K means accelerator

# Introduction

## Abbreviations

|  |  |
| --- | --- |
| Test Bench | TB |
| Register File | RF |
| Classification block | CB |
| New means calculation block | NMB |
| Convergence check block | CCB |

## The K means algorithm

The K means algorithm is an iterative algorithm which divides a given data vector to K different clusters (K is a natural number). Each cluster will be characterized by its “center of mass”, what will be referred in this paper as centroid.

### The algorithm steps

For a simpler explanation, it can be assumed that K is a constant predefined natural value. First, some symbols need o be defined:

-the cluster number "*i*" centroid

– the group of points in cluster number "*i*"

Upper index “*t*” – iteration or time

#### Initialization step

The first step in the algorithm is to randomly choose centroids for the K clusters. The “time” (“*t*”) for the initialization step will be defined as zero.

#### Classification step

In each iteration(time) of the algorithm, first each point of the input data is assigned to a cluster based on the “distance” from the point to the clusters centroid. A point will be assigned to cluster number “i” if the metrical distance between it and the cluster’s centroid is the minimum between the distances from the point to all others cluster’s centroids. To simplify:

\*In case of the distance from two different clusters is the same and is the minimum found, the chosen cluster is the one with the lowest index.

#### Centroids update step

After the classification step, the centroids of each cluster are updated to be mean of all points which belong to it in end of iteration(time) *t*. This is done by verifying if a cluster is empty(in this case the centroid is not changed) and then calculating the mean of all the clusters points:

#### Convergence check step

If the centroids of the next iteration calculated in the step above are close enough to the current centroids, then the algorithm comes to an end. Else, the iteration number(time) is increased by one and a new iteration begins with the assigning step.

##### Algorithm convergence

The k means algorithm assures convergence to a local minimum, i.e. the final centroids values are so that the variance within the clusters is minimized while the intra cluster’s variance is maximized. This minimum variance within the cluster is not always the global minimum that can be reached, the local minimum which was reached by the algorithm depends on the initialization step, specifically on the first values of the centroids.

#### Choosing K

Usually the optimal K is not known before the beginning of the algorithm. Therefore, a an error parameter can be defined to help choosing K. The most commonly known error parameter is the clustering error which is defined by:

In this formula, the elements are:

As K increases, the error decreases. For example, if K is as the number of pints in the input vector, the error will be zero. This because it cluster will have just one point which will also be its centroid, but in this case no new information was added by the algorithm.

One suggested method of choosing a natural K so the clustering error is minimized is by gradually increasing K and calculating for each increasement. The process ends when the error reaches a value so that , where is a predefined threshold.

## AMBA APB

### Introduction

The Advanced Peripheral Bus (APB) is part of the Advanced Microprocessor Bus Architecture (AMBA) protocol family. This protocol is a single master multi slave and set guidelines for transactions between the master and its low-bandwidth peripherals, the slaves. The APB protocol signal transactions are only related to the rising edge of the clock and every transaction takes at least two cycles. It can be used to provide access to the programmable control registers of peripheral devices. Furthermore, the APB is a low-cost interface that is optimal for minimal power consumption.

The figure bellow (Key to timing diagram conventions) explains the timing diagrams in the following sections. Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.



Figure 1: Key to timing diagram of APB protocol

The signals which are part of APB protocol are listed and described in the table below:

|  |  |  |
| --- | --- | --- |
| Signal | Source | Description |
| PCLK | Clock source | Clock. The rising edge of PCLK times all transfers on the APB. |
| PRESETn | System bus equivalent | Reset. The APB reset signal is active LOW. This signal is normally connected  directly to the system bus reset signal. |
| PADDR | Master | Address. This is the APB address bus. It can be up to 32 bits wide and is driven  by the peripheral bus bridge unit. |
| PSELx | Master | Select. The APB bridge unit generates this signal to each peripheral bus slave.  It indicates that the slave device is selected and that a data transfer is required.  There is a PSELx signal for each slave. |
| PENABLE | Master | Enable. This signal indicates the second and subsequent cycles of an APB  transfer. |
| PWRITE | Master | Direction. This signal indicates an APB write access when HIGH and an APB  read access when LOW. |
| PWDATA | Master | Write data. This bus is driven by the peripheral bus bridge unit during write  cycles when PWRITE is HIGH. This bus can be up to 32 bits wide. |
| PREADY | Slave | Ready. The slave uses this signal to extend an APB transfer. |
| PRDATA | Slave | Read Data. The selected slave drives this bus during read cycles when  PWRITE is LOW. This bus can be up to 32-bits wide. |
| PSLVERR | Slave | This signal indicates a transfer failure. APB peripherals are not required to  support the PSLVERR pin. This is true for both existing and new APB  peripheral designs. Where a peripheral does not include this pin then the  appropriate input to the APB bridge is tied LOW. |

Table 1: APB signal description

The PADDR,PWRITE,PWDATA signals are common among all the slaves, however there are as many PSEL signals as slaves, and for each slave one PRDATA from it to the master. The following shows the block diagram between master and slave of APB:



Figure 2: APB block diagram

#### Operating states

The figure bellow describes the operating states of the protocol:



Figure 3: APB operating states

The state machine operates through the following states:

**IDLE** - This is the default state of the APB.

**SETUP** - When a transfer is required the bus moves into the SETUP state, where the appropriate select signal, PSELx, is asserted. The bus only remains in the SETUP state for one clock cycle and always moves to the ACCESS state on the next rising edge of the clock.

**ACCESS** - The enable signal, PENABLE, is asserted in the ACCESS state. The

address, write, select, and write data signals must remain stable during

the transition from the SETUP to ACCESS state. Exit from the ACCESS state is controlled by the PREADY signal from the slave:

• If PREADY is held LOW by the slave then the peripheral bus remains in the ACCESS state.

• If PREADY is driven HIGH by the slave then the ACCESS state is exited and the bus returns to the IDLE state if no more transfers are required. Alternatively, the bus moves directly to the SETUP state if another transfer follows.

### Transfers

Each transfer consists of two cycles: one for the SETUP state and another for the ACCESS state. There are three types of transfers: write transfers, read transfers and error response transfers. In addition, write and read transfers can be with or without wait states, that are SETUP states which follow an ACCESS state instead of going to IDLE STATE.

#### Write Transfers

##### Write Transfers without wait states

A write transfer without wait states consist of two clock cycles: in the first (the SETUP STATE) the signals: address (PADDR), write data (PWDATA), write (PWRITE) and select (PSEL) are asserted. PADDR is asserted to the desired address where the data is supposed to be written, PWDATA is asserted to the desired data to be written, PWRITE is asserted HIGH and PSEL is asserted HIGH only for the specific slave which the write command is for, the rest of the PSEL lines are driven LOW. These signals remain unchanged through the second cycle.

In the second cycle (the ACCESS state) the slave sets the enable signal (PENABLE) HIGH. The ready signal (PREADY) is set HIGH by the slave in order the informed the master that the slave is ready to receive the data, which is latched by the slave in the rising edge ending the second clock cycle. After this last clock rising edge, PREADY is driven LOW by the slave, PENABLE is driven LOW by the master, and PSEL is driven LOW by the master (unless the transfer is to be followed immediately by another transfer to the same slave, in which case the signals PENABLE and PSEL remain as they are) meaning that the transfer is over.

In Figure 4 there is an example of write transaction with no wait states can be seen, with the first cycle of the transfer being from T1 to T2 and the second cycle from T2 to T3.



Figure 4: APB write transfer with no waits

##### Write Transfers with wait states

The first cycle of the transfers is the as the transfers without wait states. During the ACCESS state, when PENABLE is HIGH, the transfer can be extended by driving the PREADY LOW. The signals PADDR, PWRITE, PSEL, PENABLE and PDATA remain unchanged from the end of the first cycle (SETUP state) until the data is latched by the slave, which occurs at the first rising clock edge after the slave sets the PREADY signal HIGH. After this clock rising edge, PREADY is driven LOW by the slave, PENABLE is driven LOW by the master, and PSEL is driven LOW by the master meaning that the transfer is over (unless the transfer is to be followed immediately by another transfer to the same slave, in which case the signals PENABLE and PSEL remain as they are) .

In Figure 4: APB write transfer with no waits Figure 5 an example of write transaction with wait states can be seen, with the first cycle of the transfer being from T1 to T2,two wait states occur from T2 until T4 and the last cycle of the transfer from T4 to T5 , in which the slave sets the PREADY signal HIGH and at the end of this cycle the data is latched by the slave.



Figure 5: APB write transfer with wait states.

#### Read Transfers

##### Read Transfers without wait states

A read transfer without wait states consist of two clock cycles: in the first (the SETUP STATE) the signals: address (PADDR), write (PWRITE) and select (PSEL) are asserted. PADDR is asserted to the desired address where the data is supposed to be read, PWRITE is asserted LOW and PSEL is asserted HIGH only for the specific slave which the write command is for, the rest of the PSEL lines are driven LOW. These signals remain unchanged through the second cycle.

In the second cycle (the ACCESS state) the slave sets the enable signal (PENABLE) HIGH. The PRDATA signal is set by the slave according to the data in stored in the desired address(the address which is set in PADDR signal) and the ready signal (PREADY) is set HIGH by the slave in order the informed the master that the slave is ready to send the data. The data in PRDATA signal is latched by the master in the rising edge ending the second clock cycle. After this last clock rising edge, PREADY is driven LOW by the slave, PENABLE is driven LOW by the master, and PSEL is driven LOW by the master (unless the transfer is to be followed immediately by another transfer to the same slave, in which case the signals PENABLE and PSEL remain as they are) meaning that the transfer is over.

In Figure 6 an example of write transaction with no wait states can be seen, with the first cycle of the transfer being from T1 to T2 and the second cycle from T2 to T3.



Figure 6: APB read transfers with no wait states

##### Read Transfers with wait states

The first cycle of the transfers is the as the first cycle of transfer without wait states. During the ACCESS state, when PENABLE is HIGH, the transfer can be extended by driving the PREADY LOW. The signals PADDR, PWRITE, PSEL and PENABLE remain unchanged from the end of the first cycle (SETUP state) until the data is latched by the master, which occurs at the first rising clock edge after the slave sets the PREADY signal HIGH. After this clock rising edge, PREADY is driven LOW by the slave, PENABLE is driven LOW by the master, and PSEL is driven LOW by the master meaning that the transfer is over (unless the transfer is to be followed immediately by another transfer to the same slave, in which case the signals PENABLE and PSEL remain as they are) .

In Figure 7 an example of read transaction with wait states can be seen, with the first cycle of the transfer being from T1 to T2,two wait states occur from T2 until T4 and the last cycle of the transfer from T4 to T5 , in which the slave sets the PREADY signal HIGH and at the end of this cycle the data is latched by the master.



Figure 7:APB read transfer with wait states.

#### Error response

Some APB peripheral offer a way of indicating that an error occurred during a transfer with the PSLVERR signal. Errors can occur both in read and write transfers, and the signal PSLVERR is only considered valid during the last cycle of an APB transfer, when PSEL, PENABLE, and PREADY are all HIGH.

It is recommended, but not mandatory, that you drive PSLVERR LOW when it is not

being sampled. That is, when any of PSEL, PENABLE, or PREADY are LOW.

Transactions that receive an error, might or might not have changed the state of the

slave. This is peripheral-specific, and either is acceptable.

When a write transaction receives an error, this does not mean that the register within the slave has not been updated. Read transactions that receive an error can return invalid data.

There is no requirement for the slave to drive the data bus to all 0s for a read error.

APB slaves are not required to support the PSLVERR pin. This is true for both

existing and new APB peripheral designs. Where a slave does not include this pin

then the appropriate input to the master is tied LOW.

##### Error response in a write transfer

When there is an error in a write transfer and the slave in the transfer has an active PSLVERR signal, during the last cycle of the transfer (when PSEL, PENABLE, and PREADY are all HIGH) PSLVERR is driven HIGH, informing the master about the error in the transaction. These can be seen in Figure 8:



Figure 8: APB error in write transfer

##### Error response in a write transfer

When there is an error in a read transfer and the slave in the transfer has an active PSLVERR signal, during the last cycle of the transfer (when PSEL, PENABLE, and PREADY are all HIGH) PSLVERR is driven HIGH, informing the master about the error in the transaction. These can be seen in Figure 9:



Figure 9 : APB read in write transfer

## Paper summary

This section is a summary of the paper which is the base for this project, “*FPGA Implementation of K-means Algorithm for Bioinformatics Application: An Accelerated Approach to Clustering Microarray Data*” by Hanaa M. Hussain, Khaled Benkrid, Huseyin Seker, Ahmet T. Erdogan.

The motivation of the paper is the acceleration of the K means algorithm in order to process Microarrays which is a technique used in genome experiments to measure expression level of many thousands of genes simultaneously.

### K means clustering distance computation

According to the paper, distance computation is the most computationally demanding part, and where most of the K-means processing time occurs. Therefore, one aspect for improving the implementation of the algorithm is by accelerating the distance computation.

One of the widely used distance metrics incorporated with K-means clustering is the Euclidean metric and it is easy to implement. The Euclidean distance metric from point *x* to centroid (both with “n” dimensions) in iteration “*t*” is given by:

However, it also consumes a lot of computational resources when implemented in hardware due to the multiplication operation used for obtaining the square operation.

Thus, the paper presents an alternative distance metric called the Manhattan distance to be used for the classification step of the algorithm. The Manhattan metric from point *x* to centroid (both with “n” dimensions) in iteration “*t*” is given by:

The Manhattan metric according to the paper performed faster than the Euclidean metric, because it does not require calculating the square, offering better exploitation of parallelism and speed twice than that obtained by Euclidean distance. However, the accuracy of this distance measure was found to be slightly inferior to the Euclidean metric, but results were still within an acceptable error.

### K means past implementations and improvements

The paper shows few different implementation methods and their advantages/disadvantages regarding different aspect of performance, elaborate later.

* Implementing a hardware unit on FPGA board which calculate parallelly the distance(by using Manhattan metric) of each point from the input data to all cluster’s centroids. The input data was stored in a host, brought to the computational unit for the distance calculation and the result were sent back to the host for new means calculations.

Advantages:

This implementation allowed the input data to be at any size since the storage of the data was a responsibility of the host. Moreover, this implementation achieved a speed-up of 15x.

Disadvantages:

The communication overhead between the host and the FPGA board.

* Storing the data in a SRAM memory unit used exclusively for the distance calculation unit. Familiar to the previous implementation, the distance calculation is done in hardware by three-FPGA’s, and using Manhattan metric for distance calculation.

Advantages:

Speedup of 50x of more than the 500MHz Pentium III host processor, in part because the data retrieving from the host is done only one time during the algorithm, only in the beginning.

Truncating of bit width of input data helped the design to be faster.

Disadvantages:

The paper does not suggest disadvantages, but there is a clear limit of the data sets size due to the memory unit storage capacity.

* Implementation of a hybrid fixed and floating point arithmetic units for the calculations required during the algorithm run in hardware.

Advantages:

Data transfer throughput increased.

Disadvantages:

Larger FPGA area needed for the design implementation.

* Fully implementing the algorithm steps in hardware(except the initialization step which is done in a host).The distance was calculated by Manhattan metric.

Advantages:

Speedup of 500x over matlab implantation including I/O overhead, using 3 clusters.

Disadvantages:

Lack of memory capability restricted the size of data processing can be done at one time.

A suggested disadvantage is the fact that it was tested only for a run of the algorithm with 3 clusters, which can be to few clusters to receive significant information about the data after running the algorithm. There is a lack of proof that this implementation is efficient for problems which need more than 3 clusters.

* Fully implementing the algorithm steps in hardware (except the initialization step which is done in a host). The distance was calculated by Euclidean metric.

Advantages:

Speed-up of 2x over software implementation even though the former was running at 12.5 times lower frequency than the latter. Better accuracy duo to the use of Euclidean metric.

* Fully implementing the algorithm steps in hardware(on FPGA board). In addition, utilizing a floating point divider to calculate the new means in hardware level. This approach required the use of an extra block to convert the fixed-point data to floating point, and then after the division was done, another floating to fixed-point converter was needed.

Advantages:

The host is free while the FPGA is running the algorithm.

Disadvantages:

No speed up was achieved.

In conclusion, all the mention implementations were done at least ten years ago, so it is to be anticipated better timing performance in the same designs duo to the fact that today’s resources outnumber those used in some of the above mentioned implementations.

### The paper’s implementation proposal

The main implementation described in the paper is implementing all algorithm steps on hardware level. First, creating a module capable of running the algorithm with 8 clusters and then using the rest available area of the FPGA to duplicate this module as many times as possible in order the increase the parallelism.

One important feature of the implementation described in the paper is the use of fixed point instead of floating point. As a result of the division operation used in the algorithm in order to calculate the centroids every iteration, there is a need for representation of real numbers. Even though the most commonly used representation of real number is the floating point in software level, duo to its cost and complexity in this implementation (and in most FPGA designs) real numbers are represented by fixed point.

#### Preliminary analysis for Hardware implementation

In order to set the design requirements, the paper presents some analysis. By analyzing the future data inputs determining the common data size, dynamic range, precision and memory capacity.

In fixed point representation, the word length of the real numbers is constant and must be determined prior to the implementation. Therefore, there is need to analyze the number of bits required to represent both integer and fractional part of real numbers.

After the dynamic range and precision of the input data is determined, the word length needs to be decided. This is done in the paper by calculating the minimum number of bits required to represent any word which could be a part of the input data. The integer part of the word should have as many bits as:

The fractional part of the world should have as many bits as:

This calculation should be done for the word length of the input data, as well as the word length of the distances and the accumulators.

Moreover, there is a need the determined the memory capacitance needed input data, centroids and results. The ideal situation would be is to have enough Block Rams to store all datasets and avoid memory access bottleneck. However, this is not always the case especially when using huge datasets, thus streaming data from an external memory is a possible option.

#### Hardware architecture and design

The architecture and design of the algorithm implementation is as in the figure below:



Figure 10 : The paper's implementation design divided by blocks

As it can be seen, the design is based on three blocks. The first block is used to calculate distances. The second block consists of assigning points to one of the eight clusters based on the results of the previous block. The third block is a sequential divider which calculates the new means in hardware.

##### Minimum distance finder block

The input data is stored in Block RAMs within the FPGA, and the initial centroids are stored in registers within the FPGA. The first block initially calculates distances between each data point and the cluster’s centroids. In the paper’s design eight clusters are used, therefore it has eight distance calculating processing elements working in parallel. The first block reads one data point from the on chip Block RAMs every clock cycle, obtains the eight distances simultaneously, and then obtains the absolute values of these distances. Secondly, these eight absolute distances run through a comparator tree (as shown in Figure 10 : The paper's implementation designFigure 10 )to obtain the minimum distance for each data point and its index. This takes two clock cycles for it to complete. This whole process is fully pipelined to have a throughput of one result every clock cycle, but it has a latency of four clock cycles: one for reading the input data point and obtaining the distances, one for obtaining the absolute of these distances, two for the comparison tree to obtain the minimum distance and its index.

##### Accumulation block

The second block assigns the points to one of the eight cluster based on the results of the previous block (the point itself and the index of the cluster it should be assigned to) and counting the number of points in every cluster. The outputs of this block are eight accumulators and eight counters. This block has a throughput of one result per clock cycle with latency of one clock cycle. However, the final results will be passed to the next block when accumulators finish assigning all points to the accumulator.

##### Divider block

The third block calculates the new means in hardware. The latency of the

divider is 60 clock cycles and the throughput is one result per clock cycle. And since we are calculating eight means sequentially, the total time in which the divider will be active is 68 clock cycles. The divider itself was obtained using the core generator tool available with Xilinx ISE Design Suite 12.2. The divider block is activated by the previous block that is when the second block finishes assigning all points to clusters.

.

##### Convergence

The process iterates until an end condition is reached and tested for inside the controller, which we decided to be reaching a point where previous centroids do not vary from newly calculated ones, with an acceptable 1% error.

#### Implementation results

Simulation results show that it takes 2971 clock cycles, to cluster 2905 points (415 × 7) assuming that data are already written to Block RAMs. The algorithm converged after 25 iterations, thus taking a total of 74275 clock cycles. This result does not take into consideration the time needed to write data to the FPGA Block RAMs, which is also 2905 clock cycles. However, the result does include the time to write results to the FPGA Block Rams.

In the paper a comparison between the implementation in hardware and in software was done, not considering the time needed to write data to the FPGA block RAMs . The software implementation was done with Matlab and it took an average execution time of the model for 1000 runs of the algorithm was 0.0062 ± 1.22e-4 s, with minimum execution time of 0.0060 s and maximum execution time of 0.0072 s. These results are based on initial centroids being pre defined and given as an input to the algorithm. The hardware implementation on the other hand converged after 25 iterations, thus taking a total of 74275 clock cycles. This result does not take into consideration the time needed to write data to the FPGA Block RAMs, which is also 2905 clock cycles. However, the result does include the time to write results to the FPGA Block Rams.

### Hardware Synthesis Results

The described above design was implemented on a FPGA Xilinx XC4VLX25-10SF363 using just a single core, and a maximum clock frequency of 126 MHz was achieved. This single core occupies 2.208 slices, which is only 20% of the FPGA floor area. As a result of the large available area in the FPGA after the implementation and n order to improve results the paper authors replicated the whole design five times before running out of floor area. This approach can accelerate the run time of the algorithm by five times and provide a server solution for processing multiple datasets simultaneously. In the paper, this approach was implemented using the same datasets and obtained a maximum frequency of 124 MHz, and consumed 99 % of the FPGA floor area.

For the single core design, simulation results showed that it takes about 2971 clock cycles to complete one full iteration, and the datasets required 25 iterations to converge, thus hardware execution time is just 589 μs, given that the clock frequency is 126 MHz

The implemented hardware achieved high timing performance, with speed-up of 10.3x for the case when implementing the single core, and 51.7x for the case when implementing the five cores approach.

### Paper’s conclusion

The paper in discussion presented FPGA hardware design of the K-means algorithm. Due to applying concepts of pipelining, parallelism, and multicore processing, results show that for the input data used in the paper (Microarray data) there was a speed up potential on implementing the K means algorithm on hardware level.

# Architecture

## Block Diagram

The high-level architecture proposed by this project is as shown in Figure 11. It is essentially composed of two main modules: the “Register file” and the “K means core”.

The “Register file” interfaces with the CPU host by APB protocol, as APB slave. Besides that, it also stores important data at local registers and interfaces with the second module “K means core”, allowing to read and write to its internal registers.

The “K means core” module is the actual “brain” of the architecture. It is responsible by running the algorithm and when it is done, it throws an interrupt to the CPU host, indicating the algorithm has come to an end.

The data set with which the algorithm is done is stored in a local RAM inside the “K means core module”. In order to do so, every data point is store one by one in this RAM by a process called “Indirect Access”.

The “Indirect Access” process is as its sounds: the CPU can write to the “K means core” local RAM only though a mediator, in this case, through the “Register File”. For more about this, see section ‎2.6.1.1 .



Figure 11: K means TOP block diagram

## Block Description

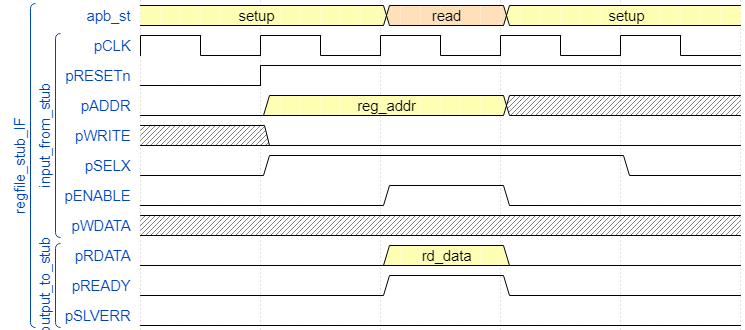
### Pipeline diagrams

TBD

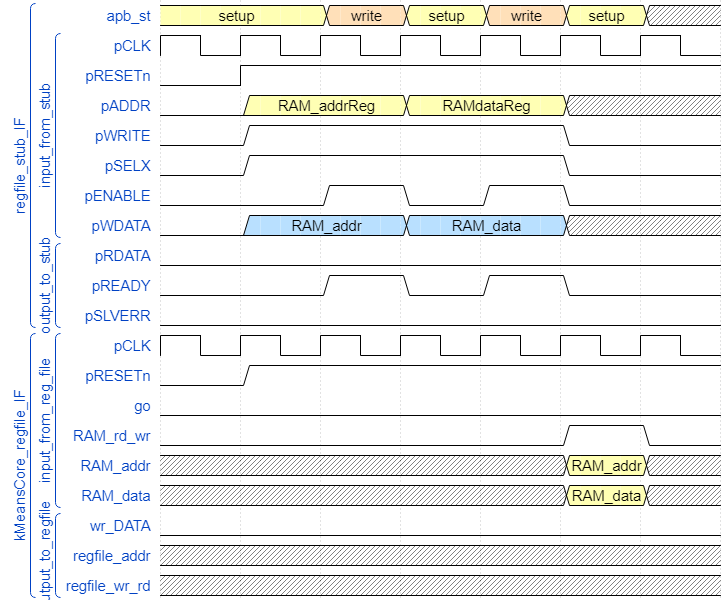
### Timing diagrams

Below can be seen timing diagram regarding "K means TOP" - STUB(out ports) interface:

* Indirect-Access Read is related to reading a register's value from "Register File" Block, timing diagram can be seen in figure TBD:



* Indirect-Access Write is done by writing a value to a register in "Register File" Block, which is then followed by writing to RAM, if there was 2 followed indirect writing's to RAM address register & RAM data register accordingly.  
  timing diagram can be seen in figure TBD:



## Pins Description

TBD

The K-means Top modules inputs and outputs are described below (an extended explanation can be seen at chapter 2.5 – interface description):

Inputs:

* PClk – clock input.
* PResetn – reset asyncronic signal(described in 2.4)
* Below inputs are part of APB interface: (as described in 1.3 – AMBA APB(hyperlink this please) PADDR,PSEL,PENABLE,PWRITE,PWDATA.

Outputs:

* Below outputs are part of APB interface: (as described in 1.3 – AMBA APB(hyperlink this please) PREADY,PRDATA,PSLVERR.
* Interupt – a flag to annonce the calculation process finished.

## Clocks and Resets

Clocks:

* There is a single domain – starting PClk signal as input to K means TOP, this clk permeates both register file and K means core, no other clk domains are implemented.
* Note: the RAM inside the K means core work with Negedge of the same clk domain explained.

Resets:

* Reset asynchronous signal – named Presetn, reset the calculation process, required between calculation requests.
* Wait for schahar response to fill this part more if needed

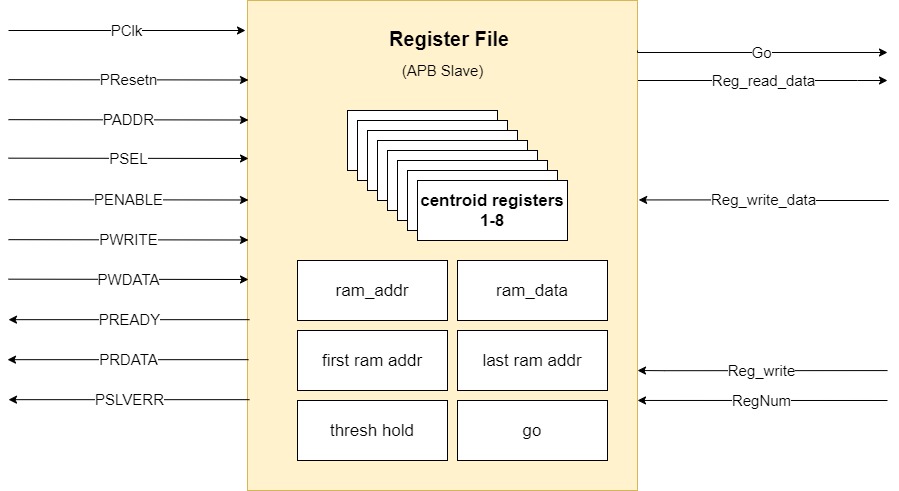
## Interfaces Description

Other than clks/resets, the APB interface signals are connected to Register File block, which explained extensively at chapter 2.6.1 – Register File, for extension of how they are driving the Register File, which is the APB slave, further read regarding APB interface signals offered at chapter 1.3 – amba apb.

## Sub-units Description

### Register File

Proposed architecture for Register File includes several registers as show in figure bla bla, each register has an enumerated value representing it, which shown in table blab la below.



TBD – REMOVE INTERNAL STAUTS REG AND THEN FILL ENUM VALUES:

|  |  |  |
| --- | --- | --- |
| Register name | Register enum | Register description |
| Ram data |  | Core's ram data (explained in chapter 2.6.1.1 indirect access) |
| Ram addr |  | Core's ram addr (explained in chapter 2.6.1.1 indirect access) |
| First ram addr |  | Address of first data point stored in core's ram |
| Last ram addr |  | Address of last data point stored in core's ram |
| Thresh hold |  | Thresh hold value for convergency check for the algorithm |
| go |  | Go instruction to K means core |
| Centroid's 1-8 |  | Value of centroids, initialized by APB master, and after convergence get new value from K means core |

Operations regarding register file includes both interface with APB master and with K means core.

#### Interface with APB master

* APB master may write/read from a specific register, by pushing on PADDR signal the enumerated value for the required register, and all relevant APB related protocol signals (chapter 1.3 – ampa apb).

Note: an operation of indirect access which writes to core's RAM may be triggered, as explained in chapter 2.6.1.2 " Interface with K means core " below.

* Indirect Access – read:

an example is shown at chapter 2.2.2 timing diagram.

#### Interface with K means core

* Indirect Access of APB master – write to core's RAM:

Writing of a data point to the RAM located in K means core, is done by APB master writing to ram\_addr reg and following that to ram\_data reg, an example is shown at chapter 2.2.2 timing diagram.

* Transfer of a posedge of "go" signal when go register is triggered, which will result by calculation begin of K means core.

### K means core

The proposed architecture for the k means core block is as described in Figure 12.This block is responsible for running the k means algorithm. It receives the input data points from the register file block by indirect access, as well as the initial centroids. The block output (to the register fil block) is the final centroids value after the algorithm’s end and an interrupt indicating the calculation has been finished.

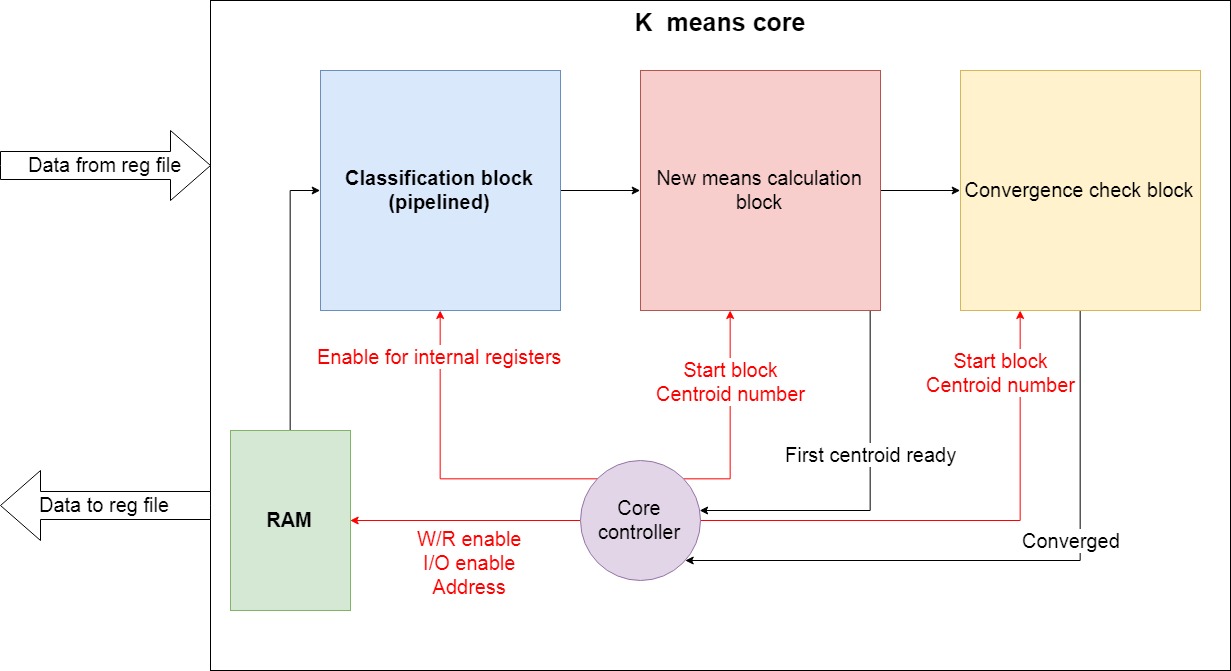


Figure 12:K means core top block diagram

The k means core block is composed of four main components:

* RAM – memory which is used to store the input data points.
* Classification block - this subblock is responsible for the classification step of the k means algorithm. It will start running only after all the input data points are stored in the RAM and the initial centroids are stored within local register of the block and it will run for each data point every iteration of the algorithm.
* New means calculation block – this block is responsible for the centroids update step of the algorithm. It will start running only after the classification block has finished classifying all input data points stored in the RAM. This block will run for as many times as the number of centroids, in the case of this IP, eight times.
* Convergence check block – after the “new means block” calculations are done, this block will be responsible for the “convergence check step” of the algorithm. In case convergence was achieved, it will inform the controller of so.
* Core controller - this is a state machine, which will control the k means core block by sending control signals to each of its internal blocks. It will receive form the register file the value of the registers and

The mains differences of the proposed here architecture to the architecture described in section ‎*1.4.3* are :

1. Different units for the architecture – the proposed paper has a specific RAM and DIV units due to academical limits – there was a choice for a DIV that will provide performance despite this limitation – extend of this in part #TODO-fill part.
2. Interface specifications – there is a CPU-STUB which represent the software commands for the CPU to activate the accelerator and doing all pre-calc and after-calc operations.
3. The Communication prototcol is a APB as presented in chaper 1 - where the CPU STUB is the master which represent the parsing of wqe's from software to commands to the accelartor and the slave is the REGFILE which should release result when the calculation will finish.
4. Micro architecture extension – the paper represent the k means-core – in the proposed here architecture will be extened with micro architecture and down to private implementation with all the details regarding implementation – there's a convergence unit added for example, which does not exist in the paper.

#### K means core controller

The k means core controller is a state machine who controls the core setting signals connected to the other blocks in the core and to the register file block.

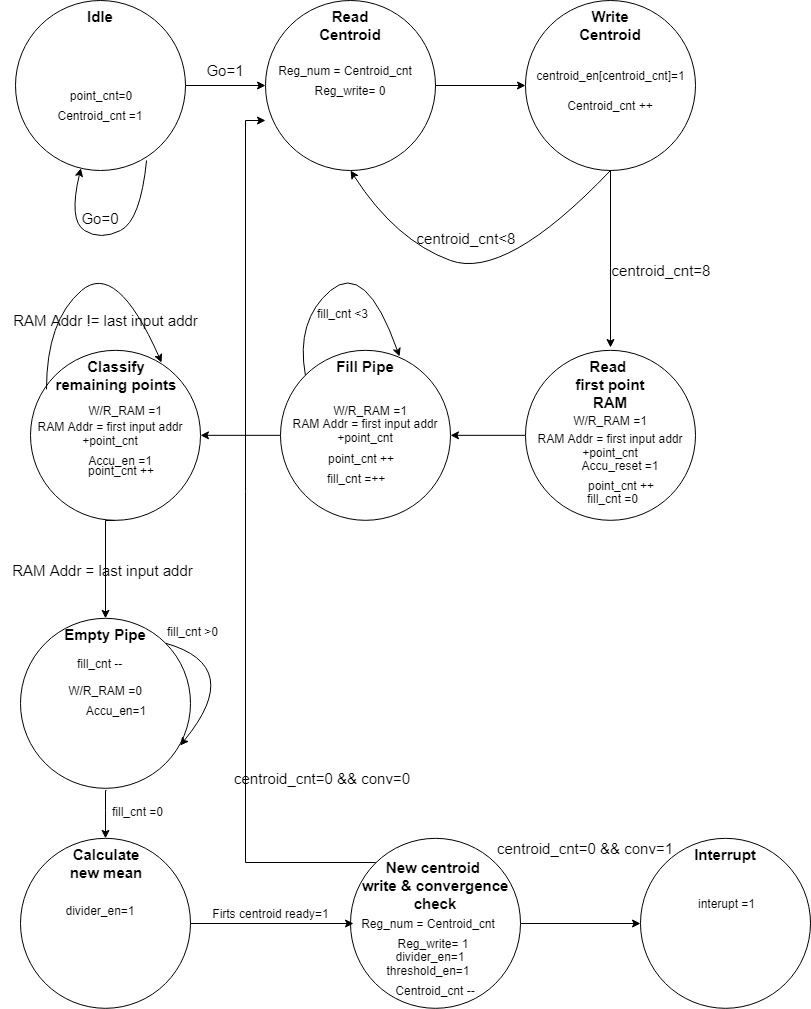


Figure 13: Core controllers state machine

The state machine has the following states:

* Idle – the machine waits in this state until it receives a “go signal”, i.e. until the output value of the “Go register” is set to 1. Once this signal is received, the state changes to the “Read Centroid” state.
* Read Centroid – In this state the centroid value in “centroid X register” of the Register File block is read (X is an integer from 1 to 8, determined by a counter named centroid\_cnt which is initialized to 1 in the idle state). This state is automatically flowed by the “Write Centroid” state.
* Write Centroid – In this state, the value read in the “Read Centroid” state is now available inside the core and it is written in a local register inside the Classification block. After this action, the centroid counter value is promoted by 1 if the counter is smaller than 8(as the number of clusters use in the algorithm) and the next stage is the “Read Centroid” state. If the centroid counter value is 8, it is not promoted, and the state becomes the “Read Input From RAM” state.
* Read First Input From RAM – In this state, the input data at RAM address “first RAM address + point\_cnt” is read(point\_cnt at this state is 0), where “firsts RAM address” is the value of “RAM first point address register” of the Register File and “point\_cnt” is a counter initialized to 0 at the “Idle” state. This state is automatically flowed by the “Fill Pipe” state.
* Fill Pipe - The “Classification block” is a pipelined component which classifies the input data points into cluster by adding the data point to one of eight registers called “Accumulator X register”(X is a integer from 1 to 8). It has a throughput of one data point (after each cycle, one data point is added to the correct accumulator) but it has a latency of four cycles(one for reading the data from the RAM, one for calculating the distance between the centroid ab the point, one to determine the closest centroid and one for adding to point to the accumulator). Therefore, there is a need to fill the pipeline. After the first point is already read, it takes three more cycles to fill the pipeline. This is the purpose of this state: to continue reading data points from the RAM but not allowing not updated data to be written in the accumulator registers of the ”Classification Block”, which are in the final stage of the pipeline. The machine remains in this state for three cycles and it is followed by the “Classify remaining points” states.
* Classify remaining points – In this state, after the pipeline of “Classification block” is full, all it remains is to read the remaining data points (and after they have been read, they automatically go through the pipeline). The machine remains in this state until all data points were read from the RAM, when the RAM Addr signal is equal to the “last input addr”, where “last input addr” is the value of “RAM last point address register” of the Register File.
* Empty – The “Classification block” is a pipelined component which classifies the input data points into cluster by adding the data point to one of eight registers called “Accumulator X register” (X is an integer from 1 to 8). It has a throughput of one data point (after each cycle, one data point is added to the correct accumulator) but it has a latency of four cycles(one for reading the data from the RAM, one for calculating the distance between the centroid ab the point, one to determine to closest centroid and one for adding to point to the accumulator). Therefore, after the last point is written to the “Classification block” local register “Input register”, it will be classified to the write cluster only 3 cycles later(the first two cycles are part of the states “Read Input from RAM” and “Write Input to Input reg”). As a result, is needed this state is needed. In this state, all the state machine do is wait for the last input data to be classified, and it stays in this state for three cycles. After these three cycles, the next state is the “Calculate new mean” state.
* Calculate new mean – In this state, the new centroid number “centroid\_cnt” is calculated by the “New Means calculation block”(“ centroid\_cnt” is 8 in the first time the machine is in this state duo to the fact that this counter stopped at 8 at the last time the machine was in “Write Centroid” state).The machine stays in this state one cycle(the amount of cycles needed by the divider to finish calculating the first new centroid). The next stage is the “New centroid write & convergence check” state.
* New centroid write & convergence check – After the “News Means calculation block” finishes the calculation of the first new centroid, it gives an output of one new centroid every cycle. Therefore, after the first new centroid has been calculated, in order to parallelize the work, the “Convergence check block” can begin by checking the convergence of the first centroid and writing it to the centroid register in the register file. The machine remains in this state for 8 cycles(in order to write the 8 new centroid to the register file and check the convergence for each centroid by comparing it to the old centroid stored in the local registers “Centroid register X” of the “Classifying block”). If there is convergence, the “Convergence check block” will send a signal to the controller pointing that the algorithm has ended. In this case, the controller goes to the final state, the “Interrupt” state. If convergence was not reached, the

machine returns to “Read Centroid” state, starting a new iteration of the algorithm.

The controller’s states machine is also described in the table below:

Table 2: K means core controller states

| **Present State** | **Next State** | | | **Controls signals from controller** | | **Receiver** | | **Meaning of control signal** | **Signals to controller** | **Sender** | **Meaning of signal to controller** | **Internal signals** | **Meaning**  **Of internal signal** | | **Notes** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Idle | Read Centroid | If Go =1 | |  | |  | |  | Go | TB | Indicates whether or not to start running algorithm | Centroid\_cnt=1 | Used to count centroids. Initiated at 1 | |  |
| Idle | If Go=0 | |
| Point\_cnt=0 | Used to count data points. Initiated at 0 | |
| Read Centroid | Write Centroid | | | Reg\_write=0 | | RF | | Enables read from RF |  |  |  |  |  | |  |
| Reg\_num= centroid\_cnt | | RF | | Register address in RF |
| Write Centroid | Read Centroid | | If Centroid\_cnt<8 | Centroid\_en[centroid\_cnt]=1 | | CB | | Enables “centroid register” of number “centroid\_cnt” local register in CB |  |  |  | Centroid\_cnt ++ if centroid\_cnt<8 | This counter is promoted by one case it is smaller than 8 | | Centroid number “centoid\_cnt” stored in register file is written to local reg in CB |
| Read first point RAM | | If Centroid\_cnt=8 |
| Read first point RAM | Fill Pipe | | | W/R\_RAM=1 | | RAM | | Enables reading from RAM |  |  |  | Point\_cnt ++ | The counter of how many points were read from RAM is promoted by 1 | | First input data point from RAM enters CB pipeline |
| RAM Addr= first input addr + point cnt | | RAM | | Set RAM address to address of first input point + number of points already read |
| Accu\_reset=1 | | CB | | Resets all sixteen local CB registers called Accumulators and accumulators counters | Fill\_cnt=0 | This counter is used to count how many points entered CB pipeline, ­­­­­in order to know if it is full | |
| Fill Pipe | Fill pipe | | Fill\_cnt<3 | W/R\_RAM=1 | | RAM | | Enables reading from RAM |  |  |  | Point\_cnt ++ | | The counter of how many points were read from RAM is promoted by 1 | Reads points from RAM, but does not allow writing to the CB’s local registers “Acummulators” and “Acummulator counters  ” |
| Classify remaining points | | Fill\_cnt=3 | RAM Addr= first input addr + point cnt | | RAM | | Set RAM address to address of first input point + number of points already read | Fill\_cnt++ | | This counter is used to count how many points entered CB pipeline, ­­­­­in order to know if it is full |
| Classify  remaining points | Empty Pipe | | RAM Addr = last input addr | W/R\_RAM=1 | | RAM | | Enables reading from RAM |  |  |  | Point\_cnt ++ | | The counter of how many points were read from RAM is promoted by 1 | Reads points from RAM, and allows writing to the CB’s local registers “Acummulators” and “Acummulator counters  ” |
| Classify  remaining points | | RAM Addr != last input addr | RAM Addr= first input addr + point cnt | | RAM | | Set RAM address to address of first input point + number of points already read |
| Accu\_en =1 | | CB | | Enables writing to CB’s internal registers  “Accumulators” and “Accumulators counters  ” |
| Empty Pipe | Empty Pipe | | Fill cnt >0 | W/R\_RAM=0 | | RAM | | Does NOT Enables reading from RAM |  |  |  | Fill\_cnt=0 | | This counter is used to count how many points entered CB pipeline, ­­­­­in order to know if it is full | Does not reads any more points from RAM, but still allows writing to the CB’s local registers “Acummulators” and “Acummulator counters  ” in order to properly process the final points. |
| Calculate new mean | | Fill cnt =0 | Accu\_en =1 | | CB | | Enables writing to CB’s internal registers  “Accumulators” and “Accumulators counters  ” |
| Calculate new mean | Calculate new mean | | First centroid ready =0 | Divider enable =1 | | NMB | | Enables the divider block to run | First centroid ready | NMB | Indicates the first new centroid has been calculated |  | |  | Calculates the first new centroid(centroid number 8,duo to the fact that centroid\_cnt is 8) |
| New centroid write & convergence check | | First centroid ready =1 |
| New centroid write & convergence check | New centroid write & convergence check | | Centroid\_cnt >0 | Divider enable =1 | | NMB | | Enables the divider block to run | conv | CCB | Indicates if convergence was reached | Centroids\_cnt -- | | This counter is demoted by one case it is larger than 0 | Writes the new calculate centroid to the RF while checking if it upholds the convergence condition |
| Read centroid | | Centroid\_cnt =0 & conv =0 | Reg\_write=0 | | RF | | Enables write to RF |
| Interrupt | | Centroid\_cnt =0 & conv =1 | Reg\_num= centroid\_cnt | RF | | Register address in RF | |
| Treshold\_en =1 | CCB | | Enables the CCB to run | |
| Interupt |  | |  | Interrupt=1 | | RF | | Sends interrupt signal as output from the core |  |  |  |  | |  | Stops the core execution |

#### Classification block

The “Classification block” is a pipelined component which classifies the input data points into cluster by adding the data point to one of eight registers called “Accumulator X register” (X is an integer from 1 to 8). It has a throughput of one data point per cycle (after each cycle, one data point is added to the correct accumulator) but it has a latency of four cycles(one for reading the data from the RAM, one for calculating the distance between the centroid and the point, one to determine to closest centroid and one for adding to point to the accumulator.

In Figure 14 the diagram of the Classification block can be seen.



Figure 14:Classification block diagram

The Classification block pipeline is composed of three mains parts:

1. The distance calculation from the input data point (stored in the “Input register”) to the centroids, which are stored in local registers called “Centroid Register X” (X being a integer from 1 to 8). This calculation is done by o module called “Distance calculator” which is basically two sub modules in series: the first calculates the subtraction of the data in the input register from the data in centroid register “X”, the second is a module which calculates the absolute value of the mentioned subtraction. It can be seen in details in Figure 15.



Figure 15: Classification block distance calculation part

1. The second part is the minimum distance calculation. In this part, all the distances from the first are compared, and in the index of the closest centroid is found and given as output of this block. It can be seen in details in Figure 16.



Figure 16: Classification block minimum distance calculation part

The third and final part of the Classification block pipeline is the accumulation of the input data point in the desired accumulator register, chosen by the index received from second part. In this part, there are sixteen registers: eight accumulators registers and eight counters registers. Each accumulator represents a cluster and stores the summed of all points assign to this cluster at a given time. The counters are register which simply keep track of how many points were assigned to each cluster. The index received from the previous part is used as a selector for two decoders, in order to choose to each accumulator, register and counter register the input data point should be added. This part can be seen in details in Figure 17.



Figure 17: Classification block accumulator part

#### New Means Calculation block

The “New Means Calculation block” is responsible for centroids update step of the algorithm. It does so by dividing the value of each accumulator(stored at the local “Accumulator reg” register in the “Classification block”) by the number of points assigned to them(stored at the local “Accumulator counter reg” register in the “Classification block”). Its architecture can be seen in Figure 18.

In this block there are two multiplexers, one for choosing the accumulator register and one for choosing its counter, while the select signal is a control signal named “centroid\_cnt” which counts how many new centroids still need to be calculated. The value stored in the accumulators (a value represented in fixed point arithmetic) needs to be translated to an integer duo to the fact that the divider chosen for the architecture is an integer divider. Therefore, after the accumulator is chosen by the multiplexer, its value goes though a module called “Fixed point to integer”. For more about this module see section *Fixed Point to Integer module*.

The accumulator value converted to integer is then inserted into the divisor as the dividend, and the value of the accumulator counter is inserted as the divisor. The divider calculates the division in hardware level (see section *Division in Hardware* for more information) , and when it is done, its signals to the core’s controller the division’s end with the signal “Division\_ready”.

The result of the division needs to be converted back into fixed point representation, which is done by the module “Integer to Fixed Point”. For more about this module see section *Integer to Fixed Point module*.

The “New Means Calculation block” is in a numb state while the “Classification block” is working, and it starts only after all data inputs were classified into accumulators. In order to enable the block’s operation, the core’s controller asserts the signal “Divide\_en”, enabling the block to start its operation.

Figure 18: New means calculation block diagram

##### Input Data Characteristics

The range of the input data set which the proposed architecture can handle is [-3.278,3.546] (range described in the paper). As described in section *Preliminary analysis for Hardware implementation* , in order to represent this range by fixed point representation 13 bits will be required: 1(the MSB) to determine the sign of the number(in two’s complement convention),2 for the integer part of the number and 10 for the fractional part of the number.

The data set size described in the paper was of a matrix of 415X7, i.e. 415 points with 7 coordinates each, so in order to represent the accumulator’s results, in the worst case where all data points enter the same accumulator, the accumulator maximum value per coordinate will be as high as:

In order to represent this value, 22 bits will be needed: 1(the MSB) to determine the sign of the number, 11 for integer part of the number and 10 for the fractional part of the number.

##### The division method

As described before, duo to the complexity and time demanding characteristics of the floating point representation, the proposed architecture works with the fixed point representation. Therefore, there was a need to find a proper way to divide fixed point values.

The proposed method was to convert the fixed point number to an integer by shifting all fractional bits left (multiplying by 1024 ), dividing the converted number by using an integer divider(with known an simple algorithms, as describe in section *Division in Hardware*), and after the division shifting right the result in order to return to the fixed point representation.

This method was tested for accuracy in an excel tab, and for 10000 data points in the range of [-1500,1500] divided by integer numbers in the range [1,415]. The result of this test was maximum error of 0.016 %, and it can be seen in the figure bellow:

Figure 19: Precise floating-point division x proposed division method values

##### Division in Hardware

Even though mathematical operations like addition and subtraction are relatively simple and intuitive in hardware implementation, the division operation isn’t. As a result, over the years, many algorithms which implement the division operation in hardware level were developed. In this section, some of these algorithms are presented.

###### Slow division methods:

Few of the slow division methods are related to division by repeated subtraction, which answer to the basic question: given integers a and b, how many times b fit in a?

There are two values for output result, which are Quotient and Remainder of the division , where Quotient represent the integer amount of times b fits in a and Remainder is the what is left after subtracting b from a as many times as possible without getting an negative result ,ranging from 0 to (b-1). To get those two values, the basic algorithm has two main methods: restoring and non-restoring.

Restoring method:

In restoring division, the divisor is shift-positioned and subtracted from the dividend. If subtraction of the divisor produces a negative result at any bit position relative to the dividend, the operation at that bit position is unsuccessful, and a 0 is placed in the corresponding location of the quotient. The divisor is added back (restored) to the result of the division operation, then the next highest bit of the dividend is shifted into the left bit position of the result. As each bit of the dividend is shifted from right to left, the quotient is built up from left to right. After n shifts, where n represents the number of bits in the dividend, the division operation is complete. Complete hardware for restoring division is shown in Figure 20. In this figure an n-bit positive divisor is loaded into register M and n- bit dividend is loaded into register Q at the start of the operation. After the division is complete, the n-bit quotient is in register Q and the remainder is in register A. The result after the last restore operation is the remainder.

Restoring division algorithm is very similar to manually performing long division.

The algorithm’s steps:

1. Set Count to 0, load the divider into register M, load the dividend into Q register and put 0 in A register.
2. Start loop for n times.
3. Shift A & Q left one binary position - (A,Q) are concatenated through the algorithm iterations, so when shifted left, A LSB becomes Q MSB and Q LBS is now a blank space.
4. Subtract M from A, placing the answer back in A.
5. If the sign of A < 0, set the new LSB of Q to 0 and add M back to A (restore A).
6. Otherwise, set the new LSB of Q to 1.
7. Check for count, when count = n-1 then stop the loop, where n is dividend width in bits.
8. The result will be store in Q and the Remainder in A.



Figure 20: Restoring division algorithm diagram

Non-Restoring method:

Non-restoring Division Algorithm comes from the restoring division. The restoring algorithm calculates the remainder by successively subtracting the shifted denominator from the numerator until the remainder is in the appropriate range. The operation in each step depends on the result of the previous step. Non-restoring division has a quotient digit set of { 1, - 1} instead of the conventional binary digit set. By the non-restoring division approach, we find the -1 of the quotient bit can be simply set to 0, and the quotient is the actual quotient that we want to find. We dismantle Q into bits.

The algorithm’s steps:

1. Set Count to 0, load the divider into register M, load the dividend into Q register and put 0 in A register.
2. Shift A & Q left one binary position - (A,Q) are concatenated through the algorithm iterations, so when shifted left, A LSB becomes Q MSB and Q LSB is now a blank space.
3. Subtract the divisor from the dividend.
4. Start loop for n times.
5. If A < 0:
   1. set the new LSB of Q to 0
   2. Shift A & Q left one binary position.
   3. Add the dividend to A
6. If A>0
   1. set the new LSB of Q to 1
   2. Shift A & Q left one binary position.
   3. Subtract the dividend to A
7. After n iterations, if A is negative, than add the dividend to A.
8. The result will be store in Q and the Remainder in A.



Figure 21: Non-Restoring division algoritm diagram

Note: Non-restoring is faster (max n+1 steps for n digits a width against the restoring method with needs up to 2n-1 steps – might need to correct/restore almost at each step of the restoring way).

SRT division:

SRT division is like non-restoring division, but it uses a lookup table based on the dividend and the divisor to determine each quotient digit.

For example, when implementing radix-4 SRT division, each quotient digit is chosen from five possibilities: { −2, −1, 0, +1, +2 }. Because of this, the choice of a quotient digit does not need to be perfect; later quotient digits can correct for slight errors. (For example, the quotient digit pairs (0, +2) and (1, −2) are equivalent, since 0×4+2 = 1×4−2). This tolerance allows quotient digits to be selected using only a few most-significant bits of the dividend and divisor, rather than requiring a full-width subtraction. This simplification in turn allows a radix higher than 2 to be used.

Like non-restoring division, the final steps are a final full-width subtraction to resolve the last quotient bit, and conversion of the quotient to standard binary form.

###### Fast division methods:

Some fast methods require doing multiply a by the inverse of b(1/b) instead of dividing a/b.

Newton-Raphson division:

Newton-Raphson uses a mathematical computation method called "Newton's method" to find a approximation of the inverse of b (the divisor) and then going towards doing multiplication of two numbers instead of dividing( D= 1/b and then a/b can be done as a\*D).

In order to apply Newton's method to find the reciprocal of b, it is necessary to find a function that has a zero at D=1/b, the choice of this function is not the obvious choice ( which is [ f(x) = bx – 1 ]) but a one that leads to an iterative algorithm by which every step result with an error for the approximation of D in which one can stop when reached a required limit for the error. Afterwards, when reached an approximation of some amount of bits of D, then the multiplication can be done to reach an approximated result of the division.

Goldschmidt division:

This method become useful when the divisor has the property of ( 0 < b < 1). There's an iterative process in which there's a multiply of both dividend and divisor by factor ( the i is for the iteration time) until the divisor is sufficiently close to 1 (b\*F1\*F2\*…\*Fn =1).

The steps for Goldschmidt division are:

1. Generate an estimate for the multiplication factor Fi.

2. Multiply the dividend and divisor by Fi.

3. If the divisor is sufficiently close to 1, return the dividend, otherwise, loop to step 1

This division method will also result with an error for approximation.

##### Fixed Point to Integer module

This module converts the accumulator value representation from fixed point to integer. In order to do so, it copies the value of the 22 bits which represent the accumulator in fixed point representation to a new variable, with also 22 bits. This new variable is now an integer, with value as the old fixed-point value multiplied by 1024.

##### Integer Divider module

The divider chosen to be part of the architecture is an integer divider, as described in section *The division method*. It receives two integers, dividend and divisor, and start signal which enables it to start its operation. When the division is done and available on its quotient output, a signal named “complete” is driven high. The length of the output in bits is as the length of the dividend, therefore, the output is 22 bits wide.

##### Integer to Fixed Point module

This module returns the output of the divider to fixed point representation. As explained before, in the range of values [-3.278,3.546] there is a need for 13 bits to represent in fixed point representation, but the output of the “Integer Divider” module is 22 bits long. Therefore, this module converts the output back to fixed point creating a new variable with 13 bits (which is the value in fixed point). Then it writes into this variable to following values:

* The MSB of the divider’s output is copied to the variable’s MSB(the sign bit).
* The 10 LSBs of the divider’s output are copied to the variable’s 10 LSBs.
* Bits number 10 and 11 of the of the divider’s output are copied to the variable’s 10th and 11th bits.

#### Convergence Check block

The convergence check block is responsible for the convergence check step of the algorithm. It does so by checking if any of the new centroids calculated in the “New Means Calculation block” value is close enough (within a pre-decided threshold stored at “Threshold register ” in the register file) to its old value(the value stored in the beginning of the iteration, stored in local registers of the classification block).

For each new centroid, the distance of new centroid to old centroid is calculated by using a subtractor and a absolute value calculator module. After this is done, this distance is goes into a comparator, which compares it to the threshold value. The result of the comparation is 1 if the distance is smaller or equal to the threshold, else it is 0. This one bit result then enters an adder with a local register on it’s output, called the convergence register. The purpose of the adder and register is to count how many of the centroid complied with the convergence requirement.

The final module of this block is the “Has converged” module, which verifies the requirements for convergence of the algorithm, i.e. it checks if all centroids were checked for convergence by checking if the control signal “centroid\_cnt” is 0 and also is the value in the local register “Convergence reg” is 8(all centroids have converged).

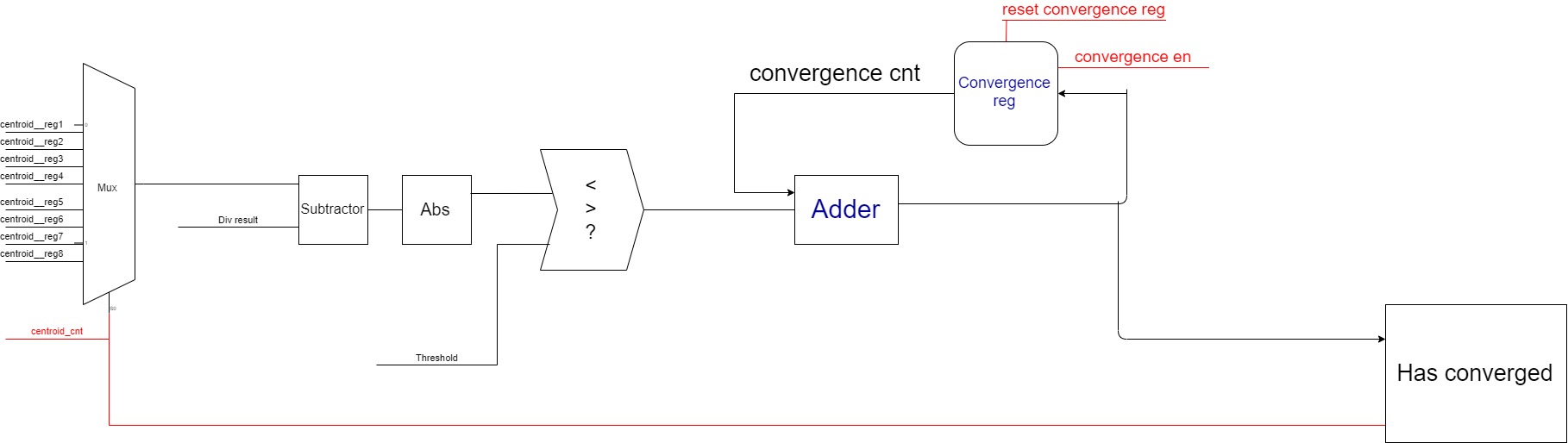


Figure 22: Convergence check block diagram

#### K means core Pipeline diagrams

K means core pipeline view is divided into two pipelines, the division can be seen in the figure below, the reason for this division is the Read-After-Write dependency between the 2 pipes.  
This results by having one pipe working while the other wait for it's results. The left pipe result is all data points classified to clusters, which then can be used to calculate the new means for next iteration (or the last means if there will be a convergence). Then for the next iteration of classifying the data points, the new means value has to be ready, therefore the first pipe will "sleep" until second pipe will finish bringing the result of calculating new means. The division for the pipes can be viewed in Figure 23.

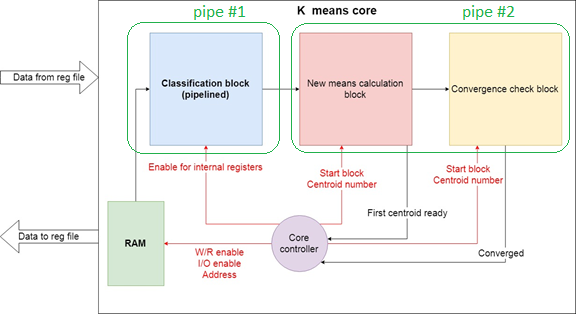


Figure 23: K means core block diagram divide into pipelines

Pipe #1 – Classification block:

The figure below shows the pipe stages of classification block – for more see ‎2.6.2.2.

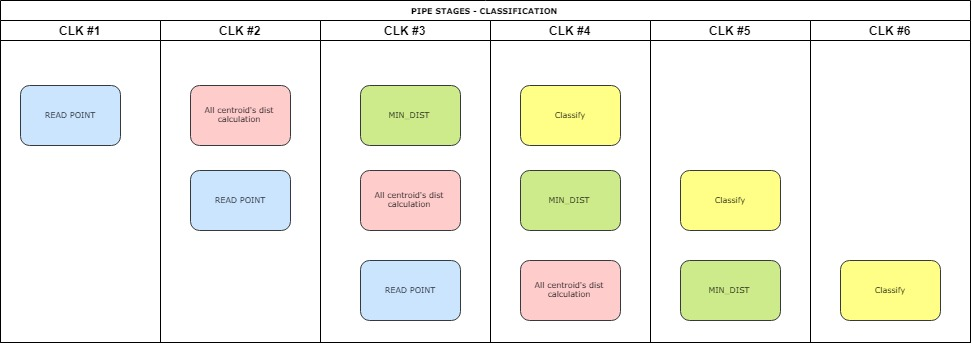


Figure 24: K means core first pipeline diagram

Pipe #2 – New means calculation (div operation mainly) & convergence check block:

The figure below shows the pipe stages of the second pipe – for more see ‎2.6.2.3

TBD – maybe remake those stages cause div shrinks into 1 cycle.

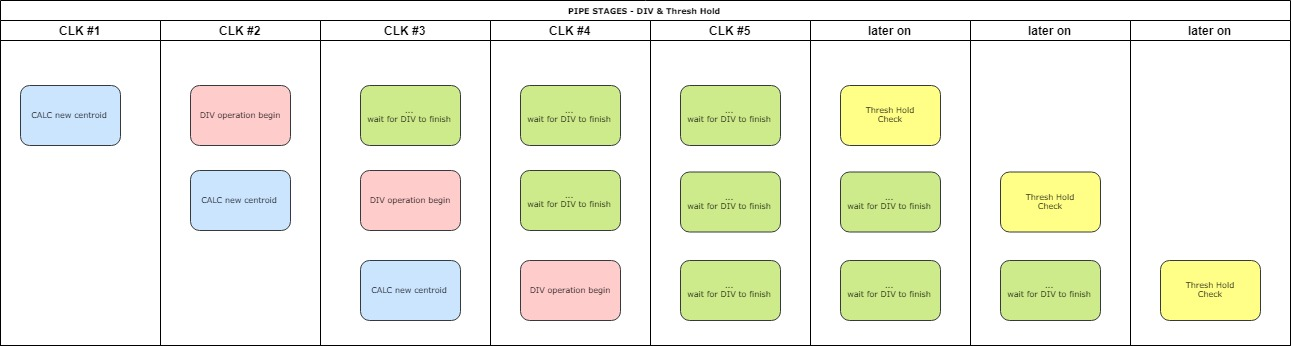


Figure 25: K means core second pipeline diagram

#### K means core timing diagrams

The first two pipeline diagrams show the most important signals inside the core, from the core to the register file and from the register file to the core.

In Figure 23, the core starts its action when “Go” signal is asserted to one. After this, the controller starts enabling the “Classification block” internal “centroids registers” to sample the read data from the register file (the registers read are the centroid registers in the register file).

After the eight local centroid registers are updated (this is known by an internal core signal called “centroid cnt”), the core starts filling the internal pipeline of the “Classification block” with input data points stored in the RAM. It does that by asserting signals to the RAM (“RAM write or read” and “RAM addres”) . Every data point that enters the pipeline is counted by ac ore internal signal “fill cnt”

When the “fill cnt” signal reaches the value 3, it means the pipe is filled, therefore the core enables the “Classification block” internal registers called ”Accumulators registers” and “Accumulators counter registers” to start sampling their inputs.

These process continues until all input data points are read from the RAM(the signal RAM ADDR from controller to RAM is no equal to the last address in the RAM where there are inputs) , the core allows the sampling of the “Classification block” internal registers(”Accumulators registers” and “Accumulators counter registers”) for three more cycles in order for the last input data point to go through all the pipeline. At the fourth cycle, the “Classification block” internal registers enables are deserted, and the “Classification block” stop being active.

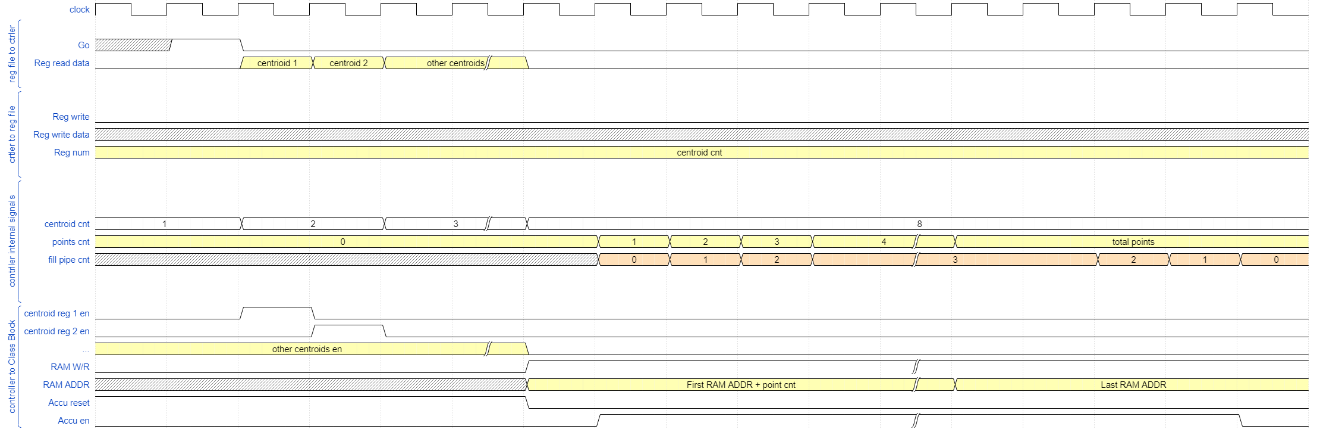


Figure 26: Core timing diagram part 1

In Figure 24 the rest of the algorithm can bee seen. The controller asserts a signal called “divider start en” every two cycles, enabling the divider in the “New Means Calculation” block to calculate the division of the accumulators by the accumulators counters.

When the division is done, the divider asserts a signal to the controller called “complete”. When the first assertion of “complete” occurs, the controller asserts the “thershnol en” signal to the “Convergence Check” block, enabling this block to sample the first new accumulator calculated in the “New Means Calculation” block, and verifying convergence.

At the same time the convergence of the first new mean calculated is checked, the second new mean is calculated, so that the “New Means Calculation” block and the “Convergence Check” block work as a pipeline, calculating all eight new means and checking them for convergence.

If the algorithm has reached convergence (case presented in Figure 24 ), the “Convergence Check” block asserts a signal called “has converged” to the controller, which receives it and then in the next cycle, asserts the interrupt signal to its host.

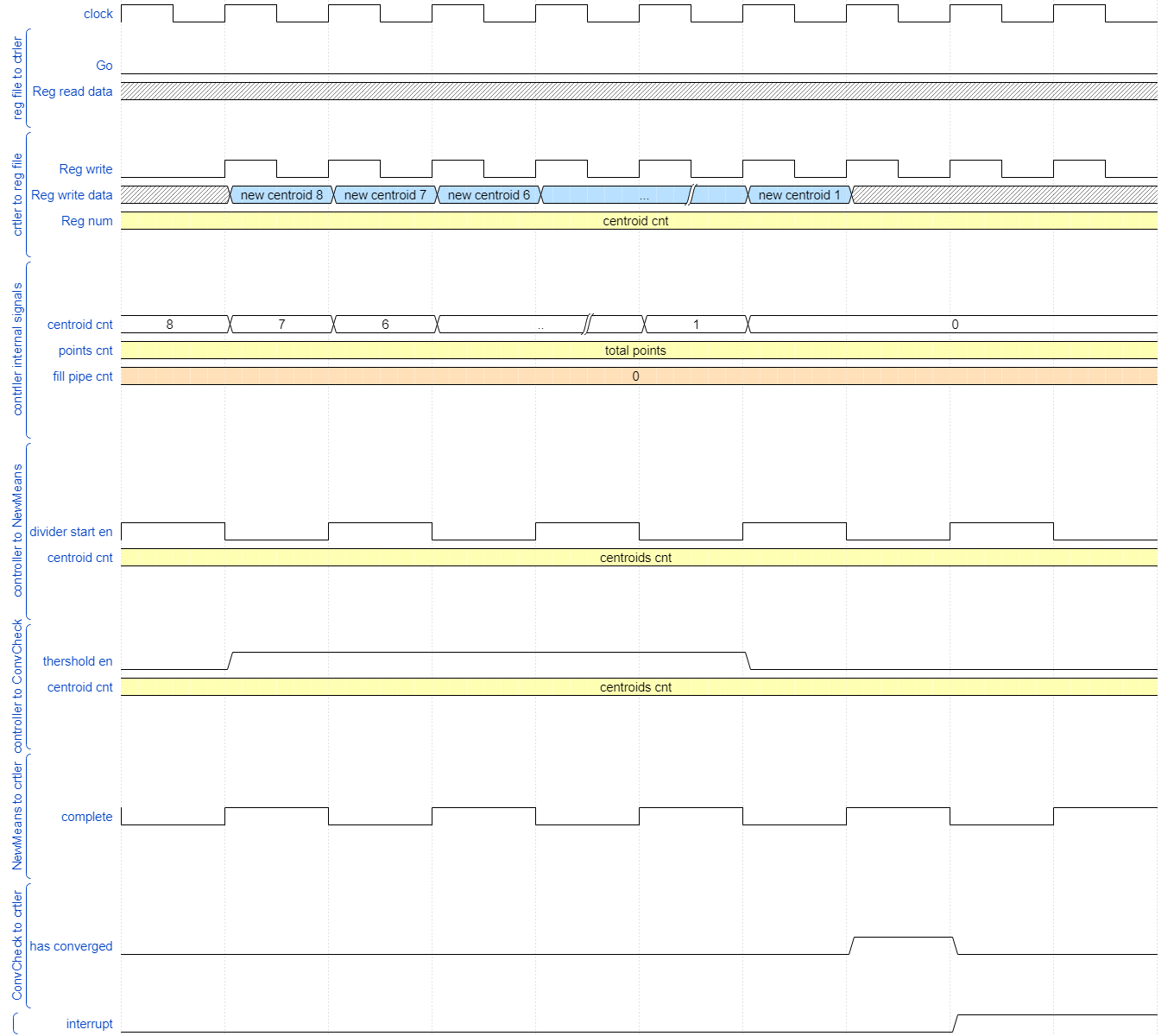


Figure 27: Core timing diagram part 2

##### K means core internal blocks timing diagrams

In this paragraph, there are timing diagrams for the expected functionality of the k means core internal blocks.

###### Classification block timing diagrams

In Figure 25, the first part of the “Classification block” timing diagram can be seen. In this part, first all eight local registers called “centroid registers” are given an enable signal, enabling them to sample their input which is the centroid of the last iteration stored at the RAM.

After all eight “centroid registers” are updated, the controller asserts the enable of the local registers “input register”, enabling it to sample the first input point.

The output of the “centroid registers” and the “input register” are used as input by the “distance calculators”( for more see section ‎2.6.2.2) which calculate the distance between the point and the centroids. The outputs of this block are the distances and indexes (1 to 8), all which are pipelined to the next part.

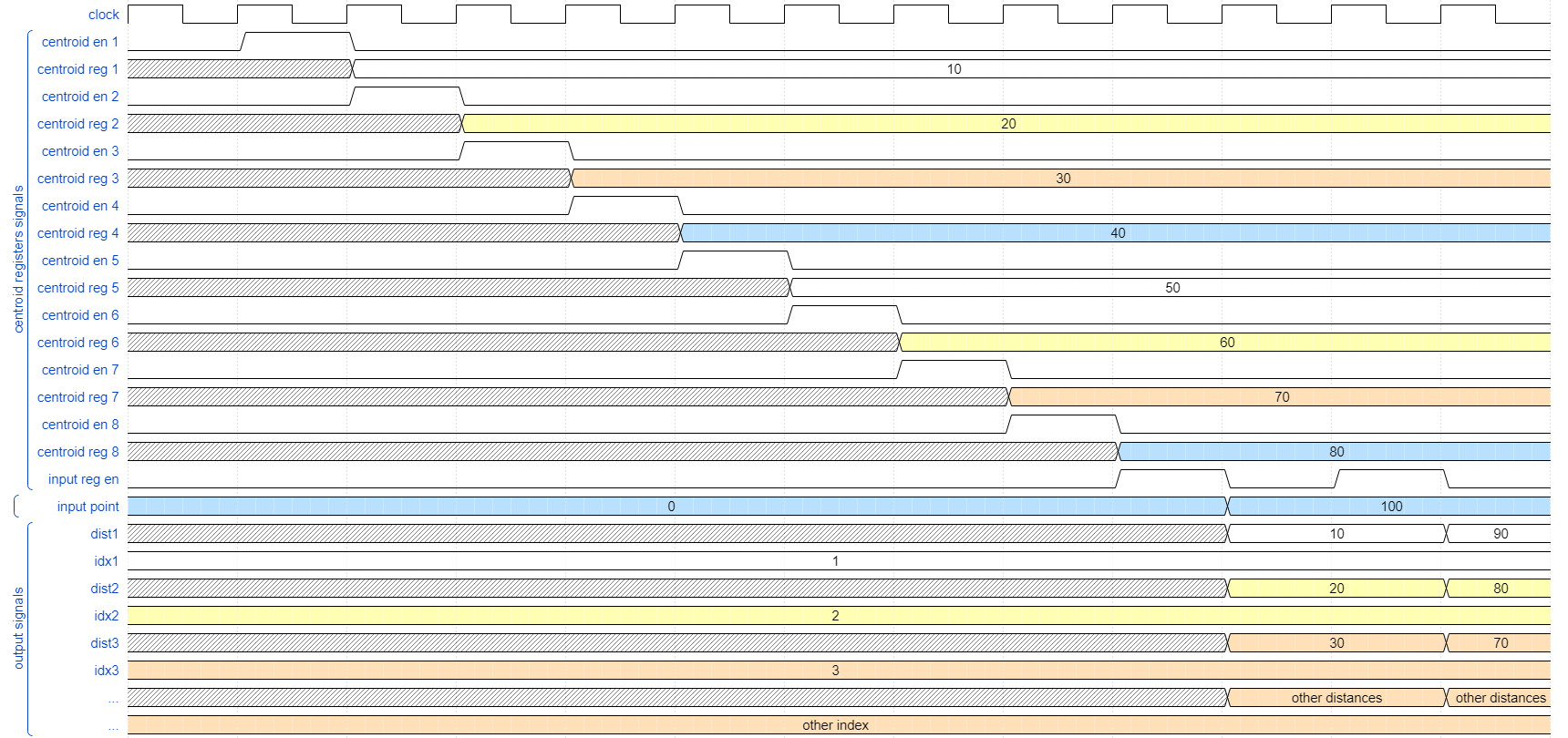


Figure 28: Classification block timing diagram part 1

The next figure, Figure 26, show the internal signals at the “Classification block” at the second part of the block: the minimum distance finder. All eight distances enter the section, and every cycle two are compared, the smallest “moves on” the comparison chain together with the index of the accumulator it represents.

The output signal is the minimum distance found and the index of the accumulator it represents.

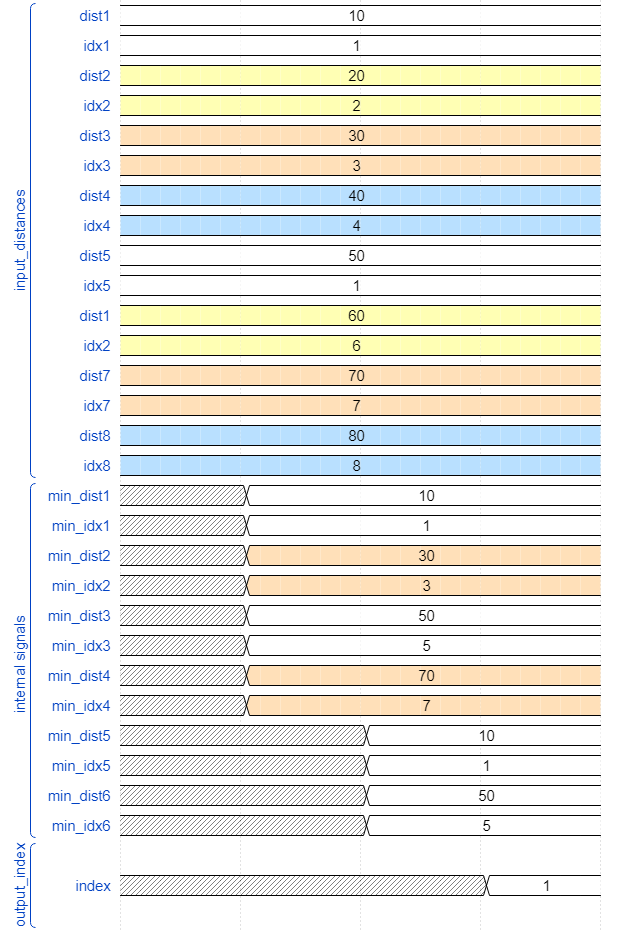


Figure 29: Classification block timing diagram part 2

Figure 26 shows the third part of the “Classification block” timing diagram. In this figure, an example is shown, an input data point reaches the last part of the pipeline, the accumulation part, and it is supposed to be part of the second accumulator.

The input point and its index are received in the second cycle. Immediately, one input of adder number 2(adder connected to the input of accumulator register number two) changes to the input point(the other is the accumulator register number two output), while the other adders inputs remain the same(one is the accumulator registers output and the other zero).

In the third cycle, the accumulators sample their inputs, duo to the change only in output of adder number two, only output of accumulator register number two changes.

The same for the accumulator counter registers, only accumulator counter register number two samples a value different from its value, which is its value plus one (the output of the adder connected to its input is the only one which changed).

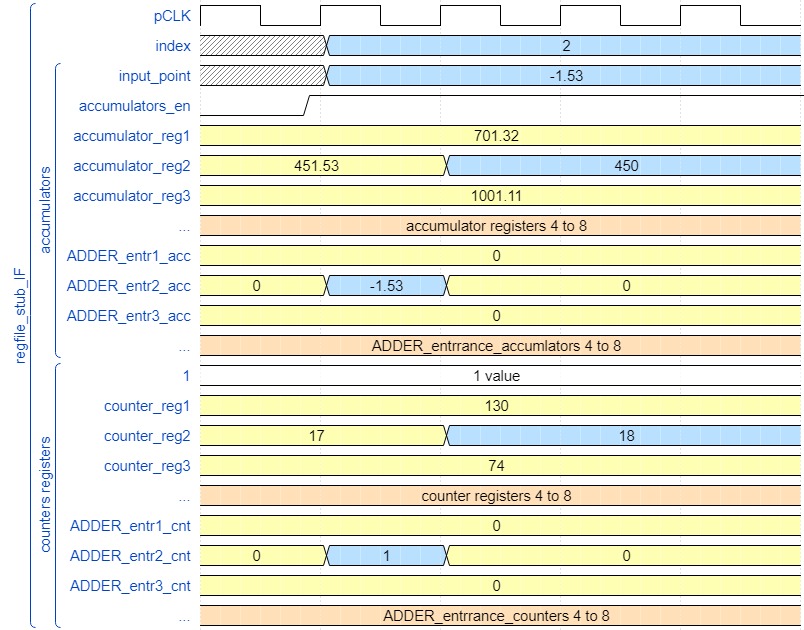


Figure 30: Classification block timing diagram part 3

###### New Means Calculation block timing diagrams

Figure 28 shows the timing diagram of the “New Means Calculation block”. As it can be seen, this block receives a signal from the controller called “start” and two cycles after that it outputs the new mean.

The new mean is the output of the divider inside the block. The input of the divider is the output of a multiplexer which receives as “selector” signal the signal “centroid\_cnt” from the controller, indicating which new mean to calculate.

When the new mean is ready, the block asserts the “complete” signal to the controller.

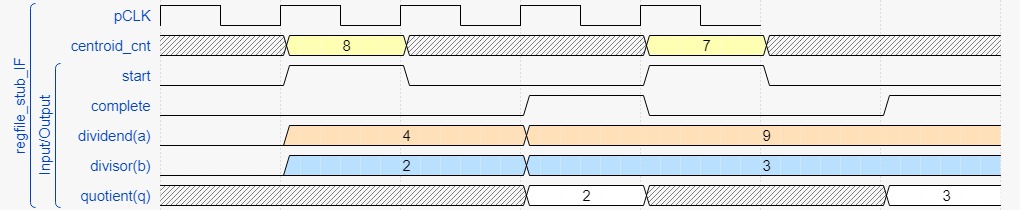


Figure 31: New Means Calculation block timing diagram

###### Convergence Check block timing diagrams

In Figure 29, the “Convergence Check block” timing diagram is described. It receives the “centroid\_cnt” singnal form controller indicating which new mean is now its input, as well as a new calculated mean, the last iteration mean and a constant input called threshold, which is the comparison threshold.

When the controller asserts the input signal to the block “convergence\_en”, the comparator is enabled and the old and the new means are compared. In the case they are as close as the threshold (case presented in the diagram below), the comparator asserts a signal which enables the convergence\_cn local register to sample its value plus one, adding to the convergence counter.

After all new means are checked, if the convergence counter reached eight (case presented in the diagram below), the block outputs a signal to the controller called “conv”, indicating the end of the algorithm.

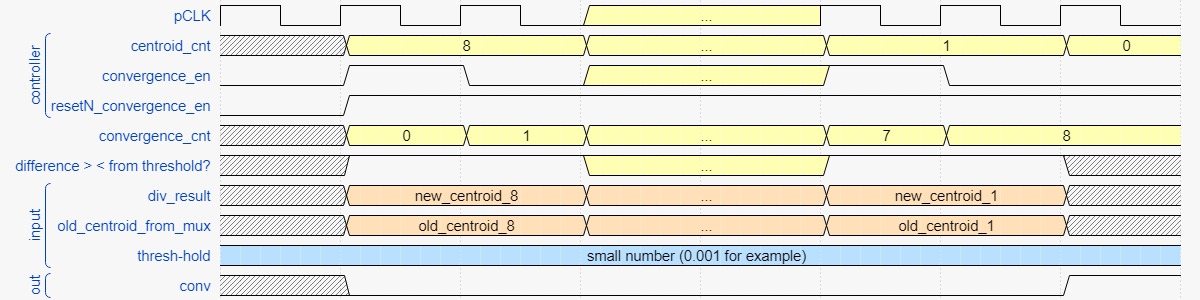


Figure 32: Convergence Check block timing diagram

## Performance

In this section, an analysis of the K means core latency and throughput is presented. The k means core has two pipelines, as described in section *‎2.6.2*, therefore this analysis is divided into the analysis of these two pipelines latencies and throughputs.

### Classification block latency, throughput and bandwidth

First, it is important to highlight that the total latency of the classification block depends directly on the number of input data points, since this block must classify all data points defined by user.

Second, the classification block is a pipelined module, therefore first latency component is the number of cycles needed to fill this pipeline. The classification block consists of three main parts, as explained in section *‎2.6.2.2*, as a result three clock cycles are needed to fill this module pipeline.

After the pipeline is filled, every clock cycle a new point is classified, resulting in a throughput of one new point classification per cycle. The bandwidth therefore can be calculated as the point size in Bytes per clock cycle. In the described design, each data point has a bit width of 91 bits, thus the bandwidth of the classification block is 91 bits per clock cycle period. For example, if the used clock has a 100 Mhz frequency, the block’s bandwidth is as described below:

As the final data point enters the pipeline, it must go thought the pipeline stages, hence three more clock cycles are needed to complete the classification block functionality.

In conclusion, the classification clock latency is as described in the equation below:

### New means calculation block and Convergency check block latency and throughput

The second pipeline is the one formed by the new means calculation block and the convergence check block. This pipeline is directly dependent on the number of centroids used in the algorithm, being it in this design eight centroids.

As this pipeline consists of two modules, it requires one clock cycle to be filled. After the pipeline is filled, each clock cycle a new centroid is calculated and the calculated in the previous cycle is checked for convergency. Accordingly, a throughput of one new centroid calculation and convergence check per cycle. The bandwidth therefore can be calculated as the centroid size in Bytes per clock cycle. In the described design, each centroid has a bit width of 91 bits, thus the bandwidth of the classification block is 91 bits per clock cycle period. For example, if the used clock has a 100 Mhz frequency, the block’s bandwidth is as described below:

When the final new centroid enters the pipe, it must go thought the second pipeline stage, hence one more clock cycle is needed to complete this pipeline functionality.

In conclusion, the new means calculation block and convergence check block pipeline latency is as described in the equation below:

### Total performance

It is important to emphasize that the total latency of the k means core depends on the number of iterations it takes to converge, which depends of the data input points and initial centroids values, thus being an unknow variable. Hence, the total latency of the k means core is not known neither constant, however it is bounded by the latency per iteration times the iteration number it took the algorithm to converge.

Furthermore, the two mention above pipelines are data dependent, i.e., the second pipeline depends on the result of the first pipeline and the opposite, the total latency per iteration is their sum.

To conclude, the total latency per algorithm iteration of the k means core is as described in the equation below:

Accordingly, the total latency of the k means core is as described in the equation below:

# Zero Order Verification

In this section, there is a description of the tests made for the described implementation.

## Block diagram of the testbench

The testbench made in order to verify the implementation of the design described herein was called top\_tb.sv. Its structure is as shown in Figure 33. It is basically composed by an instantiation of the K means TOP(which is the verified DUT) , a clock generator and a stimulus generator and driver.

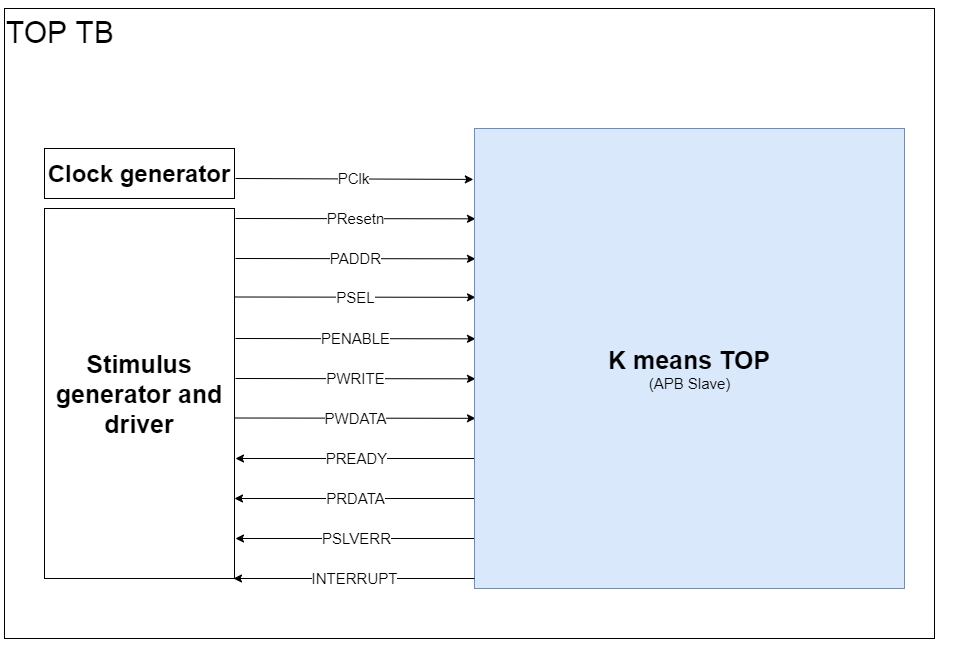


Figure 33: Top tb block diagram

## Tests description

Two actual tests were implemented in the described testbench:

* “Sanity check” test: this test verifies with a simple example the functionality of the k means top module.
* “Do nothing” test: this test verifies the case that k means top module should not change the initial centroid values because they are the same as the data points, therefore being the expected final centroid values.

### “Sanity check” test description

In this test, ten data points are inserted in the RAM, which have different values to the eight initial values of the centroids registers in the register file.

The testbench first writes via APB protocol to the eight centroids register in the register file. The data written to these registers are points containing zeros on six dimensions and in the first dimension(the 13 LSBs of the data point) they contain decimal values from 1 to 8, being 1 written to centroid register number one and similarly to the others registers.

Then, the testbench writes via APB protocol the decimal value 1 to the register which stores the first RAM address containing data. In the same way, it writes the decimal value 10 to the register which stores the last RAM address containing data.

Next, by indirect access, the testbench writes the ten data points to the RAM in the DUT. Between these ten data points, eight contained data different from zero only in their first dimension (the 13 LSBs of the data point), one has data different from zero in its second dimension ( bits 14 to 26 of the data point ) and the last point has data different from zero in its first dimension (the 13 LSBs of the data point) and third dimension ( bits 27 to 39 of the data point ).

Finally, the testbench writes the value ‘1’ to the go register in the register file, and from then it waits for the algorithm to end, which is signaled by the rising edge of the interrupt output signal of the k means top module.

### “Do nothing” test description

In this test, eight data points are inserted in the RAM, which have values equal to the eight initial values of the centroids registers in the register file.

The testbench first writes via APB protocol to the eight centroids register in the register file. The data written to these registers are points containing zeros on six dimensions and in the first dimension(the 13 LSBs of the data point) they contain decimal values from 1 to 8, being 1 written to centroid register number one and similarly to the others registers.

Then, the testbench writes via APB protocol the decimal value 1 to the register which stores the first RAM address containing data. In the same way, it writes the decimal value 8 to the register which stores the last RAM address containing data.

Next, by indirect access, the testbench writes the eight data points, which are points containing zeros on six dimensions and in the first dimension (the 13 LSBs of the data point) they contain decimal values from 1 to 8.

Finally, the testbench writes the value ‘1’ to the go register in the register file, and from then it waits for the algorithm to end, which is signaled by the rising edge of the interrupt output signal of the k means top module.

## Tests results

In this section the result of the test descripted in the above section is analyzed. As the “sanity check” test is more complex in its execution than the “do nothing” test, its analysis is more detailed, but the same could have been done for the “do nothing” test, resulting in a similar detailed analysis.

### “Sanity check” test results

In this section, an analysis of the “Sanity check” test results is made, highlighting some essetials parts of the algorithm’s implementation.

#### “Sanity check” test full simulation analysis

TBD

#### “Sanity check” test simulation parts analysis

TBD

#### “Sanity check” test numeric results analysis

As it can be seen from the Figure TBD(zoom out sanity check)The final centroids values in the registers file centroids in this test were

|  |  |
| --- | --- |
| Centroid number | Value in Hexadecimal |
| 1 | 000 0000 0000 0000 0000 e000 |
| 2 | 000 0000 0000 0000 0000 0002 |
| 3 | 000 0000 0000 0000 0000 0003 |
| 4 | 000 0000 0000 0000 0000 0004 |
| 5 | 000 0000 0000 0000 0000 0005 |
| 6 | 000 0000 0000 0000 0000 0006 |
| 7 | 000 0000 0000 0000 4400 0007 |
| 8 | 000 0000 0000 0000 0000 000e |

Table 3:Sanity check results in hexadecimal

The above results converted into the data representation chosen for this project(7 dimensions fixed point numbers, each with 13 bits ,for more details see section ‎2.6.2.3.1) are show in their real value in the table below:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Centroid number | Dim 7 | Dim 6 | Dim 5 | Dim 4 | Dim 3 | Dim 2 | Dim 1 |
| 1 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.0068 | 0.000 |
| 2 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.0020 |
| 3 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.0029 |
| 4 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.0039 |
| 5 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.0049 |
| 6 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.0059 |
| 7 | 0.000 | 0.000 | 0.000 | 0.000 | 0.0166 | 0.000 | 0.0068 |
| 8 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.0137 |

Table 4:Sanity check results values as real numbers

#### “Sanity check” test numeric results validation with MATLAB

Using MATLAB R2019b, the k means algorithm results for the same data set were calculated, and its result is shown in below:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Centroid number | Dim 7 | Dim 6 | Dim 5 | Dim 4 | Dim 3 | Dim 2 | Dim 1 |
| 1 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.0068 | 0 |
| 2 | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 3 | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 4 | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 5 | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 6 | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 7 | 0.000 | 0.000 | 0.000 | 0.000 | 0.0166 | 0.000 | 0.0068 |
| 8 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.0142 |

Table 5: Sanity check test MATLAB validation results

Note: When running in MATLAB the k means algorithm with the build in “kmeans” function, when a centroid is empty, as occurs in this case, the result centroid matrix will write “NaN” in the empty centroid result. The k means module build by this project acts differently, it simply does not change the initial centroid value in case no data points were assigned to a specific centroid.

As it can be seen from Table 4 and Table 5 the results achieved by the k means algorithm implementation as it is described through this report match the results achieved by MATLAB calculations, with an inaccuracy of only 0.001.

This inaccuracy is a consequence of the chosen architecture, specifically, due to the fact that in the division made by the implementation (in the New means calculation block of the core), the remainder of the division is neglected.

In conclusion, despite this minor inaccuracy, the results of the MATLAB simulation indeed validate the k means algorithm implementation of this project.

### “Do nothing” test results

# Bibliography

* K means algorithm – tutorial 3 of Technion winter 2018/2019 machine learning course 046195
* AMBA APB – ARM AMBA 3 APB Protocol v1.0 Specification
* AMBA APB – International Journal of Scientific Engineering and Applied Science (IJSEAS) - Volume-1, Issue-3, June 2015 ISSN: 2395-3470 www.ijseas.com 492

Design & Implementation of Advance Peripheral Bus Protocol

* FPGA Implementation of K-means Algorithm for Bioinformatics Application: An Accelerated Approach to Clustering Microarray Data
* Manpreet Manna restoring division figure at https://www.researchgate.net/figure/Flowchart-of-Restoring-Division-Algorithm\_fig4\_250612202
* Abid Rahman K answer at stackoverflow in this link https://stackoverflow.com/questions/12133810/non-restoring-division-algorithm
* https://www.quora.com/What-is-the-difference-between-restoring-and-non-restoring-divison-method
* https://en.wikipedia.org/wiki/Division\_algorithm#SRT\_division ( include SRT, Goldschmidt & Newton-Raphson division methods).