FPGA Implementation of CNN Handwritten Character Recognition

TEAM POOR HANDWRITING

Qikun LIU
Shichen (Justin) QIAO
Haining QIU
Lingkai (Harry) ZHAO

ADVISOR

Eric HOFFMAN

ECE 554 - MAY, 2023

Department of Electrical and Computer Engineering
UNIVERSITY OF WISCONSIN–MADISON

Summary

Machine Learning (ML) has been a skyrocketing field in Computer Science in recent years. As computer hardware engineers, we are enthusiastic about hardware implementations of popular software ML architectures to optimize their performance, reliability, and resource usage.

Our project involved designing a real-time device for recognizing handwritten letters and digits using an Altera DE1 FPGA Kit. We implemented and validated three different ML architectures: linear classification, a 784-64-10 fully connected neural network (NN), and a LeNet-5 CNN with ReLU activation layers and 36 classes. The training processes were done in Python scripts, and the resulting kernels and weights were stored in hex files and loaded into the FPGA's SRAM units. We wrote assembly code for our custom 32-bit floating-point instruction set architecture (ISA) to perform classification and developed a 5-stage MIPS processor in SystemVerilog to manage image processing, matrix multiplications, and user interfaces. We followed various engineering standards, including IEEE-754 32-bit Floating Point Standard, Video Graphics Array (VGA) display protocol, Universal Asynchronous Receiver-Transmitter (UART) protocol, and Inter-Integrated Circuit (I2C) protocols to achieve our project goals.

This report documents the high-level design block diagrams, interfaces between each System Verilog module, implementation details of our software and firmware components, and the potential impacts of our project on society. Additionally, we will provide a final demonstration and discuss each team member's contributions to this senior capstone project.

Project Final Report

Table of Contents

1. Instruction Set Architecture

Full ISA:
https://docs.google.com/spreadsheets/d/1PT7VjIhUPUwOg7ZNtqeGGRNTjavUGF0D/edit#gid=1203584936

Changes from proposal: Added ADDI and SUBI instructions, removed MOVC(LWI) instruction

Responses to Eric's comments:
We extended and validated the assembler according to our 32-bit ISA. Please check out Project/asm_tests/asmbl_32.pl for the implementation and Project/asm_tests/translate_test.asm for a compilation sanity check.
We kept the HLT instruction so that our Python auto-tester could take advantage of it and automatically run assembly validation tests of our processor.
We kept R0 hardwired to 0 to test floating point values efficiently. For instance, when we have an FP value in R1, and we want to branch according to the sign of this value. We can't use ADD or ADDI because the ALU will not properly set flags for FP values. Thus, with this feature, we can do "ADDF R1, R1, R0" and branch safely and efficiently right below this ADDF instruction.
We kept LLB and LHB as they were since we felt something like LLW and LHW confused the LW instruction.

Copy of ISA table (please go to the link at the top of this section for a better-formatted table):

ECE 554 32-bit ISA
General Format
3 register instruction: aaaa_axxx_xxxd_dddd_xxxs_ssss_xxxt_tttt
2 register instruction: aaaa_axxx_xxxd_dddd_xxxs_ssss_iiii_iiii
1 register instruction: aaaa_axxx_xxxd_dddd_oooo_oooo_oooo_oooo
a=opcode, c=sub_opcode, x=don't_care, d=destination, s=source, t=second_source, i=immediate, o=offset

Floating Point Format: IEEE 754: https://en.wikipedia.org/wiki/IEEE_754

1. Flag registers are Z-zero, V-overflow, N-negative/sign
2. The overflow flag denotes positive overflow as well as negative underflow
3. Register R0 is hard-wired to 32'h00000000, can't be written to
4. Jal instruction always stores the return address in register R31. Do not write R31 inside function calls if you wish to return.

Instruction	Encoding	Sample Instruction	OPCODE	Sample Explanation	Other Comments
ADD	aaaa_axxx_xxxd_dddd_xxxs_ssss_xxxt_tttt	ADD R1, R2, R3	5'b00000	R1 <= R2 + R3	Saturating arithmetic. Updates the Z, V
ADDZ		ADDZ R1, R2, R3	5'b00001	R1 <= R2 + R3 only if Z=1	
SUB		SUB R1, R2, R3	5'b00010	R1 <= R2 - R3	

					and N flag registers
AND		AND R1, R2, R3	5'b00011	R1 <= R2 & R3	Updates the Z flag register
NOR		NOR R1, R2, R3	5'b00100	R1 <= ~(R2 | R3)	
SLL	aaaa_axxx_xxxd_dddd_xxxs_ssss_xxxi_iiii	SLL R1, R2, C	5'b00101	R1 <= R2 << C	C is 5-bit unsigned immediate value Updates the Z flag register
SRL		SRL R1, R2, C	5'b00110	R1 <= R2 >> C	
SRA		SRA R1, R2, C	5'b00111	R1 <= R2 >>> C	
LW	aaaa_axxx_xxxd_dddd_xxxs_ssss_oooo_oooo	LW R1, R2, O	5'b01000	R1 <= DataMem[R2 + O]	O is 8-bit signed immediate value
SW		SW R1, R2, O	5'b01001	DataMem[R2 + O] <= R1	
LHB	aaaa_axxx_xxxd_dddd_iiii_iiii_iiii_iiii	LHB R1, C	5'b01010	R1 <= {C, R1[15:0]}	C is 16-bit signed immediate value
LLB		LLB R1, C	5'b01011	R1 <= sign-extend{C}	
B	aaaa_accc_xxxx_xxxx_xxxx_oooo_oooo_oooo				O is signed 12-bit offset in two's complement Branch target address = (Address of branch instruction + 1) + offset PC holds word addresses, each instruction is 1 word, offset is specified as the number of instructions with respect to the instruction following the branch instruction.
NEQ		B NEQ, label	5'b01100000	Branch if Z=0	
EQ		B EQ, label	5'b01100001	Branch if Z=1	
GT		B GT, label	5'b01100010	Branch if {Z,N}==2'b00	
LT		B LT, label	5'b01100011	Branch if N=1	
GTE		B GTE, label	5'b01100100	Branch if N=0	
LTE		B LTE, label	5'b01100101	Branch if N=1 or Z=1	
OVFL		B OVFL, label	5'b01100110	Branch if V=1	
UNCOND		B UNCOND, label	5'b01100111	Branch unconditionally	

JAL	aaaa_axxx_xxxx_xxxx_xxxx_oooo_oooo_oooo	JAL label	5'b01101	R31 <= address of jal instruction +1, jump to target	O is signed 12-bit offset in two's complement Jump target address = (Address of jal instruction + 1) + offset
JR	aaaa_axxx_xxxx_xxxx_xxxt_tttt_xxxx_xxxx	JR R31	5'b01110	Jump to the address in R31	Can be used to return from function calls (jal)
PUSH	aaaa_axxx_xxxx_xxxx_xxxs_ssss_xxxx_xxxx	PUSH R1	5'b10010	DataMem[SP] <= R1; Decrement SP	Stores value in R1 into data memory pointed by the stack pointer; decrements stack pointer
POP	aaaa_axxx_xxxd_dddd_xxxx_xxxx_xxxx_xxxx	POP R1	5'b10011	R1 <= DataMem[SP]; Increment SP	Loads value in data memory pointed by the stack pointer into R1; increments stack pointer
ADDI	aaaa_axxx_xxxd_dddd_xxxs_ssss_iiii_iiii	ADDI R1, R2, I	5'b10100	R1 <= R2 + C	I is 8-bit signed immediate value. Updates the Z, V and N flag registers
SUBI		SUBI R1, R2, I	5'b10101	R1 <= R2 - C	
MUL	aaaa_axxx_xxxd_dddd_xxxs_ssss_xxxt_tttt	MUL R1, R2, R3	5'b11000	R1 <= (signed) R2[15:0] * (signed) R3[15:0]	Only support 16 by 16 multiplications
UMUL		UMUL R1, R2, R3	5'b11001	R1 <= (unsigned) R2[15:0] * (unsigned) R3[15:0]	
ADDF		ADDF R1, R2, R3	5'b11010	R1 <= R2 + R3 (floating-point)	Floating point calculation 1-bit sign, 8-bit exponent, 23-bit mantissa
SUBF		SUBF R1, R2, R3	5'b11011	R1 <= R2 - R3 (floating-point)	
MULF		MULF R1, R2, R3	5'b11100	R1 <= R2*	

				R3(floating-point)	
ITF	aaaa_axxx_xxxd_dddd_xxxs_ssss_xxxx_xxxx	ITF R1, R2	5'b11101	R1 <= R2 (integer to floating-point)	
FTI		FTI R1, R2	5'b11110	R1 <= R2 (floating-point to integer)	
HLT	1111_1xxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx	HLT	5'b11111	Processor Halt	

2. Hardware Block Diagrams

2.1. Top Level

The top level includes instantiations of the following modules.

A set of modules come from the tutorial on exploring the camera. They are: Sdram_Control, RAW2GRAY, CCD_Capture, I2C_CCD_Config, Reset_Delay, and VGA_Controller.

A set of modules are fully self-implemented and added for the required function. They are image_mem, weight_rom, SPART, PLL, and rst_sync. The details are elaborated in the following sections.

The modification of the CPU is explained in the following section.

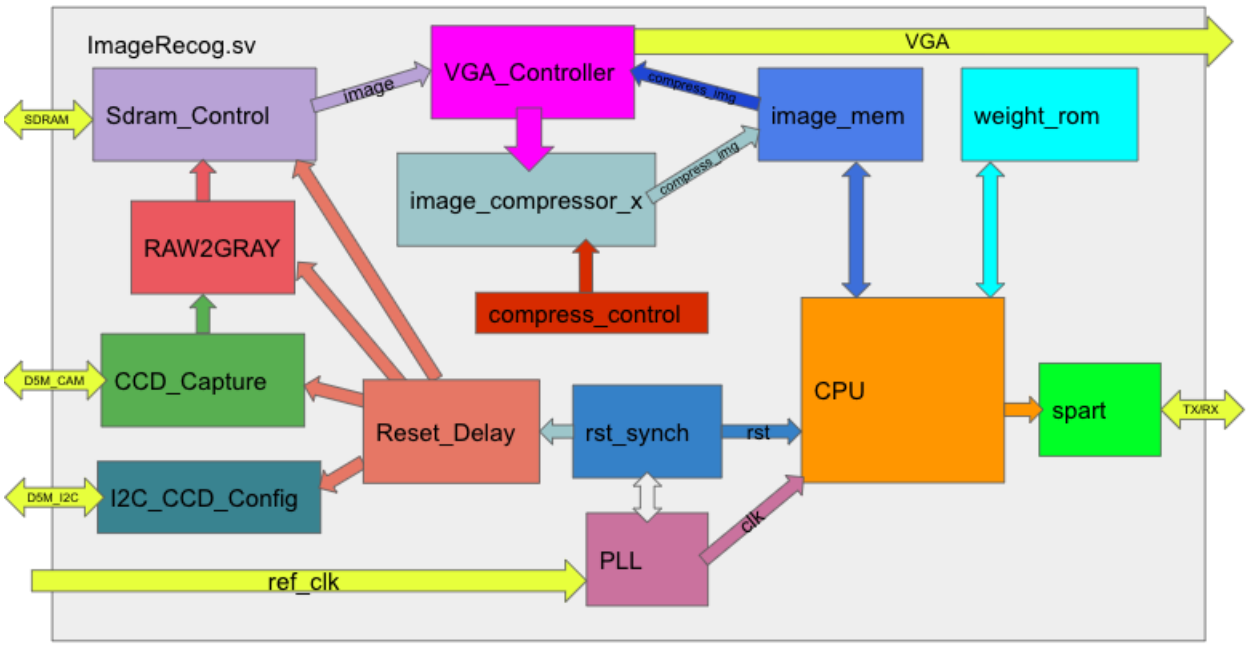

Figure 1: Top_level block diagram. The details of each module are elaborated in the following sections of this document.

2.1.1. Top Level Memory Mapped Registers

Register Address:	Description:
0x0000C000	Write to this address will write to LEDR[9:0] of board
0x0000C001	Read from this address will return the state of SW[9:0] of board
0x0000C004	Transmit Buffer (IOR/W = 0); Receive Buffer (IOR/W = 1)
0x0000C005	Status Register (IOR/W = 1)
0x0000C006	DB(Low) Division Buffer
0x0000C007	DB(High) Division Buffer
0x0000C008	Set to 1 to request compressing the image, it will pull down once the compression is finished.

2.1.2. Top-Level Memory Blocks

Two external memory modules are instantiated at the top level to support the implementation of CNN in hardware.

The image_mem is a dual-port memory module to store a 32*32 compressed image.

The weight_rom is a single-port memory module with all 63,654 weights necessary for CNN. These weights are trained and validated using software. They are then loaded into the ROM using a weight.hex file generated by software. We decided to use external memory for weight_rom for two reasons: 1. We do not want to expand the data_mem of the CPU because we want to keep the CPU as a general-purpose processor. 2. We want to ensure that the weights are stored as ROM and cannot be overwritten by software.

- The first 6x1x5x5 weights are for the first convolution layers
- The second 16x6x5x5 weights are for the second convolution layer
- The third 400x120 weights are for the first fully-connected neural network layer
- The fourth 120x84 weights are for the second fully-connected neural network layer
- The last 84x36 weights are for the third fully-connected neural network layer

For more information on the neural network architectures, please see the software section of the report.

Name:	Start Addr	End Addr	Description:
image_mem	0x00010000	0x0001030F	Image memory RAM, take inputs from image compressor and output values to VGA and processor
weight_rom	0x00020000	0x00021E9F	Weight memory ROM, values load from ML software, output to processor

2.1.3. Top Level Compress Signal Control

To implement real-time processing, we must be able to capture an image only after the last image is processed and outputted through UART. Therefore, we use this compress control logic to allow the CPU to request a new image compression. Before processing each image and having the compressed snapshot in image_mem, the assembler code must request a snapshot by storing 1 to 0xC008 in data mem (SW 1, 0xC008). The compress control will wait until the SDRAM access is synchronized with the first pixel of the snapshot and enable the compressor to process the image. The assembler code needs to

check the data in 0xC008 periodically. The compressed image is ready when the data stored in 0xC008 becomes 0.

Signal:	Dir:	Description:
uncompress_addr_x [7:0]	In	X axis of the uncompressed image address. It is used to synchronize with the first pixel of the uncompressed image
uncompress_addr_ y[7:0]	in	Y axis of the uncompressed image address. It is used to synchronize with the first pixel of the uncompressed image
we	In	Write-enable signal for requesting a snapshot. The CPU will access this when writing to 0xC008.
compress_wdata	In	The request for a snapshot. The CPU can set this to 1 to indicate a request for a snapshot, 0 to indicate no need to take a snapshot
pause	In	Input from the button. This supports the function of freezing video input by pressing a button (KEY[2]). The compress signal control will not start until the key is released.
compress_req	out	The status register of the compressor control. If it is 0, there is no compression going on. If it is 1, compression is in process. CPU is responsible for accessing 0xC008 to check this status.
compress_start	out	The control signal to start a new compression.

2.2. Camera Interface

2.2.1. D5M Camera ports

The camera interface requires the following ports:

input: D5M_D[11:0], D5M_FVAL, D5M_LVAL, D5M_PIXCLK, D5M_STROBE

output: D5M_RESET_N, D5M_SCLK, D5M_TRIGGER, D5M_XCLKIN

inout: D5M_SDATA

These ports are implemented using GPIO_0 ports on the FPGA board. We map the ports on GPIO to the corresponding pins on the D5M camera. The D5M camera captures the images, and the data is stored in the external SDRAM on the FPGA developer board. Because we do not have sufficient FPGA-embedded SRAM memory, we must use the external SDRAM.

2.2.2. SDRAM_Control

To manage the data transferred into the external SDRAM, we use the Sdram_Control module to manipulate the timing and data sent into the SDRAM. Each received pixel is divided into two parts because the width of the SDRAM memory is 8 bits, and the received pixel is 12 bits. The Sdram_Control module takes the captured image and stores the upper and lower bytes sequentially. The Sdram_Control module has a FIFO buffer to avoid loss of data.

2.2.3. I2C_CCD_Config

To manage the Camera's configuration, we use the I2C_CCD_Config module to adjust the exposure, zoom, and brightness configuration of the D5M camera through the I2C protocol. When we want to adjust one camera setting, we use the combination of a switch and a button to set the desired setting.

For example, if SW[0] is on and KEY[1] is pressed, the exposure will increase. If SW[0] is off and KEY[1] is pressed, the exposure will decrease. I2C_CDD_Config will monitor the switch and keys to update settings through the I2C protocol.

2.2.4. Raw2Gray

To Convert the captured color image into a grayscale image, we use the Raw2Gray module to process the captured pixel and convert it to a grayscale image for CNN prediction. The Raw2Gray module takes the input from the CCD_Capture module, which is captured from the camera and converts the output to grayscale values.

2.2.5. CCD_Capture

To capture the image and manage the communication protocol with the camera, we use the CCD_Capture module to control the clock and the data received. This module also keeps track of the received pixel's frame count, x_location, and y_location.

2.3. CPU

This is a high-level block diagram of our modified processor. The original 16-bit pipeline logic was mostly preserved. We expanded the data paths, word size, and instruction decode to 32 bits, expanded the register files from 16 to 32 registers, and added new modules. Note that our Instr_mem has 1K entries, and our Data_mem has 8K. For detailed interface specifications, please refer to the following sections.

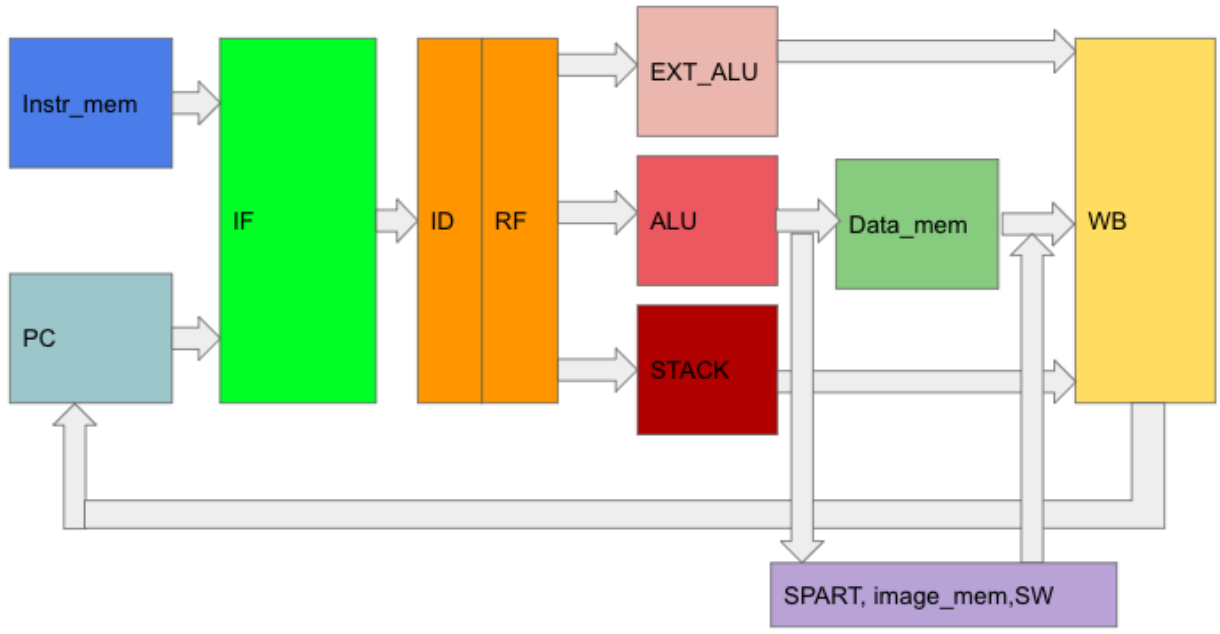

Figure 2: CPU block diagram. The original 16-bit pipeline was kept with datapaths extended to 32-bit. EXT_ALU and STACK are new modules to support our application, which will be elaborated in the following sections.

2.4. Extended ALU

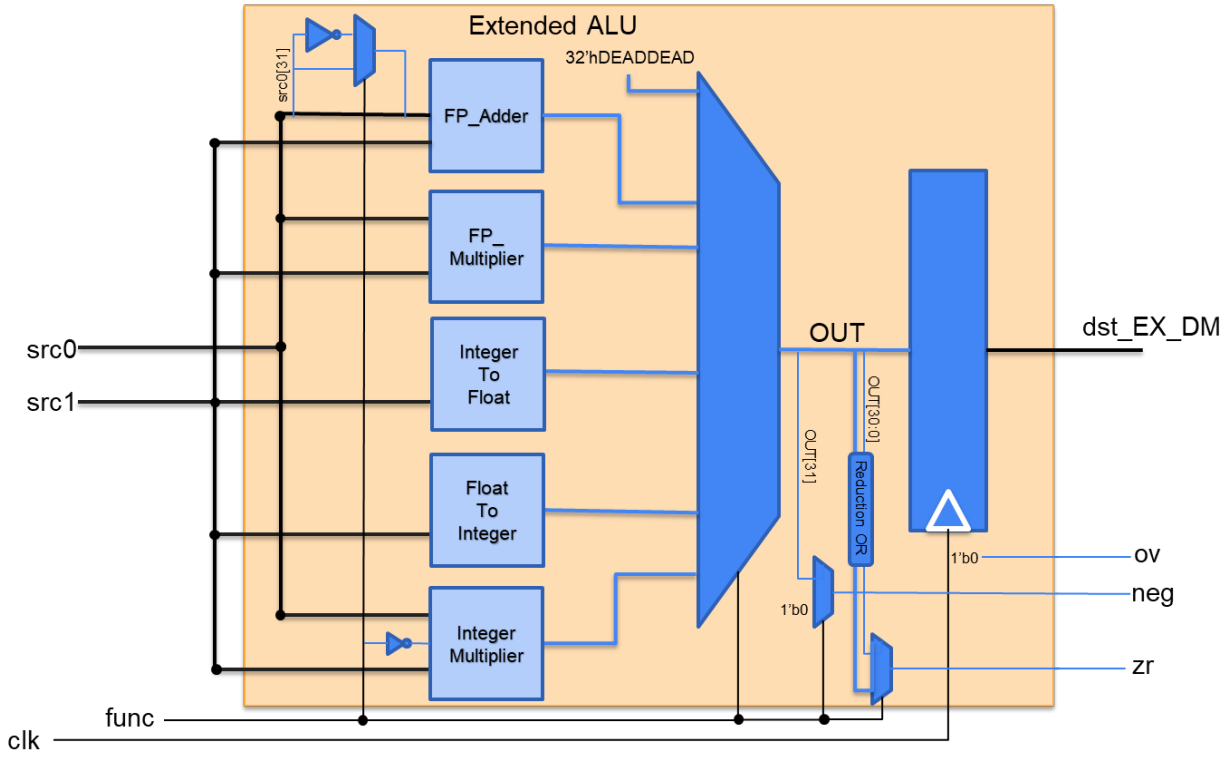

Figure 3: Extended ALU block diagram. The details of those five compute units are explained in the following sections.

The extended ALU is in the same pipeline stage as the original ALU in the processor, while its hardware supports floating-point operations and integer multiplication. It contains five submodules for floating-point addition and multiplication, conversions between float and integer, and integer multiplication. The func signal selects its output value and flags.

Signal:	Dir:	Description:
clk	in	50MHz system clock
src1[31:0]	in	32-bit source 1 into ALU
src0[31:0]	in	32-bit source 0 into ALU
func[2:0]	in	3-bit OP Code: 000 ==> MUL 001 ==> UMUL 010 ==> ADDF 011 ==> SUBF 100 ==> MULF 101 ==> ITF 110 ==> FTI 111 ==> undefined
dst_EX_DM[31:0]	in	32-bit ALU output
ov	out	Overflow flag - but this is always 0!!! Kept here for following branch ops

zr	out	Zero flag - high when output is 0 (int zero or FP zeroes)
neg	out	Negative flag - high when output is negative

2.4.1. Floating Point Adder Interface,

The addition of two IEEE-754 floating-point numbers is a complex process due to the potential difference in their exponents and unsigned mantissa. Here are the steps involved in the FP Adder process:

1. Compare the exponents and determine the smaller ones. Calculate the absolute difference between the exponents; the larger exponent will be the common exponent.
2. Prepend the common exponent to both mantissa, making them both 24-bit in length.
3. Shift the mantissa with the smaller exponent to the right, using the lower 5 bits of the exponent difference as the shift amount. The maximum shift amount should be 22-bit.
4. Convert both appended and shifted mantissa to 2's complement format. This makes both numbers 25-bit in length.
5. Add the two 25-bit numbers, and if the result overflows positively or negatively. Note that this overflow is an internal overflow, not an external value overflow.
6. Convert the 25-bit 2's complement result to a 25-bit signed number, where the MSB is the final sign, and the rest 24-bit is the unsigned value. If this 24-bit prepended mantissa starts with 0, but no internal occurs, denormalize the common exponent to 0.
7. If an overflow occurs, shift the lower 24-bit to the right and append 1 in the MSB, or shift the lower 24-bit to the left if it has leading zeros. The common exponent, if not 0, is adjusted accordingly.
8. The resulting mantissa is the lower 23-bit of the final 24-bit result, and the exponent is the final common exponent. The sign is the MSB of the final result.

Signal:	**Dir:**	**Description:**
A[31:0]	in	32-bit input interpreted as an IEEE-754 floating-point number
B[31:0]	in	32-bit input interpreted as an IEEE-754 floating-point number
Out[31:0]	out	32-bit output as an IEEE-754 floating-point number

2.4.1.1. Left Shifter Interface

Signal:	**Dir:**	**Description:**
In[23:0]	in	24-bit mantissa input to be logically left-shifted
ShAmt[4:0]	in	5-bit left shift amount
Out[23:0]	out	24-bit mantissa output normalized into IEEE-754 format

2.4.1.2. Right Shifter Interface

Signal:	**Dir:**	**Description:**
In[23:0]	in	24-bit mantissa input to be logically right-shifted
ShAmt[4:0]	in	5-bit right shift amount
Out[23:0]	out	24-bit mantissa output normalized to a common exponent

2.4.2. Floating-point Multiplier Interface,

Multiplying two 32-bit values in IEEE-754 format involves breaking the inputs into {S1, E1, M1} and {S2, E2, M2}. The signs are XORed to determine the sign of the product, and the exponents are added together with the 127 offsets accounted for. The mantissa are then appended with an implicit 1 (or 0) and multiplied. Special values like -INF, -0, +0, and +INF require special combinational logic to adjust the outputs. The resulting values are then concatenated and normalized to conform to the IEEE FP standard and output as a 32-bit value.

Signal:	Dir:	Description:
A[31:0]	in	32-bit input interpreted as an IEEE-754 floating-point number
B[31:0]	in	32-bit input interpreted as an IEEE-754 floating-point number
OUT[31:0]	out	32-bit output as an IEEE-754 floating-point number

2.4.3. Float-to-integer Unit Interface

Signal:	Dir:	Description:
FP_val[31:0]	in	32-bit input interpreted as an IEEE-754 floating-point number
signed_int_val[31:0]	out	32-bit output converted into a signed integer

2.4.4. Integer-to-float Unit Interface

Signal:	Dir:	Description:
signed_int_val[31:0]	in	32-bit input interpreted as a signed integer
FP_val[31:0]	out	32-bit output converted into an IEEE-754 floating-point number

2.4.5. 16-by-16 Integer Multiplier Interface

We choose to support a 16-by-16 multiplier because 32-by-32 is way more expensive to implement, and our design does not require high large integer multiplications.

Signal:	Dir:	Description:
A[31:0]	in	32-bit input interpreted as an integer
B[31:0]	in	32-bit input interpreted as an integer
sign	in	1 for signed multiplication; 0 for unsigned multiplication
OUT[31:0]	out	32-bit output as an integer

2.5. Stack

The stack is implemented as an individual memory module. It is placed in the Execute stage for maximum efficiency. We do not implement it as a subset of the data memory because reaching the data memory stage takes one additional cycle.

This stack is implemented in the same fashion as the FIFO buffer, except it pushes to the top of the stack and popes from the top of the stack. This stack is a First-In-Last-Out(FILO) buffer. A list of registers is instantiated, and the access address gets updated each cycle based on the commands(push, pop) to the stack. Only one command is allowed at a time (cannot push and pop in the same cycle) because each instruction from the CPU can only perform one action at a time.

Signal:	Dir:	Description:
clk	in	50M system clock
rst_n	in	active low reset
push	in	push wdata onto stack
pop	in	pop top of stack to stack_EX_DM
wdata[31:0]	in	32-bit data to be pushed
stack_EX_DM[31:0]	out	32-bit data being popped

2.6. Image Processing and Storage

This blue color is used to indicate CNN-related topics.

The image taken from the camera is stored in SDRAM and then sequentially fed into the image compressor. This image compressor sequentially takes a 224*224 8-bit image from SDRAM and compresses it into a 28*28 8-bit image by taking the average color among 8*8 blocks. Within one cycle, only one pixel is taken, and only one pixel is out.

The image compressor x is designed explicitly for the CNN model, requiring a 32*32 image with zero-padding of width 2 surrounding the original 28*28 image. The image compressor x, therefore, has a different output signal called compress_addrx, which ranges from 0 to 1023 (instead of 0 to 783). The zero-padding is achieved by skipping and not writing to the padding address in the image memory so that these addresses always contain 0. The compress_addrx signal is a combinational logic of the original output compress_addr. Note that the SRAM block of the image memory is also extended to 1024 locations.

2.6.1. Image Compressor Interface

Signal:	Dir:	Description:
clk	in	25MHz clock signal from VGA display
rst_n	in	System reset signal
start	in	Signals a valid pixel color input starting from 0
pix_color_in[7:0]	in	8-bit pixel color value from VGA DRAM (0 to 255 grayscale)
pix_haddr[7:0]	in	8-bit pixel horizontal address (0 to 223)
pix_vaddr[7:0]	in	8-bit pixel vertical address (0 to 223)
sram_wr	out	Write enable signal to the SRAM memory storing compressed image
pix_color_out[7:0]	out	8-bit compressed pixel color by taking average value among an 8*8 block
compress_addr[9:0]	out	10-bit compressed image pixel address (0 to 783 for a 28*28 image)

2.6.2. Image Compressor X Interface - only for CNN Model

Signal:	Dir:	Description:
clk	in	25MHz clock signal from VGA display
rst_n	in	System reset signal
start	in	Signals a valid pixel color input starting from 0
pix_color_in[7:0]	in	8-bit pixel color value from VGA DRAM (0 to 255 grayscale)
pix_haddr[7:0]	in	8-bit pixel horizontal address (0 to 223)
pix_vaddr[7:0]	in	8-bit pixel vertical address (0 to 223)

sram_wr	out	Write enable signal to the SRAM memory storing compressed image
pix_color_out[7:0]	out	8-bit compressed pixel color by taking average value among an 8*8 block
compress_addrx[9:0]	out	10-bit compressed image pixel address (0 to 1023 for a 32*32 image)

2.6.3. Image Compressor Registers

Register:	Description:
compress_addr[9:0]	Described in table above. Reset to 10'd784; zero-set when start asserted; incremented when sram_wr asserted. Namely, it increments to the next available SRAM address after an image memory write and stalls itself when the entire image memory gets written until the next asserted start signal.
block[13:0][0:27]	28 14-bit wide SRAM blocks to store the accumulated sum of every pixel value inside 28 8*8 blocks. We need 28 of them since at least one row of 8*8 blocks should be saved for averaging, and there are 28 blocks per row (224/8 = 28). Its address is determined by the upper 5 bits of pix_haddr, named b_haddr. 14-bit is needed since it stores the sum of 64 8-bit wide pixel color values. The average value is taken from its upper 8 bits to produce a compressed pixel color value.

2.6.4. Image Memory Interface

Signal:	Dir:	Description:
clk	in	50MHz system clock. Note that this is 2 times faster than the clock of the image compressor module, but this is safe since every correct data will get written twice at the same address.
we	in	Write enable signal of image SRAM memory from image compressor
waddr[9:0]	in	10-bit write address of SRAM from image compressor
wdata[7:0]	in	8-bit compressed pixel color value from image compressor
raddr[9:0]	in	10-bit read address of SRAM (0 to 1023)
rdata[7:0]	out	8-bit read port of SRAM for a compressed pixel color value

2.6.5. Image Memory Registers

Register:	Description:
rdata[9:0][0:1023]	Described in table above. There are 1024 SRAM blocks just enough for a compressed 32*32 image with zero-padding of width 2. This SRAM is written by the image compressor.

3. Software

3.1. Machine Learning Model - Liner Classification

Input:	Output:
keras.datasets.emnist - 120,000 images with labels	weight.hex - contains 28,224 8-digits hex numbers, one per line.

One high-level software is used for this project. The purpose of the software is to provide the weight matrix that is used to make predictions by performing matrix multiplication on our input image data. This software is written in Python on Google Colaboratory. It includes a Python notebook and a Python script file. The script file contains helper functions, and the notebook will read input, call functions, and generate output.

The software works in three steps: it reads train data, trains on the data, and produces the weight matrix. It currently uses the Keras dataset that contains 60,000 different images of 28*28 pixels of handwritten numbers and corresponding labels to indicate their values. The first 40,000 images are used for training, the next 10,000 images are used for validation during training, and the last 10,000 images are used for testing the accuracy of the trained model.

The model used for this software is a softmax loss linear classifier, which utilizes the difference between ideal logistic probabilities and the current logistic probabilities as the loss function and employs gradient descent to minimize the loss function. This training procedure is repeated at least 200 times. The training process uses the PyTorch library to help boost performance and reduce training time. With appropriate learning rate and regularization factors, the training accuracy can be over 90%, and the testing accuracy is around 88.6% for recognizing individual digits.

After finding the best performance model, we extract the transpose of the weight matrix. The transposed weight matrix is flattened into a list of 784*36 = 28,224 floating-point numbers (7,840 for digits-only models). Each row of the 28 numbers is parsed into hex numbers and pasted into a .hex file, where each line is in the format of {LINE_NUMBER} {HEX_NUMBER}. This file can be directly loaded into the FPGA board as a ROM, which is later read for the matrix multiplication.

3.2. Machine Learning Model - Neural Network

Input:	**Output:**
keras.datasets.emnist - 425,600 images with labels	weight_nn.hex - contains 52,480 8-digit hex numbers, one per line

Since Linear Classification didn't perform too well with around 89% accuracy for digits only and 67% accuracy for digits and letter, we chose to change our model to use Neural Network. A Neural Network(NN) model typically has better performance in classification tasks when compared to Linear Classification(LC) model. This model is also written with Python but using Jupyter Notebook instead since we localized the training dataset.

We used the PyTorch library for this NN model, which is publicly available and has a built-in NN model. Specifically, we used the LeNet NN model with two neuron layers: 784*64 and 64*36, and there is a ReLu hidden layer between these two layers. We set these configurations for the torch.nn library and let it train/test for us. For predicting each image, we will first perform matrix multiplication with the image data and the first layer, which is 1*784 dot 784*64, which results in a matrix of 1*64. Then, we change all the negative numbers in this 1*64 matrix to 0. This simulates the ReLu layer in our NN model. Next, we will perform another matrix multiplication between 1*64 and 64*36. This results in 1*36 numbers, representing the score for each class (10 for digits and 26 for letters - both upper case letters and lower

case letters classify into the same category). The final result is around 99% accuracy for digits only and 91% for digits and letters. However, the performance doesn't appear to be this high once we load it onto an FPGA.

We only need to export the two neuron layers to extract the weights. model.lin1 and model.lin2 were printed to the weightnn.hex file, one on each line. model.lin1 has dimension of 784*64 = 50,176, and model.lin2 has dimension of 64*36 = 2,304. This totals up to 50,176+2,304 = 52,480. This hex file will later be read into weight_nn_rom.sv. Each number is first converted to a float, then cast into a hex, and then re-structured into an 8-digit hex num that is 32 bits large. The format of each line is the same as LC, so refer to the end of section 3.1.

3.3. Machine Learning Model - Convolutional Neural Network

Input:	**Output:**
keras.datasets.emnist - 425,600 images with labels	weight_cnn.hex - contains 63,654 8-digit hex numbers, one per line.

The NN model was much better than the CL model in terms of model accuracy, but we weren't satisfied with its performance on FPGA. Having only two layers limits the model's real-time accuracy performance. But if we were to have more layers for the NN model, our FPGA would have run out of FPGA memory, as we need to store the weight information on the board. Another model that was then introduced was the Convolutional Neural Network(CNN), which would shrink the input size and only extract key features from the dataset, then send the convoluted data to NN layers. This helps us to keep the total number of weights manageable, even if we use more layers for NN.

For the CNN model, we kept most of the structure of the NN model but added more convolutional layers before the linear layers. Two convolutional layers are added, each of their kernels having the dimensions of 6*1*5*5 and 16*6*5*5, which works like two filters for the input data. The input data is still 784(28*28*1). The first convolutional layer will add paddings of 2 pixels on each of the four sides of this picture, which will be 32*32*1. The padding added will all have a value of 0. This 32*32*1 matrix will be sent to the model.cov1 layer, which uses 6*1*5*5 kernels to extract information. This operation will then use a 28*28*6 matrix as the first middle data. We then change all negative numbers to a 0, serving as the ReLu layer. This processed 28*28*6 data will then be average pooled with 2*2 kernels and stride = 2, which means each time we take a 2*2 matrix and grab the average, and move by two each time. This will then give us a 14*14*6 matrix as the input to the second convolution layer. The model.cov2 has 16*6*5*5 kernels, and by operating on the 14*14*6 matrix, which will turn into a 10*10*16 matrix as the second middle data. Again, we take all negative values out and substitute them with 0. Then we do average polling again with 2*2 kernels and a stride of 2. This will finally give us a 5*5*16 matrix as our data. The data has a dimension of 1*400 after being flattened, which is way less than the original input of 1*784. The total number of weights for the convolution layers is the sum of the two convolutional layers, in which model.conv1 is 6*1*5*5 = 150, and model.conv2 is 16*6*5*5 = 2400. In total, the dimension of the two convolutional layers is 150 + 2,400 = 2,550 numbers.

This 1*400 matrix will then be sent to the NN model, which has three layers: 400*120, 120*84, and 84*36(84*10 for the digits-only model). This is similar to the two-layer NN model we've used before, so refer to section 3.2. The total numbers of weight for the NN portion is the sum of these three layers. model.lin1 has 400 * 120 = 48,000, model.lin2 has 120 * 84 = 10,080, and model.lin3 has 84*36 = 3,024. In total, the dimension of the three linear layers is 61,104. The total dimension that is used by weight_cnn_rom.sv will then be 61,104 + 2,550 = 63654. For the digits-only model, the total number of weights would then instead be 48,000 + 10,080 + 840 + 2,550 = 61,470

3.4. Assembly Firmware

From the firmware's perspective, the input image (integers between 0 and 255) is stored in image_mem starting at 0x00010000, and the pre-trained kernels or weights (in 32-bit Floating Point format) are stored in weight_rom (or weight_nn_rom or weight_cnn_rom) starting at 0x00020000. With help from the same machine learning model as the training processes described above, the firmware would accomplish the classification processes through programs written in our customized assembly language.

3.4.1. Main Function (CNN-Supercharged)

We chose to only explain the main function in CNN_supercharged.asm in this report, as it's the most complicated one compared to the main functions among all the firmware we developed.

The main function is an infinite loop, tailoring the different layers together through data memory (DM) and indefinitely triggering CNN classifications. Before entering a CNN, MAIN sends a snapshot request to the image processing unit and waits in SNAPSHOT_WAIT until an image is captured from the camera, compressed, padded, and stored in image_mem. Then MAIN would call PRE_PROCESS to convert the integers in image_mem to 32-bit floating point format and store them in DM.

After pre-processing, MAIN setup parameters such as a pointer to weight ROM, input matrix size, channel lengths, DM pointer to output matrix, and so on for the different layers of the CNN and call CONV, AVG_POOL, MATRIX_MUL, and OUTPUT_LAYER in the order as defined in section 3.3 above to do the work. For DM and RF usage, please refer to the following tables:

Starting Addr	**Ending Addr**	**Usage in this Program**
0	1023	Preprocessed input image after HW padding (2 on each side)
1024	5727	Output of first convolution layer
5728	6903	Output of first pooling layer
(re-usage start)		
0	1599	Output of second convolution layer
1600	1999	Output of second pooling layer
2000	2119	Output of first full NN layer
2120	2203	Output of second full NN layer
(re-usage end)		
7000	7399	Workzone for matrix multiplications
8000	8009	Final Scores of the ten classes

Register Name	Usage in this Function
R0	hard-wired 0
R1	snapshot trigger
R2	pointer to weight ROM
R3	pointer of image MEM
R4	input matrix size
R5	output matrix size
R6	convolution input channel length
R7	convolution output channel length
R27	Snapshot status
R28	reserved for result of matrix multiplication
R29	DM pointer to output matrix
R30	0x0000C000 base address of peripherals
R31	reserved for JAL/JR

3.4.2. Pre-Process Function

The PRE_PROCESS is fairly simple. It takes in a starting address and a matrix size as parameters, reads out each entry, converts each integer to FP format, and stores the results back to DM starting at address 0.

Params:	
R3	pointer of image matrix
R4	matrix size
Local Variables:	
R0	hard-wired 0
R5	DM pointer
R6	temp reg holding value being converted
R31	reserved for JAL/JR

3.4.3. Convolution Layer

The convolution layer is essential in the Convolution Neural Network. Research has shown that it proves the accuracy of the classification significantly. It also reduces the space required for weights. For fully-connected neural networks, we observed lower accuracy and less efficient use of memory than those of a Convolution Neural Network. The Convolution Layer extracts useful features, such as edge detection, Gaussian Blur, and filter, from the original image, reducing the input size for the next layers. The below image is an example of how the convolution is performed with a 3x3 kernel: A 3x3 subset of the original image is multiplied by the elements in the kernel correspondingly. And the sum of the 3x3 multiplication is outputted as a pixel in the output image.

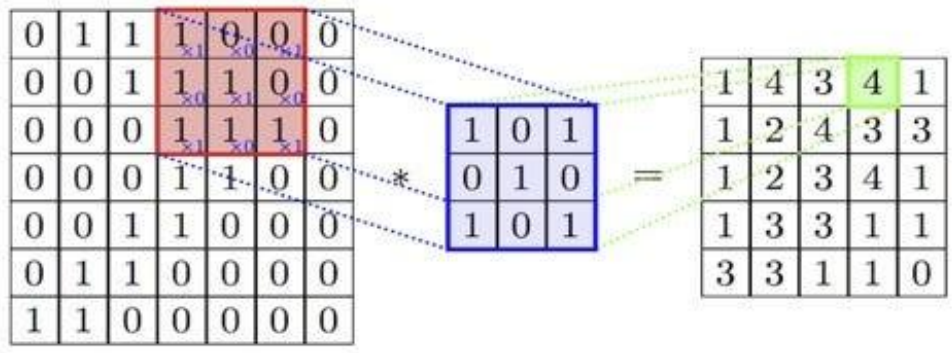

Figure 4: Visual representation of convolution. This image was adapted from towardsai.net.

Our architecture uses a 5x5 convolution kernel in each convolution layer. To perform the convolution with a 5x5 kernel, we need to perform 5x5x#input_channels to get a pixel for the output. Depending on the size of the output and #output_channel, we need to generate number-of-output-image pixels. For example, if the input is 6 of 32x32 images (6 channels) and we want to output 16 images, then the output is 16 of 28x28 images, and we need to perform 16x28x28x5x5x6 multiplications. We also need to perform 16x28x28x5x5x6 additions to finish this convolution layer.

To use our assembly firmware, the function needs to know the address of the input kernel, the address of the image, the size of the input image, the input channel length, the output channel length, and the address of the output. We used 25 registers to implement the convolution layer. The registers and use of the registers are listed below.

Params:	
R2	addr_kernel (start address of kernel)
R3	addr_image (start address of image)
R4	side_length_input (input side_length)
R6	in_channel_length (repeat the convolution calculation for multiple images with the same kernel)
R7	out_channel_length (repeat the convolution calculation for same image with different kernels)
R29	addr_output (start address of output address, increase by one when a pixel is calculated)
Local Variables:	
R4	side_length_output, same reg as input (will set to side_length_input - 4 to reflect the output side_length)
R5	Set to R4 - 1 for branch purposes
R8	x_result, x location of output images (start as 0, increase by one once a pixel is calculated. set to 0 when reaching side_length_output)
R9	y_result, y location of output images (start as 0, increase by one when x_result reaches side_length_output)

R10	pix_sum (sum of result at a result_pix)
R11~R15	5 weight registers (shared between all 25 weights)
R16~R20	5 pix registers (shared between 25 pixels, it also stores the mult result)
R21	image_length (use this jump distance to switch between input channels)
R22	base (a temp base location for pixel load)
R23	temp (intermediate for base address calculation
R24	channel_id (a down counter for keep track of the channel id in process)
R25	side_length_output, same reg as input (will set to side_length_input - 4 to reflect the output side_length)
R26	static copy of param out_channel_length

3.4.4. Average Pooling Layer

The average pooling layer comes after every convolution layer in our CNN. This layer has a function similar to the image compressor, but is performed by firmware. It takes a 2*2 pixel block from an image specified by the layer starting address, adds 4 pixel values, calculates the average value, and stores it back to data memory as a compressed pixel for the new image. It performs the steps above for images in every channel and generates new images of the number of channels. The aim of this layer is to reduce the amount of computation required while preserving the features of the image. Compressing the image also helps to prevent overfitting and improves the network's generalization ability.

Params:	
R3	layer starting address
R4	image width
R6	number of image channels
R29	output starting address
Local Variables:	
R0	hard-wired 0
R2	0.25F
R7~R10	pooled pixel

3.4.5. Neural Network (Matrix Multiplication) Layer

Fully connected NN can be abstracted to loops of 1D matrix multiplications. Thus, we designed this MATRIX_MUL function which takes two starting addresses, one pointing to the weight and the other pointing to the image (or input data of a middle layer), multiplies the two one-dimensional (flattened) matrices together, and returns the accumulated value. The caller would record and manage the real input and output dimensions and reconstruct the output matrix.

Params:	
R2	pointer to weight matrix
R3	pointer of image matrix
R4	matrix size
Return Value:	

R28	result of matrix multiplication
Local Variables:	
R0	hard-wired 0
R5	intermediate mult result store address
R6	image pixel value
R7	weight value
R8	multiplication result
R31	reserved for JAL/JR

3.4.6. Output Layer

The output layer is responsible for making a prediction based on the input data. In our case, there are 36 possible outputs, and the output layer determines the maximum result among all the possibilities. To achieve this, the layer loops through all 36 results and compares them with the current maximum value, replacing the current maximum when necessary. Once the maximum result is found, the output layer transmits the final prediction to the SPART module, which is connected to the processor. The prediction is in the form of an ASCII value that represents the predicted character.

Params:	
R29	base pointer to the output layer
Local Variables:	
R0	hard-wired 0
R2	loop index i
R3	loop terminate condition = 35
R4	current number
R5	current number - current max
R6	35 - i
R7	current max
R8	current max index
R9	0x00000030 ASCII number offset
R10	temp reg to distinguish between digits and letters
R30	0x0000C000 base address of peripherals
R31	reserved for JAL/JR

3.5. Self-checking Assembly Tests and Python Auto-Tester

3.5.1. Python Auto-Tester

The Python auto-tester works together with the cpu_tb.sv file and the asm tests. The protocol between these parts is that the asm tests are stuck at 0x00AD if they pass, and at 0x00DD if they fail. The cpu_tb.sv will wait for 300 cycles (which is more than most tests), then it will check if the PC is around 0x00AD or 0x00DD, and output a message of “Test pass” or “Test fail.”

The Python auto-tester has four functionalities: help, run one file, run all files, and clean.

"***python test.py help***" will display a help message on how to use the tester.

"***python test.py <filename>.asm***" will look for that .asm file in the test directory and assemble, compile, and simulate it.

"***python test.py***" will look for all .asm files in the test directory and assemble, compile, and simulate it.

"***python test.py clean***" will clean all .hex files that are in the test folder.

The Python script has to be in the same directory as all the .v and .sv files, and there needs to be another directory called "***test***" in this directory that contains all the .asm files and the assembler file for the corresponding ISA. The Python script will strictly look for instr_mem.sv for instruction memory file modifications, asmbl_32.pl for the assembler, and cpu_tb.sv for the testbench file. These names need to be unchanged. All the asm file names can be arbitrary.

For the testing functionality, the Python script will have to modify the instr_mem.sv file to run multiple tests. Before it does anything to this file, test.py will first remember what is currently in instr_mem.sv and save it. The script compiles all .v and .sv files once before all the other procedures. Then, it will run the assembler on any .asm files that need to be tested to generate a .hex file. The name of the .hex file will then be replaced in instr_mem.sv. The script will then re-compile the instr_mem.sv files so that the file name change will go through. Next, the script simulates the project and stores the output of the cpu_tb.sv into an output file. cup_tb.sv checks the pc and sees where it is stuck. If the pc is stuck around the pass loop, then it will display a "test passed" message, otherwise, it displays "test failed." The test.py script will check if the output file test.output contains "pass", then we can remove the output file and proceed to the next asm file. If the file contains "fail", then the script adds information about which test failed in the output file and blocs further testing. After all the testing procedures are finished, the test script will restore instr_mem.sv with its original content and display the testing results.

3.5.2. ASM Test Coverage

Our Assembly tests are designed to comprehensively validate the processor architecture, ensuring that all firmware are executed correctly. To achieve this, we have created unit tests for every original instruction and each newly introduced instruction. All test assembly codes consist of two infinite loops, each containing a branch to itself. The passing loop is at 0x00AD, while the failing loop is at 0x00DD. Each instruction test contains several test cases, all branch to the failing loop when an error occurs - only the last test case branches to the passing loop upon success. As a result, a passing unit test will eventually have its Program Counter (PC) at 0x00AD, while a failing test will have its PC at 0x00DD.

Our tests focus on validating the correct flags and data, bypassing instructions related to our extended ALU rather than the actual calculations. This is because the extended ALU primarily handles floating-point arithmetic, which has already undergone extensive testing as a Verilog module. The floating-point module's design team has conducted over 10 million randomized and 256 corner-case tests with special values, demonstrating the module's correctness and accuracy.

4. Engineering Standards Employed in your Design

4.1. IEEE 1800-2009 SystemVerilog

SystemVerilog is a hardware description and verification language used to design digital systems, with the aim of improving productivity in the verification of hardware designs. It is an extension of Verilog, which includes additional features like assertions, constrained random testing, and coverage measurement. For example, we used the casex feature of SystemVerilog to implement the floating point operations. We also used SystemVerilog to validate our design.

4.2. IEEE-754 32-bit Floating Point Standard

IEEE-754 is a standard for floating-point arithmetic published by the Institute of Electrical and Electronics Engineers (IEEE) in 1985. The standard defines formats for representing and manipulating floating-point numbers, which are used to approximate real numbers in computers. This project employs the single-precision 32-bit format to perform addition, subtraction, multiplication, and integer conversion on floating-point numbers. Hardware support for IEEE-754 32-bit floating-point arithmetic provides a broader range of representable values and greater accuracy than integer calculations.

4.3. VGA

The Video Graphics Array (VGA) protocol is an industry standard for displaying images on supported monitors. To meet the protocol's requirements for a stable output frequency of 25.2MHz on a 640x480 display, we utilized the embedded PLL to generate a 25MHz clock, achieving a frame rate of approximately 59 frames per second (FPS). We used a VGA Controller to control the data and timing requested by the VGA protocol. We feed our pixels directly from the SDRAM into the VGA Controller and reset the timing when the board is initialized. The VGA protocol operates at half the processor speed, with the data to be displayed being flopped in the VGA Controller.

4.4. UART (SPART)

The Universal Asynchronous Receiver-Transmitter (UART) is a widely-used industry-standard protocol for transmitting data between two devices. Our project used the UART protocol to transmit predicted results to the monitor via a USB cable. To enable buffering of the transmitted and received data, we added two FIFO buffers on either side of the UART signal. Each FIFO buffer can store up to eight one-byte data. We have named the upgraded module that facilitates this functionality SPART, and used it to send predicted characters to the display.

4.5. I2C

The Inter-Integrated Circuit (I2C) is a widely-used synchronous, multi-master/multi-slave (controller/target), packet-switched, single-ended, serial communication bus. In our project, we utilized the I2C protocol to adjust the camera's exposure, pause, zoom, and brightness settings. Specifically, we used I2C_CCD_Config to change the camera settings. Once the camera captures an image, it is processed (using image2Gray) and stored directly in the SDRAM.

5. Potential Societal Impacts of Our Design

5.1. Computing Efficiency in both Time and Cost

This hardware implementation of a Convolutional Neural Network (CNN) has the capability to recognize a single character within 90 ms and costs less than 500 USD in total. Unlike software that relies on extravagant GPUs, the FPGA is specialized and dedicated solely to this computing task. While its NRE cost is higher than that of software development, its deployment at scale is much less expensive, and it consumes less computing power than GPU-based systems. Additionally, the pipelined processor architecture and configurable assembly functions, such as matrix multiplication, convolution, and average pooling, allow for easy expansion to accommodate various image recognition demands by introducing new algorithms.

5.2. Smart Monitoring and Internet of Things (IoT)

Our system offers a versatile solution to monitoring readings from industrial equipment and sensors that lack external data ports, allowing for non-stop monitoring. Take, for example, a multimeter in a grid substation that is unable to transmit voltage/current readings to the data center due to its lack of capability. Rather than redesigning the multimeter or hiring additional laborers, our design offers an instant and effective solution that connects the multimeter to the IoT, enabling transmission of data to others. Our system can also pre-process readings to generate more useful data for users, improving reliability while reducing the need for human intervention. While our solution may cause manpower displacement, it ultimately provides greater reliability and non-stop monitoring capabilities.

5.3. Trend of Edge Computing Using ASIC Devices

The core paradigm behind this design is Edge Computing, which aims to bring computation and data storage closer to the location where the data is generated. By processing data locally and transmitting only the necessary data, edge devices reduce latency and bandwidth requirements, sometimes eliminating the need for data transmission. This approach improves reliability and scalability by distributing computing resources efficiently without relying on a centralized cloud infrastructure. It also enhances security by keeping critical data within a hardware system that is closer to its source. The applications of edge computing are rapidly emerging in various industries, including smart grid systems, autonomous vehicles, and industrial manufacturing.

6. Validation

In addition to module level System Verilog testbenches and self-checking assembly validations, we also validated our top level design with ModemSim simulations. The intermediate matrices and final scores reported by simulations were compared to their counterparts generated by our ML software using the same fixed image hex files. Note that with fixed hex images, our design takes about 3.7 million simulation cycles to classify one character. Since our software utilizes 64-bit floating point operations but our design uses 32-bit format, small errors at the least significant bits (when represented in decimal format) is allowed. Some example Python code we used to dump weight matrices, fixed images, and intermediate matrices are shared below:

```
write_file = open('test_weightfix.hex', 'w')
for i in range(len(model.conv1.weight.reshape(-1))):
    s_print = "@"+hex(i)[2:].zfill(4)+" "+hex(struct.unpack('<I',
struct.pack('<f', model.conv1.weight.reshape(-1)[i].item()))[0])[2:].zfill(8)
    write_file.write(s_print.strip()+'\n')
write_file.close()
```

Coding Example 1: Weight Matrix Dump

```
write_file = open('test_fixed_image_cnn.hex', 'w')
for i in range(len(comb_test[sample_idx][0].reshape(-1))):
    s_print = "@"+hex(i)[2:].zfill(4)+"
"+hex(int(comb_test[sample_idx][0].reshape(-1)[i]*255))[2:].zfill(2)
    write_file.write(s_print.strip()+'\n')
write_file.close()
```

Coding Example 2: Image Data Dump

```
write_file = open('test_con1.txt', 'w')
for i in activation['conv1'].reshape(6*28*28):
    write_file.write(str(i.item())+"\n")
write_file.close()
write_file = open('test_pool1.txt', 'w')
for i in
F.avg_pool2d(F.relu(activation['conv1']),kernel_size=2,stride=2).view(-1):
    write_file.write(str(i.item())+"\n")
write_file.close()
```

Coding Example 3: Prediction Data Dump

We compared selected sets of the intermediate layers and the final 36 scores of the classes and validated that our hardware design precisely executed all our different ML architectures. Selected screenshots of our validation processes are shared below:

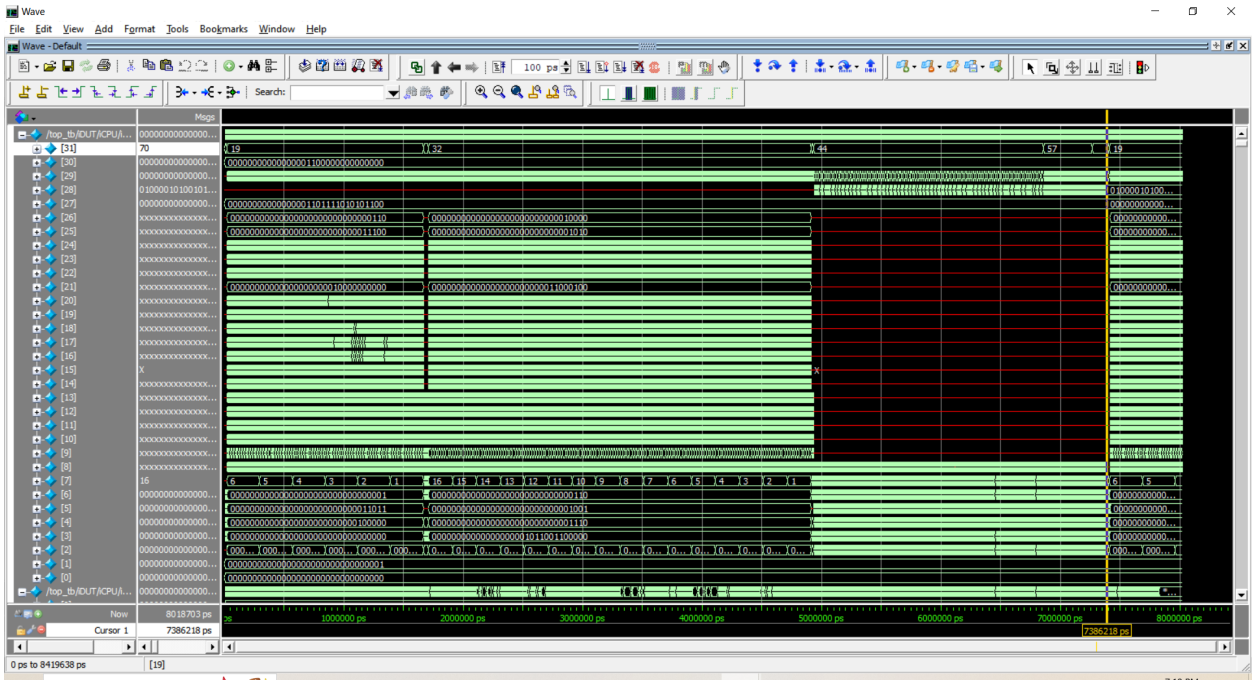

Figure 5: Register usage of a complete CNN classification process.

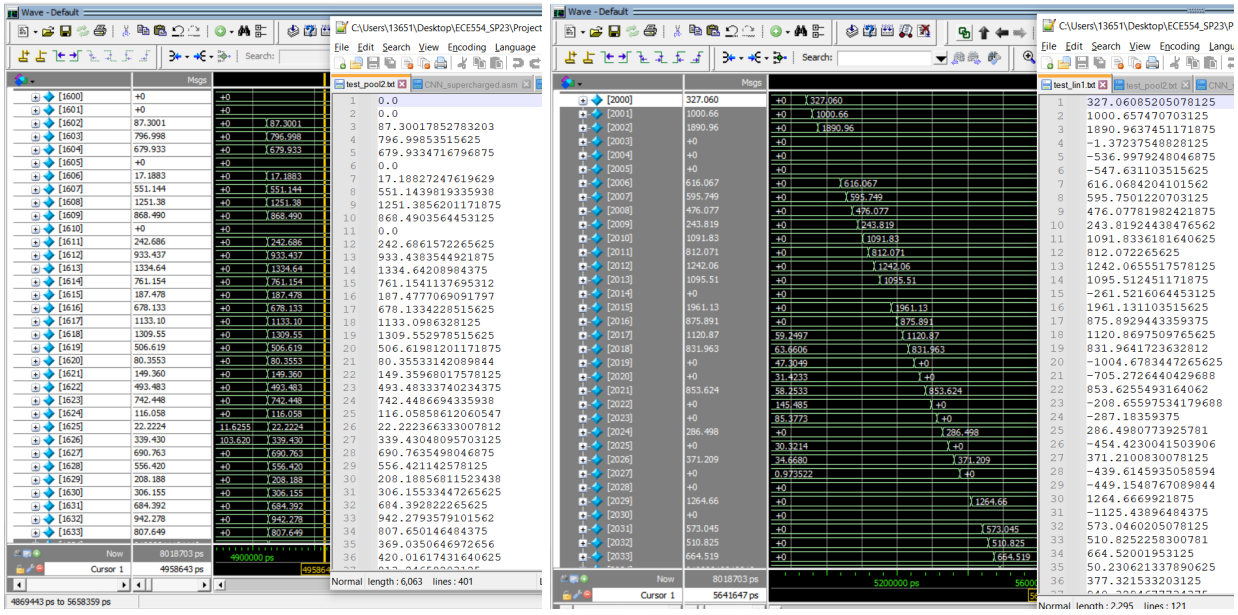

Figure 6: Intermediate layer validation - comparing FP values in DM of our processor to software dumps.

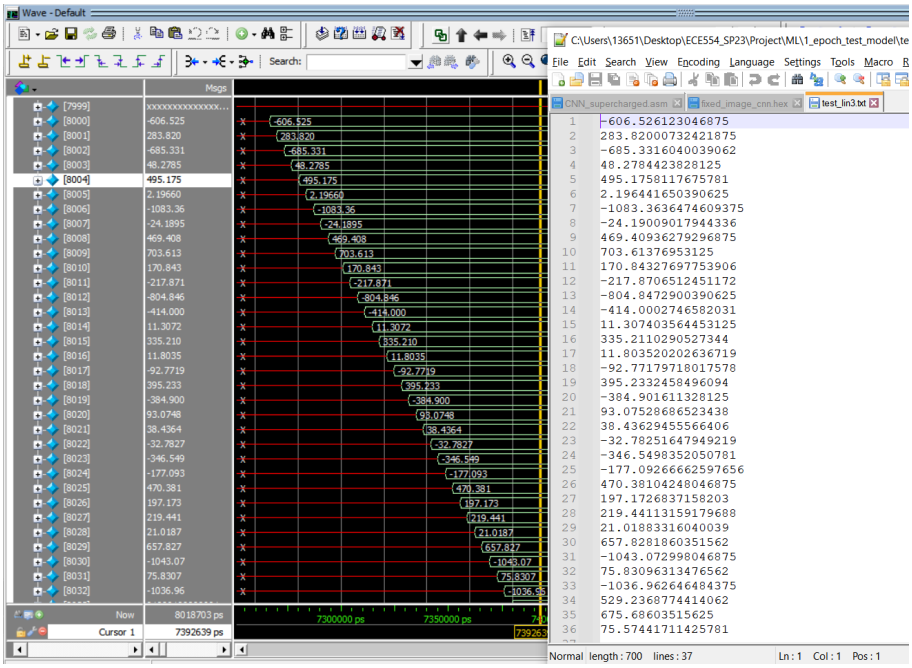

Figure 7: Final output validation of the 36 scores.

7. Final Application Demonstration

A demonstration video is posted on YouTube at https://youtu.be/7T7qIo2IxYQ

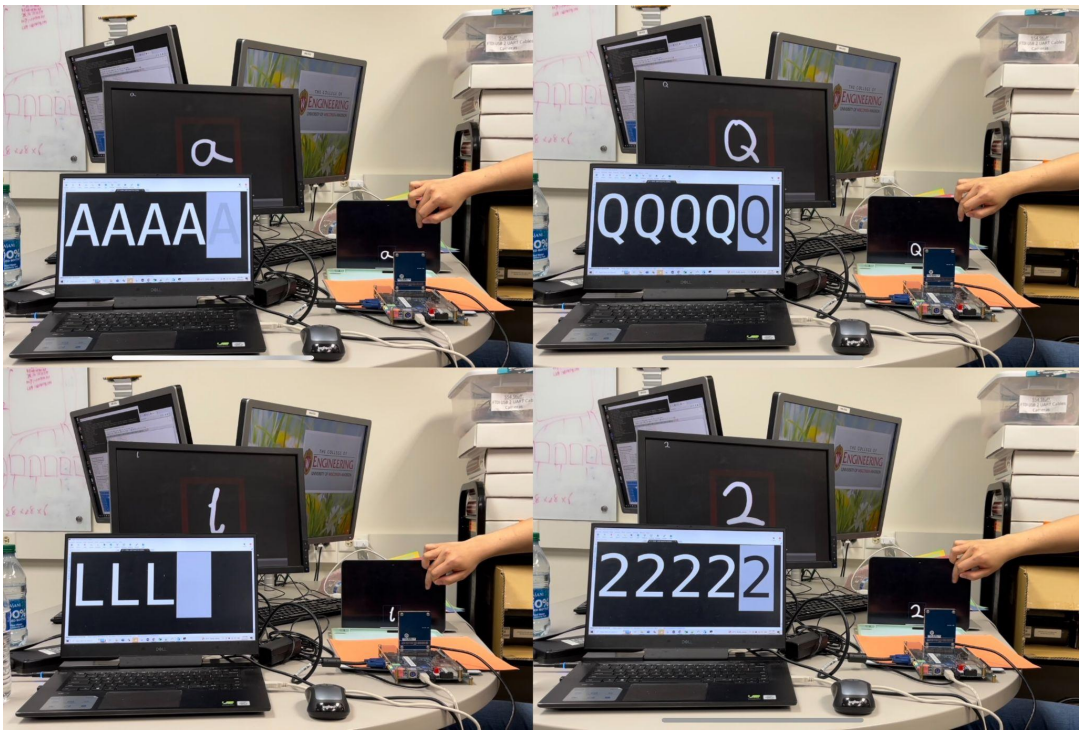

Figure 8: Selected screenshots from our video demo.

The images above display the results of our FPGA-CNN model's prediction of several handwritten characters. The bottom-left screen shows the prediction result obtained through UART, while the bottom-right corner features our FPGA board equipped with a camera. On the image's right-hand side, a hand can be seen swiping between sample handwriting on an iPad. Meanwhile, the monitor in the background displays the video feed captured by the camera. The red frame indicates the range of letter prediction. In the top-left corner of the monitor, a compressed 32x32 image of the target letter is echoed back.

8. Contributions of Individuals

Justin Qiao	Haining Qiu	Harry Zhao	Qikun Liu
Team Management	FP Adder Module	Top Level Integration	ML Software
Assembly Firmware	Image Processing HW	CPU Pipeline Expansion	Weight ROM Interface
Extended ALU Modules	Pooling Layer ASM	Camera Interface	Python Auto-Tester
FP Verilog Testbenches	New Inst Validation	Convolution ASM	Old Inst Validation