ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA ĐIỆN TỬ - VIỄN THÔNG

MSSV 21207001
Bùi Thành Đạt

BÁO CÁO THỰC TẬP THỰC TẾ

Thiết kế và Tích hợp Bộ điều khiển DMA Tùy chỉnh trên FPGA
trong Hệ thống SoC Nios V/m

NGÀNH KỸ THUẬT ĐIỆN TỬ - VIỄN THÔNG
CHƯƠNG TRÌNH CHẤT LƯỢNG CAO

Tp. Hồ Chí Minh, tháng 06/2025

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA ĐIỆN TỬ - VIỄN THÔNG

MSSV 21207001
Bùi Thành Đạt

BÁO CÁO THỰC TẬP THỰC TẾ

Thiết kế và Tích hợp Bộ điều khiển DMA Tùy chỉnh trên FPGA
trong Hệ thống SoC Nios V/m

NGÀNH KỸ THUẬT ĐIỆN TỬ - VIỄN THÔNG
CHƯƠNG TRÌNH CHẤT LƯỢNG CAO

GIÁO VIÊN HƯỚNG DẪN
TS. Huỳnh Hữu Thuận

Tp. Hồ Chí Minh, tháng 06/2025

MỤC LỤC

DANH SÁCH CHỮ VIẾT TẮT

BOOM Berkeley Out-of-Order Machine. 3

CHISEL Constructing Hardware In a Scala Embedded Language. 3

FIRRTL Flexible Intermediate Representation for RTL. 3

FPGA Field-Programmable Gate Array. 3, 4

RTL Register Transfer Level. 3

SoC System-on-Chip. 4

DANH SÁCH CÁC HÌNH

DANH SÁCH CÁC BẢNG

CHƯƠNG 1: INTRODUCTION

CHƯƠNG 2: THEORETICAL FOUNDATION

2.1 Chipyard Framework

2.1.1 Overview of Chipyard

Chipyard is an open-source System-on-Chip (SoC) design framework that facilitates the agile development of complex, customized hardware systems [1]. It provides an integrated environment that brings together a vast collection of Register Transfer Level (RTL) (Register Transfer Level) generators, simulation tools, and implementation flows. The primary goal of Chipyard is to enable rapid architectural exploration, design, and evaluation of SoCs, particularly those incorporating specialized hardware accelerators and heterogeneous core configurations.

2.1.2 Key Components of Chipyard

- **RTL Generators:** Chipyard leverages a rich library of open-source, generator-based IP blocks. These generators, primarily written in Constructing Hardware In a Scala Embedded Language (CHISEL) (Constructing Hardware In a Scala Embedded Language), allow for highly parameterized and composable hardware designs. This includes RISC-V cores (like Rocket and Berkeley Out-of-Order Machine (BOOM)), caches, interconnects, peripherals, and domain-specific accelerators [1].

- **SoC Configuration:** Designs in Chipyard are specified through a powerful configuration system. Users can define the SoC architecture, select IP blocks, and customize their parameters without directly modifying the RTL source code. This promotes modularity and reusability [1, p. 27].

- **FIRRTL Intermediate Representation:** Chisel designs are elaborated into Flexible Intermediate Representation for RTL (FIRRTL) (Flexible Intermediate Representation for RTL). Chipyard utilizes FIRRTL transformations to adapt the generated hardware for various downstream tools and flows, such as software simulation, Field-Programmable Gate Array (FPGA) emulation, and ASIC implementation [1].

- **Simulation and Verification:** Chipyard supports multiple simulation methodologies:

- Software RTL Simulation: Cycle-accurate simulation using tools like Verilator or commercial simulators for detailed debugging and verification [1, p. 29].
- FPGA-Accelerated Simulation (FireSim): For faster, full-system validation with long-running workloads, Chipyard integrates FireSim. FireSim enables cycle-exact simulation on cloud-hosted FPGAs, providing a scalable and accessible platform for pre-silicon performance evaluation [5, 1].
- VLSI Implementation (Hammer): For physical design, Chipyard includes Hammer, a VLSI flow that abstracts process-technology and EDA-tool specifics, enabling more portable and reusable physical design scripts [1].
- Workload Management (FireMarshal): Chipyard provides FireMarshal for generating and managing software workloads, including Linux distributions, to run on the simulated or emulated hardware [1].

Chipyard's methodology promotes an agile hardware development process, allowing for iterative design, continuous integration, and validation of physically realizable custom System-on-Chips (SoCs). This research utilizes Chipyard as the foundational framework for constructing and evaluating the Rocket Chip-based SoC with the Gemmini accelerator.

2.2 Rocket Chip and RISC-V

The foundation of the processing system developed in this thesis lies in the RISC-V instruction set architecture and its implementation through the Rocket Chip SoC generator.

2.2.1 The RISC-V Instruction Set Architecture (ISA)

The RISC-V ISA represents a paradigm shift in processor design, moving away from proprietary, closed architectures towards a free and open standard. This openness is a crucial enabler for innovation, allowing researchers, academics, and industry professionals to design, implement, and extend processor architectures without restrictive licensing fees or access limitations [2]. RISC-V is designed with simplicity, modularity, and extensibility at its core.

Key characteristics of the RISC-V ISA relevant to this work include:

- **Modularity:** The ISA is defined as a small base integer instruction set (e.g., RV32I for 32-bit, RV64I for 64-bit systems) with multiple standard optional extensions that can be added to meet specific application needs. This thesis focuses on an RV64I base. Common extensions include 'M' for integer multiplication and division, 'A' for atomic instructions, 'F' for single-precision floating-point, 'D' for double-precision floating-point, and 'C' for compressed instructions. The combination of IMAFD is often referred to as 'G' for general-purpose computing [3].

- **Extensibility:** RISC-V reserves a significant portion of its opcode space for custom, non-standard extensions. This allows for the tight integration of specialized hardware, such as the Gemmini accelerator used in this research, directly into the processor's instruction set if desired, or via coprocessor interfaces.

- **Privileged Architecture:** To support complex operating systems like Linux, RISC-V defines a privileged architecture with multiple modes of operation, including Machine (M), Supervisor (S), and User (U) modes. The Supervisor mode, along with mechanisms for virtual memory management (Memory Management Unit - MMU) and interrupt handling, is essential for OS functionality [6].

- **Growing Ecosystem:** The open nature of RISC-V has fostered a rapidly expanding ecosystem of compatible cores, development tools (compilers like GCC and LLVM, simulators like Spike and QEMU), and operating system ports (including Linux and FreeBSD).

2.2.2 Rocket Chip SoC Generator

Rocket Chip is a highly influential open-source SoC generator that produces synthesizable RTL for systems based on the RISC-V ISA [3]. It is a cornerstone of the Chipyard framework and is itself developed using Chisel. Instead of being a fixed processor design, Rocket Chip is a generator capable of producing a wide variety of SoC configurations, from simple microcontrollers to complex multi-core systems capable of running full operating systems.

Key aspects of the Rocket Chip generator utilized or relevant to this thesis include:

- **Core Generators:**

  - Rocket Core: This is a highly configurable, in-order, scalar RISC-V core generator. It can be parameterized to implement various ISA subsets (e.g., RV64G as used in this project). Crucially for this research, it features a sophisticated Memory Management Unit (MMU) supporting page-based virtual memory, configurable branch prediction mechanisms, and non-blocking data caches. These features, along with its support for Supervisor mode, make it well-suited for running operating systems like Linux [3, p. 4]. This is the primary core utilized in this thesis.
  - BOOM (Berkeley Out-of-Order Machine): Rocket Chip also includes a generator for BOOM, an out-of-order, superscalar core designed for higher performance applications, demonstrating the generator's versatility [3, p. 5].

- **Cache Hierarchy:** The generator provides configurable L1 instruction and data caches for each core, as well as an optional shared L2 cache. Parameters such as cache size, associativity, number of banks, and replacement policies can be customized to explore different points in the design space [3, p. 3].

- **TileLink Interconnect:** Rocket Chip employs TileLink, an open-source, chip-scale interconnect protocol designed to connect various components within the SoC, including cores, caches, memory controllers, and peripherals. TileLink supports cache coherence, enabling the construction of complex, coherent multi-core systems and accelerator integrations [3, p. 6] [1, p. 15].

- **RoCC (Rocket Custom Coprocessor) Interface:** This standardized interface facilitates the integration of custom hardware accelerators, termed RoCC accelerators, with Rocket or BOOM cores. Accelerators connected via RoCC can be issued custom instructions decoded by the core, can access core registers, and can interact with the memory system. They can share the core's L1 data cache and Page Table Walker (PTW) for virtual memory support, or connect directly to the TileLink fabric for higher bandwidth memory access [3, p. 5] [1, p. 13]. The Gemmini accelerator in this thesis is integrated using this interface.

This research specifically leverages the Rocket core generator within the Chipyard framework to construct a 64-bit RISC-V system (based on the Rocket64b1gem16

configuration), which is then targeted for implementation and evaluation on a Xilinx VC707 FPGA board.

2.3 Gemmini Accelerator

Gemmini is a highly configurable, open-source systolic array-based matrix multiplication accelerator generator, designed to accelerate machine learning workloads, particularly deep neural networks [4, 1]. It is typically integrated into a Chipyard-based SoC as a RoCC accelerator, allowing it to be controlled by custom RISC-V instructions executed on a host core like Rocket.

Key Features of Gemmini:

- **Systolic Array Architecture:** Gemmini employs a 2D systolic array of processing elements (PEs) to perform matrix multiplications efficiently. The dimensions of this array, dataflow (e.g., weight-stationary, output-stationary), and data types are configurable.

- **On-Chip Scratchpad Memory:** It includes dedicated on-chip SRAM (scratchpad memory) for storing input activations, weights, and partial sums, reducing reliance on off-chip memory bandwidth.

- **Configurability:** Gemmini offers extensive parameterization, allowing designers to explore a wide design space. This includes the systolic array dimensions, scratchpad memory sizes, data precision, and the interface to the memory system.

- **RoCC Integration:** As a RoCC accelerator, Gemmini can be issued commands by the CPU. These commands can configure the accelerator, load data into its scratchpads, and initiate matrix multiplication operations. It can leverage the CPU's virtual memory system through the RoCC interface for memory accesses [1, p. 13].

In this thesis, the Gemmini accelerator is integrated with the Rocket core to provide hardware acceleration for the matrix multiplication operations that are fundamental to many machine learning algorithms. The successful execution of tests from gemmini-rocc-tests confirms the functional integration of Gemmini within the built system.

2.4 Linux on RISC-V SoCs

Running a full-fledged operating system like Linux on a custom-designed RISC-V SoC involves several key software and hardware components working in concert. The ability to boot Linux signifies a mature and functional hardware platform capable of supporting complex software stacks. The project eugene-tarassov/vivado-risc-v, referenced in the current results, provides a specific environment and set of configurations for achieving this on a Xilinx VC707 board.

General components and processes involved include:

- **RISC-V Privileged Architecture:** The RISC-V ISA includes a privileged architecture specification that defines different privilege modes (Machine, Supervisor, User), control and status registers (CSRs), and mechanisms for memory management (MMU) and interrupt handling. A Linux-capable core must implement the Supervisor mode and support virtual memory [6]. Rocket Chip cores, like the one used, provide this support [3].

- **Bootloader:** A bootloader is the first piece of software that runs after system reset. Its primary role is to initialize the hardware (e.g., memory controller, console) and then load the operating system kernel into memory and transfer execution to it. Common bootloaders for RISC-V include:

  - Berkeley Boot Loader (BBL): Often used with Rocket Chip, BBL can run in Machine mode and provides a Supervisor-mode execution environment for the Linux kernel. It typically includes a device tree blob.
  - U-Boot: A more versatile and widely used bootloader that supports multiple architectures, including RISC-V.

- **Device Tree (DTB):** The Device Tree Blob is a data structure passed to the kernel by the bootloader. It describes the hardware components of the system (e.g., number of cores, memory map, peripherals, interrupt controllers) in a standardized way, allowing the kernel to be hardware-agnostic to a certain extent.

- **Linux Kernel:** A port of the Linux kernel for the RISC-V architecture is required. This kernel must include drivers for the specific peripherals present in the SoC (e.g., UART for console, network interface, block device controller if present).

- **Root Filesystem:** Once the kernel is running, it mounts a root filesystem which contains the user-space applications, libraries, and system utilities that constitute the Linux distribution (e.g., Debian, Fedora, Buildroot).

The vivado-risc-v project likely provides pre-compiled versions or build scripts for these components, specifically tailored for the Rocket Chip configuration deployed on the VC707, including necessary FPGA-specific configurations (e.g., for DDR memory controller, Ethernet, UART through JTAG or physical pins). Successfully booting Debian Linux demonstrates that the custom SoC, including the Rocket core, memory system, and peripherals, are functioning correctly and are sufficiently robust to support a complex operating system.

2.5 Time-Series Anomaly Detection

This subsection will be elaborated upon once the machine learning model for arrhythmia time-series anomaly detection is developed and implemented. It will cover the theoretical underpinnings of the chosen anomaly detection algorithms, relevant time-series analysis techniques, and how these map to the capabilities of the Gemmini-accelerated RISC-V platform.

CHƯƠNG 3: IMPLEMENTATION

This chapter details the hardware architecture of the custom System-on-Chip (SoC) designed for this research, and its realization on an FPGA platform. The SoC was architected using the Chipyard SoC generation framework, which integrates the Rocket Chip generator (Asanović et al., 2016) for the core SoC infrastructure and CPU, all described using the Chisel hardware construction language. The practical FPGA deployment, enabling a full Debian Linux environment on an AMD/Xilinx VC707 board (the target for this work), was achieved by leveraging and adapting the infrastructure provided by the vivado-risc-v project (Tarassov, n.d.). This project supplies the necessary scripts, Verilog-based peripheral controllers (e.g., for DDR memory via Vivado IP, and open-source UART, SD card, and Ethernet controllers), and build system modifications to bridge the gap between the generated SoC RTL and a functional Linux-capable FPGA system. This approach combines the flexibility of academic SoC generators with a robust path to hardware prototyping. The use of the open RISC-V Instruction Set Architecture (ISA) is central to this endeavor, promoting transparency and customizability.

The specific SoC configuration developed for this thesis is descriptively termed the "Rocket Core 64-bit Big Core with Gemmini 16x16 accelerator." Within the project's build and configuration system, this specific design point is identified as `Rocket64b1gem16`. This system integrates a Gemmini systolic array accelerator—a specialized hardware unit designed for efficient Deep Neural Network (DNN) computation (Genc et al., 2021)—with a general-purpose RISC-V Rocket CPU core. The overall design aims to create a balanced system capable of running a full Linux distribution (Debian) while providing significant hardware acceleration for the target time-series anomaly detection workloads. The following sections describe the key architectural features of this SoC and the underlying design decisions.

3.1 Target SoC Architecture: Rocket Core 64-bit Big Core with Gemmini 16x16 Accelerator (`Rocket64b1gem16`)

The `Rocket64b1gem16` system embodies a complete SoC. At its heart is a 64-bit RISC-V Rocket CPU core, specifically configured as a "big" core variant. This CPU acts as the main general-purpose processor, responsible for executing the Debian Linux operating system, managing system resources and peripherals, and orchestrat-

ing tasks for the specialized Gemmini accelerator. As documented by Asanović et al. (2016), the Rocket core is typically a 5-stage, single-issue, in-order scalar processor. The "big" core designation in this context signifies the inclusion of critical features like a Memory Management Unit (MMU), which is indispensable for supporting virtual memory and thus enabling a sophisticated operating system like Linux. It also incorporates a Floating Point Unit (FPU) for handling scalar floating-point calculations, although the primary computationally intensive DNN tasks are offloaded to Gemmini.

The central accelerator component is the Gemmini unit, configured with a 16×16 systolic array of Processing Elements (PEs). Each PE within this array is designed to perform a multiply-accumulate (MAC) operation on the specified data types per clock cycle. This array structure forms a highly parallel computation engine, exceptionally effective for the matrix multiplication and convolution operations that are foundational to most DNN workloads. Gemmini interacts with the rest of the SoC via a 64-bit wide Direct Memory Access (DMA) engine, which facilitates high-bandwidth data transfers between Gemmini's internal memories and the main system memory.

The SoC's memory subsystem includes a shared L2 cache. This cache is designed to be inclusive, meaning it maintains a superset of the data stored in the CPU's private L1 caches. The L2 cache aims to reduce the average memory access latency for the CPU and can also enhance the performance of Gemmini's DMA transfers by serving as a fast, on-chip data buffer. For software development and debugging purposes, the system is configured with multiple hardware breakpoints accessible via JTAG, a feature supported by the ‘vivado-risc-v‘ infrastructure. Essential system settings, such as the overall memory map (including DDR RAM interfaced via a controller typically provided by the Vivado IP catalog within the ‘vivado-risc-v‘ project), boot procedures from an SD card, and interrupt handling mechanisms, are established to create a fully functional computing platform.

3.2 Key Architectural Components and Their Configuration

The SoC's architecture is constructed by integrating and parameterizing several key hardware blocks. This modular design philosophy, inherent in Chipyard and Rocket Chip, allows for the creation of SoCs tailored to specific requirements. The configuration and integration of the Gemmini accelerator are particularly critical for achieving the research objectives related to DNN acceleration.

3.2.1 Gemmini Accelerator Integration and System Interface

The Gemmini accelerator is incorporated into the SoC fabric as a specialized co-processor unit. It is connected to the CPU and the system bus utilizing the RoCC (Rocket Custom Coprocessor) interface. This standard interface, a well-established part of the Rocket Chip ecosystem, enables the CPU to issue custom RISC-V instructions specifically decoded by Gemmini. These instructions are used to configure Gemmini's operational parameters, manage its internal state, and initiate data transfers and computations.

The computational core of Gemmini, the 16×16 array of PEs, is fed data through its dedicated DMA engine. This DMA engine communicates with the main system memory over a 64-bit wide data bus. The system's primary interconnect, a TileLink-based cache-coherent fabric as described by Asanović et al. (2016), is configured with a corresponding 64-bit (8-byte) data transfer granularity (`beatBytes`) to ensure efficient and protocol-compliant communication between Gemmini and other memory system components like the L2 cache and main DDR memory.

3.2.2 Gemmini's Internal Architecture and Operational Parameters

Gemmini itself is a highly configurable accelerator generator, designed around a systolic array architecture. Systolic arrays are particularly well-suited for DNN computations because they can effectively exploit the inherent parallelism and significant data reuse opportunities present in these algorithms, thereby minimizing the bottleneck of off-chip memory accesses (Genc et al., 2021). The specific instance of Gemmini within the `Rocket64b1gem16` SoC is configured with several important features that dictate its performance and capabilities:

- **Data Representation and Quantization Strategy**: The accelerator is configured to process 8-bit signed integer input data and utilizes 32-bit signed integers for its internal accumulators. This configuration directly supports 8-bit quantized integer arithmetic for DNN inference. Quantization is a widely adopted technique to reduce the memory footprint of DNN models, lessen data transfer bandwidth, and decrease computational energy, often with only a minor impact on the model's predictive accuracy. The use of 32-bit accumulators is crucial for preserving numerical precision during the accumulation of many 8-bit product terms. Furthermore, Gemmini's hardware includes dedicated logic for applying per-tensor or per-channel scaling factors associated with quanti-

zation (as defined by parameters like `mvin_scale_args` and `acc_scale_args` in its configuration), which is essential for correctly performing arithmetic in the quantized domain.

- **Systolic Array Dataflow Versatility**: The Gemmini architecture, by default and in this configuration, supports both Weight Stationary (WS) and Output Stationary (OS) dataflows within its systolic array. This architectural flexibility allows the runtime software or compiler to select the most efficient data movement and computation scheduling strategy for different types of neural network layers. For instance, WS dataflow is often optimal for convolutional layers where filter weights are reused extensively across the input feature map, while OS dataflow can be more advantageous for layers like fully-connected layers or where output values are reused. This adaptability enhances overall computational efficiency across diverse DNN models (Genc et al., 2021).

- **On-Chip Memory Subsystem**: Gemmini incorporates a substantial internal memory hierarchy specifically designed to sustain high throughput in the systolic array:

  - **Scratchpad Memory**: A 256KB SRAM-based scratchpad memory is provisioned within Gemmini. This high-speed, software-managed local memory is used to buffer input feature maps (activations) and model weights (filters) close to the PEs. Staging data in the scratchpad is fundamental to exploiting data reuse and minimizing latency by avoiding frequent accesses to slower off-chip system memory. The scratchpad is internally banked (typically configured with 4 banks) to allow for concurrent memory accesses, which helps in sustaining the data rate required by the PEs.

  - **Accumulator Memory**: A separate 64KB memory is dedicated to storing the partial sums and final accumulated values computed by the PEs. This accumulator memory often features wider data paths to accommodate the 32-bit accumulated values and is also banked (typically with 2 banks) to facilitate concurrent read/write operations as results are updated and potentially passed back to the scratchpad or to output buffers.

- **Hardware Acceleration for Ancillary DNN Operations**: Beyond the core matrix multiplication and convolution capabilities provided by the sys-

tolic array, the Gemmini unit is typically configured with hardware support for other common DNN operations. These often include max pooling and various non-linear activation functions (e.g., ReLU). Offloading these auxiliary operations to dedicated hardware within Gemmini further reduces the computational burden on the host CPU and streamlines the DNN processing pipeline.

- **Local TLB for Efficient Virtual Memory Addressing**: To operate effectively within the virtual memory environment managed by the Linux operating system running on the host CPU, Gemmini includes its own small Translation Lookaside Buffer (TLB), configured with 4 entries. This local TLB caches recently used virtual-to-physical address translations specifically for Gemmini's DMA operations. This significantly reduces the latency associated with address translation for Gemmini's memory requests by minimizing its reliance on the CPU's main page table walking hardware, which is especially beneficial when dealing with the large, and potentially non-contiguously allocated, data tensors common in DNN applications (Genc et al., 2021).

- **Instruction Control and Execution**: The host CPU controls Gemmini by issuing specialized custom instructions that are dispatched via the RoCC interface. These instructions fall into several categories: configuration instructions to set up Gemmini's operational modes and parameters (e.g., dataflow, activation functions); data movement instructions to trigger DMA transfers for loading input data and weights into the scratchpad (often termed 'mvin') and for storing results from the accumulator or scratchpad back to system memory ('mvout'); and computation instructions that initiate core operations like matrix multiplications or convolutions on the data resident in Gemmini's local memories.

This comprehensive set of features results in a Gemmini instance capable of performing 256 multiply-accumulate (MAC) operations per clock cycle on 8-bit input data, backed by a robust on-chip memory system and efficient mechanisms for data handling and control, making it a powerful DNN accelerator.

3.2.3 CPU Core and Base System Configuration

The SoC's general-purpose computational needs are met by a single "big" Rocket CPU core. This configuration is critical for running the Debian Linux operating

system, which in turn manages the overall system, file systems (on the SD card), networking (if an Ethernet controller from the ‘vivado-risc-v‘ project is included), and other user-space applications. The underlying base system configuration establishes essential parameters such as the physical memory map (defining address ranges for the DDR RAM, memory-mapped peripherals like UART and SD card controller, and the boot ROM), the initial boot sequence from the SD card (managed by OpenSBI and U-Boot, components typically part of the ‘vivado-risc-v‘ boot flow), and the setup for platform-level interrupt controllers. This forms the foundational environment for the CPU, the operating system, and the Gemmini accelerator.

3.3 Memory Hierarchy, Coherence, and System Interconnect

The SoC employs a multi-level memory hierarchy to ensure efficient data access for all components. The Rocket CPU is equipped with private L1 instruction and data caches, providing low-latency access to frequently used code and data. A system-level L2 cache, configured to be inclusive of the L1 caches, sits at the next level. This L2 cache acts as a larger, shared buffer, reducing the need for accesses to the main off-chip DDR memory for both the CPU and potentially for Gemmini's DMA engine if frequently accessed data blocks are resident in the L2. The inclusive nature of this L2 cache simplifies the cache coherence protocols required to maintain a consistent view of memory across the system.

All major components, including the CPU core(s), the L2 cache, and the RoCC interface for Gemmini, are connected via the TileLink interconnect fabric. TileLink is a cache-coherent, chip-scale interconnect protocol developed as part of the Rocket Chip project (Asanović et al., 2016). It supports mechanisms to ensure that all bus masters (like the CPU and Gemmini's DMA engine) have a consistent view of memory. This is crucial for correctness when data is produced by one component (e.g., the CPU pre-processing data) and consumed by another (e.g., Gemmini processing that data), or vice-versa. Gemmini's DMA operations are thus coherent with the CPU caches, meaning data transfers correctly reflect the latest state of memory across the system.

3.4 FPGA Realization and Software Stack for Linux Environment

The translation of the designed SoC into a functional hardware prototype on the AMD/Xilinx VC707 FPGA is facilitated by the ‘vivado-risc-v‘ project. This project not only provides Vivado project generation scripts for synthesizing the RISC-V SoC

(including the custom `Rocket64b1gem16` configuration with Gemmini) and producing an FPGA bitstream, but also orchestrates the preparation of a complete software stack necessary to boot and run a Debian Linux distribution. The Xilinx Vivado Design Suite (version 2024.2 as used in this work) is employed for synthesis, place-and-route, and bitstream generation. The ‘vivado-risc-v‘ Makefile automates much of this complex software and hardware build process.

3.4.1 Boot Process and Software Components

The ability to run a full Linux distribution like Debian on the custom SoC is critical for complex application development, system testing, and leveraging a rich ecosystem of tools and libraries. The boot process on the FPGA involves several stages, with key software components prepared and integrated by the ‘vivado-risc-v‘ build system:

1. **FPGA Boot ROM:** Upon power-on or reset, the RISC-V core begins execution from a small Boot ROM embedded within the FPGA logic. The contents of this Boot ROM are generated during the SoC build process. The ‘vivado-risc-v‘ Makefile processes Device Tree Source (DTS) files (‘system.dts‘ specific to the Rocket Chip configuration, augmented with board-specific DTS information like ‘board/vc707/bootrom.dts‘) to create a complete device tree and compile it into the ‘bootrom.img‘. This Boot ROM performs minimal hardware initialization and its primary role is to locate and load the first stage of the main bootloader, typically ‘boot.elf‘, from a designated location on the SD card.

2. **RISC-V Open Source Supervisor Binary Interface (OpenSBI):** The ‘boot.elf‘ file loaded from the SD card begins with OpenSBI. OpenSBI serves as a crucial firmware layer, providing a standardized Supervisor Binary Interface between the underlying hardware (in M-mode, Machine mode) and the subsequent bootloader or operating system kernel (which runs in S-mode, Supervisor mode). It handles low-level hardware initializations, platform-specific configurations, and provides runtime services (e.g., console I/O, inter-processor interrupts in multi-core systems, timer management) to the S-mode software. The ‘vivado-risc-v‘ Makefile, as seen in its targets for ‘opensbi/build/platform/vivado-risc-v/firmware/fw_payload.elf‘, compiles OpenSBI, often applying specific patches or configurations (‘patches/opensbi/*‘) and crucially packaging the next stage bootloader (U-Boot) as its payload.

3. **U-Boot (Universal Boot Loader):** Once OpenSBI initializes the necessary M-mode environment, it transfers control to its payload, which is U-Boot. U-Boot

is a versatile and widely used bootloader in embedded systems. Its responsibilities include more extensive hardware initialization (e.g., setting up the DDR memory controller, initializing network interfaces if network boot is intended, interacting with the SD card controller). A key function of U-Boot in this context is to read the Linux kernel image (‘Image‘) and the flattened device tree blob (DTB) from the SD card into DDR memory. The ‘vivado-risc-v‘ Makefile details the process for building U-Boot (‘u-boot/u-boot-nodtb.bin‘), which involves applying patches (‘patches/u-boot.patch‘) and using a board-specific configuration (‘vivado_riscv64_defconfig‘, ‘vivado_riscv64.h‘) tailored for the RISC-V SoC environment. After loading, U-Boot prepares the boot arguments and jumps to the Linux kernel entry point.

4. **Linux Kernel:** The Linux kernel is the heart of the operating system. The ‘vivado-risc-v‘ project facilitates the compilation of a RISC-V Linux kernel (from ‘linux-stable‘), customized for the target hardware. The Makefile shows that specific patches (‘patches/linux.patch‘) are applied, custom drivers for FPGA-specific AXI-based peripherals (like ‘fpga-axi-sdc.c‘ for the SD card controller and ‘fpga-axi-eth.c‘ for Ethernet, which are open-source Verilog controllers in the ‘vivado-risc-v‘ design) are integrated, and a specific kernel configuration (‘patches/linux.config‘) is used. This ensures that the kernel can correctly recognize and manage the SoC's custom hardware components, including the peripherals essential for a functional system.

5. **Debian Root Filesystem:** Once the Linux kernel is booted, it mounts a root filesystem, which contains all the user-space applications, libraries, and system utilities that constitute the Debian distribution. The ‘vivado-risc-v‘ project simplifies this by providing mechanisms to use a pre-built Debian RISC-V root filesystem (‘rootfs.tar.gz‘), which is then typically extracted and written to a partition on the SD card by the ‘./mk-sd-card‘ utility script. This allows the system to boot into a familiar Linux environment with access to package management (‘apt‘) and a vast collection of pre-compiled software.

The ‘vivado-risc-v‘ Makefile orchestrates the download of prebuilt toolchains (e.g., ‘workspace/gcc/tools.tar.gz‘), patching of submodules (OpenSBI, U-Boot, Linux kernel, Rocket Chip itself), compilation of these boot components, and generation of HDL from Chisel sources. The ‘./mk-sd-card‘ script then assembles the ‘boot.elf‘ (OpenSBI+U-Boot), Linux kernel image, and the Debian root filesystem onto an SD card, making it bootable on the FPGA once programmed with the corresponding bitstream.

This comprehensive software stack, coupled with the custom hardware, transforms

the FPGA into a fully operational RISC-V Linux server, providing a rich platform for developing and testing the time-series anomaly detection application accelerated by the Gemmini co-processor.

3.5 Chipyard Framework and Rationale for Time-Series Anomaly Detection

`Rocket64b1gem16`) configuration was developed to address the computational demands inherent in time-series anomaly detection tasks that rely on Deep Neural Networks. The 16×16 systolic array within Gemmini delivers substantial parallel processing capability, ideal for the matrix-vector and matrix-matrix multiplications, as well as convolutions, that form the backbone of modern neural network architectures (such as LSTMs, Transformers, or specialized CNNs) frequently employed for analyzing sequential data. The 256KB on-chip scratchpad memory in Gemmini is dimensioned to locally store significant segments of input time-series data and the corresponding model parameters (weights and biases) during the processing of individual neural network layers. This local storage and reuse minimize the latency and energy overhead associated with frequent accesses to off-chip DDR memory.

The adoption of 8-bit quantized integer arithmetic, supported by the configured Gemmini instance, is a strategic choice for this application. Quantization significantly reduces the memory footprint of the deployed DNN models and lessens the bandwidth required for transferring weights and activations. These efficiencies are paramount when dealing with potentially long and high-volume time-series datasets, with the goal of maintaining high anomaly detection accuracy. The robust host CPU, running a full Debian Linux environment, provides the flexibility required for complex data preprocessing (e.g., normalization, feature extraction from raw time-series signals), post-processing of anomaly detection outputs, managing network communication, and handling overall system control and user interaction, while the Gemmini accelerator is dedicated to the high-performance execution of the core DNN inference computations.

This carefully architected SoC, realized on the VC707 FPGA via the ‘vivado-risc-v‘ project and running a full Debian Linux distribution, represents a balanced hardware platform. It combines the versatility of general-purpose processing with the efficiency of specialized, high-throughput hardware acceleration, specifically tailored to meet the challenges of deploying sophisticated DNN models for time-series anomaly detection.

3.6 Implementation of the Time-Series Anomaly Detection System

TÀI LIỆU THAM KHẢO

Tiếng Anh

[1] Alon Amid et al. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (2020), pp. 10–21. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996616.

[2] Krste Asanovi'c and David A. Patterson. *Instruction sets should be free: The case for RISC-V*. Tech. rep. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, 2014. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[3] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. Electrical Engineering and Computer Sciences, University of California, Berkeley, 2016. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[4] Hasan Genc et al. *Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures*. 2019. arXiv: 1911.09925 [cs.AR].

[5] Sagar Karandikar et al. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 29–42. DOI: 10.1109/ISCA.2018.00013.

[6] Andrew Waterman et al. *The RISC-V instruction set manual volume II: Privileged architecture version 1.7*. Tech. rep. UCB/EECS-2015-49. EECS Department, University of California, Berkeley, 2015. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-49.html.

CHƯƠNG A: MÃ NGUỒN CỦA THIẾT KẾ

A.1 Mã Verilog Top-Level cho DMANiosV

CHƯƠNG B: MỘT SỐ LỖI THƯỜNG GẶP

Phần này bao gồm một số lỗi thường gặp và cách khắc phục trong quá trình triển khai và gỡ lỗi hệ thống Nios V với DMA.

B.1 Lỗi Tạo BSP hoặc Biên dịch Ứng dụng

Một lỗi phổ biến khi chạy lệnh `niosv-app` là báo không tìm thấy tệp hoặc thư mục nguồn.

- **Nguyên nhân:** Đường dẫn đến thư mục `app`, `bsp`, hoặc tệp mã nguồn C (`-s=...`) không chính xác so với vị trí hiện tại (hoặc không tồn tại) của Nios V Shell.
- **Khắc phục:** Đảm bảo đã `cd` vào thư mục gốc của dự án Quartus (`DMANiosVIntern`) trước khi chạy lệnh `niosv-app`. Kiểm tra lại các đường dẫn tương đối (`software/app`, `software/bsp`) trong lệnh. Tên tệp mã nguồn C trong thư mục `app` phải khớp với tên được chỉ định bởi tham số `-s`.