Dolunay: Architectural Support for Independent Thread Scheduling in a RISC-V SIMT Accelerator

Ahmet Zahit Can
Computer Engineering, A. I. and Data Engineering
Yıldız Technical University
Istanbul, Türkiye
ahmetzahit.can@yildiz.edu.tr

Erkan Uslu
Computer Engineering
Yıldız Technical University
Istanbul, Türkiye
euslu@yildiz.edu.tr

Abstract— **Single-Instruction Multiple-Thread (SIMT) architectures have revolutionized data-parallel computing by providing a high-throughput abstraction that simplifies vector management. However, traditional stack-based SIMT models do not support intra-warp synchronization primitives such as mutexes and spin-locks. This work introduces Dolunay, a RISC-V-based Independent Thread Scheduling (ITS) SIMT accelerator. By only adding three custom instructions, Dolunay employs a cooperative multitasking model and explicit synchronization barriers at the hardware-level, and provides the forward-progress guarantees necessary to implement starvation-free algorithms. We evaluate Dolunay on the Cmod A7-35T FPGA module and demonstrate its ability to correctly execute kernels that deadlock on traditional stack-based architectures while still achieving parallel execution for conventional compute-heavy kernels.**

Keywords-Independent Thread Scheduling, SIMT, GPGPU, RISC-V, Forward-progress guarantee, Barrier synchronization, Cooperative multitasking

I. INTRODUCTION

Popularized by NVIDIA [1], Single-Instruction Multiple-Thread (SIMT) architectures employ lockstep execution, a model in which a single instruction stream drives multiple independent threads simultaneously [1], [2]. These threads are logically grouped into a warp; at the microarchitectural level, this organization allows individual threads to occupy SIMD lanes while the control logic operates at warp granularity. This abstraction maximizes computational throughput without imposing the burden of manual vector management on the programmer [1], [3]. Consequently, the SIMT model has become the architectural foundation of General-Purpose GPU (GPGPU) computing and has proven highly effective across a broad range of data-parallel workloads [4].

However, NVIDIA's pre-Volta SIMT designs were unable to support intra-warp synchronization primitives such as mutexes. To address this limitation, NVIDIA introduced Independent Thread Scheduling (ITS) in the Volta architecture, thereby enabling support for such primitives [5], [6].

The SIMT design philosophy has also been applied to RISC-V in several implementations, bringing SIMT computing to the RISC-V ecosystem [7]–[10]. However, none of these designs provide sufficient forward-progress guarantees required to support the synchronization primitives discussed above. This work introduces Dolunay, a RISC-V-based ITS SIMT accelerator. Each path in the Dolunay architecture has its own PC value, and the RISC-V ISA is extended with a custom extension that supports cooperative multitasking and explicit reconvergence via synchronization barriers, thereby enabling a variety of synchronization primitives. Consequently, starvation-free algorithms can be implemented on Dolunay. To the best of our knowledge, this is the first RISC-V SIMT accelerator to provide this capability. The source code for Dolunay is published at https://github.com/ahmetzahitcan/dolunay, licensed under Apache License 2.0. The exact variant described in this paper is found under branch `arch__dsd2026`.

II. RELATED WORK

SIMT architectures organize threads into warps, with each warp sharing a single instruction stream. A primary challenge in such architectures is branch divergence, in which conditional branches cause threads within a warp to follow different execution paths. Because the hardware employs a shared control unit, it must serialize these paths by executing them sequentially while masking lanes that are inactive on the current path. Ideally, once the paths reconverge at a common instruction, all lanes are unmasked to restore full SIMD throughput [1], [2], [4], [11].

In pre-Volta NVIDIA architectures [5], reconvergence was managed using an Immediate Post-Dominator (IPDOM) stack model. This mechanism uses a hardware stack to track divergence events; because nested branches may occur before an earlier branch reconverges, the stack preserves the execution state associated with each level of divergence. Each stack entry typically stores the active mask for a divergent path together with the address of its reconvergence point, that is, the IPDOM. When execution reaches a reconvergence point, the corresponding entry is popped and the previous mask is restored, allowing the warp to resume unified execution [1], [11]. Beyond NVIDIA architectures, this model has also been implemented in several RISC-V SIMT accelerators, including Vortex [7] and e-GPU [8].

However, a significant limitation of the stack-based model is its inability to support intra-warp synchronization primitives,

Manuscript received May 11, 2026; revised July 15, 2026; accepted August 21, 2026; published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.137

such as mutexes and spin-locks. Because divergent paths execute serially under a shared warp execution context, a thread holding a lock may be inactive while another thread in the same warp spins while waiting for it, resulting in deadlock [5]. To address this limitation, NVIDIA transitioned to ITS beginning with the Volta architecture [5]. In the ITS model, threads maintain independent execution state, including separate program counters and call stacks, which allows the scheduler to make forward-progress decisions at thread granularity [5], [6].

This design allows the hardware to schedule threads more flexibly by dynamically grouping those with matching PCs while avoiding the lockstep deadlock scenarios associated with stack-based SIMT execution. The shift fundamentally strengthens the programming model by enabling fine-grained synchronization and more complex data structures within a single warp [5].

Beyond stack-based and ITS models, a third approach is transparent, or microarchitectural, SIMT, in which SIMT execution is implemented entirely below the ISA level. This approach is exemplified by Simty [9], SIMTIGHT [10], and SIMT-X [12]. It relies on dynamic hardware detection of divergence and reconvergence without requiring explicit software annotations, thereby enabling the SIMT execution of general-purpose binaries [9]. Although this approach shares the use of per-thread PCs with the ITS model, the aforementioned implementations do not support the synchronization primitives discussed above.

Dolunay was initially inspired by transparent SIMT architectures. Early designs explored combining implicit hardware reconvergence with preemptive scheduling to support intra-warp forward progress. However, efficiently detecting reconvergence among independently scheduled execution contexts required substantial additional hardware complexity, and unconstrained scheduling policies could lead to suboptimal reconvergence rates, thereby incurring significant performance overhead. Consequently, Dolunay adopts an architectural ITS approach based on cooperative scheduling and explicit synchronization barriers for reconvergence.

III. Methodology

A. Instruction Set

Dolunay implements a variety of standard extensions as well as the custom Xdolunay extension on top of the RV32I base.

TABLE I. XDOLUNAY CUSTOM INSTRUCTIONS

Instruction	Encoding	Description
`BINIT imm(rs1)`	I-format (opcode = `0xB`, funct3 = `1`, rd = `x0`)	Initializes a barrier in an aligned 4-byte region of work RAM.
`BSYNC imm(rs1)`	I-format (opcode = `0xB`, funct3 = 2, rd = `x0`)	Updates the barrier in an aligned 4-byte region of work RAM and either parks the calling threads or releases the barrier.
`YIELD`	Word (`0x0000400B`)	Yields control to another path.

The standard extensions Zicsr and Zalrsc are mandatory. Xdolunay depends on Zicsr for custom CSRs, while Zalrsc provides the LR/SC operations required to implement synchronization primitives. Although Xdolunay supports independent thread execution with per-thread program counters, it does not introduce synchronization primitives beyond those already provided by Zalrsc.

The standard extensions Zicond and Zba were included to improve performance for operations expected to be common in Dolunay's target workloads. Zicond reduces the need for short conditional branches, mitigating some forms of divergence and thereby reducing, though not eliminating, the need for explicit convergence through barriers, as detailed in Section III-A1. Zba improves address-generation efficiency for accesses indexed by warp and/or thread identifiers.

The standard extensions Zicntr and Zihpm are used for performance monitoring. The hardware performance monitors (HPMs) are described in detail in Section III-A3.

1) Custom Instructions

RISC-V SIMT architectures differ in whether their divergence and reconvergence mechanisms are implicit or explicit. In some architectures, such as Simty [9] and SIMTIGHT [10], these mechanisms are implicit, meaning that they require no software annotation. Divergence occurs automatically on non-uniform branch and jump instructions. For reconvergence, paths are sorted by priority, and reconvergence occurs when the active path (with the highest priority) and the next path (with the second-highest priority) are detected to have the same PC. The priority-sorting mechanism ensures that this condition is sufficient to cover all reconvergence cases. Together, these features enable microarchitectural SIMT, allowing the hardware to execute traditional SPMD programs in SIMT fashion [9].

In other architectures, such as Vortex [7] and e-GPU [8], these mechanisms are explicit and must be marked using custom instructions, either by the programmer or the compiler. Failure to mark these points may result in incorrect execution. Vortex and e-GPU both use the same custom extension [8], which supports an IPDOM-stack-based approach in which all threads within a warp share a single PC. These custom instructions manipulate the IPDOM stack and the active thread mask, and all branches and jumps are assumed to be uniform across the active thread mask [7].

Dolunay adopts an intermediate approach: divergence is implicit, whereas reconvergence is explicit. Similar to Simty [9] and SIMTight [10], Dolunay makes no assumption of uniformity for branches and jumps and automatically splits paths when a non-uniform branch or jump is detected. However, because Dolunay allows threads to yield control to one another, it has no notion of path priority. As a result, implicit reconvergence detection would require checking whether the active path shares a PC with any inactive path, which would introduce significant hardware complexity. Therefore, Dolunay requires all reconvergence points to be explicitly annotated in software using custom instructions.

To achieve this, Dolunay adds 3 custom instructions: BINIT, BSYNC and YIELD. Encoding and a brief description for these instructions is given in Table I.

Dolunay uses warp-level structures called barrier contexts to support reconvergence via synchronization barriers. Barrier contexts are stored in memory, and both BINIT and BSYNC therefore perform memory accesses. Unlike branches and jumps, Dolunay assumes uniformity for barrier context addresses; violating this assumption results in undefined behavior. Storing the barrier contexts in memory was preferred over dedicated registers to avoid the need to spill and restore warp-level state. As described in Section III-C, the programming model uses unmodified RISC-V GCC and inserts the custom instructions via .insn assembler directives. Because the toolchain is not warp-aware, spilling and restoring such registers would be particularly cumbersome.

The barrier context contains two thread masks. The first is the participation mask. For each bit, a value of 1 indicates that the corresponding thread participates in the synchronization barrier and that the other participating threads must wait for it, whereas a value of 0 indicates that the thread does not participate in that barrier. The second is the parked mask. For each bit, a value of 1 indicates that the corresponding thread has parked at the barrier and is waiting for the remaining participating threads, whereas a value of 0 indicates that the thread has either not yet parked at that barrier, if its participation bit is 1, or does not participate in that barrier, if its participation bit is 0.

BINIT initializes the barrier context. The participation mask is set to the active thread mask, meaning that synchronization applies to the threads that execute BINIT simultaneously. The parked mask is initialized to all zeros.

BSYNC performs synchronization using the barrier context. Its operation proceeds as follows:

1) **Compute the new barrier context:** The new parked mask is computed as the bitwise OR of the active thread mask and the previous parked mask. This value is always written back to memory.
2) **Evaluate the new barrier context:** The new parked mask is compared with the participation mask. Depend- ing on the result, either step 3 or step 4 is executed.
3) **Release the barrier:** If the condition in step 2 evaluates to true, all parked threads are merged into the current path.
4) **Park the threads:** If the condition in step 2 evaluates to false, the current path is deactivated, and the next active path is selected using the round-robin scheduler, which is the same mechanism used by YIELD.

The algorithm for BSYNC is given in Fig. 1. A notable consequence of this implementation is that the reconvergence point itself is not stored in the barrier context. Instead, it is encoded implicitly by the location of BSYNC in the instruction stream. One implication is that if multiple BSYNC instructions use the same barrier context, an individual thread may park at one BSYNC instruction and later resume at another. This behavior is not currently used by any of the software developed for this architecture.

1: **procedure** BSYNC(addr)
2: ▷ T is the number of threads per warp
3: $m_{parked} \leftarrow MEM[addr][T-1:0]$
4: $m_{participation} \leftarrow MEM[addr][2T-1:T]$
5: $m_{parked_new} \leftarrow m_{parked} \vee m_{active}$
6: **if** $m_{parked_new} = m_{participation}$ **then**
7: $m_{active} \leftarrow m_{participation}$
8: **else**
9: $m_{active} \leftarrow 0$
10: YIELD
11: **end if**
12: $MEM[addr][T-1:0] \leftarrow m_{parked_new}$
13: **end procedure**

Figure 1. BSYNC algorithm

YIELD switches the active path using a round-robin scheduler.

2) CSRs

In addition to the three custom instructions, Xdolunay defines three custom CSRs. Their names, addresses, and brief descriptions are given in Table II.

All three CSRs are read-only. `xwarpid` and `xthrid` are constant for a given thread. The standard CSR `mhartid` is also implemented. The lower bits of `mhartid` are equal to `xthrid`, and the remaining bits are equal to `xwarpid`.

`xrole` can be used to determine whether the current threadis the leader of the active path. The leader is the thread with the lowest ID among those enabled in the active thread mask. Every path has exactly one leader and zero or more follower threads.

All other CSRs are implemented as read-only zero.

3) Hardware Performance Monitors

Dolunay implements the three standard HPMs together with two custom HPMs. The standard HPMs `cycle` and `time` are implemented as shadows and are read from a counter that increments every cycle. The standard HPM `instret` indicates the number of instructions retired by the current thread.

TABLE II. XDOLUNAY CUSTOM CSRS

CSR	Address	Description
`xwarpid`	`0xCC0`	Warp identifier
`xthrid`	`0xCC1`	Thread identifier within the warp. Threads from different warps may have the same xthrid value.
`xrole`	`0xCC2`	Indicates whether the thread is a leader (0) or a follower (1).

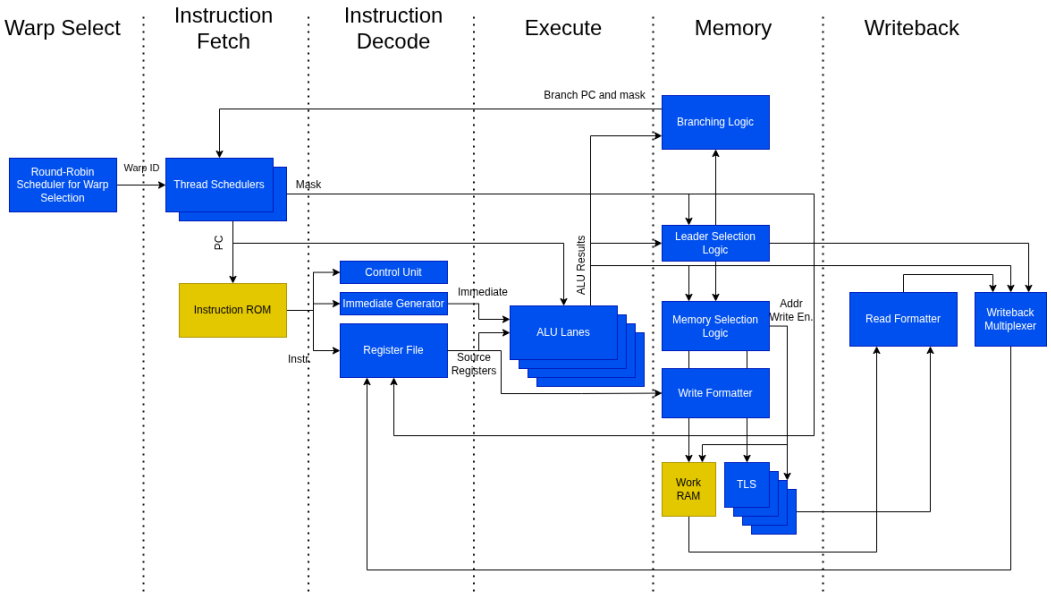


Figure 2. Dolunay core. Yellow modules are external to the core.

The custom HPM `wtinstret`, mapped to `hpmcounter4`, is a warp-level counter that indicates the total number of instructions executed by the warp. It is always equal to the sum of the `instret` values of all threads within the warp.

The custom HPM `wuinstret`, mapped to `hpmcounter3`, is a warp-level counter that indicates the number of instructions executed uniquely by the warp. Simultaneous executions of an instruction are counted as a single unique execution of that instruction. In the absence of divergence, `wuinstret` is equal to `wtinstret` divided by the number of threads warp.

B. Microarchitecture

Similar to other SIMT architectures [1], Dolunay is designed to execute multiple threads in SIMD fashion. A group of co-executing threads is referred to as a warp. Dolunay also interleaves the execution of multiple warps. As detailed in Section III-B4, four is the minimum supported number of warps, although a larger number may also be selected before synthesis. Likewise, the number of threads per warp can be chosen before synthesis, provided that all warps use the same thread count.

As the threads in a warp diverge, they are divided into subgroups called paths. As per the ITS design, paths run using a cooperative multitasking design, allowing the successful execution of starvation-free algorithms. Paths are managed by the thread schedulers, the number of thread schedulers is equal to the number of warps. The implementation of the thread schedulers is detailed in Section III-B2.

A simplified overview of the core is given in Fig. 2.

1) Pipeline

The pipeline has 6 stages: Warp Select, Instruction Fetch, Instruction Decode, Execute, Memory and Writeback.

1. **Warp Select:** This stage contains a round-robin scheduler that selects the next warp to run.
2. **Instruction Fetch:** During this stage, the relevant thread scheduler is run, selecting a path to execute. The PC of the selected path is sent to the Instruction ROM to fetch the instruction. The mask of the selected path is also pushed into the pipeline.
3. **Instruction Decode:** This stage contains the control unit. The control unit generates the control signals based on the instruction read from the ROM. The source register indices are simultaneously used to read from the register file. The register file is synchronous and provides two read ports and one write port. To implement this structure using FPGA BRAMs, we employ a shadowing technique in which two BRAMs, each with one read port and one write port, share a common write interface.
4. **Execute:** This stage contains the ALUs for computation. The number of ALU lanes is equal to the number of threads per warp.
5. **Memory:** This stage contains several units running in parallel. As the name implies, memory accesses happen in this stage. The leader selection and coalescing logic used for memory accesses are reused for JALR and branching. Divergence is detected and the relevant thread scheduler is notified, updating its paths. Instruction retired and instruction replay signals are also generated in this stage, consumed by the relevant thread scheduler and the hardware performance monitors.

6. **Writeback:** This stage is used for writing the results into the register file.

2) Thread Schedulers

Thread schedulers are hardware units that manage thread divergence and convergence. Each warp is assigned a dedicated scheduler, which groups threads into paths. A path is a simple data structure that contains a PC and a thread mask with at least one active thread. At startup, there is only one path, which contains all threads and has a PC value of zero. Divergence increases the number of paths, up to a maximum equal to the number of threads per warp, corresponding to the worst-case scenario in which each thread has a different PC. Reconvergence decreases the number of paths, thereby allowing more threads to execute in parallel.

It is worth noting that, at the microarchitectural level, Dolunay does not assign a unique PC to each thread, but rather to each path. However, because Dolunay correctly handles the worst-case scenario of one thread per path, this distinction is not architecturally visible.

Upon receiving a branch signal, the scheduler determines whether divergence has occurred. If the branch is non uniform, the threads that take the branch are partitioned into a new path. For uniform branches, in which all active threads follow the same trajectory, the PC of the current path is simply updated.

As explained in Section III-A1, Dolunay allows software to switch to a different path. This is achieved by a yield signal generated when the YIELD custom instruction reaches the Memory stage. The signal is qualified by the warp ID associated with that stage. When the signal is asserted, the scheduler performs a context switch to the next active path within the warp, thereby enabling algorithms that require a forward-progress guarantee, which is a key focus of this work. The next active path is selected by a modified round-robin scheduler that skips inactive paths.

The scheduler receives two input signals related to instruction completion: an instruction-complete signal and an instruction replay mask. When the instruction-complete signal is asserted, the replay mask is checked to determine whether any threads must replay the instruction. If so, the instruction is replayed using a mask that includes only those threads; otherwise, the PC of the active path is incremented.

The BSYNC instruction, described in Section III-A1, executes in two passes. A single-bit internal register is maintained for each warp to track the current pass. In the first pass, the barrier context is loaded from memory into an internal register. In the second pass, that register is used either to park the threads or to release the barrier. In both cases, the active thread mask is updated, either to zero or to the participation mask stored in the barrier context. In the parking case, control is additionally yielded to another path.

3) Memory

Dolunay separates the address space into three regions: Instruction ROM (IROM), Work RAM (WRAM), and Thread-Local Storage (TLS). The corresponding address ranges for these three regions are given in Table III.

TABLE III. MEMORY REGIONS

Memory Region	Start Address	End Address
IROM	`0x00000000`	`0x3FFFFFFF`
WRAM	`0x40000000`	`0x7FFFFFFF`
TLS	`0x80000000`	`0xFFFFFFFF`

To determine which memory region is being accessed, the pipeline checks the highest two bits of the address. Although the current configuration allocates 1 GiB to IROM, 1 GiB to WRAM, and 2 GiB to TLS, the corresponding memory units are not large enough to cover the full ranges. Therefore, the memory is effectively shadowed because the upper address bits are ignored.

WRAM has a 32-bit-wide read-write port with byte-write enable. For each access to WRAM, a leader thread is selected to provide the address. The leader's operation always succeeds. Any follower thread whose address differs from the leader's replays the operation. Any follower thread whose address matches the leader's completes the operation simultaneously with the leader. For load operations, this means that such threads read the same value as the leader. For store operations, writes from non-replayed follower threads are ignored. This choice is architecturally valid because it is indistinguishable from the case in which the leader is the last thread to perform its store.

IROM has two 32-bit-wide read-only ports. One of these ports is used to fetch instructions, and only the IROM region is executable. The other port is used to load constants. As with WRAM, a leader thread is selected to provide the address. The leader's load operation always succeeds. Any follower thread with a matching address completes its load simultaneously with the leader. Other follower threads replay. Stores to IROM are ignored.

TLS instances are per-thread read-write memories. Each execution lane has an independent TLS with a 32-bit-wide read-write port with byte-write enable. The warp ID is appended to the address to ensure that threads with the same thread ID in different warps do not access one another's TLS. TLS accesses always complete in a single pass.

Because different threads can generate different addresses, different threads may access different regions. IROM and WRAM accesses use the same leader-selection logic, which selects only one leader, so only one of IROM or WRAM can be accessed in a single cycle. Threads accessing TLS are exempt from leader selection.

LR/SC operations are valid only on WRAM. LR reserves the entire WRAM region, and SC invalidates the reservations held by other threads. When multiple threads execute SC simultaneously, a leader thread is selected to perform the operation. All other threads report failure by storing 1 in `rd`. SC instructions never replay.

This memory design, although simple, is inefficient. As shown in Section IV-B, the extensive serialization of memory

accesses significantly degrades performance. We explored more efficient memory implementations, but implementing them within the limited area and BRAM resources of the Cmod A7-35T proved cumbersome. We therefore chose this simpler, less efficient implementation, as the primary goal of Dolunay is to serve as a proof of concept for a RISC-V ITS SIMT architecture rather than as a high-performance soft-core accelerator.

4) FGMT

Dolunay avoids the need for complex dependency-tracking hardware by employing Fine-Grain Multithreading (FGMT), which interleaves instructions from different warps [13]. Although the Dolunay pipeline has six stages, four warps are sufficient to ensure correct execution. The argument is as follows.

Consider instruction P entering the pipeline at cycle N . Because the pipeline issues another instruction from the same warp only after W cycles, where W is the number of warps, instruction P + 1 enters the pipeline at cycle N + W . The progression of these two instructions through the pipeline, together with the corresponding cycle numbers, is shown in Table IV.

The following hazards must be avoided:

1. **Hazard #1 — Register-file write before read:** The register write of instruction P at the Writeback stage, which occurs in cycle N + 5, must complete before the register read of instruction P + 1 at the Instruction Decode stage, which occurs in cycle N + W + 2.
2. **Hazard #2 — Thread-scheduler state update:** The thread-scheduler state update of instruction P at the Memory stage, which occurs in cycle N + 4, must complete before the thread scheduler generates the PC and mask for instruction P + 1 at the Instruction Fetch stage, which occurs in cycle N + W + 1.
3. **Hazard #3 — BSYNC toggle:** As explained in Section III-B2, the BSYNC instruction executes in two passes, and a register is toggled to track the current pass. This register must be updated at the Memory stage in cycle N + 4 before the corresponding control signals are generated at the Instruction Decode stage in cycle N + W + 2.

Therefore, the following inequalities must hold:

$$N + W + 2 > N + 5 \Rightarrow W > 3$$

$$N + W + 1 > N + 4 \Rightarrow W > 3$$

$$N + W + 2 > N + 4 \Rightarrow W > 2$$

TABLE IV. PIPELINE CYCLE NUMBERS FOR INSTRUCTIONS

Inst.	WS	IF	ID	EX	MEM	WB
P	N	N+1	N+2	N+3	N+4	N+5
P+1	N+W	N+W+1	N+W+2	N+W+3	N+W+4	N+W+5

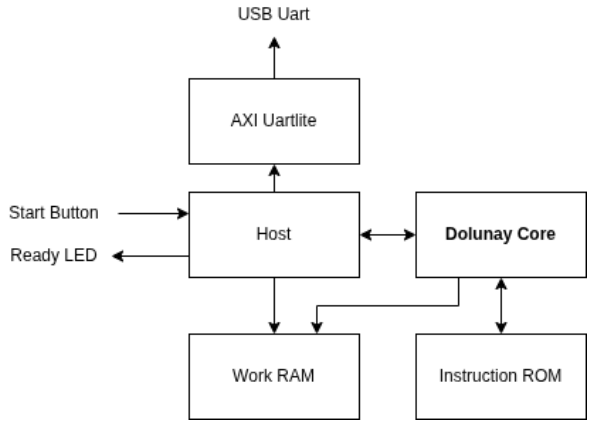


Figure 3. Testbench organization and hardware integration on the Cmod A7-35T FPGA.

Hence, W ≥ 4 is required to avoid hazards. When W = 3, hazards #1 and #2 occur; when W < 3, all three hazards occur.

C. Programming Model

For the programming model, we implemented a minimal C toolchain. Custom instructions are inserted into C code via asm directives, and because the assembler is also unaware of these custom instructions, .insn directives are used to instantiate them. A simple linker script maps ELF sections to memory regions, and a crt0.s file initializes the stack pointer, copies globals from IROM to WRAM or TLS, and initializes two BSS sections, one in WRAM and the other in TLS. We used unmodified RISC-V GCC to generate ELF objects and converted them to binaries using the objdump tool.

IV. RESULTS

To test the core, we connect it to a simple FSM host via a start/ready mechanism. The FSM host also drives AXI Uartlite llowing to print messages and dump memory. A diagram of the design is given in Fig. 3. This design as a whole is then synthesized and implemented for Cmod A7-35T using Vivado 025.2 toolchain. Flow PerfOptimized high and Performance Explore strategies are used.

A. Area and Frequency

We synthesized our testbench module with warp counts of four, eight, and sixteen; thread-per-warp counts of four, six, eight, and twelve; and TLS sizes of 512, 1024, 2048 and 4096 bytes. To name the various configurations, we use the following three-character format:

1. **v vs V vs W:** v indicates 4 warps, V indicates 8 warps, W indicates 16 warps.
2. **t vs s vs T vs S:** t indicates 4 threads per warp, s indicates 6 threads per warp, T indicates 8 threads per warp, S indicates 12 threads per warp. Admittedly, thread-per-warp counts of six and twelve are problematic because these values are not powers of two. However, higher powers of two

failed to implement, so we evaluated these configurations to examine how area scales with the number of threads per warp.

3. **h vs 1 vs 2 vs 4:** h indicates 512 bytes per TLS, 1 indicates 1024 bytes per TLS, 2 indicates 2048 bytes per TLS, 4 indicates 4096 bytes per TLS.

The post-implementation reported LUT, FF and BRAM usages as well as WNS and Fmax for each successful combination is given in Table V. All configurations were implemented with a 50MHz clock rate, Fmax is theoretical and calculated from WNS. Graphs for area scaling according to warp/thread configuration, for TLS size of 512 bytes, are given in Fig. 4.

We also tested whether changing the base ISA to RV32E could reduce BRAM usage by shrinking the register file; however, this was not beneficial except at extremely high warp counts. Because increasing the warp count raises area usage without improving IPC, we chose RV32I instead.

We selected the vT2 configuration for the performance benchmarks. We increased the IROM size to 8 KiB and the WRAM size to 32 KiB, which fully utilized the BRAM resources of the Cmod A7-35T in this configuration. Compute-heavy Kernels

We used four compute-heavy kernels to evaluate Dolunay: vecadd, saxpy, nearn, and gemm. For each warp, we extracted the wtinstret and wuinstret values described in Section III-A3 and calculated their ratio. This ratio is referred to as Lane Utilization Ratio, or LUR, and the obtained values are shown in Fig. 5a.

TABLE V. POST-IMPLEMENTATION RESOURCE USAGE AND PERFORMANCE CHARASTERICTICS ACROSS DOLUNAY CONFIGURATIONS

Config.	LUT	FF	BRAM	WNS	Fmax
vth	5594	3161	8	2.983ns	58.76MHz
vt1	5597	3161	10	2.446ns	56.96MHz
vt2	5599	3161	14	3.808ns	61.75MHz
vt4	5600	3161	22	3.333ns	59.99MHz
vsh	8746	4237	11	2.194ns	56.16MHz
vs1	8747	4237	14	1.709ns	54.67MHz
vs2	8742	4237	20	3.037ns	58.95MHz
vs4	8751	4235	32	2.612ns	57.51MHz
vTh	11167	5335	14	2.444ns	56.96MHz
vT1	11180	5335	18	2.922ns	58.55MHz
vT2	11230	5337	26	2.955ns	58.66MHz
vT4	11221	5339	42	1.758ns	54.81MHz
vSh	16924	7633	20	0.898ns	52.35MHz
vS1	19634	7633	26	1.124ns	52.97MHz
vS2	17007	7637	38	0.514ns	51.31MHz
Vth	7273	5336	10	2.126ns	55.94MHz
Vt1	7274	5336	14	1.883ns	55.19MHz
Vt2	7286	5336	22	3.108ns	59.19MHz
Vt4	7283	5345	38	2.722ns	57.87MHz
Vsh	11267	7287	14	1.989ns	55.52MHz
Vs1	11369	7308	20	0.948ns	52.48MHz
Vs2	11367	7281	32	1.763ns	54.83MHz
VTh	14412	9282	18	2.517ns	57.19MHz
VT1	14452	9284	26	1.869ns	55.15MHz
VT2	14436	9288	42	2.504ns	57.15MHz
Wth	10440	9667	14	2.291ns	56.46MHz
Wt1	10433	9671	22	2.002ns	55.56MHz
Wt2	10440	9675	38	1.424ns	53.82MHz
Wsh	16220	13410	20	0.638ns	51.64MHz
Ws1	16215	13416	32	1.400ns	53.76MHz

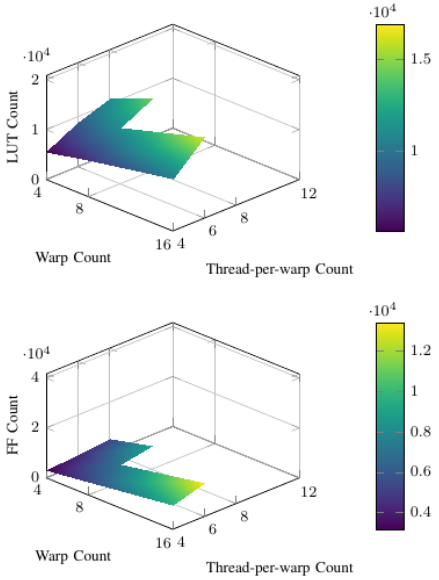


Figure 4. Scaling of LUT and FF counts according to warp/thread configuration

We also calculated the instructions-per-cycle value by dividing `wtinstret` by the cycle count. The cycle count was obtained by measuring the time required for Dolunay to assert ready after receiving the start signal, and it may therefore differ from the cycle counter. The resulting values are shown in Fig. 5b.

As explained in Section III-B3, we were unable to implement an efficient memory system because of the limitations of the Cmod A7-35T. All non-uniform, non-TLS memory accesses must be serialized, which significantly reduces IPC.

B. Concurrecy Stress Kernels

To demonstrate that Dolunay can correctly execute kernels that require a forward-progress guarantee, we designed two microkernels that cannot run on IPDOM-stack-based SIMT hardware.

The first is called ping-pong. Threads are grouped into pairs that share an accumulator; one thread executes the ping function, and the other executes the pong function. Both functions contain a loop that terminates once the accumulator reaches a predetermined value, which is 128 in our test case. Within the loop, both functions check whether the accumulator is currently even or odd. The ping function increments it by one if it is even and yields if it is odd. The pong function increments it by one if it is odd and yields if it is even. Without yielding, both functions would loop indefinitely while waiting for the value to change. Therefore, Dolunay's ability to complete this microkernel demonstrates that it can yield control between diverging threads.

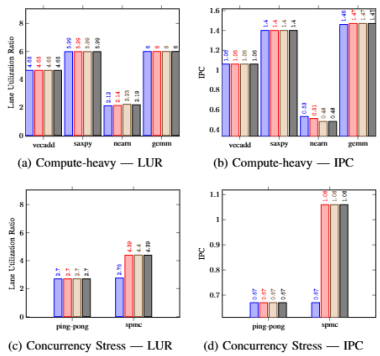


Figure 5. Performance metrics for all kernels

The second is called spmc, which stands for Single-Producer Many-Consumers. One thread, selected globally across all warps rather than per warp, acts as the producer and reads values from an array into a shared variable. The consumer threads copy the value in the shared variable into their own arrays, thereby creating copies of the original array. The producer thread must wait until all consumer threads have completed their reads before it can write the next value, and all consumer threads must wait for the producer to write the data. This microkernel is intended to stress-test Dolunay's synchronization capabilities.

As above, we measured the LUR and IPC values, which are shown in Fig. 5c and Fig. 5d. Although both kernels ran to completion, they exhibited lower performance than the compute-heavy kernels. This is expected, because compute-heavy kernels rely on parallelism to utilize as many ALU lanes as possible. Indeed, the ping-pong kernel is limited to a maximum lane-utilization rate of four even in the best-case scenario, because in each warp half of the threads execute the ping function and the other half execute the pong function. The other kernel, spmc, showed that the warp containing the producer thread, which therefore executes two roles at the same time, performed significantly worse than the all-consumer warps, thereby clearly demonstrating the effect of running heterogeneous kernels on SIMT performance.

V. Conclusion

In this paper, we presented Dolunay, a RISC-V SIMT accelerator designed to overcome the synchronization limitations of traditional stack-based models. By implementing an Independent Thread Scheduling (ITS) architecture supported by three custom instructions, we demonstrated that cooperative scheduling and explicit barrier-based synchronization can provide an intra-warp forward-progress guarantee. Our results show that Dolunay successfully executes kernels that rely on this guarantee.

The Dolunay core presented here is primarily intended as a proof of concept. Future work should extend it into a more practical design. Most importantly, a more efficient memory system should be developed, support for floating-point arithmetic should be added, a more extensive software stack such as OpenCL should be implemented, and the design should be evaluated with more lanes on larger FPGAs. Another promising direction is to combine Dolunay's innovations with those of other designs. [10] showed that dynamic scalarization can substantially reduce on-chip memory usage. [14] introduced a novel floorplan solver to support more efficient permutation operations. Finally, [15] implemented a texture unit for Vortex, accelerating 3D applications. These ideas could be combined with ITS capabilities in more advanced GPGPU designs.

References

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," Micro, IEEE, vol. 28, pp. 39 – 55, 04 2008.

[2] J. Nickolls and W. Dally, "The gpu computing era," Micro, IEEE, vol. 30, pp. 56 – 69, 05 2010.

[3] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," Queue , vol. 6, pp. 40–53, 03 2008.

[4] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," Proceedings of the IEEE , vol. 96, pp. 879–899, 05 2008.

[5] NVIDIA Corporation, "Nvidia tesla v100 gpu architecture: The world's most advanced data center gpu," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, accessed: 2026-03-28.

[6] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," 2018. [Online]. Available: https://arxiv.org/abs/1804.06826

[7] F. Elsabbagh, B. Tine, P. Roshan, E. Lyons, E. Kim, D. E. Shim, L. Zhu, S. K. Lim, and H. kim, "Vortex: Opencl compatible risc-v gpgpu," 2020. [Online]. Available: https://arxiv.org/abs/2002.12151

[8] S. Machetti, P. D. Schiavone, L. Orlandic, D. Huang, D. Kasap, G. Ansaloni, and D. Atienza, "e-gpu: An open-source and configurable risc-v graphic processing unit for tinyai applications," 2026. [Online]. Available: https://arxiv.org/abs/2505.08421

[9] C. Collange, "Simty: generalized SIMT execution on RISC-V," in First Workshop on Computer Architecture Research with RISC-V , ser. First Workshop on Computer Architecture Research with RISC-V, vol. 6, Boston, United States, Oct. 2017, p. 6. [Online]. Available: https://inria.hal.science/hal-01622208

[10] M. Naylor, A. Joannou, A. Markettos, P. Metzger, S. Moore, and T. Jones, "Advanced dynamic scalarisation for risc-v gpgpus," 11 2024, pp. 260–267.

[11] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," 01 2008, pp. 407–420.

[12] A. Tino, C. Collange, and A. Seznec, "Simt-x: Extending single-instruction multi-threading to out-of-order cores," ACM Trans. Archit. Code Optim. , vol. 17, no. 2, May 2020. [Online]. Available: https://doi.org/10.1145/3392032

[13] M. Nemirovsky and D. M. Tullsen, Fine-Grain Multithreading . Cham: Springer International Publishing, 2013, pp. 25–31. [Online]. Available: https://doi.org/10.1007/978-3-031-01738-4_4

[14] J. Liu, Q. Li, Y. Luo, H. Zhang, J. Lu, S. Fan, J. Ye, Y. Liu, X. Liu, Y. Yang, Z. Ye, Y. Zeng, A. Shen, R. Huang, W. Cong, X. Zou, and M. Gao, "Titan-i: An open-source, high performance risc-v vector core," in Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture , ser. MICRO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 675–690. [Online]. Available: https://doi.org/10.1145/3725843.3756059

[15] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, "Vortex: Extending the risc-v isa for gpgpu and 3d-graphics," in MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture , ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 754–766. [Online]. Available: https://doi.org/10.1145/3466752.3480128