Analysis of Chipyard's Rocket Core, BOOM and their Communication Infrastructure

Presented By: Ayesha Qazi and Humail Nawaz

Objectives

- Understanding Rocket and BOOM Specifications
- Comparative analysis of core pipelines
- Introduction to a Cache Coherent Communication Protocol

Core generation using Rocket Chip

1. Setting up Prerequisites
2. Cloning Chipyard Repository
3. Building the RISC-V Toolchain
4. Generation of Core Configuration
5. Simulation
6. Core Customization (Optional)
7. Verilog Generation
8. Core Synthesis

Rocket Core

The Rocket Core is an open-source, in-order, 5-stage RISC-V processor core designed at the University of California, Berkeley, as part of the RISC-V ecosystem. It is a simple, scalar core that follows the RISC-V ISA (Instruction Set Architecture) and is suitable for many general-purpose computing applications. The Rocket Core serves as the base for many RISC-V-based projects and research, providing an efficient, flexible design for educational, industrial, and research purposes.

Features

In-order Execution:
The Rocket Core is an **in-order** processor, meaning that it executes instructions strictly in the order they are fetched.

Optional FPU:
Rocket Core supports an optional **Floating-Point Unit (FPU)** that can be included in the core for applications requiring floating-point arithmetic.

Scalar Core:
The Rocket Core is scalar, meaning it executes one instruction per cycle if there are no pipeline hazards (such as data or control hazards).

MMU and Virtual Memory Support:
Rocket includes a Memory Management Unit (MMU) for virtual memory support. The MMU implements page-based virtual memory translating virtual addresses to physical addresses using page tables, with support for TLB.

Configurable Parameters:
The Rocket Core is designed to be **parameterized**, allowing the user to configure various aspects of the core during synthesis, such as cache size and word size.

Support for Custom Extensions:
Rocket Core allows modification and extension. Users can add custom instructions or hardware accelerators by modifying the Chisel source code.

Multicore Support:
Rocket Core can be integrated into a **multicore system**, where several Rocket cores share a memory hierarchy.

Boots Linux: Rocket Core is capable of running a full **operating system** like Linux, which requires support for virtual memory and user-level execution.

Fetch

Instructions are fetched from the instruction cache (I-Cache). The PC holds the address of the current instruction and is updated (usually incremented) after each instruction fetch.

Decode

In this stage, the fetched instruction is decoded to determine its type (e.g., arithmetic, load/store, branch) and the required operands are identified. The register file is accessed, and the necessary register values are read. Immediate values from the instruction are extracted as well.

Execute

The ALU performs required computation For arithmetic and logical instructions, the ALU processes the operation using the source operands. For memory operations, the effective memory address is computed. For branch instructions, the condition is evaluated, and the PC may be updated based on the result.

Memory Access:

In the Memory Access stage, memory operations are executed. For load instructions, data is read from the Data Cache (D-Cache) or main memory. For store instructions, data is written to the D-Cache. Non-memory instructions skip this stage but pass through it to maintain pipeline flow.

Write Back

The Write-back stage is the final stage of the pipeline, where the results of the instruction (from the ALU or memory) are written back to the register file. This completes the instruction's execution, making the result available for future instructions.

BOOM – Berkeley Out-of-Order Machine.

The Berkeley Out-of-Order Machine (BOOM) is a synthesizable and parameterizable open source RV64GC RISC-V core written in the Chisel hardware construction language. While BOOM is primarily ASIC optimized, it is also usable on FPGAs. Its focus is to create a high performance, synthesizable, and parameterizable core for architecture research.

Fetch

Instructions are fetched from instruction memory and pushed into a Fetch Buffer. Branch prediction also occurs in this stage, redirecting the fetched instructions as necessary.

Decode and Rename:

Decode pulls instructions out of the Fetch Buffer and generates the appropriate Micro-Op(s) to place into the pipeline. The ISA register specifiers (e.g. x0-x31) are then renamed into “physical” register specifiers

Rename and Dispatch:

The micro-operation is then dispatched, or written, into a set of Issue Queues.

Issue and Register Read:

µ-OPs sitting in Issue Queue wait until all of their operands are ready and are then issued. This is the beginning of the out–of–order piece of the pipeline. Issued µ-Ops first read their register operands from the unified Physical Register File.

Execute:

In the execution stage, the functional units reside. Issued memory operations perform their address calculations in the Execute stage, and then store the calculated addresses in the Load/Store Unit which resides in the Memory stage.

Memory Access:

The Load/Store Unit consists of three queues: a Load Address Queue a Store Address Queue and a Store Data Queue. Loads are fired to memory when their address is present in the LAQ. Stores are fired to memory at Commit time.

Write Back:

ALU operations and load operations are written back to the Physical Register File.

Features

Synthesizable:
BOOM is designed Chisel, which allows it to be synthesized into actual hardware (e.g., on FPGAs or in ASIC designs).

Floating Point (IEEE 754-2008):
Includes a **Floating Point Unit (FPU)** that conforms to the **IEEE 754-2008** floating-point standard, which defines how floating-point arithmetic is handled.

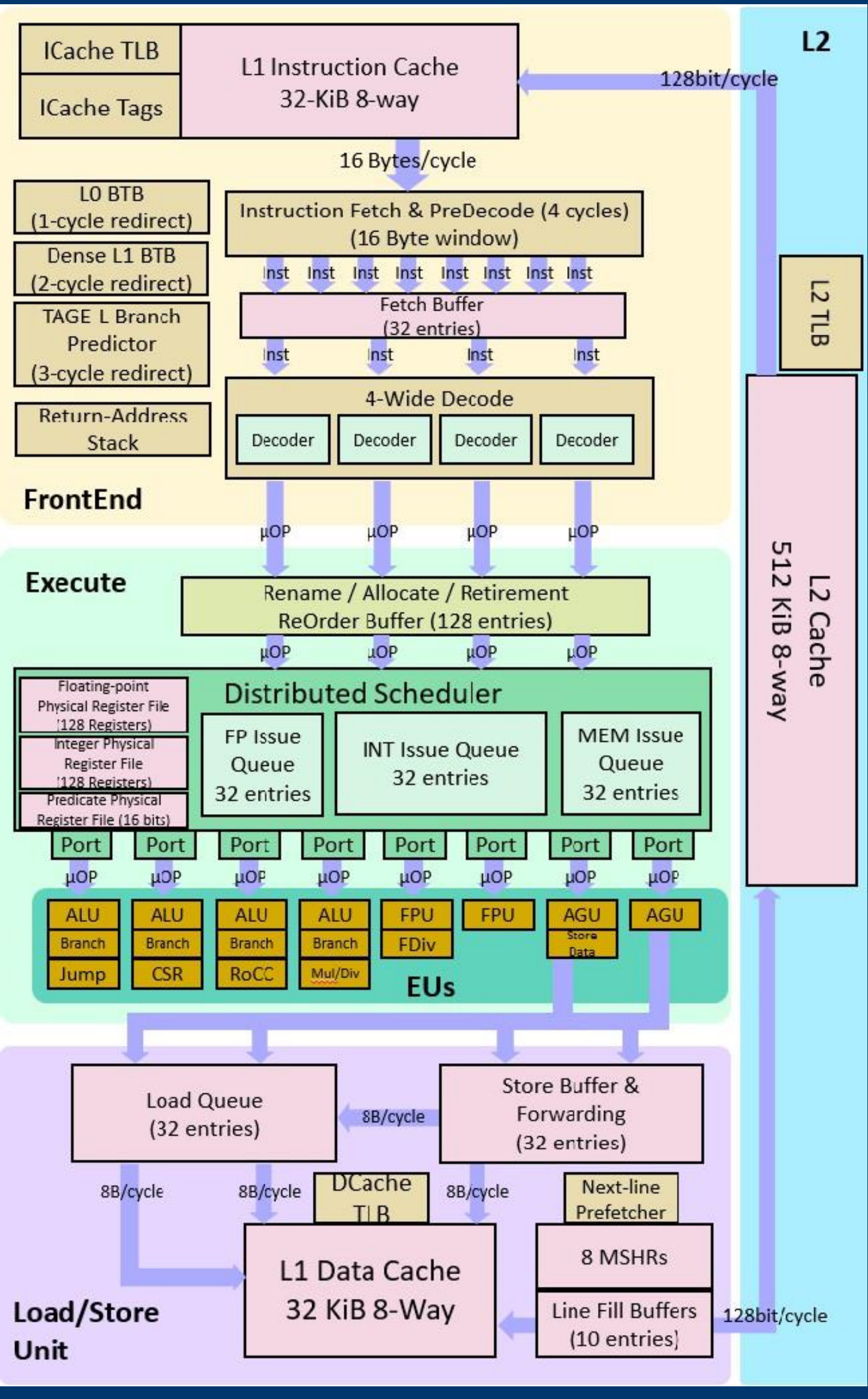
Atomic Memory Operation Support: BOOM supports **atomic memory operations (AMOs)**, which allow certain memory operations (like read-modify-write) to be performed atomically without interference from other cores or memory accesses.

Caches and Virtual Memory: Supports cache hierarchy, (typically L1 I and D caches, and possibly L2) and an MMU for virtual memory

Parameterized: Various architectural aspects (e.g., number of execution units, cache sizes, issue width, etc.) can be adjusted by simply changing parameters in the code.

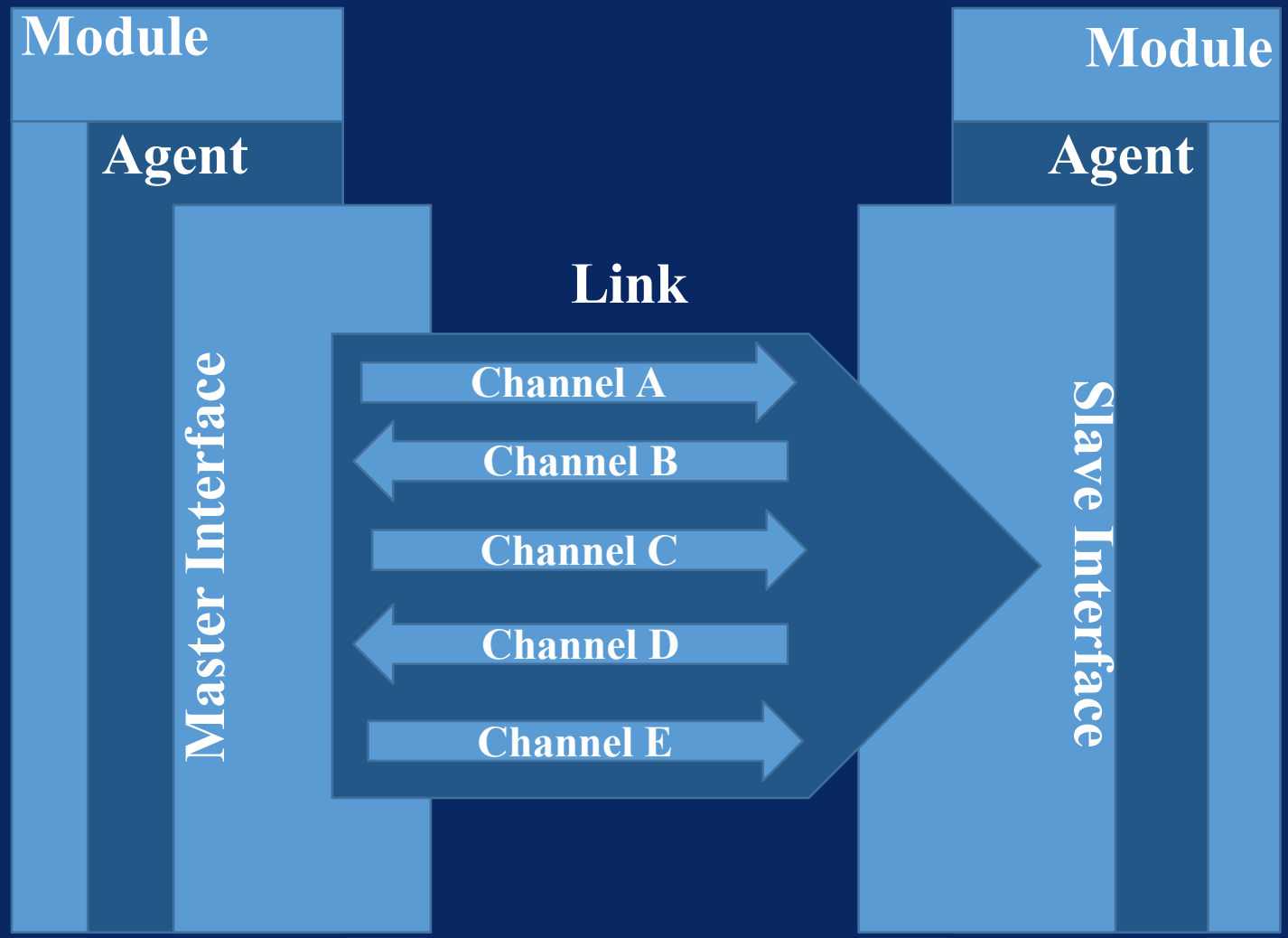
Boots Linux: BOOM is capable of running a full **operating system** like Linux, which requires support for virtual memory and user-level execution.

External Debug: Built-in support for external debugging, allowing developers to halt execution, inspect registers, step through code, and diagnose issues.


ICache TLB
ICache Tags
L1 Instruction Cache
32-KiB 8-way
L2
128bit/cycle
16 Bytes/cycle
L0 BTB
(1-cycle redirect)
Dense L1 BTB
(2-cycle redirect)
TAGE L Branch
Predictor
(3-cycle redirect)
Return-Address
Stack
Instruction Fetch & PreDecode (4 cycles)
(16 Byte window)
Inst Inst Inst Inst Inst Inst Inst Inst
Fetch Buffer
(32 entries)
Inst Inst Inst Inst
4-Wide Decode
Decoder
Decoder
Decoder
Decoder
L2 TLB
FrontEnd
µOP µOP µOP µOP
Execute
Rename / Allocate / Retirement
ReOrder Buffer (128 entries)
µOP µOP µOP µOP
Floating-point
Physical Register File
(128 Registers)
Integer Physical
Register File
(128 Registers)
Predicate Physical
Register File (16 bits)
Distributed Scheduler
FP Issue
Queue
32 entries
INT Issue Queue
32 entries
MEM Issue
Queue
32 entries
L2 Cache
512 KiB 8-way
Port Port Port Port Port Port Port Port
µOP µOP µOP µOP µOP µOP µOP µOP
ALU
Branch
Jump
ALU
Branch
CSR
ALU
Branch
RoCC
ALU
Branch
Mul/Div
FPU
FDiv
FPU
AGU
Store
Data
AGU
EUs
Load Queue
(32 entries)
8B/cycle
Store Buffer &
Forwarding
(32 entries)
8B/cycle
8B/cycle
DCache
TLB
8B/cycle
Next-line
Prefetcher
8 MSHRs
L1 Data Cache
32 KiB 8-Way
Line Fill Buffers
(10 entries)
128bit/cycle
Load/Store
Unit

TileLink

TileLink is a RISC-V standard coherent interconnect protocol that manages communication between cores, memory controllers, peripherals, and caches.



Communication

In a RISC-V system with out-of-order and in-order cores, TileLink ensures cache coherence and efficient memory access. Atomic operations enable core-to-core communication, and shared memory is synchronized with `fence` instructions. Challenges are managed with synchronization primitives and lock-free structures, while Chipyard provides integration and simulation, and Tilelink ensures coherent, low-latency communication.