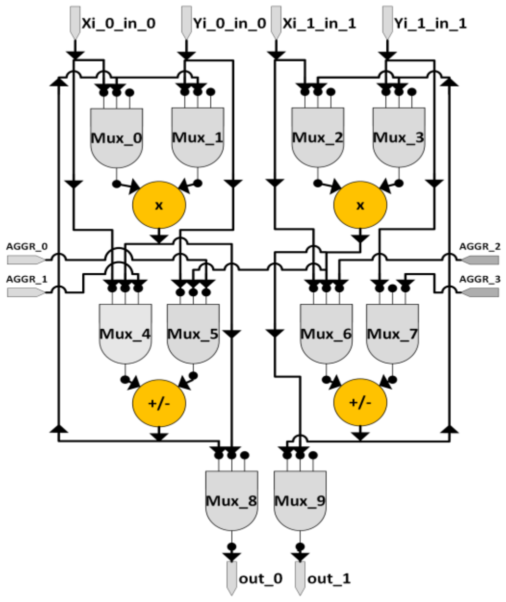
1.1.1.1.1 PE microarchitecture

The PE microarchitecture is illustrated in **Figure 10**. Each PE supports 4 (STC) types of operation. These are Euclidean distance calculation (optype=1), Manhattan distance calculation (optype=2), vector dot product (optype=3) and weighted vector pooling (optype=4). The PE is completely feedforward and does not support any aggregation. The datapath is configurable and can be programmed to support the optypes mentioned above. Each PE has 8 bytes of input from the DDR and 8 bytes from the values stored in XNU local memory. This value can be either a query vector (denoted as Y in case of similarity search) or a scalar weight (denoted as W in the case of recommendation, graph neural nets). The total size of input from DDR in each cycle can be a maximum of 128B (64B each from two channels). Hence a total of 16 PEs is required. The arithmetic datapath in each PE is FP32, but can be easily changed to support simpler arithmetic such as int16 or fp16. The PE is fully pipelined with multiple cycles of latency. The adders and multipliers might require retiming for meeting a specific target frequency. Aggregation logic following the PEs is WIP.



Design :

PE design is simple and is using following modules :

- 4 Data Inputs , 10 Multiplexers (**Mux_X**), 2 Adder/Subtractor modules , 2 Multiplier modules , 4 Aggregation Inputs(**AGGR_X**) and 2 Data Outputs (**out_X**).

All inputs and outputs are configured for 32 bits and data format can be int16/int32/fp32.

Aggregation inputs from each PE module can be configured/connected together with internal Mux modules (**Mux_4, Mux_5 , Mux_6 , Mux_7**) to forward output data from PE to internal Adder/Subtractor modules.

All Mux modules are using a Sel pins (**m_X_sel**) to select what Input (from 3 options) will be used to forward data to output port. (not in PE diagram)

All Add/Sub modules are using Operation pins (**addsub_X_op**) to select addition/subtraction operations. (not in PE diagram)

Implementation and Validation :

PE module was implemented in Scala programming language using Chisel3 library and FIRRTL framework.

Chisel3 is a hardware design language that facilitates advanced circuit generation and design reuse for both ASIC/FPGA digital logic designs and add hardware construction primitives to the Scala programming language. (https://github.com/chipsalliance/chisel3)

Chisel3 is powered by **FIRRTL** (Flexible Intermediate Representation for RTL), a hardware compiler framework that performs optimizations of Chisel-generated circuits and supports custom user-defined circuit transformations. (https://github.com/chipsalliance/firrtl)

Each module from PE project was implemented individualy in Scala and for floating point operations was used Hardfloat library.

Hardfloat project contains hardware floating-point units written in Chisel and conversions between floating-point conversions with different precision. (https://github.com/ucb-bar/berkeley-hardfloat)

Validation tests for all PE modules was implemented in Scala using **chisel-testers** library (https://github.com/freechipsproject/chisel-testers) and random generated values for all supported operations was done in python using **Simfloat** library (https://github.com/robclewley/ieee754_simulation)

All commands to build, run tests or to clean components from PE project are implemented in a **Makefile**.

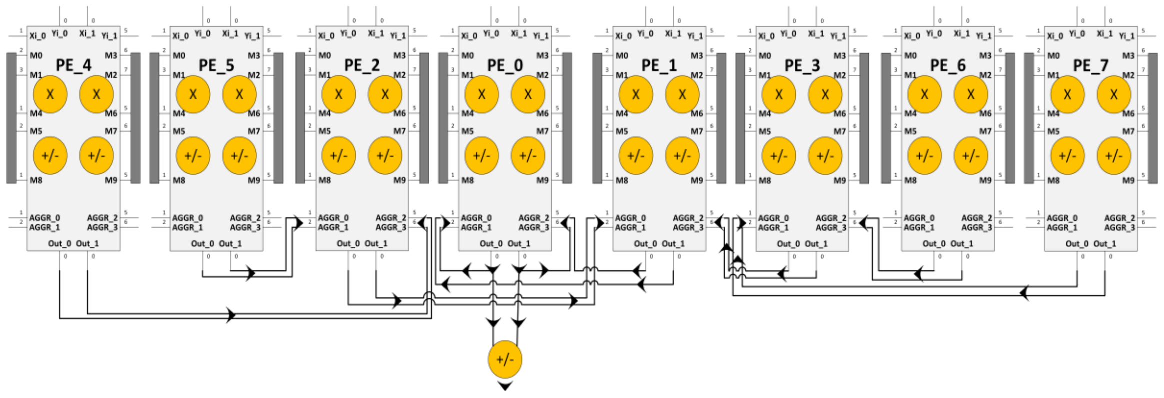
Final project was uploaded to : [**git link**]

Requirements :

One requirement for this project was to support total size of input from DDR : 128 bits.

Solution for this case was to implement 2 modules with 8 PEs (64 bits input each) to cover inputs from dual channels.

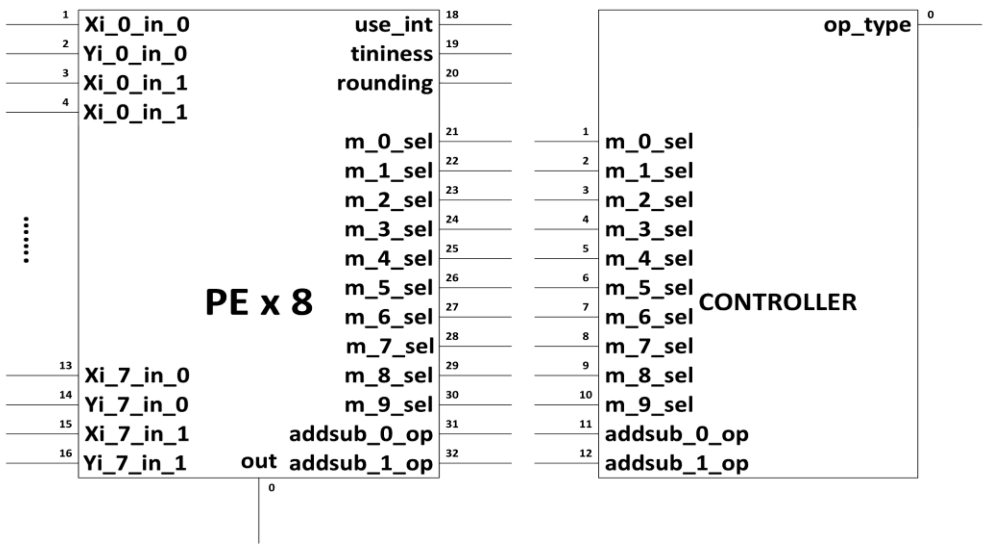
Following diagram is covering design of 8 PE modules and design logic for aggregation.



Aggregation logic is based on a tree arhictecture where one central module (PE_0) is collecting output data from above modules (directly connected). AGGR ports are used to loop output data from PE_0 directly to internal Adder modules. Rest of PE modules are identified as secondary tree nodes and are forwarding both outputs (out_0 and out_1) to the directly connected nodes.

Another requirement for all PE modules was to implement a logic to switch from int16/int32 to fp32 and this option is available connecting pin use_int.

Final design (8 PE modules configured in one PEx8 module & controller) :



➢ Operation type input (**op_type**) :

Operation type	**Bits (2)**
Euclidean (L2)	0b00
Manhattan (L1)	0b01
Dot product	0b10
Scalar-Vector product	0b11

➢ Option to use INT input (**use_int**):

Operation name	**Bits (1)**
True – inputs/outputs are configured for INT (32 bits)	0b1
False – inputs/outputs are configured for FP (32 bits)	0b0

➢ Option to select tininess for output data (**tininess**):

In the terminology of the IEEE Standard, HardFloat can detect tininess for underflow either before or after rounding.

Operation name / status	**Bits (1)**
tininess is detected before rounding	0b0
tininess is detected after rounding	0b1

- **Detecting tininess after rounding is usually slightly better because it results in fewer spurious underflow signals**. The option for detecting tininess before rounding is provided for compatibility with some systems.

➢ Option to select different rounding types (**rounding**):

Operation name / status	**Bits (3)**
round_near_even - round to nearest, with ties to even	0b000
round_minMag - round to minimum magnitude (toward zero)	0b001
round_min - round to minimum (down)	0b010
round_max - round to maximum (up)	0b011
round_near_maxMag - round to nearest, with ties to maximum magnitude (away from zero)	0b100
round_odd - round to odd (jamming)	0b110

- **Rounding mode round_near_maxMag is usually better for most of the cases and was used for testing all PE modules.**

<u>ONLY for INT inputs</u> : rounding and tininess are NOT used by internal logic (Add/Sub/Mul)

➢ Values for selectors on Mux modules (**m_X_sel**):

Operation name / status	**Bits (2)**
in_0 – input 0	0b00
in_1 – input 1	0b01
in_2 – input 2	0b10
in_3 – input 3	0b11

- Values for operation type on Adder/Subtractor modules (**addsub_0_op / addsub_1_op**):

Operation name / status	Bits (1)
False – for addition operation	0b0
True – for subtraction operation	0b1

- Input ports (**Xi_0_in_0, Yi_0_in_0**, … , **Xi_7_in_1, Yi_7_in_1**) :

Are total of 16 inputs for 8 PE modules. All inputs are configured for 32 bits.

- Ouput port (**out**) :

Output is configured for 32 bits.

- **Validation cycles for PEx8 module :**

Startup cycle is used only once to init all modules from PE and then the base logic is started : Init -> Operation Logic -> Aggregation Loic -> Addition -> Output -> Init

Operation Type	Operation Name	Startup	Init	Operation Logic	Aggregation Logic	Addition	Output
00	L2	1	1	15	56	2	1
01	L1	1	1	10	53	2	1
10	DOT prod	1	1	10	53	2	1
11	S-V prod	1	1	15	56	2	1

Following section will present all 4 operations supported by PE x 8 module.

Note : dbg_fsm is used for debugging to check the cycles order

Debug Step	Debug ID
Startup	9
Init	1

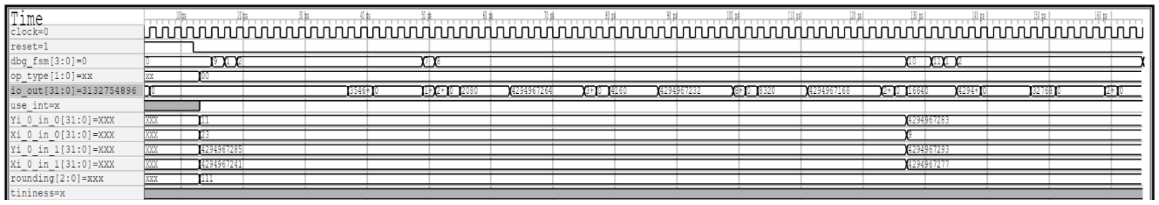
L2	2
L1	3
Dot product	4
Scalar-Vector product	5
Final addition step	10
Output	11

ONLY for INT inputs : rounding and tininess are NOT used by internal logic (Add/Sub/Mul)

- **Set 1 of inputs** : Xi_0 (23) / Yi_0 (11) / Xi_1 (-55) / Yi_1 (-11)
- **Set 2 of inputs** : Xi_0 (9) / Yi_0 (-13) / Xi_1 (-19) / Yi_1 (--3)
- **ALL PE modules** have the same set of inputs for following tests.

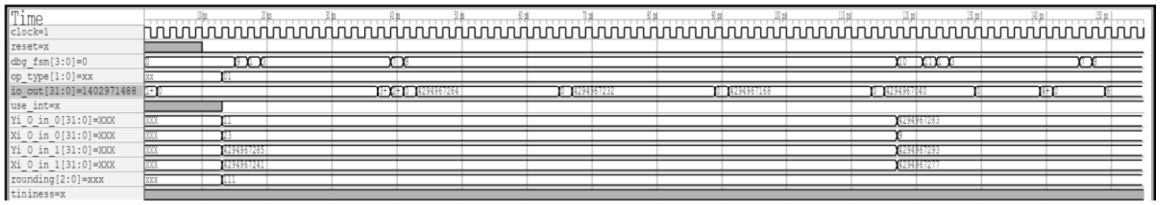
Operation type	**Bits (2)**
Euclidean (L2)	0b00

Test for L2 is using 2 sets of input data and INT values are represented in unsigned INT.



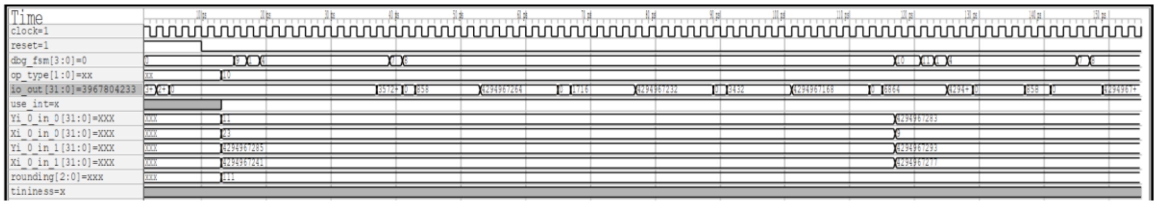
Operation type	**Bits (2)**
Manhattan (L1)	0b01

Test for L1 is using 2 sets of input data and INT values are represented in unsigned INT.



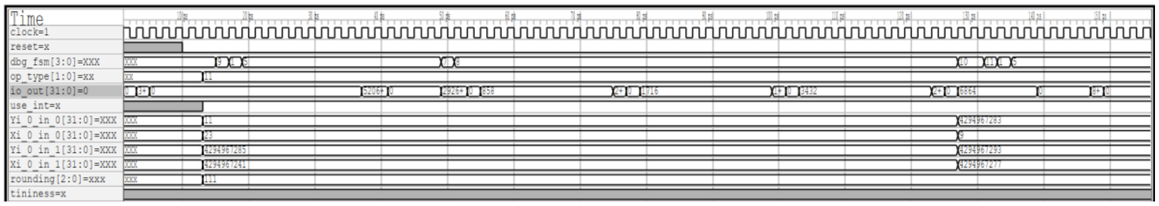
Operation type	Bits (2)
Dot product	0b10

Test for DOT prod is using 2 sets of input data and INT values are represented in unsigned INT.



Operation type	Bits (2)
Scalar-Vector product	0b11

Test for S-V prod is using 2 sets of input data and INT values are represented in unsigned INT.


Time
clock=1
reset=x
dbg_fsm[3:0]=XXX
op_type[1:0]=xx
io_out[31:0]=0
use_int=x
Yi_0_in_0[31:0]=XXX
Xi_0_in_0[31:0]=XXX
Yi_0_in_1[31:0]=XXX
Xi_0_in_1[31:0]=XXX
rounding[2:0]=xxx
tininess=x