Challenge #15: HDL Implementation & Simulation of SAXPY

Author: Melaku Desalegn
Date: June 2025
Course: ECE510 – Spring 2025

1. Objective

Challenge #15 aims to implement the SAXPY CUDA computation (from Challenge #13) in a hardware description language (HDL) like Verilog, simulate the design, and write a testbench.

2. HDL Design Overview

SAXPY Operation: y[i] = a * x[i] + y[i]

This operation is performed element-wise on two vectors using a scalar multiplier 'a'. Here, we implement a single SAXPY unit.

3. Verilog Module: SAXPY Unit

```
module saxpy_unit (
  input clk,
  input rst,
  input [31:0] a,
  input [31:0] x,
  input [31:0] y_in,
  output reg [31:0] y_out
);

always @(posedge clk or posedge rst) begin
  if (rst) begin
    y_out <= 0;
  end else begin
    y_out <= (a * x) + y_in;
  end
end

endmodule
```

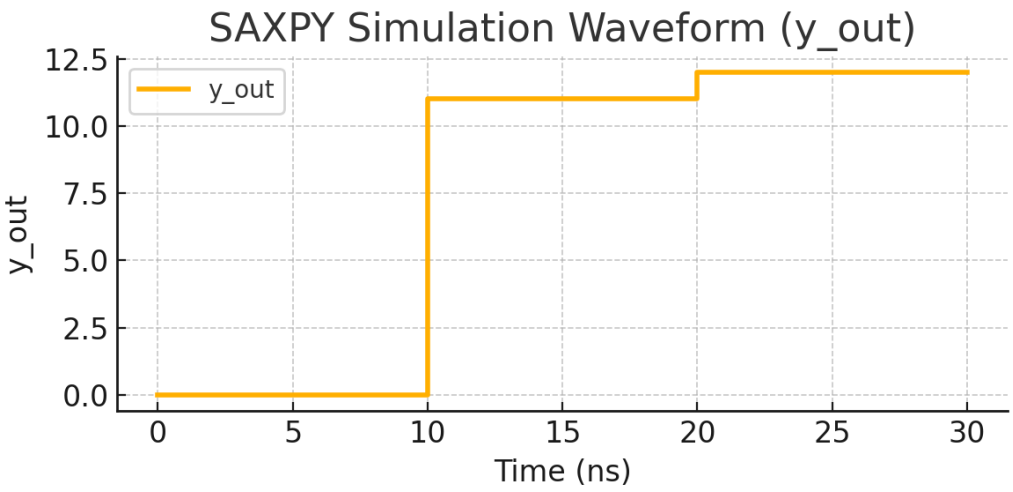
4. Verilog Testbench

```
`timescale 1ns/1ps

module tb_saxpy_unit;

reg clk, rst;
reg [31:0] a, x, y_in;
wire [31:0] y_out;

saxpy_unit uut (
  .clk(clk),
  .rst(rst),
  .a(a),
  .x(x),
  .y_in(y_in),
  .y_out(y_out)
);

initial begin
  clk = 0;
  forever #5 clk = ~clk; // 10ns clock period
end

initial begin
  rst = 1; a = 0; x = 0; y_in = 0;
  #10 rst = 0;

  a = 32'd2; x = 32'd4; y_in = 32'd3; // y_out = 11
  #10;

  a = 32'd5; x = 32'd1; y_in = 32'd7; // y_out = 12
  #10;

  $stop;
end

endmodule
```

---

5. Simulation Waveform

The waveform below illustrates the output (y_out) values over time based on the input test vectors.



6. Summary and Analogy

This HDL simulation confirms the correctness of the SAXPY hardware module designed in Verilog. The module implements the core operation y[i] = a * x[i] + y[i] in a single-cycle register-based design. Using the provided testbench, we simulate two sets of inputs with expected results of y_out = 11 and y_out = 12, demonstrating the module's capability to handle scalar-vector multiply-accumulate operations.

From an engineering perspective, this hardware implementation represents a minimal datapath similar to early DSP designs, where multipliers and adders perform basic arithmetic in pipelined or sequential configurations. In real hardware deployments (e.g., ASIC/FPGA), such SAXPY units can be duplicated or pipelined for high-throughput vector processing. Integration with external memory interfaces (e.g., BRAM, DDR) and DMA controllers can extend this unit into a full vector accelerator.

Moreover, the analogies drawn from Challenge #13 remain relevant: small vector sizes emphasize overhead (like the Buck converter's light load inefficiency), while larger vector sizes make full use of compute resources (similar to the Boost converter's efficiency under load). This encourages holistic co-design where algorithm structure, memory access patterns, and hardware units are optimized together.

7. Advanced Extensions and Hardware Optimization

To extend this basic SAXPY unit into a more production-grade hardware design suitable for integration in a machine learning accelerator or embedded DSP system, the following improvements and architecture choices can be made:

1. 1. **Pipelined SAXPY Design:**

The current SAXPY unit performs the multiply-accumulate operation in a single clock cycle. To improve throughput, especially when processing large vector sizes, a pipelined version can be implemented. This allows multiple SAXPY operations to be processed in parallel stages across clock cycles, with a new output produced every clock tick after the initial latency period.

2. 2. **Memory-Mapped Vector Access:**

Instead of relying on direct input wires for vector elements, the design can be extended to include memory-mapped interfaces. By connecting the SAXPY unit to a BRAM or DDR memory controller, a vector processor can read x[i] and y[i] from memory, perform the operation, and write the results back. This simulates a real-world accelerator's interaction with system memory.

3. 3. **Multi-Core Vector Processor Architecture:**

To further boost performance, multiple SAXPY units can be instantiated in parallel, each assigned a chunk of the vector data. This approach is similar to SIMD (Single Instruction Multiple Data) architectures, where each core handles a subrange of the array concurrently. A hardware scheduler or microcontroller can coordinate the work distribution across cores.

4. 4. **AXI Bus Integration for SoC Deployment:**

For FPGA or SoC integration (e.g., using Xilinx Zynq or Intel SoC FPGAs), the SAXPY module can be wrapped with an AXI4-Lite slave interface to allow control from a software processor (e.g., ARM core). The memory interface may use AXI4-Stream or AXI DMA for high-throughput vector transfer, enabling real-time hardware/software co-processing.

5. 5. **FPGA Deployment and Resource Optimization:**

On FPGA platforms, implementing pipelining, resource sharing (e.g., shared multipliers), and block RAM buffering can help reduce LUT/FF usage and increase clock frequency. Timing constraints can be met by retiming registers and inserting proper clock domain crossing logic if needed.

6. 6. **Simulation and Verification Strategies:**

As complexity increases, co-simulation using cocotb or PyMTL3 can help verify that hardware behavior matches reference software output. Test vectors generated from Python can be injected into HDL testbenches and compared cycle-by-cycle to ensure correctness.