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

KHÓA LUẬN TỐT NGHIỆP CỬ NHÂN

Nghiên cứu FPGA ứng dụng AI

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

KHÓA LUẬN TỐT NGHIỆP CỬ NHÂN

Nghiên cứu FPGA ứng dụng AI

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

MỤC LỤC i

DANH SÁCH CÁC HÌNH iii

DANH SÁCH CÁC BẢNG v

1 INTRODUCTION 1

2 RISC-V Microprocessor Instruction Set Architecture 3
  2.1 Overview of RISC-V 3
    2.1.1 RISC-V 32-bit and the RV32I Base Instruction Set 5
    2.1.2 RISC-V 64-bit and the RV64I Base Instruction Set 8
    2.1.3 RISC-V Processor Cores 11
    2.1.4 Pipeline Architecture in Microprocessors 11
    2.1.5 Out-of-Order Architecture in Microprocessors 12
    2.1.6 Open-Source RISC-V Processor Cores 13
    2.1.7 Chipyard - Rocket Chip 15

3 A Multi-Core RISC-V Processor System 19
  3.1 System Design Methodology with `Chisel` 19
    3.1.1 `Chisel` and `Scala` 19
    3.1.2 `Verilator` Simulation Tool 20
  3.2 The `Rocket` Processor Core 21
  3.3 `Rocket Chip` SoC Structure 22
    3.3.1 Bus and Interconnect 22
    3.3.2 Memory and Cache 26
    3.3.3 Core-Complex 27
    3.3.4 Peripheral Components 28

4 The Gemmini Deep Learning Accelerator 33
  4.1 Introduction to Gemmini 33
  4.2 Gemmini Architecture 33
    4.2.1 Systolic Array (Spatial Array) 34
    4.2.2 Local Scratchpad Memory 34
    4.2.3 Accumulator Memory 35

4.2.4 Controller and DMA Engine 35
4.2.5 Optional Functional Units 35
4.2.6 Virtual Address Translation Support 36
4.3 Gemmini Integration as a RoCC Accelerator 36
4.4 Gemmini Programming Model 37
4.4.1 Low-Level: Gemmini ISA (RoCC Instructions) 37
4.4.2 Mid-Level: Hand-Tuned C Library 38
4.4.3 High-Level: Compiler Support (Conceptual) 39
4.5 Gemmini Configuration in This Thesis 39
4.6 Summary 39

TÀI LIỆU THAM KHẢO **41**

DANH SÁCH CÁC HÌNH

2.1 RISC-V is managed by RISC-V International. Source [6] 3
2.2 Instruction encoding formats in RV32I. Source [10] 6
2.3 Instructions in the RV32I base instruction set. Source [10] 7
2.4 Instruction encoding format for ADDIW in 64-bit architecture. [10] . . 9
2.5 Encoding format for 64-bit I-type shift instructions. [10] 9
2.6 Encoding format for 64-bit R-type shift instructions for word operations. [10] 10
2.7 Encoding formats for 64-bit LOAD and STORE instructions. [10] . . . 10
2.8 Process of handling instructions through pipeline stages. 11
2.9 Pipeline of Rocket core. Source [1] 13
2.10 Berkeley Out-of-Order Machine Processor Architecture. Source [3] . . . 14
2.11 Chipyard, developed by UC Berkeley. Source: [11] 16
2.12 Component structure of Chipyard. Source: [11] 16

3.1 `Chisel` as a library within `Scala`. Source: CHIPS Alliance. 19
3.2 Structure of a `Rocket` Tile. 22
3.3 Proposed structure of a single-core RISC-V processor system with a `Gemmini` `RoCC` accelerator. 23
3.4 Connection model in `TileLink` and its interfaces. Source: [8] 24
3.5 Channels in a `TileLink` connection between two agents. Source: [8] . . 25
3.6 `TileLink` `TL-UL` connection structure. Source: [8] 27
3.7 Implementation structure of a heterogeneous multi-core processor system. 28
3.8 Peripherals in the Rocket Chip base system. 29
3.9 `RoCC` co-processor core and MMIO peripheral core in `Rocket Chip`. . . 30

4.1 High-level overview of the `Gemmini` accelerator integrated with a host CPU (`Rocket Core`) and system memory hierarchy. 33
4.2 Conceptual diagram of the `Gemmini` RoCC interface with the host CPU (`RISC-V CPU`), PTW, and L2 Cache/System Memory. 36
4.3 Layered programming model for `Gemmini`. 37

DANH SÁCH CÁC BẢNG

2.1 ABI names and functions of registers in RV32I. Source [10] 6
2.2 Pseudo-instructions and their corresponding standard instructions. Source [10] 8
2.3 Development status of extended RISC-V instruction sets. Source [10] . 8

3.1 `TileLink` operational configurations. Source: [8] 24

CHƯƠNG 1: INTRODUCTION

CHƯƠNG 2: RISC-V Microprocessor Instruction Set Architecture

2.1 Overview of RISC-V

Designing and developing a microprocessor, or Central Processing Unit (CPU), is a complex process requiring the collaboration of many experts in fields such as digital logic design, compiler development, and operating systems, along with significant financial resources. Major vendors like ARM, with prominent processor lines such as Cortex-M and Cortex-A, often impose high licensing fees and strict confidentiality requirements on their designs. This creates considerable barriers to hardware and software research and customization, particularly for academic institutions or small businesses. In response to these limitations, the RISC-V architecture [10, 9] was developed and introduced as an open-source solution, providing a flexible platform that eliminates licensing costs and allows for need-based customization. By fostering innovation and opening up a freer research ecosystem, RISC-V is reshaping the traditional approach to microprocessor development, aiming for more efficient and comprehensive solutions.

Hình 2.1: RISC-V is managed by RISC-V International. Source [6]

Essentially, RISC-V is an open-source microprocessor instruction set architecture (ISA), designed and developed based on the Reduced Instruction Set Computer (RISC) methodology, and it represents the fifth generation of this design approach. For a program executing on a microprocessor, the RISC design methodology stipulates that the microprocessor executes only simple assembly (machine code) instructions, which have a fixed length and are executed within a single, uniform clock cycle. A complex task is performed by multiple assembly instructions, making the instruction set simple, compact, and easy to develop hardware for rapidly. This is entirely different from the Complex Instruction Set Computer (CISC) architecture, which aims to complete a task with the fewest possible assembly instructions. To achieve this, CISC microprocessors must be designed to execute structurally complex assembly instructions, capable of performing multiple specialized functions sequentially and executing over multiple clock cycles.

Originating as a project developed by researchers and graduate students at the

University of California, Berkeley in 2010, the initial purpose of RISC-V was to create a practical ISA to support research and teaching in computer architecture, deployable on both hardware and software without licensing requirements. The original authors of RISC-V were individuals with considerable experience in computer design. Consequently, for stable development, RISC-V aimed for several goals, most of which have been achieved, including:

- A complete, fully open-source ISA, free for academic, educational, and industrial use, with practical applicability suitable not only for experimental simulation but also for hardware implementation.
- An ISA not oriented towards complex microarchitectures or fixed by any specific implementation technology (FPGA, ASIC, full-custom, etc.) but still allowing for efficient implementation in any of these directions.
- An ISA divided into base ISAs, usable for developing hardware accelerators or for educational purposes, and optional standard extensions to support general software development.
- An ISA supporting extensive instruction set extensions with specialized functionalities and numerous variants.
- Support for the IEEE 754-2008 floating-point standard.
- Support for parallel, multi-core, and heterogeneous many-core implementations.
- Both 32-bit and 64-bit address space variants developed for applications, operating system kernels, and hardware implementations.

Due to its open-source nature, RISC-V allows developers to freely use the design architecture without paying intellectual property and licensing fees. This creates development opportunities for RISC-V itself as it is adopted by the research community and educational institutions, gradually becoming popular in industry, and attracting significant attention from many large companies, corporations, and even governments of various countries. Furthermore, designers using the RISC-V ISA can utilize it without limitations in both hardware and software. This means developers are encouraged, but not required, to publicly disclose their designs and projects developed using RISC-V.

Currently, the development process and rights regarding the publication, release, and maintenance of intellectual property and official documentation related to RISC-V are managed by RISC-V International, based in Switzerland. This meets the standardization requirements for designers and users for commercial purposes, who demand a system using the RISC-V ISA that can operate and exist for many years.

To suit various implementers and ensure flexibility in designs with specialized characteristics, the RISC-V instruction set is divided and standardized into two main parts: the base instruction set and standard extensions. The RISC-V instruction set supports 32-bit, 64-bit, and 128-bit architectures with the respective base instruction sets RV32I, RV32E, RV64I, and RV128I. Among these, RV32I is the most important and common set for 32-bit microprocessors, featuring 40 integer manipulation instructions. The fact that RISC-V only standardizes and provides the instruction set allows hardware designers the freedom to choose how to implement their designs.

2.1.1 RISC-V 32-bit and the RV32I Base Instruction Set

RV32I [10] is the base instruction set present in almost all 32-bit RISC-V microprocessor architectures. The assembly instructions in this set use 32 32-bit registers, named `x0` - `x31`, to store integers. Each register, in addition to being numbered from 0 to 31, is also referred to by its Application Binary Interface (ABI) name, which reflects its specific function. For example, register `x0` is a read-only register containing the constant value "0", and register `x1` is used as the return address register, storing the return address when a function is executed. Table 2.1 lists these registers and their functions in detail.

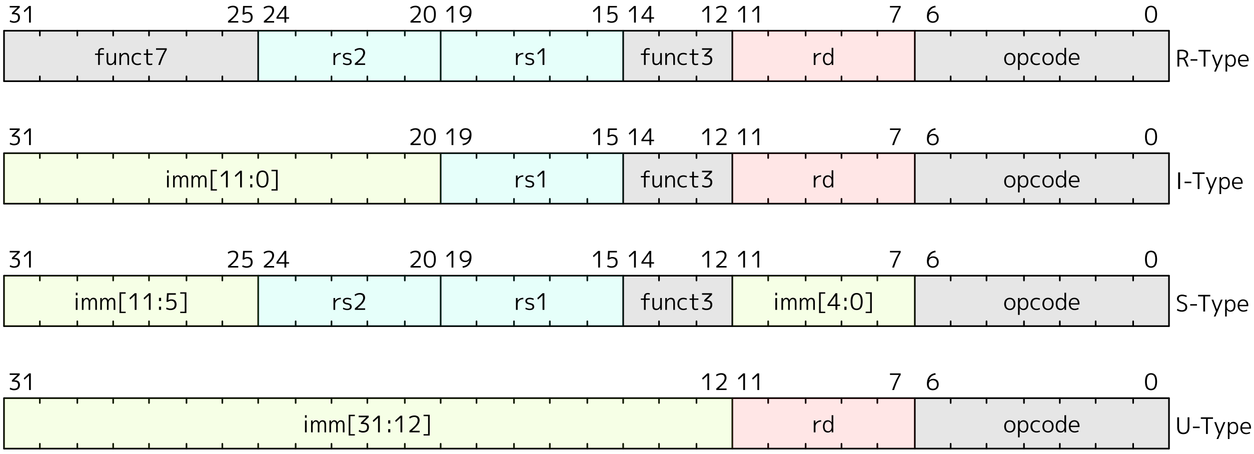
The assembly instructions in RV32I use six different 32-bit instruction formats, depending on the function of the instruction. These include R, I, S, U formats, as well as two additional formats for immediate values: B and J. These formats are clearly shown in Figure 2.2. Notably, the source registers (`rs1` and `rs2`) and the destination register (`rd`) are kept in the same position in most formats, simplifying instruction decoding. The only discernible difference lies in how `rs2`, `rd`, and immediate values are organized within the instruction.

Figure 2.3 details all instructions in the RV32I base instruction set with their respective formats, specific opcode values, and function codes (`funct3` and `funct7`). Functionally, the RV32I set is divided into five distinct groups:

- 21 integer computation instructions: Perform arithmetic (addition, subtraction, bit shifts), logic (AND, OR, XOR, NOT, etc.), and comparison operations

Bảng 2.1: ABI names and functions of registers in RV32I. Source [10]

Register	ABI Name	Description
x0	zero	Constant "0"
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5 → x7	t0 → t2	Temporary
x8	s0/fp	Memory/frame pointer
x9	s1	Memory
x10, x11	a0, a1	Function parameter/return value
x12 → x17	a2 → a7	Function parameter
x18 → x27	s2 → s11	Memory
x28 → x31	t3 → t6	Temporary



Hình 2.2: Instruction encoding formats in RV32I. Source [10]

Note: This figure should depict the R, I, S, U, B, and J formats as shown in the RISC-V specification.

on signed and unsigned numbers. Typically use R-type and I-type formats.

- 8 load-store instructions: LOAD and STORE byte, half-word, word (signed and unsigned), typically using I-type and S-type formats.
- 8 control-flow/branch instructions: Conditional branches, performing comparisons between two registers to decide whether to branch (BEQ/BNE, BLT, BGE), typically using I-type format.
- 2 system access instructions for the execution environment - ECALL and breakpoint - EBREAK, using I-type format.

- 1 memory access ordering instruction - FENCE, used by other processing cores (in multi-core scenarios) or other peripheral devices in the system. The I-type format is used for this instruction with some specific bit encoding rules and opcodes.

Placeholder for RV32I Instruction List Diagram

Hình 2.3: Instructions in the RV32I base instruction set. Source [10]
Note: This figure should list the RV32I instructions as shown in the RISC-V specification.

With these 40 instructions, the microprocessor can perform most basic operations on integers. Additionally, several standard pseudo-instructions are defined to add tasks for which there are no dedicated instructions in the set. For example, the MOVE instruction - "`mv rd, rs`" - performs the function of moving/copying data between two registers. In a program, this instruction might be generated in the assembly code by a RISC-V compiler but is actually converted by assemblers into a standard instruction when translating to machine code, namely ADDI - Add Immediate "`addi rd, rs, 0`", which adds the source register to the constant "0" and assigns it to the destination register, achieving a similar function. Some other pseudo-instructions and their equivalent standard instructions are shown in Table 2.2.

As mentioned, a goal of RISC-V is to support implementations requiring more features beyond the base instruction set. This is reflected in RISC-V defining extensions [10] for its instruction set architecture. These extensions define additional instruction encodings for various functionalities. For example, the standard M extension adds instructions for integer multiplication and division, while the standard F extension defines instructions for single-precision floating-point computation and manipulation.

Table 2.3 above clearly lists the standard extension sets developed and published by RISC-V International. It can be seen that many instruction sets have been ratified and can be used in designs. When implemented, architectures supporting the RV32I base instruction set (or RV64I depending on the architecture) combined with the standard extensions M, A, F, D, Zicsr, and Zifencei are often summarized and referred to as the RV32G (or RV64G) instruction set, named for its general-purpose utility.

Bảng 2.2: Pseudo-instructions and their corresponding standard instructions. Source [10]

Pseudo-instruction	**Corresponding Core Instruction**	**Meaning**
nop	addi rd, x0, x0	No operation
li rd, imm	addi rd, x0, imm	Load constant (immediate)
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement (invert)
neg rd, rs	sub rd, x0, rs	Two's complement (negative)
seqz rd, rs	sltiu rd, rs, 1	Compare equal to zero
snez rd, rs	sltiu rd, x0, rs	Compare not equal to zero
sltz rd, rs	slt rd, rs, x0	Compare less than zero
sgtz rd, rs	slt rd, x0, rs	Compare greater than zero
beqz rs, offset	beq rs, x0, offset	Branch if equal to zero
bnez rs, offset	bne rs, x0, offset	Branch if not equal to zero
blez rs, offset	bge x0, rs, offset	Branch if less than or equal to zer
bgez rs, offset	bge rs, x0, offset	Branch if greater than or equal to
bltz rs, offset	blt rs, x0, offset	Branch if less than zero
bgtz rs, offset	blt x0, rs, offset	Branch if greater than zero
j offset	jal x0, offset	Jump to label (offset)
jal offset	jal x1, offset	Jump and link register
ret	jalr x0, 0(x1)	Return from subroutine

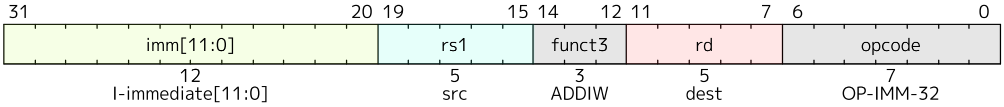
Bảng 2.3: Development status of extended RISC-V instruction sets. Source [10]

Instruction Set	**Function Description**	**Version**	**Status**
M	Integer multiplication and division instruction set	2.0	Approved
A	Atomic memory operation instruction set	2.1	Approved
F	Single-precision floating-point computation	2.2	Approved
D	Double-precision floating-point computation	2.2	Approved
Q	Quad-precision floating-point computation	2.2	Approved
C	Compressed instruction set	2.0	Approved
Zifence	Instruction-Fetch Fence instruction set	2.0	Approved
Zicsr	Control and Status Registers (CSRs) instruction set	2.0	Approved
Ztso	Total Store Ordering instruction set	1.0	Approved

2.1.2 RISC-V 64-bit and the RV64I Base Instruction Set

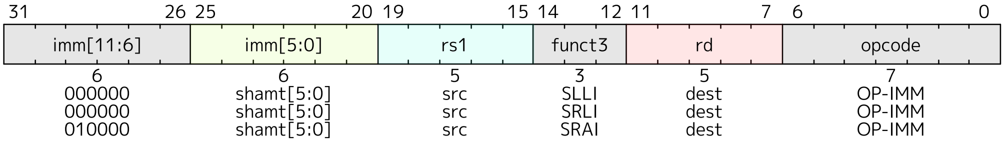
Similar to RV32I, RV64I is the base instruction set for 64-bit microprocessor architectures, featuring 64-bit wide registers and supporting 64-bit addressing (XLEN=64), though instructions remain 32-bits wide. Architectures compliant with RV64I are compatible with RV32 instructions and include additional instructions to support operations between 32-bit and 64-bit data. For example, the `LW` instruction loads a 32-bit value and sign-extends it into a 64-bit register, while the new `LD` instruction loads a 64-bit word. The `ADD` instruction now performs 64-bit addition, while the `ADDW` instruction handles 32-bit addition, ignoring the upper 32 bits and returning a 32-bit signed value, sign-extended to 64 bits. These instructions enabling backward compatibility typically have a "W" suffix to denote 32-bit wide operations.

The compiler and calling conventions maintain the invariant that all 32-bit values are stored in sign-extended format within 64-bit registers. Even 32-bit unsigned integers will have bit 31 extended into bits 63 through 32. Consequently, converting between 32-bit signed and unsigned integers is a no-operation, as is converting from a 32-bit signed integer to a 64-bit signed integer. Existing 64-bit instructions like `SLTU` (set less than unsigned) and branch comparisons continue to operate correctly on 32-bit unsigned integers under this rule. Similarly, 64-bit logical operations on 32-bit sign-extended integers preserve the sign-extension property. A few new instructions (`ADD[I]W/SUBW/SxxW`) are necessary for addition and shifts to ensure reasonable performance for 32-bit values.


31
20
19
15
14
12
11
7
6
0
imm[11:0]
rs1
funct3
rd
opcode
12
I-immediate[11:0]
5
src
3
ADDIW
5
dest
7
OP-IMM-32

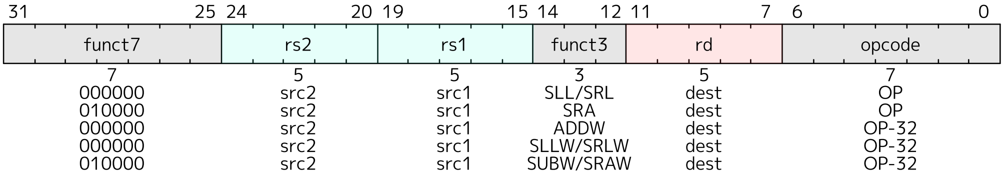
Hình 2.4: Instruction encoding format for ADDIW in 64-bit architecture. [10]

Some distinctly different instructions found only in RV64I include the `ADDIW` instruction, as shown in Figure 2.4. `ADDIW` performs an addition between a 12-bit sign-extended immediate value and register `rs1`, producing a proper 32-bit sign-extended result in `rd`. Overflows are ignored, and the result consists of the lower 32 bits of the sum, sign-extended to 64 bits. Note, the instruction "`addiw rd, rs1, 0`" writes the sign-extension of the lower 32 bits of register `rs1` into register `rd` (pseudo-instruction `SEXT.W`).


31
26
25
20
19
15
14
12
11
7
6
0
imm[11:6]
imm[5:0]
rs1
funct3
rd
opcode
6
000000
000000
010000
6
shamt[5:0]
shamt[5:0]
shamt[5:0]
5
src
src
src
3
SLLI
SRLI
SRAI
5
dest
dest
dest
7
OP-IMM
OP-IMM
OP-IMM

Hình 2.5: Encoding format for 64-bit I-type shift instructions. [10]

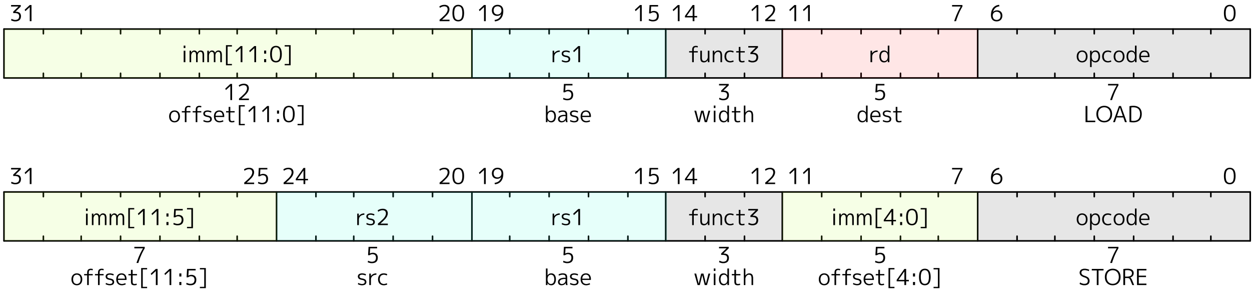
I-type shift operations in RV64I (Figure 2.5) use the same opcode as RV32I with some modifications. Register `rs1` contains the operand, and the shift amount is encoded in the lower 6 bits of the I-immediate field. Bit 30 determines the shift type: `SLLI` (logical left shift, zeros shifted into lower bits), `SRLI` (logical right shift, zeros shifted into upper bits), and `SRAI` (arithmetic right shift, sign bit copied into vacated upper bits).


31 25 24 20 19 15 14 12 11 7 6 0
funct7
rs2
rs1
funct3
rd
opcode
7 5 5 3 5 7
000000 src2 src1 SLL/SRL dest OP
010000 src2 src1 SRA dest OP
000000 src2 src1 ADDW dest OP-32
000000 src2 src1 SLLW/SRLW dest OP-32
010000 src2 src1 SUBW/SRAW dest OP-32

Hình 2.6: Encoding format for 64-bit R-type shift instructions for word operations. [10]

The `ADDW` and `SUBW` instructions in RV64I, similar to `ADD` and `SUB` for 64-bit operations, operate on 32-bit data and produce a 32-bit signed result, which is then sign-extended to 64-bits before being written to the destination register. Overflows are also ignored. The `SLL`, `SRL`, and `SRA` instructions perform logical left, logical right, and arithmetic right shifts on the value in `rs1`, with the shift amount determined by the lower 6 bits of `rs2`. The `SLLW`, `SRLW`, and `SRAW` instructions (Figure 2.6), unique to RV64I, operate on 32-bit values and use the lower 5 bits of `rs2` to determine the shift amount.

Finally, load-store instructions can be mentioned. RV64I extends the address space to 64 bits, but the number of accessible addresses may vary depending on the execution environment.


31 20 19 15 14 12 11 7 6 0
imm[11:0]
rs1
funct3
rd
opcode
12 5 3 5 7
offset[11:0] base width dest LOAD
31 25 24 20 19 15 14 12 11 7 6 0
imm[11:5]
rs2
rs1
funct3
imm[4:0]
opcode
7 5 5 3 5 7
offset[11:5] src base width offset[4:0] STORE

Hình 2.7: Encoding formats for 64-bit LOAD and STORE instructions. [10]

The instruction formats are shown in Figure 2.7. The `LD` instruction here will load a 64-bit value, `LW` loads a 32-bit value (sign-extended to 64-bits) from memory into a register, and `LWU` zero-extends a 32-bit value from memory into register `rd`. Similarly, `LH/LHU` and `LB/LBU` handle 16-bit and 8-bit values. The `SD`, `SW`, `SH`, and `SB` instructions store 64-bit, 32-bit, 16-bit, and 8-bit values from register `rs2` to memory, respectively.

2.1.3 RISC-V Processor Cores

With its open-source nature, system developers can freely implement designs based on the RISC-V architecture, supported by an increasingly robust and growing ecosystem. The RISC-V ISA ecosystem mentioned here includes a diverse range of components from hardware design tools, low-level firmware development, bootloaders, to fully functional operating system kernels. Currently, development tools such as GCC/LLVM compilers, QEMU and Verilator simulators, and even the Linux operating system have incorporated RISC-V architecture and are under active development. As of the time of writing this thesis, there are over 100 open-source hardware processor cores and more than 40 open-source SoC platforms using the RISC-V ISA that have been and continue to be developed.

This section will provide a general overview of pipeline, in-order, and out-of-order architectures commonly implemented in microprocessors. Subsequently, a brief survey of open-source RISC-V processor cores will be conducted to select the most suitable design for implementation.

2.1.4 Pipeline Architecture in Microprocessors

A modern general-purpose CPU is often designed using a pipeline architecture. This technique draws inspiration from industrial assembly lines, dividing the processing of an assembly instruction into multiple stages or segments, each performed by different specialized hardware units connected sequentially throughout the process and capable of continuous operation. This architecture is commonly found in many microprocessor designs using RISC in general and RISC-V ISA in particular, due to its higher performance compared to processing instructions sequentially and waiting for results.

Placeholder for Pipeline Stages Diagram

Hình 2.8: Process of handling instructions through pipeline stages.

The concept of pipelining is illustrated in Figure 2.8. While one assembly instruction is being processed in later stages, another assembly instruction begins processing in earlier stages, in parallel with previous instructions. It can be seen that at the fourth clock cycle, the pipeline is processing four different instructions in parallel, each in a distinct stage. A basic microprocessor pipeline consists of five stages. However, depending on the implementation and design, a microprocessor may use more or fewer stages, but still follows the sequence of stages explained below:

- **Instruction Fetch (IF):** Retrieves instructions from the instruction memory, also known as program memory. Each instruction code that the CPU must execute is stored in the program memory. These instruction codes are binary sequences compiled from C/C++ code or assembly code.
- **Instruction Decode (ID):** The binary of each instruction is decoded, and corresponding control signals are activated.
- **Execute (EX):** Performs calculations and processing corresponding to the decoded instruction, such as addition, subtraction, multiplication, division, shifts, assignments, branches, etc.
- **Memory Access (MEM):** Accesses memory and performs read/write operations of data from/to memory.
- **Writeback (WB):** Saves the results after instruction execution to the location specified by the instruction, such as registers.

2.1.5 Out-of-Order Architecture in Microprocessors

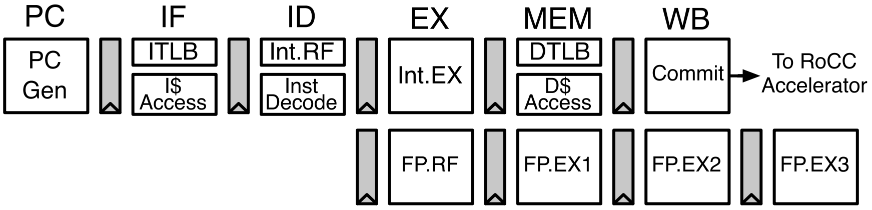
Alternatively, out-of-order execution architecture is a crucial paradigm in modern CPU design, delivering significant performance gains by allowing instructions to be executed as soon as their input operands are ready, rather than strictly adhering to the program order as in a pipeline. This architecture is also a core element in high-performance systems used in scientific computing and artificial intelligence research. This flexible execution model addresses performance bottlenecks caused by data dependencies, cache latencies, and pipeline stalls, ensuring more efficient utilization of computational resources.

Key components of this architecture include the instruction window, which stores multiple instructions for dependency analysis, and dynamic scheduling mechanisms such as Tomasulo's algorithm or scoreboarding, which track dependencies and resource availability. Another critical feature is the Re-order Buffer (ROB), which helps maintain program order during instruction retirement, ensuring correct architectural state updates and precise exception handling. By decoupling instruction fetch and execution stages, out-of-order processors achieve higher instruction throughput and sustain the performance of execution units, optimizing efficiency across various applications.

Despite its many strengths and implementation benefits, out-of-order processors also present significant design challenges. The hardware blocks required for components like the ROB and branch prediction increase complexity, chip area, and power consumption, making energy efficiency optimization an important research area. Furthermore, extending out-of-order architecture to meet the demands of heterogeneous processing and large-scale parallel computing requires innovative approaches in dynamic scheduling and resource management.

2.1.6 Open-Source RISC-V Processor Cores

Rocket [1], or Rocket Core, is a RISC-V instruction set architecture microprocessor core designed by the Berkeley Architecture Research group. Rocket features a 5-stage, in-order architecture and implements the RISC-V RV32G and RV64G instruction sets.



Hình 2.9: Pipeline of Rocket core. Source [1]

The design of this microprocessor integrates many advanced features, including a Memory Management Unit (MMU) that supports virtual memory, enabling efficient memory access and address translation. Besides its role as a general-purpose microprocessor, Rocket also serves as a library of reusable components, including functional units, caches, Translation Lookaside Buffer (TLB), Page Table Walker (PTW), and Control and Status Registers (CSRs), which are reused in many other microprocessor designs. This makes Rocket Core a foundational component in Chipyard - RocketChip, allowing for the rapid development and creation of experimental systems ranging from simple to complex.

Rocket is considered the most prominent microprocessor core in the open-source RISC-V development community due to its well-updated and stably supported development documentation. It is built using Chisel, a relatively new hardware construction language based on Scala, employing an object-oriented approach similar to software development. Therefore, the flexibility of the system development process and the applicability of Rocket Core are very high, making it suitable for implementing multi-core systems.

Along with Rocket Core, the Berkeley Out-of-Order Machine (BOOM) [3, 2] is also a prominent open-source RISC-V core, designed for high performance and out-of-order execution. Also developed within the microprocessor architecture research group at the University of California, Berkeley, BOOM is aimed at applications in advanced academic research and industry, featuring a flexible design and high operational performance. BOOM is also written in the Chisel language as a generator, like Rocket Core, allowing for rapid deployment and customization of the core according to the RV64GC RISC-V architecture. The latest version, SonicBOOM [12] (also known as BOOMv3), achieves a performance of 6.2 CoreMarks/MHz, capable of competing directly with high-performance commercial processor cores, setting a high standard for open-source processor design.

Placeholder for BOOM Architecture Diagram

Hình 2.10: Berkeley Out-of-Order Machine Processor Architecture. Source [3]

Structurally, BOOM features a 10-stage pipeline architecture including Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback, and Commit. These stages allow for detailed control and optimization of the instruction flow, maintaining the "out-of-order" functionality at the high speeds necessary for modern processors. Specifically, each stage performs as follows:

- **Fetch:** Instructions are fetched from instruction memory and stored in a fetch buffer to prepare for processing.
- **Decode:** Instructions are converted into simpler micro-ops to optimize execution within the pipeline.
- **Register Rename:** Logical registers in the ISA are renamed to independent physical registers, reducing data conflicts and increasing processing efficiency.
- **Dispatch:** Micro-ops from the decode stage are sent to the Issue window, awaiting necessary operands.
- **Issue:** Instructions in the Issue window are checked for operand readiness, and when conditions are met, micro-ops are sent to an execution unit.

- **Register File (RF) Read:** Micro-ops read operands from the physical register file or bypass network, preparing data for the next stage.

- **Execute:** Micro-ops are processed by functional units, performing computational tasks and memory address calculations.

- **Memory:** Manages memory operations via Load Address Queue (LAQ), Store Address Queue (SAQ), and Store Data Queue (SDQ), performing loads when addresses are ready and stores when both address and data are available.

- **Writeback:** Results from operations are written back to the physical register file, ensuring updated values are stored for future instructions.

- **Commit:** The Reorder Buffer tracks the status of instructions in the pipeline and commits results when instructions complete, including storing data to memory.

Similar to Rocket Core, this is also considered one of the most prominent open-source microprocessor cores. However, BOOM's design is more geared towards ASIC implementation, and its numerous features along with the use of Chisel for implementation might pose difficulties in creating a multi-core design in Verilog and deploying it on FPGAs.

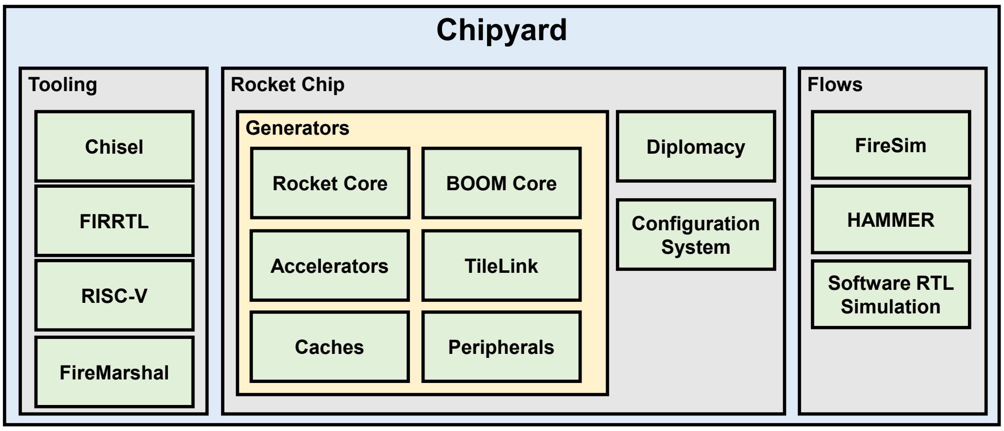
2.1.7 Chipyard - Rocket Chip

The Rocket Chip project, also developed by the Berkeley Architecture Research (BAR) group at the University of California, Berkeley, marked a significant advancement in open-source processor design. This project provides a flexible, scalable, and configurable platform, enabling the creation of custom RISC-V microprocessors tailored to diverse performance and application requirements. Rocket Chip is essentially a library of hardware generators that can be configured and interconnected in various ways, producing diverse SoC designs. This library can generate an in-order core (Rocket) or an out-of-order core (BOOM), and these cores can be connected to co-processors via an interface named RoCC (Rocket Custom Coprocessor) [1].

Subsequently, to develop and expand the capabilities of Rocket Chip, Chipyard was created. Chipyard is an open-source framework that integrates system design, simulation, and implementation for SoC development, and it has become one of the most actively embraced and developed frameworks by the community. Chipyard, also

Hình 2.11: Chipyard, developed by UC Berkeley. Source: [11]

originating from UC Berkeley, is the result of continuous development and integration of Chisel, Rocket Chip, and the Rocket Core.


Chipyard
Tooling
Chisel
FIRRTL
RISC-V
FireMarshal
Rocket Chip
Generators
Rocket Core
BOOM Core
Accelerators
TileLink
Caches
Peripherals
Diplomacy
Configuration System
Flows
FireSim
HAMMER
Software RTL Simulation

Hình 2.12: Component structure of Chipyard. Source: [11]

By utilizing Chisel, the FIRRTL compiler, the Rocket Chip generator, and other tools, Chipyard facilitates the development of highly customizable RISC-V SoCs. Chipyard includes a diverse set of processor cores (Rocket Core, BOOM, Ariane), accelerators (Hwacha, Gemmini, SHA3), along with comprehensive memory systems, peripheral devices, and tools to support the construction of a full-featured SoC. Chipyard supports multiple development flows, allowing for flexible hardware extension and experimentation. These development flows include software-based Register Transfer Level (RTL) simulation, FPGA-accelerated simulation using FireSim, automated VLSI flows with Hammer, and FireMarshal, which enables the creation and loading of software on both bare-metal systems (similar to microcontrollers) and Linux-based systems.

Rocket Chip supports deployment on FPGAs, with operational frequencies reaching 25-100 MHz depending on the configuration and the type of FPGA used. Currently, Rocket Chip is being utilized by SiFive [7], a company founded by some of the authors of RISC-V. SiFive produces RISC-V processors on silicon based on Rocket and provides tools for developing, testing, and evaluating their designs.

Using Chipyard (or Rocket Chip) as a foundation for system construction can pose several challenges due to its complexity in terms of the number of components and the design language used—Chisel. However, overcoming this barrier yields numerous benefits due to Rocket Chip's high flexibility and system customization capabilities. Its components are developed independently, have detailed development documentation, and are easily extensible. Furthermore, the Rocket Chip support community is very active, with more frequent updates and improvements compared to other frameworks.

CHƯƠNG 3: A Multi-Core RISC-V Processor System

From the surveys and analyses in the preceding sections regarding processor cores and the two SoC development frameworks, it is evident that each of these research subjects possesses distinct advantages and limitations. However, for the research orientation and implementation of a processor system featuring a custom accelerator, the `Chipyard` framework and its library components are deemed most suitable for the objectives of this thesis. Not only does `Chipyard` effectively meet current requirements, but it also holds the potential for expansion to support more complex systems in the future.

This chapter will present a more detailed exposition of the component structure and the process of constructing an SoC system, while also delving into the architecture of the `Rocket` processor core. The thesis will utilize high-level design languages `Chisel/Scala` to implement a single-core processor system integrated with the `Gemmini RoCC` accelerator, focusing on leveraging the components within an SoC system, and building the hardware and software design based on the `Chipyard` framework.

3.1 System Design Methodology with `Chisel`

As previously introduced, this thesis undertakes system design based on the components and tools within `Chipyard`. This framework and its library components are constructed from a diverse range of elements, all based on `Chisel/Scala`.

3.1.1 `Chisel` and `Scala`

CHISEL
Scala
FIRRTL
CHIPS
ALLIANCE

Hình 3.1: `Chisel` as a library within `Scala`. Source: CHIPS Alliance.

`Chisel` is a Hardware Construction Language (HCL) developed at UC Berkeley, initially intended to enable hardware design through highly parameterized and hierarchically specific code generators. `Chisel` is used to define these generators, which then produce designs in `Verilog`, suitable for industry-standard RTL design flows. `Chisel` is classified as a Domain-Specific Language (DSL); more precisely, it is implemented as a library package for `Scala`, a high-level, multi-paradigm programming language that integrates features of both Object-Oriented Programming (OOP) and functional programming on the `Java` Virtual Machine (JVM). Furthermore, `Scala`'s functional programming capabilities support higher-order functions and immutable data structures, facilitating more maintainable code, which is crucial for reproducibility and collaboration in scientific research.

`Chisel`'s design aims to address the lack of object-oriented programming and preprocessing structures in existing Hardware Description Languages (HDLs), supporting the construction of more flexible generators. By being based on `Scala`, `Chisel` gains the flexibility to describe hardware with high levels of abstraction and modularity. This helps hardware designers describe circuits in a way that is easy to maintain, modify, and extend, especially for large and complex designs such as microprocessors and specialized digital signal processing cores. `Scala`'s robust type system, diverse numerical representations and variables, immutability, and support for functional programming make it an ideal foundation for `Chisel`, aiding in error checking during hardware and circuit description and encouraging reusable design patterns.

In the context of hardware design based on `Chisel` and `Scala`, `sbt` (Simple Build Tool) plays a crucial role, optimizing the design management and development process. Built specifically for `Scala` and `Java` projects, `sbt` is the build tool used by the majority of `Scala` developers. Essentially, `sbt` acts as the compiler when working with `Chisel` and `Scala`. Using `sbt` simplifies the process of structuring projects for hardware design with `Chisel` in `Chipyard`, as well as enabling integration with many other open-source projects.

3.1.2 `Verilator` Simulation Tool

In the hardware design process, evaluating the functionality of the design is critically important. With `Chipyard`, system designs, from simple to complex structures, can be effectively realized and tested thanks to integrated tools and RTL simulators like `Verilator`, which are readily available within the project framework.

`Verilator` is an open-source program, prominent in the field of RTL simulation due to its unique method of converting ("Verilating") `Verilog` or `SystemVerilog` into `C++` or `SystemC`, creating a highly optimized executable program. Unlike traditional simulators, `Verilator` does not directly interpret `Verilog` during simulation. Instead, it generates `C++` or `SystemC` program files by parsing the hardware code, performing syntax checking, linting, and optionally inserting debug points and coverage analysis to enhance the correctness and accuracy of evaluation and verification. This "Verilated" output file can be configured to run in single-threaded or multi-threaded mode on a development server, offering an advantage for simulating complex or computationally demanding designs. After the "Verilated" code is generated, it is compiled by a standard `C++` compiler such as `GCC`, `Clang`, or `MSVC++`, and is typically combined with a user-provided wrapper file to manage the initialization of the Verilated model.

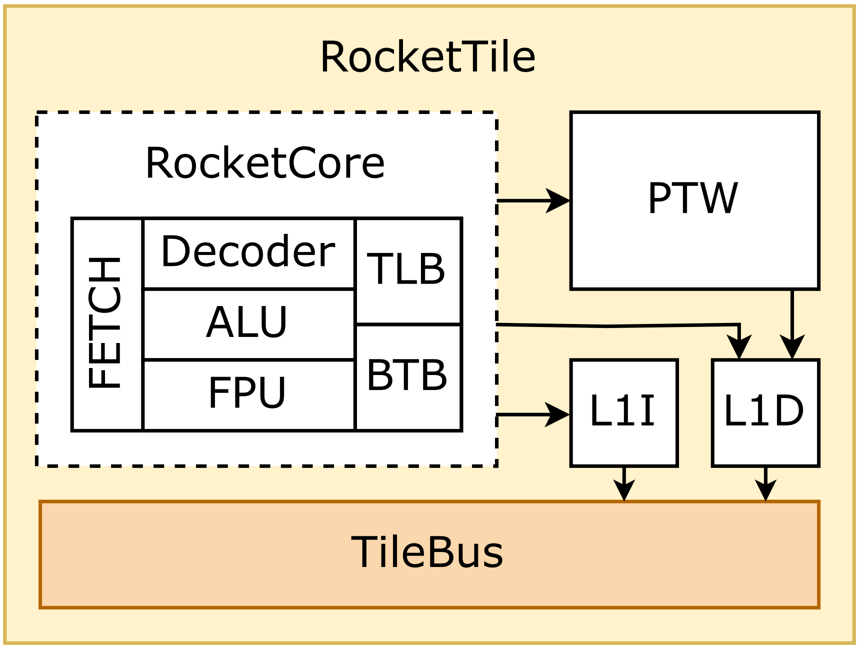
`Verilator` offers high simulation performance, especially for designs requiring substantial resources, due to its high speed and cycle accuracy. `Verilator`'s compilation approach makes it up to 100 times faster than interpretive simulators like Icarus `Verilog` (which is also open-source), and with multi-threading, performance can increase by an additional 2-10 times, achieving a total improvement of up to 1000 times. This makes `Verilator` suitable for RISC-V multi-core processor designs ranging from simple to complex, and competitive even with commercial simulators such as Synopsys VCS, Cadence Incisive, or Mentor ModelSim.

3.2 The `Rocket` Processor Core

As introduced in the previous chapter, `Rocket` is a 5-stage, in-order processor core compliant with the RISC-V instruction set architecture, with RV32G and RV64G variants. Currently, the development of the `Rocket` core is maintained by the CHIPS Alliance. `Rocket` is written in the `Chisel` hardware design language, based on the `Scala` programming language, and is a key component of the `Rocket Chip` SoC Generator toolchain.

When implemented, each `Rocket` core is combined with L1 data and instruction caches. This memory operates locally for each core and, in a multi-core system, is responsible for storing temporary variables generated during the execution of the pipeline, helping to reduce congestion when multiple cores attempt to store data. Together with the Page Table Walker (`PTW`), a Tile within the SoC is formed. This `Rocket` Tile (Figure 3.2) is considered a fundamental unit in the `Rocket Chip` SoC system and can be replicated to create multi-core processor systems, meeting the demands for performance and flexibility in modern SoC design.

Due to the object-oriented nature of `Chisel`, the language used for its design, `Rocket` itself is written from object classes that can be defined and reconfigured according to implementation needs, such as `WithNBigCore()`, `WithNMedCore()`, `WithNSmallCore(` and `WithNTinyCore()`. Additionally, instead of implementing the `Rocket` core, the



Hình 3.2: Structure of a `Rocket` Tile.

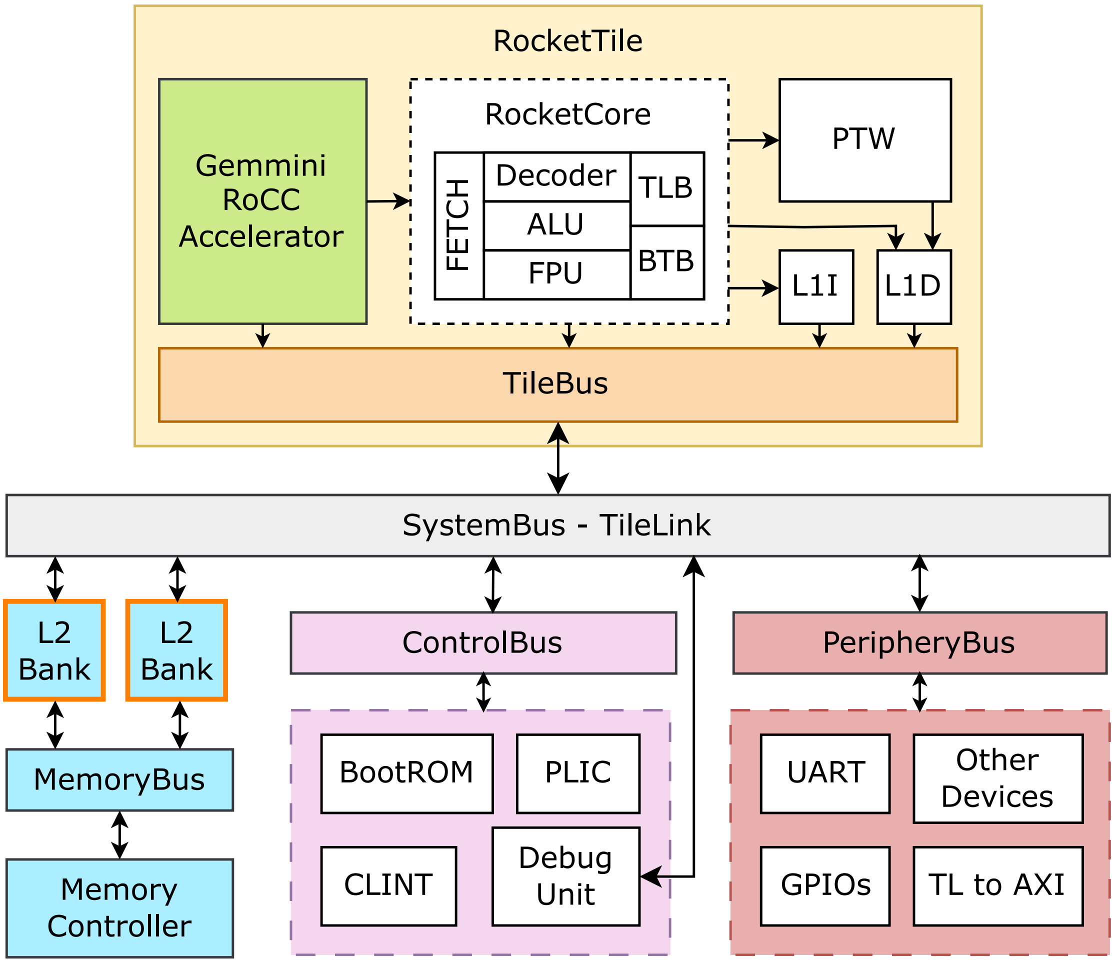
`BOOM` core can also be integrated in its place, and many other peripherals can be integrated into a Tile unit.

3.3 `Rocket Chip` SoC Structure

This system generator includes multiple components, not just the processor core but also the on-chip system bus (SoC bus), specifically `TileLink`. `TileLink` comprises a peripheral bus, control bus, and memory bus, along with arbiters and system controllers. Additionally, the system integrates peripheral components such as DRAM, GPIO, UART, and, critically for this thesis, supports tightly-coupled accelerators like `Gemmini` via the `RoCC` interface. Figure 3.3 below depicts the generalized structure of the single-core RISC-V processor system with a `Gemmini RoCC` accelerator, which forms the basis for the development in this thesis.

3.3.1 Bus and Interconnect

In the design of Systems-on-Chip (SoCs), bus architecture plays a backbone role, connecting and coordinating the operation of all components within the system. Particularly for modern multi-core processor systems, development trends focus on using crossbar-type bus architectures instead of Time Shared Buses. A crossbar, specifically a fully-connected crossbar structure, is considered a form of hardware switching

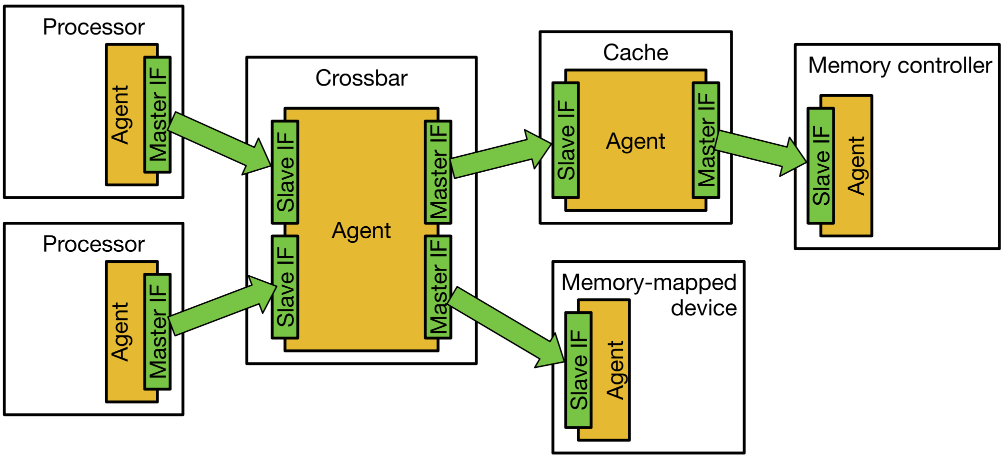

RocketTile
Gemmini RoCC Accelerator
RocketCore
FETCH
Decoder
ALU
FPU
TLB
BTB
PTW
L1I
L1D
TileBus
SystemBus - TileLink
L2 Bank
L2 Bank
MemoryBus
Memory Controller
ControlBus
BootROM
PLIC
CLINT
Debug Unit
PeripheryBus
UART
Other Devices
GPIOs
TL to AXI

Hình 3.3: Proposed structure of a single-core RISC-V processor system with a `Gemmini RoCC` accelerator.

fabric, allowing any input port to connect directly to any output port simultaneously. The outstanding advantage of this architecture lies in its ability to provide high bandwidth and low latency, effectively meeting performance requirements in applications such as networking equipment, multi-core processors, and high-performance storage systems. The optimization in data transmission offered by this architecture is a key factor in improving the overall performance of modern SoC systems.

For `Rocket Chip`, the `TileLink` [8] system communication bus was developed concurrently with the project and the RISC-V instruction set, and is widely used in open-source system research at present.

`TileLink` is a highly parameterizable, open-source, shared memory interconnect protocol based on the aforementioned crossbar structure, designed to connect diverse modules for SoCs. `TileLink` is implemented via a hierarchical, point-to-point network that can be easily scaled. This protocol provides memory-mapped access for multiple


Processor
Agent
Master IF
Processor
Agent
Master IF
Crossbar
Slave IF
Slave IF
Agent
Master IF
Master IF
Cache
Slave IF
Agent
Master IF
Memory controller
Slave IF
Agent
Memory-mapped device
Slave IF
Agent

Hình 3.4: Connection model in `TileLink` and its interfaces. Source: [8]

controllers, ensuring memory coherency while supporting efficient communication between processors, DMAs, and peripheral devices.

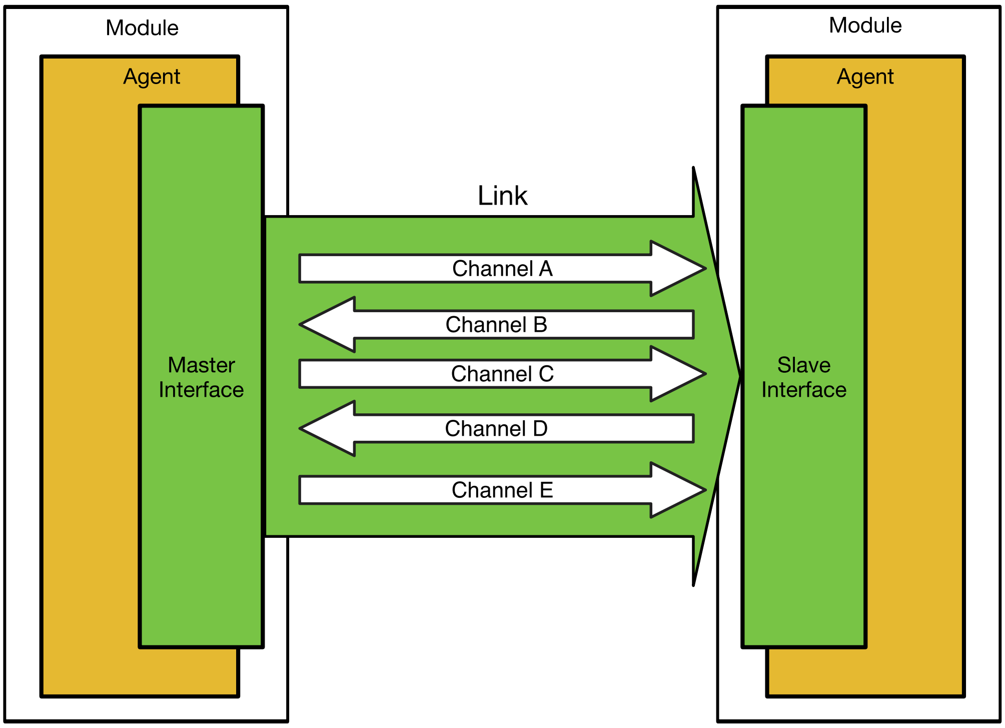
A `TileLink` bus system can support a combination of multiple communicating agents, each capable of supporting different configured subsets of the protocol. The `TileLink` specification includes three levels of operational configuration for agents, presented in Table 3.1.

Bảng 3.1: `TileLink` operational configurations. Source: [8]

	TL-UL	**TL-UH**	**TL-C**
Read/Write operations	y	y	y
Multibeat messages	.	y	y
Atomic operations	.	y	y
Hint operations	.	y	y
Cache block transfers	.	.	y
Channels B+C+E	.	.	y

`TileLink` clearly distinguishes between cached and uncached communications. The simplest level is `TileLink` Uncached Lightweight (`TL-UL`), which only supports basic read and write operations to memory, for individual words. The next, more complex level is `TileLink` Uncached Heavyweight (`TL-UH`), which adds features such as "hints," "atomic" operations, and burst accesses but does not support coherent caching. Here, "hints" refers to a feature where a request command seeks data information external to the executing command from masters, and "atomic" refers to operations that lock access rights in a system with multiple masters, which could be microprocessors. Finally, `TileLink` Cached (`TL-C`) is the full protocol, supporting the use of coherent caches within the system.

For integrating peripheral hardware, all three bus configuration levels can be used. The communication connection diagram, at its most complete level, between a custom peripheral hardware and an agent is shown in Figure 3.5.



Hình 3.5: Channels in a `TileLink` connection between two agents. Source: [8]

In this `TileLink` protocol, the defined communication channels include `A`, `B`, `C`, `D`, and `E`, each undertaking specific roles in the interaction process between a master (M) and a slave (S), and all have a defined direction of transmission as depicted in the figure above.

- **Channel A:** This is the initiation channel, where M sends requests to S, establishing the initial step of the communication process.
- **Channel B:** This channel is used when M requests a cache block from S, ensuring necessary data is queried.
- **Channel C:** Used for S to respond to M's request for a cache block, playing a crucial role in providing necessary data to M.
- **Channel D:** This is the general response channel, where S sends the final response to M after processing requests.

- **Channel E:** This channel ensures transaction completion by performing the final handshake for cache block transfer between the two parties.

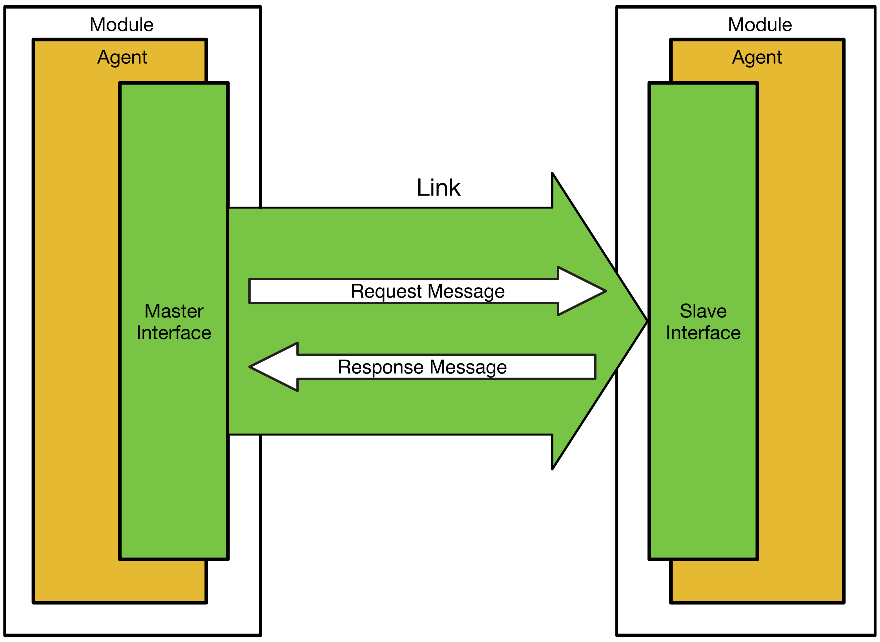
The priority order for these channels is as follows: `A` < `B` < `C` < `D` < `E`, where channel `A` has the lowest priority and `E` has the highest. Prioritization in the `TileLink` bus system ensures that messages transmitted through the system do not fall into a hold-and-wait loop. In other words, the message flow through all channels between agents is maintained as a directed acyclic graph, enabling the `TileLink` protocol to operate without encountering deadlocks, ensuring continuous and uninterrupted data flow within the system.

The choice of `TileLink` bus configuration is relevant for connecting various memory-mapped peripherals within the system. For general-purpose MMIO peripherals, which might include cryptographic algorithm accelerators designed in `Verilog` HDL (if they were to be integrated as MMIO devices), the `TL-UL` configuration, comprising only channels `A` and `D` between M and S (as shown in Figure 3.6), is often sufficient. However, for the primary accelerator in this thesis, the `Gemmini` core, integration is achieved through the `RoCC` interface, which provides a more tightly-coupled path to the processor core, rather than as a `TileLink`-attached MMIO peripheral. Details on the `RoCC` interface are discussed further in Section 3.3.4. For other standard MMIO peripherals, a wrapper from the `Rocket Chip` library, using `Chisel/Scala`, can be inherited and customized to connect their signals to the appropriate system bus, typically the peripheral bus.

At the system configuration level, a wrapper from the `Rocket Chip` library, using `Chisel/Scala`, can be inherited and customized to convert the signals of the accelerator core to the system bus, in this case, the peripheral bus. Details for this interface will be further elaborated in subsequent sections of this thesis.

3.3.2 Memory and Cache

Each Tile in the system is integrated with a processor core and an L1 cache that is flexibly configurable. By default design with `Chisel`, the L1 cache has a size of 16 KiB, organized multi-dimensionally with sets and ways. The L1 cache uses a set-associative cache structure, dividing the memory region into sets, each set containing a fixed number of data blocks (ways). When a memory address access request occurs, the index derived from the address maps it to a specific set. The cache controller then searches for the data block within the ways of that set. This design supports



Hình 3.6: `TileLink` `TL-UL` connection structure. Source: [8]

both instruction and data storage, optimizing access performance and increasing the hit rate compared to caches with lower associativity.

In addition to the L1 cache, the system also integrates a shared L2 cache, connected to all cores via the `TileLink` bus system. The L2 cache acts as an intermediate storage repository, allowing processor cores and other components in the system to perform read/write data requests. This is a coordination mechanism that helps reduce pressure on the main memory and improves overall processing performance. The L2 cache controller also ensures fairness in resource allocation, using a queuing system to manage access requests.

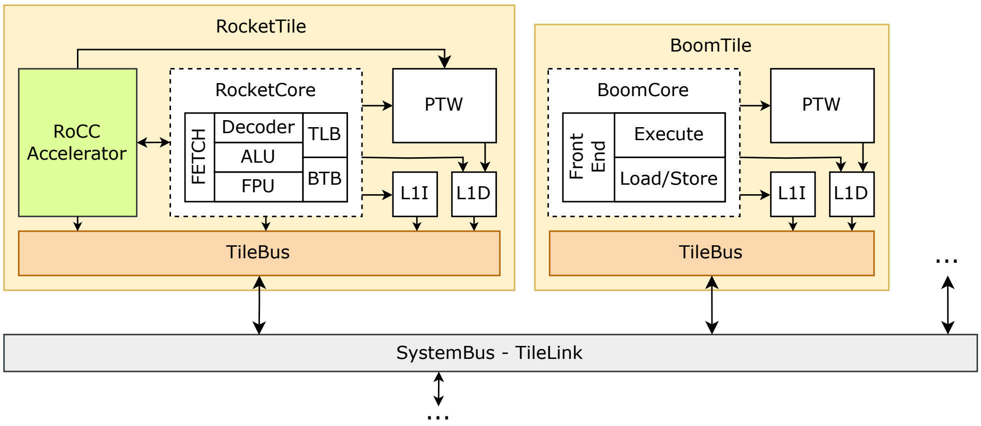
At a higher level, the system also supports the integration of external memory, such as DRAM or SRAM, through open-source controllers or specialized IPs from FPGA implementation toolkits. Additionally, an interface with `DRAMSim` is supported for integration when performing full-system simulation. This provides flexibility and scalability, meeting various requirements in practical applications.

3.3.3 Core-Complex

As previously mentioned, the `Chipyard` framework and library consist of many component definition classes that can form a `Rocket Chip` system. This allows for the formation of a complex system with diverse processor cores and peripherals, specifically heterogeneous architecture systems comprising multiple different processors like

`Rocket` and `BOOM` coexisting on the system.

In these multi-core processor structures, each individual core will be uniquely identified with a different hart ID, ordered according to the system configuration. An example can be referenced below with a configuration comprising one `Rocket` core and one `BOOM` core. The corresponding block diagram configuration when generating the system will be as shown in Figure 3.7 below.



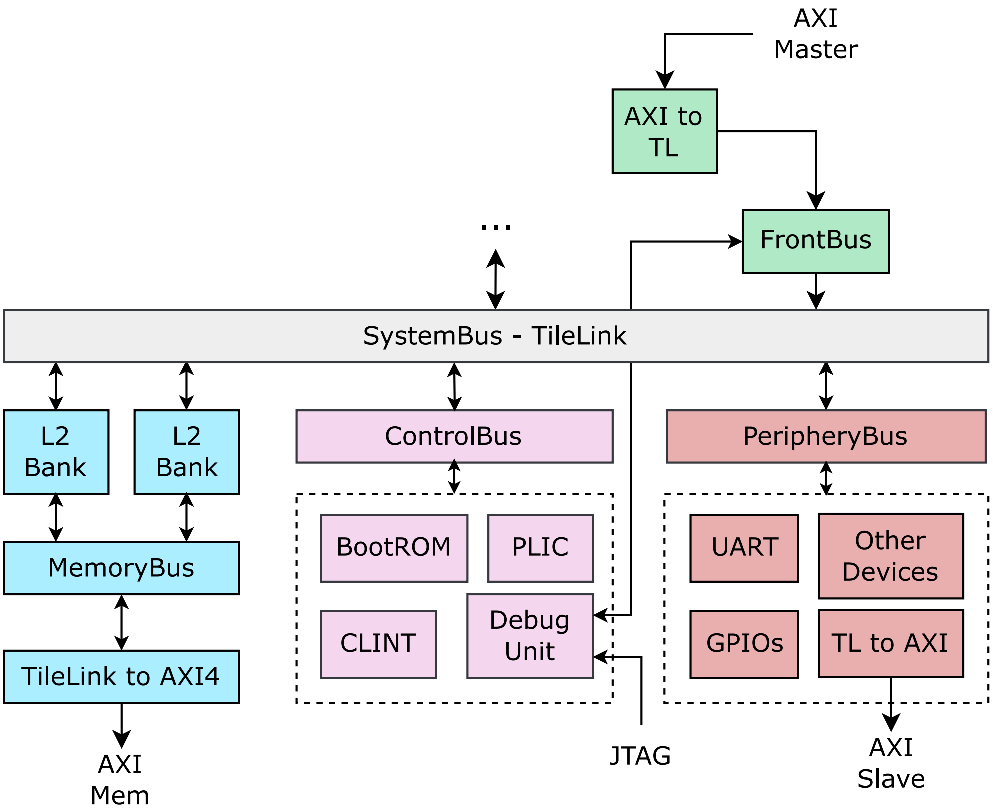
Hình 3.7: Implementation structure of a heterogeneous multi-core processor system.

A heterogeneous SoC system is an implementation approach also considered in this thesis to combine different processor cores to optimize performance and energy efficiency. At the application scale, lightweight cores can handle simple tasks with low power consumption, while high-performance cores support hyper-threading and out-of-order execution to process complex tasks. This design ensures flexibility, resource optimization, and superior performance for diverse applications.

3.3.4 Peripheral Components

In the base system configuration class of `Rocket Chip`, many peripheral components are initialized and integrated by default to support both simulation and deployment on FPGA hardware for the SoC system. This configuration includes basic I/O devices such as PWM, interrupt controllers, JTAG, ROM, external memory, and standard peripheral interfaces, ensuring that basic communication, storage, and control requirements are met. In particular, common peripheral devices like UART, SPI Flash, and GPIO are implemented to support serial communication, storage management, and flexible control within the system.

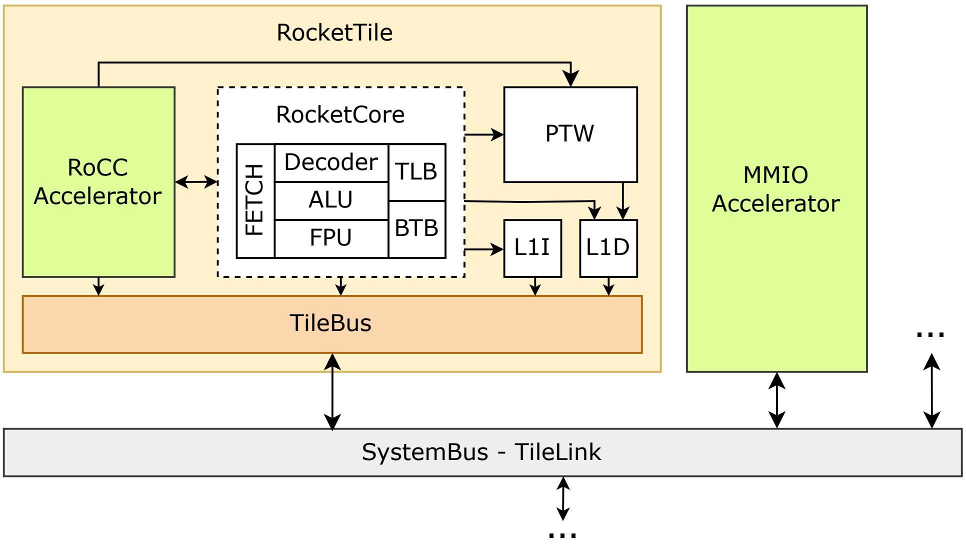
All these components are developed based on the open-source `Rocket Chip` library, with hardware designs like `Rocket` and `BOOM` realized on the `Chisel` platform.


AXI
Master
AXI to TL
FrontBus
...
SystemBus - TileLink
L2 Bank
L2 Bank
MemoryBus
TileLink to AXI4
AXI
Mem
ControlBus
BootROM
PLIC
CLINT
Debug Unit
JTAG
PeripheryBus
UART
Other Devices
GPIOs
TL to AXI
AXI
Slave

Hình 3.8: Peripherals in the Rocket Chip base system.

When external memory is needed in design and simulation, the `AXI4` (Advanced eXtensible Interface 4) interface can be integrated as an intermediate layer, allowing effective connection with DRAM or SPI Flash models. Concurrently, serial protocols like JTAG are integrated to ensure accurate debugging capabilities and efficient serial communication, configured to connect with the `TileLink` bus. The `Rocket Chip` system also supports flexible configuration through special control ports, including custom boot pins and chip identification ports. Furthermore, the `Rocket Chip` system provides the capability to directly map bus interfaces such as `AXI4` Memory and MMIO (Memory-Mapped IO), allowing for expanded compatibility and optimized performance in complex SoC applications.

To expand the capability of integrating components and peripheral cores to form highly parameterized custom systems suitable for specific applications, the `Rocket Chip` architecture provides two flexible methods for peripheral integration. Specifically, the system can integrate a peripheral core as a `RoCC` (`Rocket` Custom Coprocessor), located within each tile, or integrate at the system communication level via the MMIO mechanism. These methods facilitate in-depth customization, meeting various requirements for each specific application.



Hình 3.9: `RoCC` co-processor core and MMIO peripheral core in `Rocket Chip`.

MMIO Peripherals are custom peripherals attached directly to the `TileLink` bus, while Tightly-Coupled `RoCC` Accelerators are custom peripherals attached directly to the arbiter and processor within each Tile, in parallel with the processor core. With the `TileLink`-Attached MMIO method, the processor communicates with MMIO devices through registers mapped into memory at addresses on the system. Through these registers, control mechanisms or data transfer can be customized to suit the application of the peripheral core. Conversely, in the `RoCC` Accelerator method, communication is performed via a custom protocol and non-standard ISA instructions, which are configured and defined separately within the encoding space of the RISC-V ISA. For `Rocket` and `BOOM`, each core can control up to four accelerators through custom opcodes, sharing resources with the microprocessor. Instructions executed with `RoCC` cores have the format `customX rd, rs1, rs2, funct`. Here, `X` is a number from 0-3, determining the opcode that controls the routing of the instruction to a specific accelerator. The fields `rd`, `rs1`, `rs2` are the register numbers for the destination register and two source registers, and the `funct` field is a 7-bit integer that the accelerator can use to differentiate various instructions.

Peripherals connected as `RoCCs` have a distinct advantage due to their high customizability and direct connection and control via custom ISA instructions. This allows for specialized performance optimization, particularly useful for cores requiring large, complex computational workloads or applications needing extremely fast response times. Communication via a custom protocol and leveraging the RISC-V ISA helps `RoCC` utilize CPU resources efficiently, while also offering flexibility in designing specialized accelerator processors. The `Gemmini` accelerator, central to this thesis, is a prime example of a `RoCC`-based component, designed for high-throughput matrix multiplication and tightly integrated with the `Rocket` core.

While `RoCC` implementation can require a custom software toolchain for its specialized instructions, the performance benefits for demanding applications like those targeted by `Gemmini` often justify this investment. For this thesis, the integration of the `Gemmini` accelerator via the `RoCC` interface is deemed the most appropriate approach to achieve the desired performance for deep learning workloads.

Conversely, the MMIO `TileLink`-Attached method is widely used for integrating general-purpose memory-mapped peripheral devices. In this model, peripheral devices provide a set of registers (e.g., `Chisel` Registers) to communicate with microprocessors or other masters on the system bus. This method is suitable for peripherals where the tight coupling and custom instruction set of `RoCC` are not necessary.

To minimize design complexity for MMIO peripherals, the `Rocket Chip` architecture provides the `regmap` interface, which allows for the automatic generation of much of the necessary linking logic. Designers can use `TLRegisterRouter` or create a `LazyModule` and initialize `TLRegisterNode` to integrate such devices. Since `TileLink` is the official protocol used in SoCs based on `Rocket Chip` and `Chipyard`, designing MMIO peripheral devices (other than the primary `RoCC` accelerator) would typically focus on integration with this protocol.

CHƯƠNG 4: The Gemmini Deep Learning Accelerator

As established in the previous chapter, the System-on-Chip (SoC) developed in this thesis utilizes a single `Rocket Core` augmented with a specialized accelerator for deep learning workloads. This chapter delves into the specifics of this accelerator: `Gemmini`. `Gemmini` is an open-source, full-stack Deep Neural Network (DNN) accelerator generator, originating from UC Berkeley, designed to facilitate the systematic evaluation of deep-learning architectures [5]. Its integration as a Rocket Custom Coprocessor (`RoCC`) within the `Chipyard` framework makes it a powerful tool for research and development in domain-specific hardware acceleration.

4.1 Introduction to Gemmini

The proliferation of Deep Neural Networks (DNNs) across various domains, from computer vision to natural language processing, has created a significant demand for efficient hardware solutions. While general-purpose CPUs can execute DNNs, their performance and energy efficiency are often suboptimal for these computationally intensive tasks. `Gemmini` addresses this by providing a generator for producing highly configurable, systolic array-based DNN accelerators [4].

A key advantage of `Gemmini` is its full-stack approach. It not only generates the synthesizable Register Transfer Level (RTL) code for the accelerator hardware but also provides flexible programming stacks and enables integration into complete SoCs with shared resources. This allows for a comprehensive evaluation of system-level effects, such as memory contention and OS overheads, which are often overlooked when designing accelerators in isolation [5]. For this thesis, leveraging `Gemmini` allows for the exploration of a powerful DNN accelerator within a realistic SoC environment built using `Chipyard`.

4.2 Gemmini Architecture

The `Gemmini` architecture is designed around a flexible template that can be parameterized to generate a wide range of accelerator instances. The core components of a `Gemmini` accelerator are detailed below.

Hình 4.1: High-level overview of the `Gemmini` accelerator integrated with a host CPU (`Rocket Core`) and system memory hierarchy.

4.2.1 Systolic Array (Spatial Array)

The heart of `Gemmini` is its systolic array, also referred to as a spatial array. This array consists of a grid of Processing Elements (`PEs`), each capable of performing Multiply-Accumulate (`MAC`) operations. The systolic array is highly configurable in terms of:

- **Dimensions:** The number of rows and columns in the `PE` grid (e.g., 16x16, 32x32).
- **Dataflow:** `Gemmini` supports different dataflow strategies, primarily Weight Stationary (`WS`) and Output Stationary (`OS`). In `WS` dataflow, weights are preloaded into the `PEs` and remain stationary while input activations stream through. In `OS` dataflow, partial sums of the output activations remain stationary within the `PEs` while inputs and weights are streamed. The choice of dataflow can significantly impact performance and energy efficiency depending on the DNN layer characteristics and memory access patterns.
- **Pipelining:** The degree of pipelining within the array and `PEs` affects the clock speed and throughput.

This configurable systolic array allows `Gemmini` to efficiently execute dense matrix multiplications, which are fundamental operations in many DNN layers, particularly fully connected and convolutional layers (often after an `im2col` transformation).

4.2.2 Local Scratchpad Memory

`Gemmini` includes a dedicated on-chip scratchpad memory. This is a software-managed SRAM used for staging input activations, weights, and intermediate feature maps close to the systolic array.

- **Purpose:** Reduces the latency and energy consumption associated with fetching data from or writing data to higher levels of the memory hierarchy (e.g., L2 cache or main memory/DRAM).
- **Configurability:** The capacity of the scratchpad (e.g., 256 KB) and the number of banks it is divided into are configurable parameters. Multiple banks allow for parallel access, potentially improving bandwidth.

Efficient management of the scratchpad memory is crucial for achieving high utilization of the systolic array.

4.2.3 Accumulator Memory

Separate from the main scratchpad, `Gemmini` features an accumulator memory.

- **Purpose:** Stores the partial sums generated by the `MAC` operations in the systolic array. These sums are typically of higher precision than the input data types to maintain accuracy during accumulation.
- **Configurability:** The capacity and banking of the accumulator memory are also configurable.

After accumulation, the results can be scaled, quantized, and passed through activation functions before being written back to the scratchpad or main memory.

4.2.4 Controller and DMA Engine

A dedicated controller manages the overall operation of the `Gemmini` accelerator. It decodes instructions received from the host CPU via the `RoCC` interface and orchestrates the data movements and computations. `Gemmini` incorporates a Direct Memory Access (`DMA`) engine responsible for:

- Transferring data between the external memory (L2 cache/DRAM) and the internal scratchpad memory.
- Moving data between the scratchpad and the accumulator memory.

The DMA engine operates in parallel with the systolic array computations to hide memory latency.

4.2.5 Optional Functional Units

`Gemmini` can be configured with several optional hardware units to accelerate common non-GEMM (General Matrix Multiply) DNN operations. These include:

- **Activation Functions:** Hardware support for functions like ReLU.
- **Pooling Layers:** Max-pooling or average-pooling operations.
- **Normalization/Scaling:** For quantizing or scaling results.
- **Transposer:** For matrix transposition if needed by the dataflow or layer.

- **Im2col Unit:** A dedicated unit to perform the `im2col` transformation (converting image patches into columns for convolution via GEMM) directly in hardware, which can offload the host CPU.

These units can be included or excluded during the `Gemmini` generation process (elaboration time) based on target application requirements, allowing for trade-offs between area/flexibility and performance.

4.2.6 Virtual Address Translation Support

To operate within a system using virtual memory (like Linux-based SoCs), `Gemmini`'s DMA needs to translate virtual addresses (used by software) to physical addresses (used by hardware).

- `Gemmini` typically utilizes the host CPU's Page Table Walker (`PTW`) for this translation.
- It can also be configured with its own local Translation Lookaside Buffer (`TLB`) to cache recent address translations, reducing the latency of PTW lookups. The capacity and hierarchy (e.g., L2 TLB) of Gemmini's TLB are configurable.

4.3 Gemmini Integration as a RoCC Accelerator

As discussed in Chapter 3, the `Chipyard` framework supports the integration of custom accelerators via the `RoCC` interface. `Gemmini` is designed as such a `RoCC` accelerator, enabling tight coupling with a `RISC-V` core like `Rocket`.

Hình 4.2: Conceptual diagram of the `Gemmini` RoCC interface with the host CPU (`RISC-V CPU`), PTW, and L2 Cache/System Memory.

The interaction involves:

1. **Custom Instructions:** The host CPU (e.g., `Rocket Core`) issues custom `RISC-V` instructions to `Gemmini`. These instructions are part of the reserved `RoCC` opcode space. The format is typically ‘customX rd, rs1, rs2, funct7‘, where ‘X‘ selects one of up to four `RoCC` units, ‘rd‘ is the destination CPU register (for status or small results), ‘rs1‘ and ‘rs2‘ are source CPU registers (often holding data or pointers to data/configuration structures), and ‘funct7‘ provides further instruction differentiation for `Gemmini`.

2. **Operand Passing:** Operands from the CPU's register file (‘rs1‘, ‘rs2‘) are passed to `Gemmini`. These can include configuration parameters, memory addresses for DMA, or scalar values.

3. **Status Reporting:** `Gemmini` can signal its status (e.g., ‘busy‘) back to the CPU. The CPU can poll this status or use fences for synchronization.

4. **Memory Access:** For bulk data, `Gemmini`'s DMA engine directly accesses the memory system (L2 cache, DRAM) via the `TileLink` bus, using physical addresses obtained after translation by the PTW. This data path bypasses the CPU's core execution pipeline for high-bandwidth transfers.

`Gemmini` employs decoupled access-execute pipelines, often managed through separate instruction queues for load (memory to scratchpad), execute (computation on scratchpad data), and store (scratchpad to memory) operations. This decoupling allows for overlapping computation with data movement, enhancing performance.

4.4 Gemmini Programming Model

`Gemmini` supports a multi-level programming model, catering to different needs from direct hardware control to higher-level abstractions.

Hình 4.3: Layered programming model for `Gemmini`.

4.4.1 Low-Level: Gemmini ISA (RoCC Instructions)

At the lowest level, `Gemmini` is controlled by its custom `RoCC` instruction set. These instructions are typically wrapped in C preprocessor macros or inline assembly for easier use from C/C++ code. Key instruction categories include:

- **Configuration Instructions:** To set up `Gemmini`'s internal state, data types, addresses for DMA, and activation functions.

- **Data Movement Instructions (‘mvin‘, ‘mvout‘):**

  - ‘`mvin`‘ (move in): Transfers data from main memory to `Gemmini`'s scratchpad.

  - ‘`mvout`‘ (move out): Transfers data from `Gemmini`’s scratchpad/accumulator to main memory. It can also apply activation functions and pooling during this process.

- **Execution Instructions (‘preload‘, ‘compute‘):**

  - ‘`preload`‘: Loads stationary data (e.g., weights for WS dataflow, or partial sums for OS dataflow) into the systolic array’s internal registers.
  - ‘`compute`‘: Initiates a matrix multiplication using data already preloaded into the array and data streamed from the scratchpad.

`Gemmini` also offers "complex" instructions like ‘`loop_matmul`‘ or ‘`loop_conv`‘ which encapsulate common looping structures and data tiling, unrolled by hardware Finite State Machines (`FSMs`). These can simplify programming for common layer types by abstracting scratchpad management to some extent, enabling dynamic scheduling and potentially better overlap of load/execute/store operations.

4.4.2 Mid-Level: Hand-Tuned C Library

To abstract the complexities of direct ISA programming, `Gemmini` provides a C library (often through ‘`gemmini.h`‘). This library offers functions for common DNN operations, such as:

- ‘`tiled_matmul`‘: Performs tiled matrix multiplication, handling data movement and computation for larger matrices than can fit in the scratchpad at once.

- ‘`tiled_conv`‘: Performs tiled convolution (often using `im2col` and `tiled_matmul` internally).

- Functions for residual additions, pooling, etc.

These library functions utilize heuristics for loop tiling and data movement to maximize buffer usage and systolic array utilization. They rely on a configuration header file (e.g., ‘`gemmini_params.h`‘) that reflects the specific hardware parameters of the generated `Gemmini` instance (e.g., array dimensions, scratchpad size). This allows the same C code to be used with different `Gemmini` hardware configurations.

4.4.3 High-Level: Compiler Support (Conceptual)

While not the primary focus of this thesis's implementation, the `Gemmini` ecosystem aims to support higher-level compilation flows. This involves tools that can take DNN models described in frameworks like ONNX (Open Neural Network Exchange) or TensorFlow Lite and compile them down to `Gemmini` instructions, potentially leveraging the mid-level C library or directly generating low-level ISA calls. This further simplifies the deployment of DNNs on `Gemmini`-accelerated SoCs.

4.5 Gemmini Configuration in This Thesis

The `Gemmini` accelerator integrated into the SoC for this thesis is based on a specific configuration suitable for the target AI applications. Within the `Chipyard` framework, configurations are typically defined in Scala. A common starting point for a single `Rocket Core` with `Gemmini` is the '`TutorialGemminiRocketConfig`' or a derivative.

For this project, the `Gemmini` instance was configured with [**Specify key parameters here: e.g., systolic array dimensions (e.g., 16x16), scratchpad size (e.g., 256KB), accumulator size (e.g., 64KB), dataflow type (e.g., Weight Stationary), whether the hardware im2col unit was included, etc. These details should come from your actual implementation choices.**]. These parameters were chosen to balance performance for target DNN workloads with the resource constraints typical of an FPGA-based or academic ASIC implementation. The detailed performance implications of these choices will be explored in subsequent chapters.

The integration follows the standard `RoCC` methodology provided by `Chipyard`, ensuring that `Gemmini` coexists with the `Rocket Core`, shares the memory system hierarchy, and can be programmed using custom `RISC-V` instructions dispatched to its `RoCC` slot.

4.6 Summary

`Gemmini` serves as the core computational engine for DNN acceleration in the SoC designed for this thesis. Its flexible, generator-based architecture allows for tailoring to specific needs, and its integration as a `RoCC` accelerator provides a tight coupling with the host `RISC-V` processor. The multi-level programming model offers pathways for both detailed hardware control and higher-level application development.

By leveraging `Gemmini` within the `Chipyard` framework, this thesis aims to demonstrate an effective approach to building and evaluating specialized hardware for AI applications.

TÀI LIỆU THAM KHẢO

References

[1] K. Asanovic et al. *The Rocket Chip Generator*. Tech. Rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, 2016.

[2] C. Celio, D. Patterson, and K. Asanovic. *The Berkeley Out-of-Order Machine (BOOM) Design Specification*. University of California, Berkeley. 2016.

[3] C. Celio, D. A. Patterson, and K. Asanovic. *The Berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor*. Tech. Rep. UCB/EECS-2015-167. EECS Department, University of California, Berkeley, 2015.

[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] Hasan Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *Proceedings of the 58th Annual ACM/IEEE Design Automation Conference*. IEEE Press, 2022, 769–774. ISBN: 9781665432740. URL: `https://doi.org/10.1109/DAC18074.2021.9586216`.

[6] RISC-V International. *Members*. Accessed: Dec. 30, 2024. 2024. URL: `https://riscv.org/members/`.

[7] SiFive. *A Winning Processor Portfolio*. Accessed: Dec. 30, 2024. 2024. URL: `https://www.sifive.com/risc-v-core-ip`.

[8] SiFive Inc. *SiFive TileLink Specification*. Version 1.7-draft. Accessed: Dec. 30, 2024. 2017. URL: `https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf`.

[9] A. 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.

[10] Andrew Waterman et al. *The RISC-V instruction set manual, volume I: User-level ISA, version 2.0*. Tech. Rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, 2014.

[11] J. Zhao. *Chipyard Intro and Fundamentals*. Presentation Slides, ISCA 2021 Tutorial. Accessed: Dec. 30, 2024. 2021. URL: `https://fires.im/isca21-slides-pdf/02_chipyard_basics.pdf`.

[12] J. Zhao et al. “Sonicboom: The 3rd generation Berkeley out-of-order machine”. In: *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2020, pp. 1–7.