Hardware Design of

Real-time Classification Algorithm

*TEAM POOR HANDWRITING*

Haining Qiu, Harry Zhao, Justin Qiao, Qikun Liu

A picture containing logo

Description automatically generated

Interface Document

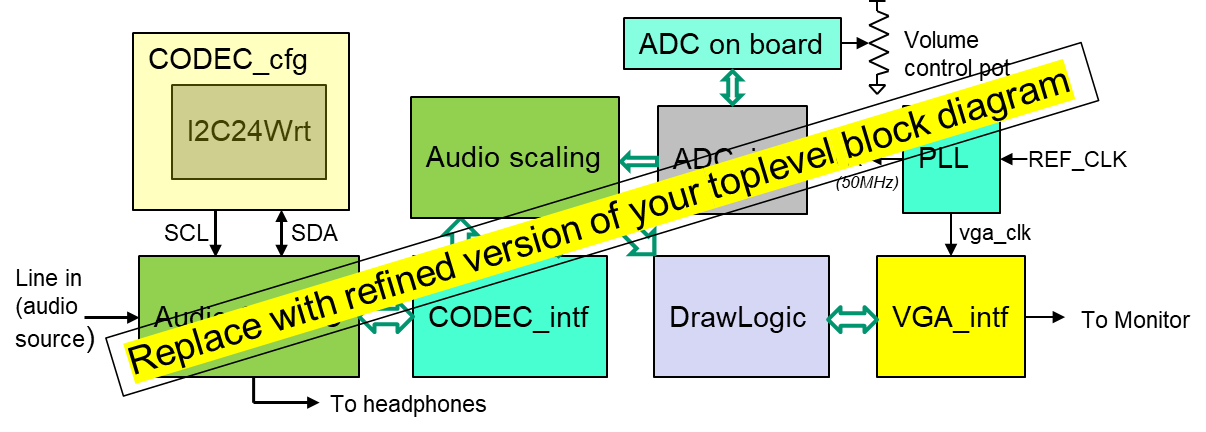
1. Hardware Block Diagram
   1. Top Level
   2. CPU
      1. PC Interface
      2. PC Registers
   3. Extended ALU
      1. Floating-point Adder
         1. Left Shifter Interface
         2. Right Shifter Interface
      2. Floating-point Multiplier
      3. Float-to-integer Unit
      4. Integer-to-float Unit
      5. 16-by-16 Integer Multiplier
   4. Stack
   5. Image Compressor and Storage
      1. Image Compressor Interface
      2. Image Compressor Register
      3. Image Memory Interface
      4. Image Memory Registers
2. Software
   1. Machine Learning Software
   2. Assembly Firmware

# Hardware Block Diagrams

## Top Level

<TODO>

Refine the hardware block diagram in this document. You may need separate block diagrams for some of the more complex functional units. The CPU would fall into this category. Each unit that you are defining interfaces for should be a “block” in a presented block diagram. CPU block is too complicated to write as an interface. In addition to signal interface we wish to see a description of any significant registers within a unit.



## CPU

This is a high level block diagram of our modified processor. For detailed interface specifications, please refer to the following sections.



### PC (Program Counter) Interface

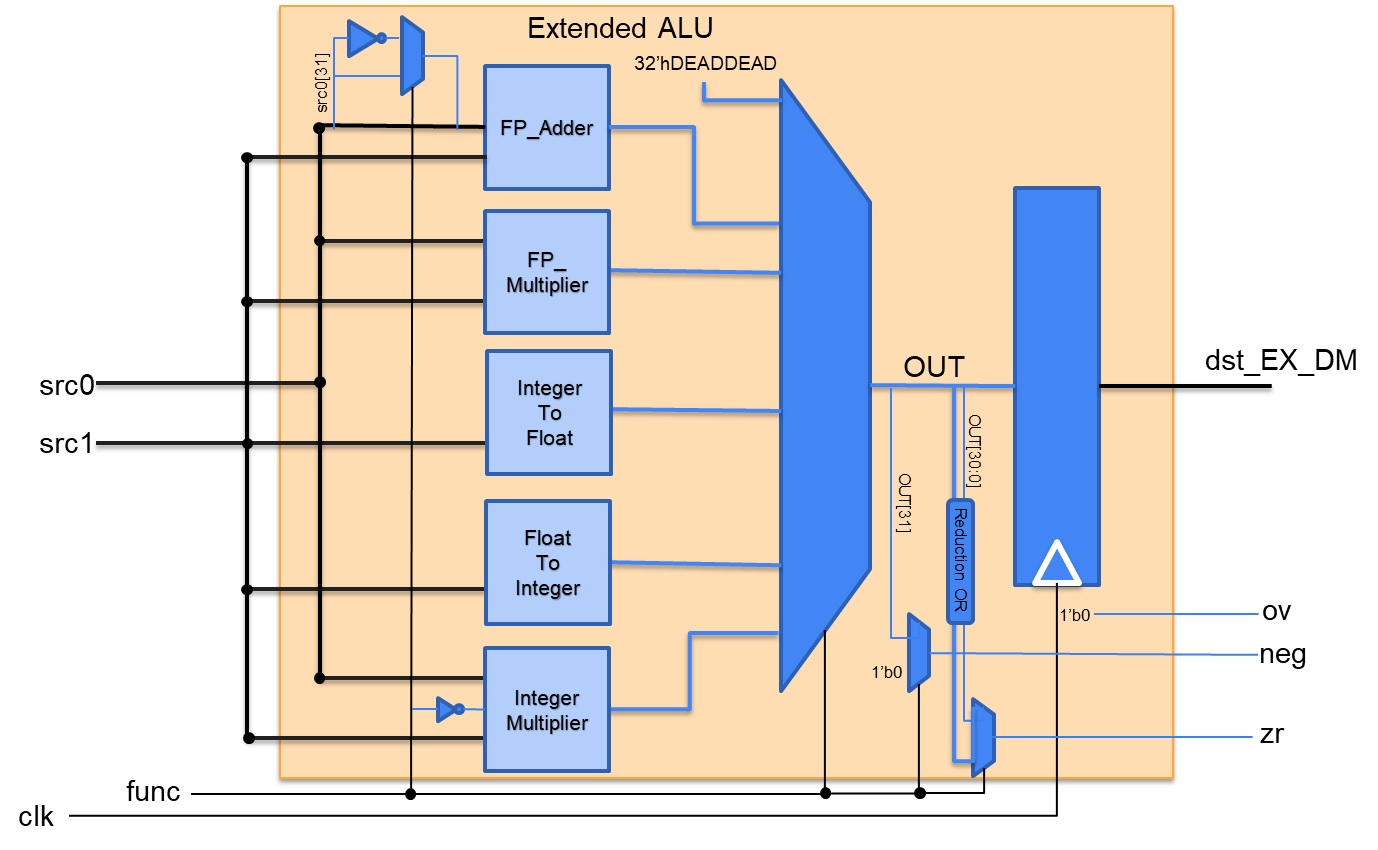
The PC controls the flow of instruction execution. This unit compute the next instruction to fetch based on the instruction flow and CPU stall logic.

| **Signal:** | **Dir:** | **Description:** |
| --- | --- | --- |
| Flow\_change\_ID\_EX | In | Indicates PC will be updated due to branch or jump (from ID unit) |
| stall\_IM\_ID | in | Asserted if stalling pipe due to load/use or MOVC related stall |
| dst\_ID\_EX[31:0] | In | Branch target address from ALU |
| pc[31:0] | out | Forms address to instruction memory |
| PC\_ID\_EX[31:0] | out | Piped version needed in EX stage for computing new branch target |
| PC\_EX\_DM[31:0] | out | Piped to EX\_DM for JAL instruction store in R15 |

### PC (Program Counter) Registers

| **Register:** | **Description:** |
| --- | --- |
| dst\_ID\_EX, pc, PC\_ID\_EX,PC\_EX\_DM  All 32-bits | Described in table above |
| Flow\_change\_ID\_EX, stall\_IM\_ID  All 1 bit | Described in table above |
| pc\_IM\_ID[31:0] | Updated PC if no stall happens. |
| nxt\_pc[31:0] | The expected next pc if no branch or jump instruction (pc+1) |

## 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 |

### Floating Point Adder Interface

| **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 |

### 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 |

### 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 |

### Floating-point Multiplier Interface

| **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 |

### 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 |

### 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 |

### 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 |

## Stack

| **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 |

## Image Processing and Storage

### 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 an 28\*28 image) |

### 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. |

### 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 783) |
| rdata[7:0] | out | 8-bit read port of SRAM for a compressed pixel color value |

### Image Memory Registers

| Register: | Description: |
| --- | --- |
| rdata[9:0][0:783] | Described in table above. There are 784 SRAM blocks just enough for a compressed 28\*28 image. This SRAM is written by the image compressor. |

# Software

## **Machine Learning Software**

| **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.

## **Assembly Firmware**

| **Register:** | **Description:** |
| --- | --- |
| R1 | 1 |
| R2 | pointer to weight matrix |
| R3 | pointer of image matrix |
| R4 | pointer to DM |
| R5 | loop index |
| R6 through R16 | compute regs |
| R17 | loop number 9 (i <= 9) |
| R18 | current number |
| R19 | current number - current max |
| R20 | 9 - i |
| R21 | current max |
| R22 | current max index |
| R23 | SW status |
| R24 | SW1 mask |
| R25 | step size |
| R27 | 0x00000030 ASCII number offset |
| R28 | 0x0000C000 base address of peripherals |
| R29 | matrix index |
| R30 | result pointer, results will be in DM at addr = 1000 through 1009 |

Our firmware contains three main layers. The first layer is executed once for each requested matrix multiplication. It obtains one number from image\_mem and one number from weight\_mem, multiplies them together and stores the result to the DM. 784 of such multiplies are calculated and stored in DM[0] through DM[783] in FP format. The second layer is a tree adder of the 784 products to obtain a score. We first add every 7 consecutive numbers together to reduce the 784 numbers to 114. Then add every 7 consecutive numbers of the 114 values again to get 16 numbers. FInally we add the 16 numbers together to get the ultimate score. Each three sub-layers of additions are done with tree adders at the assembly level to mitigate FP addition errors caused by adding two FP numbers with big differences in their exponent values. Finally, a third layer is used to find the max value of the calculated scores for each possible character, and print the prediction result (a ASCII character) to the terminal via Spart.