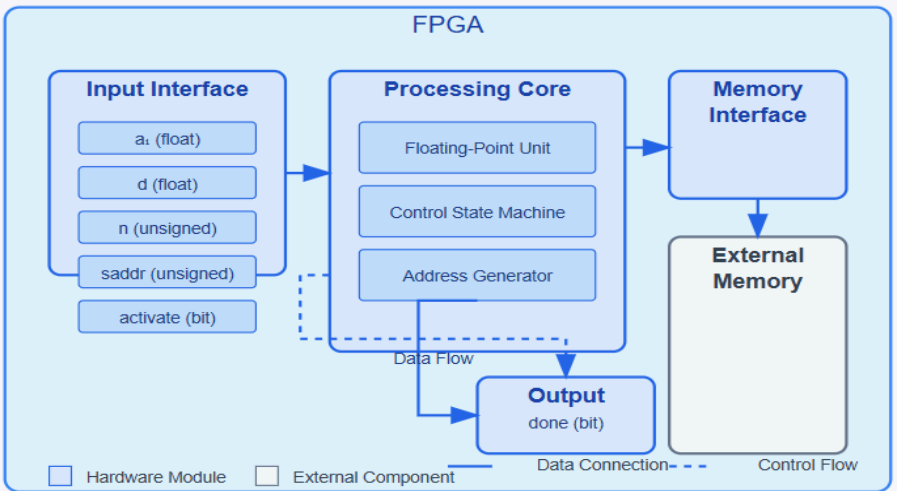
ASG - Arithmetic Series Generator: A Hardware Acceleration Journey

Posted on April 20, 2025 by Team FPGAcelerate



Architecture of a ASG FPGA accelerator

Group 6

Team Members:
Morris Nkomo (NKMMOR001)
Muzerengwa Vincent (MZRVIN001)
Teddy Umba Muba (TDDMUB001)
Mpumelelo Mpanza (MPNMPU002)

1. Project Description

An arithmetic series is a series of numbers where each term increases or decreases by a constant difference to form a sequence[1]. The series follows the form:

$$a(n) = a_1 + (n - 1) \cdot d$$

Where,

- a1, is the first term
- d, is step size
- n, is term index

Arithmetic Series are used in various applications. An example of such an application would be graphics and imaging pipelines where linear addressing is needed to walk through pixels in a line or a block, an arithmetic series would be ideal for calculating such addresses.

Computer programs can be used to easily calculate the sequence for small values of n however, challenges begin when:

- n becomes too large
- Real-time processing is required
- Floating point arithmetic is used

In such cases,software implementation can become inefficient. This is because a CPU must compute and store each term in memory which can lead to performance bottlenecks. Systems which use traditional memory (e.g. DDR) may experience memory access latency which may slow down storage of each term[2].

2. Proposed Solution

To address these limitations, a proposed solution is to implement an Arithmetic Signal Generator (ASG) on an FPGA [3]. When the ASG is activated, it generates **n** terms of the arithmetic sequence and writes them sequentially into memory, starting at a specified base address, **saddr.** Once the sequence is complete, the ASG sends a **done** signal to indicate successful completion.

Advantages of using FPGA to implement ASG is that:

- It can compute and store one term per cycle
- An FPGA can achieve lower latency as compared to software implementations
- It can efficiently handle floating point arithmetic using custom logic
- It can write results directly to memory with minimal overheads

3. Prototype Specification

The prototype implementation of the Arithmetic Series Generator (ASG) is designed to harness the parallel processing capabilities of Field-Programmable Gate Arrays (FPGAs) for high-throughput arithmetic sequence generation. This specification outlines the core architecture, functional modules, interfaces, parameterization capabilities, and targeted performance metrics of the prototype, in alignment with the goals stated in the following section.

3. 1. System Overview

At a high level, the ASG accelerator is architected around a pipeline that generates arithmetic terms in real-time and writes them to memory with minimal latency. The design emphasizes one-term per cycle throughput, low latency memory access and support for both fixed and floating point formats. The system architecture includes:

- Configuration Interface
- Sequence Generation Core (SGC)
- Floating-Point Unit (FPU)
- Memory Controller
- Control & Status Registers (CSR)
- Interrupt-capable Done Signal Generator

Each component is optimised for low-latency operation and minimal resource utilisation while preserving computation precision.

3. 2. Functional Components

3.2.1. Configuration Interface

The ASG is configured by writing to memory-mapped control registers and it receives the following configuration parameters from a host processor:

- **`a1(float)`**: First term of the sequence
- **`d(float)`**: Common difference
- **`n(unsigned int)`**: Number of terms
- **`saddr(unsigned int)`**: Start address in memory for writing output
- **`activate(bit):`** Start signal

These parameters are loaded into dedicated control registers via a memory-mapped AXI-lite interface.

3.2.2. Sequence Generation Core (SGC)

The SGC is the computational heart of the ASG. It iteratively computes the i-th term using the recurrence relation:

$$a_i = a_1 + (i - 1) \cdot d$$

The SGC operates in a fully pipelined fashion to sustain one output per clock cycle under optimal conditions.

The generator supports:

- **Streaming architecture:** New term computed every cycle
- **Loop unrolling**
- Support for pipeline stalls if memory is not ready

3.2.3. Floating-Point Unit (FPU)

A custom-designed floating-point unit, compliant with IEEE 754 single-precision format, is integrated for high-accuracy operations. Features include:

- Guard bits to minimize rounding errors
- Support for pipelined multiply-add operations
- Configurable rounding modes
- Latency-optimised datapath

3.2.4. Memory Controller

The memory subsystem uses a burst-based AXI4 interface to perform sequential writes. It features:

- Write buffering to decouple computation from memory latency
- Burst generation logic for contiguous address writes
- Backpressure support to avoid stalling the generator pipeline

Also handles the writing of generated terms into memory:

- AXI4 burst transfers
- FIFO buffering decouples computation from memory writes
- Burst size auto-adjusts based on **n**
- Backpressure-aware: stalls pipeline if memory not ready

3.2.5. Control and Status Registers (CSR)

The CSR block maintains internal state and supports:

- Start/stop control
- Progress tracking
- Error reporting
- Completion indication (*done* signal)

3.3. Parameterization

The ASG is designed to be scalable and reusable. Key parameters are:

- **`DATA_WIDTH`**: Configurable from 32 to 64 bits
- **`TERM_COUNT_WIDTH`**: Adjustable bit width for n to support small and large sequences
- **`ADDR_WIDTH`**: Memory address size adaptability
- **`FP_MODE`**: Compile-time switch between fixed-point and floating-point modes

3.4. Timing and Throughput

- Initiation Interval: 1 cycle (new term every cycle)
- Latency per term: 3-5 cycles (depending on FPU pipeline depth)
- Total latency: Approximately **`init_latency + n`** cycles for sequence length n
- Peak throughput: 1 term per clock cycle

3.5. Resource Utilization (Targeted)

For a typical 32-bit floating-point configuration on a Xilinx Artix-7 or Intel Cyclone V:

- LUTs: ≤ 2,500
- FFs: ≤ 1,500
- DSP Blocks: ≤ 10
  BRAM: ≤ 4 blocks (for buffering and CSR)

3.6. Clocking and Reset

- Clock frequency: Target 100–150 MHz
- Reset: Active-low asynchronous reset with pipeline flush logic

3.7. Verification Strategy

The prototype is accompanied by a testbench suite that verifies:

- Correctness across a range of **`a1`**, **`d`**, and **`n`**
- Edge conditions (e.g., **`n = 0, d = 0`**)
- Floating-point precision against IEEE-compliant software models
- Stress tests for memory backpressure and long sequences

3.8. Integration Considerations

- Interfaces with a soft-core processor or host CPU via AXI
- Compatible with standard FPGA design workflows (Vivado, Quartus)
- Synthesis scripts provided for RTL-to-bitstream generation
- Potential for HLS migration for future scalability

3.9. Prototype Implementation Milestones

- Phase 1: RTL design and functional simulation
- Phase 2: FPGA synthesis, floorplanning, and timing closure
- Phase 3: Hardware-in-the-loop (HIL) testing with host integration
- Phase 4: Optimization for area and power efficiency
- Phase 5: Documentation and performance benchmarking

4. Project Goals

Our team has established the following goals for the ASG accelerator:

1. **Performance Optimization**: Create an accelerator that generates arithmetic sequences at least 8x faster than equivalent software implementations on general-purpose processors.
2. **Resource Efficiency**: Design the accelerator to use minimal FPGA resources (LUTs, DSP blocks, memory) while maintaining high performance.
3. **Floating-Point Precision**: Ensure accurate floating-point calculations that match software implementations within acceptable error margins.
4. **Memory Interface Efficiency**: Develop a memory interface that can handle high-speed sequential writes without becoming a performance bottleneck.
5. **Scalability**: Support generation of sequences with varying lengths (from small to very large values of n) with consistent performance.

6. **Verification Framework**: Create a comprehensive testing framework to verify correctness across various input parameters.
7. **Documentation and Analysis**: Provide thorough documentation of the design decisions, implementation details, and performance analysis.
8. **Real-World Application**: Demonstrate the accelerator's utility in at least one practical application scenario.

References

[1] Siyavula, "1.1 Arithmetic sequences," in *Mathematics Grade 12: Sequences and Series*, 1st ed., Cape Town, South Africa: Siyavula Education, 2017. [Online]. Available: https://www.siyavula.com/read/za/mathematics/grade-12/sequences-and-series/01-sequences-and-series-01. [Accessed: April, 2025].

[2] B. Bailey, "CPU Performance Bottlenecks Limit Parallel Processing Speedups," *Semiconductor Engineering*, Dec. 2, 2020. [Online]. Available: https://semiengineering.com/cpu-performance-bottlenecks-limit-parallel-processing-speedups/. [Accessed: April, 2025].

[3] Altera Corporation, “Implementing High-Performance Floating-Point Arithmetic in FPGAs”, White Paper WP-01222-1.2, Intel FPGA, San Jose, CA, USA, Nov. 2015.

Comments Section

Have you worked on hardware acceleration projects before? We'd love to hear your experiences in the comments below!

John_Doe92:

This is a fascinating project! I’ve worked on a similar FPGA-based accelerator for geometric series. How do you plan to handle floating-point rounding errors?

Team FPGAcelerate (Reply):

Great question! We're using custom floating-point units with guard bits to minimize rounding errors. We'll also include error analysis in our verification framework.

Trever_libdt:

Have you considered using HLS (High-Level Synthesis) for this, or is it all RTL?

Team FPGAcelerate (Reply):

We're starting with RTL for fine-grained control over performance and resources, but we might explore HLS for higher-level optimizations later.

This project is part of the EEE4120F High Performance Embedded Systems course at the University of Cape Town.