Succinct Boot Camp: SP1 Core

Lecture Notes

September 17, 2024

1 Introduction to SP1 Core

SP1 Core is the central component of the Succinct zkVM system. It encompasses the essential elements that enable the execution and proof generation for RISC-V programs in a zero-knowledge context. The core is primarily divided into two main components: the Executor and the Machine Crate.

The SP1 Core is designed to:

- Simulate the execution of RISC-V programs
- Generate arithmetic representations of the program execution
- Produce efficient zero-knowledge proofs of correct execution

By understanding the SP1 Core, we gain insight into how general-purpose computation can be transformed into provable statements, bridging the gap between traditional programming and zero-knowledge cryptography.

In this lecture, we will explore:

- The RISC-V architecture and compilation process
- The Executor component and its role in program simulation
- The Machine Crate and its implementation of the CPU AIR (Algebraic Intermediate Representation)
- The multi-table architecture and lookup arguments
- Memory handling and the timestamp-based memory argument
- Optimization techniques used in SP1

Understanding these components will provide a comprehensive view of how SP1 transforms high-level programs into efficient zero-knowledge proofs.

2 RISC-V and Compilation Process

2.1 What is RISC-V?

RISC-V is a reduced instruction set computer (RISC) instruction set architecture (ISA). Unlike complex instruction set computers (CISC), RISC-V aims for simplicity and efficiency:

- Reduced set of instructions, each with a simple and well-defined purpose
- Designed to be modular and extensible
- Open-source architecture, allowing for customization and innovation

RISC-V is gaining popularity due to its openness and flexibility, making it suitable for various applications from embedded systems to high-performance computing.

2.2 Rust to RISC-V Compilation

The compilation process from Rust to RISC-V involves several steps:

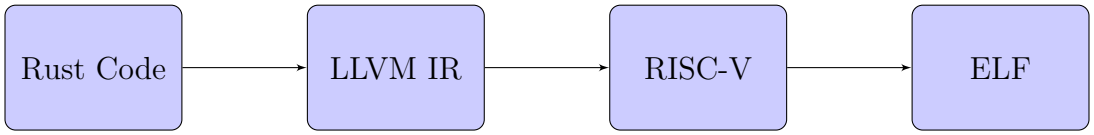

Figure 1: Rust to RISC-V Compilation Pipeline

1. Rust code is first compiled to LLVM IR (Intermediate Representation)
2. LLVM then compiles this IR to RISC-V assembly
3. Finally, the assembly is assembled into an ELF (Executable and Linkable Format) file

LLVM (Low Level Virtual Machine) is a compiler infrastructure that supports multiple front-end languages (like Rust, C, C++) and multiple back-end targets (like RISC-V, x86, ARM).

2.3 RISC-V Specifics for SP1

SP1 uses 32-bit RISC-V with the IM extension. Let's break this down:

- 32-bit: This refers to the width of the registers and the size of memory addresses. It means the system can directly address up to 4GB of memory.
- IM extension:

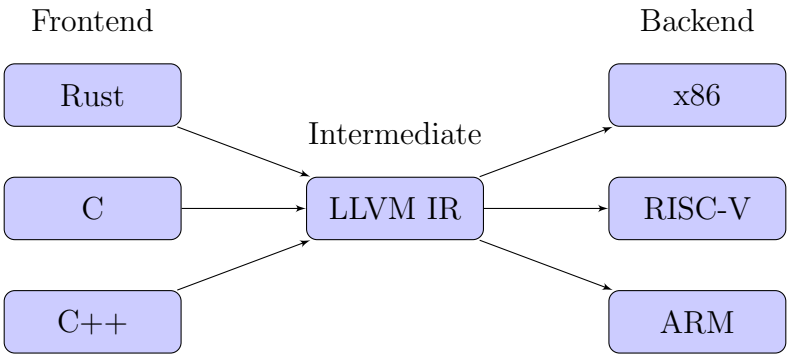

Figure 2: LLVM Compilation Flow

- I: Base Integer Instruction Set
- M: Integer Multiplication and Division

The M extension is particularly important for SP1 as it provides hardware support for multiplication and division operations, which are crucial for many cryptographic and mathematical computations.

The full target name used in SP1 is:
`riscv32im-succinct-zkvm-elf`
This target name specifies:

- riscv32: 32-bit RISC-V architecture
- im: Integer (I) and Multiplication (M) extensions
- succinct: Indicates it's for the Succinct zkVM
- zkvm: Zero-Knowledge Virtual Machine
- elf: The output format is ELF (Executable and Linkable Format)

2.4 SP1 Toolchain

SP1 uses a custom fork of the Rust compiler with specific modifications for zkVM compatibility. These modifications include:

- Custom target specification for the Succinct zkVM
- Implementation of zkVM-specific system calls
- Modifications to the standard library to support the zkVM environment

Here's an example of how you might specify the SP1 target when building:

```
cargo build --target riscv32im-succinct-zkvm-elf
```

This command tells Cargo (Rust's package manager) to use the SP1-specific target when compiling the project.

It's worth noting that the SP1 toolchain includes a fork of the Rust compiler with some specific modifications. These modifications are relatively small but crucial for zkVM compatibility. For instance, they implement custom system calls and modify the standard library to work within the zkVM environment.

The SP1 toolchain ensures that Rust code can be compiled to RISC-V in a way that's compatible with the zkVM environment, allowing developers to write high-level Rust code that can be efficiently proven in zero-knowledge.

3 SP1 Core: Executor

3.1 Overview of the Executor

The SP1 Executor is responsible for simulating the execution of RISC-V instructions and collecting the necessary information for proof generation. It's a crucial component that bridges the gap between the compiled RISC-V code and the zero-knowledge proof system.

3.2 Instruction Execution

The executor follows a fetch-decode-execute cycle:

1. Fetch: Retrieve the instruction at the current program counter
2. Decode: Interpret the instruction to determine the operation and operands
3. Execute: Perform the operation and update the system state

Here's a simplified pseudocode representation of the execution loop:

```
while true:
    instruction = fetch(program_counter)
    opcode, operands = decode(instruction)
    execute(opcode, operands)
    update_program_counter()
    if is_terminal_instruction(opcode):
        break
```

3.3 Memory and Register Handling

SP1 uses a unified memory model where registers are treated as part of the memory space. This is different from physical hardware where registers are separate, faster storage units. In SP1:

- Registers are simply specific memory addresses

- There's no speed difference between accessing registers and memory
- This simplifies the model but may require different optimization strategies compared to physical hardware

3.4 System Calls (E-calls)

E-calls (Environment Calls) are used for system operations and interactions with the host environment. They allow the RISC-V program to perform operations that are beyond the basic instruction set, such as:

- Input/Output operations
- Cryptographic operations (e.g., hashing)
- Interaction with the zkVM environment

In SP1, e-calls are implemented as special instructions that are interpreted by the executor. For example, an e-call might be used to read input data or to perform a complex cryptographic operation that's optimized in the zkVM.

Here's a simplified example of how an e-call might be implemented in the SP1 executor:

```
fn execute_ecall(&mut self, syscall_id: u32) -> Result<(),
    ExecutionError> {
     match syscall_id {
         SYSCALL_READ_INPUT => self.read_input(),
         SYSCALL_WRITE_OUTPUT => self.write_output(),
         SYSCALL_KECCAK => self.keccak_precompile(),
         // ... other syscalls ...
         _ => Err(ExecutionError::UnknownSyscall(syscall_id)),
     }
}
```

This function would be called when the executor encounters an e-call instruction, with the `syscall_id` determining which operation to perform.

3.5 Execution Record

As the executor runs the program, it generates an execution record. This record contains all the information needed to later prove the correct execution of the program. It includes:

- The sequence of instructions executed
- The state of memory and registers at each step
- Any system calls made and their results
- Additional metadata needed for the proof generation

The execution record is crucial for the next stage of SP1, where the arithmetic circuits for zero-knowledge proofs are generated.

3.6 Optimization Techniques in the Executor

The SP1 executor employs several optimization techniques to improve performance:

3.6.1 Paged Memory

Instead of implementing memory as a simple hash map, SP1 uses a paged memory system. This approach is inspired by how real computers manage memory:

- **Concept**: Memory is managed in larger blocks called pages, typically 4 kilobytes in size.
- **Motivation**: Real programs often access memory locations that are near each other, exhibiting spatial locality.
- **Implementation**: Recently used pages are kept in a cache-like structure, improving access times for frequently used memory regions.
- **Benefits**: This system can significantly speed up memory operations by reducing the overhead of individual memory accesses.

Here's a simplified example of how paged memory might be implemented:

```
struct PagedMemory {
    pages: HashMap<u32, Vec<u8>>,
    page_size: usize,
}

impl PagedMemory {
    fn read(&self, address: u32) -> u8 {
        let page_num = address / self.page_size as u32;
        let offset = address % self.page_size as u32;
        self.pages.get(&page_num)
            .map(|page| page[offset as usize])
            .unwrap_or(0)
    }
    // ... write and other methods ...
}
```

3.6.2 Byte Lookup Table

For certain operations, SP1 uses a pre-computed byte lookup table:

- **Purpose**: This table contains the results of common operations on bytes.
- **Operations covered**: Typically includes bitwise operations (AND, OR, XOR) and other byte-level manipulations.
- **Benefit**: Reduces the complexity of constraints for these operations in the zero-knowledge proof, leading to more efficient proofs.

3.6.3 Multi-table Architecture

Instead of having a single, complex table for all operations, SP1 uses multiple, simpler tables for different operations:

- **Concept**: Separate tables for different types of operations (e.g., addition, multiplication, bitwise operations).
- **Implementation**: Uses lookup arguments to connect the main CPU table with these specialized operation tables.
- **Advantages**:
  - Simplifies constraints for individual operations
  - Allows for more efficient use of table space
  - Enables potential parallelization of proof generation for different operation types

3.6.4 Efficient Range Checks

SP1 employs clever techniques to optimize range checks, which are crucial for many operations:

- **Byte decomposition**: Instead of checking if a 32-bit number is within a certain range, SP1 often decomposes the number into bytes and checks each byte individually.
- **Lookup-based checks**: Utilizes the byte lookup table for efficient small-range checks.
- **Combined lookups**: For larger range checks, SP1 combines multiple lookups to reduce the overall constraint complexity.

These optimization techniques work together to significantly improve the performance of the SP1 executor, allowing it to handle complex programs while maintaining reasonable proving times. The paged memory system, in particular, helps to bridge the gap between the abstract zkVM environment and the memory access patterns of real-world programs, leading to more efficient execution and proof generation.

4 SP1 Core: Machine Crate and CPU AIR

4.1 Overview of the Machine Crate

The Machine Crate is a crucial component of SP1 Core that implements the arithmetic circuits necessary for zero-knowledge proofs. It takes the execution record generated by the Executor and transforms it into a set of polynomial constraints that can be efficiently proven and verified.

4.2 CPU AIR (Algebraic Intermediate Representation)

The CPU AIR is a table representation of the program execution. Each row in the AIR represents one step of the program's execution:

PC	Opcode	opa	opb	opc	a	b	c	aux
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 3: AIR Table Structure

Where:

- PC: Program Counter
- Opcode: The operation being performed
- opa, opb, opc: Operand addresses
- a, b, c: Operand values
- aux: Auxiliary information for proof generation

4.3 Constraint Types

The AIR includes several types of constraints to ensure the correctness of the execution:

1. **Instruction correctness**: Ensures that the instruction at each PC is valid
2. **Opcode processing**: Verifies that each opcode is executed correctly
3. **Memory and register updates**: Checks that memory and register values are updated correctly
4. **Program counter updates**: Ensures that the PC is updated correctly after each instruction

For example, for an ADD instruction, we might have constraints like:

- $c = a + b \pmod{2^{32}}$ (for 32-bit addition)
- $PC' = PC + 4$ (move to next instruction)

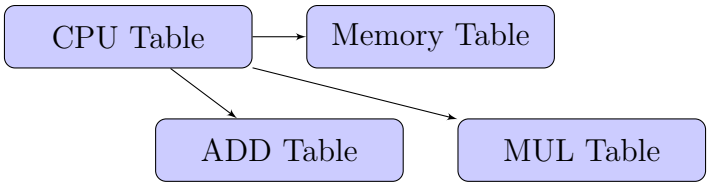

Figure 4: Multi-table Architecture

4.4 Multi-table Architecture

SP1 uses a multi-table architecture with lookup arguments:

This architecture allows for more efficient proof generation by separating different types of operations into their own tables. The CPU table then uses lookup arguments to reference these other tables when needed.

Here's a simplified example of how the ADD table might be implemented:

```
struct AddTable {
    a: Column<F>,
    b: Column<F>,
    c: Column<F>,
    carry: Column<F>,
}

impl AddTable {
    fn generate_constraints(&self, builder: &mut ConstraintBuilder<F>)
        {
        builder.add_constraint(
            "a + b = c + carry * 2^32",
            self.a + self.b - self.c - self.carry * F::from(1u64 << 32)
        );
        // Additional constraints for carry, etc.
    }
}
```

The multi-table architecture offers several advantages:

- Simplifies constraints for individual operations
- Allows for more efficient use of table space
- Enables parallel processing of different operation types

4.5 Lookup Arguments

Lookup arguments are a key component of SP1's multi-table architecture. They allow the CPU table to efficiently reference entries in other tables without needing to replicate all the constraint logic.

The lookup argument works by computing a kind of "hash" of each row in both tables and then ensuring that these hashes match. Mathematically, it's implemented using a polynomial identity, which can be efficiently verified in the proof system.

Here's a simplified explanation of how a lookup argument works:

1. Compute a hash-like function for each row in both tables: $h(a, b, c) = \frac{1}{a+\alpha b+\alpha^2 c}$ where α is a random value provided by the verifier.
2. Compute the product of these values for each table.
3. If the products match, with high probability, every row in the first table appears in the second table.

This approach allows SP1 to maintain separate tables for different operations while still ensuring consistency between them.

4.6 Memory Argument

The memory argument uses a timestamp-based approach with accumulators:

$$\text{Accumulator} = \sum_i \text{Hash}(\text{address}_i, \text{value}_i, \text{timestamp}_i) \cdot \text{sign}_i \tag{1}$$

Where:

- address_i: Memory address being accessed
- value_i: Value being read or written
- timestamp_i: When the access occurred
- sign_i: +1 for reads, -1 for writes

This approach allows for efficient verification of memory consistency throughout the program execution. It works by ensuring that every memory read is matched with the most recent write to that address, and that the final memory state is consistent with all operations performed.

Here's a simplified example of how this might be implemented:

```
struct MemoryAccess {
    address: Column<F>,
    value: Column<F>,
    timestamp: Column<F>,
    is_read: Column<F>,
}

impl MemoryAccess {
    fn generate_constraints(&self, builder: &mut ConstraintBuilder<F>)
        {
```

```
        let sign = self.is_read * F::ONE + (F::ONE - self.is_read) * (-
            F::ONE);
        builder.add_to_accumulator(
            "memory_accumulator",
            hash(self.address, self.value, self.timestamp) * sign
        );
        // Additional constraints to ensure correct memory access
    }
}
```

This memory argument is a key part of ensuring the integrity of memory operations in the zero-knowledge proof.

5 Optimization Techniques and Future Directions

5.1 Current Optimization Techniques

SP1 employs several optimization techniques to improve the efficiency of proof generation and verification:

5.1.1 Byte Lookup Table

The byte lookup table is a pre-computed table that contains the results of common operations on bytes. This optimization is particularly useful for bitwise operations and range checks.

- **How it works**: Instead of constraining bitwise operations directly, which can be expensive in arithmetic circuits, SP1 uses lookups to a pre-computed table.
- **Benefits**: Reduces the complexity of constraints for bitwise operations, leading to more efficient proofs.

Here's a simplified example of how the byte lookup table might be implemented:

```
struct ByteLookupTable {
    input1: Column<F>,
    input2: Column<F>,
    and_result: Column<F>,
    or_result: Column<F>,
    xor_result: Column<F>,
}

impl ByteLookupTable {
    fn generate_table() -> Vec<(u8, u8, u8, u8, u8)> {
        let mut table = Vec::with_capacity(256 * 256);
        for i in 0..=255u8 {
            for j in 0..=255u8 {
                table.push((i, j, i & j, i | j, i ^ j));
            }
        }
```

```
            table
        }
    }
```

5.1.2 Range Check Optimizations

SP1 uses clever techniques to optimize range checks, which are crucial for many operations:

- **Decomposition into bytes**: Instead of checking if a 32-bit number is within a certain range, SP1 often decomposes the number into bytes and checks each byte individually.
- **Combined lookups**: For larger range checks, SP1 combines multiple lookups to reduce the overall constraint complexity.

5.1.3 Multi-table Architecture

As discussed earlier, the multi-table architecture allows for more efficient use of table space and simplifies constraints for individual operations.

6 Conclusion

SP1 represents a significant advancement in the field of zero-knowledge proofs, particularly in the context of general-purpose computation. By leveraging the RISC-V architecture and innovative proof techniques, SP1 provides a flexible and efficient platform for generating zero-knowledge proofs of program execution.

Key takeaways from this overview include:

- The use of RISC-V as a base architecture provides a well-defined and extensible instruction set for zkVM implementation
- The multi-table architecture with lookup arguments allows for efficient proof generation by separating different types of operations
- The unified memory model simplifies certain aspects of the implementation but requires careful optimization
- Ongoing research and development in areas such as arithmetization and lookup optimizations hold promise for further improving the efficiency and capabilities of the SP1 system

As zero-knowledge proofs continue to gain importance in areas such as blockchain technology, privacy-preserving computation, and secure multi-party computation, systems like SP1 will play a crucial role in enabling practical and efficient implementations of these technologies.

Future work in this area will likely focus on further optimizing the proof generation process, expanding the range of supported operations and programming paradigms, and exploring novel applications of zkVM technology across various domains.