

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



## DANH SÁCH CÁC HÌNH





## DANH SÁCH CÁC BẢNG



## 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 [riscv:manual:user:2024, waterman2015riscvprivileged] 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 [riscv2024members]

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

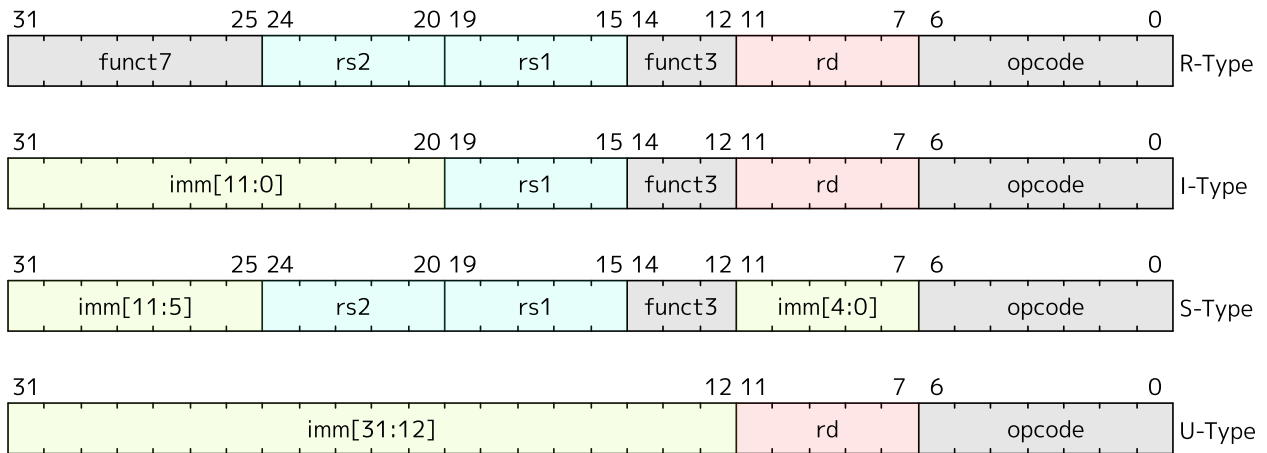
RV32I [riscv:manual:user:2024] is the base instruction set present in almost all 32-bit RISC-V microprocessor architectures. The assembly instructions in this set use 32 32-bit registers, named `x0` - `x31`, to store integers. Each register, in addition to being numbered from 0 to 31, is also referred 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 ?? 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 ?. 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 ? details all instructions in the RV32I base instruction set with their respective formats, specific opcode values, and function codes (`funct3` and `funct7`).

Bảng 2.1: ABI names and functions of registers in RV32I. Source [riscv:manual:user:2024]

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 [riscv:manual:user:2024]

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 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.

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 ??.

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 [riscv:manual:user:2024] 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 ?? 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.

### 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

RV32I Base Instruction Set								
imm[31:12]				rd		0110111	LUI	
imm[31:12]				rd		0010111	AUIPC	
imm[20 10:1 11 19:12]				rd		1101111	JAL	
imm[11:0]			rs1	000	rd		1100111	JALR
imm[12 10:5]		rs2	rs1	000	imm[4:1 11]		1100011	BEQ
imm[12 10:5]		rs2	rs1	001	imm[4:1 11]		1100011	BNE
imm[12 10:5]		rs2	rs1	100	imm[4:1 11]		1100011	BLT
imm[12 10:5]		rs2	rs1	101	imm[4:1 11]		1100011	BGE
imm[12 10:5]		rs2	rs1	110	imm[4:1 11]		1100011	BLTU
imm[12 10:5]		rs2	rs1	111	imm[4:1 11]		1100011	BGEU
imm[11:0]			rs1	000	rd		0000011	LB
imm[11:0]			rs1	001	rd		0000011	LH
imm[11:0]			rs1	010	rd		0000011	LW
imm[11:0]			rs1	100	rd		0000011	LBU
imm[11:0]			rs1	101	rd		0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]		0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]		0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]		0100011	SW
imm[11:0]			rs1	000	rd		0010011	ADDI
imm[11:0]			rs1	010	rd		0010011	SLTI
imm[11:0]			rs1	011	rd		0010011	SLTIU
imm[11:0]			rs1	100	rd		0010011	XORI
imm[11:0]			rs1	110	rd		0010011	ORI
imm[11:0]			rs1	111	rd		0010011	ANDI
0000000		shamt	rs1	001	rd		0010011	SLLI
0000000		shamt	rs1	101	rd		0010011	SRLI
0100000		shamt	rs1	101	rd		0010011	SRAI
0000000		rs2	rs1	000	rd		0110011	ADD
0100000		rs2	rs1	000	rd		0110011	SUB
0000000		rs2	rs1	001	rd		0110011	SLL
0000000		rs2	rs1	010	rd		0110011	SLT
0000000		rs2	rs1	011	rd		0110011	SLTU
0000000		rs2	rs1	100	rd		0110011	XOR
0000000		rs2	rs1	101	rd		0110011	SRL
0100000		rs2	rs1	101	rd		0110011	SRA
0000000		rs2	rs1	110	rd		0110011	OR
0000000		rs2	rs1	111	rd		0110011	AND
fm	pred	succ	rs1	000	rd		0001111	FENCE
1000	0011	0011	00000	000	00000	0001111	FENCE.TSO	
0000	0001	0000	00000	000	00000	0001111	PAUSE	
000000000000			00000	000	00000	1110011	ECALL	
000000000001			00000	000	00000	1110011	EBREAK	

Hình 2.3: Instructions in the RV32I base instruction set. Source [riscv:manual:user:2024]

Bảng 2.2: Pseudo-instructions and their corresponding standard instructions. Source [riscv:manual:user:2024]

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 zero
bgez rs, offset	bge rs, x0, offset	Branch if greater than or equal to zero
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

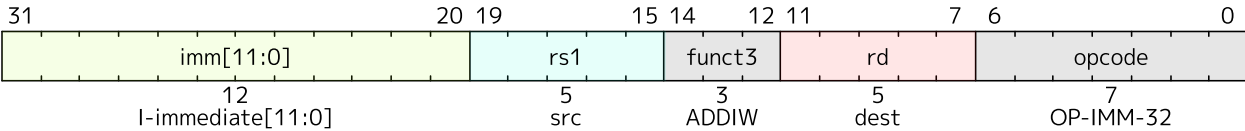
Bảng 2.3: Development status of extended RISC-V instruction sets. Source [riscv:manual:user:2024]

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

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

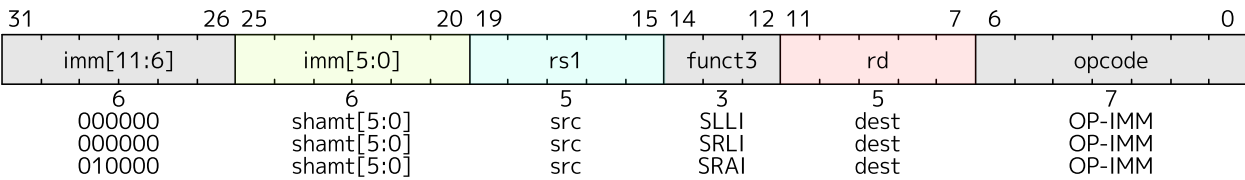
ADDW instruction handles 32-bit addition, ignoring the upper 32 bits and returning a 32-bit signed value, sign-extended to 64 bits. These instructions enabling backward compatibility typically have a "W" suffix to denote 32-bit wide operations.

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.



Hình 2.4: Instruction encoding format for ADDIW in 64-bit architecture. [riscv:manual:user:2024]

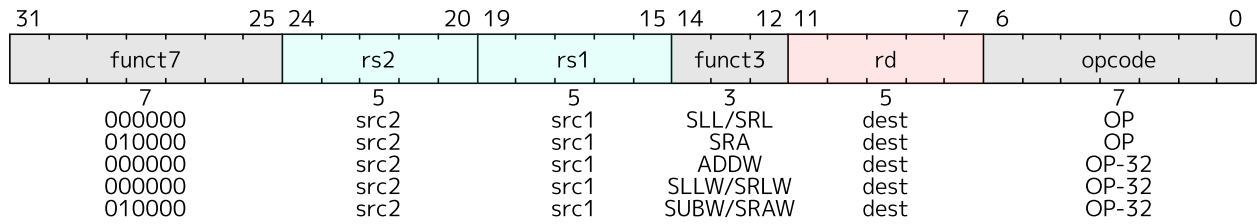
Some distinctly different instructions found only in RV64I include the ADDIW instruction, as shown in Figure ???. ADDIW performs an addition between a 12-bit sign-extended immediate value and register `rs1`, producing a proper 32-bit sign-extended result in `rd`. Overflows are ignored, and the result 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`).



Hình 2.5: Encoding format for 64-bit I-type shift instructions. [riscv:manual:user:2024]

I-type shift operations in RV64I (Figure ??) 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:

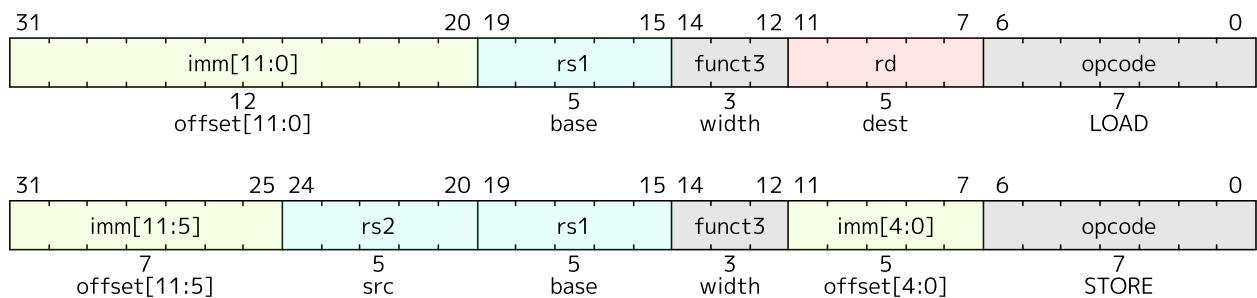
SLLI (logical left shift, zeros shifted into lower bits), SRLI (logical right shift, zeros shifted into upper bits), and SRAI (arithmetic right shift, sign bit copied into vacated upper bits).



Hình 2.6: Encoding format for 64-bit R-type shift instructions for word operations. [riscv:manual:user:2024]

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 ??), 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.



Hình 2.7: Encoding formats for 64-bit LOAD and STORE instructions. [riscv:manual:user:2024]

The instruction formats are shown in Figure ?. 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.

## 2.2 RISC-V Processor Cores

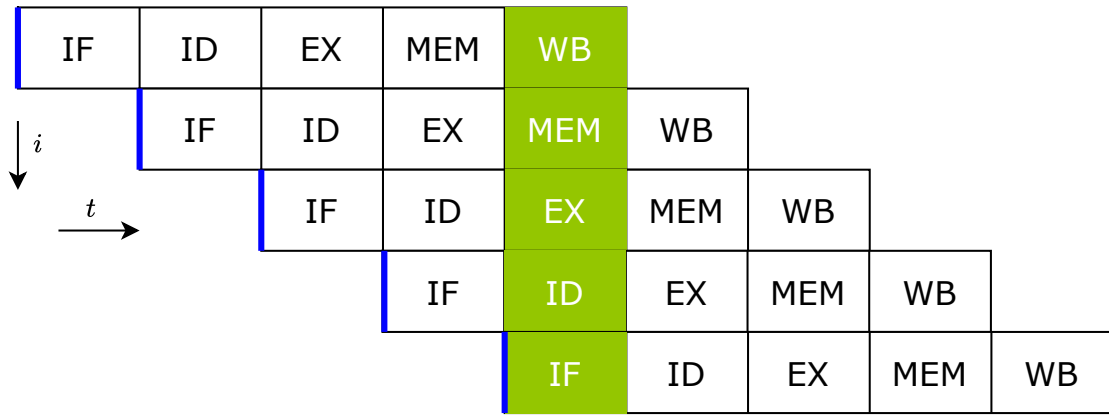
With its open-source nature, system developers can freely implement designs based on the RISC-V architecture, supported by an increasingly robust and growing ecosystem. The RISC-V ISA ecosystem mentioned here includes a diverse range of components from hardware design tools, low-level firmware development, 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.

### 2.2.1 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.

The concept of pipelining is illustrated in Figure ???. 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



Hình 2.8: Basic five-stage pipeline in a RISC machine. The vertical axis is successive instructions; the horizontal axis is time. So in the green column, the earliest instruction is in WB stage, and the latest instruction is undergoing instruction fetch.

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.

### 2.2.2 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

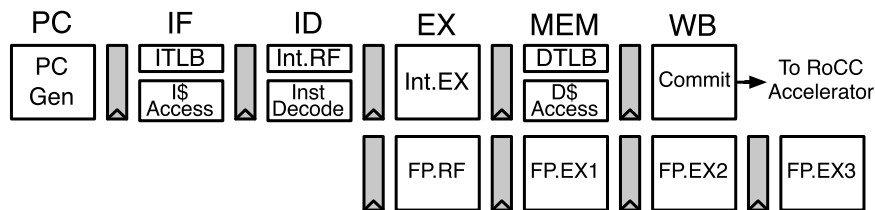
the program order as in a pipeline. This architecture is also a core element in high-performance systems used in scientific computing and artificial intelligence research. This flexible execution model addresses performance bottlenecks caused by data dependencies, cache latencies, and pipeline stalls, 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.

### 2.2.3 Open-Source RISC-V Processor Cores

Rocket [asanovic2016rocket], 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 [asanovic2016rocket]

The design of this microprocessor integrates many advanced features, including a



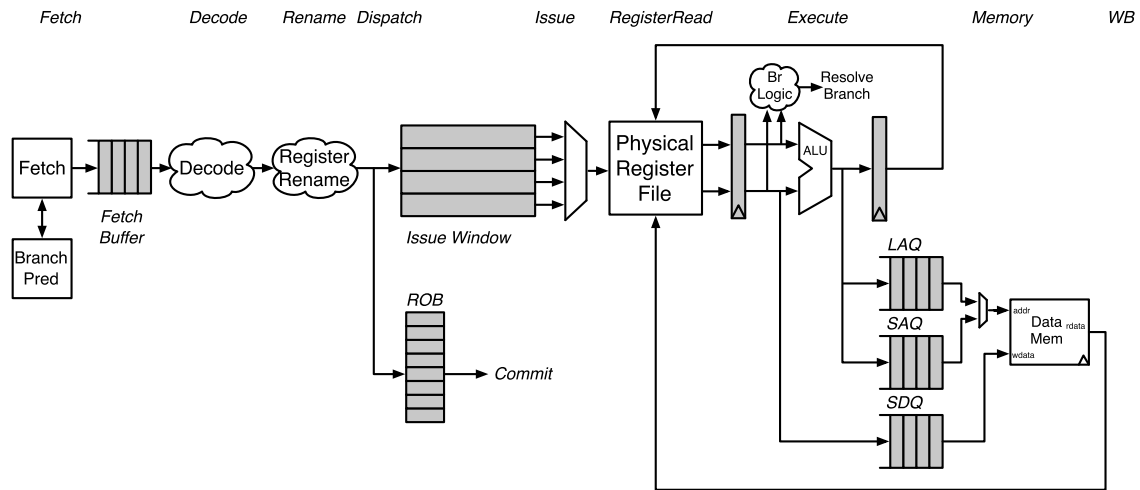
Memory Management Unit (MMU) that supports virtual memory, enabling efficient memory access and address translation. Besides its role as a general-purpose microprocessor, Rocket also serves as a library of reusable components, including functional units, caches, Translation Lookaside Buffer (TLB), Page 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) [[celio2015boom](#), [celio2016boom\\_spec](#), [celio\\_phdthesis\\_2018boom](#)] 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 [[zhao2020sonicboom](#)] (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.

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.



Hình 2.10: Berkeley Out-of-Order Machine Processor Pipeline Architecture. Source [celio\_phdthesis\_2018boom]

- **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.

### 2.3 System-on-Chip Design Framework: Chipyard - Rocket Chip

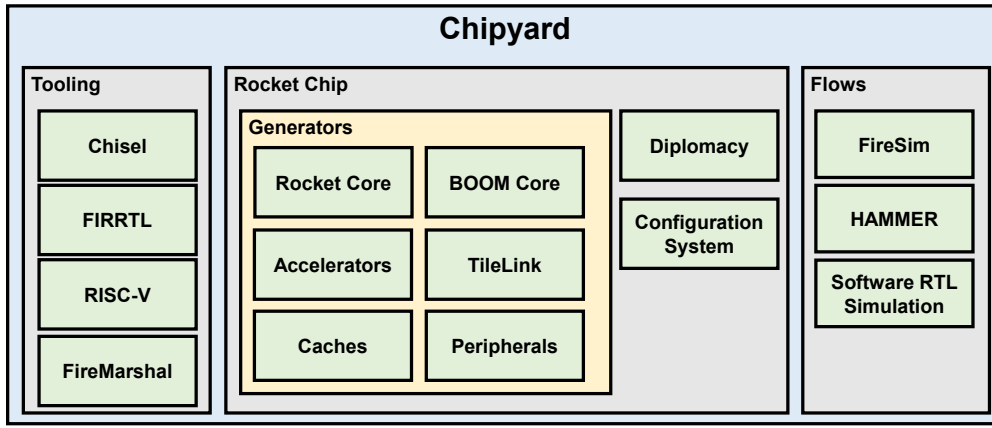
The Rocket Chip project, also developed by the Berkeley Architecture Research (BAR) group at the University of California, Berkeley, marked a significant advancement in open-source processor design. This project provides a flexible, scalable, and configurable platform, enabling the creation of custom 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) [asanovic2016rocket].



Hình 2.11: Chipyard, developed by UC Berkeley. Source: [zhao2021chipyard]

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

By utilizing Chisel, the FIRRTL compiler, the Rocket Chip generator, and other tools, Chipyard facilitates the development of highly customizable RISC-V SoCs.



Hình 2.12: Component structure of Chipyard. Source: [zhao2021chipyard]

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 [sifive2024portfolio], 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 Single-Core with Gemmini RoCC Accelerator 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



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 pre-processing 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.

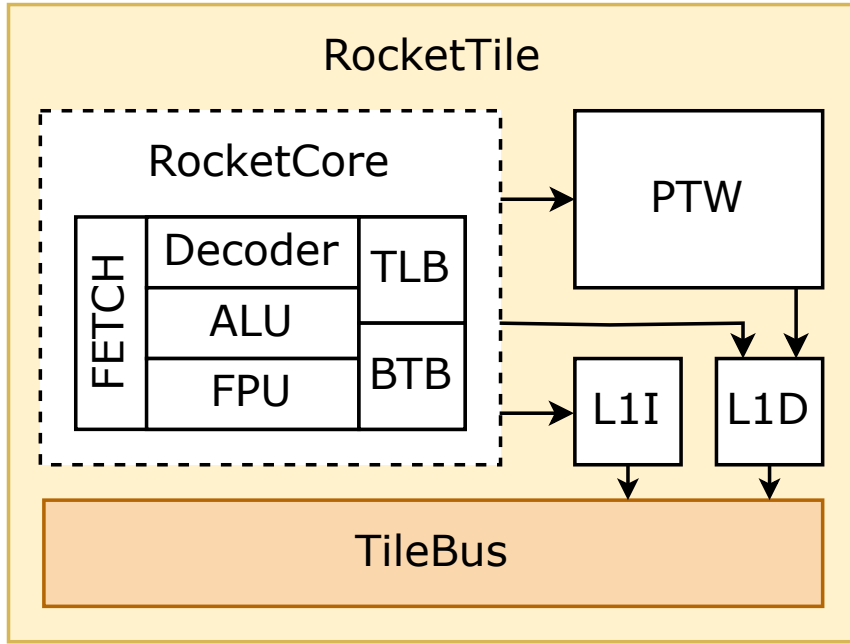
**Verilator** offers high simulation performance, especially for designs requiring substantial resources, due to its high speed and cycle accuracy. **Verilator**'s compilation approach makes it up to 100 times faster than interpretive simulators like **Icarus Verilog** (which is also open-source), and 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 ??) 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 ac-



Hình 3.2: Structure of a Rocket Tile.

cording to implementation needs, such as `WithNBigCore()`, `WithNMedCore()`, `WithNSmallCore()`, and `WithNTinyCore()`. Additionally, instead of implementing the Rocket core, the BOOM core can also be integrated in its place, and many other peripherals can be integrated into a Tile unit.

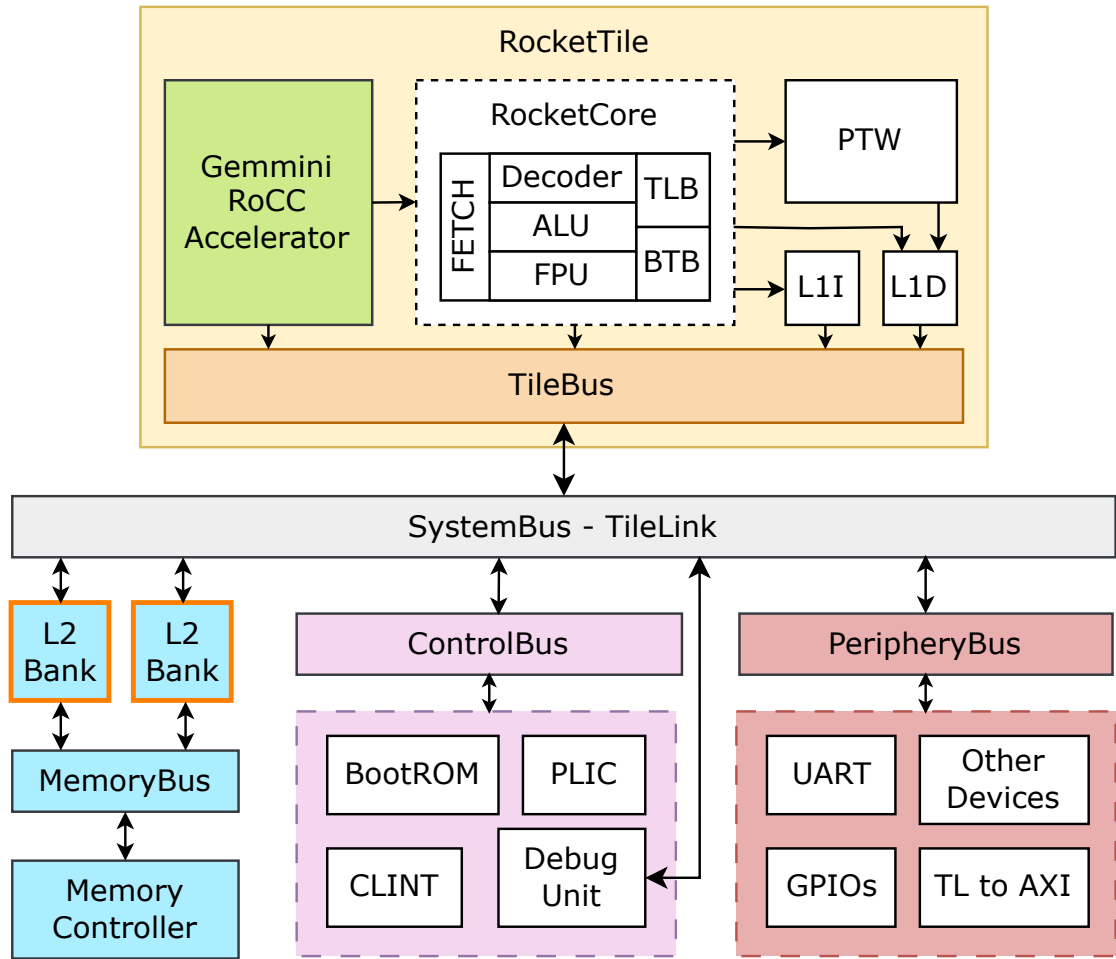
### 3.3 Rocket Chip SoC Structure

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 ?? 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



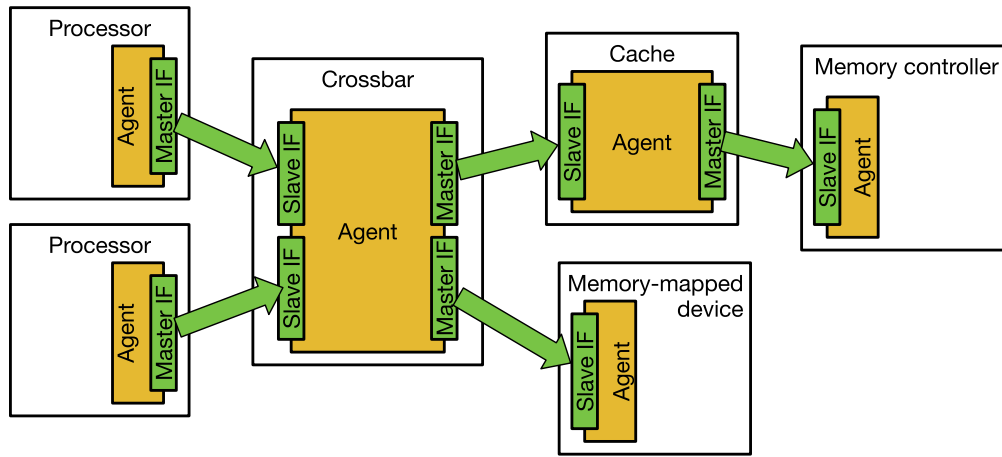


Hình 3.3: Proposed structure of a single-core RISC-V processor system with a Gemmini RoCC accelerator.

crossbar-type bus architectures instead of Time Shared Buses. A crossbar, specifically a fully-connected crossbar structure, is considered a form of hardware switching 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** [sifive2018tilelink] 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



Hình 3.4: Connection model in TileLink and its interfaces. Source: [sifive2018tilelink]

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 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 ??.

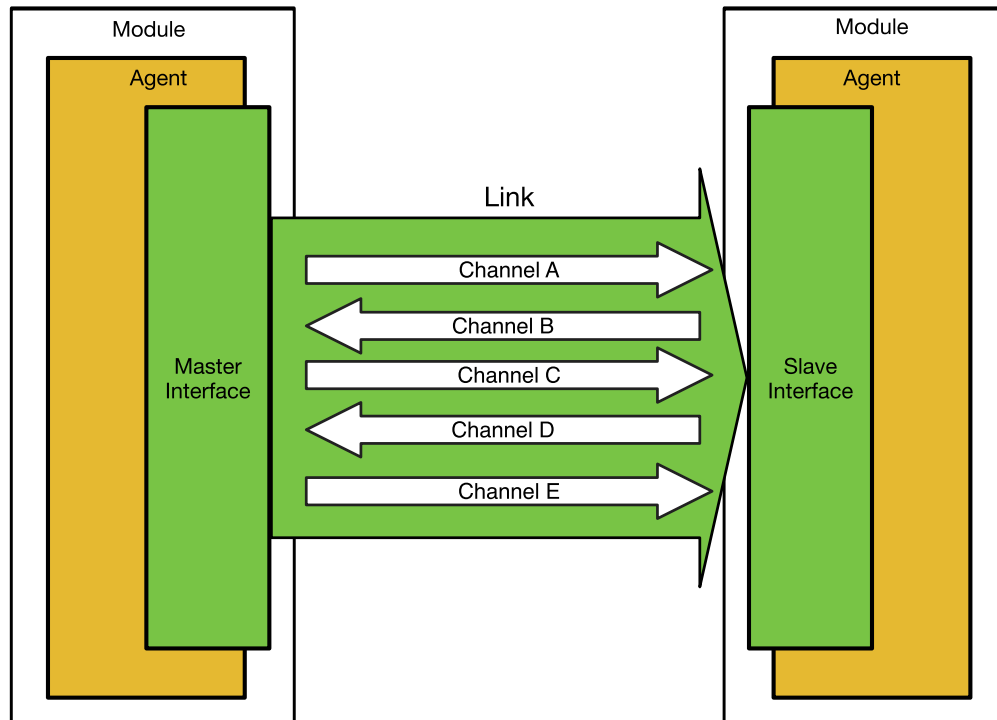
Bảng 3.1: TileLink operational configurations. Source: [sifive2018tilelink]

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



Hình 3.5: Channels in a **TileLink** connection between two agents. Source: [sifive2018tilelink]

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.

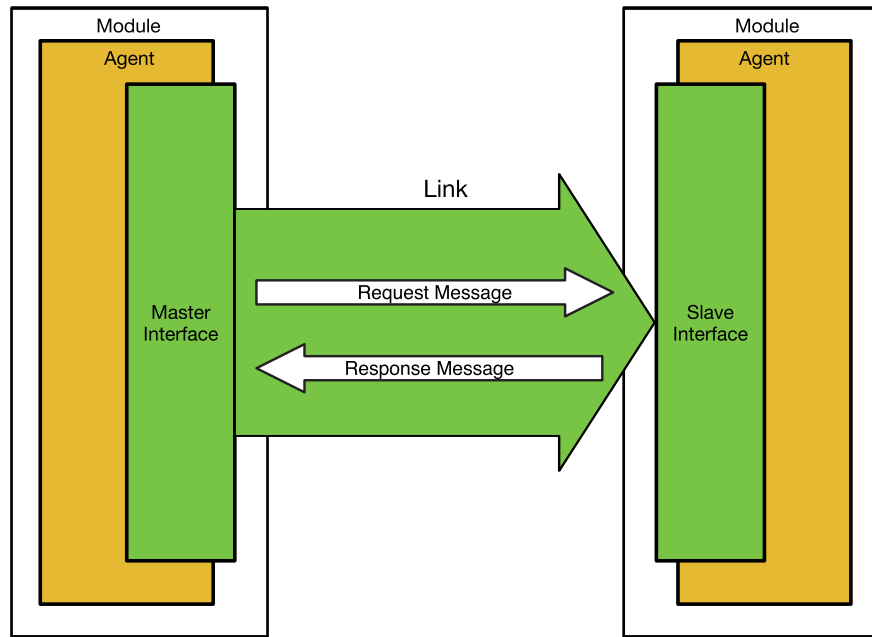
The priority order for these channels is as follows:  $A < B < C < D < E$ , where channel A has the lowest priority and E has the highest. Prioritization in the **TileLink** bus system ensures that messages transmitted through the system do not fall into a hold-and-wait loop. In other words, the message 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 ??), 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 ?. 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,



Hình 3.6: TileLink TL-UL connection structure. Source: [sifive2018tilelink]

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 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.

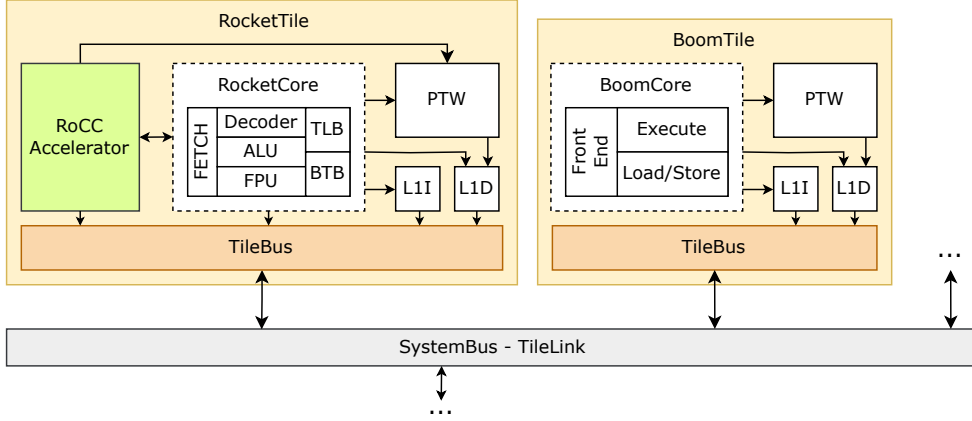
At a higher level, the system also supports the integration of external memory, such as DRAM or SRAM, through open-source controllers or specialized IPs from FPGA implementation toolkits. Additionally, an interface with **DRAMSim** is supported for integration when performing full-system simulation. 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 ?? below.

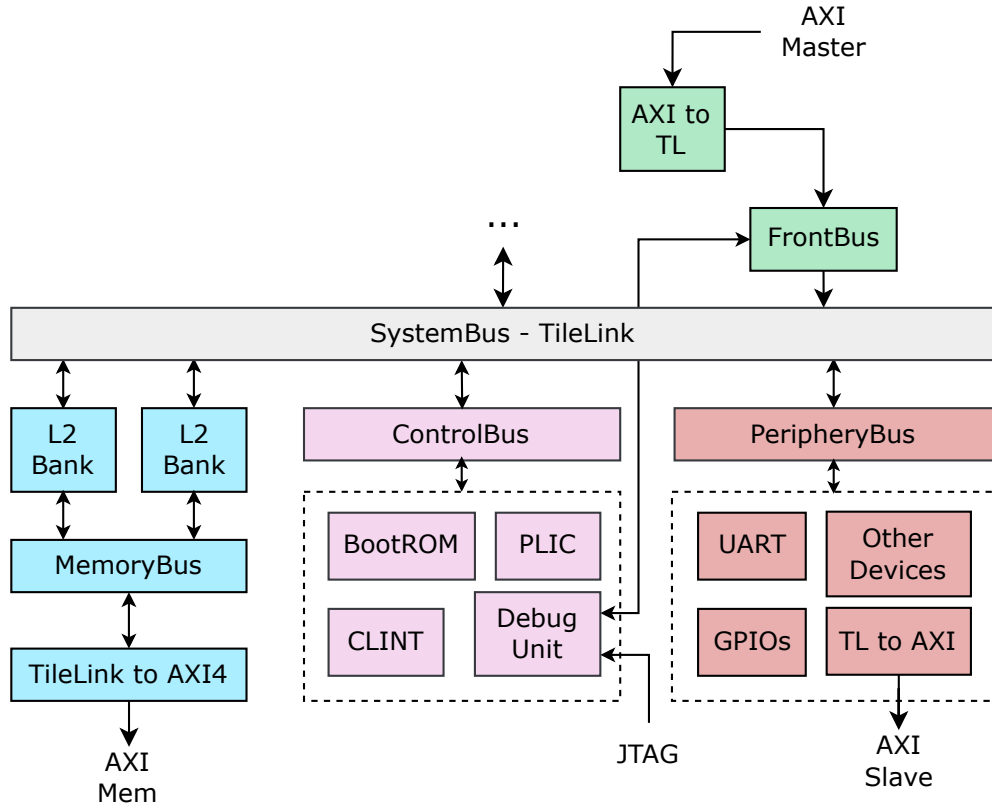


Hình 3.7: Implementation structure of a heterogeneous multi-core processor system.

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.

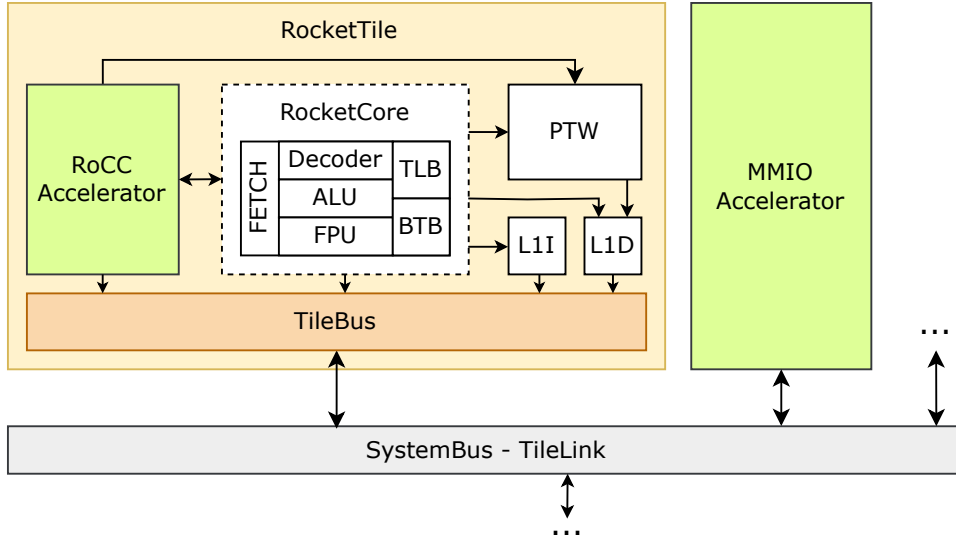


Hình 3.8: Peripherals in the Rocket Chip base system.

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. 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 Co-processor), 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: Architectural Foundations of DNN Acceleration

The efficient execution of Deep Neural Networks (DNNs) presents one of the most significant challenges and opportunities in modern computer architecture. While today’s general-purpose CPUs are highly optimized for a wide range of tasks, their fundamental design principles are often mismatched with the unique computational and data movement patterns of large-scale neural networks. This chapter will establish the foundational concepts that motivate the need for specialized hardware. We will begin by exploring the core problem—the immense cost of data movement—and then introduce the classic architectural paradigm designed to solve it: the systolic array. Finally, we will examine how this timeless principle is embodied in state-of-the-art industrial accelerators, providing the necessary context to understand the design of the Gemmini accelerator, which is the central subject of this thesis.

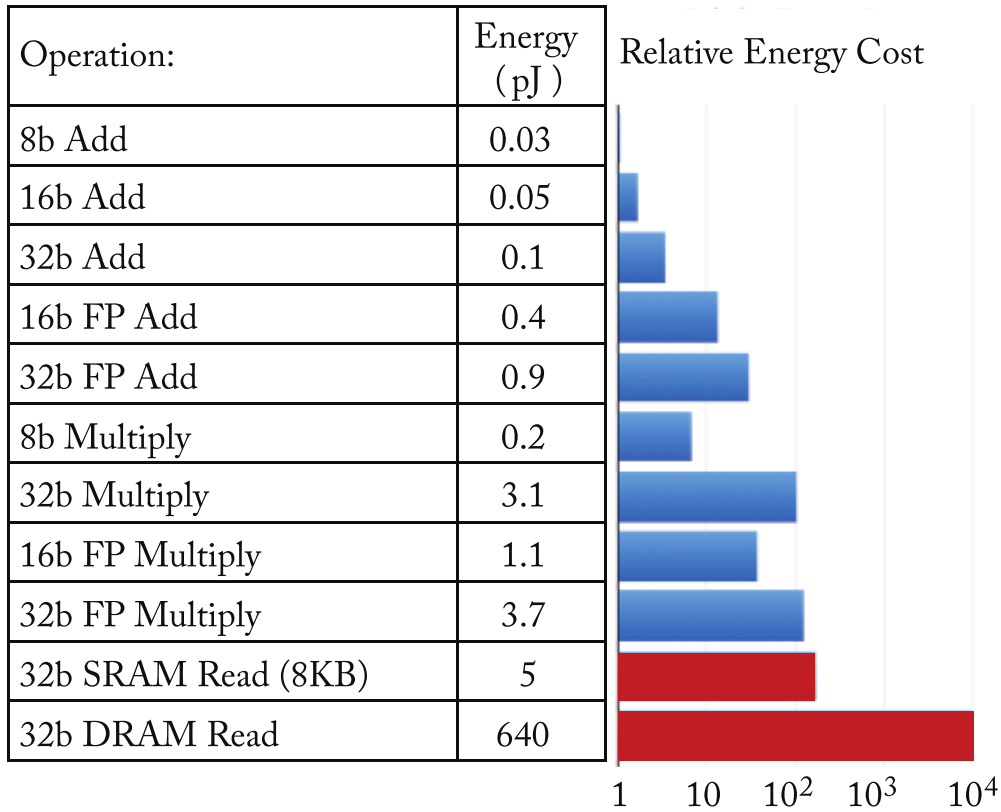
### 4.1 The Data-Compute Divide: The Memory Wall

The performance of any computing system is ultimately limited by two factors: the speed of its computations and the speed at which it can supply data to its computational units. For decades, Moore’s Law drove exponential growth in the number of transistors on a chip, leading to dramatic increases in processing power. However, the speed of off-chip memory systems, such as DRAM, has improved at a much slower pace. This growing disparity between processor speed and memory speed is famously known as the **Memory Wall** [wulf1995hitting].

For data-intensive workloads like DNNs, this is the primary performance bottleneck. The problem can be understood with a simple analogy: imagine a master chef (the processor) who can perform culinary tasks with superhuman speed. However, their ingredients (the data) are stored in a large warehouse across the street (DRAM). Even though the chef is exceptionally fast, their overall productivity is dominated by the time spent walking to and from the warehouse to fetch each ingredient.

This bottleneck is not just about latency; it is fundamentally about energy. As quantified by Horowitz [horowitz2014energy], the energy required to perform a complex 32-bit floating-point computation is dwarfed by the energy needed to fetch its operands from off-chip DRAM (Figure ??). This immense energy cost of data movement means that for DNNs, which perform billions of operations, any viable high-performance hardware solution must be designed with a primary objective: to

minimize data movement.



Hình 4.1: The relative energy cost of various arithmetic and memory operations. A 32-bit DRAM read is orders of magnitude more costly than a 32-bit floating-point addition. This disparity is the primary motivation for specialized accelerator architectures. Adapted from Horowitz, as presented in Sze et al. [sze2020efficient, horowitz2014energy].

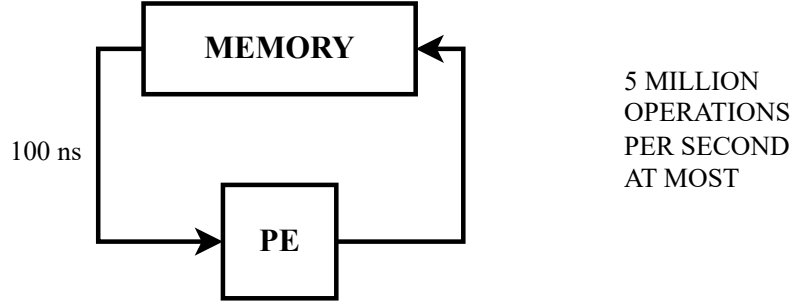
## 4.2 A Classic Solution: The Systolic Principle

In his seminal 1982 paper, H.T. Kung proposed a novel architectural paradigm designed specifically to address the memory wall: the **systolic array** [kung1982systolic]. The architecture is named by analogy to the human circulatory system. Just as the heart rhythmically pumps blood to the body's cells, in a systolic system, memory "pumps" data through a regular grid of simple Processing Elements (PEs).

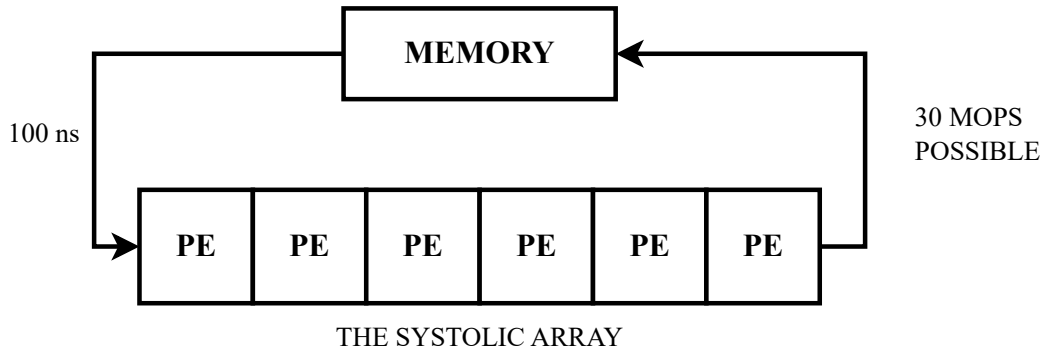
The core principle is to **maximize the number of computations performed for each data item fetched from memory**. Instead of fetching an operand, using it once in a central ALU, and then discarding it (the conventional approach), a systolic array passes an operand from one PE to the next in a pipelined fashion. At each step of its journey, the data element participates in another computation. This structure,

illustrated conceptually in Figure ??, amortizes the high cost of the initial memory access over dozens or even hundreds of useful operations.

**INSTEAD OF:**



**WE HAVE:**



Hình 4.2: A conceptual comparison of a conventional von Neumann architecture and a systolic architecture. The conventional approach (top) is bottlenecked by the memory bus. The systolic approach (bottom) uses an array of PEs to achieve high throughput with the same memory bandwidth by reusing data internally. Adapted from the presentation of Kung’s work [kung1982systolic].

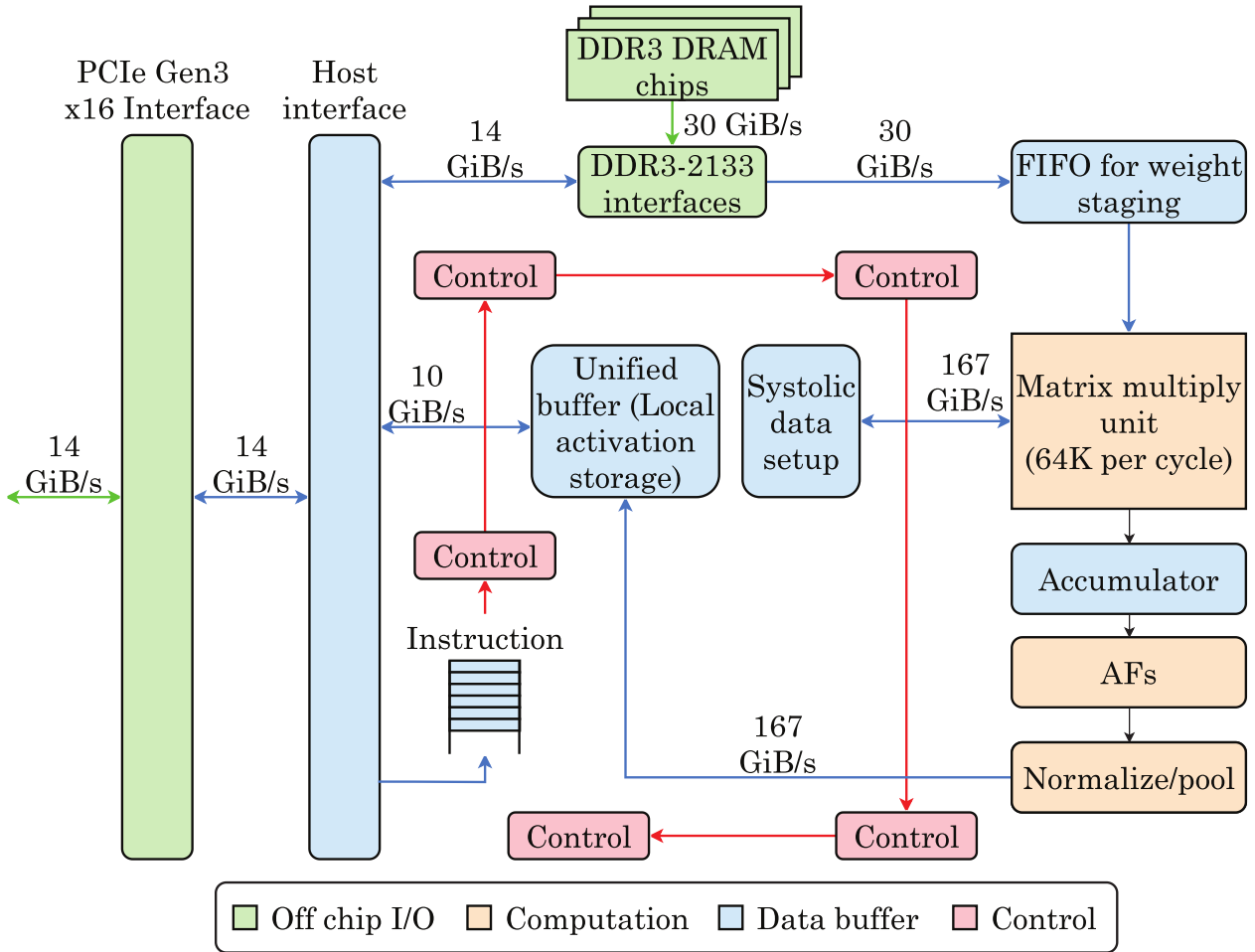
### 4.3 Case Studies in Modern Systolic Design

While the systolic principle is four decades old, it has become the dominant paradigm for modern high-performance DNN accelerators. We will examine two landmark industrial designs that showcase this principle in practice.

#### 4.3.1 Google’s TPuv1: A Datacenter-Scale Design

Google’s Tensor Processing Unit (TPU) was one of the first large-scale custom ASICs designed specifically for accelerating DNN inference in datacenters [jouppi2017tpu]. As shown in Figure ??, its architecture is a direct embodiment of the systolic prin-

ciple. The heart of the TPU is a massive 256x256 Matrix Multiply Unit, which is a two-dimensional systolic array of 65,536 8-bit multiply-accumulate (MAC) units. To feed this vast computational engine, the TPU relies on a large, software-controlled on-chip memory called the Unified Buffer (a 24 MiB scratchpad), which stores input activations and intermediate partial sums. The design philosophy of the TPU prioritizes deterministic, high-throughput performance for inference tasks where low latency is critical.

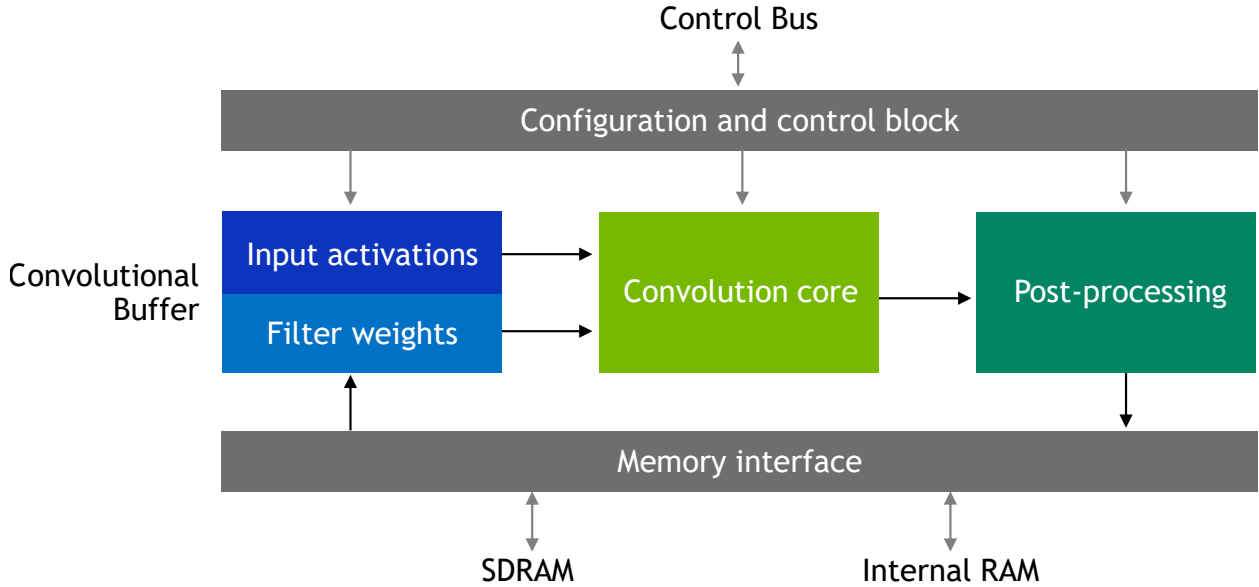


Hình 4.3: A high-level block diagram of the Google TPUv1. The design is dominated by the systolic Matrix Multiply Unit and its large on-chip Unified Buffer. Adapted from Jouppi et al. [jouppi2017tpu, mittal2021survey].

#### 4.3.2 NVIDIA NVDLA: An Open-Source Accelerator IP

In contrast to the monolithic, datacenter-focused TPU, the NVIDIA Deep Learning Accelerator (NVDLA) is an open-source, configurable IP core designed for integration into a wide range of Systems-on-Chip (SoCs), particularly for embedded

systems [farshchi2019nvdla]. As shown in Figure ??, the NVDLA has a more modular design, with a dedicated Convolutional Core (which contains the MAC units), a Post-Processing unit for activation functions and pooling, and a Convolutional Buffer for storing weights and inputs. Unlike the TPU’s custom interface, the NVDLA uses industry-standard bus protocols (AXI and APB) for memory access and control, simplifying its integration into third-party SoC designs. Its configurability (e.g., ‘nv\_small’, ‘nv\_large’) allows a system designer to trade off area and performance, making it a flexible solution for the embedded space.



Hình 4.4: The high-level architecture of the NVDLA. The design is partitioned into distinct functional units connected via standard bus interfaces. Adapted from Farshchi et al. [farshchi2019nvdla].

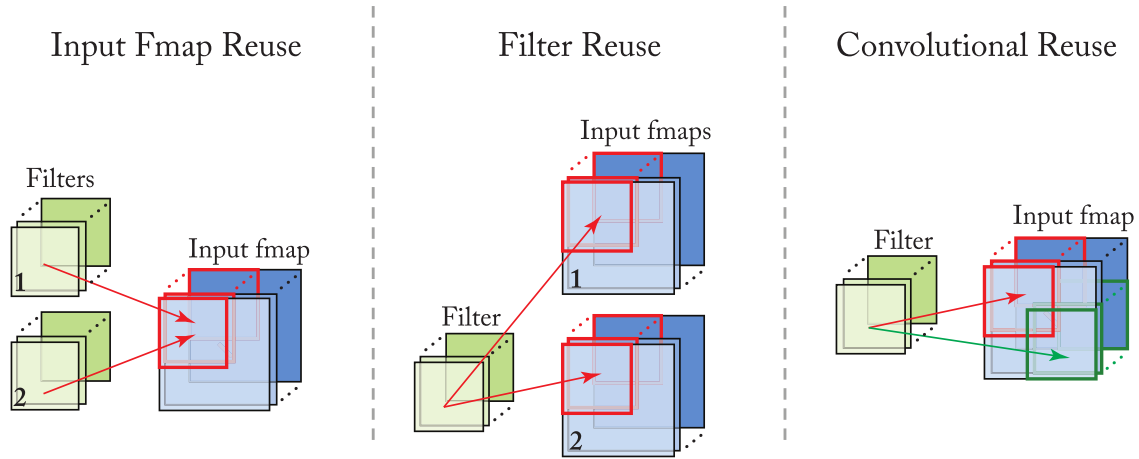
#### 4.4 A Formal Framework: Dataflows

The design choices made in architectures like the TPU and NVDLA can be formally described using the concept of a **dataflow**. The dataflow defines the strategy for moving data through the systolic array to maximize reuse. The effectiveness of a given dataflow depends on the structure of the DNN layer being executed. The two most common dataflows, which we will see are directly relevant to Gemmini, are:

- **Weight Stationary (WS):** This dataflow prioritizes weight reuse. Filter weights are pre-loaded into the PEs and remain stationary, while input activations are streamed in. This strategy, used by the Google TPU, is highly efficient for layers with large input feature maps and relatively small filters.

- **Output Stationary (OS):** This dataflow prioritizes minimizing the movement of partial sums. Each PE is responsible for accumulating the final value for a single output activation, which remains stationary in its accumulator. Inputs and weights are streamed to the PE as needed. This can be more efficient for layers where partial sum read/write energy is a dominant cost.

These strategies, visually contrasted in Figure ?? and ??, represent a fundamental trade-off in accelerator design. The choice of dataflow directly impacts which types of data reuse, shown in Figure ??, are most effectively exploited.

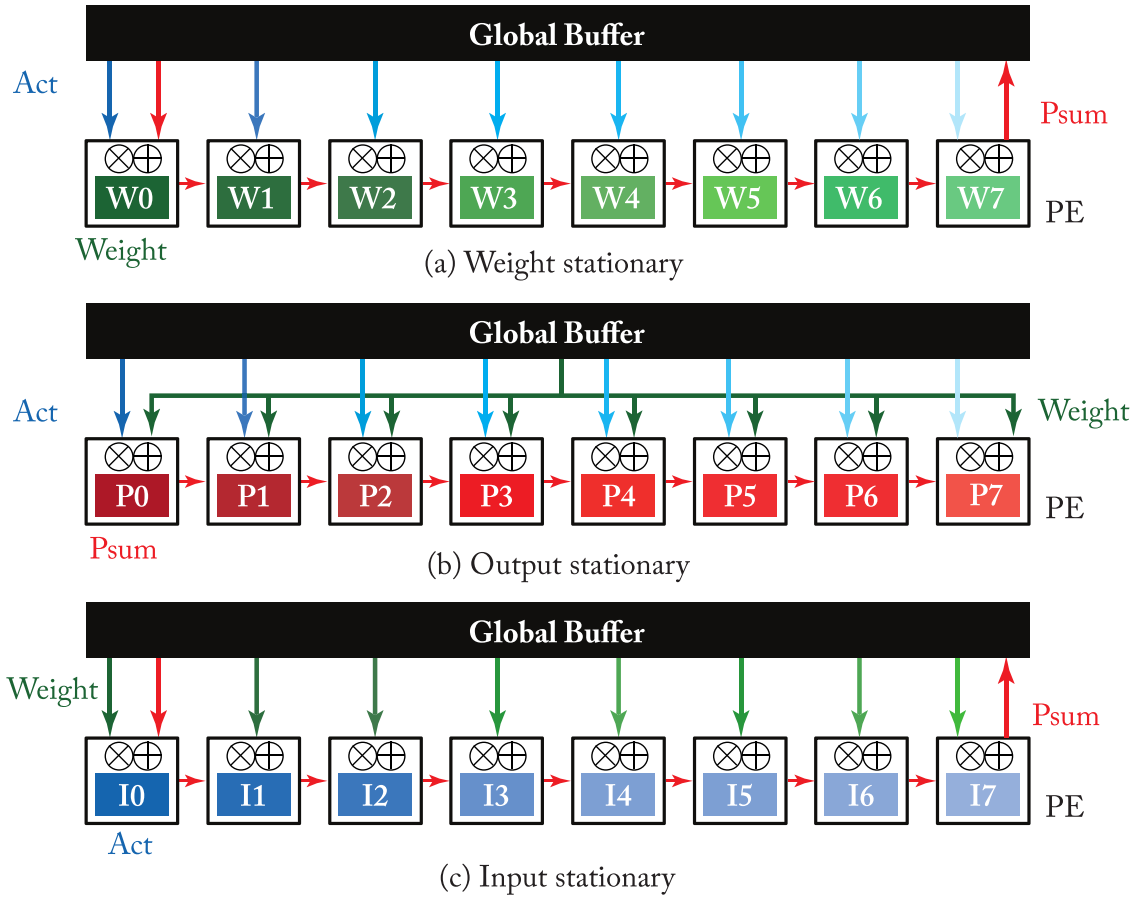


Hình 4.5: The three primary forms of data reuse in convolutional layers: convolutional, filter, and input reuse. Adapted from Sze et al. [sze2020efficient].

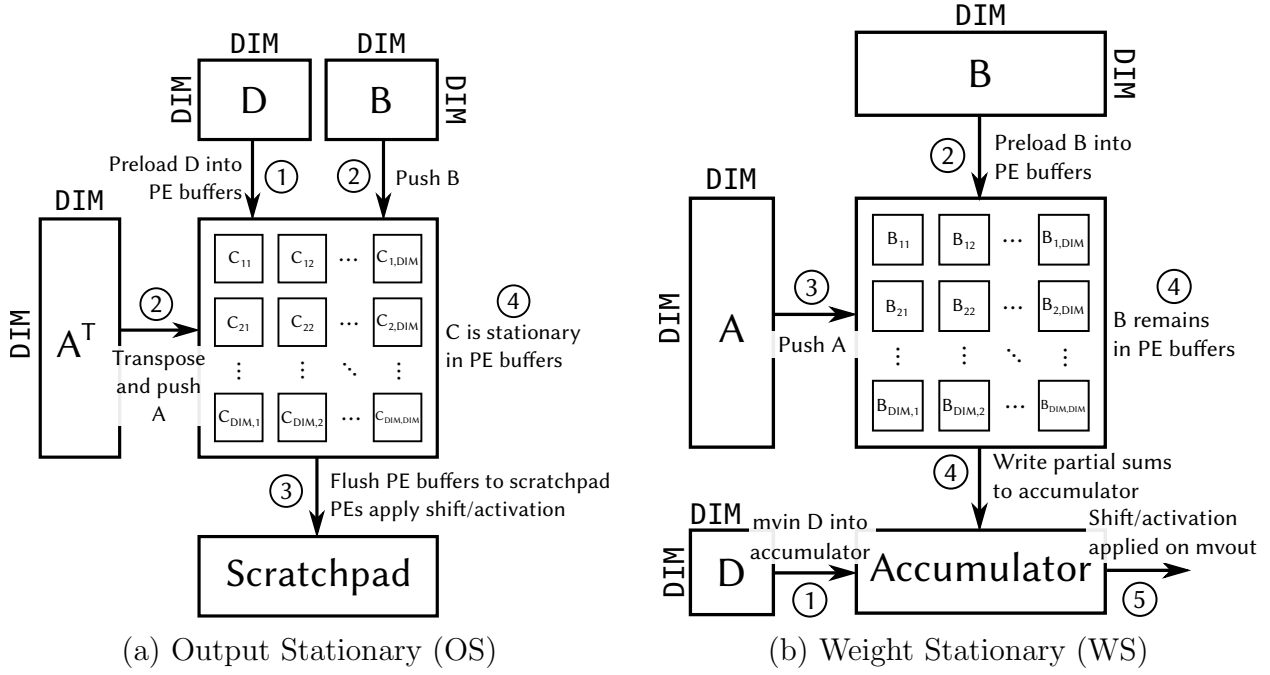
## 4.5 Chapter Summary

This chapter has built a foundational argument for the necessity of specialized hardware for DNN acceleration. We began with the core architectural challenge—the Memory Wall—and introduced the systolic array as a classic and enduring solution that minimizes costly data movement by maximizing data reuse. By examining modern industrial accelerators like the Google TPU and NVDLA, we have seen this principle in action. Finally, we formalized these strategies with the concept of dataflows, such as Weight Stationary and Output Stationary. These foundational concepts provide the necessary framework to understand the design and operation of the Gemmini accelerator, which will be the focus of the subsequent chapters.





Hình 4.6: A visual comparison of Weight Stationary (left) and Output Stationary (right) dataflows. Adapted from Sze et al. [sze2020efficient].



Hình 4.7: A visual comparison of the two primary dataflows supported by Gemmini. (a) In the **Output Stationary** dataflow, the result matrix  $C$  remains stationary in the PEs to accumulate partial sums. (b) In the **Weight Stationary** dataflow, the weight matrix  $B$  is preloaded and remains stationary in the PEs to maximize weight reuse. The choice between these strategies represents a fundamental trade-off in accelerator design. Source: [gemini-dac].

## CHƯƠNG 5: The Gemmini DNN Accelerator: An Architectural Overview

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 architecture of this accelerator: **Gemmini**, a sophisticated component designed to overcome the fundamental limitations of general-purpose processors for Deep Neural Network (DNN) tasks.



Hình 5.1: Gemmini Logo. Source: [gemini-dac].

### 5.1 Introduction: The Full-Stack Accelerator Philosophy

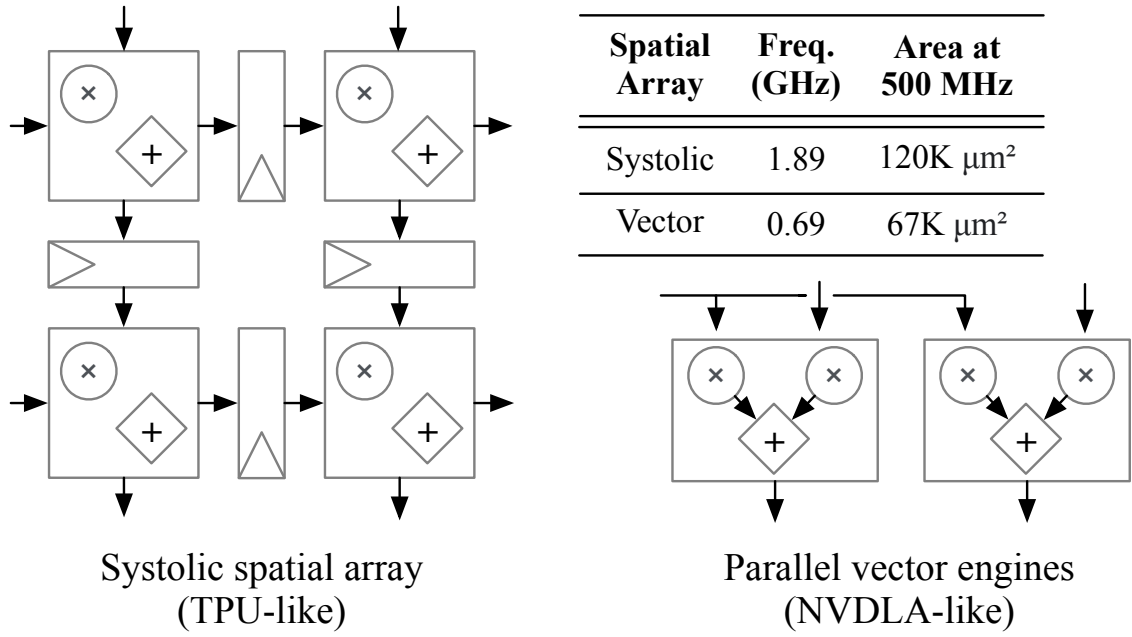
**Gemmini** is an open-source, full-stack DNN accelerator *generator* developed at the University of California, Berkeley [gemini-dac]. It is designed not merely as a standalone hardware component, but as a complete ecosystem for exploring and evaluating domain-specific hardware acceleration.

Its design is rooted in the philosophy of "full-stack" evaluation. Many accelerators are designed and tested in isolation, which fails to account for critical system-level effects such as memory bus contention, cache coherency overhead, and operating system interactions (e.g., page faults and context switches) [gemini-dac]. By generating not only the accelerator RTL but also a complete, Linux-capable SoC using the **Chipyard** framework, **Gemmini** enables a comprehensive analysis of these cross-stack interactions. This allows for a more realistic assessment of real-world performance and energy efficiency [gemini-dac].

Crucially, as a generator written in Chisel, **Gemmini** is not a single, fixed hardware design. As illustrated in Figure ??, its flexible template can be configured to produce a wide spectrum of systolic architectures, from highly pipelined designs to parallel-vector engines. This makes it an exceptionally powerful platform for the academic exploration of accelerator architectures.

For this thesis, leveraging **Gemmini** provides the opportunity to explore a powerful

DNN accelerator within a realistic SoC environment.



Hình 5.2: The design space enabled by the Gemini generator, allowing it to produce a wide variety of spatial array structures. Source: [gemini-dac].

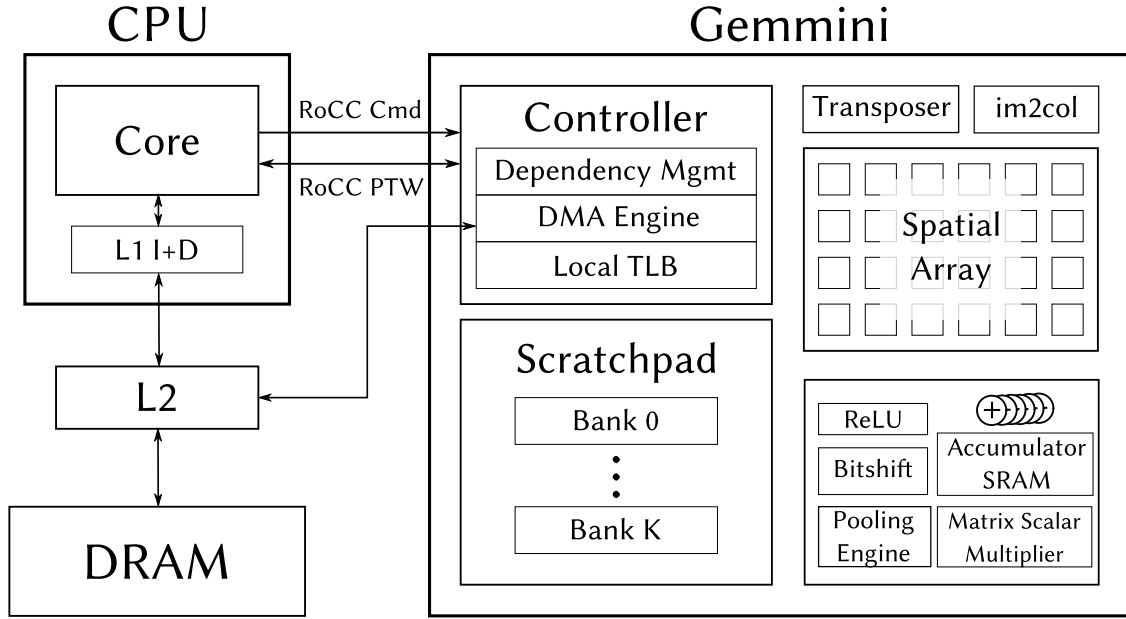
## 5.2 Top-Level Architectural Blueprint

The **Gemmini** generator can produce a wide range of accelerator instances based on a flexible architectural template. To understand the **Gemmini** architecture, we first examine the high-level template shown in Figure ???. This blueprint illustrates the three primary components of the accelerator—the Controller, the on-chip memory (Scratchpad and Accumulator), and the compute core (Spatial Array)—and their interface with the host CPU and system DRAM. The following sections provide a detailed analysis of each of these blocks.

## 5.3 Analysis of Core Architectural Components

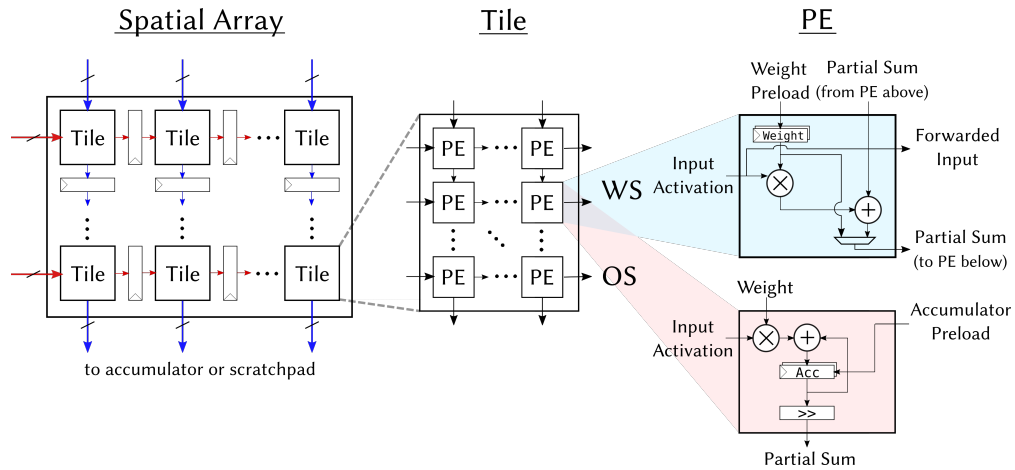
### 5.3.1 The Systolic Compute Engine

The computational heart of **Gemmini** is its systolic array, a highly configurable spatial array that functions as a powerful General Matrix-Matrix Multiplication (GEMM) engine. The array's microarchitecture, as detailed in Figure ???, features a two-level hierarchy composed of **Tiles** and individual Processing Elements (PEs). Each PE



Hình 5.3: High-level overview of the Gemini accelerator template integrated with a host CPU and system memory hierarchy. Source: [gemini-dac].

is capable of performing a Multiply-Accumulate (MAC) operation in a single clock cycle. This hierarchical and configurable design allows **Gemmini** to efficiently execute the dense matrix multiplications fundamental to modern Deep Neural Networks (DNNs).



Hình 5.4: The two-level hierarchy of Gemini's spatial array, composed of tiles and Processing Elements (PEs). Source: [gemini-dac].

The key parameters of the systolic array can be specified at generation time:

- **Dimensions:** The size of the PE grid, structured into a hierarchy of tiles, can be configured (e.g., a 16x16 array composed of 2x2 tiles, each with 8x8

PEs). This allows for a trade-off between hardware area and parallel processing capability.

- **Dataflow:** *Gemmini* primarily supports two dataflow strategies: Weight Stationary (WS) and Output Stationary (OS).
  - In **WS dataflow**, the DNN weights are pre-loaded into the PEs and remain stationary, maximizing weight reuse as input activations are streamed through.
  - In **OS dataflow**, partial sums of the output activations remain stationary within the PEs, while both inputs and weights are streamed. This is particularly beneficial for layers with low weight reuse.

The choice of dataflow has a significant impact on performance and energy efficiency.

- **Pipelining:** The degree of pipelining within the array and its PEs can be configured, affecting the maximum clock frequency and overall throughput.

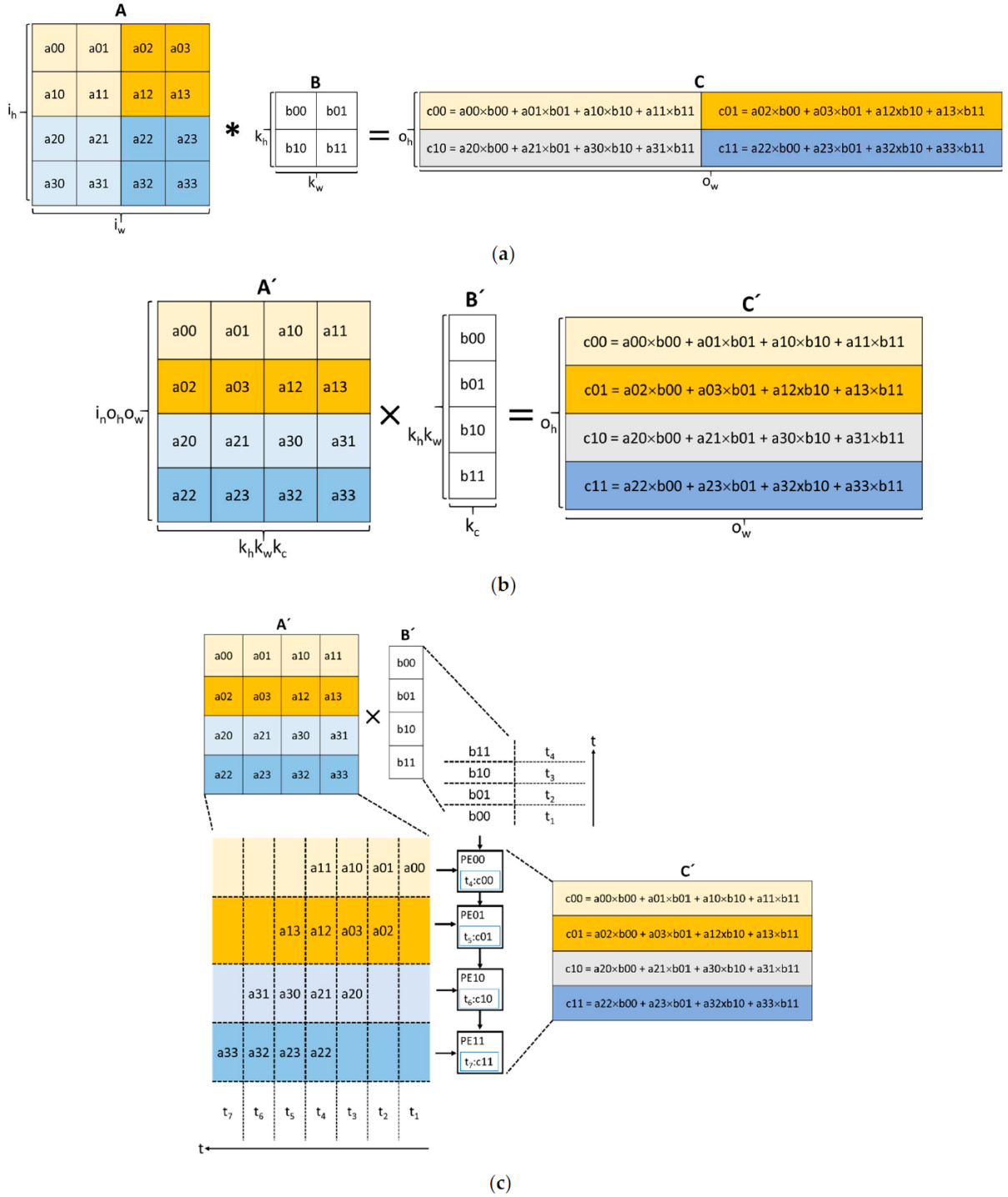
#### 5.3.1.1 Mapping Convolution to Hardware: The `im2col` Transformation

While systolic arrays are fundamentally designed to accelerate General Matrix-Matrix Multiplication (GEMM), they are adept at accelerating 2D convolutions through a crucial data-restructuring technique known as `im2col` ("image-to-column"). This process unrolls the overlapping input image patches from a convolutional layer into the columns of a new, larger matrix. As shown in Figure ??, this new matrix can then be multiplied with a corresponding weight matrix, effectively **converting the convolution into a GEMM operation** that the systolic array can process with maximum efficiency.

Recognizing the computational cost of this pre-processing step, *Gemmini* can be configured with a dedicated hardware unit to perform the `im2col` transformation on-the-fly. This offloads the data restructuring from the host CPU, freeing it for other tasks and streamlining the overall computation pipeline. The decision to include this unit represents a classic area-versus-performance trade-off in the accelerator design.

#### 5.3.2 The On-Chip Memory Subsystem

To feed its high-throughput systolic array, *Gemmini* includes a specialized on-chip memory system. This design represents a deliberate trade-off between hardware-



Hình 5.5: From convolution to systolic array computation: (a) direct convolution operation; (c) example of a systolic array operation; (b) A visualization of the `im2col`-based GEMM operation. The 3D input tensor (left) is transformed into a 2D matrix (center) that can be processed by the systolic array. Adapted from Gookyi et al. [gookyi2023gemmini\_case\_study].

managed caches and software-managed memories.

- **Scratchpad Memory:** A software-managed SRAM used for staging input activations, weights, and intermediate feature maps close to the systolic array. Its purpose is to reduce the latency and energy consumption associated with fetching data from higher levels of the memory hierarchy, such as the shared L2 cache or main memory (DRAM). Its capacity and banking are configurable.
- **Accumulator Memory:** Separate from the main scratchpad, **Gemmini** features a dedicated memory to store the partial sums generated by the MAC operations. This memory typically uses a higher precision data type (e.g., 32-bit integers) than the input data (e.g., 8-bit integers) to maintain accuracy during accumulation.

## 5.4 SoC Integration and System-Level Features

### 5.4.1 The RoCC Coprocessor Interface

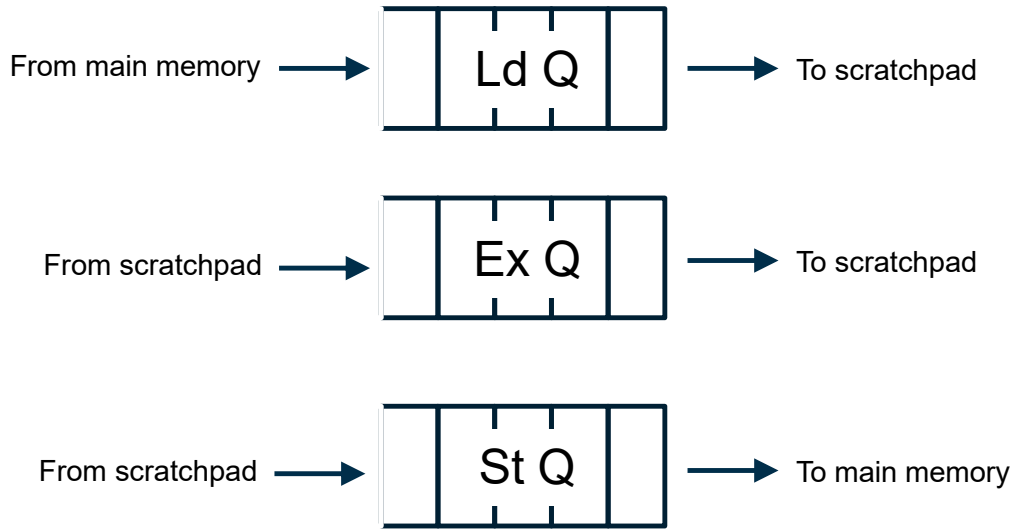
**Gemmini** integrates into the Chipyard SoC via the **Rocket Custom Coprocessor (RoCC)** interface. As a tightly-coupled coprocessor, it receives custom RISC-V instructions dispatched directly by the host **Rocket** Core's pipeline. This allows for low-latency command and control.

This integration uses a decoupled access-execute model, as illustrated conceptually in Figure ???. The host CPU issues three main types of commands to **Gemmini**: load (from main memory to scratchpad), execute (perform computation on data within the scratchpad), and store (from scratchpad to main memory). These commands are placed into separate hardware queues, allowing **Gemmini**'s internal controller then processes these commands asynchronously, and **Gemmini**'s DMA engine to handle data movement in parallel with the systolic array's computations. This overlapping of communication and computation is crucial for hiding memory latency and maximizing the utilization of the systolic array.

### 5.4.2 Virtual Memory Support

To operate correctly under a full-featured operating system like Linux, an accelerator's DMA engine must be able to translate virtual memory addresses into physical addresses. **Gemmini** is designed with this "full-stack" capability. It can be configured





Hình 5.6: Conceptual view of Gemini’s decoupled access-execute pipelines, managed by separate hardware command queues for Load, Execute, and Store operations. Source: [gemini-dac].

to share the host CPU’s Page Table Walker (PTW) to perform its own address translations, and it includes its own local Translation Lookaside Buffer (TLB) to cache these translations and reduce latency.

## 5.5 Summary

In summary, **Gemmini** is a sophisticated, highly configurable DNN accelerator generator that embodies the principles of systolic computation. Its key architectural features include a powerful systolic array, a specialized on-chip memory hierarchy, and a tightly-coupled integration with a host processor via the **RoCC** interface, including support for system-level features like virtual memory. This design enables the high-performance execution of DNN workloads within a full-stack, system-level context.

Having established this architectural overview, the following chapter will now perform a deep dive into the specifics of **Gemmini**’s internal command processing and its multi-level programming model.

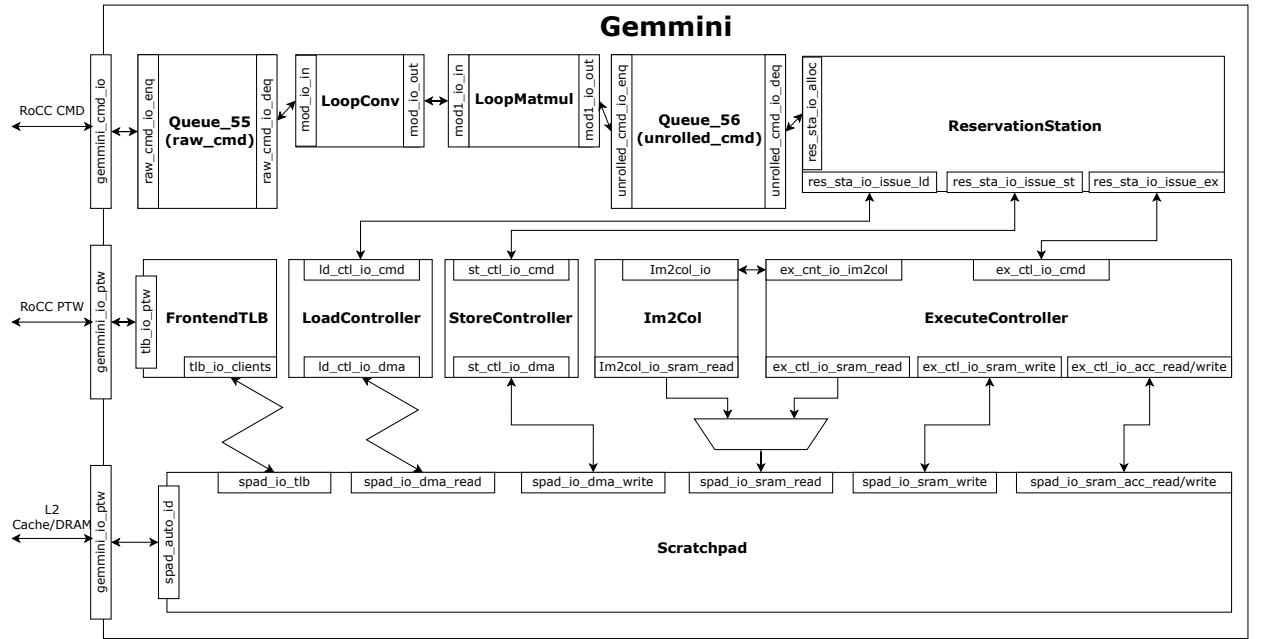


## CHƯƠNG 6: Gemini's Internal Architecture and Programming Model

While Chapter ?? described the conceptual architecture of **Gemini**, this chapter explores its internal structure and the software interfaces used to control it. A deeper understanding of these implementation-level details is necessary to effectively program the accelerator and analyze its performance.

### 6.1 Internal Command and Control Flow

A custom RoCC instruction sent from the **Rocket** core is not executed directly by the systolic array. Instead, it triggers a complex sequence of events within the **Gemini** accelerator's internal controller modules. The overall structure of the generated hardware, as analyzed by Gookyi et al. [gookyi2023gemmini\_case\_study], is shown in Figure ?. This section details the key components of this internal pipeline.

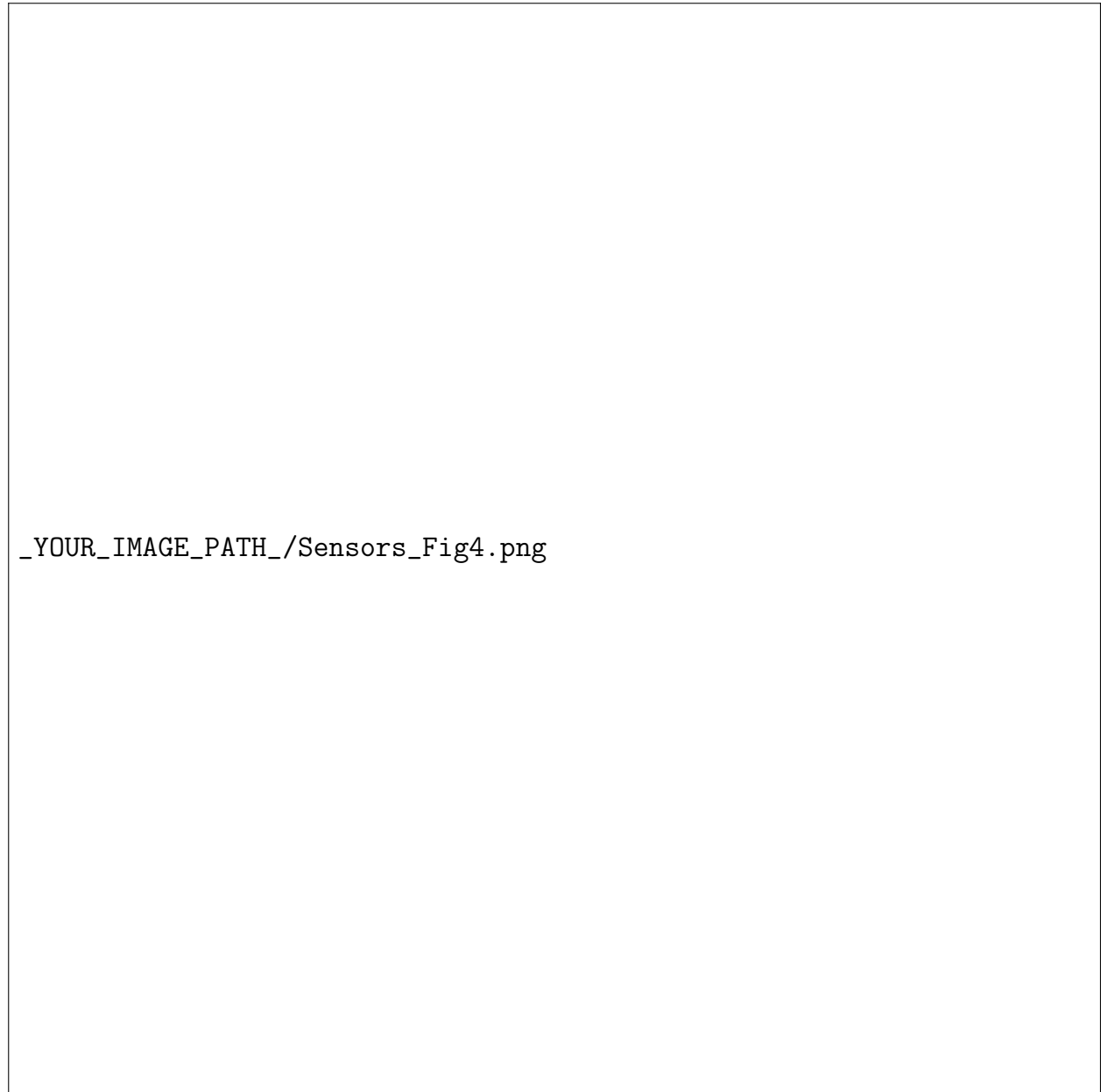


Hình 6.1: Overview of the internal modules in the generated Gemini hardware architecture, showing the flow of commands from the RoCC interface through various queues and controllers. Source: [gookyi2023gemmini\_case\_study].

#### 6.1.1 Command Unrolling and Buffering

The initial RoCC instructions received from the CPU are often "complex" commands that encapsulate entire loops (e.g., 'loop\_matmul'). As shown in Figure ??,

modules like `LoopMatmul` and `LoopConv` are responsible for unrolling these high-level tasks into a stream of simpler, primitive micro-operations for loading, storing, and executing on tiled sub-matrices. These primitive commands are then buffered in a queue before being sent to a central dispatcher.



Hình 6.2: Gemmini’s command unrolling modules (`LoopConv` and `LoopMatmul`), which translate high-level commands into a series of primitive operations. Source: [gookyi2023gemmini\_case\_study].

### 6.1.2 Reservation Station and Execution Control

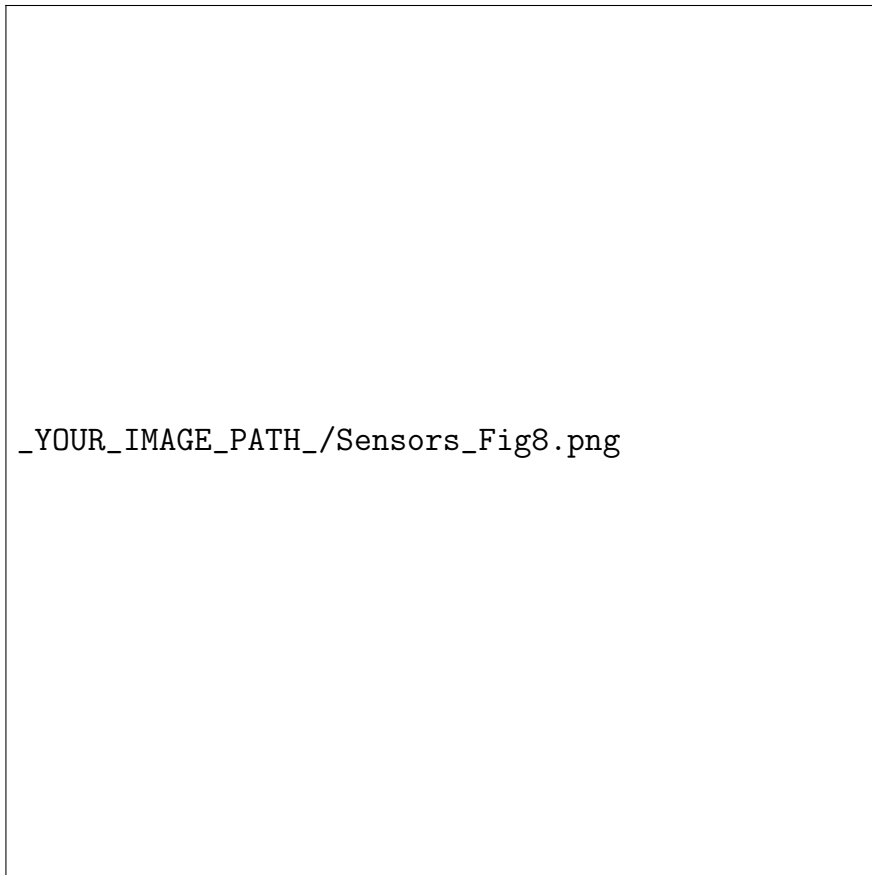
The stream of primitive commands is received by the `ReservationStation` module, which reorders and dispatches them to the appropriate back-end controller. The `ExecuteController` (Figure ??) manages the actual computation. It issues commands to the systolic array and orchestrates the movement of data between the scratchpad and the computational units. It also contains logic to manage optional units, such as the hardware `im2col` unit or data transposer.

\_YOUR\_IMAGE\_PATH\_/Sensors\_Fig6.png

Hình 6.3: The `ExecuteController` module and its associated SRAM control interfaces. Source: [gookyi2023gemmini\_case\_study].

### 6.1.3 The Mesh and Scratchpad


The `ExecuteController` ultimately controls the two lowest-level hardware blocks: the `Mesh` and the `Scratchpad`. The `Mesh` (Figure ??) is the direct hardware implementation of the systolic array, containing the grid of PEs. The `Scratchpad` (Figure ??) implements the on-chip memory, containing the SRAM banks and logic for interfacing with the DMA engine and the systolic array's accumulator.



Hình 6.4: The `Mesh` module, which implements the systolic array of PEs and supports different dataflows (Output Stationary and Weight Stationary). Source: [gookyi2023gemmini\_case\_study].

## 6.2 The Multi-Level Programming Model

To cater to different user needs, from application developers to hardware architects, `Gemmini` supports a multi-level programming model. This stack provides layers of abstraction, allowing users to work at a level appropriate for their task, from direct hardware control to high-level framework integration.



`_YOUR_IMAGE_PATH_/Sensors_Fig9.png`

Hình 6.5: The internal structure of the **Scratchpad** module, showing the various interfaces to the TLB, DMA, and internal memory banks. Source: [gookyi2023gemmini\_case\_study].

### 6.2.1 Low-Level: Gemmini ISA

At the lowest level, **Gemmini** is controlled by its custom RoCC instruction set. These instructions are typically wrapped in C preprocessor macros for easier use from C/C++ code. The instruction set includes primitive commands like `mvin` (move data in), `mvout` (move data out), `preload` (load data into the PE array), and `compute`. It also features more "complex" instructions like `loop_matmul` that abstract away the manual tiling and scheduling of primitive commands, offloading the loop control to hardware state machines [Genc2022GemminiMLSys].

### 6.2.2 Mid-Level: Hand-Tuned C Library

To abstract the complexities of direct ISA programming, **Gemmini** provides a C library (typically accessed via `gemmini.h`). This library offers functions for common DNN operations, such as `tilled_matmul` and `tilled_conv`. These functions implement optimized heuristics for loop tiling and data movement to maximize scratchpad utilization based on the specific hardware parameters of the generated **Gemmini** instance.

### 6.2.3 High-Level: Compiler Support (Conceptual)

The **Gemmini** ecosystem is designed to support higher-level compilation flows. This involves tools that can take DNN models described in standard frameworks like ONNX (Open Neural Network Exchange) and compile them down to **Gemmini** instructions, either by calling the mid-level C library or by directly generating low-level ISA calls. This further simplifies the deployment of DNNs on **Gemmini**-accelerated SoCs.

## 6.3 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 and FPGA resource constraints. Within the **Chipyard** framework, this configuration is defined in a **Scala** configuration file. [inline]Specify the key parameters for your **Gemmini** instance here. For example: The instance was configured with a 16x16 systolic array using 8-bit integer data types for inputs and weights, and a 32-bit integer accumulator. The scratchpad was configured with 256KB of capacity and the accumulator memory



with 64KB. The Weight Stationary (WS) dataflow was selected, and the hardware `im2col` unit was included to offload this pre-processing step from the host CPU. These parameters were chosen to balance performance for target DNN workloads with the resource constraints of the target FPGA. 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 system memory hierarchy, and can be programmed using custom RISC-V instructions dispatched to its RoCC slot.



## TÀI LIỆU THAM KHẢO