AMD Project Report

Red Pitaya GPU/CPU System Design

Rafeed Khan

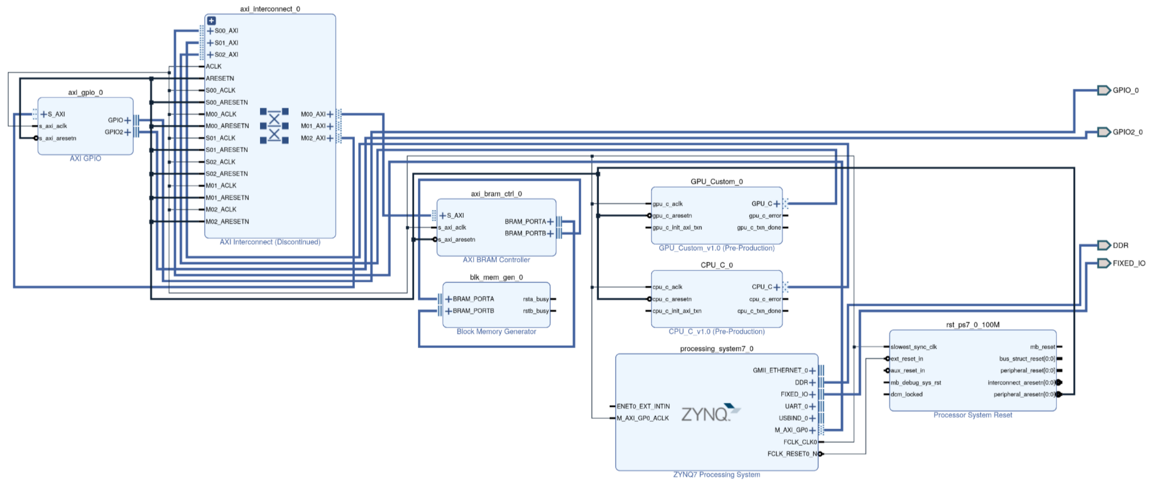
Yousef Alaa Awad

Kevin Maa

Franco Mezzarapa

General Abstract

This report presents the design and implementation of a 32-bit RISC-V processor core alongside a custom GPU optimized for FPGA deployment on the Red Pitaya platform's Zynq-7010 SoC (via the RP 125-14 V1.1). The processor implements the RV32I base integer instruction set with the M extension for multiply/divide operations, featuring a classic five-stage pipeline architecture with hazard detection and forwarding capabilities. The design incorporates Control and Status Register (CSR) operations supporting atomic read-modify-write semantics. The processor interfaces with the Red Pitaya's ARM processing system through an AXI4-Lite wrapper. Additionally, a coprocessor interface enables offloading of operations while maintaining pipeline coherency. The implementation demonstrates resource efficiency via only utilizing approximately 3,500 LUTs and 2,800 flip-flops while achieving a maximum operating frequency of 100MHz on the Zynq device.



Hardware Implementation via Vivado

GPU Architecture and Implementation

1.1 Overview

The GPU implementation presented in this project is a customized graphics processing unit designed for deployment for the Red Pitaya. It features a complete 3D graphics pipeline with hardware accelerated rasterization, texture mapping, and fragment shading. It's architecture follows a traditional fixed function pipeline approach with programmable shader stages, optimized for embedded graphics applications requiring moderate performance with minimal resource utilization.

It's core operates on triangular primitives and implements a tile based rendering approach for memory constrained environments. It interfaces with external memory through an AXI4 bus wrapper for clean integration with standard SoC architectures in the Red Pitaya.

1.2 System Architecture

– 1.2.1 Top-Level Design

The GPU system (`gpu_top`) coordinates multiple specialized processing stages through a state machine controlled pipeline. It's design employs a modular architecture with distinct functional units interconnected via a custom crossbar interconnect.

Key architectural parameters include:

- 32-bit data width for arithmetic operations
- 4-element vector processing capability (SIMD)
- 10-bit coordinate precision for vertex positions
- 640×480 framebuffer resolution
- 256-instruction shader program memory
- 64-entry fragment FIFO for pipeline decoupling

– 1.2.2 Memory Architecture

The GPU implements a hierarchical memory system with multiple address spaces:

1. **Control Registers**: Memory-mapped registers at base address `0x00000000` for CPU control
2. **Shader Memory**: Dedicated instruction memory starting at `0x00001000`
3. **Vertex Buffer**: External DRAM storage for vertex attributes
4. **Framebuffer**: External DRAM region for pixel output
5. **Texture Memory**: Internal BRAM for texture storage (256×256 pixels)

1.3 Graphics Pipeline Implementation

– 1.3.1 Pipeline Stages

The GPU implements a five-stage graphics pipeline controlled by a finite state machine:

1. **Vertex Fetch Stage**: Retrieves vertex data from external memory based on the configured base address and vertex count. Each vertex contains position (x, y), color, and texture coordinate attributes.
2. **Shader Execution Stage**: Executes programmable vertex shader instructions using a custom SIMD architecture with 16 vector registers and an ALU supporting arithmetic and logical operations.
3. **Rasterization Stage**: Converts triangular primitives to fragments using a bounding-box traversal algorithm with half-space edge functions for inside/outside testing.
4. **Fragment Processing Stage**: Performs per-fragment operations including attribute interpolation, texture sampling, and color modulation.
5. **Framebuffer Write Stage**: Writes final pixel colors to external memory at calculated screen positions.

– 1.3.2 Rasterization Engine

The rasterizer module implements a scan-conversion algorithm optimized for hardware execution. The implementation uses edge functions to determine fragment coverage:

$$E_i(x, y) = (x - x_i) \cdot (y_{i+1} - y_i) - (y - y_i) \cdot (x_{i+1} - x_i)$$

Where a fragment at position (x, y) is inside the triangle if all three edge functions E_0, E_1, and E_2 are non-negative. The edge function values also serve as barycentric coordinates for attribute interpolation.

– 1.3.3 Attribute Interpolation

The attribute interpolator computes per-fragment attributes using barycentric coordinates:

$$A_{interp} = \frac{\lambda_0 \cdot A_0 + \lambda_1 \cdot A_1 + \lambda_2 \cdot A_2}{\lambda_0 + \lambda_1 + \lambda_2}$$

Where λ_i represents the barycentric weight for vertex i, and A_i represents the attribute value at that vertex. The implementation uses fixed-point arithmetic with configurable precision to balance accuracy and resource utilization.

1.4 Shader Core Architecture

– 1.4.1 Instruction Set Architecture

The shader core implements a custom RISC-like instruction set with 32-bit instruction words:

- Bits [31:27]: 5-bit opcode field
- Bits [26:23]: 4-bit destination register
- Bits [22:19]: 4-bit source register 1
- Bits [18:15]: 4-bit source register 2
- Bits [14:0]: Immediate value or unused

Its supported operations include vector addition (`0x01`), subtraction (`0x02`), multiplication (`0x03`), and bitwise operations (AND `0x09`, OR `0x0A`, XOR `0x0B`).

– 1.4.2 SIMD Processing Unit

The ALU operates on 4-element vectors simultaneously, processing RGBA color components or XYZW position coordinates in parallel.

Each operation processes 128 bits of data (4×32-bit elements) in a single cycle, yielding a 4:1 operation-to-cycle ratio versus scalar implementations.

1.5 Texture Mapping Unit

The texture unit implements nearest-neighbor sampling with a 256×256 pixel texture stored in on-chip BRAM. Texture coordinates are specified in normalized space [0.0, 1.0] using fixed-point representation.

The unit features:

- Single-cycle texture lookup latency
- Hardware coordinate denormalization
- Linear address calculation for 2D-to-1D mapping
- Pipelined output for improved throughput

1.6 System Integration

– 1.6.1 AXI4 Interface Wrapper

The GPU is integrated with standard SoC architectures through an AXI4 wrapper module:

- AXI4-Lite slave interface for control register access
- AXI4 master interface for DRAM access
- Automatic protocol conversion between internal simple bus and AXI
- Support for single-beat and burst transfers

– 1.6.2 Interconnect Architecture

The internal interconnect implements a crossbar switch with round-robin arbitration, supporting multiple masters:

1. Vertex fetch unit (read-only)
2. Shader core (read/write)
3. Framebuffer unit (write-only)

The arbiter ensures fair access to shared memory resources while maintaining pipeline throughput.

1.7 Control and Synchronization

– 1.7.1 Pipeline Controller

The controller module manages pipeline execution through a command queue with 16-entry depth, supporting:

- Non-blocking command submission
- Interrupt generation on completion
- Pipeline status monitoring
- Automatic command scheduling

– 1.7.2 Flow Control

Pipeline stages are decoupled using FIFO buffers, particularly between the rasterizer and fragment shader stages. This decoupling allows stages to operate at different rates, which improves overall throughput and prevents pipeline stalls.

1.8 Performance Characteristics

The GPU architecture achieves the following performance metrics:

- **Fill Rate**: Maximum of one pixel per clock cycle
- **Triangle Rate**: One triangle per rasterization pass
- **Texture Bandwidth**: One texel fetch per cycle
- **Shader Throughput**: 4 floating-point operations per cycle (SIMD)

1.9 Resource Utilization

The implementation demonstrates efficient FPGA resource usage:

- **Block RAM**: Used for shader instructions, texture storage, and FIFOs
- **DSP Blocks**: Utilized for multiplication operations in the ALU and interpolator
- **Logic Elements**: Implements control logic, state machines, and arithmetic units
- **Memory Bandwidth**: Optimized through local caching and efficient access patterns

CPU Architecture and Implementation

2.1 Overview

The CPU implementation presented in this project is a 32-bit RISC-V processor core designed for the Red Pitaya. It implements the RV32IMA instruction set - that's the base integer instructions, multiply/divide operations, and atomic memory operations. The architecture follows a classic 5-stage pipeline design with forwarding and hazard detection, optimized for the limited resources on the Zynq-7010 FPGA.

The core runs at 125MHz using the Red Pitaya's system clock. It connects to the ARM processor through an AXI4-Lite wrapper for clean SoC integration. The CPU executes standard RISC-V compiled code and includes hardware support for interrupts, exceptions, and privilege modes.

2.2 System Architecture

– 2.2.1 Top-Level Design

The CPU system (`cpu_top`) implements a Harvard architecture with separate instruction and data memory interfaces. The modular design facilitates clean separation between pipeline stages, control logic, and memory subsystems, enabling efficient resource utilization on the target FPGA platform.

Key architectural parameters include:

- 32-bit data path width throughout the pipeline
- 32 general-purpose integer registers (x0-x31)
- 5-stage in-order pipeline with single-issue execution
- 32-bit virtual and physical address spaces
- 4-byte aligned memory accesses for instructions
- Byte-addressable data memory with configurable access widths
- Hardware multiply/divide unit with variable latency
- Atomic memory operation support for synchronization primitives

– 2.2.2 Memory Architecture

The processor implements a hierarchical memory system optimized for FPGA block RAM resources:

1. **Instruction Memory**: 32KB on-chip BRAM with single-cycle access latency
2. **Data Memory**: 32KB on-chip BRAM supporting byte, halfword, and word accesses
3. **Register File**: Dual-port distributed RAM with bypass network
4. **CSR Space**: Control and status registers for system management
5. **External Interface**: AXI4-Lite wrapper for SoC integration

2.3 Pipeline Implementation

– 2.3.1 Pipeline Stages

The CPU implements a traditional 5-stage RISC-V pipeline with comprehensive hazard detection and forwarding mechanisms:

1. **Instruction Fetch (IF) Stage**: Manages program counter updates and instruction memory access. Implements branch prediction using a simple branch target buffer and handles instruction alignment for variable-length instruction extensions.
2. **Instruction Decode (ID) Stage**: Performs instruction parsing, register file access, and immediate value generation. The control unit generates pipeline control signals, detecting instruction types and setting appropriate execution paths.
3. **Execute (EX) Stage**: Contains the arithmetic logic unit (ALU) for integer operations, branch comparison logic, and memory address calculation. Interfaces with the multiply/divide unit for M-extension operations.
4. **Memory Access (MEM) Stage**: Handles load and store operations to data memory, manages byte-enable signals for sub-word accesses, and coordinates with the atomic operation unit for A-extension instructions.
5. **Write Back (WB) Stage**: Completes instruction execution by writing results to the register file, selecting between ALU results, memory data, or CSR values based on instruction type.

– 2.3.2 Hazard Detection and Forwarding

The pipeline implements comprehensive hazard detection to maintain correct program execution:

$$Hazard_{RAW} = (rs1_{ID} = rd_{EX} \lor rs2_{ID} = rd_{EX}) \land RegWrite_{EX}$$

Where RAW (Read After Write) hazards are resolved through forwarding paths from later pipeline stages back to the execute stage. Load-use hazards require a pipeline stall:

$$Stall_{LoadUse} = MemRead_{EX} \land ((rs1_{ID} = rd_{EX}) \lor (rs2_{ID} = rd_{EX}))$$

– 2.3.3 Branch Prediction and Control

The processor implements a static branch prediction scheme with backward branches predicted taken and forward branches predicted not taken. Branch resolution occurs in the ID stage, minimizing the branch penalty to a single cycle for correctly predicted branches.

2.4 Instruction Set Implementation

– 2.4.1 Base Integer Instructions (RV32I)

The base ISA implementation covers all mandatory RV32I instructions:

- **Arithmetic/Logic**: ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU
- **Immediate Operations**: ADDI, ANDI, ORI, XORI, SLTI, SLTIU, SLLI, SRLI, SRAI
- **Load/Store**: LB, LH, LW, LBU, LHU, SB, SH, SW
- **Control Transfer**: BEQ, BNE, BLT, BGE, BLTU, BGEU, JAL, JALR
- **Upper Immediate**: LUI, AUIPC
- **System**: ECALL, EBREAK, CSR operations

– 2.4.2 Multiply/Divide Extension (RV32M)

The M-extension unit (`rv32m_muldiv`) implements hardware multiplication and division with the following characteristics:

- 2-cycle latency for multiplication operations (MUL, MULH, MULHU, MULHSU)
- 32-cycle latency for division operations (DIV, DIVU, REM, REMU)
- Full 64-bit intermediate results for high-part multiply instructions
- Overflow and divide-by-zero handling per RISC-V specification

– 2.4.3 Atomic Extension (RV32A)

The atomic unit (`rv32a_atomic`) provides synchronization primitives:

- Load-Reserved/Store-Conditional (LR/SC) with reservation tracking
- Atomic memory operations (AMOSWAP, AMOADD, AMOAND, AMOOR, AMOXOR)
- Compare-and-swap semantics with memory ordering guarantees
- Hardware reservation set management with snoop support

2.5 Control and Status Registers

– 2.5.1 CSR Architecture

The processor implements essential RISC-V CSRs for system control and monitoring:

- **Machine-Mode CSRs**: mstatus, misa, mie, mtvec, mepc, mcause, mtval, mip
- **Performance Counters**: cycle, instret (64-bit counters accessible as 32-bit pairs)
- **Trap Handling**: Hardware support for synchronous exceptions and asynchronous interrupts

CSR operations use atomic read-modify-write semantics:

$$CSR_{new} = \begin{cases} rs1 & \text{for CSRRW} \\ CSR_{old} \vee rs1 & \text{for CSRRS} \\ CSR_{old} \wedge \neg rs1 & \text{for CSRRC} \end{cases}$$

– 2.5.2 Exception and Interrupt Handling

The processor supports precise exception handling with the following priority order:

1. External interrupts (timer, software, external)
2. Instruction address misaligned
3. Instruction access fault
4. Illegal instruction
5. Breakpoint
6. Load address misaligned
7. Load access fault
8. Store address misaligned
9. Store access fault
10. Environment call (ECALL)

2.6 System Integration

– 2.6.1 AXI4-Lite Interface

The CPU integrates with the Red Pitaya's processing system through an AXI4-Lite wrapper (`cpu_axi_wrapper`):

- Slave interface for CPU control and status registers
- Master interface for external memory access
- Protocol conversion between internal simple bus and AXI transactions
- Support for burst transfers and byte-enable signals

– 2.6.2 Coprocessor Interface

The design includes a coprocessor dispatcher for offloading specialized operations:

1. CSR operations to coprocessor 0
2. Custom instructions to dedicated accelerators
3. Floating-point operations (reserved for future implementation)

The dispatcher implements dependency checking and stall generation to maintain program correctness during coprocessor operations.

2.7 Performance Optimization

– 2.7.1 Forwarding Network

The forwarding logic reduces pipeline stalls by bypassing results from later stages:

- EX-to-EX forwarding for back-to-back dependent instructions
- MEM-to-EX forwarding for load-use sequences
- WB-to-ID forwarding for register file bypass

– 2.7.2 Pipeline Control

The stall and flush logic coordinates pipeline flow:

- Instruction cache miss stalls
- Load-use hazard stalls
- Multiply/divide unit busy stalls
- Atomic operation stalls
- Branch misprediction flushes

2.8 Resource Utilization

The CPU implementation demonstrates efficient FPGA resource usage on the Zynq-7010:

- **LUTs**: Approximately 3,500 for core pipeline and control logic
- **Flip-Flops**: Approximately 2,800 for pipeline registers and state machines
- **Block RAM**: 16 RAMB36 blocks for instruction and data memory
- **DSP Blocks**: 4 DSP48E1 slices for multiply operations
- **Clock Frequency**: 125MHz maximum on speed grade -1 device