Implementation of Nock Combinator Logic in Hardware: The NockPU Project

John Smith ~ `mopfel-winrux`
Native Planet

Abstract

This paper presents the design, implementation, and evaluation of NockPU, a hardware-based processor that directly executes Nock, a minimalist combinator calculus that serves as the foundation for the Urbit computing platform. While most Nock interpreters operate in software, this research explores the opportunities and challenges of implementing Nock's combinator operations directly in hardware using an FPGA. The paper details the design decisions involved in representing nouns in memory, the stackless approach to tree traversal, and the architectural components required for efficient execution. Performance analysis reveals both the advantages of hardware-based execution for certain operations and the significant memory challenges inherent to combinator reduction. The NockPU architecture demonstrates new possibilities for specialized functional hardware that maintains semantic equivalence with software implementations while potentially offering advantages in determinism and power efficiency. This work contributes to the broader understanding of hardware-based approaches to executing non-traditional computation models while highlighting specific optimization techniques for combinator reduction.

Manuscript submitted for review.

Address author correspondence to ~ `mopfel-winrux`.

Contents

1 Introduction

Combinator calculi represent a fundamental approach to computation based on the application and reduction of combinators - operators that compose and transform data without requiring named variables. Nock is a specific combinator calculus developed as the foundation of Urbit, a clean-slate computing platform. It provides a deterministic, stateless computing environment that trades performance for perfect semantic clarity. While Nock interpreters typically run in software environments (e.g., the C-based Vere interpreter and Rust-based NockVM interpreter for Urbit), little exploration has been done regarding the direct implementation of Nock in hardware.

Implementing Nock in hardware presents unique challenges fundamentally different from traditional assembly languages. While conventional processors operate on byte buffers stored in linear memory arrays, Nock computation operates on *nouns* - recursive tree structures that exist as linked data rather than contiguous memory blocks. This fundamental difference means that traditional hardware architectures, optimized for sequential memory access patterns and fixed-width data types, are poorly suited for the tree traversal and dynamic memory allocation patterns inherent to Nock execution. The challenge lies not merely in implementing the Nock operations themselves, but in developing memory representations and traversal mechanisms that can efficiently handle the recursive, pointer-based data structures that define Nock's computational model.

The motivation behind this research was initially quite straightforward: to address skepticism about whether Nock could be effectively implemented in hardware. However, the investigation quickly revealed deeper insights about the relationship between combinator reduction and hardware architecture, along with opportunities to develop novel memory management techniques for tree-based computation.

1.1 Research Context and Gap

Hardware implementation of functional languages has historical precedent, with notable examples including the SKIM (S, K, I reduction machine) developed at Cambridge in the 1970s and more recent work on The Reduceron at the University of York. However, these projects have typically focused on more traditional functional programming languages rather than the minimal, axiomatic approach of Nock. Additionally, most prior research has not sufficiently addressed the challenges of scaling such systems to utilize both on-chip and off-chip memory effectively.

1.2 Research Questions and Objectives

This project aimed to answer several key questions:

1. How can Nock's nouns and operations be efficiently represented in hardware memory?
2. What hardware architecture best supports the pattern of execution required by Nock?
3. How can tree traversal be implemented without a stack-based approach?
4. What performance characteristics emerge from a hardware implementation compared to software interpreters?

The primary objective was to build a Verilog-based NockPU that could perform all standard Nock operations, thereby demonstrating the feasibility of hardware-based combinator reduction while identifying optimal design patterns for such an implementation.

2 Background and Related Work

2.1 Nock Specification and Semantics

Nock is a minimalist combinator calculus defined by a small set of axiomatic rules (`~sorreg-namtyv`, 2013). Its specification begins with: "A noun is an atom or a cell. An atom is a natural number. A cell is an ordered pair of nouns." The computational core of Nock is expressed through reduction rules that transform nouns based on operator codes. For example, the reduction rule `*[a 0 b]` performs a slot operation (tree addressing), while `*[a 2 b c]` evaluates `*[*[a b] *[a c]]`, and so forth.

Nock's extreme simplicity makes it an interesting target for hardware implementation. It requires only a handful of operations, has no need for floating-point arithmetic, and functions in a completely deterministic manner (`~sorreg-namtyv` et al., 2016). Yet, this same simplicity can lead to computational inefficiency when compared to traditional architectures, as Nock requires graph transformations for even basic arithmetic operations.

2.2 Digital Hardware Design Fundamentals

Field-Programmable Gate Arrays (FPGAs) allow for reconfigurable digital hardware design, making them ideal for prototyping novel processor architectures. Unlike traditional software that executes sequentially, hardware designs in languages like Verilog describe circuits where operations occur in parallel, governed by clock cycles.

The design process involves:

1. Creating a high-level design using Hardware Description Language (HDL)
2. Building test benches to verify functionality
3. Synthesis (converting HDL to netlist)
4. Place and Route (mapping netlist to FPGA resources)
5. Timing analysis and program file generation

2.3 Existing Hardware Implementations of Functional Languages

Several significant projects have attempted to implement functional programming languages directly in hardware:

SKIM (S, K, I reduction machine): Developed by Clarke et al. at Cambridge in the 1970s and 1980s (Clarke et al.; Norman, Clarke, and Stoye, 1980; 1984), SKIM implemented combinatory logic directly in hardware. It pioneered many of the techniques for graph reduction that influenced later work.

The Reduceron: More recently, work at the University of York by Naylor and Runciman has produced The Reduceron (Naylor; Naylor and Runciman, 2009; 2008), an FPGA-based graph reduction machine initially designed for executing Haskell programs using basic combinators (S, K, I, B, C). Later versions moved to more complex supercombinator implementations.

While these implementations provide valuable precedent, they differ from the NockPU in several important ways:

1. They primarily target conventional functional languages rather than a minimal combinator calculus like Nock

2. They typically employ stack-based reduction strategies

3. They have focused less on scalability across on-chip and off-chip memory boundaries

2.4 Stack-based vs. Stackless Architectures

Traditional approaches to traversing tree structures in hardware often rely on stack-based mechanisms, wherein a stack stores return addresses or intermediate state during traversal. While intuitive, this approach can introduce complexity in hardware implementation and potentially limit parallelism.

The stackless approach, in contrast, embeds navigational information directly within the tree structure itself, allowing for traversal without external stack state. This technique requires careful consideration of how to mark nodes during traversal and how to restore the tree to its original form afterward, but can offer advantages in certain hardware contexts.

2.5 Memory Representation Challenges in Functional Computing

Representing functional data structures efficiently in memory presents several challenges. Traditional memory is organized as a linear array of words, while functional data often takes the form of trees or graphs. Additionally, functional programs frequently create and discard temporary structures during evaluation, necessitating effective memory management strategies.

In hardware implementations, memory access patterns significantly impact performance. On-chip memory offers fast access but limited capacity, while off-chip memory provides greater capacity at the cost of higher latency. Effective design must carefully balance these tradeoffs.

3 NockPU Architecture

3.1 System Overview and Design Philosophy

The NockPU is designed as a specialized processor that directly executes Nock operations in hardware. Its architecture emphasizes several key principles:

1. **Memory-centric design**: Since combinator reduction is fundamentally about memory manipulation, the architecture prioritizes efficient memory operations.

2. **Deterministic execution**: The system maintains Nock's deterministic nature, ensuring consistent results for identical inputs.

3. **Stackless traversal**: Rather than relying on a traditional stack for tree traversal, the NockPU embeds traversal state within the memory structure itself.

4. **Scalability across memory boundaries**: The design accommodates both on-chip and off-chip memory, allowing for larger computations than could fit in on-chip memory alone.

At a high level, the NockPU consists of a memory traversal unit (MTU) that controls the overall execution flow, specialized modules for executing different Nock operators, and memory management components including a memory unit (MU) and garbage collector.

The architecture implements explicitly single-threaded execution, consistent with Nock's deterministic semantics. This design choice ensures that all computations proceed in a predictable, sequential manner, eliminating race conditions and maintaining the mathematical purity that characterizes Nock evaluation.

3.2 Memory Representation Model

3.2.1 28-bit Nouns in 64-bit Words

The NockPU represents Nock nouns using a custom memory format. Each memory cell is 64 bits wide, divided as follows:

- 8 tag bits (highest bits)
- 28 bits for the head noun
- 28 bits for the tail noun

This representation allows for efficient storage of Nock's binary tree structure while providing room for necessary metadata. The 28-bit limitation for nouns was chosen based on practical FPGA constraints: most commercially available FPGAs have at most 256MB of directly accessible on-board memory, making 28-bit addressing sufficient while maintaining efficient 64-bit word alignment. This design choice facilitates straightforward scaling across different FPGA platforms, as the architecture can be trivially adjusted for larger memory configurations when available, though the architecture includes provisions for handling arbitrarily large atoms through linked representations.

3.2.2 Tag Bit Utilization

The 8 tag bits serve several crucial functions:

- **Execute bit** (bit 63): Marks a cell as requiring execution
- **Stack bit** (bit 62): Identifies cells containing an operation code and its operand
- **Reserved bit** (bit 61): Reserved for future architectural extensions
- **Large atom bit** (bit 60): Indicates an atom larger than 28 bits
- **Head traversal bit** (bit 59): Tracks traversal state for the head
- **Tail traversal bit** (bit 58): Tracks traversal state for the tail
- **Head tag bit** (bit 57): Distinguishes whether the head is an atom or cell
- **Tail tag bit** (bit 56): Distinguishes whether the tail is an atom or cell

These tag bits allow the processor to efficiently determine the nature of each memory cell and maintain traversal state without requiring an external stack.

For example, a simple Nock cell `[42 43]` would be represented as:

- Tag bits: `00000011` (indicating both head and tail are atoms)
- Head: `0x000001A` (28-bit representation of 42)
- Tail: `0x000002B` (28-bit representation of 43)

3.2.3 Atom and Cell Representation

In the NockPU memory model, atoms (natural numbers) are represented directly within the 28-bit fields when possible. For atoms that exceed this limit, the system can use multiple memory cells linked together.

Cells are represented as pointers to other memory locations. When both the head and tail of a cell are atoms, they can be stored directly within a single memory word. When either is a cell, the corresponding field contains a pointer to another memory location.

3.3 Stackless Tree Traversal Mechanism

3.3.1 Program Pointer and Back Pointer Methodology

The NockPU implements a stackless approach to tree traversal (Burrows, 2009) using two primary pointers:

1. **Program Pointer (P)**: Points to the current node being processed
2. **Back Pointer (B)**: Points to the previous node in the traversal

Together, these pointers allow the system to navigate the tree structure without requiring a separate stack. The approach fundamentally works by leaving ”breadcrumbs” in the form of modified pointers that enable retracing steps back up the tree after descending.

3.3.2 Breadcrumb Trail Implementation

As the processor traverses the tree, it modifies the memory cells it visits, effectively leaving a trail that can be followed back up. When descending into a subtree, the processor redirects the pointer in the left part of the cell to point to the parent node, from which execution flow just came. This creates a pathway back up the tree.

The tag bits for head and tail traversal (bits 59 and 58) track which branches have been visited, allowing the processor to determine which subtree to explore next during traversal.

This mechanism ensures that:

1. The processor can always retrace its steps
2. The original tree can be reconstructed after traversal

3. Shared subtrees remain unmodified, preserving the integrity of the graph

3.4 Control Flow Architecture

3.4.1 Memory Traversal Control

The Memory Traversal Unit (MTU) serves as the master controller for the NockPU, orchestrating the overall execution flow. It performs several key functions:

1. Initiates tree traversal to find nodes marked for execution
2. Maintains the program and back pointers
3. Passes control to specialized execution modules when appropriate
4. Coordinates memory access through the memory multiplexer

The MTU operates according to a finite state machine that manages the complex coordination between memory operations, tree traversal, and execution control. The state machine includes states for memory access initiation, traversal coordination, execution delegation, and result handling. State transitions are triggered by completion signals from subordinate modules, memory operation acknowledgments, and the detection of execution markers within the traversed tree structure. This state-driven approach ensures that all memory operations complete properly before proceeding and that control flow remains deterministic throughout the computation process.

3.4.2 Execute Module

The Execute module handles the reduction of Nock operations when triggered by the MTU. It receives the address and data for a cell marked for execution, performs the appropriate transformation according to the Nock operation code, and returns control to the MTU when complete.

This module contains specialized logic for each Nock operator (0 through 11), implementing their specific reduction rules. For operators that generate nested executions, the Execute module restructures the memory to reflect the transformed computation and marks the relevant cells for future execution.

3.4.3 Operational Modules

In addition to the core Execute module, the NockPU includes several specialized modules for specific operations:

- **Cell Block**: Handles type checking for the cell operator (op code 3)
- **Increment Block**: Implements increment operations (op code 4)
- **Equal Block**: Performs equality comparisons (op code 5)
- **Edit Block**: Handles tree modification for the replace operator (op code 10)

These specialized modules allow for more efficient implementation of specific operations and better utilization of hardware parallelism.

3.4.4 Error Handling and System Integrity

The NockPU includes a comprehensive error detection and reporting system to maintain system integrity during execution. When an error condition is encountered—such as malformed nouns, invalid operation codes, or memory access violations—the system raises an error signal and provides diagnostic information through an error code bus. This approach allows for graceful error handling while maintaining the deterministic nature of Nock execution, ensuring that invalid computations are detected rather than producing undefined results.

3.4.5 Garbage Collection Implementation

The NockPU implements a Cheney-style copying garbage collector based on the algorithm described by Clark (Clark, 1976). This implementation addresses the significant memory consumption challenges inherent in combinator reduction by reclaiming memory occupied by intermediate structures that are no longer reachable.

The garbage collection process operates as follows:

1. **Traversal Reset**: Before initiating garbage collection, the system resets any active tree traversal state, ensuring that breadcrumb modifications are properly unwound and the memory representation returns to its canonical form.

2. **Copying Phase**: Using Cheney's two-space copying algorithm, reachable nouns are copied from the current memory space to a clean memory partition, with pointer updates maintaining referential integrity.

3. **Space Flip**: Once copying is complete, the roles of the two memory spaces are exchanged, making the compacted space the active working memory.

A key advantage of this design is that computation state does not need to be recreated after garbage collection. Since the NockPU's execution model is based on marking cells for execution rather than maintaining complex execution stacks, the collector can preserve all necessary computational context during the copying process. This allows garbage collection to occur transparently without requiring expensive state reconstruction. The tradeoff is that the traversal reset process takes some clock cycles, but this approach eliminates the need to maintain extra state during garbage collection.

The copying collector approach is particularly well-suited to the NockPU's stackless architecture, as it eliminates the need to traverse and update complex stack structures during collection. The collector operates entirely through memory scanning and pointer updating, maintaining compatibility with the breadcrumb-based traversal mechanism used throughout the system.

4 Implementation Details

4.1 Hardware Design Process and Tools

4.1.1 Verilog Implementation

The NockPU was implemented in Verilog, a hardware description language that allows for precise control over the digital circuit design. The implementation follows a modular approach, with separate components for different functional aspects of the processor.

Key Verilog modules include:

- Memory Unit (`memory_unit.v`)
- Memory Traversal Unit (`mem_traversal.v`)
- Execute Module (`execute.v`)
- Specialized Operation Blocks
- Memory Multiplexer (`memory_mux.v`)
- Control Multiplexer (`control_mux.v`)

Each module was designed with clear interfaces and tested independently before integration.

4.1.2 Testing and Verification Methodology

A comprehensive testing framework was developed to verify the correctness of the NockPU implementation across multiple levels of abstraction:

End-to-End Testing: The primary verification approach uses an end-to-end test bench (`execute_tb`) that inputs known Nock formulas and compares the resulting output with a reference Nock interpreter. This approach ensures semantic equivalence between the hardware implementation and established software interpreters, validating that the NockPU produces correct results for complete Nock programs.

Component-Level Testing: Individual subsystems are verified through dedicated test benches:

- Memory traversal test bench: Verifies the correctness of the stackless tree traversal mechanism, ensuring proper navigation and breadcrumb management
- Memory operations test bench: Validates basic memory read, write, and allocation operations across the bisected memory architecture

This modular testing approach allows for isolation of functionality and systematic debugging, enabling verification of both individual components and their integration. The test benches provide comprehensive coverage of the processor's operational modes and edge cases, ensuring robust implementation of the Nock specification.

4.2 Nock Operation Implementation

4.2.1 Basic Operations (Slot, Zero, One)

The simplest Nock operations are implemented directly within the memory traversal and execution modules:

Slot Operation (`*[a 0 b]`): Implemented by traversing the subject tree according to the address pattern specified by `b`. The implementation uses a bit-wise approach where each bit in the address determines whether to follow the head or tail pointer.

Constant Operations (`*[a 1 b]`): Simply returns the constant `b` regardless of the subject. This is implemented by writing the value of `b` directly to the result cell.

4.2.2 Tree Manipulation Operations

Operations that transform the tree structure are implemented through carefully orchestrated memory manipulations:

Evaluation (`*[a 2 b c]`): Constructs a new tree representing `*[*[a b] *[a c]]`, marking the appropriate cells for execution.

Cell Testing (`*[a 3 b]`): Examines whether the result of `*[a b]` is a cell, returning the appropriate truth value.

Increment (`*[a 4 b]`): Increments the result of `*[a b]` by one, using specialized logic.

Equality (`*[a 5 b c]`): Compares the results of `*[a b]` and `*[a c]` for equality.

4.2.3 Conditional Operation Implementation

The conditional operation (`*[a 6 b c d]`) is particularly challenging due to its branching nature. In the NockPU, it is implemented by constructing the equivalent expression `*[a *[[c d] 0 *[[2 3] 0 *[a 4 4 b]]]]` directly in memory.

This approach, while computationally expensive, maintains semantic equivalence with the Nock specification and demonstrates how even complex operations can be expressed through graph transformation rather than traditional control flow.

4.2.4 Composition, Push, and Call Operations

Operations 7, 8, and 9 share a common implementation pattern in the NockPU: they perform graph reduction by writing the equivalent expanded expression directly into memory, then rely on the normal execution order to evaluate the result correctly.

Composition (`*[a 7 b c]`): The NockPU writes the graph reduction `*[*[a b] c]` directly into memory. The normal execution order ensures that `*[a b]` is evaluated first, with its result becoming the subject for the subsequent evaluation of `c`.

Push (`*[a 8 b c]`): Implemented by writing the expansion `*[[*[a b] a] c]` into memory. The execution system constructs a cell containing both the result of `*[a b]` and the original subject `a`, creating the augmented context needed for evaluating `c`.

Call (`*[a 9 b c]`): Constructs the graph reduction `*[*[a c] 2 [0 1] 0 b]` in memory. This creates the function call frame structure, with normal execution order ensuring proper evaluation sequence: first `*[a c]` to obtain the function, then the constructed evaluation context.

This unified approach eliminates the need for specialized control flow logic in hardware. Instead, the NockPU leverages its graph reduction capabilities and the inherent ordering properties of the traversal mechanism to achieve correct execution semantics for all three operations.

4.2.5 Edit Operation Implementation

The edit operation (`*[a 10 [b c] d]`) performs tree modification through the specialized Edit Block module. This operation implements the tree editing function `#[b *[a c] *[a d]]`, which modifies the tree structure at address `b` by replacing the value with the result of `*[a c]` and continuing evaluation with `*[a d]`.

The Edit Block uses careful pointer manipulation to modify tree structures while preserving shared subtrees and maintaining memory consistency. This operation requires sophisticated address calculation and memory management to ensure that tree modifications do not corrupt other parts of the computation.

4.2.6 Hint Operation Implementation

The hint operation (`*[a 11 [b c] d]`) provides optimization opportunities by transforming to `*[[*[a c] *[a d]] 0 3]`. While semantically equivalent to its expansion, hints in the NockPU architecture are designed to enable hardware-specific optimizations or jetting.

The current implementation constructs the hint structure in memory and proceeds with standard evaluation. However, the architecture includes provisions for recognizing specific hint patterns that could be accelerated through specialized hardware modules or optimized execution paths, representing a key area for future performance improvements.

4.3 Memory Management

4.3.1 Heap Allocation Strategy

The NockPU employs a heap-based memory allocation strategy. A free memory chain links together all available memory cells, and operations claim cells from this chain as needed. The free memory pointer (`F`) tracks the next available cell.

This approach allows for dynamic allocation without requiring complex memory management hardware, albeit at the cost of potential fragmentation over time.

4.3.2 Memory Access Patterns

The memory architecture is designed to optimize the prevalent access patterns in Nock execution:

1. **Cell-Based Memory Model**: Each memory word represents a complete cell containing both head and tail components within the 64-bit word structure. Single atoms are represented as cells `[a NIL]`, where NIL is the maximum direct atom value (28-bits).

2. **Parallel Memory Operations**: The design allows up to three memory operations per processor cycle: two reads and one write, enabling efficient traversal and manipulation of tree structures.

3. **Memory Management Unit (MMU)**: The NockPU is shielded from off-chip memory complexity through an MMU that handles the translation between the processor's cell-based addressing and the underlying memory hierarchy.

5 Evaluation and Analysis

5.1 Test Methodology

5.1.1 Test Bench Design

A comprehensive test bench was developed to evaluate the NockPU's functionality and performance. The test bench allows for loading different Nock programs into memory, executing them, and analyzing the results.

The testing framework includes:

- Memory initialization from hex files
- Execution control and timing
- Result validation
- Performance measurement through cycle counting

5.1.2 Operation Verification

Each Nock operation was verified through specific test cases designed to exercise different aspects of its functionality. Waveform analysis allowed for detailed examination of the processor's behavior during execution, ensuring correctness and identifying potential optimization opportunities.

5.2 Performance Metrics

5.2.1 Execution Time Analysis

Performance analysis was conducted using a variety of benchmark programs, including:

1. **Decrement Operation**: A classic test case that requires recursive application of Nock operators to perform what would be a single instruction on a traditional CPU.
2. **Fibonacci Calculation**: Tests recursive function evaluation performance.
3. **Equality Testing**: Evaluates the performance of tree comparison operations.

Results show that simple operations complete in hundreds of cycles, while more complex operations requiring extensive tree manipulation can take thousands to millions of cycles.

5.2.2 Memory Usage Patterns

Memory usage analysis reveals a clear pattern: Nock programs quickly consume memory due to the graph reduction approach.

Even simple operations like decrement create multiple intermediate structures.

For example, decrementing the number 3 created approximately 640 milliseconds worth of memory operations and used a significant portion of available memory, demonstrating the memory-intensive nature of combinator reduction.

5.2.3 Operation Complexity

The complexity of different operations varies significantly:

- **Simple Operations** (constants, slot access): O(log n) in tree depth
- **Arithmetic Operations** (increment): O(1) when directly implemented
- **Recursive Operations** (decrement, equality): O(n) to O(2^n) depending on input size
- **Conditional Operations**: Particularly expensive due to tree reconstruction

5.3 Limitations and Challenges

5.3.1 Memory Consumption Issues

The most significant limitation encountered was excessive memory consumption. Without garbage collection, even moderately complex programs quickly exhaust available memory due to the creation of intermediate structures during reduction.

For instance, a decrement operation on the number 10 was found to crash the system due to memory exhaustion, highlighting the critical need for memory management.

5.3.2 Performance Bottlenecks

Several performance bottlenecks were identified:

1. **Memory Access Latency**: Particularly when using off-chip memory, which is significantly slower than on-chip memory.

2. **Sequential Reduction**: The inherently sequential nature of certain reduction patterns limits parallelism.
3. **Tree Traversal Overhead**: The need to navigate complex tree structures introduces overhead compared to direct operations.

5.3.3 Conditional Operation Overhead

The implementation of the conditional operation (`*[a 6 b c d]`) proved particularly inefficient due to its expression as a complex tree transformation rather than a simple branch. This approach, while semantically pure, introduces significant overhead compared to traditional conditional execution.

6 Future Work

6.1 Arbitrary-size Atom Support

While the current implementation supports 28-bit atoms, supporting arbitrarily large atoms is necessary for full Nock compatibility. A proposed approach involves using a linked representation where multiple words store segments of a large atom, with special tag bits indicating continuation.

This enhancement would require modifications to all operations that manipulate atoms to handle the multi-word representation.

6.2 Hardware Jetting Strategy

”Jetting” refers to the replacement of inefficient Nock code patterns with optimized implementations. A promising approach for hardware jetting involves using a secondary processor that:

1. Detects specific patterns in the Nock code
2. Executes optimized hardware implementations
3. Returns results to the main NockPU

This hybrid approach could dramatically improve performance for common operations while maintaining semantic equivalence with pure Nock.

6.3 ASIC Implementation Considerations

While the current implementation targets FPGAs, future work could explore an Application-Specific Integrated Circuit (ASIC) implementation. This would offer potential advantages in power efficiency and performance but would require significant additional design work and manufacturing costs.

6.4 Power Efficiency Potential

Preliminary analysis suggests that a dedicated NockPU could offer significant power efficiency advantages over general-purpose CPUs running Nock interpreters. Future work should include comprehensive power analysis and optimization of the architecture for low-power operation.

7 Discussion and Implications

7.1 Theoretical Implications for Computer Architecture

The NockPU project raises interesting questions about the relationship between computation models and hardware architecture. Traditional von Neumann architectures are optimized for imperative, state-based computation, while the NockPU demonstrates an alternative approach optimized for functional, stateless computation.

This exploration suggests that there may be benefits to developing specialized architectures for different computational paradigms, rather than forcing all computation onto a single architectural model.

7.2 Practical Applications

7.2.1 Low-power Computing

The deterministic nature of Nock, combined with the potential power efficiency of specialized hardware, suggests applications in low-power computing environments where predictability is valued over raw performance.

7.2.2 Verifiable Computing

The simplicity and determinism of the NockPU architecture could make it easier to formally verify than complex general-purpose CPUs, opening possibilities for applications requiring high assurance of correctness.

7.3 Comparison with Traditional Architectures

When compared to traditional CPU architectures, the NockPU reveals fundamental tradeoffs:

1. **Performance vs. Simplicity**: The NockPU sacrifices raw performance for semantic clarity and simplicity.

2. **Flexibility vs. Specialization**: General-purpose CPUs offer flexibility across computing paradigms, while the NockPU is specialized for combinator reduction.

3. **Memory Efficiency vs. Semantic Purity**: The NockPU's approach to computation requires more memory operations than optimized imperative code.

These tradeoffs suggest that the NockPU and similar architectures may find their niche in specialized applications rather than general-purpose computing.

8 Conclusion

The NockPU project has demonstrated the feasibility of implementing the Nock combinator calculus directly in hardware, while highlighting both the challenges and opportunities presented by such an approach. The stackless traversal mechanism and specialized memory representation provide a foundation for efficient execution of combinator operations, while the challenges of memory consumption and performance bottlenecks point to areas for future improvement.

This work contributes to the broader understanding of hardware-based approaches to functional computation and offers specific insights into the implementation of combinator reduction systems. While a hardware Nock implementation may not compete with optimized software interpreters for general-purpose computing, it offers advantages in determinism, potential power efficiency, and semantic clarity.

Future work building on this foundation has the potential to address the identified limitations and further explore the unique capabilities of specialized hardware for functional computation paradigms.

References

Burrows, E. (2009). "A Combinator Processor." In: *Part II Computer Science Tripos.*

Clark, Douglas W. (1976). "An Efficient List-Moving Algorithm Using Constant Workspace." In: *Communications of the ACM* 19.6, pp. 352–354. DOI: 10.1145/360238.360249.

Clarke, T. J. W. et al. (1980). "SKIM—The S, K, I Reduction Machine." In: *Proceedings of the 1980 ACM Conference on LISP and Functional Programming.* New York, NY, USA: ACM, pp. 128–135. DOI: 10.1145/800087.802799.

Naylor, Matthew (2009). "Hardware-Assisted and Target-Directed Evaluation of Functional Programs." PhD thesis. York, UK: University of York.

Naylor, Matthew and Colin Runciman (2008). "The Reduceron: Widening the von Neumann Bottleneck for Graph Reduction Using an FPGA." In: *Implementation and Application of Functional Languages.* Vol. 5083. Lecture Notes in Computer Science. Springer, pp. 129–146.

Norman, A. C., T. J. W. Clarke, and W. R. Stoye (1984). "Some Practical Methods for Rapid Combinator Reduction." In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. New York, NY, USA: ACM, pp. 159–166. DOI: 10.1145/800055.802036.

~sorreg-namtyv, Curtis Yarvin (2013) "Nock 4K". URL: https://docs.urbit.org/language/nock/reference/definition (visited on ~2024.2.20).

~sorreg-namtyv, Curtis Yarvin et al. (2016). *Urbit: A Solid-State Interpreter*. Whitepaper. Tlon Corporation. URL: https://media.urbit.org/whitepaper.pdf (visited on ~2024.1.25).