Vector Dot Product Accelerator: Software vs Hardware

Shaba Altaf Shaon and Sanjoy Dev
Instructor: Dr. Aleksandar Milenkovic
Course: CPE 523 Hardware/Software Co-design

1 Abstract

General-purpose central processing units (CPUs) execute operations sequentially in a fetch-decode-decode style, leading to performance bottlenecks for compute- or memory-intensive tasks [1]. A hardware accelerator addresses this issue by enabling parallelism and pipelining, significantly speeding up critical operations while reducing CPU load [2, 3]. Therefore, this project presents the design, implementation, and performance evaluation of a pipelined hardware accelerator for computing the dot product of two vectors on the Terasic DE1-SoC FPGA platform. The accelerator is built using SystemVerilog and leverages the Avalon-MM interface to communicate with external SDRAM and the ARM Cortex-A9 processor. Our design includes a direct memory access (DMA)-enabled master interface for autonomous memory access, first-in, first-out (FIFO) buffers to decouple memory latency from computation, and a 64-bit multiply-accumulate (MAC) unit that operates on a fully pipelined architecture. To evaluate the benefits of the hardware accelerator, we implement two approaches: (i) a baseline software version using a simple C loop on the ARM CPU, and (ii) an efficient pipelined hardware version that uses DMA to streams data from SDRAM with minimal CPU intervention. Performance is measured using the Altera Avalon Interval Timer, a cycle-accurate hardware timer running at 100 MHz, to compare the execution time of software and hardware dot product implementations. Results show that the pipelined hardware accelerator achieves significant speedup over the software version. Therefore, this project demonstrates the effectiveness of custom hardware acceleration for memory-bound operations such as dot product, with applications across many engineering and computational domains.

2 Background and Introduction

In modern engineering and scientific computing, vector operations such as the dot product are fundamental building blocks used in domains including machine learning, signal processing, robotics, and control systems. While these operations are mathematically simple, they often become performance bottlenecks in large-scale applications due to their memory-bound nature, where data movement, rather than computation, limits execution speed.

2.1 Vector Dot Product

The dot product of two vectors A and B of equal length N is a scalar value obtained by multiplying corresponding elements of the vectors and summing the results. Hence, it is also known as the scalar product. Mathematically, if

$$\mathbf{A} = [a_1, a_2, \ldots, a_N], \quad \mathbf{B} = [b_1, b_2, \ldots, b_N], \tag{1}$$

then the dot product is defined as [4]

$$\mathbf{A} \cdot \mathbf{B} = \sum_{n=1}^{N} a_n b_n. \tag{2}$$

2.2 Physical Interpretation of Vector Dot Product

The dot product of two vectors measures how much one vector "acts in the direction of" another. Geometrically, it is defined as

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}||\mathbf{B}|cos(\theta), \tag{3}$$

where θ is the angle between the two vectors. If the dot product is positive, the vectors point in approximately the same direction. However, if the dot product is negative, they point in opposite directions. Moreover, if the dot product is zero, the vectors are perpendicular(i.e., orthogonal) to each other. Physically, it is often used to compute work done by a force acting along a displacement.

2.3 Applications of Vector Dot Product

Vector Dot Product is widely used in numerous engineering computations. Some of its application domains are discussed below.

***Machine learning*:** In machine learning, the dot product is used extensively in models such as linear regression, logistic regression, and neural networks. In a neural network, the output of a neuron is computed as the dot product between the input feature vector and the weight vector, followed by an activation function. This operation determines how strongly a particular set of features activates a neuron.

***Signal processing*:** In digital signal processing, the dot product is used to compute convolution, correlation, and finite impulse response filters. When filtering a signal, a sliding window of input samples is multiplied element-wise with a filter kernel and summed. This contributes in tasks such noise reduction, audio processing, and image smoothing.

***Scientific Computing*:** The dot product is a fundamental operation in many scientific computing and numerical simulations. For instance, in linear algebra, it is used for solving systems of equations, computing vector projections, and performing matrix-vector multiplications. These operations are also essential in physics simulations, structural analysis, and engineering modeling where performance and numerical stability are critical.

3 Goals and Objectives

In this project, we aim to design, implement, and evaluate a hardware accelerator for computing the dot product of two vectors using a pipelined architecture on an FPGA, and to compare its performance against a traditional software implementation on a general-purpose processor. More specifically, the objectives of our project are as follows:

- Design and implement a pipelined hardware core for computing the dot product of two vectors using SystemVerilog on the DE1-SoC FPGA platform, with DMA access to external SDRAM.
- Develop a baseline software implementation of the vector dot product using C on the ARM Cortex-A9 processor integrated in the DE1-SoC.
- Compare and evaluate the performance of the hardware and software approaches using Avalon Interval Timer to measure execution time and identify speedup achieved through acceleration.

4 Methods

To evaluate the performance benefits of hardware acceleration for dot product computation, we implemented and tested three distinct approaches on the Terasic DE1-SoC platform. They are described below:

- **Software implementation on ARM Cortex-A9:** We design a baseline software version in C and execute it on the ARM Cortex-A9 processor. This approach utilizes a simple loop to iterate over vector elements stored in SDRAM. Along the way, it performs element-wise multiplication and accumulates the result. Here, random vector data is generated and initialized in SDRAM before computation begins.
- **Pipelined hardware accelerator (with DMA, SDRAM, and FIFOs):** We design our main fully pipelined hardware accelerator in SystemVerilog that uses an Avalon-MM master interface to autonomously fetch vector data from SDRAM and process it using a 64-bit MAC unit. Dual FIFO buffers are used to decouple memory access from computation, enabling one dot product operation per clock cycle. Performance for all methods was measured using the Altera Avalon Interval Timer to evaluate execution time and calculate speedup over the software baseline.

4.1 Design of Our Main Hardware Accelerator

We now present a detailed breakdown of all the main components of our pipelined hardware accelerator design. The main hardware accelerator was implemented in SystemVerilog and designed to execute vector dot product operations efficiently using a streamed, pipelined architecture. The following are the key components that enable high-throughput performance:

***Avalon-MM Slave Interface*:** This interface allows the CPU (i.e., ARM Cortex-A9) to configure and control the accelerator through memory-mapped registers. It provides access to:

- The vector length register
- Base addresses for vectors A and B in SDRAM
- A control register to start computation and check busy status
- Result registers for reading the 64-bit dot product output (split into low and high resisters)

Address	Name	Access	Description
0	Control & Busy	R/W	Bit 0 is busy flag; writing starts the process
1	VEC_Length	R/W	Size of vectors N
2	vecA_baseAddr	R/W	Base address of Vector A in SDRAM
3	vecB_baseAddr	R/W	Base address of Vector B in SDRAM
4	RES_LO	R	Lower 32 bits of dot product result
5	RES_HI	R	Upper 32 bits of dot product result

Table 1: Register map of the vector dot product accelerator.

This interface makes the accelerator appear like a peripheral device to the CPU, allowing seamless communication over the lightweight HPS-to-FPGA bridge (lwh2f bridge).

***Avalon-MM Master Interface*:** The master interface is responsible for autonomously reading data from external SDRAM. Key features include:

- Generating alternating read requests for $\mathbf{A}[n]$ and $\mathbf{B}[n]$
- Using byte-aligned addresses via avm_address with avm_bytenable = 4'b1111
- Sending one-word burst count transactions using avm_burstcount = 1
- Handling avm_waitrequest to pause read requests if SDRAM is busy
- Receiving 32-bit data via avm_readdata and detecting its validity with avm_readdatavalid.

This DMA mechanism offloads memory access from the CPU, ensuring data is streamed into the accelerator without much CPU involvement.

***Core Computational Logic*:** In order to decouple the slow and unpredictable memory latency from fast MAC computation, our design uses:

- FIFO_A: Buffer for vector A data.
- FIFO_B: Buffer for vector B data.

Each FIFO

- Has a depth of 64 entries, each 32 bits wide

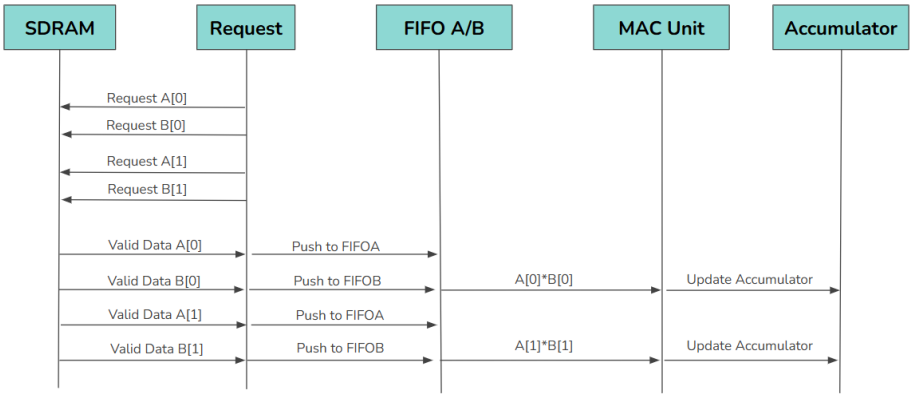

Figure 1: Pipelined Mechanism using SDRAM

- Is managed by separate read/write pointers and counters
- Accepts new data when avm_readdatavalid is high
- Feeds the MAC unit to do calculation when both FIFOs have at least one valid data word
- A toggling mechanism is implemented for both requesting data from the SDRAM and pushing the response to each of the two FIFOs. For each clock pulse, the data either request data for vector A or vector B. Same goes for the response phase.

This buffering ensures that once data begins flowing, the MAC unit can compute one result per clock cycle.
***MAC Unit*:** At the core of the design is a 64-bit multiply-accumulate unit, which performs:

$$\text{accum} \leftarrow \text{accum} + (\mathbf{A}[n] \times \mathbf{B}[n]) \tag{4}$$

It operates once per clock cycle when FIFOs are not empty and accumulates results in a 64-bit register to avoid overflow. This unit is tightly integrated with the FSM so that once data is available, computation proceeds continuously without CPU intervention.
***Finite state machine (FSM)*:** A compact 3-state FSM controls the accelerator. The states are:

- **IDLE:** Waits for start signal; resets pointers and counters.
- **RUN:** Alternates memory read requests, buffers incoming data, and performs MAC when buffers are non-empty.
- **DONE:** Signals completion and transitions back to IDLE automatically in the next clock cycle.

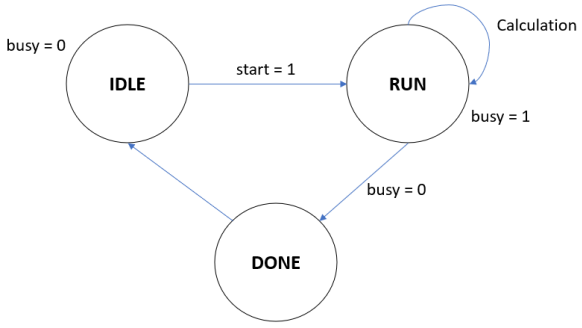

Figure 2: State Diagram

***Result Register Bank*:** After the full dot product is computed, the 64-bit result is split into two 32-bit registers. The result is accessible to the CPU via the Avalon-MM slave interface. The result is latched only when the FSM transitions from RUN to DONE state. This enables the ARM processor to read the final result in two consecutive register reads.

***Avalon Interval Timer*:** The Avalon Interval Timer is used to measure execution time in clock cycles. We then evaluate speedup compared to software version. Thus, we confirm the effectiveness of pipelined hardware accelerator for vector dot product. It is initialized, started, and read by the ARM software during both hardware and software dot product executions.

5 Cost Analysis

Metric	**Full DE1-SoC System**	**Accelerator Only**
Logic utilization (ALMs)	24,498 / 32,070 (76%)	320 / 32,070 (< 1%)
Total registers	31,288	518
Total pins	368 / 457 (81%)	151 / 457 (33%)
Total block memory bits	2,395,802 / 4,065,280 (59%)	4,096 / 4,065,280 (< 1%)
Total DSP blocks	21 / 87 (24%)	6 / 87 (7%)
Total PLLs	3 / 6 (50%)	0 / 6 (0%)
Total DLLs	1 / 4 (25%)	0 / 4 (0%)

Table 2: Resource utilization of the complete DE1-SoC computer system versus the pipelined vector dot-product accelerator.

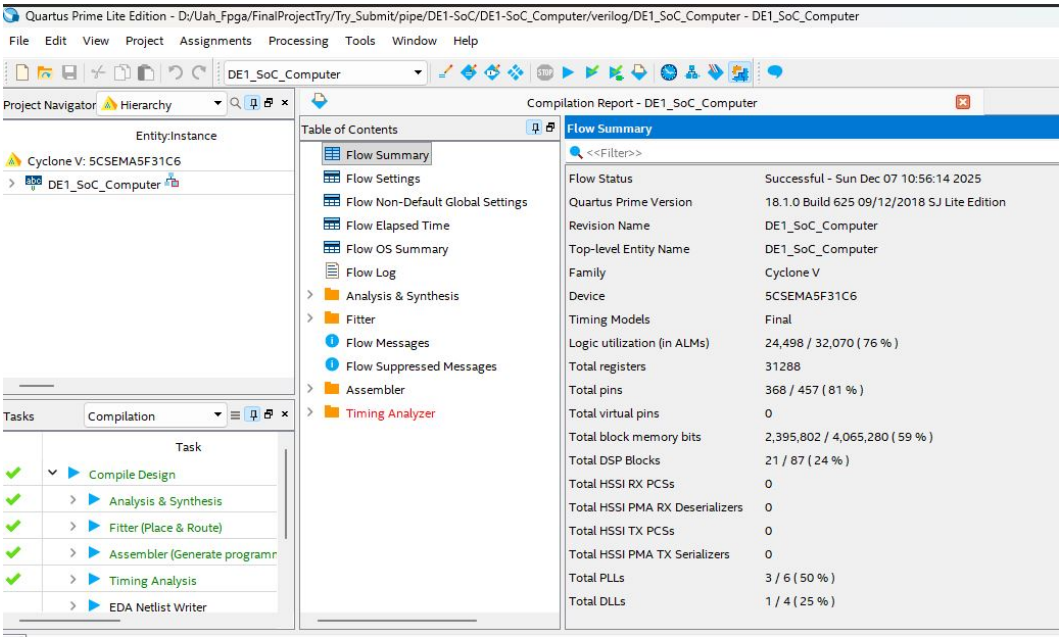

Figure 3: Compilation Report of the entire system including Accelerator

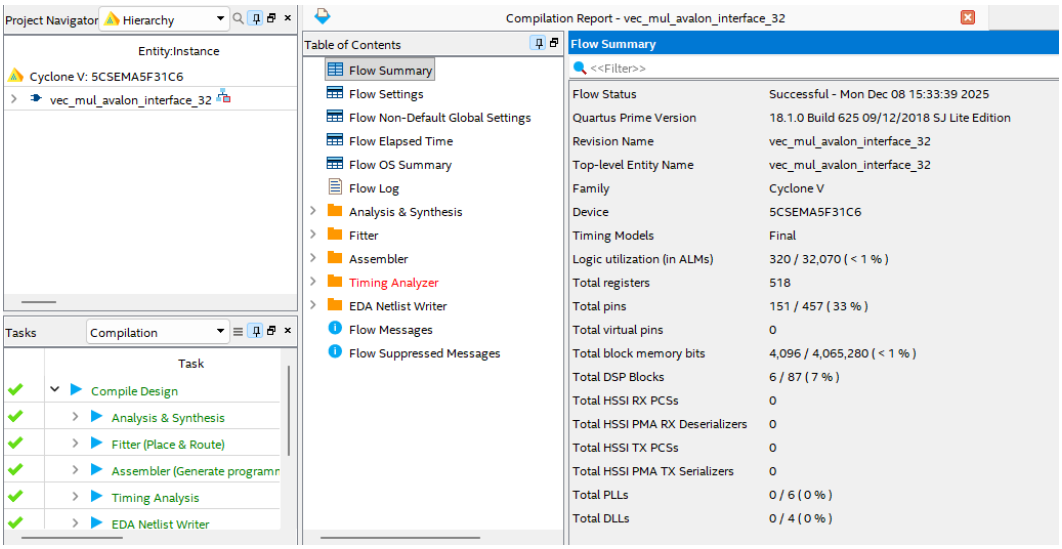

Figure 4: Compilation Report of the Accelerator

6 Experiments and Results

To evaluate the performance of the proposed vector dot product accelerator, we tested our implementations using randomly generated input vectors ranging from 0-9. Number of execution cycles was measured across two methods: software and the final pipelined hardware with SDRAM and dual FIFOs. Among these, the fully pipelined design consistently outperformed the software implementation, particularly for moderate to large vector sizes. The result achieved from the Pipelined+SDRAM hardware accelerator is stated below:

Value of N (Vector Size)	SW Cycles	HW Cycles	Speedup (SW/HW)
10	980	579	1.69
50	4112	915	4.49
100	8108	1376	5.89
200	9948	1416	7.03
300	16067	3401	4.72
500	31548	11183	2.82
1000	63293	22442	2.82
2000	126484	44617	2.83
5000	316050	111144	2.84
10000	632031	221996	2.85

Table 3: Measured software and hardware cycle counts and resulting speedup for various vector sizes.

Our experimental results revealed three distinct operational zones:

- **Startup zone:** For small vectors, for instance $N = 10, 50$, the hardware setup and SDRAM latency overhead dominate computation time, resulting in low or even sub-unity speedup. This reveals that for small workloads, the CPU can execute the task faster.

- **Sweet spot:** If the vectors are of lengths in the mid-range, for example $N = 100$ to 300, our final pipelined hardware core with SDRAM becomes fully utilized. The FIFOs effectively addresses the issue of SDRAM latency. Therefore, the MAC unit achieves near-peak throughput. This is because once the FIFOs have sufficient data, the MAC unit does the multiply-accumulation task in every clock cycle until the buffers are non-empty. This is the region of vector length where our final accelerator design demonstrates optimal performance, with speedup increasing significantly.

- **Saturation zone:** Beyond vector length $N = 500$, the speedup begins to plateau. Because of the limited memory bandwidth of the 16-bit SDRAM interface, the system reaches a bottleneck where further increases in vector size no longer yield proportional speedup. This behavior may occur as now it requires 2 trips to load a 32-bit value from the SDRAM.

A comparison of Pipelined+SDRAM design with Baseline approach is shown below:

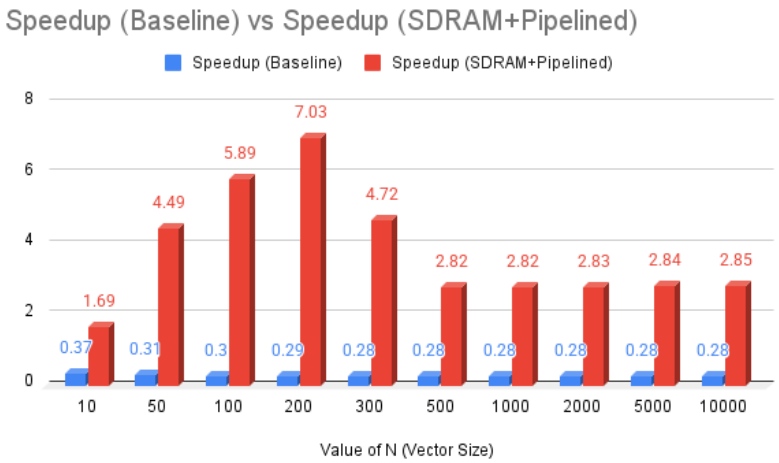

Figure 5: Comparison of Pipelined+SDRAM design with baseline approach.

7 Conclusion

In this project, we have designed and implemented a hardware accelerator for vector dot product computation on the Terasic DE1-SoC platform. By utilizing SystemVerilog and Avalon interfaces, our pipelined architecture with dual FIFO buffers and DMA-based memory access significantly reduced execution latency in terms of clock-cycles compared to a software-only implementation and another hardware approach without piplelining or DMA. More specifically, in this project, we have developed a complete C-based software implementation for reference and also introduced a non-pipelined hardware baseline to better understand the benefits of pipelining and data streaming. Experimental results across varying vector sizes have clearly demonstrated that our pipelined accelerator achieves substantial speedup, especially in the mid-to-large vector size range. Therefore, this project highlights the importance of custom hardware design in accelerating memory-intensive operations by offloading some tasks from the CPU and opens pathways for deploying similar accelerators in real-time embedded systems. However, in our current design, since the SDRAM is 16 bits wide while our core requests 32-bit data, each 32-bit read requires two trips to the SDRAM. This introduces a memory access bottleneck that limits the throughput. In future work, we plan to implement a double-buffering scheme that will help address the SDRAM latency and allow sustained data flow to the pipeline, improving overall system efficiency.

A Necessary Screenshots

We now present screenshots illustrating the progression of our project through its different implementation stages.

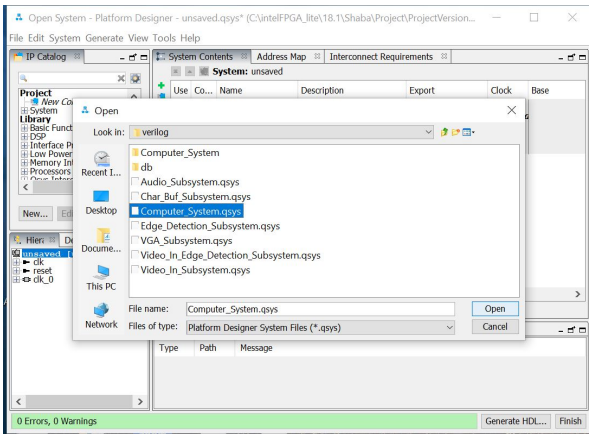

Figure 6: Loading the existing Computer_System.qsys and creating new component.

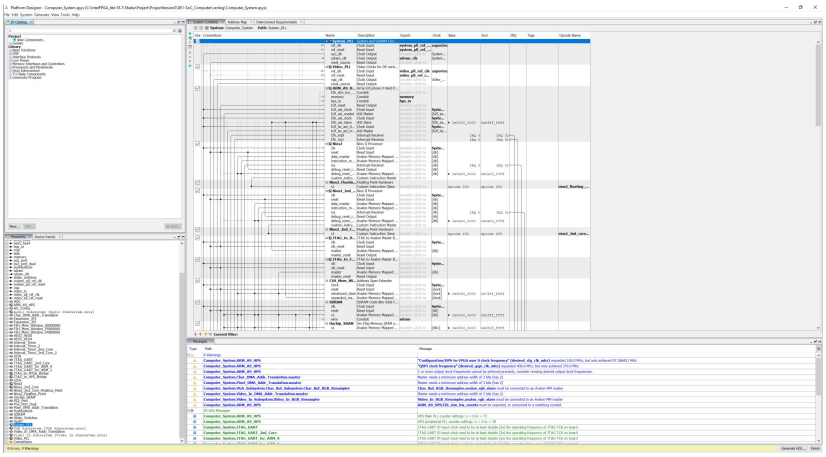

Figure 7: Loading the Platform Designer from Quartus.

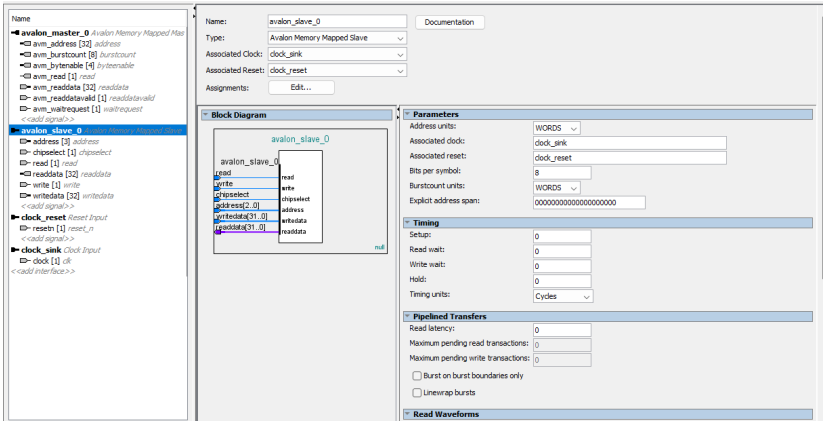

Figure 8: Avalon slave in the new component.

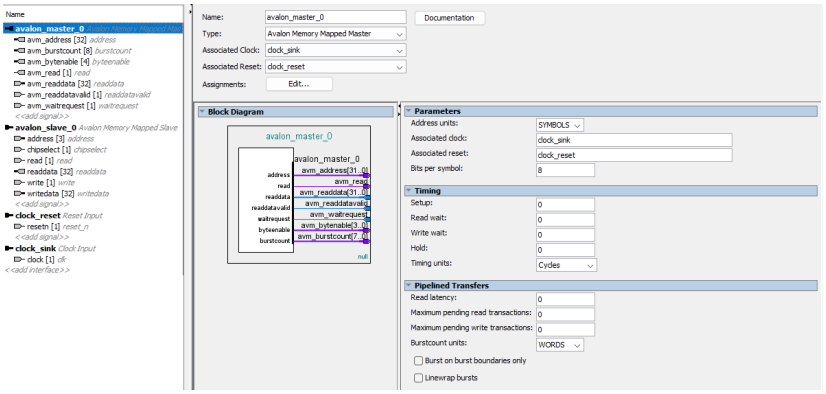

Figure 9: Avalon master in the new component.

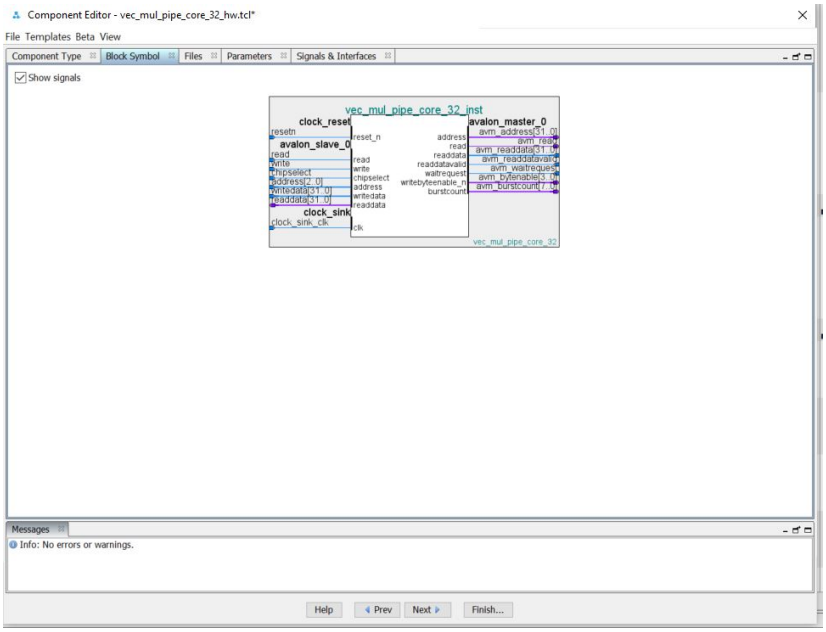

Figure 10: Block symbol of the new component.

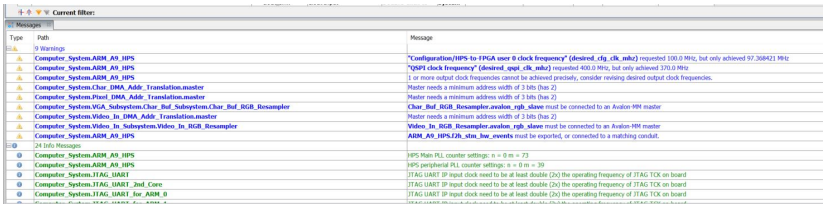

Figure 11: No error shown after connecting the new component to the existing computer system.

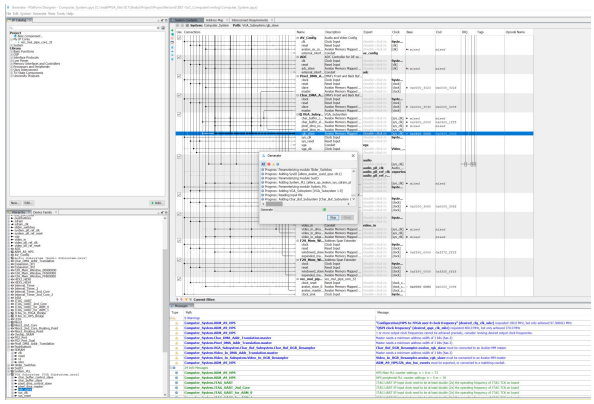

Figure 12: Generating HDL.

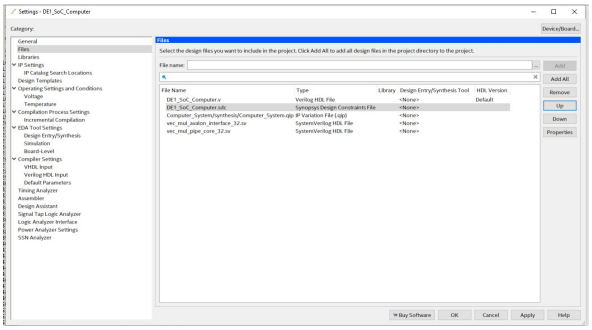

Figure 13: Removing previous Computer_System.qip file and adding the newly generated Computer_System.qip file.

```
Terminal
Enter the size of Vector: 200

 SW Result: 3859; Clock Cycles: 9948 cc
 HW Result: 3859; Clock Cycles: 1416 cc
Speedup: 7.03x
Continue (y/n)? y

Enter the size of Vector: 300

 SW Result: 6101; Clock Cycles: 16067 cc
 HW Result: 6101; Clock Cycles: 3401 cc
Speedup: 4.72x
Continue (y/n)? y

Enter the size of Vector: 500

 SW Result: 10659; Clock Cycles: 31548 cc
 HW Result: 10659; Clock Cycles: 11183 cc
Speedup: 2.82x
Continue (y/n)? y

Enter the size of Vector: 1000

 SW Result: 19952; Clock Cycles: 63293 cc
 HW Result: 19952; Clock Cycles: 22442 cc
Speedup: 2.82x
Continue (y/n)? y

Enter the size of Vector: 2000

 SW Result: 40562; Clock Cycles: 126484 cc
 HW Result: 40562; Clock Cycles: 44617 cc
Speedup: 2.83x
Continue (y/n)? y

Enter the size of Vector: 5000

 SW Result: 100608; Clock Cycles: 316050 cc
 HW Result: 100608; Clock Cycles: 111144 cc
Speedup: 2.84x
Continue (y/n)? y

Enter the size of Vector: 10000

 SW Result: 200852; Clock Cycles: 632031 cc
 HW Result: 200852; Clock Cycles: 221996 cc
Speedup: 2.85x
Continue (y/n)?
```

Figure 14: Outputs on terminal.

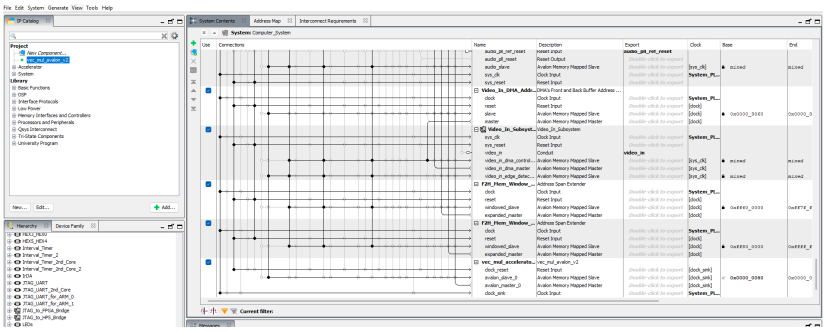

Figure 15: Platform design.

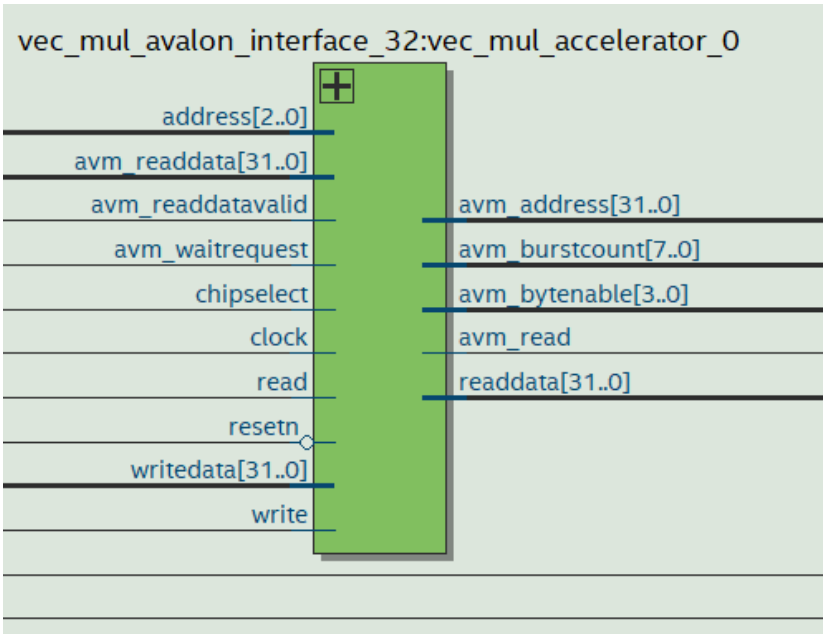

Figure 16: RTL view.

B Software Implementation in C

```
#include<stdio.h>
#include<stdlib.h>

#define H2F_BASE 0xFF200000
#define VEC_MUL_BASE 0x00000080
#define FPGA_SDRAM_BASE 0xC0000000
#define ACC_SDRAM_BASE 0x00000000

//Interval Timer
#define INTERVAL_TIMER_BASE 0xFF202000
#define CLK_FREQ 100000000.0

//Register offset
#define CTRL_REG 0
#define VEC_LENGTH 1
#define vecA_BaseAddress_REG 2
#define vecB_BaseAddress_REG 3
#define RES_LO 4
#define RES_HI 5

//Timer Functions
static inline void init_timer(volatile unsigned int* tbase){
 *(tbase + 1) = 0x8;     // STOP=1 in control reg
    *(tbase + 2) = 0xFFFF;  // period low
```

```
    *(tbase + 3) = 0xFFFF;  // period high
    *(tbase + 1) = 0x6;     // Start=1, CONT=1 (continuous mode)
}


static inline unsigned int read_timer(volatile unsigned int* tbase) {

 *(tbase + 4) = 0;                    // trigger snapshot
    unsigned int lo = *(tbase + 4);     // read SNAP Lower
    unsigned int hi = *(tbase + 5);     // read SNAP Higher
    return (hi << 16) | (lo & 0xFFFF);  //return the snapped cc
}



static inline void stop_timer(volatile unsigned int* tbase){
 *(tbase + 1) = 0x8;     // STOP=1 in control reg

}


long long vec_mul_hw_64(volatile unsigned int *vec_reg, unsigned int addressA,
                            unsigned int addressB, int N, unsigned int* total_time){

 volatile unsigned int * TIMER_ptr = (volatile unsigned int *) INTERVAL_TIMER_BASE;


 long long result = 0;


 init_timer(TIMER_ptr);
 unsigned int start_time = read_timer(TIMER_ptr); //Start timer

 //Write Registers
 vec_reg[VEC_LENGTH] = N;
 vec_reg[vecA_BaseAddress_REG] = addressA;
 vec_reg[vecB_BaseAddress_REG] = addressB;

 //Start signal for accelerator
 vec_reg[CTRL_REG] = 1;

 //This is the busy bit for the control reg. BUSY = 1 (RUN), BUSY = 0 (DONE)
 // When done we can get the result
 while((vec_reg[CTRL_REG] & 0x1) == 1);

 //Read Result HI and LO
```

```
 unsigned int res_lo = vec_reg[RES_LO];
 unsigned int res_hi = vec_reg[RES_HI];
 result =  ((long long) res_hi <<32) | res_lo;

 stop_timer(TIMER_ptr);
 unsigned int end_time = read_timer(TIMER_ptr);

 //Calculate the time and stop the timer
 *total_time = start_time-end_time;


 //Concat the results first shift res_hi to upper 32 bits and then concat with res_lo
 return result;

}

long long vec_mul_sw_64 (volatile int *mem_ptr, int addressA,
                            int addressB, int N, unsigned int* total_time){
 volatile unsigned int * TIMER_ptr = (volatile unsigned int *) INTERVAL_TIMER_BASE;


 long long result = 0;

 init_timer(TIMER_ptr);
 unsigned int start_time = read_timer(TIMER_ptr); //Start timer

 //We initialize the vectors which points to the start address of SDRAM's each vector
 volatile int *vec_a = mem_ptr + addressA;
 volatile int *vec_b = mem_ptr + addressB;

 //Calculate the vector dot product and accumulate in result
 for (int i = 0; i<N; i++){
  result+=(long long) vec_a[i] * vec_b[i];
 }

 stop_timer(TIMER_ptr);

 unsigned int end_time = read_timer(TIMER_ptr);
 //Calculate the time and stop the timer
 *total_time = start_time-end_time;


 return result;

}
```

```
int main(){
    //connects to the accelerator
 volatile unsigned int *vec_reg = (volatile unsigned int *)(VEC_MUL_BASE + H2F_BASE);
 volatile int *sdram_ptr = (volatile int *)(FPGA_SDRAM_BASE);

 unsigned int time_sw, time_hw;


 printf("Welcome to Vector Dot Product Multiplication Calculator! \n");
 while(1){
  int N = 0;
  printf("\nEnter the size of Vector: ");
  scanf("%d", &N);

  //Start index in SDRAM
  int addressA_startIndex = 0;
  int addressB_startIndex = N;

  //for the accelerator
  unsigned int vecA_address = ACC_SDRAM_BASE + (addressA_startIndex*4);
  unsigned int vecB_address = ACC_SDRAM_BASE + (addressB_startIndex*4);

  srand(0);
  //Put random numbers in both the vectors
  for(int i=0; i<N; i++){
   //generate random numbers from 0-9
   sdram_ptr[addressA_startIndex+i] = rand() % 10;
   sdram_ptr[addressB_startIndex+i] = rand() % 10;
  }

  //Software RUN
  long long res_sw = vec_mul_sw_64 (
                                        sdram_ptr,
                                        addressA_startIndex,
                                        addressB_startIndex,
                                        N,
                                        &time_sw
                                        );

  printf("\n SW Result: %lld; Clock Cycles: %u cc ", res_sw, time_sw);

  //Hardware RUN
  long long res_hw = vec_mul_hw_64 (
                                        vec_reg,
                                        vecA_address,
                                        vecB_address,
```

```
                                        N,
                                        &time_hw
                                        );

  printf("\n HW Result: %lld; Clock Cycles: %u cc ", res_hw, time_hw);


  //Check Speedup
  printf("\nSpeedup: %.2fx\n", (double)time_sw/(double)time_hw);

  printf("Continue (y/n)? ");
        char sel; scanf(" %c", &sel);
        if (sel == 'n' || sel == 'N') break;


 }
 return 0;

}
```

C System Verilog Code:

C.1 Avalon Memory Mapped Interface: vec_mul_avalon_interface_32.sv

```
module vec_mul_avalon_interface_32(
 //Avalon MM slave interface
 input logic clock,
 input logic resetn,
 input logic read,     //read signal from CPU
 input logic write,     //Write signal from CPU
 input logic chipselect,   //Component selection
 input logic [2:0] address,   //Register MAP offset
 input logic [31:0] writedata,  //Data from CPU
 output logic [31:0] readdata,  //Data to CPU

 //Avalon MM master interface (connected to SDRAM)
 output logic [31:0] avm_address,   //address to read from SDRAM
 output logic avm_read,      //read signal to SDRAM
 input logic [31:0] avm_readdata,   //data from SDRAM
 input logic avm_readdatavalid,    //new data delivery signal
 input logic avm_waitrequest,    //Memory busy signal (need to wait)
 output logic [3:0] avm_bytenable,   //bytes to read
 output logic [7:0] avm_burstcount  //number of items to read
```

```
);


 //The register MAP used:
 //Address 0 Control&Busy (Read/Write) Bit 0: busy flag; Writing to this reg starts
    the process
 //Address 1 VEC_Length (Read/Write) Number of elements "N"
 //Address 2 vecA_baseAddr (Read/Write)  Base address of Vector A in SDRAM
 //Address 3 vecB_baseAddr (Read/Write)  Base address of Vector B in SDRAM
 //Address 4 RES_LO (Read)  Lower 32 Bits of dot product Result
 //Address 5 RES_HI  (Read)  Upper 32 Bits of dot product Result
 //Total 64 Bits for result


 //Internal Registers
 logic [31:0] vec_length;   //Value of N, number of elements
 logic [31:0] vecA_baseAddr;  //Base address of A in SDRAM
 logic [31:0] vecB_baseAddr;  //Base address of B in SDRAM
 logic [31:0] RES_LO;    //Lower 32 Bits of dot product Result
 logic [31:0] RES_HI;   //Upper 32 Bits of dot product Result

 //Signals for core computation
 logic dot_start;
 logic dot_busy;
 logic dot_busy_prev;
 //64 bit Dot product result
 logic [63:0] dot_result;


 //computation start logic
 assign dot_start = chipselect & write & (address == 3'd0); //Address 0 in for
    Control, writing anything starts the computation


 vec_mul_pipe_core_32 coreComp (
  //Avalon MM slave
  .clock(clock),
  .resetn(resetn),
  .start(dot_start),      //start of process
  .length(vec_length),   //length of the vectors
  .vecA_baseAddr(vecA_baseAddr), //start address of Vector A
  .vecB_baseAddr(vecB_baseAddr),  //Start address of Vector B
  .busy(dot_busy),      //busy status
  .result(dot_result),   //accumulated result
```

```
 //Avalon MM master (connected to the SDRAM)
 .avm_address(avm_address),  //address to read from SDRAM
 .avm_read(avm_read),     //read signal to SDRAM
 .avm_readdata(avm_readdata),  //data from SDRAM
 .avm_readdatavalid(avm_readdatavalid),   //new data delivery signal
 .avm_waitrequest(avm_waitrequest),   //Memory busy signal (need to wait)
 .avm_bytenable(avm_bytenable),  //bytes to read
 .avm_burstcount(avm_burstcount) //number of items to read


);

//Write logic from CPU to FPGA
always_ff @(posedge clock or negedge resetn) begin

 if(!resetn) begin
  vecA_baseAddr <= 32'b0;
  vecB_baseAddr <= 32'b0;
  RES_LO <= 32'b0;
  RES_HI <= 32'b0;
  dot_busy_prev <= 1'b0;

 end

 else begin
  if (chipselect & write) begin
   case(address)
    3'd1:  vec_length <= writedata;   //Writes length
    3'd2:  vecA_baseAddr <= writedata;   //Writes Base address of A
    3'd3:  vecB_baseAddr <= writedata;   //Writes Base address of B
    default: ;
   endcase
  end

  //Will need to store the result
  //check if it is busy in the previous cycle
  dot_busy_prev <= dot_busy;
  if(dot_busy_prev && !dot_busy) begin
   RES_LO <= dot_result[31:0];
   RES_HI <= dot_result[63:32];
  end
 end
end

//Read logic from FPGA to CPU
always_comb begin
```

```
  readdata = 32'b0;

  if(chipselect & read) begin
   case(address)
    3'd0: readdata = {31'b0, dot_busy}; //Control/Busy Register
    3'd1: readdata = vec_length;   //Read back Length N
    3'd2: readdata = vecA_baseAddr;  //Read back Base address of A
    3'd3: readdata = vecB_baseAddr;  //Read back Base address of B
    3'd4: readdata = RES_LO;    //Read Lower 32 bits Result
    3'd5: readdata = RES_HI;    //Read Upper 32 bits Result
    default: readdata = 32'b0;
   endcase
  end
 end

endmodule
```

C.2 Core Computational Logic: vec_mul_pipe_core_32.sv

```
//We will use 3 states here: IDLE, RUN, DONE

module vec_mul_pipe_core_32 (
 //Avalon MM slave
 input logic clock,
 input logic resetn,
 input logic start,      //start of process
 input logic [31:0] length,   //length of the vectors
 input logic [31:0] vecA_baseAddr, //start address of Vector A
 input logic [31:0] vecB_baseAddr,  //Start address of Vector B
 output logic busy,      //busy status
 output logic [63:0] result,   //accumulated result


 //Avalon MM master (connected to the SDRAM)
 output logic [31:0] avm_address,  //address to read from SDRAM
 output logic  avm_read,     //read signal to SDRAM
 input logic [31:0] avm_readdata,  //data from SDRAM
 input logic avm_readdatavalid,   //new data delivery signal
 input logic avm_waitrequest,   //Memory busy signal (need to wait)
 output logic [3:0] avm_bytenable,  //bytes to read
 output logic [7:0] avm_burstcount //number of items to read



);
```

```
//2 FIFO Buffers
//One for Vector A and One for Vector Buffers
//Used for calculation of the vector dot product so that the calculation will not
   totally depend on the memory
//also while fetching from memory some data might lost
localparam FIFO_DEPTH = 64;
logic [31:0] FIFO_A [0:FIFO_DEPTH-1];       //FIFO for vector A
logic [31:0] FIFO_B [0:FIFO_DEPTH-1];       //FIFO for vector B
logic [$clog2(FIFO_DEPTH):0] FIFO_A_count, FIFO_B_count; //available items
   in the FIFOs
logic [$clog2(FIFO_DEPTH)-1:0] write_A_ptr, write_B_ptr, read_A_ptr, read_B_ptr;
   //Fifo read and write pointers

//state machine
typedef enum logic [1:0] {IDLE, RUN, DONE} state_t;
state_t state;

//Internal logics
logic [31:0] read_req_index;   //number of read requests sent, also works as index
logic [31:0] calculation_index;  //number of calculations done
logic [31:0] vec_length;    //vector length
logic [63:0] accum;     //accumulated dot product
logic readReq_b;      //0 = request is A, 1 = request if B
logic dataRecv_b;      //0 = data received is A, 1 = data received is B

assign busy = (state != IDLE);   //busy if not in IDLE
assign result = accum;     //Get the total vector dot product from accum

assign avm_burstcount = 8'b1;   //read 1 word at a time
assign avm_bytenable = 4'b1111;   //read all 4 bytes (32 bits)

//Combinational logic to calculate the next address
always_comb begin
 avm_address = 0;
 avm_read = 0;

 //Requests for Data in RUN state and until everything is fetched
 if (state == RUN && read_req_index < vec_length) begin
  //checks if FIFO has space
  if(FIFO_A_count < (FIFO_DEPTH-2) && FIFO_B_count < (FIFO_DEPTH-2)) begin
   avm_read = 1'b1;  //read signal to SDRAM

   //Address calculation (Base + )
   if(!readReq_b)
```

```
    avm_address = vecA_baseAddr + (read_req_index << 2); //Request vec A data
   else
    avm_address = vecB_baseAddr + (read_req_index << 2); //Request vec B data
  end
 end
end

//Main Sequential Logic

always_ff @(posedge clock or negedge resetn) begin

 if(!resetn) begin
  //reset everything to 0
  state <= IDLE;
  read_req_index <= 0;
  calculation_index <= 0;
  vec_length <= 0;
  accum <= 0;
  readReq_b <= 0;
  dataRecv_b <= 0;
  write_A_ptr <=0;
  write_B_ptr <= 0;
  read_A_ptr <= 0;
  read_B_ptr <= 0;

 end

 else begin
  case(state)
   IDLE: begin
    if (start) begin
     state <= RUN;   //Next state will go to RUN
     vec_length <= length;  //vector length
     //else everything in IDLE state will remain 0
     read_req_index <= 0;
     calculation_index <= 0;
     accum <= 0;
     readReq_b <= 0;
     dataRecv_b <= 0;
     write_A_ptr <=0;
     write_B_ptr <= 0;
     read_A_ptr <= 0;
     read_B_ptr <= 0;
    end

   end
```

```
 RUN: begin

  //if master_read (signal) is high and no wait from SDRAM
  if (avm_read && !avm_waitrequest) begin

   //If requested B, then the pair just finished (..A, B) so we will
                      inrease the index for the next pair
   if (readReq_b) begin
    read_req_index <= read_req_index + 1;
   end

   //we toggle from A to B or B to A
   readReq_b <= ~readReq_b;
  end

  //If all the index's calculation is done up to length then the next state
                 will be DONE
  if (calculation_index == vec_length) begin
   state <= DONE;

  end
 end

 DONE: begin
  state <= IDLE;  //in DONE state we just transit to the IDLE state
 end
endcase


//Data push in FIFO
//Checks if the new data is ready
if (avm_readdatavalid) begin
 //whose turn? A or B
 if(!dataRecv_b) begin
  FIFO_A[write_A_ptr] <= avm_readdata;  //push it into buffer
  write_A_ptr <= write_A_ptr + 1;    //increase the write pointer
  FIFO_A_count <= FIFO_A_count + 1;    //increase the FIFOA counter
 end

 else begin
  FIFO_B[write_B_ptr] <= avm_readdata;  //push it into buffer
  write_B_ptr <= write_B_ptr + 1;    //increase the write pointer
  FIFO_B_count <= FIFO_B_count + 1;    //increase the FIFOB counter
 end

 //toggle to the next A to B or B to A
```

```
     dataRecv_b <= ~dataRecv_b;
    end

    //Calculation in MAC
    //So after receiving the response from the SDRAM to FIFOs when we have values
              in the Buffer we can start out MAC
    if (state == RUN && (FIFO_A_count > 0) && (FIFO_B_count > 0)) begin
     //runs the main calculation logic using the read pointer of FIFO
     accum <= accum + (($signed(FIFO_A[read_A_ptr]))*($signed(FIFO_B[read_B_ptr])));

     //after calcualtion we increase the read pointer and the calculation index
     read_A_ptr <= read_A_ptr+1;
     read_B_ptr <= read_B_ptr+1;
     calculation_index <= calculation_index+1;

     //Done with the calculation
     //Need to update the FIFO counters
     //In pipelined, both a paired Data can be received and another calcualted
                 at same cycle (counter remains same)
     //If only calculated, we decrement the counter
     //For FIFO A
     if(avm_readdatavalid && !dataRecv_b) begin
      FIFO_A_count <= FIFO_A_count;  //Both Data received and calculated
     end

     else begin
      FIFO_A_count <= FIFO_A_count-1; //Only calculated
     end

     //For FIFO B
     if (avm_readdatavalid && dataRecv_b) begin
      FIFO_B_count <= FIFO_B_count; //Both Data rececived and calculated
     end

     else begin
      FIFO_B_count <= FIFO_B_count-1; //Only calculated
     end
    end


   end
  end

endmodule
```

References

[1] D. A. Patterson and J. L. Hennessy, "Computer organization and design," *The Hardware/Soft*, 1803.

[2] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "An efficient hardware accelerator for sparse convolutional neural networks on fpgas," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 17–25.

[3] H. Jin, "Digital scope implemented on altera de1-soc," *Cornell University, School of Electrical and Computer Engineering, MEng. Project Report*, 2016.

[4] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, "Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication," in *Proceedings of the 53rd annual design automation conference*, 2016, pp. 1–6.