T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

# 論文 / 著書情報
# Article / Book Information

| | |
|---|---|
| 題目(和文) | RISC-VをベースとしたAI画像認識システムのための特定用途向け命令セットプロセッサ |
| Title(English) | Application-Specific Instruction-set Processor for AI Visual System based on RISC-V |
| 著者(和文) | 汪漢森 |
| Author(English) | Hansen Wang |
| 出典(和文) | 学位:博士(工学),<br>学位授与機関:東京工業大学,<br>報告番号:甲第12861号,<br>授与年月日:2024年9月20日,<br>学位の種別:課程博士,<br>審査員:一色 剛,髙橋 篤司,本村 真人,佐々木 広,藤木 大地 |
| Citation(English) | Degree:Doctor (Engineering),<br>Conferring organization: Tokyo Institute of Technology,<br>Report number:甲第12861号,<br>Conferred date:2024/9/20,<br>Degree Type:Course doctor,<br>Examiner:,,,, |
| 学位種別(和文) | 博士論文 |
| Type(English) | Doctoral Thesis |

Powered by T2R2 (Science Tokyo Research Repository)

Doctoral Dissertation

# Application-Specific Instruction-set Processor for AI Visual System based on RISC-V

Hansen Wang

Department Information and Communications Engineering,
Tokyo Institute of Technology

Supervisor: Professor Tsuyoshi Isshiki

June 2024

# Abstract

This article introduces a programmable Artificial Intelligence Visual System on Chip (AIV-SoC) featuring an application-specific instruction set, optimized for deploying various AI models, including YOLOv8, RESNET, and VGG. For YOLOv8, the system processes input images at a resolution of 352×352 to perform object detection, instance segmentation, and pose detection, achieving impressive Frames per Second (FPS) rates of 67.1, 55.2, and 64.9, respectively. Remarkably, the AIV-SoC demonstrates superior power efficiency compared to other GPU and FPGA platforms, underscoring its practicality for various industries. The AIV-SoC's efficiency makes it particularly suitable for deployment in resource-constrained environments such as healthcare, agriculture, and surveillance. This capability redefines the landscape of computer vision in real-world scenarios, suggesting a future where these technologies play a pivotal role in transforming our interactions with machines and our interpretation of visual information.

# Acknowledgments

First and foremost, I would like to express my deepest gratitude to Professor Tsuyoshi Isshiki for his invaluable guidance, support, and encouragement throughout the course of my research. His insights and expertise have been instrumental in shaping the direction and quality of this work. I am also profoundly grateful to Assistant Professor Dongju Li, whose assistance and advice have been crucial at various stages of my research. Her constructive feedback and willingness to help have significantly contributed to the completion of this thesis.

I would like to extend my heartfelt thanks to all my fellow students. Their camaraderie, collaboration, and support have made this journey much more enjoyable and enriching. The numerous discussions and shared experiences have been a source of great learning and motivation.

Furthermore, I am immensely thankful for the financial support provided by the New Energy and Industrial Technology Development Organization (NEDO), JPNP23015 project. This funding has been essential in enabling me to pursue and complete this research.

Thank you all for your invaluable contributions and unwavering support.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Computer Vision & Artificial Intelligence

Object detection, a pivotal computer vision technology, is dedicated to discerning instances of specific semantic objects within digital images and videos [1]. Its versatile applications span a spectrum of computer vision tasks, encompassing image annotation and retrieval, vehicle counting, activity recognition, face detection and video object co-segmentation. Moreover, object detection plays a pivotal role in autonomous driving systems [2].

In conjunction with object detection, instance segmentation represents a nuanced extension of this field. Unlike object detection, which localizes objects with bounding boxes, instance segmentation delves further into the delineation of object boundaries at the pixel level. In comparison to semantic segmentation, the task of instance segmentation is notably more fine-grained, intricate, and demanding [3]. Noteworthy methodologies, such as Mask R-CNN [4], initially perform object detection and subsequently assign a binary label to each pixel within the bounding box. The work presented in [5] seamlessly integrates with cutting-edge one-stage detection frameworks, surpassing Mask R-CNN under the same training schedule and achieving faster processing times.

The synergistic integration of object detection and instance segmentation has found applications in various domains, including scene understanding for robotics [6], medical image analysis for precise organ delineation [7][8], video surveillance, augmented reality [9], and image compression. This amalgamation enhances the granularity of information extraction, proving particularly valuable in scenarios where object boundaries and spatial relationships play a pivotal role.

Pose detection, another key task in computer vision, involves estimating the spatial arrangement of key body joints or keypoints within an image or video. This critical field plays a pivotal role in diverse applications such as human-computer interaction, surveillance, and augmented reality. The extracted pose information is instrumental in tasks ranging from anomaly detection in surveillance systems to enhancing virtual try-on experiences in e-commerce [10][11]. Additionally, pose detection contributes significantly to sports analytics, facilitating precise motion tracking for athletes and aiding in skill refinement and training optimization [12].

## 1.2 Single Instruction Multiple Data (SIMD) Devices for AI Acceleration

The conventional approach to AI model implementation involves utilizing GPUs along with their corresponding APIs, such as Compute Unified Device Architecture (CUDA) [13]. However, GPUs have the disadvantage of high power consumption compared with FPGAs and Application-Specific Integrated Circuits (ASICs). This makes GPUs unsuitable for applications with low power requirements, such as edge and IoT devices.

FPGAs are hardware devices that offer configuration to execute a wide range of digital functions, including CNN inference tasks. They allow for rapid configuration of hardware resources to adapt to various neural network architectures. FPGAs also show promise in power efficiency, attributed to their ability to instantiate only the necessary computational elements for a specific task, resulting in lower power consumption compared to general-purpose GPUs. Recently, several C-based High-Level Synthesis (HLS) tools for FPGA development have emerged [14, 15]. However, FPGA programming and optimization necessitate specialized skills, requiring users to invest significant time in tuning performance directives, thus making development more complex compared to CUDA-based GPU programming. Additionally, it's important to note that the design flexibility of FPGAs is somewhat constrained by their fixed hardware resources, lacking the versatility of some other platforms.

ASICs, as the name implies, are integrated circuits meticulously designed for specific applications or tasks. ASICs are tailored to execute predetermined functions with higher efficiency than FPGAs and GPUs, making them an ideal choice for applications prioritizing performance, power efficiency, and miniaturization. In CNN applications, convolution, a key neural network layer, primarily consists of regular and predictable multiplication and addition operations. Therefore, we propose the utilization of ASICs to implement CNNs, aiming for high efficiency and low power consumption.

However, ASICs come with their own set of limitations. The design and fabrication of ASICs require significant time and resources, making them less flexible for rapidly evolving applications. Additionally, their lack of versatility can be a drawback, as ASICs excel only in their designated roles. To mitigate these drawbacks, we introduce the RISC-V architecture as a fundamental platform, leveraging its modular design to enhance flexibility.

## 1.3 Contributions of the Thesis

The major contributions of our RISC-V custom instruction-driven accelerator architecture are summarized as follows:

- **Dedicated on-chip high bandwidth memory architecture for sustaining high utilization of MAC (multiply-accumulate) compute resources:** This architecture ensures that data is readily available for computation, minimizing idle times and maximizing the efficiency of the MAC units. By maintaining a high throughput of data, the architecture sustains peak performance levels, crucial for handling the demanding computations of CNNs like YOLOv8, RESNET, and VGG.

- **Dedicated on-chip DMA controller for hiding data transfer cycles behind AI compute cycles:** The on-chip DMA controller effectively overlaps data transfer

operations with computation cycles. This means that while the accelerator performs computations, the DMA controller simultaneously prepares the next set of data, thereby minimizing latency and ensuring a seamless and continuous data flow. This approach significantly boosts overall system efficiency and performance.

- **RISC-V custom instructions for controlling the accelerator hardware:** Custom instructions tailored to the RISC-V architecture allow for direct and efficient control of the accelerator hardware. These instructions eliminate the need for complex hardware drivers, simplifying the programming model and making it easier for software developers to leverage the accelerator's capabilities. This direct control streamlines the integration process and enhances the usability and flexibility of the system.

These contributions address several key challenges faced by AI accelerator devices. The high bandwidth memory architecture and DMA controller work together to mitigate data transfer bottlenecks, ensuring that the computational units are fully utilized without being stalled by data movement delays. The custom RISC-V instructions simplify the control and programming of the hardware, reducing development time and complexity.

The impact of these contributions is significant: they enhance the energy efficiency and performance of the accelerator, making it ideal for low-power, high-performance AI applications. The architecture achieves superior power efficiency by reducing the energy and time lost during data transfer compared to traditional CPU + FPGA, GPU, or ASIC solutions. This efficiency is particularly beneficial for deploying AI models in resource-constrained environments, such as edge devices and low-power terminals, ultimately enabling more widespread and effective use of AI technologies.

## 1.4 Structure of the Thesis

In this paper, we first introduce a novel basic design called DNN-AS, which we then upgrade to the Artificial Intelligence Visual System on Chip (AIV-SoC). The AIV-SoC emphasizes the use of Int8 precision and a 352×352 input resolution. Our contribution lies in the introduction of an innovative and adaptable circuit tailored for the integration of new, computationally intensive layers in YOLOv8, such as SiLU and softmax. Leveraging advanced software programming techniques, the AIV-SoC demonstrates remarkable proficiency in efficiently executing the complete end-to-end processes of object detection, instance segmentation, and pose detection tasks.

The structure of the paper is organized as follows: Section 2 provides an overview of relevant works. Section 3 describes the Deep Neural Network accelerator SoC (DNN-AS) presented in [16] and [18], including the experimental results of RESNET. In Section 4, our works [17] and [19] extend the DNN-AS to the Artificial Intelligence Visual System on Chip (AIV-SoC), addressing more functionalities of YOLOv8. Section 5 explains the programming approach for different models and layers. Finally, Section 6 summarizes the strengths and weaknesses of our AIV-SoC and suggests potential future work.

# Chapter 2

# Related Works

## 2.1 RISC-V Architecture

The RISC-V Instruction Set Architecture (ISA) was initially developed at the University of California, Berkeley, in 2010 [20]. Over time, it has garnered substantial recognition within both academic and industrial circles. The ISA holds a pivotal role in the realm of Central Processing Units (CPUs) as it serves as the linchpin connecting hardware and software. It accomplishes this by translating high-level software constructs into low-level instructions that the CPU can execute, in tandem with compilers. In comparison to other architectural designs, RISC-V boasts several advantages that render it exceptionally conducive to development.

One of its foremost strengths lies in its open-source nature, affording any interested party unrestricted access to the core's source intellectual property (IP) without the encumbrance of licensing issues. While the ISA's development adheres to open-source principles, its foundational features have been meticulously defined and stabilized. This, in turn, beckons software development endeavors. Additionally, RISC-V lends itself to the seamless integration of supplementary functionalities through a set of extensions, which, too, are exhaustively delineated upon stabilization.

The modular composition of the ISA renders it amenable to a diverse spectrum of applications, spanning both high-performance and low-power integrated circuit scenarios. Furthermore, it simplifies the creation of application-specific processors equipped with dedicated accelerators. These attributes grant developers the freedom to craft their own cores in alignment with the RISC-V ISA guidelines.

## 2.2 Cadence AI Platform

Recent advancements in artificial intelligence (AI) have significantly impacted various fields, including electronic design automation (EDA). Cadence Design Systems [21] has been at the forefront of integrating AI into EDA, offering a comprehensive AI platform that enhances chip design through generative AI and big data analytics. Their Cadence.AI Generative AI Platform is particularly notable for its ability to optimize design performance and improve engineering productivity by leveraging vast amounts of data throughout the design process.

The Cadence Cerebrus platform, part of this AI suite, focuses on digital design optimization, enabling engineers to achieve better performance with less manual intervention. Additionally, the Allegro X AI extends these benefits to printed circuit board (PCB) design,

automating tasks such as component placement and routing, thus reducing time-to-market and improving overall design efficiency. By comparing these AI-driven solutions to traditional methods, it is evident that Cadence's AI integration significantly enhances the design workflow, making it a pivotal development in the field of EDA.

## 2.3 Esperanto ET-SoC-1 Chip

Esperanto Technologies developed the ET-SoCC-1 chip [22], integrating over 1,000 low-power RISC-V processors on a single chip to enhance ML recommendation workloads. This chip, fabricated with TSMC's 7-nm technology, features over 24 billion transistors and includes three types of RISC-V-compatible processors: a single 64-bit service processor for system boot, 1,088 64-bit ET-Minion processors for ML computation, and four 64-bit ET-Maxion processors for high-performance tasks. It includes over 160 million bytes of on-die memory and LPDDR4x DRAM controllers supporting up to 32 GB of external memory. The ET-SoC-1 achieves peak computation rates between 100 and 200 TOPS while operating under 20 W of power, making it highly efficient for various workloads.

Traditional x86 servers allocate a PCIe slot for accelerator cards, which must outperform the host CPU within a 75-120 W power budget. Esperanto's ET-SoC-1 shows substantial improvements in performance per watt compared to conventional server platforms. Preliminary data suggests that Esperanto-based accelerator cards could achieve over a hundred times better performance per watt for the MLPerf Deep Learning Recommendation Model benchmark compared to standard server platforms.

Esperanto operates its ET-Minion cores at low voltages to enhance energy efficiency significantly. At around 0.3 V, each chip consumes only 8.5 W, allowing up to six chips to function within a 120-W power budget. This setup provides a performance boost approximately 2.5 times higher and an energy efficiency 20 times better than a single-chip solution at high voltage. The scalability of the ET-SoC-1 chip is also critical. By distributing processing and I/O across multiple chips, the solution can effectively scale performance, memory capacity, and bandwidth by adding more chips, using low-cost, low-power LPDDR4X DRAM instead of more expensive, high-power alternatives.

Esperanto's low-power technology allows multiple ET-SoC-1 chips within a PCIe card slot's power constraints. A single accelerator card can house six Esperanto chips and 192 GB of DRAM, achieving up to 836 TOPS (INT8) peak performance when all ET-Minions operate at 1 GHz. This configuration enables deployment in large data centers, potentially involving millions of accelerator chips across thousands of racks.

In summary, Esperanto Technologies' ET-SoC-1 chip advances ML recommendation workloads with its innovative use of low-power RISC-V processors, scalable architecture, and efficient low-voltage operation, positioning it as a highly effective solution for modern data centers and ML applications.

## 2.4 Graphcore Intelligence Processing Unit (IPU)

The Graphcore Intelligence Processing Unit (IPU) [23] architecture presents a novel approach to AI computing, distinct from traditional GPU and TPU architectures. One of the key features of the IPU is its use of a large amount of on-die SRAM, offering 896MiB with a

bandwidth of 47TB/s, which allows for unprecedented access to arbitrarily-structured data. This design choice addresses the challenge of data transport energy consumption by minimizing it, thus enabling more processor silicon to be deployed within a power budget. Additionally, it reduces the memory cost for AI models, allowing for a more cost-effective deployment of processor silicon.

Graphcore's IPU architecture supports fine-grained parallelism, which is crucial for future AI models and data. The architecture includes features such as native graph abstraction, codelet-level parallelism, and pipeline-oblivious threads. These features collectively enable rapid software evolution and eliminate concurrency hazards. The Bulk Synchronous Parallel (BSP) model employed by the IPU ensures that tile processors execute asynchronously until data exchange is necessary, synchronizing globally in approximately 150 cycles on-chip.

The IPU's memory model diverges from the high-bandwidth memory (HBM) approach used by GPUs and TPUs, which is typically expensive and thermally demanding. Instead, the IPU utilizes a combination of on-die SRAM for bandwidth and off-chip DDR for capacity, offering a more economical and scalable solution. This combination allows for efficient handling of model states and optimizer states, supporting various model scales without the need for excessive off-chip memory bandwidth.

Graphcore's approach also includes innovative mechanisms for random number generation and stochastic rounding, which are vital for efficient training of models using reduced precision formats like f16. Each tile in the IPU can generate 128 random bits per cycle, using an enhanced xoroshiro128+ PRNG and a 6th-order Irwin Hall Gaussian shaper.

Overall, the Graphcore IPU represents a significant advancement in AI processor design, providing a flexible, efficient, and scalable architecture that is well-suited for the evolving demands of AI workloads.

## 2.5 Cerebras Architecture Deep Dive

In the realm of deep learning hardware and software co-design, Cerebras [24] Systems has established a formidable presence with its innovative approach to amplifying Moore's Law through the development of its Wafer-Scale Engine (WSE). The company's architectural deep dive, as elucidated in their blog post titled "Cerebras Architecture Deep Dive: First Look Inside the HW/SW Co-Design for Deep Learning," provides a comprehensive overview of their cutting-edge technology.

The traditional scaling paradigm within semiconductors, governed by Moore's Law, has predominantly focused on increasing the density of transistors on a chip. However, Cerebras has transcended this incremental approach by designing the second-generation WSE-2, which stands as the largest chip ever constructed. This chip, boasting an unprecedented 56 times larger surface area compared to the largest CPUs and housing 2.6 trillion transistors, integrates 850,000 cores on a single piece of silicon. This monumental feat is underpinned by a specially designed system, the Cerebras CS-2, which facilitates the utilization of the wafer-scale chip in a standard datacenter environment.

The architectural cornerstone of the WSE-2 is its fabric, a 2D mesh topology that enables efficient and high-performance communication across the entire wafer. This fabric is characterized by a simple 5-port design with 32-bit bi-directional interfaces, which allows for single clock cycle latency between nodes and a low-cost, lossless flow control with minimal buffering. The fabric's design is further optimized through the use of static routing and the introduction

of "colors" to enable multiple routes on the same physical link.

Cerebras' architecture also addresses the challenge of scaling beyond a single die by extending the fabric across die boundaries, utilizing high-level metal layers within the TSMC process. This results in a fully homogeneous array of cores across the entire wafer, with a highly efficient, source-synchronous, parallel interface that incorporates redundancy to account for potential defects in the fabrication process.

A pivotal aspect of Cerebras' innovation is the concept of weight streaming, which facilitates the execution of extremely large neural networks on a single chip. The WSE-2 decouples the neural network model, memory weights, and compute, storing all model weights externally in a device called MemoryX. These weights are streamed onto the CS-2 system as needed, enabling the computation to proceed without the constraints of on-chip memory capacity.

The computational prowess of the CS-2 is further highlighted by its ability to leverage all 850,000 cores as a single, giant matrix multiplier, a capability that is particularly advantageous for transformer models such as GPT. The architecture's dataflow mechanisms and on-chip broadcast fabric allow for efficient weight broadcast and reductions, as well as the execution of FMAC operations directly triggered by the weights.

Cerebras' approach to deep learning hardware and software co-design represents a significant leap forward in the field, offering unparalleled performance and scalability. With the ability to support neural networks of all sizes and the capacity to handle matrices as large as 100,000 by 100,000 without partitioning, the WSE-2 and CS-2 system stand as a testament to the company's commitment to pushing the boundaries of what is possible in deep learning computation.

## 2.6 Roofline Model and Data Flow

The Roofline model is an analytical framework proposed to establish a correlation between processor performance and off-chip memory traffic [27]. In the foreseeable future, off-chip memory bandwidth is expected to be a critical resource that limits system throughput. To address this concern, the concept of "operational intensity" is introduced, which quantifies the number of operations performed per byte of DRAM traffic. For certain tasks such as matrix operations and Fast Fourier Transformation (FFT), the operational intensity increases with the problem size. The on-chip cache size and optimizations will also reduce the number of memory accesses, thereby increasing the operational intensity.

Convolutional Neural Networks (CNNs) predominantly process data within the chip, and the primary bottleneck in input-output (IO) operations is associated with weight data. While higher bit-width can effectively reduce transfer times, it also imposes more stringent requirements on circuit pins and packaging. Our work will employ the concept of "operational intensity" to investigate the necessity of such high bit-width. Moreover,specifically tailored to YOLOv8, we propose an optimal bit-width configuration that utilizes the fewest possible pins while ensuring overall throughput.

## 2.7 Activation Function

[28] provides a circuit design for configurable sigmoid and Tanh activation functions using second-order approximation and deviation compensation. This design exhibits superior speed

and area efficiency compared to traditional lookup table or polynomial approximation methods. It also utilize the function to perform the post-processing of YOLOv3 and reduced about 0.06% of the MAP accuracy. YOLOv8 also utilize the sigmoid function to calculate the confidence value. However, YOLOv8 introduces more non-linear activity function other that signoid such as the SiLU after convolution, softmax in the DFL layer. Implementing the second-order approximation in hardware solely for the sigmoid function does not yield significant advantages.

[29] presents an enhanced Rectified Linear Unit (ReLU) segmentation correction activation function, termed SignReLU. Experimental findings demonstrate that the proposed activation function expedites convergence, effectively mitigates the gradient vanishing issue, and substantially enhances the accuracy of neural network identification. Due to the ease of hardware implementation and versatility associated with piecewise linear functions (PLFs), we have devised a strategy employing general PLFs to approximate the diverse activation functions within YOLOv8.

## 2.8 LLVM-C2RTL Design Framework

In [25], the authors propose a novel method for directly describing the RTL (Register Transfer Level) structure of a pipelined RISC-V processor with cache, memory management unit (MMU), and AXI bus interface using the C++ programming language. This processor C++ model serves as a near cycle-accurate simulation model of the RISC-V core. Additionally, the authors introduce the C2RTL framework, which translates the processor C++ model into a cycle-accurate RTL description in Verilog-HDL and an RTL-equivalent C model.

The unique aspect of their design methodology lies in the fact that both the simulation model and the RTL model are derived from the same C++ source, significantly simplifying the design verification and optimization processes. The effectiveness of this methodology is demonstrated on a RISC-V processor, which successfully runs Linux OS on a field-programmable gate array (FPGA) board.

In [26], High-Level Synthesis (HLS) tools are used to implement the YOLOv7-tiny network on FPGAs. However, controlling their cycle-level behavior directly from the software description can be challenging, often requiring tool-specific pragma annotations and coding styles. To address this, the proposed C2RTL framework allows designers to describe the cycle-level behavior of the logic design in C++ using a data flow style while specifying hardware attributes with GCC-style attribute keywords. This approach enables fast simulation, design transparency, and avoids the use of proprietary languages.

In conclusion, the C2RTL design framework has been utilized in the development of AI accelerators, significantly impacting the efficiency of design implementation and verification of both the AI accelerator hardware and AI application software. By using C2RTL, we were able to describe the cycle-level behavior of the logic design in C++ with a data flow style, while specifying hardware attributes with GCC-style attribute keywords. This approach allowed for fast simulation and greater design transparency. Moreover, the framework enabled us to derive both the simulation model and the RTL model from the same C++ source, simplifying the design verification and optimization processes. This methodology facilitated the design of the AIV-SoC for YOLO, resulting in a more efficient and accessible development process for complex processors and system components, particularly in the context of IoT devices requiring flexible and low-power instruction sets.

# Chapter 3

# Deep Neural Network Accelerator System-on-Chip

## 3.1 RISC-V Pipeline Design

In [16] and [18], we presented the design of DNN-AS with maximum Int8 precision at 256×256 input resolution, supporting traditional ResNet and VGG networks. Figure 3.1 portrays the design of the 5-stage RISC-V pipeline, featuring a dedicated instruction set extension and zero-overhead loop. For the sake of simplicity, certain auxiliary circuits, such as those responsible for adventure control and branch prediction, have been omitted from the diagram. The CE control block serves as an Application Programming Interface (API) for the instruction set extension designed for DNN. Employing custom instructions to govern the CE represents a superior approach in contrast to conventional co-processor methods. Custom instructions seamlessly integrate specialized functionalities directly into the primary processor, resulting in performance and efficiency enhancements. They are instrumental in optimizing critical operations, minimizing overhead, and fostering improved hardware-software co-design. Furthermore, custom instructions offer the advantages of flexibility, adaptability, and the potential to revolutionize computer architecture, thereby paving the way for more efficient and potent computing systems.

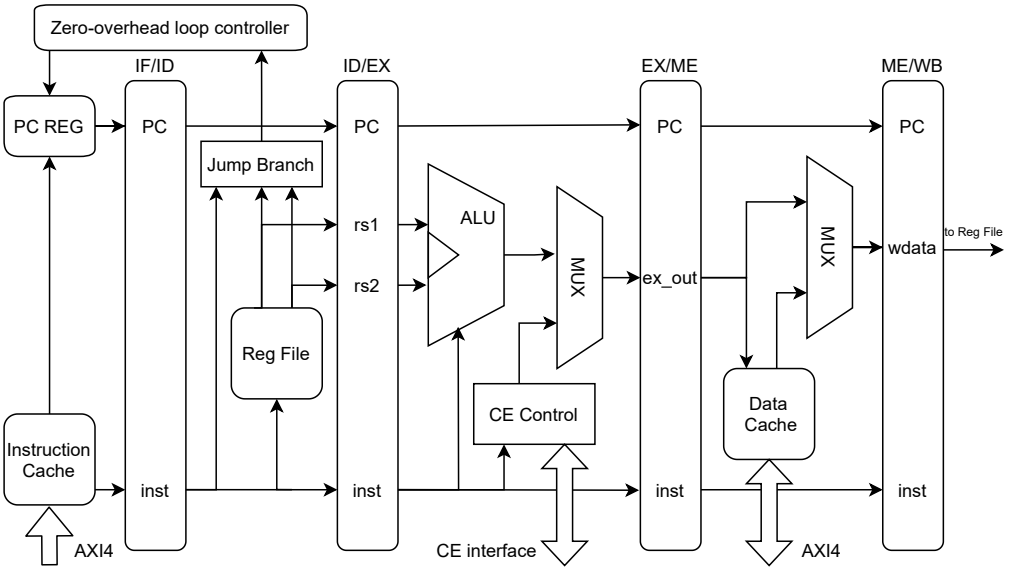



Figure 3.1: RV32IMA with zero-overhead loop control circuits

Figure 3.2 illustrates the placement of the CE and its interaction with the RISC-V architecture. During the execution of DNN calculations, data pertaining to weights and feature maps are transferred via Direct Memory Access (DMA) without affecting the Dcache within the RISC-V pipeline. To ensure synchronization, it is imperative to invalidate and flush the DCache when handling feature map data residing in DRAM.

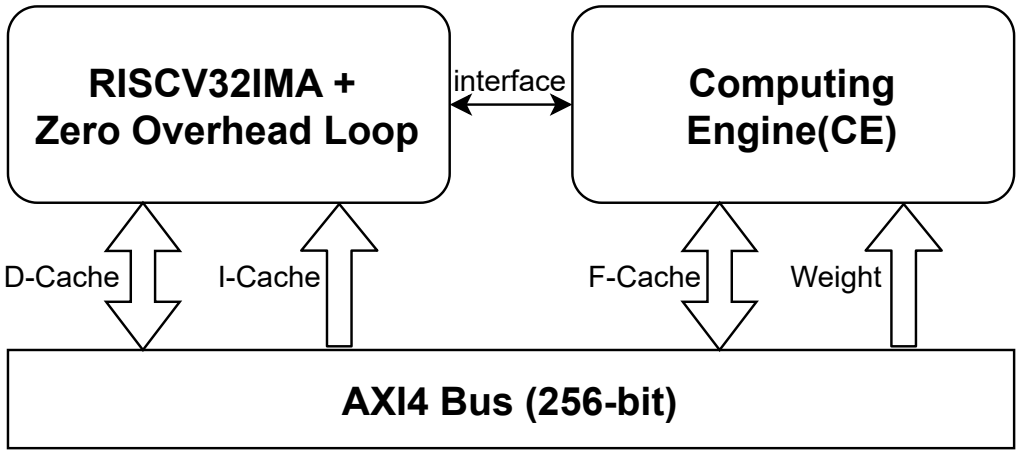



Figure 3.2: Overall Architecture of DNN-AS

## 3.2 Feature Map Cache (FMC)

As shown in Figure 4.10, we have designed 8 FMC pages, each composed of 16 32-bit 8192-depth static random-access memory (SRAM) cells, providing data widths of 512 bits. To accommodate different fixed point number precision requirements, the cache offers two input modes: 32-bit mode and 8-bit mode. In the 32-bit mode, the input is interpreted as 16 32-bit signed integers, which are commonly used to store intermediate calculation results that

have not been quantized yet. In the 8-bit mode, the input is processed as 64 8-bit signed or unsigned integers, depending on a signed bit register. Typically, it stores the layer's result after quantization. The use of two memory modes enables more efficient use of on-chip memory with minimal loss of accuracy.

Consider the 32-bit mode. For instance, the pixel in the $f$ channel at position $(x, y)$ will be stored in the $f$-th cell's $(y \times Nix + x)$ depth. In this way, a single page can store a feature map of size $90 \times 90$. However, due to significant variations in channel height and width parameters across different layers in YOLOv8, directly storing data with such a mapping relationship may not be suitable. Therefore, we have adopted a stacked and carry mechanism to effectively manage the data in FMC.

Figure 3.3 illustrates these mechanisms in the 32-bit mode. The top half picture demonstrates a layer with $Nif = 16$ and $(Niy \times Nix) = 12288$, where the overflowing 4096 depth is carried to the next 16 cells, which resides in the other FMC page. The bottom half image showcases a layer with $Nif = 32$ and $(Niy \times Nix) = 2048$, where the 17th to 32nd layers are stacked up to the 1st to 16th layers. By adopting this methodology, the release of the yellow FMC page for alternative purposes will be facilitated. Given the limitations imposed by on-chip storage technology, this investigation has constrained the cache size to 4MB, thereby restricting the input image resolution of YOLOv8 to a maximum of 352.

While it is possible to concatenate pages to achieve a larger volume, the calculation process allows only one page to be selected for input data, and one page for output data storage at a time.

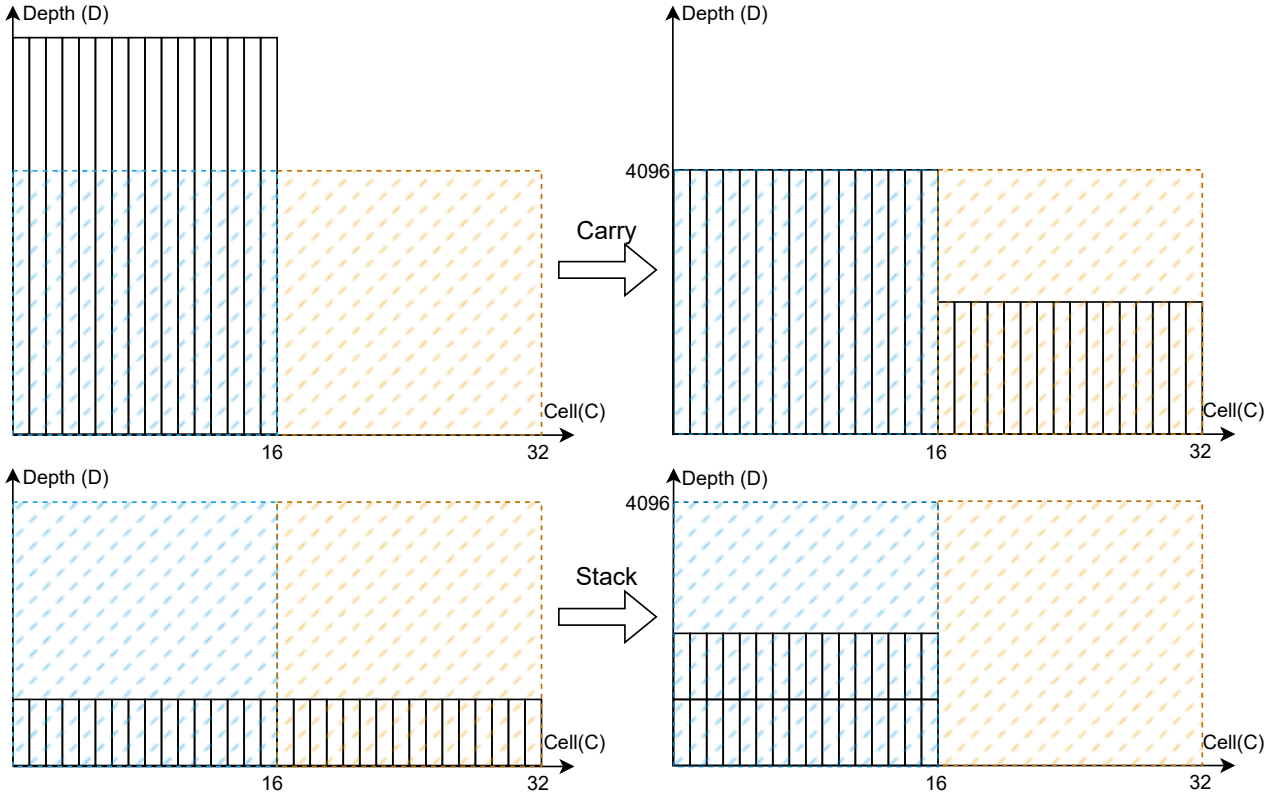



Figure 3.3: Carry and stack mechanisms

## 3.3 Convolution Loop Unrolling Strategy

Similar to Graphics Processing Units (GPUs), Single Instruction, Multiple Data (SIMD) plays a pivotal role in the optimization of large-scale neural networks. Enabling SIMD necessitates the hardware-level implementation of certain loops. Numerous established loop optimization methodologies, including but not limited to loop unrolling, tiling, and interchange [30], prove to be highly effective in mitigating these challenges. These strategies involve the decomposition of the overarching loop into more manageable subsets, with subsequent unrolling of each subset directly onto the chip.

In [16], we have compiled a comprehensive list detailing the convolution design variables of ResNet suitable for loop unrolling. These variables encompass $(Pkx, Pky)$, $Pif$, $(Pox, Poy)$, and $Pof$, symbolizing the degree of calculation parallelism. Constraints for these variables are defined as $1 \leq P^* \leq N^*$, where $P^*, N^*$ represent any variable with a prefix of P, N. YOLOv8 incorporates various operations beyond convolution, including max-pooling, matrix multiplication, transposed convolution, and up-sampling. Despite the structural distinctions among these operations, they can be categorized into four loops based on their alignment with the addresses of the FMC.

In Figures 3.4, 3.5 and 3.6, we provide pseudo-code representations of convolution, max-pool and matrix multiplication, incorporating 32-bit mode FMC as delineated in Table 3.1. The symbol $N^*$ within the context of FMC signifies its maximal capacity to hold data along a specific dimension, whereas $P^*$ denotes the upper limit of the attainable loop unrolling coefficient within its input/output bandwidth.

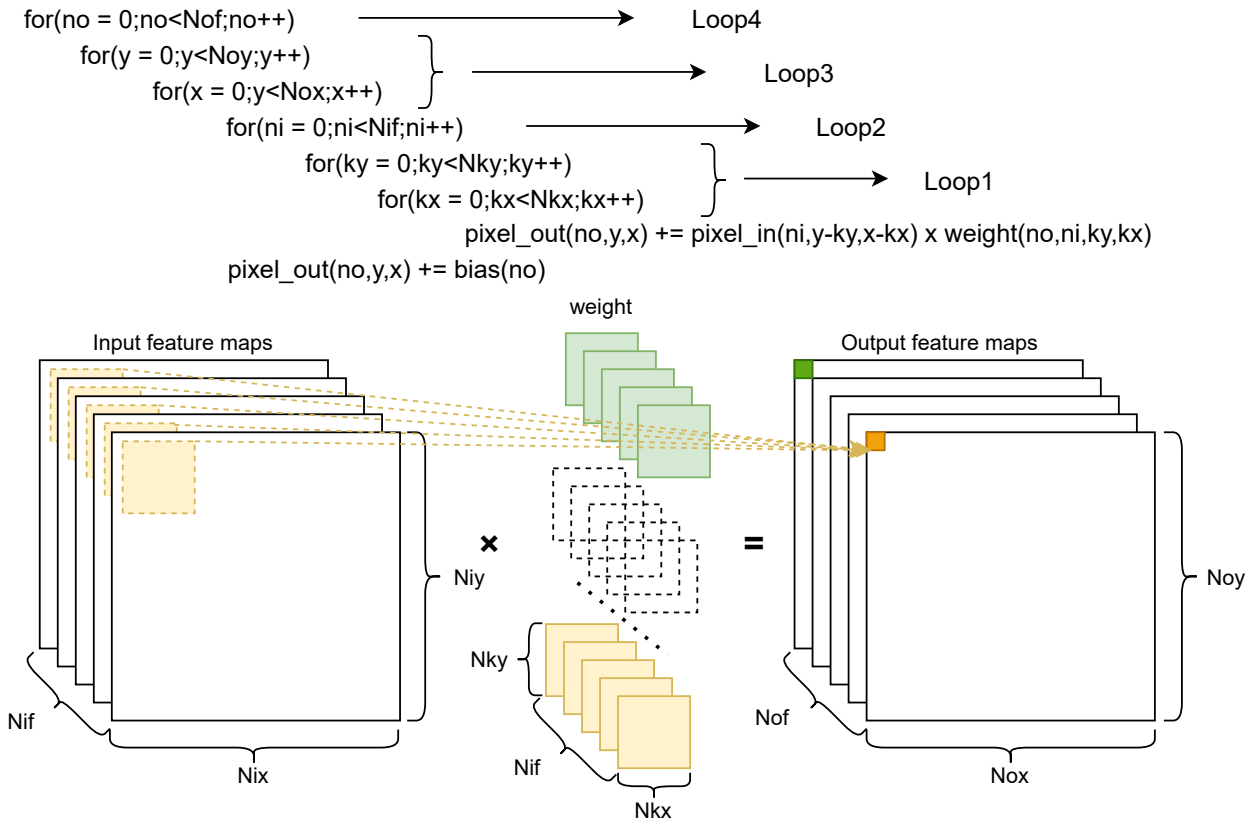



Figure 3.4: Structure of convolution

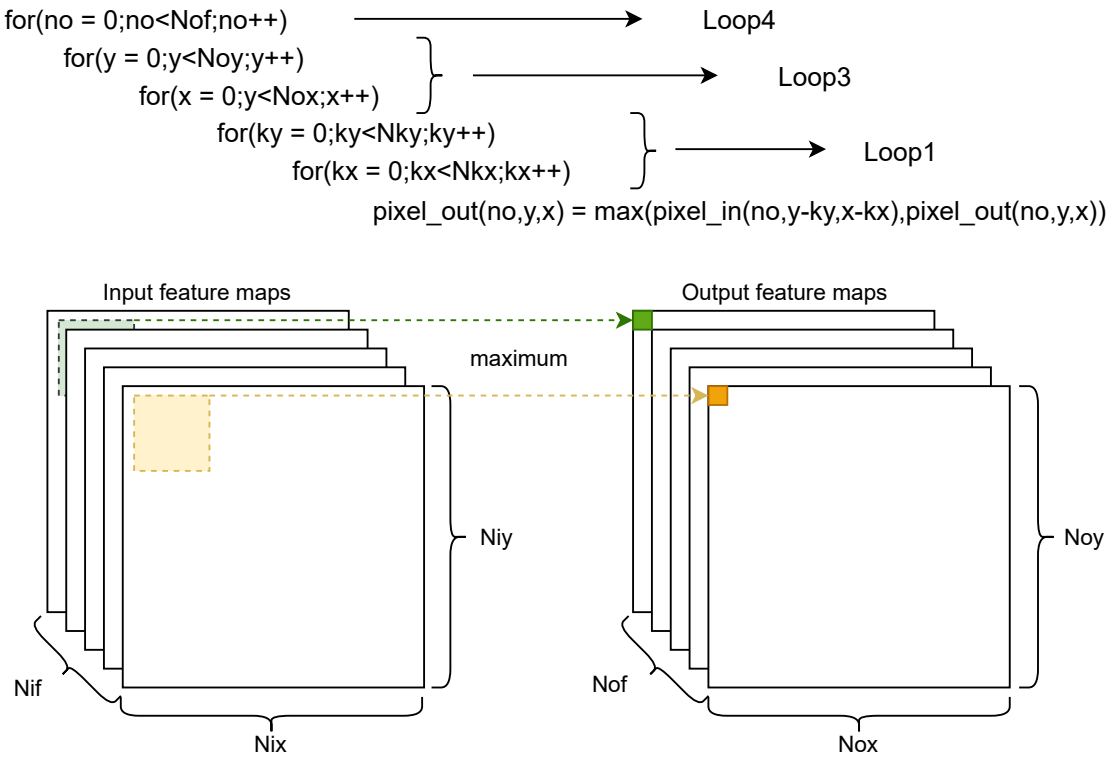



Figure 3.5: Structure of max-pool

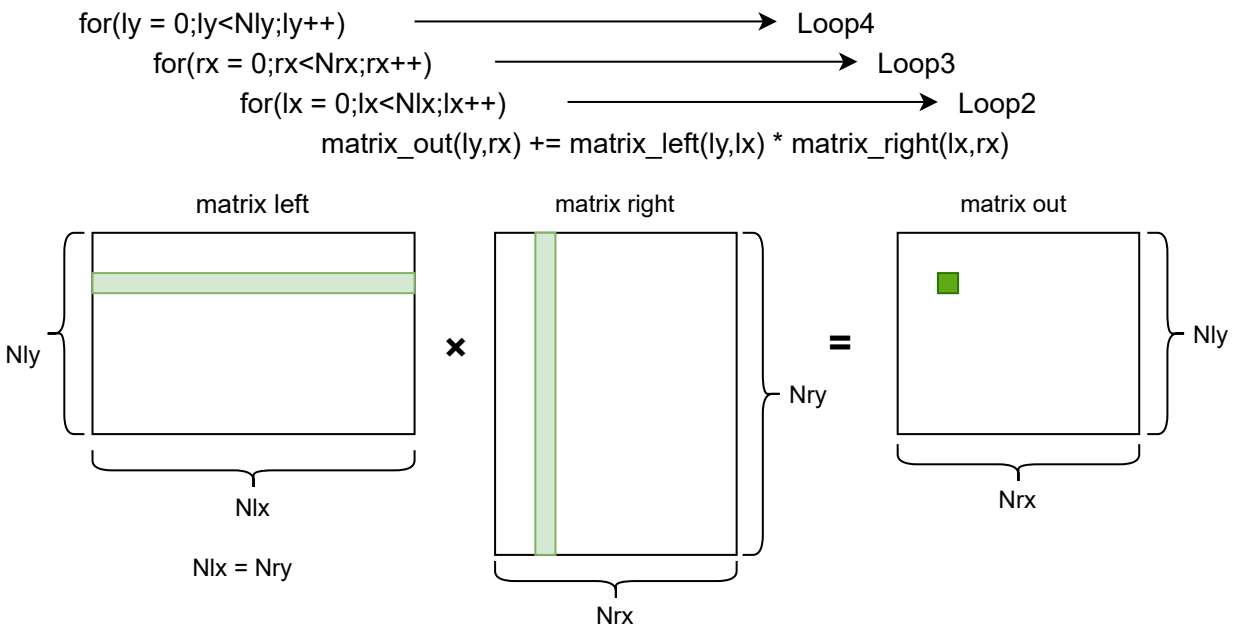



Figure 3.6: Structure of matrix multiplication

Additionally, data reuse plays a pivotal role in reducing the number of I/O operations. Two forms of data reuse are particularly relevant: spatial reuse and temporal reuse [31]. Spatial reuse implies that data is employed by multiple parallel multipliers, whereas temporal reuse

denotes the use of data for multiple consecutive clock cycles. In convolution, weight data can be temporally reused in loop 3, input feature map data can be spatially reused in loop 3 and 4.

In Loop1 of Table 3.1, numerous unrolling coefficients extended the FMC limitation, introducing certain challenges. However, convolution and max-pooling operations rely on successive sliding windows, resulting in a significant overlap between adjacent windows. To leverage this crucial characteristic and enhance the efficiency of external memory bandwidth, an innovative line-buffer and data-core caching scheme were introduced by [32], involving two-port RAMs. Since two-port RAMs entail greater area and power consumption compared to their single-port counterparts, we employs a combination of SRAM and Register structures. Consequently, we opt to implement the sliding window's caching scheme as shown in Figure 3.7.

Table 3.1: Loop dimensions with unrolling variable

| | Coefficient | Loop1 | Loop2 | Loop3 | Loop4 |
|---|---|---|---|---|---|
| FMC (32-bit mode) | Dimension(N*) | in-depth | in-cell | out-depth | out-cell |
| | Unrolling (P*) | 1 | 16 | 1 | 16 |
| Convolution | Dimension(N*) | (Nkx, Nky) | Nif | (Nox, Noy) | Nof |
| | Unrolling (P*) | (Pkx, Pky) | Pif | (Pox, Poy) | Pof |
| Max-pooling | Dimension(N*) | (Nkx, Nky) | / | (Nox, Noy) | Nof |
| | Unrolling (P*) | (Pkx, Pky) | / | (Pox, Poy) | Pof |
| Matrix Multiply | Dimension(N*) | / | Nlx | Nrx | Nly |
| | Unrolling (P*) | / | Plx | Prx | Ply |

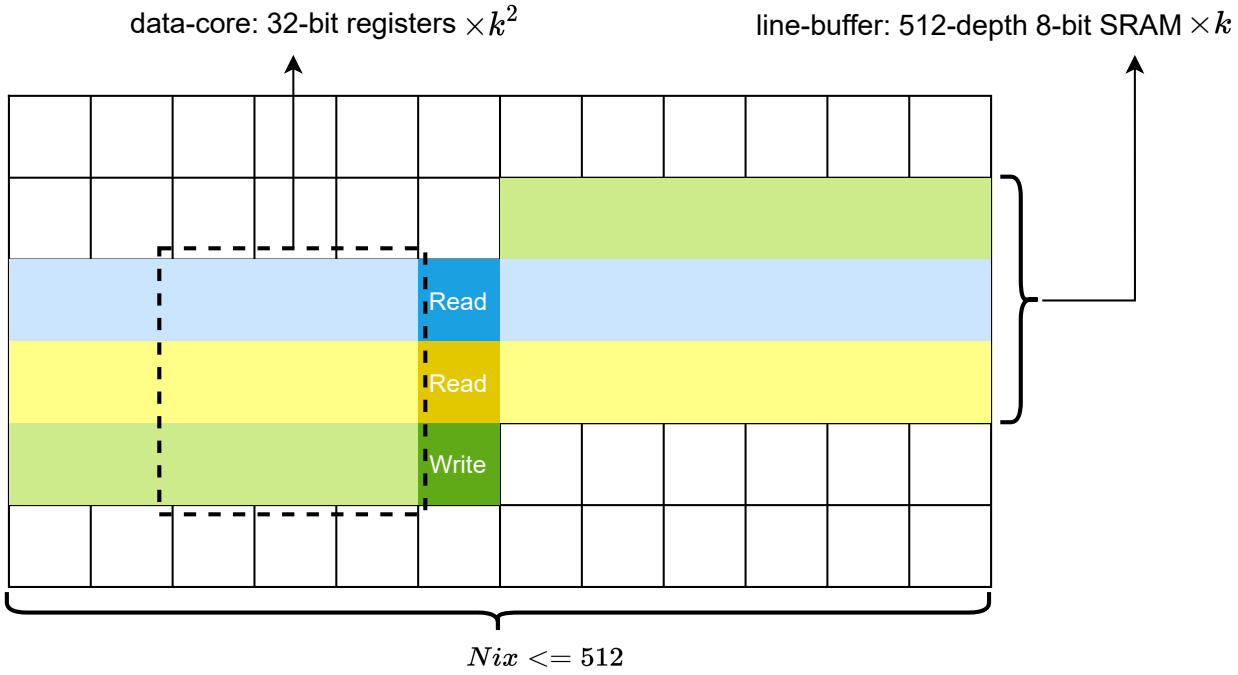



Figure 3.7: Line-buffer and data-core caching scheme

For a kernel with a square size denoted as $(k, k)$, when $k > 1$, data from the FMC is transferred to line-buffer and data-core caching scheme. Figure 3.7 illustrates the hardware

resource utilization for a $k \times k$ kernel when unrolling loop1. The existing data from the line-buffer (depicted in dark blue and yellow) and the new data from the FMC (depicted in dark green) are then transferred to the data-core register array, refreshing sliding windows at the next clock cycle. As shown in Figure 3.8, our approach allows for an increased input data bit-width compared to traditional vector instructions, resulting in significantly higher throughput.

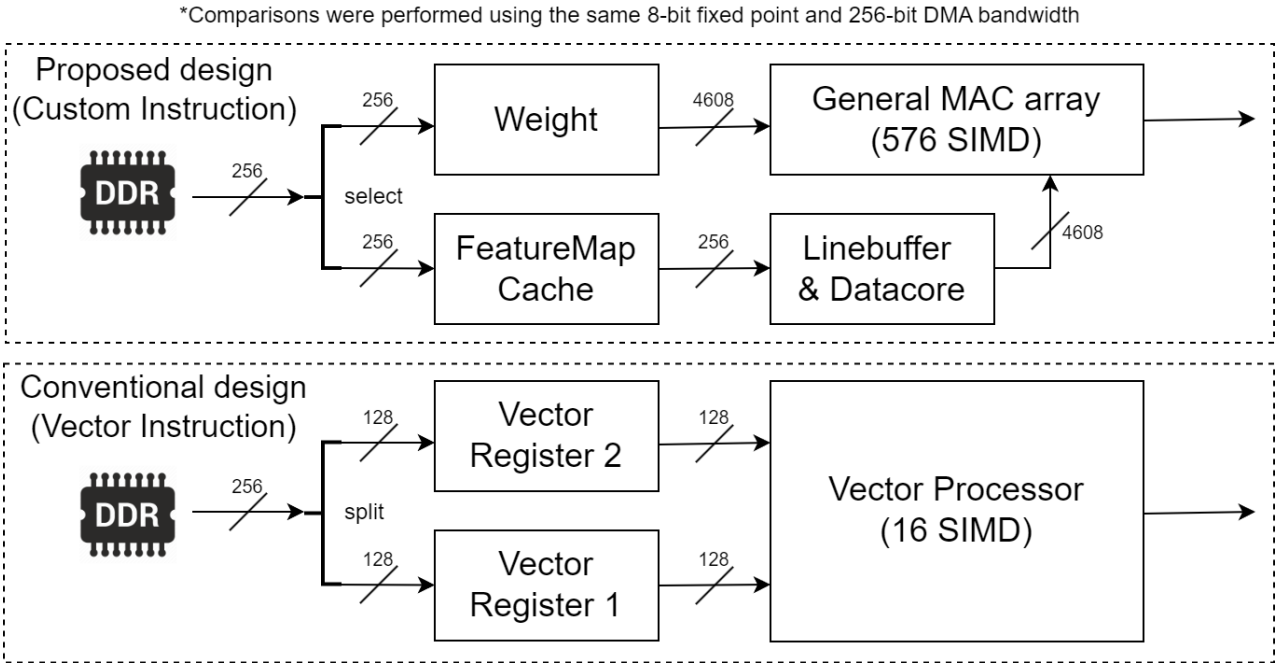



Figure 3.8: Input bit-width amplification

Max-pooling consumes $Pof \times k$ line-buffers and $Pof \times k^2$ data-cores, while convolution requires $Pif \times k$ line-buffers and $Pif \times k^2$ data-cores. Considering the overall hardware resources and constraints, in this paper, we establish 24 32-bit 512-depth SRAM line-buffers, supporting a maximum input feature map size ($Nix$) of 512. The number of data-cores is set to 100 to accommodate max-pooling with $(5, 5)$ kernel size and $Pof = 4$. Detailed information regarding the $P^*$ coefficient for various layers in YOLOv8 is presented in Table 3.2.

Table 3.2: Unrolling Parameters (P*) of different layers

| | Loop1 | Loop2 | Loop3 | Loop4 |
|---|---|---|---|---|
| Maxpool 5×5 | (5,5) | / | (1,1) | 4 |
| Matrix Multiplication | / | 16 | (1,1) | 16 |
| Convolution 3×3 | (3,3) | 8 | (1,1) | 16 |
| Convolution 1×1 | (1,1) | 64 | (1,1) | 16 |

## 3.4 General Multiply-Accumulate Tree Array

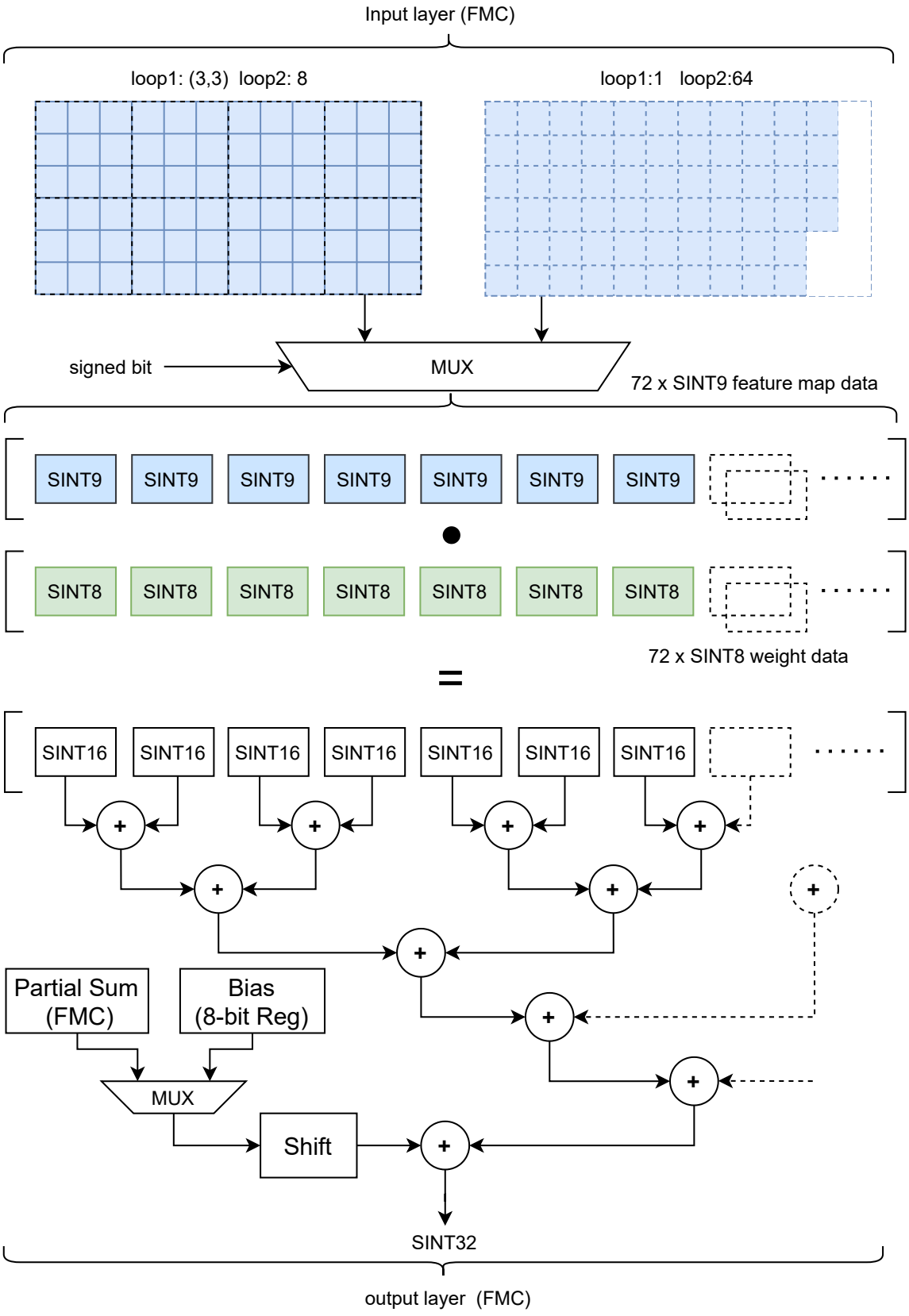



Figure 3.9: MAC tree

To unroll the convolution and matrix multiplication loops 1, 2, and 4, denoted as $P^*$ in Table 3.2, we have developed a versatile MAC tree array comprising 16 independent MAC tree circuits. These 16 MAC trees share the input feature map's data while employing distinct weights. Consequently, a set of 16 contiguous channels in loop 4 can be computed concurrently.

Each MAC tree is responsible for the hardware implementation of loops 1 and 2. Given that the majority of layers in YOLOv8 utilize traditional 3x3 convolutions, we set the number of MAC operations for each MAC tree as $loop1 \times loop2 = (3 \times 3) \times 8 = 72$. Figure 3.9 illustrates a MAC tree, showcasing incorporation of adjusted bias data or partial sums from the preceding group. The blue square represents a pixel on the feature map, while the white square indicates zero padding. Despite introducing some hardware inefficiency, given the widespread utilization of 3x3 convolutions in YOLOv8, the associated loss is considered negligible.

## 3.5 Statistic Block for Dynamic Fix-Point (DFP) Scheme

In circuit design, the floating-point multiplication operation requires numerous shift circuits, which can lead to increased circuit delay. Therefore, we have opted to replace the float instruction extension with the DFP scheme to minimize unnecessary shift manipulations.

[33] also introduced DFP scheme for feature map and weight data. Considering a convolution layer made of $Lf(Nof \times Noy \times Nox)$ feature map data and $Lw(Nif \times Nof \times Nkx \times Nky)$ weight data. They assigned one public exponent to each of them, which may lead to significant truncation error if Lf or Lw are very large. To address this challenge, we introduced "per-group" quantization [17] as shown in Figure 3.10, where each group has its own exponent. As shown in Table 4.3, each convolution output group consists of $Gl(16 \times Noy \times Nox)$ feature map data, corresponding to $Gw(16 \times Nif \times Nkx \times Nky)$ weight data. This more meticulous quantization helps reduce the truncation error. Additionally, because unquantized data occupies four times more space than quantized data, the "per-group" quantization also reduces the on-chip cache requirement.

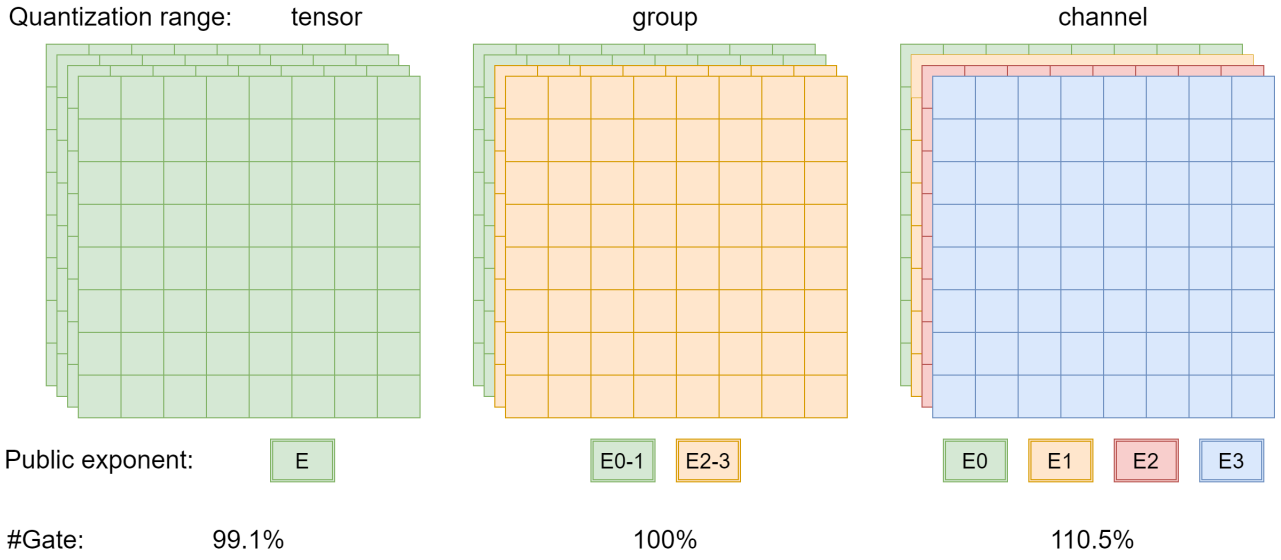



Figure 3.10: Quantization in different range

It is easy to implement the fixed-point scheme on hardware for its lower delay and smaller

size. However, the range of the fix-point is too small compared with the floating point. We introduce a DFP scheme to solve this problem. Considering a group made of $Gl(16 \times Noy \times Nox)$ pixels or $Gw(16 \times Nif \times Nkx \times Nky)$ weights, we add a public exponent E for it. The value representation example of an unsigned number is shown in Figure 3.11.

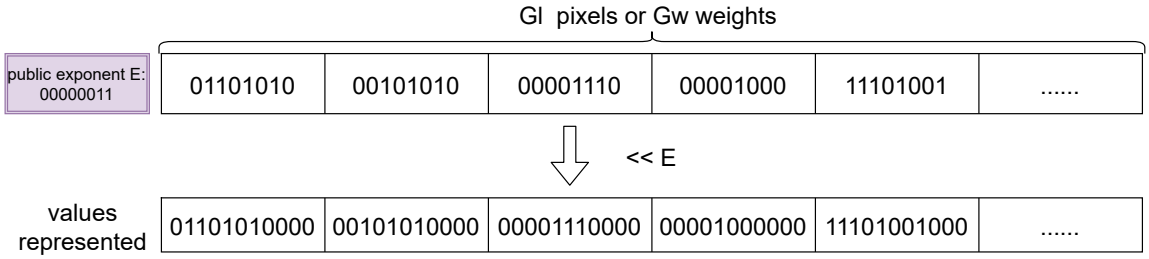



Figure 3.11: Representation of DFP scheme

To simplify the data flow, let us assume the bias is zero. The input feature map is M, the output feature is N, and the weight is W. The following Figure 3.12 shows that a floating multiplication in the DFP scheme will be divided into an addition of public exponent and a multiplication of fixed point mantissa. As depicted in Figure 4.10, when the WE signal of the feature map cache is active, the write data copy will be automatically sent to the statistics module. Subsequently, the statistics module will accumulate the distribution information of the highest bit. Once all calculations within the group are completed, the statistics module will determine and generate the optimal exponent (truncation bit) for the current group.

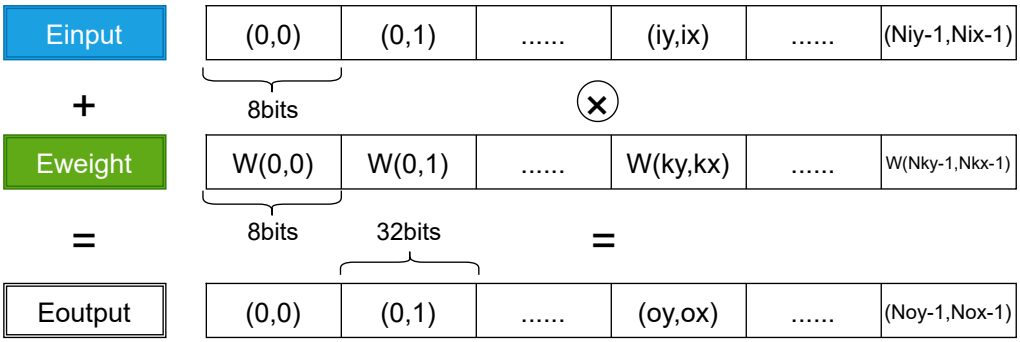



Figure 3.12: The output of convolution in DFP scheme

During inference(eval mode), the weight is static,that is to say, The $Gw$ weights is quantized before inference. While for the group of $Gl$ pixels is dynamic, so we have to quantize them during main computing process.The unquantized $Gl$ pixels from the MAC array is 32-bit integer data, while the quantized input for the next layer is 8-bit integer data. So we need to quantize these SINT32 to SINT8 for each group respectively. Usually, the public exponent of a group is decided by the maximum absolute value of its elements. This strategy is reasonable because the operands with a great absolute value usually dominate the result of the DNN operations. However, the quantization error made up of parameters in small absolute values is not negligible. First, we sort each $L_x$ in one group from the smallest to largest according to its absolute value. To simplify the discussion, let's assume that the system is signed mode,

we only display the 7-bit absolute value for SINT8 in Gl. In Figure 3.13, we truncate pixels whose absolute value is larger than $2^{shiftbit+7}$ with Round-to-the-Nearest schemes.

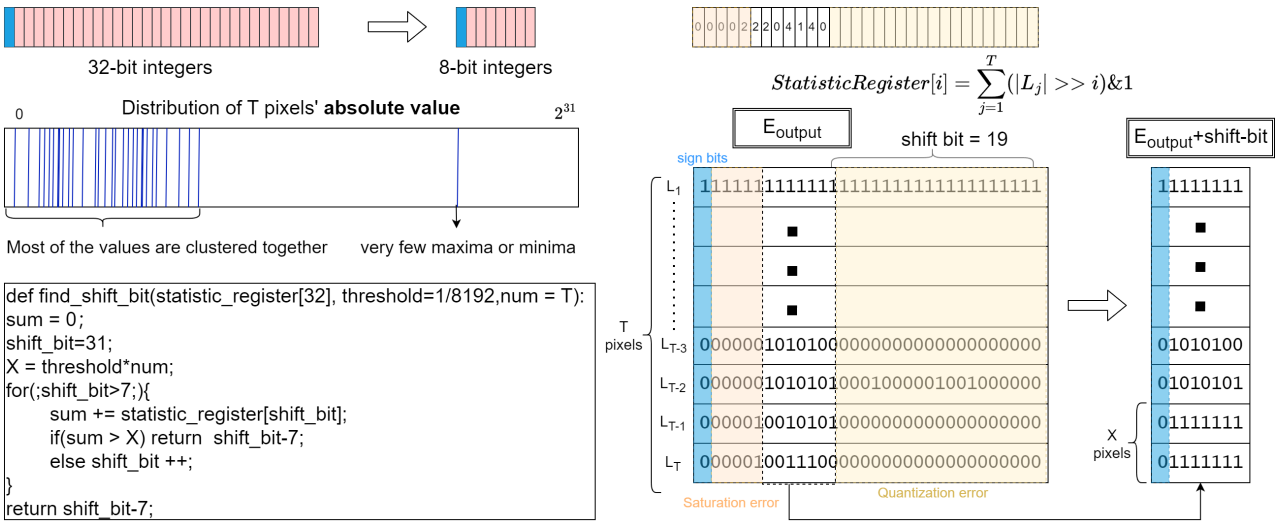



Figure 3.13: Truncation error analysis

The quantization error obeys even distribution $U(-2^{shiftbit-1}, 2^{shiftbit-1})$. Usually, $Gl$ is larger than 1000 for most group, in which case, the sum of quantization error is approximate to continuous Gaussian distribution. The variance of it only depends on shift-bit, which is Eq.(3.1).However, it is hard to evaluate the truncation errors because the number of truncation is small compared to the quantization error. Here we suppose the distribution quantization error is $U(-2^{shiftbit+7}, 2^{shiftbit+7})$ with corresponding variance Eq.(3.2). Under this distribution, the total variance is shown in Eq.(3.3).

As shown in Figure 3.13, the number of truncation pixels X is vital for the total variance. Thus, as shown in Eq.(3.4) we need to make a balance between the truncation error and the quantization error. We define $X/Gl$ as a tolerance threshold which means we can allow some pixels to be truncated. In other words, the public exponent is determined by the most significant portion of the data set rather than the largest data, which increases the system's robustness. we can get a suitable tolerance threshold is around $2^{-16}$ because the truncation and quantization errors have the same order of magnitude in this proportion if our suppose Eq.(3.2) is true. After many experiment, we use $2^{-13} = 1/8192$ as the tolerance threshold $X/Gl$.

Because we need all the output pixels of one group to be truncated before we can decide on the X pixels to be trimmed, when performing convolution, we need to record the highest "1" bit's position of each pixel's absolute value. As shown in Figure 3.3, we use 32 on-chip registers to store statistic information in the statistics module. After finishing all the pixels in current output group, we will use these statistical data to quantize current group.

$$Var(Err_{quan}) = Var(\sum_{i=0}^{Gl} Err_{quan,i}) = \frac{Gl}{12}(2^{shiftbit})^2 \tag{3.1}$$

$$Var(Err_{trun}) = Var(\sum_{i=0}^{X} Err_{trun,i}) = \frac{X}{12}(2^{shiftbit+8})^2 \tag{3.2}$$

$$Var(Err_{total}) = \frac{Gl + 2^{16}X}{12}(2^{shiftbit})^2 \tag{3.3}$$

$$Err_{total} = Err_{trun} + Err_{quan} \tag{3.4}$$

## 3.6 Custom Instruction Design

The DNN-AS architecture is developed based on RISC-V32IMA. By customizing the instruction set, we can seamlessly deploy various DNN models, including RESNET, while ensuring both execution efficiency and maximizing compatibility. Additionally, the ease of modifying the customized instruction set gives us the potential to adapt to emerging new models in the future.

### 3.6.1 Zero-overhead Loop Instruction

Apart from hardware-based loop unrolling, software is required for implementing convolution loops. To enhance program efficiency, we introduce the Zero-overhead loop command, as depicted in Figure 3.14. We construct the model during the decode stage, similar to the diagram presented in Figure 4.10. This model enables the execution of a non-nested loop with a maximum length of seven commands. In the zero-overhead loop command, funct3 'a' represents the length of the loop's body, while register 'b' or an immediate number determines the number of iterations. Our software code, illustrated in Algorithm 1, demonstrates the use of a zero-overhead loop for a loop comprising three commands, repeated 100 times.

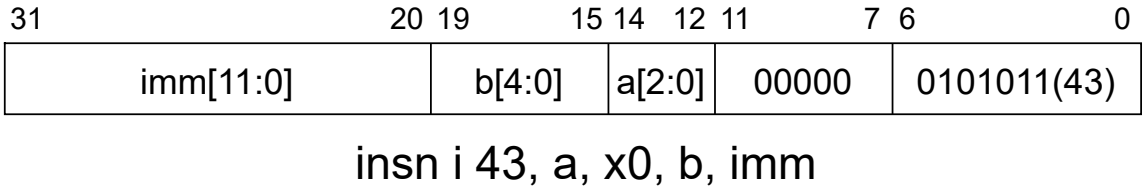



Figure 3.14: Zero-overhead loop in I format

**Algorithm 1** Example of performing a 100 times loop of 3 commands

```
int iteration = 100;
asm volatile(".insn i 43, 3, x0, %0, 0"::"r"(iteration));   ▷ a = 3, b = iteration
loop command x;
loop command y;
loop command z;
```

### 3.6.2 Control Instruction

To implement extension instructions for CNN, the chip must obtain detailed information, such as the layer's channels and size, convolution kernel, and the Rectified Linear Unit (ReLU) activation function. Specific registers store this information within RISC-V. Initially, we utilize

standard instructions like `lw` or `sw` to transfer data from DDR-memory to `REGFILE` via the `Dcache`. Subsequently, we employ custom instructions, as shown in Figure 3.15 and detailed in Table 3.3, to configure the special registers using values from `rs1` and `rs2` in the register file. Control instructions offer us the flexibility to manage various types of neural network layers, enabling dynamic configuration of the neural network via software. Also, we need to get the statistical information from the computation to help us in the next quantization operation. The feedback of public exponent is delivered by the `rd` register.

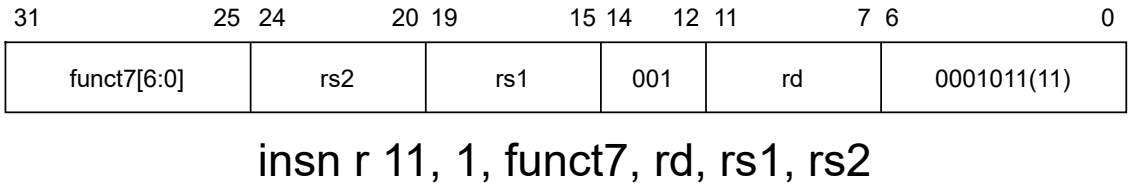



Figure 3.15: Control and IO in R format

Table 3.3: Control instruction

| Instruction | Description |
|---|---|
| GEN_EXP(rd) | Generate public exponent of out-layer using Statistic block |
| INIT_LAYER(rs2,rs1) | Initialize and read size of layers |
| RESET | Reset all the register in CE |
| SET_LAYER(rs1) | Set the residual, partial sum, input and output layers in memory slices |
| SET_EXP(rs1) | Set the public exponent of input-layer |
| LOAD_BIASES(rs2,rs1) | Load 8 consecutive-addresses biases |
| SET_CHANNEL(rs2) | Set the number of input-layer's channels |
| CLEAR_WEIGHTS | Clear all the weights loaded |

### 3.6.3 AXI4 DMA Instruction

The original I/O command of RISC-V is routed through the Dcache. Consequently, the efficiency of `lw` and `sw` commands is suboptimal. This inadequacy becomes particularly evident when dealing with the substantial data transfers associated with weights and feature maps. Using the port as a control instruction is an impractical choice for such tasks.

To address this issue, we have introduced the AXI4 DMA instruction for loading weights and feature maps, as illustrated in Table 3.4 and Figure 3.16. These DMA commands adopt the same format as the control instructions depicted in Figure 3.15. Notably, the "LOAD WEIGHTS" command does not introduce idle cycles into the pipeline. Consequently, we can perform computations and transfer data simultaneously.

It's worth noting that the Fully Connected (FC) layer typically involves a substantial volume of weights but only a few calculation operations. Due to the constraints imposed by the AXI4 bandwidth, the calculation commands in the FC layer must await the DMA data transfer commands.

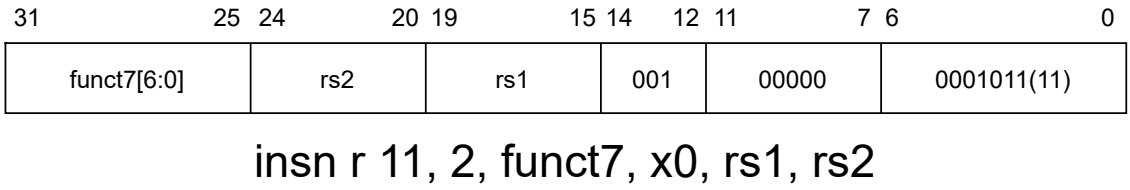



Figure 3.16: AXI DMA instruction in R format

Table 3.4: AXI4 DMA INSTRUCTION

| Instruction | Description |
|---|---|
| LOAD_WEIGHTS(rs1,rs2,x) | Initialize AXI4 reading with `x` burst-length loading 8 * `x` INT64 data for weights in address start from (rs1+rs2) |
| FREAD(rs1) | Initialize AXI4 reading with 16 burst-length loading 8 * 16 INT64 data for Fcache in address start from (rs1) |
| FWRITE(rs1) | Initialize AXI4 writing with 16 burst-length writing 8 * 16 INT64 data from Fcache to address start from (rs1) |

### 3.6.4 Compute Instruction

The calculation instructions encompass various operations, including convolutions of sizes 3x3, 7x7, and 1x1, as well as fully connected layers, all of which utilize the general Multiply-Accumulate (MAC) array. The format of these calculation instructions is depicted in Figure 3.17, which follows the R-type format. However, it's worth noting that in this context, the registers rs1, rs2, and rd remain unused. A comprehensive breakdown of their specific usage is presented in Table 3.5.

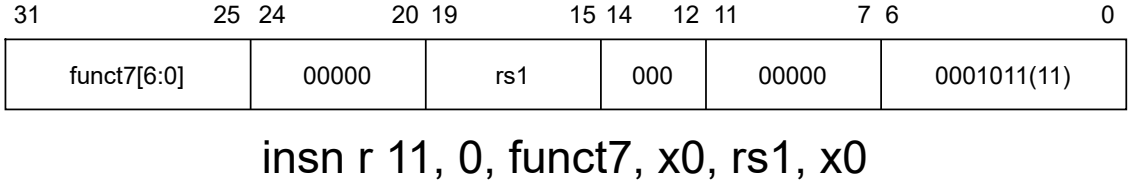



Figure 3.17: Calculation instruction in R format

Table 3.5: Compute instruction

| Instruction | Description |
|---|---|
| CALC_MAXPOOL3 | Perform 16 max-pool of 3x3 size |
| CALC_AVGPOOL | Perform average-pool |
| CALC_FC | Perform 16*64 MAC in fully connected layer |
| CALC_CONV1 | Perform 16*64 convolutions of 3x3 kernel |
| CALC_CONV3 | Perform 16*8 convolutions of 3x3 kernel |
| CALC_CONV7 | Perform 16 convolutions of 7x7 kernel |

## 3.7 Experimental Results

### 3.7.1 Hardware Simulation

The hardware design is implemented using C++ and compiled using LLVM 11.0.0 and C2RTL 4.1.10 versions. The C++ code is then converted into an RTL Verilog file. The Front-End hardware design is synthesized with Synopsys Design compiler. Initially, we employ the TSMC 28nm SRAM memory compiler, TSN28HPCPD127SPSRAM_20120200, to produce 16 individual single port 32-bit 4096-depth SRAM cells. Subsequently, we utilize the compiler TSN28HPCPUHDSPSRAM_20120200 to synthesize the circuit incorporating these memory cells. The simulation constraint and outcomes for DNN-AS in comparison to the original RISC-V processor are illustrated in Table 3.6.

Table 3.6: Hardware Resource Utilization

| | DNN-AS | RISCV32IMA |
|---|---|---|
| Frequency | 540Mhz | 740Mhz |
| Clk period | 1.851ns | 1.35ns |
| Area | $8.3mm^2$ | $0.11mm^2$ |
| Memory size | 4,328KB | 35KB |
| Power | 137.7mW | 34.8mW |
| Temperature | 25°C | |
| Voltage | 0.9V | |
| Clk uncertainty | 0.3ns | |
| Max transition | 0.3ns | |
| Max fanout | 32 | |

To underscore the power efficiency of our design, we conducted a comprehensive comparison between our DNN-AS and other related platforms as shown in Table 3.7. To make a fair comparison of resource utilization and architecture efficiency, we introduce the concepts of throughput efficiency (TE) and power efficiency (PE).TE is defined as the ratio of actual throughput (AT) to the peak throughput (PT). The TE is similar to the "TRP_Eff" concept described in [34], and it quantifies the real-time efficiency of general MAC operations.

PE is calculated by dividing the power consumption by actual throughput. It is equivalent to the actual energy (mJ) for per Giga operation (Energy/GOPs). These metrics help us evaluate the effectiveness of our DNN-AS implementation in terms of both computational performance (throughput) and energy efficiency (power consumption).

The actual throughput depends on the neural network's architecture. We assume that each inference of RESNET requires N multiplication operations. Assuming we have L convolution layers, where the subscript $i$ represents the $i$th layer among the L layers, the variables $Nox$, $Noy$, $Nkx$, $Nky$, $Nif$, and $Nof$ correspond to the same definitions as depicted in Figure **??**.

$$N = \sum_{i=0}^{L} (Nox_i \times Noy_i \times Nkx_i \times Nky_i \times Nif_i \times Nof_i) \tag{3.5}$$

While the accelerator consumes T clocks in performing inference at a frequency of F, one multiplication operation based on the matrix multiplier must correspond to an addition

operation in the convolution operation. Then, the actual throughput of the accelerator is given by:

$$ActualThroughput(AT) = \frac{N \times F}{T} \tag{3.6}$$

On the other hand, from a hardware perspective, the circuit resources for multiplication are limited. Without considering the IO bottleneck, there is a theoretical peak throughput. Assuming we have "mult" multipliers on the chip, the peak throughput is given by:

$$PeakThroughput(PT) = mult \times F \tag{3.7}$$

TE is defined as the ratio of actual throughput (Eq.(3.6)) to peak throughput (Eq.(3.7)):

$$TE = \frac{N}{T \times mult} \tag{3.8}$$

Table 3.7 presents the TE, PE and precision alongside other relevant information. In this context, the abbreviation `FP` denotes floating-point representation, whereas `INT` signifies fixed-point representation. Specifically, `INT8/16` indicates that certain portions of the network's quantization utilize a 16-bit representation, while others employ an 8-bit representation. `Reconfigurable` refers to the capability of this platform to modify model weights or types through software. For the reference [35], due to the absence of time consumed data in the author's work, only the PT can be obtained. The data referenced in [36], [37] and [38] was also obtained from [34]. However, these 6 references didn't explicitly specify whether operations related to external storage were included in computing AT. The simulation environment in our study is Linux-based. Therefore, while calculating AT, we considered all RISC-V instructions, including Linux system commands and interrupts. For the data of [13], we use our own device to implement the same model as DNN-AS, we will also include operation other than GPU, such as CPU task and memory transfer time, in the AT calculation.

Table 3.7: Comparison with other acceleration platform

| | [34] | [36] | [37] | [37] | [38] | [35] | [13] | DNN-AS | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Platform | VX980T | XC7Z045 | 10GX2800 | 10GX1150 | XC7K325T | TSMC65nm | GTX1080Ti | TSMC28nm | | |
| Processor | ✗ | Cortex-A9 | ✗ | ✗ | ✗ | ASIC | GP102 | RISCV32IMA | | |
| Reconfigurable | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | | |
| Model | RESNET101 | VGG16 | VGG16 | VGG16 | RESNET101 | VGG16 | RESNET101 | RESNET34 | RESNET50 | RESNET101 |
| Precision | INT8/16 | INT16 | INT8/16 | INT8/16 | INT8 | INT16 | FP32 | INT8 | | |
| Frequency(MHz) | 100 | 150 | 300 | 240 | 200 | 962 | 1481 | 540 | | |
| AT(GOP/s) | 600 | 137 | 1605 | 968 | 342 | / | 300.7 | 357.5 | 270.5 | 314.3 |
| PT(GOP/s) | 624 | 235 | 4916 | 1505 | 410 | 91.45 | 11347 | 622 | | |
| TE(%) | 96.12 | 58.3 | 32.65 | 64.31 | 83.5 | / | 2.65 | 57.5 | 43.5 | 50.5 |
| Bandwidth(GB/s) | 12.8 | 4.2 | 16.9 | 16.9 | / | / | 484.4 | 17.3 | | |
| Power(W) | 12.39 | 9.63 | 100 | 40 | 16.5 | 0.394 | 250 | 0.138 | | |
| PE(mJ/GOPs) | 20.64 | 70.27 | 62.11 | 41.32 | 48.21 | 4.31* | 833 | 0.386 | 0.51 | 0.44 |

* PT is used instead of AT to calculate PE.

### 3.7.2 Tolerance Threshold for the Quantization

In Figure 3.13, it is demonstrated that the data of the feature map ought to be quantized for storage in on-chip memory in an 8-bit mode. To define a feasible portion as the tolerance threshold for the quantization, Table 3.8 illustrates the Top1 and Top5 accuracy of RESNET's implementation when employing Int8 layer quantization at different tolerance thresholds. To mitigate potential interference, the weights remain unquantized and are utilized as 32-bit

floating-point numbers throughout the simulations in Table 3.8. The accuracy of the original unquantized float implementation (denoted as 'float' in quantization column) is also provided. After comprehensive analysis of the test outcomes, we have opted for a tolerance threshold of 1/8192 for our simulation.

Table 3.8: Accuracy of various tolerance Threshold

| Model | Tolerance Threshold | Top1 | Top5 |
|---|---|---|---|
| RESNET34 | float | 73.30% | 91.42% |
| | 0 | 73.31% | 91.45% |
| | 1/8192 | 73.32% | 91.40% |
| | 1/4096 | 73.29% | 91.43% |
| | 1/2048 | 73.19% | 91.41% |
| | 1/1024 | 72.94% | 91.31% |
| RESNET50 | float | 76.15% | 92.87% |
| | 0 | 76.09% | 92.87% |
| | 1/8192 | 76.07% | 92.87% |
| | 1/4096 | 76.10% | 92.93% |
| | 1/2048 | 76.00% | 92.90% |
| | 1/1024 | 75.91% | 92.78% |
| RESNET101 | float | 77.37% | 93.56% |
| | 0 | 77.29% | 93.55% |
| | 1/8192 | 77.31% | 93.49% |
| | 1/4096 | 77.25% | 93.52% |
| | 1/2048 | 77.30% | 93.49% |
| | 1/1024 | 77.12% | 93.49% |

### 3.7.3 ILSVRC2012 Image Classification

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 dataset is a pivotal resource in the field of computer vision, offering over 1.2 million high-resolution images across 1000 distinct categories, which are organized in a hierarchical structure to reflect the relationships between different classes. The annotations accompanying each image, including class labels and bounding boxes, are crucial for supervised learning tasks, and the dataset's public availability has democratized access to a wealth of visual data, enabling a broad spectrum of researchers and practitioners to advance in their studies and projects. The ILSVRC2012 dataset has been a catalyst for significant progress in deep learning and convolutional neural networks, including RESNET and VGG, and has set a benchmark for evaluating the capabilities of visual recognition systems.

Table 3.9: Accuracy of various quantization scale

| Model | Quantization | Top1 | Top5 |
|---|---|---|---|
| RESNET34 | float | 73.30% | 91.42% |
| | per-channel | 73.08% | 91.34% |
| | per-group | 72.77% | 91.19% |
| | per-tensor | 72.35% | 90.88% |
| RESNET50 | float | 76.15% | 92.87% |
| | per-channel | 75.88% | 92.85% |
| | per-group | 75.84% | 92.74% |
| | per-tensor | 75.42% | 92.63% |
| RESNET101 | float | 77.37% | 93.56% |
| | per-channel | 77.26% | 93.52% |
| | per-group | 76.69% | 93.23% |
| | per-tensor | 75.33% | 92.38% |
| VGG19 | float | 72.39% | 90.87% |
| | per-channel | 72.33% | 90.86% |
| | per-group | 77.26% | 90.85% |
| | per-tensor | 77.19% | 90.76% |
| VGG16 | float | 71.59% | 90.39% |
| | per-channel | 71.57% | 90.35% |
| | per-group | 71.42% | 90.35% |
| | per-tensor | 71.34% | 90.35% |

# Chapter 4

# Artificial Intelligence Visual System-on-Chip

## 4.1 YOLOv8 architecture

YOLO, an acronym for "You Only Look Once," stands out as a real-time object detection algorithm that partitions the input image into a grid. It predicts bounding boxes and class probabilities directly within each grid cell [39]. YOLOv8, the latest release in the YOLO series by Ultralytics [40], stands out for its ability to achieve a balance between accuracy and real-time inference speed, distinguishing itself from earlier YOLO versions and alternative object detection architectures. YOLOv8 exhibits competitive performance when evaluated against other state-of-the-art models such as EfficientDet and Cascade R-CNN. To make it easier for readers to understand our hardware design, we focus on the architecture of YOLOv8 in this chapter.

### 4.1.1 Bottleneck

YOLOv8 adopts a modified CSPDarknet53 architecture by replacing C3 (CSP Bottleneck with 3 convolutions) with C2f (Faster CSP Bottleneck with 3 convolutions). BottleNet architectures leverage residual connections, inspired by ResNet structures as shown in Figure 4.1, to facilitate the flow of information through the network. These connections enable the network to efficiently learn and propagate relevant information while mitigating the risk of vanishing or exploding gradients during training.

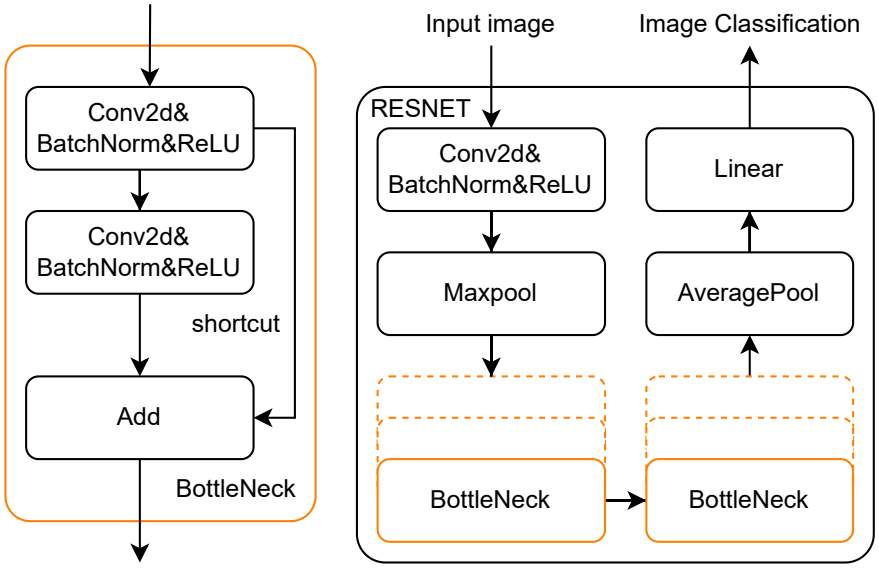



Figure 4.1: Structure of ResNet and Bottleneck

In CSPDarknet53, C3 functions as a module for constructing feature extraction layers [41]. The C2f structure addresses the challenge of insufficiently fused defect information. Illustrated in Figure 4.2, the C2f structure divides the input feature map into two segments, processes them through two branches, and subsequently merges the branch outcomes. This amplifies feature representation, facilitating information flow across diverse branches and strengthening the fusion effect of bearing defect features. Some C2f structures include shortcuts in the Bottleneck, while others do not. The 2D convolution employs batch normalization and utilizes the Sigmoid Linear Unit (SiLU) activation function to introduce non-linearity.The variable $N$ is dependent on model sizes, such as YOLOv8n (nano), YOLOv8s (small), YOLOv8m (medium), YOLOv8l (large), and YOLOv8x (extra-large).

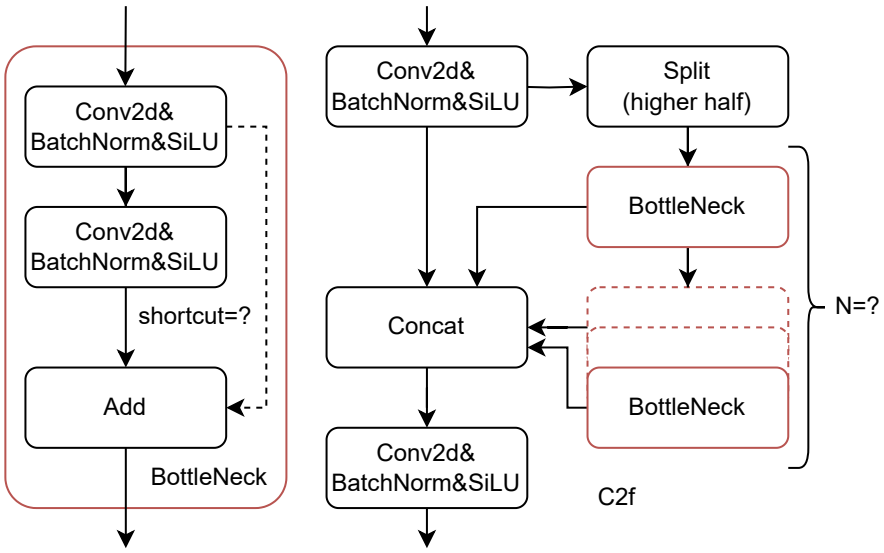



Figure 4.2: Structure of C2f and Bottleneck

### 4.1.2 Spatial Pyramid Pooling Fast (SPPF)

The YOLOv8 architecture incorporates the Spatial Pyramid Pooling Fast (SPPF) from YOLOv5, as introduced by Glenn Jocher [42]. As illustrated in Figure 4.3, the original Spatial Pyramid Pooling (SPP) structure utilizes pooling kernels of $5 \times 5$, $9 \times 9$, and $13 \times 13$ to extract features between first convolution and concatenation layer[43], preserving spatial information while reducing the image size. In contrast, SPPF performs three consecutive pooling layers with $5 \times 5$ kernels before concatenation, significantly enhancing computational efficiency compared to traditional SPP structures.

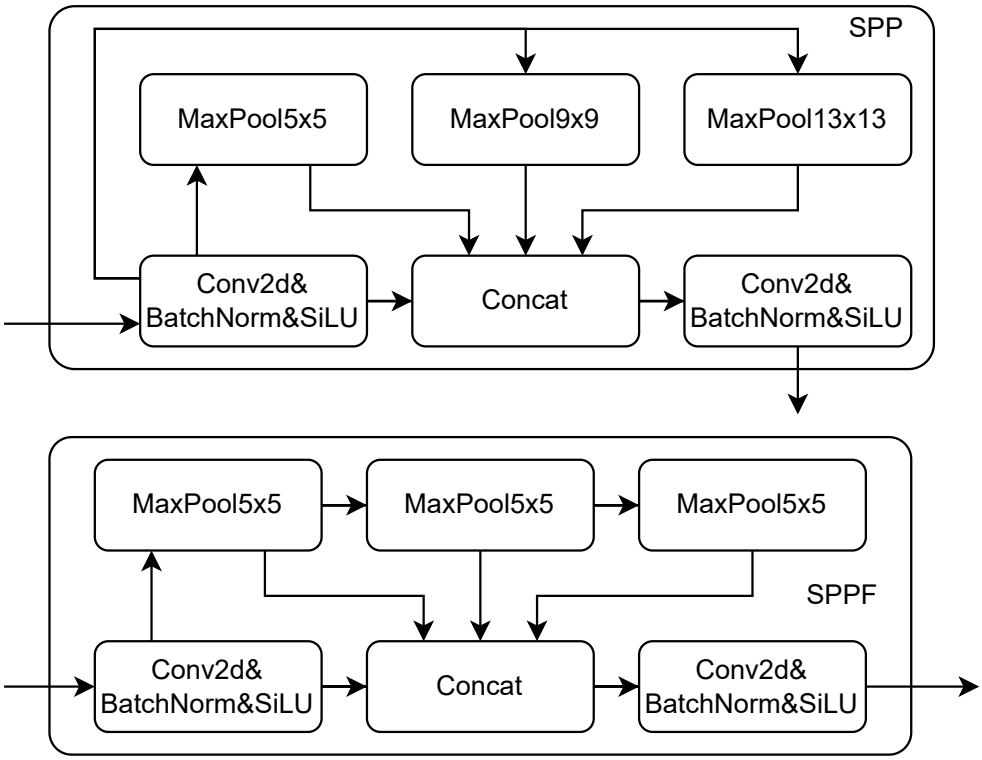



Figure 4.3: Comparism of SPP and SPPF

### 4.1.3 Distribution Focal Loss (DFL)

YOLOv8 employs classification and regression branches for its loss computation[44]. The classification branch utilizes Binary Cross-Entropy (BCE) loss, while the regression branch in this paper employs the Distribution Focal Loss (DFL)[45] and Complete IoU (CIoU) loss functions. In the regression branch, YOLOv8 utilizes the integral module proposed in [46], illustrated in Figure 4.4, where $C$ represents the channel, and the default value for $reg\_max$ is set to 16. The output of DFL comprises only 4 channels, representing the box's position ($x$, $y$) and dimensions ($w$, $h$) before being located in the grid cell.

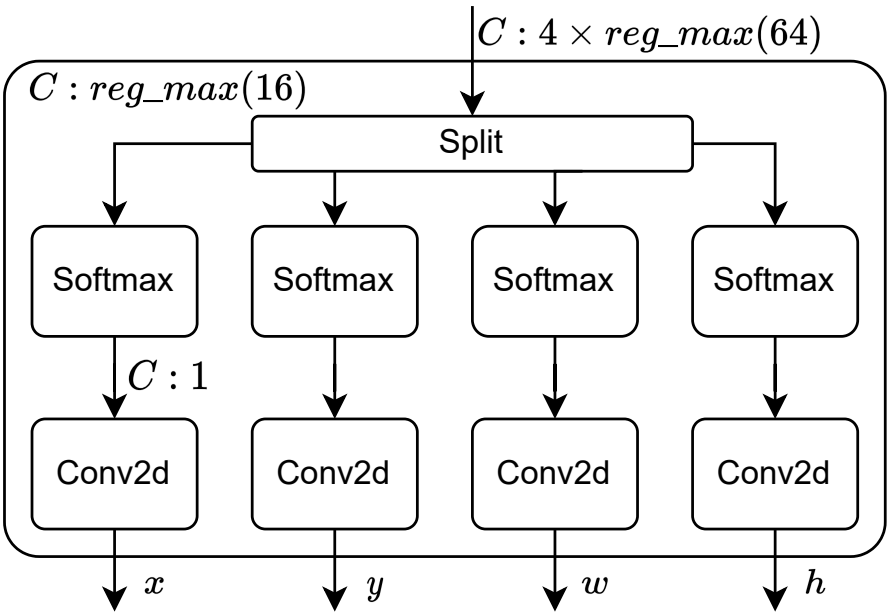



Figure 4.4: Structure of integral DFL

### 4.1.4 Path Aggregation Network

YOLOv8 supports dynamic input resizing with image dimensions represented as $S_x \times S_y$, where both $S_x$ and $S_y$ must be integer multiples of 32. This improves adaptability to various object scales during inference.

The PANet, or Path Aggregation Network [47], functions as the core backbone and head network for YOLOv8, as illustrated in Figure 4.5. PANet represents a significant advancement in instance segmentation, introducing a unique path aggregation mechanism and contextual awareness module. The Feature Pyramid Network (FPN) comprises levels $P_1$ to $P_5$, where the feature map size in layer $P_i$ is $(S_x/2^i, S_y/2^i)$. By augmenting the FPN, PANet effectively tackles the challenges of multi-scale object recognition, particularly excelling in improving object boundaries and capturing intricate details.

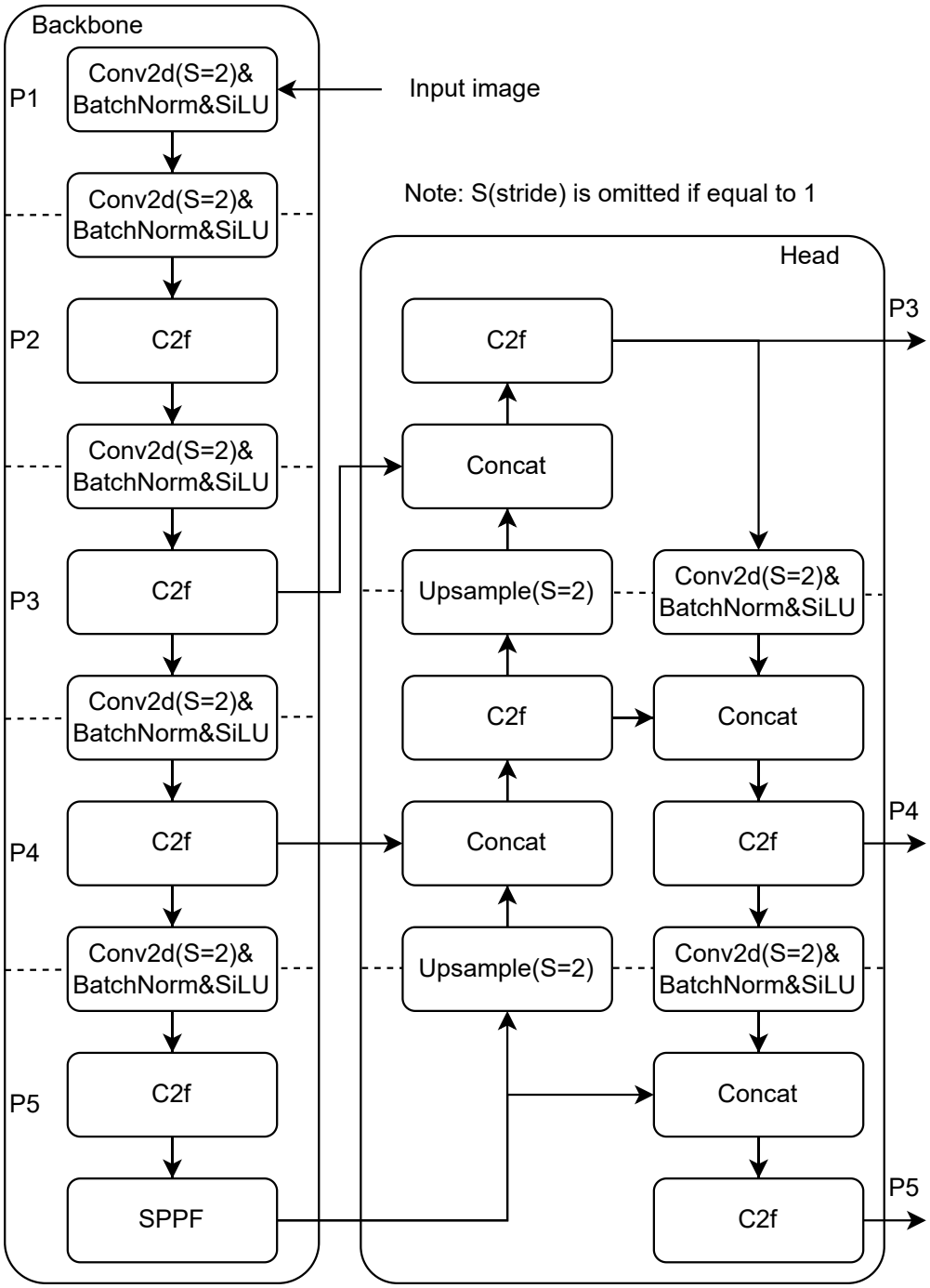



Figure 4.5: YOLOv8's backbone and head strucure

### 4.1.5 Multi-functional Head

In Figures 4.7, 4.6 and 4.8 , YOLOv8 supports multiple computer vision tasks, including object detection, instance segmentation, and pose estimation, by integrating branch module networks such as Keypoint, Mask, and prototype above the Head. In this paper, we utilize the YOLOv8 small model from [48]. The three models utilized are 'yolov8s.pt' for object detection, 'yolov8s-seg.pt' for instance segmentation, and 'yolov8s-pose.pt' for pose estimation.

We decompose complex backbone and head structures, excluding DFL, NMS, and MASK, to derive fundamental deep learning operations, as illustrated in Table 4.1.

For object detection and instance segmentation tasks, the COCO dataset [49] defines a class count ($nc$) of 80. However, in the context of pose detection within the COCO dataset, the parameter $nc$ is restricted to 1. It is important to note that other datasets may exhibit variations in the number of classes as well. These tasks address distinct objectives and find applications in various scenarios, making YOLOv8 versatile for surveillance, autonomous vehicles, and robotics.

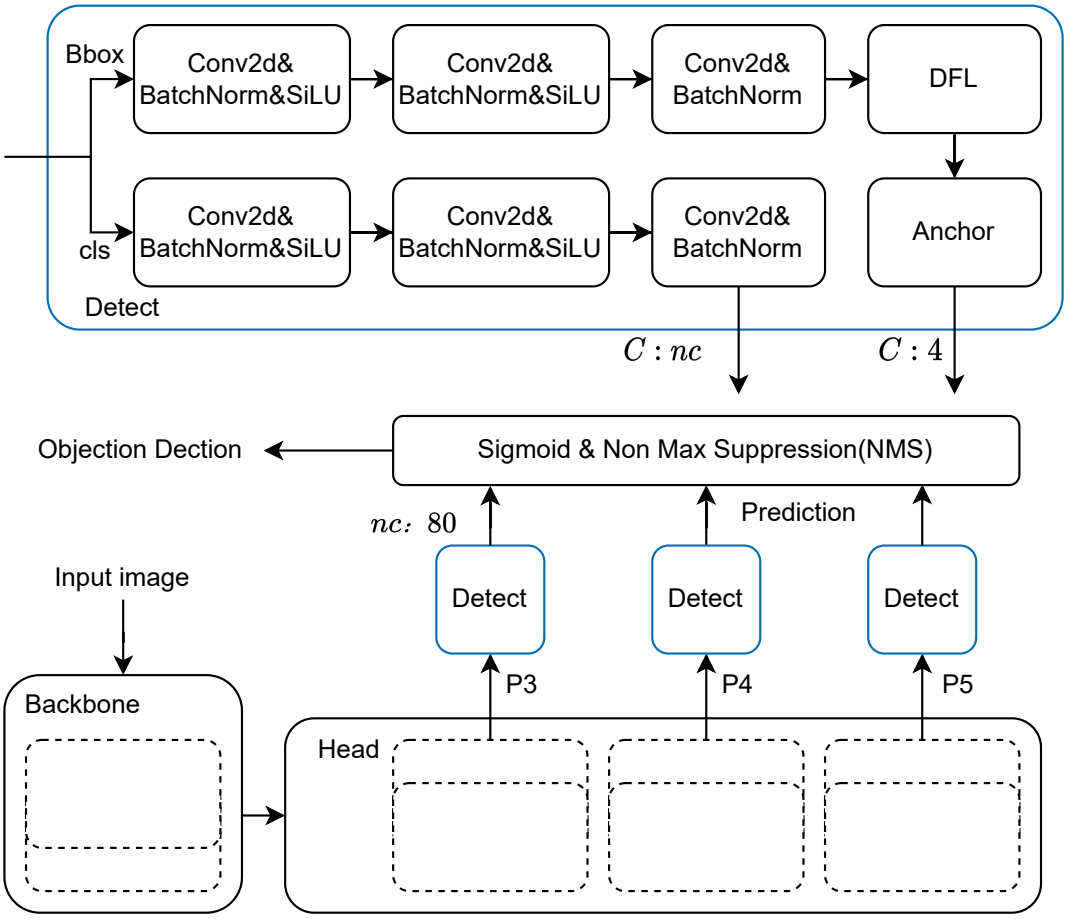



Figure 4.6: YOLOv8's structure for object detection

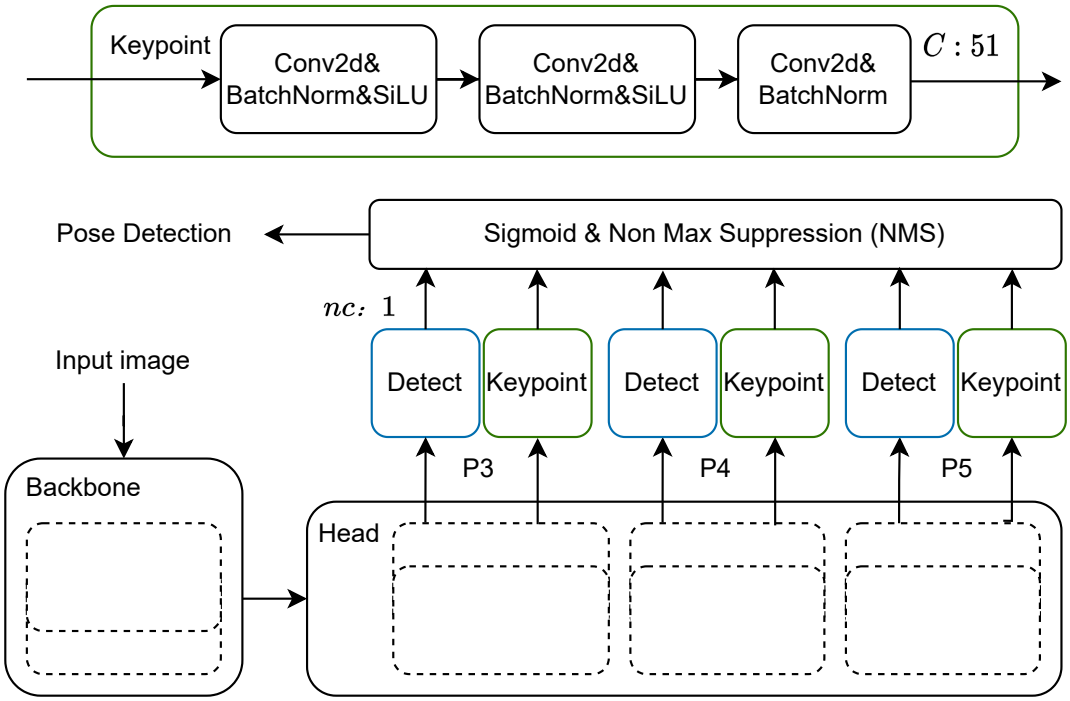



Figure 4.7: YOLOv8's structure for pose detection

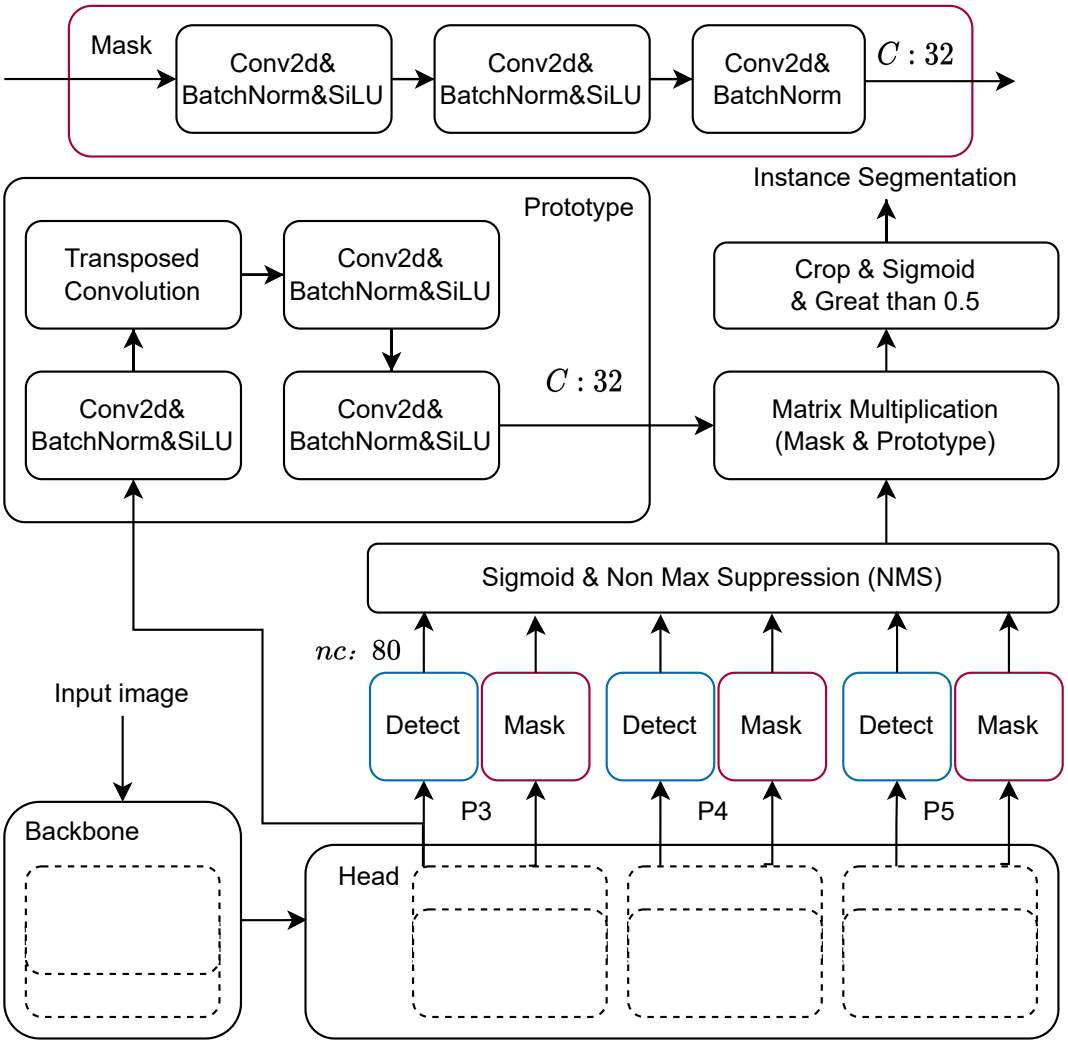



Figure 4.8: YOLOv8's structure for instance segmentation

Table 4.1: Total number of basic functions(layers)

| Model | yolov8s | yolov8s-seg | yolov8s-pose |
|---|---|---|---|
| Convolution 3x3 | 39 | 47 | 45 |
| Convolution 1x1 | 32 | 36 | 35 |
| Transposed convolution | 0 | 1 | 0 |
| Maxpool 5x5 | 3 | 3 | 3 |
| Addition | 6 | 6 | 6 |
| Concat | 13 | 13 | 13 |
| SiLU | 65 | 74 | 71 |
| Sigmoid | 3 | 3 | 3 |
| Upsample | 2 | 2 | 2 |

## 4.2 Improvements and Optimizations to Computing Engine

In our newer works [17] and [19], we expanded and reformed the DNN-AS to AIV-SoC, enhancing support for more models and tasks and achieving higher throughput. As depicted in Figure 4.9, the RISC-V module, initially RV32IMA, has been upgraded to RV32IMASU, introducing extensions for both user and supervisor modes. The four data ports—the instruction cache (I-cache), data cache (D-cache), feature map caches (FMC), and weight cache—share the same 64-bit AXI4 master port. Additionally, we have integrated a memory controller and a Universal Asynchronous Receiver/Transmitter (UART) module on the AXI4 bus to streamline execution and debugging on FPGA.

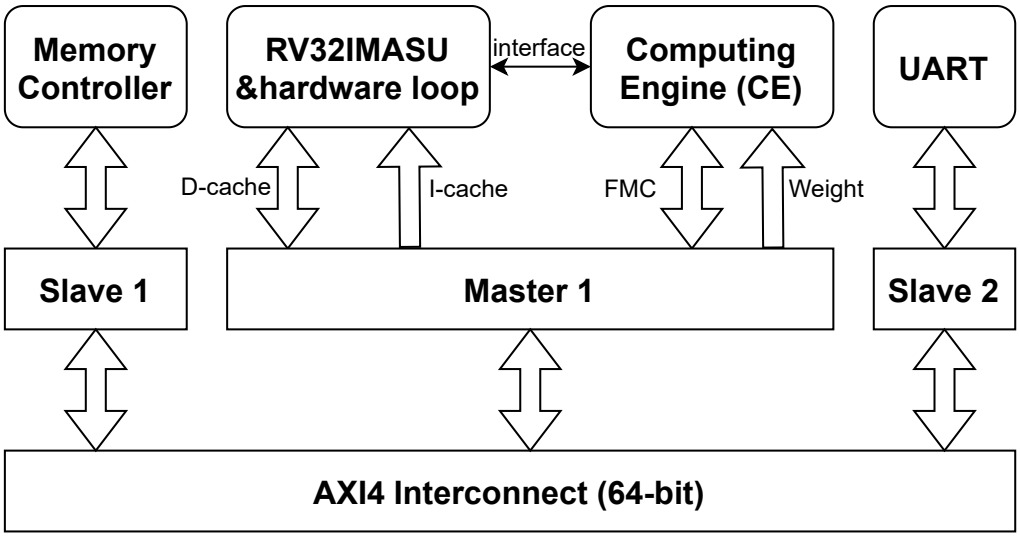



Figure 4.9: Overall architecture of AIV-SoC

In this chapter, specific modifications and enhancements have been introduced to handle the intricate and multifaceted YOLOv8. In Figures 4.1, 4.7 and 4.8, it is evident that YOLOv8 surpasses traditional ResNet networks in complexity. As a result, our AIV-SoC is equipped with a more versatile and efficient CE depicted in Figure 4.10. Table 4.2 outlines the implementation of these novel functions (layers) using reformed CE hardware or RISC-V software. Subsection 2.8 elaborates on our C2RTL-based design methodology, which facilitates easy hardware modifications. Therefore, even if there are new functionalities beyond those outlined in Table 4.2, we can readily implement them by adapting the hardware design in C++.

Table 4.2: Implementation of functions(layers)

| function/layer | implementation |
|---|---|
| SiLU | PLF circuit |
| Sigmoid | PLF circuit |
| Softmax | PLF circuit; Softmax shift,sum,multiply circuit |
| Upsample | Copy circuit |
| Concat | Copy circuit |
| Non-maximum suppression | RV32IMASU software |
| Crop | RV32IMASU software |
| Greater than | RV32IMASU software |
| Matrix multplication | General multiply-Accumulate (MAC) tree array |
| Transposed convolution | General MAC tree array |
| Add | General MAC tree array |
| Anchor | RV32IMASU software; General MAC tree array |

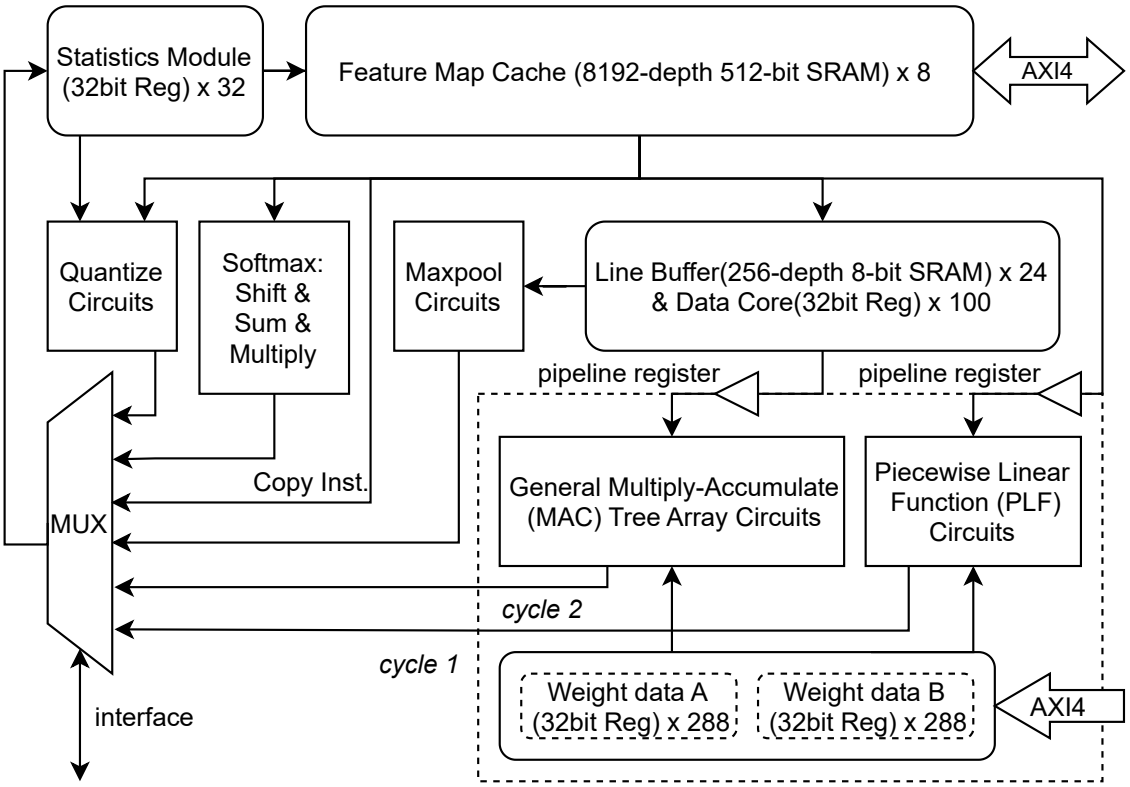



Figure 4.10: Structure of the CE

Moreover, the frequency of DNN-AS is diminished due to the prolonged critical path of the general Multiply-Accumulate (MAC) tree array. Our YOLOv8 CE in AIV-SoC tackles this issue by introducing a multi-cycle path. Concretely, we allocate the MAC tree array and its associated weight register to cycle 2, while the remaining calculation and storage modules are situated in cycle 1.

## 4.3 Maxpooling Circuits

Thanks to the new SPPF structure, we only need to focus on max-pooling with a $5 \times 5$ kernel. Taking into account register resources, we designate $P_4$ as 4 for the max-pool5x5 operation, resulting in 4 max-selection circuit trees, as illustrated in Figure 4.11.

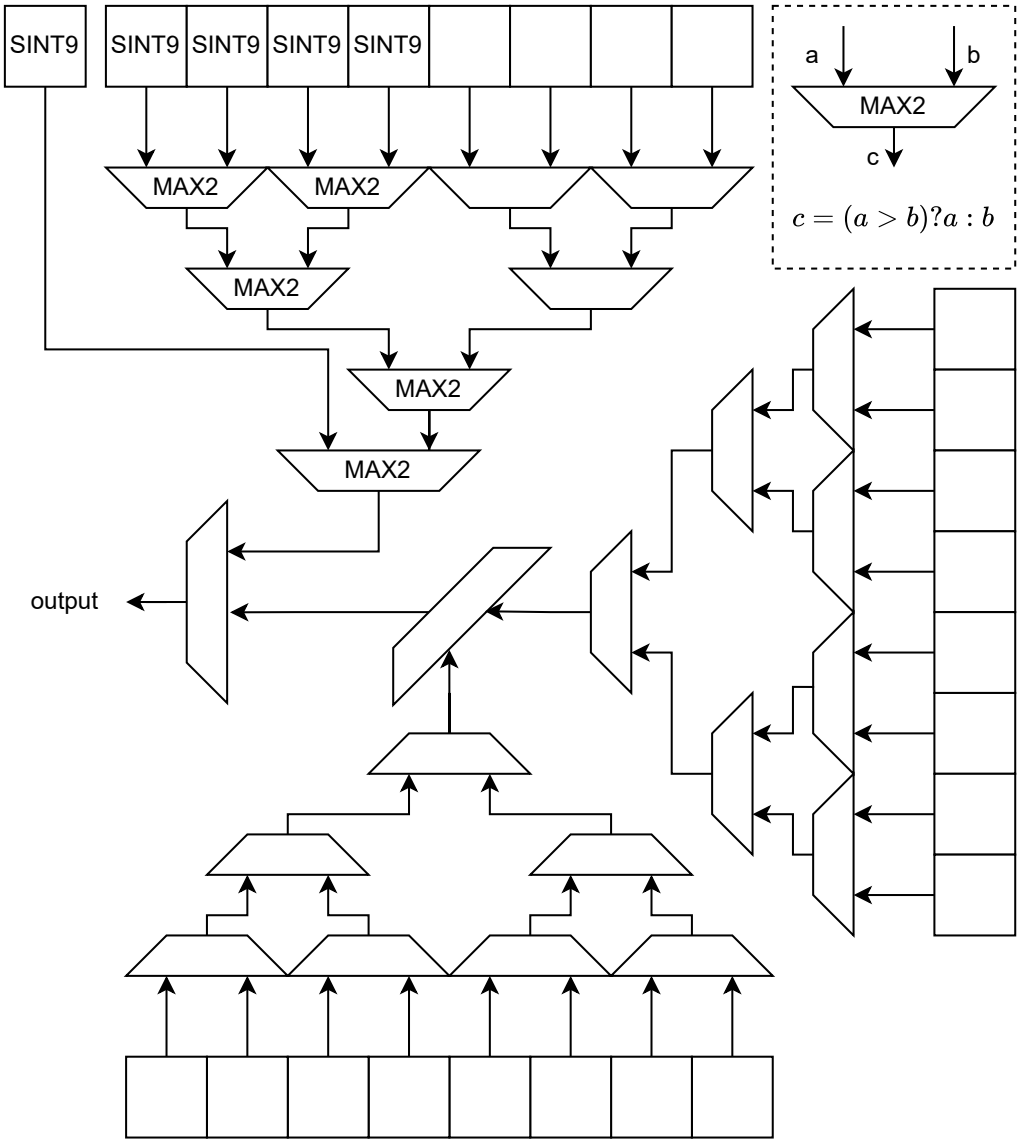



Figure 4.11: max-selection circuit tree

## 4.4 Piecewise Linear Function Module

The YOLOv8 framework incorporates various non-linear mapping functions, specifically SiLU, softmax, and sigmoid. The softmax function can be deconstructed into its inverse function and the exponential function. Traditionally, the exponential function is approximated through the Taylor series method [50]. However, due to the necessity of introducing additional multiplication and addition terms in the Taylor expansion, this significantly amplifies the overall

computational workload. To mitigate this challenge, we employ the PLF module to approximate the values of the non-linear functions.

To accommodate the input range of the majority of non-linear functions, we have divided the interval $[-8, 8]$ into 32 segments with a step size of 0.5. For each segment, we derived the corresponding slope $k$ and intercept $b$ from the weight register. Assuming that the $frac()$ function returns the fractional portion of a tensor, the resulting computation is expressed as $k \times frac(|2x|) + b$. As illustrated in Figure 4.12, a limited number of mantissa bits are adequate for accurately representing high-precision input. The outputs are quantized on a per-group basis, following the same procedure as other operations utilizing the statistics module.

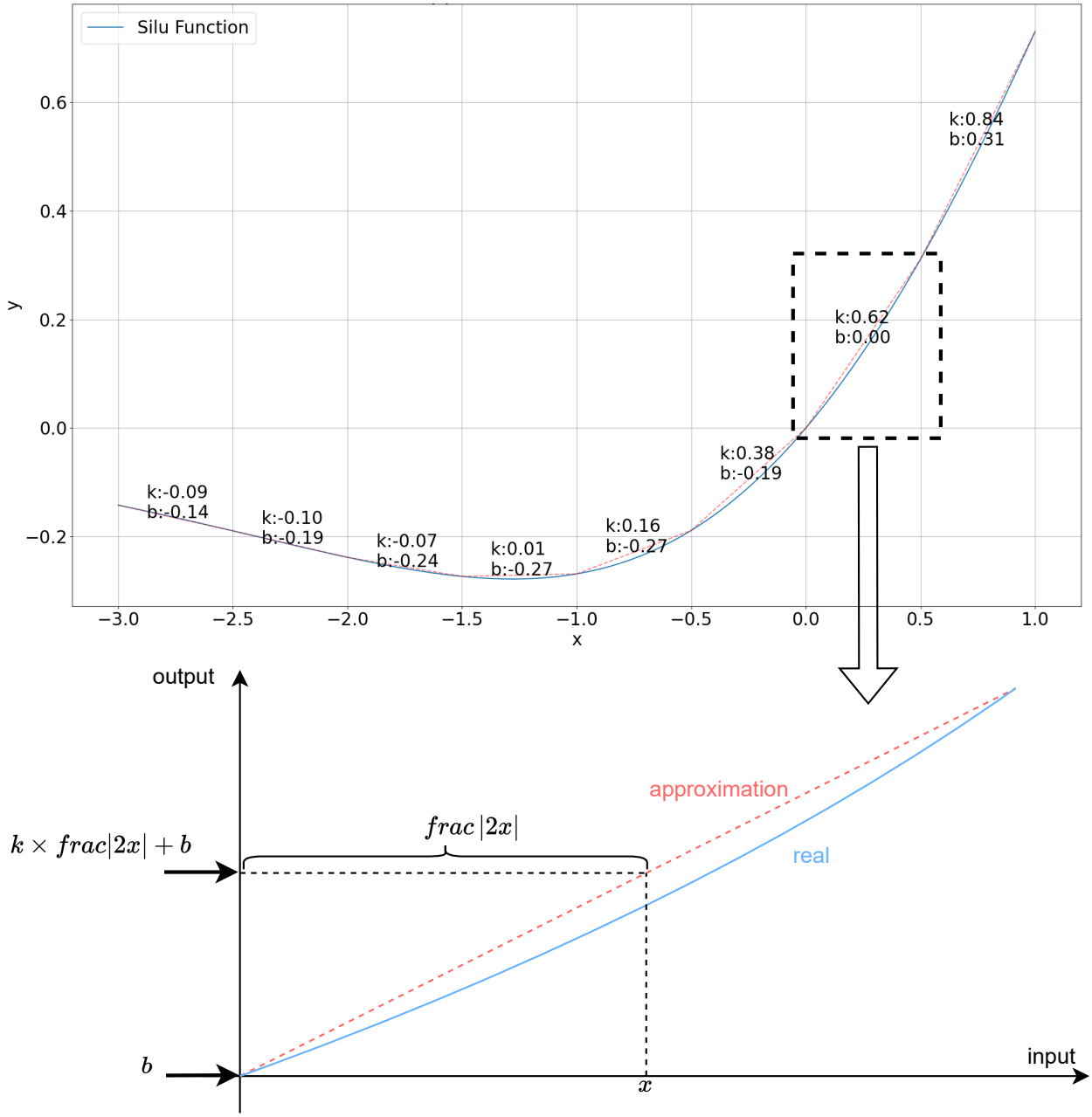



Figure 4.12: PLF approximation scheme

## 4.5 Transposed Convolution

Transposed convolution, also known as fractionally strided convolution or deconvolution, is a technique used in neural networks for up-sampling or increasing the spatial resolution of feature maps. Unlike standard convolution layers that perform down-sampling through the use of pooling operations, transposed convolution layers increase the spatial resolution of input feature maps. The transposed convolution operation involves using learnable kernels to map input pixels to a larger output space. During the transposed convolution, the kernels are applied to the input with a specific stride, resulting in the expansion of the feature map.

This operation is particularly useful in tasks such as image segmentation and image generation, where the network needs to learn to generate high-resolution details from low-resolution inputs [51]. Transposed convolution layers are often used in decoder parts of neural networks, especially in architectures like U-Net and various generative models. In terms of implementation, transposed convolution layers are typically realized using operations like "convolution with fractional input stride" in deep learning frameworks. These layers have learnable parameters (weights) that are adjusted during training to enable the network to effectively up-sample and generate detailed information from coarser representations.

It's worth noting that the term "deconvolution" in the context of neural networks is a bit of a misnomer, as it doesn't perform true deconvolution in the mathematical sense. Instead, it refers to the transposed convolution operation, which has the effect of up-sampling the input.

[52] and [53] introduces a hardware acceleration design transposed convolutions, enabling real-time execution of segmentation tasks. Given that the majority of transposed convolution in segmentation model employ a stride of 2, it implies the necessity to insert zeros between each datum in the input layer during convolution. To addresses the computational intensity and inefficiency issues, [53] introduced a dataflow exploration method by dividing filters and corresponding input feature maps into four patterns and applying the Winograd algorithm. Due to the additional latency and computational overhead introduced by image scaling, the majority of models tend to favor the first approach. The YOLOv8 architecture, employed in this study, also adopts the strategy of utilizing a kernel size divisible by the stride for its transposed convolutions, where both the stride and kernel size are set to 2.

[54] highlights another challenge known as "uneven overlap" in transposed convolution, where certain areas receive more emphasis than others, especially when the kernel size is not divisible by the stride. This uneven overlap manifests as a checkerboard-like pattern, which is more pronounced in two dimensions due to the squared effect. Two suggested approaches to mitigate these issues include using a kernel size divisible by the stride (equivalent to "sub-pixel convolution") and separating up-sampling from convolution by resizing the image before applying convolution layers.

In light of the overarching constraints and implementation considerations, we formulated the transposed convolution with a stride of 2 in CE. This design specifically mandates a kernel size that is divisible by the given stride. Similar to the four configurations delineated in [53], we have also partitioned our transposed convolution kernel into four distinct patterns as shown in Figure 4.13. Each pattern is equivalent to a traditional convolution with a stride of 1 and a kernel size of $(\frac{Nkx}{2}, \frac{Nky}{2})$. Implementing this 4-pattern mode introduces a repetition of loop2, occurring four times. As shown in Table 4.3, we listed the $P^*, N^*$ for the YOLOv8's transposed convolution in both normal and 4-pattern mode. By computing the expression in ( 4.1), where "Inst" denotes the number of instructions, and $P_i, N_i$ denote the dimensions or

unrolling variables under loop "i", it is discerned that employing the 4-pattern mode consumes only one-fourth of the number instruction compared to the normal mode.

$$Inst = \prod_1^4 \frac{N_i}{P_i} \tag{4.1}$$

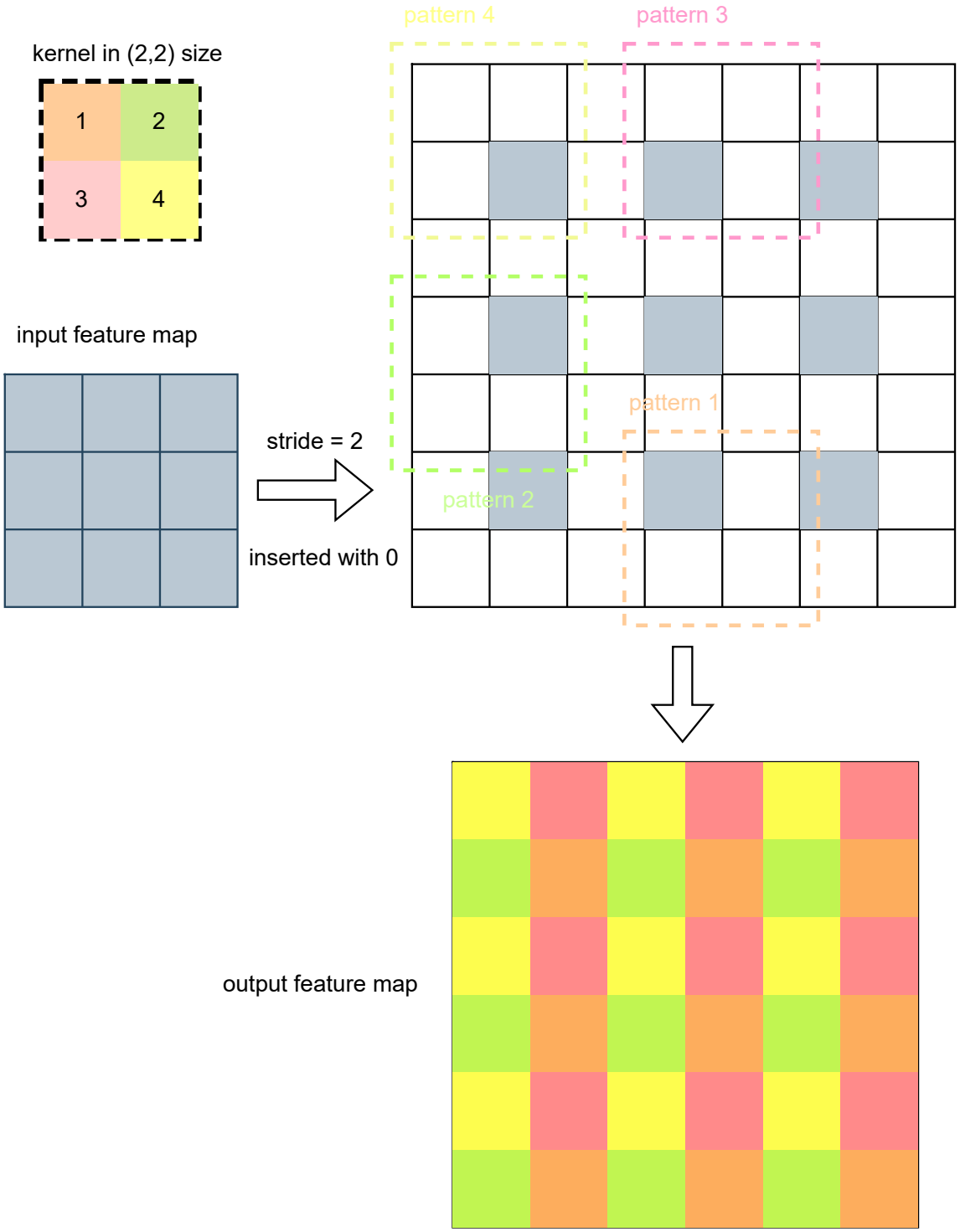



Figure 4.13: 4-pattern mode of Transposed Convolution

Table 4.3: YOLOv8's transposed convolution N* and P*

| Mode | Coefficient | Loop1 | Loop2 | Loop3 | Loop4 |
|---|---|---|---|---|---|
| normal | Dimension(N*) | (2, 2) | 128 | (2*Nix, 2*Niy) | 128 |
| | Unrolling (P*) | (2, 2) | 16 | (1, 1) | 16 |
| 4-pattern | Dimension(N*) | (1, 1) | 512 | (Nix, Niy) | 128 |
| | Unrolling (P*) | (1, 1) | 64 | (1, 1) | 16 |

## 4.6 Experimental Results

### 4.6.1 Hardware Simulation

The hardware design is implemented using C++ and compiled using LLVM 11.0.0 and C2RTL 4.2.8 versions. The C++ code is then converted into an RTL Verilog file. The Front-End hardware design is synthesized with Synopsys Design compiler. Initially, we employ the TSMC 28nm SRAM memory compiler, TSN28HPCPD127SPSRAM_20120200, to produce 8 individual single port 32-bit 8192-depth SRAM cells. Subsequently, we utilize the compiler TSN28HPCPUHDSPSRAM_20120200 to synthesize the circuit incorporating these memory cells.

The simulation constraint and outcomes for AIV-SoC in comparison to the basic RISC-V SoC without CE are illustrated in Table 4.4. In comparison to [16] and [17], our enhancements involve the incorporation of a memory controller and UART module within the AXI4 bus architecture. Consequently, the area and power consumption of the base RISC-V design have increased. However, the power differential between the AIV-SoC and the basic RISC-V SoC remains nearly constant.

Table 4.4: Hardware Resource Utilization

| | AIV-SoC | RISC-V SoC |
|---|---|---|
| Frequency | 595Mhz | 632Mhz |
| Clk period | 1.68ns | 1.58ns |
| Area | $7.91mm^2$ | $0.34mm^2$ |
| Memory size | 4,276KB | 69KB |
| Power | 202.5mW | 114.3mW |
| Temperature | 25°C | |
| Voltage | 0.9V | |
| Clk uncertainty | 0.3ns | |
| Max transition | 0.3ns | |
| Max fanout | 32 | |

Prior works ([16], [17]) adopt traditional single-cycle path designs, wherein achieving higher clock frequencies can be challenging due to the stringent timing requirements imposed on critical paths. To address this challenge, we employ a multi-cycle patch design in CE, segmenting the critical paths across two clock cycles. This approach provides flexibility in meeting timing constraints and opens the possibility of achieving higher clock frequencies. Consequently, the SoC presented in this paper attains a higher frequency than those in the referenced works.

Furthermore, the flexibility offered by the multi-cycle path design contributes to improved design scalability. The segmentation strategy can be readily adapted to accommodate changes in technology, environment, or other design constraints.

To underscore the power efficiency of our design, we also conducted a comprehensive comparison between our AIV-SoC and other contemporary platforms. This comparative analysis centered on the average energy consumption and frame processing time when handling unprocessed images with YOLOv8s, featuring a 352-input image size. To show our AIV-SoC's compatibility, we listed the results from resnet50. although throughput is not as good as YOLOv8.

Given the absence of existing literature on FPGA implementation of YOLOv8, we referred to related models such as YOLOv2 [32] and YOLOv7-tiny [26]. We also present the results for the GTX1080Ti (GP102) using the official PyTorch API function *val*() from [48]. Its using the default COCO dataset. The input resolution of the measurement is 352 and the default batch size is 16. Since each platform deploys different models, we standardized the evaluation metrics to ensure fairness, utilizing throughput (TOP/s) and energy consumption per billion operations (Energy per GOPs) to assess performance. To begin, we utilize the "thop" library to acquire the FLOPs (Floating Point Operations Per Second) of each model. By encompassing the entire execution cycle and frequency, we can derive the throughput of the platform. Subsequently, dividing the power by the throughput enables us to ascertain the Energy per GOPs.

Due to the absence of chip fabrication and validation, as well as the cost implications tied to chip production volume, we are currently unable to estimate the retail price of our AIV-SoC. As a reference point, we have compared it with the K230 chip, the latest generation SoC product in Canaan Technology's Kendryte series of AIOT chips,It has 2 CPU with max 800Mhz and 1600hz respectively, it can deal with many AI model including YOLOv5. In Table 4.5 we present the result with int8 yolov5s. The k230 is priced at approximately $50. Therefore, if our design were to enter commercial production at scale, leveraging the cost advantages of utilizing the more economical TSMC28nm process, our production costs would undoubtedly be lower.

As depicted in Table 4.5, our AIV-SoC achieved a throughput (TOP/s) equivalent to 13% of the GTX1080Ti. Moreover, our power efficiency, defined by the Energy per GOPs, was approximately 160 times superior to that of the GTX1080Ti, 3 times better than K230, and 37 times superior to the Arria-10 GX1150 FPGA. Compared to FPGA-based designs, our AIV-SoC has the potential for lower costs on a mass production scale. With these advantages, Considering these factors, our AIV-SoC has great potential for applications in lower-end devices such as autonomous vehicles or agricultural monitoring systems.

Table 4.5: Comparison of Hardware Metrics

| | This work | | | | NVIDIA GTX1080Ti | | |
|---|---|---|---|---|---|---|---|
| Frequency (MHz) | 595 | | | | 1481 | | |
| Technology | TSMC 28nm | | | | TSMC 16nm | | |
| Bandwidth(GB/s) | 4. 76 | | | | 484 | | |
| Precision | Int8 | | | | Float32 | | |
| Area ($mm^2$) | 7.9 | | | | 471 | | |
| Power (W) | 0.2 | | | | 250 | | |
| Model | yolov8s | yolov8s-seg | yolov8s-pose | resnet50 | yolov8s | yolov8s-seg | yolov8s-pose |
| Throughput(TOP/s) | 0.58 | 0.71 | 0.592 | 0.455 | 4.35 | 5.35 | 4.34 |
| Energy/GOPs (mJ) | 0.345 | 0.282 | 0.338 | 0.43 | 57.5 | 46.7 | 57.6 |

| | Arria-10 GX1150[32] | XC7Z7100[26] | K230[56] | XC7Z045[36] | ASIC[35] |
|---|---|---|---|---|---|
| Frequency (MHz) | 204 | 100 | 800/1600 | 150 | 962 |
| Technology | Intel 20nm | TSMC 28nm | TSMC 12nm | Xilinx 28nm | TSMC 65nm |
| Precision | Int8 | Int13 | Int8/16 | Int16 | Int16 |
| Power (W) | 26 | 10.79 | 1 | 9.63 | 0.394 |
| Model | yolov2 | yolov7-tiny | yolov5s | VGG16 | VGG16 |
| Throughput(TOP/s) | 1.98 | 0.918 | 0.912 | 0.137 | / |
| Energy/GOPs (mJ) | 13.12 | 11.72 | 1.09 | 70.0 | 4.31 |

### 4.6.2 COCO Objection Detection Simulation


(a) Per-group Quantization

(b) Float

Figure 4.14: COCO objection detection mAP loss of quantization

Compared to the PLF, Int8 quantization introduces a higher level of error. In this paper, we are undertaking a comparative analysis among three distinct quantization models: per-tensor, per-group, and per-channel INT8 quantization models, alongside the original official model in

[48]. Table 4.6 presents a juxtaposition of the mean Average Precision (mAP) scores at input resolutions of 256, 352, and 640 for YOLOv8s Object Detection on the COCO [49] val2017 dataset. The simulation results of an exemplary image at an input resolution of $352 \times 352$ are presented in Figure 4.14, utilizing both the AIV-SoC's per-group quantization model and the original floating-point model, visualizing a 0.8% mAP gap.

Table 4.6: YOLOv8s mAP 0.5-0.95 of objection detection

| Image size | Quantization | mAP 0.5-0.95 | | | |
|---|---|---|---|---|---|
| | | all | small | medium | large |
| 640 | Float | 44.9% | 25.7% | 49.9% | 61.0% |
| | Per-channel | 44.5% | 25.5% | 49.6% | 60.6% |
| | Per-group | 43.8% | 24.9% | 48.8% | 60.0% |
| | Per-tensor | 43.2% | 24.5% | 48.0% | 59.2% |
| 352 | Float | 37.3% | 13.6% | 40.8% | 61.7% |
| | Per-channel | 37.1% | 13.3% | 40.6% | 61.6% |
| | **Per-group** | 36.5% | 12.8% | 39.7% | 60.9% |
| | Per-tensor | 36.0% | 12.7% | 38.8% | 59.8% |
| 256 | Float | 31.3% | 8.0% | 32.8% | 57.6% |
| | Per-channel | 31.1% | 8.0% | 32.3% | 57.2% |
| | Per-group | 30.6% | 7.6% | 31.6% | 56.1% |
| | Per-tensor | 30.0% | 7.5% | 31.0% | 55.0% |

### 4.6.3 Global Wheat 2000 Objection Detection Simulation

The Global Wheat 2000 [55] dataset is pivotal for wheat genetics and agricultural research. It offers high-resolution images and detailed annotations, enabling researchers to develop algorithms for various tasks like wheat variety identification and disease detection. Additionally, the Global Wheat Head Dataset is widely used for wheat head detection tasks, providing diverse images for plant phenotyping and crop management. These datasets play a crucial role in advancing wheat production and food security initiatives.

We trained the Global Wheat 2000 object detection model using the YOLOv8s architecture, employing the same training methodology as the example program detailed in [48], with 100 epochs and a default input image size of 640. In the Global Wheat 2000 dataset, the number of classes ($nc$) is 1, which impacts the head detection process. Consequently, the overall inference time is slightly smaller compared to the COCO val2017 dataset. However, during our testing, the time difference on both AIV-SoC and GTX1080ti devices is negligible. Therefore, we only present the mAP table and a pair of example image.

While training with an image size of 352 may yield better precision for inference at 352 resolution, it is noteworthy that deploying pre-trained generic models tailored to specific resolutions presents challenges. Hence, we opted to adhere to the default resolution of 640 for training, aligning with typical real-world application scenarios.

Table 4.7: YOLOv8s objection detection mAP in Global Wheat 2000

| Image size | Quantization | mAP 0.5-0.95 | mAP 0.5 |
|---|---|---|---|
| 640 | Float | 70.6% | 98.2% |
| | Per-channel | 69.5% | 98.2% |
| | Per-group | 69.4% | 98.1% |
| | Per-tensor | 68.3% | 98.1% |
| 352 | Float | 64.1% | 96.3% |
| | Per-channel | 63.1% | 96.1% |
| | **Per-group** | 62.2% | 95.7% |
| | Per-tensor | 60.5% | 95.7% |
| 256 | Float | 50.2% | 88.6% |
| | Per-channel | 49.6% | 88.3% |
| | Per-group | 49.1% | 87.9% |
| | Per-tensor | 47.4% | 87.4% |


(a) Per-group Quantization


(b) Float

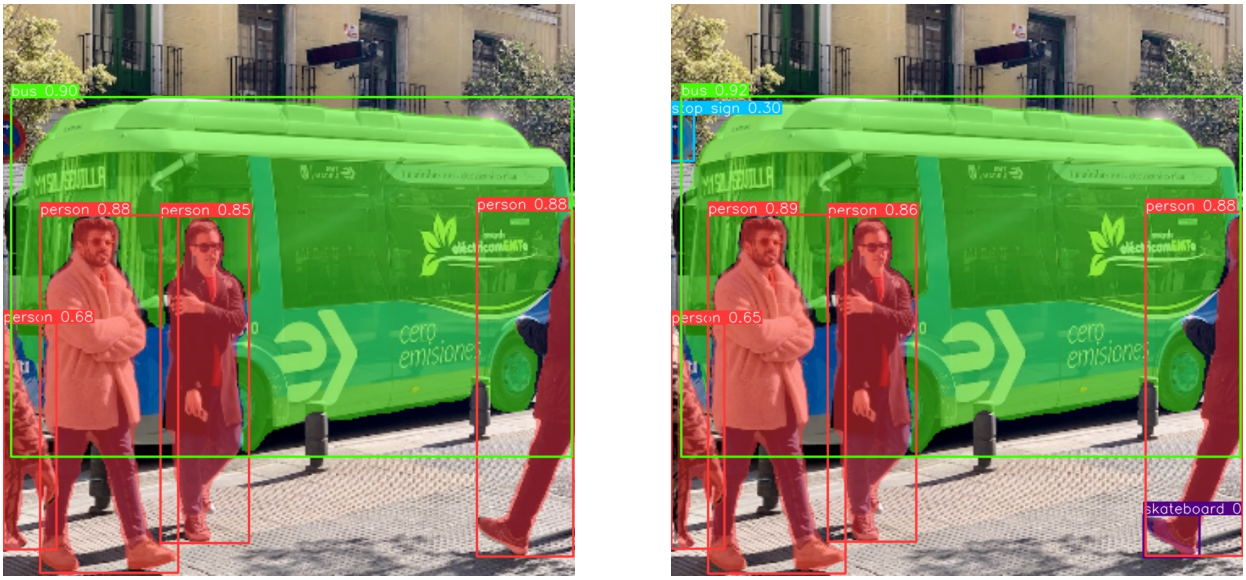
Figure 4.15: Global Wheat 2000 objection detection mAP loss of quantization

### 4.6.4 COCO Instance Segmentation Simulation

In parallel with Table 4.6, we conducted a comparative analysis of four distinct quantization models applied to the YOLOv8 instance segmentation model, as illustrated in Table 4.8. The test dataset is COCO-seg [49], an extension of the COCO dataset. Given the absence of prior research dedicated to the instance segmentation of YOLO, our power efficiency comparison is limited to our AIV-SoC and the 1080Ti, as presented in Table 4.5. Similar to Figure 4.14, Figure 4.16 visualizes the 0.8% mAP loss observed in instance segmentation when using an image size of 352, attributable to quantization.

Table 4.8: YOLOv8s instance segmentation mAP 0.5-0.95 in COCO

| Image size | Quantization | mAP 0.5-0.95 | | | |
|---|---|---|---|---|---|
| | | all | small | medium | large |
| 640 | Float | 36.8% | 17.0% | 41.2% | 53.7% |
| | Per-channel | 36.3% | 16.9% | 40.9% | 53.3% |
| | Per-group | 35.9% | 16.0% | 40.6% | 52.9% |
| | Per-tensor | 35.4% | 16.1% | 40.0% | 52.6% |
| 352 | Float | 30.3% | 8.1% | 32.6% | 53.0% |
| | Per-channel | 30.1% | 8.0% | 32.3% | 53.1% |
| | **Per-group** | 29.5% | 7.8% | 31.8% | 52.2% |
| | Per-tensor | 29.1% | 7.7% | 31.2% | 51.8% |
| 256 | Float | 24.8% | 4.4% | 24.3% | 48.8% |
| | Per-channel | 24.5% | 4.1% | 23.9% | 48.5% |
| | Per-group | 24.1% | 3.9% | 23.3% | 48.2% |
| | Per-tensor | 23.8% | 3.9% | 22.9% | 47.2% |




(a) Per-group Quantization

(b) Float

Figure 4.16: COCO instance segmentation mAP loss of quantization

### 4.6.5 Crack Instance Segmentation Simulation

The Roboflow Universe Crack [57] Segmentation Dataset emerges as an extensive repository tailored specifically for professionals engaged in transportation and public safety analyses, comprising a total of 4029 static images captured from diverse road and wall scenarios. Crack segmentation holds practical relevance in infrastructure upkeep, facilitating the detection and evaluation of structural deterioration. Moreover, it plays a pivotal role in bolstering road safety by empowering automated systems to identify and address pavement cracks promptly for timely repairs.

The number of classes ($nc$) for the Roboflow Universe Crack instance segmentation model is 1. We trained it using the same methodology employed for the Global Wheat 2000 dataset. Additionally, the overall inference time difference between COCO instance segmentation and our model is negligible. Furthermore, we present the mean Average Precision (mAP) table alongside a pair of example images. Due to the relatively small size of most images in the Roboflow Universe Crack dataset (often below a resolution of 416x416), employing an image resolution of 352 results in superior overall precision compared to using 640.

Table 4.9: YOLOv8s instance segmentation mAP in Roboflow Universe Crack

| Image size | Quantization | mAP 0.5-0.95 | mAP 0.5 |
|---|---|---|---|
| 640 | Float | 22.7% | 64.6% |
| | Per-channel | 22.2% | 62.2% |
| | Per-group | 22% | 64.1% |
| | Per-tensor | 21.5% | 63.5% |
| 352 | Float | 25.6% | 71.3% |
| | Per-channel | 25.1% | 69.7% |
| | **Per-group** | 25.6% | 73.4% |
| | Per-tensor | 26.0% | 73.5% |
| 256 | Float | 25.0% | 64.7% |
| | Per-channel | 24.8% | 64.5% |
| | Per-group | 25.2% | 66.5% |
| | Per-tensor | 25.6% | 66.0% |


(a) Per-group Quantization (b) Float

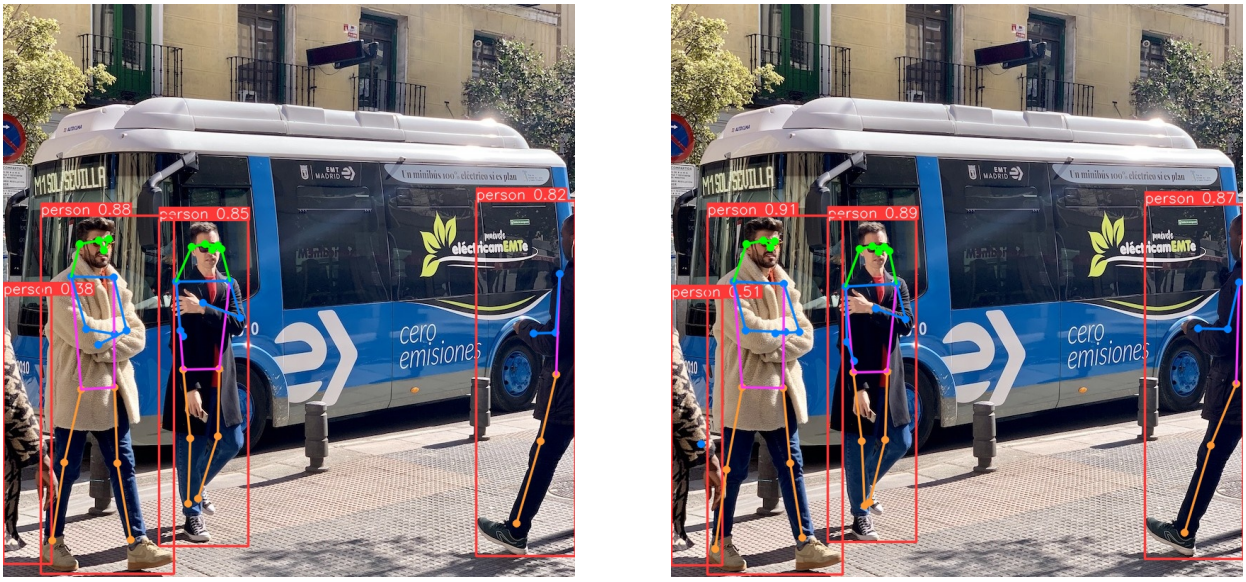
Figure 4.17: Roboflow Crack Instance Segmentation mAP loss of quantization

### 4.6.6 COCO Pose Detection Simulation

In a similar manner, we extended our investigation to YOLOv8 pose detection. We performed a comparative analysis involving four distinct quantization models, as portrayed in Table 4.10. The test dataset is COCO-Pose [49], a specialized version of the COCO dataset, designed for pose estimation tasks. Due to the absence of prior research dedicated to the pose detection of YOLO, our power efficiency analysis is singularly focused on a comparative evaluation between our AIV-SoC and the 1080Ti, as elucidated in Table 4.5. Consistent with Figure 4.14, Figure 4.18 illustrates the simulation outcomes for pose detection across 2 models at an image size of 352. Notably, there is a certain degree of mAP loss (5.4%) in pose detection, which may be attributed to its unique key-point branch. The adoption of Int8 quantization results in a greater loss of information compared to other fully convolution networks (FCN). In our future work, we will explore the feasibility of employing per-group and per-channel hybrid quantization.

Table 4.10: YOLOv8s pose detection mAP 0.5-0.95 in COCO

| Image size | Quantization | mAP 0.5-0.95 | | |
|---|---|---|---|---|
| | | all | medium | large |
| 640 | Float | 59.9% | 53.7% | 70.3% |
| | Per-channel | 58.7% | 52.2% | 69.1% |
| | Per-group | 53.5% | 45.2% | 65.7% |
| | Per-tensor | 48.9% | 41.3% | 60.1% |
| 352 | Float | 47.4% | 34.7% | 65.5% |
| | Per-channel | 45.9% | 33.1% | 64.3% |
| | **Per-group** | 42.0% | 29.9% | 58.8% |
| | Per-tensor | 36.7% | 27.2% | 50.7% |
| 256 | Float | 37.0% | 21.9% | 58.2% |
| | Per-channel | 36.0% | 21.2% | 57.1% |
| | Per-group | 32.4% | 19.4% | 51.0% |
| | Per-tensor | 28.0% | 17.9% | 42.7% |




(a) Per-group Quantization

(b) Float

Figure 4.18: COCO Pose detection mAP loss of quantization

# Chapter 5

# AI Application Software Design Using RISC-V Custom Instructions

In this study, we employed the pre-trained YOLOv8s model from Ultralytics to serve as the neural network running on the AIV-SoC platform. To optimize the network architecture, we integrated the batch normalization layers with convolution layers using the official 'fuse()' function.

## 5.1 Model Adjustments & Quantization

The YOLOv8's weights are quantized using the same quantization scales as feature map data. For example, the convolution layer's weight data consists of $L_w = Nkx \times Nky \times Nif \times Nof$ data points. The size of $L_w$ becomes prohibitively large as the dimensions of $Nif$ and $Nof$ increase in the latter part of YOLOv8. Consequently, we propose dividing them into groups to reduces the number of data sharing the same public exponent. Similar to layer's per-group quantization, each group have $G_w = Nkx \times Nky \times Nif \times 16$ data points.

As this paper primarily concentrates on the hardware design for model inference, we choose a straightforward post-training symmetric static quantization of the model. In the future, incorporating Quantization Aware Training (QAT) could potentially enhance accuracy further.

Notably, while per-channel quantization achieves accuracy closest to the original float, it necessitates a substantial number of exponents. Additionally, given that the coefficients for the majority of computation layer loop4 are set at 16, per-channel quantization will result in increased computational load within the loop. In light of the inherent trade-off between precision and circuit complexity, we have, therefore, incorporated per-group quantization into the AIV-SoC framework.

## 5.2 Model Configuration

Different from conventional hetergous architecture which use AXI protocol to control the acceleartor, we directly use custom instruction to control the CE and configure the model's parameter as shown in Figure 5.1. This figure demonstrates the parameter settings for a portion of a convolutional network; parameters for other models will vary accordingly.This approach is easier and more practical, eliminating the need to consult complex manuals, and

aligns more closely with the RISC-V design philosophy.

```
int __attribute__((noinline))conv3(int sizein,int stride,int in_channel,int out_channel,.....){
int sizeout = (stride==2)?sizein/2:sizein;
int ini_loop = sizein*1+1;
asm volatile(".insn r 11, 0, 3, x0, %0, %1"::"r"(sizein),"r"(sizeout));
asm volatile(".insn r 11, 0, 6, x0, %0, %1"::"r"(in_channel),"r"(out_channel));
asm volatile(".insn r 11, 2, 4, x0, x0, x0"::); //reset
asm volatile(".insn r 11, 0, 8, x0, x0, %0"::"r"(stride));
asm volatile(".insn r 43, 1, 0, x0, %0, x0"::"r"(ini_loop));
   asm volatile(".insn r 11, 1, 3, x0, x0, x0"::);//CE Compute command
.....
```

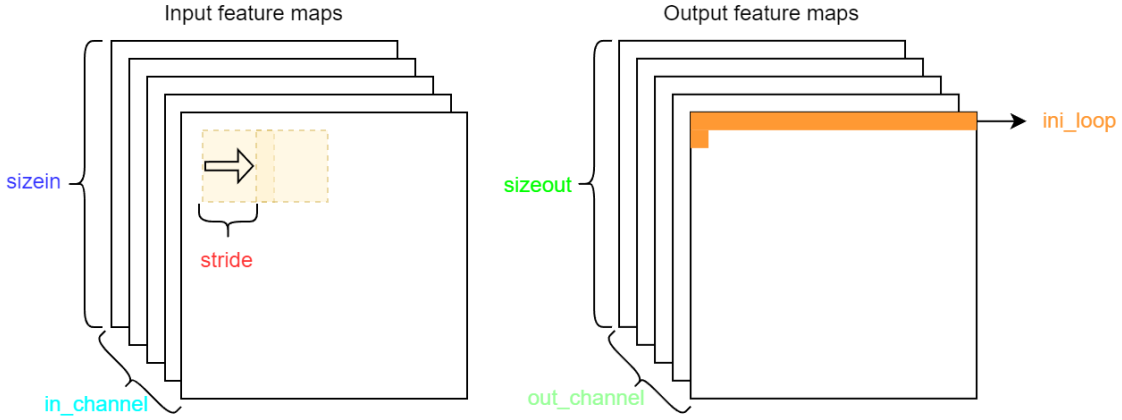



Figure 5.1: Part of C++ actual code to configure model

## 5.3 Overlap of Compute and Transfer

The overlap of compute and transfer, often implemented through a ping-pong architecture, offers significant advantages over traditional dataflow mechanisms. In a ping-pong architecture, data transfer and computation phases are interleaved, allowing one set of data to be transferred while another set is being processed. This concurrent execution minimizes idle times for processing units, thereby enhancing overall throughput and reducing latency. Traditional dataflow architectures, in contrast, typically operate in a sequential manner where data transfer and computation are distinct phases, leading to potential idle periods where processing units await data. By leveraging the ping-pong mechanism, systems can achieve higher efficiency and performance, particularly in scenarios involving large-scale data processing and real-time applications. The Figure 5.2 our advantage and how we achieve it in hardware.

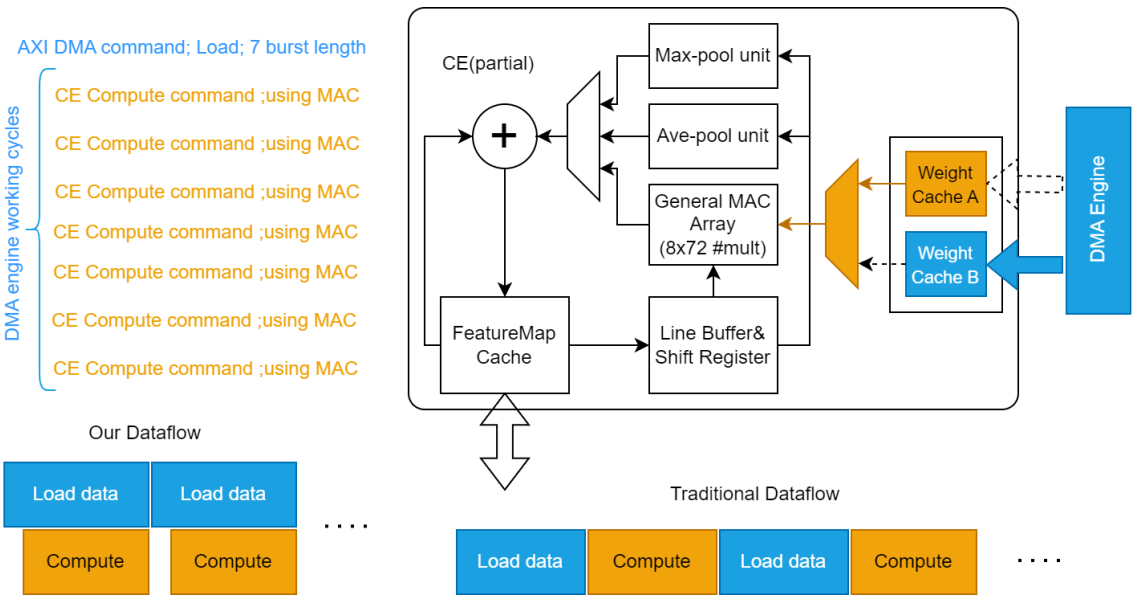



Figure 5.2: Comparison of ping-pong scheme and transitional data-flow

## 5.4 Convolution

Algorithm 2 delineates the configuration of the nested loop structure governing the convolution operation with a kernel size of $(3, 3)$. In each iteration, the *load_weights*() commands are employed to retrieve the necessary weight data for the subsequent group. In most cases, the *load_weights*() commands occurs seamlessly without disrupting the computational process, thereby enabling the concurrent execution of *calc_conv*3() alongside the weight-loading procedure.

The inner loop, featuring the *ni* statement, accumulates the partial sum from 8 adjacent input channels. As a result, we only need to save the statistics of the final iteration. Hence, we incorporate the *clean_statistics*() function to reset the data in the first register and the *save_statistics*() function to accumulate the data from the first register into the second register. Additionally, to enhance calculation efficiency, we also implement the zero overhead loop mechanism from [16] in software design.

**Algorithm 2** Convolution3x3 loop

```
1: iteration ← sizein × sizein
2: load_weights()
3: wait()
4: for no = 0 to Nof by 16 do
5:     load_bias()
6:     for ni = 0 to Nif by 8 do
7:         set_layer()
8:         clean_statistics()
9:         load_weights()
10:        for zero_over_headloop(iteration) do
11:            calc_conv3x3()
12:        end for
13:    end for
14:    save_statistics()
15:    generate_exp()
16: end for
```

## 5.5 Transposed Convolution

In Section 4.5, our approach involves employing a 4-pattern mode for transposed convolution. Within YOLOv8's segmentation model, the transposed convolution is characterized by a kernel size of 2 and a stride of 2. This operation is specifically applied to "up-sample" the prototype layer . The implementation details of this mechanism are elucidated in Algorithm 3.

**Algorithm 3** Transposed Convolution Loop

```
1: iteration ← sizein × sizein
2: load_weights()
3: wait()
4: for no = 0 to Nof by 16 do
5:     load_bias()
6:     for pattern = 0 to 3 do
7:         for ni = 0 to Nif by 64 do
8:             clean_statistics()
9:             set_layer()
10:            load_weights()
11:            for zero_over_headloop(iteration) do
12:                calc_convtranspose2x2()
13:            end for
14:        end for
15:        save_statistics()
16:    end for
17:    generate_exp()
18: end for
```

## 5.6 Matrix Multiplication

The segmentation model of YOLOv8 incorporates the task of object detection. Following the object detection process, the obtained results necessitate matrix multiplication with predictions post Non-Max Suppression (NMS) and the prototype layer. The matrix representing predictions post NMS is positioned on the left, with its variable $Nly$ denoting the count of objects remaining after NMS. Conversely, the prototype layer is situated on the right, assuming that the input image is of dimensions $(imgsx, imgsy)$. Here, $Nrx = imgsx \times imgsy/16$, and $Nlx = Nry = 32$.

To execute the matrix multiplication, predictions after NMS must initially be retrieved from the CE, followed by re-composition, and subsequently flushed back as weight data. A more in-depth understanding of this process is elucidated in Algorithm 4.

**Algorithm 4** Matrix Multiplicatio Loop

```
iteration ← Nrx
fetch_prediction_after_NMS()
reconstruct()
load_weights()
wait()
for no = 0 to Nly by 16 do
    for ni = 0 to Nlx by 64 do
        set_layer()
        load_weights()
        clean_statistics()
        for zero_over_headloop(iteration) do
            calc_convtranspose2x2()
        end for
    end for
    save_statistics()
    generate_exp()
end for
```

## 5.7 Maxpool

The spatial pyramid pooling fast (SPPF) in YOLOv8s utilizes three max-pool layers with a kernel size of 5x5. According to Table III, max-pool5x5 is composed of $Nof/4$ groups. As the max-pool operation outputs only the maximum pixel value from sliding windows in the line buffer and data core, quantization is not required. Algorithm 5 presents the pseudo code for the max-pool5x5 operation.

**Algorithm 5** Maxpool5x5 loop

```
1: iteration ← sizein × sizein
2: for no = 0 to Nof by 8 do
3:     set_layer()
4:     for zero_over_headloop(iteration) do
5:         calc_maxpool5x5()
6:     end for
7: end for
```

## 5.8 Concat & Upsample

The Concat and Upsample layers do not perform calculations; they simply copy the channels of the input layer's data. The data-flow for these layers is represented as the "Copy Inst." shown in Figure 4.10. Assuming that the first channel of the input layer will be shifted to the beginning of the output channels ($Bof$), the software algorithm is illustrated in Algorithm 6.

**Algorithm 6** Pseudo code for copy

```
1: iteration ← sizeout × sizeout
2: for no = 0 to Bof by 16 do
3:     set_layer()
4:     for zero_over_headloop(iteration) do
5:         copy()
6:     end for
7: end for
```

## 5.9 Shortcut Connection (Residual Layer)

Some bottlenecks in YOLOv8 involve shortcuts, which entail an element-wise addition of two layers. Due to the potential discrepancy in the DFP's exponents of the two input layers, we employ the MAC to perform the addition. One layer is directed to the MAC's input feature map data, while the other layer is sent to the partial sum port, where it will be appropriately shifted to align with the former. In the end, the sum result also necessitates per-group quantization. The software data-flow is illustrated in Algorithm 7.

**Algorithm 7** Pseudo code for addition

```
1: iteration ← sizein × sizein
2: load_weights()
3: wait()
4: for no = 0 to Nof by 16 do
5:     set_layer()
6:     clean_statistics()
7:     for zero_overhead_loop(iteration) do
8:         calc_add()
9:     end for
10:    save_statistics()
11:    generate_exp()
12: end for
```

## 5.10 SiLU, Sigmoid & Softmax

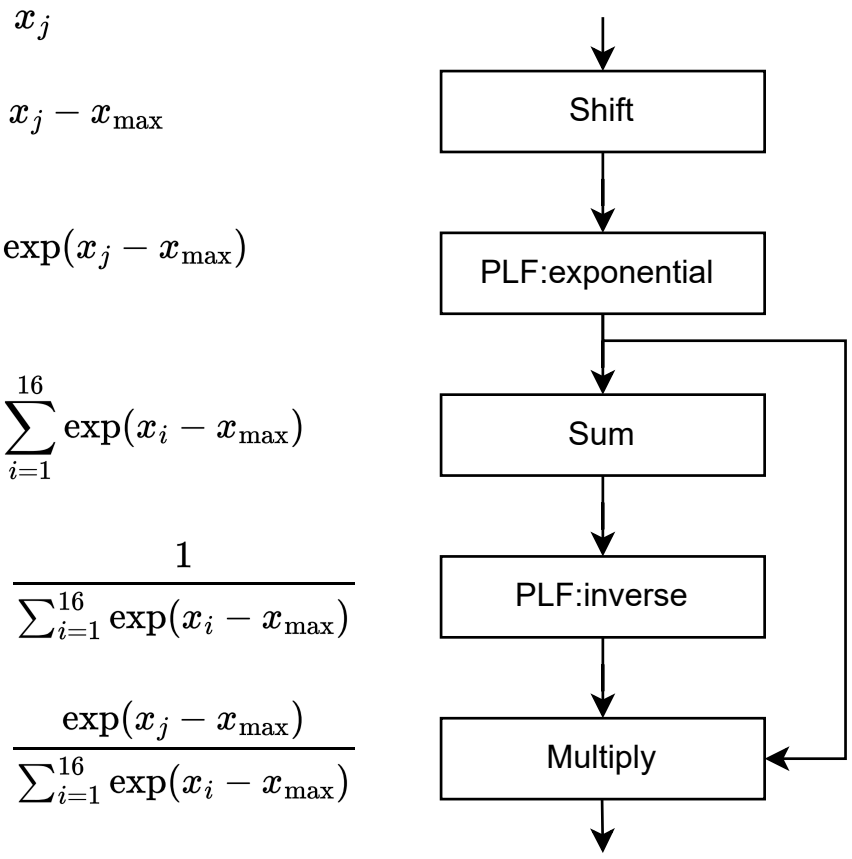



Figure 5.3: Decomposition of max-shifted softmax

YOLOv8 incorporates various non-linear mapping functions, including SiLU, softmax, and sigmoid. Modifying SiLU and sigmoid for the required task is as simple as adjusting parameters k and b within the PLF module's software. However, softmax necessitates an custom

instruction beyond PLF, as depicted in Figure 5.3. To mitigate issues related to overflow and underflow, we introduce the concept of max-shifted softmax. This technique entails subtracting the maximum value from each element before applying the exponential function, ensuring both numerical stability and computational efficiency.

## 5.11 Swap Memory

After completing each layer, the software will check if the data can be freed if the following layer does not require them. However, in some layers, there is no available space in the FMC to be freed, and the remaining memory is insufficient for the current layer. To address this issue, we have introduced a swap memory mechanism, similar to the swap memory in computers. In this context, the swap memory serves as virtual memory or paging space, which is a specialized memory area within the computer system used to temporarily store infrequently accessed feature map data when the physical on-chip SRAM becomes insufficient.

Assuming the current layer is Layer 6 with Addition operation, and its inputs are 4 and 5. Layer 2 will be used later. Layer 6 has 2 space, one the the output result of 8 bit, the other is 32-bit un-quantized data temporarily cached. Figure 5.4 depicts the operational mechanism of swap memory, involving alternate processes of addition and quantization. Upon completion of layer6 calculations, the 32-bit cache space is released, allowing Layer 2 to be flushed back into on-chip SRAM. Given the opportunity to modify YOLOv8, we would consider adjusting the layer sizes and structures to mitigate the need for swapping memory. This adjustment would allow the AIV-SoC to reduce its reliance on AXI bandwidth.

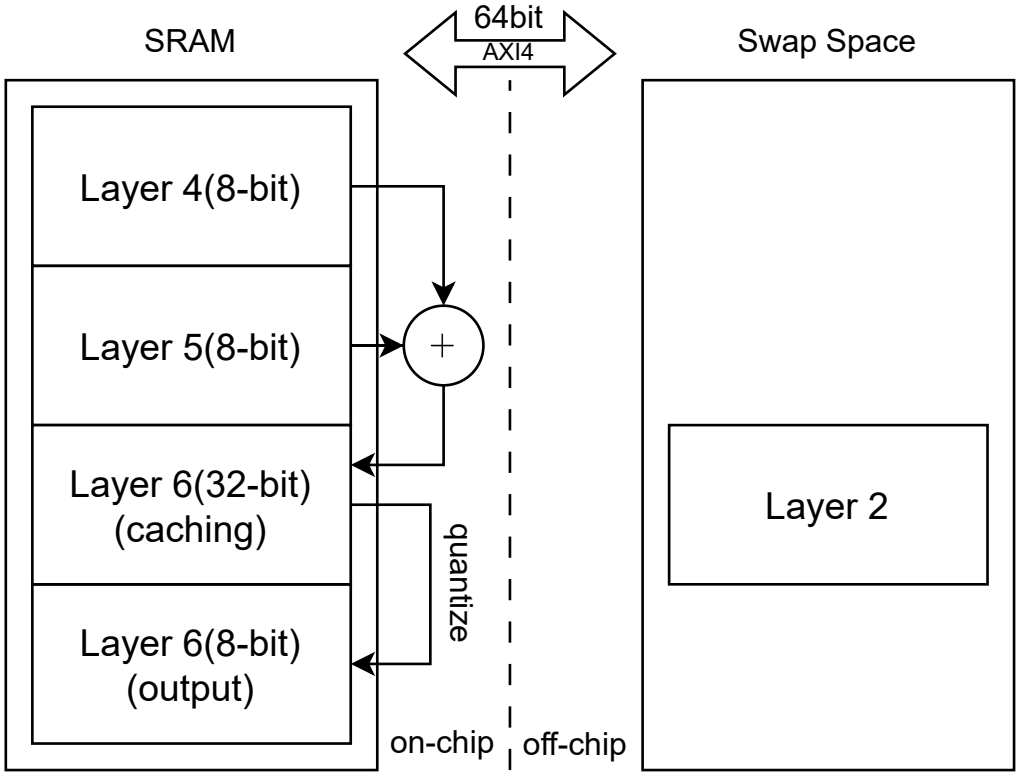



Figure 5.4: Swap memory mechanism

## 5.12 Anchor Point

YOLOv8 employs grid cells as anchors for determining box positions in an image. To account for this, grid cell bias information needs to be integrated into the boxes from the DFL layer. In Figure 4.8, there are three detect head blocks with varying layer sizes and corresponding strides: 8, 16, and 32. Consequently, the values of $(x, y, w, h)$ from the DFL are scaled by 8, 16, or 32, depending on the associated detect block. The *torch.meshgrid*() API is then utilized to generate three feature maps for grid cells, which are subsequently added to the scaled boxes.

In our AIV-SoC, leveraging the public exponent mechanism allows us to simplify the scale operation by adding an exponent of 3, 4, or 5. Following this, we create an equivalent grid cell in the off-chip swap memory. Once a softmax layer among the three is completed, its corresponding grid cell layer is transferred to on-chip SRAM. Ultimately, an addition layer, illustrated in Algorithm 7, is utilized to add the scaled boxes and grid cell. The complete data flow about anchor point is illustrated in Figure 5.5, where the dotted rectangle denotes off-chip operations.

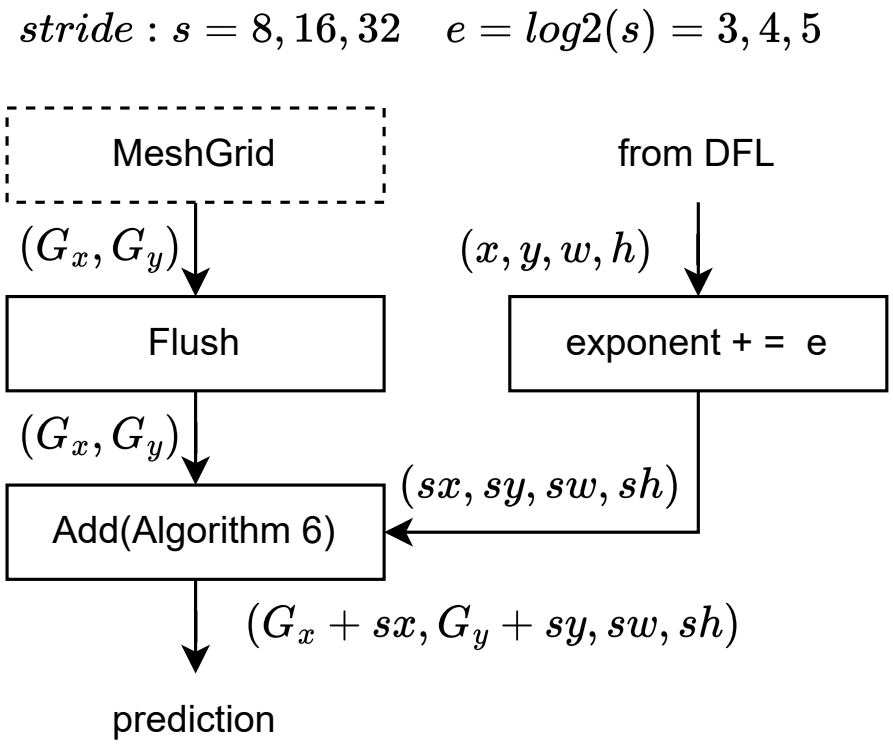



Figure 5.5: anchor point data flow

# Chapter 6

# Conclusions & Future Works

Our previous researches in DNN-AS were confined to providing support exclusively for convolution, pooling and fully connected layers. This limitation constrained its utility to elementary models like VGG and ResNet, restricting its applicability to a narrow spectrum of models and impeding its adaptability to contemporary trends in neural network evolution. Moreover, due to its exclusive consideration of image classification tasks, the maximum input resolution was confined to 256x256. Additionally, its SoC design also lacks the inclusion of a memory controller and UART module, rendering it impractical for debugging scenarios and real-world industrial applications.

In our second work, we introduce AIV-SoC, which extends support to instance segmentation and pose detection. Moreover, we have improved the input image resolution from 256x256 to 352x352, while simultaneously reducing the AXI4 bus width from 256 to 64 bits. The achieved FPS for these tasks are 67.1, 55.2, and 64.9, respectively. Despite the mAP experiencing a reduction for smaller targets, constituting half of the original 640x640 size, the mAP for larger targets remains nearly unchanged from the original 640x640 size. The intermediate-sized accuracy loss is also limited to approximately 9%. It is noteworthy that the marginal benefits of further increasing the resolution are diminishing.

In contrast to alternative architectures, our RISC-V architecture demonstrates the capability to seamlessly execute various branches of YOLOv8 on-chip, encompassing the post-processing step. Unlike ASIC or FPGA architectures tailored for CNNs, which often accommodate only object detection, our design adeptly manages YOLOv8 Object Detection, Instance Segmentation, and Pose Detection. This signifies a superior level of versatility in comparison to existing solutions. Additionally, we provide benchmark results for specific areas, a feature lacking in their designs. Furthermore, as illustrated in Table 4.5, our architecture exhibits outstanding power efficiency compared with other platforms.

Compared with the Nvidia GTX1080Ti GPU, our accelerator achieves only 0.8% of the power consumption while generating 13.3% of the throughput for the YOLOv8s model. In other words, our energy efficiency is 166 times better than the GTX1080Ti. Additionally, our design boasts adaptability and scalability, facilitated by the use of the C2RTL design toolkit and a modular RISC-V platform. These features empower us to efficiently design and simulate hardware. As indicated in [2], YOLO can be enhanced with the Swin Transformer, augmenting the environmental perception capabilities of autonomous vehicles. This suggests that we can further customize instructions to accommodate increasingly intricate neural networks in the future.

Compared to commercial embedded accelerators such as those in [22], [24], [56], and [23],

our approach uses custom instructions to handle various layers instead of relying on vector instructions. This makes our product easier to use, as it eliminates the need for additional hardware drivers and simplifies programming and debugging. Furthermore, as shown in Figure 6.1, unlike traditional ASIP architectures, our custom instruction set employs deep CISC-type instructions rather than simple register-to-register operations. Moreover, our custom instructions more efficiently utilize hardware multiplication resources and bandwidth, achieving higher throughput under the same conditions. As a result, our energy efficiency per GOPs is also three times better than the K230 from [56].

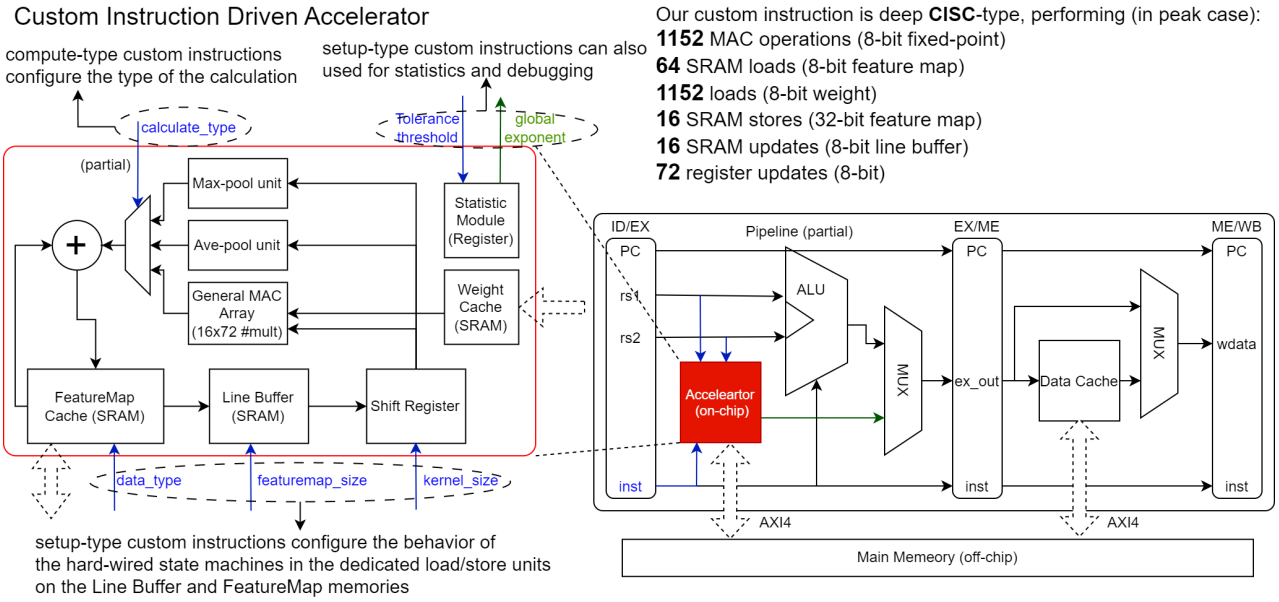



Figure 6.1: Deep CISC type operations

Since this paper primarily focuses on the hardware design for model inference, we have chosen to use straightforward post-training static quantization for the model. However, there are several areas for future work that could build upon this research:

1. **Exploration of Various CNN-Based AI Vision Applications**: This thesis has not tested a wide variety of CNN-based AI vision applications. Future work could explore the applicability and performance of the proposed hardware design across different CNN architectures and vision tasks.

2. **Applicability to Transformer-Based AI Algorithms for Large Language Models**: Transformer-based AI algorithms have transformed natural language processing (NLP), especially in developing large language models. The scalability and flexibility of transformer-based algorithms make them essential for advancing large language models. The proposed architecture's suitability for Transformer-based AI algorithms, particularly for large language models, has not been examined. Future research could investigate how well the hardware design supports these models and identify any necessary adaptations or optimizations.

3. **Quantization Aware Training (QAT)**: While we have employed straightforward post-training static quantization in this work, future studies could implement Quantization Aware Training (QAT) to potentially enhance accuracy further. QAT integrates

quantization into the training process, allowing the model to learn the quantization effects, which can lead to better performance compared to post-training quantization.

By addressing these areas, future research can further validate and extend the applicability of the proposed hardware design, ensuring its effectiveness across a broader range of AI models and tasks.

# Publications

Refereed papers

- H. Wang, D. Li, and T. Isshiki, "Reconfigurable CNN Accelerator Embedded in Instruction Extended RISC-V Core," in 2023 6th International Conference on Electronics Technology (ICET), IEEE, 2023, pp. 945–954.
- H. Wang, D. Li, and T. Isshiki, "A power-efficient end-to-end implementation of YOLOv8 based on RISC-V," in 2023 4th International Conference on Computers and Artificial Intelligence Technology (CAIT), 2023.
- Wang, H., Li, D., & Isshiki, T. A low-power reconfigurable DNN accelerator for instruction-extended RISC-V. IPSJ Transactions on System LSI Design Methodology, 17(0), 55–66. 2024.
- Wang, H., Li, D., & Isshiki, T. Energy-efficient implementation of YOLOv8, instance segmentation, and pose detection on RISC-V SoC. IEEE Access: Practical Innovations, Open Solutions, 12, 64050–64068. 2024.

# References


[1] C. P. Papageorgiou, M. Oren, and T. Poggio, "A general framework for object detection," in Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271), 2002.

[2] Y. Cao, C. Li, Y. Peng, and H. Ru, "MCS-YOLO: A multiscale object detection method for autonomous driving road environment recognition," IEEE Access, vol. 11, pp. 22342–22354, 2023.

[3] C. Yin, J. Tang, T. Yuan, Z. Xu, and Y. Wang, "Bridging the gap between semantic segmentation and instance segmentation," IEEE Trans. Multimedia, vol. 24, pp. 4183–4196, 2022.

[4] K. He, G. Gkioxari, P. Dollar, and R. Girshick, "Mask R-CNN," in 2017 IEEE International Conference on Computer Vision (ICCV), 2017.

[5] H. Chen, K. Sun, Z. Tian, C. Shen, Y. Huang, and Y. Yan, "BlendMask: Top-down meets bottom-up for instance segmentation," in 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020.

[6] C. Cadena, A. Dick, and I. D. Reid, "Multi-modal auto-encoders as joint estimators for robotics scene understanding," in Robotics: Science and Systems XII, 2016.

[7] Y. Zhou, O. F. Onder, Q. Dou, E. Tsougenis, H. Chen, and P.-A. Heng, "CIA-net: Robust nuclei instance segmentation with contour-aware information aggregation," in Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019, pp. 682–693.

[8] S. Zhou, D. Nie, E. Adeli, J. Yin, J. Lian, and D. Shen, "High-resolution encoder-decoder networks for low-contrast medical image segmentation," IEEE Trans. Image Process., pp. 1–1, 2019.

[9] H. A. Alhaija, S. K. Mustikovela, L. Mescheder, A. Geiger, and C. Rother, "Augmented reality meets deep learning for car instance segmentation in urban scenes," British machine vision conference, vol. 1, 2017.

[10] C. W. Hsieh, C. Y. Chen, C. L. Chou, H. H. Shuai, J. Liu, and W. H. Cheng, "FashionOn: Semantic-guided image-based virtual try-on with detailed human and clothing information," in Proceedings of the 27th ACM international conference on multimedia, 2019, pp. 275–283.

[11] H. Dong et al., "Towards multi-pose guided virtual try-on network," in 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 2019.

[12] C. Wang, Y. Wang, and A. L. Yuille, "An approach to pose-based action recognition," in 2013 IEEE Conference on Computer Vision and Pattern Recognition, 2013.

[13] CUDA Toolkit. (n.d.). NVIDIA Developer. Retrieved June 1, 2024, from https://developer.nvidia.com/cuda-toolkit

[14] Vitis unified software platform. (n.d.). AMD. Retrieved June 1, 2024, from https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

[15] Intel High Level Synthesis (HLS) Compiler tool. Retrieved June 1, 2024, from https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[16] H. Wang, D. Li, and T. Isshiki, "Reconfigurable CNN Accelerator Embedded in Instruction Extended RISC-V Core," in 2023 6th International Conference on Electronics Technology (ICET), IEEE, 2023, pp. 945–954.

[17] H. Wang, D. Li, and T. Isshiki, "A power-efficient end-to-end implementation of YOLOv8 based on RISC-V," in 2023 4th International Conference on Computers and Artificial Intelligence Technology (CAIT), 2023.

[18] Wang, H., Li, D., & Isshiki, T. (2024a). A low-power reconfigurable DNN accelerator for instruction-extended RISC-V. IPSJ Transactions on System LSI Design Methodology, 17(0), 55–66. https://doi.org/10.2197/ipsjtsldm.17.55

[19] Wang, H., Li, D., & Isshiki, T. (2024b). Energy-efficient implementation of YOLOv8, instance segmentation, and pose detection on RISC-V SoC. IEEE Access: Practical Innovations, Open Solutions, 12, 64050–64068. https://doi.org/10.1109/access.2024.3397536

[20] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," in EECS Department, 2014.

[21] Cadence.com. Retrieved August 2, 2024, from https://www.cadence.com/en_US/home/ai/overview.ht

[22] Ditzel, D. R. (2022). Accelerating ML recommendation with over 1,000 RISC-V/tensor processors on Esperanto's ET-SoC-1 chip. IEEE Micro, 42(3), 31–38. https://doi.org/10.1109/mm.2022.3140674

[23] Knowles, S., 2021, August. Graphcore. In 2021 IEEE Hot Chips 33 Symposium (HCS) (pp. 1-25). IEEE.

[24] Lie, S., 2022, August. Cerebras architecture deep dive: First look inside the hw/sw co-design for deep learning: Cerebras systems. In 2022 IEEE Hot Chips 34 Symposium (HCS) (pp. 1-34). IEEE Computer Society.

[25] T. Sadasue and T. Isshiki, "LLVM-C2RTL: C/C++ based system level RTL design framework using LLVM compiler infrastructure," IPSJ Trans. Syst. LSI Des. Methodol., vol. 16, no. 0, pp. 12–26, 2023.

[26] A. Hosseiny and H. Jahanirad, "Hardware acceleration of YOLOv7-tiny using high-level synthesis tools," Journal of Real-Time Image Processing, vol. 20, no. 4, 2023.

[27] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," Communications of the ACM, vol. 52, no. 4, pp. 65–76, 2009.

[28] F. Liu, B. Zhang, G. Chen, G. Gong, H. Lu, and W. Li, "A novel configurable high-precision and low-cost circuit design of sigmoid and tanh activation function," in 2021 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA), 2021.

[29] G. Lin and W. Shen, "Research on convolutional neural network based on improved Relu piecewise activation function," Procedia Comput. Sci., vol. 131, pp. 977–984, 2018.

[30] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2015.

[31] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," IEEE Trans. Very Large Scale Integr. VLSI Syst., vol. 26, no. 7, pp. 1354–1367, 2018.

[32] Z. Wang, K. Xu, S. Wu, L. Liu, L. Liu, and D. Wang, "Sparse-YOLO: Hardware/software co-design of an FPGA accelerator for YOLOv2," IEEE Access, vol. 8, pp. 116569–116585, 2020.

[33] S. Li, C. Yu, T. Xie, and W. Feng, "A power-efficient optimizing framework FPGA accelerator for YOLO," in 2022 15th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI), 2022.

[34] W. Huang et al.:FPGA-Based High-Throughput CNN Hardware Accelerator With High Computing Resource Utilization Ratio, IEEE Transactions on Neural Networks and Learning Systems, doi: 10.1109/TNNLS.2021.3055814(2021).

[35] S. S. Lee, T. D. Nguyen, P. K. Meher and S. Y. Park: Energy-Efficient High-Speed ASIC Implementation of Convolutional Neural Network Using Novel Reduced Critical-Path Design, IEEE Access, vol. 10, pp. 34032-34045, 2022, doi: 10.1109/ACCESS.2022.3162066.

[36] Guo, Kaiyuan, et al.: Angel-eye: A complete design flow for mapping CNN onto embedded FPGA, IEEE transactions on computer-aided design of integrated circuits and systems 37.1: 35-47 (2017).

[37] Ma, Yufei, et al.:Automatic compilation of diverse CNNs onto high-performance FPGA accelerators,IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39.2: 424-437 (2018).

[38] Y. Yu, C. Wu, T. Zhao, K. Wang and L. He.:OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks,IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 28, no. 1, pp. 35-47, Jan. 2020, doi: 10.1109/TVLSI.2019.2939726(2020).

[39] J. Terven and D. Cordova-Esparza, A comprehensive review of YOLO: From YOLOv1 to YOLOv8 and beyond. 2023.

[40] M. Sohan, T. Sai Ram, R. Reddy, and C. Venkata, "A Review on YOLOv8 and Its Advancements," in International Conference on Data Intelligence and Cognitive Informatics, Singapore: Springer, 2024, pp. 529–545.

[41] Y. Zhao, B. Chen, B. Liu, C. Yu, L. Wang, and S. Wang, "GRP-YOLOv5: An improved bearing defect detection algorithm based on YOLOv5," Sensors (Basel), vol. 23, no. 17, 2023.

[42] M. Qiu, L. Huang, and B.-H. Tang, "ASFF-YOLOv5: Multielement detection method for road traffic in UAV images based on multiscale feature fusion," Remote Sens. (Basel), vol. 14, no. 14, p. 3498, 2022.

[43] P. Yan, Q. Sun, N. Yin, L. Hua, S. Shang, and C. Zhang, "Detection of coal and gangue based on improved YOLOv5.1 which embedded scSE module," Measurement (Lond.), vol. 188, no. 110530, p. 110530, 2022.

[44] X. Wang, H. Gao, Z. Jia, and Z. Li, "BL-YOLOv8: An improved road defect detection model based on YOLOv8," Sensors (Basel), vol. 23, no. 20, 2023.

[45] Li, Xiang, Wenhai Wang, Lijun Wu, Shuo Chen, Xiaolin Hu, Jun Li, Jinhui Tang, and Jian Yang. "Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection," Adv. Neural Inf. Process. Syst., vol. 33, pp. 21002-21012, 2023.

[46] X. Li, C. Lv, W. Wang, G. Li, L. Yang, and J. Yang, "Generalized Focal Loss: Towards efficient representation learning for dense object detection," IEEE Trans. Pattern Anal. Mach. Intell., vol. 45, no. 3, pp. 3139–3153, 2023.

[47] K. Wang, J. H. Liew, Y. Zou, D. Zhou, and J. Feng, "PANet: Few-shot image semantic segmentation with prototype alignment," in 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 2019.

[48] Ultralytics, "Ultralytics YOLOv8 Docs." Ultralytics.com, Accessed: Apr. 29, 2024. [Online]. Available: https://docs.ultralytics.com/

[49] T.-Y. Lin et al., "Microsoft COCO: Common objects in context," in Computer Vision – ECCV 2014, Cham: Springer International Publishing, 2014, pp. 740–755.

[50] P. Bader, S. Blanes, and F. Casas, "Computing the matrix exponential with an optimized Taylor polynomial approximation," Mathematics, vol. 7, no. 12, p. 1174, 2019.

[51] A. Litvin, K. Nasrollahi, S. Escalera, C. Ozcinar, T. B. Moeslund, and G. Anbarjafari, "A novel deep network architecture for reconstructing RGB facial images from thermal for face recognition," Multimed. Tools Appl., vol. 78, no. 18, pp. 25259–25271, 2019.

[52] K.-W. Chang and T.-S. Chang, "Efficient accelerator for dilated and transposed convolution with decomposition," in 2020 IEEE International Symposium on Circuits and Systems (ISCAS), 2020.

[53] X. Di, H.-G. Yang, Y. Jia, Z. Huang, and N. Mao, "Exploring efficient acceleration architecture for Winograd-transformed transposed convolution of GANs on FPGAs," Electronics (Basel), vol. 9, no. 2, p. 286, 2020.

[54] A. Odena, V. Dumoulin, and C. Olah, “Deconvolution and Checkerboard Artifacts,” Distill, vol. 1, no. 10, 2016.

[55] E. David et al., “Global Wheat Head Detection (GWHD) dataset: A large and diverse dataset of high-resolution RGB-labelled images to develop and benchmark wheat head detection methods,” Plant Phenomics, vol. 2020, p. 3521852, 2020.

[56] Canaan, ”K230 Product Full Datasheet.” canaan-creative.com. Accessed: Apr. 29, 2024. [Online]. Available: https://developer.canaan-creative.com/k230/dev/zh/00_hardware/K230_datasheet.html/

[57] Roboflow Universe, ”crack Computer Vision Project.” roboflow.com. Accessed: Apr. 29, 2024. [Online]. Available: https://universe.roboflow.com/university-bswxt/crack-bphdr/

# Appendix A

# Original Synthesis Result

## A.1 Area

```
Report : area
Design : RVProcAXI
Version: U-2022.12
Date   : Sat Jan 27 21:15:01 2024
****************************************

Information: Updating design information... (UID-85)
Warning: Design 'RVProcAXI' contains 1 high-fanout nets. A fanout number of 1000 will be used for delay calculations involving these nets. (TIM-134)
Library(s) Used:

    tcbn28hpcplusbwp35p140tt0p9v25c_ccs (File: /home/vlsilab/wang/pdk/TSMCHOME/digital/Front_End/timing_power_noise/CCS/tcbn28hpcplusbwp35p140_180a/
    tsln28hpcphvtb8192x32m8s_tt0p9v25c (File: /home/vlsilab/wang/RVProcAXI-B/ip/sram-hv/lib/tsln28hpcphvtb8192x32m8s_180a_tt0p9v25c.db)
    tsln28hpcpuhdhvtb16x32m1s_tt0p9v25c (File: /home/vlsilab/wang/RVProcAXI-B/ip/sram/lib/tsln28hpcpuhdhvtb16x32m1s_170a_tt0p9v25c.db)
    tsln28hpcpuhdhvtb16x17m2s_tt0p9v25c (File: /home/vlsilab/wang/RVProcAXI-B/ip/sram/lib/tsln28hpcpuhdhvtb16x17m2s_170a_tt0p9v25c.db)
    tsln28hpcpuhdhvtb64x22m1s_tt0p9v25c (File: /home/vlsilab/wang/RVProcAXI-B/ip/sram/lib/tsln28hpcpuhdhvtb64x22m1s_170a_tt0p9v25c.db)
    tsln28hpcpuhdhvtb64x64m1s_tt0p9v25c (File: /home/vlsilab/wang/RVProcAXI-B/ip/sram/lib/tsln28hpcpuhdhvtb64x64m1s_170a_tt0p9v25c.db)
    tsln28hpcpuhdhvtb512x8m4s_tt0p9v25c (File: /home/vlsilab/wang/RVProcAXI-B/ip/sram/lib/tsln28hpcpuhdhvtb512x8m4s_170a_tt0p9v25c.db)

Number of ports:                       100677
Number of nets:                        620010
Number of cells:                       465983
Number of combinational cells:         416365
Number of sequential cells:             48509
Number of macros/black boxes:             312
Number of buf/inv:                      52235
Number of references:                       2

Combinational area:            330683.723299
Buf/Inv area:                   15243.480118
Noncombinational area:         105173.581886
Macro/Black Box area:         7481130.111816
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:              7916987.417001
```

Figure A.1: AIV-SoC Aera report

## A.2 Power

```
Global Operating Voltage = 0.9
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1nW


Attributes
----------
i - Including register clock pin internal power


  Cell Internal Power  = 195.0264 mW   (99%)
  Net Switching Power  =   2.1350 mW    (1%)
                         ---------
Total Dynamic Power    = 197.1615 mW  (100%)

Cell Leakage Power     =   5.3519 mW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory            98.2176           0.2002         4.7125e+06            103.1303  (  50.92%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network     95.6005           0.0000              0.0000            95.6005  (  47.20%)  i
register           0.1191        7.6059e-02         1.3671e+05             0.3318  (   0.16%)
sequential         0.0000           0.0000              0.0000             0.0000  (   0.00%)
combinational      1.1187           1.8588         5.0266e+05             3.4801  (   1.72%)
--------------------------------------------------------------------------------------------------
Total            195.0559 mW        2.1350 mW      5.3519e+06 nW       202.5427 mW
```

Figure A.2: AIV-SoC Power report

## A.3 Timing

```
-------------------------------------------------------------------------
clock clk (rise edge)                                   0.00       0.00
clock network delay (ideal)                             0.00       0.00
M0/M2/M0/M179/mem/sram16x17.M0/CLK (TS1N28HPCPUHDHVTB16X17M2S)
                                                        0.00 #     0.00 r
M0/M2/M0/M179/mem/sram16x17.M0/Q[5] (TS1N28HPCPUHDHVTB16X17M2S)
                                                        0.31       0.31 r
M0/M2/M0/M179/mem/U18/Z (MUX2OPTD4BWP35P140)            0.02       0.34 r
M0/M2/M0/M179/mem/dout[5] (sram_tsmc28_DATA_WIDTH17_ADDR_WIDTH4_MEM_SIZE16_1)
                                                        0.00       0.34 r
M0/M2/M0/M179/dout[5] (CPU_step_RAM_EXCL_RW_M_ID0_MEM_SIZE16_ADDR_WIDTH4_DATA_WIDTH17_1)
                                                        0.00       0.34 r
M0/M2/M0/M0/G__this_io_dcache_tlb_m_tag_data_2_dout[5] (CPU_step_no_mem_M_ID0)
                                                        0.00       0.34 r
M0/M2/M0/M0/U10131/ZN (ND2OPTIBD2BWP35P140)             0.01       0.35 f
M0/M2/M0/M0/U10103/ZN (ND2D1BWP35P140)                  0.01       0.36 r
M0/M2/M0/M0/U60359/ZN (OAI22D2BWP35P140)                0.01       0.37 f
M0/M2/M0/M0/U29086/ZN (NR2D1BWP35P140)                  0.01       0.38 r
M0/M2/M0/M0/U85894/ZN (ND3OPTPAD2BWP35P140)             0.01       0.40 f
M0/M2/M0/M0/U76345/ZN (NR2D3BWP35P140)                  0.02       0.41 r
M0/M2/M0/M0/U57687/ZN (INVD6BWP35P140)                  0.01       0.43 f
M0/M2/M0/M0/U9860/ZN (NR2D1BWP35P140)                   0.01       0.44 r
M0/M2/M0/M0/U56530/ZN (ND2OPTIBD2BWP35P140)             0.02       0.46 f
M0/M2/M0/M0/U9841/ZN (NR2D1BWP35P140)                   0.01       0.47 r
M0/M2/M0/M0/U95214/ZN (IAO21D1BWP35P140)                0.01       0.48 f
M0/M2/M0/M0/U95213/ZN (OAI21OPTREPBD2BWP35P140)         0.01       0.49 r
M0/M2/M0/M0/U9773/ZN (NR4D1BWP35P140)                   0.02       0.51 f
M0/M2/M0/M0/U93397/ZN (ND2OPTIBD6BWP35P140)             0.02       0.53 r
M0/M2/M0/M0/U54050/ZN (XNR2UD1BWP35P140)                0.02       0.55 r
M0/M2/M0/M0/U11851/ZN (NR4D1BWP35P140)                  0.02       0.57 f
M0/M2/M0/M0/U37016/ZN (ND2OPTIBD1BWP35P140)             0.01       0.58 r
M0/M2/M0/M0/U76499/ZN (NR3OPTPAD2BWP35P140)             0.01       0.59 f
M0/M2/M0/M0/U52555/ZN (CKND2D3BWP35P140)                0.01       0.61 r
M0/M2/M0/M0/U51938/ZN (ND3D3BWP35P140)                  0.02       0.63 f
M0/M2/M0/M0/U76274/ZN (ND2OPTPAD4BWP35P140)             0.01       0.64 r
M0/M2/M0/M0/U51441/ZN (ND2D6BWP35P140)                  0.02       0.66 f
M0/M2/M0/M0/U75038/ZN (ND2OPTPAD16BWP35P140)            0.02       0.68 r
M0/M2/M0/M0/U16502/ZN (INVD12BWP35P140)                 0.02       0.69 f
M0/M2/M0/M0/U16503/Z (BUFFD6BWP35P140)                  0.03       0.72 f
M0/M2/M0/M0/U2705/Z (AN2D4BWP35P140)                    0.02       0.74 f
M0/M2/M0/M0/U26475/ZN (INVD6BWP35P140)                  0.01       0.75 r
M0/M2/M0/M0/U261984/ZN (INVD6BWP35P140)                 0.02       0.77 f
M0/M2/M0/M0/U15783/ZN (INR2D1BWP35P140)                 0.03       0.80 f
M0/M2/M0/M0/U15399/ZN (IND2D1BWP35P140)                 0.07       0.87 f
M0/M2/M0/M0/U1931/ZN (INVD1BWP35P140)                   0.03       0.90 r
M0/M2/M0/M0/U1876/ZN (INVD3BWP35P140)                   0.03       0.94 f
M0/M2/M0/M0/U15416/ZN (INVD1BWP35P140)                  0.06       1.00 r
M0/M2/M0/M0/U102055/ZN (INR2D1BWP35P140)                0.08       1.08 f
M0/M2/M0/M0/U25351/Z (BUFFD1BWP35P140)                  0.08       1.15 f
M0/M2/M0/M0/U354602/ZN (AOI22D1BWP35P140)               0.03       1.19 r
M0/M2/M0/M0/U354603/ZN (OAI21D1BWP35P140)               0.02       1.20 f
M0/M2/M0/M0/U354605/ZN (INR4D0BWP35P140)                0.03       1.24 r
M0/M2/M0/M0/U354606/ZN (OAI211D1BWP35P140)              0.03       1.26 f
M0/M2/M0/M0/U354607/ZN (AOI21D1BWP35P140)               0.02       1.28 r
M0/M2/M0/M0/U354608/ZN (ND2D1BWP35P140)                 0.01       1.30 f
M0/M2/M0/M0/U354609/ZN (AOI21D1BWP35P140)               0.02       1.31 r
M0/M2/M0/M0/U354610/ZN (OAI21D1BWP35P140)               0.02       1.33 f
M0/M2/M0/M0/U354611/ZN (AOI21D1BWP35P140)               0.02       1.35 r
M0/M2/M0/M0/U354612/ZN (OAI21D1BWP35P140)               0.02       1.37 f
M0/M2/M0/M0/G__this_ce_data_shifted_d_reg[2][1]/D (DFCNQD1BWP35P140)
                                                        0.00       1.37 f
data arrival time                                                  1.37

clock clk (rise edge)                                   1.68       1.68
clock network delay (ideal)                             0.00       1.68
clock uncertainty                                      -0.30       1.38
M0/M2/M0/M0/G__this_ce_data_shifted_d_reg[2][1]/CP (DFCNQD1BWP35P140)
                                                        0.00       1.38 r
library setup time                                     -0.01       1.37
data required time                                                 1.37
-------------------------------------------------------------------------
data required time                                                 1.37
data arrival time                                                 -1.37
-------------------------------------------------------------------------
slack (MET)                                                        0.00
```



Figure A.3: AIV-SoC Timing report