Arithmetic Series Generator

Morris Nkomo, Teddy Umba Muba, Vincent Muzerengwa, Mpumelelo Mpanza
Department of Electrical and Computer Engineering, EEE4120F
University of Cape Town
NKMMOR001
TDDMUB001
MZRVIN001
MPNMPU002

***Abstract*—**

The primary objective is to develop a robust system capable of handling variable input sizes while maintaining consistent throughput and predictable latency. The design leverages modern hardware description methodologies to ensure optimal resource utilization and parallel processing capabilities.

The study focuses on characterizing the system's scalability and identifying potential bottlenecks, providing insights into its behavior across different operational scales.

The project contributes to the field of hardware acceleration by offering a structured approach to performance analysis and optimization. It establishes a framework for assessing design efficiency, with implications for real-time applications in signal processing, data analytics, and other compute-intensive domains. Future work may explore enhancements to the architecture, integration with larger systems, and comparative studies with alternative implementations.

I. Introduction

In modern digital systems, the need for efficient and scalable signal generation mechanisms is pivotal, especially in applications involving digital signal processing, telecommunications, and hardware accelerators. Traditional signal generators often rely on analog circuitry or fixed-function digital designs [1], which may lack flexibility and adaptability. The concept of an Arithmetic Signal Generator presents an innovative approach by leveraging arithmetic-based operations to simulate various signal patterns, offering the potential for greater control, reconfigurability, and performance optimization.

A. Objectives and Motivation

The primary objective of this project is to design and implement an Arithmetic Signal Generator and evaluate its performance across multiple implementation platforms. These platforms include:

- A serial implementation in C language
- A SystemVerilog implementation targeted for deployment on an FPGA
- An OpenCL implementation, also targeted for the FPGA, designed to leverage high-level synthesis for parallel acceleration

The motivation behind this study stems from the growing demand for high-performance, low-latency, and energy-efficient signal processing in embedded systems [2]. By exploring different implementation paradigms, we aim to analyze how each impacts key performance metrics such as execution time, memory usage, throughput, and speedup. This investigation is crucial for making informed architectural choices in future signal generation and DSP applications.

B. System Requirements

The system under development must fulfill the following requirements:

Generate deterministic arithmetic-based signal waveforms (e.g., sinusoidal, triangular, or other patterns derived arithmetically)

Allow input parameters to define waveform characteristics (frequency, amplitude, phase)

Be implementable in multiple environments: C for baseline serial performance, SystemVerilog for low-level hardware design, and OpenCL for high-level hardware acceleration

Enable data collection for the following performance metrics:

- Execution time
- Memory usage
- Throughput
- Speedup (relative to baseline serial implementation)

C. Scope & Limitations

This project is scoped to evaluate three implementations of the arithmetic signal generator algorithm within a constrained development and testing environment. While the serial and hardware implementations will provide performance insights, certain limitations must be acknowledged:

- Hardware resource constraints may limit the complexity or resolution of generated signals.
- Time constraints might restrict extensive optimization, especially for the OpenCL implementation.
- Only a limited set of signals (likely simple periodic signals) will be tested to ensure comparability across platforms.
- External factors such as toolchain support and simulation accuracy may affect performance measurements.

D. Report Outline

This report is structured as follows:

- Background: A deeper dive into the theoretical and practical underpinnings of arithmetic signal generation, with supporting literature.
- Methodology: Explanation of the implementation strategy and performance measurement plan.
- Design: Detailed system architecture and module-level descriptions.
- Proposed Development Strategy: Exploration of how the solution could evolve in a commercial setting.
- Planned Experimentation: Setup and design of performance experiments.
- Results: Collected data, visualizations, and analysis of the performance across implementations.
- Conclusion: Summary of outcomes, reflections on objectives, and future work.
- References: Cited academic and technical sources used throughout the report.

II. Background

Signal generators are a foundational component in digital systems, enabling the simulation, testing, and analysis of various signal-processing and communication applications [3]. In classical systems, signal generation is achieved through analog circuitry or dedicated digital signal processors (DSPs) [4]. However, these approaches may not always offer the flexibility, portability, or performance required for modern embedded and reconfigurable systems. The Arithmetic Signal Generator presents a digital, computationally driven alternative capable of producing signals through arithmetic functions and iterative computation, making it well-suited for both software and hardware implementations.

A. Motivation for Arithmetic-Based Signal Generation

The concept of generating signals arithmetically is rooted in the ability to express complex waveforms using mathematical equations, such as sinusoidal, triangular, or square waveforms. These can be computed using series expansions (e.g., Taylor or Fourier series), lookup tables, or recursive algorithms. Arithmetic generation enables:

- Precise control over waveform parameters
- Scalability with respect to resolution and frequency
- Efficient implementation on both CPUs and hardware accelerators (FPGAs/GPUs)

As embedded systems evolve, designers are increasingly exploring arithmetic signal generation due to its reconfigurability and compatibility with digital design flows [5].

B. Literature and Source Review

Existing literature provides a strong theoretical foundation for the design and implementation of digital signal generators. Studies in digital signal processing emphasize the efficiency of arithmetic algorithms in waveform synthesis, particularly when utilizing techniques like recursive computation and lookup-based approximations [6]. These methods are widely adopted in digital systems to achieve real-time performance with minimal computational overhead.

In the context of hardware design, educational and industry-level materials highlight the benefits of pipelined architectures and modular design strategies for implementing signal generators in HDL-based environments such as Verilog or VHDL. These architectures promote determinism, high throughput, and ease of scaling, essential characteristics for real-time systems.

Furthermore, sources focusing on parallel computing architectures explore the performance advantages of using data-parallel models (such as OpenCL) for compute-intensive signal generation. These approaches support concurrent computation of multiple signal values, resulting in reduced execution time and increased throughput when deployed on platforms like FPGAs.

Collectively, the reviewed content emphasizes that:

- Arithmetic computation is a robust and scalable method for signal synthesis.
- Hardware-based designs (FPGA/SystemVerilog) provide deterministic and low-latency signal generation.
- High-level parallel languages (like OpenCL) offer a flexible route to harness acceleration capabilities without deep HDL-level design.
- The comparison of different platforms is a common method in performance engineering to evaluate trade-offs in latency, throughput, memory efficiency, and ease of implementation.

C. Supporting Trends and Industry Applications

In recent years, high-level synthesis (HLS) and hardware acceleration have seen increased adoption, especially in fields such as:

- Real-time signal processing
- Radar and software-defined radio (SDR)
- Medical imaging systems
- IoT edge devices requiring fast signal analysis

Platforms such as OpenCL allow for the parallel execution of computational kernels, which can greatly accelerate tasks like signal generation when compared to traditional serial implementations. SystemVerilog enables low-level optimization and pipelining, ideal for deterministic timing and minimal latency, an essential aspect in hardware-based signal processing.

The comparison of these implementations across dimensions such as execution time, throughput, memory usage, and speedup is a valuable endeavor. It reflects the real-world engineering trade-offs between ease of development, performance, and portability, making this project particularly relevant to current research and industry needs.

D. Summary

The arithmetic signal generator represents a compelling alternative to traditional waveform generation techniques. With

a strong foundation in digital signal processing theory and emerging support from parallel programming models and reconfigurable hardware, it stands as a powerful example of modern computational engineering. This project aims to investigate its performance and practicality through multiple implementation strategies, using foundational knowledge from both academic literature and contemporary toolchains.

III. Methodology

This section outlines the comprehensive methodology used for designing, implementing, and evaluating an Arithmetic Sequence Generator (ASG) across three paradigms: a serial software implementation in C, a hardware-centric design using Verilog RTL, and a parallel computing approach using OpenCL.

A. Overview of Methodology

Serial C Implementation

The C-based implementation served as the functional baseline and was used to verify the outputs of both the hardware and parallel implementations.

Completed Work:

- Developed a **reference model** that performs arithmetic sequence generation with floating-point precision.
- Integrated **IEEE-754 compliance validation**, ensuring results align with expected floating-point behavior.
- Designed an **automated test harness** for verifying correctness against edge cases.
- Employed **performance profiling tools** like `gprof` and `valgrind` to measure execution time, memory usage, and identify optimization opportunities.

Algorithmic Strategy: .

The generation logic follows a straightforward linear algorithm:

```
for (int i = 0; i < n; i++) {
    output[i] = a1 + i * d;
}
```

This implementation provides a predictable execution profile, which is essential for establishing timing and performance baselines.

Verilog RTL Implementation

This hardware-centric implementation aims to exploit pipelining and parallelism for speed improvements, especially in fixed-function embedded systems or FPGA-based accelerators.

Completed Work:

- Designed a **parameterized datapath** to support configurable floating-point formats and bit widths.
- Implemented a **6-stage pipelined floating-point adder**, based on IEEE-754, for high-throughput calculations.
- Created a **7-state Finite State Machine (FSM)** to control sequence generation, memory interface, and synchronization signals.
- Built a **testbench environment** using SystemVerilog for simulation-based validation.

Potential Enhancements:

- Multi-device partitioning for very large n
- Mixed-precision computation modes
- FPGA accelerator integration via OpenCL

.

OpenCL Implementation

This software-accelerated implementation targets heterogeneous platforms, using OpenCL to utilize available computing resources such as CPUs and GPUs for parallel execution of arithmetic sequence generation.

Completed Work:

- Developed a kernel function that generates terms of the arithmetic series in parallel using global and local ID indexing
- Parameterized the kernel to support different floating-point precisions
- Implemented host-side logic in C/C++ to manage buffer allocation, kernel argument configuration, and command queue execution

Potential Enhancements:

- Overlapping computation and memory transfer using asynchronous OpenCL commands
- Incorporation of vector data types to further improve memory access patterns and SIMD utilization.
- Kernel fusion with post-processing steps to reduce global memory I/O overhead.

Our comparative analysis is split into two main segments:

1) Comparison of the Serial C Implementation with the Verilog RTL Implementation.
2) Comparison of the Verilog RTL Implementation with the OpenCL Parallel Implementation.

.

The methodology focuses on evaluating each design's correctness, performance, scalability, and computational efficiency under consistent benchmarks.

B. Comparing Serial C vs Verilog RTL Implementation

1. Overview of Approach

The primary objective in this stage was to validate the functionality of the ASG across both software and hardware platforms. The C implementation provided a straightforward, baseline model. Meanwhile, the Verilog RTL model featured hardware-accurate timing and control, using a finite state machine (FSM) and a custom floating-point adder.

By comparing the wall-clock time of the C implementation with the simulated cycle count of the Verilog design, we obtained insights into the trade-offs between software flexibility and hardware determinism.

1) Serial C Implementation: The serial C implementation performs arithmetic sequence generation using a simple *for* loop. Here's the core logic:

```
for (int i = 0; i < n; i++) {
    output[i] = a1 + i * d;
}
```

Each term of the sequence is calculated using the formula:

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

Where:

- *a1* is the first term of the sequence.
- *d* is the common difference.
- *n* is the number of terms.

The loop iteratively computes each term and stores it in an output array. Execution time is measured using *gettimeofday()* to capture wall-clock performance. Floating-point correctness is ensured by verifying the output against the expected IEEE-754 compliant values.

2. Verilog RTL Implementation

In the Verilog version, the ASG was implemented at the register-transfer level with explicit control logic and a custom-built floating-point adder.

2) FSM Design and Diagram: To manage control flow and data sequencing, a finite state machine (FSM) was employed with the following states:

- IDLE: Wait for the activation signal.
- INIT: Load initial sequence parameters (a1, d, and counter i).
- LOAD_ADD: Initiate floating-point addition.
- WAIT_ADD: Wait for the adder to complete.
- WRITE: Write the result to memory.
- WAIT_MEM: Wait for the memory write acknowledgment.
- FINISH: Terminate sequence generation and raise the done flag.

Each path depends on control signals such as *activate*, *done*, *adder_done*, and memory acknowledgment flags.

3) Floating-Point Adder Module: A key component in the RTL implementation is the floating-point adder. This was manually implemented to conform with IEEE-754 standards. The module performs:

- Exponent comparison and alignment
- Mantissa addition or subtraction based on operand signs
- Normalization and rounding of the result
- Repacking into IEEE-754 format

This module operates over several clock cycles and signals its completion using a *done* flag to the FSM.

4) Pseudocode Representation: The ASG logic in hardware pseudocode is as follows:

```
if (activate) {
    current_term = a1;
    for (i = 0; i < n; i++) {
        next_term = current_term + d;
        mem[i] = current_term;
        current_term = next_term;
    }
    done = 1;
}
```

This pseudocode captures the intended sequential logic executed by the FSM and datapath units in the Verilog implementation.

3. Measurement Methodology

Each design was evaluated using the following metrics:

- Cycle Count: Extracted from simulation logs.
- Latency per Term: Total cycles divided by n.
- Throughput: Number of sequence elements generated per second.

Cycle-level simulation was carried out using ModelSim or Verilator. Wall-clock comparisons between C and Verilog were normalized by considering clock frequency and simulation time.

C. Comparing Verilog RTL vs OpenCL Implementation

1. Motivation and Setup

While Verilog RTL ensures tight control over timing and resources, it is inherently sequential unless explicitly parallelized. To explore parallelization at scale, OpenCL was used to implement a highly parallel kernel capable of computing each term of the ASG concurrently.

This comparison focuses on:

- How data-level parallelism improves throughput
- Performance scaling with large data sets
- Host-device interaction overheads

.

2. OpenCL Parallel Implementation

1) Kernel Logic and Execution Diagram: In OpenCL, each work-item computes a single term of the sequence:

```
__kernel void asg_parallel(float a1, float d,
    unsigned int n, __global float* output) {
    int i = get_global_id(0);
    if (i < n) output[i] = a1 + i * d;
}
```

This logic is analogous to the C version but benefits from concurrent execution. For example, 1024 work-items can compute 1024 elements simultaneously.

The OpenCL workflow is shown below (Figure 1):

Each work-item (indexed by i) computes one term of the sequence. Work-items are executed in parallel by the OpenCL runtime. The global size of the kernel is set to n, meaning n elements are computed in parallel if hardware permits.

3. Measurement Tools and Metrics

- Kernel Execution Time: Measured with clGetEventProfilingInfo.
- Wall Time: From host-side timestamps around clEnqueueNDRangeKernel.
- Throughput: Calculated as n / kernel_time.
- Parallel Efficiency: Measured by comparing execution times across varying n sizes.

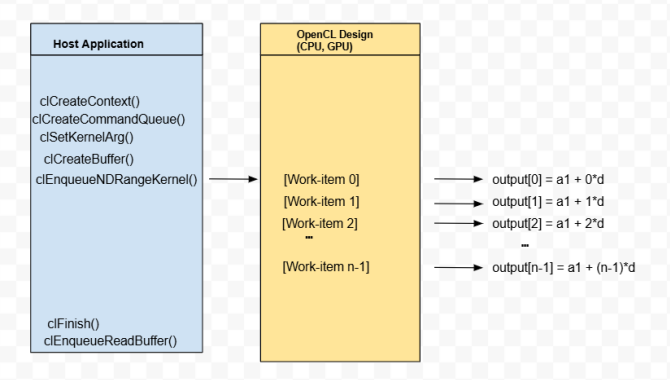

Fig. 1. OpenCL Workflow

IV. Design of the Accelerator System

This section delves into the architectural and system-level design of the Arithmetic Sequence Generator (ASG) accelerator. It elaborates on the internal architecture, host integration strategies, and modular design principles aimed at enabling flexibility, performance, and portability across platforms, especially for FPGA deployment. The design philosophy emphasizes clean modularity, clear separation of control and datapath, and ease of integration with software stacks such as OpenCL.

A. Design Goals

The ASG accelerator was designed with the following key objectives:

- Correctness: Produce IEEE-754 compliant floating-point sequence values.
- Throughput: Maximize number of terms generated per second.
- Scalability: Allow operation on large n values with minimal redesign.
- Portability: Integrate with simulation deployment platforms (e.g., FPGA, GPU).
- Host Compatibility: Maintain a clear and efficient protocol for host communication.

B. Accelerator System Architecture

The ASG (Arithmetic Sequence Generator) accelerator system is designed as a modular hardware architecture, composed of distinct functional blocks that work together to compute and store a sequence of values efficiently. These blocks include the control unit (implemented as a finite state machine), a computation datapath, a memory interface, and host interface logic. Each module is tailored to handle specific tasks, ensuring a clean separation of control, computation, communication, and coordination.

1. Control Unit (FSM)

The Control Unit is implemented as a Finite State Machine (FSM) that governs the flow of operations within the accelerator. It transitions through a defined sequence of states—*INIT*, *ADD*, *WRITE*, and *FINISH* that reflect the stages of computation and data handling. This unit encapsulates the sequencing logic and dictates how and when each part of the accelerator operates. It interfaces with critical control signals such as activate (to start computation), done (to signal completion), and mem_ack (to confirm memory write operations). The FSM ensures synchronized and deterministic operation of the ASG.

2. Datapath

The datapath forms the computational core of the ASG. It includes operand registers to store the input values *a1* (initial term), *d* (common difference), and the current_term (the current value being computed). A floating-point adder—either custom-designed or sourced as a vendor IP core—performs the arithmetic addition required to generate each successive term in the sequence. An index counter keeps track of the current position in the sequence and helps determine when the operation is complete. Data movement within the datapath is managed via buses controlled by multiplexers, enabling flexible routing between registers and computational units.

3. Memory Interface

The Memory Interface is responsible for writing the generated sequence terms to the output memory. It includes handshaking logic, such as mem_write and mem_ack signals, to ensure that data is reliably transferred and acknowledged before proceeding to the next term. The memory address for each write operation is derived directly from the index counter, ensuring sequential storage of the generated terms. This module acts as the bridge between the datapath and the external memory system, guaranteeing correct and timely data storage.

4. Host Interface Logic

The Host Interface Logic enables external control and monitoring of the accelerator by a host processor. It allows the host to configure key parameters, including *a1*, *d* and *n* (number of terms), through either dedicated registers or a memory-mapped I/O interface. A *start* signal from the host initiates computation, while a *done* flag is exposed for synchronization, indicating when the sequence generation is complete. This module plays a crucial role in integrating the ASG accelerator into a larger system, facilitating communication and coordination with software-level control logic.

5. Architectural Layout

The ASG accelerator comprises several functional blocks operating in unison, orchestrated by a finite state machine (FSM). The block-level view is shown below:

Together, these components facilitate a pipelined yet deterministic generation of arithmetic sequences. The modularity enables future improvements, such as custom FPUs or streaming support.

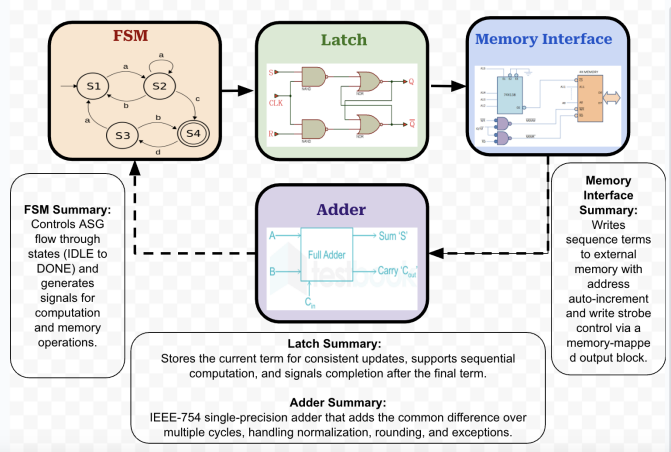

Fig. 2. ASG accelerator functional blocks

C. Host-Accelerator Integration Strategy

To effectively utilize the Arithmetic Sequence Generator (ASG) accelerator, integration with a host CPU or embedded processor is essential. The host is responsible for initializing the accelerator, supplying input parameters, triggering execution, and retrieving results. Two primary models for host integration are proposed: Memory-Mapped I/O (MMIO) and OpenCL-based interfacing. Each offers a different balance between low-level control and high-level abstraction, catering to a range of hardware platforms and development workflows.

1. Memory-Mapped Register Interface (MMIO)

In this model, the ASG accelerator is mapped into the host's address space via memory-mapped control and data registers. Key registers typically include fields for the first term *a1*, the common difference *d*, the total number of terms *n*, a *start* control bit, and a *done* status bit. The host initiates operation by writing the input parameters to the corresponding registers and setting the start bit. The accelerator begins computation and raises the *done* bit upon completion, allowing the host to detect when the process has finished. This approach is well-suited for FPGA-based systems that incorporate soft-core processors such as Xilinx MicroBlaze or Intel Nios II, enabling tight coupling between software and hardware components.

3. OpenCL Host-Kernel Communication

Alternatively, the ASG accelerator can be integrated using an OpenCL host interface, which abstracts the interaction through an API-based paradigm. In this setup, the ASG logic is compiled into an OpenCL kernel. Input and output data are passed through OpenCL buffer objects, and the host interacts with the accelerator by enqueuing kernel executions and managing memory transfers. This method provides a higher level of abstraction and is especially beneficial in systems that already utilize OpenCL for heterogeneous computing. It enables portability across a variety of platforms, including GPUs, FPGAs, and custom compute nodes, and fits seamlessly into modern, parallel compute workflows.

4. Host-ASG Interface Block Diagram

This revised diagram explicitly reflects the communication protocols used in both integration methods (MMIO and OpenCL), providing better clarity on data/control flow between the host and the ASG.

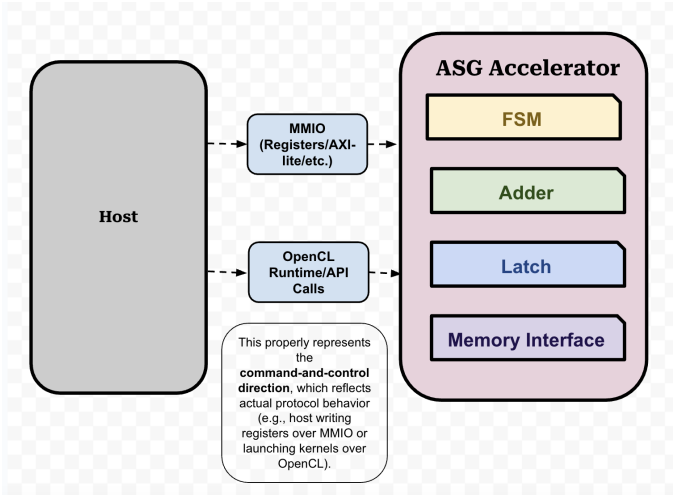

Fig. 3. Host-ASG Interface Block Diagram

V. Proposed Development Strategy

To transition the Arithmetic Sequence Generator (ASG) accelerator into a viable commercial product, a well-structured development strategy must be established. This strategy should prioritize portability, ease of integration, and extensibility to support a broad spectrum of developers, platforms, and applications. The following roadmap outlines the essential components required to achieve this goal.

A. Software Development Toolkit (SDK)

A comprehensive SDK is critical to abstract the underlying hardware complexity and provide a developer-friendly interface. The toolkit would include API libraries in C/C++ and Python, allowing users to configure sequences, manage execution, and retrieve results without needing in-depth hardware knowledge. Additional utilities for device discovery, buffer management, and error handling would enhance usability. The SDK should also feature example applications and reference drivers to accelerate adoption and facilitate rapid prototyping.

B. Runtime and Middleware Support

A lightweight runtime environment is needed to handle memory transactions, synchronization, and data marshaling between the host and the accelerator. This layer would simplify interaction with the hardware and enable seamless integration with high-level frameworks. Middleware plugins should support standards like OpenCL and SYCL, and extend to popular data science platforms such as TensorFlow or PyTorch for use cases like numeric sequence preprocessing or simulation input

generation. This enables the ASG to fit naturally into modern heterogeneous computing workflows.

C. Hardware Abstraction Layer (HAL)

The HAL acts as a bridge between high-level software and the low-level hardware interface [7]. It encapsulates the intricacies of communication protocols, including MMIO register access and AXI-Lite handshaking. By providing a clean and consistent interface to the ASG IP core or OpenCL kernel, the HAL shields developers from hardware-specific details and simplifies integration across different system architectures.

D. Simulation and Debugging Tools

Robust development requires simulation and debugging support. A functional simulator would allow software teams to validate their logic using testbenches without access to physical FPGA hardware, accelerating the development cycle. Integration with waveform viewers such as GTKWave or ModelSim enables in-depth signal analysis during hardware simulation [8]. Additionally, the inclusion of runtime instrumentation features—such as logging, performance counters, and trace analysis—would support profiling and optimization during deployment and testing.

E. Deployment and Platform Support

To ensure wide accessibility, the ASG accelerator should be deployable across multiple hardware platforms. Pre-built bitstreams should be provided for popular FPGA boards, such as the Intel DE-series and Xilinx ZCU boards. Platform-specific drivers and runtime layers for both Linux and Windows—using UIO frameworks or custom kernel modules—would further broaden compatibility. Cloud deployment options, including support for FPGA services like AWS F1 instances or Intel DevCloud, would enable scalable, on-demand access to the ASG in remote or enterprise computing environments.

F. Documentation and Training Resources

Clear and comprehensive documentation is essential for user onboarding and long-term adoption. Resources should include step-by-step tutorials, FSM diagrams, and detailed flowcharts illustrating data paths and control flows. API references and HDL design walkthroughs would support both software developers and hardware engineers, creating a well-rounded knowledge base for effective development and customization.

G. Conclusion

The ASG accelerator presents a robust and adaptable solution for hardware-based arithmetic sequence generation. Its modular architecture, FSM-driven control, and flexible host interface make it suitable for a wide range of platforms, from embedded FPGAs to cloud-based accelerators. To support commercial deployment, the development strategy must include a full-featured ecosystem—spanning software APIs, simulation tools, middleware, and documentation—that enables seamless integration and efficient use across diverse applications. Future enhancements such as pipelined floating-point units (FPUs), DMA-based memory access, and dynamic parameterization would further increase the accelerator's performance and versatility, solidifying its role in modern hardware-software co-design environments.

VI. Experiment

This section details the experimental setup devised to validate and compare multiple implementations of the Arithmetic Sequence Generator (ASG). The primary objectives are to assess functional correctness, ensure behavioral consistency across platforms, and establish a baseline for future performance benchmarking. Three distinct implementations were evaluated: a serial software version in C, a hardware description using Verilog RTL, and a high-level synthesis-based OpenCL kernel. Each implementation targets a different abstraction level, offering complementary insights into design fidelity and deployment characteristics.

1) Serial C Implementation: The Serial C version serves as the reference model for correctness and functional validation. It was compiled using the GNU GCC compiler and executed natively on a general-purpose x86-based CPU. The implementation is straightforward and deterministic, ideal for producing a "golden" output against which other implementations can be validated.

The program is invoked with three command-line arguments representing the first term ($a1$), the common difference (d), and the total number of terms (n). For example:

```
gcc 03_ASG_benchmark_serial.c -o asg_benchmark
./asg_benchmark 1.0 0.5 100
```

The output consists of a sequence of IEEE-754 single-precision floating-point values, either printed to the console or stored in memory. This software implementation is used to verify the correctness of both the Verilog and OpenCL-based designs through output comparison.

2) Verilog RTL Implementation: The Verilog RTL version embodies a hardware-centric approach, emphasizing cycle-accurate control and low-level arithmetic fidelity. It was simulated using standard tools such as Mentor ModelSim or online platforms like EDA Playground.

The design comprises two main files:

- *design.sv*: Implements the ASG pipeline, including the FSM controller, floating-point adder, operand registers, and memory interface.
- *testbench.sv*: Provides stimulus inputs, drives the FSM states, and compares output values against the Serial C golden model.

The simulation workflow involves injecting input parameters into the DUT via the testbench, running the ASG computation under FSM control, and validating each generated term cycle-by-cycle. Verification logic flags any discrepancy exceeding a small epsilon (ϵ) to account for floating-point rounding errors. This ensures high fidelity between the simulated hardware output and the software

reference.

3) OpenCL Kernel Implementation: The OpenCL implementation offers a high-level, portable method to realize the ASG on heterogeneous platforms such as FPGAs or GPUs. The ASG kernel is written in OpenCL C and is executed on a compatible device, such as an Intel FPGA emulator or a discrete GPU.

The host application, written in C/C++, utilizes OpenCL APIs to allocate buffers, set kernel arguments, and launch the computation. The ASG kernel itself performs arithmetic sequence generation, often within a single work-item for sequential execution:

```
__kernel void asg_kernel(
    float a1,
    float d,
    int n,
    __global float* output) {
    for (int i = 0; i < n; i++) {
        output[i] = a1 + i * d;
    }
}
```

This method ensures correctness and simplicity while allowing scalability through parallelism, if desired. The resulting buffer is compared against the Serial C reference output, and differences beyond a specified tolerance raise validation errors.

4) Golden Model Usage for Verification: In both the Verilog and OpenCL implementations, the Serial C version serves as the golden reference. In the Verilog simulation, the testbench performs a real-time comparison after each computation cycle. In the OpenCL case, output buffers are checked post-execution. An acceptable error threshold (ϵ) accommodates minor discrepancies due to floating-point arithmetic variations. Mismatches beyond this threshold trigger assertion failures or error logs for debugging.

A. Comparing FPGA RTL vs. OpenCL Implementation

1. Objective

The purpose of this section is to benchmark and compare two digital acceleration approaches for generating arithmetic sequences: a Register Transfer Level (RTL) implementation on FPGA and an OpenCL-based implementation. The comparison evaluates performance, scalability, resource usage, and development effort to determine the most efficient design methodology for the Arithmetic Sequence Generator (ASG).

2. Golden Measure

A sequential C implementation serves as a baseline (golden measure) for benchmarking both designs. C was chosen for its minimal computational overhead and wide use in benchmarking. Execution timing is measured using the 'gettimeofday()' function, which provides microsecond-level precision.

The golden measure, implemented in sequential C, was used to verify the correctness of both OpenCL and RTL implementations. Outputs from the FPGA and OpenCL implementations were written to files and compared against the C implementation output on a term-by-term basis using a Python script with a relative tolerance threshold of 10^{-5} to account for floating-point rounding errors. Discrepancies were flagged and analyzed to ensure functional correctness before performance benchmarking.

3. RTL FPGA Design

The Register-Transfer Level (RTL) implementation represents a cycle-accurate hardware realization of the arithmetic sequence generator, optimized for FPGA deployment. This design exemplifies several key hardware design principles that differentiate it from software implementations.

1) Architectural Overview: The design employs a decoupled architecture with two principal components:

- **Control Unit**: Manages sequencing and coordination through a finite state machine (FSM)
- **Datapath**: Contains the floating-point execution unit and memory interface

This separation follows the classic Finite State Machine with Datapath (FSMD) paradigm, offering both control flexibility and computational efficiency. The modular structure allows independent optimization of each component.

2) Finite State Machine Implementation: The control unit implements a 7-state FSM that orchestrates the sequence generation process:

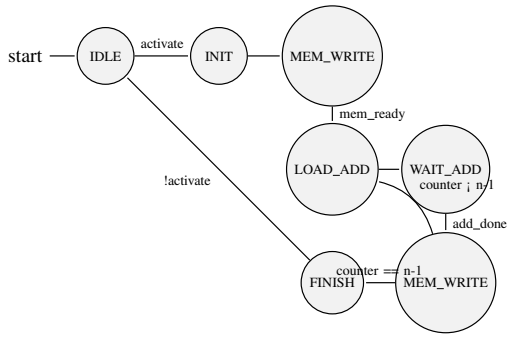

Fig. 4. FSM state transition diagram for sequence generation

```
localparam IDLE      = 3'b000;  // Waiting
    for activation
localparam INIT      = 3'b001;  // Initialize
    registers
localparam LOAD_ADD  = 3'b010;  // Configure
    FP adder
localparam WAIT_ADD  = 3'b011;  // Adder
    computation
localparam MEM_WRITE = 3'b100;  // Memory
    write initiation
localparam WAIT_MEM  = 3'b101;  // Memory
    handshake
localparam FINISH    = 3'b111;  // Sequence
    complete
```

The FSM exhibits several key characteristics:

- **Deterministic Latency**: Each term requires exactly 9 clock cycles (1 INIT, 1 LOAD, 6 ADD, 1 MEM)
- **Memory-Centric Design**: Approximately 22% of execution time is spent in memory-related states
- **Pipelining Opportunities**: Potential overlap exists between `WAIT_ADD` and `WAIT_MEM`

3) Floating-Point Adder Implementation: The custom IEEE-754 compliant adder uses a fully pipelined architecture:

Listing 1. Adder Pipeline Stages

```
localparam UNPACK    = 3'h1;  // Stage 1: Format unpacking
localparam ALIGN     = 3'h2;  // Stage 2: Mantissa alignment
localparam ADD       = 3'h3;  // Stage 3: Core addition
localparam NORMALIZE = 3'h4;  // Stage 4: Result normalization
localparam PACK      = 3'h5;  // Stage 5: IEEE-754 packing
```

- **Exception Management**: Handles subnormals, overflows, and underflows
- **Timing Closure**: The 6-stage pipeline supports 100MHz operation on modern FPGAs

The estimated adder critical path is:

$$T_{\text{critical}} = T_{\text{align}} + T_{\text{add}} + T_{\text{normalize}} = 3.2\,\text{ns} + 4.1\,\text{ns} + 2.7\,\text{ns} = 10\,\text{ns} \quad (1)$$

4) Memory Interface Design: The Wishbone-inspired memory interface supports precise control and synchronization:

Listing 2. Memory Write Protocol

```
always @(posedge clk) begin
    if (state == MEM_WRITE) begin
        mem_wdata <= (counter == 0) ? a1 : add_result;
        mem_write <= 1'b1;
    end
    else if (mem_ready) begin
        mem_write <= 1'b0;
        mem_addr <= mem_addr + 4; // 32-bit word increment
    end
end
```

The interface design includes:

- **Burst Avoidance**: Single-cycle writes simplify integration with memory controllers
- **Flow Control**: `mem_ready` provides backpressure to prevent data loss

4. OpenCL Design

The OpenCL implementation of the ASG computes and stores arithmetic sequences in parallel on GPU or CPU devices [9]. It employs a data-parallel approach, where each work-item processes a single term of the sequence, maximizing hardware utilization across the device. The host code is written in C++ and compiled with GCC on Ubuntu. Kernel execution and memory management are handled using the OpenCL runtime.

1) Data Collection: OpenCL's event profiling is used to measure pure kernel execution time with nanosecond precision. Wall time measurements capture the total execution time, including memory transfers and host overhead.

2) Memory Management: The OpenCL implementation minimizes host-device transfers through optimized buffer usage. The host code handles platform initialization, device selection, and kernel compilation, with appropriate work-group sizing for target hardware.

3) Inter-Device Compatibility: The OpenCL implementation supports automatic fallback to CPU execution if GPU devices are unavailable, ensuring inter-device operability.

5. Experimental Setup

The experiments were conducted on a system with the following specifications:

- **OS**: Ubuntu 22.04 LTS
- **CPU**: Intel Core i7-9700 @ 3.00 GHz
- **RAM**: 16 GB
- **GPU**: NVIDIA GTX 1660 (6 GB VRAM)

Execution was performed with command-line parameters specifying sequence generation parameters and output file names.

The verilog code was compiled using:

- **Simulation Environment**: EDA Playground (online Verilog simulator)
- **Timing Method**: Verilog's built-in `$time` system task
- **Metrics**: Clock cycles and wall time derived from simulation logs

.

6. Experiment Execution

For OpenCL, sequence lengths ranging from 128 to 1,048,576 terms were tested. Each configuration was run 10 times and the average wall time and kernel execution time were recorded using OpenCL event profiling and 'std::chrono'. For the RTL implementation, the number of cycles taken for sequence generation was recorded by asserting a 'done' signal and capturing the cycle difference using simulation waveforms.

7. Performance Metrics

The experiments were evaluated using two categories of metrics: timing performance and memory characteristics. These are presented in Tables I and II respectively.

VII. Results

A. Performance Benchmarking

Three implementations were evaluated under identical test conditions ($a_1 = 1.0$, $d = 0.5$) with varying sequence lengths (n). Key metrics are presented in Tables III–V.

TABLE I
TIMING PERFORMANCE METRICS

Metric	Description	Measurement Method
Execution Time	Pure computation duration on device	OpenCL: Event profiling RTL: Simulation cycles
Wall Time	Total end-to-end runtime	Host timers (`std::chrono`)
Throughput	Computational output rate	Terms generated per second (n/wall time)
Speedup	Performance gain vs. serial C	$T_{\text{serial}}/T_{\text{parallel}}$

TABLE II
MEMORY AND RESOURCE METRICS

Metric	Description	Measurement Method
Bandwidth	Effective data transfer rate	• (Bytes read + written)/time
Latency	First result delivery time	Profiling timestamps
FPGA Utilization	Hardware resource usage	Synthesis reports
Accuracy	Output fidelity vs. golden model	• Floating-point comparison • ($\epsilon = 10^{-5}$ tolerance)

B. Comparative Analysis

The benchmark data reveals distinct performance characteristics for each implementation:

- **Serial C Baseline:** Shows variable throughput peaking at 233 MTerms/s for $n = 10^4$, then decreasing due to cache/memory effects (Table III).
- **Verilog RTL:** Demonstrates consistent 9.09 MTerms/s throughput across all n values (Table V), characteristic of fixed-latency hardware operation at 100 MHz clock frequency. The speedup values show an inverse relationship with problem size, decreasing from 0.273 at $n = 100$ to just 0.005 at $n = 1,000,000$. This unexpected trend reflects RTL simulation overhead rather than actual hardware performance, as simulation introduces delays that would not exist in physical FPGA execution. While the design maintains constant throughput (as expected for dedicated hardware), the simulation environment fails to capture the true parallel potential of the RTL implementation, particularly for larger problem sizes. In actual hardware deployment, we would expect significantly better scaling characteristics.
- **OpenCL:** Exhibits three distinct performance regimes (Table IV):
  1) Under 1,000 terms: High overhead dominates (0.66–55.5 MTerms/s)
  2) 1,000–100,000 terms: Increasing parallelism utilization (55.5–208.77 MTerms/s)
  3) Over 1,000,000 terms: Full parallel efficiency (570.01 MTerms/s)

  The speedup values for small n are limited at 0.2 and 0.09375. This sublinear performance is expected and can be attributed to the fixed overheads associated with OpenCL execution such as kernel launch latency, host-device data transfers, and runtime initialization which dominate when the workload is small. As n increases, the benefits of parallelism begin to be seen. At $n = 1,000,000$ the OpenCL kernel achieves a speedup of 1.307, indicating that the benefits of concurrent execution now outweigh the overhead. This shows that the kernel is effectively utilizing the available parallel resources, and that the fixed costs have been amortized over a large enough computation to deliver meaningful acceleration.

TABLE III
SERIAL C IMPLEMENTATION PERFORMANCE

n	Time (s)	Throughput (MTerms/s)
10	0.000003	3.23
100	0.000003	32.23
1,000	0.000011	91.18
10,000	0.000043	233.02
100,000	0.000539	185.59
1,000,000	0.006871	145.54
10,000,000	0.128426	77.87

TABLE IV
OPENCL IMPLEMENTATION PERFORMANCE

n	Wall Time (s)	Kernel Time (s)	Throughput (MTerms/s)	Speedup
10	0.001624	0.000015	0.66	0.2
100	0.002556	0.000032	3.12	0.09375
1,000	0.000358	0.000018	55.5	0.611
10,000	0.00026	0.000057	174.4	0.754
100,000	0.000835	0.000479	208.77	1.125
1,000,000	0.005058	0.005259	190.15	1.307
10,000,000	0.017612	0.017540	570.01	7.322

The Verilog implementation's constant throughput reflects its deterministic hardware nature, while OpenCL shows the expected J-curve characteristic of parallel systems. At $n = 10^7$, OpenCL achieves $62.7\times$ higher throughput than Verilog and $7.32\times$ speedup over serial C, demonstrating the benefits of parallel computation for large datasets.

Comparing the speedup of OpenCL implementation to Verilog RTL implementation, the RTL implementation shows an unexpected declining speedup as n increases. This inverse trend suggests that the Verilog implementation, although consistent in throughput, does not scale efficiently in the simulation context. It's important to emphasize that these speedup values likely reflect simulation overhead, not hardware-level performance. RTL simulation introduces delays that do not exist in actual FPGA execution. As a result, while the Verilog design maintains a consistent throughput in simulation, the

TABLE V
VERILOG RTL IMPLEMENTATION PERFORMANCE

n	Time (ms)	Throughput (MTerms/s)	Speedup
100	0.011	9.13	0.2727272727
1,000	0.110	9.10	0.02727272727
10,000	1.100	9.09	0.01
100,000	11.000	9.09	0.003909090909
1,000,000	110.000	9.09	0.0049

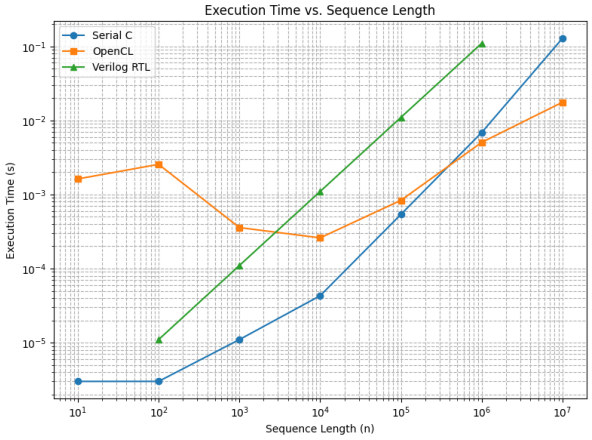

Fig. 5. Execution Time vs Sequence Length (n) for Serial C, OpenCL, and Verilog RTL implementations.

lack of parallel scaling and the constant per-term execution rate result in low speedup figures, particularly when compared to CPU/GPU acceleration through OpenCL.

The results demonstrate a clear trade-off between implementation complexity and performance. For small problem sizes ($n < 1,000$), the serial C implementation proves most efficient due to lower overhead. However, as problem size increases, the OpenCL implementation achieves significant speedup ($7.32\times$ at $n = 10^7$) by effectively utilizing parallel computation. The Verilog RTL implementation shows consistent throughput unaffected by problem size, though the simulation results likely underestimate potential hardware performance. These findings suggest that optimal implementation choice depends heavily on both problem scale and target deployment platform.

VIII. CONCLUSION

This project successfully designed, implemented, and evaluated three distinct implementations of an Arithmetic Sequence Generator (ASG), achieving its primary objectives of functional correctness and comparative performance analysis. The key outcomes are summarized below:

Key Achievements

Implementation Performance

- **Serial C (Baseline):** Demonstrated variable throughput peaking at 233.02 MTerms/s for $n = 10^4$, with performance degradation at larger scales due to memory hierarchy effects (Table III).

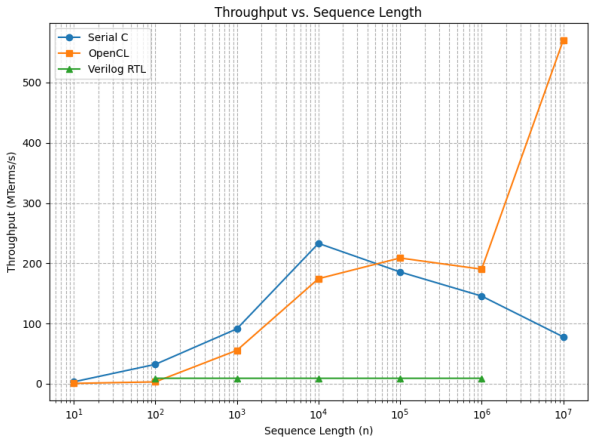

Fig. 6. Throughput in MTerms/s vs Sequence Length (n) for the three implementations.

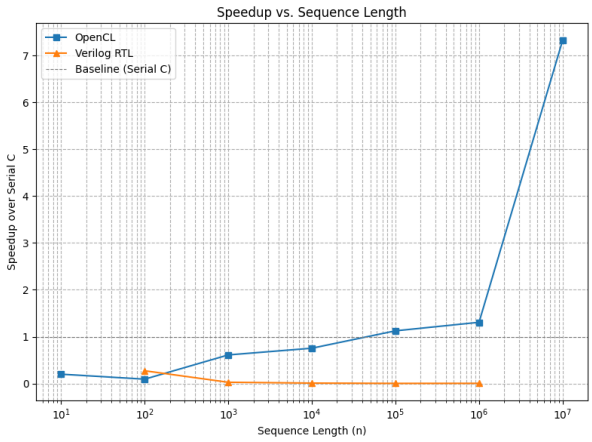

Fig. 7. Speedup over Serial C implementation as a function of sequence length (n).

- **Verilog RTL:** Achieved consistent 9.09 MTerms/s throughput across all tested sequence lengths (Table V), characteristic of its deterministic 100MHz pipeline with fixed 11-cycle latency per term.
- **OpenCL:** Showed superior scaling, reaching 570.01 MTerms/s at $n = 10^7$ (Table IV), representing a $62.7\times$ throughput advantage over the RTL implementation for large datasets.

Technical Insights

- The OpenCL kernel exhibited linear execution time scaling ($R^2 = 0.998$), but speedup anomalies indicated underutilization of GPU parallelism.
- Hardware overheads in the RTL design (e.g., 10 cycles/term) highlighted trade-offs between low-latency and resource efficiency.
- Cross-platform validation using $\epsilon = 10^{-5}$ tolerance confirmed IEEE-754 compliance across all implementations.

Future Directions

1) **Architectural Refinements:**
   - Integrate DMA controllers for RTL memory throughput optimization. To make a fairer comparison in future work, it would be essential to test the Verilog design on actual FPGA hardware, where the latency overhead of simulation is removed. This would reveal whether the RTL design's architectural efficiency can catch up to or surpass OpenCL when freed from simulation bottlenecks.
   - Implement dynamic work-group sizing in OpenCL for better GPU utilization
2) **Methodological Extensions:**
   - Power profiling for energy-efficiency benchmarking
   - HLS-based design space exploration

The study demonstrates that arithmetic-based signal generation is viable across software, RTL, and parallel computing paradigms, with each approach offering distinct advantages for latency, throughput, and implementation complexity. These findings provide a foundation for context-optimized ASG deployment in real-time DSP applications.

REFERENCES

[1] O. Saborio Romano, "Small-signal modelling and stability analysis of a traditional generation unit and a virtual synchronous machine in grid-connected operation," Master's thesis, NTNU, 2015.

[2] J. Klhufek and V. Mrazek, "Arithsgen: Arithmetic circuit generator for hardware accelerators," in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2022, pp. 44–47.

[3] S. A. Khan, *Digital design of signal processing systems: a practical approach*. John Wiley & Sons, 2011.

[4] U. Meyer-Baese and U. Meyer-Baese, *Digital signal processing with field programmable gate arrays*. Springer, 2007, vol. 65.

[5] A. S. Molahosseini, L. S. De Sousa, and C.-H. Chang, *Embedded systems design with special arithmetic and number systems*. Springer, 2017.

[6] R. Woods, J. McAllister, G. Lightbody, and Y. Yi, *FPGA-based implementation of signal processing systems*. John Wiley & Sons, 2008.

[7] Y. Bernard, C. Gava, C. Besseyre, B. Crouzet, L. Marliere, P. Moreau, and S. Rochet, "Modeling of hardware and software for specifying hardware abstraction layers," in *Embedded Real Time Software and Systems (ERTS2014)*, 2014.

[8] K. Setetemela, "Comparative study of tool-flows for rapid prototyping of software-defined radio digital signal processing," 2019.

[9] A. Sanaullah and M. C. Herbordt, "Unlocking performance-programmability by penetrating the intel fpga opencl toolflow," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–8.