RISC-V Microprocessor & DSP Element

Team Sachima

Ashwin K. Avula, Yucheng Chang, Garry Chen, Alvin Cheng

ECE 554: Project Proposal

University of Wisconsin-Madison

Table of Contents

Introduction

This project aims to design and implement a microprocessor based on the RISC-V instruction set architecture, along with an integrated digital signal processing (DSP) coprocessor. The processor will feature a Harvard architecture for efficient data transfer via bootloading. It will include memory-mapped I/O for interfacing with peripherals such as LEDs, switches, buttons, a UART module, a VGA bitmap controller, and the DSP coprocessor. The DSP coprocessor will perform efficient image processing tasks, while the VGA interface will allow original and processed images to be displayed on a monitor. The project will also develop essential software components, such as application software to showcase the hardware capabilities, and scripts to support image buffering and loading. By combining a RISC-V CPU core with a DSP co-processor, various interfaces, and image processing capabilities, this project seeks to demonstrate the strengths of the hardware design in multimedia applications that involve processing and displaying visual data.

Hardware Block Diagram

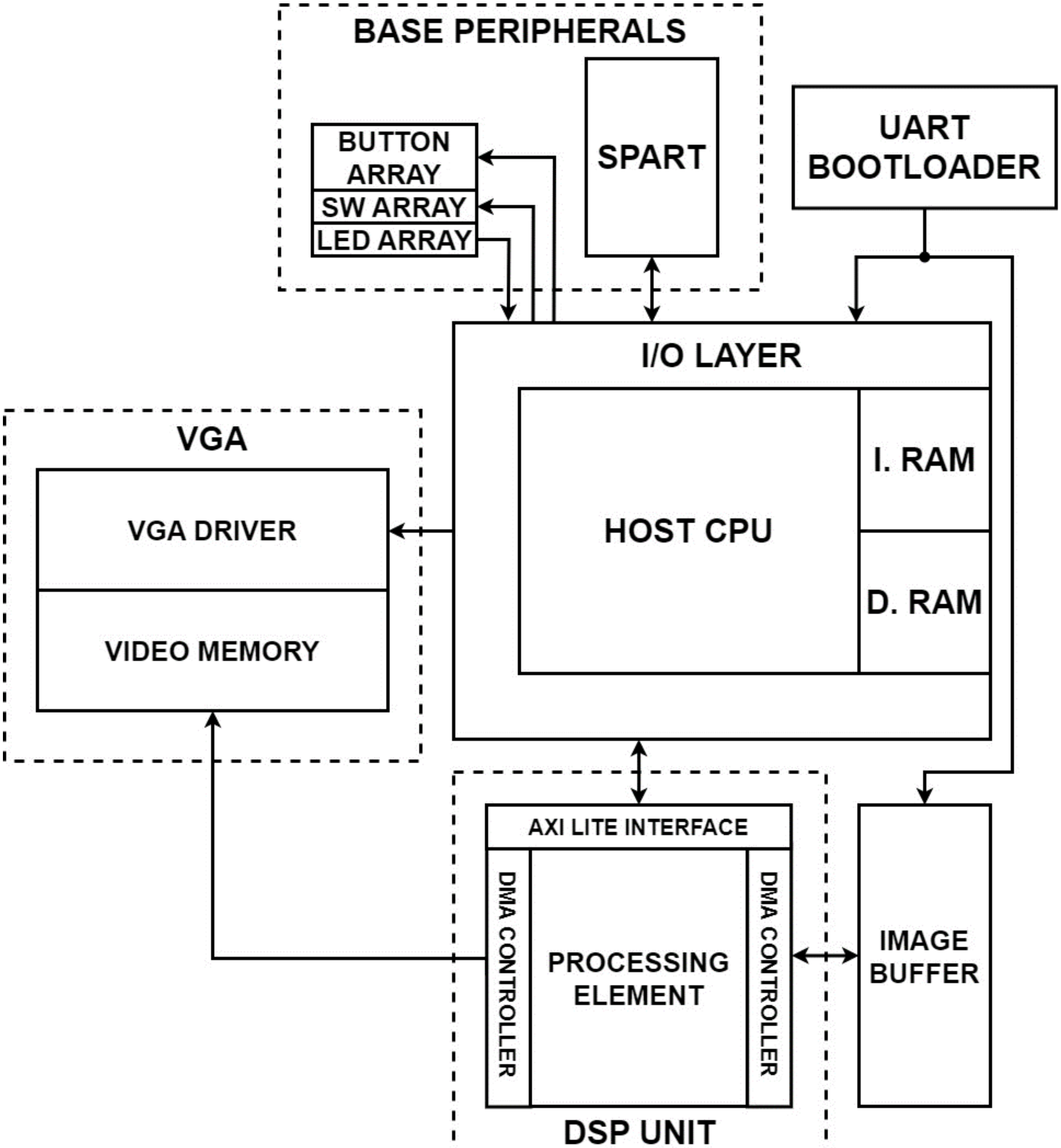

Figure 1: Top-Level RISC-V Microprocessor with DSP Element

As shown in *Figure 1* above, the HOST CPU implements the Base RISC-V instruction set architecture with discrete data and instruction memories. With an I/O Layer, the CPU can interact with peripherals via memory-mapped I/O Registers. Notable peripherals include basic switch and button arrays, two UART Peripherals for debug capabilities (SPART) and boot loading to memory, a VGA driver for output to a monitor, and finally the DSP Unit for acceleration of image processing applications.

I/O Layer Specification

Table 1 an initial I/O Layer Specification for which the HOST CPU interfaces with peripherals.

Memory Mapped Address	Function	CPU I/O Direction
0xC000	LED Array [9:0]	Output
0xC001	SW Array [9:0]	Input
0xC002	Button Array [3:0]	Input
0xC004	SPART Transmit Buffer [7:0]	Input / Output
0xC005	SPART Status Register [7:0]	Input
0xC006	SPART Division Buffer Low Byte [7:0]	Output
0xC007	SPART Division Buffer Low Byte [7:0]	Output
0xC008	VGA BMP Control Register [15:0]	Output
0xC009	VGA BMP X-Location [9:0]	Output
0xC00A	VGA BMP Y-Location [8:0]	Output
0xC00B	VGA BMP Status Register [15:0]	Input
0xC00C	Coprocessor Status Register [7:0]	Input
0xC00D	Coprocessor Address Register [7:0]	Output
0xC00E	Coprocessor Data Register [15:0]	Input / Output

Table 1: CPU Memory-Mapped Registers

DSP Coprocessor

The DSP Unit is a dedicated peripheral designed to accelerate specific image processing algorithms. The unit interfaces with the HOST CPU using an I/O Interface that modifies the AXI4-Lite communication protocol. In brief, in the AXI-Lite protocol, an AXI Master provides an ADDRESS and corresponding DATA line and expects a RESPONSE, as well as returning DATA from the AXI Slave. This protocol can be observed in *Figure 2*.

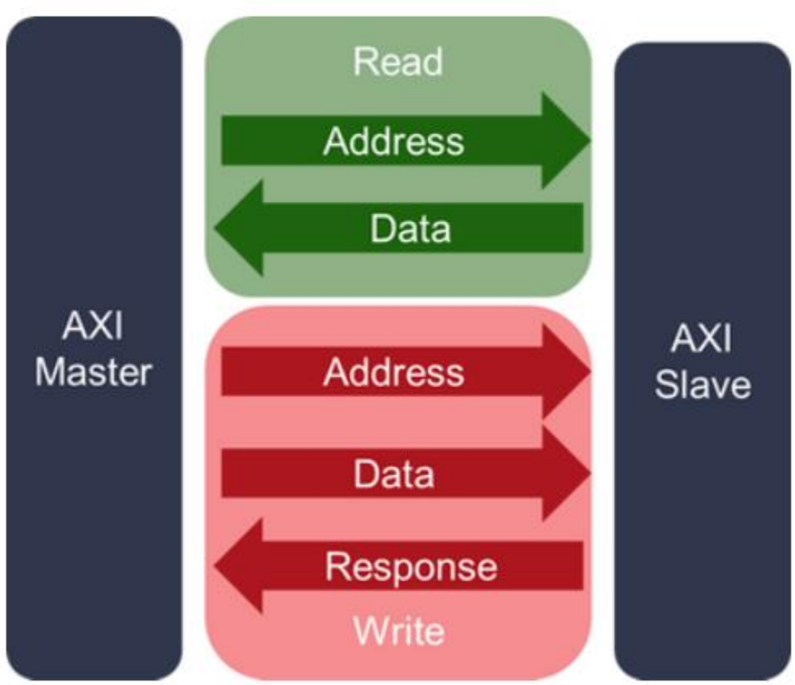

Figure 2: Simplified AXI4-Lite Protocol

To replicate this protocol, Coprocessor Address, Data, and Status Registers will be created in the HOST CPU Memory-Mapped region to monitor and control the coprocessor. Details of the memory-mapped field contents are provided below in *Table 2*.

Memory-Mapped Coprocessor Register	**Memory Address**	**Register Field Function**
Status	0xC00C	{4'b0, SLAVE ERROR, SLAVE DONE, SLAVE BUSY}
Address	0xC00D	{4'b0, HOST R/W_n, INPUT IMAGE INDEX [2:0]}
Data	0xC00E	{3'b0, RGB_or_GRAY, SLAVE FUNCTION [2:0]} or {RESULT IMAGE ADDRESS[15:0]}

Table 2: Coprocessor Memory-Mapped Register Field Functions

In this system, the coprocessor (AXI Slave) drives the Status Register as its RESPONSE to the HOST CPU (AXI Master). Throughout the interaction and coprocessor activity, the status register will change based on the coprocessor state, and the HOST CPU must monitor this register to determine its control.

The Address Register is driven by the HOST CPU, and on Writes, the HOST initiates an operation with Image Address [2:0], signifying the index of the image in the coprocessor's Image Buffer. Thus, on a HOST Write, the corresponding Image Type and Function (*Table 3*) are written on the dataline for the coprocessor to determine which operation to perform, on which image, and the necessary memory bounds when reading the image from the Image Buffer.

When the coprocessor completes its operation and its status is made clear to the HOST, the HOST CPU Read enables the coprocessor to write to the bidirectional databus, and the coprocessor responds by delivering the Image Address (in Video Memory) of the operation result for the HOST to later send to the VGA driver. This HOST Read also serves as the AXI master acknowledgment to the completed operation, and the coprocessor is able to change its state accordingly. Note, on HOST reads, the Input Image Index is redundant.

Function Opcode	**Function Mnemonic**	**Function Description**
000	INVERT	Pixel-wise inversion on current image to product image negative
001	COLOR	Pseudo-colorization on current image by mapping grey-scale pixel-contrast to RGB pixels
010	CONTRAST	Image contrast stretching to enhance contrast on current image by utilizing entire dynamic range of image
011	EDGE_DET	Generation of edge-map for current image using 3x3 Sobel edge detection convolution
100	GAUSSIAN	Image smoothening using 3x3 Gaussian Filter on current image
101	THRESH	Hysteresis Thresholding for differentiation of strong and weak edges of current image (Used AFTER EDGE_DET operation)

Table 3: DSP Coprocessor Supported Functions

The following *Figure 3* examples an interaction between the HOST CPU and coprocessor to successfully process and display a new image.

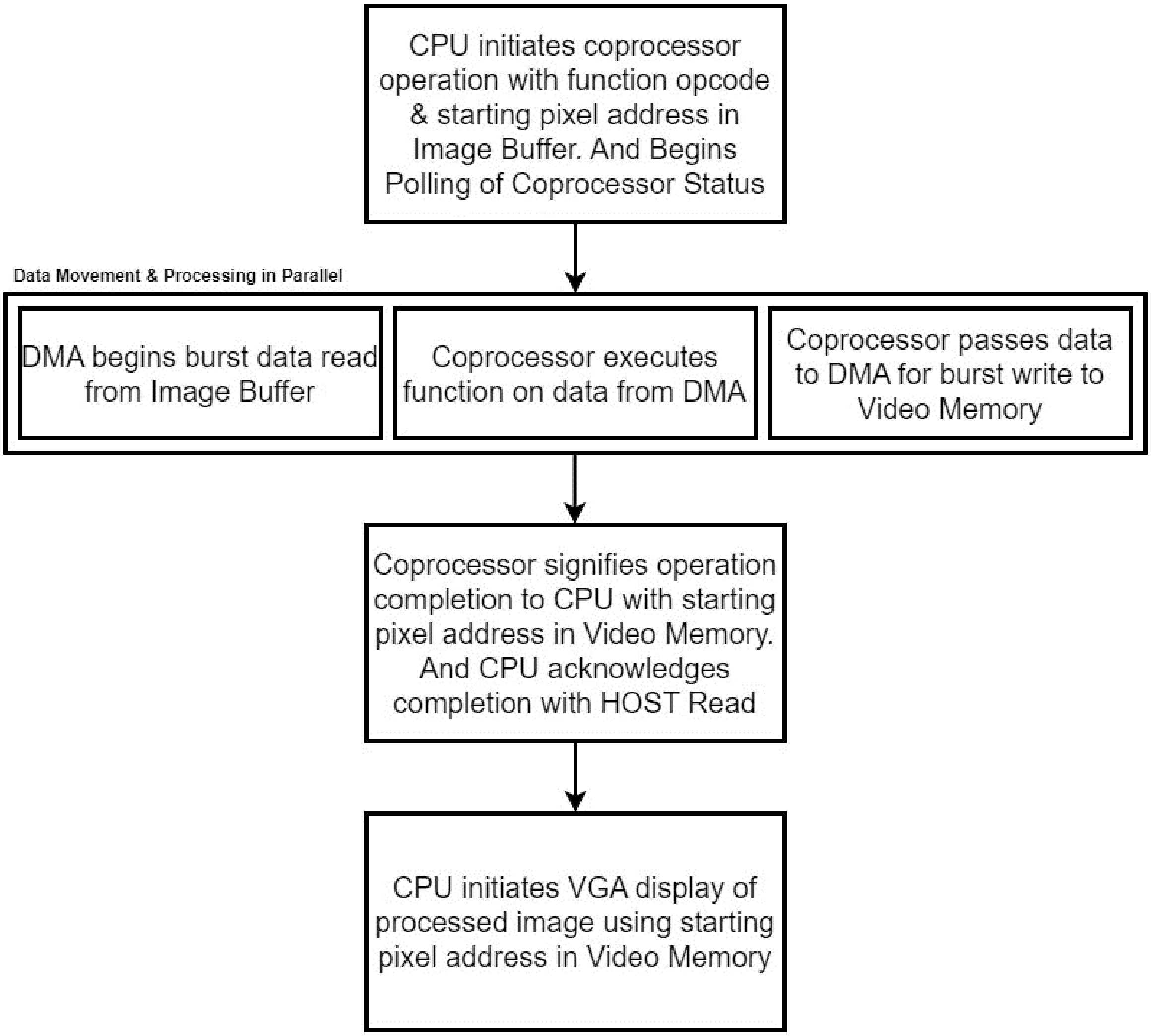

Figure 3: Example Interaction between HOST CPU and DSP Coprocessor

VGA Driver

The VGA peripheral is utilized to display image outputs from the coprocessor to a monitor display. The FPGA VGA port supports up to 6-Bit RGB, and all processing on image data will strive to maintain full resolution. The Video Memory will act as the video buffer between the VGA Driver and the coprocessor output. This memory primarily serves as temporary storage for an image before it is displayed.

HOST CPU

ISA Summary

The primary HOST CPU implements the RISC-V 32I Base Instruction Set, Version 2.0. Below is a list of the supported instructions, and more information can be found in the RISCV-Spec-v2.2 document.

Instruction Name	Description	Instruction Type
LUI	Load Upper Immediate	U

AUIPC	Add Upper Immediate to PC	U
JAL	Jump and Link	J
JALR	Jump and Link to Register	I
BEQ	Branch Equal	B
BNE	Branch Not-Equal	B
BLT	Branch Less	B
BGE	Branch Greater-Equal	B
BLTU	Branch Less Unsigned	B
BGEU	Branch Greater-Equal Unsigned	B
LB	Load Byte	I
LH	Load Halfword	I
LW	Load Word	I
LBU	Load Byte Unsigned	I
LHU	Load Halfword Unsigned	I
SB	Store Byte	S
SH	Store Halfword	S
SW	Store Word	S
SBU	Store Byte Unsigned	S
SHU	Store Halfword Unsigned	S
ADDI	Add Immediate	I
SLTI	Set Less Than Immediate	I
SLTIU	Set Less Than Immediate Unsigned	I
XORI	XOR Immediate	I
ORI	OR Immediate	I
ANDI	AND Immediate	I
SLLI	Shift Left Logical Immediate	I
SRLI	Shift Right Logical Immediate	I
SRAI	Shift Right Arithmetic Immediate	I
ADD	Standard Addition	R
SUB	Standard Subtraction	R
SLL	Shift Left Logical	R
SLT	Set Less Than	R
SLTU	Set Less Than Unsigned	R
XOR	Standard XOR	R
SRL	Shift Right Logical	R
SRA	Shift Right Arithmetic	R
OR	Standard OR	R
AND	Standard AND	R
ECALL	Execution Environment Request	I

Table 4: RISC-V Base 32I ISA Supported Operations

The instructions above follow the 6 primary instruction types (R, I, S, B, U, J) which dictate immediate types (*Figure 6*), function bits, and register addressing:

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

Figure 4: RISC-V Base 32I ISA Instruction Types

Addressing Modes

Primarily, the core processor uses base-register addressing for memory accesses. For example, with Loads, I-Type instructions offset the base register RS1 and address RAM for reads. Similarly, for Stores, S-Type instructions offset the base register RS1 and address RAM for writes.

Register File Arrangement

The main core processor contains a 32-deep 32-bit Register file where register x0 is hard-wired to 0x0000. Traditionally, the registers are used by the programmer as depicted below; however, these registers can be used in any manner.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Figure 5: RISC-V Base 32I ISA Register File Protocol

Immediate Formation

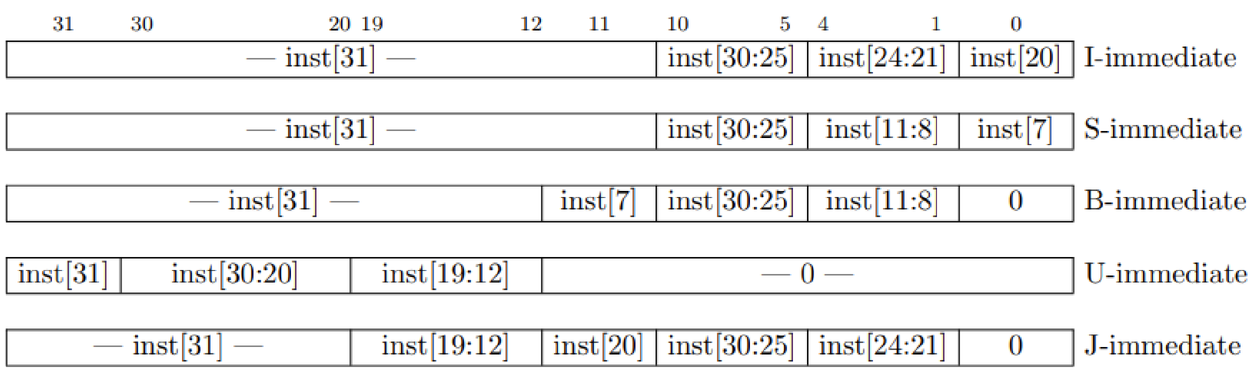

Figure 6: RISC-V Base 32I ISA Immediate Formation

Harvard vs Von Neumann

This system implements a Harvard model with discrete Data and Instruction Memories. In order to provide data memory initialization and transfer of information from instruction memory to data memory, we are implementing a data bootloader system as depicted below. This system will interface with a UART driver from a PC and write to the FPGA instruction or data memory.

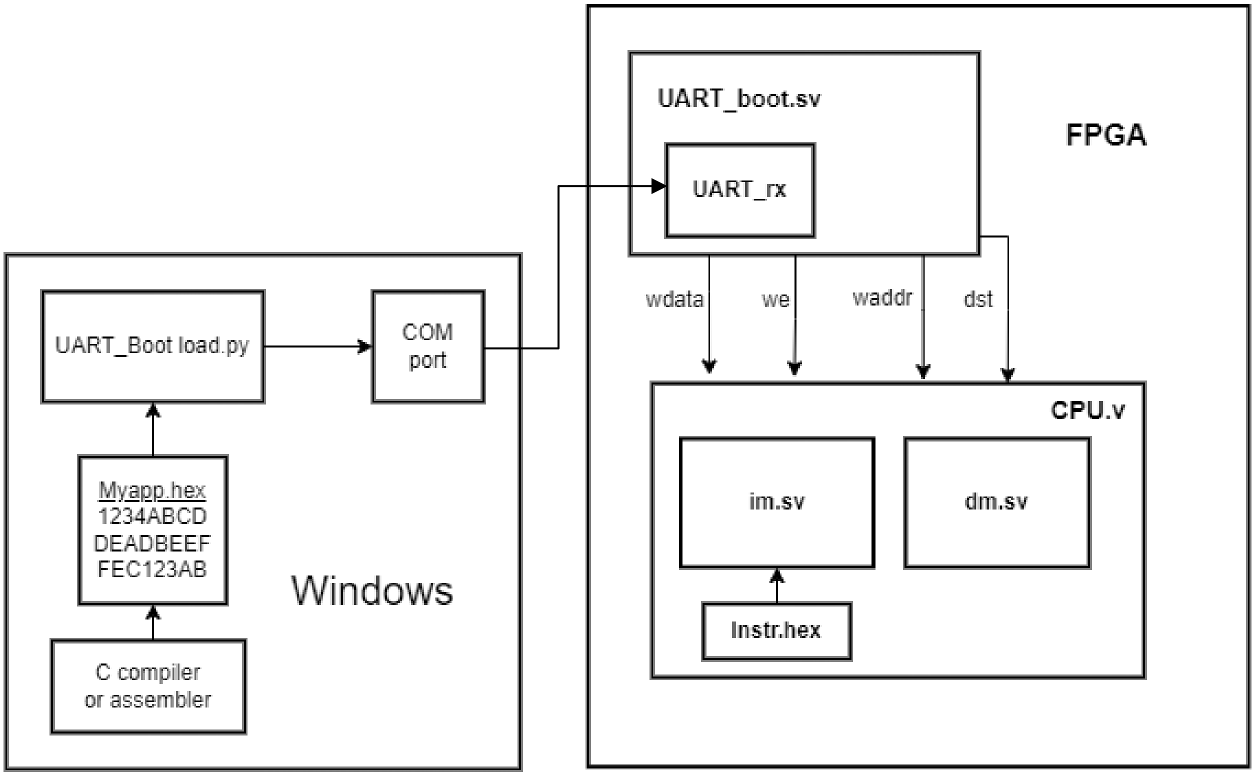

Figure 7: CPU Bootloading Process

Software Stack

Assembler & Compiler

Industry-standard C compilers and assemblers are utilized for this project, given the implementation of the RISC-V ISA for the HOST CPU.

Additional Scripts

Scripts for bootloading data and formatting initial data into memory will be implemented using Python.

Simulator

A known System Verilog based RISC-V simulation environment will be inherited and adapted for this project to generate traces of simulations on the processor.

Application

We will have a RISC-V processor and a customized coprocessor implemented on an FPGA to perform image processing tasks such as edge detection, highlighting, contrast adjustment, and recoloring. Our coprocessor is designed to execute image processing functions rapidly and efficiently, operating in parallel with the host CPU. The coprocessor supports fast and efficient parallel load and store functionalities from different memory banks and process the image in filter batches. We will use switches on the FPGA board as inputs for selecting which filter to apply to the image. As small batches of the image are processed, the completed portions will be written to the video memory. Since the image data is read out of order in parallel, we need to reformat the processed data into the original image layout before writing to the video memory. Subsequently, the coprocessor will signal the CPU when all processing is complete, and the image is ready for the VGA to display. Both the original and processed images will be displayed side by side on the VGA monitor.

The HOST CPU will primarily be utilized for polling peripherals and will behave as the control master for the coprocessor and the VGA driver.

Division of Labor

Table 5 below depicts the division of labor for the project. Tasks were assigned based on design preferences and comfort level with design.

Team Member	Designated Tasks
Ashwin	CPU Design & Verification, DSP specification
Yucheng	CPU Design & Verification, UART Bootloader & I/O
Garry	Application Software, VGA/DSP Interface, Image Buffer & Loading Support Scripts
Alvin	DSP Processing Element, DMA Controller

Table 5: Division of Labor

Team Signatures

- Ashwin K. Avula
- Yucheng Chang
- Alvin Cheng
- Garry Chen