**RISC-V Microprocessor & DSP Element**

Team Sachima

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

**ECE 554: Microarchitecture Specification**

**University of Wisconsin-Madison**

Table of Contents

[Hardware Specification 3](#_Toc161932873)

[Top-Level Interface 3](#_Toc161932874)

[UART Bootloader Architecture 4](#_Toc161932875)

[UART Protocol 5](#_Toc161932876)

[RISC-V CPU Architecture 5](#_Toc161932877)

[FETCH Stage (IF) 7](#_Toc161932878)

[DECODE Stage (ID) 8](#_Toc161932879)

[EXECUTE Stage (EX) 10](#_Toc161932880)

[MEMORY Stage (MEM) 12](#_Toc161932881)

[WRITE-BACK Stage (WB) 14](#_Toc161932882)

[Data Hazard Management: Forwarding Unit 14](#_Toc161932883)

[Control Hazard Management: Hazard Unit 15](#_Toc161932884)

[DSP Accelerator Architecture 15](#_Toc161932885)

[Basic Workflow 15](#_Toc161932886)

[Image Buffer 16](#_Toc161932887)

[Direct Memory Access Convention 17](#_Toc161932888)

[Processing Element 19](#_Toc161932889)

[Video Driver Architecture 21](#_Toc161932890)

[Software Specification 22](#_Toc161932891)

# Hardware Specification

## Top-Level Interface

A diagram of a computer hardware system

Description automatically generated

Figure 1: Top-Level Device Components

Depicted in *Figure 1*, the proposed device is controlled by a HOST RISC-V CPU core. Using an I/O Layer, the HOST can communicate with an array of peripherals via memory-mapped I/O. For example, a Special Purpose Asynchronous Receiver Transmitter (SPART) peripheral is used as a UART interface primarily for debugging purposes. The main DSP Unit employs a coprocessor architecture for accelerating various image processing operations. The DSP Unit interfaces with its own dedicated Image Buffer, which stores the 256x256 RGB images that the unit operates on. The unit utilizes Direct Memory Access (DMA) structures to maintain simultaneous read and write streaming of pixel data, in parallel with the image processing performed in the Processing Element of the DSP Unit. A dedicated video memory is employed to store the immediate images, which are output to a monitor display via a VGA Driver. Finally, a UART Bootloading system is utilized for efficient reprogramming of the CPU Instruction and Data Memories, as well as the DSP’s Image Buffer. The following sections will highlight interfaces between these primary components and the inner microarchitecture of each component. It is important to note that all modules are currently expected to map to the Altera SoC FPGA, and following discussions on memory modules detail the size constraints in order to meet this full utilization of on-chip RAM.

## UART Bootloader Architecture

Depicted in *Figure 2*, a Bootloader system is utilized for the initialization of on-chip SRAM to avoid frequent and redundant compilation and flashing onto the Altera FPGA. Using a HOST PC, application code, image data, or raw data values can be converted to Binary and sent via UART over a COM Port in real-time. The main UART\_boot module implements a general-purpose UART RX and, based on the protocol discussed in the following subsection, will store the incoming data into either the Instruction RAM, Data RAM, or the Image Buffer. During this bootloading process, the UART\_boot acts as the master for the CPU and keeps the CPU in reset until bootloading is complete. Thus, execution will resume from PC Address 0x0000 when complete.

A diagram of a computer system

Description automatically generated

Figure 2: Top-Level Bootloader Architecture

|  |  |  |
| --- | --- | --- |
| **Signal** | **Port Direction** | **Description** |
| wdata[7:0] | Output | Serial Data Byte Output |
| we | Output | Active High Write Enable |
| done | Output | Transaction Complete Flag, Active High |
| Status[1:0] | Output | Status of the state machine:  0=IDLE, 1=CTRL, 2=ADDR, 3=DATA |
| bootloading | Output | Flag signifying Bootloading in process. Used to pull down Active Low CPU Reset |
| dst[1:0] | Output | Bootloaded Data Destination:  00=IM, 01=DM, 10=IB |
| RX | Input | Serial Bit Received from HOST PC |

Table 1: UART Bootloader Interface

### UART Protocol

The Bootloader module operates on a standard UART 8N1 frame with a baud rate of 115200 bps. The protocol requires the HOST CPU to initiate the transaction with a control byte, with the 2 least-significant bits indicating which memory is being written to. The next two bytes indicate the address of the memory to be written to. Finally, the following four bytes indicate the actual desired data to be written (assuming RISC-V 32b data width). This process repeats until all data is written to the desired memory, and the HOST CPU releases the UART Line high, completing the Bootloading process.

A close up of a label

Description automatically generated

Figure 2: UART Bootloader Protocol

## RISC-V CPU Architecture

For the main CPU core, the RISC-V Base 32I ISA is adopted. This implementation will include a pipelined architecture, a Harvard-style memory system, a 32 x 32-bit Register File, byte-addressable data memory, memory-mapped I/O, and compatibility with the open-source RISC-V toolchain. The following subsections detail each pipeline stage, memory constraints, and optimizations to minimize stalling and flushing occurrences.

A diagram of a system

Description automatically generated

Figure 3: RISC-V Top-Level Pipeline Architecture

### FETCH Stage (IF)

The Fetch pipeline stage consists of the PC register, which maintains the currently fetched instruction from Memory. In the event that the currently fetched instruction is an ECALL operation, the HALT DETECT Unit will signal a HLT Signal, which will disable the PC. Thus, the CPU will be stalled in this HALT state forever. The PC source mux selects between the incremented PC and the computed TARGET\_ADDR in the event of control flow changes. The Fetch stage includes a 32KB Instruction Memory, which is a single-port single-cycle RAM. This memory was constructed to be a single-bank word-accessible memory, as instructions are only accessed in word granularity. Under the constraints of the FPGA on-chip RAM, the CPU was given only 64KB of memory to be used as instruction and data memories. Thus, the Instruction occupies 32 KB of this memory.

A diagram of a computer circuit

Description automatically generated

Figure 4: Top-Level Fetch Stage

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Port Direction** | **Description** | **Origin Stage** |
| TARGET\_ADDR[31:0] | Input | 32-Bit Address for Jump & Branch Cases | DECODE |
| PC\_SRC | Input | PC\_NEXT Mux Select | DECODE |
| HLT | Input | HALT Instruction Flag to disable PC | DECODE |
| PC[31:0] | Output | Current Program Counter | FETCH |
| PC\_INC[31:0] | Output | Current Incremented Program Counter | FETCH |
| INST[31:0] | Output | 32-Bit Instruction Fetched from Memory | FETCH |

Table 2: FETCH Top Interface

### DECODE Stage (ID)

The Decode pipeline stage handles the bulk of control generation, operand accessing, and branch/jump resolution. In this stage, the Control Unit generates all control signals using only the Instruction 7-bit Opcode, 3-bit Function field, BIT30, as well as a BRANCH flag from the BR Comparison Unit. With the generated control signals, the proper 32-Bit immediate can be generated, the correct branch/jump target address can be calculated, and the correct register locations can be accessed. All control signals not consumed in the current DECODE stage are passed to downstream stages and will propagate via pipeline registers.

A diagram of a computer system

Description automatically generated

Figure 5: Top-Level Decode Stage

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Port Direction** | **Description** | **Origin Stage** |
| TARGET\_ADDR[31:0] | Output | 32-Bit Address for Jump & Branch Cases | DECODE |
| PC\_SRC | Output | PC\_NEXT Mux Select | DECODE |
| HLT | Output | HALT Instruction Flag to disable PC | DECODE |
| PC[31:0] | Input, Output | Current Program Counter | FETCH |
| INST[31:0] | Output | 32-Bit Instruction Fetched from Memory | FETCH |
| IMM32[31:0] | Output | 32-Bit Immediate | DECODE |
| REG\_WRITE | Output | Register File Write Enable | DECODE |
| RD[4:0] | Output | Register Final Destination Address | DECODE |
| WB\_SRC\_SEL[1:0] | Output | Write Back Stage Mux Select | DECODE |
| MEM\_WRITE | Output | Data Memory Write Enable | DECODE |
| MEM\_READ | Output | Data Memory Read Enable | DECODE |
| MEM\_TYPE[2:0] | Output | Data Memory Access Type | DECODE |
| EX\_OUT\_SEL | Output | Execute Stage Output Mux Select | DECODE |
| SHAMT[4:0] | Output | ALU Shift Amount | DECODE |
| ALU\_OP[3:0] | Output | ALU Control Function | DECODE |
| ALU\_A\_SRC | Output | ALU Operand A Mux Select | DECODE |
| ALU\_B\_SRC | Output | ALU Operand B Mux Select | DECODE |
| SRA[31:0] | Output | Register File Read Port 1 Output | DECODE |
| SRB[31:0] | Output | Register File Read Port 2 Output | DECODE |
| MEM\_WB\_REG\_WRITE | Input | Pipelined Register File Write Enable | WRITE BACK |
| WB\_DATA[31:0] | Input | Write Back Data Output, Register File Destination Data | WRITE BACK |
| MEM\_WB\_RD[4:0] | Input | Register File Destination Address | WRITE BACK |

Table 3: DECODE Top Interface

#### **Immediate Generation Unit**

A diagram of a machine

Description automatically generated

Figure 6: Immediate Generation Microarchitecture

Above, Figure 6 details the microarchitecture for the generation of Immediate Values. This unit follows the RISC-V Base 32I Immediate format and utilizes the ISA Instruction field sharing optimization. This minimizes the amount of selection logic, muxes, and wiring needed to generate the various types of immediates.

#### **BRANCH Comparison Unit**

A diagram of a mathematical equation

Description automatically generated A diagram of a computer component

Description automatically generated

Figure 7: (A) General Purpose Comparator. (B) Branch Comparison Logic

Above, Figures 7A & B detail the microarchitecture for a general-purpose comparator that generates equality and inequality flags. This comparator is then implemented in the Branch Boolean Unit, which compares the Register File Read 1 & Read 2 Data outputs and, based on the instruction condition codes, determines if a branch should be taken. The comparator was designed to be general-purpose for implementation in the ALU as well.

### EXECUTE Stage (EX)

The Execute pipeline stage consists of the main ALU, which handles the bulk of the computation for instructions in the RISC-V Base 32I ISA. The ALU selects between the Register File Data outputs, PC, and 32-Bit Immediate using its SRC muxes, and additional operation control generated in the Control Unit is used to select the instruction executed within the ALU. Finally, the ALU result is muxed with the 32-Bit Immediate in the event of an LUI operation, allowing the immediate to bypass the ALU and minimizing the control within the ALU.

A diagram of a computer component

Description automatically generated

Figure 8: Top-Level EXECUTE Stage

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Port Direction** | **Description** | **Origin Stage** |
| IMM32[31:0] | Input | 32-Bit Immediate for I-type Instructions and address computation | DECODE |
| SRA[31:0] | Input | Register Data 1 Output for all ALU Instructions BUT AUIPC | DECODE |
| SRB[31:0] | Input | Register Data 2 Output for R-type Instructions | DECODE |
| PC[31:0] | Input | Program Counter for AUIPC | FETCH |
| SHAMT[4:0] | Input | Shift Amount for ALU Shift Instructions | DECODE |
| ALU\_OP[3:0] | Input | ALU Control Vector | DECODE |
| ALU\_A\_SRC | Input | Mux Select for ALU A Operand | DECODE |
| ALU\_B\_SRC | Input | Mux Select for ALU B Operand | DECODE |
| EX\_OUT\_SEL | Input | Mux Select for Execute Output | DECODE |
| EXECUTE\_OUT[31:0] | Output | Execute Output Result | EXECUTE |

Table 4: EXECUTE Top Interface

#### **Arithmetic-Logic Unit**

The following *Figure x* details the microarchitecture of the primary ALU. For the main ADDER, a 32-bit Carry Lookahead structure is implemented for ADD, SUB, AUIPC, and LOAD/STORE instructions, as A diagram of a computer program

Description automatically generated with medium confidencewell as their immediate variants. A 3-function Barrel Shifter is implemented with a maximum shift of 31 bits for SLL, SRL, and SRA instructions, as well as their immediate variants. A simple 3-function Logical Unit is implemented for XOR, AND, and OR instructions, as well as their immediate variants. Finally, the Comparator detailed in the DECODE Section is reused in the ALU for SLT, SLTU instructions, as well as their immediate variants. All intermediate outputs are muxed to the ALU\_OUT based on the ALU\_OP.

A diagram of a transmission unit

Description automatically generated

Figure 9: BASE 32I ALU

### MEMORY Stage (MEM)

A diagram of a computer chip

Description automatically generated

Figure 10: Top-Level EXECUTE Stage

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Port Direction** | **Description** | **Origin Stage** |
| MEM\_DATA\_TYPE[2:0] | Input | Data Memory Access Type | DECODE |
| MEM\_WRITE | Input | Data Memory Write Enable | DECODE |
| MEM\_READ | Input | Data Memory Read Enable | DECODE |
| MEM\_ADDR[31:0] | Input | Byte-Address of Data Accessed in Current Cycle | EXECUTE |
| MEM\_DATA\_IN[31:0] | Input | Data to be Written on a Store Instruction | EXECUTE |
| MEM\_DATA\_OUT[31:0] | Output | Data Read from Memory to be Written to Register File on a Load Instruction | Memory |

Table 5: MEMORY Top Interface

The Memory pipeline stage serves as the primary access point to Data Memory. This stage includes a 32KB Data Memory, which is a single-port single-cycle RAM. The memory was designed to be a four-bank byte-accessible memory, as data can be accessed in byte granularity (Store Byte, Load Byte, etc.). Under the constraints of the FPGA on-chip RAM, the CPU was allocated only 64KB of memory to be used as instruction and data memories. Thus, the Data occupies 32 KB of this memory.

In addition to accessing Data Memory, the Memory Stage allows for access to Memory-Mapped I/O. Note that Memory-Mapped access is omitted from Figure 10; however, the microarchitecture can be observed in *Figure 3*. Memory-Mapped Registers in the I/O layer can be found in *Table 6* below.

|  |  |  |
| --- | --- | --- |
| **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 |
| 0xC00C | Coprocessor Status Register [7:0] | Input |
| 0xC00D | Coprocessor Control Register [7:0] | Output |

Table 6: Memory-Mapped I/O Region

#### **Data Memory Unit**

A diagram of a machine

Description automatically generated

Figure 11: Data Memory Byte Banking Scheme

To achieve byte-addressability, the 32-bit Data Memory is banked into byte-regions. As detailed in the RISC-V Base 32I ISA, memory is assumed to be aligned with respect to the memory access data type. Furthermore, data input and output always begin at the least-significant byte. Thus, to input and output data from the various banks, a Right-or-Left Barrel shifter is implemented. By utilizing the least two significant bits of the memory address (Byte-Address), a bit address of the desired byte is obtained by left-shifting the byte address by 3 bits. This bit address is then used as a shift amount to right-shift Read data down to the least significant bit or to left-shift Write data up to the desired Byte Bank. Additionally, the output data is passed to an Extender unit, which manages zero and sign extension of non-word reads based on the memory data type. For example, given a Load-Byte instruction with an address having the least significant two bits of 10, a bit address of 16 is derived (reading from Byte Bank 2). Thus, the Data memory output is right-shifted so that the output of Byte Bank 2 now resides in the least-significant bit position.

### WRITE-BACK Stage (WB)

The Write-Back pipeline stage simply selects the appropriate data value to be written into the Register File. This selection is made based on the WB\_SRC\_SEL control signal defined in the Control Unit. Additionally, Register File address and write enable signals are propagated to the Write-Back stage. This behavior is depicted in *Figure 12* below, and the interface is detailed in *Table 7*.

A diagram of a rectangular object with arrows

Description automatically generated

Figure 12: RISC-V WRITE-BACK Stage Top

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Port Direction** | **Description** | **Origin Stage** |
| WB\_SRC\_SEL[1:0] | Input | Mux Select control determining value passed to Reg File Dest Data | DECODE |
| PC\_INC[31:0] | Input | Incremented PC propagated for JALx Instructions. | FETCH |
| MEM\_DATA\_OUT[31:0] | Input | Output of Data Memory for LOAD Instructions | MEMORY |
| EXECUTE\_OUT[31:0] | Input | Output for all other instructions writing to the Reg File | EXECUTE |
| WB\_DATA[31:0] | Output | Selected value to be written in the Reg File | WRITE-BACK |
| MEM\_WB\_RD[4:0] | Pass Through | Destination Address for value written to Reg File | DECODE |
| MEM\_WB\_REG\_WRITE | Pass Through | Register File Write Enable | DECODE |

Table 7: RISC-V WRITE-BACK Stage Interface

### Data Hazard Management: Forwarding Unit

To handle Data Hazards introduced by the 5-stage pipeline, various forwarding methods are implemented to bypass pipeline registers and minimize the number of stall cycles required to retrieve the proper data for an instruction. This implementation utilizes: Register File Bypassing (Mem-to-Decode) Forwarding, Ex-to-Decode Forwarding for Branch and Jump operands, Ex-to-Ex Forwarding and Mem-to-Ex Forwarding for ALU Operands, and finally Mem-to-Mem Forwarding for Store Instructions. The Forwarding Unit is an observing entity not bound to a single pipeline stage but instead spies on the control signals and addressing flowing through the pipeline to determine if forwarding is appropriate.

### Control Hazard Management: Hazard Unit

Similar to the Forwarding Unit, the Hazard Unit is an observing entity for the management of Control Hazards introduced by the 5-stage pipeline. The Hazard Unit primarily introduces pipeline bubbles into the IF/ID registers in the event of a Jump or Branch taken, such that the currently fetched instruction is invalid and must be flushed. In the event that stalling is necessary, such as in load-to-use stall, data-to-branch, or data-to-jumpR cases, the FETCH & DECODE stages are frozen by disabling the IF/ID registers and resetting the ID/EX registers. This reset of the ID/EX registers introduces a pipeline bubble into the EXECUTE stage as instructions continue downstream to resolve the data hazard. More

## DSP Accelerator Architecture

A diagram of a computer hardware system

Description automatically generated

Figure 13: Top-Level Accelerator Architecture

The proposed device implements a coprocessor in the form of a DSP Accelerator to optimize various image pre-processing and processing functions. The DSP Accelerator consists of an Image Buffer for storage of an “Original” 256x256 input image and a “Most Recent” 256x256 output image. These two images are streamed to a Processing Element via the IMG DMA. The Processing Element is capable of executing various 3x3 Convolution filters on the streaming data as well as various pre-processing functions. Additionally, a VGA driver is implemented to output the processed image data to a Monitor Display. All data is secured to the coprocessor such that the HOST CPU can only initiate operations via memory-mapped I/O. The following sections detail the microarchitecture of these coprocessor components and the dataflow within the system.

### Basic Workflow

On the very first startup, both the image buffer and the video memory will be empty. Each can store two images. The first image in both will contain the original image, while the second image will be the most recently processed one. The UART bootloader will then load the original image into the image buffer, and the CPU will send instructions to the coprocessor to perform a NOP. This operation will compute various heuristic metadata of the image and then store the original image to the video memory for VGA to display. After this setup, the coprocessor will raise the ready signal when the operation is done, and all pixels streamed from the Image Buffer will be written into the video memory for display to the Monitor. Then, the CPU can continue to send functions and control bits to the coprocessor, and these operations will all be performed on the most recently processed image (functions can stack up).

### Image Buffer

The image buffer stores two images: the original and the latest processed version. Each image has dimensions of 256x256 pixels, and every pixel encompasses 3 RGB channels, with each channel encoded with 4 bits. Furthermore, the images are segmented into three banks to support 3x3 kernel convolution operations efficiently. When reading or writing, the operations involve entire rows of 256 pixels (3072 bits) for each bank.

A diagram of a machine

Description automatically generated

Figure 14: Image Buffer Banking Methodology

|  |  |  |
| --- | --- | --- |
| **Signal** | **Port Direction** | **Description** |
| img\_idx | Input | Specifies the index of the image (2 images each for video memory and image buffer) for storing into different memory locations |
| raddr0 [6:0] | Input | Read address for bank0 |
| raddr1 [6:0] | Input | Read address for bank1 |
| raddr2 [6:0] | Input | Read address for bank2 |
| re | Input | Read enable for all the banks in image buffer |
| we | Input | Write enable for the bank specified by waddr in image buffer |
| waddr [9:0] | Input | Bit 9 specifies the image index, bits 7-8 specifies the bank address, bits 0-6 specifies the bank address |
| wdata [3071:0] | Input | Data to be written to a row. 256 pixels \* 4 pixels/channel \* 3 channels |
| rdata0 [3071:0] | Output | Data to be read from bank0 |
| rdata1 [3071:0] | Output | Data to be read from bank1 |
| rdata2 [3071:0] | Output | Data to be read from bank2 |

Table 8: Coprocessor Image Buffer Interface

#### **Image Buffer Memory**

|  |  |
| --- | --- |
| **Memory** | **Description** |
| Original Image  [256x256x12] | The original image is loaded through the bootloader and consists of 256x256 pixels. This image is divided into three banks, with each bank containing 86 rows and columns of 256 pixels, amounting to 3072 bits per row. Notably, the final row in both bank 1 and bank 2 remains unoccupied, leaving it empty. |
| Recent Image [256x256x12] | The recent image, which is the most recent output from the processing unit, is also stored in the same format as the original image. However, the last rows in bank 1 and bank 2 of this recent image are repurposed as temporary storage areas. These rows are utilized to hold processed pixel rows. |

Table 9: Image Buffer Size Specification

### Direct Memory Access Convention

When the start signal is activated, the Direct Memory Access (DMA) retrieves data from the image buffer. It extracts three rows, each containing 256 pixels or 3072 bits, from the image buffer and outputs a 3x3 pixel window designated for convolution. Additionally, the DMA receives a processed pixel (12 bits) from the co-processor, stores it in the shifting register, and writes it back to the image buffer once the register is filled.

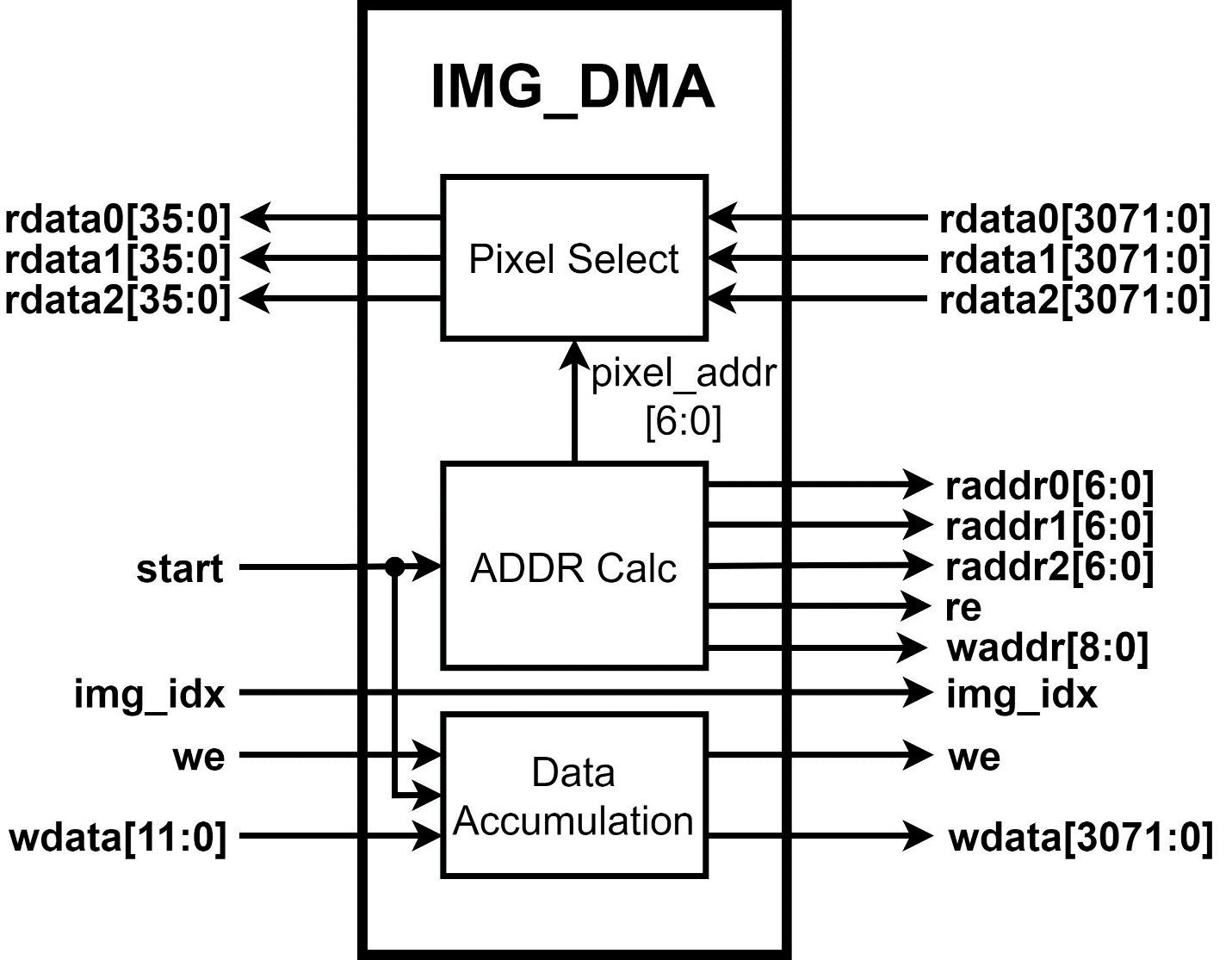


Figure 15: IMG DMA Top-Level

|  |  |  |
| --- | --- | --- |
| **Signal** | **Port Direction** | **Description** |
| rdata0\_in [3071:0] | Input | Data read from bank0. Will be stored in registers |
| rdata1\_in [3071:0] | Input | Data read from bank1. Will be stored in registers |
| rdata2\_in [3071:0] | Input | Data read from bank2. Will be stored in registers |
| img\_idx\_in | Input | Specifies the index of the image (2 images each for video memory and image buffer) for storing into different memory locations |
| we\_in | Input | Write enable input from coprocessor control unit |
| wdata\_in [11:0] | Input | Processed pixel to be written. Will first be stored in a wide register to be written into image banks later |
| start | Input | Asserted for 1 clock to start memory access |
| rdata0\_out [35:0] | Output | First row of data for a 3\*3 convolution window, 3 pixels \* 4 pixels/channel \* 3 channels |
| rdata1\_out [35:0] | Output | Second row of data for a 3\*3 convolution window |
| rdata1\_out [35:0] | Output | Third row of data for a 3\*3 convolution window |
| img\_idx\_out | Output | Specifies the index of the image (2 images each for video memory and image buffer) for storing into different memory locations |
| raddr0 [6:0] | Output | Read address for bank0 |
| raddr1 [6:0] | Output | Read address for bank1 |
| raddr2 [6:0] | Output | Read address for bank2 |
| re\_out | Output | Read enable for all the banks in image buffer |
| we\_out | Output | Write enable for the bank specified by waddr in image buffer |
| waddr [8:0] | Output | Bits 7-8 specifies the bank address, bits 0-6 specifies the bank address |
| wdata\_out [3071:0] | Output | Data to write to a row. 256 pixels \* 4 pixels/channel \* 3 channels |

Table 10: IMG DMA Interface

#### **Direct Memory Access Register**

|  |  |
| --- | --- |
| **Register** | **Description** |
| Shifting Register  [3071:0] | The shifting registers served as the temporary storage of processed pixel from the processing unit. Each time the DMA received a processed pixel, it shifts the register right by 12 bit and store the processed pixel in the most significant 12 bits. When a row within the recent image is no longer required for current processing tasks, the Direct Memory Access (DMA) transfers the data stored in the shifting registers to that specific row. |

Table 11: IMG DMA Register Specification

### Processing Element

The processing element acts as a dedicated image processor, executing tasks such as Gaussian Smoothing, sharpening, and edge detection, among others. Upon activation of the start signal, it streams a 3x3 pixel window from the DMA. If the grayscale bit is asserted by the control register set by the HOST CPU, the image undergoes conversion to grayscale before any processing. Following each processing step, the processor dispatches a single pixel (12 bits) to both the Direct Memory Access (DMA) for write-back to the Image Buffer and the Video Driver. Once the entire image has been processed, a ready signal is issued to indicate completion.

A diagram of a computer component

Description automatically generated

Figure 16: Processing Element Microarchitecture

|  |  |  |
| --- | --- | --- |
| **Signal** | **Port Direction** | **Description** |
| func [2:0] | Input | Indicates which image processing function will be performed. Function as opcodes in the coprocessor |
| grayscale | Input | Control bit to indicate if the image should be converted to grayscale before processing |
| img\_idx | Input | Specifies the index of the image (2 images each for video memory and image buffer) for storing into different memory locations |
| rdata0 [35:0] | Input | Data read from bank 0 in image buffer |
| rdata1 [35:0] | Input | Data read from bank 1 in image buffer |
| rdata2 [35:0] | Input | Data read from bank 2 in image buffer |
| start | Input | Asserted for 1 clock to start coprocessor |
| RGB\_out [11:0] | Output | Processed pixel output with four bits in each channel. If the image is grayscale, it will replicate 4 bits three times |
| we | Output | Write enable signal for both the image buffer and the video memory |
| ready | Output | Raise ready signal if the entire image is processed and written to video memory |

Table 12: Processing Element Interface

### Video Driver Architecture

The video driver architecture is designed to interface directly with the processing unit, from which it receives a 12-bit pixel after each processing operation. This pixel is first stored within the video memory. Following storage, the video driver expands the 12-bit pixel data to 24 bits. After the bit extension process, the 24-bit pixel is transmitted to the monitor for visualization. Finally, it is important to note that the VGA Monitor operates on a 25MHz clock, ½ the rate of the device Refclk. Thus, a PLL is implemented for the 25MHz clock generation.

A diagram of a vga driver

Description automatically generated

Figure 17: VGA Driver of Image Output to a Monitor Display

|  |  |  |
| --- | --- | --- |
| **Signal** | **Port Direction** | **Description** |
| we | Input | Write enable signal for video memory |
| RGB [11:0] | Input | Pixel data to be stored to video memory |
| VGA\_HS | Output | Active low horizontal synch |
| VGA\_VS | Output | Active low vertical synch |
| VGA\_CLK | Output | 25MHz VGA clock from PLL block |
| VGA\_BLANK\_N | Output | Assert (low) during non-active pixels |
| VGA\_SYNC\_N | Output | Tie it low |
| VGA\_R [7:0] | Output | R channel, will be tied as {rdata[11:8],4'b0} |
| VGA\_G [7:0] | Output | G channel, will be tied as {rdata[7:4],4'b0} |
| VGA\_B [7:0] | Output | B channel, will be tied as {rdata[3:0],4'b0} |

Table 13: VGA Driver Interface

#### **Video Memory**

|  |  |
| --- | --- |
| **Memory** | **Description** |
| Images  [640x256x12] | The original image is loaded to video memory at the start of application from the processing unit. Another image is the processed image that has been performed operations by the processing unit. Unlike the image buffer, the row of video memory stored the row of two different images since the images will be displayed side-by-side on the monitor. |

Table 14: VGA Video Memory Specification

# Software Specification

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 processes 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.

We will compare the processing speed of the images (number of clock cycles used) between using the coprocessor and using the CPU solely. When using the CPU, the coprocessor will be detached, and the image data will be stored in the Data-Memory to suit the CPU.