Parallel Computer Architecture
Simple GPGPU



Nikas Ioannis Iason **Tsogkas Panagiotis Nikolaos** **Tsiantos Dimitrios**

June 2026

Contents

1 Introduction **4**

2 Project Goals and System Overview **4**
- 2.1 Design Goals . . . 4
- 2.2 System Overview . . . 5
- 2.3 Setup & tools . . . 6

3 Hardware Architecture **6**
- 3.1 Top Level System and Host Integration . . . 6
- 3.2 The Streaming Multiprocessor (Shared frontend) . . . 6
- 3.3 The Streaming Processor . . . 6
  - 3.3.1 ALU and Multiplier Pipeline . . . 6
  - 3.3.2 Data Hazards and Forwarding . . . 7
- 3.4 Memory Arbitration: The Nx2 Crossbar . . . 7
  - 3.4.1 Global Stall Logic and Request Accumulation . . . 7
  - 3.4.2 Atomic Memory Operations (Banked Design) . . . 7

4 SoC and Host Integration **8**
- 4.1 Integration Goals . . . 8
- 4.2 System-on-Chip Design Overview . . . 8
- 4.3 Host Controller . . . 9
- 4.4 Host Command Processor . . . 9
- 4.5 Synthesis and Implementation . . . 11
- 4.6 Initial Integration Attempt Using UART in the PL . . . 11
- 4.7 Integration Using the Zynq Processing System . . . 12
- 4.8 PS-to-PL Communication Through AXI GPIO . . . 13
- 4.9 PS Host Software . . . 15

5 Software Stack and Runtime **15**
- 5.1 Program Build Flow . . . 15
- 5.2 Program Structure . . . 17
- 5.3 Memory Layout . . . 17
- 5.4 GPGPU Runtime . . . 18
- 5.5 Kernel Arguments . . . 18
- 5.6 Generic FPGA Runner . . . 19
- 5.7 Program Adapters . . . 19
- 5.8 UART Host Interface . . . 20
- 5.9 Example: N-Body Software Flow . . . 20
- 5.10 Summary . . . 21

6 Applications and Demo Workloads **21**
- 6.1 N-Body Gravitational Simulation . . . 21

7 Verification Methodology **21**
- 7.1 Automated Test Suite Ecosystem . . . 21
- 7.2 Program Safety and Constrained Random Verification . . . 22
- 7.3 Aggressive Hardware Fuzzing . . . 22
- 7.4 Top-Level End-to-End Test Extension . . . 22
- 7.5 Implementation-Compatible Testing . . . 23
- 7.6 Final FPGA End-to-End Testing . . . 23

8 Results and Evaluation **24**
8.1 Functional Validation . . . 24
8.2 Peak Compute Throughput . . . 24
8.3 N-body Performance Comparison . . . 25
8.4 Memory Bandwidth . . . 25
8.5 Host Communication Bottleneck . . . 26
8.6 Resource Utilization . . . 26

9 Limitations and Future Work **27**
9.1 UART Communication Bottleneck . . . 27
9.2 FPGA Resource Limitations . . . 28
9.3 Memory System Limitations . . . 28
9.4 Architectural Limitations . . . 28
9.5 Future Work . . . 28

10 Conclusion **29**

1 Introduction

Parallel processors are used when a program can split its work into many small independent tasks. A common example is a GPU, where many simple processing elements execute similar operations on different data. This project follows the same general idea and implements a small GPGPU-style accelerator on FPGA.

The goal of the project was to design a custom parallel architecture that can run simple RISC-V programs on multiple cores at the same time. The final design uses 32 processing elements. All of them execute the same instruction stream in lockstep, but each processing element has its own register file and works on its own data. This makes the design simpler than a full GPU, while still allowing parallel execution.

The accelerator was implemented and tested on a Digilent ZedBoard, which is based on the Xilinx Zynq-7000 SoC. The programmable logic side of the Zynq device was used for the custom GPGPU hardware, while the ARM Cortex-A9 processing system was used as the host that loads programs, starts execution, and reads back results. The FPGA design and bitstream generation were performed using Xilinx Vivado, while the host-side bare-metal integration was developed using Xilinx Vitis.

The system is not only a hardware design. It also includes the basic software flow needed to compile and run programs on the accelerator. C programs are compiled to RISC-V code, converted to instruction memory files, loaded to the FPGA, executed, and then the results are read back from data memory. This makes it possible to test real workloads instead of only small hand-written assembly programs.

The main parts of the project are:

- a 32-core lockstep GPGPU-style architecture,
- a small supported subset of the RISC-V instruction set,
- a five-stage pipeline with hazard and forwarding logic,
- a shared instruction fetch and decode path,
- a data memory crossbar for connecting 32 cores to dual-port memory,
- a host controller for loading programs and reading results,
- a software stack for compiling and running C kernels,
- demo applications such as 2D n-body simulation, 3D n-body simulation, and Mandelbrot rendering.

The project also includes a testing flow with simulation, a software reference model, program checking, and random testing. These tools were used to check that the hardware works correctly and that the programs follow the restrictions of the lockstep execution model.

Overall, this project demonstrates a complete small parallel computer system: from hardware design, to host integration, to software execution, to visual demo applications.

2 Project Goals and System Overview

The main goal of this project was to build a simple but complete parallel accelerator. The design had to be small enough to fit on the FPGA, but also general enough to run more than one example program. For this reason, the system was based on a small RISC-V subset and a simple host-controlled execution flow.

2.1 Design Goals

The design was based on the following goals:

- **Run many cores in parallel.** The accelerator should use multiple processing elements so that one program can work on many data items at the same time.
- **Use a simple RISC-V instruction subset.** The cores should execute normal integer RISC-V instructions such as arithmetic operations, branches, jumps, loads, and stores.
- **Share the instruction stream.** All processing elements should execute the same instruction at the same time. This reduces hardware complexity and cost because instruction fetch and decode do not need to be copied 32 times.

- **Keep separate per-core state.** Even though the instruction stream is shared, each processing element should have its own stack, register file, and execution path.
- **Support memory access from all cores.** All cores need to read and write data memory. Since the data memory has only two ports, the design needs a crossbar and stall logic to safely handle memory contention.
- **Allow a host CPU to control execution.** The host should be able to load instructions, write input data, start the accelerator, and read output data after the kernel finishes.
- **Provide a usable software flow.** Programs should be written in C when possible, compiled to RISC-V, loaded to the FPGA, and compared with native software reference outputs.
- **Run real demo workloads.** The system should not only run small tests. It should also run applications that produce visible results, such as n-body simulation and Mandelbrot rendering.

2.2 System Overview

The final system is made of three main layers:

1. the hardware accelerator,
2. the host interface,
3. the software stack.

The hardware accelerator contains the 32 processing elements, the instruction memory, the data memory, the shared pipeline frontend, and the memory crossbar. The processing elements execute the same instruction at the same time, but they operate on different data. Each core can identify itself using a thread ID, which is stored in register `x31`.

The host interface is responsible for controlling the accelerator. Before execution, the host loads the program instructions into instruction memory and writes the input arguments into data memory. Then it starts the accelerator. When the kernel finishes, the accelerator enters a dumping state, where the host can read the output data from memory.

The software stack connects normal C programs with the FPGA hardware. Each program can have a RISC-V kernel version and a native x86 reference version. The RISC-V version is compiled using a cross-compiler and converted into a memory file that can be loaded into the FPGA. The native version is used to check the expected behavior and to help debug the program.

The basic execution flow is:

1. write a program kernel in C
2. compile it to RISC-V
3. generate the instruction memory file
4. load the instructions to the FPGA
5. write the kernel arguments to data memory
6. start the accelerator
7. wait for the kernel to finish
8. read the output data from data memory
9. process or visualize the result on the host

This design keeps the hardware simple. The shared instruction path avoids duplicating the same fetch and decode logic for every core. At the same time, the replicated execution units allow the system to process 32 data elements in parallel.

The main limitation of this approach is that all cores must follow the same control flow. If different cores need to take different branches, the hardware cannot execute both paths independently. For this reason, the software and testing tools must make sure that programs are safe for lockstep execution.

This design is therefore a simplified GPU-like system. It does not try to match the full complexity of a modern GPU. Instead, it focuses on the main ideas: parallel execution, shared instruction control, per-core data processing, memory arbitration, and host controlled kernel execution.

2.3 Setup & tools

The accelerator was implemented and tested on a Digilent ZedBoard, which is based on the Xilinx Zynq-7000 SoC. The programmable logic side of the Zynq device was used for the custom GPGPU hardware, while the ARM Cortex-A9 processing system was used as the host that loads programs, starts execution, and reads back results. The FPGA design, testing, and bitstream generation were performed on x86 systems using Xilinx Vivado, while the host-side bare-metal integration was developed using Xilinx Vitis.

3 Hardware Architecture

The hardware of the Simple GPGPU is implemented entirely in Verilog. It functions as a basic System on Chip where a host controller directs a 32-core Streaming Multiprocessor (SM).

3.1 Top Level System and Host Integration

The top level `GPGPU` module handles the interaction between the host, the processing elements, and the dual port memory subsystems. Execution is handled by a finite state machine within the `HostController`, transitioning strictly between `CORE_LOADING`, `CORE_RUNNING`, and `CORE_DUMPING` states.

3.2 The Streaming Multiprocessor (Shared frontend)

To fit 32 processing elements onto an FPGA, fully duplicating a standard CPU pipeline is expensive. Instead, the `StreamingMultiprocessor` module employs a shared frontend, centralizing the Instruction Fetch (IF) and Instruction Decode (ID) stages.

A single global Program Counter, fetches one 32 bit instruction per clock cycle from the IMEM. This instruction is decoded once by a shared `Decoder` module, which extracts the fields including the opcode (`i_instr[6:0]`), register addresses (`rs1`, `rs2`, `rd`), and standard RISC-V immediates (`imm_31_12`, `imm_31_25`).

This design inherently enforces strict SIMD (Single Instruction, Multiple Data) lockstep execution. Because there is only one Program Counter, hardware level thread divergence is impossible. In the Verilog implementation, Core 0 acts as the master control thread. Its branch evaluation signals (`core_ex_branch_taken[0]` and `core_ex_beq_target_idx[0]`) are routed backward to the centralized fetch unit to drive the global PC and, if taken, assert a `flush` signal to clear the `ifid_program_counter` pipeline register.

3.3 The Streaming Processor

While the frontend is shared, the backend execution is fully parallelized. The SM instantiates an array of 32 independent `StreamingProcessor` (SP) cores. Each SP encapsulates the Execute (EX), Memory (MEM), and Write Back (WB) stages.

Crucially, each SP maintains its own private `Regfile`. By utilizing the core's unique Thread ID, which is accessible via software through register `x31`, the identically broadcasted instructions are applied to unique data offsets in memory.

3.3.1 ALU and Multiplier Pipeline

The Execute stage features a custom `ALU` supporting a fundamental subset of RISC-V integer operations, including logical shifts (`ALU_SLL`, `ALU_SRA`) and comparisons (`ALU_SLT`).

Because high width multiplication on an FPGA can drastically reduce the maximum clock frequency if forced into a single combinational cycle, the architecture employs a dedicated multi cycle DSP pipeline. When an R Type multiply instruction is decoded, it bypasses standard ALU execution and enters a 3 stage shift register (`mul_stage1`, `mul_stage2`, `mul_stage3`), for now the EX stage blocks all instructions that aren't multiplications from entering when a multiplication is inside the pipeline. The codebase explicitly uses the `(* use_dsp = "yes" *)` synthesis attribute on these registers to force the Vivado toolchain to map the multiplication path into the FPGA's dedicated hardware DSP48 slices.

3.3.2 Data Hazards and Forwarding

Despite the lockstep execution, individual cores can encounter Read After Write data dependencies within their local registers. Each SP implements a `ForwardingUnit`.

The `ForwardingUnit` actively monitors the `exmem_rd` and `memwb_rd` destination registers. If a dependency is detected, it generates selector signals to multiplex data directly into the ALU inputs (`ex_actual_alu_in_a`), bypassing the register file via forwarding paths.

If a load use hazard occurs, or if an instruction depends on the 3 cycle multiplier pipeline before `mul3_valid` is asserted, the `HazardUnit` safely stalls the pipeline and injects NOPs into the control path. Additionally, the Write Back stage includes a `safe_wb_rdata` latch, if the global pipeline stalls during a memory load, this latch captures the BRAM output immediately so that subsequent delayed memory requests do not overwrite the shared read bus.

3.4 Memory Arbitration: The Nx2 Crossbar

A major architectural challenge in SIMD design is memory bandwidth. The 32 SPs require independent access to the DMEM, but the underlying Block RAM only possesses two read/write ports. To prevent fatal memory collisions, the architecture routes all memory requests through a `MemoryCrossbarNx2` module.

The crossbar operates as a dynamic arbiter using a double scan priority encoder.

1. **Winner A Selection:** The arbiter utilizes a `for` loop to scan the 32 bit request vector (`i_req`) from the Most Significant Bit (`N-1`) down to the Least Significant Bit (`0`) to find the first asserting core, generating `winner_a_idx`.
2. **Winner B Selection:** A bitwise mask is dynamically applied (`req_masked = i_req & ~({{N-1{1'b0}}, 1'b1} << winner_a_idx)`) to hide Winner A. The exact same scanning process is then repeated on the masked request bus to assign `winner_b_idx`.

The crossbar also features aggressive write collision detection. If both winners attempt to write to the exact same memory address in the same clock cycle (`mux_addr_a == mux_addr_b`), the hardware flags a `collision_w` and explicitly disables Port B's write enable (`o_wen_b = mux_wen_b && !collision_w`) to prevent memory corruption.

3.4.1 Global Stall Logic and Request Accumulation

When more than two cores request memory access simultaneously, the hardware must stall the pipeline to serve them sequentially. The `StreamingMultiprocessor` generates a `global_stall` signal if any active core's memory request is denied by the crossbar (e.g., `|(active_mem_ren & ~sp_mem_grant)`).

During a global stall, the pipeline registers freeze, but the crossbar iteratively accumulates the grant signals in a dedicated `mem_satisfied` register (`mem_satisfied <= mem_satisfied | sp_mem_grant`). Active requests are continually masked against this register (`active_mem_ren = sp_mem_ren & ~mem_satisfied`) until every requesting core has been served, at which point the stall drops.

Because synchronous BRAM reads take one clock cycle to return data, the crossbar implements a 1 cycle pipeline delay for read routing. It latches the winning core states into `port_a_core_r` and `port_b_core_r`. On the subsequent clock cycle, these latched indices act as multiplexer selectors to steer the returning 32 bit data payloads exclusively into the `o_rdata` bus segments of the cores that originally requested them.

3.4.2 Atomic Memory Operations (Banked Design)

Although developed for a multi-banked memory crossbar that was excluded from the final bitstream due to time constraints, the architecture included hardware support for Atomic Memory Operations (AMO) to enable

parallel synchronization. When a core asserts an AMO request via the `i_amo` signal, the crossbar executes a two cycle Read Modify Write sequence directly within the arbitration logic, bypassing the core's standard execution pipeline.

During the first cycle, the arbiter grants the requesting core access to the memory bank, initiates a read request, and locks the targeted bank by asserting an internal active signal (`amo_active`). It also latches the target address and the addend value provided by the core into dedicated registers (`amo_addr` and `amo_addend`).

In the second cycle, the crossbar temporarily hijacks the locked memory bank. It blocks any new requests from interacting with that specific bank and performs the arithmetic addition locally, combining the returned read data with the latched addend. The crossbar then automatically asserts the write enable signal to store the newly computed value back into memory. This ensures that the entire fetch, compute, and store sequence happens atomically without risking data corruption from other cores.

4 SoC and Host Integration

At this stage of the project, the GPGPU RTL core had already been developed and validated at the RTL simulation level. However, in order to use the design on the FPGA, the core had to be integrated into a complete System-on-Chip. The goal of this part of the project was to bridge the gap between a working RTL core and a deployable FPGA system that could be controlled by software, synthesized, implemented, and tested end-to-end.

The integration work focused on wrapping the hardware accelerator with the necessary control infrastructure, exposing a communication interface to the host side, and ensuring that the final design could be validated not only in simulation but also after FPGA implementation.

4.1 Integration Goals

The main goal was to transform the isolated RTL design into a complete FPGA system. This required more than simply instantiating the compute core in a top-level module. The final system needed to support program loading, data memory initialization, kernel execution, status reporting, and result dumping.

- Package the working RTL GPGPU core into a complete synthesizable SoC.
- Provide a host-control interface for communicating with the core.
- Allow the host to initialize instruction memory and data memory.
- Provide a mechanism for starting execution and detecting when execution has completed.
- Allow the host to read back data memory contents after execution.
- Preserve a common top-level interface that could be used both in simulation and after FPGA implementation.

This integration step was necessary because RTL-level correctness alone does not guarantee that the complete system can be controlled, synthesized, implemented, or tested on the physical FPGA platform.

4.2 System-on-Chip Design Overview

The final SoC design consists of the GPGPU core, local instruction and data memories, a host controller, and a command processor. The purpose of this structure is to expose the core through a simple command-based interface rather than requiring the external host to directly manipulate internal RTL signals.

At a high level, the system is organized as follows:

- The **GPGPU core** performs the actual parallel computation.
- The **instruction memory** stores the program executed by the accelerator.
- The **data memory** stores input data, kernel arguments, intermediate values, and output data.
- The **host controller** manages the high-level state of the accelerator system.
- The **command processor** decodes host commands and translates them into memory accesses or execution-control operations.

This organization allows the accelerator to be controlled through a small set of external commands, such as loading memory, starting execution, waiting for completion, and dumping memory contents.

4.3 Host Controller

The `HostController` module is responsible for controlling the high-level execution state of the GPGPU system. It acts as the top-level control finite state machine around the accelerator and determines whether the system is currently accepting host writes, executing the kernel, or allowing the host to read back results.

The controller uses three main core states:

- `CORE_LOADING`: the host is allowed to initialize instruction memory and data memory.
- `CORE_RUNNING`: the GPGPU core is executing the loaded program.
- `CORE_DUMPING`: execution has completed and the host is allowed to read back memory contents.

The state transitions are driven by two handshake signals generated by the host command processor and by the completion signal from the GPGPU core. After reset, the system enters the loading state. While in this state, the host can write the instruction and data memories. When the host finishes loading the program and input data, it issues a write-completion command. This causes the controller to transition from `CORE_LOADING` to `CORE_RUNNING`.

During `CORE_RUNNING`, the accelerator owns the memory interfaces and executes the kernel. The host controller remains in this state until the core asserts its completion signal. Once execution completes, the controller moves to `CORE_DUMPING`, where the host can read back data memory, instruction memory, or debug-visible values. Finally, after the host finishes dumping the required output data, it issues a read-completion command, which returns the system to `CORE_LOADING` for the next test or program run. The resulting state flow is:

State	Trigger	Next state
`LOADING`	Run command received	`RUNNING`
`RUNNING`	Kernel finishes	`DUMPING`
`DUMPING`	Done command received	`LOADING`

Figure 1: Host controller FSM

This controller therefore separates the system into clear phases: memory initialization, kernel execution, and result readback. This is important because the memories are shared between the host interface and the GPGPU core. In the loading and dumping phases, the host can access the memories. In the running phase, the accelerator takes ownership of the memory ports and performs the computation.

In the top-level `GPGPU` module, the current core state is also used to generate external status signals such as `o_loading`, `o_running`, and `o_dumping`. These signals are useful for debugging and for observing the current phase of execution from outside the hardware system.

4.4 Host Command Processor

The `HostCommandProcessor` module implements the command-level interface used by the host to interact with the GPGPU system. While the host controller manages the high-level execution phase, the command processor handles individual host commands and translates them into memory accesses, readback operations, or state-transition handshakes.

The command processor receives a command code, a command-valid signal, an address, and optional write data from the host side. Internally, it latches the command, address, and write data when a valid command is accepted. It then performs the requested operation and asserts a done signal back to the host when the command has completed.

The supported commands are:

Command	Name	Purpose
0	CMD_IMEM_WRITE	Write a word to instruction memory
1	CMD_DMEM_WRITE	Write a word to data memory
2	CMD_WRITE_DONE	Signal that program/data loading is complete
3	CMD_DMEM_READ	Read a word from data memory
4	CMD_IMEM_READ	Read a word from instruction memory
5	CMD_REG_READ	Read a debug/register value
6	CMD_READ_DONE	Signal that memory dumping is complete

The command processor itself is implemented as a small finite state machine with the following states:

- S_IDLE: wait for a valid command from the host.
- S_ACCEPT: accept and decode the command.
- S_WRITE: perform a write-type operation or emit a phase-completion handshake.
- S_READ: capture read data from the selected source.
- S_DONE: report command completion to the host.

The command flow is:

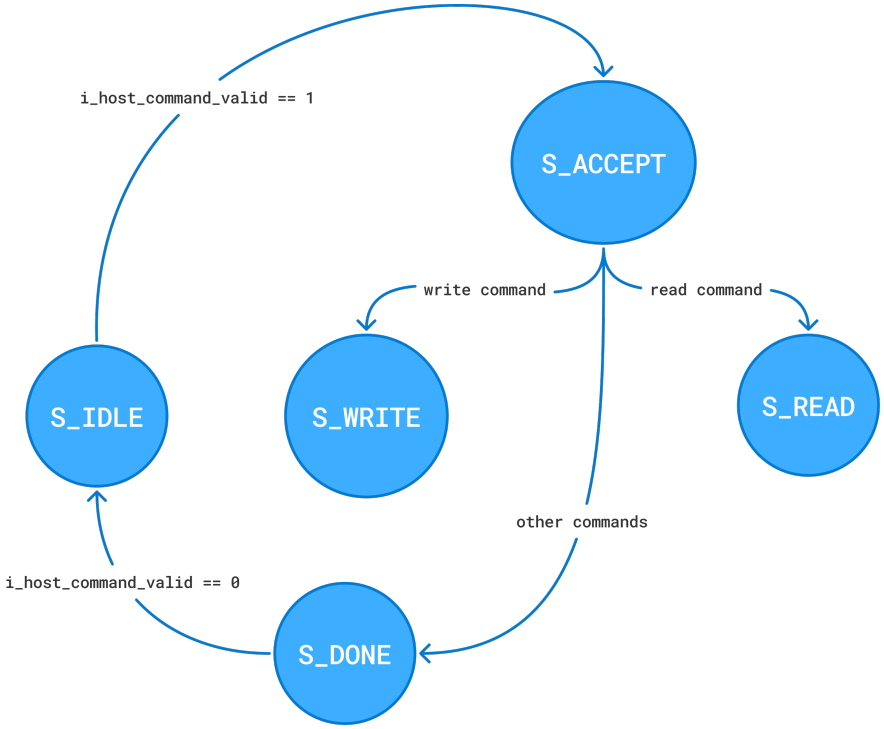


Figure 2: Host command processor FSM

Write commands are only allowed during the correct system phase. For example, instruction-memory and data-memory writes are only enabled when the core is in CORE_LOADING. This prevents the host from overwriting memories while the accelerator is running. Similarly, the CMD_WRITE_DONE command only produces the o_host_done_writing handshake while the system is in the loading state.

Read completion is also phase-gated. The CMD_READ_DONE command only produces the o_host_done_dumping handshake while the system is in CORE_DUMPING. This handshake informs the host controller that the host has finished reading the output data and that the system can return to the loading phase. For read commands, the processor captures the selected read-data signal into o_host_rdata. Depending on the command, this value can come from instruction memory, data memory, or the register/debug read path. Once the operation is complete, the processor asserts o_host_done. The processor remains in the done state until the host deasserts

`i_host_command_valid`, which creates a simple request/acknowledge protocol between the host and the hardware.

This design gives the host a compact command interface while keeping the internal accelerator protocol controlled and phase-safe. The host does not directly control the GPGPU core or memory write-enable signals. Instead, it issues commands, and the command processor generates the appropriate internal control signals only when the current core state allows the operation.

4.5 Synthesis and Implementation

After the SoC wrapper, host controller, and command processor were added, the design could be passed through the FPGA synthesis and implementation flow. This step verified that the complete integrated design was not only valid in RTL simulation but also suitable for deployment on the target FPGA platform.

The synthesis and implementation process required the complete system to be structurally valid, with a clear top-level module and well-defined interfaces. This was important because the final tests were designed to target the same top-level structure used for implementation.

For the implementation the following options were used on Vivado 2025.2:

- **Strategy:** `Congestion_SpreadLogic_high`
- **Opt Design:** `ExploreWithRemap`
- **Place Design:** `ExtraTimingOpt`
- **Post-Place Phys Opt Design:** `AggressiveExplore`
- **Route Design:** `NoTimingRelaxation`
- **Post-Place Phys Opt Design:** `AggressiveExplore`

With these settings we managed to fit 32 SPs into the FPGA's fabric and achieve a 38MHz clock.

4.6 Initial Integration Attempt Using UART in the PL

The first approach for board-level integration was to implement a UART communication module directly in the Programmable Logic. The idea was to allow an external test script to communicate directly with the hardware accelerator over UART.

In this approach, the UART module in the PL would receive commands from the host machine and forward them to the command processor. It would also transmit memory dumps and status information back to the host. This design would have avoided the need for software running on the Processing System.

The intended structure was:

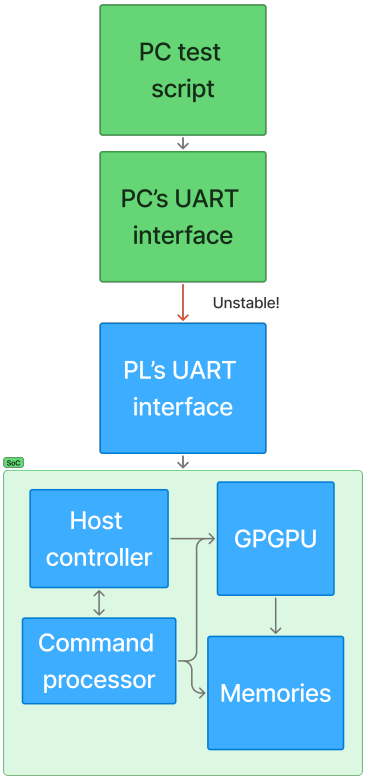


Figure 3: UART module in PL flow

However, this approach proved unstable during FPGA testing. Communication errors made it difficult to live debug using the UART interface. Because reliable communication was necessary for final end-to-end testing, this approach was abandoned in favor of using the Zynq Processing System as the communication endpoint.

4.7 Integration Using the Zynq Processing System

The final integration approach used the Zynq-7000 Processing System as the communication bridge between the external host and the accelerator implemented in the Programmable Logic. Instead of implementing the full UART interface in the PL, UART communication was handled by software running on the ARM processor in the PS. The PS then communicated with the PL using memory-mapped interfaces.

This produced a cleaner separation of responsibilities:

- The **PS** handles UART communication and host-side protocol parsing.
- The **PL** contains the core, memories, host controller, and command processor.
- The **AXI GPIO interface** provides a simple communication path between PS and PL.

The revised architecture was:

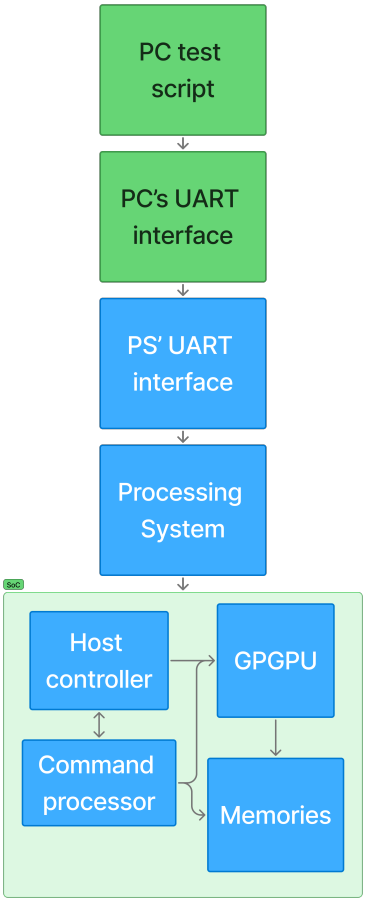


Figure 4: Revised architecture using PS for UART

This approach made the system easier to debug because UART communication was handled in software, while the PL remained focused on accelerator execution and memory control.

4.8 PS-to-PL Communication Through AXI GPIO

To connect the Processing System to the Programmable Logic, AXI GPIO blocks were used. AXI GPIO exposes simple memory-mapped registers to software running on the PS. The PS can write command and data values to these registers and read status or data values back from the PL.

This interface was sufficient for the project because the main objective was control and validation rather than high-throughput data transfer. A more complex interface such as AXI DMA could provide higher bandwidth, but AXI GPIO provided a simpler and more reliable integration path for bring-up and testing.

The PS-to-PL communication path is:

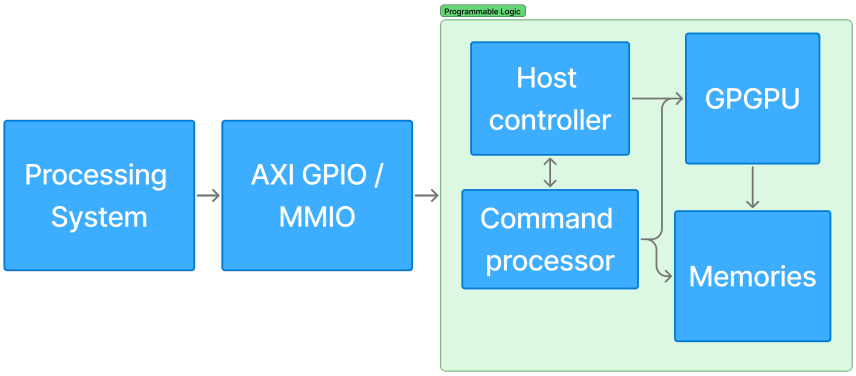
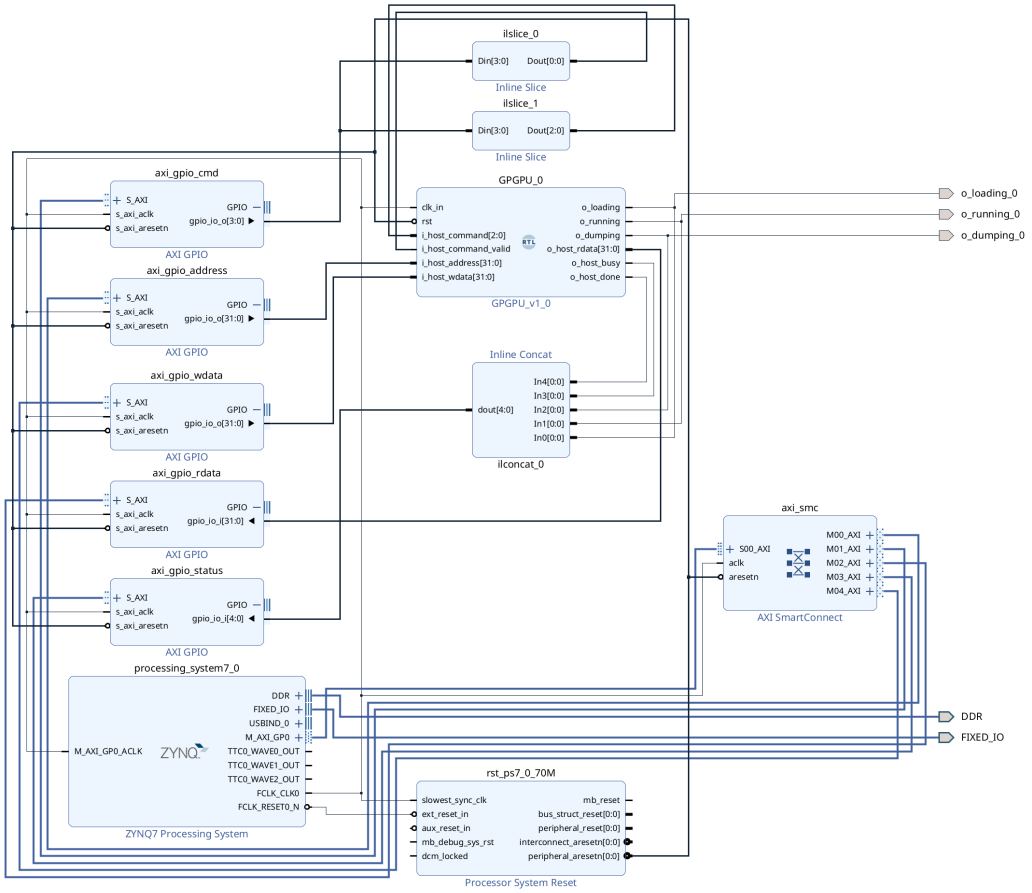


Figure 5: Flow of PS to PL communication

The AXI GPIO interface was used to exchange:

- Command words from the PS to the PL.
- Data words for memory initialization.
- Status signals from the PL to the PS.
- Memory dump data from the PL back to the PS.

This allowed the PS software to control the accelerator using simple memory-mapped reads and writes. This is the final block diagram of the design:



4.9 PS Host Software

A lightweight host program was written to run on the ARM processor in the Zynq Processing System. This program receives commands from the external test machine over UART and translates them into AXI GPIO transactions to control the PL.

The host software is responsible for:

- Receiving commands over UART.
- Parsing the requested operation.
- Writing command and data values to the AXI GPIO registers.
- Polling the PL status signals.
- Reading memory dump data from the PL.
- Returning results to the external host over UART.

This software acts as a bridge between the external testing environment and the implemented FPGA accelerator. For example, a memory load command received over UART is translated into a sequence of AXI GPIO writes, while a memory dump command is translated into AXI GPIO reads followed by UART transmission back to the test script. This structure made it possible to control the implemented FPGA design from an automated host-side testing script. The final version can be found under `host/baremetal/`.

5 Software Stack and Runtime

The hardware accelerator needs a software flow in order to run real programs. The goal of the software stack is to take a C program, compile it to RISC-V instructions, load it to the FPGA, run it, and read the results back from data memory.

The software stack is split into four main parts:

- the program build flow,
- the linker script and memory layout,
- the small GPGPU runtime,
- the host-side FPGA runner and program adapters.

This section describes how these parts work together.

5.1 Program Build Flow

Programs are stored under the `programs/` directory. Each program has its own folder and normally contains a C source file with the same name as the program. For example:

- `programs/nbody/nbody.c`
- `programs/nbody-3d/nbody-3d.c`
- `programs/differences/differences.c`

The build system is controlled by `programs/Makefile`. It builds each program in two different ways:

1. as a RISC-V bare-metal program for the FPGA,
2. as a native x86 program for the host machine.

The RISC-V version is the one that runs on the accelerator. The x86 version is used as a reference, so that the same program can also be tested on the host machine.

The RISC-V compilation flow is:

1. compile the C file to a RISC-V ELF file,
2. disassemble the ELF file with `objdump`,
3. create a clean assembly listing,
4. extract the raw instruction words,
5. write them to an instruction memory file.

The main generated files are:

- `<program>.elf`: the compiled RISC-V executable,
- `<program>_dump_real.asm`: the full disassembly,
- `<program>_program.asm`: the clean assembly listing,
- `<program>_instructions.mem`: the raw instruction memory file,
- `<program>_x86`: the native x86 executable.

For example, building the `nbody` program produces files such as:

```
programs/nbody/nbody.elf
programs/nbody/nbody_dump_real.asm
programs/nbody/nbody_program.asm
programs/nbody/nbody_instructions.mem
programs/nbody/nbody_x86
```

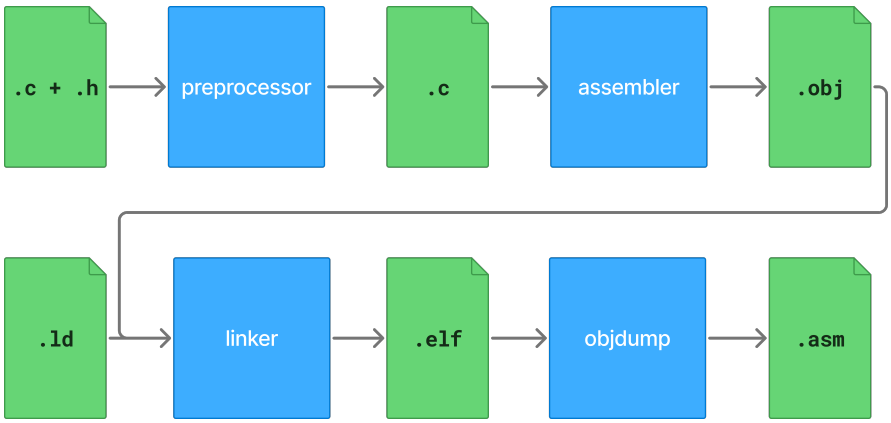


Figure 6: Diagram of the program build flow.

The instruction memory file is the file that is loaded into the FPGA instruction memory before execution. The RISC-V compiler is configured for a small 32-bit integer target. The build uses options such as:

```
-march=rv32im
-mabi=ilp32
-ffreestanding
-fno-builtin
-nostdlib
-nostartfiles
```

These options are important because the accelerator does not run a normal operating system and does not support the full standard C runtime. The program must be a small bare-metal kernel.

5.2 Program Structure

Most programs contain both the RISC-V kernel code and the native x86 reference code in the same C file. The two versions are separated using the `__riscv` preprocessor symbol.

A simplified structure is:

```
#ifdef __riscv

void kernel_main(void)
{
    unsigned int tid = gpgpu_thread_id();

    // RISC-V kernel code that runs on the accelerator
}

GPGPU_START(kernel_main)

#else

int main(void)
{
    // Native x86 reference code
    return 0;
}

#endif
```

This structure keeps the FPGA version and the reference version close together. It also makes debugging easier, because both versions can use the same constants and helper functions.

The RISC-V version is written as a kernel. Each processing element runs the same kernel code, but receives a different thread ID. The native x86 version usually simulates the same behavior with a loop over all 32 bodies or lanes.

5.3 Memory Layout

The accelerator uses separate instruction memory and data memory. The software layout is defined by the linker script in:

`programs/gpgpu.ld`

The linker script places code and data at addresses that match the hardware memory system.

The main memory regions are:

- instruction memory for program code,
- data memory for kernel arguments,
- data memory for program state and output,
- a stack area for the processing elements.

A simplified view of the data memory layout is:

```
0x00000040  GPGPU_ARGS
0x00001000  program data region
0x00001800  stack bottom
0x00002000  stack top
```

The kernel arguments are stored at `GPGPU_ARGS`. This is a small fixed area in data memory that the host writes before starting a kernel. The program then reads these values to know where its input and output data are located.

The program data region is used by the workloads to store their arrays. For example, the n-body program stores position and velocity arrays in this region.

The stack region is used by the runtime. Each processing element receives its own small stack area, so that normal C code can use temporary values and compiler-generated stack stores.

5.4 GPGPU Runtime

The file `programs/gpgpu_runtime.h` provides the small runtime used by the RISC-V kernels. It does not provide a full standard library. Instead, it only provides the basic functions and macros needed to start a kernel on the accelerator.

The most important function is `gpgpu_thread_id()`:

```
static inline unsigned int gpgpu_thread_id(void)
{
    unsigned int tid;
    __asm__ volatile("mv␣%0,␣x31" : "=r"(tid));
    return tid;
}
```

Each processing element stores its lane ID in register `x31`. The kernel uses this value to decide which data element it should process.

For example, in a simple parallel program, each lane can use its thread ID as an array index:

```
unsigned int tid = gpgpu_thread_id();
output[tid] = input[tid] + 1;
```

The runtime also defines the `GPGPU_START` macro. This macro creates the real entry point of the RISC-V program. It sets up the stack for the current processing element and then calls the kernel function.

The runtime uses the thread ID to calculate a private stack for each lane. Each lane receives a stack slice of 64 bytes. This allows the compiler to use stack memory without all cores writing to the same stack addresses.

The main idea is:

1. read the thread ID from `x31`,
2. multiply it by the stack stride,
3. subtract this offset from the top of the stack region,
4. call the kernel function.

This setup is needed because the hardware runs 32 copies of the same code at the same time. Without separate stack areas, all cores would use the same stack addresses and corrupt each other's data.

5.5 Kernel Arguments

The host passes arguments to a kernel through the `GPGPU_ARGS` area in data memory. In the runtime, this area is exposed as:

```
#define GPGPU_ARGS ((volatile int *)__gpu_args_base)
```

A kernel can read its arguments like a normal integer array.

For example, the n-body kernel uses:

```
volatile int *base = (volatile int *)(uintptr_t)GPGPU_ARGS[0];
int steps = GPGPU_ARGS[1];
int reset = GPGPU_ARGS[2];
```

In this example:

- `GPGPU_ARGS[0]` is the base address of the n-body state arrays,

- `GPGPU_ARGS[1]` is the number of simulation steps,
- `GPGPU_ARGS[2]` tells the kernel if it should reset the state.

This simple argument passing method is enough for the current workloads. It also keeps the hardware interface simple, because the host only needs to write a few words to data memory before starting the kernel.

5.6 Generic FPGA Runner

The file `programs/fpga_run.py` contains the generic host-side execution loop. This script does not know the details of each workload. Instead, it only knows how to communicate with the FPGA and run a kernel.

The generic execution flow is:

1. load the instruction memory file,
2. optionally initialize data memory,
3. ask the program adapter for the next kernel arguments,
4. write the kernel arguments to data memory,
5. start the accelerator,
6. wait until the kernel finishes,
7. dump the requested output window from data memory,
8. pass the output words back to the program adapter,
9. repeat for the requested number of kernel calls,
10. run final processing or visualization.

This design separates common FPGA communication from program-specific logic. The runner handles the common tasks, while each program adapter explains what arguments and outputs its own kernel uses.

5.7 Program Adapters

Each complex program can define a program adapter in:

`programs/<program>/fpga.py`

The adapter defines the host-side ABI of the program. It tells the generic runner:

- what arguments to write before each kernel call,
- which data memory window to read after the kernel finishes,
- how to convert the raw output words into useful results,
- whether any initial data memory setup is needed,
- whether a visualization step should run at the end.

The important adapter methods are:

- `kernel_arguments()`,
- `output_window()`,
- `process_output()`,
- `initial_dmem()`,
- `finalize()`.

For example, the n-body adapter passes the base pointer, the number of steps, and the reset flag to the kernel. After the kernel finishes, it reads the position arrays from data memory and writes them to a CSV file.

This adapter-based design makes it easier to add new workloads. A new program does not need to rewrite the UART or FPGA control logic. It only needs to define how its own arguments and outputs are arranged in memory.

5.8 UART Host Interface

The Python runner communicates with the FPGA through the UART monitor in:

`host/baremetal/gpgpu_uart.py`

This file provides a higher-level Python interface for the commands exposed by the bare-metal host monitor. It supports the following operations:

- loading instruction memory,
- loading data memory,
- starting execution,
- waiting for the dumping state,
- reading data memory,
- returning the accelerator to the loading state.

The two most important execution commands are:

- `run()`, which starts the accelerator,
- `done()`, which tells the host controller that output reading is complete.

The host and accelerator follow a simple state sequence:

```
LOADING -> RUNNING -> DUMPING -> LOADING
```

During `LOADING`, the host writes instructions and input data. During `RUNNING`, the accelerator executes the kernel. During `DUMPING`, the host reads the output data.

5.9 Example: N-Body Software Flow

The n-body workload shows how all parts of the software stack work together.

First, the C code is compiled into a RISC-V instruction memory file. Then the generic FPGA runner loads this file into instruction memory. Before each kernel call, the n-body adapter writes the kernel arguments:

- the base address of the n-body state arrays,
- the number of simulation steps,
- the reset flag,
- one reserved word.

Inside the kernel, each processing element uses its thread ID to update one body. The state arrays are stored in data memory:

- `pos_x[32]`,
- `pos_y[32]`,
- `vel_x[32]`,
- `vel_y[32]`.

After the kernel finishes, the adapter reads the position arrays from data memory and appends the results to a CSV file. The output can then be visualized on the host.

This example shows the full path from C code to FPGA execution and then back to host-side output processing.

5.10 Summary

The software stack makes the accelerator usable as a small programmable system. The hardware alone can execute instructions, but the software flow provides the tools needed to build programs, load them, pass arguments, run kernels, and read results.

The main advantage of this design is that the common execution logic is shared. The same build system, runtime, UART interface, and FPGA runner can be used for different workloads. Only the program adapter and the kernel code need to change for each new application.

6 Applications and Demo Workloads

To verify the functionality of the hardware and demonstrate its capabilities, we developed practical compute workloads written in C. These programs were compiled into RISC-V machine code using our custom software stack and loaded onto the FPGA. We specifically chose two distinct workloads to stress different aspects of the architecture: an N-Body simulation to stress shared memory access, and a Mandelbrot set renderer to stress the compute and multiplier pipelines.

6.1 N-Body Gravitational Simulation

The N-body simulation calculates the gravitational forces and resulting trajectories of interacting particles. This is a classic $O(N^2)$ problem, making it an ideal candidate for parallel acceleration.

In our implementation, the workload is mapped directly to the hardware limitations: the simulation consists of exactly 32 bodies. Each of the 32 Streaming Processors (SPs) is statically assigned to one body based on its Thread ID (`tid`).

During a simulation step, each core reads the X and Y coordinates of all 32 bodies from the shared Data Memory. It calculates the relative distances (`dx = xj - xi`) and accumulates the net force acting on its assigned body. Because the Simple GPGPU does not possess a Hardware Divider or a Floating Point Unit (FPU), the software relies on bitwise right-shifts (`ax >> 2`) to scale the acceleration and compute the new velocity vectors.

This workload heavily tests the memory subsystem. Because all 32 cores must loop through and read the positions of every other body simultaneously, it aggressively exercises the `MemoryCrossbarNx2` and the global stall logic to safely arbitrate the massive burst of read requests.

7 Verification Methodology

Verifying a 32 core SIMD processor presents unique challenges, primarily because of the shared front end and the custom memory crossbar. A simple Verilog testbench is not enough to guarantee that the pipeline handles all hazard conditions and memory collisions correctly. To address this, we built a custom automated verification ecosystem heavily relying on Python and Bash scripts to validate both the software constraints and the hardware execution.

7.1 Automated Test Suite Ecosystem

The entire verification flow is orchestrated by a master script, `run_tests.sh`. Because we do not use a standard assembler for our low level hardware tests, we wrote a custom one. This script parses our hand written RISC-V assembly files, resolves labels, and translates the instructions into machine code, generating the `.mem` initialization files required by the Block RAM.

To verify that the hardware executes the instructions correctly, we rely on a "Golden Model" approach. We developed `expected_generator.py`, a Python based software simulator that mimics our architecture's expected behavior. For every test program, the Golden Model executes the instructions and predicts the exact cycle by cycle register states and final memory contents. We then run the physical Verilog simulation using our testbenches (such as `tb_GPGPU.v`) and compare the final memory dumps and register write backs against the Golden Model's predictions. If there is a perfect match, the test passes.

7.2 Program Safety and Constrained Random Verification

Because the hardware executes in strict lockstep, multi core programs must follow specific safety rules to prevent undefined hardware behavior. We implemented Constrained Random Verification (CRV) through a script called `program_checker.py` to validate the assembly code before it ever reaches the hardware.

The checker enforces three main safety constraints:

1. **Infinite Loop Prevention:** The checker simulates the program and enforces a strict maximum execution limit of 10,000 cycles. If a program exceeds this limit or attempts to execute an invalid backwards jump, it is flagged as a stalled loop and fails validation.
2. **Divergence Checking:** The checker simulates branch decisions across all 32 active cores. If it detects a state where one core attempts to take a branch while another core falls through, the program immediately fails validation.
3. **Memory Corruption Prevention:** The checker concurrently scans the computed memory write addresses of all 32 cores during store operations. If it detects that two or more cores are attempting to write to the exact same memory location in the same cycle, it flags a collision error to prevent data corruption. As we mentioned above this is also checked and handled in hardware, but the checker is used as a means to tell the programmer that his program is most likely wrong.

7.3 Aggressive Hardware Fuzzing

To ensure the pipeline is robust against complex data dependencies and memory contention, we used Random Instruction Testing (RIT). We built a hardware fuzzer, `random_tester.py`, which automatically generates highly biased RISC-V assembly programs specifically designed to break the processor.

Instead of generating purely random instructions, the fuzzer intentionally targets the `ForwardingUnit` and `HazardUnit`. It applies a 50 percent bias to register selection, meaning there is a high probability that an instruction will use the destination register of the immediately preceding instruction as its source operand. This heavily stresses the pipeline's ability to forward data correctly across the EX/MEM and MEM/WB registers.

Furthermore, the fuzzer purposefully injects "evil sequences". These are deliberate edge case combinations, such as:

- **Load Use Dependencies:** A load instruction immediately followed by an arithmetic instruction requiring the loaded data, forcing the hardware to stall and use the `safe_wb_rdata` latch.
- **Multi Cycle DSP Traps:** R Type multiply instructions injected directly before dependent instructions to verify that the hazard unit correctly tracks the 3 cycle delay of the DSP pipeline (`mul3_valid`) without dropping data.
- **Store Load Bypasses:** Back to back memory writes and reads to the same addresses to aggressively stress the timing of the `MemoryCrossbarNx2` arbiter.

7.4 Top-Level End-to-End Test Extension

To verify the integrated SoC, the test suite was extended to instantiate the top-level module rather than only testing isolated internal modules. This allowed the tests to exercise the same command interface used by the host controller and command processor. The purpose of this extension was to make the tests more realistic. Instead of directly modifying internal memories or signals, the tests interact with the design through the top-level host interface. This makes the test flow closer to the way the final FPGA system is controlled. The top-level tests validate:

- Instruction memory loading.
- Data memory loading.
- Kernel argument setup.
- Execution start and completion.
- Data memory dumping.

- Comparison against expected output.

This helped verify not only the compute core, but also the memory interface, command processor, and host controller.

7.5 Implementation-Compatible Testing

A key goal of the verification extension was to keep the tests compatible with the implemented top-level design. By targeting the same top-level interface used for FPGA implementation, the tests reduce the gap between simulation and hardware execution.

This approach provides stronger confidence than module-level tests alone, because it verifies that the complete system can be controlled through its external interface. It also helps catch integration bugs that may not appear when testing modules in isolation.

The methodology can be summarized as:

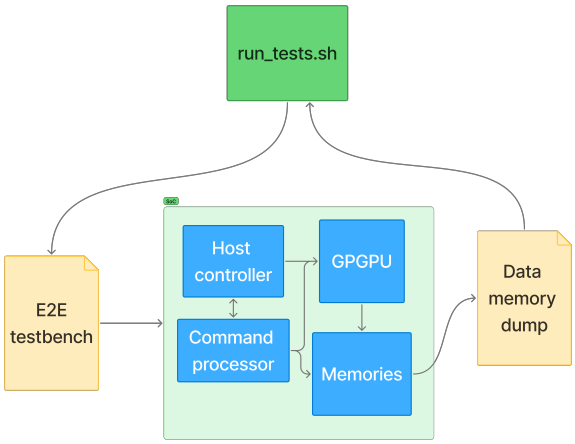


Figure 7: End-to-end testing flow

The same general command sequence is preserved across these levels, which makes the verification flow consistent from simulation to physical FPGA testing.

7.6 Final FPGA End-to-End Testing

The final validation step tested the complete implemented system on the FPGA board. In this flow, the external host communicates with the Zynq Processing System over UART. The PS host software then controls the PL through AXI GPIO.

The final end-to-end test flow is:

1. The host test script generates the input data and expected output.
2. The host sends commands to the PS over UART.
3. The PS software translates these commands into AXI GPIO transactions.
4. The PL command processor loads instruction memory and data memory.
5. The host triggers kernel execution.
6. The GPGPU core runs in the PL.
7. After completion, the host requests a data memory dump.
8. The PS reads the dump from the PL and sends it back over UART.
9. The host compares the returned memory contents against the expected generator.

This flow validates the complete path:

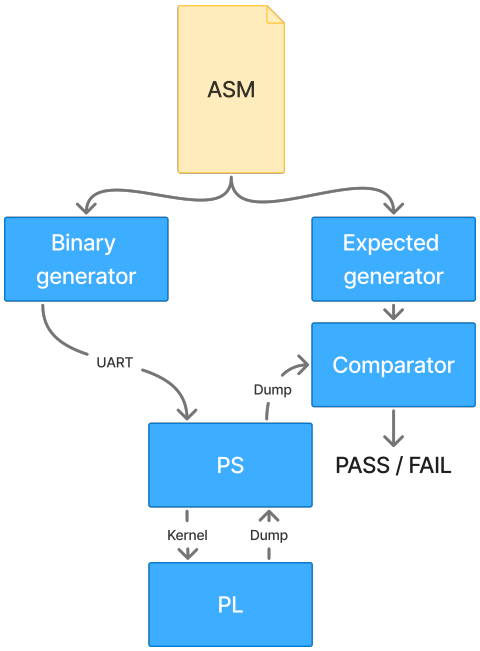


Figure 8: FPGA End-to-End test flow

This final test verifies more than just the computation result. It also validates the UART protocol, PS host software, AXI GPIO bridge, PL command processor, memory system, accelerator execution, and result readback path.

8 Results and Evaluation

The GPGPU system was evaluated in terms of functional correctness, performance, memory bandwidth, communication overhead, and FPGA resource utilization. The evaluation focuses mainly on the final 32-core SIMD design running on the Zynq-7000 FPGA fabric, with host communication handled by the ARM Cortex-A9 processing system.

8.1 Functional Validation

Before running the design on the FPGA, the hardware was validated through the custom Verilog test suite. The test infrastructure assembles RISC-V programs, runs them on the RTL model, and compares the resulting register and memory state against a Python golden model. This was especially important because the design includes forwarding, hazard detection, branching, jumps, multiplication, and shared memory access through a crossbar.

The software workloads were also structured so that the same C source file can be compiled in two modes:

- as a bare-metal RISC-V kernel for the GPGPU,
- as a native host executable for reference execution.

For example, the n-body program contains both the RISC-V kernel and the native reference implementation. On the GPGPU, each processing element obtains its logical thread ID from register `x31` and updates one body. On the host, the reference implementation iterates over the same 32 bodies sequentially. This allows the output of the FPGA execution to be compared against a known-good software implementation.

8.2 Peak Compute Throughput

The final hardware design contains 32 streaming processors operating in lockstep. Since all lanes share the same instruction stream, the ideal peak integer throughput can be approximated as:

$$\text{Performance}_{peak} = \#SPs \times IPC \times f \tag{1}$$

For the implemented design:

$$\text{Performance}_{peak} = 32 \times 1 \times 38.5\,\text{MHz} \approx 1.23\,\text{GINTOPS} \tag{2}$$

This number represents an ideal upper bound assuming that all 32 processing elements are active, each lane completes one useful integer operation per cycle, and the pipeline is not stalled by memory dependencies, hazards, or host communication.

In practice, the achieved performance depends strongly on the workload. Kernels with enough independent work per thread can benefit from the 32-way SIMD execution model, while memory-heavy kernels are limited by memory arbitration and by host-side transfers.

8.3 N-body Performance Comparison

The most important performance result comes from the n-body simulation. In this workload, 32 bodies are simulated, and each body is assigned to one processing element. Each lane computes the force contribution from all bodies and updates its own position and velocity.

The measured execution time for one n-body simulation step is shown in Table 1.

Table 1: N-body simulation performance comparison.

Metric	**GPGPU**	**ARM Cortex-A9**
Clock frequency	$\sim 38.5\,\text{MHz}$	$\sim 667\,\text{MHz}$
Bodies	32	32
Execution time	$\sim 129\,\mu s$	$\sim 308\,\mu s$
Speedup	$\sim 2.38\times$	$1\times$

Even though the ARM Cortex-A9 runs at a much higher clock frequency, the custom GPGPU is faster for this workload. The measured speedup is approximately:

$$\text{Speedup} = \frac{308\,\mu s}{129\,\mu s} \approx 2.38\times \tag{3}$$

This result demonstrates the main advantage of the architecture, the n-body kernel exposes enough data-level parallelism for the 32 processing elements to work simultaneously. The ARM Cortex-A9 executes the same body updates mostly sequentially, while the GPGPU evaluates the 32 body updates in parallel.

This result is especially notable because the GPGPU operates at only approximately $38.5\,\text{MHz}$, while the Cortex-A9 runs at approximately $667\,\text{MHz}$. Therefore, the speedup does not come from frequency, but from parallel execution.

8.4 Memory Bandwidth

The implemented design uses a shared data memory accessed through a crossbar. The final bitstream uses a dual-port BRAM-based data memory. Since each memory port is 32 bits wide, the theoretical internal data-memory throughput is:

$$2 \times 4\,\text{bytes} \times 38\,\text{MHz} \approx 307\,\text{MB/s} \tag{4}$$

Table 2 summarizes the internal data-memory configuration.

Table 2: Internal GPGPU data-memory bandwidth.

Metric	**Single shared data memory**
Latency	1 cycle
Data width	32 bits
Ports	2
Theoretical throughput	$\sim 307\,\text{MB/s}$

This bandwidth is the bandwidth inside the FPGA fabric between the processing elements and the BRAM data memory. It is not the same as the bandwidth between the host and the FPGA. The internal memory system is fast enough for small kernels, but contention still exists because 32 processing elements share only two data-memory ports. When many lanes issue loads or stores at the same time, the crossbar grants access to only two requests per cycle and stalls the other lanes until their memory operations are served.

A multi-banked memory system was also explored during development. Such a design could increase the number of independent memory accesses per cycle and reduce contention between lanes. However, the final FPGA bitstream uses the dual-port BRAM approach, mainly because it was simpler and more reliable within the available development time.

8.5 Host Communication Bottleneck

Although the internal BRAM bandwidth is relatively high, the complete system is limited by host communication. The accelerator is controlled by the Zynq-7000 processing system, which runs on the ARM Cortex-A9 side. The host loads instruction memory, initializes data memory, starts execution, waits for the kernel to finish, and then reads output data back from the accelerator.

In the current implementation, this communication path uses UART at 115200 baud. Assuming 10 serial bits per transferred byte, this corresponds to an effective throughput of approximately:

$$\frac{115200}{10} = 11520\,\text{bytes/s} \approx 11.52\,\text{KB/s} \tag{5}$$

This is much lower than the internal BRAM bandwidth. Therefore, for workloads that require frequent transfers between the host and the accelerator, the total runtime is dominated by communication rather than computation.

The generic FPGA runner follows the sequence:

1. load the instruction memory,
2. optionally initialize data memory,
3. write kernel arguments to the `GPGPU_ARGS` region,
4. start the accelerator,
5. wait until the design enters the dumping state,
6. read the requested data-memory window,
7. return the system to the loading state.

This flow is convenient for debugging, but it also means that every kernel launch includes host-side overhead. For small kernels, the overhead of loading and dumping memory can be much larger than the actual compute time. Thus, the design is primarily **memory-transfer bound** at the system level.

8.6 Resource Utilization

The final design uses a significant fraction of the available FPGA resources. The measured utilization is summarized in Table 3.

Table 3: FPGA resource utilization of the final design.

Resource	Utilization %	Utilization
LUT	88%	46551
LUTRAM	1%	2
FF	41%	43357
BRAM	3%	4
DSP	44%	96
IO	2%	3
BUFG	6%	2

The highest utilization is in LUTs, reaching approximately 88%. This is expected because the design instantiates 32 processing lanes, forwarding logic, hazard detection, a shared front-end, and a memory crossbar. The DSP utilization is also significant at 44%, due to the multiplication hardware used by the processing elements.

The BRAM utilization is low, around 3%, because the current design uses small instruction and data memories. This suggests that the design is not limited by available memory capacity. Instead, it is limited mainly by logic resources and by the external communication path.

The utilization results show that the design is close to the practical logic capacity of the target FPGA. Adding more processing elements, more memory banks, or more complex scheduling logic would likely require either further optimization or a larger FPGA.

9 Limitations and Future Work

Although the final system successfully demonstrates a working 32-lane RISC-V-based SIMD accelerator on FPGA, the current implementation is still a prototype. Its main limitations are not related only to the compute core itself, but also to the surrounding system: the host communication path, the available FPGA resources, and the simplicity of the current programming/runtime model.

9.1 UART Communication Bottleneck

The most important system-level limitation is the communication interface between the host and the accelerator. In the current implementation, the Zynq-7000 processing system communicates with the GPGPU through UART. This interface is simple, easy to debug, and useful during development, but it has very low bandwidth.

The UART link runs at 115200 baud. Assuming 10 serial bits per transferred byte, the effective transfer bandwidth is approximately:

$$\frac{115200}{10} = 11520\,\text{bytes/s} \approx 11.52\,\text{KB/s} \tag{6}$$

This bandwidth is much lower than the internal memory bandwidth of the FPGA fabric. As a result, the end-to-end execution time of a program can be dominated by transfers between the ARM Cortex-A9 processing system and the accelerator.

This is especially important because the host-side execution flow currently requires several communication steps:

1. load the instruction memory,
2. initialize or update data memory,
3. write kernel arguments,
4. start the accelerator,
5. wait for completion,
6. dump output data memory,
7. return the accelerator to the loading state.

For small kernels, the actual compute time on the GPGPU can be much smaller than the time spent transferring data. Therefore, even though the accelerator can outperform the ARM Cortex-A9 in compute-heavy kernels such as n-body, the full system is still strongly limited by host-to-accelerator communication.

This makes the current prototype not suitable for workloads that require frequent small kernel launches or large amounts of output data per launch. It works best when the computation per transferred byte is high.

9.2 FPGA Resource Limitations

A second major limitation is the size of the target FPGA. The final design uses a large fraction of the available logic resources. The measured utilization is shown in Table 3.

The most critical resource is LUT usage, which reaches approximately 88%. This leaves limited headroom for adding more complex features, such as more streaming multiprocessors, larger control logic, more advanced scheduling, or a more complex memory subsystem.

DSP usage is also significant at 44%, mainly because the design includes multiplication hardware in the processing lanes. Adding more arithmetic units, supporting more complex operations, or scaling the number of lanes would further increase DSP pressure.

9.3 Memory System Limitations

The current design uses a shared data memory connected to the 32 processing elements through a crossbar. The final implementation uses a dual-port BRAM memory, so only two memory accesses can be served per cycle.

This creates an internal memory bottleneck when many processing elements issue loads or stores at the same time. The crossbar arbitrates between the 32 lanes, grants access to two of them, and stalls the remaining lanes until their memory operations can be served.

A multi-banked memory system was considered and explored during the project. Such a design would allow more independent memory accesses per cycle and reduce contention between lanes. However, due to time constraints and FPGA resource limitations, the final bitstream uses the simpler dual-port BRAM solution.

9.4 Architectural Limitations

The current GPGPU follows a SIMD execution model. All 32 processing elements execute the same instruction stream in lockstep. This is efficient for regular data-parallel workloads, but it limits the types of programs that can be executed efficiently.

The main architectural limitations are:

- **No warp scheduling**: there is only one active instruction stream, so the design cannot switch between multiple warps to hide memory latency.
- **No hardware support for warp divergence**: all lanes must follow the same control-flow path. Programs with lane-dependent branches must be written carefully or transformed in software.
- **Limited instruction support**: the current design supports the integer subset needed by the implemented workloads, but does not support floating-point operations or division.
- **Static memory model**: memory regions and kernel arguments are statically managed by the host and linker script. There is no dynamic memory allocation on the accelerator.

These limitations are acceptable for a first working prototype, but they would need to be addressed for a more general-purpose programmable accelerator.

9.5 Future Work

The most important direction for future work is improving the communication path between the host and the FPGA. Replacing UART with a higher-bandwidth interface would significantly improve end-to-end performance, especially for workloads that transfer input and output data frequently.

A better integration with the Zynq processing system could include:

- memory-mapped communication between the ARM Cortex-A9 and the accelerator,
- DMA-based transfers for instruction and data memory,
- a driver or runtime running under a real operating system,
- fewer host round trips per kernel launch,
- keeping data resident in FPGA memory across multiple kernel calls.

Another important improvement is moving to a larger FPGA with a higher-bandwidth host interface, such as PCIe. A larger FPGA would allow the design to scale beyond the current resource limits and would make it possible to implement:

- more streaming multiprocessors,
- more processing lanes per SM,
- a multi-banked memory system,
- more complex scheduling logic,
- larger instruction and data memories.

On the compute side, future versions could extend the supported instruction set with floating-point operations and division. This would make the accelerator usable for a wider range of workloads. For example, more realistic physics simulations, image processing algorithms, and scientific applications often require floating-point arithmetic. Finally, the programming model could be improved by adding support for warp scheduling, warp divergence handling, and dynamic memory allocation. These features would make the system closer to a real GPU programming model and would reduce the amount of manual work required from the programmer.

Overall, the current prototype validates the main idea: a custom 32-lane RISC-V-based SIMD accelerator can be implemented on FPGA and can accelerate parallel integer workloads. The next step is to remove the system-level bottlenecks, especially UART communication and FPGA resource constraints, so that the compute capabilities of the design can be used more effectively.

10 Conclusion

In this project, we designed and implemented a simple GPGPU-like accelerator on FPGA, based on a custom 32-lane RISC-V SIMD architecture. The final system is able to execute C kernels compiled with the RISC-V GNU toolchain, run them on the FPGA, and communicate with the Zynq-7000 processing system through a host-side runtime.

The design evolved from a simple scalar RISC-V core to a 5-stage pipelined processor with forwarding, hazard detection, branching, jumps, and multiplication support. This core was then transformed into a SIMD architecture by sharing the instruction fetch and decode stages across 32 processing elements. As a result, all lanes execute the same instruction stream in lockstep, while each lane operates on different data through its own thread ID.

A complete software stack was also developed around the hardware. This includes a linker script for separating instruction and data memory, a small bare-metal runtime for stack setup and thread ID access, a build flow for compiling C programs to RISC-V instruction memory images, and a Python-based FPGA runner for loading programs, passing kernel arguments, launching kernels, and reading results back from data memory. This software infrastructure made it possible to run higher-level workloads such as adjacent differences, Mandelbrot rendering, and n-body simulation.

The evaluation shows that the accelerator can provide useful speedup for data-parallel workloads. In particular, the n-body simulation demonstrated that the custom GPGPU can outperform the ARM Cortex-A9 despite running at a much lower clock frequency. This confirms that the architecture is effective when the workload exposes enough parallelism and can be mapped to the 32 SIMD lanes. At the same time, the results also reveal the main limitations of the current prototype. The most important bottleneck is the communication path between the host and the accelerator. The UART interface is simple and reliable for debugging, but its effective bandwidth is very low compared to the internal FPGA memory bandwidth. Therefore, the end-to-end runtime is often limited by memory transfers rather than by computation. The second major limitation is the size of the target FPGA, as the final design already uses a large fraction of the available LUTs and DSPs, leaving limited room for more advanced features.

Overall, the project successfully demonstrates the feasibility of building a small programmable GPU-style architecture on FPGA. The final system combines a custom RISC-V hardware design, a working SIMD execution model, a software runtime, and real FPGA execution. Future work should focus on improving the host interface, using higher-bandwidth communication such as DMA or PCIe, scaling the memory system, and extending the architecture with features such as floating-point arithmetic, warp scheduling, divergence handling, and support for larger FPGA platforms.