FPGA Based Audio DSP Coprocessor for TinyRV2 CPU

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University in Partial Fulfillment of the Requirements for the Degree of Master of Engineering, Electrical and Computer Engineering

Submitted

by

Han Mo

MEng Advisor: Van Hunter Adams

Degree Date: May 2025

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: FPGA Based Audio DSP Coprocessor for TinyRV2 CPU

Author: Han Mo

Abstract: This project focuses on extending a five-stage RISC-V pipelined processor with specialized Audio Digital Signal Processing (DSP) instruction sets and dedicated Audio DSP coprocessor on an FPGA platform. The primary objective of this project is to enhance the computational efficiency of Audio DSP tasks, particularly operations such as Multiply-Accumulate (MAC) and Infinite Impulse Response (IIR) Filtering. By integrating custom DSP instruction sets and designing specialized Audio DSP coprocessors, the project aims to address the limitations of general-purpose RISC-V processors in performing Audio DSP workloads. The implementation is justified using common DSP applications like voice vocoders, tested on the DE1-SoC FPGA board. The findings emphasize the scalability and flexibility of RISC-V architecture enhanced with Audio DSP coprocessors, contributing to efficient and cost-effective Audio DSP solutions suitable for real-time and resource-constrained applications.

1. Executive Summary

This project focuses on extending a five-stage pipelined RISC-V processor with Audio DSP capabilities and hardware coprocessor, implemented on the DE1-SoC FPGA platform. In the real world, Audio DSP operations are computationally demanding in applications ranging from voice processing to audio/video analysis. These operations require specialized coprocessors to achieve real-time performance. General-purpose CPUs often struggle to meet the speed requirements for DSP workloads due to their sequential execution features and lack of specialized instruction sets and relevant coprocessors. The motivation for this project arises from the requirement to improve computational efficiency for Audio DSP workload. In this project, FPGAs serve as an ideal verification platform due to their reconfigurability, parallelism, and ability to rapidly test designs in hardware. The implementation was tested on the FPGA using Audio DSP benchmark, which was voice vocoding, demonstrating significant improvements in audio DSP workload execution.

During the Fall 2024 semester, I focused primarily on familiarizing myself with the use of the DE1-SoC FPGA board and laying the groundwork for a subsequent implementation of a coprocessor for Audio DSP. I began by completing the Lorenz System project, which solves and visualizes the Lorenz attractor equation in real-time by implementing a hardware ODE solver on an FPGA. This not only helped me to consolidate the basic skills of Verilog programming and hardware module connection but also helped me to master the basic flow of audio playback and signal visualization. In addition, I built the basic architecture of an audio DSP coprocessor and implemented and tested a voice vocoder with an IIR filter on an FPGA. With the created IIR filter, this voice vocoder could successfully translate a human-voice input into a robotic output, laying the foundation for the development of subsequent DSP modules.

Moving into the Spring 2025 semester, I shifted my focus to implementing a RISC-V processor on the DE1-SoC platform and integrating the audio DSP coprocessor with the RISC-V CPU. During this phase, I designed and added custom audio DSP instruction extensions, as well as built a complete on-chip memory system, including instruction SRAM, data SRAM, source/sink SRAM, and a memory-mapped interface via HPS. I also added a handshaking protocol to the original five-stage pipelined RISC-V processor architecture (based on TinyRV2) to realize the interaction between the CPU and the coprocessor. In the end, the modified processor and on-chip memory module successfully passed all simulation tests, verifying the correctness of the behavioral logic with waveform graphs and assertions. After that, the whole system was synthesized and implemented on the FPGA platform for hardware validation. Several general-purpose workloads such as vector addition and ascending order sorting and voice vocoder configurations with varying IIR filter tap lengths, center frequencies, and pitch shift parameters are utilized for the

FPGA onboard whole system testing. The whole system uses architecture which enables the HPS interfaced instruction and data loading mechanism, allowing the dynamic workload execution without regenerating the FPGA bitstream. This architecture establishes an efficient framework for rapid hardware testing and validation of the complete integrated system. According to the testing results, the FPGA was confirmed to function correctly when integrated with the TinyRV2 CPU, successfully executing voice vocoding operations under control of the customized IIR instructions incorporated into the baseline RISC-V processor architecture.

2. Introduction

Digital Signal Processing (DSP) is a fundamental technology used to manipulate signals in the digital domain. It begins with converting the analog signals into digital signals for ADC conversion. The digital signals are then processed and analyzed using specific algorithms. After that, the processed data will be converted back to analog signals through DAC. This process allows accurate and real-time processing of audio, video, and other voice-based data. As shown in Figure 1, an Audio DSP system operates in the digital domain, connecting an analog input signal from a microphone or sensor with a modified analog output to a loudspeaker or actuator. Nowadays, Audio DSPs have become indispensable, with a wide range of applications:

- Audio processing: audio loopback, noise reduction and real-time speech analyzing in mobile devices.
- Speech recognition: providing support for voice assistants such as Siri.

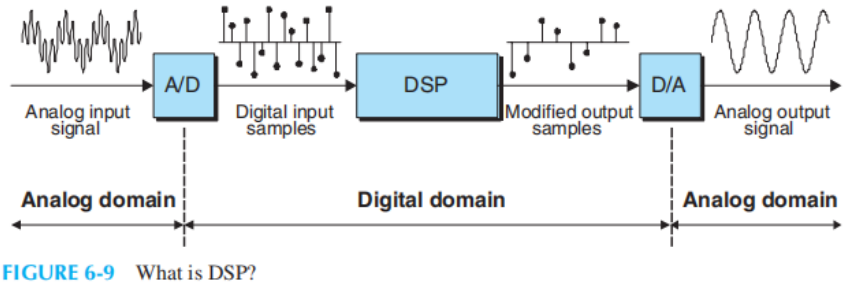


Figure 1. Digital signal processing procedure [1].

Audio DSP algorithms such as infinite impulse response (IIR) filtering, fast Fourier transform (FFT) and multiply-accumulate (MAC) operations require high computational intensity and parallel processing. Traditional general-purpose central processing unit (CPU) is not suitable for these tasks because:

- Sequential execution: CPUs process tasks sequentially, leading to great latency which is not acceptable in real-time signal processing workloads.

- Inefficient instruction sets: CPUs lack specialized instructions for Audio DSP algorithms. Executing DSP algorithms through the basic mathematical algorithms makes the system inefficient.
- No specialized accelerator: CPUs lack the specialized coprocessor that can efficiently execute the DSP workload.

To cope with the efficiency bottleneck of traditional CPUs when dealing with Audio DSP workloads, the industry has also explored some solutions:

- Optimized Software Library Supports DSP Functions for Processors:

One way is to support DSP functions through software libraries. For example, the ARM Cortex-M series (e.g., Cortex M0) can rely on software libraries optimized for the instruction set to implement DSP operations. The advantage of this approach is that it can run directly on existing processor architectures without additional hardware support. However, the problem is obvious: without a dedicated hardware acceleration module, this approach has obvious limitations in efficiency and real-time, making it difficult to meet the high latency and throughput requirements of application scenarios.

- Dedicated DSP Processors:

Another approach is to use a dedicated DSP processor, such as Analog Devices' SHARC series. These processors have been optimized at the architectural level for DSP applications, such as using the VLIW (Very Long Instruction Word) architecture, which greatly improves processing power. However, this type of solution usually means a higher development threshold - from compilers, debuggers to software libraries all need to be customized, which not only has a long development cycle, but also lacks in versatility and flexibility.

Given these limitations, a suggested solution is to offload Audio DSP tasks to specialized hardware integrated into processors. Processors with DSP instruction extensions offer the necessary parallelism, flexibility, and efficiency to accelerate Audio DSP operations, making them ideal for real-time and energy-efficient applications.

2.1 RISC-V Architecture for Audio DSP Applications

In this project, the RISC-V architecture is chosen to develop a system with enhanced DSP capabilities. Because RISC-V is an open-source and free Instruction Set Architecture (ISA), which allows modular design and easy customization. However, standard RISC-V processors lack support for common Audio DSP tasks, such as Multiply-Accumulate (MAC), Fast Fourier Transform (FFT), and Infinite Impulse Response (IIR) filtering. The absence of these critical instructions and hardware accelerators prevents RISC-V processors from achieving highly efficient performance in Audio DSP workloads. To

overcome these limitations, this project aims to enhance the RISC-V processor through the following innovations:

- Custom Audio DSP Instruction Extensions: Adding specialized instructions to offload DSP operations to the coprocessor.
- Dedicated Audio DSP Hardware Coprocessor: Offloading computationally intensive tasks to optimized hardware coprocessor.

Prior research has demonstrated that RISC-V is an ideal candidate for heterogeneous computing platforms. Gautschi et al. [2] proposed a near-threshold energy-efficient RISC-V core with DSP extensions tailored for IoT endpoints, showing measurable gains in throughput and power reduction.

Therefore, by integrating the specialized accelerator, this project will unlock the potential of RISC-V processors for real-time, high-performance DSP applications. The constructed processor with customized hardware will be implemented and validated on an FPGA platform to demonstrate the functionality and effectiveness of this approach. In the design of this project, I combine and configure multiple M10K memory blocks to construct the following four types of the

2.2 Memory Interface Design with M10K Blocks

The RISC-V processor implemented in this project uses the FPGA M10K memory blocks to build an efficient memory system. These configurable memory blocks are embedded in the Cyclone V FPGA. Each of them provides 10Kb independent storage and is suitable for implementing high-speed and low-latency local memory access. In this project, I combine and configure several M10K memory blocks, constructing the following different types of functional memories.

- Instruction/Data Memory: used to store the RISC-V instruction and data set
- Source/Sink Memory: used to store the data going into or outside of the processor
- Tap/Frequency/Pitch Memory: used to store parameters for the Audio DSP coprocessor including the tap number of the IIR, the center frequency of IIR and the pitch shift parameters for the voice vocoder.

2.3 Audio DSP Coprocessor Interface

The interface between the RISC-V processor and the Audio DSP coprocessor is critical to the whole system. The Audio DSP coprocessing interface is based on the request and response protocol and FIFO buffering. The processor sends a request signal to initiate the

customized instruction on the audio DSP coprocessor related to audio DSP processing. After the audio DSP coprocessor initiates the relevant audio operation, the DSP coprocessor returns a completion signal to the RISC-V processor and waits for new audio instructions to come in. During the request and response process, the valid and ready handshaking protocols are utilized to ensure the correct functionality of both sender and receiver. This protocol is also used with the RISC-V processor to receive parameters such as the IIR segment number, IIR center frequency, and pitch offset number. Besides this interface, the Audio DSP coprocessor has another interface with the FIFO input and output. These two FIFO buffers are utilized to contemporarily store the sampled or processed for smooth output from the ADC or into the DAC. These interfaces were specifically designed for real-time audio processing, ensuring predictable timing characteristics essential for stable audio output.

2.4 Voice Vocoder on Audio DSP Coprocessor

To demonstrate the functionality of the RISC-V processor with custom DSP instructions and specialized hardware accelerators, this project is expected to implement a benchmark application on the DE1-Soc FPGA: Voice vocoder. Served as real DSP workloads, this benchmark is used to validate the real time performance of the Audio DSP Coprocessor.

The voice vocoder benchmark, as shown in Figure 2, is a signal processing application that separates audio signals into multiple frequency bands using bandpass filters. After filtering, the signals are rectified and passed through low-pass filters (LPF) to extract their amplitude envelopes. These envelopes are then used to modulate a carrier signal, and the modulated signals are summed to reconstruct the output voice. The voice vocoder is designed to modulate the input voice and generate the characteristic robotic sound. The main computational workload comes from IIR filtering and multiply-accumulate (MAC) operations, both of which are computationally intensive. To satisfy the real-time processing requirements, these operations are offloaded to a hardware coprocessor implemented on the FPGA board. This strategy is expected to significantly reduce system latency and enable efficient real-time audio processing.

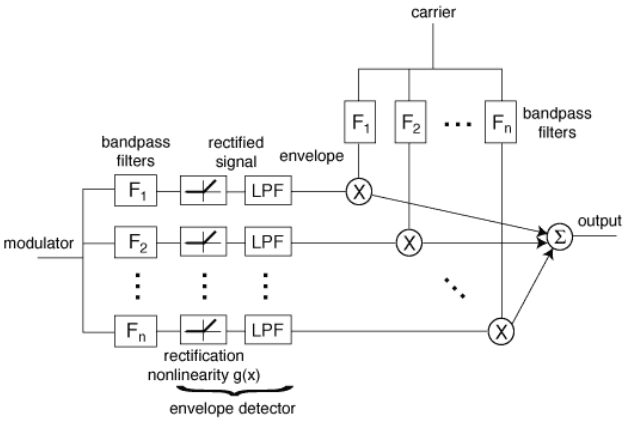


Figure 2. Channel voice vocoder [3].

2.5 FPGA Implementation Platform

As mentioned above, the FPGA DE1-Soc board is chosen as the platform for the implementation of the constructed RISC-V processor extended with DSP instructions and relevant accelerators. The adoption of the DE1-Soc board is related to the parallel computation capabilities, custom hardware acceleration and rich peripheral interfaces of the FPGA. DE1-SoC provides high flexibility for the custom design and test of the RISC-V processor. The rich peripherals on the board which include the microphone, speaker, and VGA display supports the implementation and testing of DSP benchmarks such as voice vocoder and signal waveform visualization.

2.6 Project Objectives

This project aims to extend the functionality of a RISC-V processor by integrating custom Audio DSP instructions and a specialized hardware coprocessor to enhance its performance for real-time audio signal processing tasks. The whole system will be implemented and validated on the DE1-SoC FPGA platform using benchmarks such as IIR filter and voice vocoder. The outcomes are expected to show the real-time processing capabilities of the whole system for the Audio DSP workloads, indicating the suitability of the audio DSP coprocessor for resource-constrained scenarios. Besides this, the construction of the FPGA based DSP coprocessor system is also expected to deepen my understanding of the RISC-V architecture, DSP algorithms, and FPGA hardware design, establishing a strong foundation for future work in hardware design and digital verification.

3. Implementation

The implementation section of this project involves the design, execution, and validation of the RISC-V processor with Audio DSP instruction extensions and dedicated hardware Audio DSP coprocessor on the DE1-SoC FPGA platform. The FPGA was used as the target platform to achieve real-time digital signal processing tasks, including IIR frequency filtering and pitch shifting.

3.1 DE1-Soc FPGA Platform

The DE1-SoC FPGA board, as shown in Figure 4, is chosen for its flexibility, rich peripheral and integration of both FPGA and ARM Cortex-A9 processors. This FPGA board is based on the Cyclone V FPGA from Intel and supports parallel processing, which is critical for executing computationally intensive Audio DSP tasks. This board contains the audio codec and the microphone input and line-out ports which enable real-time audio input and output for the voice vocoder. The hybrid nature of the board combining FPGA and HPS provides the flexibility needed to implement and test the RISC-V processor while integrating Audio DSP coprocessor. Moreover, the DE1-SoC FPGA board includes a lot of customized M10K memory blocks which can be used for efficient on-chip memory implementation. The AXI bus interface of the FPGA also enables direct memory access from both HPS and FPGA side. This architecture provides the flexibility required for the implementation and testing of the RISC-V processor with the Audio DSP Coprocessor.

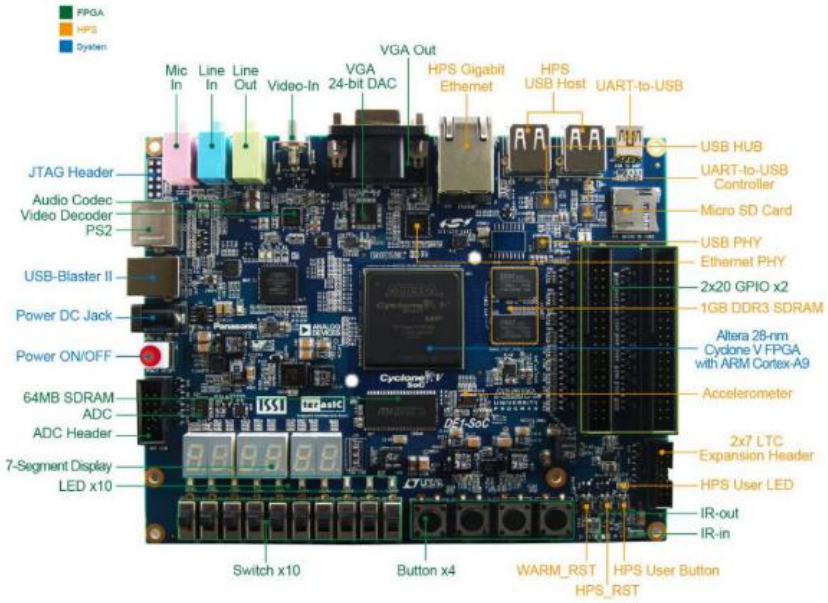


Figure 3. DE1-SoC development board [4].

3.2 Overall System Structure

To fully utilize the capabilities of the DE1-SoC FPGA, an entire system of RISC-V processor with Audio DSP instruction extensions and dedicated Audio DSP Coprocessor were constructed on the board, including the memory network implemented based on the M10K unit on the FPGA board. To construct the whole system, the implementation workflow is as follows:

1. **Instruction Storage and Execution**: Instructions for Audio DSP workloads are stored in the SRAM within the FPGA. The RISC-V processor fetches and decodes instructions sequentially to execute relevant operations.

2. **Fetch and Execute Instructions through the Processor**: The processor fetches instructions from SRAM, decodes them, and executes the required operations. Custom Audio DSP instructions trigger the corresponding hardware coprocessor for computationally intensive audio tasks.

3. **Get the Audio Input**: Real-time audio signals are captured through the microphone input of the DE1-SoC board. The analog inputs are digitized using the on-board audio Codec.

4. **Execute the Voice Vocoder when Reaching Relevant Instructions**: When DSP instructions for the voice vocoder are detected, the processor offloads the tasks to the hardware coprocessor. The processed signal is reconstructed and output through the line-out port to produce the modified audio (e.g., robotic voice).

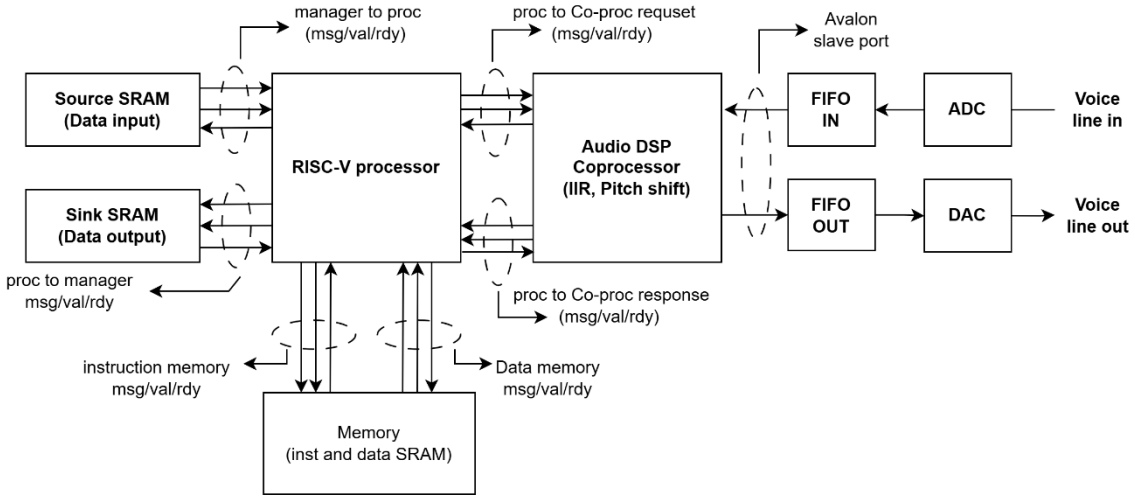


Figure 4. Structure of the whole system.

Figure 5 above shows the detailed structure of the whole system. According to Figure 4, the main implementation can be divided into three parts: the RSIC-V processor, on-chip SRAM blocks and Audio DSP coprocessor. These three submodules are tightly coupled through

dedicated latency insensitive interfaces and handshaking communication protocols to form a complete audio processing system.

According to the implementation workflow listed above, the first step of this project focuses on gaining a solid understanding of FPGA concepts and Verilog programming. Therefore, the Lorenz system project was implemented, involving solving the Lorenz attractor equations using a hardware ODE solver and visualizing the results on the FPGA in real time. These serve as the foundation for implementing real-time audio signal processing and developing skills in signal visualization. After that, the voice vocoder was implemented on the FPGA to figure out the structure required for the accelerator on the FPGA and prepare for the integration with the general-purpose RISC-V processor. In the next step, the on-chip SRAM blocks were implemented and configured to ensure the correct loading and manipulating of the dataset. Finally, these three submodules are integrated together on the FPGA board to do the whole system testing and verification.

3.3 Voice Vocoder Implementation

To implement the voice vocoder on the DE1-SoC FPGA, a second-order Infinite Impulse Response (IIR) filter is used as the fundamental building block. The equation used to achieve the second order IIR filter is defined as:

$$y(n) = b_1 \cdot x(n) + b_2 \cdot x(n-1) + b_3 \cdot x(n-2) - a_2 \cdot y(n-1) - a_3 \cdot y(n-2)$$

IIR filter is one type of digital filter which uses both current and past input samples as well as past output samples to compute the current output samples, making them recursive. Their structure consists of feed forward coefficients applied to inputs and feedback coefficients applied to previous outputs as indicated in the equation above. This IIR filter processes the input voice signal across multiple frequency bands to separate signals required for each frequency band, forming the basis of the vocoder system.

The voice vocoder is constructed based on the classic channel vocoder architecture. Prior works such as Rabiner and Schafer [5] provide foundational theory on speech processing and filter bank vocoding, which guide the envelope detection and pitch modulation process. Figure 5 shows the complete structure of the voice vocoder. The voice vocoder system processes an audio signal to extract frequency-specific envelopes and reconstruct the modulated output signals for both left and right audio channels. The structure, as shown in Figure 5, consists of the following four important components.

The first component is the input audio signal processing. The input audio is passed through two parallel pipelines for left and right channels. Each pipeline contains 32 IIR filters that transfer the input signal into 32 frequency bands. These filters operate independently and

are implemented using second-order IIR filter. After that, each IIR bandpass filter extracts a specific frequency range from the signal input. This decomposition allows the system to analyze the spectral characteristics of the input signal.

The third component is the envelope detection through the low pass filtering. The outputs of the IIR bandpass filters are rectified and smoothed using lowpass filters. This step ensures that the amplitude envelopes of the band-specific signals are obtained efficiently. Then the envelope signals are multiplied by sinusoidal carriers for reconstruction. Pitch shift parameters are set and adjusted to get the most robotic like voice when performing the voice vocoder. In the end, the modulated signals from all 32 bands are summed to form the final left and right channel outputs. These outputs are then sent to external interfaces such as DAC for audio output.

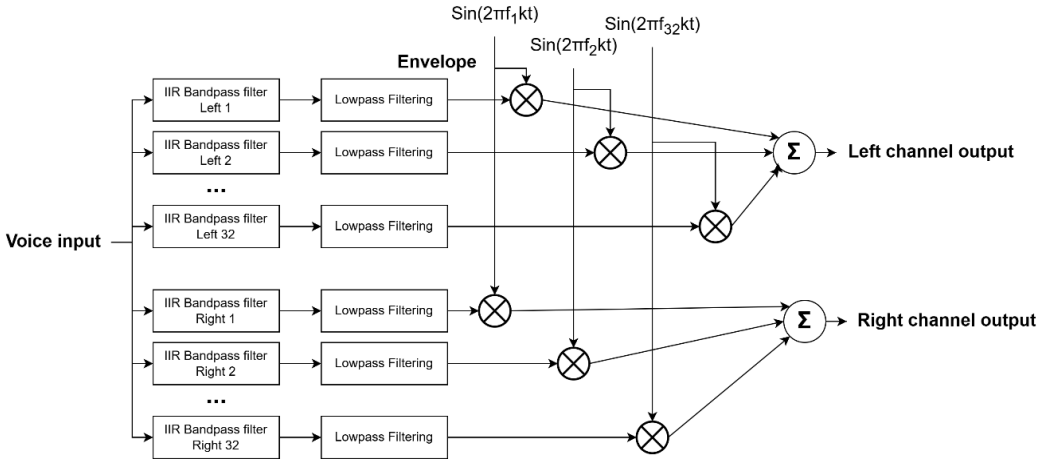


Figure 5. Structure of voice vocoder.

During the voice vocoder implementation, the 32 IIR bandpass filters for both channels were implemented in parallel to maximize throughput. Therefore, each IIR filter used DSP slices for multiply-accumulate (MAC) operations to improve computation efficiency. The sinusoidal carriers were precomputed and stored in Lookup Tables (LUTs). To maintain synchronization across the filters, rectifiers, and modulation stages, the design incorporated clock-driven pipelines with precise control logic.

3.4 RISC-V Processor Implementation

After constructing the voice vocoder, the RISC-V processor should also be designed and implemented. As shown in Figure 6, this processor is built based on the materials from the ECE 4750 lab2. I have revised the processor code in the ECE 4750 to make it synthesizable and developed the memory system and interfaces that can be used to interact with the Audio DSP Coprocessor [6]. The processor contains five stages: Fetch, Decode, Execute, Memory, and Writeback. These stages operate in parallel to enable the pipeline of the system. And this processor also includes stalling and squashing techniques to resolve hazards problems.

The processor uses the RISC-V instruction set architecture (ISA), specifically the Tiny RISC-V ISA, which enables the execution of simple workloads through C program [7].

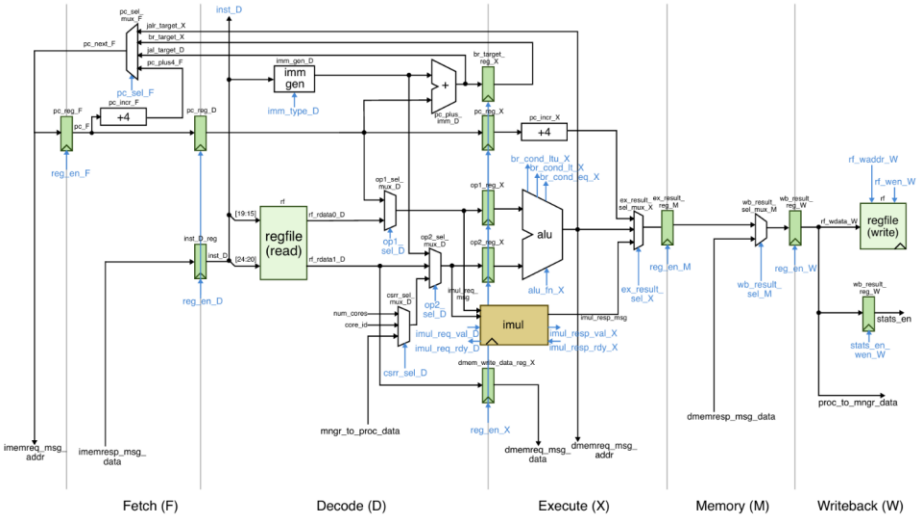


Figure 6. Five stage pipelined processor with stalling and bypassing logics [6].

3.4 Audio DSP Coprocessor Interface

The interface between the RISC-V processor and the Audio DSP coprocessor was carefully designed to ensure reliable and efficient communication. Extra control logic is implemented in the RISC-V processor to manage the data flow and coordinate the operation between different processors. As shown in Figure 4 above, the request and response protocol are utilized to synchronize the operations between the processor and the coprocessor through handshaking mechanism. This req/resp protocol is also implemented to ensure the correct configuration of the data transferring including the filter coefficients, the center frequency and the pitch shift values. When customized IIR instructions are decoded by the RISC-V processor, the controller is triggered to open the interface and sends the corresponding parameters to the coprocessor through the handshaking protocol. Besides the interface with the processor, the Audio DSP coprocessor has another interface with the input and output FIFO modules, which are utilized to buffering the input and output data, coordinating the speed difference between the ADC/DAC and coprocessor.

3.5 Custom Instruction Set Extensions

To enable communication with the Audio DSP coprocessor, the standard RISC-V processor needs customized instruction sets, which involve:

1. Custom Opcode: Defining new instruction formats within the RISC-V ISA framework for Specific Audio DSP operations

2. Modify Instruction Decoder: Modifying the decode stage in the RISC-V processor to recognize custom DSP instructions
3. Generate Control Signal: Creating specialized control signals for Audio DSP coprocessor integration.

Recent academic research has provided a comprehensive overview of RISC-V instruction set extensions. Cui et al. point out that the modular architecture of RISC-V is well suited for the integration of DSP-oriented extensions, with a particular emphasis on its potential for customized application areas such as audio processing [8]. This trend is consistent with the goal of this project, which is to enable audio DSP performance improvements through custom instructions and coprocessors.

In this project, I implemented custom instruction for audio IIR filtering by reusing the existing CSRW and CSRR instructions from the RISC-V ISA. Rather than defining new opcodes, I utilized the standard CSR to associate several IIR-related operations with a range of reserved control status registers from address 0x7E0 to 0x7FF. This idea is inspired by the Cornell ECE6745 course about the TinyRV2 Accelerator RTL design, in which those CSR instructions are customized to control several status registers from a piece of address to enable parameters transmission to the Audio DSP coprocessor [7].

```
#void bubble_sort( int *array, int size ) {
#    for ( int i = 0; i < size-1; i++ )
#      for ( int j = 0; j < size-i-1; j++ )
#        if ( array[j] > array[j+1] )
#          swap( array[j], array[j+1] );
csrr  x1, mngr2proc < 10         # Read array lengtgh
csrr  x2, mngr2proc < 0x2000     # Read array base address
addi  x5, x1, -1
outer_loop:
    beq   x5, x0, done_sort
    add   x6, x5, x0
    add   x7, x0, x0
inner_loop:
    beq   x6, x0, finish_inner
    slli  x8, x7, 2
    add   x8, x2, x8
    lw    x9, 0(x8)
    addi  x10, x8, 4
    lw    x10, 0(x10)
    slt   x11, x10, x9
    beq   x11, x0, no_swap
    sw    x10, 0(x8)
    addi  x12, x8, 4
    sw    x9, 0(x12)
```

```
no_swap:
    addi  x7, x7, 1
    addi  x6, x6, -1
    jal   x0, inner_loop

finish_inner:
    addi  x5, x5, -1
    jal   x0, outer_loop

done_sort:
    csrw  proc2mngr, x0 > 0     # Finish sorting
csrr  x3, mngr2proc < 32    # Get the tap size of Audio coprocessor
IIR  0x7e4, x3             # Send the tap size to the Audio coprocessor
IIR  0x7e0, x0             # Start Audio coprocessor
IIR  x0,    0x7e0          # Response success start of Audio coprocessor
```

Figure 7. Assembly for System to Adds Arrays and Start Audio DSP Coprocessing.

The figure above shows both general-purpose workloads (e.g., bubble sort) and customized IIR instructions executed using TinyRV2 assembly. TinyRV2 is a teaching-oriented subset based on the official RISC-V 32-bit instruction set architecture, which is standardized and maintained by RISC-V International [9]. As shown in Figure 7 above, when the IIR 0x7E4, x3 is executed, the processor reads data from general purpose register x3 and writes the value to the IIR coprocessor's input register via the internal xcel.req interface. The coprocessor then starts the filtering operation based on the input parameters and writes the signal for the start of the filtering to another register address (e.g. 0x7E0). By executing

the IIR x0, 0x7E0 instruction, the RISC-V processor can read from the coprocessor the signal that filtering has started.

Although the form is still csrw/csrr in assembly, I regard these operations as customized IIR instructions in terms of functional semantics, forming a clear interface decoupled from the hardware coprocessor, which facilitates instruction scheduling and module reuse. At the same time, this mechanism does not require modification of the processor's decoder and only needs to add support in the CSR parsing logic and coprocessor control channel, which is highly portable and scalable.

3.6 On-chip SRAM Implementation

To implement the RISV-processor and the Audio DSP coprocessor on the FPGA, the synthesizable memory system becomes necessary. The memory system was needed to store instructions, data, and Audio DSP parameters while also enabling data flow through handshaking interfaces and ready/response protocols. The DE1-SoC FPGA board used in this project provides on-chip SRAM configuration through configurable M10K memory blocks. These memory blocks offer several advantages:

- **Dual Port**: Enable M10K memory can be accessed from both FPGA and HPS side.
- **Simplified Programming**: Direct memory initialization can be achieved through C code on the HPS side, simplifying the loading of assembly for RISC-V processor.
- **Flexible Size:** Data width and total memory size for each M10K block are configurable and programmable.
- **Selectable Latency:** Able to configure the read latency of the M10K block as 1 cycle or 2 cycle.

In this project, I utilized a total of eight M10K blocks distributed across the system. Four M10K blocks were dedicated to RISC-V processor memory functions, including instruction storage, data memory, sink and source buffers, while the remaining four blocks are used to store Audio DSP parameters such as IIR filter coefficients, center frequencies, and pitch shifting values. Each memory block was configured with specific dual-port settings and suitable data width. Before integrating the M10K memory unit into the processor, basic tests about the read and write latency of M10K memory were went through first. Though the M10K unit was configured to be 1 cycle late during read, the combinational reading behavior was observed when doing the test. However, the RISC-V processor implemented was using the protocol which assumes the memory latency is 1 cycle. Therefore, all the M10K unit used in this project is configured to have 2 cycle latency in the Qsys page of the Quartus, making the behavior aligns with the expectation of the processor. The figure below shows the Qsys layout for the M10K SRAM units.

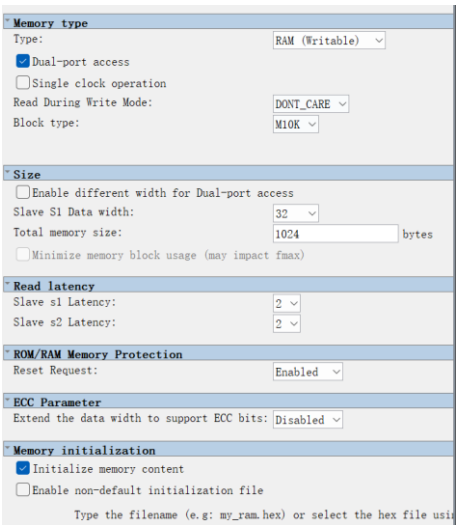


Figure 8. Qsys layout for the M10K SRAM units.

After configuring the M10K units, these memory units cannot be directly integrated into processor since the real memory system will not always synchronize with the processor. The Val/Rdy handshaking protocol is needed to make the memory system become latency insensitive when the RISC-V processor is in the process of stalling or squashing. To implement the delay-insensitive val/rdy interface, I adapted the SRAM wrapper concept from Professor Christopher Batten's SRAM tutorial. This approach creates a robust abstract layer between the processor and memory system, effectively managing timing differences and ensuring reliable data transfers regardless of pipeline stalls or memory access delays [10]. The structure of the SRAM wrapper is shown in figure below. The message will only be transmitted when both valid and ready signals are high. Queue is utilized in the wrapper to buffering the elements when stalling happens in the output side of the SRAM. By using this wrapper, the M10K unit works smoothly when integrated into the processor.

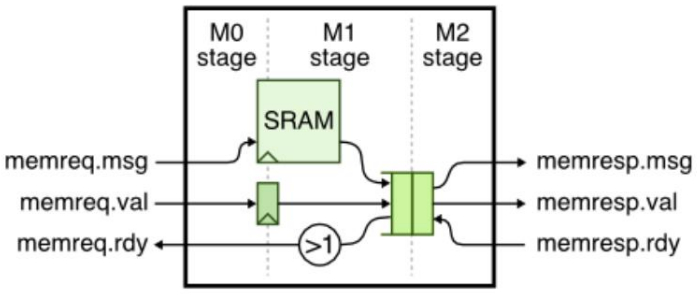


Figure 9. SRAM latency insensitive val/rdy wrapper.

3.8 HPS-FPGA Communication

After integrating on-chip SRAM blocks, I also addressed how to initialize SRAM contents. As discussed previously, the dual-port M10K blocks enable direct memory initialization

from the HPS through the C program. To support this, I established and tested the full HPS–FPGA communication flow. The specified C program was developed on the ARM HPS side to control RISC-V processor execution and load test programs. The system workflow when running this HPS-side C program is as follows:

- **Program Selection**: User selects which assembly program wants to be executed
- **Reset Sequence**: Reset the entire processor system
- **Memory Configuration**: The program initializes instruction memory, data memory, and source memory based on disassembled machine code.
- **Clock Start**: The program triggers a constant clock signal to the processor
- **IIR filter check**: Voice vocoder starts with pitch shifted voice output when the customized IIR instruction is executed.
- **Completion Detection**: Program checks the "done" signal to determine if the processor has finished execution.
- **Visualization**: Dumps all memory to verify contents and compares register values.

To ensure correct execution, address offsets are also applied during the process of the memory initialization. Instruction and data memory starts at 0x200 and 0x2000 respectively. However, the M10K blocks used for all the memory blocks in this project are addressed from 0x0. Therefore, all addresses got from the machine code hex file are adjusted by subtracting 0x200 for instruction memory and 0x2000 for data memory [11]. When the assembly program is selected, the system reads its corresponding hex file, which contains both machine instructions and the expected register outputs. The program then loads the values into the relevant M10K blocks from the ARM HPS side, using pointer arithmetic to account for address offsets on the shared AXI bus. Additionally, the machine code must be byte-reordered accordingly before written into memory since the processor follows a little-endian memory format.

Through careful integration of a RISC-V processor, audio DSP coprocessor, and memory system, the complete audio processing system is successfully built on a DE1-SoC FPGA platform. This system utilizes a customized IIR instruction extension set to achieve efficient communication between the audio processing coprocessor and the main processor. Through the val/rdy handshake protocol, the data transfer between system components is ensured to be reliable. This implementation not only validates the scalability and flexibility of the RISC-V architecture in the audio DSP filed but also demonstrates the effectiveness of FPGA-based hardware coprocessor schemes in real-time audio processing tasks. Next, I will show specific results of the system tests and evaluate the advantages of this scheme in real applications.

4. Result

Through the implementation of the RISC-V processor with Audio DSP coprocessor, several tests were conducted to verify the functionality of individual components and the integrated system. This section presents the results of these tests, demonstrating the successful implementation of the project and highlighting its performance characteristics. Through the implementation of the RISC-V processor with Audio DSP coprocessor, several tests were conducted to verify the functionality of individual modules and the whole integrated system. These tests include the simulation results obtained through VCD waveforms after running SystemVerilog testbenches, which provided functional verification prior to implementation; and actual hardware testing results from the complete system running on the DE1-SoC FPGA board, which validated the system's performance in a real-world environment. This section presents the results of these tests, demonstrating the successful implementation of the project.

4.1 SRAM Test

As mentioned in the implementation section, the M10K memory unit with a fixed latency of 1 cycle is used to store the source, sink, instruction, and data values of the processor. The original memory system of the processor needed to be reconstructed to make it synthesizable for FPGA implementation. To achieve this, I developed sink and source modules integrated with an interface that get the same fixed 1-cycle latency behavior as the M10K unit, allowing the use of SystemVerilog to accurately simulate and test the memory system behavior. For the source memory testing, I configured the test to verify the reading functionality. As shown in Figure 10, the source memory was initialized with 6 test values. These values were intended to be used for initializing the processor registers and were successfully read out in the correct sequence (0x10101010, 0x21212121, ..., 0x76767676). The VCD waveform show that each memory address was correctly accessed and the appropriate data was retrieved with the expected timing. When the last value was read, the done signal was asserted high, showing successful completion of the read operation as expected.

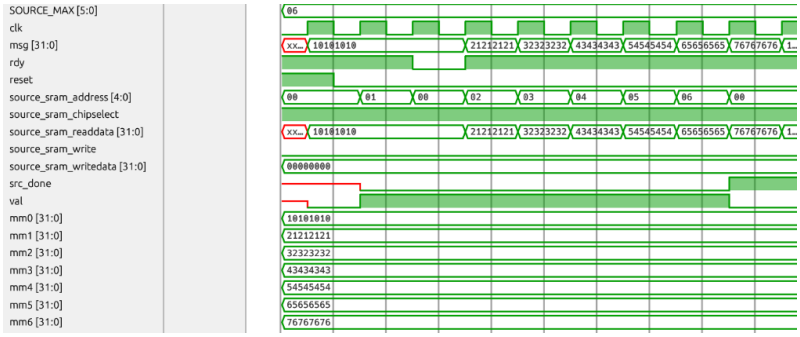


Figure 10. Source SRAM testing results.

For the sink memory testing, I configured the test to verify the writing functionality. As shown in Figure below, I performed write operations with 10 sequential values: 0x10101010, 0x21212121, ..., 0xa9a9a9a9. The waveform confirms that each value was correctly stored at its intended address with the proper timing. After the final value was written to the sink memory, the done signal was asserted high, confirming successful completion of the write operation sequence as expected.

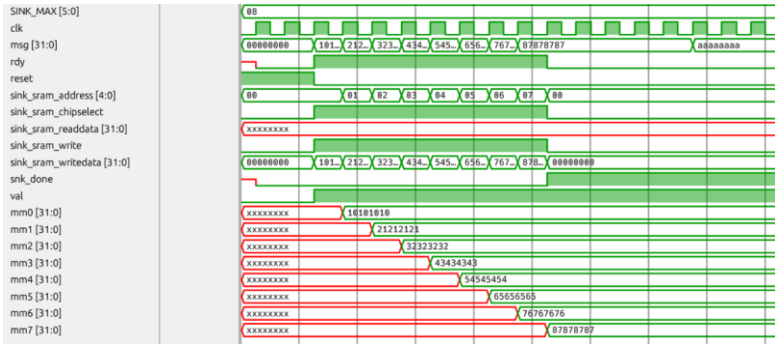


Figure 11. Sink SRAM testing results.

In addition to sink and source, the SRAM blocks store instructions and data. This requires a wrapper for the internal SRAM to support the latency insensitive interface. The correct operation of these SRAM wrappers for operations such as instruction fetches and data access will be verified at the RISC-V processor level, making sure that the SRAM modules are being utilized correctly in the system.

4.2 RISC-V Processor

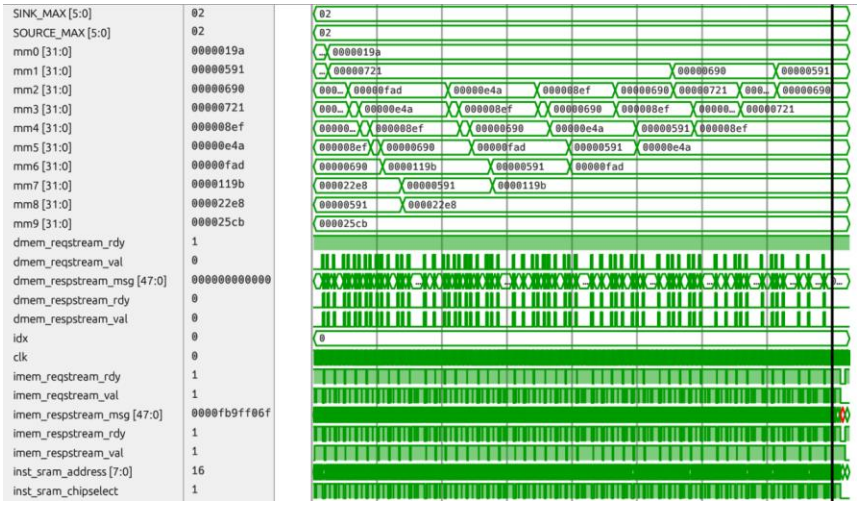
As mentioned above, the baseline RISC-V processor was built based on materials from the ECE 4750 lab2. To implement this processor on the DE1-Soc FPGA platform, the original non-synthesized memory system was replaced by the synthesizable SRAM modules with the relevant wrapper. This wrapper was necessary for ensuring the latency-insensitive behavior of the system in memory interactions. The figure below shows the waveform from RISC-V processor testing with a sorting algorithm after successful integration with the synthesizable SRAM components (source, sink, instruction, and data memory). The assembly code shown in Figure 7 was utilized as the processor testing benchmark. In this test, the bubble sort algorithm was executed on an array which was initialized with the following values: 0x721, 0x19a, 0x119b, 0xfad, 0xe4a, 0x8ef, 0x690, 0x22e8, 0x591, and 0x25cb. The waveform in the figure below demonstrates the operation of the processor during the sorting execution. When the sorting instructions were completed, the data was dumped from the data SRAM. The dumped data indicated that the original values had been successfully sorted in ascending order. The successful completion of the sort operation indicates that:

- The RISC-V processor with pipelined stages functioned properly
- The control logic for branch instructions executed properly
- The ALU operations were executed properly
- The memory access operations worked properly with the synthesizable SRAM

Besides the sorting algorithm, additional workloads were tested to further prove the correct functionality of the constructed RISC-V processor. The processor is proven to correctly do the addition of two arrays:

  - First array: 168, 162, 114, 101, 145
  - Second array: 282, 984, 470, 145, 200

The expected sums were found in the SRAM-dumped results, verifying the arithmetic abilities of the processor. In addition to all this, the RISC-V processor was tested under an even more complex control flow with multiple branches and memory access, and all passed as well. These tests have shown the successful integrations of the RISC-V processor with the synthesizable memories system with the expected outputs under different workloads. With these observations all the processor components such as instruction fetch, decode, execute, memory, and writeback stage are implemented properly and the memory interface would work with the custom SRAM wrapper.



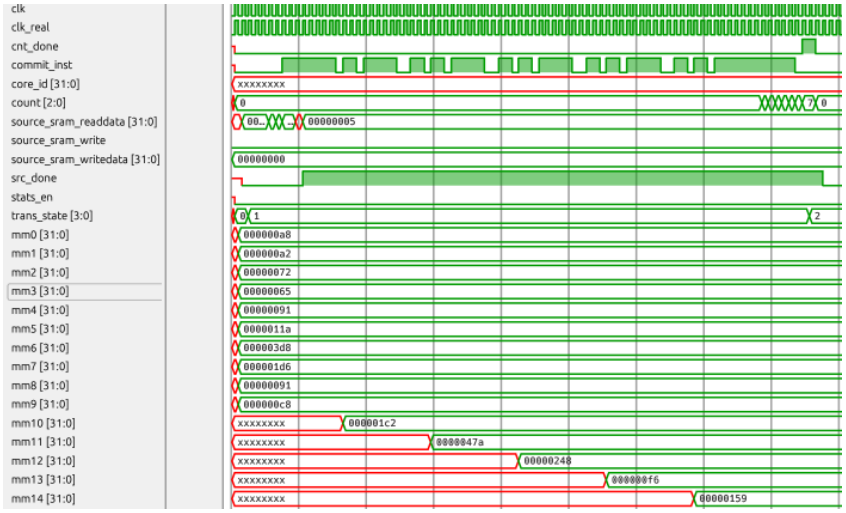


Figure 12. RISC Processor testing results of sorting (upper) and vector addition (lower).

4.3 Audio DSP Coprocessor Filtering

As mentioned in the implementation session above, the Audio DSP coprocessor included 32 bandpass IIR filters to effectively separate the audio input signal into distinct frequency bands for voice vocoding. The figure below shows the frequency response characteristics of these filters in the human voice range from 300 Hz to 3500 Hz.

The filter bank was designed using MATLAB. In the MATLAB, the second-order IIR filters were created with carefully selected parameters. Rather than using a linear frequency distribution, the Mel scale is used to get the distributed center frequencies of the filter, which better approximates human auditory perception.

Besides this, each bandpass filter was implemented using a second-order Butterworth filter with calculated cutoff frequencies. The diagram below indicated that each filter has a sharp peak at its center frequency with appropriate roll-off characteristics. This roll-off characteristics ensures effective separation of the frequency components. The series of IIR filters show good selectivity while maintaining sufficient overlap to prevent gaps in the frequency coverage. In theory and in implementation, the correct coefficients and center frequencies have been successfully determined. The IIR simulation shows the correct IIR filtering behaviors across all 32 channels. Since it is difficult to visualize the filtered and pitch-shifted voice in images, the actual audio results of the IIR filter implementation, including the filtered and pitch-shifted voice output are included in the demonstration video available on my YouTube channel. The successful implementation of these filters provides the foundation for the voice vocoder functionality, enabling the separation of audio signals into frequency bands that can be independently processed and recombined to create the characteristic "robotic voice" effect.

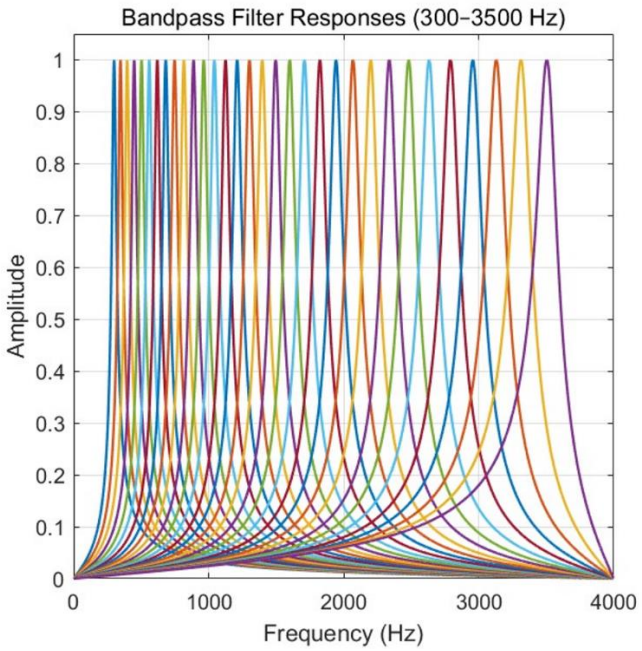


Figure 13. Frequency response for 32 IIR filters in frequency range: 300 ~ 3500 Hz.

4.4 Integrated System Simulation

After the separate verification of the RISC-V processor and the Audio DSP coprocessor, an integrated system simulation to verify the communication between the processing elements was realized. The validation of the interaction is specifically concentrated on the accelerator protocol (xcel protocol) realization. This protocol is necessary to convey parameters to the Audio DSP coprocessor from the RISC-V processor for the execution of decoded IIR DSP custom instructions. The simulation waveforms of the combined system are presented in Figure below that demonstrates the xcel_reqstream_msg used for the communication between the processor and the coprocessor.

The simulation demonstrates the execution of both general-purpose instructions and the specialized IIR instructions implemented. The xcel protocol successfully transmitted the IIR filter tap number as a parameter to the Audio DSP coprocessor, as shown in the xcel_reqstream_msg signal. This parameter defines the number of filter coefficients used in the IIR filter implementation, which directly affects the filtering characteristics. The waveform shows the proper implementation of the handshaking mechanism through the xcel_reqstream_val and xcel_reqstream_rdy signals. When the processor executes a custom IIR instruction, these signals coordinate the data transfer, ensuring reliable communication between the components. This integrated simulation verified that the custom instruction mechanism worked as designed, with the processor correctly identifying the IIR instructions and initiating the appropriate communication with the coprocessor. The successful parameter transmission and protocol operation confirmed that

the hardware accelerator interface was properly implemented, providing a solid foundation for the physical implementation on the FPGA.

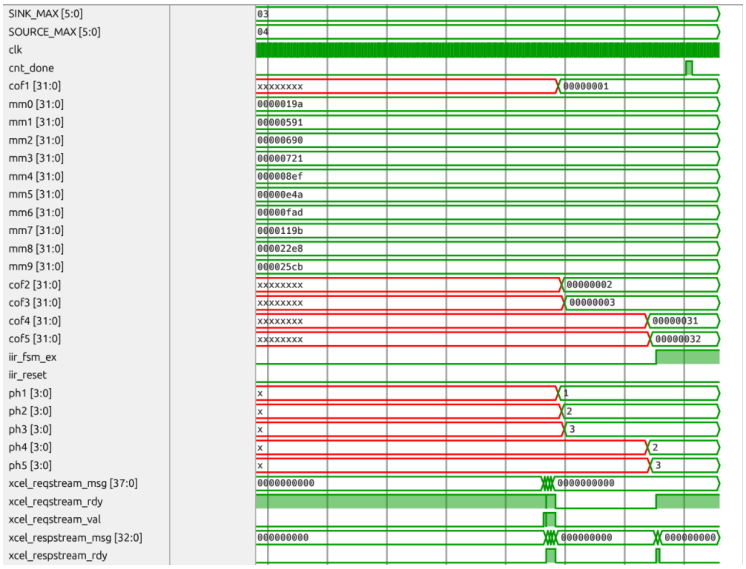


Figure 14. System simulation for sorting algorithms and customized IIR execution.

4.5 Integrated System Testing on FPGA

After simulating successfully, the whole integrated system has been implemented and tested on DE1-SoC FPGA platform to prove its performance on actual hardware. As discussed in HPS-FPGA Communication, I have written a separate C program on the ARM HPS side, to manage the execution of the RISC-V processor, and to load the test programs, allowing for the testing conditions to be easily adjusted without having to reconfigure the FPGA. The visualization of the memory dump after running the C code and successful HPS-FPGA communication is shown in Figure 15. This test included both a general-purpose sort and a specialized audio algorithm (robotic voice effect using IIR filter with pitch shift). The assembly code for this test was already presented in figure 7. This figures shows how the system was programmed to make two things at the same time: sort elements and initialize voice vocoder mode.

1. **Sorting Operation**: The dumped value from the data SRAM has indicated that the whole array has been correctly sorted in ascending order. Therefore, the RISC-V processor is confirmed to be correctly executed the sorting algorithm on the hardware.

2. **Voice Vocoder Initialization**: The system successfully initiated the voice vocoder with the specified parameters (tap number 32, pitch shift 3).

Due to the difficulty of visualizing audio processing results in static images, a detailed

demonstration of the system's operation, including audio input and output, is included in the demonstration video referenced in the Appendix. This demonstration provides audible confirmation of the voice vocoder's functionality, transforming normal speech into the characteristic "robotic voice" effect with the specified pitch shifting parameter of 3. The successful hardware testing confirmed that the system design was correctly synthesized and implemented on the FPGA, validating all aspects of the integrated RISC-V processor with Audio DSP coprocessor system.

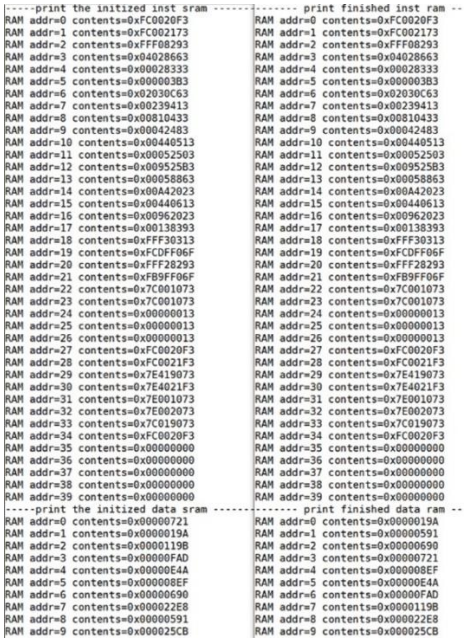


Figure 15. Dumped SRAM result after the sorting algorithm execution on FPGA.

4.6 FPGA Resource Utilization

The resource utilization of the FPGA should also be considered for the implementation of the RISC-V processor with an Audio DSP coprocessor to ensure efficient hardware usage. The figure below shows the detailed resource utilization report generated by the Quartus.

Timing Models	Final
Logic utilization (in ALMs)	21,858 / 32,070 (68 %)
Total registers	20911
Total pins	368 / 457 (81 %)
Total virtual pins	0
Total block memory bits	296,464 / 4,065,280 (7 %)
Total DSP Blocks	66 / 87 (76 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	2 / 6 (33 %)
Total DLLs	1 / 4 (25 %)

Figure 16. Resource utilization report.

According to the figure above, the mostly used resources were:

1. Pins (81% usage): This high pin utilization is due to the complex internal connections between the RISC-V processor, Audio DSP coprocessor, memory system, and external interfaces for audio input/output.

2. DSP Blocks (76% usage): The high level of DSP block usage is due to the MACs added by the IIR filters in the Audio DSP coprocessor. For the implementation of the second-order IIR equation, each of the used 32 IIR filters (left channel and right channel) need more than one DSP block so the DSP resources will be used heavily.

3. Logic Elements (68% usage): The five stage pipeline and the control logic of the RISC-V processor and audio DSP coprocessor's custom interfaces drove up the logic resource.

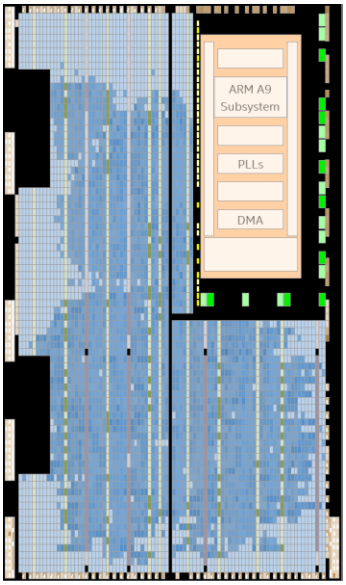


Figure 17. Chip planner visualization.

Figure 17 shows a visual representation of the components placed and routed on the FPGA.

As illustrated in the chip planner view above, the blue shaded regions are the active logic utilization in the FPGA fabric. To the right are the ARM A9 Subsystem, PLLs and DMA controller. These are the HPS parts of Soc. The resource utilization analysis shows that the system consumes most of the FPGA resources, pins and DSP blocks. It is well known that DSP block utilization is a key constraint in high-performance signal processing designs. The implementation of this project is in line with the findings of Meyer-Baese [12], showing the benefits of optimized filter mapping and the use of DSP fragments in FPGAs

for MAC-heavy operations.

4.7 System Demonstration

The functionality of the voice vocoder system is difficult to visualize since the transformation is produced on the audio side. To demonstrate the functionality of the integrated RISC-V processor with Audio DSP coprocessor system and clarify the real-time audio processing capabilities, the video demonstration of the whole system was recorded as shown in the Appendix. This demonstration video provides concrete evidence of the system functionality and performance under real-world applications.

After evaluating the project outcomes, it can be confirmed that the implementation proceeded following the project plan and successfully achieved the essential goals established. The primary objective of constructing an FPGA-Based Audio DSP Coprocessor for the TinyRV2 CPU was fully accomplished.

While all essential goals were accomplished, there was one stretched goal that remains to be reached in the future: the FFT visualization of the voice. Fast Fourier Transform (FFT) is another fundamental workload for audio DSP applications. This stretched goal was identified as a potential extension of the current work. As detailed in the Future Work section, the addition of FFT capabilities would further enhance the system's functionality and provide additional audio processing options.

5. Conclusion

This project has successfully achieved the extension of a five-stage RISC-V pipelined processor with customized Audio Digital Signal Processing (DSP) capabilities through specialized instruction sets and hardware coprocessor. The implementation on the DE1-SoC FPGA platform has validated the effectiveness of this approach for enhancing computational efficiency in Audio DSP tasks, particularly for operations such as Infinite Impulse Response (IIR) filtering and voice vocoding.

The project achieved results including integration of the RISC-V processor with the audio DSP coprocessor through the handshaking protocol, the implementation of M10K memory blocks through SRAM wrappers, and customized IIR instructions for audio operations. The voice vocoder successfully transformed human voice into robotic output with minimal latency. Moreover, flexible HPS-FPGA communication achieved dynamic program loading without recompiling the FPGA. Despite these achievements, several challenges were encountered, including memory latency which requires adjustment to 2-cycle latency, high pin and DSP block utilization, precision limitations due to the fixed-point arithmetic, and complex requirements during the integration. In conclusion, this project demonstrated that the RISC-V architecture, when extended with specialized hardware coprocessor, can

effectively address audio processing demands.

6. Future work

The stretched goal for future development of this project is the FFT visualization of the input voice. The Fast Fourier Transform (FFT) is considered as the fundamental algorithm in the audio DSP applications.

As shown in the figure below, the FFT algorithm uses a divide-and-conquer strategy which divides the input signal into smaller parts and enables computing the N/2-point Discrete Fourier Transforms (DFTs) separately for samples with even and odd indices. The outcomes are then combined based on the symmetrical properties of the DFT to get the complete frequency spectrum. This approach achieves the efficient conversion of time-domain signals into their frequency-domain representations, enabling spectral analysis of audio signals. Implementing this FFT visualization benchmark would require extending the current Audio DSP coprocessor to handle the computational workload of FFT, which includes intensive multiplications, additions, and data movement operations. These operations are very suitable for hardware acceleration and could leverage the existing FPGA infrastructure. The processed frequency-domain results could be visualized in real-time on the VGA display, taking advantage of the DE1-SoC board's rich peripheral interfaces.

This future work would build upon the foundation established in the current project, further demonstrating the flexibility and extensibility of the RISC-V architecture for those specialized Audio DSP applications.

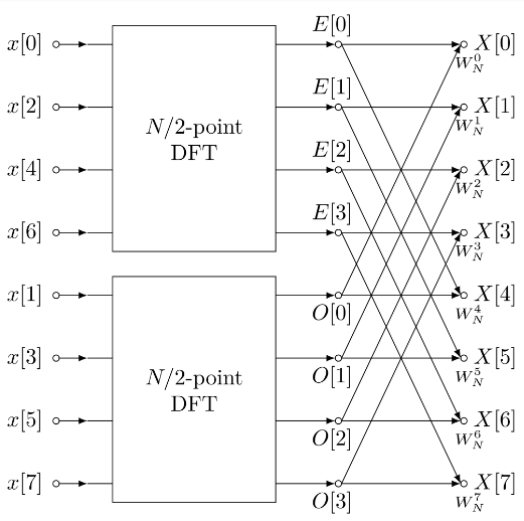


Figure 3. Structure of FFT algorithm [13].

7. Project Budget

This project requires specific hardware components to successfully implement and test the FPGA-based Audio DSP Coprocessor for the TinyRV2 CPU. The core of the implementation relies on the DE1-SoC FPGA development board ($377), which provides the necessary FPGA resources, ARM HPS system, and integrated peripherals essential for developing the RISC-V processor with custom DSP extensions. To enable high-quality audio processing capabilities, the system requires audio input/output interfaces, including a USB microphone ($15) for voice capture and a stereo audio player ($20) for playback of the processed audio signals. For proper connectivity between components, the project utilizes several 3.5mm stereo audio cables ($5 each) to connect the FPGA board to external audio devices.

Acknowledgement

I extend my deepest gratitude to Dr. Hunter Adams, whose steady guidance and encouragement made this work possible. I also want to acknowledge Dr. Anne Bracy; the Lab 2 materials from her Cornell ECE 4750 Computer Architecture course shaped the core of our processor design. My thanks go to Prof. Christopher Batten for ECE 6745 lab2 materials and SRAM tutorials on accelerator-processor interfaces and SRAM handshaking protocol, which proved crucial to my SRAM and coprocessor implementation. The FPGA processor concepts were also inspired by last year's ECE 5760 RISC-V CPU teams—Han Yang, Zhuoer Shao, Huzhe Liu, Tongyuan Liu, and Jiacheng Tu. Ideas for the voice vocoder drew on the ECE 5760 Speech Vocoder project by João Pedro Carvão, Justin Joco, and Thinesiya Krishnathasan. Their prior work laid the groundwork for many of the concepts realized here.

Reference

[1] FPGAs Key, "Learn FPGA: The ultimate guide to understand FPGA tutorial," [Online]. Available: https://www.fpgakey.com/tutorial/section348. [Accessed: Nov. 20, 2024].

[2] M. Gautschi *et al*., "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *arXiv preprint arXiv:1608.08376*, Aug. 2016.

[3] W. A. Sethares, "Channel Vocoder," *University of Wisconsin-Madison*, Accessed: Apr. 27, 2024. [Online]. Available: https://sethares.engr.wisc.edu/vocoders/channelvocoder.html

[4] "DE1-SoC User Manual v.1.2.2," *Cornell University ECE 5760 Course*, Accessed: Apr. 27, 2024. [Online]. Available

[5] L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, 1978.

[6] "ECE 4750 Lab 2: Processor Design," School of Electrical and Computer Engineering, Cornell University, [Online]. Available: https://www.csl.cornell.edu/courses/ece4750/handouts/ece4750-lab2-proc.pdf. [Accessed: Nov. 18, 2024].

[7] Cornell ECE 6745 Course Staff, "ECE 6745 Tutorial 09: Building an RTL Accelerator," *Cornell University*, Apr. 2025. [Online]. Available: https://cornell-ece6745.github.io/ece6745-docs/ece6745-tut09-xcel-rtl/

[8] E. Cui, T. Li, and Q. Wei, "RISC-V instruction set architecture extensions: A survey," *IEEE Access*, vol. 11, pp. 24696–24711, 2023.

[9] RISC-V International, "RISC-V Instruction Set Manual," [Online]. Available: https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications. [Accessed: May 15, 2025].

[10] Cornell ECE 6745 Course Staff, "ECE 6745 Tutorial 10: Integrating SRAM with OpenRAM," *Cornell University*, Apr. 2025. [Online]. Available: https://cornell-ece6745.github.io/ece6745-docs/ece6745-tut10-sram/

[11] S. Liu, J. Tang, and T. Liu, "Voice Controlled Piano Using FPGA," *Cornell ECE 5760 Final Project Report*, May 2024. [Online]. Available: https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/s2024/sl2973_jt842_tl839/sl2973_jt842_tl839/finalreport.html

[12] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, 3rd ed., vol. 65. Berlin, Germany: Springer, 2007.

[13] "DIT-FFT butterfly," Wikimedia Commons, Accessed: Apr. 27, 2024. [Online]. Available: https://commons.wikimedia.org/wiki/File:DIT-FFT-butterfly.svg

Appendix:

Appendix A: YouTube Channel Link:

Demo of this project: https://www.youtube.com/watch?v=Se5XtEpPGkQ

Appendix B: Code

The following section shows the part of the important code I implemented for this project. As the project includes code permitted for use from ECE 4750 Computer Architecture Lab 2, these segments cannot be shared publicly to preserve their potential use in future course materials. The full code will be shared privately with the supervisor, Van Hunter Adams.

a. Code for the HPS Side:

//
//

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <math.h>

// main bus; PIO
#define FPGA_AXI_BASE   0xC0000000
#define FPGA_AXI_SPAN   0x00001000
// main axi bus base
void *h2p_virtual_base;
volatile unsigned int * axi_pio_ptr = NULL ;
volatile unsigned int * axi_pio_read_ptr = NULL ;

// lw bus; PIO
#define FPGA_LW_BASE    0xff200000
#define FPGA_LW_SPAN    0x00001000
// the light weight bus base
void *h2p_lw_virtual_base;
// HPS_to_FPGA FIFO status address = 0
volatile unsigned int * lw_pio_ptr = NULL ;
volatile unsigned int * lw_pio_read_ptr = NULL ;

// read offset is 0x10 for both busses
// remember that eaxh axi master bus needs unique address
#define FPGA_PIO_READ   0x10
#define FPGA_PIO_WRITE  0x00

// SRAM implementation
#define FPGA_ONCHIP_BASE      0xC8000000
#define FPGA_ONCHIP_SPAN      0x00040000
void *fp_ram_virtual_base;
volatile unsigned int * inst_ram_ptr = NULL;
volatile unsigned int * data_ram_ptr = NULL;
volatile unsigned int * source_ram_ptr = NULL;
volatile unsigned int * sink_ram_ptr = NULL;
volatile unsigned int * coff_ram_ptr = NULL;
volatile unsigned int * pitch_ram_ptr = NULL;
volatile unsigned int * cb13_ram_ptr = NULL;
volatile unsigned int * ca2_ram_ptr = NULL;
volatile signed   int * ca3_ram_ptr = NULL;

#define INST_SRAM   0x0000
#define DATA_SRAM   0x1000
#define SOURCE_SRAM 0x2000
```

```
#define SINK_SRAM   0x3000
#define COFF_SRAM   0x3100
#define PITCH_SRAM  0x3200
#define CB13_SRAM   0x3300
#define CA2_SRAM    0x3400
#define CA3_SRAM    0x3500

// My pio implement
volatile unsigned int * done_pio_ptr = NULL;
#define MEM_INIT 0x10
volatile unsigned int * mem_init_ptr = NULL;

#define SRC_MAX 0x20
#define SNK_MAX 0x30
#define RESTART_SIG 0x30

volatile unsigned int * src_max_ptr = NULL;
volatile unsigned int * sink_max_ptr = NULL;
volatile unsigned int * restart_sig_ptr = NULL;


// /dev/mem file id
int fd;

int main(void)
{

   // Declare volatile pointers to I/O registers (volatile
   // means that IO load and store instructions will be used
   // to access these pointer locations,

   // === get FPGA addresses ==================
   // Open /dev/mem
   if( ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 )    {
      printf( "ERROR: could not open \"/dev/mem\"...\n" );
      return( 1 );
   }

   // === get RAM parameter addresses ==================
   fp_ram_virtual_base = mmap(NULL, FPGA_ONCHIP_SPAN, ( PROT_READ |
PROT_WRITE ), MAP_SHARED, fd, FPGA_ONCHIP_BASE );
   if( fp_ram_virtual_base == MAP_FAILED ) {
      printf( "ERROR: mmap3M10K() failed...\n" );
      close( fd );
      return(1);
   }

   //=============================================
   // get virtual addr that maps to physical
   // for light weight AXI bus
   h2p_lw_virtual_base = mmap( NULL, FPGA_LW_SPAN, ( PROT_READ | PROT_WRITE ),
```

```
MAP_SHARED, fd, FPGA_LW_BASE );
   if( h2p_lw_virtual_base == MAP_FAILED ) {
      printf( "ERROR: mmap1() failed...\n" );
      close( fd );
      return(1);
   }
   // Get the addresses that map to the two parallel ports on the light-weight bus
   lw_pio_ptr = (unsigned int *)(h2p_lw_virtual_base);
   lw_pio_read_ptr = (unsigned int *)(h2p_lw_virtual_base + FPGA_PIO_READ);

   //=============================================

   // =============================================
   // get virtual address for
   // AXI bus addr
   h2p_virtual_base = mmap( NULL, FPGA_AXI_SPAN, ( PROT_READ | PROT_WRITE ),
MAP_SHARED, fd, FPGA_AXI_BASE);
   if( h2p_virtual_base == MAP_FAILED ) {
      printf( "ERROR: mmap3() failed...\n" );
      close( fd );
      return(1);
   }
   // Get the addresses that map to the two parallel ports on the AXI bus
   axi_pio_ptr =(unsigned int *)(h2p_virtual_base);
   axi_pio_read_ptr =(unsigned int *)(h2p_virtual_base + FPGA_PIO_READ);
   //=============================================
   // done_sig_pio
   done_pio_ptr = (unsigned int *)(h2p_lw_virtual_base);
   mem_init_ptr = (unsigned int *)(h2p_lw_virtual_base + MEM_INIT);
   src_max_ptr = (unsigned int *)(h2p_lw_virtual_base + SRC_MAX);
   sink_max_ptr = (unsigned int *)(h2p_lw_virtual_base + SNK_MAX);
   restart_sig_ptr = (unsigned int *)(h2p_lw_virtual_base + RESTART_SIG);

   inst_ram_ptr = (unsigned int *)(fp_ram_virtual_base+ INST_SRAM);
   data_ram_ptr = (unsigned int *)(fp_ram_virtual_base + DATA_SRAM);
   source_ram_ptr = (unsigned int *)(fp_ram_virtual_base + SOURCE_SRAM);
   sink_ram_ptr = (unsigned int *)(fp_ram_virtual_base + SINK_SRAM);
   coff_ram_ptr = (unsigned int *)(fp_ram_virtual_base + COFF_SRAM);
   pitch_ram_ptr = (unsigned int *)(fp_ram_virtual_base + PITCH_SRAM);
   cb13_ram_ptr = (unsigned int *)(fp_ram_virtual_base + CB13_SRAM);
   ca2_ram_ptr = (unsigned int *)(fp_ram_virtual_base + CA2_SRAM);
   ca3_ram_ptr = (signed int *)(fp_ram_virtual_base + CA3_SRAM);

   int addr_inst = 0;
   int addr_data = 0;
   int addr_src = 0;
   int addr_snk = 0;

   int inst_arr[40] = {
      0xFC0020F3,
      0xFC002173,
```

```
    0xFFF08293,
    0x04028663,
    0x00028333,
    0x000003B3,
    0x02030C63,
    0x00239413,
    0x00810433,
    0x00042483,
    0x00440513,
    0x00052503,
    0x009525B3,
    0x00058863,
    0x00A42023,
    0x00440613,
    0x00962023,
    0x00138393,
    0xFFF30313,
    0xFCDFF06F,
    0xFFF28293,
    0xFB9FF06F,
    0x7C001073,
    0x7C001073,
    0x00000013,
    0x00000013,
    0x00000013,
    0xFC0020F3,
    0xFC0021F3,
    0x7E419073,
    0x7E4021F3,
    0x7E001073,
    0x7E002073,
    0x7C019073,
    0xFC0020F3,
    0,
    0,
    0,
    0,
    0
};

int data_arr[20] = {
    0x00000721,
    0x0000019A,
    0x0000119B,
    0x00000FAD,
    0x00000E4A,
    0x000008EF,
    0x00000690,
    0x000022E8,
    0x00000591,
    0x000025CB,
```

```
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0
};

int source_arr[5] = {
    0x0000000A,
    0x00002000,
    0x00000008,
    0x00000020,
    0
};

int sink_arr[5] = {
    0,
    0,
    0,
    0,
    0
};

int coff_arr[32] = {
    26843600,
    31083210,
    35523697,
    40174581,
    45045829,
    50147884,
    55491680,
    61088673,
    66950858,
    73090801,
    79521663,
    86257227,
    93311931,
    100700897,
    108439960,
    116545711,
    125035523,
    133927593,
    143240981,
    152995649,
    163212506,
    173913451,
```

```
    185121421,
    196860439,
    209155666,
    222033458,
    235521416,
    249648451,
    264444843,
    279942308,
    296174062,
    313174898
};

int pitch_arr[32] = {1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1
};

// int pitch_arr[32] = {2,2,2,2,2,2,2,2,
//  2,2,2,2,2,2,2,2,
//  2,2,2,2,2,2,2,2,
//  2,2,2,2,2,2,2,2
// };

int cb13_arr[32] = {
    45,52,59,67,
    75,84,93,102,
    112,122,133,144,
    156,168,181,195,
    209,223,239,255,
    272,290,309,328,
    349,370,392,416,
    440,466,493,521
};

int ca2_arr[32] = {
    130880,130832,130776,130711,
    130636,130551,130455,130345,
    130220,130080,129921,129743,
    129543,129318,129067,128787,
    128473,128124,127735,127303,
    126823,126289,125698,125042,
    124317,123514,122627,121647,
    120566,119374,118061,116615
};

int ca3_arr[32] = {
    -65445,-65431,-65416,-65401,
    -65384,-65367,-65349,-65331,
    -65311,-65291,-65269,-65247,
    -65223,-65198,-65173,-65145,
```

```
    -65117,-65088,-65056,-65024,
    -64990,-64954,-64917,-64878,
    -64837,-64794,-64750,-64703,
    -64654,-64603,-64549,-64493
};

int addr = 0;
while(1)
{

    // ------------------- Initialize the input mem ---------------------
    // send to PIO, the mem init no done
    *(mem_init_ptr) = 0;
    *(restart_sig_ptr) = 0;
    printf("------- init sram ------- \n\r");
    // init
    for (addr=0; addr <40; addr++)
    {
        *(inst_ram_ptr+addr) = inst_arr[addr];
    }

    // data
    for (addr=0; addr <20; addr++)
    {
        *(data_ram_ptr+addr) = data_arr[addr];
    }

    // source
    for (addr=0; addr <5; addr++)
    {
        *(source_ram_ptr+addr) = source_arr[addr];
    }

    // sink
    for (addr=0; addr <5; addr++)
    {
        *(sink_ram_ptr+addr) = sink_arr[addr];
    }

    // coff
    for (addr=0; addr <32; addr++)
    {
        *(coff_ram_ptr+addr) = coff_arr[addr];
    }

    // pitch
    for (addr=0; addr <32; addr++)
    {
        *(pitch_ram_ptr+addr) = pitch_arr[addr];
    }
```

```
// cb13
for (addr=0; addr <32; addr++)
{
   *(cb13_ram_ptr+addr) = cb13_arr[addr];
}

// ca2
for (addr=0; addr <32; addr++)
{
   *(ca2_ram_ptr+addr) = ca2_arr[addr];
}

// ca3
for (addr=0; addr <32; addr++)
{
   *(ca3_ram_ptr+addr) = ca3_arr[addr];
}

/// ------------ inst sram init -----------------------
printf("-----print the initized inst sram ------- \n\r");
for (addr=0; addr <40; addr++)
{
   printf("RAM addr=%d contents=0x%08X \n\r",addr, *(inst_ram_ptr+addr));
}


/// ------------ data sram init -----------------------
printf("-----print the initized data sram ------- \n\r");
for (addr=0; addr <20; addr++)
{
   printf("RAM addr=%d contents=0x%08X \n\r",addr, *(data_ram_ptr+addr));
}

/// ------------ source sram init -----------------------
printf("-----print the initized source sram ------- \n\r");
for (addr=0; addr <5; addr++)
{
   printf("RAM addr=%d contents=0x%08X \n\r",addr, *(source_ram_ptr+addr));
}

/// ------------ sink sram init -----------------------
printf("-----print the initized sink sram ------- \n\r");
for (addr=0; addr <5; addr++)
{
   printf("RAM addr=%d contents=0x%08X \n\r",addr, *(sink_ram_ptr+addr));
}

/// ------------ coff sram init -----------------------
printf("-----print the initized coff sram ------- \n\r");
for (addr=0; addr <32; addr++)
{
```

```
   printf("RAM addr=%d contents=0x%08X \n\r",addr, *(coff_ram_ptr+addr));
}

/// ------------ pitch sram init -----------------------
printf("-----print the initized pitch sram ------- \n\r");
for (addr=0; addr <32; addr++)
{
   printf("RAM addr=%d contents=0x%08X \n\r",addr, *(pitch_ram_ptr+addr));
}

*(src_max_ptr) = 4;
*(sink_max_ptr) = 3;
*(mem_init_ptr) = 1;
// ------------------ printed the output result --------------------
while (*(done_pio_ptr)==0){}

//------------------------------------------------------------------------
//-------------- SRAM Printed Results ---------------------------------
//------------------------------------------------------------------------
printf("------- print finished inst ram ------- \n\r");
for (addr=0; addr <40; addr++)
{
   printf("RAM addr=%d contents=0x%08X \n\r",addr, *(inst_ram_ptr+addr));
}

printf("------- print finished data ram ------- \n\r");
for (addr=0; addr <20; addr++)
{
   printf("RAM addr=%d contents=0x%08X \n\r",addr, *(data_ram_ptr+addr));
}

printf("------- print finished source ram ------- \n\r");
for (addr=0; addr <5; addr++)
{
   printf("RAM addr=%d contents=0x%08X \n\r",addr, *(source_ram_ptr+addr));
}


printf("------- print finished sink ram ------- \n\r");
for (addr=0; addr <5; addr++)
{
   printf("RAM addr=%d contents=0x%08X \n\r",addr, *(sink_ram_ptr+addr));
}

*(mem_init_ptr) = 0;
*(src_max_ptr) = 0;
*(sink_max_ptr) = 0;
// ask new try
int continue_flag = 0;
printf("Have the new try? (1 for continue, other for exit): ");
```

```
        // refresh fifo for input in C
        fflush(stdin);  //

        char input_char = getchar();

        // check '1' for continue ?
        if (input_char == '1') {
            continue_flag = 1;
            *(restart_sig_ptr) = 1;
        } else {
            printf("Exiting the program...\n\r");
            break;
        }

        // refresh
        while ((input_char = getchar()) != '\n' && input_char != EOF);
        // exit
        if (!continue_flag) {
            break;
        }
    } // end while(1)
} // end main
```

b. Code used for the Source SRAM testing (Test Source testbench):

```
`timescale 1ns/1ps

`include "TestSource.sv"
module vc_TestSource_tb;
  // para define
  parameter p_msg_nbits = 32;
  parameter p_num_msgs  = 10;

  parameter logic [p_msg_nbits-1:0] p_test_msgs[p_num_msgs] = '{
  'h10101010, 'h21212121, 'h32323232, 'h43434343, 'h54545454,
  'h65656565, 'h76767676, 'h87878787, 'h98989898, 'ha9a9a9a9
};

  // sig def
  logic                clk;
  logic                reset;
  logic                val;
  logic                rdy;
  logic [p_msg_nbits-1:0] msg;
  logic src_done;
  logic [5:0]            SOURCE_MAX;

  // ---------------- sram transfer ------------
  // sram_source
```

```
logic [4:0]  source_sram_address;
logic        source_sram_chipselect;
logic        source_sram_write;
logic [31:0] source_sram_writedata;   // not used here, just tie to 0
logic [31:0] source_sram_readdata;

// instantialize the DUT module
vc_TestSource #(
  .p_msg_nbits(p_msg_nbits)
) dut (
  .clk  (clk),
  .reset(reset),
  .val  (val),
  .rdy  (rdy),
  .msg  (msg),
  .done (src_done),
  .num_msgs (SOURCE_MAX),
// sram ports
  .source_sram_addr (source_sram_address),
  .source_sram_en   (source_sram_chipselect),
  .source_sram_wen  (source_sram_write),
  .source_sram_wdata(source_sram_writedata),   // not used here, just tie to 0
  .source_sram_rdata(source_sram_readdata)
);

// sram_SramVRTL#(32,128) sram_source
sram_SramVRTL #(32,32) sramsource
(
  .clk        (clk),
  .reset      (reset),
  .port0_idx   (source_sram_address), // addr
  .port0_type  (source_sram_write), // 0=read, 1=write
  .port0_val   (source_sram_chipselect), // port enable
  .port0_wdata (source_sram_writedata), // write data
  .port0_rdata (source_sram_readdata) // read data
);

// generate clock
always #10 clk = ~clk;

// test cases
initial begin
  $display("Starting TestSource testbench...");
  // load msg to inner mem
  foreach (p_test_msgs[i])
    sramsource.sram.mem[i] = p_test_msgs[i];
    // dut.m[i] = p_test_msgs[i];

  SOURCE_MAX = 6;

  // initialization
```

```
    clk = 0;
    reset = 1;
    rdy = 1;
    #20 reset = 0;

    #30
    rdy = 0;
    #20
    rdy = 1;

    // wait unitl the end of transition
    @(posedge src_done);
    $display("All messages sent, testbench complete.");
    #50
    $finish;
  end

  // time out
  initial begin
    for( integer i = 0; i < 50; i = i + 1 ) begin
      @( negedge clk );
    end
    $display( "TIMEOUT: Testbench didn't finish in time" );
    $finish;
  end

  // dump waveform
  initial begin
    $dumpfile("waveform.vcd");
    $dumpvars(0, vc_TestSource_tb);            // Record the signals of the top module and all
submodules
  end
endmodule

//------------------------------------------------------------------------
// The called module
//------------------------------------------------------------------------
module sram_SramVRTL
#(
  parameter p_data_nbits  = 32,
  parameter p_num_entries = 256,

  // Local constants not meant to be set from outside the module
  parameter c_addr_nbits  = $clog2(p_num_entries),
  parameter c_data_nbytes = (p_data_nbits+7)/8 // $ceil(p_data_nbits/8)
)(
  input  logic                    clk,
  input  logic                    reset,
  input  logic                    port0_val,
  input  logic                    port0_type,
  input  logic [c_addr_nbits-1:0]   port0_idx,
```

```
  input  logic [p_data_nbits-1:0]   port0_wdata,
  output logic [p_data_nbits-1:0]   port0_rdata
);

  logic                  clk0;
  logic                  web0;
  logic                  csb0;
  logic [c_addr_nbits-1:0]  addr0;
  logic [p_data_nbits-1:0]  din0;
  logic [p_data_nbits-1:0]  dout0;

  assign clk0  = clk;
  assign web0  = ~port0_type;
  assign csb0  = ~port0_val;
  assign addr0 = port0_idx;
  assign din0  = port0_wdata;

  assign port0_rdata = dout0;

  generate

     sram_SramGenericVRTL#(p_data_nbits,p_num_entries) sram (.*);

  endgenerate

endmodule

module sram_SramGenericVRTL
#(
  parameter p_data_nbits  = 1,
  parameter p_num_entries = 2,

  // Local constants not meant to be set from outside the module
  parameter c_addr_nbits  = $clog2(p_num_entries),
  parameter c_data_nbytes = (p_data_nbits+7)/8 // $ceil(p_data_nbits/8)
)(
  input  logic                  clk0,  // clk
  input  logic                  web0,  // bar( write en )
  input  logic                  csb0,  // bar( whole SRAM en )
  input  logic [c_addr_nbits-1:0]   addr0, // address
  input  logic [p_data_nbits-1:0]   din0,  // write data
  output logic [p_data_nbits-1:0]   dout0  // read data
);

  logic [p_data_nbits-1:0] mem[p_num_entries-1:0];

  logic [p_data_nbits-1:0] data_out1;
  logic [p_data_nbits-1:0] wdata1;

  // // Inside sram_SramGenericVRTL module
  // task automatic init_mem(input logic [c_addr_nbits-1:0] addr, input logic [p_data_nbits-1:0]
```

```
data);
  //   mem[addr] = data;
  // endtask

  logic [31:0] mm0,mm1,mm2,mm3,mm4,mm5,mm6,mm7;

  assign mm0 = mem[0];
  assign mm1 = mem[1];
  assign mm2 = mem[2];
  assign mm3 = mem[3];
  assign mm4 = mem[4];
  assign mm5 = mem[5];
  assign mm6 = mem[6];
  assign mm7 = mem[7];
  assign mm8 = mem[8];
  assign mm9 = mem[9];

  always @( posedge clk0 ) begin

    // Read path

    if ( ~csb0 && web0 )
      data_out1 <= mem[addr0];
    else
      data_out1 <= {p_data_nbits{1'bx}};

  end

  // Write path

  genvar i;
  generate
    for ( i = 0; i < c_data_nbytes; i = i + 1 )
    begin : write
      always @( posedge clk0 ) begin
        if ( ~csb0 && ~web0 )
          mem[addr0][ (i+1)*8-1 : i*8 ] <= din0[ (i+1)*8-1 : i*8 ];
      end
    end
  endgenerate
  assign dout0 = data_out1;

endmodule
```

Code used for the Sink SRAM testing:

```
`timescale 1ns/1ps
`include "TestSink.sv"

module vc_TestSink_tb;
  // Parameters
```

```
parameter p_msg_nbits = 32;
parameter p_num_msgs  = 10;
parameter logic [p_msg_nbits-1:0] p_test_msgs[p_num_msgs] = '{
  'h10101010, 'h21212121, 'h32323232, 'h43434343, 'h54545454,
  'h65656565, 'h76767676, 'h87878787, 'h98989898, 'ha9a9a9a9
};


// Signal definitions
logic                clk;
logic                reset;
logic                val;
logic                rdy;
logic [p_msg_nbits-1:0] msg;
logic                snk_done;

// sram_sink
logic [4:0]  sink_sram_address;
logic        sink_sram_chipselect;
logic        sink_sram_write;
logic [31:0] sink_sram_writedata;   // not used here, just tie to 0
logic [31:0] sink_sram_readdata;

logic [5:0] SINK_MAX;

// Instantiate the DUT
vc_TestSink #(
  .p_msg_nbits(p_msg_nbits)
) dut (
  .clk   (clk),
  .reset (reset),
  .val   (val),
  .rdy   (rdy),
  .msg   (msg),
  .done  (snk_done),
  .num_msgs (SINK_MAX),

  .sink_sram_addr   (sink_sram_address),
  .sink_sram_en     (sink_sram_chipselect),
  .sink_sram_wen    (sink_sram_write),
  .sink_sram_wdata  (sink_sram_writedata),
  .sink_sram_rdata  (sink_sram_readdata)
);

  // sram_SramVRTL#(32,128) sram_source
sram_SramVRTL #(32,32) sramsink
(
  .clk         (clk),
  .reset       (reset),
  .port0_idx   (sink_sram_address), // addr
  .port0_type  (sink_sram_write), // 0=read, 1=write
```

```
  .port0_val   (sink_sram_chipselect), // port enable
  .port0_wdata (sink_sram_writedata), // write data
  .port0_rdata (sink_sram_readdata) // read data
);

// Generate clock
always #5 clk = ~clk; // 10ns period

// Test stimulus
initial begin
  $display("Starting TestSink testbench...");
  SINK_MAX = 8;
  // Initialize signals
  clk = 0;
  reset = 1;
  val = 0;
  msg = 0;

  // Apply reset
  repeat(3) @(posedge clk);
  reset = 0;
  val = 1; // Keep valid high throughout transmission
  // Send all messages in sequence
  for (int i = 0; i < SINK_MAX; i++) begin
    msg = p_test_msgs[i];
    @(posedge clk);
    while (!rdy) @(posedge clk); // Wait if sink not ready

    // Optional: Add debug prints
    $display("Time=%0t Sending message[%0d]: %h", $time, i, msg);
  end

  // Wait a cycle after reset
  @(posedge clk);
  // Send test messages with continuous valid
  @(posedge clk);
  @(posedge clk);
  @(posedge clk);
  @(posedge clk);

  // Keep sending the last message for a few cycles (as shown in waveform)
  msg = 'haaaaaaaa;
  repeat(3) @(posedge clk);

  // End simulation
  #100;
end

// Monitor memory writes
always @(posedge clk) begin
  if (val && rdy) begin
```

```
      $display("Time=%0t Stored msg[%0d]: %h", $time, dut.index, msg);
    end
  end


  // dump waveform
  initial begin
    $dumpfile("waveform.vcd");
    $dumpvars(0, vc_TestSink_tb);            // Record the signals of the top module and all
submodules
  end


  // time out
  initial begin
    for( integer i = 0; i < 50; i = i + 1 ) begin
      @( negedge clk );
    end
    $display( "TIMEOUT: Testbench didn't finish in time" );
    $finish;
  end

endmodule

//------------------------------------------------------------------------
// The called module
//------------------------------------------------------------------------
module sram_SramVRTL
#(
  parameter p_data_nbits  = 32,
  parameter p_num_entries = 256,

  // Local constants not meant to be set from outside the module
  parameter c_addr_nbits  = $clog2(p_num_entries),
  parameter c_data_nbytes = (p_data_nbits+7)/8 // $ceil(p_data_nbits/8)
)(
  input  logic                    clk,
  input  logic                    reset,
  input  logic                    port0_val,
  input  logic                    port0_type,
  input  logic [c_addr_nbits-1:0]   port0_idx,
  input  logic [p_data_nbits-1:0]   port0_wdata,
  output logic [p_data_nbits-1:0]   port0_rdata
);

  logic                   clk0;
  logic                   web0;
  logic                   csb0;
  logic [c_addr_nbits-1:0]  addr0;
  logic [p_data_nbits-1:0]  din0;
  logic [p_data_nbits-1:0]  dout0;
```

```
  // // Inside sram_SramVRTL
  // task automatic init_mem(input logic [c_addr_nbits-1:0] addr, input logic [p_data_nbits-1:0]
data);
  //   sram.init_mem(addr, data);
  // endtask

  assign clk0  = clk;
  assign web0  = ~port0_type;
  assign csb0  = ~port0_val;
  assign addr0 = port0_idx;
  assign din0  = port0_wdata;

  assign port0_rdata = dout0;

  generate

      sram_SramGenericVRTL#(p_data_nbits,p_num_entries) sram (.*);

  endgenerate

endmodule

module sram_SramGenericVRTL
#(
  parameter p_data_nbits  = 1,
  parameter p_num_entries = 2,

  // Local constants not meant to be set from outside the module
  parameter c_addr_nbits  = $clog2(p_num_entries),
  parameter c_data_nbytes = (p_data_nbits+7)/8 // $ceil(p_data_nbits/8)
)(
  input  logic                    clk0,  // clk
  input  logic                    web0,  // bar( write en )
  input  logic                    csb0,  // bar( whole SRAM en )
  input  logic [c_addr_nbits-1:0]   addr0, // address
  input  logic [p_data_nbits-1:0]   din0,  // write data
  output logic [p_data_nbits-1:0]   dout0  // read data
);

  logic [p_data_nbits-1:0] mem[p_num_entries-1:0];

  logic [p_data_nbits-1:0] data_out1;
  logic [p_data_nbits-1:0] wdata1;

  // // Inside sram_SramGenericVRTL module
  // task automatic init_mem(input logic [c_addr_nbits-1:0] addr, input logic [p_data_nbits-1:0]
data);
  //   mem[addr] = data;
  // endtask
```

```
  logic [31:0] mm0,mm1,mm2,mm3,mm4,mm5,mm6,mm7;

  assign mm0 = mem[0];
  assign mm1 = mem[1];
  assign mm2 = mem[2];
  assign mm3 = mem[3];
  assign mm4 = mem[4];
  assign mm5 = mem[5];
  assign mm6 = mem[6];
  assign mm7 = mem[7];
  assign mm8 = mem[8];
  assign mm9 = mem[9];

  always @( posedge clk0 ) begin

    // Read path

    if ( ~csb0 && web0 )
      data_out1 <= mem[addr0];
    else
      data_out1 <= {p_data_nbits{1'bx}};

  end

  // Write path

  genvar i;
  generate
    for ( i = 0; i < c_data_nbytes; i = i + 1 )
    begin : write
      always @( posedge clk0 ) begin
        if ( ~csb0 && ~web0 )
          mem[addr0][ (i+1)*8-1 : i*8 ] <= din0[ (i+1)*8-1 : i*8 ];
      end
    end
  endgenerate
  assign dout0 = data_out1;

endmodule
```

c. FPGA side code:

```
module DE1_SoC_Computer (
            ////////////////////////////////////
            // FPGA Pins
            ////////////////////////////////////

            // Clock pins
            CLOCK_50,
            CLOCK2_50,
```

```
CLOCK3_50,
CLOCK4_50,

// ADC
ADC_CS_N,
ADC_DIN,
ADC_DOUT,
ADC_SCLK,

// Audio
AUD_ADCDAT,
AUD_ADCLRCK,
AUD_BCLK,
AUD_DACDAT,
AUD_DACLRCK,
AUD_XCK,

// SDRAM
DRAM_ADDR,
DRAM_BA,
DRAM_CAS_N,
DRAM_CKE,
DRAM_CLK,
DRAM_CS_N,
DRAM_DQ,
DRAM_LDQM,
DRAM_RAS_N,
DRAM_UDQM,
DRAM_WE_N,

// I2C Bus for Configuration of the Audio and Video-In Chips
FPGA_I2C_SCLK,
FPGA_I2C_SDAT,

// 40-Pin Headers
GPIO_0,
GPIO_1,

// Seven Segment Displays
HEX0,
HEX1,
HEX2,
HEX3,
HEX4,
HEX5,

// IR
IRDA_RXD,
IRDA_TXD,

// Pushbuttons
```

```
KEY,

// LEDs
LEDR,

// PS2 Ports
PS2_CLK,
PS2_DAT,

PS2_CLK2,
PS2_DAT2,

// Slider Switches
SW,

// Video-In
TD_CLK27,
TD_DATA,
TD_HS,
TD_RESET_N,
TD_VS,

// VGA
VGA_B,
VGA_BLANK_N,
VGA_CLK,
VGA_G,
VGA_HS,
VGA_R,
VGA_SYNC_N,
VGA_VS,

///////////////////////////////////
// HPS Pins
///////////////////////////////////

// DDR3 SDRAM
HPS_DDR3_ADDR,
HPS_DDR3_BA,
HPS_DDR3_CAS_N,
HPS_DDR3_CKE,
HPS_DDR3_CK_N,
HPS_DDR3_CK_P,
HPS_DDR3_CS_N,
HPS_DDR3_DM,
HPS_DDR3_DQ,
HPS_DDR3_DQS_N,
HPS_DDR3_DQS_P,
HPS_DDR3_ODT,
HPS_DDR3_RAS_N,
HPS_DDR3_RESET_N,
```

```
HPS_DDR3_RZQ,
HPS_DDR3_WE_N,

// Ethernet
HPS_ENET_GTX_CLK,
HPS_ENET_INT_N,
HPS_ENET_MDC,
HPS_ENET_MDIO,
HPS_ENET_RX_CLK,
HPS_ENET_RX_DATA,
HPS_ENET_RX_DV,
HPS_ENET_TX_DATA,
HPS_ENET_TX_EN,

// Flash
HPS_FLASH_DATA,
HPS_FLASH_DCLK,
HPS_FLASH_NCSO,

// Accelerometer
HPS_GSENSOR_INT,

// General Purpose I/O
HPS_GPIO,

// I2C
HPS_I2C_CONTROL,
HPS_I2C1_SCLK,
HPS_I2C1_SDAT,
HPS_I2C2_SCLK,
HPS_I2C2_SDAT,

// Pushbutton
HPS_KEY,

// LED
HPS_LED,

// SD Card
HPS_SD_CLK,
HPS_SD_CMD,
HPS_SD_DATA,

// SPI
HPS_SPIM_CLK,
HPS_SPIM_MISO,
HPS_SPIM_MOSI,
HPS_SPIM_SS,

// UART
HPS_UART_RX,
```

```
        HPS_UART_TX,

        // USB
        HPS_CONV_USB_N,
        HPS_USB_CLKOUT,
        HPS_USB_DATA,
        HPS_USB_DIR,
        HPS_USB_NXT,
        HPS_USB_STP
);

//=======================================================
//  PARAMETER declarations
//=======================================================


//=======================================================
//  PORT declarations
//=======================================================

///////////////////////////////////
// FPGA Pins
///////////////////////////////////

// Clock pins
input                       CLOCK_50;
input                       CLOCK2_50;
input                       CLOCK3_50;
input                       CLOCK4_50;

// ADC
inout                       ADC_CS_N;
output                  ADC_DIN;
input                       ADC_DOUT;
output                  ADC_SCLK;

// Audio
input                       AUD_ADCDAT;
inout                       AUD_ADCLRCK;
inout                       AUD_BCLK;
output                  AUD_DACDAT;
inout                       AUD_DACLRCK;
output                  AUD_XCK;

// SDRAM
output      [12: 0]  DRAM_ADDR;
output      [ 1: 0]  DRAM_BA;
output                  DRAM_CAS_N;
output                  DRAM_CKE;
output                  DRAM_CLK;
output                  DRAM_CS_N;
```

```
inout       [15: 0]  DRAM_DQ;
output               DRAM_LDQM;
output               DRAM_RAS_N;
output               DRAM_UDQM;
output               DRAM_WE_N;

// I2C Bus for Configuration of the Audio and Video-In Chips
output               FPGA_I2C_SCLK;
inout                FPGA_I2C_SDAT;

// 40-pin headers
inout       [35: 0]  GPIO_0;
inout       [35: 0]  GPIO_1;

// Seven Segment Displays
output      [ 6: 0]  HEX0;
output      [ 6: 0]  HEX1;
output      [ 6: 0]  HEX2;
output      [ 6: 0]  HEX3;
output      [ 6: 0]  HEX4;
output      [ 6: 0]  HEX5;

// IR
input                IRDA_RXD;
output               IRDA_TXD;

// Pushbuttons
input       [ 3: 0]  KEY;

// LEDs
output      [ 9: 0]  LEDR;

// PS2 Ports
inout                PS2_CLK;
inout                PS2_DAT;

inout                PS2_CLK2;
inout                PS2_DAT2;

// Slider Switches
input       [ 9: 0]  SW;

// Video-In
input                TD_CLK27;
input       [ 7: 0]  TD_DATA;
input                TD_HS;
output               TD_RESET_N;
input                TD_VS;

// VGA
output      [ 7: 0]  VGA_B;
```

```
output                                  VGA_BLANK_N;
output                                  VGA_CLK;
output          [ 7: 0]   VGA_G;
output                                  VGA_HS;
output          [ 7: 0]   VGA_R;
output                                  VGA_SYNC_N;
output                                  VGA_VS;



///////////////////////////////////
// HPS Pins
///////////////////////////////////

// DDR3 SDRAM
output          [14: 0]  HPS_DDR3_ADDR;
output          [ 2: 0]  HPS_DDR3_BA;
output                                  HPS_DDR3_CAS_N;
output                                  HPS_DDR3_CKE;
output                                  HPS_DDR3_CK_N;
output                                  HPS_DDR3_CK_P;
output                                  HPS_DDR3_CS_N;
output          [ 3: 0]   HPS_DDR3_DM;
inout           [31: 0]  HPS_DDR3_DQ;
inout           [ 3: 0]   HPS_DDR3_DQS_N;
inout           [ 3: 0]   HPS_DDR3_DQS_P;
output                                  HPS_DDR3_ODT;
output                                  HPS_DDR3_RAS_N;
output                                  HPS_DDR3_RESET_N;
input                                           HPS_DDR3_RZQ;
output                                  HPS_DDR3_WE_N;

// Ethernet
output                                  HPS_ENET_GTX_CLK;
inout                                           HPS_ENET_INT_N;
output                                  HPS_ENET_MDC;
inout                                           HPS_ENET_MDIO;
input                                           HPS_ENET_RX_CLK;
input           [ 3: 0]   HPS_ENET_RX_DATA;
input                                           HPS_ENET_RX_DV;
output          [ 3: 0]   HPS_ENET_TX_DATA;
output                                  HPS_ENET_TX_EN;

// Flash
inout           [ 3: 0]   HPS_FLASH_DATA;
output                                  HPS_FLASH_DCLK;
output                                  HPS_FLASH_NCSO;

// Accelerometer
inout                                           HPS_GSENSOR_INT;
```

```
// General Purpose I/O
inout       [ 1: 0]   HPS_GPIO;

// I2C
inout                 HPS_I2C_CONTROL;
inout                 HPS_I2C1_SCLK;
inout                 HPS_I2C1_SDAT;
inout                 HPS_I2C2_SCLK;
inout                 HPS_I2C2_SDAT;

// Pushbutton
inout                 HPS_KEY;

// LED
inout                 HPS_LED;

// SD Card
output                HPS_SD_CLK;
inout                 HPS_SD_CMD;
inout       [ 3: 0]   HPS_SD_DATA;

// SPI
output                HPS_SPIM_CLK;
input                 HPS_SPIM_MISO;
output                HPS_SPIM_MOSI;
inout                 HPS_SPIM_SS;

// UART
input                 HPS_UART_RX;
output                HPS_UART_TX;

// USB
inout                 HPS_CONV_USB_N;
input                 HPS_USB_CLKOUT;
inout       [ 7: 0]   HPS_USB_DATA;
input                 HPS_USB_DIR;
input                 HPS_USB_NXT;
output                HPS_USB_STP;

//=======================================================
//  REG/WIRE declarations
//=======================================================

wire        [15: 0]  hex3_hex0;
//wire        [15: 0]  hex5_hex4;

//assign HEX0 = ~hex3_hex0[ 6: 0]; // hex3_hex0[ 6: 0];
//assign HEX1 = ~hex3_hex0[14: 8];
//assign HEX2 = ~hex3_hex0[22:16];
//assign HEX3 = ~hex3_hex0[30:24];
//assign HEX4 = 7'b1111111;
```

```
//assign HEX5 = 7'b1111111;
//assign HEX3 = 7'b1111111;
// assign HEX2 = 7'b1111111;
// assign HEX1 = 7'b1111111;
// assign HEX0 = 7'b1111111;

 HexDigit Digit0(HEX0, SOURCE_MAX[4:0]);
 HexDigit Digit1(HEX1, SINK_MAX[4:0]);
 HexDigit Digit2(HEX2, reset_audio);
 HexDigit Digit3(HEX3, iir_fsm_ex);
 HexDigit Digit4(HEX4, trans_state);
 HexDigit Digit5(HEX5, restart_sig[4:0]);

//  HexDigit Digit2(HEX2, iir_fsm_ex[0:0]);

// HexDigit Digit2(HEX2, source_data_buffer[4:0]);
// iir_fsm_ex
//=======================================================
// implement Proc
//=======================================================
  logic        iir_reset;
  logic        reset;
  logic        clk;
  logic [31:0]  mngr2proc_msg;
  logic        mngr2proc_val;
  logic        mngr2proc_rdy;

  // To mngr streaming port

  logic [31:0]  proc2mngr_msg;
  logic        proc2mngr_val;
  logic        proc2mngr_rdy;

  // Instruction Memory Request Port

  mem_req_4B_t  imem_reqstream_msg;
  logic        imem_reqstream_val;
  logic        imem_reqstream_rdy;

  // Instruction Memory Response Port

  mem_resp_4B_t imem_respstream_msg;
  logic        imem_respstream_val;
  logic        imem_respstream_rdy;

  // Data Memory Request Port

  mem_req_4B_t  dmem_reqstream_msg;
  logic        dmem_reqstream_val;
  logic        dmem_reqstream_rdy;
```

```
// Data Memory Response Port

mem_resp_4B_t dmem_respstream_msg;
logic        dmem_respstream_val;
logic        dmem_respstream_rdy;

xcel_req_t    xcel_reqstream_msg;
logic        xcel_reqstream_val;
logic        xcel_reqstream_rdy;

// Xcel Response Port

xcel_resp_t   xcel_respstream_msg;
logic        xcel_respstream_val;
logic        xcel_respstream_rdy;
// extra ports; note that core_id is an input port rather than a
// parameter so that the module only needs to be compiled once. If it
// were a parameter, each core would be compiled separately.

logic [31:0]  core_id;
logic        commit_inst;
logic        stats_en;

logic src_done;
logic snk_done;
logic [5:0] SINK_MAX;
logic [5:0] SOURCE_MAX;
logic [5:0] restart_sig;
// ---------------------- sram transfer --------------------
// sram_source
logic [4:0]  source_sram_address;
logic       source_sram_chipselect;
logic       source_sram_write;
logic [31:0] source_sram_writedata;   // not used here, just tie to 0
logic [31:0] source_sram_readdata;

// sram_sink
logic [4:0]  sink_sram_address;
logic       sink_sram_chipselect;
logic       sink_sram_write;
logic [31:0] sink_sram_writedata;   // not used here, just tie to 0
logic [31:0] sink_sram_readdata;

// sram_inst
logic [7:0]  inst_sram_address;
logic       inst_sram_chipselect;
logic       inst_sram_write;
logic [31:0] inst_sram_writedata;   // not used here, just tie to 0
logic [31:0] inst_sram_readdata;

// sram_data
```

```
logic [7:0]  data_sram_address;
logic        data_sram_chipselect;
logic        data_sram_write;
logic [31:0] data_sram_writedata;   // not used here, just tie to 0
logic [31:0] data_sram_readdata;

// sram_coeff
logic [5:0]  coeff_sram_address;
logic        coeff_sram_chipselect;
logic        coeff_sram_write;
logic [31:0] coeff_sram_writedata;   // not used here, just tie to 0
logic [31:0] coeff_sram_readdata;

// cb13_sram
logic [5:0]  cb13_sram_address;
logic        cb13_sram_chipselect;
logic        cb13_sram_write;
logic [31:0] cb13_sram_writedata;   // not used here, just tie to 0
logic [31:0] cb13_sram_readdata;

// ca2_sram
logic [5:0]  ca2_sram_address;
logic        ca2_sram_chipselect;
logic        ca2_sram_write;
logic [31:0] ca2_sram_writedata;   // not used here, just tie to 0
logic [31:0] ca2_sram_readdata;

// ca3_sram
logic [5:0]  ca3_sram_address;
logic        ca3_sram_chipselect;
logic        ca3_sram_write;
logic [31:0] ca3_sram_writedata;   // not used here, just tie to 0
logic [31:0] ca3_sram_readdata;

  // pitch_sram
logic [5:0]  pitch_sram_address;
logic        pitch_sram_chipselect;
logic        pitch_sram_write;
logic [7:0]  pitch_sram_writedata;   // not used here, just tie to 0
logic [7:0]  pitch_sram_readdata;

assign pitch_sram_address = coeff_sram_address;
assign pitch_sram_chipselect = coeff_sram_chipselect;
assign pitch_sram_write = coeff_sram_write;
assign pitch_sram_writedata = 4'b0;

assign cb13_sram_address = coeff_sram_address;
assign cb13_sram_chipselect = coeff_sram_chipselect;
assign cb13_sram_write = coeff_sram_write;
assign cb13_sram_writedata = 4'b0;
```

```
assign ca2_sram_address = coeff_sram_address;
assign ca2_sram_chipselect = coeff_sram_chipselect;
assign ca2_sram_write = coeff_sram_write;
assign ca2_sram_writedata = 4'b0;

assign ca3_sram_address = coeff_sram_address;
assign ca3_sram_chipselect = coeff_sram_chipselect;
assign ca3_sram_write = coeff_sram_write;
assign ca3_sram_writedata = 4'b0;

vc_TestSource
#(
  .p_msg_nbits ( 32 )
) src (
  .clk        (            clk ),
  .reset      (          reset ),

  .val        (     mngr2proc_val ),
  .rdy        (     mngr2proc_rdy ),
  .msg        (     mngr2proc_msg ),

  .done       (       src_done ),
        .num_msgs        (SOURCE_MAX),

// sram ports
  .source_sram_addr   (source_sram_address),
  .source_sram_en     (source_sram_chipselect),
  .source_sram_wen    (source_sram_write),
  .source_sram_wdata  (source_sram_writedata),   // not used here, just tie to 0
  .source_sram_rdata  (source_sram_readdata)
);


vc_TestSink
#(
  .p_msg_nbits ( 32 )
) sink (
  .clk        (            clk ),
  .reset      (          reset ),

  .val        (      proc2mngr_val ),
  .rdy        (      proc2mngr_rdy ),
  .msg        (      proc2mngr_msg ),

  .done       (       snk_done ),
        .num_msgs        (SINK_MAX),

  .sink_sram_addr   (sink_sram_address),
  .sink_sram_en     (sink_sram_chipselect),
  .sink_sram_wen    (sink_sram_write),
  .sink_sram_wdata  (sink_sram_writedata),
```

```
    .sink_sram_rdata  (sink_sram_readdata)
  );


proc_Proc
#(
  .p_num_cores(1)
)
DUT
(
  .clk(clk),
  .reset(reset),

  // From mngr streaming port

  .mngr2proc_msg(mngr2proc_msg),
  .mngr2proc_val(mngr2proc_val),
  .mngr2proc_rdy(mngr2proc_rdy),

  // To mngr streaming port

  .proc2mngr_msg(proc2mngr_msg),
  .proc2mngr_val(proc2mngr_val),
  .proc2mngr_rdy(proc2mngr_rdy),

  .xcel_reqstream_msg (xcel_reqstream_msg),
  .xcel_reqstream_val (xcel_reqstream_val),
  .xcel_reqstream_rdy (xcel_reqstream_rdy),

  .xcel_respstream_msg (xcel_respstream_msg),
  .xcel_respstream_val (xcel_respstream_val),
  .xcel_respstream_rdy (xcel_respstream_rdy),

  // Instruction Memory Request Port

  .imem_reqstream_msg(imem_reqstream_msg),
  .imem_reqstream_val(imem_reqstream_val),
  .imem_reqstream_rdy(imem_reqstream_rdy),

  // Instruction Memory Response Port

  .imem_respstream_msg(imem_respstream_msg),
  .imem_respstream_val(imem_respstream_val),
  .imem_respstream_rdy(imem_respstream_rdy),

  // Data Memory Request Port

  .dmem_reqstream_msg(dmem_reqstream_msg),
  .dmem_reqstream_val(dmem_reqstream_val),
  .dmem_reqstream_rdy(dmem_reqstream_rdy),
```

```
  // Data Memory Response Port

  .dmem_respstream_msg(dmem_respstream_msg),
  .dmem_respstream_val(dmem_respstream_val),
  .dmem_respstream_rdy(dmem_respstream_rdy),

  // extra ports; note that core_id is an input port rather than a
  // parameter so that the module only needs to be compiled once. If it
  // were a parameter, each core would be compiled separately.

  .core_id(0),
  .commit_inst(commit_inst),
  .stats_en(stats_en)
);

tut8_sram_SramMinionVRTL_Inst inst_mem
(
  .clk(clk),
  .reset(reset),

  // Memory request port interface -- inst
  .minion_req_val(imem_reqstream_val),
  .minion_req_rdy(imem_reqstream_rdy),
  .minion_req_msg(imem_reqstream_msg),

  // Memory response port interface -- inst
  .minion_resp_val(imem_respstream_val),
  .minion_resp_rdy(imem_respstream_rdy),
  .minion_resp_msg(imem_respstream_msg),

  // SRAM output interface
  .inst_sram_wen_M0         (inst_sram_write),
  .inst_sram_addr_M0        (inst_sram_address),
  .inst_sram_en_M0          (inst_sram_chipselect),
  .inst_memreq_msg_data_M0  (inst_sram_writedata),
  .inst_sram_read_data_M1   (inst_sram_readdata)
);

tut8_sram_SramMinionVRTL_Data data_mem (
  .clk(clk),
  .reset(reset),

  // Memory request port interface -- data
  .minion_req_val(dmem_reqstream_val),
  .minion_req_rdy(dmem_reqstream_rdy),
  .minion_req_msg(dmem_reqstream_msg),

  // Memory response port interface -- data
  .minion_resp_val(dmem_respstream_val),
  .minion_resp_rdy(dmem_respstream_rdy),
  .minion_resp_msg(dmem_respstream_msg),
```

```
   // SRAM interface
  .data_sram_addr_M0       (data_sram_address),
  .data_sram_wen_M0        (data_sram_write),
  .data_sram_en_M0         (data_sram_chipselect),
  .data_memreq_msg_data_M0  (data_sram_writedata),
  .data_sram_read_data_M1   (data_sram_readdata)
);

//=================== init coeff store buffer here ===================
logic [7:0] pitch_store_reg [0:31];

logic [31:0] cb13_store_reg [0:31];
logic [31:0] ca2_store_reg  [0:31];
logic [31:0] ca3_store_reg  [0:31];

logic [31:0] coeff_store_reg [0:31];

//=================== ============================ ==================


logic [2:0] state_ex_reg;

logic       iir_fsm_ex;
logic [31:0] tap_num_ex;

xcel_IIRXcel xcel_iir (
  .clk                (clk),
  .reset              (iir_reset),
  .xcel_reqstream_msg   (xcel_reqstream_msg),
  .xcel_reqstream_val   (xcel_reqstream_val),
  .xcel_reqstream_rdy   (xcel_reqstream_rdy),

  .xcel_respstream_msg  (xcel_respstream_msg),
  .xcel_respstream_val  (xcel_respstream_val),
  .xcel_respstream_rdy  (xcel_respstream_rdy),

  .state_reg           (state_ex_reg),
  .coeff_sram_addr      (coeff_sram_address),
  .coeff_sram_en        (coeff_sram_chipselect),
  .coeff_sram_wen       (coeff_sram_write),
  .coeff_sram_wdata     (coeff_sram_writedata), // not used, always set to 0
  .iir_fsm             (iir_fsm_ex),
  .tap_num              (tap_num_ex)
);

always_comb begin
  if (state_ex_reg == 3'b1)begin
    if (coeff_sram_address > 0) begin
      coeff_store_reg [coeff_sram_address - 1] = coeff_sram_readdata;
      pitch_store_reg [coeff_sram_address - 1] = pitch_sram_readdata;
```

```
      cb13_store_reg [coeff_sram_address - 1] = cb13_sram_readdata;
      ca2_store_reg [coeff_sram_address - 1]  = ca2_sram_readdata;
      ca3_store_reg [coeff_sram_address - 1]  = ca3_sram_readdata;

    end
  end
end

logic [2:0] count;
logic cnt_done;

wire inst_sram_clken;
wire data_sram_clken;
wire source_sram_clken;
wire sink_sram_clken;
wire coeff_sram_clken;
wire pitch_sram_clken;

wire cb13_sram_clken;
wire ca2_sram_clken ;
wire ca3_sram_clken ;

// -------------------- FSM for mem init and clk start --------------------
logic mem_done;

// ==================== FSM for the State Transition =======================
assign clk = CLOCK_50;
//assign LEDR = test_clk;

logic [3:0] trans_state;
logic [3:0] next_state;

//assign              inst_sram_clken = 1'b1;
//assign data_sram_clken = 1'b1;
//assign source_sram_clken = 1'b1;
//assign sink_sram_clken = 1'b1;

always_comb begin
    case (trans_state)
        4'd0: begin
                                inst_sram_clken = 1'b0;
                                data_sram_clken = 1'b0;
                                source_sram_clken = 1'b0;
                                sink_sram_clken = 1'b0;
                                coeff_sram_clken = 1'b0;
                                pitch_sram_clken = 1'b0;
              cb13_sram_clken  = 1'b0;
              ca2_sram_clken   = 1'b0;
              ca3_sram_clken   = 1'b0;

                                reset = 1'b1;
```

```
                        iir_reset = 1'b1;
    end
    4'd1: begin
                        inst_sram_clken = 1'b1;
                        data_sram_clken = 1'b1;
                        source_sram_clken = 1'b1;
                        sink_sram_clken = 1'b1;
                        coeff_sram_clken = 1'b1;
                        pitch_sram_clken = 1'b1;

        cb13_sram_clken  = 1'b1;
        ca2_sram_clken   = 1'b1;
        ca3_sram_clken   = 1'b1;

                        reset = 1'b0;
                        iir_reset = 1'b0;
    end
            4'd2: begin
                        inst_sram_clken = 1'b0;
                        data_sram_clken = 1'b0;
                        source_sram_clken = 1'b0;
                        sink_sram_clken = 1'b0;
                        coeff_sram_clken = 1'b0;
                        pitch_sram_clken = 1'b0;

        cb13_sram_clken  = 1'b0;
        ca2_sram_clken   = 1'b0;
        ca3_sram_clken   = 1'b0;

                        reset = 1'b1;
                        iir_reset = 1'b0;

            end
    default: begin
                        inst_sram_clken = 1'b0;
                        data_sram_clken = 1'b0;
                        source_sram_clken = 1'b0;
                        sink_sram_clken = 1'b0;
                        coeff_sram_clken = 1'b0;
                        pitch_sram_clken = 1'b0;

        cb13_sram_clken  = 1'b0;
        ca2_sram_clken   = 1'b0;
        ca3_sram_clken   = 1'b0;

                        reset = 1'b1;
                        iir_reset = 1'b0;
    end
  endcase
end
```

```
always_ff @(posedge CLOCK_50 or negedge KEY[0]) begin
   if (!KEY[0])
      trans_state <= 4'd0;
   else
      trans_state <= next_state;
end

always_comb begin
   case (trans_state)
      4'd0: next_state = mem_done ? 4'd1 : 4'd0;
      4'd1: next_state = cnt_done ? 4'd2 : 4'd1;
      4'd2: next_state = (!restart_sig) ? 4'd2 : 4'd0;
      default: next_state = 4'd0;
   endcase
end

  vc_BasicCounter #(
    .p_count_nbits(3),
    .p_count_clear_value(0),   //
    .p_count_max_value(7)      //
  ) my_counter (
    .clk(clk),
    .reset(~KEY[0]),
    .clear(1'b0),              //
    .increment(snk_done),          //
    .decrement(1'b0),          //
    .count(count),
    .count_is_zero(),
    .count_is_max(cnt_done)        //
  );

//=======================================================
// Bus controller for AVALON bus-master
//=======================================================
// computes DDS for sine wave and fills audio FIFO
// reads audio FIFO and loops it back
// configure (in Qsys) Audio Config module:
// -- Line in to ADC
// The audio_input_ready signal goes high for one
// cycle when there is new audio input data
// --
// 32-bit data is on
//   right_audio_input, left_audio_input ;
// Every write requires 32-bit data on
//   right_audio_output, left_audio_output ;
wire [15:0] right_filter_output , left_filter_output;

wire [15:0] right_filter_output_arr [0:31];
wire [15:0] left_filter_output_arr  [0:31];

reg [31:0] bus_addr ; // Avalon address
```

```
// see
//
ftp://ftp.altera.com/up/pub/Altera_Material/15.1/University_Program_IP_Cores/Audio_Video/Au
dio.pdf
// for addresses
wire [31:0] audio_base_address = 32'h00003040 ;  // Avalon address
wire [31:0] audio_fifo_address = 32'h00003044 ;  // Avalon address +4 offset
wire [31:0] audio_data_left_address = 32'h00003048 ;  // Avalon address +8
wire [31:0] audio_data_right_address = 32'h0000304c ;  // Avalon address +12
reg [3:0] bus_byte_enable ; // four bit byte read/write mask
reg bus_read  ;       // high when requesting data
reg bus_write ;      //  high when writing data
reg [31:0] bus_write_data ; //  data to send to Avalog bus
wire bus_ack;       //  Avalon bus raises this when done
wire [31:0] bus_read_data ; // data from Avalon bus
reg [30:0] timer ;
reg [3:0] state ;
wire state_clock ;
wire reset_audio;

assign reset_audio = ~KEY[0];

// current free words in audio interface
reg [7:0] fifo_space ;
// debug check space
assign LEDR = fifo_space ;

// audio input/output from audio module FIFO
reg [15:0] right_audio_input, left_audio_input ;
reg audio_input_ready ;
wire [15:0] right_audio_output, left_audio_output ;

// For audio loopback, or filtering
assign right_audio_output = SW[0] ? right_filter_output : right_audio_input ;

assign left_audio_output =  SW[0] ? left_filter_output : left_audio_input ;

wire [2:0] shift_pitch = SW[3:1];

// get some signals exposed
// connect bus master signals to i/o for probes
assign GPIO_0[0] = bus_write ;
assign GPIO_0[1] = bus_read ;
assign GPIO_0[2] = bus_ack ;
assign GPIO_0[3] = audio_input_ready ;

assign GPIO_0[4] = right_audio_input ;
assign GPIO_0[5] = right_filter_output;
assign GPIO_0[7] = left_filter_output;

// ============================================================
```

```
// === Filters ============================================
// ========================================================

wire [15:0] right_filter_sum, left_filter_sum;
integer jr, jl;

 always_comb begin
  right_filter_sum = 0;
  for (jr = 0; jr < 32; jr = jr + 1) begin
    if ((tap_num_ex == 32 && jr < 32) ||
        (tap_num_ex == 16 && jr < 16) ||
        (tap_num_ex ==  8 && jr <  8) ||
                    (tap_num_ex ==  1 && jl <  1) ) begin
      right_filter_sum = right_filter_sum + right_filter_output_arr[jr];
    end
  end
end
// get the summed output
assign right_filter_output = right_filter_sum;


 always_comb begin
  left_filter_sum = 0;
  for (jl = 0; jl < 32; jl = jl + 1) begin
    if ((tap_num_ex == 32 && jl < 32) ||
        (tap_num_ex == 16 && jl < 16) ||
        (tap_num_ex ==  8 && jl <  8) ||
                    (tap_num_ex ==  1 && jl <  1) ) begin
      left_filter_sum = left_filter_sum + left_filter_output_arr[jl];
    end
  end
end
// get the summed output
assign left_filter_output = left_filter_sum;

wire [5:0] base_lpf = {1'd0,SW[9:6]};

genvar i;
generate
  for (i = 0; i < 32; i = i + 1) begin : gen_filters
    // i==0 → base_alpha+5, i==1 → base_alpha+4, …, i==4 → base_alpha+1, other→
base_alpha
    wire [5:0] this_lpf = (i == 0) ? (base_lpf + 5'd5) :
                    (i == 1) ? (base_lpf + 5'd4) :
                    (i == 2) ? (base_lpf + 5'd3) :
                    (i == 3) ? (base_lpf + 5'd2) :
                    (i == 4) ? (base_lpf + 5'd1) :
                               base_lpf;

    // --- right channel ---
    IIR2_18bit_fixed filter_right (
```

```
    .audio_out          ( right_filter_output_arr[i]      ),
    .audio_in           ( right_audio_input               ),
    .b1                 ( cb13_store_reg[i][17:0]         ),
    .b2                 ( 18'sd0                          ),
    .b3                 ( -$signed(cb13_store_reg[i][17:0])  ),
    .a2                 ( ca2_store_reg[i][17:0]          ),
    .a3                 ( ca3_store_reg[i][17:0]          ),
    .sine_mo            ( coeff_store_reg[i] <<< pitch_store_reg[i] ),
    .lpf_coeff          ( this_lpf                        ),
    .state_clk          ( CLOCK_50                        ),
    .audio_input_ready  ( audio_input_ready               ),
    .reset              ( reset_audio                     )
  );

  // --- left channel ---
  IIR2_18bit_fixed filter_left (
    .audio_out          ( left_filter_output_arr[i]       ),
    .audio_in           ( left_audio_input                ),
    .b1                 ( cb13_store_reg[i][17:0]         ),
    .b2                 ( 18'sd0                          ),
    .b3                 ( -$signed(cb13_store_reg[i][17:0])  ),
    .a2                 ( ca2_store_reg[i][17:0]          ),
    .a3                 ( ca3_store_reg[i][17:0]          ),
    .sine_mo            ( coeff_store_reg[i] <<< pitch_store_reg[i] ),
    .lpf_coeff          ( this_lpf                        ),
    .state_clk          ( CLOCK_50                        ),
    .audio_input_ready  ( audio_input_ready               ),
    .reset              ( reset_audio                     )
  );
 end
endgenerate

Computer_System The_System (
        ////////////////////////////////////
        // FPGA Side
        ////////////////////////////////////

        // PIO
        .done_sig_ext_export(cnt_done),
        .mem_init_sig_ext_export(mem_done),
        .source_max_ext_export(SOURCE_MAX),
        .sink_max_ext_export(SINK_MAX),
        .restart_sig_ext_export(restart_sig),
        //SRAM M10K units
        // inst_sram
        .inst_sram_s1_address    (inst_sram_address),                    //
onchip_sram_s1.address
        .inst_sram_s1_clken      (inst_sram_clken),                      //                          .clken
    .inst_sram_s1_chipselect (inst_sram_chipselect),
//                           .chipselect
        .inst_sram_s1_write                (inst_sram_write),
```

```
//                      .write
        .inst_sram_s1_readdata   (inst_sram_readdata),
//                      .readdata
        .inst_sram_s1_writedata (inst_sram_writedata),
//                      .writedata
        .inst_sram_s1_byteenable         (4'b_1111),
//                      .byteenable

        // data_sram
        .data_sram_s1_address                   (data_sram_address),            //
data_sram_s1.address
        .data_sram_s1_clken                     (data_sram_clken),
//                      .clken
        .data_sram_s1_chipselect                (data_sram_chipselect),
//                      .chipselect
        .data_sram_s1_write                             (data_sram_write),
//                      .write
        .data_sram_s1_readdata                  (data_sram_readdata),
//                      .readdata
        .data_sram_s1_writedata                         (data_sram_writedata),
//                      .writedata
        .data_sram_s1_byteenable                (4'b_1111),

        // source_sram
        .source_sram_s1_address                         (source_sram_address),
//                  data_sram_s1.address
        .source_sram_s1_clken                           (source_sram_clken),
//                      .clken
        .source_sram_s1_chipselect                      (source_sram_chipselect),
//                      .chipselect
        .source_sram_s1_write                           (source_sram_write),
//                      .write
        .source_sram_s1_readdata                        (source_sram_readdata),
//                      .readdata
        .source_sram_s1_writedata                       (source_sram_writedata),
//                      .writedata
        .source_sram_s1_byteenable                      (4'b_1111),

        // sink_sram
        .sink_sram_s1_address                   (sink_sram_address),            //
data_sram_s1.address
        .sink_sram_s1_clken                     (sink_sram_clken),
//                      .clken
        .sink_sram_s1_chipselect                    (sink_sram_chipselect),
//                      .chipselect
        .sink_sram_s1_write                             (sink_sram_write),
//                      .write
        .sink_sram_s1_readdata                  (sink_sram_readdata),
//                      .readdata
        .sink_sram_s1_writedata                         (sink_sram_writedata),
//                      .writedata
```

```
        .sink_sram_s1_byteenable                    (4'b_1111),

    .ca2_sram_s1_address            (ca2_sram_address),                 //
ca2_sram_s1.address
        .ca2_sram_s1_clken              (ca2_sram_clken),
//                      .clken
        .ca2_sram_s1_chipselect          (ca2_sram_chipselect),
//                      .chipselect
        .ca2_sram_s1_write              (ca2_sram_write),
//                      .write
        .ca2_sram_s1_readdata            (ca2_sram_readdata),
//                      .readdata
        .ca2_sram_s1_writedata           (ca2_sram_writedata),
//                      .writedata
        .ca2_sram_s1_byteenable          (4'b_1111),
//                      .byteenable

    .ca3_sram_s1_address            (ca3_sram_address),                 //
ca3_sram_s1.address
        .ca3_sram_s1_clken              (ca3_sram_clken),
//                      .clken
        .ca3_sram_s1_chipselect          (ca3_sram_chipselect),
//                      .chipselect
        .ca3_sram_s1_write              (ca3_sram_write),
//                      .write
        .ca3_sram_s1_readdata            (ca3_sram_readdata),
//                      .readdata
        .ca3_sram_s1_writedata           (ca3_sram_writedata),
//                      .writedata
        .ca3_sram_s1_byteenable          (4'b_1111),
//                      .byteenable

    .cb13_sram_s1_address           (cb13_sram_address),                //
cb13_sram_s1.address
        .cb13_sram_s1_clken             (cb13_sram_clken),
//                      .clken
    .cb13_sram_s1_chipselect         (cb13_sram_chipselect),
//                      .chipselect
    .cb13_sram_s1_write             (cb13_sram_write),
//                      .write
        .cb13_sram_s1_readdata           (cb13_sram_readdata),
//                      .readdata
        .cb13_sram_s1_writedata          (cb13_sram_writedata),
//                      .writedata
        .cb13_sram_s1_byteenable         (4'b_1111),
//                      .byteenable

    .coeff_sram_s1_address          (coeff_sram_address),               //
coeff_sram_s1.address
    .coeff_sram_s1_clken            (coeff_sram_clken),
//                      .clken
```

```
    .coeff_sram_s1_chipselect           (coeff_sram_chipselect),
//                      .chipselect
    .coeff_sram_s1_write                (coeff_sram_write),
//                      .write
        .coeff_sram_s1_readdata             (coeff_sram_readdata),
//                      .readdata
        .coeff_sram_s1_writedata            (coeff_sram_writedata),
//                      .writedata
    .coeff_sram_s1_byteenable           (4'b_1111),

    .pitch_sram_s1_address              (pitch_sram_address),                   //
pitch_sram_s1.address
    .pitch_sram_s1_clken                (pitch_sram_clken),
//                      .clken
    .pitch_sram_s1_chipselect           (pitch_sram_chipselect),
//                      .chipselect
    .pitch_sram_s1_write                (pitch_sram_write),
//                      .write
        .pitch_sram_s1_readdata             (pitch_sram_readdata),
//                      .readdata
        .pitch_sram_s1_writedata            (pitch_sram_writedata),
//                      .writedata
        // Global signals
        .system_pll_ref_clk_clk                                 (CLOCK_50),
        .system_pll_ref_reset_reset                     (1'b0),
        .sdram_clk_clk                                                  (state_clock),

        // AV Config
        .av_config_SCLK
        (FPGA_I2C_SCLK),
        .av_config_SDAT
        (FPGA_I2C_SDAT),

        // Audio Subsystem
        .audio_pll_ref_clk_clk                                  (CLOCK3_50),
        .audio_pll_ref_reset_reset                              (1'b0),
        .audio_clk_clk                                                  (AUD_XCK),
        .audio_ADCDAT
        (AUD_ADCDAT),
        .audio_ADCLRCK
        (AUD_ADCLRCK),
        .audio_BCLK
        (AUD_BCLK),
        .audio_DACDAT
        (AUD_DACDAT),
        .audio_DACLRCK
        (AUD_DACLRCK),

        // bus-master state machine interface
        .bus_master_audio_external_interface_address     (bus_addr),
        .bus_master_audio_external_interface_byte_enable (bus_byte_enable),
```

```
.bus_master_audio_external_interface_read       (bus_read),
.bus_master_audio_external_interface_write      (bus_write),
.bus_master_audio_external_interface_write_data (bus_write_data),
.bus_master_audio_external_interface_acknowledge (bus_ack),
.bus_master_audio_external_interface_read_data  (bus_read_data),


///////////////////////////////////
// HPS Side
///////////////////////////////////
// DDR3 SDRAM
.memory_mem_a                   (HPS_DDR3_ADDR),
.memory_mem_ba                  (HPS_DDR3_BA),
.memory_mem_ck                  (HPS_DDR3_CK_P),
.memory_mem_ck_n            (HPS_DDR3_CK_N),
.memory_mem_cke             (HPS_DDR3_CKE),
.memory_mem_cs_n            (HPS_DDR3_CS_N),
.memory_mem_ras_n           (HPS_DDR3_RAS_N),
.memory_mem_cas_n           (HPS_DDR3_CAS_N),
.memory_mem_we_n            (HPS_DDR3_WE_N),
.memory_mem_reset_n (HPS_DDR3_RESET_N),
.memory_mem_dq                  (HPS_DDR3_DQ),
.memory_mem_dqs             (HPS_DDR3_DQS_P),
.memory_mem_dqs_n           (HPS_DDR3_DQS_N),
.memory_mem_odt             (HPS_DDR3_ODT),
.memory_mem_dm                  (HPS_DDR3_DM),
.memory_oct_rzqin           (HPS_DDR3_RZQ),

// Ethernet
.hps_io_hps_io_gpio_inst_GPIO35     (HPS_ENET_INT_N),
.hps_io_hps_io_emac1_inst_TX_CLK    (HPS_ENET_GTX_CLK),
.hps_io_hps_io_emac1_inst_TXD0      (HPS_ENET_TX_DATA[0]),
.hps_io_hps_io_emac1_inst_TXD1      (HPS_ENET_TX_DATA[1]),
.hps_io_hps_io_emac1_inst_TXD2      (HPS_ENET_TX_DATA[2]),
.hps_io_hps_io_emac1_inst_TXD3      (HPS_ENET_TX_DATA[3]),
.hps_io_hps_io_emac1_inst_RXD0      (HPS_ENET_RX_DATA[0]),
.hps_io_hps_io_emac1_inst_MDIO      (HPS_ENET_MDIO),
.hps_io_hps_io_emac1_inst_MDC           (HPS_ENET_MDC),
.hps_io_hps_io_emac1_inst_RX_CTL    (HPS_ENET_RX_DV),
.hps_io_hps_io_emac1_inst_TX_CTL    (HPS_ENET_TX_EN),
.hps_io_hps_io_emac1_inst_RX_CLK    (HPS_ENET_RX_CLK),
.hps_io_hps_io_emac1_inst_RXD1      (HPS_ENET_RX_DATA[1]),
.hps_io_hps_io_emac1_inst_RXD2      (HPS_ENET_RX_DATA[2]),
.hps_io_hps_io_emac1_inst_RXD3      (HPS_ENET_RX_DATA[3]),

// Flash
.hps_io_hps_io_qspi_inst_IO0    (HPS_FLASH_DATA[0]),
.hps_io_hps_io_qspi_inst_IO1    (HPS_FLASH_DATA[1]),
.hps_io_hps_io_qspi_inst_IO2    (HPS_FLASH_DATA[2]),
.hps_io_hps_io_qspi_inst_IO3    (HPS_FLASH_DATA[3]),
.hps_io_hps_io_qspi_inst_SS0    (HPS_FLASH_NCSO),
```

```
.hps_io_hps_io_qspi_inst_CLK  (HPS_FLASH_DCLK),

// Accelerometer
.hps_io_hps_io_gpio_inst_GPIO61       (HPS_GSENSOR_INT),

//.adc_sclk                  (ADC_SCLK),
//.adc_cs_n                   (ADC_CS_N),
//.adc_dout                   (ADC_DOUT),
//.adc_din                   (ADC_DIN),

// General Purpose I/O
.hps_io_hps_io_gpio_inst_GPIO40       (HPS_GPIO[0]),
.hps_io_hps_io_gpio_inst_GPIO41       (HPS_GPIO[1]),

// I2C
.hps_io_hps_io_gpio_inst_GPIO48       (HPS_I2C_CONTROL),
.hps_io_hps_io_i2c0_inst_SDA          (HPS_I2C1_SDAT),
.hps_io_hps_io_i2c0_inst_SCL          (HPS_I2C1_SCLK),
.hps_io_hps_io_i2c1_inst_SDA          (HPS_I2C2_SDAT),
.hps_io_hps_io_i2c1_inst_SCL          (HPS_I2C2_SCLK),

// Pushbutton
.hps_io_hps_io_gpio_inst_GPIO54       (HPS_KEY),

// LED
.hps_io_hps_io_gpio_inst_GPIO53       (HPS_LED),

// SD Card
.hps_io_hps_io_sdio_inst_CMD (HPS_SD_CMD),
.hps_io_hps_io_sdio_inst_D0    (HPS_SD_DATA[0]),
.hps_io_hps_io_sdio_inst_D1    (HPS_SD_DATA[1]),
.hps_io_hps_io_sdio_inst_CLK  (HPS_SD_CLK),
.hps_io_hps_io_sdio_inst_D2    (HPS_SD_DATA[2]),
.hps_io_hps_io_sdio_inst_D3    (HPS_SD_DATA[3]),

// SPI
.hps_io_hps_io_spim1_inst_CLK              (HPS_SPIM_CLK),
.hps_io_hps_io_spim1_inst_MOSI       (HPS_SPIM_MOSI),
.hps_io_hps_io_spim1_inst_MISO       (HPS_SPIM_MISO),
.hps_io_hps_io_spim1_inst_SS0          (HPS_SPIM_SS),

// UART
.hps_io_hps_io_uart0_inst_RX  (HPS_UART_RX),
.hps_io_hps_io_uart0_inst_TX  (HPS_UART_TX),

// USB
.hps_io_hps_io_gpio_inst_GPIO09       (HPS_CONV_USB_N),
.hps_io_hps_io_usb1_inst_D0           (HPS_USB_DATA[0]),
.hps_io_hps_io_usb1_inst_D1           (HPS_USB_DATA[1]),
.hps_io_hps_io_usb1_inst_D2           (HPS_USB_DATA[2]),
.hps_io_hps_io_usb1_inst_D3           (HPS_USB_DATA[3]),
```

```
        .hps_io_hps_io_usb1_inst_D4         (HPS_USB_DATA[4]),
        .hps_io_hps_io_usb1_inst_D5         (HPS_USB_DATA[5]),
        .hps_io_hps_io_usb1_inst_D6         (HPS_USB_DATA[6]),
        .hps_io_hps_io_usb1_inst_D7         (HPS_USB_DATA[7]),
        .hps_io_hps_io_usb1_inst_CLK        (HPS_USB_CLKOUT),
        .hps_io_hps_io_usb1_inst_STP        (HPS_USB_STP),
        .hps_io_hps_io_usb1_inst_DIR        (HPS_USB_DIR),
        .hps_io_hps_io_usb1_inst_NXT        (HPS_USB_NXT)
);
endmodule

//////////////////////////////////////////////////
////////////////// signed multy 2.16 ////////////
//////////////////////////////////////////////////
module signed_mult (out, a, b);

        output  signed  [17:0]  out;
        input   signed  [17:0]  a;
        input   signed  [17:0]  b;

        wire    signed  [35:0]  mult_result_out;

        assign mult_result_out = a * b;
        assign out = {mult_result_out[35], mult_result_out[32:16]};
endmodule
//////////////////////////////////////////////////


////////////////////////////////////////////////////////////////
/// Second order IIR filter ////////////////////////////////////
////////////////////////////////////////////////////////////////
module IIR2_18bit_fixed (audio_out, audio_in,
                         b1, b2, b3,
                         a2, a3,
                         sine_mo,
                         lpf_coeff,
                         state_clk, audio_input_ready, reset) ;

always @ (posedge state_clk)
begin
endmodule
//////////////////////////////////////////////////
////////////        Sin Wave ROM Table   //////////////
//////////////////////////////////////////////////
// produces a 2's comp, 16-bit, approximation
// of a sine wave, given an input phase (address)
module sync_rom (clock, address, sine);
input clock;
input [7:0] address;
output [15:0] sine;
reg signed [15:0] sine;
```

```
always@(posedge clock)
begin
   case(address)
         endcase
end
endmodule

////////////////////////////////////////////////
// === References/Links ======================
////////////////////////////////////////////////
/*
Used the IIR filter code provided by the Dr.Bruce Land as the foundation web:
https://people.ece.cornell.edu/land/courses/ece5760/DE2/fpgaDSP.html
DSP web:
https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/HPS_peripherials/DSP_index.htm
l
The sinewave generation code is inherited from the lorenz project provided by the Dr.Hunter
Adams
Used the Pitch shift code provided by the previouse year FPGA Speech Vocoder:
https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/s2019/jc2697_jaj263_tk455/jc2
697_jaj263_tk455/jc2697_jaj263_tk455/index.html
*/
```

c. RISC-V Processor testing code:

```
`timescale 1ps/1ps

`include "ProcBase.sv"
// `include "trace.sv"
`include "SramMinionVRTL_data.sv"
`include "SramMinionVRTL_inst.sv"
// `include "vc/TestRandDelaySource.v"
// `include "vc/TestRandDelaySink.v"
`include "TestSink.sv"
`include "TestSource.sv"

//------------------------------------------------------------------------
// Top-level module
//------------------------------------------------------------------------
module top;

  logic        reset;
  logic        clk;
  logic [31:0]  mngr2proc_msg;
  logic        mngr2proc_val;
  logic        mngr2proc_rdy;

  // To mngr streaming port

  logic [31:0]  proc2mngr_msg;
```

```
logic        proc2mngr_val;
logic        proc2mngr_rdy;

// Instruction Memory Request Port

mem_req_4B_t  imem_reqstream_msg;
logic        imem_reqstream_val;
logic        imem_reqstream_rdy;

// Instruction Memory Response Port

mem_resp_4B_t imem_respstream_msg;
logic        imem_respstream_val;
logic        imem_respstream_rdy;

// Data Memory Request Port

mem_req_4B_t  dmem_reqstream_msg;
logic        dmem_reqstream_val;
logic        dmem_reqstream_rdy;

// Data Memory Response Port

mem_resp_4B_t dmem_respstream_msg;
logic        dmem_respstream_val;
logic        dmem_respstream_rdy;

// extra ports; note that core_id is an input port rather than a
// parameter so that the module only needs to be compiled once. If it
// were a parameter, each core would be compiled separately.

logic [31:0]  core_id;
logic        commit_inst;
logic        stats_en;

logic src_done;
logic snk_done;
logic mem_clear;

// localparam SINK_SOURCE_MAX = 10;
localparam SOURCE_MAX = 3;//2
localparam SINK_MAX = 2;//1
logic [  31:0 ] src_msgs [ SOURCE_MAX-1:0 ];
logic [ 31:0 ] snk_msgs [ SINK_MAX-1:0 ];

vc_TestSource
#(
  .p_msg_nbits ( 32 ),
  .p_num_msgs  ( SOURCE_MAX )
) src (
  .clk        (            clk ),
```

```
  .reset       (           reset ),

  .val         (     mngr2proc_val ),
  .rdy         (     mngr2proc_rdy ),
  .msg         (     mngr2proc_msg ),

  .done        (        src_done )
);



vc_TestSink
#(
  .p_msg_nbits ( 32 ),
  .p_num_msgs  ( SINK_MAX )
) sink (
  .clk         (             clk ),
  .reset       (           reset ),

  .val         (      proc2mngr_val ),
  .rdy         (      proc2mngr_rdy ),
  .msg         (      proc2mngr_msg ),

  .done        (         snk_done )
);



lab2_proc_ProcBase
#(
  .p_num_cores(1)
)
DUT
(
  .clk(clk),
  .reset(reset),

  // From mngr streaming port

  .mngr2proc_msg(mngr2proc_msg),
  .mngr2proc_val(mngr2proc_val),
  .mngr2proc_rdy(mngr2proc_rdy),

  // To mngr streaming port

  .proc2mngr_msg(proc2mngr_msg),
  .proc2mngr_val(proc2mngr_val),
  .proc2mngr_rdy(proc2mngr_rdy),

  // Instruction Memory Request Port
```

```
  .imem_reqstream_msg(imem_reqstream_msg),
  .imem_reqstream_val(imem_reqstream_val),
  .imem_reqstream_rdy(imem_reqstream_rdy),

  // Instruction Memory Response Port

  .imem_respstream_msg(imem_respstream_msg),
  .imem_respstream_val(imem_respstream_val),
  .imem_respstream_rdy(imem_respstream_rdy),

  // Data Memory Request Port

  .dmem_reqstream_msg(dmem_reqstream_msg),
  .dmem_reqstream_val(dmem_reqstream_val),
  .dmem_reqstream_rdy(dmem_reqstream_rdy),

  // Data Memory Response Port

  .dmem_respstream_msg(dmem_respstream_msg),
  .dmem_respstream_val(dmem_respstream_val),
  .dmem_respstream_rdy(dmem_respstream_rdy),

  // extra ports; note that core_id is an input port rather than a
  // parameter so that the module only needs to be compiled once. If it
  // were a parameter, each core would be compiled separately.

  .core_id(0),
  .commit_inst(commit_inst),
  .stats_en(stats_en)

);

tut8_sram_SramMinionVRTL_Inst inst_mem
(
  .clk(clk),
  .reset(reset),

  // Memory request port interface -- inst
  .minion_req_val(imem_reqstream_val),
  .minion_req_rdy(imem_reqstream_rdy),
  .minion_req_msg(imem_reqstream_msg),

  // Memory response port interface -- inst
  .minion_resp_val(imem_respstream_val),
  .minion_resp_rdy(imem_respstream_rdy),
  .minion_resp_msg(imem_respstream_msg)
);


tut8_sram_SramMinionVRTL_Data data_mem (
  .clk(clk),
```

```
  .reset(reset),

  // Memory request port interface -- data
  .minion_req_val(dmem_reqstream_val),
  .minion_req_rdy(dmem_reqstream_rdy),
  .minion_req_msg(dmem_reqstream_msg),

  // Memory response port interface -- data
  .minion_resp_val(dmem_respstream_val),
  .minion_resp_rdy(dmem_respstream_rdy),
  .minion_resp_msg(dmem_respstream_msg)
);

integer idx = 0;
// string temp ;
initial begin
  reset=1;
  // mem_clear=1;
  #10
  // ------------- manully initialize the value ------
  // initialize mem test --- expected out=9 ok
  src.m[0] = 32'h00002008;
  src.m[1] = 32'h00000005;
  src.m[2] = 32'h00000002;
  inst_mem.sraminst.srami.mem[0] =  32'hFC0020F3 ;
  inst_mem.sraminst.srami.mem[1] =  32'hFC002173 ;
  inst_mem.sraminst.srami.mem[2] =  32'hFC002273 ;
  inst_mem.sraminst.srami.mem[3] =  32'h00410233 ;
  inst_mem.sraminst.srami.mem[4] =  32'h0040A023 ;
  inst_mem.sraminst.srami.mem[5] =  32'h00120213 ;
  inst_mem.sraminst.srami.mem[6] =  32'h0000A283 ;
  inst_mem.sraminst.srami.mem[7] =  32'h7C021073 ;
  inst_mem.sraminst.srami.mem[8] =  32'h7C029073 ;
  inst_mem.sraminst.srami.mem[9] =  32'hFC0020F3 ;
  data_mem.sramdata.sramd.mem[0] =  32'h000041A7 ;
  data_mem.sramdata.sramd.mem[1] =  32'h60B7ACD9 ;

  #10
  reset=1;
  // mem_clear=1;
  #10
  reset=0;
  // mem_clear=0;
end

// wait for input and output
initial begin

  @(negedge clk);
  @(negedge clk);
  @(negedge clk);
```

```
  // while((!snk_done))begin
  //   @(negedge clk);
  //   $display("while test");
  // end

  while((!src_done) | (!snk_done) ) @(negedge clk);
  @(negedge clk);
  @(negedge clk);
  @(negedge clk);
  @(negedge clk);
  @(negedge clk);
  @(negedge clk);
  @(negedge clk);
  @(negedge clk);
  @(negedge clk);


  $display("Testbench finished.  Dumping memory");
  $finish();
end

// time out
initial begin
  for( integer i = 0; i < 20000; i = i + 1 ) begin
    @( negedge clk );
  end
  $display( "TIMEOUT: Testbench didn't finish in time" );
  $finish;
end

// dump waveform
initial begin
  $dumpfile("waveform.vcd");
  $dumpvars(0, top);            // Record the signals of the top module and all submodules
end

initial begin
  clk = 0;
end
  always #5 clk = ~clk;
endmodule

//////////////////////////////////////////////////
// === References/Links ======================
//////////////////////////////////////////////////
/*
Used the cornell ece4750 lab2 built 5 stage pipelined RISC-V processor. Web:
https://www.csl.cornell.edu/courses/ece4750/handouts/ece4750-lab2-proc.pdf
```

Other code: will be shared privately with the supervisor.