Project 2 [ECE 720]

Digvijay Anand
200478940
[danand3@ncsu.edu]

Design and Simulation of a FIR Accelerator using HLS, TLM and ISS for SoC integration

---

Goal:

Minimize the performance metric of the designed 16 tap FIR Accelerator: cycles*critical delay using HLS+TLM+ISS.

1. Design a Finite Impulse Response (FIR) Filter Accelerator for use in an SoC
2. Partition the computation between hardware and software
3. Show that the Accelerator can be synthesized with HLS
4. Examine the speedup and area overhead of the relative to a software-only implementation

Project Requirements:

1. The FIR accelerator is the only module to be modified.
2. Compare sw-only (baseline) vs hw/sw design.
3. Compare the cycle-by-cycle performance of the Accelerator during *sc* and *vsim* simulation.
4. Summarize resource consumption by the Accelerator and compare the total assumed area-score.
5. Discuss your solution and how I achieve the design objectives.

All Tests and simulation **PASSED** [in vsim, hls, rocket_sim and sc]

Introduction

The design and simulation of hardware accelerators are critical for modern System-on-Chip (SoC) architectures, enabling efficient processing of complex workloads. High-Level Synthesis (HLS), Transaction-Level Modeling (TLM), and Instruction Set Simulation (ISS) play a pivotal role in this process, providing abstraction, efficiency, and verification capabilities. HLS enables developers to describe hardware accelerators in high-level languages, such as C++, which are then synthesized into Register Transfer Level (RTL) code. This approach accelerates design time while allowing optimization for performance, area, and power.

ISS serves as a functional model of the processor, simulating the execution of instructions to manage and verify accelerator integration. It allows testing of software and hardware interactions, including data transfers and control flow, without requiring physical hardware. The combination of TLM for modeling, HLS for synthesis, and ISS for functional validation offers a streamlined and efficient methodology for designing accelerators in SoC systems.

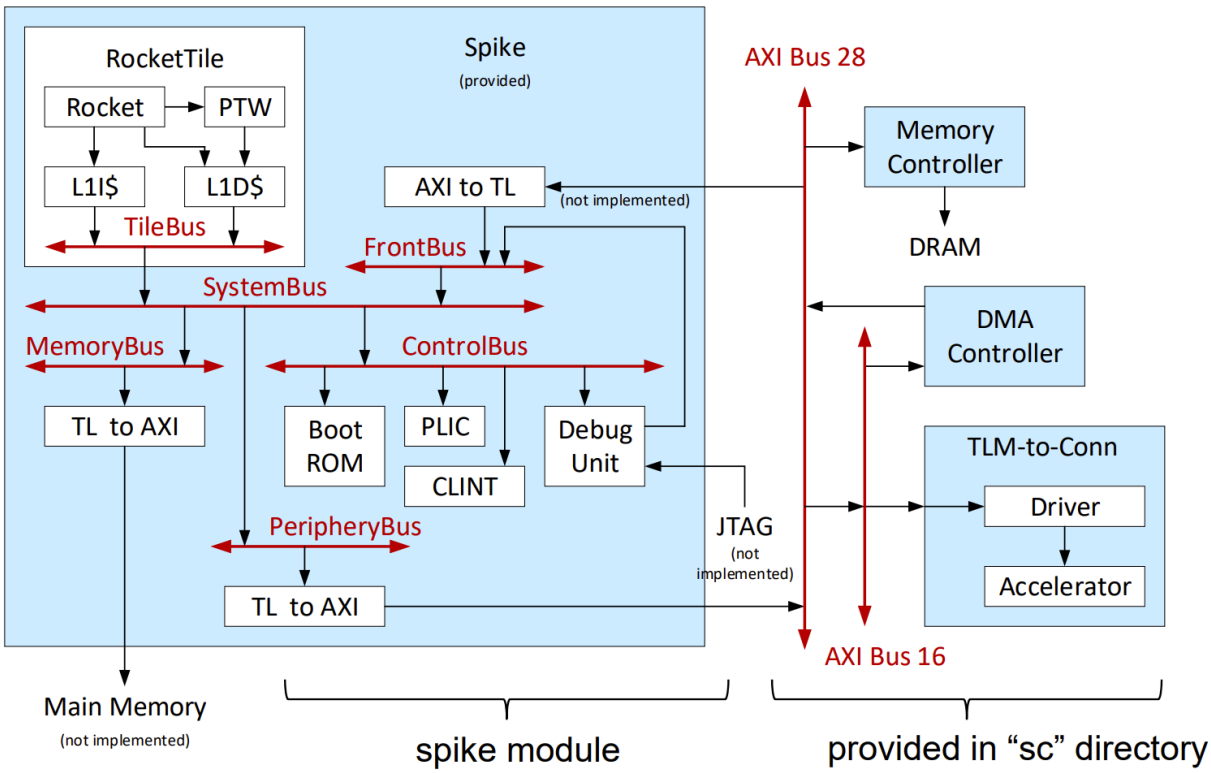

Figure 1. The complete System-on-Chip (SoC) used for HW/SW codesign of the Accelerator. It consists of a RISC-V based Rocket-Chip with access to an external memory controller and drivers for transaction to and fro our designed Accelerator and SoC.

The figure 1 below illustrates the design and simulation of a System-on-Chip (SoC) environment used for hardware-software co-design, featuring a RISC-V-based Rocket-Chip core, external memory controller, DMA controller, and a custom FIR accelerator. The architecture facilitates

efficient data processing and communication through modular components interconnected using AXI buses and transaction-level interfaces. In this project, we have used the following 5 major components used in this project.

1. FIR Accelerator is designed for hardware-accelerated FIR filtering, processing 16 time steps of data per iteration. It features 64-bit input ports (*w_in* and *x_in*), a 64-bit output port (*z_out*), and control/status registers (*CTRL* and *ST*) for managing operations. Burst transactions may ensure efficient data handling, and the accelerator operates with minimal CPU intervention, enabling high-throughput processing.
2. DMA Controller facilitates direct data movement between DRAM and the FIR accelerator, bypassing the CPU to reduce overhead. By supporting burst transactions, the DMA optimizes memory bandwidth, ensuring timely delivery of input data and retrieval of output results.
3. Memory Controller interfaces with DRAM, where input data, coefficients, and output results are stored. It supports high-speed burst transactions for efficient communication with the FIR accelerator. The controller ensures smooth data flow between the CPU, DMA, and accelerator, crucial for maintaining system performance.
4. Spike Simulator serves as a functional RISC-V CPU model, orchestrating the system's control flow and data handling. It configures DMA transfers, initiates accelerator operations, and validates output correctness. Its components, including RocketTile, caches, and interrupt controllers, provide system-level debugging and simulation capabilities.
5. TLM-to-Conn Driver bridges high-level Transaction-Level Modeling (TLM) and low-level AXI protocols, ensuring seamless integration of the FIR accelerator into the SoC. It converts high-level TLM transactions into AXI-compliant signals, maintaining protocol adherence and enabling efficient communication.

FIR - Algorithm:

$$y[n] = \sum_{m=0}^{TAPS-1} h[m] \cdot x[n-m]$$

The FIR filter algorithm processes an input signal *x[n]* by applying a set of filter coefficients *h[m]* to produce a filtered output *y[n]* through a convolution operation. For each time step *n* of the output y[n], the algorithm iterates through the filter coefficients h[m], multiplying each coefficient by the corresponding delayed input value x[n−m]. In the next sections, we will develop an accelerator for this operation using HW/SW co-design.

Design Methodology

In this section, we will discuss the algorithm (1) used by Accelerator to perform 16 TAPS FIR filtering. The accelerator begins in the reset state, where all internal buffers and pointers are cleared. In the coefficient loading state, the system reads coefficients from the *w_in* FIFO,

storing them as 16-bit values in the coefficient buffer. Once coefficients are loaded, the system transitions to the input data processing state. Here, input data is fetched from *x_in*, and the FIR operation is performed by convolving the input buffer with the coefficients. The algorithm below describes the processing by the design:

```
1: Initialize all buffers and registers to zero.
2: while true do
3:     if ctrl == 0xFF then                                  ▷ Reset State
4:         Reset all internal states and pointers.
5:     else if ctrl == 0x01 then                             ▷ Load Coefficients
6:         if !w_in.Empty() then
7:             Read coefficients from w_in and store in data_burst16_2[].
8:         end if
9:     else if ctrl == 0x02 then                             ▷ Process Input Data
10:        if !x_in.Empty() then
11:            Read x_in64 and update data_burst16[].
12:            Perform FIR filtering and store results in output[].
13:        end if
14:        if output_ready == 0xF and !z_out.Full() then
15:            Pack and push the 64-bit result to z_out.
16:        end if
17:        Reset output_ready and output[].
18:    end if
19:    Wait for the next clock cycle.
20: end while
```

Algorithm 1: Pseudo-code for Accelerator with 16 TAPS [fixed]

The FIR accelerator operates as a state machine with the following states:

1. Reset State (CTRL = 0xFF): Initializes all internal buffers, registers, and pointers. This state ensures that the system is in a clean state before any operation begins.
2. Load Coefficients (CTRL = 0x01): Reads coefficients from the input FIFO (w_in) and stores them in the internal coefficient buffer (data_burst16_2[]). Coefficients are packed into 64-bit transactions and split into 16-bit chunks.
3. Process Input Data (CTRL = 0x02): Reads input data from x_in, updates the input buffer (data_burst16[]), and performs FIR filtering operation. The results are accumulated into the output buffer and packed into a 64-bit output format for transmission via z_out.
4. Output State: Combines the FIR output values into a 64-bit word and pushes it to the output FIFO if space is available.

The central part is the FIR operation. The code implements a sliding window FIR filtering operation with efficient data management and output generation. It begins by updating a 16-element input buffer (*data_burst16*) using a sliding mechanism, where the oldest 4 values are stored in a secondary buffer (*burst_buf*), and the newest 4 values are loaded from the input (*x_in64*). The FIR computation is performed for 4 output values in parallel, convolving the

16-element input buffer with 16 filter coefficients. Depending on the index, data is fetched either from *data_burst16* or wrapped around using *burst_buf*. The computed outputs are accumulated into a 4-element output buffer, and when all outputs are ready, they are packed into a single 64-bit word and pushed to the output FIFO (*z_out*). This process ensures efficient data reuse, minimizes latency, and enables high-throughput FIR filtering in hardware. In the next subsequent section we will validate these findings.

The pseudo-code for FIR Filter Accelerator algorithm is included above in 2. It executes FIR filtering by leveraging a hardware accelerator and DMA for optimized data movement.

- Initially, coefficients, input, and output buffers are defined in memory.
- For the hardware-accelerated operation, coefficients are first transferred from memory to the accelerator's coefficient register (*accel_w*) using DMA.
- Input data is then transferred to the input register (*accel_x*), and the accelerator performs the FIR computation.

```
1: Define constants: TAPS = 16, TSTEP1 = 32, TSTEP2 = 48.
2: Define memory-mapped addresses for DMA and Accelerator.
3: procedure FIR Accelerator Execution
4:     Initialize coef, input, and output buffers in memory.
                                        ▷ Accelerator-Based FIR Filtering
5:     Set accel_ctrl = 1 to load coefficients.
6:     Use DMA to transfer coefficients from coef to accel_w.
7:     Wait for DMA to complete.
8:     Set accel_ctrl = 2 to load input data.
9:     Use DMA to transfer input from input to accel_x.
10:    Wait for DMA to complete.
11:    Transfer output from accel_z to output using DMA.
                                                  ▷ Error Checking
12:    total_error = 0.
13:    for n = 0 to TSTEP1 + TSTEP2 - 1 do
14:        Compute error: error = abs(expected[n] - output[n]).
15:        Accumulate total error: total_error += error.
16:    end for
17:    Print total_error for validation.
                                                  ▷ Finalization
18:    Set accel_ctrl = 0x0f to exit.
19: end procedure
```

Algorithm 2: Pseudo-code for SW-Only handling the designed accelerator for processing input streams. In the provided pseudo-code we load and store values to memory and do error checks.

- Output data is fetched from the accelerator's output register (*accel_z*) and written back to memory.

clk_per	realops	latency	throughput	critpath
5	403	161	160	3.862839

Table 1: The table provides a detailed breakdown of the performance metrics for a hardware design

- The DMA enables burst transactions, reducing latency and ensuring efficient communication between memory and the accelerator.
- After execution, the output is validated by comparing it with the expected results, and the total error is calculated.
- The algorithm concludes by resetting the accelerator's internal states and issuing an exit command, ensuring proper resource cleanup and readiness for subsequent operations.

Performance Results[1]

In terms of latency, the table below provides a clear comparison of latency and speedup achieved through hardware-software (HW/SW) co-design in different simulation frameworks compared to a software-only implementation. The HW/SW simulation with vsim achieves significant acceleration by offloading computational workloads to the accelerator while, SystemC provides even greater acceleration due to higher-level modeling and efficient resource utilization.

While relative comparisons are concerned, sc simulation for the accelerator is *~1.64x* faster than vsim but then, it lacks the detailed accurate traces and are highly optimized simulation techniques. Also, the timing will be more comparable with larger designs. On the other hand, the area overhead is negligible in comparison to the software-only implementation, showcasing the efficiency of the hardware accelerator. The system currently has a latency of *161* cycles, throughput of *160* operations, and has *403* real operations processed.

	Latency (*ns*)		Area Overhead	
S/W Only	HW/SW vsim	HW/SW sc	S/W Only	HW/SW hls
170881	4751	2890	1,500,000	10398.2
Speed Up (\xtimes)	**35.96**	**59.12**	**-**	**0.006884 ~ roughly 0.7 %**

Table 2: The table provides a clear comparison of latency and speedup achieved through hardware-software (HW/SW) co-design in different simulation frameworks compared to a software-only implementation.

[1] For performance evaluation, I have used the default toolkit with the changes suggested in the Moodle.

- # of clock cycles = 4751
- Critical Path Delay (critpath) = $3.86\ ns$

Total time to complete the operations = $4751\ *\ 3.86\ ns\ =\ 18,352.34\ ns$

Vsim vs SC Simulation Time:

Figure 2 highlights the differences in how simulation environments handle increasing computational workloads. Both vsim and SystemC (sc) exhibit a linear increase in execution time with higher loop counts, reflecting the iterative nature of FIR filtering and its proportional computational demands. Here, I have used 80 FIR operations in each loop iteration.

However, SystemC consistently outperforms vsim, showing significantly lower execution times across all loop counts. This performance difference arises because SystemC abstracts cycle-level hardware details, focusing on system-level behavior, which allows for faster simulations while maintaining functional accuracy. In contrast, vsim models low-level hardware operations, including cycle-by-cycle execution, leading to slower performance as workloads grow. At higher loop counts, such as 3000, vsim becomes impractical ("TOO LONG") due to its resource-intensive nature, whereas SystemC remains efficient and scalable, providing results even for large and complex simulations. These findings underscore the trade-offs between simulation accuracy and speed, with vsim excelling in hardware-level precision and SystemC offering a more scalable and efficient solution for system-level design and verification.

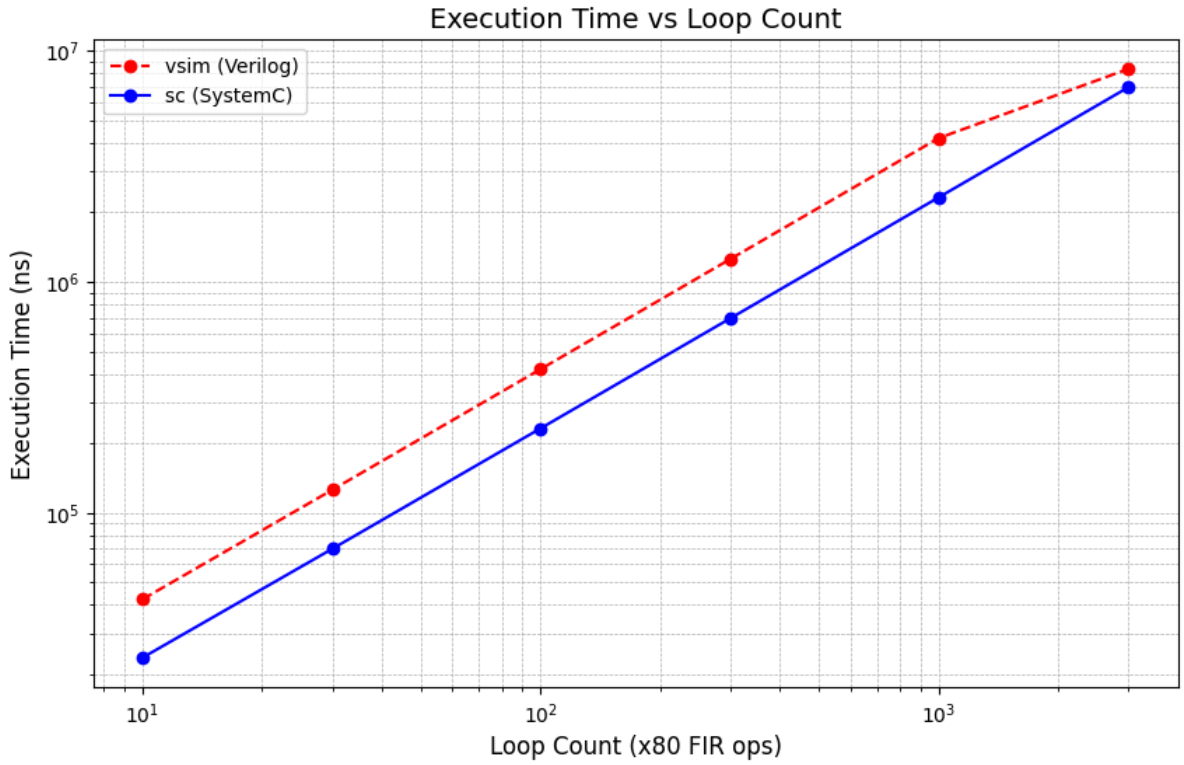

Figure 2: This figure compares the execution time vs. Loop Count for vsim and sc simulations.

Loop count	10	30	100	300	1000	3000
vsim	42,294	125,934	418,650	1,255,050	4,182,450	TOO LONG
sc	23,644	69,984	232,150	695,550	2,317,450	6,951,451

Table 3: The table provides execution time results (in nanoseconds) for the FIR filtering operation across different loop counts, comparing vsim (a Verilog simulation) and sc (a SystemC simulation).

Resource Summary:

Component	Post-Assignment
Total Area Score	10382.7 (100%)
Total Reg	6431.9 (62%)
MUX	2148.2 (21%)
FUNC	744.7 (7%)
LOGIC	1057.4 (10%)
BUFFER	0.0
MEM	0.0
ROM	0.0
REG	6415.9 (62%)
FSM-REG	16.0 (97%)
FSM-COMB	0.5 (3%)

Table 4: Area breakdown of the hardware design after post-assignment optimization, highlighting the contributions of registers, logic, and multiplexers, with detailed analysis of FSM components.

The table 4 provides an area breakdown of the hardware design after post-assignment optimization. The total area score is 10,382.7, with 62% of the total area contributed by registers. Among functional components, multiplexers account for 21% of the area, while logic and functional units (LOGIC and FUNC) contribute 10% and 7%, respectively. No area is allocated to buffers, memory, or ROM since we are using DRAM. The breakdown also includes the finite state machine, where FSM-REG (register-based state storage) occupies 97% of the FSM area, and FSM-COMB (combinational logic) occupies the rest. This analysis highlights the dominance of registers and multiplexers in area utilization, reflecting design trade-offs for performance and functionality.

Conclusion:

The results from the implementation and simulation of the FIR Accelerator demonstrate the significant benefits of leveraging hardware-software (HW/SW) co-design for computationally intensive tasks. Compared to a software-only implementation, the HW/SW approach drastically reduces latency by offloading the FIR filtering operation to a dedicated accelerator. The simulation results highlight that SystemC (sc) achieves a latency reduction of approximately 59.12x compared to the software-only baseline, while Verilog simulation (vsim) achieves a 35.96x reduction. We notice that the design time and the amount of know-how required before

designing a SystemC level designs is less and often faster. These findings underline the advantages of integrating hardware accelerators into System-on-Chip (SoC) architectures. The results also validate the utility of simulation frameworks like SystemC for high-level modeling and efficient resource utilization, providing valuable insights for future accelerator designs in resource-constrained environments.

Simulations:

RISCV_SIM ?= ../vsim/simv [no errors with sc/vsim simulations]:

```
cpu main FIR total error: 0
9680 ns tlm2conn WRITE len:0x8 addr:0x8
9681 ns tlm2conn.driver WRITE addr=0x8 length=0x8 data=0xf
9681 ns tlm2conn transaction complete
9681 ns tlm2conn received exit signal

Info: SystemC: Simulation stopped by user.
Simulation time: 9681 ns
Wall clock time: 2 seconds
          V C S   S i m u l a t i o n   R e p o r t
Time: 9681000 ps
CPU Time:      0.780 seconds;       Data structure size:   0.0Mb
Wed Dec 11 03:08:06 2024

real    0m2.766s
user    0m0.619s
sys     0m0.196s
stty sane
trace -f spike -o fir.riscv.dump fir.spike.out > fir.spike.trace
```

RISCV_SIM ?= ../vsim/simv:

```
4751 ns tlm2conn.driver WRITE addr=0x8 length=0x8 data=0xf
4751 ns tlm2conn transaction complete
4751 ns tlm2conn received exit signal

Info: SystemC: Simulation stopped by user.
Simulation time: 4751 ns
Wall clock time: 2 seconds
          V C S   S i m u l a t i o n   R e p o r t
Time: 4751000 ps
CPU Time:      0.570 seconds;       Data structure size:   0.0Mb
Wed Dec 11 04:04:22 2024

real    0m1.773s
user    0m0.495s
sys     0m0.111s
stty sane
trace -f spike -o fir.riscv.dump fir.spike.out > fir.spike.trace
[danand3@grendel46 rocket_sim]$
```

SystemC Simulation:

```
2890 ns tlm2conn received exit signal

Info: /OSCI/SystemC: Simulation stopped by user.
Simulation time: 2890 ns
Wall clock time: 0 seconds

real    0m0.199s
user    0m0.023s
sys     0m0.019s
stty sane
trace -f spike -o fir.riscv.dump fir.spike.out > fir.spike.trace
[danand3@grendel46 rocket_sim]$
```

Software:

```
170881 ns tlm2conn received exit signal

Info: /OSCI/SystemC: Simulation stopped by user.
Simulation time: 170881 ns
Wall clock time: 0 seconds

real    0m0.481s
user    0m0.274s
sys     0m0.036s
stty sane
trace -f spike -o fir.riscv.dump fir.spike.out > fir.spike.trace
```