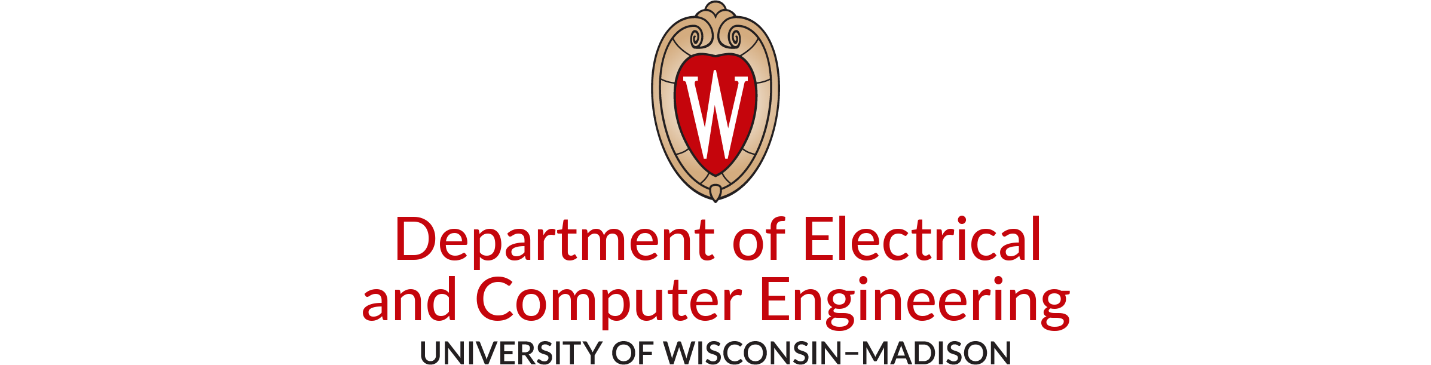
Hardware Design of

Real-time Classification Algorithm

TEAM POOR HANDWRITING

Haining Qiu, Harry Zhao, Justin Qiao, Qikun Liu



Project Proposal Document

**Table of Contents**

[1. Introduction 2](#_heading=h.gjdgxs)

[2. Hardware Block Diagram 3](#_heading=h.30j0zll)

[2.1. << Block X Description >> 3](#_heading=h.1fob9te)

[3. Processor 3](#_heading=h.3znysh7)

[3.1. ISA Summary 3](#_heading=h.2et92p0)

[3.2. Condition Codes 4](#_heading=h.tyjcwt)

[3.3. Addressing Modes 4](#_heading=h.3dy6vkm)

[4. Software Blocks 4](#_heading=h.1t3h5sf)

[4.1. Assembler/Compiler 5](#_heading=h.4d34og8)

[4.2. Simulator 5](#_heading=h.2s8eyo1)

[4.3. Application 5](#_heading=h.17dp8vu)

[5. Division of Labor 5](#_heading=h.3rdcrjn)

# Introduction

The purpose of this project is to recognize a handwritten character image using the TRDM-D5M camera mounted on the DE1 FPGA board. The captured image will be processed by the hardware, namely the extended 552 processor with our assembly code.

The weight of the softmax linear classification is obtained using python pre-training algorithm and is preloaded into the ROM on the FPGA.

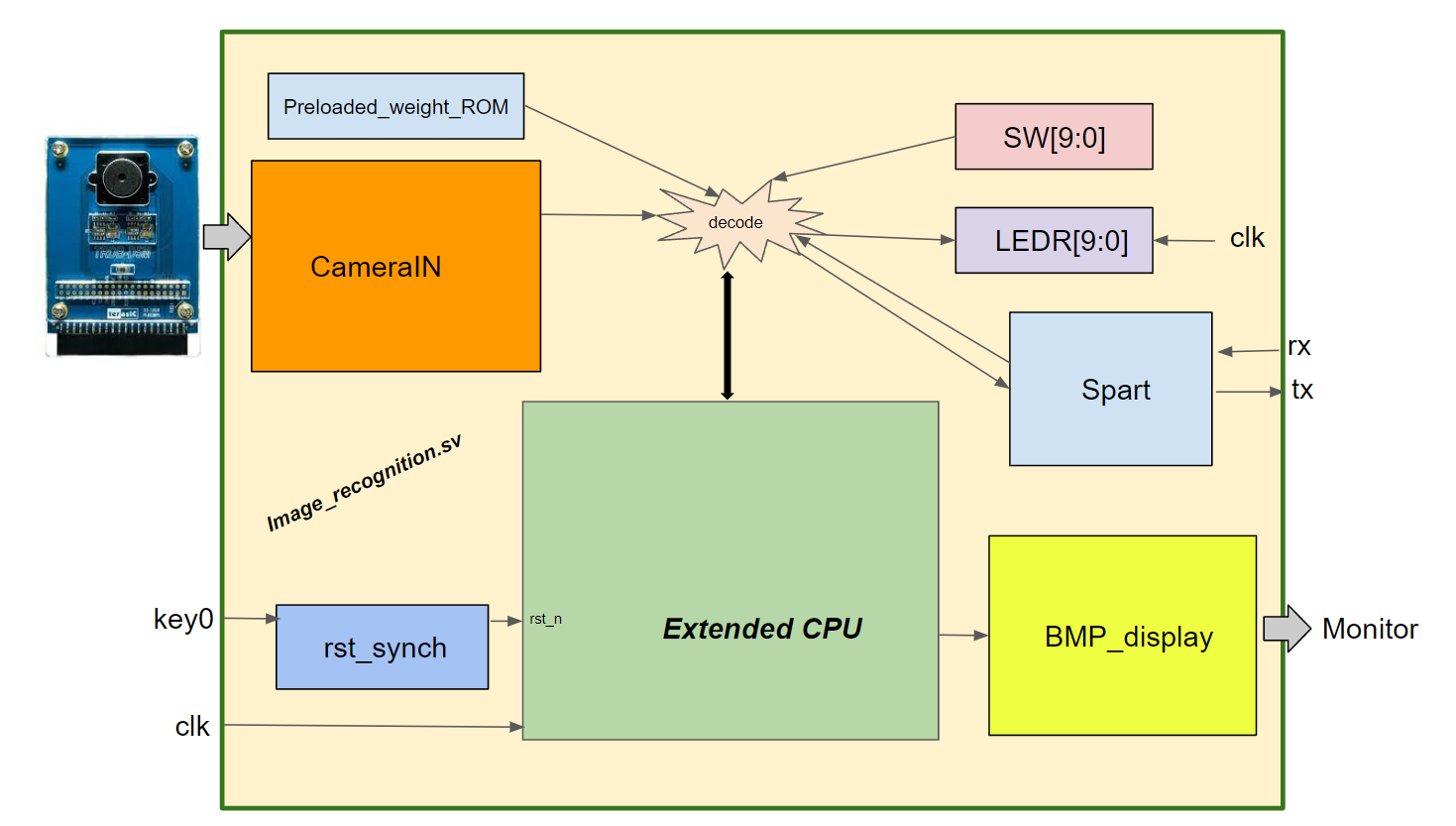
By using floating point matrix multiplication to process the image, we will obtain the predicted image category with high precision.

Finally, we will display the predicted image through UART to a host machine. An alternative would be displaying the classified result through VGA.

# Hardware Block Diagram

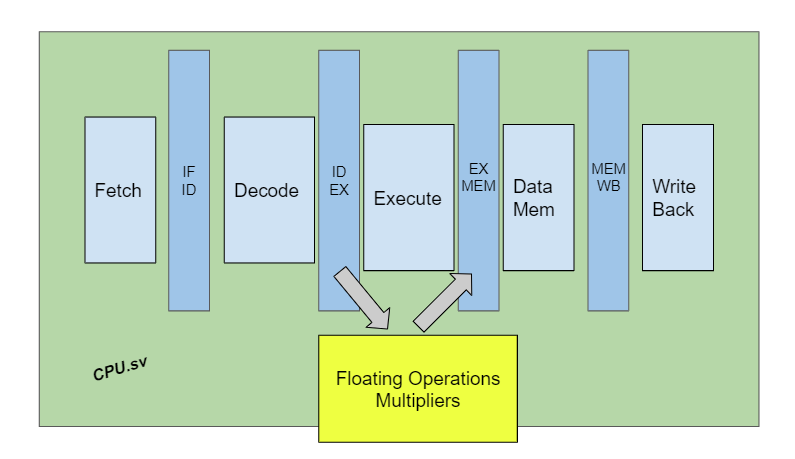
## Top Level Block Description

The top level of the design takes the signal of a camera into a reserved memory , then the image is filtered into grayscale. We will use the CPU to multiply the received image with preloaded weight. The SW and LEDR are used for debugging and they can be accessed at address 0xC000, 0xC001, respectively. Spart can be accessed using addresses from 0xC004~0xC007. The Spart module is used to communicate with the host computer for displaying results. The Extended CPU will be able to perform multiplication with 32 bit floating inputs. The BMP\_display module will contain a VGA output signal control and a memory to store outputs on the VGA signal. It will echo a part of the received image.



## CPU Block Description

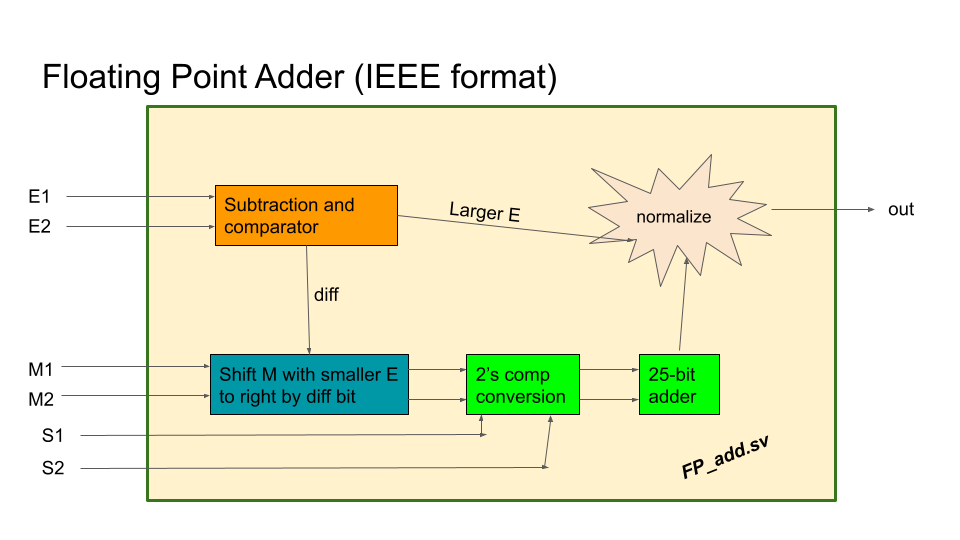
The CPU block will be extended into 32 bit wide for instruction memory and data memory. The depth of data memory and instruction memory will be determined based on the available memory on the FPGA board. The CPU will support the new ISA defined in section 3.1. The floating point operations and multiplier will be added into the execute stage and combined into an advanced ALU. If we decide to up scale the project scope, we will add multiple ALU (regular and advanced) in the execute stage to implement parallel instruction execution. If superscalar is implemented, we will need wide fetch, wide decode, wide data memory write and wide write back. The stall logic will be more complex. If we decide to add out of order processing, then we need to add two more stages, Issue stage and reorder buffer stage.



## Floating Point Adder Block Description

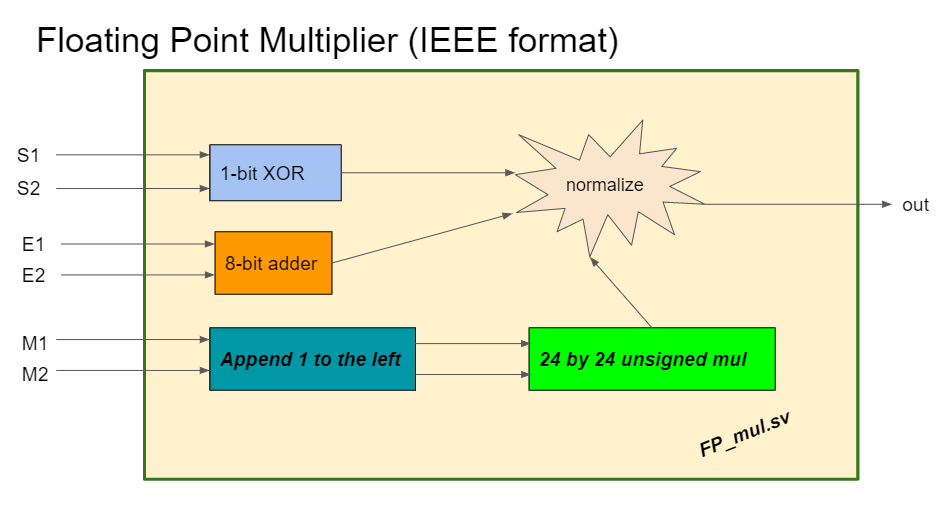
The addition of two floating-point numbers is complex, since their exponents could differ by a great amount and their mantissas are unsigned representations. The following steps describe how the FP Adder works in detail.

1. Compare two exponents, determine the smaller one, and calculate the absolute difference between them. The larger E will be the common E (for now).
2. Append |E (reduction OR of exponent) in front of both mantissas. Now both numbers of interest are of 24-bit.
3. Shift the mantissa (including the appending digit) with a smaller exponent to the right, where the shift amount is the lower 5 bits of the difference between exponents. The maximum shift amount should be 22-bit, or the shifted number is too small to be considered.
4. Convert both appended then shifted mantissa into 2’s complement format by looking at the sign bit. Now both numbers are in 25-bit 2’s complement.
5. Add two 25-bit numbers and get a 25-bit result. If the result overflows either positively or negatively, an increment in the common exponent is needed. Note that this overflow is NOT an external value overflow (the FP number cannot be represented) but an internal overflow (the result can be correctly represented).
6. Convert the 25-bit 2’s complement back to a 25-bit signed number with the MSB being the final sign and the rest 24-bit being the unsigned value. This step should have no loss of precision. Now, the MSB is the sign, and the rest 24-bit is to be further processed.
7. If an overflow occurs, shift the lower 24-bit to the right, appending 1 in MSB and discarding the LSB; if the lower 24-bit has leading zeros, a left shift is needed, where the shift amount is determined by the number of leading zero(s), and the common exponent is decremented by the shift amount. This 24-bit result should have a MSB of 1 now.
8. The resulting mantissa will be the lower 23-bit of the final 24-bit result. The exponent is the final common exponent. The sign is the MSB of is the final sign.



## Floating Point Multiplier Block Description

The inputs are two 32-bit values in IEEE FP format. They are first broken into {S1, E1, M1} and {S2, E2, M2}. During normal operations, the signs are XORed to get the sign of product; the exponent values are added together (with the 127 offset handled); and the mentisas are appended with an implicit 1 (or 0) and then multiplied together. When special values (such as -INF, -0, +0, and +INF) are involved, the outputs are adjusted by special combinational logic. Finally, the resulting values are concatenated and normalized to compile the IEEE FP standard and output as a single 32-bit value.



## Processor

### ISA Summary

**CPU** **HW resources**: 32 bit Instruction Memory (size TBD), 32 of 32-bit logical registers (two physical copies for implementation, R0 reserved for read-only 32’h00000000, R31 reserved for JAL and JR), 32 bit Data Memory (Size TBD)

We expand the provided 552 ISA to a 32-bit one while maintaining the original 16-bit structure. All original instructions preserve their opcodes with an extra bit 0 added in MSB. All new instructions start with bit 1 for clear differentiation. The register file is also expanded to 32 registers with a 5-bit selecting address. This expansion enables functions including floating-point operations, integer multiplication, stack management, and instruction read (LWI).

https://docs.google.com/spreadsheets/d/1PT7VjIhUPUwOg7ZNtqeGGRNTjavUGF0D/edit?usp=sharing&ouid=115717939913064443857&rtpof=true&sd=true

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 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'b01100 000 | 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 instruction is 1 word,  offset is specified as the number of instructions with  respect to the instruction following the branch  instruction. |
| EQ | B EQ, label | 5'b01100 001 | Branch if Z=1 |
| GT | B GT, label | 5'b01100 010 | Branch if {Z,N}==2'b00 |
| LT | B LT, label | 5'b01100 011 | Branch if N=1 |
| GTE | B GTE, label | 5'b01100 100 | Branch if N=0 |
| LTE | B LTE, label | 5'b01100 101 | Branch if N=1 or Z=1 |
| OVFL | B OVFL, label | 5'b01100 110 | Branch if V=1 |
| UNCOND | B UNCOND, label | 5'b01100 111 | 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) |
|  |  |  |  |  |  |
| LWI (movec) | aaaa\_axxx\_xxxd\_dddd\_xxxs\_ssss\_oooo\_oooo | LWI R1, R2, O | 5'b10000 | R1 <= InstMem[R2 + O] | Load instruction with address = R2 + 2's complement Offset signal |
|  |  |  |  |  |  |
| 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 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| 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 |  |

### Condition Codes Z-zero, V-overflow, N-negative/sign

### Addressing Modes

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  
For more specialized address modes, please refer to the ISA table above.

#### Immediate

We support 5-bit unsigned, 8-bit signed, 12-bit signed, and 16-bit signed immediate(offset) for different instructions - please refer to the ISA for detail.

#### Harvard vs Von Neumann

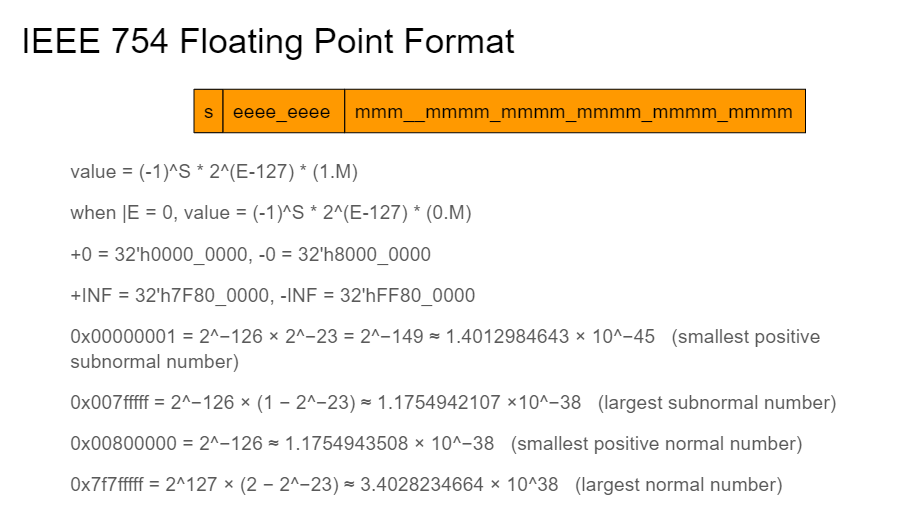
We will have two copies of instruction memory. Therefore, we will use one instruction memory to perform regular instruction execution. We will use another instruction memory to perform movc. When the board is loaded with instructions, two same copies of instructions will be loaded to allow multiple access at the same time. If we proceed with implementing super-scalar, then we need to instantiate more copies of instruction memory to support wide-fetch.

#### Special Features

We have not planned any special performance feature for the basic project goals, but we plan to do superscalar if time allows. See section 3.8 for details.

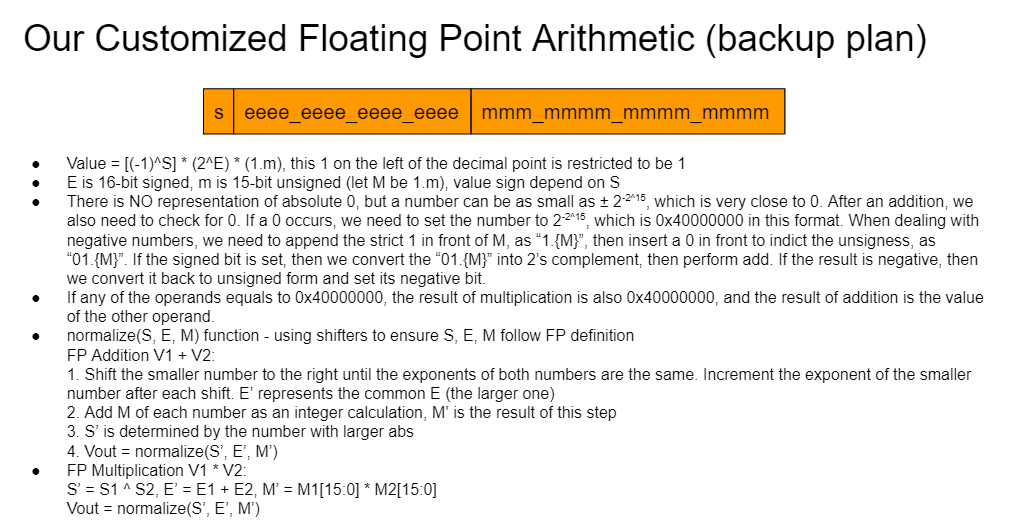
#### Co-processors

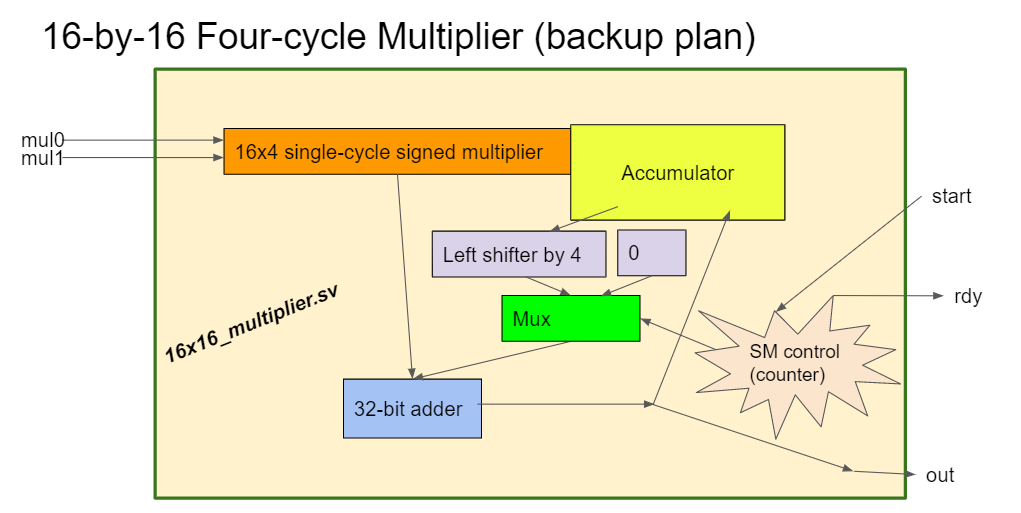
We support IEEE floating point standard additions, subtractions, and multiplications. FP addition and subtraction will be one module; FP multiplication and 16 by 16 (signed/unsigned) multiplication will be another module; and these two modules will sit aside the ALU during pipeline processing.



We will share a 25 by 25 (appending sign bit accordingly) multiply multiplier with just a \* sign for IEEE FP multiplication, 16 by 16 signed multiplication and 16 by 16 unsigned multiplication in the beginning. This entails we will have a special module called multipliers sitting aside as the ALU to handle these three operations.

If this logic does not fit on the FPGA, we will proceed with the multi-cycle multiply, this entails we won’t follow the standard IEEE floating point definition, since we will only support 16 by 16 integer multiplication.





#### Other architectural features

We plan to implement superscalar processing (maybe with out of order execution) after we validate all aspects of the design described above. More specifically:

1. Specialized operation for the matrix multiplication on FPGA - we are considering the possibility of loading multiple data of the matrix and performing multiple multiplications simultaneously. (Tiny expansion of superscalar, VLIW, multi-cycle calculation)
2. Superscalar processor to perform the calculation of matrix multiplication (without Out-Of-Order schedule algorithm). This project would have little dependency in assembly code, so a superscalar processor could still perform well without OOO processing.
3. Out-Of-Order execution. This is the ultimate form of the project (unlikely to be accomplished). This will increase the performance of the classifier.

# Software Blocks

Mention all the software blocks present in your project. These can be one or more of the following.

1. Assembler
2. Python
3. Compiler
4. Simulator
5. Application

## Assembler

We will implement a 32 bit assembler to support translation between assembly code to machine code based on our 32-bit ISA. The assembler will correctly ignore comments, translate both hex and decimal address to hex instructions. It will have line numbers.

* 1. **Firmware Overview**

A different firmware will be used to train the linear classifier before we load the model into our processor. Specifically, we will use Python and its PyTorch library for such purposes. The data used for this process will be either generated by manually taking pictures with the TRDM-D5M camera, and then transferring the data into smaller 2-d matrices, or finding handwriting data from open online sources, and then modify the data into the same format with what we read in from the camera; depending on the success of finding open sources. Then, softmax training techniques will be used for training one single linear classifier. The test validation accuracy can be around 40% - 80%.

The finished model will be a 2-dimensional matrix, with dimensions of C\*D, where C stands for the number of classes, in this case it will be 26(letters)+10(digits) = 36(classes); and the D stands for the number of dimensions for one input, which will be the width\*height of the camera picture(TBD). This will be used later to predict the label of an input image. This will be done by performing matrix multiplication between the weight and the input image, which the dimensions will be (C\*D).(D\*1) = (C\*1), which means we will end up having a total of C numbers, representing the final score our model gave to each of the classes. Among these C numbers, the highest value will be chosen as our final prediction; and our processor should then transmit the prediction to the UART device.

* 1. **Compiler**

A compiler will also be used for coding this application. Since matrix multiplication needs to be used, and the processor only supports floating point addition and multiplication, the software will need to unroll the matrix multiplication into many floating point multiplications and additions. A compiler could be used for smoothing this process. The compiler will be able to scanner the basic version of Java, including all the basic operations and critical words, such as “+”, “-”, “\*”, “=”, “==”, “{”, “}”, “(”, “)”, “;”, “,”, “.”, “&”, “|”, “<”, “>”, “<=”, “>=”, “boolean”, “int”, “true”, “false”, “scan”, “float”, “print”, “else”, “while”, etc. Then the compiler will parse them into an AST, and finally translate AST into our ISA.

## Simulator

We will substantially use Modelsim for module testbench simulations. We plan to make the top level configurable through a one bit parameter, SIMMODE. When SIMMODE is set, PLL will be disabled, so that we can potentially run top level simulations in Modelsim as well. This worked for BMP capability exploration, but we are unsure if we can do this again for the project.

## Application

Our application is to use the CPU on FPGA to apply a machine learning classifier on an image feed from the camera. It will be able to classify a single hand-written character(or multiple characters or image if we want to up scope). The classified result will be shown on the screen through UART. The software will demonstrate our design in two perspectives. First, it will use the floating point hardware on the CPU to perform the matrix multiplication. This hardware support will increase the precision of the classification result. Second, if we implement the superscalar and carefully write the assembly to reduce dependency, then we would expect a significant performance boost because multiple instructions can be executed in parallel.

# Division of Labor

|  |  |  |  |
| --- | --- | --- | --- |
| **High-level Image Classification Algorithm Training** | Qikun Liu | **Out of order processing - add reorder buffer** | Haining & Justin |
| **Camera Interface and Demo Assembly Code** | Qikun Liu & Haining Qiu | **Out of order processing - add scheduler** | Harry Zhao |
| **Integrate classifier weights into top level** | Qikun Liu | **Out of order processing - verification** | Everyone |
| **Assembler for New ISA** | Harry Zhao | **Super Scalar Top level plan** | Harry Zhao |
| **Floating-Point Adder (and subtractor)** | Haining Qiu | **Super scalar - stall logic** | Haining Qiu |
| **Floating-Point (and integer) Multipliers** | Justin Qiao | **Super scalar - Compute unit integration** | Justin Qiao |
| **Int-FP and FP-Int Converters** | Justin Qiao | **Super scalar - wide fetch, wide decode** | Harry Zhao |
| **Processor Expansions for 32-bit ISA** | Harry Zhao | **Super scalar - memory modification** | Qikun Liu |
| **Unit Tests and Top-level Testbench Design** | Everyone | **Synthesis - Memory Management and Timing** | Harry Zhao |
| **Compiler to translate Java into asm code** | Qikun Liu | **Test Coverage** | Justin Qiao |
| **Generating Java code for the application** | Qikun Liu |  |  |