FPGA Implementation of CNN Handwritten Character Recognition

TEAM POOR HANDWRITING

Haining Qiu, Harry Zhao, Justin Qiao, Qikun Liu

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 in 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-like 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 used a 5-stage MIPS processor that we designed 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 individual contributions to this senior capstone project.

Project Final Report

Table of Contents

1. Repeat of ISA Table (updated from your project proposal doc)

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 checkout 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 can take advantage of it and run assembly validation tests of our processor automatically.

We kept R0 hardwired to 0 to efficiently test floating point values. For instance, when we have a FP value in R1, and we want to branch according to the sign of this value. We can't use ADD or ADDI for this because the ALU will not set flags properly for FP values. Thus, with this feature, we can do ADDF R1, R1, R0, and then branch safely and efficiently right below this ADDF instruction.

We still kept LLB and LHB as they were, since we feel something like LLW and LHW is confusing to 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_	ADD	5'b00	R1 <= R2 + R3	Saturating arithmetic.

	dddd_xxxs_ssss_xxxt_tttt	R1, R2, R3	000		Updates the Z, V and N flag registers
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		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				
NEQ		B NEQ, label	5'b01100000	Branch if Z=0	O is signed 12-bit offset in two's complement Branch target address = (Address of branch instruction + 1) + offset PC holds word addresses, each
EQ		B EQ, label	5'b01100	Branch if Z=1	

			001		instruction is 1 word, offset is specified as the number of instructions with respect to the instruction following the branch instruction.
GT		B GT, label	5'b01 100 010	Branch if {Z,N}==2'b00	
LT		B LT, label	5'b01 100 011	Branch if N=1	
GTE		B GTE, label	5'b01 100 100	Branch if N=0	
LTE		B LTE, label	5'b01 100 101	Branch if N=1 or Z=1	
OVFL		B OVFL, label	5'b01 100 110	Branch if V=1	
UNCOND		B UNCOND, label	5'b01 100 111	Branch unconditionally	
JAL	aaaa_axxx_xxxx_xxxx_xxxx_oooo_oooo_oooo	JAL label	5'b01 101	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'b01 110	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'b10 010	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'b10 011	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'b10 100	R1 <= R2 + C	I is 8-bit signed immediate value. Updates the Z, V and N flag registers
SUBI		SUBI R1, R2,	5'b10 101	R1 <= R2 - C	

		I			
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

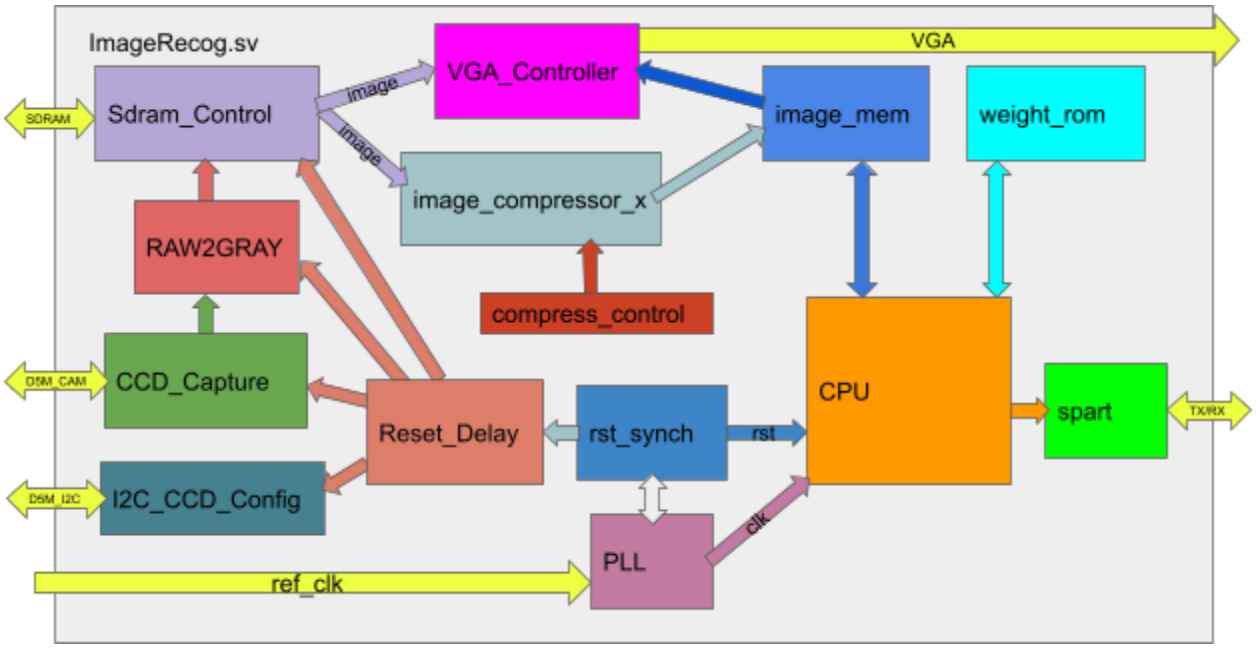
2.1. Top Level

The top level includes instantiations of the following modules.

A set of modules come from the tutorial of exploring 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 explained in the following sections.

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



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 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 ported 32x32 memory module to store one compressed image. It allows writing from the image_compressor_x and reading from CPU external memory access. When a image is captured and stored in image_mem, the data automatically adds 2 paddings to each side (from 28x28 to 32x32) by image_compressor_x to implement the LeNet-5 CNN architecture.

The weight_rom is a single ported memory module with all the weights necessary for CNN. There are a total 63,654 weights needed to implement the CNN. The CPU can read any weight in the weight_rom with a starting address of 0x20000. These weights are trained, tested 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 6x25 weights are for the first convolution layers
- The second 6x16x25 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 the 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. When the data stored in 0xC008 becomes 0, the compressed image is ready for use.

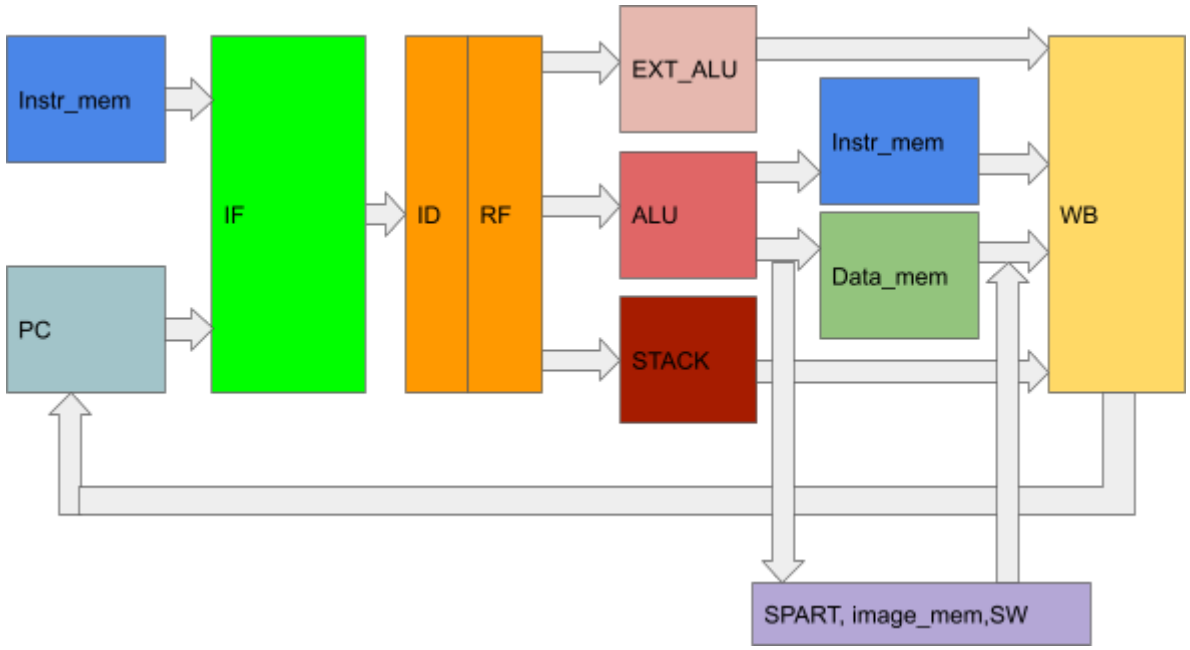
Signal:	Dir:	Description:
uncompress_addr_x [7:0]	In	X axis of the uncompressed image address. It is used to synchronize with the first pixel of the uncompressed image
uncompress_addr_ y[7:0]	in	Y axis of the uncompressed image address. It is used to synchronize with the first pixel of the uncompressed 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 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 compress control. If it is 0, there is no compression going on. If it is 1, a compression is in process. CPU has responsibility to access 0xC008 to check this status.
compress_start	out	The control signal to start a new compression.

2.2. Camera Interface

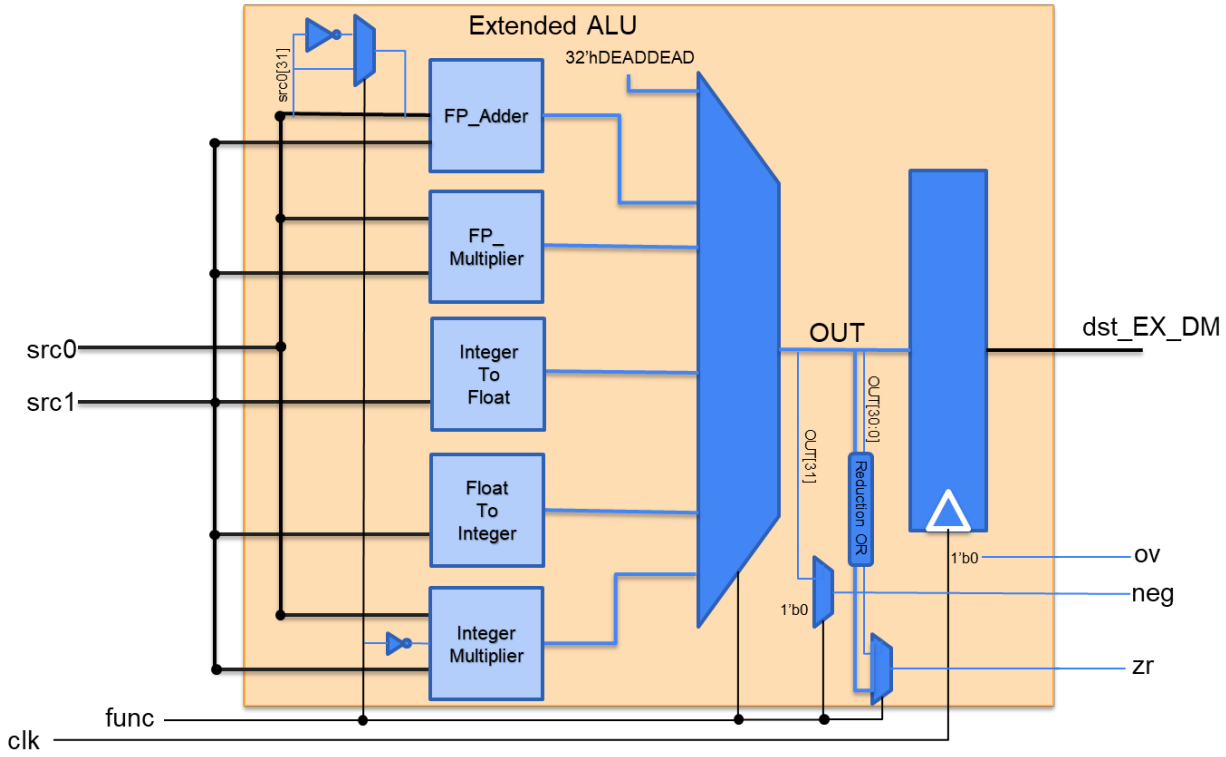
TODO

2.3. CPU

This is a high level block diagram of our modified processor. The original 16-bit pipeline logic was mostly preserved. We only expanded the data paths to 32 bits and added new modules. For detailed interface specifications, please refer to the following sections.



2.4. Extended ALU



The extended ALU is in the same pipeline stage of 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. Its output value and flags are selected by the func signal.

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 mantissas. Here are the steps involved in the FP Adder process:

1. Compare the exponents and determine the smaller one. Calculate the absolute difference between the exponents, and the larger exponent will be the common exponent.
2. Prepend the common exponent to both mantissas, 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 mantissas 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, increment the common exponent. Note that this overflow is an internal overflow, not an external value overflow.
6. Convert the 25-bit 2's complement result back to a 25-bit signed number, where the MSB is the final sign and the rest 24-bit are 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

The process for 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 offset accounted for. The mantissas are then appended with an implicit 1

(or 0) and multiplied together. 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 single 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

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 it takes one additional cycle to reach the data memory stage.

This stack is implemented in the same fashion as the FIFO buffer, except that it will push to the top of the stack and pop from top of the stack. This stack is a First-In-Last-Out(FILO) buffer. A list of registers is instantiated and the address of access 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

The image taken from the camera is stored in SDRAM, which is then sequentially fed into the image compressor. This image compressor sequentially takes a 224*224 8-bit image from SDRAM and compress 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 specifically designed for CNN model, which requires a 32*32 image with zero-padding of width 2 that surrounds 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)

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)

2.6.2. 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.

2.6.3. 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

2.6.4. 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.mnist - 60,000 images with labels	weight.hex - contains 7840 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 does its work in three steps: it reads train data in, trains on the data, and produces the weight matrix. It is currently using the Keras dataset that contains 60,000 different images of 28*28

pixels of handwritten number, and corresponding label to indicate their values. The first 40,000 images are used for training, and 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 softmax loss linear classifier, which utilizes the difference of 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 the performance and reduce the 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 flatten into a list of 784 = (28*28) floating-point numbers. For each row, each 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

3.3. Machine Learning Model - Convolutional Neural Network

3.4. Assembly Firmware

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

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 that was first 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 compared to integer calculations.

4.3. VGA

The Video Graphics Array (VGA) protocol is an industry-standard protocol 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. In our project, we utilized 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 as SPART, and it is used 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 exposure, pause, zoom, and brightness settings of the camera. 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 into 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.

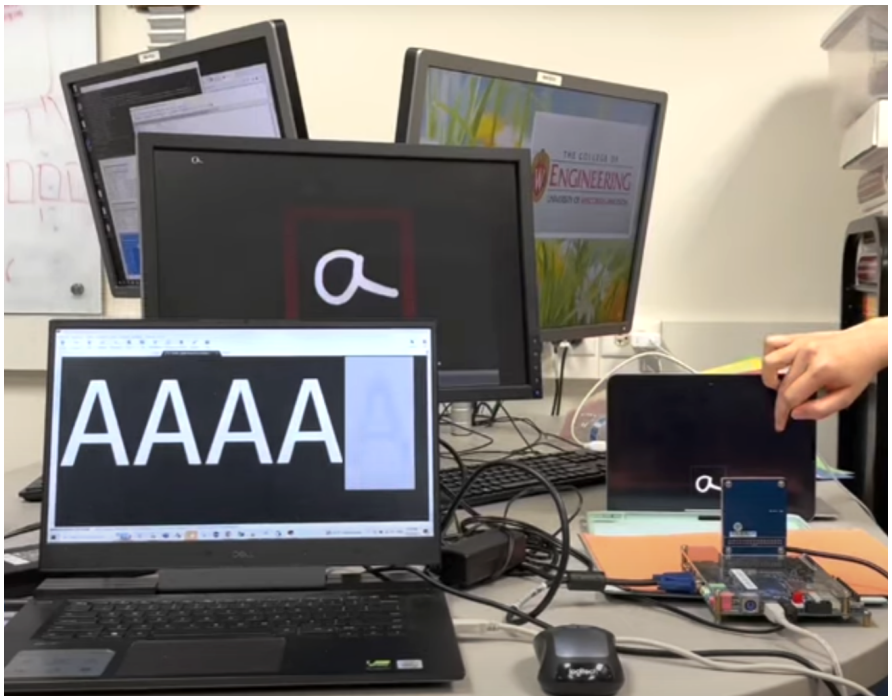
5.3. Trend of Edge Computing Using ASIC Devices

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. Final Application Demonstration

6.1. Demonstration of Application

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



The above image shows that our FPGA-CNN model predicts the hand-written “a” as a letter A. (The bottom-left corner of the image is the predicted result from UART, the bottom-right corner of the image

has the FPGA board with a camera installed. The hand on the right side of the image is swiping between sample images on an iPad. The monitor in the back shows the video captured by the camera. The red frame indicates the range of letter prediction. On the top-left corner of the monitor, a compressed 32x32 image of the target letter is echoed back.)

6.2. Validation

6.2.1. Convolution Layer Validation

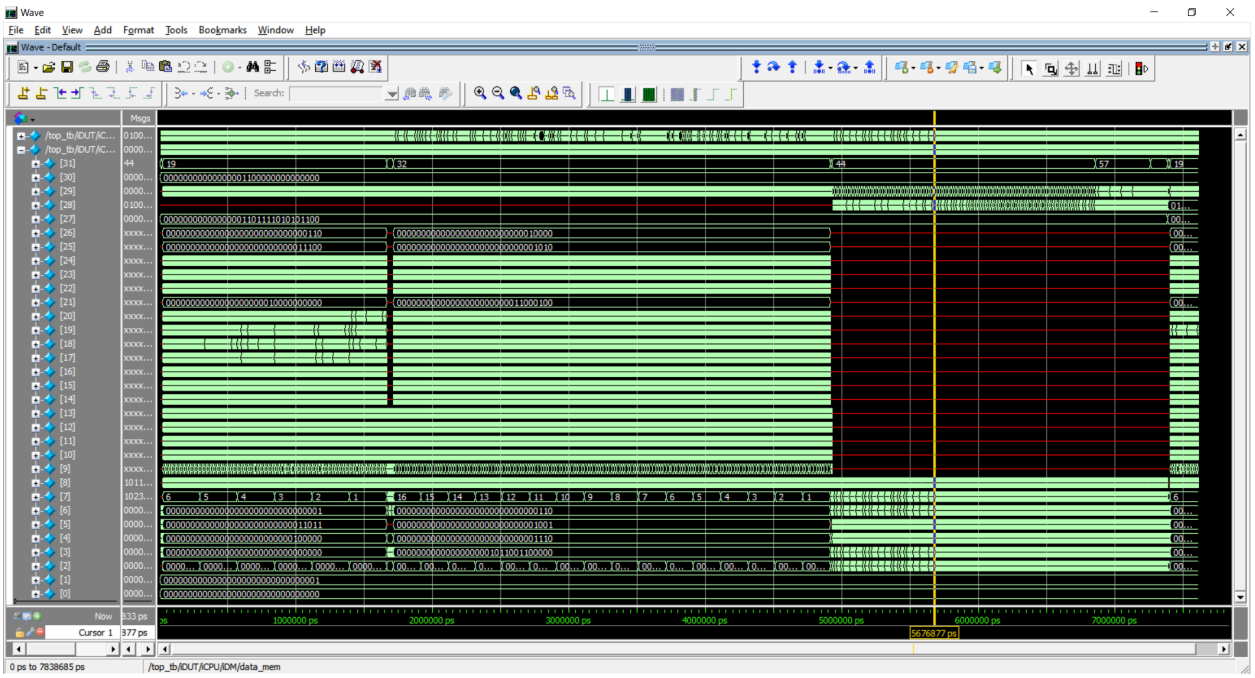
6.2.2. Neural Network Layer Validation

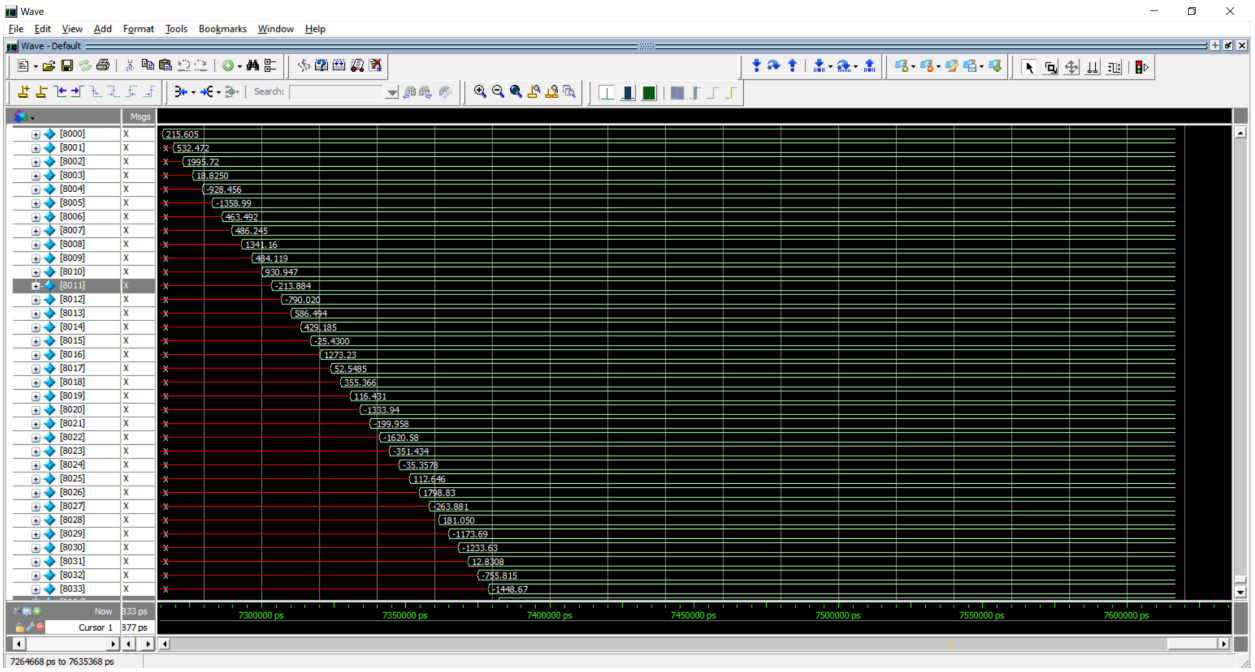
6.2.3. Final Result Validation

Need to mention the time takes to finish the simulation (3690608 cycles)

6.3. Demonstration of Application

6.4. Demonstration of Application





7. Contributions of Individuals

Justin Qiao	Haining Qiu	Harry Zhao	Qikun Liu