Modular Deductive Verification of a GPU Hardware Implementation

by

Joonhyup Lee

B.S., Seoul National University, (2023).

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2026

Authored by: Joonhyup Lee
Department of Electrical Engineering and Computer Science
August 14, 2026

Certified by: Adam Chlipala
Arthur J. Conner (1888) Professor of Computer Science, Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Modular Deductive Verification of a GPU Hardware Implementation

by

Joonhyup Lee

Submitted to the Department of Electrical Engineering and Computer Science
on August 14, 2026 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

ABSTRACT

Previous work on GPU semantics typically assumes an abstract machine whose relationship to actual hardware is unproven. This thesis presents Mangrove, a parameterized GPU implementation described in a rule-based hardware-description language embedded in the Rocq proof assistant, with a mechanized proof that it refines its ISA semantics. The main technical contribution is a specification of the processor core, independent of the memory system, that abstracts away all microarchitectural detail and leaves only the architecturally visible state and externally visible requests. We prove the implementation refines this specification, then compose it with a memory-system specification and prove the composed implementation also refines a system-level specification — a sequentially consistent operational semantics that transitions to undefined behavior on a detected data race — yielding a DRF-SC (sequential consistency under data-race freedom) result established directly against the hardware implementation. We further give a notion of per-kernel correctness in terms of Hoare triples, lift it to a theorem about the host-observable trace over multiple kernel launches, and verify a vector-addition kernel correct, obtaining an end-to-end theorem from assembly down to hardware. The implementation is synthesized on an FPGA and compared to a reference implementation, achieving comparable performance on compute-bound workloads and scaling characteristics representative of a GPU.

Thesis supervisor: Adam Chlipala
Title: Arthur J. Conner (1888) Professor of Computer Science

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Graphics processing units (GPUs) have evolved far beyond their original purpose of rendering graphics. Today, they serve as the computational backbone of large-scale machine learning, scientific simulation, and general-purpose parallel computing. As their role in critical applications grows, so too does the importance of understanding precisely what guarantees a GPU program can provide.

The semantics of GPU programs (also called *kernels*) have received considerable attention from the research community. There are systems that aim to give better abstractions to program GPUs [21, 4]. There are systems that test or statically analyze kernels to prove they are safe [5, 14, 9]. There are also formal specifications of GPU memory models [2, 26] and the execution model of GPUs [11]. All of the projects previously mentioned assume some abstract machine model of the GPU: something that has never been grounded in a hardware specification proven to be realized by an actual implementation. The link between a formal model and the hardware it purports to describe is left implicit or simply assumed.

This thesis aims to close that gap. Mangrove, our verified GPU implementation, has proofs of correctness against its ISA semantics *mechanized* in the Rocq theorem prover. Thus, we have established a foundation for GPU program semantics that is anchored in concrete hardware rather than an idealized abstraction. The central problem we address is the following: *how can one formally verify that a hardware implementation of a GPU correctly realizes its ISA semantics, in a way that is both modular and practically meaningful?* This problem has two intertwined dimensions. The first is a question of correctness: does the hardware do what its specification says it should? The second is a question of structure: what is the right decomposition of the verification effort such that the proof is manageable, informative, and reusable? Answering these questions requires not only building and verifying an implementation but also arriving at a suitable specification. As we shall see, finding the right specification — one that is simultaneously faithful to the hardware and clean enough to be reusable — is itself a significant theoretical challenge.

1.2 Main Result

The central result of this thesis, stated informally, is the following.

Theorem 1.1 (Mangrove is correct)**.** Assume a list of kernels $[\{P_i\}\, k_i\, \{Q_i\}]_{i=1}^{n}$ bundled with their preconditions (P_i) and postconditions (Q_i). If the host performs a series of communications that programs the instruction memory and data memory, starting from a state satisfying Q_{i-1} (with Q_0 being the initial state) so as to satisfy P_i for each i, then the data downloaded between kernel k_i and k_{i+1} must satisfy Q_i.

Here, the term *host* refers to the machine responsible for dispatching programs to the GPU, typically a CPU in practice. Importantly, the theorem quantifies only over host-observable behavior: whatever the GPU does internally, it will produce correct data (according to each postcondition Q_i) at the point when the host downloads results. In practice, driver code communicating with the GPU enforces this external protocol, ensuring that the GPU is set up correctly before each kernel is launched.

A verified GPU implementation serves purposes beyond a one-time correctness guarantee. Hardware design is not static; microarchitectural choices evolve, and those choices have consequences for the programming model exposed to software. With a formally verified baseline implementation, one can iterate: modify the implementation, reverify, and observe precisely how a change propagates through to the semantics that programmers must reason about. Verification thus becomes not merely a correctness argument but a living tool for exploring the hardware-software interface.

1.3 Modular Verification Structure

Hardware is naturally structured in terms of modules. Physical constraints demand it, and sound engineering practice reinforces it. A standard decomposition made by hardware architects is the separation between a *processor core* and the *memory system*: the core issues requests, the memory system processes them, and a well-defined interface mediates between the two. Our verification effort mirrors this structure, specifying and verifying each component separately before composing them into a whole-system result.

The main theoretical contribution of this thesis arises from the attempt to give a *modular* specification for the processor core: one that is independent of the memory system. A modern processor core is a complex artifact: instructions are pipelined, executed out-of-order, and subject to intricate dependencies on register state. One might therefore expect a faithful specification of the dependency-tracking mechanisms inside the core to reflect this complexity in kind. Surprisingly, it need not. We show that the processor core can be specified as a simple *per-instruction sequential abstraction*: each instruction is described solely by its individual input-output behavior, with no explicit mention of register dependencies, pipeline stages, or out-of-order scheduling. The pipelined, out-of-order nature of the implementation is entirely abstracted away using a purely *semantic* device, and verification then consists of showing that the implementation realizes exactly this clean abstraction via a refinement proof for the core module in isolation.

This result is noteworthy in the context of prior work. Recent efforts have explored separating ISA semantics from memory semantics [29], lending support to the idea that this decomposition is a productive way to study how the interaction between core and memory gives rise to memory-model complexity. However, that work still involves computing dependencies by recording events such as register reads and writes that are not observable to an external client. A truly abstract specification for the core should not need to reason about whether register `t0` or `t1` prevents the reordering of requests. Our specification allows a client to reason directly from the fact that a request originated from an instruction with all dependencies resolved, without any exposure of which parts of the core's internal state were involved.

1.4 Contributions

This thesis makes the following contributions.

- **A verified GPU implementation.** We present a concrete, parameterized, and modular implementation of a GPU and verify it against its ISA semantics in Rocq, providing a formally grounded basis for reasoning about GPU program behavior.
- **A novel core specification.** We give a per-instruction sequential specification of the processor core that abstracts entirely from microarchitectural details such as pipelining, out-of-order execution, and register dependencies between instructions. We prove that the implementation satisfies this specification via a refinement proof for the core module in isolation, demonstrating that the abstraction is both sound and practically useful.
- **A modular proof of the whole system.** By separating the core specification from the memory system, we obtain a verification structure that mirrors the physical modularity of hardware and makes explicit the interface across which the two components interact. The top-level specification for Mangrove is a sequentially consistent operational semantics that raises undefined behavior upon detection of a *data race*, establishing a *sequential consistency under data race freedom* (DRF-SC) result proven directly against the hardware implementation.

1.5 Thesis Organization

The remainder of this thesis is structured as follows. Chapter 2 presents the ISA semantics that our system is proven against. Chapter 3 describes how we formalize hardware descriptions within Rocq and what correctness guarantee our proofs provide. Chapter 4 describes the hardware implementation and its parameterized structure. Chapter 5 develops the core specification and the refinement proof. Chapter 6 describes the composition of the core and memory specifications into the whole-system correctness theorem. Chapter 7 evaluates our system by comparing with a reference implementation in Verilog and providing an example proof of a kernel, mechanized in Rocq. Chapter 8 revisits previous work and compares with our approach. Chapter 9 concludes.

Chapter 2

The Instruction Set Architecture

2.1 Background: SIMT Execution Model

A parallel programming model popularized by GPUs is called single-instruction, multiple-thread (SIMT). In this model, programmers write code as if it were executed by a single thread operating on scalar data. In reality, the code is automatically executed in parallel across a group of threads, each running the same instruction on a different vector lane. NVIDIA coined the term *warp* to refer to the smallest such group of threads sharing a single instruction, and we adopt this terminology throughout [28].

This programming model diverges from single-threaded programming when data-dependent control flow causes threads within the same warp to split into different execution trajectories. This divergence is harmful for performance, as a split warp performs less useful work than one in which all threads follow the same path. Hardware therefore provides mechanisms to regroup diverged threads back into warps. This dynamic split/join behavior is what distinguishes the SIMT programming model. The component responsible for managing this behavior is realized concretely in both GPU implementations and our ISA semantics: the *scheduler*.

2.2 Assembly Language for Mangrove

We now introduce the instruction set architecture (ISA) of Mangrove. The language is RISC-V augmented with the scheduling operations hinted at in the previous section. The extensions were first proposed and implemented by the open-source Vortex GPGPU project [33].

The semantic domains used to formulate the ISA are laid out in Fig. 2.1. All domains are parameterized by the number of warps per core (W) and threads per warp (T): we have proven our theorems against any $W > 1, T > 1$ that are powers of 2. The unit of execution in this machine is a warp, which is a 3-tuple of a warp identifier, a thread mask (specifying the vector lanes the instructions run on), and a program counter. The machine is assumed to have separate data and instruction memories; the latter is assumed to be writable between program executions but immutable while a program is running. There are three kinds of registers on the machine: general-purpose registers, floating-point registers, and control-and-status registers (CSRs). The first two kinds live in $\mathcal{R}$, and the third kind lives in the CSR

Warps per core	W	$\in$	$\mathbb{N}$	
Threads per warp	T	$\in$	$\mathbb{N}$	
Thread index	t	$\in$	TIdx	$\triangleq \{0, \ldots, T-1\}$
Thread mask	μ	$\in$	Mask	$\triangleq \mathbb{B}^T$
Warp identifier	ω	$\in$	WId	$\triangleq \{0, \ldots, W-1\}$
Program counter	π	$\in$	PC	$\triangleq \mathbb{B}^{32}$
Warp	w	$\in$	Warp	$\triangleq$ Mask $\times$ WId $\times$ PC
Memory address	a	$\in$	Addr	$\triangleq \mathbb{B}^{32}$
Data value	v	$\in$	Val	$\triangleq \mathbb{B}^{32}$
Register	r	$\in$	RIdx	$\triangleq \mathbb{B} \times \{0, \ldots, 31\}$
Register file	$\mathcal{R}$	$\in$	RF	$\triangleq$ WId $\to$ RIdx $\to$ TIdx $\to$ Val
Instruction memory	im	$\in$	IMem	$\triangleq \mathbb{B}^{30} \rightharpoonup$ Val
Data memory	dm	$\in$	DMem	$\triangleq$ Addr $\rightharpoonup \mathbb{B}^8$
Scheduler	$\mathcal{S}$	$\in$	Sched	
CSR file	$\mathcal{C}$	$\in$	CSRF	$\triangleq$ Val
Processor state	p	$\in$	Proc	$\triangleq \mathcal{P}$(Warp) $\times$ RF $\times$ DMem $\times$ Sched $\times$ CSRF

Figure 2.1: Semantic domains for the GPU core ISA. The first component for RIdx tells whether the register is a GPR or a FPR. The control-and-status-register file (CSR file) currently has one stateful element, which holds the address where the argument to the kernel is stored in data memory. The scheduler state is explained in Fig. 2.3.

file, which can be modified by the host before a program begins. Within a single program, the execution can be described as the interleaving execution of warps that are created and terminated through scheduling operations. The formal description of each instruction can be found in Fig. 2.2.

It is worth noting that we parameterize over the semantics of the arithmetic instructions. Verifying, for example, an FPU is a worthwhile endeavor, but we deem it out-of-scope. Modular verification cleanly separates concerns: Mangrove is proven against an ISA *fully generalized* over its arithmetic operations.

The most interesting part of the machine, which distinguishes it from CPUs, is the scheduling unit. Its stucture directly corresponds to the scheduling operations added by the Vortex ISA:

Instruction	**Arguments**	**Function**
`wspawn`	Number of warps	Spawns new warps with fresh warp identifiers.
`tmc`	Mask	Sets the mask of the executing warp.
`pred`	Mask	Filters the warp's mask with the given mask.
`split`	Mask, frame pointer	Allocates a new divergence stack entry.
`join`	Divergence pointer, fp	Installs itself into the specified stack entry.
`bar`	Barrier ID, count	Waits until the specified number of warps converge.

The structure of the scheduler unit is given in Fig. 2.3. A scheduler state $\mathcal{S}$ is a 5-tuple, the first 3 being data structures that hold information in order to process `wspawn`/`pred`/`tmc` ($\mathcal{F}$), `split`/`join` (Δ), and `bar` ($\mathcal{B}$).

The freelist $\mathcal{F}$ keeps a bitvector of free warp IDs, and the barrier map keeps track, per barrier ID, of which warps are waiting and how many more warps a barrier is expect-

$$\textsc{Core-Step}\quad \frac{w = (\mu, \omega, \pi) \in \mathcal{W} \qquad im(\pi) = \iota \qquad p^- = (\mathcal{W} \setminus \{w\}, \mathcal{R}, dm, \mathcal{S}, \mathcal{C}) \qquad p' \in \mathrm{exec}(\iota, w, p^-)}{(\mathcal{W}, \mathcal{R}, dm, \mathcal{S}, \mathcal{C}) \rightsquigarrow_{im} p'}$$

Instruction	$p' \in \mathrm{exec}(\iota,\ w,\ p^-)$ iff $\exists \vec{v}_1, \vec{v}_2, \vec{v}_3.\ (\forall i, t.\ \mu[t] = 1 \implies \vec{v}_i[t] = \mathcal{R}(\omega, rs_i, t))$ and:
Arithmetic f, rd, rs_1, rs_2	$p' = p^-\big[\mathcal{R}(\omega, rd, t) \leftarrow_\mu f(\vec{v}_1[t], \vec{v}_2[t]),\ \mathcal{W}\ \mathrel{+}= w\big]$
Load rd, $[rs_1{+}i]$	$p' = \textbf{if}$ aligned $\textbf{then}\ p^-\big[\mathcal{R}(\omega, rd, t) \leftarrow_\mu \mathcal{M}[\vec{v}_1[t]{+}i],\ \mathcal{W}\ \mathrel{+}= w\big]\ \textbf{else}\ \bot$
Store $[rs_1{+}i]$, rs_2	$p' = \textbf{if}$ aligned $\textbf{then}\ p^-\big[dm[\vec{v}_1[t]{+}i] \leftarrow_\mu \vec{v}_2[t], \mathcal{W}\ \mathrel{+}= w\big]\ \textbf{else}\ \bot$
Jal rd, i	$p' = p^-\big[\mathcal{R}(\omega, rd, t) \leftarrow_\mu \pi{+}4,\ \mathcal{W}\ \mathrel{+}= (\mu, \omega, \pi{+}i)\big]$
Auipc rd, i	$p' = p^-\big[\mathcal{R}(\omega, rd, t) \leftarrow_\mu \pi{+}i,\ \mathcal{W}\ \mathrel{+}= w\big]$
Jalr rd, rs_1, i	$p' = p^-\big[\mathcal{R}(\omega, rd, t) \leftarrow_\mu \pi{+}4,\ \mathcal{W}\ \mathrel{+}= \overline{\{(\mu_\pi, \omega, \pi) \mid \mu_\pi[t] = (\mu[t] \wedge \vec{v}_1[t] + i = \pi)\}}\big]$
Branch f, rs_1, rs_2, i	$p' = p^-\big[\mathcal{W}\ \mathrel{+}= \overline{\{(\mu^+, \omega, \pi{+}i),\ (\mu \wedge \neg\mu^+, \omega, \pi{+}4) \mid \mu^+ = \mu \wedge f(\vec{v}_1, \vec{v}_2)\}}\big]$
Sched f, rs_1, rs_2	$p' \in p^-\big[\mathcal{S} \leftarrow \mathcal{S}'\big]$, for $\mathcal{S}' \in \mathrm{sched}(f, \vec{v}_1, \vec{v}_2)$
Csr rd, c, rs_1	$p' = p^-\big[\mathcal{R}(\omega, rd, t) \leftarrow_\mu \mathcal{C}(c),\ \mathcal{C}(c) \leftarrow \vec{v}_1[\min\{t \mid \mu[t] = 1\}],\ \mathcal{W}\ \mathrel{+}= w\big]$

Figure 2.2: Core ISA operational semantics. The notation $\overline{S} \triangleq \{w \in S \mid w.\mu \neq 0\}$ is the operation that filters out every warp with a zeroed-out mask. The notation $\cdot(t) \leftarrow_\mu \cdot(t)$ means that the write happens ranging over all t such that $\mu[t] = 1$ in a nondeterministic order. The active warp $w = (\mu, \omega, \pi)$ is removed from $\mathcal{W}$ to form p^- before exec is applied. For Sched, no warp is directly added to $\mathcal{W}$; the warp is consumed, and new warps are emitted asynchronously via the pool of scheduler responses within the scheduler.

ing. How the divergence stack Δ works is more interesting. Each divergence stack entry (*next*, *divMask*, *convMask*, *ipdom*, *fp*) holds:

1. The *next* divergence stack entry the subwarp needs to join with (since there may be multiple levels of divergence).
2. The mask of the threads that are still *diverged*.
3. The mask of the threads that have *converged*.
4. The *join point* ($= \pi_{\mathsf{JOIN}}$), computed statically as $\pi_{\mathsf{SPLIT}}{+}12$ when the entry is allocated.
5. The frame pointer, set by the first arriving subwarp and used to distinguish recursive call frames that share the same static SPLIT PC: a joiner with $fp < e.fp$ is a spurious pass-through; one with $fp > e.fp$ evicts the current resident.

The reason why the divergence stack needs to store the PC and the frame pointer of where the join is supposed to happen is because of an optimization we perform; we do not allocate a stack entry when there is actually no divergence. The naïve implementation, which allocates an entry for every split request, will overflow the stack if the depth of divergence is deeper than the maximum depth set in hardware. With this fix, the maximum depth of allocation cannot be above T, the worst case being when all individual threads within a warp have diverged.

Divergence pointer	ptr	$\in$	SPtr	
Divergence entry	e	$\in$	StkEnt	$\triangleq \mathsf{SPtr}^? \times \mathsf{Mask} \times \mathsf{Mask} \times \mathsf{PC} \times \mathbb{B}^{32}$ $(next,\ divMask,\ convMask,\ ipdom,\ fp)$
Divergence stack	Δ	$\in$	Stk	$\triangleq \mathsf{WId} \to \mathsf{SPtr} \rightharpoonup \mathsf{StkEnt}$
Barrier ID	b	$\in$	BId	
Barrier entry	β	$\in$	BarEnt	$\triangleq \mathsf{WId} \rightharpoonup (\mathsf{Warp} \times \mathbb{N})$
Barrier map	$\mathcal{B}$	$\in$	Bar	$\triangleq \mathsf{BId} \to \mathsf{BarEnt}$
Warp freelist	$\mathcal{F}$	$\in$	Free	$\triangleq \mathsf{WId} \to \mathbb{B}$
Split request	$sreq$	$\in$	SplitReq	$\triangleq \mathsf{Warp} \times \mathsf{Mask} \times \mathsf{SPtr}^?$
Scheduler response	$srsp$	$\in$	SchedResp	$\triangleq \mathsf{Warp} \times \mathsf{SPtr}^?$
Scheduler state	$\mathcal{S}$	$\in$	Sched	$\triangleq \mathsf{Free} \times \mathsf{Bar} \times \mathsf{Stk} \times \mathsf{list}(\mathsf{SplitReq}) \times \mathcal{P}(\mathsf{SchedResp}) \times \mathbb{N}$ $(\mathcal{F},\ \mathcal{B},\ \Delta,\ sreqs,\ srsps,\ done)$

Figure 2.3: Semantic domains for the scheduler state $\mathcal{S}$. $\mathcal{F}$ is the warp-ID freelist. $\mathcal{B}$ records, per-barrier, the set of arrived warps and the remaining arrival count; a barrier releases all arrived warps once *count* reaches zero. Δ is the per-warp divergence stack used for SPLIT/JOIN reconvergence; each entry is linked through *next* and is indexed by warp ID and stack pointer. *sreqs* is a FIFO of SPLIT requests deferred from the instruction stream and resolved one step at a time, decoupling divergence-stack allocation from instruction fetch. *done* counts *all-warps-terminated* tokens (at most 1 in practice).

The scheduler adds three more kinds of transitions to $\rightsquigarrow_{im}$:

$$\textsc{Split-Step}\quad \frac{\mathcal{S}.sreqs = sreq :: sreqs^- \qquad \mathrm{alloc}(\mathcal{S}.\Delta, sreq) = \Delta'}{(\mathcal{W}, \mathcal{R}, dm, \mathcal{S}, \mathcal{C}) \rightsquigarrow_{im} (\mathcal{W}, \mathcal{R}, dm, \mathcal{S}[\Delta \mapsto \Delta', sreqs \mapsto sreqs^-], \mathcal{C})}$$

$$\textsc{Start-Step}\quad \frac{\mathcal{S}.srsps = \{((\mu, \omega, \pi), ptr^?)\} \uplus srsps^- \qquad \forall \mu', \pi'.\ (\mu', \omega, \pi') \in \mathcal{W} \implies \mu \perp \mu'}{(\mathcal{W}, \mathcal{R}, dm, \mathcal{S}, \mathcal{C}) \rightsquigarrow_{im} (\mathcal{W} \cup \{(\mu, \omega, \pi)\}, \mathcal{R}(\omega, 0, t) \leftarrow_{\mu} ptr^?, dm, \mathcal{S}[srsps \mapsto srsps^-], \mathcal{C})}$$

$$\textsc{Done-Step}\quad \frac{\forall \omega.\ \mathcal{F}_0(\omega) = 1 \qquad \forall \omega.\ \mathcal{B}_0(\omega) = \bot \qquad \forall \omega, ptr.\ \Delta_0(\omega, ptr) = \bot}{(\varnothing, \mathcal{R}, dm, (\mathcal{F}_0, \mathcal{B}_0, \Delta_0, [], \varnothing, 1), \mathcal{C}) \rightsquigarrow_{im} (\varnothing, \mathcal{R}, dm, (\mathcal{F}_0, \mathcal{B}_0, \Delta_0, [], \varnothing, 0), \mathcal{C})}$$

Note that although we have only specified safe transitions for $\rightsquigarrow_{im}$, the relation can also trigger undefined behavior (UB). For example, an unaligned memory access, which cannot fit in one request to the DRAM, triggers UB as can be seen in Fig. 2.2. The scheduler semantics (sched) can also trigger UB if a WSPAWN tries to spawn warps beyond the allowed maximum W. Similarly, the alloc function in Split-Step may raise UB, Start-Step raises UB if μ overlaps with the mask of an existing warp in $\mathcal{W}$ with the same ω, and Done-Step raises UB if $done > 0$ even though the processor has leftover work.

2.2.1 Difference with the Vortex ISA Semantics

The semantics of the `SPLIT` and `JOIN` instructions are different between our architecture and what the Vortex implementation provides. Vortex uses a per-warp immediate post-dominator (IPDOM) stack. When the program executes a `SPLIT` instruction, the divergent warps are pushed onto a stack, and only one of the split-off warps is released to continue. Execution

thus proceeds serially: one subwarp runs to completion (or to a nested reconvergence), the stack is popped, and the complementary subwarp runs. Once both paths reach the post-dominator, the warp reconverges, and execution resumes with the full mask.

In our implementation, both of the subwarps are released back into the core, with its join point statically determined by its program counter and the frame pointer. Having this assumption baked into the hardware means that the compiler must maintain a well-formedness condition on the program to ensure correct execution. To exploit this intrawarp parallel execution, for example by letting subwarps that are not waiting for memory operations to defer their joins until their siblings are done, the compiler must ensure that each subwarp jump back to the correct join point. Writing a formally verified compiler targeting this architecture is future work.

Chapter 3

Verification Methodology

Before delving into the details of the implementation of Mangrove, we first describe what it means for a hardware implementation to be proven against its specification.

3.1 Correctness as Refinement Between LTSes

Our correctness criterion is the classical notion of *refinement* between *labeled transition systems* (LTSes). We define a hardware-description language (HDL) and embed it in Rocq, allowing us to write hardware descriptions within the proof assistant. A program in the language is denoted into a LTS and is proven to refine the specification LTS, its transitions being written directly in mathematics.

A LTS is defined in terms of *state transitions*, each transition optionally annotated with a label. This notion of a *label* can be read as analogous to *methods* in an object-oriented language, which defines what kind of interactions are allowed by each object. An object, or a *module instantiation*, hides its internal state and communicates with its client only through its visible methods. To make the notion of LTSes more concrete, we first give some definitions to clarify what a *module* is.

Definition 3.1 (Signature)**.** A *signature* for a method is some $\sigma = (\vec{\tau}_{\text{args}}, \tau_{\text{ret}}) \in \mathsf{Sig} \triangleq \mathsf{list}(\mathbb{N}) \times \mathbb{N}$ that gives the bitwidths of its arguments and the bitwidth of its return value.

Definition 3.2 (Method)**.** A *value method* following a signature $\sigma = (\vec{\tau}_{\text{args}}, \tau_{\text{ret}})$ given a state space S is some $V \in \mathsf{VMet}_\sigma$ that relates the initial state and arguments of the method call with its possible return values. An *action method* following a signature $\sigma = (\vec{\tau}_{\text{args}}, \tau_{\text{ret}})$ given a state space S is some $A \in \mathsf{AMet}_\sigma$ that additionally relates the updated state after the method call.

$$\mathsf{VMet}_\sigma \triangleq \mathcal{P}\left(S \times \prod_i \mathbb{B}^{\vec{\tau}_{\text{args}}(i)} \times \mathbb{B}^{\tau_{\text{ret}}}\right) \qquad \mathsf{AMet}_\sigma \triangleq \mathcal{P}\left(S \times \prod_i \mathbb{B}^{\vec{\tau}_{\text{args}}(i)} \times \mathbb{B}^{\tau_{\text{ret}}} \times S\right)$$

Both products here are *dependent*. In $\prod_i \mathbb{B}^{\vec{\tau}_{\text{args}}(i)}$, the type of the i-th component varies with i. In other words, each argument to a method is a tuple of which the type of its components are dependent on their positions. The same notation recurs whenever a family

of types is indexed: $\prod_{vm \in N_V} \mathsf{VMet}_{\sigma_V(vm)}$ in Definition 3.4 is a table assigning each method name a relation at *that* method's signature. Where such a family is more naturally read as a function we apply rather than as a tuple we index, we write it $\forall x.\, T(x)$, which is the family of functions returning a value of type $T(x)$ dependent on the argument x.

Definition 3.3 (Interface)**.** An *interface* for a module is an element I of

$$\mathsf{Ifc} \triangleq \{(N_V, \sigma_V, N_A, \sigma_A) \mid \sigma_V \in N_V \to \mathsf{Sig}, \sigma_A \in N_A \to \mathsf{Sig}\}$$

giving, for each method, its name ($vm \in N_V, am \in N_A$) and signature ($\sigma_V(vm), \sigma_A(am)$).

Definition 3.4 (Module)**.** A *module* M following an interface $I = (N_V, \sigma_V, N_A, \sigma_A)$ given a state space S is an element (s_0, m_V, m_A, R) of

$$\mathsf{Mod}_I \triangleq S \times \prod_{vm \in N_V} \mathsf{VMet}_{\sigma_V(vm)} \times \prod_{am \in N_A} \mathsf{AMet}_{\sigma_A(am)} \times \mathcal{P}(\mathcal{P}(S \times S))$$

giving the initial state $init(M) \triangleq s_0 \in S$ and the transitions that are labeled either with $(vm, args, ret)$ or $(am, args, ret)$ or are silent (rules in R, labeled with τ).

Now it is clear that modules are LTSes that are characterized through their interfaces, which tells which methods can be called. Modules that follow the same interface can differ between how each method will affect the internal state and what response would be given to the environment. What it means for a module M to implement, or refine, a module M' is that they are externally indistinguishable: that all method-call sequences accepted by M are also accepted by M' with the same return values. That containment is called "trace inclusion," where a trace is precisely the sequence of labels observable starting from some specified initial state. The judgment $\mathit{TraceOf}_M(s, l)$ formalizes that notion of a trace: it holds iff l is a valid finite trace of module M starting from state s.

$$\textsc{Tr-Nil}\ \frac{}{\mathit{TraceOf}(s, [])} \qquad \textsc{Tr-Tau}\ \frac{s \xrightarrow{\tau} s' \qquad \mathit{TraceOf}(s', l)}{\mathit{TraceOf}(s, l)} \qquad \textsc{Tr-Vis}\ \frac{s \xrightarrow{\ell} s' \qquad \mathit{TraceOf}(s', l)}{\mathit{TraceOf}(s, \ell :: l)}$$

Trace inclusion is an intuitive characterization of correctness; however, the proofs of trace inclusion do not compose nicely. Our notion of refinement is defined so that it allows *modular verification*: modules are verified separately against their specifications, and the specifications themselves are composed to be refined against a more abstract specification. The refinement relation $\sqsubseteq$ we use allow two axes of composition: horizontal and vertical. The statement of horizontal compositionality requires terminology given in Section 3.2, but we will place it here for completeness.

Theorem 3.1 (Horizontal compositionality)**.** Let $\mathtt{M}$ be a syntactic module. Let Σ and Σ' be possible submodule semantics maps for $\mathtt{M}$. Then:

$$(\forall m.\, \Sigma(m) \sqsubseteq \Sigma'(m)) \implies [\![\mathtt{M}]\!]\Sigma \sqsubseteq [\![\mathtt{M}]\!]\Sigma'$$

Theorem 3.2 (Vertical compositionality)**.** $\sqsubseteq$ is transitive.

$\sqsubseteq$ is "correct" in the sense that $M \sqsubseteq M'$ implies trace inclusion.

Theorem 3.3 ($\sqsubseteq$ implies trace inclusion)**.** If $M \sqsubseteq M'$, then for every label sequence l,

$$\mathit{TraceOf}_M(\mathit{init}(M), l) \implies \mathit{TraceOf}_{M'}(\mathit{init}(M'), l).$$

The definition of $\sqsubseteq$ is what is commonly known as a *simulation*, where a relation ϕ connecting the internal state of the implementation and the specification is preserved through each transition of the module.

Definition 3.5 (Refinement)**.** A module $M = (S, s_0, m_V, m_A, R)$ refines $M' = (S', s'_0, m'_V, m'_A, R')$ if they have the same interface $I = (N_V, \sigma_V, N_A, \sigma_A)$ and $s_0 \preceq s'_0$, where $\preceq$ is defined as the greatest fixed point of the monotonic functor $F \in \mathcal{P}(S \times S') \to \mathcal{P}(S \times S')$ defined as:

$$\begin{aligned} F(\phi) \triangleq\ & \{(s, s') \mid \forall vm, a, r.\ (s, a, r) \in m_V(vm) \implies (s', a, r) \in m'_V(vm)\} \\ & \cap \{(s, s') \mid \forall am, a, r, t.\ (s, a, r, t) \in m_A(am) \implies \exists t'.\ (s', a, r, t') \in m'_A(am) \wedge (t, t') \in \phi\} \\ & \cap \{(s, s') \mid \forall \rho \in R, t.\ (s, t) \in \rho \implies \exists \rho' \in (R')^*, t'.\ (s', t') \in \rho' \wedge (t, t') \in \phi\} \end{aligned}$$

where $(R')^*$ is the set of all relations constructed by the composition of zero or more relations chosen from R'.

The first conjunct says that ϕ guarantees the specification can return r if the implementation returns r for every value method. The second conjunct says that ϕ guarantees the specification can return the same value and preserve ϕ through every action method. The third conjunct says that ϕ is preserved through every implementation rule by some number of silent steps on the specification side.

3.2 The Embedded DSL

Now we give the syntax and semantics of the embedded DSL we use to write hardware implementations. The language can be thought of as a pared-down version of Bluespec, a rule-based hardware-description language (HDL). "Rule-based" means that one can specify hardware on the level of LTS transitions and compile it to register-transfer level (RTL) [27]. Verified compilers for rule-based languages have been presented: Kôika, for example, is an implementation of such a compiler [8]. The semantics of the DSL presented in this section is essentially the semantics of Kôika actions, except that we allow arbitrary submodules that are not just registers. Our DSL also has a verified compilation pipeline targeting an RTL datatype in Rocq. The compiler leverages `GolemMa`, a new hardware-verification and verified-compilation framework with a rule-based surface language, currently under submission elsewhere and led by a different author [24].

Mangrove is implemented in Bluespec and hand-translated to our DSL. It is compiled using both compilers: the open-source Bluespec compiler and the verified compiler in Rocq. The output of the verified compiler exists to mainly validate our claim that we have verified an *implementation* of a GPU: that there exists a physical realization that refines the top-level specification of our system. The output of the Bluespec compiler is loaded onto an FPGA and is used to evaluate the architecture by running actual kernels in Chapter 7.

Type/width	τ	$\in$	$\mathbb{N}$	e°	$\in$	Expr°_τ	
Variable	x	$\in$	Var_τ		$\to$	$x \mid c \mid \mathsf{rollback} \mid ?$	
Constant	c	$\in$	$\mathbb{B}^\tau$		$\mid$	$f_1(e^\circ) \mid f_2(e^\circ, e^\circ) \mid m.vm(\vec{e}^\circ)$	
n-ary op.	f_n	$\in$	$\Pi_i \mathbb{B}^{\vec{\tau}_{\text{args}}(i)} \to \mathbb{B}^{\tau_{\text{ret}}}$		$\mid$	$\mathsf{let}\ x := e^\circ\ \mathsf{in}\ e^\circ$	
Submodule interface	$\mathcal{I}$	$\in$	$\mathsf{Name} \to \mathsf{Ifc}$		$\mid$	$\mathsf{if}\ e^\circ\ \mathsf{then}\ e^\circ\ \mathsf{else}\ e^\circ$	
Submodule semantics	Σ	$\in$	$\forall m.\ \mathsf{Mod}_{\mathcal{I}(m)}$	$e^\bullet$	$\in$	$\mathsf{Expr}^\bullet_\tau$	
Submodule	m	$\in$	$\mathrm{dom}(\mathcal{I})$		$\to$	$\mathsf{ret}(e^\circ) \mid m.am(\vec{e}^\circ)$	
Value method	$m.vm$	$\in$	$\mathcal{I}(m).N_V$		$\mid$	$\mathsf{let}\ x \leftarrow e^\bullet\ \mathsf{in}\ e^\bullet$	
Action method	$m.am$	$\in$	$\mathcal{I}(m).N_A$		$\mid$	$\mathsf{if}\ e^\circ\ \mathsf{then}\ e^\bullet\ \mathsf{else}\ e^\bullet$	

Figure 3.1: Syntax of DSL expressions (`Expr.t`). $\vec{e}^\circ$ denotes a typed argument list of pure expressions. The let and if forms are effect-polymorphic, appearing in both categories. rollback models an aborted computation that never produces a value. ? models a nondeterministically chosen bit vector.

Each expression of the DSL is tagged by whether it is pure or effectful. Pure expressions stem from constants, variables, or submodule value-method calls $m.vm(\vec{e})$. Effectful expressions stem from submodule action-method calls $m.am(\vec{e})$. One language construct that deserves further explanation is rollback. It serves a role similar to `panic` in software languages, where the whole expression *aborts* its execution and does not perform any updates if the execution reaches that point. The mechanism to express rollback in hardware is a *ready* wire in the interface for each method, which is held high iff the precondition for calling that method (i.e. the precondition that the execution does not abort) is internally satisfied.

A syntactic module `M` following an interface I is given by:

1. For each value/action method having signature $(\vec{\tau}_{\text{args}}, \tau_{\text{ret}})$, a pure/effectful expression `vm`/`am` of type τ_{ret} parametrized by variables $\vec{x}_{\text{args}}$.

2. A finite list of effectful expressions, each specifying a rule transition.

The semantics for each expression is given as a judgment $St; U \vdash e \Downarrow P$ in Fig. 3.2. It means: under committed global state $St \in \forall m.\ \Sigma(m)$ and accumulated local update $U \in \forall m.\ \Sigma(m)^?$, all outcomes of e must fall in $P \subseteq \mathbb{B}^\tau \times \forall m.\ \Sigma(m)^?$. The judgment follows the big-step omnisemantics style of giving operational semantics to languages with nondeterminism: we specify the conditions for a *set* of results to be a valid postcondition of the initial configuration [10]. We also prove equivalence with respect to a more traditional small-step operational semantics in the style of Kôika.

Then the interpretation of a syntactic module $[\![\mathtt{M}]\!]\Sigma$ is given as:

- *Value methods.* A call to a value method `vm` with arguments $\vec{a}$ returns r in state U when, for every P such that $init(\Sigma); U \vdash \mathtt{vm}(\vec{a}) \Downarrow P$, we have $r \in P$. Here $P \subseteq \mathbb{B}^{\tau_{\text{ret}}}$ constrains only the return value; the state is left unchanged.

- *Action methods.* A call to action method `am` with arguments $\vec{a}$ returns r and transitions from U to U' when, for every P such that $init(\Sigma); U \vdash \mathtt{am}(\vec{a}) \Downarrow P$, we have $(r, U') \in P$.

- *Rules.* The rule set is the disjoint union of *local* rules and *submodule* rules. Local rules are the same as action methods with no arguments. Each submodule rule that takes $s = init(\Sigma)[U](m)$ to s' is lifted to take U to $U[m \mapsto s']$.

$\boxed{St;\, U \vdash e \Downarrow P}$

$$\textsc{ECst}\ \frac{(c, U) \in P}{St;\, U \vdash c \Downarrow P} \qquad \textsc{ERollback}\ \frac{}{St;\, U \vdash \mathsf{rollback} \Downarrow P} \qquad \textsc{EUndef}\ \frac{\forall v.\ (v, U) \in P}{St;\, U \vdash ? \Downarrow P}$$

$$\textsc{EOp}\ \frac{\forall i.\ St;\, U \vdash e_i \Downarrow Q_i \qquad \forall \vec{v}.\ (\forall i.\ (v_i, U) \in Q_i) \implies (f_n(\vec{v}), U) \in P}{St;\, U \vdash f_n(\vec{e}) \Downarrow P}$$

$$\textsc{EVMet}\ \frac{\forall i.\ St;\, U \vdash e_i \Downarrow Q_i \qquad \forall \vec{v}, r.\ (\forall i.\ (v_i, U) \in Q_i) \implies m.vm\ (\vec{v},\, St[U](m)) \rightsquigarrow r \implies (r, U) \in P}{St;\, U \vdash m.vm(\vec{e}) \Downarrow P}$$

$$\textsc{EAMet}\ \frac{\begin{array}{c}\forall i.\ St;\, U \vdash e_i \Downarrow Q_i \\ \forall \vec{v}, r, s'.\ (\forall i.\ (v_i, U) \in Q_i) \implies m.am\ (\vec{v},\, St[U](m)) \rightsquigarrow (r,\, s') \implies (r, U[m \mapsto s']) \in P\end{array}}{St;\, U \vdash m.am(\vec{e}) \Downarrow P}$$

$$\textsc{ERet}\ \frac{St;\, U \vdash e^{\circ} \Downarrow P}{St;\, U \vdash \mathsf{ret}(e^{\circ}) \Downarrow P} \qquad \textsc{EBind}\ \frac{St;\, U \vdash a_1 \Downarrow Q \qquad \forall v, U'.\ (v, U') \in Q \implies St;\, U' \vdash a_2[v/x] \Downarrow P}{St;\, U \vdash \mathsf{let}\ x \leftarrow a_1\ \mathsf{in}\ a_2 \Downarrow P}$$

$$\textsc{EIf}\ \frac{St;\, U \vdash e^{\circ} \Downarrow Q \qquad (\mathbf{1}, U) \in Q \implies St;\, U \vdash con \Downarrow P \qquad (\mathbf{0}, U) \in Q \implies St;\, U \vdash alt \Downarrow P}{St;\, U \vdash \mathsf{if}\ e^{\circ}\ \mathsf{then}\ con\ \mathsf{else}\ alt \Downarrow P}$$

Figure 3.2: Evaluation semantics of DSL expressions (`eval`). The predicate $St;\, U \vdash e \Downarrow P$ is defined inductively. The notation $St[U](m) \triangleq \textbf{if}\ m \in \mathrm{dom}(U)\ \textbf{then}\ U(m)\ \textbf{else}\ St(m)$. ERollback has no premise: any postcondition holds vacuously, modeling abort. EUndef requires P for all values, modeling nondeterminism. Both arguments in EBOp are evaluated independently from the same U, as both are pure. Σ is fixed throughout; U grows via EAMet ($U[m \mapsto s']$) and is threaded sequentially by EBind. The condition of EIf is always a pure expression e°, so both branches resume from the original U. $m.vm\,(\vec{v}, s) \rightsquigarrow r$ and $m.am\,(\vec{v}, s) \rightsquigarrow (r, s')$ are the value- and action-method step relations from module m's specification.

3.3 Verified Compilation to RTL

Now we describe the process of compiling a syntactic module `M` down to RTL. `GolemMa` represents RTL in Rocq as a flat array of registers paired with sequences of combinational assignments, one sequence for each rule and method. We call this datatype a *netlist*. Each sequence connects input wires holding the latest value for each register to a new set of wires, producing both the updated register values and the signal to rollback. Since an assignment may only refer to already-defined wires, there are no combinational loops by construction. Sequencing the assignments through each rule and method in turn then gives a cycle-level update function, whose behavior is one-rule-at-a-time.

The compiler that does the lowering to RTL is not ours. As mentioned in Section 3.2, we target the surface language of `GolemMa`. What we add are the two ends: a translation from our DSL into `GolemMa`'s surface language, and a route from the compiler-correctness result of `GolemMa` back into the refinement relation $\sqsubseteq$ that the rest of this thesis is stated in.

3.3.1 Three Languages

Three languages are involved, and each comes with a semantics of a different flavor.

Our DSL. The language of Section 3.2: pure and effectful expressions, submodule method calls, and rollback for abort. Its semantics is given by an LTS whose transitions are *relations* between states, and the notion of refinement is also defined in terms of relations.

`GolemMa`'s surface language. Also rule-based and close enough to ours that the translation between them is structural. A syntactic module in `GolemMa`'s surface language is denoted into $\mathsf{Mod}_{\mathsf{G}}$, which is identical to our Mod type except for the fact that each rule and method specification is given by a *predicate transformer* rather than a relation. A method's transformer takes *two* postconditions:

$$[\![am]\!]_{\mathsf{G}} \in \mathcal{P}(\{\text{state}\} \times \{\text{arguments}\} \times \{\text{commit post } P\} \times \{\text{abort post } \rho\}),$$

where a method that commits must establish P of its outcome and a method that aborts must establish ρ. The second postcondition is what corresponds to our rollback. A rule's transformer takes only the first: rules are silent, so no caller has to be told that one aborted, and an aborted rule is simply one that leaves the state as it found it.

`GolemMa` has its own notion of refinement, which we write $\sqsupseteq_{\mathsf{G}}$, with the specification module on the left. `GolemMa`'s refinement relation is parameterizable over whether each method allows *stuttering* on the specification side, but we fix this parameter to forbid stuttering on refinement between methods. Fixing this parameter enables us to recover $\sqsubseteq$ from $\sqsupseteq_{\mathsf{G}}$.

`GolemMa`'s RTL representation. A netlist for an interface I is an array of registers that is typed by a vector of bitwidths $\vec{\alpha}$, together with combinational-update expressions, one per each rule and method. Each expression is a tree whose leaves are either data wires of primitive registers or input wires and whose nodes are either muxes or simple Boolean

operations. Roughly, if we let *assign* be the denotation function for a combinational-update expression am corresponding to an action method:

$$assign(\mathsf{am}) \in \Pi_i \mathbb{B}^{\vec{\alpha}(i)} \to \Pi_i \mathbb{B}^{\vec{\tau}_{\text{args}}(i)} \to \left(\Pi_i \mathbb{B}^{\vec{\alpha}(i)} \times \mathbb{B}^{\tau_{\text{ret}}}\right) \times \mathbb{B}.$$

The three kinds of expressions differ only in their output blocks. An action method returns the updated register array, its return value, and a ready signal, as displayed; a value method returns only the latter two, since value methods are read-only; and a rule returns only the register array and the ready signal, having no return value to give.

A netlist is denoted into an element of Mod_G, same as the surface language. For notational convenience, let *rdy* be the selector for the ready signal from the output of a combinational-update expression, let *regs* be the selector for the next-state wires, and let *ret* be the selector for the return value. Then the predicate transformer derived from am as described previously that relates the initial register array *arr* and argument $\vec{a}$ to postconditions P and ρ is:

$$(rdy(out) = \mathbf{1} \implies (regs(out), ret(out)) \in P) \wedge (rdy(out) = \mathbf{0} \implies \rho),$$

where $out = assign(\mathsf{am})(arr, \vec{a})$.

3.3.2 Connecting the Three Semantics

The three semantics are connected by two edges, both stated as $\sqsupseteq_\mathsf{G}$. To establish the first $\sqsupseteq_\mathsf{G}$, we first define an *embedding* function $\mathcal{G}$ from Mod to Mod_G. For a method transition $(s, \vec{a}) \rightsquigarrow (s', r)$ relating an initial state s with argument $\vec{a}$ to the next state s' with return value r, we relate $(s, \vec{a})$ to P and ρ as follows:

$$(\forall s', r.\ (s, \vec{a}) \rightsquigarrow (s', r) \implies (s', r) \in P) \wedge (\neg(\exists s', r.\ (s, \vec{a}) \rightsquigarrow (s', r)) \implies \rho),$$

sending a method's possible outcomes to P and its being disabled to ρ.

The first $\sqsupseteq_\mathsf{G}$ is established by us. We translate a syntactic module M of our DSL into a syntactic module $compile(\mathsf{M})$ in $\mathsf{GolemMa}$ and prove that the semantics of the surface language refines $\mathcal{G}([\![\mathsf{M}]\!]\Sigma)$. The second edge is $\mathsf{GolemMa}$'s compiler-correctness theorem: the netlist obtained by lowering $compile(\mathsf{M})$ refines the semantics of the $\mathsf{GolemMa}$ syntax. By transitivity of $\sqsupseteq_\mathsf{G}$, we obtain a correspondence between the netlist and our DSL.

The two denotations on the $\mathsf{GolemMa}$ side are each parametrized by the semantics of the submodules. The surface language is interpreted over the embedding of the semantics of each submodule, $\mathcal{G}(\Sigma)$. Lowering to a netlist, by contrast, gives each submodule a slot in $\vec{\alpha}$ and replaces every call on it by a combinational expression, so what a netlist is interpreted over is a realization of each submodule in the following sense:

Definition 3.6 (Netlist realization)**.** A *netlist realization* of a $\mathsf{GolemMa}$ module M_G consists of a bit-level state type, a reset value, one combinational update expression for each method of M_G and each of its rules, and proofs that these expressions simulate corresponding transitions of M_G. We write $\mathcal{N}(M_\mathsf{G})$ for the collection of all possible realizations and lift that to families pointwise in the notation of Section 3.1: $\mathcal{N}(\Sigma_\mathsf{G}) \triangleq \forall m.\ \mathcal{N}(\Sigma_\mathsf{G}(m))$.

Writing $net(\mathsf{M})$ for the lowered netlist, the series of refinements is:

$$\forall N \in \mathcal{N}(\mathcal{G}(\Sigma)).\quad \mathcal{G}([\![\mathsf{M}]\!]\Sigma) \sqsupseteq_{\mathsf{G}} [\![compile(\mathsf{M})]\!]_{\mathsf{G}}\, \mathcal{G}(\Sigma) \sqsupseteq_{\mathsf{G}} [\![net(\mathsf{M})]\!]_{\mathsf{N}} N, \tag{3.1}$$

where $[\![\cdot]\!]_{\mathsf{G}}$ and $[\![\cdot]\!]_{\mathsf{N}}$ are `GolemMa`'s denotation functions for the surface language and the netlist, respectively. The lowering itself also reads its slot layout off N; we leave that dependence implicit and record N only on the denotation. The quantifier is what makes the compiler correctness module-local: `M` compiles against a contract for each submodule.

3.3.3 Lifting a Netlist Back to $\sqsubseteq$

Equation (3.1) lives in predicate transformers, and the rest of this thesis does not, so we need the inverse of $\mathcal{G}$: a construction $\mathcal{L}$ that reads an LTS off a module in $\mathsf{Mod}_{\mathsf{G}}$ by relating an outcome iff it is contained in *all* postconditions. For an action method am, for instance, the relational LTS transition recovered from the predicate transformer $[\![am]\!]_{\mathsf{G}}$ is:

$$(s, \vec{a}) \rightsquigarrow (s', r) \triangleq \forall P.\ (s, \vec{a}, P) \in [\![am]\!]_{\mathsf{G}} \implies (s', r) \in P,$$

when the abort postcondition $\rho = \top$.

$\mathcal{L}$ transports $\sqsupseteq_{\mathsf{G}}$ back to $\sqsubseteq$ since our instantiation of the `GolemMa` refinement is *method-lockstep*: the specification takes no silent steps while simulating a method.

Theorem 3.4 (Recovery)**.** For $M \in \mathsf{Mod}$ and $M_{\mathsf{G}} \in \mathsf{Mod}_{\mathsf{G}}$,

$$\mathcal{G}(M) \sqsupseteq_{\mathsf{G}} M_{\mathsf{G}} \implies \mathcal{L}(M_{\mathsf{G}}) \sqsubseteq M.$$

A netlist's semantics under $\mathcal{L}$ can be characterized further as a state-update function on the register array.

Theorem 3.5 (Netlist as a State-Update Function)**.** Let $N \in \mathcal{N}(\mathcal{G}(\Sigma))$ and $L = \mathcal{L}([\![net(\mathsf{M})]\!]_{\mathsf{N}} N)$, with slots $\vec{\alpha}$ and expressions as in Section 3.3.1. The state space of L is exactly the register array $\prod_i \mathbb{B}^{\vec{\alpha}(i)}$. Writing *out* for *assign* of the expression belonging to the rule or method concerned, applied to s and to any arguments,

$$\begin{aligned} s \xrightarrow{\tau} s' &\iff s' = regs(out), \\ s \xrightarrow{(am,\vec{a},r)} s' &\iff rdy(out) = \mathbf{1} \wedge s' = regs(out) \wedge r = ret(out), \\ s \xrightarrow{(vm,\vec{a},r)} s &\iff rdy(out) = \mathbf{1} \wedge r = ret(out). \end{aligned}$$

Every clause is an "if and only if," so the recovered LTS admits neither more nor fewer transitions than the three displayed: a method whose ready bit is $\mathbf{0}$ relates no states, which is how blocking is modeled, and a rule is a total function of the register array. Determinism is what makes $\mathcal{L}$ lossless and hence what makes the left-hand side of (3.2) the RTL itself rather than an abstraction of it. Instantiating Theorem 3.4 at a netlist realization:

$$\forall N \in \mathcal{N}(\mathcal{G}(\Sigma)).\quad \mathcal{L}([\![net(\mathsf{M})]\!]_{\mathsf{N}} N) \sqsubseteq [\![\mathsf{M}]\!]\Sigma. \tag{3.2}$$

3.3.4 Flattening the Tree

In order to produce a *flat* netlist, we need to *inline* each submodule into the bodies of each rule and method of the parent. Inlining needs a netlist realization for each submodule in the sense of Definition 3.6. Thus every method of the submodule is lowered, and so is every one of its internal rules, readdressed to the slot that submodule occupies in $\vec{\alpha}$ and costing one proof obligation apiece; Bluespec likewise keeps a submodule's rules when it flattens a design. We recursively compile the module-tree hierarchy into netlists, and the end result is a flat netlist that bottoms out to one realization written by hand rather than derived: that of a register. Assembly of the refinement from a netlist up to the semantics of our DSL is done using $\sqsubseteq$, over a family Σ^{N} of netlist submodules and the realization map $N \in \mathcal{N}(\mathcal{G}(\Sigma^{\mathsf{N}}))$, which is supplied by the submodules' own netlists:

$$\mathcal{L}([\![net(\mathtt{M})]\!]_{\mathsf{N}} N) \sqsubseteq [\![\mathtt{M}]\!]\Sigma^{\mathsf{N}} \sqsubseteq [\![\mathtt{M}]\!]\Sigma.$$

The first $\sqsubseteq$ is (3.2), and the second $\sqsubseteq$ is horizontal compositionality.

Chapter 4

The GPU We Verified

Now we explain the implementation of Mangrove in terms of the interface each key module follows and the microarchitectural optimizations performed.

4.1 Interfaces of Key Modules

```
interface Core;
  MemReq getDMemReq();
  Warp getIMemReq();
  CsrReq getCsrReq();
  SchedReq getSchedReq();
  Void getError();
  Void putDMemResp(MemResp resp);
  Void putIMemResp(Data instr);
  Void putCsrResp(CsrResp resp);
  Void start(SchedResp resp);
endinterface
```

```
interface Proc;
  Void hostToProc(Addr addr, Data data);
  Data procToHost();
  Bool nextStage(Stage stage);
endinterface
```

Figure 4.1: The interface to a single core (Core) and the interface to the top-level system (Proc), expressed in a Bluespec-like syntax.

First, we need to ground ourselves on the interfaces of the modules we are most interested in, since they precisely specify what we can observe during their execution. The interfaces are presented in Fig. 4.1 using a syntax close to that of Bluespec. The function signatures in the interface are the signatures for the action methods, with **Void** referring to the case when $\tau_{\text{ret}} = 0$.

The processor core can issue 4 kinds of requests: to the data memory, to the instruction memory, to the CSR file, and to the scheduler. They are the modules that need to live outside of the core, as they are either programmed by the host or provides interwarp operations and therefore need freedom of physical placement. The interesting part is that besides the requests to memory, which are already present in CPU cores, there is a method to send requests to the scheduler. This additional method, and how it is allowed to be interleaved with other requests, creates interesting implications for the proof of the whole system.

The top-level system proceeds in terms of stages ($\in$ Stage):

$$\texttt{DOWNLOAD\_DATA} \rightarrow \texttt{LOAD\_DATA} \rightarrow \texttt{LOAD\_KERNEL} \rightarrow \texttt{EXEC}.$$

`nextStage` is the method that advances the stage, returning `False` when the GPU is not ready to advance the stage or if the host thought the GPU was in a different stage. It is used to poll the GPU to confirm its status. Depending on the stage, what `hostToProc` does changes. In the idle stage (`DOWNLOAD_DATA`), the host is free to read content from its data memory stored at the argument `addr`. In the next stage (`LOAD_DATA`), the host is free to load data onto the data memory directly by specifying the destination and data to store. Then the host can program the instruction memory by streaming in the program (`LOAD_KERNEL`), and the program can start execution in the next stage. The argument in the `EXEC` case programs the CSR file with `data`, which will be the pointer to the data memory where the loaded data is, and starts the program at $\pi =$ `addr`.

4.2 Implementation of Mangrove

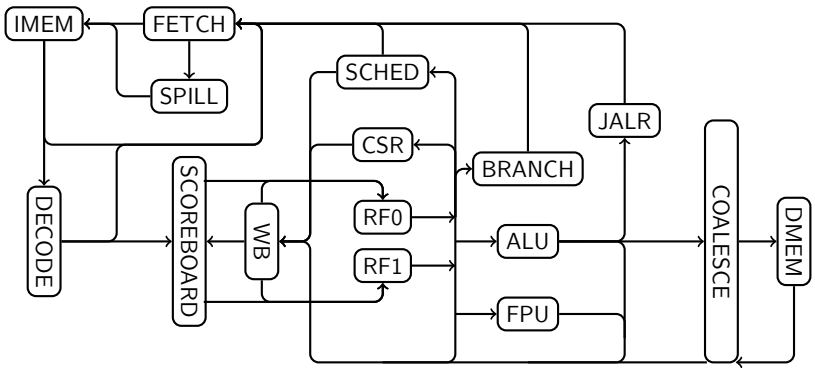

Figure 4.2: Block diagram of the SIMT processor. All modules except the instruction memory, scheduler, CSR file, coalescer, and the data memory are internal to the core.

Fig. 4.2 shows the submodules instantiated for the whole system. The whole system (`mkGpu`) is decomposed into the core (`mkCore`), the CSR file, the scheduler, the memory coalescer, the instruction memory, and the data memory.

We believe that the current system captures what differentiates GPUs from CPUs in their programming model and how they optimize for parallel computation. GPUs are *throughput machines*: the implementation allows for long-latency operations, but hides latency by filling each core with enough warps so independent computations are overlapped in time. Also, GPUs amortize the cost of communication with memory by grouping multiple threads into warps. Our implementation also performs optimizations to prevent interwarp contention over resources and to minimize the amount of off-core communication. Even though the current system instantiates only a single core, we believe that such a system already displays the scaling characteristics typical of GPUs, confirmed by running benchmark kernels. The system even starts to display behavior that is not sequentially consistent due to multiple warps completing out-of-order.

4.2.1 Optimizations

One optimization we perform to minimize interwarp contention is *banked register files* (RF0, RF1). Because of the large amount of thread-local state the core must store in its register file, the register files must be implemented as SRAMs, each of which supports only a finite number of read and write ports. To reduce contention on the register file, we partition the register files across warp IDs, with even warp IDs and odd warp IDs living on different banks. The issue logic and the scoreboard implementation must also be banked to support this optimization.

We also perform *memory coalescing* (COALESCE) to minimize the number of RAM requests. That is, per-thread memory requests are packed as much as possible within a single RAM request, as RAM transfers data using a much wider width compared to a word.

In our implementation, the coalescing operation is done by a parameterized module that takes in a vector of key-value pairs and streams out a merged list of key-value-mask pairs, the mask specifying which lanes the original pairs came from. This module is fully parameterized over the merge function, the number of vector lanes, and the input datatype. Although not the highlight of this work, we have proven the implementation of this module correct for all possible instantiations of the parameters. This proof is reused in multiple places, as the JALR unit that merges warps with the same jump target also instantiates the same module with different parameters. Already, this reuse is a showcase of why parameterized, modular verification is desirable.

4.2.2 Out-of-Order Completion

To further prevent useful work from being blocked by long-latency operations, we structure the pipeline to allow out-of-order completion of independent instructions. The current design features a scoreboard that holds the instruction stream per warp ID in program order and dispatches each warp as soon as its registers are free. Instructions branch off to their independent pipelines as early as possible, allowing for out-of-order completion. Since our design does not have speculative execution nor is it intended to support precise exceptions, out-of-order completion does not break the illusion of sequential execution. Indeed, the refinement proof of the system against the ISA machine is proof that our implementation is sound.

For instance, here is a "litmus test" that reaches an outcome forbidden under SC but is actually observable from the implementation:

(b) `lw t0, 4(x0)`	(d) `csrr t0, mscratch`
(c) `csrw mscratch, t1`	(a) `sw t1, 4(x0)`

The instructions on the left-hand side are executed by a warp with $\omega = 0$ and the right-hand side by a warp with $\omega = 1$. The current implementation allows **(a)** the store from warp 1 to run ahead, **(b)** be observed by warp 0, **(c)** then the **`csrw`** from warp 0 **(d)** be read by warp 1. Here is why this non-SC outcome is reachable: since the register file is banked, and the CSR requests are pulled to the side as soon as they leave the register file, (1) CSR requests from different warps interleave and (2) memory requests can go ahead of CSR requests.

Chapter 5

Specification of the Core

Now we present our modular specification for the core in isolation from the memory system.

5.1 Problem: In-Order Specification for a Pipelined Implementation

The desired specification for the processor core is a one-instruction-at-a-time system that, when an instruction is fetched, immediately runs the instruction and commits its effects. A specification that executes instructions instantaneously is nice because any client who wants to use the specification can reason about each instruction using only the ISA-level semantics. Furthermore, we demand that the specification be *modular* and *abstract*.

Our criterion for modularity is that the specification should be able to be linked with *arbitrary* memory systems without adaptation to the refinement proof between the core implementation and specification. The refinement should be proven once, and the specification should be reusable under every possible environment with which the core could be instantiated.

Our criterion for abstraction is that the state type for the specification LTS does not expose any explicit machinery to *constrain* the observable trace of the core. For example, the scoreboard that tracks register dependencies should not be a part of the internal state of the specification. In fact, we would like to eliminate any explicit dependency-tracking mechanism from the specification. We believe that baking in such mechanisms is detrimental to the flexibility of the specification to be adapted to different architectures.

5.1.1 Difficulty of Giving an In-Order Specification

Even without out-of-order execution, the implementation being *pipelined* already causes problems when trying to give an in-order specification. Pipelining means that there is a mismatch between the *timing* between external requests and responses. In a naïve description of a single-instruction execution, the external responses are assumed to be available instantaneously. If we write down the trace of such an execution, it would be as if the responses come immediately after their requests:

Instantaneous: $\mathsf{Req} \to \mathsf{Resp} \to \mathsf{Req} \to \mathsf{Resp}$ Pipelined: $\mathsf{Req} \to \mathsf{Req} \to \mathsf{Resp} \to \mathsf{Resp}$

However, because of pipelining, there may be multiple memory loads that are waiting for their responses, provided that they do not write to the same register. It is not obvious to see how to support the trace on the right-hand side while the specification is still in-order.

We are not the first to tackle the problem of giving in-order specifications to pipelined cores.

Kami. The multiprocessor system verified in Kami is already decomposed into the cores and the memory system [13]. This decomposition is possible because Kami modules are allowed to call external modules, not just submodules that are entirely encapsulated within the parent module. These external calls are visible as *questions* ($f(a) = b$) when denoting a syntactic module into an LTS. The set of traces generated by this LTS contains all possible return values b, allowing the caller to *assume* the return value at the point the method is called. The value of b is constrained and hidden from the external world when the module is *composed* with a context that emits a matching *answer* ($\overline{f}(a) = b$).

Using this capability to call external methods, the Kami way of giving a modular specification to a core is by composing it with a simple *wrapper*. The core uses internal queues to track outstanding requests to memory; the wrapper resolves the requests immediately by calling an external memory module. This wrapped version of the core can be proven to refine a specification where the call to external memory is *inlined* at the point the request is made in the original implementation.

Kami's solution is almost ideal, since it is both modular and abstract. However, allowing calls to external modules complicates both the LTS denotation of syntax and the definition of linking between modules. Moreover, the definition of refinement in Kami is parameterized over a *label map* that may filter some external calls; this label map is what is instantiated per usage context. We believe that our system of syntactic modules, which allows calls only to submodules, makes more obvious the connection with a physical implementation. The refinement relation is also general over the context under which it is used, meaning that if we can give a specification using $\sqsubseteq$, it will be usable unconditionally.

As we will see, although we do not have external module calls, we are able to formulate a specification that is as pleasant as Kami's.

Fjfj. Fjfj is a hardware verification framework that uses the same kind of semantics and refinement relation as ours [7]. Confronted by this problem, the generalized processor specification of Fjfj employs *nondeterministic load buffers* that may always issue loads and cache their results. Load instructions then resolve their requests by reading directly from this internal structure, circumventing the decoupled timing of requests and responses. Basically, it is pulling the L1 cache inside the core.

This idea shows its limits when there are multiple methods that may interleave *writes* out-of-order. For example, in the current GPU implementation, stores and CSR writes can go out-of-order even if there is only one resident warp. This reordering does not violate SC per se, since the memory and CSR file are disjoint; however, we cannot make the load buffer emit arbitrary stores. To allow this trace, a nontrivial extension to the specification is required.

5.2 Solution: A Tree We Can Reach Deep Inside

Thus we naturally came upon a familiar data structure: trees as execution traces. A tree with requests as nodes and branches over the possible responses is precisely the form we want, as each continuation proceeds *as if* the response were given instantly. Each new instruction binds to all leaves, adding a new node if the instruction causes observable effects. The QueTree construction in Fig. 5.1 realizes such a data structure. Each $t_q \in \mathsf{QueTree}$ represents the complete set of possible future pipeline states: a node $\mathsf{ld}(q) \gg\!= k$ records a pending load with continuation $k : \mathsf{LdResp} \to \mathsf{QueTree}$ that branches over every possible response — equivalently, every value the instruction could write to a destination register. It can be viewed as a freer monad or interaction tree specialized to the effect signature of the core [20, 34, 6]. It can also be viewed as the materialization of Kami's external calls as an explicit data structure.

Load request	q_L	$\in$	$\mathsf{LdReq} \triangleq \mathsf{Addr}^T \times \mathsf{Mask} \times \mathsf{WId}$
Load response	r_L	$\in$	$\mathsf{LdResp} \triangleq \mathsf{Val}^T$
Store request	q_S	$\in$	$\mathsf{StReq} \triangleq \mathsf{Addr}^T \times \mathsf{Val}^T \times \mathsf{Mask} \times \mathsf{WId}$
CSR request	q_C	$\in$	$\mathsf{CsrReq} \triangleq \mathbb{B}^{12} \times \mathsf{Val} \times \mathbb{B}^\dagger \times \mathsf{Mask} \times \mathsf{WId}$
CSR response	r_C	$\in$	$\mathsf{CsrResp} \triangleq \mathsf{Val}^T$
Scheduler request	q_{sc}	$\in$	$\mathsf{SchedReq}$
Core-internal state	c	$\in$	$\mathsf{Core} \triangleq \mathcal{P}(\mathsf{Warp}) \times \mathsf{RF}$
Question tree	t_q	$\in$	$\mathsf{QueTree}$
		$\to$	c
		$\mid$	$\mathsf{ld}(q_L) \gg\!= \lambda r_L.\, t_q(r_L) \;\mid\; \mathsf{st}(q_S) \gg\!= t_q$
		$\mid$	$\mathsf{csr}(q_C) \gg\!= \lambda r_C.\, t_q(r_C) \;\mid\; \mathsf{sc}(q_{sc}) \gg\!= t_q \;\mid\; \mathsf{err}(\omega) \gg\!= t_q$
Answer tree	t_a	$\in$	$\mathsf{AnsTree}$
		$\to$	$t_q \;\mid\; \mathsf{ld} \gg\!= \lambda r_L.\, t_a(r_L) \;\mid\; \mathsf{csr} \gg\!= \lambda r_C.\, t_a(r_C)$

† Indicates whether the CSR request writes to the destination CSR ($\in \mathbb{B}^{12}$) or not.

Figure 5.1: Node types for QueTree and AnsTree. Nodes ld and csr branch on response values; st, sc, and err have deterministic continuations. AnsTree additionally tracks outstanding requests to external components (ld, csr) not yet serviced by memory or the CSR file.

However, there is a problem when actually trying to extract requests from a question tree. The trees are constructed in program order, yet the requests may be pulled out-of-order. To allow this reordering of requests, we need to be able to extract requests hiding *deep inside the tree.*

Thus the crux boils down to determining *what is the criterion to be able to pull out a request buried deeply.* The obvious answer turns out to be the correct one, and we believe that it captures the essence of *what it means for an instruction to be independent of another instruction.*

A request q may be extracted from a tree t if q is present in *every* path from the root to the leaf of t. This convention is made precise in the commutation rules QLd-Ld and QLd-Csr in Fig. 5.2: to extract a load past a preceding load or CSR from a different warp, the premise requires $\forall s.\, k(s) \xrightarrow{\mathsf{ld}?q!r} k'(s)$: the target request must be present for all responses s. Because s is the value written to a register by the preceding instruction, requiring this property for all s is precisely the statement that the later request does not depend on any register value

$$\textsc{QLd-Match}\quad \frac{}{\mathsf{ld}(q) \gg\!= k \xrightarrow{\mathsf{ld}?q!r} k(r)}$$

$$\textsc{QLd-Ld}\quad \frac{q'.\omega \neq q.\omega \qquad \forall s.\ k(s) \xrightarrow{\mathsf{ld}?q!r} k'(s)}{\mathsf{ld}(q') \gg\!= k \xrightarrow{\mathsf{ld}?q!r} \mathsf{ld}(q') \gg\!= k'}$$

$$\textsc{QLd-St}\quad \frac{q'.\omega \neq q.\omega \qquad k \xrightarrow{\mathsf{ld}?q!r} k'}{\mathsf{st}(q') \gg\!= k \xrightarrow{\mathsf{ld}?q!r} \mathsf{st}(q') \gg\!= k'}$$

$$\textsc{QLd-Csr}\quad \frac{\forall s.\ k(s) \xrightarrow{\mathsf{ld}?q!r} k'(s)}{\mathsf{csr}(q') \gg\!= k \xrightarrow{\mathsf{ld}?q!r} \mathsf{csr}(q') \gg\!= k'}$$

$$\textsc{QLd-Sc}\quad \frac{k \xrightarrow{\mathsf{ld}?q!r} k'}{\mathsf{sc}(q') \gg\!= k \xrightarrow{\mathsf{ld}?q!r} \mathsf{sc}(q') \gg\!= k'}$$

$$\textsc{QLd-Err}\quad \frac{k \xrightarrow{\mathsf{ld}?q!r} k'}{\mathsf{err}(\omega') \gg\!= k \xrightarrow{\mathsf{ld}?q!r} \mathsf{err}(\omega') \gg\!= k'}$$

The above is for when the label is $\mathsf{ld}?q!r$. The other labels differ only in their blocking conditions as follows:

Relation	Blocked by	Commutes past
$\mathsf{ld}?q!r$	st	csr, sc, err
$\mathsf{st}?q$	ld	csr, sc, err
$\mathsf{csr}?q!r$	—	ld, st, sc, err
$\mathsf{sc}?q$	ld, st, csr	err
$\mathsf{err}?\omega$	—	all

Figure 5.2: Question-extraction relations on QueTree, shown in full for $\mathsf{ld}?q!r$. Blocking is the absence of a rule: where the table lists a node as blocking, no premise applies and extraction cannot proceed past it. A relation's own constructor appears in neither column, being the base case rather than a commutation: the head node is either the request sought, which is emitted (QLd-Match), or a different request of the same kind, which blocks. The $\forall s$ quantification appears exactly when commuting past a response-branching node (ld or csr), since s ranges over every value that node could write to a register. Both columns describe nodes carrying the same ω as the request; a question from a different ω can always be commuted past.

produced earlier. The core specification thus avoids register dependencies entirely: instead of tracking which instruction reads which register written by which earlier instruction, it requires only that every extractable request appears structurally in every response branch of every preceding instruction.

The blocking rules in Fig. 5.2 constrain what kinds of reorderings are possible between requests. The rules listed apply to Mangrove, and these constraints determine what constitutes a "data race" for our implementation to maintain sequentially consistent semantics. Different instantiations of these rules give rise to distinct memory-consistency models, although we have not worked out most in full detail. One notable constraint is that a scheduler request acts like a *fence* to all memory/CSR requests preceding it. That is, a scheduler request cannot be pulled out until all preceding requests have been drained. This constraint is necessary because the memory is an externally visible state: the host can read it through `hostToProc`. When the scheduler signals that all warps have terminated back to the host, all of the memory requests must have been committed to memory for the host to observe up-to-date data. Also, the CSR is also modifiable by the host through `procToHost`; all CSR request must have also been drained at the time of kernel termination.

Question trees thus serve as the state type for the core's specification module, almost. There is actually one more layer of nodes that are waiting for responses, aptly called *answer*

trees. The transition from a question to an answer is performed by binding the transition for extracting a single question to all leaves of an answer tree, which are question trees themselves. For example, for `getDMemReq` this transition would roughly be:

$$\frac{\forall r.\ t_q \xrightarrow{\mathsf{ld?}q\mathsf{!}r} k_a(r)}{t_q \xrightarrow{\mathsf{ld?}q} \mathsf{ld} \gg\!= k_a} \qquad \frac{\forall r.\ k_a(r) \xrightarrow{\mathsf{ld?}q} k'_a(r)}{\mathsf{ld} \gg\!= k_a \xrightarrow{\mathsf{ld?}q} \mathsf{ld} \gg\!= k'_a} \qquad \frac{\forall r.\ k_a(r) \xrightarrow{\mathsf{ld?}q} k'_a(r)}{\mathsf{csr} \gg\!= k_a \xrightarrow{\mathsf{ld?}q} \mathsf{csr} \gg\!= k'_a}$$

These transitions can actually be computed by recursion on the tree structure, since all response types are finitely enumerable and the premises are on subterms of the trees in the conclusion. The answer tree, when given a response, chooses one branch as detailed in Fig. 5.3.

$$\text{ALd}\ \frac{}{\mathsf{ld} \gg\!= k \xrightarrow{\mathsf{ld!}r} k(r)} \qquad \text{ALd-Csr}\ \frac{\forall s.\ k(s) \xrightarrow{\mathsf{ld!}r} k'(s)}{\mathsf{csr} \gg\!= k \xrightarrow{\mathsf{ld!}r} \mathsf{csr} \gg\!= k'} \qquad \text{ACsr}\ \frac{}{\mathsf{csr} \gg\!= k \xrightarrow{\mathsf{csr!}r} k(r)} \qquad \text{ACsr-Ld}\ \frac{\forall s.\ k(s) \xrightarrow{\mathsf{csr!}r} k'(s)}{\mathsf{ld} \gg\!= k \xrightarrow{\mathsf{csr!}r} \mathsf{ld} \gg\!= k'}$$

Figure 5.3: Answer-delivery relations on AnsTree. $\mathsf{ld!}r$ delivers memory response r to the outermost ld node, commuting past any csr nodes with universal quantification over CSR responses, and symmetrically for $\mathsf{csr!}r$.

The actual state type of the core module is a *set* of answer trees, paired with a list of warps waiting for responses from the instruction memory and a bitvector keeping track of what threads (ω, t) are resident on the core.

$$\mathtt{mkCore}.S = (\mathcal{P}(\mathsf{AnsTree}) \times \mathsf{list}(\mathsf{Warp}) \times (W \times T \to \mathbb{B}))^?$$

The bitvector is instrumentation to enforce the invariant that there cannot be more than one thread with the same (ω, t). If the scheduler enqueues a warp that violates this invariant, the module transitions into None (raises UB). The reason the core module holds a set of trees, not a single tree, is because of internal nondeterminism allowing inactive lanes to read garbage values from the register file. Since the garbage values manifest as memory requests at the interface as well, this nondeterminism has to be taken into account.

The resulting specification module has *no* rules and only methods, each method binding its own operations to the leaves of all answer trees in the set. A request can be observed only if there *exists* some tree in the set that the request can be pulled out from. A response can be given even if the core is not waiting for any responses; the core will raise UB in this case, and it is the obligation of the environment to give answers only when the core is expecting them. Most importantly, the transition of `putIMemResp` binds all trees in the set with a tree corresponding to the semantics of *each instruction* as if it were executed within a single cycle.

5.3 Proof Sketch

The proof of the refinement follows a pattern that we believe can be generalized to other pipelines as well. The crux of the simulation relation is a relation between an implementation

state st and a single answer tree t_a, a relation we call the *flushing* relation. Recall that the branches of an answer tree range over all future pipeline stages; for example, each continuation in the pipeline waiting for a memory response would correspond to a ld node in the answer tree, in the order they would be answered. Similarly, for each token tok in the pipeline, we can lift tok into a function $\lceil tok \rceil$ of the type $\mathsf{QueTree} \to \mathsf{AnsTree}$ that binds to the innermost question tree. Flushing each token incrementally deepens the answer tree, eventually resulting in a tree spanning all possible future pipeline states.

The flushing relation $st \rhd k \Downarrow t_a$ relates a partially flushed implementation state st and an accumulated continuation $k \in \mathsf{QueTree} \to \mathsf{AnsTree}$ with an answer tree t_a. We informally write $st = tok_1 :: tok_2 :: \cdots :: tok_n :: c$, where $c \in \mathsf{Core}$ is the currently committed architecturally visible state in the implementation, and tok_i is each token in the pipeline, with tok_1 at the latest stage. In most processors, tok_1 would be the token at the head of the commit stage. Then the definition of the flushing relation can loosely be given as:

$$\textsc{Flush-Drained}\quad \frac{}{c \rhd k \Downarrow k(c)} \qquad \textsc{Flush-Cont}\quad \frac{tl \rhd \lambda t_q.\ \mathbf{let}\ t'_q \leftarrow k(t_q)\ \mathbf{in}\ \lceil hd \rceil(t'_q) \Downarrow t_a}{hd :: tl \rhd k \Downarrow t_a}$$

where $\mathbf{let}\ t_q \leftarrow t_a\ \mathbf{in}\ f(t_q)$ is notation for binding $f \in \mathsf{QueTree} \to \mathsf{AnsTree}$ to t_a.

With the flushing relation in hand, it is tempting to define the simulation relation as

$$\{t_a \mid st \rhd id \Downarrow t_a\} \subseteq T$$

where $T \subseteq \mathsf{AnsTree}$ is the first component of the specification state, and id is the trivial lifting of a question tree into an answer tree. It turns out it is not quite the case, due to the fact that our pipeline is branching, and the actual definition of our flushing relation fixes a canonical order as to simplify the induction principle. This canonical order is sound, because we can preserve the invariant that for all t_a that the implementation flushes to, there exists a $t'_a \in T$ such that all traces observable from t_a are also observable from t'_a:

$$\forall t_a.\ st \rhd id \Downarrow t_a \implies \exists t'_a.\ t'_a \in T \wedge t_a \lesssim t'_a.$$

The $t_a \lesssim t'_a$ relation is defined by structural recursion, built bottom-up from $t_q \lesssim t'_q$:

$$\textsc{QLe-Leaf}\quad \frac{}{c \lesssim c} \qquad \textsc{QLe-Ld}\quad \frac{\forall r.\ \exists t'_q.\ t_q \xrightarrow{\mathsf{ld}?q!r} t'_q \wedge k(r) \lesssim t'_q}{\mathsf{ld}(q) \gg\!= k \lesssim t_q} \qquad \textsc{QLe-St}\quad \frac{\exists t'_q.\ t_q \xrightarrow{\mathsf{st}?q} t'_q \wedge k \lesssim t'_q}{\mathsf{st}(q) \gg\!= k \lesssim t_q}$$

$$\textsc{QLe-Csr}\quad \frac{\forall r.\ \exists t'_q.\ t_q \xrightarrow{\mathsf{csr}?q!r} t'_q \wedge k(r) \lesssim t'_q}{\mathsf{csr}(q) \gg\!= k \lesssim t_q} \qquad \textsc{QLe-Sc}\quad \frac{\exists t'_q.\ t_q \xrightarrow{\mathsf{sc}?q} t'_q \wedge k \lesssim t'_q}{\mathsf{sc}(q) \gg\!= k \lesssim t_q} \qquad \textsc{QLe-Err}\quad \frac{\exists t'_q.\ t_q \xrightarrow{\mathsf{err}?\omega} t'_q \wedge k \lesssim t'_q}{\mathsf{err}(\omega) \gg\!= k \lesssim t_q}$$

and extended to $t_a \lesssim t'_a$:

$$\textsc{ALe-Ld}\quad \frac{\forall r.\ \exists t'_a.\ t_a \xrightarrow{\mathsf{ld}!r} t'_a \wedge k(r) \lesssim t'_a}{\mathsf{ld} \gg\!= k \lesssim t_a} \qquad \textsc{ALe-Csr}\quad \frac{\forall r.\ \exists t'_a.\ t_a \xrightarrow{\mathsf{csr}!r} t'_a \wedge k(r) \lesssim t'_a}{\mathsf{csr} \gg\!= k \lesssim t_a}$$

It can be proven that $\lesssim$ is reflexive and transitive. Also, $\lesssim$ is monotone under bind and the question-extraction and answer-delivery operations.

For each method emitting label ℓ, the proof obligation boils down to:

$$\forall st', t'_a.\ st' \rhd id \Downarrow t'_a \implies \forall st.\ st \xrightarrow{\ell} st' \implies \exists t_a. st \rhd id \Downarrow t_a \wedge \exists t''_a.\ t_a \xrightarrow{\ell} t''_a \wedge t'_a \lesssim t''_a.$$

By definition of a simulation between action methods, we first assume a transition of the implementation from st to st'. We want to prove that the simulation relation holds on st' as well: we quantify over all answer trees t'_a reached by flushing st' and prove that there must have been a way to cover t'_a using the same step in the specification. The proofs are carried out by induction on the flushing relation by generalizing over id into a continuation k by adding suitable well-formedness conditions on st and k. The conditions are needed to commute each $\lceil tok \rceil$ past other constructors of the flushing relation.

Chapter 6

Specification of the Whole System

Now we illustrate how the specification of the whole system is given. The specification executes according to the ISA semantics in Section 2 by interleaving instructions from different warps; it is a sequentially consistent model. The refinement proof of the system against this semantics establishes a DRF-SC-type result, where a "data race" causes the specification to step into UB, else sequential consistency is maintained [1]. We will first describe what constitutes a data race under our semantics, then move on to the main theorem, which allows composing per-kernel correctness proofs into a proof about the trace of the machine across multiple kernel launches.

6.1 DRF-SC for Mangrove

The only reordering that causes the implementation to violate sequential consistency is the reordering between same-warp-ID memory operations and CSR operations. First, cross-warp-ID operations can be freely interleaved by definition of sequential consistency. Second, same-warp-ID, same-kind operations are in program order. Third, commuting with scheduler operations is either impossible or the operation must be independent of the scheduler op. The reordering is impossible if the operation precedes the scheduler operation, else the operation is after the scheduler operation. Then by the invariant that there cannot be more than one thread identified by (ω, t), the operation must act on lanes disjoint to those of the warp being issued to the scheduler. Thus, the only reordering that violates SC is between CSR and memory operations, as illustrated in Section 4.2.1.

We are in luck, since the intended use cases for CSR operations are read-only when there are multiple warps resident: a thread reads either its id (ω, t), its thread mask, or the scratch register, which is written by an orchestrator *before* spawning multiple warps. Thus, the "data race" condition that we use in our specification forbids *all CSR writes* when there are multiple warp IDs being used in the core.

6.1.1 The Refinement Proof

The state type for the whole system is

$$\mathsf{mkGpu}.S = \mathcal{P}(\mathsf{Proc}^?) \times \mathsf{IMem} \times \mathsf{Stage} \times \mathsf{list}(\mathsf{Data}).$$

The first component is a set of possible processor states, each stepping by the ISA semantics $\rightsquigarrow_{im}$ of Section 2. Stepping to None means undefined behavior was triggered, either by an invalid instruction or a data race. The second component is the instruction memory, the third component is the current stage of the GPU (e.g., `EXEC`), and the last component is the data to be downloaded by the host through `procToHost`.

The simulation relation relates a specification state that has triggered UB (which has None as an element of the first component) with any implementation state. Otherwise it relates the two whenever *one* interleaving accounts for the whole machine: a single partial trace shared by every tree, along which the specification set can be evolved so as to cover every answer the implementation's trees have already received.

We can see why such a relation can be maintained in three steps. First, the implementation runs ahead: it has requests in flight that the specification, which executes one instruction at a time, has not performed yet. The specification is therefore allowed to *lag*, the pending requests acting as a log — its copy of a resource is the implementation's copy with that log undone — and it catches up by taking silent steps whenever the implementation completes a prefix of the trace. Catching up matters most when the implementation becomes ready to accept host requests, since the specification must be equally ready: there the implementation can take no more answers, and the relation forces the last step to complete the partial trace on the specification too, so that it can run to termination.

Second, a lag is sound only while nobody can observe it, and the argument that nobody can is the one given at the start of this section. Operations from distinct warp IDs may be interleaved freely, same-kind operations from one warp ID are already in program order, and scheduler operations either cannot commute or act on disjoint lanes. What is left over is a single warp ID's private view of a resource that no other warp ID reads, and a private lag is undetectable.

Third, the one case that is not covered is a resource visible to two warp IDs while a write to it is still pending. There the lag *is* observable, and no amount of catching up repairs it — a data race, thus the specification state steps to None, which is related to every implementation state. The DRF condition is a safe boundary at which the lag argument stops working.

6.2 Correctness Proofs Per Kernel

Since the specification state keeps track of a *set* of processor states, the spec transitions by mapping the per-processor step relation to all elements. For the purpose of verifying kernels, however, we would like to reason in terms of a single processor state. Thus we state the correctness criterion for a kernel as a Hoare triple $\{P\}\, k\, \{Q\}$, with the precondition P being satisfied by all members of the set. The proof obligation boils down to proving that *each* processor state satisfying P *eventually* satisfies Q. The inductive eventually relation we use $(\rightarrow^{\diamond}_{im})$ is stated in terms of the ISA-level step relation $(\rightsquigarrow_{im})$ on a single processor state.

$$\textsc{Ev-Here}\ \frac{p \in P}{p \rightarrow^{\diamond}_{im} P} \qquad \textsc{Ev-Step}\ \frac{\exists p^{?}.\ p \rightsquigarrow_{im} p^{?} \qquad \forall p'.\ p \rightsquigarrow_{im} \mathsf{Some}\, p' \implies p' \rightarrow^{\diamond}_{im} P \qquad \neg(p \rightsquigarrow_{im} \mathsf{None})}{p \rightarrow^{\diamond}_{im} P}$$

$p \to^{\diamond}_{im} P$ holds when every execution path from p with instruction memory fixed as im reaches a state satisfying P in finitely many steps without triggering UB.

Now we can give a definition on what a *kernel* is along with the appropriate notion of a Hoare triple for a kernel.

Definition 6.1 (Kernel)**.** A *kernel* $k = (im, \pi, a) \in \mathsf{IMem} \times \mathsf{PC} \times \mathsf{Addr}$ is an instruction memory paired with the starting PC and the pointer to the arguments of the kernel in the data memory.

Definition 6.2 (Hoare triple)**.** Let $k = (im, \pi, a)$ be a kernel and $P, Q \subseteq \mathsf{DMem}$ be pre- and postconditions. Let $\mathcal{S}_0 = (\mathcal{F}_0, \mathcal{B}_0, \Delta_0, [], \varnothing, 0)$ be the drained scheduler, where $\forall\omega.\ \mathcal{F}_0(\omega) = 1$, $\forall\omega.\ \mathcal{B}_0(\omega) = \bot$, and $\forall\omega, ptr.\ \Delta_0(\omega, ptr) = \bot$. Then a *Hoare triple* $\{P\}\, k\, \{Q\}$ is defined as:

$$\forall \mathcal{R}, dm.\ dm \in P \implies \\ (\{(1, 0, \pi)\}, \mathcal{R}, dm, \mathcal{S}_0[\mathcal{F}[0] \mapsto 0], a) \to^{\diamond}_{im} \{(\varnothing, \mathcal{R}', dm', \mathcal{S}_0[done \mapsto 1], a') \mid dm' \in Q\}$$

That is, the processor with the data memory initialized to satisfy P eventually reaches a terminated state with no warps, the scheduler done, and the data memory satisfying Q under im.

Having the kernel pin down the instruction memory precisely may seem too constraining. However, it is not the case. We can quantify over im outside the triple and prove that the triple holds for any instruction memory that is initialized at π to the list of instructions we are interested in executing.

6.3 Correctness Across Multiple Kernels

Now we lift Hoare triples to a theorem about the trace of the whole system. For notational convenience, let $poll(stg)$ denote the sequence of labels $(\mathsf{nextStage}, stg, \bot)^*(\mathsf{nextStage}, stg, \top)$, which is the host poll of the GPU to transition to the next stage when the current stage is stg. Likewise, let $fold(m, l)$ denote the memory obtained by applying every $\mathsf{hostToProc}$ write in the label sequence l to m in order, which is the memory the host leaves behind once it has finished uploading; it is applied below to both the instruction and the data memory. Then we have the following:

Theorem 6.1 ($\mathsf{Mangrove}$ is correct)**.** Let l satisfy $TraceOf_{\mathsf{mkGpu}}(init(\mathsf{mkGpu}), l)$. Given a list of Hoare triples $[\{P_i\}\, k_i\, \{Q_i\}]_{i=1}^n$, a list of data-memory initializations $dmi = [(\mathsf{hostToProc}, a, v)_i^*]_{i=1}^n$, a list of instruction-memory initializations $imi = [(\mathsf{hostToProc}, \pi, \iota)_i^*]_{i=1}^n$, and a list of downloads $dl = [((\mathsf{hostToProc}, a, 0)(\mathsf{procToHost}, v))_i^*]_{i=1}^n$, if

$$l = \left[\begin{array}{l} poll(\mathsf{DOWNLOAD\_DATA}) +\!\!+ dmi_i +\!\!+ poll(\mathsf{LOAD\_DATA}) +\!\!+ imi_i +\!\!+ \\ poll(\mathsf{LOAD\_KERNEL}) +\!\!+ [(\mathsf{hostToProc}, k_i.\pi, k_i.a)] +\!\!+ poll(\mathsf{EXEC}) +\!\!+ dl_i \end{array}\right]_{i=1}^n,$$

$\forall i.\ fold(k_{i-1}.im, imi_i) = k_i.im$, and $\forall i, dm.\ dm \in Q_{i-1} \implies fold(dm, dmi_i) \in P_i$, then we have:

$$\forall i, a, v.\ (\mathsf{hostToProc}, a, 0)(\mathsf{procToHost}, \mathsf{v}) \in dl_i \implies \exists dm \in Q_i.\ dm[a] = v$$

Thus we have proven that $\mathsf{Mangrove}$ remains correct across multiple kernel launches and have validated that our definition of Hoare triples is a useful one.

Chapter 7

Evaluation

To breathe life into our main theorem, we ran case studies to demonstrate the validity of our theorem statement: that we can *observe* a trace for actual kernels and that we can *verify* the kernels with respect to their Hoare triples. As mentioned in Section 3, we have synthesized our implementation using the Bluespec compiler and run actual kernels on an FPGA. The results show that our optimizations enable Mangrove to have performance comparable to that of the Vortex reference implementation. We also illustrate how a Hoare triple can be proven by verifying a simple vector-addition kernel.

7.1 Comparing Performance with the Vortex Reference Implementation

	$(W, T) = (8, 4)$				$(W, T) = (16, 4)$			
	Vortex		Ours		Vortex		Ours	
Benchmark	Cycles	IPC	Cycles	IPC	Cycles	IPC	Cycles	IPC
`vecadd`	19 530	0.46	9 136	**0.99**	33 573	0.47	13 793	**1.15**
`sgemm`	125 037	**2.20**	293 025	0.94	140 198	**2.01**	159 372	1.77
`mstress`	93 435	**1.07**	173 328	0.58	231 707	**0.86**	345 994	0.58
`madmax`	844 721	1.43	417 067	**2.89**	870 915	1.39	412 842	**2.93**

Table 7.1: Comparison of the instructions per cycle (IPC) of 4 benchmark programs (`vecadd`, `sgemm`, `mstress`, `madmax`) with the Vortex reference implementation. Our implementation simulates a conservative 120 cycles of latency for accessing main memory and runs on an FPGA. We compare against Vortex release version 2.3, using the `rtlsim` driver instantiated with 1 core. The FPGA synthesis closes timing at 125[MHz], approximately half of that of Vortex.

We evaluate our implementation against the Vortex reference (release version 2.3) on four benchmarks that span the GPU workload spectrum [32]. `vecadd` is a simple element-wise vector addition and serves as a low-intensity baseline. `sgemm` is a single-precision matrix

multiplication with mixed arithmetic and memory traffic. `mstress` is a memory-stress microbenchmark dominated by main-memory accesses, and `madmax` is an arithmetic-intensive kernel whose working set fits comfortably in registers.

Arithmetic-intensive workloads. On `madmax`, our implementation achieves an IPC of 2.89 versus Vortex's 1.43 at $(W, T) = (8, 4)$; roughly double. Because our FPGA synthesis closes timing at 125[MHz], approximately half the operating frequency of Vortex, the two implementations deliver comparable wall-clock throughput on compute-bound code. The same doubling of IPC offsets the frequency gap for `vecadd`, where our IPC of 0.99 is again approximately double Vortex's 0.46.

Memory-intensive workloads. On `mstress`, our IPC of 0.58 falls below Vortex's 1.07. The shortfall is expected: our implementation has no data cache and models main-memory latency by adding a constant overhead to average around 120 cycles. Vortex's cache hierarchy absorbs a significant fraction of memory traffic, and the gap on `mstress` directly reflects the absence of a cache in our design. That design choice is orthogonal to our verification effort; a properly implemented memory subsystem would satisfy the same specification as a memory without a cache, meaning our proofs do not have to change. On `sgemm`, which mixes arithmetic and memory traffic, our IPC at $(W, T) = (8, 4)$ is 0.94 against Vortex's 2.20, reflecting the same cache-less penalty on the memory-bound fraction of the kernel.

Scaling with warp count. A key property of a GPU is the ability to hide memory latency through oversubscription: increasing the number of warps W keeps the execution units busy while outstanding memory requests drain. On `sgemm`, doubling the warp count from $W = 8$ to $W = 16$ increases our IPC from 0.94 to 1.77 — an approximately 1.9× improvement — while Vortex's IPC remains nearly flat ($2.20 \rightarrow 2.01$, already near its cache-assisted ceiling). The 1.9× improvement confirms that our implementation scales as a GPU should: it hides latency by oversubscription.

Compatibility caveat. Our `JOIN` semantics, as specified in Section 2, assumes a statically allocated join point: the join PC is fixed at compile time and recorded in the divergence-stack entry at `SPLIT` time. To be compatible with Vortex's semantics, we allow the first arriving subwarp to install its own PC as the join point at runtime. To run the Vortex benchmark binaries unmodified we made a ∼10-line modification to the scheduler that adopts this first-joiner-sets-PC convention. This change is local to the scheduler and does not affect the verification of the core or the proof of linking with the memory interface.

Summary. Taken together, the results show that our formally verified implementation achieves performance on par with the Vortex reference implementation on compute-bound workloads and that its performance gap on memory-bound workloads is attributable entirely to the absence of a cache — a design choice independent of verification. Verification does not prohibitively hurt performance.

```
; -- prologue --
0x00: LI   x12, -1       ; x12 := all-ones
0x04: TMC  x12           ; widen mask
0x08: CSRR x5,  scratch  ; read arg. addr
0x0C: LW   x10, 0(x5)    ; x10 := A
0x10: LW   x11, 4(x5)    ; x11 := B
0x14: LW   x13, 8(x5)    ; x13 := C
0x18: LW   x14, 12(x5)   ; x14 := N
; -- kernel setup --
0x1C: CSRR x1,  tid      ; i := thread id
0x20: LI   x12, -1       ; entering mask
; -- loop head --
0x24: SLT  x3,  x1, x14  ; x3 := (i < N)
0x28: PRED x3,  x12      ; narrow mask
0x2C: BR   x3,  +8       ; -> body
0x30: J    +36           ; -> epilogue
; -- loop body --
0x34: SLLI x4,  x1, 2    ; x4 := i * 4
0x38: LW   x5,  x4, x10  ; x5 := A[i]
0x3C: LW   x6,  x4, x11  ; x6 := B[i]
0x40: ADD  x7,  x5, x6   ; x7 := A[i] + B[i]
0x44: ADD  x8,  x4, x13  ; x8 := &C[i]
0x48: SW   x7,  0(x8)    ; C[i] := x7
0x4C: ADD  x1,  x1, T    ; i += T
0x50: J    -44           ; -> loop head
; -- epilogue --
0x54: TMC  0             ; retire warp
```

Figure 7.1: The vecadd kernel: a 22-instruction RISC-V program comprising a 7-instruction boot prologue (mask widening via TMC and argument loading from a parameter block at the scratch-CSR pointer), a 15-instruction kernel body (grid-stride loop with SIMT predication), and an epilogue. TMC parks the warp and reinstalls it with a specified mask; PRED narrows the active mask to in-range lanes, resetting to all-ones when all lanes have exhausted their work.

7.2 Case Study: Verified Vector Addition on a GPU

To demonstrate that the machine semantics supports end-to-end reasoning about GPU programs, we verify a complete vector-addition kernel running on the GPU. The kernel computes $C[i] = A[i] + B[i]$ for all $i < N$ using a standard grid-stride loop, and we prove in Rocq that every execution terminates, never exhibits undefined behavior, and leaves the correct result in memory.

7.2.1 The Program

The kernel is a 22-instruction assembly program, comprising a 7-instruction *boot prologue* and a 15-instruction *kernel body*. Lane t of iteration i processes element $t + i \times T$.

Prologue. The host launchees the kernel with a *single* active lane ($w = (1, 0, \texttt{0x00})$), passing a pointer to the arguments A, B, C, N through the scratch CSR. Because loads write only to active lanes, the argument registers must be loaded *after* widening the mask to all lanes. The prologue achieves this widening via the TMC instruction: it first loads the all-ones word into x12 (LI x12, -1), then executes TMC x12, which parks the warp in $\mathcal{S}.srsps$ and reinstalls it with $\mu = \neg\vec{0}$ (all lanes active). Only then is the argument-block pointer read from the scratch CSR and the four kernel arguments loaded: base pointers A (x10), B (x11), C (x13), and element count N (x14).

Kernel body. Each lane sets its initial loop index to its hardware thread identifier (`CSRR x1, tid`). A second `LI x12, -1` establishes the entering mask. The grid-stride loop then repeats: compare the index against N (`SLT`), narrow the execution mask to the in-range subset of lanes (`PRED`), branch into the body or fall through to the exit jump (`BR`/`J`), perform the per-lane load-add-store, increment the index by T, and jump back to the loop head. The `PRED` instruction is the key GPU-specific operation: it intersects the warp's execution mask with the per-lane predicate in `x3`. When the narrowed mask would be zero (all lanes have exhausted their elements), `PRED` resets it to all-ones so the subsequent `BR` falls through uniformly to the exit jump. The epilogue executes `TMC 0`, which frees the warp ID back to the scheduler and deposits the kernel's done token.

7.2.2 The Correctness Statement

Given the vectors $\vec{v}_A, \vec{v}_B \in \mathsf{Val}^N$, the `vecadd` kernel k satisfies

$$\{P(k.a, A, B, C, N, \vec{v}_A, \vec{v}_B)\}\, k\, \{\, Q(C, N, \vec{v}_A, \vec{v}_B)\,\},$$

where the three components are defined as follows. Following Verilog's indexed part-select, we write $dm[b{+}{:}\,w]$ for the w bytes of dm beginning at address b; every use below has $w = 4$, one machine word.

The kernel. $k = (im, \texttt{0x00}, a)$, where im is any instruction memory whose first 22 words decode to the `vecadd` program of Figure 7.1, and a is the argument-block pointer passed as the launch's `data` value (placed in the scratch CSR at launch, read by the boot prologue via `CSRR x5, scratch`).

The precondition. $P(a, A, B, C, N, \vec{v}_A, \vec{v}_B)(dm)$ holds when:

1. The argument block is word-aligned ($4 \mid a$) and defined in dm:

$$dm[a +{:}\, 4] = A \wedge dm[a{+}4 +{:}\, 4] = B \wedge dm[a{+}8 +{:}\, 4] = C \wedge dm[a{+}12 +{:}\, 4] = N.$$

2. The data block is word-aligned ($4 \mid A$, $4 \mid B$, $4 \mid C$) and defined in dm:

$$\forall i < N.\ dm[A{+}4i +{:}\, 4] = \vec{v}_A[i] \wedge dm[B{+}4i +{:}\, 4] = \vec{v}_B[i].$$

3. The address range $[C, C + 4N)$ is disjoint with $[A, A + 4N)$ and $[B, B + 4N)$.

4. $N + T \leq 2^{32}$ (no index wraparound).

5. The address range $[C, C + 4N)$ does not alias.

The postcondition. $Q(C, N, \vec{v}_A, \vec{v}_B)(dm)$ holds when

$$\forall i < N.\ dm[C{+}4i +{:}\, 4] = \vec{v}_A[i] + \vec{v}_B[i]$$

7.2.3 Proof

The proof proceeds by providing a loop invariant and a strictly decreasing measure.

The invariant. The invariant `vecadd_inv` is a four-way disjunction:

- **Done.** The state already satisfies the postcondition, and the scheduler is drained.
- **Running.** The warp is resident in the core at program point `pcidx` $\in \{0, \ldots, 21\}$ and iteration counter i. A per-`pcidx` table (`vecadd_pc`) pins the exact register values and execution mask at each of the 22 program points.
- **Loop-parked.** The warp has been suspended by `PRED` at `0x28` and queued in the scheduler's start queue with its narrowed mask; it will reenter at `0x2C` (`BR`).
- **Boot-parked.** The warp has been suspended by the boot `TMC` at `0x04` and queued in the start queue with the full all-ones mask; it will reenter at `0x08` (`CSRR scratch`).

The invariant also carries two global memory assertions. `mem_inv` states that the source bytes for A and B are byte-for-byte identical to the initial memory throughout execution (they are never overwritten, by the non-aliasing conditions in `vecadd_wf`). `stored_prefix` tracks the advancing write frontier: after the `SW` at `0x48` in iteration i, every word at $C + 4j$ for $j < i \times T$ holds $\vec{v}_A[j] + \vec{v}_B[j]$. At loop exit, when $i \times T \geq N$, `stored_prefix` covers $[0, N)$ and directly yields the postcondition.

The termination measure. Termination is witnessed by the lexicographic order on the triple

$$\mu(st) \;=\; (\underbrace{(N+1) - \min(N+1, i)}_{\text{remaining strides}},\; \underbrace{92 - \pi}_{\text{body phase}},\; \underbrace{\varphi}_{\text{scheduling phase}}),$$

where i is lane 0's loop index, clamped to 0 for $\pi < 36$ (the prologue and kernel setup, before `x1` holds a valid loop index) and reduced by T at $\pi = 80$ (`0x50`, the back-edge jump, so that *remaining strides* is constant through the body and drops by exactly one at the back edge), and $\varphi \in \{0, 1\}$ distinguishes running (0) from parked (1).

The three components decrease strictly in lexicographic order: a drop in *remaining strides* outranks any rise in *body phase* at the back-edge jump; within a stride, *body phase* ($= 92 - \texttt{pc}$) decreases at every straight-line step; each park-reinstall handshake (at the boot `TMC` or `PRED`) costs one unit in the *scheduling phase* while leaving the other two components unchanged.

Step lemmas. The bulk of the proof is a collection of per-transition sublemmas. For each running program point (`step_pc0`–`step_pc21`), the lemma establishes: (a) *progress* (at least one successor exists); (b) *invariant preservation* (every defined successor satisfies `vecadd_inv`); (c) *measure decrease* (μ is strictly smaller); and (d) *no UB* (the machine cannot step to `None`). Two additional lemmas handle the parked states (`step_parked` for loop-parked and its boot-parked analogue): the sole enabled transition is the Start-Step reinstall, which is total and decrements only φ.

Chapter 8

Related Work

8.1 Hardware Verification

We build upon a rich body of work on hardware verification, mostly focusing on CPU implementations featuring various optimizations. To the best of our knowledge, our work seems to be the first that tackles a parallel architecture with an explicit scheduler, with the core exposing a method to schedule new threads (`start`). We also seem to be unusual in the form of our main theorem concerning correctness across multiple kernel launches; existing proofs focus on semantics of each ISA instruction or a single program.

8.1.1 Deductive Verification Methods

We fall into the category of *deductive hardware verification*, in which a mathematical theorem about the hardware is proven. The strength of this approach is the expressivity of the specification we can give to the hardware, as we have the full range of mathematical logic at our disposal. Theorems proven about hardware using this method typically feature universal quantification over device parameters, soundness for unbounded executions, and results tied to the actual programming-language-level semantics.

Kami, Kôika, and Fjfj are a series of hardware-verification frameworks that use rule-based languages embedded in Rocq to formalize hardware descriptions [13, 8, 7]. The syntax is denoted into an LTS, and hardware verification amounts to proving that the denotation refines a specification LTS, using various notions of refinement established in the literature. Kami and Fjfj has been used to verify processor implementations against their specifications. Kôika has been used to implement MTIsolation, a hardware enclave system with formal proofs of security guarantees [23]. The processor in Kami has been used to implement end-to-end stacks ranging from the software-level semantics of programming languages down to the trace observable from bare metal [15].

Similar proofs of end-to-end systems have also been carried out by embedding Verilog into theorem provers. The CakeML project implements and proves a verified processor using a functional semantics for Verilog in Isabelle/HOL that *precisely computes* the updates the syntax should perform in a cycle given the initial register state [25]. Recent work denotes the synthesizable fragment of the IEEE semantics for Verilog into an interaction tree (itree) in

Rocq, which is an alternative formulation of LTSs, and uses the standard refinement between itrees to verify a processor against its ISA semantics [12].

8.1.2 Push-Button Verification Methods

There are also approaches to hardware verification that aim to provide more "lightweight" methods to check the properties of hardware. Such approaches involve instrumenting the source file (usually written in Verilog) with basically what is the refinement mapping of the implementation state to its specification state. SystemVerilog assertions are then inserted in appropriate places to check if desired properties hold on the instrumented state through model checking.

It has been reported that ARM is employing this method to verify that their hardware implementations uphold the semantics of each instruction according to the ARM ISA specification [17, 31]. Methods to automatically discover invariants of the implementation that imply per-instruction reachability of execution traces that violate correctness have also been demonstrated to be scalable and accurate [18, 19].

However, it is a fundamental limit of these methods that the connection between the top-level theorem they want to prove and the assertions they are tested for is still fuzzy. Unlike in the case of deductive verification, where one can precisely state in mathematical terms the property we want on the hardware, there is no formal guarantee that the properties they have proven actually correspond to their notion of correctness.

That is why we chose deductive verification: to provide a bridge with software, we want a verification methodology expressive-enough to state precisely the guarantees required by the hardware-software contract.

8.2 Memory Models

Our quest for an in-order, instantaneous specification of the processor core in isolation led us naturally to a construction that we believe would be of interest to those familiar with the literature on memory models. Non-SC memory models arise because of thread-local optimizations that are sound only under absence of other threads. In our specification, thread-locality is represented by three conditions: each instruction binds to all leaves of the tree (locally in-order), requests can be pulled out only if reachable over all branches (no dependency on previous responses), and commuting past previous requests is thread-locally sound.

Traditional weak memory models seem to be recoverable through linking a $t_a \in \mathsf{AnsTree}$ with a memory system to resolve its requests directly from the memory's response, without creating an answer node waiting for responses. The processor state would be instead a set of $(t_q, dm) \in \mathsf{QueTree} \times \mathsf{DMem}$, ignoring the CSR file and the scheduler for illustration purposes. Our proof of DRF-SC is thus two proofs that are transitively composed: one is a proof against the weak memory model involving a set of question trees, another one is a proof of DRF-SC between this weak memory model against a sequentially consistent semantics under the DRF assumption.

We believe this modularity is an interesting observation worth further investigation, since memory models are traditionally monolithic: they are stated in terms of the entire system without decomposition. We provide a survey of memory-model work of varying flavors below: it would be interesting to see if our specification can give a modular decomposition to other memory models as well.

8.2.1 Axiomatic Memory Models

Since the first definition of sequential consistency by Lamport, memory models have been specified using *events* that represent requests by each thread to the memory and a constraint on the *ordering* on those events [22]. Weak memory models have evolved to feature complex sets of relations between events that constrain the execution trace formed by those events, to the point that tooling for formally specifying custom constraints on candidate executions has been developed by major vendors such as ARM [3]. Of particular interest to us is the memory model of PTX, NVIDIA's pseudo-assembly language for GPUs [26].

Our specification is reminiscent of axiomatic memory models in that we constrain the ordering between issued requests. We connect those constraints to the actual hardware by proving refinement against the core implementation, which is novel in the context of memory models.

8.2.2 Operational Memory Models

An operational memory model presents an abstract machine that encapsulates microarchitectural detail and exposes only the behaviors relevant to the model. Such a machine enables reasoning in terms of each of its steps, which is arguably better than axiomatic models that reason about entire executions at once. Our specification can also be viewed as operational, due to its being described in terms of steps of an LTS.

I^2E is an operational model that is similar to ours in the aspect that it describes each instruction by instantaneous execution [35]. Promising semantics, an operational model that maintains an ordering between memory events by using per-location *views* (basically a vector clock), has been instantiated for ARM and RISC-V with formal proofs of correspondence with axiomatic models and an executable implementation of the abstract machine [30]. Compound memory models, which consider linking between memory models of heterogeneous devices, have also been specified in an operational fashion [16]. All of the models assume a monolithic system where the core and memory are already linked; we believe that decomposition allows better understanding of where these relaxed behaviors stem from.

Chapter 9

Conclusion

This thesis presented Mangrove, a parameterized GPU implementation described in an embedded rule-based HDL, verified in Rocq against the ISA semantics of Chapter 2, synthesized through the Bluespec toolchain, and evaluated on an FPGA. Its central technical contribution is a specification of the processor core, independent of the memory system, that executes one instruction at a time and exposes no scoreboard, register-dependency tracking, or pipeline state. Composed with a memory-system specification behind the refinement relation of Chapter 3, the core specification yields a system-level specification — a sequentially consistent operational semantics that transitions to undefined behavior on a detected data race — and the system-level refinement proof is a DRF-SC result established directly against the hardware implementation. The only reordering the implementation exhibits that sequential consistency forbids is between same-warp CSR and memory operations, motivating a race condition that forbids CSR writes while more than one warp is resident. Per-kernel correctness is stated as a Hoare triple and lifted to a theorem about the trace observed by the host across multiple kernel launches.

9.1 Limitations

- The implementation is a single core with no data cache.
- The race condition is sound but coarse: forbidding all CSR writes while multiple warps are resident is sufficient for the intended use of CSRs (reading a thread's identity, its thread mask, or a scratch register set before warps are spawned), but it also rules out barrier-synchronized cross-warp CSR communication that the hardware executes correctly.
- The arithmetic, floating-point, and branch units are parameters rather than concrete, verified implementations.
- The trusted computing base includes the toolchain used to synthesize the bitstream; our FPGA evaluation uses the open-source Bluespec compiler, which is not itself verified.

9.2 Future Work

- Extending the memory system — with a verified cache or with multiple cores sharing memory — can be done behind the existing interface without modifying the core refinement proof.
- The core specification appears to admit a compositional account of weak memory models: resolving the outstanding-request tree directly against a memory system, rather than recording an explicit answer node, yields a candidate weak memory model, and a DRF-SC proof over that model would factor our system-level result.
- The race condition could be sharpened, for instance to admit barrier-synchronized cross-warp CSR communication, by replacing the race predicate without altering the simulation relation itself.
- Kernel verification is currently manual; a program logic over the ISA semantics, or a verified compiler that produces kernels satisfying their triples, would reduce the cost of discharging per-kernel obligations at scale.

9.3 Use of Generative AI in this Work

Generative AI has been extensively used in this work to generate the proof scripts for the DRF-SC proof and the proofs for combinational modules. The specifications and simulation relations were written by hand, with corrections when AI pointed out errors that made it impossible to prove refinement. Mechanizing our proofs means that only the theorem statement has to be audited in order to trust the proofs, and the only axiom used throughout the refinement proof was functional extensionality.

The translation function from our DSL to `GolemMa`'s surface language and proofs of its correctness were also mainly written by an LLM. The function itself is almost an identity function that translates each constructor of the DSL into its corresponding `GolemMa` constructor, and the proofs were also mostly mechanical. Again, auditing the theorem statement and checking that there were no axioms used beyond what `GolemMa` assumes enable us to trust the AI-generated output.

Bibliography

[1] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering—a New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. Association for Computing Machinery, New York, NY, USA, 2–14. doi:10.1145/325164.325100

[2] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 577–591. doi:10.1145/2694344.2694391

[3] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 8 (July 2021), 54 pages. doi:10.1145/3458926

[4] Manya Bansal, Daniel Sainati, Joseph W. Cutler, Saman Amarasinghe, and Jonathan Ragan-Kelley. 2026. Modular GPU Programming with Typed Perspectives. *Proc. ACM Program. Lang.* 10, PLDI, Article 212 (June 2026), 25 pages. doi:10.1145/3808290

[5] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. 2015. The Design and Implementation of a Verification Technique for GPU Kernels. *ACM Trans. Program. Lang. Syst. (TOPLAS)* 37, 3, Article 10 (May 2015), 49 pages. doi:10.1145/2743017

[6] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andy Wright, and Adam Chlipala. 2023. Flexible Instruction-Set Semantics via Abstract Monads (Experience Report). *Proc. ACM Program. Lang.* 7, ICFP, Article 192 (Aug. 2023), 17 pages. doi:10.1145/3607833

[7] Thomas Bourgeat, Jiazheng Liu, Adam Chlipala, and Arvind. 2025. Making Concurrent Hardware Verification Sequential. *Proc. ACM Program. Lang. (PACMPL)* 9, PLDI, Article 228 (June 2025), 25 pages. doi:10.1145/3729331

[8] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the*

41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. doi:10.1145/3385412.3385965

[9] Soham Chakraborty, S. Krishna, Andreas Pavlogiannis, and Omkar Tuppe. 2025. GPUMC: A Stateless Model Checker for GPU Weak Memory Concurrency. In *Computer Aided Verification*. Springer Nature Switzerland, Cham, 321–346.

[10] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Trans. Program. Lang. Syst.* 45, 1, Article 5 (March 2023), 43 pages. doi:10.1145/3579834

[11] Zheyuan Chen, Naomi Rehman, Guido Martínez, and Tyler Sorensen. 2026. SIMT-Step Execution: A Flexible Operational Semantics for GPU Subgroup Behavior. *Proc. ACM Program. Lang.* 10, PLDI, Article 219 (June 2026), 25 pages. doi:10.1145/3808297

[12] Joonwon Choi, Jaewoo Kim, and Jeehoon Kang. 2025. Revamping Verilog Semantics for Foundational Verification. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 306 (Oct. 2025), 28 pages. doi:10.1145/3763084

[13] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang. (PACMPL)* 1, ICFP, Article 24 (Aug. 2017), 30 pages. doi:10.1145/3110268

[14] Tiago Cogumbreiro, Julien Lange, Dennis Liew Zhen Rong, and Hannah Zicarelli. 2021. Checking Data-Race Freedom of GPU Kernels, Compositionally. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, 403–426. doi:10.1007/978-3-030-81685-8_19

[15] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification Across Software and Hardware for a Simple Embedded System. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 604–619. doi:10.1145/3453483.3454065

[16] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models. *Proc. ACM Program. Lang.* 7, PLDI, Article 153 (June 2023), 24 pages. doi:10.1145/3591267

[17] Arm Holdings. [n. d.]. *Learn the architecture - A64 Instruction Set Architecture Guide 1.3*. https://developer.arm.com/documentation/102374/0103

[18] Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. 2021. Synthesizing Formal Models of Hardware from RTL for Efficient Verification

of Memory Model Implementations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 679–694. doi:10.1145/3466752.3480087

[19] Yao Hsiao, Nikos Nikoleris, Artem Khyzha, Dominic P. Mulligan, Gustavo Petri, Christopher W. Fletcher, and Caroline Trippel. 2024. RTL2MμPATH: Multi-μPATH Synthesis with Applications to Hardware Security Verification. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 507–524. doi:10.1109/MICRO61859.2024.00045

[20] Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 94–105. doi:10.1145/2804302.2804319

[21] Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. 2024. Descend: A Safe GPU Systems Programming Language. *Proc. ACM Program. Lang.* 8, PLDI, Article 181 (June 2024), 24 pages. doi:10.1145/3656411

[22] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. doi:10.1109/TC.1979.1675439

[23] Stella Lau, Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. 2024. Specification and Verification of Strong Timing Isolation of Hardware Enclaves. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS)* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 1121–1135. doi:10.1145/3658644.3690203

[24] Jiazheng Liu, Joonhyup Lee, Andres Erbsen, and Adam Chlipala. 2026. Unifying Functional Correctness and Timing in One Modular Hardware-Verification Framework. (2026). Under review.

[25] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1041–1053. doi:10.1145/3314221.3314622

[26] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 257–270. doi:10.1145/3297858.3304043

[27] R. Nikhil. 2004. Bluespec System Verilog: Efficient, Correct RTL from High-Level Specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 69–70. doi:10.1109/MEMCOD.2004.1459818

[28] NVIDIA. [n. d.]. *Parallel Thread Execution ISA Version 9.3.* https://docs.nvidia.com/cuda/parallel-thread-execution/

[29] Thibaut Pérami, Thomas Bauereiss, Brian Campbell, Zongyuan Liu, Nils Lauermann, Alasdair Armstrong, and Peter Sewell. 2026. ArchSem: Reusable Rigorous Semantics of Relaxed Architectures. *Proc. ACM Program. Lang.* 10, POPL, Article 8 (Jan. 2026), 31 pages. doi:10.1145/3776650

[30] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: a Simpler and Faster Operational Concurrency Model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3314221.3314624

[31] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of Processors with ISA-Formal. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 42–58.

[32] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. [n. d.]. *Vortex Release v2.3.* https://github.com/vortexgpgpu/vortex/releases/tag/v2.3

[33] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. 2021. Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 754–766. doi:10.1145/3466752.3480128

[34] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. doi:10.1145/3371119

[35] Sizhuo Zhang, Muralidaran Vijayaraghavan, and Arvind. 2017. Weak Memory Models: Balancing Definitional Simplicity and Implementation Flexibility. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 288–302. doi:10.1109/PACT.2017.29