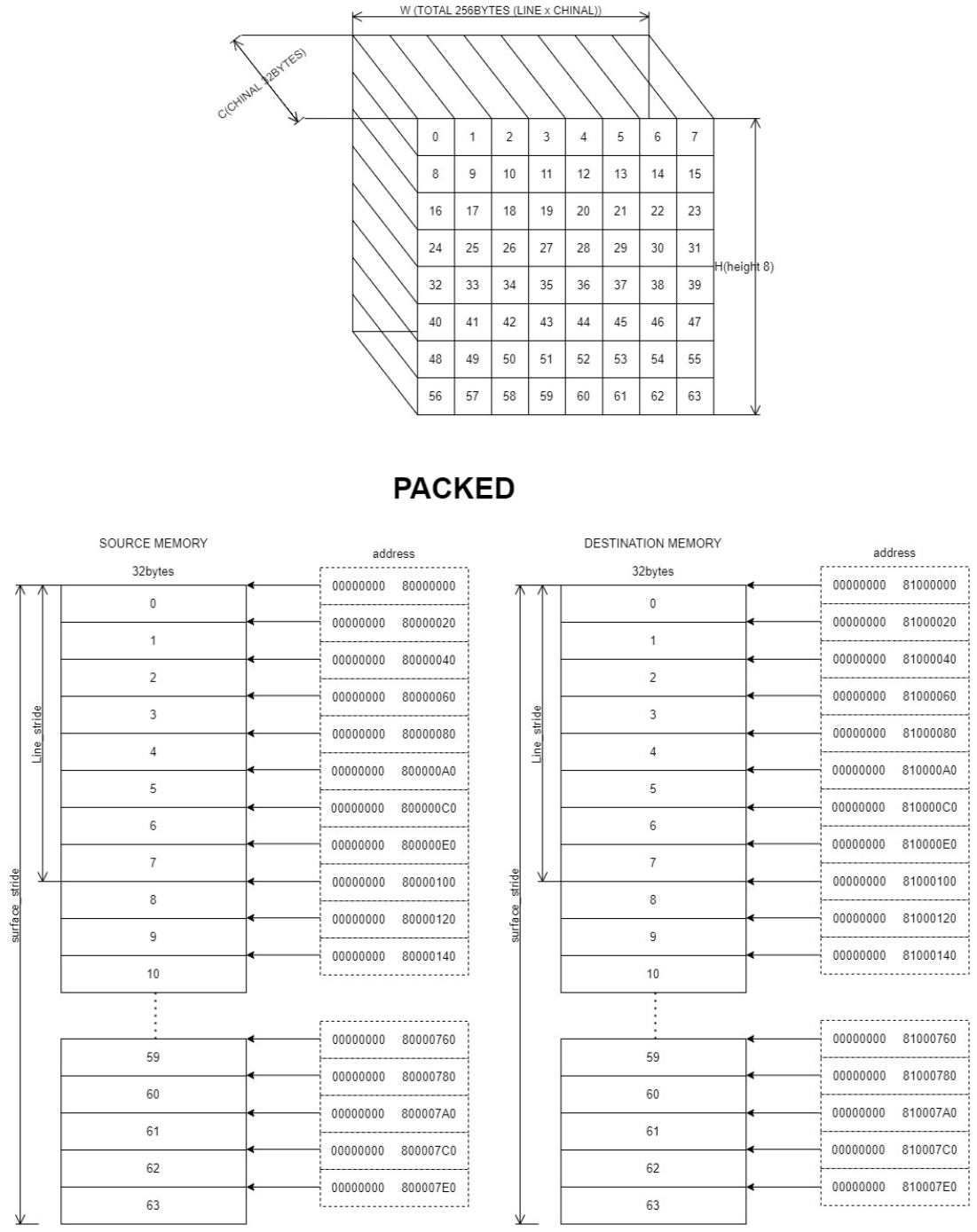
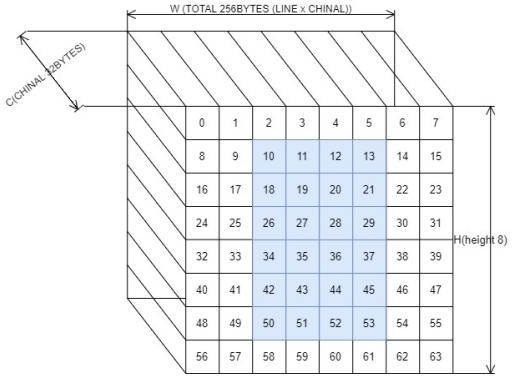


Figure 4.2: NVDLA Packed Cube



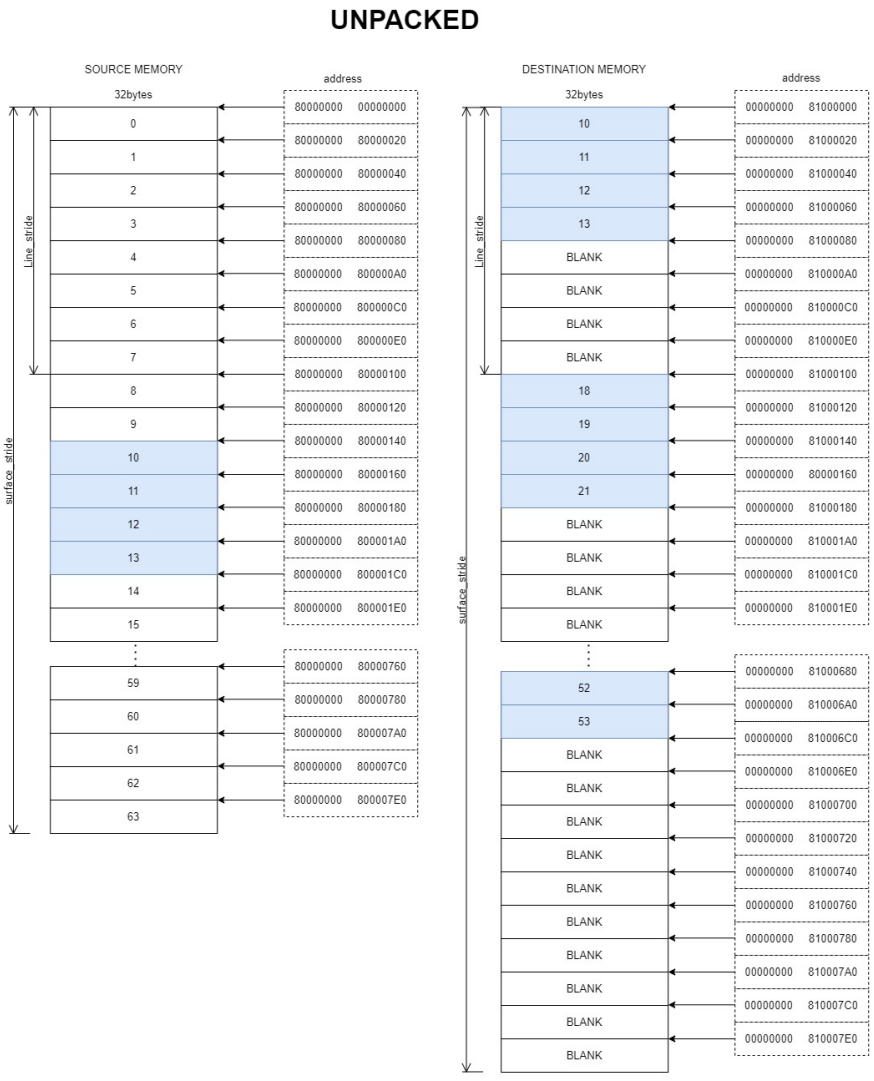


Figure 4.3: NVDLA Unpacked Cube

When C exceeds 32bytes, NVDLA first reads the data from 0-31*H*W(Also called 1 surface), and then reads the data from 32-63*H*W, which are stored in memory in the same format as the packed and unpacked ones described above.

The storage in the file is in the form of 32bytes per line, starting with 0 for the first surface, followed by the second surface

4.1.2 Weight Data Cube

The basic weights of direct convolution are the most basic and common weight format. The weight mapping rule for direct convolution is:

- Distribute the kernels into groups. For int16 and fp16 weight, one group has 16 kernels. For int8, one group has 32 kernels. Last group can have fewer kernels.

- Divide each kernel to 1x1x64-element small cubes. For int16/fp16 the small cube is 128 bytes each; and for int8 the small cube is 64 bytes each. Do not append 0 if channel size is not divisible by 128/64.

- After division, all weights are stored in 1x1xC' small cubes, where C' is no more than 128 bytes.

- Scan the 1x1xC' small cubes in a group with C'->K->W->H->C sequence. Here C' changes fastest and C changes slowest. And map them compactly as scanning sequence.

- Map the weight groups compactly. Do not append any 0s between group boundaries.

- Append 0s at end of all mapped weight for 128-byte alignment.

Here is a schematic of how the weight data cube is mapped in memory

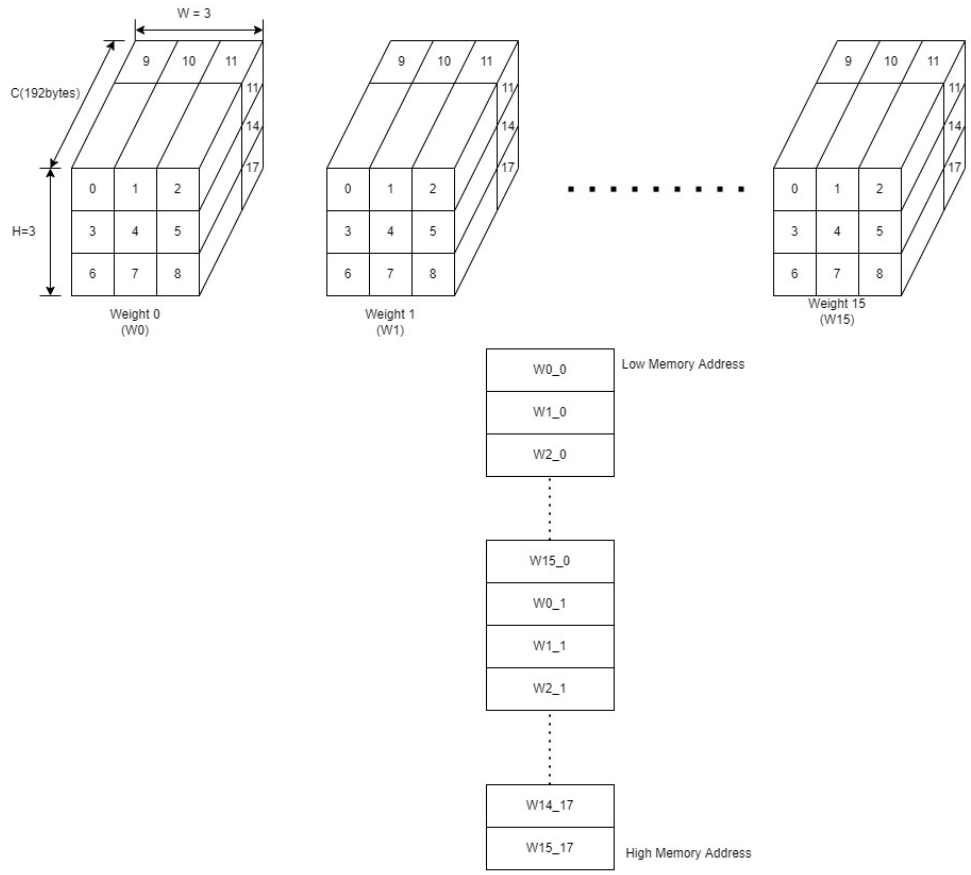


Figure 4.4: NVDLA Weight Cube

Unlike the feature data cube, the weight data cube is stored in the file as 32bytes per line, first storing the C-direction data in kernal from 0 to 15 in order, and then storing them in a W-first and then H-first manner.

4.2 NVDLA Configuration

In the official website of NVDLA, there is no specific method to configure the registers, while the address space layout in the official website is also inaccurate, but in the SW/kmd/include/opendla/initial.h file provided by it, there are the names of its corresponding registers and the corresponding addresses, combined with the description in the document of NVDLA, collated the following main use of the registers and their configurations. This paper focuses on the configuration of direct convolution, using three different data types, fp16, int16, and int8, for convolution operations.

The following are the main registers that need to be configured

Table 4.1: NVDLA Address Space Layout

Name	Address	Description
NVDLA_CDMA.D_MISC_CFG_0	0xffff1405	Control the type of input FEATURE and WEIGHT data, the type of operations on the data, and the data format.
NVDLA_CDMA.D_DATAIN_SIZE_0	0xffff1407	Control the number of H and W
NVDLA_CDMA.D_DATAIN_SIZE_1	0xffff1408	Control the number of C
NVDLA_CDMA.D_LINE_STRIDE_0	0xffff1410	Control the number of data in each line
NVDLA_CDMA.D_SURF_STRIDE_0	0xffff1412	Control the number of data in a surface
NVDLA_CDMA.D_ENTRY_PER_SLICE_0	0xffff1418	Control the size of each slice(W*1*C)
NVDLA_CDMA.D_WEIGHT_SIZE_0	0xffff141b	Control the size of weight(one kernal)
NVDLA_CDMA.D_WEIGHT_SIZE_1	0xffff141c	Control the number of kernal
NVDLA_CDMA.D_WEIGHT_BYTES_0	0xffff1420	Control the size of all weight(all kernal)
NVDLA_CDMA.D_BANK_0	0xffff142f	Control the number of banks of weight and data
NVDLA_CSC.D_MISC_CFG_0	0xffff1803	Control convolution methods and data types.
NVDLA_CSC.D_DATAIN_SIZE_EXT_0	0xffff1805	Control the number of H and W
NVDLA_CSC.D_DATAIN_SIZE_EXT_1	0xffff1806	Control the number of C
NVDLA_CSC.D_ENTRY_PER_SLICE_0	0xffff1809	Control the size of each slice(W*1*C)

NVDLA_CSC.D_WEIGHT_SIZE_EXT_0	0xffff180b	Control the number of H and W
NVDLA_CSC.D_WEIGHT_SIZE_EXT_1	0xffff180c	Control the number of C
NVDLA_CSC.D_DATAOUT_SIZE_0	0xffff180f	Control the H and W of the output data
NVDLA_CSC.D_DATAOUT_SIZE_1	0xffff1810	Control the C of the output data
NVDLA_CSC.D_BANK_0	0xffff1817	Control the number of banks of weight and data
NVDLA_CMAC_A.D_MISC_CFG_0	0xffff1c03	Control convolution methods and data types.
NVDLA_CMAC_B.D_MISC_CFG_0	0xffff2003	Control convolution methods and data types.
NVDLA_CACC.D_MISC_CFG_0	0xffff2403	Control convolution methods and data types.
NVDLA_CACC.D_DATAOUT_SIZE_0	0xffff2404	Control the H and W of the output data
NVDLA_CACC.D_DATAOUT_SIZE_1	0xffff2405	Control the C of the output data
NVDLA_CACC.D_LINE_STRIDE_0	0xffff2408	Control the number of data in each line of output data
NVDLA_CACC.D_SURF_STRIDE_0	0xffff2409	Control the number of data in each surface of output data
NVDLA_SDP.D_DATA_CUBE_WIDTH_0	0xffff2c0f	Control the width of the output data
NVDLA_SDP.D_DATA_CUBE_HEIGHT_0	0xffff2c10	Control the height of the output data
NVDLA_SDP.D_DATA_CUBE_CHANNEL_0	0xffff2c11	Control the channel of the output data
NVDLA_SDP.D_DST_LINE_STRIDE_0	0xffff2c14	Control the number of data in each line of output data

NVDLA_SDP.D_DST_SURFACE_STRIDE_0	0xffff2c15	Control the number of data in each surface of output data

In the table, H stands for height, W stands for width, and C stands for channel. For the calculation of banks the following formula is used

$$bank\_of\_data = ceil(\frac{entry_per_slice * H}{256}) \tag{4.1}$$

$$bank\_of\_weight = ceil(\frac{R * S * C * K * BPE}{256 * 128}) \tag{4.2}$$

where R refers to the number of rows or height of the convolution kernel. S refers to the number of columns or width of the convolution kernel. C refers to the number of input channels. BPE refers to Block Processing Engine, which indicates the number of processing units or parallelism. K refers to the number of convolutional kernels. More register usage is described in the appendix!! not added yet

4.3 RTL

NVIDIA provided the entire RTL code for NVDLA, and the relevant CMAC code structure is

```
cmac
├── NV_NVDLA_cmac.v
├── NV_NVDLA_CMAC_reg.v
├── NV_NVDLA_CMAC_REG_single.v
├── NV_NVDLA_CMAC_REG_dual.v
├── NV_NVDLA_CMAC_core.v
├── NV_NVDLA_CMAC_CORE_slcg.v
├── NV_NVDLA_CMAC_CORE_rt_out.v
├── NV_NVDLA_CMAC_CORE_rt_in.v
├── NV_NVDLA_CMAC_CORE_mac.v
├── NV_NVDLA_CMAC_CORE_MAC_nan.v
├── NV_NVDLA_CMAC_CORE_MAC_mul.v
├── NV_NVDLA_CMAC_CORE_MAC_exp.v
├── NV_NVDLA_CMAC_CORE_cfg.v
└── NV_NVDLA_CMAC_CORE_active.v
```

Among them, the N_NVDLA_CMAC_CORE_mac contains mainly 64 NV_NVDLA_CMAC_MAC_mul, and a CSA tree, where the CSA tree is included as part of the mac code, but in order to test the CSA tree separately, we need to separate the CSA tree as a separate component in mac. There are eight NV_NVDLA_CMAC_CORE_macs mainly contained in NV_NVDLA_CMAC_core.

4.4 Synthesize

In order to be able to test the circuits at the gate level, we need to convert the RTL to netlist files, a process that we synthesize using the design compiler. We use NanGate_15nm_OCL_typical_conditional_ccs for our library. NVDLA's synthesis folder is structured as follows.

```
hw
└── syn
    ├── scripts
    │   ├── syn_launch.sh
    │   ├── default_config.sh
    │   ├── dc_run.tcl
    │   ├── dc_interactive.tcl
    │   └── dc_app_vars.tcl
    ├── templates
    │   ├── config.sh
    │   └── cg_latency_lut.tcl
    └── cons
        ├── NV_NVDLA_partition_a.sdc
        ├── NV_NVDLA_partition_c.sdc
        ├── NV_NVDLA_partition_m.sdc
        ├── NV_NVDLA_partition_o.sdc
        └── NV_NVDLA_partition_p.sdc
```

To be able to run the synthesis, we need to change the config.sh file in the templates folder. The main changes are listed in the table below.

Table 4.2: NVDLA synthesize config

name	comments
NVDLA_ROOT	Location on disk for the NVDLA source "hw" directory.

TOP_NAMES	Space separated list of TOP_NAMES to synthesize. we use "NV_NVDLA_cmac" "NV_NVDLA_CMAC_CORE_MAC_csa_tree" "NV_NVDLA_CMAC_CORE_MAC_mul" and "NV_NVDLA_CMAC_CORE_mac"
RELEASE_DIR	Location of design compiler
CGLUT_FILE	Location of the cg_latency_lut.tcl file
TARGET_LIB	Address of target library
LINK_LIB	Address of the link library

Once these parameters are set, we can run the following command under hw to run the synthesis

```
./syn/scripts/syn_launch.sh -mode wlm -config ./syn/templates/(your
    config.sh) > log.txt
```

After run the commend,The netlist will be generated by the design compiler, note that if there is no design ware library, NVDLA provides one, but the DESIGNWARE_NOEXIST parameter needs to be declared as 1, which can be defined in RTL.

4.5 Generate Test Pattern For Multiplier

For the multiplier part, since it is relatively simple combinational logic, we use TETRA MAX to automatically generate test vectors for the multiplier. The multiplier supports three types of precision, int8, int16 and fp16, and for different types of precision, we need to do some different constraints on the inputs. The multiplier input and output ports are reported in Listing 4.1.

Listing 4.1: multiplier in/out port

```
input:
exp_sft
op_a_dat
op_a_nz
op_b_dat
op_b_nz
NVDLA_core_clk
NVDLA_core_rstn
cfg_is_fp16
cfg_is_int8
```

```
cfg_reg_en
op_a_pvld
op_b_pvld

output:
res_a
res_b
res_tag
```

After synthesizing the multiplier's netlist file, only the two control signals cfg_is_int8_d1 and cfg_is_fp16_d1 are synthesized into a flip-flop, so we need to make some changes to the netlist, we remove these two flip-flops and connect the inputs and outputs of the registers directly so that the multiplier is a complete logic circuit, we can use ATPG to generate the test vector.

We will present them in the order of int16, int8, fp16.

4.5.1 Int16

We start by running the command

```
tmax -shell (your scripts).tcl
```

The code in one of the script files is as follows

Listing 4.2: int16 TETRA MAX scripts

```
read_netlist NanGate_15nm_OCL_conditional.v -library -insensitive
read_netlist NV_NVDLA_CMAC_CORE_MAC_mul_noflipflop.v -master -
    insensitive
run_build_model NV_NVDLA_CMAC_CORE_MAC_mul

add_pi_constraints 1 NVDLA_core_rstn
add_pi_constraints 1 cfg_reg_en
add_pi_constraints 0 cfg_is_fp16
add_pi_constraints 0 cfg_is_int8
add_pi_constraints x exp_sft[0]
add_pi_constraints x exp_sft[1]
add_pi_constraints x exp_sft[2]
add_pi_constraints x exp_sft[3]
add_pi_constraints 1 op_a_nz[0]
add_pi_constraints 1 op_a_nz[1]
add_pi_constraints 1 op_b_nz[0]
add_pi_constraints 1 op_b_nz[1]
add_pi_constraints 1 op_b_pvld
add_pi_constraints 1 op_a_pvld
run_drc
set_faults -model stuck
```

```
add_faults -all
set_patterns -internal
run_atpg -auto_compression
report_patterns -all
write_patterns output_patterns_int16.stil -internal -format stil -
    replace
write_faults fault_list_int16.gz -all -uncollapsed -compress gzip
```

After the run is completed we can get 38 pairs of patterns in the report (each pair contains a feature data and a weight data), in order to count the test vectors of all data types and the corresponding coverage, we also need to run the two types of test vector generation scripts, int8 and fp16, the following is the test vector generation script for int8.

4.5.2 Int8

Listing 4.3: int8 TETRA MAX scripts

```
read_netlist NanGate_15nm_OCL_conditional.v -library -insensitive
read_netlist NV_NVDLA_CMAC_CORE_MAC_mul_noflipflop.v -master -
    insensitive
run_build_model NV_NVDLA_CMAC_CORE_MAC_mul

add_pi_constraints 1 NVDLA_core_rstn
add_pi_constraints 1 cfg_reg_en
add_pi_constraints 0 cfg_is_fp16
add_pi_constraints 1 cfg_is_int8
add_pi_equivalences {op_b_dat[0] op_b_dat[8]}
add_pi_equivalences {op_b_dat[1] op_b_dat[9]}
add_pi_equivalences {op_b_dat[2] op_b_dat[10]}
add_pi_equivalences {op_b_dat[3] op_b_dat[11]}
add_pi_equivalences {op_b_dat[4] op_b_dat[12]}
add_pi_equivalences {op_b_dat[5] op_b_dat[13]}
add_pi_equivalences {op_b_dat[6] op_b_dat[14]}
add_pi_equivalences {op_b_dat[7] op_b_dat[15]}
add_pi_constraints x exp_sft[0]
add_pi_constraints x exp_sft[1]
add_pi_constraints x exp_sft[2]
add_pi_constraints x exp_sft[3]
add_pi_constraints 1 op_a_nz[0]
add_pi_constraints 1 op_a_nz[1]
add_pi_constraints 1 op_b_nz[0]
add_pi_constraints 1 op_b_nz[1]
add_pi_constraints 1 op_b_pvld
add_pi_constraints 1 op_a_pvld
run_drc
set_faults -model stuck
read_faults fault_list_int16.gz -retain_code
```

```
set_patterns -internal
run_atpg -auto_compression
report_patterns -all
write_patterns output_patterns_int8.stil -internal -format stil -
    replace
write_faults fault_list_int8.gz -all -uncollapsed -compress gzip
```

After the run is completed we can get 10 pairs of patterns in the report. For the test vector generation script for int8, we need to make some changes to the constraints because we want to generate as many test vectors as possible as small as possible, and also the multiplier for int8 is used by splitting the multiplier in two and using the higher and lower octets of the two operands for computation respectively, so we set the constraints on the corresponding bits of the higher and lower octets to be equal. Also in order to inherit the tested test points from the int16 coverage, we need to read the int16 part of the saved fault list.

4.5.3 Fp16

In order to generate test vectors for all the data types, we also need to generate test vectors for fp16, here is the script for generating fp16 test vectors.

Listing 4.4: fp16 TETRA MAX scripts

```
read_netlist NanGate_15nm_OCL_conditional.v -library -insensitive
read_netlist NV_NVDLA_CMAC_CORE_MAC_mul_noflipflop.v -master -
    insensitive

run_build_model NV_NVDLA_CMAC_CORE_MAC_mul

add_pi_constraints 1 NVDLA_core_rstn
add_pi_constraints 1 cfg_reg_en
add_pi_constraints 1 cfg_is_fp16
add_pi_constraints 0 cfg_is_int8
add_pi_constraints 0 exp_sft[0]
add_pi_constraints 0 exp_sft[1]
add_pi_constraints 0 exp_sft[2]
add_pi_constraints 0 exp_sft[3]
#weight 0 sft feature 0 sft:
{#wt_constraints
add_pi_constraints 0 op_a_dat[14]
add_pi_constraints 0 op_a_dat[13]
add_pi_constraints 0 op_a_dat[12]
add_pi_constraints 0 op_a_dat[11]


#feature_constraints
add_pi_constraints 0 op_b_dat[14]
add_pi_constraints 0 op_b_dat[13]
```

```
25 add_pi_constraints 0 op_b_dat[12]
26 add_pi_constraints 0 op_b_dat[11]}
27
28 #weight 1 sft feature 0 sft:
29 {#wt_constraints
30 add_pi_constraints 0 op_a_dat[14]
31 add_pi_constraints 0 op_a_dat[13]
32 add_pi_constraints 0 op_a_dat[12]
33 add_pi_constraints 0 op_a_dat[0]
34
35
36 #feature_constraints
37 add_pi_constraints 0 op_b_dat[14]
38 add_pi_constraints 0 op_b_dat[13]
39 add_pi_constraints 0 op_b_dat[12]
40 add_pi_constraints 0 op_b_dat[11]}
41
42 #weight 2 sft feature 0 sft:
43 {#wt_constraints
44 add_pi_constraints 0 op_a_dat[14]
45 add_pi_constraints 0 op_a_dat[13]
46 add_pi_constraints 0 op_a_dat[1]
47 add_pi_constraints 0 op_a_dat[0]
48
49
50 #feature_constraints
51 add_pi_constraints 0 op_b_dat[14]
52 add_pi_constraints 0 op_b_dat[13]
53 add_pi_constraints 0 op_b_dat[12]
54 add_pi_constraints 0 op_b_dat[11]}
55
56 #weight 3 sft feature 0 sft:
57 {#wt_constraints
58 add_pi_constraints 0 op_a_dat[14]
59 add_pi_constraints 0 op_a_dat[2]
60 add_pi_constraints 0 op_a_dat[1]
61 add_pi_constraints 0 op_a_dat[0]
62
63
64 #feature_constraints
65 add_pi_constraints 0 op_b_dat[14]
66 add_pi_constraints 0 op_b_dat[13]
67 add_pi_constraints 0 op_b_dat[12]
68 add_pi_constraints 0 op_b_dat[11]}
69
70 #weight 0 sft feature 1 sft:
71 {#wt_constraints
72 add_pi_constraints 0 op_a_dat[14]
73 add_pi_constraints 0 op_a_dat[13]
```

```
74 add_pi_constraints 0 op_a_dat[12]
75 add_pi_constraints 0 op_a_dat[11]
76
77
78 #feature_constraints
79 add_pi_constraints 0 op_b_dat[14]
80 add_pi_constraints 0 op_b_dat[13]
81 add_pi_constraints 0 op_b_dat[12]
82 add_pi_constraints 0 op_b_dat[0]}
83
84 #weight 1 sft feature 1 sft:
85 {#wt_constraints
86 add_pi_constraints 0 op_a_dat[14]
87 add_pi_constraints 0 op_a_dat[13]
88 add_pi_constraints 0 op_a_dat[12]
89 add_pi_constraints 0 op_a_dat[0]
90
91
92 #feature_constraints
93 add_pi_constraints 0 op_b_dat[14]
94 add_pi_constraints 0 op_b_dat[13]
95 add_pi_constraints 0 op_b_dat[12]
96 add_pi_constraints 0 op_b_dat[0]}
97
98 #weight 2 sft feature 1 sft:
99 {#wt_constraints
100 add_pi_constraints 0 op_a_dat[14]
101 add_pi_constraints 0 op_a_dat[13]
102 add_pi_constraints 0 op_a_dat[1]
103 add_pi_constraints 0 op_a_dat[0]
104
105
106 #feature_constraints
107 add_pi_constraints 0 op_b_dat[14]
108 add_pi_constraints 0 op_b_dat[13]
109 add_pi_constraints 0 op_b_dat[12]
110 add_pi_constraints 0 op_b_dat[0]}
111
112 #weight 3 sft feature 1 sft:
113 {#wt_constraints
114 add_pi_constraints 0 op_a_dat[14]
115 add_pi_constraints 0 op_a_dat[2]
116 add_pi_constraints 0 op_a_dat[1]
117 add_pi_constraints 0 op_a_dat[0]
118
119
120 #feature_constraints
121 add_pi_constraints 0 op_b_dat[14]
122 add_pi_constraints 0 op_b_dat[13]
```

```
123 add_pi_constraints 0 op_b_dat[12]
124 add_pi_constraints 0 op_b_dat[0]}
125
126 #weight 0 sft feature 2 sft:
127 {#wt_constraints
128 add_pi_constraints 0 op_a_dat[14]
129 add_pi_constraints 0 op_a_dat[13]
130 add_pi_constraints 0 op_a_dat[12]
131 add_pi_constraints 0 op_a_dat[11]
132
133
134 #feature_constraints
135 add_pi_constraints 0 op_b_dat[14]
136 add_pi_constraints 0 op_b_dat[13]
137 add_pi_constraints 0 op_b_dat[1]
138 add_pi_constraints 0 op_b_dat[0]}
139
140 #weight 1 sft feature 2 sft:
141 {#wt_constraints
142 add_pi_constraints 0 op_a_dat[14]
143 add_pi_constraints 0 op_a_dat[13]
144 add_pi_constraints 0 op_a_dat[12]
145 add_pi_constraints 0 op_a_dat[0]
146
147
148 #feature_constraints
149 add_pi_constraints 0 op_b_dat[14]
150 add_pi_constraints 0 op_b_dat[13]
151 add_pi_constraints 0 op_b_dat[1]
152 add_pi_constraints 0 op_b_dat[0]}
153
154 #weight 2 sft feature 2 sft:
155 {#wt_constraints
156 add_pi_constraints 0 op_a_dat[14]
157 add_pi_constraints 0 op_a_dat[13]
158 add_pi_constraints 0 op_a_dat[1]
159 add_pi_constraints 0 op_a_dat[0]
160
161
162 #feature_constraints
163 add_pi_constraints 0 op_b_dat[14]
164 add_pi_constraints 0 op_b_dat[13]
165 add_pi_constraints 0 op_b_dat[1]
166 add_pi_constraints 0 op_b_dat[0]}
167
168 #weight 3 sft feature 2 sft:
169 {#wt_constraints
170 add_pi_constraints 0 op_a_dat[14]
171 add_pi_constraints 0 op_a_dat[2]
```

```
172 add_pi_constraints 0 op_a_dat[1]
173 add_pi_constraints 0 op_a_dat[0]
174
175
176 #feature_constraints
177 add_pi_constraints 0 op_b_dat[14]
178 add_pi_constraints 0 op_b_dat[13]
179 add_pi_constraints 0 op_b_dat[1]
180 add_pi_constraints 0 op_b_dat[0]}
181
182 #weight 0 sft feature 3 sft:
183 {#wt_constraints
184 add_pi_constraints 0 op_a_dat[14]
185 add_pi_constraints 0 op_a_dat[13]
186 add_pi_constraints 0 op_a_dat[12]
187 add_pi_constraints 0 op_a_dat[11]
188
189
190 #feature_constraints
191 add_pi_constraints 0 op_b_dat[14]
192 add_pi_constraints 0 op_b_dat[2]
193 add_pi_constraints 0 op_b_dat[1]
194 add_pi_constraints 0 op_b_dat[0]}
195
196 #weight 1 sft feature 3 sft:
197 {#wt_constraints
198 add_pi_constraints 0 op_a_dat[14]
199 add_pi_constraints 0 op_a_dat[13]
200 add_pi_constraints 0 op_a_dat[12]
201 add_pi_constraints 0 op_a_dat[0]
202
203
204 #feature_constraints
205 add_pi_constraints 0 op_b_dat[14]
206 add_pi_constraints 0 op_b_dat[2]
207 add_pi_constraints 0 op_b_dat[1]
208 add_pi_constraints 0 op_b_dat[0]}
209
210 #weight 2 sft feature 3 sft:
211 {#wt_constraints
212 add_pi_constraints 0 op_a_dat[14]
213 add_pi_constraints 0 op_a_dat[13]
214 add_pi_constraints 0 op_a_dat[1]
215 add_pi_constraints 0 op_a_dat[0]
216
217
218 #feature_constraints
219 add_pi_constraints 0 op_b_dat[14]
220 add_pi_constraints 0 op_b_dat[2]
```

```
add_pi_constraints 0 op_b_dat[1]
add_pi_constraints 0 op_b_dat[0]}

#weight 3 sft feature 3 sft:
{#wt_constraints
add_pi_constraints 0 op_a_dat[14]
add_pi_constraints 0 op_a_dat[2]
add_pi_constraints 0 op_a_dat[1]
add_pi_constraints 0 op_a_dat[0]


#feature_constraints
add_pi_constraints 0 op_b_dat[14]
add_pi_constraints 0 op_b_dat[2]
add_pi_constraints 0 op_b_dat[1]
add_pi_constraints 0 op_b_dat[0]}


add_pi_constraints 1 op_a_nz[0]
add_pi_constraints 1 op_a_nz[1]
add_pi_constraints 1 op_b_nz[0]
add_pi_constraints 1 op_b_nz[1]
add_pi_constraints 1 op_b_pvld
add_pi_constraints 1 op_a_pvld
run_drc
set_faults -model stuck
read_faults fault_list_int8.gz -retain_code
set_patterns -internal
run_atpg -auto_compression
report_patterns -all
write_patterns output_patterns_fp16_no_expsft.stil -internal -format
    stil -replace
```

After the run is completed we can get 23 pairs of patterns in the report. For the test vector generation script of fp16, we need to make some modifications to the constraints, for the exp_sft part, the main principle is to first compare the exponent part of all 64 input feature data and weight data, with every 4 multipliers as a group, and then select the largest exponent among them with the other indices for the Some shift operations are calculated, but since this method treats 64 multipliers as the same input data, we constrain all the bits of exp_sft to 0. At the same time, we constrain the corresponding possible shifts of feature data and weight data, where weight data and feature data may be shifted from 0 to 3 bits, respectively. The weight data and the feature data may be shifted from 0 to 3 bits respectively, and we will generate the corresponding test vectors by arranging and combining all the possible cases.

4.5.4 Test Vectors To Feture / Weight Data

After completing the generation of all the test vectors, for fp16, we need to reduce the obtained test vectors into the corresponding FEATURE DATA and WEIGHT DATA, because NVDLA first preprocesses the data and then passes it into the multiplier afterward, so we use Python according to the preprocessing method of NVDLA to reduce it into the corresponding So we use python to reduce it to the corresponding feature data and weight data according to the preprocessing method of NVDLA so that we can get the correct feature data weight data. Below is the Python code used

Listing 4.5: fp16 test vector to original input data

```
fp16_in = '0000000000000000'
fp16    = '1011011111001000'
for i in range(65535):
    fp16_mts_ori0 = '01' + fp16_in[6:16]
    fp16_mts_ori1 = '0' + fp16_in[6:16] + '0'

    fp_sft = int(fp16_in[4:6],2)

    fp16_mts_sft_0_0 = '000' + fp16_mts_ori0
    fp16_mts_sft_0_1 = int(fp16_mts_sft_0_0,2) << fp_sft
    fp16_mts_sft_0_2 = bin(fp16_mts_sft_0_1)
    fp16_mts_sft_0_3 = fp16_mts_sft_0_2[2 : (len(fp16_mts_sft_0_2) +
    1)]

    fp16_mts_sft_1_0 = '000' + fp16_mts_ori1
    fp16_mts_sft_1_1 = int(fp16_mts_sft_1_0,2) << fp_sft
    fp16_mts_sft_1_2 = bin(fp16_mts_sft_1_1)
    fp16_mts_sft_1_3 = fp16_mts_sft_1_2[2:(len(fp16_mts_sft_1_2) + 1)
    ]

    for x in range(len(fp16_mts_sft_0_3),15):
        fp16_mts_sft_0_3 = '0' + fp16_mts_sft_0_3
    fp16_result = fp16_in[0] + fp16_mts_sft_0_3

    for x in range(len(fp16_mts_sft_1_3),15):
        fp16_mts_sft_1_3 = '0' + fp16_mts_sft_1_3
    fp16_result_1 = fp16_in[0] + fp16_mts_sft_1_3
    if fp16_result == fp16:
        if int(fp16_in[1:6],2) != 0:
            "print('matched1')"
            print(fp16_in)


    if fp16_result_1 == fp16:
        if int(fp16_in[1:6],2) == 0:
            "print('matched2')"
```

```
35              print(fp16_in)
36
37
38      fp16_in_temp_0 = int(fp16_in,2) + 1
39      fp16_in_temp_1 = bin(fp16_in_temp_0)
40      fp16_in = fp16_in_temp_1[2 : (len(fp16_in_temp_1)+1)]
41      for n in range(len(fp16_in),16):
42          fp16_in = '0' + fp16_in
43  print('end')
```

The Python code searches for possible original data starting from 0. When there is a corresponding original data that can be computed to be equal to the test vector of fp16 generated by TETRA MAX, then it is proved that the multiplier input corresponding to this original data is the test vector we want.

4.5.5 Simulated And Fault Emulation

When we have the corresponding original data, we can configure the original data as the corresponding data cube, and at the same time set up the corresponding register configuration, start the simulation, and dump down the waveform file to the z01x for fault simulation. In order to run the simulation, we need to run the following command in the hw/verif/sim folder

```
1      make run DUMP=1 DUMPER=VERDI TESTDIR=../traces/traceplayer/(your
      test dir)
```

Among other things, the test folder should contain the appropriate feture data file, weight data file, and register configuration file. After the run is completed, the corresponding waveform file is in hw/verif/sim/(your test dir). Since we have dumped all the waveforms, but we only need the waveforms corresponding to the input and output ports of the multiplier, in order to be able to use the waveform file in Z01X again, we need to split the waveform file generated by the simulation, and we use the following instruction to split the waveform file.

```
1      vcsplit -o (destination).vcd -include scopefile_mul.txt (source).
      vcd
```

where scopefile_mul.txt contains the names of all the ports you wish to keep. This is shown below

```
1  top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
       NVDLA_core_clk
```

```
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    NVDLA_core_rstn
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    cfg_is_fp16
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    cfg_is_int8
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    cfg_reg_en
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    exp_sft
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    op_a_dat
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    op_a_nz
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    op_a_pvld
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    op_b_dat
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    op_b_nz
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    op_b_pvld
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    res_a
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    res_b
top.NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0.
    res_tag
```

In order to run the Z01X, we need to do some configuration, there are 4 configuration files for the Z01X, they are:

- fault_list.sff
  This file is used to set the fault generation constraints and to create faults on all ports in the hierarchy The contents of the file are as follows.

```
# Set fault generation constraints
FaultGenerate
{
   # Create faults on all ports in hierarchy
    NA [0,1] { PORT "NV_NVDLA_CMAC_CORE_MAC_mul.**" }
}

```

- load_design.f
  This file is used to declare the addresses of the netlist and library files for Z01X compilation. The contents of the file are as follows.

```
+incdir+./bin

-v (Address and directory of the netlist file).v
(Address and directory of library files).v

```

- permanent_faults_gen.fmsh
  This file is to declare the type of simulation and some of its parameters, to run tests and simulations, and to generate commands for reports. The contents of the file are as follows.

```
set(var=[resources], messages=[all])
set(var=[defines], format=[standard])
set(var=[defines], dictionary.enable=[1])
set(var=[fsim], dictionary.values=[all])
set(var=[fdef], method=[fr], fr.fr=[./bin/fault_list.sff],abort=[
    error])
set (var=[fsim],hyperfault=[0])
design()
addtst(test=[NV_NVDLA_CMAC_CORE_MAC_mul], stimtype=[vcd],
    dictionary.enable=[1], stim=[(The address of the vcd file and
    its directory).vcd],dut.stim=[NV_NVDLA_CMAC_CORE_MAC_mul,top.
    NVDLA_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_mul_0],stim_options=[+TESTNAME=NV_NVDLA_CMAC_CORE_MAC_mul] )
fsim()
coverage(file=[SAF_coverage.sff])
coverage (type=[dictionary],file=[saf_dic_long.txt],test=[
    NV_NVDLA_CMAC_CORE_MAC_mul])

```

- strobes.sv
  This file declares the timescale, Inject faults, and strobe methods. The contents of the file are as follows.

```
// ZOIX MODULE FOR FAULT INJECTION AND STROBING
`timescale 1ns / 1ns
module strobe;
    // Inject faults
    initial begin
        $display("ZOIX INJECTION");
        $fs_inject;  //      .. by default
        $fs_delete;
        $fs_add(NV_NVDLA_CMAC_CORE_MAC_mul);
```

```
10      end
11      initial begin
12          #30
13          forever  begin
14          $fs_strobe(NV_NVDLA_CMAC_CORE_MAC_mul.res_a);
15          $fs_strobe(NV_NVDLA_CMAC_CORE_MAC_mul.res_b);
16          $fs_strobe(NV_NVDLA_CMAC_CORE_MAC_mul.res_tag);
17          #40;
18          end
19      end
20  endmodule
21
```

- launch_zoixs.sh
  This file is the script file used to run Z01X. The contents of the file are as follows.

```
1  #!/bin/bash
2  ./clean.csh
3  rm -rf zoix.sim sim.zdb faults.fdef simout-N0* zoix.log zoix.
       progress* zoix_rt.log *.log* *.cdf
4  zoix -f ./bin/load_design.f ./bin/strobes.sv +timescale+override
       +1ns/1ns +top+NV_NVDLA_CMAC_CORE_MAC_mul+strobe +sv +
       notimingchecks +define+ZOIX +suppress+cell +delay_mode_fault -
       l log/zoix_compile.log
5  fmsh -blast sim
6  ./zoix.sim +vcd+file+"(Address and directory of vcd files).vcd" +
       vcd+dut+NV_NVDLA_CMAC_CORE_MAC_mul+top.NVDLA_top.
       u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.u_mul_0 +vcd+
       verify +vcd+verbose +vcd+limit+mismatch+20000 -l logic_sim.log
7  fmsh -load ./bin/permanent_faults_gen.fmsh
8
```

After setting up these files, we can use the following command to run Z01X.

```
1      ./launch_zoixs.sh
```

Once the run is complete, we get the following main report file.

- SAF_coverage.sff

- saf_dic_long.txt

Both files contain all the fault information, but the saf_dic_long.txt file contains the relevant time information. At this point, we have completed the test vector generation session for the multiplier part.

4.6 Generate Test Pattern For CSA Tree

The CSA tree is the addition part after all the multipliers have produced their results, and the input of the CSA tree is the output of the multiplier, which belongs to the intermediate variables and is difficult to be de-activated, and the CSA tree belongs to the Sequential Circuit of the multistage pipeline.Therefore, in order to detect the CSA tree at the functional level, we use the randomized input variables with the constraints We use a constrained randomization of input variables to detect the CSA tree at the functional level.

4.6.1 Random Generation Of Feature/weight Data

We used python to write an automation script that can be configured with randomized feature data and weight data of different sizes. Following is the script for randomly generating data.

```
    import numpy as np
import random
import os
def hex_to_dec(hex_str):
    if hex_str[0] in '01234567':
        dec_data = int(hex_str, 16)
    else:
        width = 32
        d = 'FFFF' + hex_str
        dec_data = int(d, 16)
        if dec_data > 2 ** (width - 1) - 1:
            dec_data = 2 ** width - dec_data
            dec_data = 0 - dec_data
    if(dec_data > 2**15-1 or dec_data < -2**15):
        print('out of range')
    return(dec_data)

#-----------------------------for input data--------------------------------
channels = 128  #need to be config(unit is bytes)
rows = 32       #need to be config(unit is number of rows)
colums = 32     #need to be config(unit is number of colums)
surfaces = int(np.ceil(channels/32))
print(surfaces)

```

```
25 src = np.zeros((rows,colums,channels), dtype = int)
26
27 list_txt = []
28 file = open('sample_surf_random.dat',mode = 'a')
29 print(int(np.ceil(channels/2)))
30
31 if int(np.ceil(channels/2)) < 16:
32     n = int(np.ceil(channels/2))
33 else:
34     n = 16
35
36 for surface in range(surfaces):
37     for row in range(rows):
38         for colum in range(colums):
39             for channel in range(0,n):
40                 a = hex(random.randrange(256))
41                 b = hex(random.randrange(256))
42                 if len(a) != 4:
43                     str_list_0 = list(a)
44                     str_list_0.insert(2,'0')
45                     a = ''.join(str_list_0)
46                 if len(b) != 4:
47                     str_list_1 = list(b)
48                     str_list_1.insert(2,'0')
49                     b = ''.join(str_list_1)
50                 c = a[2:4] + b[2:4]
51                 src[row][colum][(surface*16)+channel] = hex_to_dec(c)
52                 d = b + ' ' + a + ' '
53                 if channel == 15 or channel == 31 or channel == 47 or
       channel == 63 or channel == 79 or channel == 95 or channel == 111
       or channel == 127:
54                     d = b + ' ' + a + '\r'
55                 list_txt.append(d)
56             file.writelines(list_txt)
57             list_txt = []
58 file.close()
59
60 #-----------------------for weight data----------------
61 weight_channels = 128   #need to be config(unit is bytes)
62 weight_rows = 32         #need to be config(unit is number of rows)
63 weight_colums = 32       #need to be config(unit is number of colums)
64 kernals = 1             #need to be config(unit is number of kernals)
65 weight_surfaces = int(np.ceil(weight_channels / 128))
66 print(weight_surfaces)
67
68 weight_src = np.zeros((kernals,weight_rows,weight_colums,
       weight_channels), dtype = int)
69
70 weight_list_txt = []
```

```
71 weight_file = open('weight_random.dat',mode = 'a')
72
73 if int(np.ceil(weight_channels/2)) < 64:
74     n = int(np.ceil(weight_channels/2))
75 else:
76     n = 64
77
78 for surface in range(weight_surfaces):
79     for row in range(weight_rows):
80         for colum in range(weight_colums):
81             for kernal in range(kernals):
82                 for channel in range(0,n):#1 channel is 128bytes and
    64 elements
83                     weight_a = hex(random.randrange(256))
84                     weight_b = hex(random.randrange(256))
85                     if len(weight_a) != 4:
86                         weight_str_list_0 = list(weight_a)
87                         weight_str_list_0.insert(2,'0')
88                         weight_a = ''.join(weight_str_list_0)
89
90                     if len(weight_b) != 4:
91                         weight_str_list_1 = list(weight_b)
92                         weight_str_list_1.insert(2,'0')
93                         weight_b = ''.join(weight_str_list_1)
94
95                     weight_c = weight_a[2:4] + weight_b[2:4]
96                     weight_src[kernal][row][colum][(surface*64)+
    channel] = hex_to_dec(c)
97                     weight_d = weight_b + ' ' + weight_a + ' '
98                     if channel == 15 or channel == 31 or channel ==
    47 or channel == 63 or channel == 79 or channel == 95 or channel
    == 111 or channel == 127:
99                         weight_d = weight_b + ' ' + weight_a + '\r'
100                    weight_list_txt.append(weight_d)
101                weight_file.writelines(weight_list_txt)
102                weight_list_txt = []
103 weight_file.close()
104
105 final_data = open('sample_surf_random.dat',mode = 'r')
106 temp_list1 = final_data.readlines()
107 temp0 = temp_list1[len(temp_list1)-1].strip()
108 del temp_list1[len(temp_list1)-1]
109 temp_list1.append(temp0)
110 final_data1 = open('sample_surf.dat',mode = 'w')
111 final_data1.writelines(temp_list1)
112 final_data1.close()
113 os.remove("sample_surf_random.dat")
114
115 final_weight = open('weight_random.dat',mode = 'r')
```

```
116 temp_list2 = final_weight.readlines()
117 temp1 = temp_list2[len(temp_list2)-1].strip()
118 del temp_list2[len(temp_list2)-1]
119 temp_list2.append(temp1)
120 final_weight1 = open('weight.dat',mode = 'w')
121 final_weight1.writelines(temp_list2)
122 final_weight1.close()
123 os.remove("weight_random.dat")
```

By running this script, the corresponding feature data and weight data will be generated automatically.

4.6.2 Simulated And Fault Emulation

The corresponding waveform file can be obtained by simulating the generated data with the corresponding configuration file. Same as before, we need to split the waveform file, use the following command and scopefile to keep the input and output ports related to the csa tree in the waveform file. Since the scopefile is too large, it will be in the appendix Listing 7.1.

```
1    vcsplit -o (destination).vcd -include scopefile_csa_tree.txt (
   source).vcd
```

As in the multiplier section, we only need to change the corresponding parameters to run Z01X to generate coverage and fault reports. It is important to note that in the CSA tree section, we loop through the generation of different feature data and weight data, and by using the timing information in the saf_dic_long.txt file, we can locate which feature data and weight data are contributing to increase the test coverage, and record them, and finally synthesize the new feature data and weight data for the final coverage test. For each of the three different data types, we looped int16, int8, and fp16 500 times, and used the cumulative test coverage as the final csa tree coverage.

Chapter 5

Results

In this thesis, the test vectors generated for the cmac part of NVDLA using the customized method. After converting the generated test vectors into corresponding NVDLA input data. Achieve relatively good results. Comparing with the sanity3 test provided by NVDLA. The test vectors in this thesis far exceed the sanity3 test provided by NVDLA in terms of test coverage. The following are the results of the test coverage in this thesis and the sanity3 test of NVDLA Coverage results.

Table 5.1: Results

Component	Thesis Fault Coverage	NVDLA Sanity3 Fault Coverage
Single Multiplier	98.56%	77.50%
Total Multiplier	98.56%	77.50%
Csa Tree	76.36%	41.48%
Single Mac Unit(compute part)	93.18%	69.54%
CMAC(compute part)	93.18%	69.54%
CMAC	77.06%	57.68%

Table 5.2: Duration and Size

	Thesis vectors	NVDLA sanity3
Number of patterns	6251	512
Patterns size	1.55MBytes	4KBytes
Duration(0.025Ghz)	250.04us	20.48us
Duration(2.5Ghz)	2.5004us	0.2048us

Table 5.3: Faults Information

Component	Total SA Faults	Untestable Faults	Detected
Single Multiplier	10226	1112	8983
Total Multiplier	654464	71168	574912
Csa Tree	197951	11536	142363
Single Mac Unit (compute part)	852415	82704	717275
CMAC(compute part)	13638640	1323264	11476400

Chapter 6

Discussion And Conclusions

With the method described in this paper, we can target manual test vector design of single stuck-at model for AI-Oriented Hardware Accelerators for In-field test. Applying ATPG to the design of test vectors can greatly reduce the time. For structures with high repetitiveness, we can extend to all repetitive parts by testing one of them, which simplifies the design of test vectors, and obtains relatively high fault coverage but relatively small size test vectors. However, the method in this paper requires in-depth understanding of the hardware, and by analyzing the structure of the hardware, we can delineate the parts that can be applied to the ATPG and understand the algorithms, and through some constraints, we can exclude the parts that can be applied to the test vectors. algorithms, and passing some constraints to exclude some cases of impossible functions. This requires some time and effort and is not too easy to accomplish.

Chapter 7

Appendix

Listing 7.1: scopefile_csa_tree

```
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.nvdla_core_clk
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.nvdla_wg_clk
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.nvdla_core_rstn
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.cfg_is_fp16
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.cfg_is_int16
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.cfg_is_int8
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.cfg_is_wg
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.cfg_reg_en
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.dat_actv_pvld
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.wt_actv_pvld
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.out_nan_pvld
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.exp_max
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.exp_pvld
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.exp_sft_00
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
  u_csa_tree.exp_sft_01
```

```
16  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_02
17  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_03
18  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_04
19  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_05
20  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_06
21  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_07
22  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_08
23  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_09
24  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_10
25  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_11
26  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_12
27  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_13
28  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_14
29  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_15
30  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_16
31  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_17
32  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_18
33  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_19
34  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_20
35  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_21
36  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_22
37  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_23
38  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_24
39  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_25
```

```
40  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_26
41  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_27
42  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_28
43  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_29
44  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_30
45  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_31
46  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_32
47  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_33
48  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_34
49  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_35
50  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_36
51  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_37
52  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_38
53  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_39
54  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_40
55  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_41
56  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_42
57  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_43
58  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_44
59  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_45
60  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_46
61  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_47
62  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_48
63  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_49
```

```
64  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_50
65  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_51
66  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_52
67  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_53
68  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_54
69  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_55
70  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_56
71  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_57
72  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_58
73  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_59
74  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_60
75  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_61
76  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_62
77  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.exp_sft_63
78  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_00
79  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_01
80  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_02
81  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_03
82  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_04
83  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_05
84  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_06
85  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_07
86  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_08
87  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_09
```

```
88  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_10
89  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_11
90  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_12
91  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_13
92  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_14
93  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_15
94  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_16
95  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_17
96  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_18
97  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_19
98  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_20
99  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_21
100 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_22
101 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_23
102 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_24
103 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_25
104 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_26
105 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_27
106 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_28
107 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_29
108 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_30
109 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_31
110 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_32
111 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_33
```

```
112 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_34
113 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_35
114 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_36
115 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_37
116 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_38
117 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_39
118 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_40
119 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_41
120 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_42
121 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_43
122 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_44
123 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_45
124 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_46
125 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_47
126 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_48
127 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_49
128 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_50
129 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_51
130 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_52
131 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_53
132 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_54
133 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_55
134 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_56
135 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_57
```

```
136 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_58
137 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_59
138 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_60
139 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_61
140 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_62
141 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_a_63
142 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_00
143 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_01
144 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_02
145 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_03
146 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_04
147 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_05
148 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_06
149 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_07
150 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_08
151 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_09
152 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_10
153 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_11
154 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_12
155 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_13
156 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_14
157 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_15
158 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_16
159 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_17
```

```
160  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_18
161  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_19
162  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_20
163  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_21
164  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_22
165  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_23
166  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_24
167  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_25
168  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_26
169  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_27
170  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_28
171  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_29
172  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_30
173  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_31
174  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_32
175  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_33
176  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_34
177  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_35
178  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_36
179  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_37
180  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_38
181  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_39
182  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_40
183  top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
       u_csa_tree.res_b_41
```

```
184 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_42
185 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_43
186 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_44
187 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_45
188 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_46
189 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_47
190 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_48
191 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_49
192 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_50
193 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_51
194 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_52
195 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_53
196 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_54
197 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_55
198 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_56
199 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_57
200 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_58
201 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_59
202 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_60
203 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_61
204 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_62
205 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_b_63
206 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_tag_0
207 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_tag_1
```

```
208   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_10
209   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_11
210   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_12
211   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_13
212   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_14
213   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_15
214   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_16
215   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_17
216   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_18
217   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_19
218   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_2
219   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_20
220   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_21
221   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_22
222   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_23
223   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_24
224   top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_25
225 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_tag_26
226 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_tag_27
227 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_tag_28
228 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_tag_29
229 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_tag_3
230 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_tag_30
231 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
      u_csa_tree.res_tag_31
```

```
232 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_32
233 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_33
234 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_34
235 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_35
236 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_36
237 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_37
238 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_38
239 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_39
240 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_4
241 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_40
242 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_41
243 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_42
244 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_43
245 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_44
246 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_45
247 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_46
248 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_47
249 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_48
250 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_49
251 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_5
252 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_50
253 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_51
254 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_52
255 top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
        u_csa_tree.res_tag_53
```

```
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_54
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_55
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_56
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_57
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_58
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_59
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_6
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_60
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_61
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_62
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_63
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_7
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_8
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.res_tag_9
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.mac_out_data
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.mac_out_nan
top.nvdla_top.u_partition_ma.u_NV_NVDLA_cmac.u_core.u_mac_0.
    u_csa_tree.mac_out_pvld
```

Table 7.1: NVDLA Register Map

Name	Address	Description
NVDLA_GLB. HW_VERSION	0x0000	HW version of NVDLA
NVDLA_GLB. INTR_MASK	0x0001	Interrupt mask control
NVDLA_GLB. INTR_SET	0x0002	Interrupt set control

NVDLA_GLB. INTR_STATUS	0x0003	Interrupt status
NVDLA_MCIF. CFG_RD_WEIGHT_0	0x0800	Register0 to control the read weight of clients in MCIF
NVDLA_MCIF. CFG_RD_WEIGHT_1	0x0801	Register1 to control the read weight of clients in MCIF
NVDLA_MCIF. CFG_RD_WEIGHT_2	0x0802	Register2 to control the read weight of clients in MCIF
NVDLA_MCIF. CFG_WR_WEIGHT_0	0x0803	Register0 to control the write weight of clients in MCI
NVDLA_MCIF. CFG_WR_WEIGHT_1	0x0804	Register1 to control the write weight of clients in MCI
NVDLA_MCIF. CFG_OUTSTANDING_CNT	0x0805	Outstanding AXI transactions in unit of 64Byte
NVDLA_MCIF. STATUS	0x0806	Idle status of MCIF
NVDLA_SRAMIF. CFG_RD_WEIGHT_0	0x0C00	Register0 to control the read weight of clients in SRAMIF(CVIF)
NVDLA_SRAMIF. CFG_RD_WEIGHT_1	0x0C01	Register1 to control the read weight of clients in SRAMIF(CVIF)
NVDLA_SRAMIF. CFG_RD_WEIGHT_2	0x0C02	Register2 to control the read weight of clients in SRAMIF(CVIF)
NVDLA_SRAMIF. CFG_WR_WEIGHT_0	0x0C03	Register0 to control the write weight of clients in SRAMIF(CVIF)
NVDLA_SRAMIF. CFG_WR_WEIGHT_1	0x0C04	Register1 to control the write weight of clients in SRAMIF(CVIF)

NVDLA_SRAMIF. CFG_OUTSTANDING_CNT	0x0C05	Outstanding AXI transactions in unit of 64Byte
NVDLA_SRAMIF. STATUS	0x0C06	Idle status of SRAMIF(CVIF)
NVDLA_BDMA. CFG_SRC_ADDR_LOW	0x1000	Lower 32bits of source address
NVDLA_BDMA. CFG_SRC_ADDR_HIGH	0x1001	Higher 32bits of source address when axi araddr is 64bits
NVDLA_BDMA. CFG_DST_ADDR_LOW	0x1002	Lower 32bits of dest address
NVDLA_BDMA. CFG_DST_ADDR_HIGH	0x1003	Higher 32bits of dest address when axi awaddr is 64bits
NVDLA_BDMA. CFG_LINE	0x1004	Size of one line
NVDLA_BDMA. CFG_CMD	0x1005	Ram type of source and destination
NVDLA_BDMA. CFG_LINE_REPEAT	0x1006	Number of lines to be moved in one surface
NVDLA_BDMA. CFG_SRC_LINE	0x1007	Source line stride
NVDLA_BDMA. CFG_DST_LINE	0x1008	Destination line stride
NVDLA_BDMA. CFG_SURF_REPEAT	0x1009	Number of surfaces to be moved in one operation
NVDLA_BDMA. CFG_SRC_SURF	0x100A	Source surface stride
NVDLA_BDMA. CFG_DST_SURF	0x100B	Destination surface stride
NVDLA_BDMA. CFG_OP	0x100C	This register is not used in NVDLA 1.0
NVDLA_BDMA. CFG_LAUNCH0	0x100D	Set it to 1 to kick off operations in group0
NVDLA_BDMA. CFG_LAUNCH1	0x100E	Set it to 1 to kick off operations in group1

NVDLA_BDMA. CFG_STATUS	0x100F	Enable/Disable of counting stalls
NVDLA_BDMA. STATUS	0x1010	Status register: idle status of bdma, group0 and group1
NVDLA_BDMA. STAUS_GRP0_READ_STALL	0x1011	Counting register of group0 read stall
NVDLA_BDMA. STATUS_GRP0_WRITE_STALL	0x1012	Counting register of group0 write stall
NVDLA_BDMA. STAUS_GRP1_READ_STALL	0x1013	Counting register of group1 read stall
NVDLA_BDMA. STATUS_GRP1_WRITE_STALL	0x1014	Counting register of group1 write stall
NVDLA_CDMA. S_STATUS	0x1400	Idle status of two register groups
NVDLA_CDMA. S_POINTER	0x1401	Pointer for CSB master and data path to access groups
NVDLA_CDMA. S_ARBITER	0x1402	"WMB and Weight share same port to access external memory. This register controls the weight factor in the arbiter."
NVDLA_CDMA. S_CBUF_FLUSH_STATUS	0x1403	Indicates whether CBUF flush is finished after reset.
NVDLA_CDMA. D_OP_ENABLE	0x1404	Set it to 1 to kick off operation for current register group
NVDLA_CDMA. D_MISC_CFG	0x1405	Configuration of operation: convolution mode, precision, weight reuse, data reuse.
NVDLA_CDMA. D_DATAIN_FORMAT	0x1406	Input data format and pixel format
NVDLA_CDMA. D_DATAIN_SIZE_0	0x1407	Input cube's width and height

NVDLA_CDMA. D_DATAIN_SIZE_1	0x1408	Input cube's channel
NVDLA_CDMA. D_DATAIN_SIZE_EXT_0	0x1409	Input cube's width and height after extension
NVDLA_CDMA. D_PIXEL_OFFSET	0x140A	For image-in mode, horizontal offset and vertical offset of the 1 st pixel.
NVDLA_CDMA. D_DAIN_RAM_TYPE	0x140B	Ram type of input RAM
NVDLA_CDMA. D_DAIN_ADDR_HIGH_0	0x140C	Higher 32bits of input data address when axi araddr is 64bits
NVDLA_CDMA. D_DAIN_ADDR_LOW_0	0x140D	Lower 32bits of input data address
NVDLA_CDMA. D_DAIN_ADDR_HIGH_1	0x140E	Higher 32bits of input data address of UV plane when axi araddr is 64bits
NVDLA_CDMA. D_DAIN_ADDR_LOW_1	0x140F	Lower 32bits of input data address of UV plane
NVDLA_CDMA. D_LINE_STRIDE	0x1410	Line stride of input cube
NVDLA_CDMA. D_LINE_UV_STRIDE	0x1411	Line stride of input cube's UV plane
NVDLA_CDMA. D_SURF_STRIDE	0x1412	Surface stride of input cube
NVDLA_CDMA. D_DAIN_MAP	0x1413	Whether input cube is line packed or surface packed
NVDLA_CDMA.RESERVED	0x1414	This address is reserved
NVDLA_CDMA.RESERVED	0x1415	This address is reserved
NVDLA_CDMA. D_BATCH_NUMBER	0x1416	Number of batches
NVDLA_CDMA. D_BATCH_STRIDE	0x1417	The stride of input data cubes when batches > 1

NVDLA_CDMA. D_ENTRY_PER_SLICE	0x1418	Number of CBUF entries used for one input slice
NVDLA_CDMA. D_FETCH_GRAIN	0x1419	Number of slices to be fetched before sending update information to CSC
NVDLA_CDMA. D_WEIGHT_FORMAT	0x141A	Whether weight is compressed or not
NVDLA_CDMA. D_WEIGHT_SIZE_0	0x141B	The size of one kernel in bytes
NVDLA_CDMA. D_WEIGHT_SIZE_1	0x141C	Number of kernels
NVDLA_CDMA. D_WEIGHT_RAM_TYPE	0x141D	Ram type of weight
NVDLA_CDMA. D_WEIGHT_ADDR_HIGH	0x141E	Higher 32bits of weight address when axi araddr is 64bits
NVDLA_CDMA. D_WEIGHT_ADDR_LOW	0x141F	Lower 32bits of weight address
NVDLA_CDMA. D_WEIGHT_BYTES	0x1420	Total bytes of Weight
NVDLA_CDMA. D_WGS_ADDR_HIGH	0x1421	Higher 32bits of wgs address when axi araddr is 64bits
NVDLA_CDMA. D_WGS_ADDR_LOW	0x1422	Lower 32bits of wgs address
NVDLA_CDMA. D_WMB_ADDR_HIGH	0x1423	Higher 32bits of wmb address when axi araddr is 64bits
NVDLA_CDMA. D_WMB_ADDR_LOW	0x1424	Lower 32bits of wmb address
NVDLA_CDMA. D_WMB_BYTES	0x1425	Total bytes of WMB
NVDLA_CDMA. D_MEAN_FORMAT	0x1426	Whether mean registers are used or not

NVDLA_CDMA. D_MEAN_GLOBAL_0	0x1427	"Global mean value for red in RGB or Y in YUV Global mean value for green in RGB or U in YUV"
NVDLA_CDMA. D_MEAN_GLOBAL_1	0x1428	"Global mean value for blue in RGB or V in YUV Global mean value for alpha in ARGB/AYUV or X in XRGB"
NVDLA_CDMA. D_CVT_CFG	0x1429	Enable/disable input data converter in CDMA and number of bits to be truncated in the input data converter
NVDLA_CDMA. D_CVT_OFFSET	0x142A	Offset of input data convertor
NVDLA_CDMA. D_CVT_SCALE	0x142B	Scale of input data convertor
NVDLA_CDMA. D_CONV_STRIDE	0x142C	Convolution x stride and convolution y stride
NVDLA_CDMA. D_ZERO_PADDING	0x142D	Left, right,top.bottom padding size
NVDLA_CDMA. D_ZERO_PADDING_VALUE	0x142E	Padding value
NVDLA_CDMA. D_BANK	0x142F	Number of data banks and weight banks in CBUF
NVDLA_CDMA. D_NAN_FLUSH_TO_ZERO	0x1430	Enable/Disable flushing input NaN to zero
NVDLA_CDMA. D_NAN_INPUT_DATA_NUM	0x1431	Count NaN number in input data cube, update per layer
NVDLA_CDMA. D_NAN_INPUT_WEIGHT_NUM	0x1432	Count NaN number in weight kernels, update per layer

NVDLA_CDMA. D_INF_INPUT_DATA_NUM	0x1433	Count infinity number in input data cube, update per layer
NVDLA_CDMA. D_INF_INPUT_WEIGHT_NUM	0x1434	Count infinity number in weight kernels, update per layer
NVDLA_CDMA. D_PERF_ENABLE	0x1435	Enable/disable performance counter
NVDLA_CDMA. D_PERF_DAT_READ_STALL	0x1436	Count blocking cycles of read request of input data, update per layer
NVDLA_CDMA. D_PERF_WT_READ_STALL	0x1437	Count blocking cycles of read request of weight data, update per layer
NVDLA_CDMA. D_PERF_DAT_READ_LATENCY	0x1438	Count total latency cycles of read response of input data, update per layer
NVDLA_CDMA. D_PERF_WT_READ_LATENCY	0x1439	Count total latency cycles of read request of weight data, update per layer
NVDLA_CDMA. D_CYA	0x143A	not use
NVDLA_CSC. S_STATUS	0x1800	Idle status of two register groups
NVDLA_CSC. S_POINTER	0x1801	Pointer for CSB master and data path to access groups
NVDLA_CSC. D_OP_ENABLE	0x1802	Set it to 1 to kick off operation for current register group
NVDLA_CSC. D_MISC_CFG	0x1803	Configuration of operation: convolution mode, precision, weight reuse, data reuse.
NVDLA_CSC. D_DATAIN_FORMAT	0x1804	Input data format and pixel format

NVDLA_CSC. D_DATAIN_SIZE_EXT_0	0x1805	Input cube's width and height after extension
NVDLA_CSC. D_DATAIN_SIZE_EXT_1	0x1806	Input cube's channel after extension
NVDLA_CSC. D_BATCH_NUMBER	0x1807	Number of batches
NVDLA_CSC. D_POST_Y_EXTENSION	0x1808	Post extension parameter for image-in
NVDLA_CSC. D_ENTRY_PER_SLICE	0x1809	Number of CBUF entries used for one input slice
NVDLA_CSC. D_WEIGHT_FORMAT	0x180A	Whether weight is compressed or not
NVDLA_CSC. D_WEIGHT_SIZE_EXT_0	0x180B	Weight's width and height after extension
NVDLA_CSC. D_WEIGHT_SIZE_EXT_1	0x180C	Weight's channel after extension and number of weight kernels
NVDLA_CSC. D_WEIGHT_BYTES	0x180D	Total bytes of Weight
NVDLA_CSC. D_WMB_BYTES	0x180E	Total bytes of WMB
NVDLA_CSC. D_DATAOUT_SIZE_0	0x180F	Output cube's width and height
NVDLA_CSC. D_DATAOUT_SIZE_1	0x1810	Output cube's channel
NVDLA_CSC. D_ATOMICS	0x1811	Equals to output data cube width * output data cube height - 1
NVDLA_CSC. D_RELEASE	0x1812	Slices of CBUF to be released at the end of current layer
NVDLA_CSC. D_CONV_STRIDE_EXT	0x1813	Convolution x stride and convolution y stride after extension
NVDLA_CSC. D_DILATION_EXT	0x1814	Dilation parameter
NVDLA_CSC. D_ZERO_PADDING	0x1815	Left,right,top,bottom padding size

NVDLA_CSC. D_ZERO_PADDING_VALUE	0x1816	Padding value
NVDLA_CSC. D_BANK	0x1817	Number of data banks and weight banks in CBUF
NVDLA_CSC. D_PRA_CFG	0x1818	PRA truncate in Winograd mode, range: 0 2
NVDLA_CMAC_A. S_STATUS	0x1C00	Idle status of two register groups
NVDLA_CMAC_A. S_POINTER	0x1C01	Pointer for CSB master and data path to access groups
NVDLA_CMAC_A. D_OP_ENABLE	0x1C02	Set it to 1 to kick off operation for current register group
NVDLA_CMAC_A. D_MISC_CFG	0x1C03	Configuration of operation: convolution mode, precision, etc.
NVDLA_CMAC_B. S_STATUS	0x2000	Idle status of two register groups
NVDLA_CMAC_B. S_POINTER	0x2001	Pointer for CSB master and data path to access groups
NVDLA_CMAC_B. D_OP_ENABLE	0x2002	Set it to 1 to kick off operation for current register group
NVDLA_CMAC_B. D_MISC_CFG	0x2003	Configuration of operation: convolution mode, precision, etc.
NVDLA_CACC. S_STATUS	0x2400	Idle status of two register groups
NVDLA_CACC. S_POINTER	0x2401	Pointer for CSB master and data path to access groups
NVDLA_CACC. D_OP_ENABLE	0x2402	Set it to 1 to kick off operation for current register group

NVDLA_CACC. D_MISC_CFG	0x2403	Configuration of operation: convolution mode, precision, etc.
NVDLA_CACC. D_DATAOUT_SIZE_0	0x2404	Input cube's width and height after extension
NVDLA_CACC. D_DATAOUT_SIZE_1	0x2405	Input cube's channel after extension
NVDLA_CACC. D_DATAOUT_ADDR	0x2406	Address of output cube
NVDLA_CACC. D_BATCH_NUMBER	0x2407	Number of batches
NVDLA_CACC. D_LINE_STRIDE	0x2408	Line stride of output cube
NVDLA_CACC. D_SURF_STRIDE	0x2409	surface stride of output cube
NVDLA_CACC. D_DATAOUT_MAP	0x240A	Whether output cube is line packed or surface packed
NVDLA_CACC. D_CLIP_CFG	0x240B	Number of bits to be truncated before sending to SDP
NVDLA_CACC. D_OUT_SATURATION	0x240C	Output saturation count
NVDLA_CACC. D_CYA	0x240D	Not use
NVDLA_SDP_RDMA. S_STATUS	0x2800	Idle status of two register groups
NVDLA_SDP_RDMA. S_POINTER	0x2801	Pointer for CSB master and data path to access groups
NVDLA_SDP_RDMA. D_OP_ENABLE	0x2802	Set it to 1 to kick off operation for current register group
NVDLA_SDP_RDMA. D_DATA_CUBE_WIDTH	0x2803	Input cube's width
NVDLA_SDP_RDMA. D_DATA_CUBE_HEIGHT	0x2804	Input cube's height
NVDLA_SDP_RDMA. D_DATA_CUBE_CHANNEL	0x2805	Input cube's channel

NVDLA_SDP_RDMA. D_SRC_BASE_ADDR_LOW	0x2806	Lower 32bits of input data address
NVDLA_SDP_RDMA. D_SRC_BASE_ADDR_HIGH	0x2807	Higher 32bits of input data address when axi araddr is 64bits
NVDLA_SDP_RDMA. D_SRC_LINE_STRIDE	0x2808	Line stride of input cube
NVDLA_SDP_RDMA. D_SRC_SURFACE_STRIDE	0x2809	Surface stride of input cube
NVDLA_SDP_RDMA. D_BRDMA_CFG	0x280A	Configuration of BRDMA: enable/disable, data size, Ram type, etc.
NVDLA_SDP_RDMA. D_BS_BASE_ADDR_LOW	0x280B	Lower 32bits address of the bias data cube
NVDLA_SDP_RDMA. D_BS_BASE_ADDR_HIGH	0x280C	Higher 32bits address of the bias data cube when axi araddr is 64bits
NVDLA_SDP_RDMA. D_BS_LINE_STRIDE	0x280D	Line stride of bias data cube
NVDLA_SDP_RDMA. D_BS_SURFACE_STRIDE	0x280E	Surface stride of bias data cube
NVDLA_SDP_RDMA. D_BS_BATCH_STRIDE	0x280F	Stride of bias data cube in batch mode
NVDLA_SDP_RDMA. D_NRDMA_CFG	0x2810	Configuration of NRDMA: enable/disable, data size, Ram type, etc.
NVDLA_SDP_RDMA. D_BN_BASE_ADDR_LOW	0x2811	Lower 32bits address of the bias data cube
NVDLA_SDP_RDMA. D_BN_BASE_ADDR_HIGH	0x2812	Higher 32bits address of the bias data cube when axi araddr is 64bits
NVDLA_SDP_RDMA. D_BN_LINE_STRIDE	0x2813	Line stride of bias data cube
NVDLA_SDP_RDMA. D_BN_SURFACE_STRIDE	0x2814	Surface stride of bias data cube

NVDLA_SDP_RDMA. D_BN_BATCH_STRIDE	0x2815	Stride of bias data cube in batch mode
NVDLA_SDP_RDMA. D_ERDMA_CFG	0x2816	Configuration of ERDMA: enable/disable, data size, Ram type, etc.
NVDLA_SDP_RDMA. D_EW_BASE_ADDR_LOW	0x2817	Lower 32bits address of the bias data cube
NVDLA_SDP_RDMA. D_EW_BASE_ADDR_HIGH	0x2818	Higher 32bits address of the bias data cube when axi araddr is 64bits
NVDLA_SDP_RDMA. D_EW_LINE_STRIDE	0x2819	Line stride of bias data cube
NVDLA_SDP_RDMA. D_EW_SURFACE_STRIDE	0x281A	Surface stride of bias data cube
NVDLA_SDP_RDMA. D_EW_BATCH_STRIDE	0x281B	Stride of bias data cube in batch mode
NVDLA_SDP_RDMA. D_FEATURE_MODE_CFG	0x281C	Operation configuration: flying mode, output destination, Direct or Winograd mode, flush NaN to zero, batch number.
NVDLA_SDP_RDMA. D_SRC_DMA_CFG	0x281D	RAM type of input data cube
NVDLA_SDP_RDMA. D_STATUS_NAN_INPUT_NUM	0x281E	Input NaN element number
NVDLA_SDP_RDMA. D_STATUS_INF_INPUT_NUM	0x281F	Input Infinity element number
NVDLA_SDP_RDMA. D_PERF_ENABLE	0x2820	Enable/Disable performance counting
NVDLA_SDP_RDMA. D_PERF_MRDMA_READ_STALL	0x2821	Count stall cycles of M read DMA for one layer
NVDLA_SDP_RDMA. D_PERF_BRDMA_READ_STALL	0x2822	Count stall cycles of B read DMA for one layer
NVDLA_SDP_RDMA. D_PERF_NRDMA_READ_STALL	0x2823	Count stall cycles of N read DMA for one layer

NVDLA_SDP_RDMA. D_PERF_ERDMA_READ_STALL	0x2824	Count stall cycles of E read DMA for one layer
NVDLA_SDP. S_STATUS	0x2C00	Idle status of two register groups
NVDLA_SDP. S_POINTER	0x2C01	Pointer for CSB master and data path to access groups
NVDLA_SDP. S_LUT_ACCESS_CFG	0x2C02	LUT access address and type
NVDLA_SDP. S_LUT_ACCESS_DATA	0x2C03	Data register of read or write LUT
NVDLA_SDP. S_LUT_CFG	0x2C04	LUT's type: exponent or linear. And the selection between LE and LO tables.
NVDLA_SDP. S_LUT_INFO	0x2C05	LE and LO LUT index offset and selection
NVDLA_SDP. S_LUT_LE_START	0x2C06	Start of LE LUT's range
NVDLA_SDP. S_LUT_LE_END	0x2C07	End of LE LUT's range
NVDLA_SDP. S_LUT_LO_START	0x2C08	Start of LO LUT's range
NVDLA_SDP. S_LUT_LO_END	0x2C09	End of LO LUT's range
NVDLA_SDP. S_LUT_LE_SLOPE_SCALE	0x2C0A	Slope scale parameter for LE LUT underflow and overflow, signed value
NVDLA_SDP. S_LUT_LE_SLOPE_SHIFT	0x2C0B	Slope shift parameter for LE_LUT underflow and overflow, signed value
NVDLA_SDP. S_LUT_LO_SLOPE_SCALE	0x2C0C	Slope scale parameter for LO LUT underflow and overflow, signed value

NVDLA_SDP. S_LUT_LO_SLOPE_SHIFT	0x2C0D	Slope shift parameter for LO_LUT underflow and overflow, signed value
NVDLA_SDP. D_OP_ENABLE	0x2C0E	Set it to 1 to kick off operation for current register group
NVDLA_SDP. D_DATA_CUBE_WIDTH	0x2C0F	Input to sdp cube's width
NVDLA_SDP. D_DATA_CUBE_HEIGHT	0x2C10	Input to sdp cube's height
NVDLA_SDP. D_DATA_CUBE_CHANNEL	0x2C11	Input to sdp cube's channel
NVDLA_SDP. D_DST_BASE_ADDR_LOW	0x2C12	Lower 32bits of output data address
NVDLA_SDP. D_DST_BASE_ADDR_HIGH	0x2C13	Higher 32bits of output data address when axi awaddr is 64bits
NVDLA_SDP. D_DST_LINE_STRIDE	0x2C14	Line stride of output data cube
NVDLA_SDP. D_DST_SURFACE_STRIDE	0x2C15	Surface stride of output data cube
NVDLA_SDP. D_DP_BS_CFG	0x2C16	Configurations of BS module: bypass, algorithm, etc.
NVDLA_SDP. D_DP_BS_ALU_CFG	0x2C17	Source type and shifter value of BS ALU
NVDLA_SDP. D_DP_BS_ALU_SRC_VALUE	0x2C18	Operand value of BS ALU
NVDLA_SDP. D_DP_BS_MUL_CFG	0x2C19	Source type and shifter value of BS MUL
NVDLA_SDP. D_DP_BS_MUL_SRC_VALUE	0x2C1A	Operand value of BS MUL
NVDLA_SDP. D_DP_BN_CFG	0x2C1B	Configurations of BN module: bypass, algorithm, etc.
NVDLA_SDP. D_DP_BN_ALU_CFG	0x2C1C	Source type and shifter value of BN ALU

NVDLA_SDP. D_DP_BN_ALU_SRC_VALUE	0x2C1D	Operand value of BN ALU
NVDLA_SDP. D_DP_BN_MUL_CFG	0x2C1E	Source type and shifter value of BN MUL
NVDLA_SDP. D_DP_BN_MUL_SRC_VALUE	0x2C1F	Operand value of BN MUL
NVDLA_SDP. D_DP_EW_CFG	0x2C20	Configurations of EW module: bypass, algorithm, etc.
NVDLA_SDP. D_DP_EW_ALU_CFG	0x2C21	Source type and bypass control of EW ALU
NVDLA_SDP. D_DP_EW_ALU_SRC_VALUE	0x2C22	Operand value of EW ALU
NVDLA_SDP. D_DP_EW_ALU_CVT_OFFSET_VALUE	0x2C23	Converter offset of EW ALU
NVDLA_SDP. D_DP_EW_ALU_CVT_SCALE_VALUE	0x2C24	Converter scale of EW ALU
NVDLA_SDP. D_DP_EW_ALU_CVT_TRUNCATE_VALUE	0x2C25	Converter truncate of EW ALU
NVDLA_SDP. D_DP_EW_MUL_CFG	0x2C26	Source type and bypass control of EW MUL
NVDLA_SDP. D_DP_EW_MUL_SRC_VALUE	0x2C27	Operand value of EW MUL
NVDLA_SDP. D_DP_EW_MUL_CVT_OFFSET_VALUE	0x2C28	Converter offset of EW MUL
NVDLA_SDP. D_DP_EW_MUL_CVT_SCALE_VALUE	0x2C29	Converter scale of EW MUL
NVDLA_SDP. D_DP_EW_MUL_CVT_TRUNCATE_VALUE	0x2C2A	Converter truncate of EW MUL
NVDLA_SDP. D_DP_EW_TRUNCATE_VALUE	0x2C2B	Truncate of EW
NVDLA_SDP. D_FEATURE_MODE_CFG	0x2C2C	Operation configuration: flying mode, output destination, Direct or Winograd mode, flush NaN to zero, batch number.
NVDLA_SDP. D_DST_DMA_CFG	0x2C2D	Destination RAM type

NVDLA_SDP. D_DST_BATCH_STRIDE	0x2C2E	Stride of output cubes in batch mode
NVDLA_SDP. D_DATA_FORMAT	0x2C2F	Data precision
NVDLA_SDP. D_CVT_OFFSET	0x2C30	Output converter offset
NVDLA_SDP. D_CVT_SCALE	0x2C31	Output converter scale
NVDLA_SDP. D_CVT_SHIFT	0x2C32	Output converter shifter value
NVDLA_SDP. D_STATUS	0x2C33	Output of equal mode
NVDLA_SDP. D_STATUS_NAN_INPUT_NUM	0x2C34	Input NaN element number
NVDLA_SDP. D_STATUS_INF_INPUT_NUM	0x2C35	Input Infinity element number
NVDLA_SDP. D_STATUS_NAN_OUTPUT_NUM	0x2C36	Output NaN element number
NVDLA_SDP. D_PERF_ENABLE	0x2C37	Enable/Disable performance counting
NVDLA_SDP. D_PERF_WDMA_WRITE_STALL	0x2C38	Count stall cycles of write DMA for one layer
NVDLA_SDP. D_PERF_LUT_UFLOW	0x2C39	Element number of both table underflow
NVDLA_SDP. D_PERF_LUT_OFLOW	0x2C3A	Element number of both table overflow
NVDLA_SDP. D_PERF_OUT_SATURATION	0x2C3B	Element number of both table saturation
NVDLA_SDP. D_PERF_LUT_HYBRID	0x2C3C	Element number of both hit,or both miss situation that element underflow one table and at the same time overflow the other.
NVDLA_SDP. D_PERF_LUT_LE_HIT	0x2C3D	Element number of only LE table hit
NVDLA_SDP. D_PERF_LUT_LO_HIT	0x2C3E	Element number of only LO table hit

NVDLA_PDP_RDMA. S_STATUS	0x3000	Idle status of two register groups
NVDLA_PDP_RDMA. S_POINTER	0x3001	Pointer for CSB master and data path to access groups
NVDLA_PDP_RDMA. D_OP_ENABLE	0x3002	Set it to 1 to kick off operation for current register group
NVDLA_PDP_RDMA. D_DATA_CUBE_IN_WIDTH	0x3003	Input data cube's width
NVDLA_PDP_RDMA. D_DATA_CUBE_IN_HEIGHT	0x3004	Input data cube's height
NVDLA_PDP_RDMA. D_DATA_CUBE_IN_CHANNEL	0x3005	Input data cube's channel
NVDLA_PDP_RDMA. D_FLYING_MODE	0x3006	Indicate source is SDP or external memory
NVDLA_PDP_RDMA. D_SRC_BASE_ADDR_LOW	0x3007	Lower 32bits of input data address
NVDLA_PDP_RDMA. D_SRC_BASE_ADDR_HIGH	0x3008	Higher 32bits of input data address when axi araddr is 64bits
NVDLA_PDP_RDMA. D_SRC_LINE_STRIDE	0x3009	Line stride of input cube
NVDLA_PDP_RDMA. D_SRC_SURFACE_STRIDE	0x300A	Surface stride of input cube
NVDLA_PDP_RDMA. D_SRC_RAM_CFG	0x300B	RAM type of input data cube
NVDLA_PDP_RDMA. D_DATA_FORMAT	0x300C	Input data cube
NVDLA_PDP_RDMA. D_OPERATION_MODE_CFG	0x300D	Split number
NVDLA_PDP_RDMA. D_POOLING_KERNEL_CFG	0x300E	Kernel width and kernel stride
NVDLA_PDP_RDMA. D_POOLING_PADDING_CFG	0x300F	Padding width
NVDLA_PDP_RDMA. D_PARTIAL_WIDTH_IN	0x3010	Partial width for first, last and middle partitions

NVDLA_PDP_RDMA. D_PERF_ENABLE	0x3011	Enable/Disable performance counting
NVDLA_PDP_RDMA. D_PERF_READ_STALL	0x3012	Element number that for both LUT underflow.
NVDLA_PDP. S_STATUS	0x3400	Idle status of two register groups
NVDLA_PDP. S_POINTER	0x3401	Pointer for CSB master and data path to access groups
NVDLA_PDP. D_OP_ENABLE	0x3402	Set it to 1 to kick off operation for current register group
NVDLA_PDP. D_DATA_CUBE_IN_WIDTH	0x3403	Input data cube's width
NVDLA_PDP. D_DATA_CUBE_IN_HEIGHT	0x3404	Input data cube's height
NVDLA_PDP. D_DATA_CUBE_IN_CHANNEL	0x3405	Input data cube's channel
NVDLA_PDP. D_DATA_CUBE_OUT_WIDTH	0x3406	Output data cube's width
NVDLA_PDP. D_DATA_CUBE_OUT_HEIGHT	0x3407	Output data cube's height
NVDLA_PDP. D_DATA_CUBE_OUT_CHANNEL	0x3408	Output data cube's channel
NVDLA_PDP. D_OPERATION_MODE_CFG	0x3409	pooling method,flying mod,split number
NVDLA_PDP. D_NAN_FLUSH_TO_ZERO	0x340A	Option to flush input NaN to zero
NVDLA_PDP. D_PARTIAL_WIDTH_IN	0x340B	Partial width for first, last and middle partitions of input cube
NVDLA_PDP. D_PARTIAL_WIDTH_OUT	0x340C	Partial width for first, last and middle partitions of output cube
NVDLA_PDP. D_POOLING_KERNEL_CFG	0x340D	Kernel width and kernel stride
NVDLA_PDP. D_RECIP_KERNEL_WIDTH	0x340E	Reciprocal of pooling kernel width

NVDLA_PDP. D_RECIP_KERNEL_HEIGHT	0x340F	Reciprocal of pooling kernel height
NVDLA_PDP. D_POOLING_PADDING_CFG	0x3410	Left/right/top/bottom padding size
NVDLA_PDP. D_POOLING_PADDING_VALUE_1_CFG	0x3411	Padding_value*1
NVDLA_PDP. D_POOLING_PADDING_VALUE_2_CFG	0x3412	Padding_value*2
NVDLA_PDP. D_POOLING_PADDING_VALUE_3_CFG	0x3413	Padding_value*3
NVDLA_PDP. D_POOLING_PADDING_VALUE_4_CFG	0x3414	Padding_value*4
NVDLA_PDP. D_POOLING_PADDING_VALUE_5_CFG	0x3415	Padding_value*5
NVDLA_PDP. D_POOLING_PADDING_VALUE_6_CFG	0x3416	Padding_value*6
NVDLA_PDP. D_POOLING_PADDING_VALUE_7_CFG	0x3417	Padding_value*7
NVDLA_PDP. D_SRC_BASE_ADDR_LOW	0x3418	Lower 32bits of input data address
NVDLA_PDP. D_SRC_BASE_ADDR_HIGH	0x3419	Higher 32bits of input data address when axi araddr is 64bits
NVDLA_PDP. D_SRC_LINE_STRIDE	0x341A	Line stride of input cube
NVDLA_PDP. D_SRC_SURFACE_STRIDE	0x341B	Surface stride of input cube
NVDLA_PDP. D_DST_BASE_ADDR_LOW	0x341C	Lower 32bits of output data address
NVDLA_PDP. D_DST_BASE_ADDR_HIGH	0x341D	Higher 32bits of output data address when axi awaddr is 64bits
NVDLA_PDP. D_DST_LINE_STRIDE	0x341E	Line stride of output cube
NVDLA_PDP. D_DST_SURFACE_STRIDE	0x341F	Surface stride of output cube
NVDLA_PDP. D_DST_RAM_CFG	0x3420	RAM type of destination cube

NVDLA_PDP. D_DATA_FORMAT	0x3421	Precision of input data
NVDLA_PDP. D_INF_INPUT_NUM	0x3422	Input infinity element number
NVDLA_PDP. D_NAN_INPUT_NUM	0x3423	Input NaN element number
NVDLA_PDP. D_NAN_OUTPUT_NUM	0x3424	Output NaN element number
NVDLA_PDP. D_PERF_ENABLE	0x3425	Enable/disable performance counting
NVDLA_PDP. D_PERF_WRITE_STALL	0x3426	Counting stalls of write requests
NVDLA_CDP_RDMA. S_STATUS	0x3800	Idle status of two register groups
NVDLA_CDP_RDMA. S_POINTER	0x3801	Pointer for CSB master and data path to access groups
NVDLA_CDP_RDMA. D_OP_ENABLE	0x3802	Set it to 1 to kick off operation for current register group
NVDLA_CDP_RDMA. D_DATA_CUBE_WIDTH	0x3803	Input data cube's width
NVDLA_CDP_RDMA. D_DATA_CUBE_HEIGHT	0x3804	Input data cube's height
NVDLA_CDP_RDMA. D_DATA_CUBE_CHANNEL	0x3805	Input data cube's channel
NVDLA_CDP_RDMA. D_SRC_BASE_ADDR_LOW	0x3806	Lower 32bits of input data address
NVDLA_CDP_RDMA. D_SRC_BASE_ADDR_HIGH	0x3807	Higher 32bits of input data address when axi araddr is 64bits
NVDLA_CDP_RDMA. D_SRC_LINE_STRIDE	0x3808	Line stride of input cube
NVDLA_CDP_RDMA. D_SRC_SURFACE_STRIDE	0x3809	Surface stride of input cube
NVDLA_CDP_RDMA. D_SRC_DMA_CFG	0x380A	RAM type of input data cube
NVDLA_CDP_RDMA. D_SRC_COMPRESSION_EN	0x380B	This register is not used in OpenDLA 1.0

NVDLA_CDP_RDMA. D_OPERATION_MODE	0x380C	Split number
NVDLA_CDP_RDMA. D_DATA_FORMAT	0x380D	Input data cube
NVDLA_CDP_RDMA. D_PERF_ENABLE	0x380E	Enable/Disable performance counting
NVDLA_CDP_RDMA. D_PERF_READ_STALL	0x380F	Counting stalls of read requests
NVDLA_CDP. S_STATUS	0x3C00	Idle status of two register groups
NVDLA_CDP. S_POINTER	0x3C01	Pointer for CSB master and data path to access groups
NVDLA_CDP. S_LUT_ACCESS_CFG	0x3C02	LUT access address and type
NVDLA_CDP. S_LUT_ACCESS_DATA	0x3C03	Data register of read or write LUT
NVDLA_CDP. S_LUT_CFG	0x3C04	LUT's type: exponent or linear. And the selection between LE and LO tables.
NVDLA_CDP. S_LUT_INFO	0x3C05	LE and LO LUT index offset and selection
NVDLA_CDP. S_LUT_LE_START_LOW	0x3C06	Lower 32bits of start of LE LUT's range
NVDLA_CDP. S_LUT_LE_START_HIGH	0x3C07	Higher 6bits of start of LE LUT's range
NVDLA_CDP. S_LUT_LE_END_LOW	0x3C08	Lower 32bits of end of LE LUT's range
NVDLA_CDP. S_LUT_LE_END_HIGH	0x3C09	Higher 6bits of end of LE LUT's range
NVDLA_CDP. S_LUT_LO_START_LOW	0x3C0A	Lower 32bits of start of LO LUT's range
NVDLA_CDP. S_LUT_LO_START_HIGH	0x3C0B	Higher 6bits of start of LO LUT's range
NVDLA_CDP. S_LUT_LO_END_LOW	0x3C0C	Lower 32bits of end of LO LUT's range
NVDLA_CDP. S_LUT_LO_END_HIGH	0x3C0D	Higher 6bits of end of LO LUT's range

NVDLA_CDP. S_LUT_LE_SLOPE_SCALE	0x3C0E	Slope scale parameter for LE LUT underflow and overflow, signed value
NVDLA_CDP. S_LUT_LE_SLOPE_SHIFT	0x3C0F	Slope shift parameter for LE_LUT underflow and overflow, signed value
NVDLA_CDP. S_LUT_LO_SLOPE_SCALE	0x3C10	Slope scale parameter for LO LUT underflow and overflow, signed value
NVDLA_CDP. S_LUT_LO_SLOPE_SHIFT	0x3C11	Slope shift parameter for LO_LUT underflow and overflow, signed value
NVDLA_CDP. D_OP_ENABLE	0x3C12	Set it to 1 to kick off operation for current register group
NVDLA_CDP. D_FUNC_BYPASS	0x3C13	Square sum process bypass control and multiplier after interpolator bypass control
NVDLA_CDP. D_DST_BASE_ADDR_LOW	0x3C14	Lower 32bits of output data address
NVDLA_CDP. D_DST_BASE_ADDR_HIGH	0x3C15	Higher 32bits of output data address when axi awaddr is 64bits
NVDLA_CDP. D_DST_LINE_STRIDE	0x3C16	Line stride of output cube
NVDLA_CDP. D_DST_SURFACE_STRIDE	0x3C17	Surface stride of output cube
NVDLA_CDP. D_DST_DMA_CFG	0x3C18	RAM type of output data cube
NVDLA_CDP. D_DST_COMPRESSION_EN	0x3C19	This register is not used in OpenDLA 1.0
NVDLA_CDP. D_DATA_FORMAT	0x3C1A	Precision of input data

NVDLA_CDP. D_NAN_FLUSH_TO_ZERO	0x3C1B	Option to flush input NaN to zero
NVDLA_CDP. D_LRN_CFG	0x3C1C	Normalization length
NVDLA_CDP. D_DATIN_OFFSET	0x3C1D	Input data convertor offset
NVDLA_CDP. D_DATIN_SCALE	0x3C1E	Input data convertor scale
NVDLA_CDP. D_DATIN_SHIFTER	0x3C1F	Input data convertor shifter value
NVDLA_CDP. D_DATOUT_OFFSET	0x3C20	Output data convertor offset
NVDLA_CDP. D_DATOUT_SCALE	0x3C21	Output data convertor scale
NVDLA_CDP. D_DATOUT_SHIFTER	0x3C22	Output data convertor shifter value
NVDLA_CDP. D_NAN_INPUT_NUM	0x3C23	input NaN element number
NVDLA_CDP. D_INF_INPUT_NUM	0x3C24	input Infinity element number
NVDLA_CDP. D_NAN_OUTPUT_NUM	0x3C25	output NaN element number
NVDLA_CDP. D_OUT_SATURATION	0x3C26	saturated element number.
NVDLA_CDP. D_PERF_ENABLE	0x3C27	Enable/Disable performance counting
NVDLA_CDP. D_PERF_WRITE_STALL	0x3C28	Element number that for both LUT underflow
NVDLA_CDP. D_PERF_LUT_UFLOW	0x3C29	Element number that for both LUT underflow
NVDLA_CDP. D_PERF_LUT_OFLOW	0x3C2A	Element number that for both LUT over-flow
NVDLA_CDP. D_PERF_LUT_HYBRID	0x3C2B	Element number that for both LUT miss, one is over-flow and the other is overflow

NVDLA_CDP. D_PERF_LUT_LE_HIT	0x3C2C	Element number that for LE_lut hit only
NVDLA_CDP. D_PERF_LUT_LO_HIT	0x3C2D	Element number that for LO_lut hit only
NVDLA_RUBIK. S_STATUS	0x4000	Idle status of two register groups
NVDLA_RUBIK. S_POINTER	0x4001	Pointer for CSB master and data path to access groups
NVDLA_RUBIK. D_OP_ENABLE	0x4002	Set it to 1 to kick off operation for current register group
NVDLA_RUBIK. D_MISC_CFG	0x4003	Operation mode and precision
NVDLA_RUBIK. D_DAIN_RAM_TYPE	0x4004	RAM type of input cube
NVDLA_RUBIK. D_DATAIN_SIZE_0	0x4005	Input data cube's width and height
NVDLA_RUBIK. D_DATAIN_SIZE_1	0x4006	Input data cube's channel
NVDLA_RUBIK. D_DAIN_ADDR_HIGH	0x4007	Higher 32bits of input data address when axi araddr is 64bits
NVDLA_RUBIK. D_DAIN_ADDR_LOW	0x4008	Lower 32bits of input data address
NVDLA_RUBIK. D_DAIN_LINE_STRIDE	0x4009	Line stride of input data cube
NVDLA_RUBIK. D_DAIN_SURF_STRIDE	0x400A	Surface stride of input data cube
NVDLA_RUBIK. D_DAIN_PLANAR_STRIDE	0x400B	Input data planar stride, for merge mode only
NVDLA_RUBIK. D_DAOUT_RAM_TYPE	0x400C	RAM type of output cube
NVDLA_RUBIK. D_DATAOUT_SIZE_1	0x400D	Output data cube's channel
NVDLA_RUBIK. D_DAOUT_ADDR_HIGH	0x400E	Higher 32bits of output data address when axi awaddr is 64bits

NVDLA_RUBIK. D_DAOUT_ADDR_LOW	0x400F	Lower 32bits of output data address
NVDLA_RUBIK. D_DAOUT_LINE_STRIDE	0x4010	Line stride of output data cube
NVDLA_RUBIK. D_CONTRACT_STRIDE_0	0x4011	Input stride for each X step.
NVDLA_RUBIK. D_CONTRACT_STRIDE_1	0x4012	Output stride corresponding to each line in input cube
NVDLA_RUBIK. D_DAOUT_SURF_STRIDE	0x4013	Surface stride of output data cube
NVDLA_RUBIK. D_DAOUT_PLANAR_STRIDE	0x4014	Output data planar stride, for split mode only
NVDLA_RUBIK. D_DECONV_STRIDE	0x4015	Deconvolution x stride and y stride
NVDLA_RUBIK. D_PERF_ENABLE	0x4016	Enable/Disable performance counting
NVDLA_RUBIK. D_PERF_READ_STALL	0x4017	RD_STALL Count stall cycles of read DMA for one layer
NVDLA_RUBIK. D_PERF_WRITE_STALL	0x4018	WR_STALL Count stall cycles of write DMA for one layer

Bibliography

[1] Subhasish Mitra et al. «The resilience wall: Cross-layer solution strategies». In: *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*. 2014, pp. 1–11. DOI: 10.1109/VLSI-DAT.2014.6834933 (cit. on p. 2).

[2] Yanjing Li, Samy Makar, and Subhasish Mitra. «CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns». In: *2008 Design, Automation and Test in Europe*. 2008, pp. 885–890. DOI: 10.1109/DATE.2008.4484786 (cit. on pp. 2, 21).

[3] Yanjing Li, Onur Mutlu, Donald S. Gardner, and Subhasish Mitra. «Concurrent autonomous self-test for uncore components in system-on-chips». In: *2010 28th VLSI Test Symposium (VTS)*. 2010, pp. 232–237. DOI: 10.1109/VTS.2010.5469571 (cit. on p. 2).

[4] G. Tshagharyan, G. Harutyunyan, and Y. Zorian. «An effective functional safety solution for automotive systems-on-chip». In: *2017 IEEE International Test Conference (ITC)*. 2017, pp. 1–10. DOI: 10.1109/TEST.2017.8242075 (cit. on p. 2).

[5] Sugako Otani et al. «2.7 A 28nm 600MHz Automotive Flash Microcontroller with Virtualization-Assisted Processor for Next-Generation Automotive Architecture Complying with ISO26262 ASIL-D». In: *2019 IEEE International Solid-State Circuits Conference - (ISSCC)*. 2019, pp. 54–56. DOI: 10.1109/ISSCC.2019.8662300 (cit. on p. 2).

[6] Nilanjan Mukherjee et al. «Test Time and Area Optimized BrST Scheme for Automotive ICs». In: *2019 IEEE International Test Conference (ITC)*. 2019, pp. 1–10. DOI: 10.1109/ITC44170.2019.9000133 (cit. on p. 2).

[7] Pavan Kumar Datla Jagannadha et al. «Special Session: In-System-Test (IST) Architecture for NVIDIA Drive-AGX Platforms». In: *2019 IEEE 37th VLSI Test Symposium (VTS)*. 2019, pp. 1–8. DOI: 10.1109/VTS.2019.8758636 (cit. on p. 2).

[8] Geoffrey E. Hinton Vinod Nair. «Rectified Linear Units Improve Restricted Boltzmann Machines». In: 2010 (cit. on p. 6).

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification». In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123 (cit. on p. 6).

[10] URL: http://nvdla.org/ (cit. on p. 11).

[11] Takumi Uezono, Yi He, and Yanjing Li. «Achieving Automotive Safety Requirements through Functional In-Field Self-Test for Deep Learning Accelerators». In: *2022 IEEE International Test Conference (ITC)*. 2022, pp. 465–473. DOI: 10.1109/ITC50671.2022.00054 (cit. on p. 23).

[12] Yi He, Takumi Uezono, and Yanjing Li. «Efficient Functional In-Field Self-Test for Deep Learning Accelerators». In: *2021 IEEE International Test Conference (ITC)*. 2021, pp. 93–102. DOI: 10.1109/ITC50571.2021.00017 (cit. on p. 24).