Kami Processor: Status and Comparison

Larry Lee

May 7, 2020

Abstract

This report summarizes the current state of the open source Kami Processor and compares it against SiFive's S2-series processors.

Both the Kami Processor and SiFive's S2-series processors belong to similar architectural classes. Both are organized around single issue 3-5 stage pipelines.[1] Both possess 32 and 64 bit data paths and support privilege mode hierarchies.

Whereas SiFive's S2-series processors are modeled using Chisel, a domain-specific language (DSL) embedded within Scala, the Kami Processor uses Kami, a DSL embedded within Gallina - the functional programming language used by the Coq interactive theorem prover.

Gallina is a dependently typed programming language, and Kami uses Gallina's type system to encode properties possessed by its components. It uses Coq's macro language, Ltac to encode mathematical proofs verifying that these system components posses the asserted properties.

Kami's ability to encode and verify properties enables it to provide stronger assurances about its correctness than SiFive's chisel based cores, which rely on traditional model checking and stimulus test protocols.

We believe that these stronger assurances uniquely qualify the Kami Processor for high-reliability embedded applications.

Background

RISC-V

RISC-V is an open instruction set architecture (ISA) originally developed at Berkley and Stanford.

As the name indicates, RISC-V is a reduced instruction set computer architecture.

[1] The Kami processor currently performs all of its processing stages in a single clock cycle, but the design may be extended to support pipelined execution.

Following the philosophy of other RISC architectures, RISC-V defines a small set of core processor instructions while eschewing the more complicated instructions provided by traditional complex instruction set computer (CISC) architectures.

This approach is justified by the observation that RISC processors are able to increase instruction execution rates by minimizing the length of their critical paths. They achieve this by eliminating the combinatorial circuitry needed to implement the complex instructions that characterize CISC architectures.

This approach represents a fundamental tradeoff. By reducing their instruction sets to a small number of core operations, RISC processors can typically execute these instructions within fewer and shorter clock cycles than CISC architectures. However, they require programs that are executing more complex operations to perform those operations in software - i.e. to decompose these operations into a series of instructions. This typically results in slower execution times for more complex operations.

As manufacturing processes approach the limits of transistor densities achievable using CMOS technologies, processor designers are no longer able to include the combinatorial circuits needed so support the full range of complex operations needing hardware acceleration. This has spurred the rise of domain-specific processors (DSPs). These processors are characterized by RISC core architectures with dedicated circuitry to support a small number of complex operations critical to specific application domains. SiFive and other companies are designing DSPs tailored for realtime signal processing, AI inferencing, and other applications.

RISC-V was designed to be an extensible RISC architecture. It includes provisions for adding specialized complex instructions and for standardizing sets of instructions critical to common applications.

Consequently, RISC-V is well positioned to provide a common architectural platform for DSP designs.

This extensibility means that conformant RISC-V processors implement a subset of the extensions defined by "The RISC-V Instruction Set Manual," referred to as "the RISC-V spec." The RISC-V community uses a letter code to indicate which extensions are supported by a given RISC-V processor. For example, SiFive's E24 processor uses the following code: "RV32IMAFC," which indicates that the E24 processor has a 32 bit data path and supports the I, M, A, F, and C extensions.

The Kami processor supports a larger set of RISC-V extensions, namely: I, M, A, F, D, C, S, U, Zicsr, Zifencei.

The RISC-V Foundation, an international consortium, currently directs the development of the RISC-V standard.

Kami

Kami is a modeling language intended to model digital circuits. Its language syntax and semantics are derived from BlueSpec. It is a domain-specific language embedded withing Gallina, Coq's dependently typed programming language. Kami uses Gallina's dependent types to specify component properties. Kami verifies that these component properties are true using mathematical proofs expressed in Coq's Ltac and Gallina. Ltac is a macro language that Coq interprets to generate Gallina terms. These Gallina terms represent mathematical proofs.

A mathematical proof is a sequence of assertions in which every assertion is either an axiom or can be derived from one or more of the preceding assertions by applying a rule of logical deduction. Proofs correspond to Gallina terms, and Coq's type checker verifies that these proofs are logically sound.

Formally-verified software development is the practice of developing mathematical proofs that verify that a software model satisfies certain constraints. This methodology circumvents the combinatorial explosion that limits the assurance provided by traditional model checking. Consequently, formally verified software development has the potential to provide stronger correctness assurances than traditional model checking.

A Kami model of a synchronous digital circuit consists of one or more Kami modules and a scheduler. A Kami module comprises a set of registers, rules, and methods. A rule represents a combinatorial circuit that the scheduler tries to execute as frequently as possible. A method is a combinatorial circuit that posses a well defined set of inputs and outputs and which may be shared by circuits provided by other modules. Both rules and methods consist of Kami actions. A Kami action is a combinatorial circuit that may read or write to one or more registers and which may "call" one or more methods. Kami actions consist of a series of Kami expressions, which are combinatorial circuits that do not write to registers or call methods, and Kami statements that read and write to registers and call methods.

All of the constructs of the Kami language, modules, actions, methods, and expressions represent elements of physical circuits. When compiled into these elements, these constructs map onto circuit gates, wires, and registers. These physical elements possess certain constraints. For example, a wire cannot transmit more than one signal at a time, a register may not store more than one value at once, etc. The Kami semantics implicitly encode these constraints and ensure that no two circuits drive more than one signal on the same line at a time, that no two circuits attempt to write different values to the same register at the same time, and so on. These constraints express themselves in the distinction

that Kami makes between rules, methods, actions, and expressions. For example, because Kami actions may write to registers, we must ensure that no two actions write to the same registers before we can execute them within the same cycle.

Kami Processor

Overview

SiFive started the Kami Processor project in 2018 intending to develop a canonical model of the RISC-V specification. Over the subsequent year, SiFive sponsored development of the open source model.

The Kami Processor models a single cycle RISC-V processor. However, the model can be extended to model a RISC-V processor possessing a five stage pipeline. The model consists of a collection of subunits that can be composed to form this five stage pipeline.

Status

The Kami processor supports the following RISC-V extensions: I, M, A, D, F, C, S, U, Zifencei, and Zcsr. This means that the Kami processor supports the mandatory core set of integer arithmetic and logic instructions prescribed by the RISC-V ISA (I). In addition, it supports integer multiplication (M), atomic memory operations (A), 64 bit floating point operations (D), 32 bit floating point operations (F), compressed instruction encodings (C), multi threaded synchronization support (ZiFencei), and support for RISC-Vs control status register operations (Zicsr).

The RISC-V spec allows conformant processors to dynamically change the width of their internal data paths. The spec refers to the data path width as XLEN. Kami Processor supports both 32 and 64 bit data paths and can be configured to include support for dynamically changing XLEN.

Additionally, the Kami Processor supports the RISC-V privilege mode hierarchy. Applications running on the Kami Processor have access to the RISC-V Machine, Supervisor, and User modes. These privilege modes are used by Linux and other Kernels to protect computing resources from malicious applications.

The RISC-V memory model includes memory virtualization. The Kami Processor includes a page table walker which allows applications to translate between physical and virtual memory addresses. Kami Processor includes a translation

look-aside buffer that caches address translations in a buffer that uses a pseudo-LRU allocation algorithm.

The RISC-V spec defines several virtual memory modes. These are denoted by a code that indicates the size of the words used to encode virtual addresses. For example, "sv39" indicates that the virtual addresses are 39 bits wide. The Kami Processor supports virtual address modes sv32, sv39, and sv48.

The RISC-V memory protection model prescribes a physical memory protection scheme based around physical protection control status registers (PMP CSRs). The Kami Processor supports the full suite of PMP CSRs and all of the RISC-V address matching schemes, namely: top of range (TOR), naturally aligned four-byte regions (NA4), and naturally aligned power of two regions (NAPOT). Its address granularity is configurable.

SiFive designed the Kami Processor to use the TileLink interchip connect protocol. The TileLink protocol defines a set of conformance levels. Kami Processor's device interface communicates using the TileLink Uncached Heavyweight (TL-UH) message protocol. This protocol extends the basic read and write messages provided by the TileLink Uncached Lightweight (TL-UL) message protocol by adding atomic memory operations. By supporting the TileLink protocol, the Kami Processor can be integrated into systems using a number of interchip bus protocols either directly or through a conversion bridge.

The Kami Processor has been partially verified using a combination of traditional stimulus testing and formal verification. The current version of the Kami Processor successfully executes all of the test programs included within the RISC-V Foundations test suite. SiFive has also successfully used the Kami Processor to execute a significant fraction of the Linux Kernel's boot loader. In addition, SiFive has verified several components used within the Kami Processor using the formal theorem proving methods described below.

SiFive's formal verification methodology revolved around developing a canonical model of the various components used within the Kami Processor. The SiFive team then proved that components used within the Kami Processor were functionally equivalent to these components using properties of the underlying Kami semantics. Using this methodology, the SiFive team verified the correctness of the Fifos, register arrays, free lists, decoders, and other processor components.

Model Architecture

Conceptually, the Kami Processor can be divided into the following subunits: a set of pipeline units, a memory interface, and a set of device models. Each of these units corresponds to a set of Gallina terms that generate a set Kami

registers, rules, methods, actions, and expressions. The model combines these elements into a single Kami module which models a RISC-V processor core. Additionally, the model includes device models which may be used to represent memory and IO devices such as RAM and UARTs.

It's important to note that the Kami Processor is a highly configurable processor model. It consists primarily of a set of Gallina functions that generate Kami constructs based on a set of given parameter values and reference tables. The constructs generated by these generator functions only include elements needed to support the selected features. For example, Kami Processor may be configured to support a subset of the extensions listed above. If a developer configures the Kami Processor without floating point instruction support, the generator functions will not generate the floating point functional units that execute these instructions within the Executer, they will omit the floating point registers, and modify the Commit Unit accordingly.

The Kami Processor includes the following pipeline units: Config Read, Fetch, Decode, Register Read, Input Transform, Execute, Output Transform, Memory Unit, and Commit. Each of these units corresponds to a Kami action that may be wrapped within a single Kami rule. In the current design, the Decode, Register Read, Input Transform, Execute, Output Transform, and Memory Unit.sendMemReq units are wrapped within a single Kami rule. Whereas the Config Read, Fetch, Memory Unit.recvMemRes, and Commit units are wrapped within dedicated rules.

The memory interface consists of five conceptual subunits: the Fetcher, Address Translator, PMA/PMP Checker, Arbiter, and Device Router. The pipeline units send requests, and receive responses from, the memory interface, which communicates with the memory and IO devices that are connected to the processor core through the TileLink interface.

The Fetcher speculatively fetches instructions from memory and caches them in a double headed Fifo with a toggle pointer that references the lower 16 bits of the next instruction in the fifo. The Address Translator consists of a Translation Look-Aside buffer (TLB). This TLB consists of a Content Addressible Memory (CAM) cache that caches address translations and a page table walker that executes the walk algorithm prescribed by the RISC-V spec. The Cam uses a psuedo-LRU algorithm to manage cache entries.

The Arbiter accepts memory requests from its three clients: the fetcher, the Address Translator, and the pipeline Memory Unit; selects one of these requests to forward to the Device Router; and routes incoming memory responses back to the associated client. The Device router, in turn, forwards requests from the Arbiter to the designated device and encodes/decodes the TileLink request and response packets.

The Pipeline units are conventional. Config Read, reads the state registers such as the Program Counter (PC), Privilege Mode register (MPP), and so on to establish the current execution context. The Fetch unit sends an instruction fetch request to the Fetcher which caches the fetched instruction in a double headed Fifo as described above. The Decode unit retrieves the fetched instruction and decodes it using the Instruction table. It includes a Decompressor which translates compressed 16 bit instructions into their equivalent 32 bit instructions before decoding. The Register Read unit reads the registers associated with the decoded instruction and writes these into a Context Execution Packet.

Many instructions perform similar operations as others. The Kami Processor model referes to these shared operations as Semantic Functions. These semantic functions are performed by a functional unit. The Kami Processor implements each instruction by transforming the data read by the Register Read unit, passing the transformed values to the Semantic Function provided by a functional unit, and then transforming the values returned by the Semantic Function.

The Input Transform applies the instruction specific transformation to the values returned by the Register Read unit. The Execute unit performs the semantic function associated with the current instruction. The Output Transform applies the instruction specific transform to the value returned by the Execute unit, and generates an Execution Context Update packet. Conceptually Execute Context Update packets contain of one or more commit/memory operation calls. Each call consists of an operation code and an operation argument.

The pipeline memory unit consists of two subcomponents: Memory Unit.sendMemReq and Memory Unit.recvMemRes. If the current instruction performs a memory operation, the Memory Unit.sendMemReq sends the appropriate memory request to the memory interface. The memory interface passes the response to the Memory Unit.recvMemRes unit.

Finally, the Commit unit updates the processor registers with the results of the instruction. It may flush the various caches, increment the Program Counter, change the Privilege mode, etc. The operations that it performs depend on the commit/memory operation calls emitted by the Output Transform, the exception signals passed to it, etc.

System Architecture

As mentioned above, the Kami Processor is implemented as a set of Gallina terms that generate a Kami model. These generating functions parse a set of tables implemented as lists of Gallina records. These tables define the semantics of the supported RISC-V instructions; the compressed instruction encodings; the semantics of the memory operations supported by the TileLink protocol;

the properties of the CSR interface registers; and the properties of the devices mapped onto the processor's physical address space.

The system comprises the following elements: the Gallina terms that generate the Kami constructs that comprise the Kami model; the Gallina terms that specify the configuration parameters used by the generators; and Coq proofs verifying the properties of the generated model components and the generator framework itself. In addition, the larger framework, RiscvSpecFormal, includes utilities for simulating the Kami model, executing the RISC-V test suite, and generating an equivalent model in Verilog.

The RiscvSpecFormal system is broadly divided into four parts: ProcKami, which defines the Kami Processor; StdLibKami, which defines resusable circuit components that are used within the Kami Processor; The Kami library, and auxiliary libraries, which define the Kami language, verification framework, Verilog compiler, and simulators; and the supporting utilities.

Processor Comparisons

SiFive's S21 series processor is a single issue 64 bit processor with a 3 stage pipeline. The S21 supports the I, M, A, and C extensions, and provides two privilege mode levels: Machine and User mode. Floating point support is optionally supported. It also supports physical memory protection through 4 PMP registers.

Like the Kami Processor, the S21 issues instruction fetch requests using the TileLink protocol and its fetch unit fetches 32 bit words per request allowing unaligned fetch requests.

The S21's branch prediction follows that implicitly used by the Kami Processor. Both processors predict conditional branches as not-taken. And similarly both processors incur a single cycle penalty for conditional branch misses. [2]

The S21's pipeline consists of three sequential stages: instruction fetch, execute, and writeback. The instruction fetch and writeback stages correspond to Kami Processor's Fetch and Commit stages respectively.

Both the S21 and Kami processor have a peak execution rate of one instruction per cycle. The S21's execute pipeline stage corresponds to the Kami Processor's Decode, Input Transform, Execute, Output Transform, and Memory Unit.sendReq stages.

Similarly, when the S21 executes memory load instructions that require multiple execution cycles, it interlocks its pipeline as Kami Processor does.

[2] When Kami detects a branch prediction failure, it flushes its instruction cache.

Unlike the Kami Processor which performs division in a single cycle, the S21 executes these instructions over multiple clock cycles. This increases instruction latency for these operations but conforms with the S21's objective of optimizing area.

Neither the S21 nor the Kami Processor incur a stall if their TileLink buses acknowledge their memory requests on the cycle after they are sent. Both interlock however until their requests are acknowledged.

The S21 does not appear to support memory virtualization or Supervisor mode.

Conclusion

Overall, the Kami Processor is comparable to SiFive's S2 series processors. It supports a larger set of RISC-V extensions and offers support for memory virtualization, and additional privilege modes, but largely overlaps with the behavior and features provided by the S2 series. Whereas SiFive's S2 series aims to provide an efficient processor that minimizes its core area, the Kami Processor strives to provide high assurance for use in critical applications. This difference in focus is exemplified by the way in which the two processors implement integer division. The Kami Processor performs integer multiplication and division within a single clock cycle. This decision enlarges the Kami Processor's area footprint while minimizing the execution time for this operation. In contrast, the S2 series accepts increased latency for this operation, but shortens its internal critical paths and minimizes its area consumption.

The Kami Processor's value lies in its verification methodology. Ultimately, formal verfication is the only verification methodology that can provide strong correctness assurances without being blocked by the combinatorial wall that limits the applicability of traditional model checking and stimulus based testing. In exchange for this increased assurance, the Kami Processor trades performance for simplicity and verifiability.