Laboratory 3

Lab Objective:

The aims of this laboratory session are to provide practice to enhance the Verilog HDL coding skills. Specifically, you are required to complete two tasks.

Task 1 - Processor implementation

(1) Build and test the working of a simple processor to obtain better understanding of more complex logic design.

(2) Perform functional simulation of the processor using the Xilinx Vivado Simulator tool used in earlier labs.

Task 2 - K-means clustering implementation

(1) Build and test a popular clustering algorithm in machine learning, k-means.
(2) Perform functional test of the k-means algorithm.

Learning Outcomes:

At the end of this lab session, you would have learned the following:

1. A deeper understanding of processor design and k-means design, together with their control flows.
2. Micro-architectural concepts like data-path, control unit, memory components, Instruction sets, etc.
3. Hands-on with more complex logic design and test benches using Verilog HDL
4. Including a definition file to keep the definitions of miscellaneous items like opcodes, state descriptions, etc.
5. Functional and timing simulation of your design using Vivado Simulator.

Introduction:

TASK 1: Processor implementation

First, we will take a look at processor description and functionality. We will also examine the instruction set, and understand the different constituent modules and control logic for our processor. The programmable algorithmic state machine (PASM) model of a computer processor could be partitioned into three parts:

1) A data path subsystem
2) A control unit subsystem (e.g. a next state generator and an output decoder).
3) A memory subsystem.

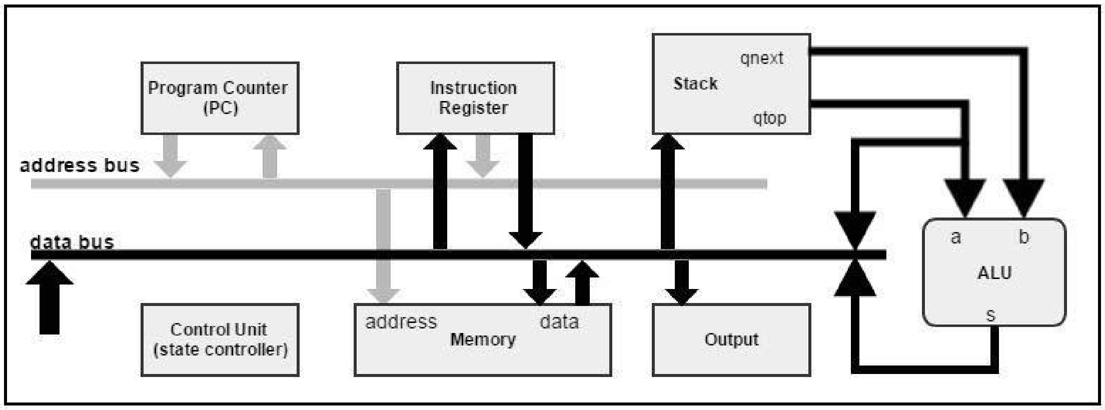


Figure 1: Block diagram of Processor architecture

Fig. 1 shows a block diagram of the processor architecture, showing the internal logic blocks and the bus connections between them. Control connections, clocks, reset lines etc., are not shown.

Firstly, we shall examine the seven modules, their interfaces and consider their functionality:

Program Counter	stores the 12-bit address pointing the instruction code in the memory
Instruction Register	16-bit register to store the instruction code to be executed
Control Unit	3-bit register that holds the current state and some other controller logic
Stack	array of registers to store intermediate values for evaluating formulas
ALU	combinational circuit to compute resulting values for specified operation
Memory	block RAM to store instruction codes of a program and values used by the program
Output Buffer	16-bit register to store the output of the program

The state machine controller (inside control unit) is shown unconnected to other units within the processor, and it is true that does not connect to the data or address buses, however it does connect widely to almost every unit and bus driver with processor.

We will examine the sequence of operations within this processor and its control aspects.

Processor Control:

On power up the processor begins in an IDLE state, in which the processor is not operating. A run signal causes the processor to begin normal operation, fetching the first instruction from memory. Once the processor enters this normal operating mode, it will execute instructions one at a time. Only the issuing of a HALT instruction (or an unrecognized instruction) from RAM will cause the processor to re-enter IDLE state. This normally indicates the end point of a program. (You must browse the code along with the description to get a better understanding. See the different connections between the various modules in the design to see the big picture.)

FETCHA is the first state encountered during normal operation. ***FETCHA*** causes the address in the program counter (pc0, initially set to zero) to be output on the address bus, abus. The ram0 module will then look up the content of that memory address, and once it has found this, to output it.

FETCHA will then automatically be followed one clock cycle later by ***FETCHB*** state, to conclude the instruction fetch process. In state ***FETCHB***, the memory content that ram0 has found will be allowed to drive dbus, and ir0 will latch this in. At this point, register ir0 will now contain the instruction machine code from the ram0 address that pc0 pointed to.

The program counter will have automatically been incremented at the transition from state ***FETCHA*** to ***FETCHB***. ***FETCHB*** is followed one clock cycle later by the first of the execution states, ***EXECA***. The execute states ***EXECA*** has the general responsibility of beginning the execution of the instruction by first performing an instruction decode function. For some instructions ***EXECA*** will need to be followed by a continuation state ***EXECB***, but at other times the processor will be ready to execute the next instruction, and thus transition back to ***FETCHA***. In cases where the instruction stored in ir0 is zero, this indicates the HALT instruction, and so the processor must transition to IDLE mode in this case (Refer to Figure 2 at the end of the manual for the state transition diagram).

Now, let's discuss the instruction set.

Processor Instructions:

Mnemonic	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Hex
HALT	0	0	0	0	X (don't care)												0XXX
PUSHI I	0	0	0	1	I (signed integer)												1000+I
PUSH A	0	0	1	0	A (unsigned integer)												2000+A
POP A	0	0	1	1	A												3000+A
JMP A	0	1	0	0	A												4000+A
JZ A	0	1	0	1	A												5000+A
JNZ A	0	1	1	0	A												6000+A
IN	1	1	0	1	X												D000
OUT	1	1	1	0	X												E000
OP *f*	1	1	1	1	X						*f*						F000+*f*

The current 16-bit instructions for the processor are summarized in the Table 1 (above), with the following meanings:

• HALT is used to discontinue processor operation. This is a legitimate event that occurs at the termination of a program, but is also a safety feature that may come into play when an unintended program branch causes the CPU to jump outside of it's program code. In this case a jump to uninitialized memory would likely result in an 'instruction' being read in with the numerical value of zero – causing the processor to halt rather than continue operation incorrectly.

• PUSHI I pushes the immediate operand into the stack. However the value of I can be a maximum of 12 bits, and yet the machine (including stack) is 16 bits wide. Thus the immediate value which is loaded to stack must be first sign extended.

• PUSHA retrieves the content of memory location A from ram0 and pushes this onto the stack.

• POP A performs the reverse operation, popping the top value from stack and storing to RAM address A.

• JMP A, JZ A and JNZ A each jump to execute the next instruction from address A either always, or only when a data item popped from the top of the stack is zero (JZ) or non-zero (JNZ) respectively. The process of jumping involves the absolute address A encoded in the machine code program being loaded into the program counter. Note that no other conditionals apart from NZ and Z are supported in the jump command, and thus the full range of traditional conditionals (such as GT, LE and so on) must be supported in another way.

• IN reads the input port and pushes the value found there onto the stack.
• OUT pops the top item from the stack and latches it into the output buffer.

• OP f is not an instruction in itself, it is a class of instructions. These cause the ALU to perform the requested function encoded in f. Since the ALU is wired to the top two stack locations, the function can use either of these. The OP f instruction class currently encodes 19 separate operations. 16 of them involve two operands (from the stack qtop and qnext outputs), and will thus pop the stack before writing the result output back into the stack. Three operations are unary – taking only the stack qtop, operating on this, and then loading back into stack. In this case no pop is required since the single value from the stack top used in the instruction will be directly overwritten by the result.

For the simple processor design we are using, a continuation state (EXECB) is only necessary for the PUSH instruction. The reason is that there is a memory lookup required as part of this instruction (just like the memory lookup for instruction loading requires two fetch states). Remember that PUSH A loads the value from memory address A into the stack. On entry to state EXECA, the instruction will already have been loaded into ir0. During EXECA, the value of memory address A is output from the instruction register to abus. The RAM module ram0 then looks up the content of this memory address, but cannot drive this value onto dbus immediately as it will take some short time to retrieve it from the memory array. Thus a second execution state EXECB exists during which RAM drives this value onto dbus and the stack is simultaneously instructed to push the current dbus content. EXECB state is always followed by a fetch of the next instruction, FETCHA (Refer to Figure 2 at the end of the manual for the state transition diagram).

The OP class of instructions are shown in Table 2, below:

Mnemonic	4	3	2	1	0	Hex	Stack top	Popped?
ADD	0	0	0	0	0	F000	next + top	Y
SUB	0	0	0	0	1	F001	next - top	Y
MUL	0	0	0	1	0	F002	next * top	Y
SHL	0	0	0	1	1	F003	next >> top	Y
SHR	0	0	1	0	0	F004	next << top	Y
BAND	0	0	1	0	1	F005	next & top	Y
BOR	0	0	1	1	0	F006	next | top	Y
BXOR	0	0	1	1	1	F007	next ^ top	Y
AND	0	1	0	0	0	F008	next && top	Y
OR	0	1	0	0	1	F009	next || top	Y
EQ	0	1	0	1	0	F00A	next == top	Y
NE	0	1	0	1	1	F00B	next != top	Y
GE	0	1	1	0	0	F00C	next >= top	Y
LE	0	1	1	0	1	F00D	next <= top	Y
GT	0	1	1	1	0	F00E	next > top	Y
LT	0	1	1	1	1	F00F	next < top	Y
NEG	1	0	0	0	0	F010	-top	N
BNOT	1	0	0	0	1	F011	~top	N
NOT	1	0	0	1	0	F012	!top	N

Programming the Processor:
Now that we have gained familiarity with the architecture and instruction set, we will learn how to program the processor stack machine. Let us take the simple example of how subtraction is implemented on the processor. We will develop a very simple program to read a value from the input port, subtract a constant from that, and then load the result into the output register. We assume that the constant is located in memory with a label cnst. The code is as follows:

```
IN
PUSH cnst
SUB
OUT
HALT
cnst: 3
```

The IN command will read something from the input port and place it onto the stack. In order to subtract a constant from that, we need to also load the constant into the stack. If this was an immediate constant we would use PUSHI, but in this case the constant resides in memory, so we need to retrieve a value from memory and push it into the stack, instead using PUSH cnst.

We then perform the subtraction operation: SUB which will pop the two input operands off the stack, perform the subtraction, and then push the result back onto the stack. So finally we can now load the result into the output register: OUT. We also need to add a location to store the constant.

Next we determine the machine code (hexadecimal) identifiers for each of the instructions in turn from the instruction set tables given. For example, looking up the IN instruction, in Table 1 we can see that it is represented by the hexadecimal value D000. The second instruction, PUSH cnst is represented by the hexadecimal value 2000 + A where A is the address at which the constant is stored. In this case we need to convert the label 'cnst' to an address – and we can do this by simply counting which address it is at.

From the listing it's the sixth line, but since the computer counts address locations starting from location zero, then the address of the constant is actually five. Thus the hexadecimal value of this instruction would become 2005. (But it is 2805 in the code below. Please check "Notes on Memory" part for the reason.) Repeating this process for the remaining instructions, we would end up with machine code as shown:

```
D000 \\ IN
2805 \\ PUSH cnst
F001 \\ SUB
E000 \\ OUT
0000 \\ HALT
0003 \\ cnst: 3
```

Examine the *ram.v* module for the instruction code already inserted beginning from the first Ram location address h'000.

Notes on Memory:

Note that we have a 12-bit address space each 16-bit wide. This means we have 4096 words each being 16-bit wide. We will explicitly put a restriction on it and divide the RAM into 2 equal halves: 1) Program memory, 2) Data memory. Hence, locations h'000 to h'7FF is categorized as program memory and locations h'800 to h'FFF is categorized as Data memory. Keep this in mind while writing your machine codes for the processor. Any overlap could lead to corruption of program.

Laboratory Exercises:

Exercise 3A:

(i) Implement the ALU (in *alu.v*) in accordance with the instruction set given.

(ii) From the given state diagram for the control unit (responsible for the processor state transitions), complete the implementation of the same in *state.v* file. Note:

  a. The reset here is asynchronous.
  b. Dotted lines indicate reset=1 and hard lines mean reset=0.

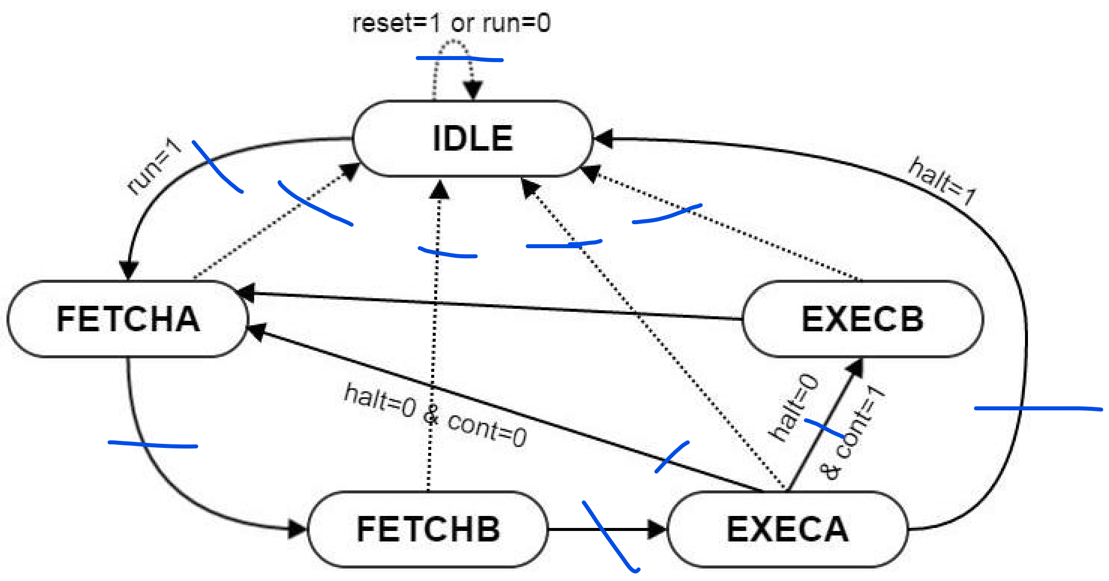


Figure 2: State diagram for next state transition

Exercise 3B:

(i) Perform behavioral simulation using Vivado Simulator for the sample code that is already in the memory (*ram.v*) and the provided test bench (*cpu_tb.v*).

(ii) Design a program to decrement a particular value 'n' obtained from the input port, 'n' times. The values n, n-1, . . ., 0 should be written, in turn, to the output buffer. Remove the previous code from 3B.(i) and modify *ram.v* with your own code and perform functional simulation using Vivado. (Hints: please take care of the exact behavior of JMP, JZ, JNZ and OP, including whether they will pop, overwrite, or shift the positions of anything in the stack. These will affect how you write the assembly code. You can do this by searching for important variables like 'pop', 'push', 'dbus2qtop' in *cpu.v*, and the variables that they are connected to in *stack.v*)

TASK 2: K-means implementation

K-means introduction:

Cluster analysis is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (based on some sorts of similarities) to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis. It has been widely used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, bioinformatics, data compression, and computer graphics, etc.

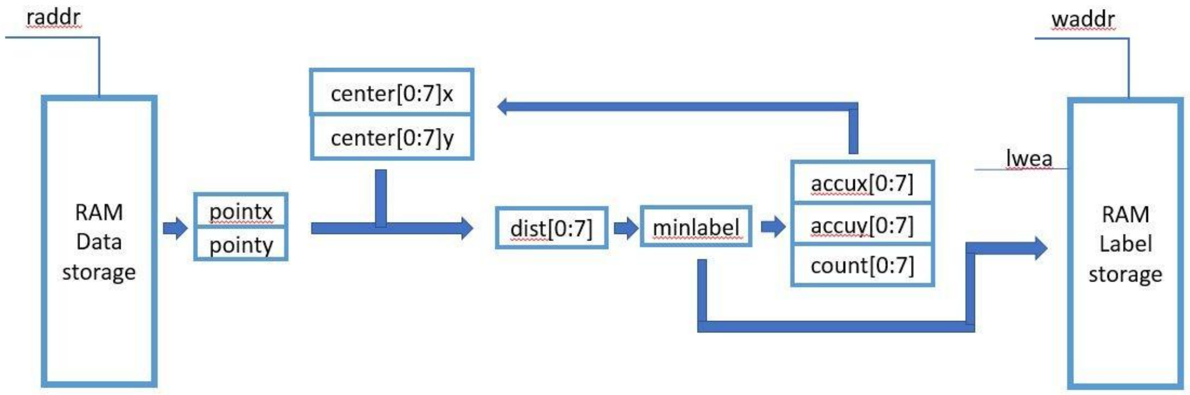
K-means algorithm is one of the most popular method in cluster analysis. K-means clustering tries to determine k centers which partition the whole sample set into k subsets. The goal is to minimize the within-cluster sum of square for the sample set.

Mathematically, given a set of observations (x1, x2, …, xn), where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k (≤ n) sets S = {S1, S2, …, Sk} and the objective is defined as:

$$\underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} \sum_{\mathbf{x}\in S_i} \|\mathbf{x}-\boldsymbol{\mu}_i\|^2 = \underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i$$

where μi is the mean of points in Si. K-means algorithm will execute several iterations to update the centers until finally the result converges (i.e., the centers become stable at some specific locations and do not change given more iterations).

K-means Implementation:
In this experiment, you need to complete a k-means algorithm with 8 centers and 1024 data points. The hardware skeleton of this implementation is given to you as below.

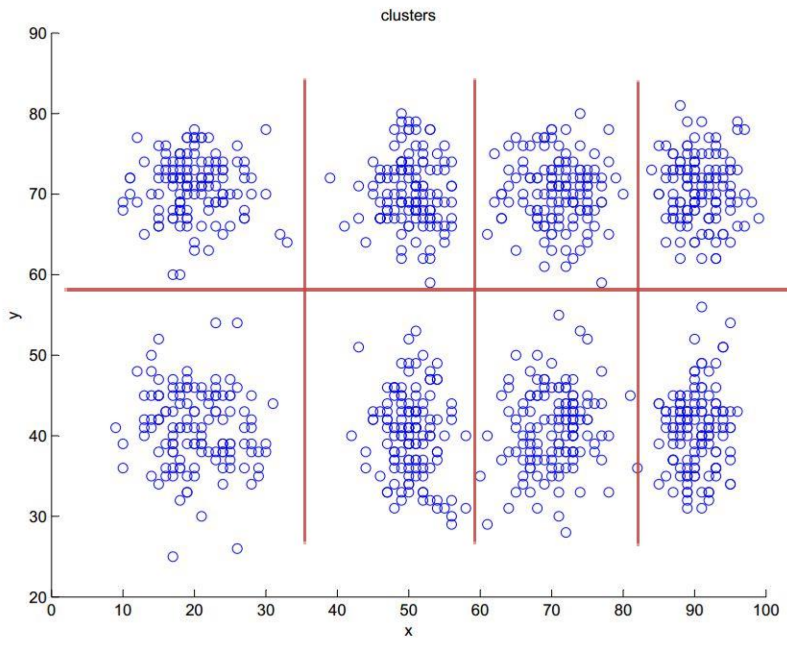


To begin with, k-means algorithm initializes k centers randomly. After that, several iterations will be executed with the following steps in each iteration.

1) One data sample is read out from a memory in each cycle. Coordinate (x,y) of this point is stored into "pointx" and "pointy" registers respectively. Also the reading address "raddr" increases.
2) This data sample will firstly go through a so called "**sample assigning phase**". The Euclidian distances of this data sample to eight centers are calculated and the eight results should be accordingly stored into "dist0" to "dist7" registers.
3) The eight distance values are then compared to find the minimum and the cluster index "minlabel". In this stage, the data point will be assigned to the center minimizing the distance between them.
4) Finally, the x value and y value of the data point are accumulated in the "accux" and "accuy" registers for the center it has been assigned to. At the same time, a "count" register updates the current number of samples for a specific cluster, by increment.
5) After all the samples are clustered, we go through a "**center updating phase**". Each of the eight centers should be updated, using the mean values of the coordinates for all the samples assigned to this center (or cluster). In the code, this operation is achieved by calculating "accux/count" and "accuy/count".

6) The updated center values are compared with the center values in the former iteration for convergence judgement; that is, if the center coordinates remain unchanged between two iterations, the results converge and the algorithm terminates.

Data points to be classified are stored in Data Storage RAM which will be initialized by *kmeans.coe* file. The data can be visualized as the following figure shows.



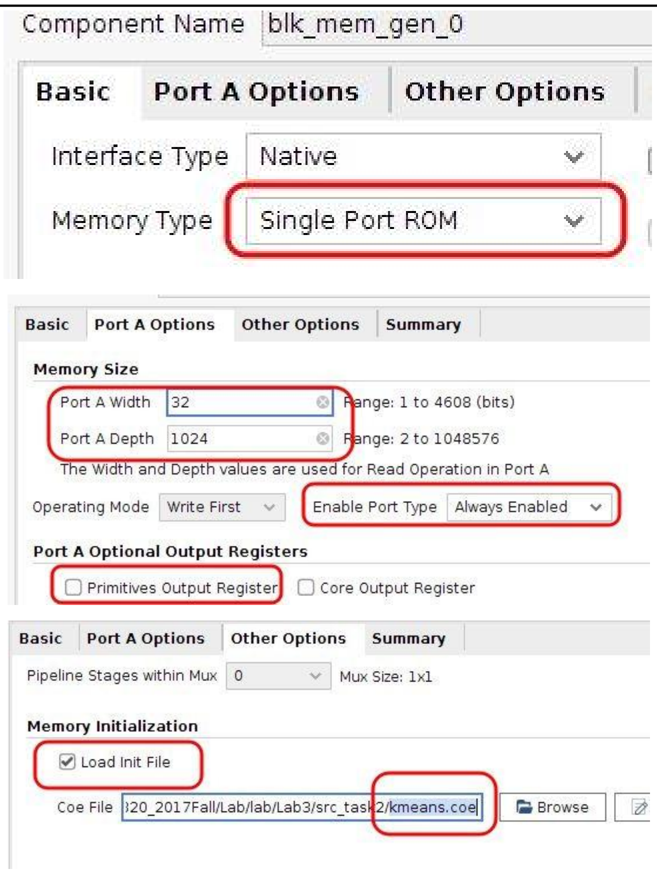
Laboratory Exercises:

Exercise 3C:

You are expected to finish several sub-functions to complete the k-means algorithm described above and report the final center positions and data labels in the simulation. To be more specific, you are required to:

(i) Fill the blank parts in *cal_dist.v, find_min.v, update_centers.v* and *xyaccumulator.v* to complete the k-means design.

(ii) Instantiate the memory module (*data_storage*) using Block RAM IP core with appropriate width and depth. Use the *kmeans.coe* to initialize it. The settings are listed below:

Instantiate the memory module (*data_storage*)


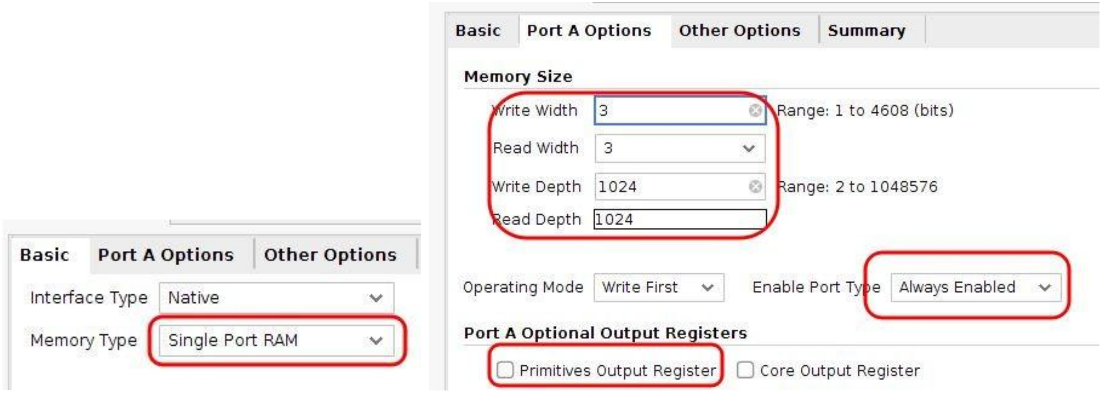
Component Name: blk_mem_gen_0

Basic — Interface Type: Native; Memory Type: Single Port ROM

Port A Options — Memory Size: Port A Width: 32 (Range: 1 to 4608 (bits)); Port A Depth: 1024 (Range: 2 to 1048576). The Width and Depth values are used for Read Operation in Port A. Operating Mode: Write First; Enable Port Type: Always Enabled. Port A Optional Output Registers: ☐ Primitives Output Register ☐ Core Output Register

Other Options — Pipeline Stages within Mux: 0; Mux Size: 1x1. Memory Initialization: ☑ Load Init File; Coe File: }20_2017Fall/Lab/lab/Lab3/src_task2/kmeans.coe

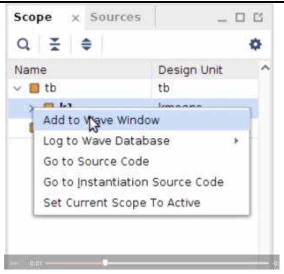
(iii) Instantiate the memory module (*label_storage*) using Block RAM IP core with appropriate width and depth. The settings are listed below:

Instantiate the memory module (*data_storage*)


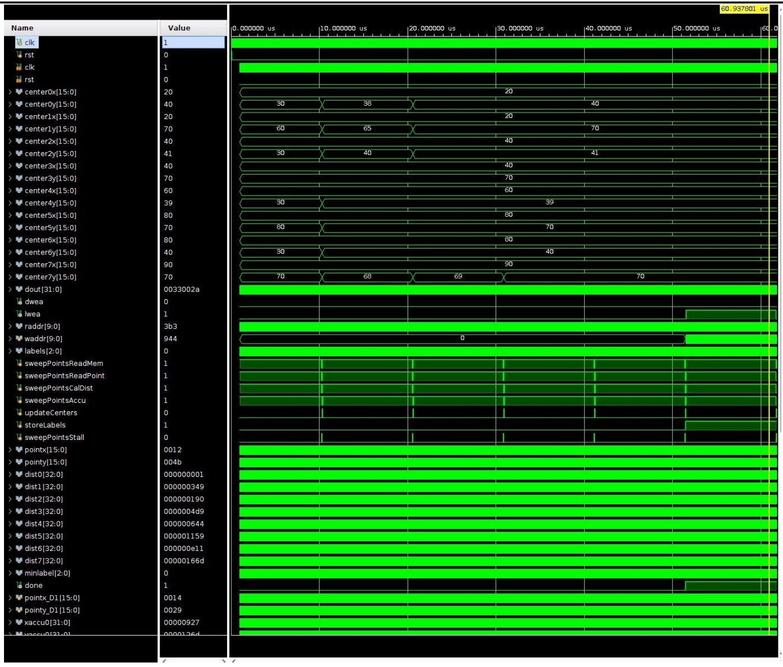
Basic — Interface Type: Native; Memory Type: Single Port RAM

Port A Options — Memory Size: Write Width: 3 (Range: 1 to 4608 (bits)); Read Width: 3; Write Depth: 1024 (Range: 2 to 1048576); Read Depth: 1024. Operating Mode: Write First; Enable Port Type: Always Enabled. Port A Optional Output Registers: ☐ Primitives Output Register ☐ Core Output Register *(iv)* Perform behavioral simulation using Vivado Simulator for the complete k-means design. Use the provided testbench, add "k1 (kmeans.v)" in Scope tab into wave window and click "Run All (F3)" in the simulation. **Remember that don't include testbench in your design source in Vivado**. <u>Include two screenshots in your report. One should demonstrate the overall simulation waveform in kmeans.v and the other one should zoom in to show the output labels of first 10 points.</u>

Add k1 module into your wave window

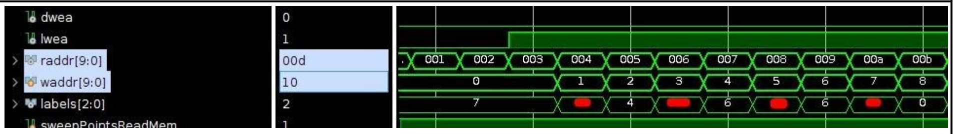


Example of Simulation Screenshot 1



Name	Value
clk	1
rst	0
clk	1
rst	0
center0x[15:0]	20
center0y[15:0]	40
center1x[15:0]	20
center1y[15:0]	70
center2x[15:0]	40
center2y[15:0]	41
center3x[15:0]	40
center3y[15:0]	70
center4x[15:0]	60
center4y[15:0]	39
center5x[15:0]	80
center5y[15:0]	70
center6x[15:0]	80
center6y[15:0]	40
center7x[15:0]	90
center7y[15:0]	70
dout[31:0]	0033002a
dwea	0
lwea	1
raddr[9:0]	3b3
waddr[9:0]	944
labels[2:0]	0
sweepPointsReadMem	1
sweepPointsReadPoint	1
sweepPointsCalDist	1
sweepPointsAccu	1
updateCenters	0
storeLabels	1
sweepPointsStall	0
pointx[15:0]	0012
pointy[15:0]	004b
dist0[32:0]	000000001
dist1[32:0]	000000349
dist2[32:0]	000000190
dist3[32:0]	0000004d9
dist4[32:0]	000000644
dist5[32:0]	000001159
dist6[32:0]	000000e11
dist7[32:0]	00000166d
minlabel[2:0]	0
done	1
pointx_D1[15:0]	0014
pointy_D1[15:0]	0029
xaccu0[31:0]	00000927

Example of Simulation Screenshot 2


dwea 0
lwea 1
raddr[9:0] 00d
waddr[9:0] 10
labels[2:0] 2
sweepPointsReadMem 1
001 002 003 004 005 006 007 008 009 00a 00b
0 1 2 3 4 5 6 7 8
7 4 6 6 0