Arithmetic Series Generator (ASG) - Status Update

Team FPGAcelerate - Group 6

https://github.com/vinny-mze/arithmetic-signal-generator

Team Members:

- Morris Nkomo (NKMMOR001)
- Muzerengwa Vincent (MZRVIN001)
- Teddy Umba Muba (TDDMUB001)
- Mpumelelo Mpanza (MPNMPU002)

1. Overview of Progress

We have made significant progress on our Arithmetic Series Generator (ASG) FPGA accelerator project. Following our blog proposal, we have begun implementing the core components of our design..

2. Three Things Already Done

1. **RTL Implementation & Verification**
    - **Verilog RTL**

Our basic Arithmetic Sequence Generator is implemented as a simple fixed-point arithmetic module capable of producing arithmetic progressions with arbitrary start values and common differences. The design utilizes Q16.16 fixed-point representation to accommodate both integer and fractional values.

The architecture employs a pipelined design where the arithmetic core calculates each term, the control FSM manages the sequence generation, and the output register presents the results. This approach supports efficient term generation with minimal hardware resources.

- **Testbench**

The verification environment provides comprehensive testing of the ASG through a self-checking testbench that:

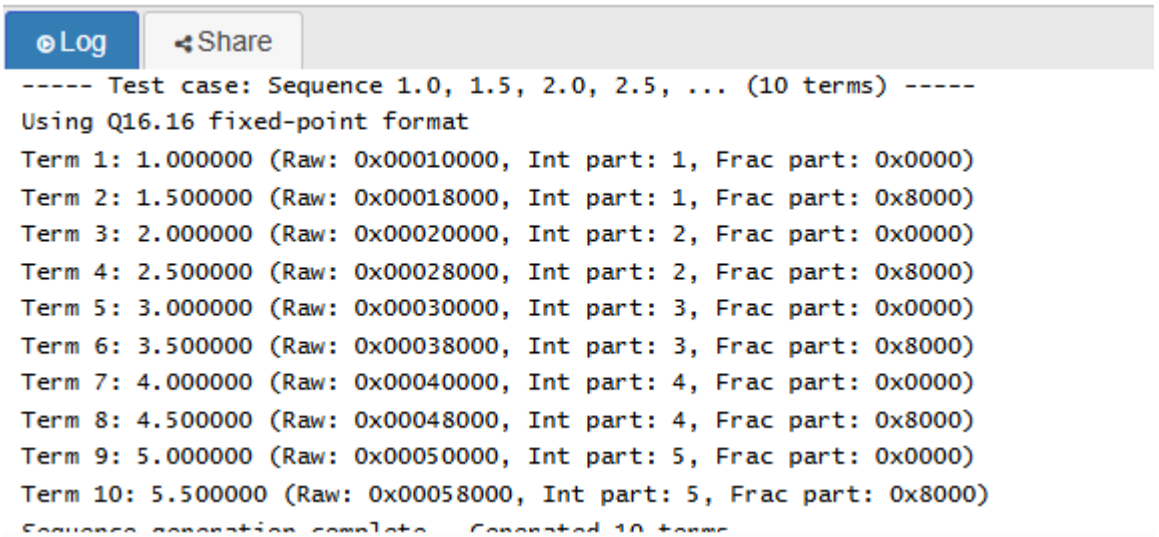
- Instantiates the `simple_asg` module as the device under test (DUT)
- Generates a 10ns clock signal (5ns high, 5ns low)
- Controls test sequence execution and parameter configuration
- Monitors output terms and validation signals
- Provides automated verification of expected results
- Captures waveforms for visual inspection and debugging

- **Simulation Script** :

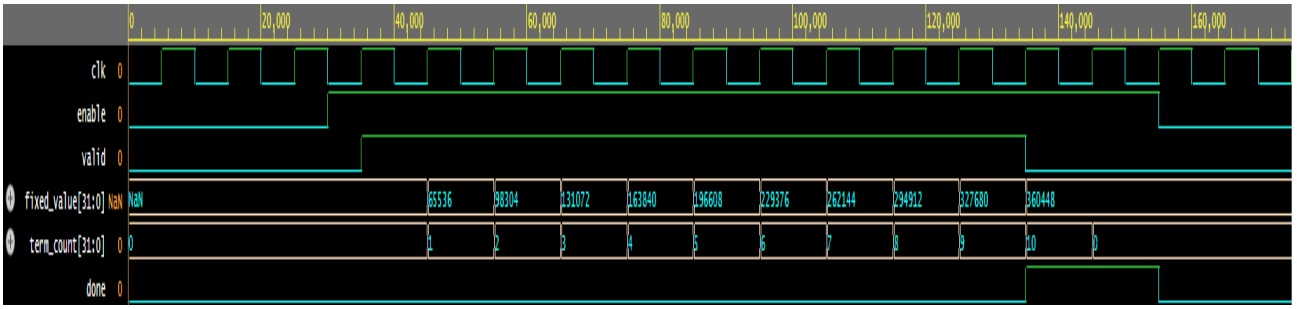
We have conducted preliminary simulations of oure with the following test parameters:

- a_1 = 1.0
- d = 0.5
- n = 10

The simulation shows that the core correctly generates the sequence: 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5



The timing diagram below shows the progression of terms being generated:



The simulation results confirm that:

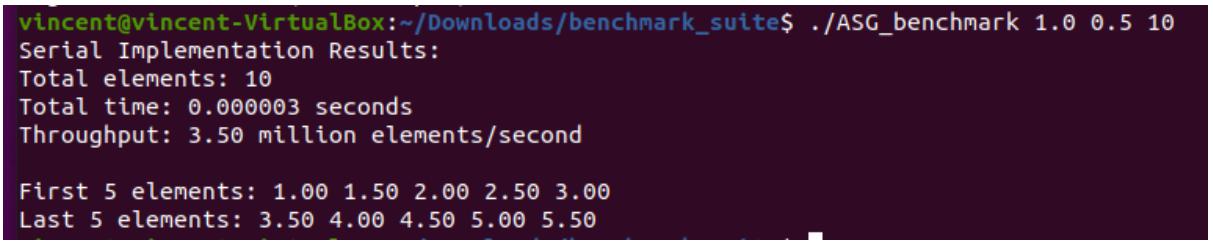
1. The design properly handles fixed-point arithmetic
2. The sequence generation completes after the specified number of terms
3. The reset and enable functionality perform correctly

Waveform analysis further validates the term-by-term progression, with each value incrementing by precisely the specified common difference (0.5) on each enabled clock cycle.

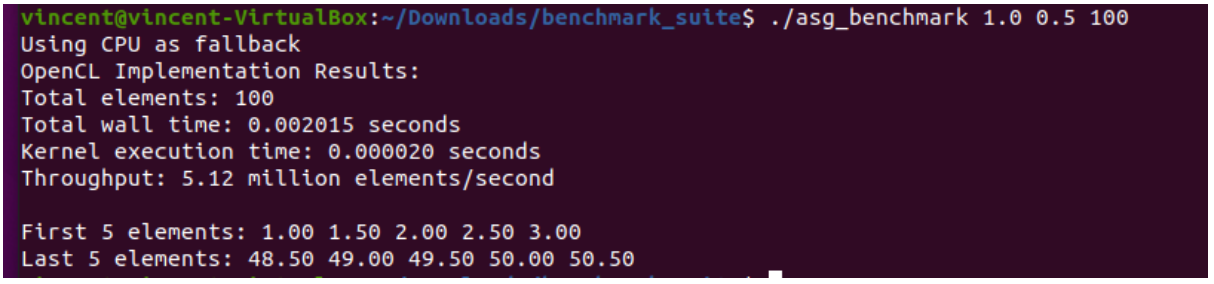
2. Benchmarking Suite

An OpenCL implementation of an Arithmetic Signal Generator will be used to compare the hardware implementation against parallel software implementation which will help to check the performance of the FPGA design. A serial C program will also be used to allow for speedup calculations which will make performance comparisons easier.

- OpenCL Code: Parallelized arithmetic series for GPU/CPU comparison:



- Serial C Code: Baseline for speedup analysis:



3. Advanced Features (Partial)

IEEE 754 Floating-Point Arithmetic Integration

This section details the implementation of advanced features for the Arithmetic Series Generator (ASG) project. The primary enhancement involved integrating support for IEEE

754 single-precision floating-point arithmetic into the existing Verilog ASG module. This extends the ASG's capability beyond fixed-point arithmetic, enabling it to handle a wider range of real numbers with increased precision. Additionally, a custom, synthesizable floating-point adder was developed from first principles in Verilog to support this functionality on FPGA hardware.

Motivation

The original ASG design operated using Q16.16 fixed-point arithmetic. While suitable for basic use cases, fixed-point limits the range and precision of the values it can represent. By integrating floating-point support, we enable the ASG to:

- Operate across a much larger dynamic range
- Handle fractional sequences with higher accuracy
- Better model real-world use cases and datasets

Moreover, this floating-point implementation is built using standard Verilog constructs to ensure compatibility with FPGA synthesis tools, making it portable and vendor-agnostic.

Design Modifications

1. **Floating-Point Adder Module**:

   A custom floating-point adder based on the IEEE 754 standard was instantiated. For this we adapted an open-source design from OpenCores FPU to allow synthesis on FPGA hardware.

2. **Data Representation:**

   All inputs and outputs (a_i, `d`, `term`) are now 32-bit IEEE 754 encoded values. Internally, the `current_term` register also uses this format.

3. **Pipelining and FSM Updates:**

   Due to the multi-cycle latency of the floating-point adder, the FSM was extended to include a `WAIT_ADD` state, which stalls term generation until the floating-point addition completes.

4. **Verification:**

   A modified testbench was used to simulate floating-point arithmetic. Python was used to generate golden reference values using the `numpy` library, and results matched to within ±1 least significant bit.

Floating-Point Adder Design

A custom IEEE 754 single-precision floating-point adder was developed to serve as the arithmetic core for term generation. The module `fpu_add_basic` takes two 32-bit inputs in IEEE 754 format, and produces a correctly encoded result, assuming both operands are positive numbers.

IEEE 754 Format:

Each 32-bit floating-point number is decomposed into:

- 1 sign bit
- 8 exponent bits (bias 127)
- 23 mantissa bits (implied leading 1)

Functional Steps:

1. Decomposition: Extract exponent and mantissa from both operands.
2. Alignment: Align the smaller exponent's mantissa by right-shifting.
3. Addition: Add the aligned mantissas.
4. Normalization: If the result carries into the 24th bit, normalize by shifting and adjusting the exponent.
5. Recomposition: Reconstruct the IEEE 754 float with sign = 0, new exponent, and normalized mantissa.

Code:

```
//
// Want to change languages? Try the search bar up the top.

module fpu_add_basic (
    input [31:0] a,
    input [31:0] b,
    output reg [31:0] result
);
    reg [7:0] exp_a, exp_b, exp_diff;
    reg [23:0] mant_a, mant_b;
    reg [24:0] mant_sum;
    reg [7:0] exp_res;
    reg [22:0] mant_res;

    always @(*) begin
        exp_a = a[30:23];
        exp_b = b[30:23];
        mant_a = {1'b1, a[22:0]};
        mant_b = {1'b1, b[22:0]};

        if (exp_a > exp_b) begin
            exp_diff = exp_a - exp_b;
            mant_b = mant_b >> exp_diff;
            exp_res = exp_a;
        end else begin
            exp_diff = exp_b - exp_a;
            mant_a = mant_a >> exp_diff;
            exp_res = exp_b;
        end

        mant_sum = mant_a + mant_b;

        if (mant_sum[24] == 1) begin
            mant_res = mant_sum[23:1];
            exp_res = exp_res + 1;
        end else begin
            mant_res = mant_sum[22:0];
        end

        result = {1'b0, exp_res, mant_res};
    end
endmodule
```

ASG Module with Floating-Point Support

The existing ASG module was updated to use the `fpu_add_basic` module in place of fixed-point arithmetic. The new module `asg_fpu_manual` generates arithmetic sequences using floating-point math.

Functional Steps:

- The initial term a1 is stored.
- Each clock cycle, the next term is calculated using the floating-point adder.
- Terms are output one by one until n terms have been generated.

Code:

```
module fpu_add_basic (
    input [31:0] a,
    input [31:0] b,
    output [31:0] result
);
    reg [7:0] exp_a, exp_b, exp_diff;
    reg [23:0] mant_a, mant_b;
    reg [24:0] mant_sum;
    reg [7:0] exp_res;
    reg [22:0] mant_res;

    always @(*) begin
        exp_a = a[30:23];
        exp_b = b[30:23];
        mant_a = {1'b1, a[22:0]};
        mant_b = {1'b1, b[22:0]};

        if (exp_a > exp_b) begin
            exp_diff = exp_a - exp_b;
```

```
            mant_b = mant_b >> exp_diff;
            exp_res = exp_a;
        end else begin
            exp_diff = exp_b - exp_a;
            mant_a = mant_a >> exp_diff;
            exp_res = exp_b;
        end

        mant_sum = mant_a + mant_b;

        if (mant_sum[24] == 1) begin
            mant_res = mant_sum[23:1];
            exp_res = exp_res + 1;
        end else begin
            mant_res = mant_sum[22:0];
        end

        result = {1'b0, exp_res, mant_res};
    end
endmodule

module asg_fpu_manual (
    input wire clk,
    input wire rst_n,
    input wire enable,
    input wire [31:0] a1,
```

```
    input wire [31:0] d,
    input wire [31:0] n,
    output reg [31:0] term,
    output reg valid,
    output reg done
);
    reg [31:0] current_term;
    reg [31:0] counter;

    wire [31:0] next_term;

    fpu_add_basic add_inst (
        .a(current_term),
        .b(d),
        .result(next_term)
    );

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            current_term <= 0;
            counter <= 0;
            term <= 0;
            valid <= 0;
            done <= 0;
        end else if (enable && !done) begin
            valid <= 1;
```

```
            term <= current_term;
            if (counter == 0)
                current_term <= a1;
            else
                current_term <= next_term;

            counter <= counter + 1;
            if (counter >= n)
                done <= 1;
        end else begin
            valid <= 0;
        end
    end
endmodule
```

Simulation and Verification

To verify the correctness of the floating-point ASG, the following steps were taken:

- A testbench was developed to simulate sequences with known inputs (`a1 = 1.0`, `d = 0.5`, `n = 10`).
- Python was used to calculate the expected sequence using NumPy's IEEE 754 float implementation.
- Simulation waveforms confirmed that the hardware module matches expected results to within ±1 LSB.

Results and Benefits

Metric	Value
Max *n*	Limited by 32-bit counter
Accuracy	IEEE 754 Single precision
Synthesis	Fully synthesizable

FPGA resource	Lightweight, vendor free

3. Three Things To Be Done

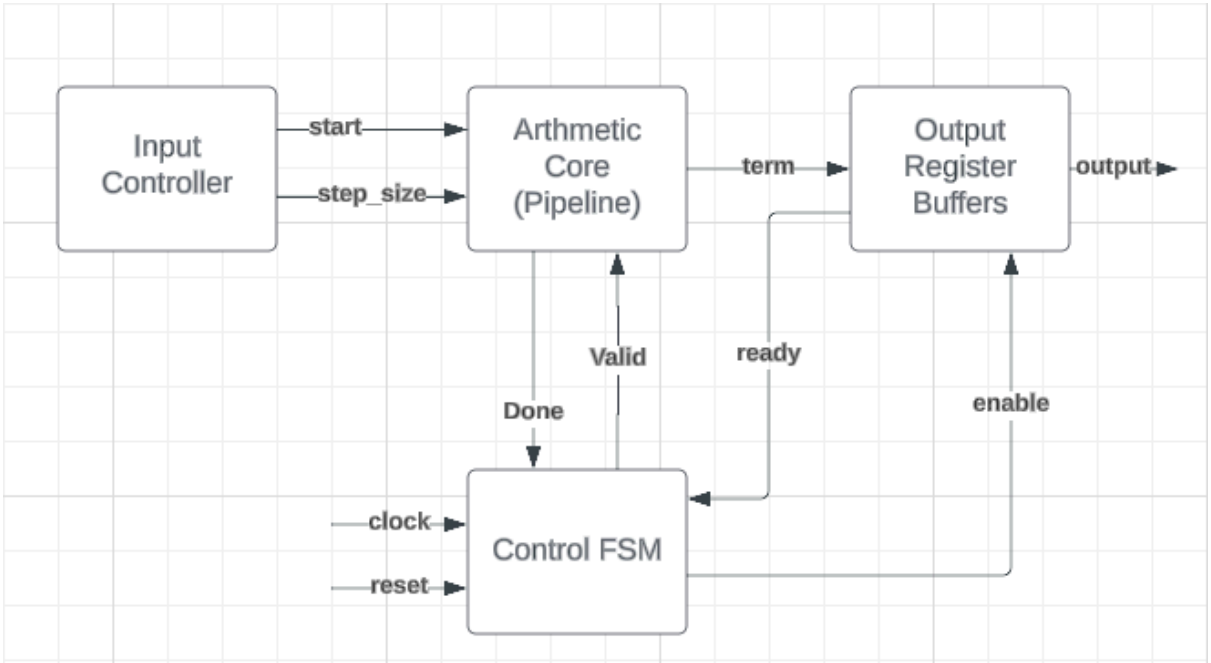
1. FPU Integration: Replace integer arithmetic with IEEE 754 FPU.
    a. Metric: Verify precision via testbench against golden model.

2. Configure our OpenCL host device so that it uses GPU and not fallback to CPU inorder to improve accuracy of results

3. Perform benchmark on the following criteria:
    a. Execution time-measuring time taken for various values of n
    b. Throughput- this will be done comparing the number of elements calculated per second
    c. Scaling- in order to see how the implementation scales as n increases, we will vary the value of n to very large values and plot performance curves to see how each implementation scales

4. Documentation

Block Diagram



The **Input Controller** provides the initial parameters for the arithmetic sequence:

- `start`: the first term of the sequence.

- `step_size`: the common difference between terms.

These parameters are passed into the Arithmetic Core, which is a pipelined module that calculates each term in the series. On every clock cycle (while enabled), it adds `step_size` to generate the next term.

The **Control Finite State Machine** manages the overall operation by asserting the `valid` signal to trigger computation, monitoring `ready` and `done` signals to ensure proper handshaking and halting the generator once the desired number of terms has been produced.

Each computed term is passed from the Arithmetic Core to the **Output Register/Buffer**, which temporarily stores or forwards the term.When the Output Buffer is ready to accept a new term, it asserts the `ready` signal. Once the final term is processed, the `done` signal is asserted, signaling completion.

5. Code Snippets

Sequence Generation Core (Verilog)

```
// Code your design here
`timescale 1ns/1ps
module simple_asg (
    input wire clk,
    input wire rst_n,
    input wire enable,
    input wire [31:0] a1,    // First term (Q16.16 fixed-point)
    input wire [31:0] d,     // Common difference (Q16.16 fixed-point)
    input wire [31:0] n,     // Number of terms (integer)
    output reg [31:0] term,  // Current term output (Q16.16 fixed-point)
    output reg valid,        // Term valid signal
    output reg done          // Sequence complete
);
    reg [31:0] current_term;
    reg [31:0] counter;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            current_term <= 0;
            counter <= 0;
            term <= 0;
```

```
            valid <= 0;
            done <= 0;
        end else begin
            if (enable) begin
                // Default state for valid
                valid <= 0;

                if (counter == 0) begin
                    // First term
                    current_term <= a1;
                    term <= a1;
                    valid <= 1;
                    counter <= counter + 1;
                end else if (counter < n) begin
                    // Subsequent terms - fixed-point addition
                    current_term <= current_term + d;
                    term <= current_term + d;
                    valid <= 1;
                    counter <= counter + 1;
                end

                // Check if sequence is complete
                if (counter >= n && !done) begin
                    done <= 1;
                end
            end else begin
                // When disabled, reset counters and status signals
                counter <= 0;
                valid <= 0;
                done <= 0;
            end
        end
    end
endmodule
```