FLUID

A New Way to Think About Computing

An Introduction to Token-Based Dataflow Architecture

Hannes Lehmann
sistemica GmbH

Version 1.0 — February 2026

Part of a Research Initiative at the
Center for Applied Complexity & Intelligence

Contents

Part I

The Big Picture

For everyone curious about the future of computing

1. Why Do We Need a New Kind of Computer?

Imagine a factory where every worker must ask permission from a single foreman before doing anything. Worker A finishes their task and sets down their output. Before Worker B can pick it up and continue, they have to wait for the foreman to notice that A is done, walk over, check the paperwork, and give explicit instructions to B. Meanwhile, Workers C through Z stand idle at their stations, even though they have everything they need to work and their tasks don't depend on what A and B are doing.

This is how your computer works today. The architecture of modern computers was designed in 1945 by John von Neumann, at a time when memory was extraordinarily expensive, transistors didn't exist yet, and the idea of parallel processing was pure science fiction. The solution von Neumann proposed was elegant for its time: fetch one instruction from memory, execute it, fetch the next instruction, execute it, and repeat forever. This sequential model made sense when building a computer meant wiring together thousands of vacuum tubes by hand.

Eighty years later, we still use this basic design. We've made it dramatically faster through billions of transistors, multiple processor cores, and clock speeds measured in gigahertz. We've added caches, branch predictors, speculative execution, and countless other optimizations. But the fundamental bottleneck remains: a single stream of instructions flowing through a central processor, one after another, with elaborate machinery trying to squeeze more work out of each cycle.

This is the von Neumann bottleneck, and its consequences touch every aspect of modern computing. Your phone gets hot when running AI models because the processor is working furiously to push data through a pipeline designed for sequential math, not the massively parallel operations that neural networks require. Security vulnerabilities like Spectre and Meltdown exist because processors try to guess what instruction comes next and execute it speculatively, creating side channels that attackers can exploit. Parallel programming remains notoriously difficult because programmers must explicitly manage threads, locks, and synchronization to wring parallelism from hardware that fundamentally wants to do one thing at a time. Power consumption keeps climbing despite efficiency improvements because the instruction fetch-decode-execute cycle itself consumes energy regardless of what computation actually needs to happen.

These problems are well understood, and they are not new. John Backus identified the von Neumann bottleneck by name in his 1977 Turing Award lecture. Since the 1970s, researchers have pursued fundamentally different architectures. Dataflow machines at MIT and Manchester built processors where computations fired when their inputs arrived, eliminating the program counter entirely. The Inmos transputer in the 1980s put message-passing communication directly into silicon, with each chip wired to four neighbors and programmed in occam, a language built on Tony Hoare's Communicating Sequential Processes. Thinking Machines' Connection Machine packed 65,536 simple processors into a single system, betting that massive parallelism could overcome the limitations of individual cores. More recently, neuromorphic chips like Intel's Loihi and IBM's TrueNorth emulate biological neural structures, processing information through spikes rather than clock-driven instruction streams.

Some of these approaches succeeded spectacularly within specific domains. GPUs repurposed parallel graphics pipelines to become the dominant platform for machine learning. Google's Tensor Processing Unit, built around systolic arrays, powers much of the world's AI inference.

Neuromorphic hardware shows promise for edge computing where power budgets are measured in milliwatts. Analog computing, once considered obsolete, is experiencing a renaissance for workloads where approximate, energy-efficient computation outperforms digital precision.

Yet no alternative has displaced the von Neumann model for general-purpose computing. The dataflow machines of the 1980s demonstrated that data-driven execution worked but struggled with compiler technology, the cost of token-matching hardware, and an industry that was pouring investment into making sequential processors faster. The transputer arrived ahead of its time, before the software ecosystem could exploit hardware-level message passing. Neuromorphic chips excel at inference tasks but do not offer a general programming model. Each approach solved part of the problem — parallelism, efficiency, or domain-specific performance — but none addressed the full challenge: an architecture that makes parallelism natural, security intrinsic, and general-purpose computing practical, all from the same set of design principles.

2. What If We Removed the Foreman?

FLUID, which stands for *Functional Logic Unit for Integrated Dataflow*, builds on the core insight that dataflow architects have pursued for five decades: remove the foreman entirely. Return to our factory analogy, but now imagine each worker at their station knows exactly what to do when parts arrive. No foreman walks the floor issuing commands. Instead, each part carries a label specifying its next destination, and workers simply do their job when materials show up. Worker A finishes and places the output on a conveyor. The part's label says "go to Worker B, input slot 2." Worker B receives it, and when all required inputs are present, B works. Meanwhile Workers C through Z are already busy with their own parts, because nobody had to wait for permission.

In FLUID, the workers are called Processing Elements, or PEs. Each PE is a small hardware unit dedicated to one specific task: an arithmetic PE adds and subtracts, a memory PE reads and writes storage, a branch PE routes data based on conditions. The parts moving between workers are called tokens. A token is a small self-describing packet that carries its data, knows its destination, and includes security credentials governing what memory it may access. When tokens arrive at a PE with all required inputs present, the PE fires automatically, performs its operation, and emits results as new tokens that flow onward.

Think of it like a modern automated warehouse. Packages arrive with barcodes that specify their destination, and conveyor belts route them automatically to the right station. At each station, a specialized machine scans, sorts, packs, or labels without waiting for a central controller to issue commands. When one machine finishes processing a package, the conveyor belt immediately carries the result to the next station. Everything happens in parallel because the packages themselves carry the information needed to route and process them.

In FLUID, computation works the same way. When one Processing Element finishes its work, its output token automatically flows to the next element that needs it. If a downstream element isn't ready yet, the token waits in a queue without blocking anyone else. Multiple independent computations proceed simultaneously because they don't share state and don't need permission from a central authority. The system exploits parallelism naturally, without the programmer writing threading code or worrying about race conditions.

This isn't a minor optimization on existing computer architecture, nor is it without precedent. The dataflow machines of the 1970s and 1980s demonstrated that data-driven execution is sound. What FLUID adds to this tradition is a specific synthesis: self-describing tokens that carry security credentials alongside routing and data, heterogeneous Processing Elements that can incorporate fundamentally different computational substrates, and a design informed by fifty years of lessons about what made earlier alternatives succeed in principle but struggle in practice.

2.1. A Flexible Foundation for Future Computing

The Processing Element model offers a crucial advantage that becomes clear when thinking about the future of computing: extensibility. Adding a new kind of computation to FLUID means designing a new PE type that consumes and produces tokens according to the same protocol as every other element. The router doesn't care what happens inside a PE; it only knows how to deliver tokens to destinations and collect outputs. This separation means that FLUID can incorporate radically different computational substrates without changing its fundamental architecture.

Consider the emerging landscape of hybrid computing. Analog computing is experiencing a renaissance for certain workloads, particularly neural network inference, where approximate calculations in the analog domain can be orders of magnitude more energy-efficient than digital equivalents. Optical computing promises interconnects and certain operations at the speed of light. Quantum computing offers exponential speedups for specific problem classes. Each of these technologies has unique strengths, but integrating them into traditional von Neumann systems requires awkward bridges between fundamentally different execution models.

FLUID's token-based approach provides a natural integration point. An analog PE could accept tokens specifying matrix operations and return approximate results. An optical PE could handle certain communication patterns. A quantum PE could process tokens representing quantum circuits and return measurement outcomes. From the perspective of the rest of the system, these are simply Processing Elements that take time to produce outputs. The dataflow model already handles variable latencies gracefully through its queue-based flow control.

This flexibility extends to customizing processors for specific applications. Rather than buying a general-purpose chip and hoping it handles your workload efficiently, FLUID enables designing a processor around your software. Need heavy floating-point throughput? Add more FPU elements. Memory-bound workload? Configure more memory ports. Unusual operations that would require library calls on traditional hardware? Design a specialized PE that handles them directly.

This approach echoes what made RISC-V successful in the processor IP market. RISC-V didn't try to define one perfect instruction set; it defined a minimal core with a clear extension mechanism, allowing implementers to add exactly the capabilities they need. FLUID takes this philosophy further by making the computational elements themselves modular. Just as RISC-V vendors can license or design custom instruction extensions, FLUID system designers can compose exactly the Processing Elements their application requires. The difference is that FLUID's modularity operates at a coarser grain, making it easier to incorporate fundamentally different technologies, while RISC-V extensions must still fit within the sequential instruction execution model.

There may be a deeper connection between FLUID and analog computing worth investigating. In traditional analog computers, engineers physically wire together operational amplifiers, integrators, and multipliers on a patch panel. Each wire represents a data path, and each component performs a specific mathematical operation. The resulting circuit *is* the program, with no separate instruction stream to fetch or decode. The computation emerges directly from how components are connected.

This bears striking resemblance to FLUID's model, where Processing Elements connected by token flows define the computation through their graph topology. A FLUID program is, in essence, an abstract patch panel. This correspondence suggests that FLUID might serve as a programming model that spans computational substrates, with a single graph specifying that certain nodes execute as digital PEs while others map to physical analog components.

If this vision proves practical, the implications would be significant. The FLUID toolchain could potentially generate patch configurations for analog hardware alongside digital PE instantiation, or even drive electronically-reconfigurable analog systems directly. A programmer would work with one graph, one set of tools, and one mental model, regardless of whether the underlying execution happens in digital logic, analog circuits, or some combination of both. Whether the practical challenges of timing, calibration, and noise make this feasible remains an open question that motivates our research program.

3. A Simple Example: Adding Two Numbers

To understand how FLUID differs from traditional computing, consider the simplest possible computation: adding two numbers. On a conventional RISC-V processor, even this trivial operation involves a sequence of instructions that must be fetched, decoded, and executed one by one:

```
Traditional RISC-V Assembly: Adding 5 + 3
──────────────────────────────────────────

li   a0, 5       # Load immediate 5 into register a0
                 #   → fetch instruction from memory
                 #   → decode opcode and operands
                 #   → write 5 to register file

li   a1, 3       # Load immediate 3 into register a1
                 #   → fetch instruction from memory
                 #   → decode opcode and operands
                 #   → write 3 to register file

add  a2, a0, a1  # Add a0 + a1, store result in a2
                 #   → fetch instruction from memory
                 #   → decode opcode, identify source/dest registers
                 #   → read a0 and a1 from register file
                 #   → perform addition in ALU
                 #   → write result to a2 in register file

# Result: a2 = 8

Total: 3 instructions, each requiring fetch-decode-execute
Plus: program counter updates, pipeline management,
      potential stalls, branch prediction overhead
```

Listing 1: Traditional sequential execution on RISC-V

Each instruction must pass through the entire pipeline: fetch from memory, decode the opcode and operands, read from registers, execute the operation, and write back results. A program counter tracks which instruction comes next. The processor speculatively predicts branches and may need to flush work when predictions fail. Even for this trivial computation, the machinery of sequential execution imposes overhead.

In FLUID, the same computation looks fundamentally different:

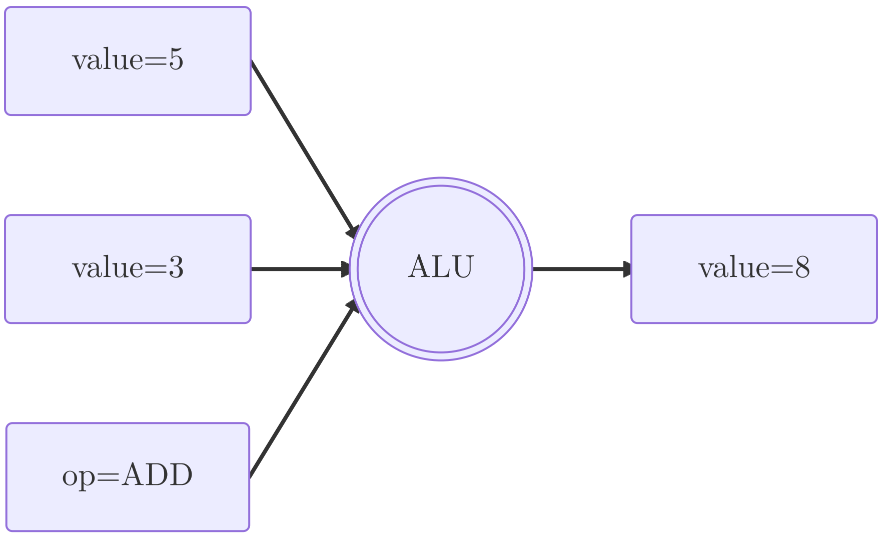

Figure 1: Three tokens arrive at an ALU and produce a result

Three tokens arrive at an arithmetic unit: one carrying the value 5, one carrying the value 3, and one specifying the ADD operation. The ALU doesn't need to be told when to fire; the presence of all three inputs triggers computation automatically. The result, a new token carrying the value 8, flows to wherever the ALU is configured to send its output.

There's no instruction fetch because the operation code arrived as a token. There's no register file because values travel directly between processing elements. There's no program counter because data dependencies determine execution order. The computation is the communication.

This example comes directly from the FLUID emulator. The JSON program that implements it defines two Processing Elements (the ALU and a Sink to collect output), wires them together, and specifies three initial tokens. When the emulator runs this program, it completes in exactly two cycles: one to route the tokens to the ALU, one to perform the computation and route the result to the Sink. The output is a token with value 8.

4. What Makes FLUID Different?

4.1. Computation Becomes Communication

In traditional computers, computation and communication are separate concerns that the programmer must carefully coordinate. You load data from memory into registers, perform calculations, store results back to memory, and hope that the cache hierarchy makes this tolerable. Each step involves explicit instructions, and the processor spends significant time and energy just moving data around rather than doing useful work.

FLUID inverts this model by making the token itself both the data and the trigger for computation. When a token arrives at an arithmetic unit, its very presence causes the computation to happen. The result becomes a new token that automatically routes to the next destination based on information embedded in the token itself. There's no instruction fetch cycle, no decode stage, no program counter ticking through memory addresses. The flow of data is the program.

This approach eliminates overhead that traditional processors treat as unavoidable. More importantly, it makes the structure of computation explicit and visible. Instead of reasoning about what a sequence of instructions does to hidden state, you can see directly how data flows through a system of processing elements.

4.2. Security Lives in the Hardware

Every FLUID token carries a capability, which is essentially a permission slip specifying exactly what memory regions the token can access and what operations it can perform there. The design intent is that this becomes a hardware-enforced constraint rather than a software construct that clever attackers might bypass.

When a token tries to read from or write to memory, the Memory Processing Element examines the capability embedded in that token before anything happens. If the access falls within the bounds specified by the capability and the operation is permitted by the capability's rights, the access proceeds. If not, the hardware generates an error token and the invalid access never occurs. Capabilities can only be created by trusted hardware, and the architecture enforces that capabilities can only be narrowed, never widened.

This design aims to eliminate entire categories of security vulnerabilities at the architectural level. Buffer overflows should not be possible because every memory access is bounds-checked against the capability. Use-after-free bugs should not be exploitable because capabilities can be revoked when memory is freed. Studies of major software projects at Microsoft and Google have found that approximately 70% of security vulnerabilities stem from memory safety issues. Whether FLUID achieves its security goals in practice requires validation beyond the emulator, including formal verification of the security model and testing against real attack patterns. The architecture provides the foundation; proving its effectiveness is ongoing research.

4.3. Parallelism Emerges Naturally

Writing parallel code on traditional systems requires the programmer to explicitly create threads, protect shared data with locks, and reason carefully about what might happen when operations interleave. This is difficult because you're fighting against the architecture. The hardware wants to execute instructions sequentially, and parallelism is an afterthought that the programmer must carefully orchestrate.

In FLUID, parallelism is the natural state of affairs. If two computations do not depend on each other's results, they can run at the same time. The programmer describes what needs to happen and how data flows through the system; the hardware determines when things can run in parallel. There are no threads to create, no locks to acquire, and the architecture aims to make race conditions structurally impossible by eliminating shared mutable state.

This doesn't mean that all programs become magically parallel. Sequential dependencies still exist when one computation truly needs the result of another. But the parallelism that does exist in a problem is exploited automatically, without programmer effort and without the bugs that plague traditional parallel programming.

5. Architectural Lineage

FLUID does not emerge from a vacuum. The idea that data dependencies rather than a program counter should drive execution dates to the early 1970s, when Jack Dennis at MIT proposed the first dataflow schemas and architectures [1]. Dennis's static dataflow model established the core principle: a computation fires when all its inputs are available, with no central controller deciding execution order. The concept was significantly extended by the Manchester Dataflow Machine [2] and MIT's Tagged-Token Dataflow Architecture [3], which introduced dynamic dataflow with tagged tokens. In these systems, tokens carry tags that identify which invocation of a computation they belong to, enabling multiple instances of the same subgraph to execute concurrently. This was a crucial advance: static dataflow could only have one set of tokens per graph arc at a time, severely limiting parallelism for loops and recursive computations.

FLUID inherits this dynamic, tagged-token lineage. Like the MIT TTDA, FLUID tokens carry metadata that enables concurrent execution of multiple computation instances. But FLUID diverges from its predecessors in several important ways. The 256-bit token format embeds capability-based security credentials alongside routing and data, making memory protection intrinsic to the execution model rather than an external concern. The system uses heterogeneous Processing Elements — 17 distinct types at present — rather than the relatively uniform processing nodes of earlier dataflow machines. And the crossbar router with bounded queues and backpressure provides flow control without the token-matching stores that were a performance bottleneck in earlier dynamic dataflow systems. For a comprehensive survey of the dataflow architecture lineage, see Veen [4] and Johnston et al. [5].

A different branch of parallel architecture also deserves mention, because it shares surface similarities with FLUID while differing fundamentally. Systolic arrays, introduced by H. T. Kung and Leiserson [6] and elaborated in Kung's influential "Why Systolic Architectures?" [7], are regular grids of identical processing elements through which data flows rhythmically, like blood pulsing through the circulatory system — the term *systole* referring to the heart's contraction phase. Each element performs the same fixed operation, typically multiply-accumulate, and passes results only to its immediate neighbors. Systolic arrays achieve remarkable efficiency for specific workloads: Google's Tensor Processing Unit uses a systolic array at its core for neural network matrix multiplication.

The resemblance to FLUID is real but limited. Both architectures distribute computation across arrays of processing elements and move data between them without a central instruction stream. Both reject the von Neumann bottleneck. But systolic arrays are homogeneous where FLUID is heterogeneous, fixed-topology where FLUID uses programmable routing, and single-purpose where FLUID aims for generality. A systolic array is a fixed conveyor belt optimized for one product; FLUID is a programmable factory that can be reconfigured for different workloads. Put differently, a systolic array could be expressed as a restricted special case within FLUID — a grid of identical ALU PEs with nearest-neighbor-only routing — but the converse is not true. FLUID's generality comes at a cost in routing overhead that purpose-built systolic arrays avoid, a classic tradeoff between flexibility and efficiency that FPGA implementation will help quantify.

Property	**Systolic Array**	**Classic Dataflow**	**FLUID**
Processing Elements	Homogeneous (identical)	Relatively uniform	Heterogeneous (17 types)
Topology	Fixed regular grid	Static graph	Programmable crossbar
Routing	Nearest-neighbor only	Fixed connections	Self-routing tokens
Operations	Single fixed (e.g. MAC)	General arithmetic	General + memory, control, I/O, security
Security	None	None	Inline capabilities per token
Data model	Streams pulse through grid	Tagged tokens	256-bit self-describing tokens
Scope	Single workload (e.g. matrix multiply)	General-purpose	General-purpose + heterogeneous substrates

Table 1: Comparison of parallel architectures

6. A More Complex Example: Chained Computation

Real programs involve more than single operations. Consider computing $(5 + 10) + 3$, where the second addition depends on the result of the first. In FLUID, this creates a chain of processing elements where data flows from one to the next:

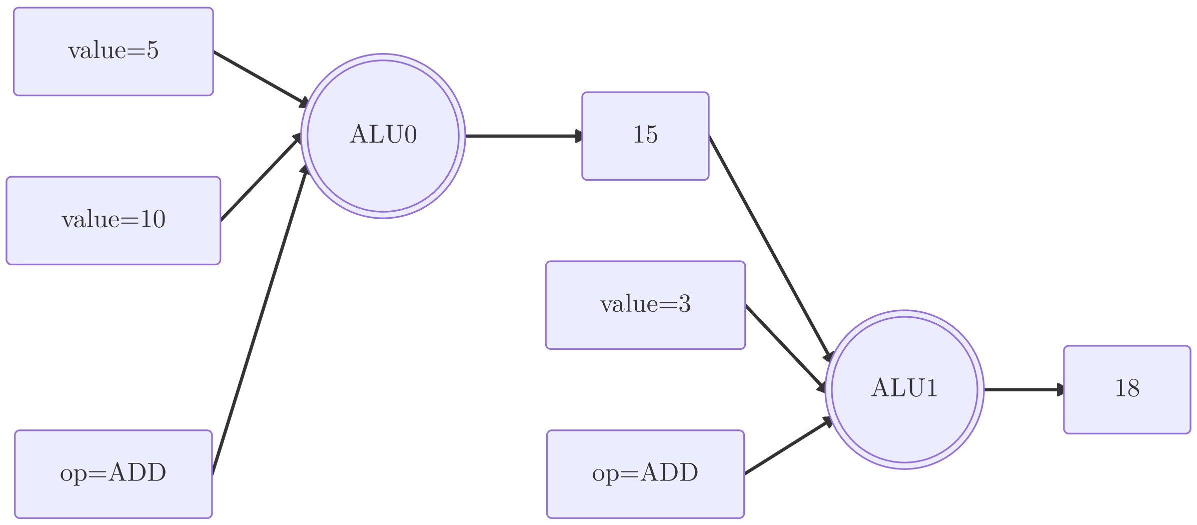

Figure 2: Two ALUs chained together: (5+10)+3=18

The first ALU receives three tokens (5, 10, and ADD), computes 15, and sends it to the second ALU. Meanwhile, the second ALU has already received its other inputs (3 and ADD) and is waiting. The moment the 15 arrives, computation fires and produces 18. The emulator executes this in three cycles.

What makes this interesting is what happens with independent computations. If we had two separate additions that didn't depend on each other, they would execute simultaneously in the same cycle. The programmer doesn't need to create threads or worry about synchronization; independence is implicit in the data flow graph, and the system exploits it automatically.

7. Who Is This For?

Where might FLUID find its place? The honest answer is that we don't yet know how far this architecture can reach. The vision is ambitious: a computing model that could eventually serve as a general-purpose foundation, not merely an accelerator for niche workloads. Whether FLUID can achieve true general-purpose computing remains an open research question. The emulator demonstrates that the model works; hardware implementation will reveal whether it performs as the architecture suggests it should.

That said, certain domains appear particularly well-suited to FLUID's characteristics, and these represent likely early adoption paths even if the broader vision takes longer to realize.

Embedded and safety-critical systems represent one such domain. Medical devices that must never crash, industrial controllers that require predictable behavior, and automotive systems where security failures could cost lives all need guarantees that traditional architectures struggle to provide. FLUID's execution model is deterministic in the emulator: the same inputs to the same program graph produce the same outputs in the same number of cycles. If this property holds in hardware, it would simplify certification and testing significantly. Whether this determinism translates to real-time guarantees is an open question. A single program running on dedicated PEs has predictable cycle counts, but sharing PEs between multiple programs introduces scheduling decisions that affect timing. The capability-based security model should prevent entire classes of exploits by architectural construction, though this remains to be validated outside simulation.

Artificial intelligence workloads represent another natural fit, at least in principle. Neural networks are fundamentally parallel structures consisting of millions of simple operations that can happen simultaneously with well-defined dependencies. Running them on sequential processors requires elaborate batching, tiling, and memory management to keep the hardware fed with data. FLUID's dataflow model matches how neural networks are actually structured. Whether this architectural alignment translates to better efficiency in practice requires benchmarks that don't yet exist.

Edge computing devices present similar opportunities. IoT sensors that need to process data locally before sending results upstream would benefit from low power consumption and robust security. The architectural argument is sound: eliminating instruction fetch overhead should save energy, and capability-based isolation should prevent compromised devices from attacking the rest of a network. These are reasonable projections from the architecture, not measured results.

We state these potential applications with appropriate humility. The emulator validates that the computational model is sound and that programs execute correctly. The security properties hold in simulation. But claims about energy efficiency, performance advantages, and practical suitability for specific domains remain hypotheses until hardware exists and benchmarks can be run. This paper describes an architecture and its implementation in software; proving its value requires the next phase of the research program.

8. The Vision

FLUID is part of a larger research initiative at sistemica called the Center for Applied Complexity and Intelligence. The thesis underlying this work is that computation, like intelligence more broadly, should emerge from the interaction of simple components rather than being forced through a central bottleneck. This principle applies not just to processor design but to operating systems, programming models, and how we think about complex systems generally.

FLUID embodies this philosophy by distributing computation across independent processing elements that communicate through messages. There's no central processor making decisions; behavior emerges from the flow of tokens through the system. This architectural approach enables properties that are difficult or impossible to achieve when everything funnels through a single point of control.

We're not building a processor in isolation. FLUID is being co-designed with AionCore, a capability-based microkernel that shares the same philosophical foundation. The eventual goal is a kernel that communicates through tokens with embedded capabilities, matching the hardware's native language with no translation layers or legacy interfaces. This tight integration between hardware and system software should enable optimizations and guarantees that aren't possible when hardware and software are designed separately and bridged through decades-old abstractions.

AionCore currently runs on conventional amd64 hardware, which might seem contradictory but serves a deliberate purpose. The kernel implements message passing between processes treated as actors, with capability-based security enforced at the system call boundary. Even on traditional hardware, this design exercises the programming model and system concepts that will eventually run natively on FLUID. Developing a microkernel is a substantial undertaking; doing so on mature, well-tooled hardware lets us focus on getting the abstractions right before the additional complexity of novel hardware enters the picture. AionCore on amd64 is a research vehicle, a way to explore capability-based kernel design and validate our ideas about process isolation and inter-process communication. The lessons learned will directly inform the FLUID-native kernel that follows.

This co-design philosophy extends, at least in our aspirations, to how FLUID might integrate with emerging computing technologies. We envision a future where analog, photonic, or neuromorphic substrates are exposed through the PE abstraction, allowing the kernel to schedule computations across heterogeneous resources without translation layers. Whether this vision is achievable, and what obstacles arise in practice, are core questions driving our research.

If capabilities can govern access to analog channels as naturally as they govern memory regions, and if process isolation can extend to accelerator resources, then the traditional distinction between processor and device may dissolve. This is speculative, but it suggests a research direction worth pursuing: not merely an operating system that supports accelerators, but one where diverse computational substrates are managed, scheduled, and protected through unified primitives.

Traditional systems treat GPUs, TPUs, and analog accelerators as I/O devices, with all the complexity that entails. Memory must be copied across protection boundaries, synchronization happens through explicit fences, and errors propagate through opaque status codes. We hypothesize that FLUID's token-based model could avoid much of this complexity, but proving this hypothesis requires the research program we are undertaking.

The goal is a computing platform that is parallel by default, secure by construction, and energy-efficient by design. These properties should emerge from architectural choices rather than being bolted on afterward. Whether we achieve this goal, and how far the architecture can scale toward general-purpose computing, remains to be demonstrated. What we have today is a working emulator, a developing kernel, and a clear research path forward.

Part II

Under the Hood

For engineers, architects, and the technically curious

9. The Token: Self-Describing Computation

At the heart of FLUID lies the token, a 256-bit message that carries everything needed to route, process, and validate a piece of computation. Unlike traditional architectures where instructions and data are separate concerns stored in different places and managed through different mechanisms, a FLUID token is self-contained. It specifies its destination, carries its payload, and includes the security credentials needed to access memory.

```
┌───────────────────────────────────────────────────────────────────────┐
│                          256-bit FLUID Token                          │
├──────────────┬──────────────────┬──────────────┬──────────────────────┤
│   ROUTING    │     METADATA     │     DATA     │      CAPABILITY      │
│   32 bits    │     64 bits      │   64 bits    │       96 bits        │
├──────────────┼──────────────────┼──────────────┼──────────────────────┤
│ destination  │ request_id (8b)  │              │ cap_tag (8b)         │
│   (16b)      │ flags (8b)       │    value     │ cap_rights (8b)      │
│ port (8b)    │ sequence_id (16b)│   (64-bit)   │ cap_base (32b)       │
│ type_tag (8b)│ reserved (32b)   │              │ cap_bound (32b)      │
└──────────────┴──────────────────┴──────────────┴──────────────────────┘
```

Listing 2: 256-bit FLUID Token Structure

The choice of 256 bits is deliberate. At 32 bytes, exactly two tokens fit in a standard 64-byte cache line with no wasted space. This alignment means that token movement through the system maps efficiently onto existing memory hierarchies, even though FLUID's execution model is fundamentally different from the sequential programs those hierarchies were designed for. The size is also large enough to carry meaningful data inline while small enough to move efficiently through routing fabric.

A token divides into four logical sections that serve distinct purposes. The routing section specifies where the token should go: which Processing Element receives it and which input port on that element. The metadata section carries information for coordination, including a request identifier that allows responses to be matched with requests and sequence numbers for ordering when order matters. The data section holds the actual payload, typically a 64-bit value representing an integer, floating-point number, pointer, or other primitive. The security section contains the capability that governs what memory accesses this token is permitted to make.

This structure means that when a token arrives at a Processing Element, everything needed to process it is immediately available. The PE doesn't need to fetch additional data from memory to learn what to do or check permissions against some external table. The token itself contains the complete context for its operation.

9.1. The Routing Section

The routing section occupies 32 bits and tells the system where this token should go. The destination field uses 16 bits to identify the target Processing Element, allowing for up to 65,536 distinct elements in a system. The addressing scheme encodes PE type in the upper bits and instance number in the lower bits, so 0x1000 means "ALU, instance 0" while 0x1001 means “ALU, instance 1” and 0x2000 means “Memory PE, instance 0”. This creates a natural namespace that supports heterogeneous systems mixing different kinds of processing elements.

The port field uses 8 bits to specify which input port on the destination PE should receive the token. Many PEs have multiple inputs that serve different purposes, such as an arithmetic unit that needs two operands and an operation code, or a branch unit that needs both a condition to test and data to route based on that condition. The port field directs each token to the right input.

The type tag uses the remaining 8 bits to describe what kind of data the token carries. This enables runtime type checking and allows PEs to handle different data types appropriately. An arithmetic unit can distinguish between signed and unsigned integers, for instance, and perform the correct operation. A memory unit can distinguish between pointers and raw data. This type information travels with the token, eliminating any ambiguity about how to interpret the payload.

9.2. The Metadata Section

The metadata section occupies 64 bits and carries information needed for coordination between Processing Elements. The request identifier is an 8-bit field that enables request-response patterns. When multiple clients access a shared resource, each request carries a unique identifier that the resource echoes in its response, allowing responses to be routed back to the correct requester. This pattern appears throughout FLUID programs, from register file access to memory operations to inter-process communication.

The flags field provides 8 bits for control information: whether this is a control token or a data token, priority hints for scheduling, and whether the sequence information is meaningful for this particular token. The sequence identifier uses 16 bits and serves double duty: for protocols that require ordering, it specifies the position in a sequence; for register access, it encodes which register to read or write.

The remaining 32 bits are reserved for future extensions. Current use includes process identification for multi-process systems, where the upper bits identify which process originated a token, enabling hardware-enforced process isolation.

9.3. The Data Section

The data section is straightforward: 64 bits that hold the token’s payload. The interpretation depends on the type tag in the routing section. For integers, this is simply a 64-bit value. For pointers, it’s a memory address. For floating-point numbers, it’s an IEEE 754 double. For error tokens, it’s an error code that describes what went wrong.

When tokens need to reference data larger than 64 bits, the value holds a pointer to memory where the larger data resides. The capability in the security section governs access to that memory, ensuring that even large data transfers remain protected.

9.4. The Security Section

The security section occupies 96 bits and contains a capability that specifies what memory accesses this token is authorized to make. The capability tag identifies what kind of capability this is. A tag of zero indicates a null capability that provides unrestricted access, used during testing and bootstrap but never in production. A tag of one indicates a memory capability with enforced bounds and permissions.

The rights field specifies what operations are permitted: read, write, execute, or combinations thereof. The base and bound fields define the memory region as 32-bit addresses marking the inclusive lower bound and exclusive upper bound of accessible memory. Any access outside this region or any operation not permitted by the rights will be rejected by hardware.

This inline capability model differs fundamentally from traditional protection mechanisms. There's no page table to consult, no TLB to miss, no permission bits stored separately from the data they protect. The token carries its permissions with it, and those permissions are checked at every access without any lookup overhead.

10. Processing Elements: Hardware Actors

FLUID has no central processor that executes instructions in sequence. Instead, computation happens in Processing Elements, each of which is an independent unit that receives tokens on input ports, performs operations when it has the inputs it needs, and emits result tokens to destinations specified in its configuration. This architectural pattern matches the actor model from computer science, where independent actors communicate through messages and maintain their own private state.

Each PE implements a specific operation or small set of related operations. An arithmetic unit performs addition, subtraction, multiplication, and other mathematical operations. A memory unit handles loads and stores with capability checking. A branch unit routes tokens based on conditions. A fork unit duplicates tokens to multiple destinations. This specialization means that each PE can be optimized for its specific task, and the overall system gains capability by composing simple elements rather than by making a single processor more complex.

The firing rule for a PE determines when it executes. Most PEs wait until all required input ports have received tokens, then perform their operation atomically and emit results. This dataflow-style execution means that computations happen as soon as their inputs are ready, without any central scheduler making decisions about what to run next. The data dependencies implicit in how tokens flow through the system determine the execution order naturally.

10.1. The Processing Element Inventory

The FLUID emulator currently implements 17 different Processing Element types, covering computation, memory access, control flow, coordination, input/output, and security. The ALU handles arithmetic and logical operations with 18 different operations including addition, subtraction, bitwise operations, shifts, and comparisons. The Memory PE provides byte-addressable access to a memory space with capability-based bounds checking on every access.

Control flow elements include the Branch PE for conditional routing and the FSM (Finite State Machine) PE for implementing loops and complex iteration patterns. The FSM PE is particularly powerful: it maintains internal state, emits a configured sequence of tokens for each iteration, and signals completion when done. This enables expressing loop constructs in pure dataflow without any central controller.

Coordination elements handle request-response patterns and token duplication. The Dispatcher PE assigns unique request identifiers to outgoing tokens, enabling multiple concurrent requests to a shared resource. The ResponseRouter PE uses those identifiers to route responses back to the correct requester. The Fork PE duplicates tokens to multiple destinations when a value needs to be used in several places simultaneously.

The Mailbox PE provides actor-model message queuing with bounded FIFO buffers. It enables asynchronous communication patterns where producers and consumers run at different speeds, with configurable policies for handling overflow. The UART PE provides serial console output, enabling programs to print diagnostic messages.

For multi-process operation, the Timer PE generates periodic tick tokens that drive scheduling, the Scheduler PE manages round-robin process switching, and the StampPid PE tags tokens with process identifiers to enforce isolation. The CapMint PE is the privileged component that can create new capabilities; user programs cannot access it directly, ensuring that capabilities cannot be forged.

10.2. The Arithmetic and Logic Unit

The ALU PE illustrates how processing elements work in practice. It has three input ports: port 0 receives operand A, port 1 receives operand B, and port 2 receives an operation code specifying which computation to perform. When tokens arrive at all three ports, the ALU performs the requested operation and emits a result token to its configured output destination.

```
┌─────────────────────────────────────────────┐
│              ALU Processing Element         │
│                                             │
│  Port 0 ──▶ [Operand A]                     │
│                       \                     │
│  Port 1 ──▶ [Operand B] ─\──▶ [COMPUTE] ──▶ Output
│                        ─/                   │
│  Port 2 ──▶ [Operation] /                   │
│                                             │
│  Operations: ADD, SUB, AND, OR, XOR,        │
│              SHL, SHR, SRA, EQ, NE,         │
│              LT, LTU, GE, GEU, DIV,         │
│              DIVU, MOD, MODU                │
└─────────────────────────────────────────────┘
```

Listing 3: ALU PE with three inputs and one output

This three-input design makes the ALU general-purpose without requiring the operation to be hard-coded. A computation that performs different operations at different times simply sends different operation codes to port 2. The same ALU instance can add in one cycle and compare in the next, determined entirely by the tokens it receives.

10.3. Processing Elements as Domain Bridges

We envision that the PE abstraction need not be limited to digital operations. Because the token protocol specifies only what data arrives and where results go, not how computation happens internally, a Processing Element could in principle encapsulate any computational substrate that accepts inputs and produces outputs. The abstraction boundary sits at the token interface, leaving the implementation behind that boundary unconstrained.

Consider what an Analog PE might look like. It could receive tokens specifying parameters for a differential equation, translate these into voltage settings on analog computing hardware, wait for the circuit to settle, sample the result, and emit a token containing the solution. From the perspective of the FLUID router, this PE would simply exhibit higher and variable latency. The queue-based flow control should handle such variability naturally, though this remains to be validated in practice.

A more ambitious possibility emerges from the structural correspondence between FLUID graphs and analog patch panels. If each analog component type, such as integrators, multipliers, summers, and comparators, were represented as a distinct PE type, then a FLUID program graph would directly describe an analog circuit topology. The same graph could potentially execute in multiple modes. In pure emulation mode, all PEs would run as digital approximations in the FLUID emulator, useful for development and debugging. In hybrid execution mode, digital PEs would handle control flow while analog PEs map to physical hardware, combining the strengths of both domains. In analog generation mode, the graph would compile to patch panel wiring instructions or drive an electronically-reconfigurable analog system, removing manual wiring from the programming workflow entirely.

This would mean that programming an analog computer becomes programming a FLUID graph, using the same tooling, debugging approaches, and composition patterns that work for purely digital programs. Whether practical challenges such as timing constraints, calibration requirements, noise sensitivity, and dynamic reconfiguration latency make this feasible is precisely the kind of question our research program aims to explore.

Similar patterns might apply to other computational domains. Photonic PEs could interface with silicon photonic components for operations where optical computing offers advantages. Neuromorphic PEs could bridge to spiking neural network hardware, translating between token streams and spike trains. Each domain would require careful co-design of the PE interface and the underlying hardware, work we hope to pursue through academic collaborations.

10.4. The Memory PE

The Memory PE provides access to a byte-addressable memory space with capability-based protection. When it receives a token requesting a memory operation, it first validates the request against the capability embedded in that token. The validation checks that the target address falls within the capability's bounds and that the operation type is permitted by the capability's rights. Only if these checks pass does the actual memory access occur.

For read operations, the Memory PE emits a token containing the data read from the specified address. For write operations, it performs the write and emits an acknowledgment. If validation fails, it emits an error token describing the violation: bounds exceeded, rights denied, or invalid capability. This error token flows through the system like any other token and can be handled by downstream elements designed to deal with errors.

The validation logic is straightforward but crucial:

```
┌──────────────────────────────────────────────────────────┐
│              Memory Access Validation                    │
│                                                          │
│  1. Check capability tag                                 │
│     └─ tag=0 (NULL): unrestricted access (test only)     │
│     └─ tag=1 (MEM): proceed to bounds check              │
│     └─ other: reject with InvalidCapability error        │
│                                                          │
│  2. Check bounds                                         │
│     └─ if address < cap_base: BoundsViolation            │
│     └─ if address+size > cap_bound: BoundsViolation      │
│                                                          │
│  3. Check rights                                         │
│     └─ READ requires READ bit set                        │
│     └─ WRITE requires WRITE bit set                      │
│     └─ Missing right: RightsDenied error                 │
│                                                          │
│  4. All checks pass: perform memory operation            │
└──────────────────────────────────────────────────────────┘
```

Listing 4: Memory access validation steps

The overhead of capability checking should be minimal because all information needed for validation is present in the token itself. There is no table lookup, no TLB miss, no context switch to kernel mode. The hardware simply examines the capability fields, compares them against the requested operation, and proceeds or rejects. Quantifying this overhead precisely requires hardware implementation and benchmarking, which remains future work.

11. The Router and Execution Model

Tokens flow between Processing Elements through a router that functions as a crossbar with bounded queues. When a PE emits a token, the router examines the token's destination field and places the token in the queue for that destination. The destination PE pulls tokens from its input queues as it becomes ready to process them.

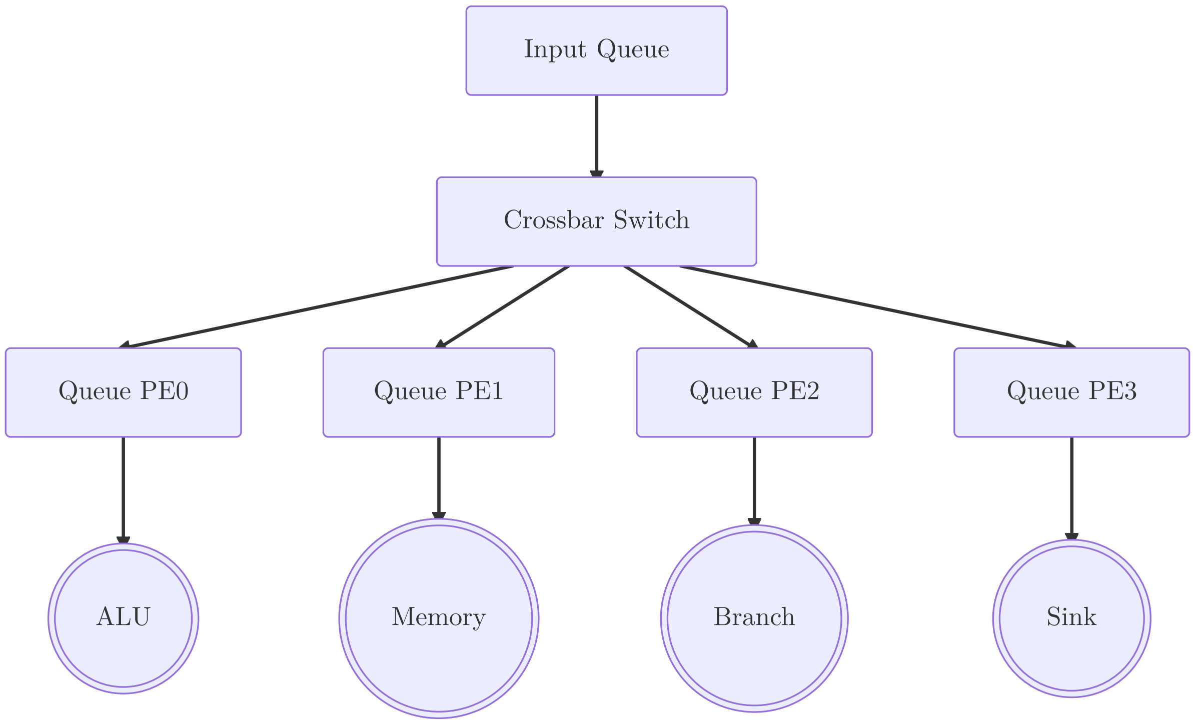

Figure 3: Token router with crossbar switch and per-PE queues

This architecture provides natural flow control through backpressure. Each queue has a finite size, and if a destination's queue fills because that PE is processing slowly or is stalled waiting for other inputs, the router simply cannot accept more tokens for that destination. This backpressure propagates through the system: if a PE can't emit because its output queue is full, it stalls, which eventually causes its input queues to fill, and so on. The system automatically throttles to match the speed of its slowest component without requiring any explicit flow control logic.

Execution proceeds in cycles. Eliminating the program counter does not mean eliminating the clock. What FLUID removes is the fetch-decode-execute cycle that forces instructions through a sequential pipeline. The clock remains because synchronous execution offers practical advantages: simpler hardware implementation, predictable timing for a given program graph, and deterministic behavior that simplifies testing and verification. Each cycle, the router attempts to move tokens from each PE's output to the appropriate destination queues, subject to queue capacity. Then each PE that has all its required inputs fires, consuming its input tokens and producing output tokens. These outputs enter the PE's output queues, ready to be routed in the next cycle. The difference from traditional processors is that the clock no longer sequences through instruction addresses. Instead, it provides a heartbeat for the dataflow fabric, with actual execution order determined by data availability rather than a program counter.

Programs in FLUID are graphs that specify Processing Elements, connections between them, and initial tokens that start computation. The emulator loads a program by instantiating the specified PEs, configuring their output connections, and injecting the initial tokens into the router. Execution continues until no tokens remain in transit and no PE has pending inputs: the computation has completed.

This model is deterministic given the same inputs and the same program graph. There's no scheduler making choices about what to run, no thread interleaving to cause non-determinism, no cache behavior affecting execution order. The same program with the same initial tokens produces the same execution trace every time.

12. Current Implementation

The FLUID emulator exists as a Rust implementation comprising, at the time of writing, more than 19,000 lines of code with around 400 tests validating behavior from individual PE operations through complex multi-PE programs. The emulator is cycle-accurate, meaning it models exactly how tokens flow through the system on each clock cycle, providing a precise reference for future hardware implementations.

12.1. Integration with QEMU

The emulator integrates with QEMU to demonstrate realistic system operation. A RISC-V processor boots firmware that initializes the system, loads a FLUID program graph, and hands off execution to the dataflow substrate. This integration proves that FLUID can function as an accelerator alongside traditional processors, a likely deployment model for early adoption.

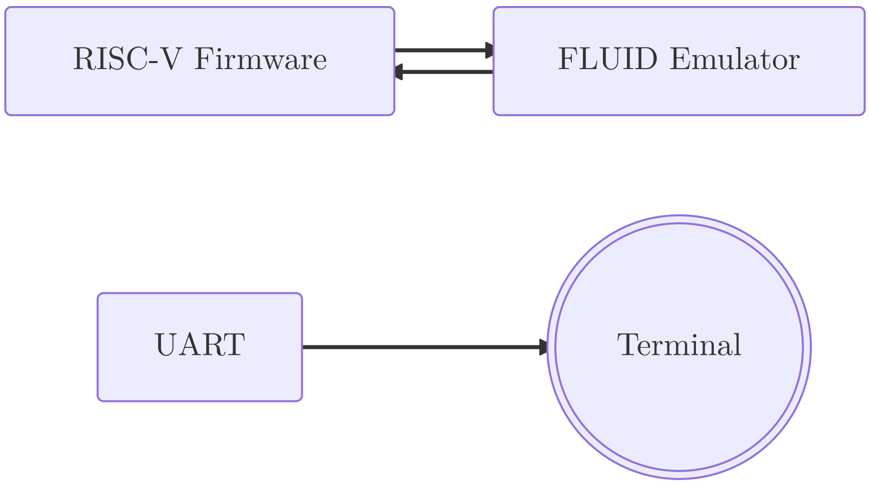

Figure 4: QEMU integration: Firmware communicates via MMIO with FLUID Emulator, UART outputs to Terminal

The firmware communicates with the FLUID emulator through memory-mapped I/O registers. To load a program, it writes the address of a JSON graph description to the GRAPH_ADDR register and its length to GRAPH_LEN, then sets the LOAD_GRAPH bit in the control register. The emulator parses the graph, instantiates PEs, and injects initial tokens. Setting the HANDOFF bit runs the emulator until completion, after which the firmware can read result tokens from the RX FIFO.

Serial output enables programs to print diagnostic messages. The UART PE converts token values to ASCII characters and sends them to a simulated serial port, which QEMU displays on the terminal. This capability proved essential during development for debugging and is available to FLUID programs that need to produce human-readable output.

12.2. Example: Iterative Fibonacci

The most complex validated program computes the Fibonacci sequence iteratively using an FSM (Finite State Machine) PE to control iteration, Register PEs to store state, and Fork PEs to duplicate values where needed:

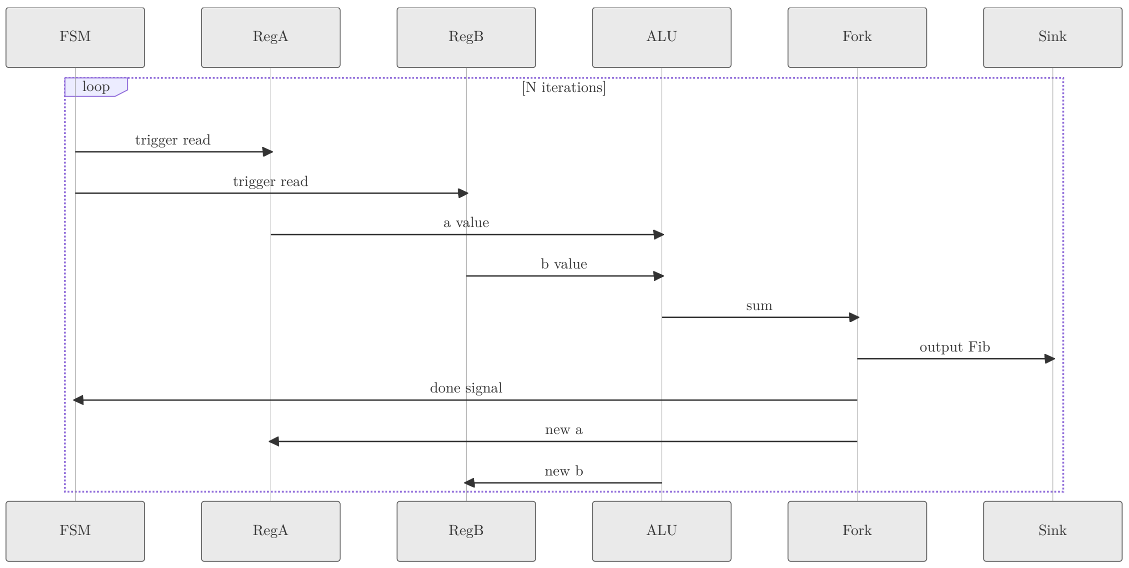

Figure 5: Fibonacci sequence diagram with loop. Output after 5 iterations: [1, 1, 2, 3, 5]

This program demonstrates that complex stateful computations are possible in pure dataflow without any central controller. The FSM PE emits a precisely timed sequence of tokens that trigger register reads, the ALU computes the sum, Fork PEs duplicate values where needed, and the results flow to both the output Sink and back to update state for the next iteration.

12.3. Validation Programs

Beyond Fibonacci, the test suite includes programs that stress different aspects of the system. Pointer-chasing workloads validate that capability enforcement works correctly under realistic memory access patterns where each load returns the address for the next load. Ring-buffer implementations stress the router's throughput capabilities with high token rates. UART output programs demonstrate I/O capabilities by converting integers to decimal strings and printing them character by character.

The test suite covers unit tests for individual PEs, integration tests for multi-PE interactions, stress tests for capability enforcement, and end-to-end tests that run complete programs and verify their output.

13. Research Directions

FLUID opens several research questions that we believe merit investigation, whether through our own work or through collaborations with academic partners. The following sections outline potential thesis topics and paper directions that could advance the state of knowledge in dataflow architectures, hybrid computing, and hardware-software co-design. We present these not as a complete research agenda but as starting points for conversations with researchers who share our interests.

13.1. Hybrid Analog-Digital Dataflow

The structural correspondence between FLUID program graphs and analog patch panels suggests that the same graph representation could target both digital and analog domains. This raises a fundamental question: can FLUID serve as a unified programming model for hybrid analog-digital systems?

Pursuing this question would require understanding how analog component characteristics should be exposed in the PE abstraction. Settling time, precision limits, and temperature drift behave quite differently from the deterministic timing of digital logic, and the abstraction must account for this without overwhelming the programmer with low-level concerns. A successful hybrid model would also need compilation strategies that minimize boundary crossings between digital and analog domains while respecting the resource constraints of both. Perhaps most intriguingly, the capability-based security model might extend to analog channel access, governing which processes may drive which voltage rails or sample which outputs.

Whether a FLUID graph could automatically generate patch configurations for analog hardware, reducing or eliminating manual wiring, remains an open question. Collaboration with researchers in analog computing and reconfigurable analog systems would be valuable in exploring these possibilities.

13.2. Dataflow Graph Analysis and Optimization

FLUID programs are static graphs, but understanding how to analyze and transform these graphs effectively remains an open problem. Unlike sequential code where profiling measures time spent in functions, dataflow programs require different metrics that capture token flow patterns, queue utilization, and PE idle time.

Graph-theoretic properties may correlate with runtime behavior in predictable ways. Highly connected hub PEs might become bottlenecks when many data paths converge, while long chains of dependent operations limit available parallelism. Developing tools that identify such patterns before execution, and suggest graph transformations that improve throughput without changing semantics, would make FLUID more practical for real-world use. The explicit structure of dataflow graphs may make such analysis more tractable than equivalent optimizations on sequential code, where control flow and data dependencies are entangled.

Automatic graph transformation is another avenue worth exploring. Given a correct but inefficient graph, can we systematically derive equivalent graphs with better performance characteristics? Techniques from term rewriting, graph grammars, and compiler optimization may apply, though the specific constraints of FLUID's token protocol and PE semantics would require adaptation.

13.3. Capability Systems for Heterogeneous Resources

FLUID's inline capabilities currently govern memory access, but extending this model to other resource types raises interesting questions. Analog channels, optical wavelengths, accelerator time-slices, and other non-memory resources might benefit from capability-based protection, but the appropriate formats and semantics are not obvious.

The fundamental challenge is expressing access rights for continuous resources like analog voltages in a framework designed for discrete digital operations. Capability delegation across domain boundaries introduces additional complexity, particularly when the domains have different timing models and error semantics. There may also be opportunities to encode quality-of-service guarantees within capabilities, specifying not just what a process may access but what precision or latency bounds it requires. Whether these extensions preserve the simplicity that makes capability systems attractive remains to be seen.

13.4. Formal Verification of Dataflow Programs

The absence of hidden state and explicit token flow may make FLUID programs more amenable to formal methods than sequential code with implicit control flow and shared mutable state. This hypothesis deserves investigation.

Type systems might be able to statically guarantee capability safety for FLUID graphs, ensuring at compile time that no token can ever exceed its permitted bounds. Some properties of dataflow graphs that are undecidable for sequential programs might become decidable, simply because the explicit structure removes ambiguity about what can affect what. Existing dataflow verification techniques could be adapted to FLUID's specific token format and PE semantics, potentially yielding practical tools for high-assurance system development. The research value lies not just in verification tools themselves but in understanding which program properties become tractable when the programming model makes data dependencies explicit.

13.5. Hardware-Software Co-Design Methodology

FLUID's design assumes tight integration between architecture and system software, but the principles that guide effective co-evolution are not yet well understood. The emulator provides a starting point, but moving from software model to FPGA implementation to production hardware involves decisions that ripple across abstraction levels.

PE design decisions affect kernel scheduling strategies in ways that are difficult to predict without experience. The feedback loops between hardware implementation choices and software usage patterns may be the most important determinant of whether a design succeeds, yet we lack systematic methods for navigating these interactions. Whether the progression from emulator to FPGA to ASIC can be systematized as a repeatable methodology, rather than an ad hoc process requiring deep expertise at every step, is a question with implications beyond FLUID itself.

13.6. Timing Predictability and Real-Time Systems

FLUID's determinism, where the same inputs produce the same outputs in the same cycle count, does not automatically translate to real-time guarantees. Understanding the conditions under which FLUID can provide bounded worst-case execution time is an open problem.

For a single program running on dedicated Processing Elements, timing analysis may be tractable. The program graph is static, the firing rules are deterministic, and queue depths are bounded. Deriving worst-case cycle counts from graph structure could enable certification for safety-critical applications.

The challenge intensifies when multiple programs share PEs. In this scenario, a scheduler must decide which process receives service on each cycle. The choice of scheduling policy, whether round-robin, priority-based, or something more sophisticated, directly affects timing guarantees. An event-driven architecture where execution depends on token arrival creates dependencies between programs that complicate analysis. If Program A produces a token that triggers Program B, the timing of B depends on A's progress.

Whether FLUID can provide meaningful real-time guarantees under PE sharing, and what constraints on program structure or scheduling policy would be required, deserves careful investigation. This question has practical implications for embedded and industrial applications where timing predictability is essential.

We welcome collaboration on these and related topics. Researchers interested in pursuing thesis work or joint publications in these areas are encouraged to reach out through the contact information at the end of this paper.

14. Roadmap

14.1. What Exists Today

The FLUID emulator represents a complete implementation of the core dataflow model with capability-based security. Development proceeded through several milestones, each building on the previous:

The foundation came first: defining the 256-bit token format, implementing the crossbar router with backpressure, and building the initial Processing Elements for arithmetic, memory access, and control flow. This established that the computational model was sound and that programs could execute correctly.

Integration with QEMU followed, demonstrating that FLUID could operate alongside a traditional RISC-V processor. The firmware boots, loads a program graph, hands off execution to the dataflow substrate, and retrieves results. This proved the accelerator model and provided a realistic test environment.

Multi-process support came next, adding the Timer, Scheduler, and StampPid PEs needed for process isolation. With these elements, multiple independent program graphs can share the hardware with proper separation, a prerequisite for any operating system functionality.

Most recently, work has focused on practical tooling: a binary graph format for efficient program representation, improved debugging facilities, and a growing test suite validating behavior from individual PE operations through complex multi-PE programs.

14.2. Near-Term: FPGA Implementation

The next milestone is implementing FLUID on an FPGA to validate that the architecture maps efficiently to real hardware. The target platform is the Tang Primer 20K development board, which provides 20,736 LUTs and 1,008 Kb of block RAM. Estimates suggest this is sufficient for approximately 50 Processing Elements, enough to run meaningful programs.

The FPGA implementation will answer questions that emulation cannot: actual area requirements per PE type, achievable clock frequencies, power consumption, and whether the router architecture scales as expected. The emulator already exports trace files that serve as test vectors for RTL simulation, enabling continuous validation that the hardware implementation matches the software reference.

14.3. Medium-Term: Operating System Primitives

Building a complete operating system on FLUID requires additional components beyond what exists today. A Syscall PE would enable user programs to invoke kernel services with proper privilege transitions. A Memory Allocator PE would manage dynamic allocation with capability-based protection. Additional I/O elements for GPIO, networking, and storage would connect FLUID systems to the outside world.

These additions align with AionCore development. The integration contract ensures that FLUID graphs map cleanly to kernel concepts: a “process” is a graph with its own capability domain, IPC happens through Mailbox PEs, and syscalls become token messages to privileged kernel PEs.

14.4. Long-Term: Compiler and Applications

General-purpose adoption would require a compiler toolchain that can target FLUID from high-level languages. An LLVM backend could map standard compiler IR to FLUID graph patterns: loads and stores become Memory PE operations, arithmetic becomes ALU operations, branches become Branch PE routings. The dataflow model is SSA (Static Single Assignment) by nature, which simplifies the mapping from LLVM's own SSA-based IR.

With a compiler in place, existing codebases could potentially target FLUID with minimal modifications. A POSIX compatibility layer might provide familiar APIs backed by FLUID's capability-based primitives. The long-term vision is a complete computing stack where the benefits of dataflow execution and hardware-enforced security are available without requiring programmers to understand tokens and PEs directly. Whether this vision is achievable, and on what timeline, depends on results from the nearer-term milestones.

15. Rethinking How We Write Programs

The deeper opportunity in FLUID lies not in running existing programs on new hardware, but in rethinking how we express computation in the first place. Traditional programming languages evolved around the von Neumann model: sequential statements that modify shared state, with parallelism as an afterthought expressed through threads and locks. Compilers for these languages perform heroic optimizations to extract parallelism that the programmer never explicitly described, fighting against abstractions that assume sequential execution.

FLUID invites a different approach. When the hardware executes dataflow graphs natively, the natural programming model becomes one of describing data dependencies rather than instruction sequences. A compiler for FLUID need not extract parallelism from sequential code; it can accept parallelism as the default and only introduce sequencing where dependencies require it. This inversion changes what optimization means: instead of finding hidden parallelism, the compiler ensures that necessary orderings are preserved while everything else flows freely.

Functional programming languages already think this way. Pure functions with explicit data dependencies map naturally to dataflow graphs. Languages like Haskell have always described what to compute rather than how to sequence operations, leaving execution order to the runtime. FLUID provides hardware that executes this model directly, potentially making functional programming not just elegant but efficient.

New language designs could go further, making capabilities and dataflow explicit in the syntax. Imagine a language where memory regions are first-class values with statically-checked bounds, where functions declare their capability requirements, and where the compiler can verify at build time that no capability violations are possible. Such a language would not merely run on FLUID; it would expose FLUID's security guarantees to the programmer as compile-time checks.

The graph-based program representation also opens possibilities for program synthesis and AI-assisted development. Generating correct dataflow graphs may prove easier than generating correct sequential code, because the explicit data dependencies constrain what connections are valid. An AI system that understands the semantics of Processing Elements could propose graph transformations that preserve correctness while improving performance, a form of optimization that operates on the program's actual structure rather than on source code that hides that structure.

These ideas remain speculative, but they suggest that FLUID's impact could extend beyond hardware efficiency into how we think about programming itself. The current work focuses on proving the architecture; the reimagining of software development is a horizon beyond that.

16. Conclusion

FLUID represents a fundamental rethinking of computer architecture for the demands of modern computing. By replacing the sequential instruction stream of von Neumann machines with parallel token flow, FLUID eliminates the central bottleneck that limits performance, complicates parallel programming, and creates security vulnerabilities. By embedding capabilities directly into every token, FLUID provides hardware-enforced security that doesn't depend on software correctness.

The architecture is not purely theoretical; it is implemented in a working emulator with hundreds of tests, integrated with QEMU for system-level operation, and demonstrated through programs ranging from simple arithmetic through iterative algorithms. The emulator serves as a research platform and reference implementation, not production software. The path to hardware is outlined, with FPGA implementation as the next milestone.

FLUID is developed by sistemica GmbH as part of the Center for Applied Complexity and Intelligence research initiative. The project is open source and welcomes collaboration from researchers and engineers interested in the future of computing architecture.

17. References

[1] Dennis, J. B. and Misunas, D. P. “A Preliminary Architecture for a Basic Data-Flow Processor.” *Proceedings of the 2nd Annual Symposium on Computer Architecture (ISCA)*, pp. 126–131, 1974.

[2] Gurd, J. R., Kirkham, C. C., and Watson, I. “The Manchester Prototype Dataflow Computer.” *Communications of the ACM*, 28(1):34–52, 1985.

[3] Arvind and Nikhil, R. S. “Executing a Program on the MIT Tagged-Token Dataflow Architecture.” *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[4] Veen, A. H. “Dataflow Machine Architecture.” *ACM Computing Surveys*, 18(4):365–396, 1986.

[5] Johnston, W. M., Hanna, J. R. P., and Millar, R. J. “Advances in Dataflow Programming Languages.” *ACM Computing Surveys*, 36(1):1–34, 2004.

[6] Kung, H. T. and Leiserson, C. E. “Systolic Arrays (for VLSI).” *Sparse Matrix Proceedings*, pp. 256–282, 1978.

[7] Kung, H. T. “Why Systolic Architectures?” *Computer*, 15(1):37–46, 1982.

Contact: hannes.lehmann@sistemica.de
Website: https://fluidcomputing.io
GitHub: https://github.com/fluidcomputing