

# **MEL G421- CAD FOR IC DESIGN**

## **PROJECT REPORT**

**HARDWARE-SOFTWARE CO-DESIGN OF K MEANS  
CLUSTERING ALGORITHM ON FPGA**

**&**

**MIN ARCHITECTURE DESIGN - CISC PROCESSOR**



**Submitted By**

**NIKHIL GUPTA 2022H123P**

**AMURT PRAKASH 2022H123P**

**AATIB MOHAMMAD 2022H1230239P**

**AVADH RAJESH HARKISHANKA 2018HS230322P**

**Submitted To**

**Dr. Abhijit Asati**

**Electrical and Electronics Department, BITS Pilani (Pilani Campus)**

## **A. Hardware Software Co-Design of K- Means Clustering Algorithm on FPGA**

### **Implementation of K-Means algorithm on hardware:**

- K-Means algorithm
- PS Part of Zedboard
- PL Part of Zedboard
- Memory in Zedboard
- PS-PL communication
- Design and datasheet
- Design 1
- Design 2
- Design 3
- Comparison of the three designs
- Conclusion

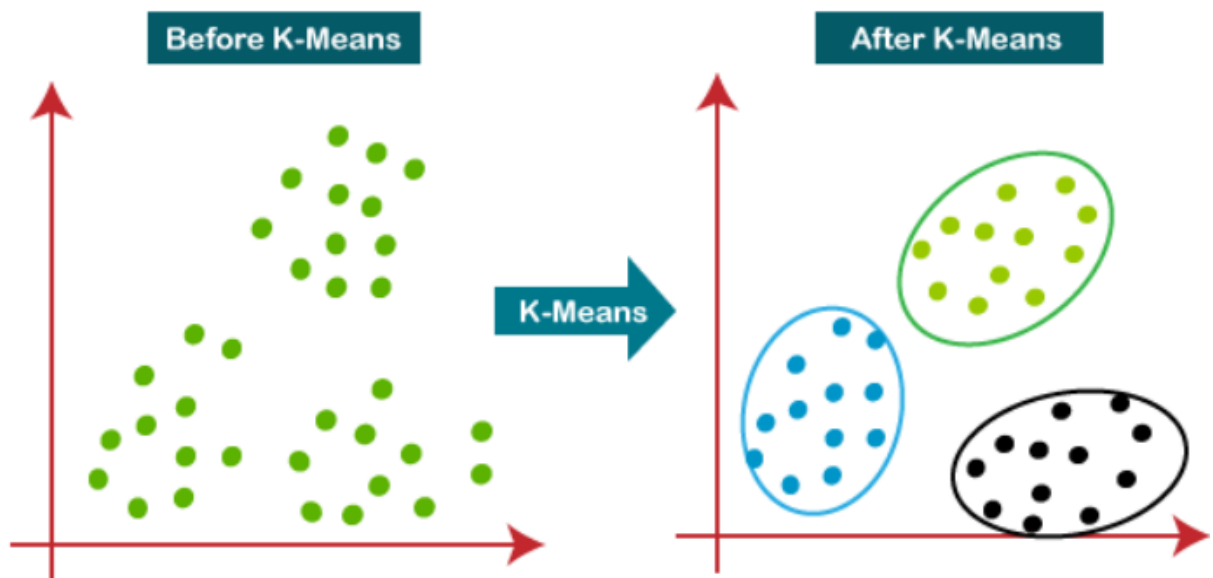
## K-Means algorithm:

K-means clustering is a type of unsupervised machine learning algorithm used to partition a set of data points into K clusters based on their similarity. It is a popular clustering algorithm because it is easy to implement and computationally efficient.

The algorithm works by randomly selecting K cluster centers and assigning each data point to its nearest cluster center based on the Euclidean distance between the data point and the cluster center. The algorithm then recalculates the cluster centers based on the mean of all data points assigned to each cluster. The process of assigning data points to clusters and recalculating the cluster centers is repeated until convergence criteria are met.

K-means clustering can be used for various applications, such as:

1. Customer segmentation: grouping customers based on their purchase behavior, demographics, and other characteristics to better understand their needs and preferences.
2. Image segmentation: dividing an image into distinct regions or objects to simplify image analysis and computer vision tasks.
3. Anomaly detection: identifying outliers or anomalies in a dataset that do not conform to the expected patterns.
4. Recommender systems: suggesting similar products or services based on a customer's purchase history or behavior.
5. Market research: segmenting markets based on customer preferences, demographics, and other factors to help businesses target their marketing efforts more effectively.



In summary, K-means clustering is a versatile algorithm that can be used for a wide range of applications to group data points into clusters based on their similarity, and it can provide valuable insights and information to businesses, researchers, and other data-driven fields.

## **PS Part of Zedboard:**

The PS (Processing System) part of Zedboard is a high-performance ARM processor that is integrated with programmable logic. It includes a dual-core ARM Cortex-A9 MPCore processor with a NEON media processing engine, a DDR3 memory controller, an AMBA 4 AXI interconnect, and various other peripherals and interfaces.

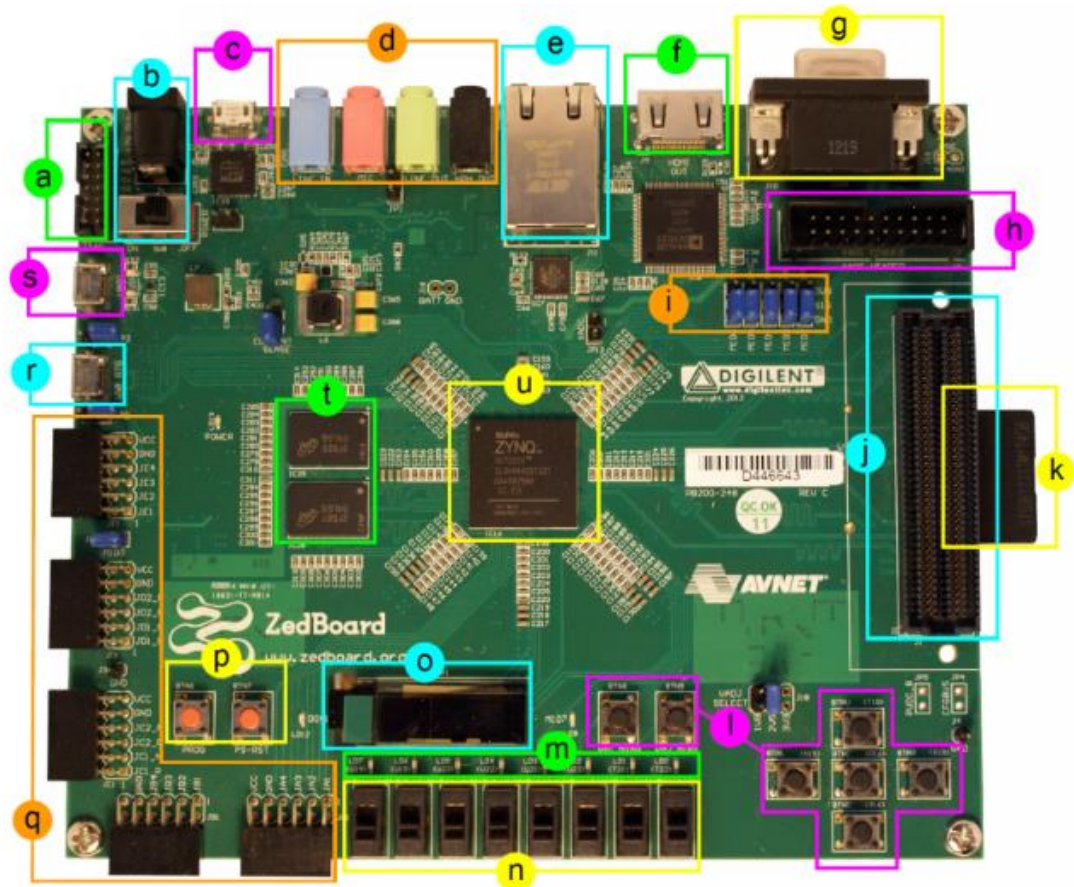
The PS part of the Zedboard is responsible for running the operating system and software applications, while the programmable logic in the PL (Programmable Logic) part of the board is responsible for implementing custom hardware logic and interfacing with external devices.

The PS part of the Zedboard supports a wide range of interfaces and protocols, including:

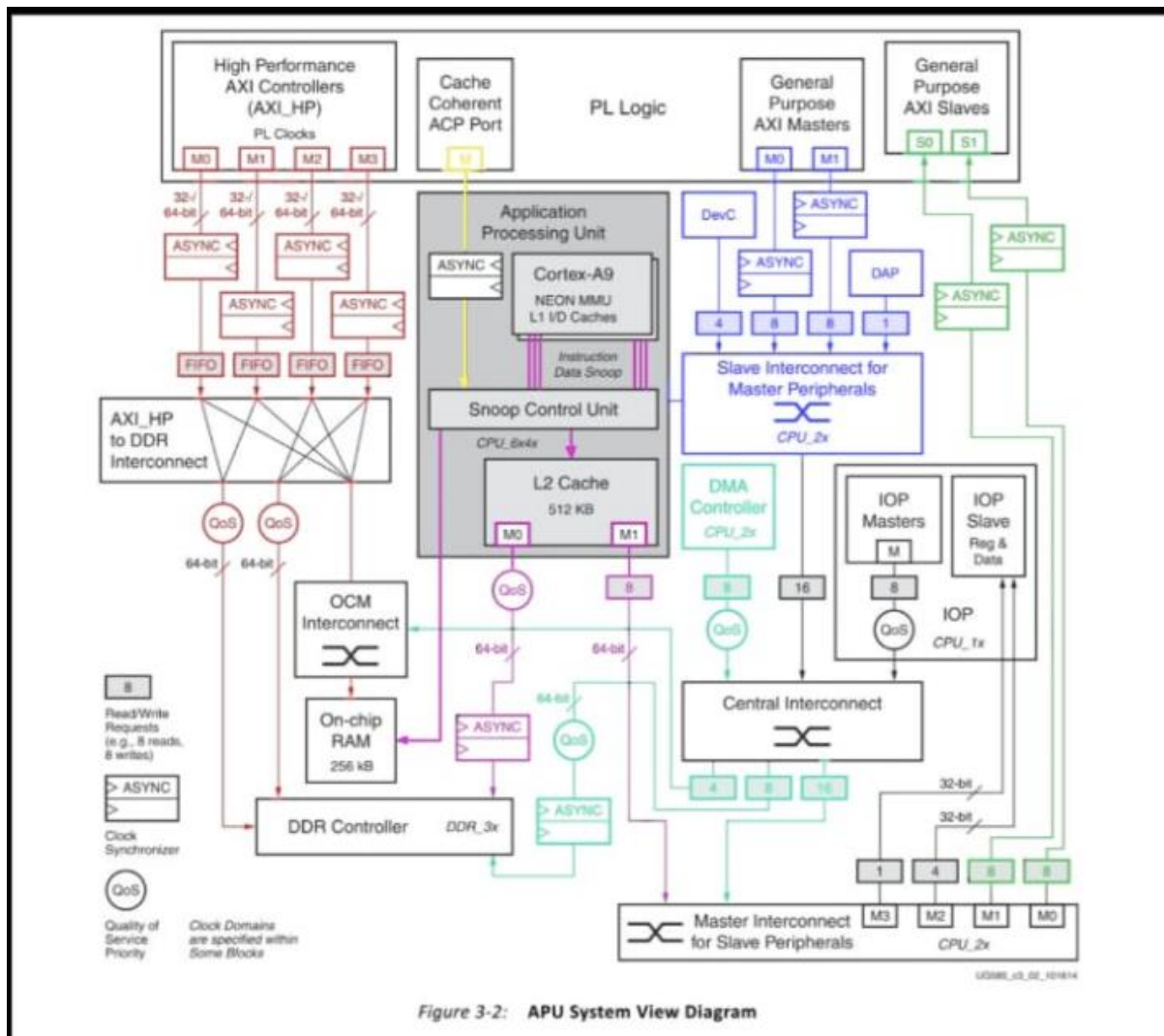
1. Ethernet: The board has a 10/100/1000 Ethernet port that can be used for networking and communication with other devices.
2. USB: The board has a USB 2.0 OTG port that can be used to connect USB devices such as keyboards, mice, and storage devices.
3. SD card: The board has a microSD card slot that can be used for storage and data transfer.
4. UART: The board has a UART interface that can be used for serial communication with other devices.
5. SPI: The board has a SPI interface that can be used for low-speed data transfer with other devices.
6. I2C: The board has an I2C interface that can be used for communication with sensors and other devices.

The PS part of the Zedboard can be programmed using a variety of development tools, including Xilinx Vitis, and supports a range of operating systems, including Linux.

The PS part of the Zedboard is a powerful ARM processor that provides a range of interfaces and peripherals for running software applications and communicating with other devices, while the programmable logic in the PL part of the board provides flexibility and customizability for implementing hardware logic and interfacing with external devices.



- |                                 |                                |                                    |
|---------------------------------|--------------------------------|------------------------------------|
| <b>a</b> Xilinx JTAG connector  | <b>h</b> XADC header port      | <b>o</b> OLED display              |
| <b>b</b> Power input and switch | <b>i</b> Configuration jumpers | <b>p</b> Prog & reset push buttons |
| <b>c</b> USB-JTAG (programming) | <b>j</b> FMC connector         | <b>q</b> 5 x Pmod connector ports  |
| <b>d</b> Audio ports            | <b>k</b> SD card (underside)   | <b>r</b> USB-OTG peripheral port   |
| <b>e</b> Ethernet port          | <b>l</b> User push buttons     | <b>s</b> USB-UART port             |
| <b>f</b> HDMI port (output)     | <b>m</b> LEDs                  | <b>t</b> DDR3 memory               |
| <b>g</b> VGA port               | <b>n</b> Switches              | <b>u</b> Zynq device (+ heatsink)  |



## PL Part of Zedboard:

The PL (Programmable Logic) part of the Zedboard is an FPGA (Field Programmable Gate Array) that can be programmed to implement custom hardware designs. The PL part of the board is connected to the PS (Processing System) part of the board through the AXI (Advanced eXtensible Interface) bus, which allows for data transfer between the PL and PS parts.

The PL part of the Zedboard can be programmed using tools such as Vivado, which allows designers to create custom digital circuits using HDL (Hardware Description Language) such as Verilog or VHDL. The PL part of the board contains a large number of programmable logic cells, which can be configured to implement logic gates, adders, multipliers, and other digital circuit elements. In addition, the PL part of the board contains dedicated hardware resources such as DSP (Digital Signal Processing) blocks, memory blocks, and I/O interfaces.

The PL part of the Zedboard can be used to implement custom hardware designs such as digital signal processing algorithms, image processing pipelines, and high-speed communication interfaces. In addition, the PL part of the board can be used to accelerate computationally intensive algorithms by offloading the processing from the PS part of the board to the FPGA.

Overall, the PL part of the Zedboard provides a flexible and programmable hardware platform that can be used to implement custom digital circuits and accelerate computationally intensive applications.

### **Memory on Zedboard:**

The Zedboard has several types of memory that can be used for storing data and code:

1. **DDR3 memory:** The Zedboard has a 1GB DDR3 memory that is used for storing program instructions and data. The DDR3 memory is connected to the PS (Processing System) part of the board, and can be accessed using the AMBA AXI bus.
2. **QSPI Flash memory:** The Zedboard has a 128MB QSPI Flash memory that is used for storing boot code and configuration data. The QSPI Flash memory is connected to the PS part of the board, and can be accessed using the Quad SPI (QSPI) interface.
3. **SD card:** The Zedboard has a microSD card slot that can be used for storing data and code. The SD card is connected to the PS part of the board, and can be accessed using the SDIO interface.
4. **On-board RAM:** The Zedboard has 256MB of on-board RAM that can be used for storing data and code. The on-board RAM is connected to the PL (Programmable Logic) part of the board, and can be accessed using the AXI bus.
5. **DDR memory on the PL side:** The PL part of the Zedboard can be connected to external DDR memory chips for additional storage.

In addition to these memory types, the Zedboard also supports virtual memory through its MMU (Memory Management Unit), which allows the processor to access more memory than is physically available by mapping virtual addresses to physical memory locations.

Overall, the Zedboard provides a variety of memory options that can be used for storing data and code, and the choice of memory type will depend on the specific requirements of the application being developed.

### **PS-PL communication:**

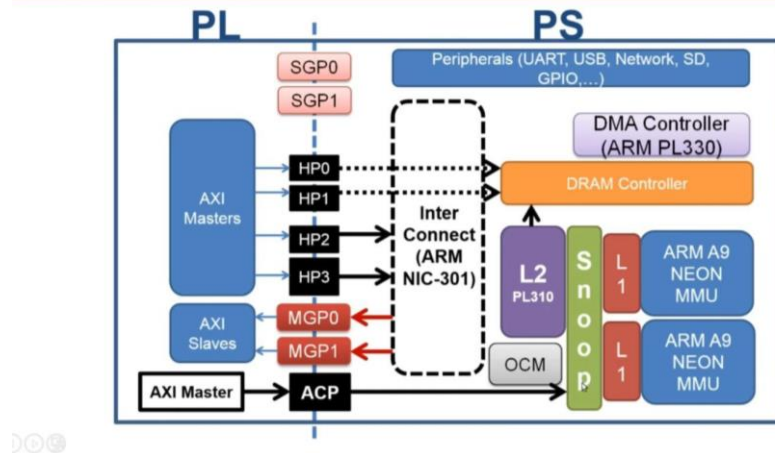
The Zedboard consists of two main parts: the PS (Processing System) and the PL (Programmable Logic) part. The PS is a hard processor system that runs software, while the PL is an FPGA that can be programmed to implement custom digital circuits. The two parts communicate through the AXI (Advanced eXtensible Interface) bus, which allows for data transfer between the PS and the PL.

The AXI bus provides a high-speed, low-latency interface between the PS and PL parts of the Zedboard. The AXI bus consists of multiple channels for data transfer, including data, address, and control channels. The AXI protocol is a standard interface protocol that defines how data is transferred between different IP blocks in a system-on-chip (SoC) design.

To facilitate communication between the PS and PL parts of the Zedboard, the Xilinx Vitis software development environment provides an interface for creating custom IP (Intellectual Property) blocks that can be integrated into the FPGA fabric. These IP blocks can be designed using HDL (Hardware Description Language) such as Verilog or VHDL and can be accessed from software running on the PS.



## Xilinx ZYNQ Architecture



The PS can communicate with the PL by writing data to and reading data from memory-mapped registers in the IP blocks. The PL can also communicate with the PS by generating interrupts or writing data to shared memory regions. In addition, the PS and PL can communicate through DMA (Direct Memory Access) transfers, which allow for high-speed data transfer between memory locations in the PS and PL.

Overall, the PS and PL parts of the Zedboard communicate through the AXI bus, which provides a flexible and high-speed interface for data transfer between the two parts. This allows for efficient integration of custom hardware designs into software running on the PS, enabling accelerated processing of computationally intensive tasks.

### Design and datasheet:

A brief explanation of the k-means algorithm based on the implementation :

Initialize the cluster centroids: Select an initial set of cluster centroids. In our case, we have selected 4 points A, B, C, and D.

1. Calculate the distance between each data point and each cluster centroid: In your implementation, this step is performed in the PS part, where the distances between the cluster centroids and all data points are calculated.
2. Assign each data point to the nearest centroid: In your implementation, this step is performed in the PL part, where the distances between each data point and each centroid are compared, and each data point is assigned to the nearest centroid.
3. Calculate new cluster centroids: After assigning each data point to its nearest centroid, calculate the new cluster centroids based on the mean of the data points assigned to each centroid. In your implementation, this step is also performed in the PS part.
4. Repeat steps 2-4 until the cluster centroids converge: Calculate the distances between each data point and the new centroids and repeat the process of assigning each data point to the nearest centroid and calculating new centroids until the centroids no longer change significantly or a maximum number of iterations is reached.



**Datasheet:**

- Data is sent from PS to Bram which will be 8 values of distance.
- Points taken for clustering are:-  $a=(1,1)$ ;  $b=(2,6)$ ;  $c=(2,3)$ ;  $d=(4,5)$ ;
- Initial clustering taken as (0,0,1,1) or (a and b) & (c and d)
- Therefore initial centroid values:-  $c1=(1.5,3.5)$ ,  $c2=(3,4)$ .
- Distance PL is supposed to calculate during 1<sup>st</sup> iteration:-  
 $ac1=2.5495$ ,  $ac2=3.6056$ ,  $bc1=2.549$ ,  $bc2=2.2361$ ,  $cc1=0.7071$ ,  $cc2=1.414$ ,  $dc1=2.9155$ ,  $dc2=1.414$
- Data which we should get from BRAM (written by PL) would be (0,1,0,1)
- So, cluster regrouped after 1<sup>st</sup> iteration (a and c) & (b and d)
- For iteration 2 we will get the new centre as  $c1=(1.5,2)$ ,  $c2=(3,5.5)$ .
- Distance PL is supposed to calculate during 2<sup>nd</sup> iteration:-  
 $ac1=1.118$ ,  $ac2=4.924$ ,  $bc1=1.118$ ,  $bc2=2.6926$ ,  $cc1=3.905$ ,  $cc2=1.118$ ,  $dc1=4.0311$ ,  $dc2=1.118$
- Data which we should get from BRAM would be (0,1,0,1)
- After 2<sup>nd</sup> iteration the error function will become zero and iteration will be completed.  
So, expected result will be:-

Cluster of point a=1

Cluster of point b=2

Cluster of point c=1

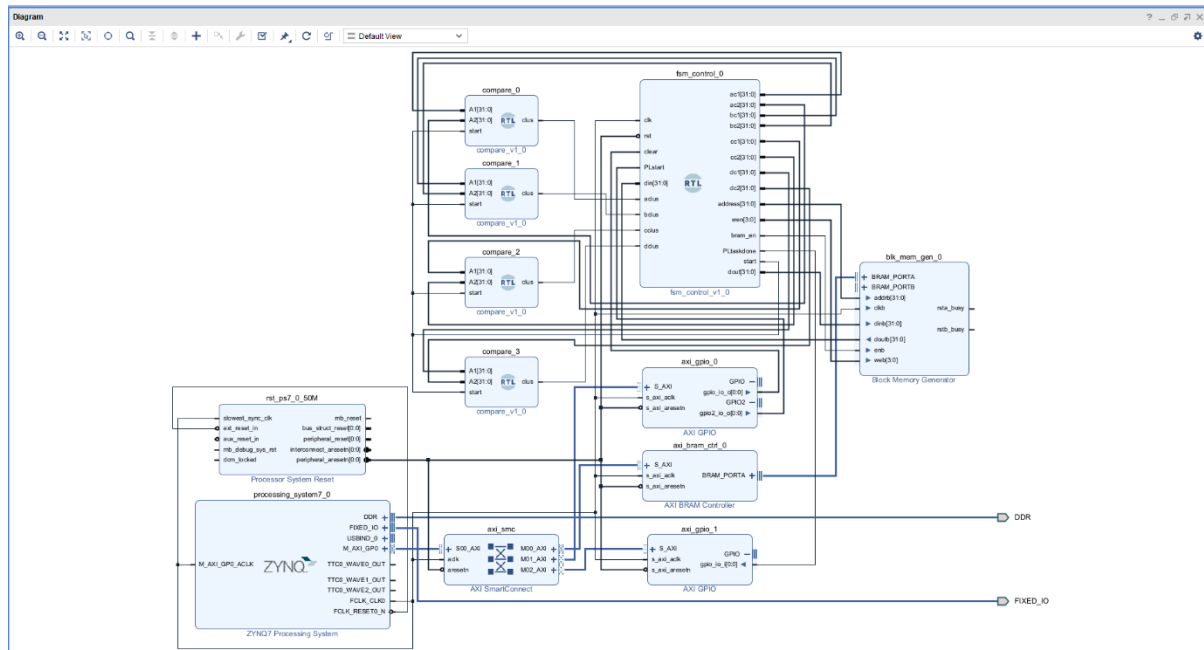
Cluster of point d=2

**Design 1:** In this design, a memory is used to store all the distance data, and then the data is sent in parallel to the comparison blocks before being written to BRAM.

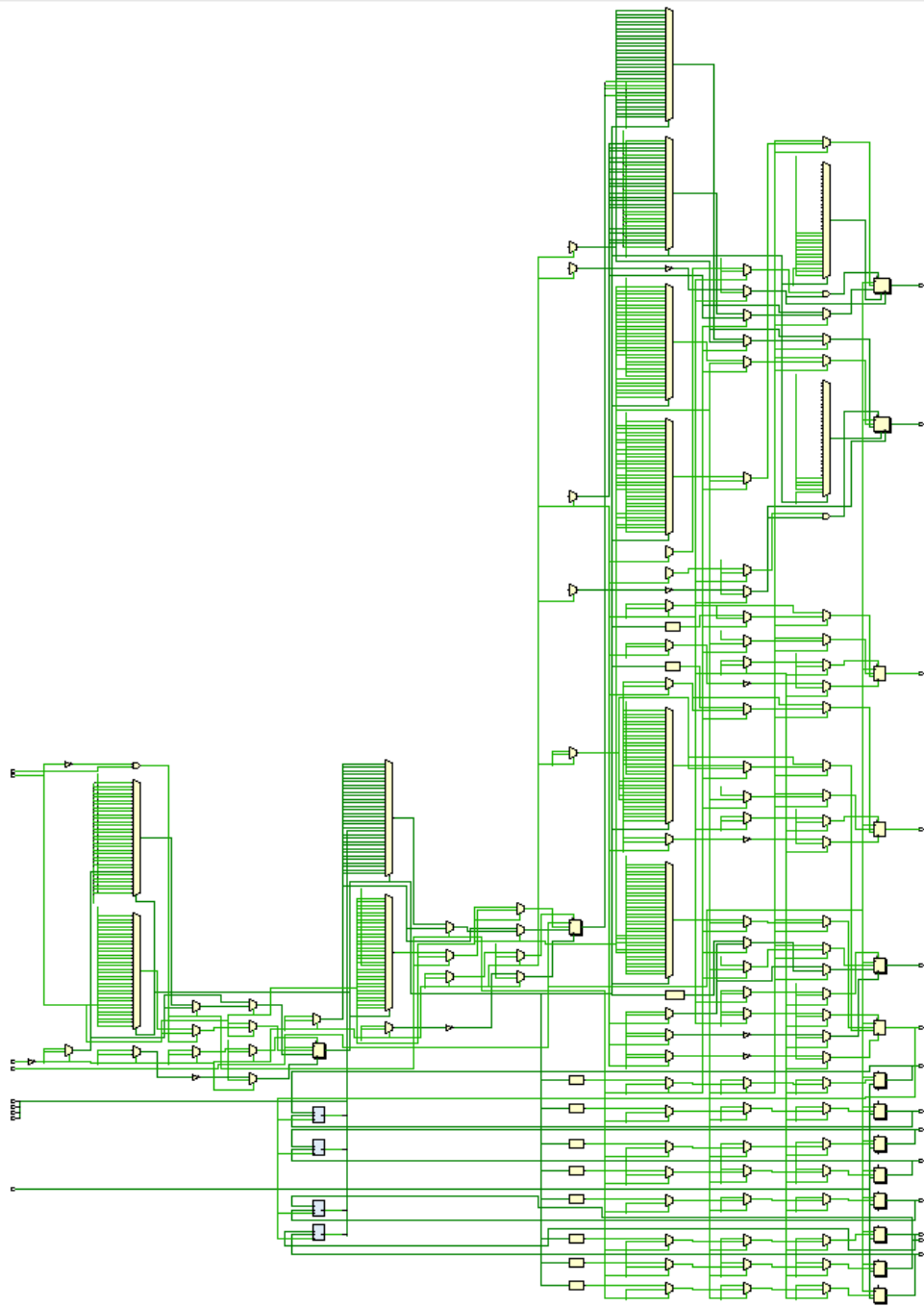
1. Calculate all distances and store them in memory: The distances between each data point and each cluster centroid are calculated, and the results are stored in memory.
2. Send the distance data to the comparison blocks: The distance data is sent in parallel to the comparison blocks, which are used to compare the distances between each data point and each centroid.
3. Compare distances and write to BRAM: The comparison blocks receive the distance data and compare the distances between each data point and each centroid. The comparison block outputs the index of the nearest centroid, which is then written to BRAM.
4. Read comparison data from BRAM: After the comparison blocks have finished comparing distances and writing the results to BRAM, the PS reads the comparison data from the BRAM.
5. Calculate new cluster centroids based on the comparison data: The PS reads the comparison data from the BRAM and calculates the new cluster centroids based on the data.
6. Repeat steps 1-5 until convergence: The process of calculating distances, comparing distances, writing results to BRAM, reading comparison data from BRAM, and calculating new cluster centroids is repeated until the cluster centroids converge.

In this approach, the use of a memory to store all the distance data and the use of parallel comparison blocks can help speed up the processing time, as multiple distances can be

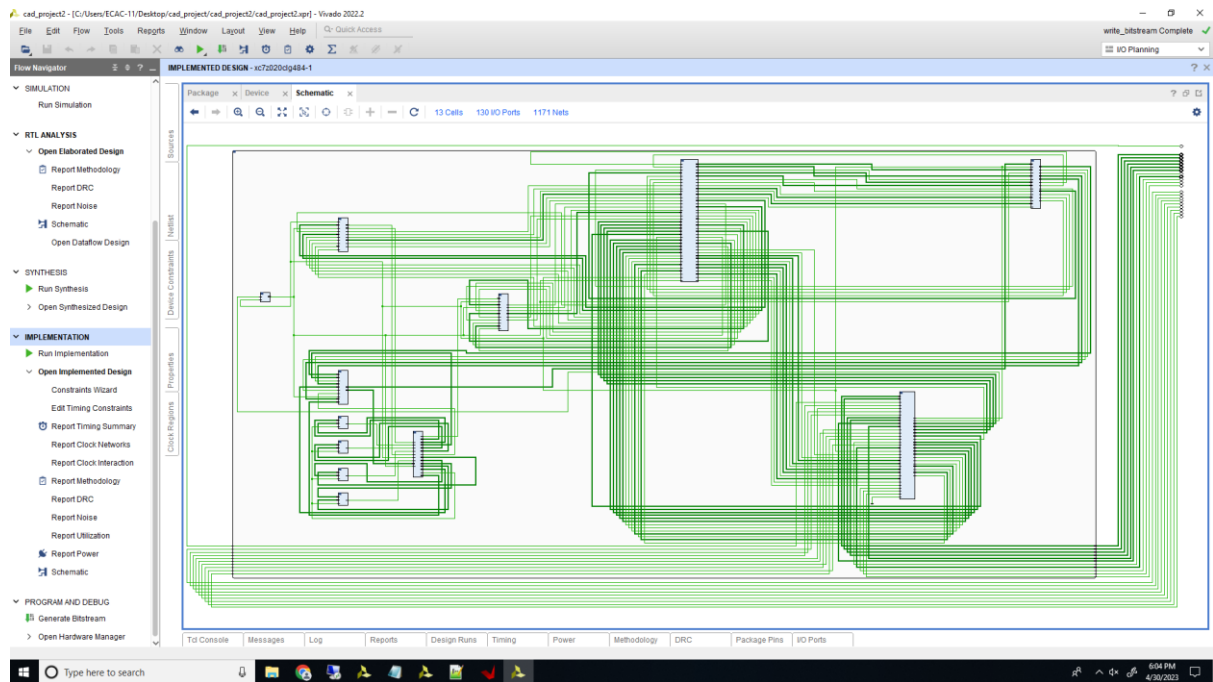
compared simultaneously. However, this approach may require more hardware resources compared to the approach of calculating and comparing distances sequentially.



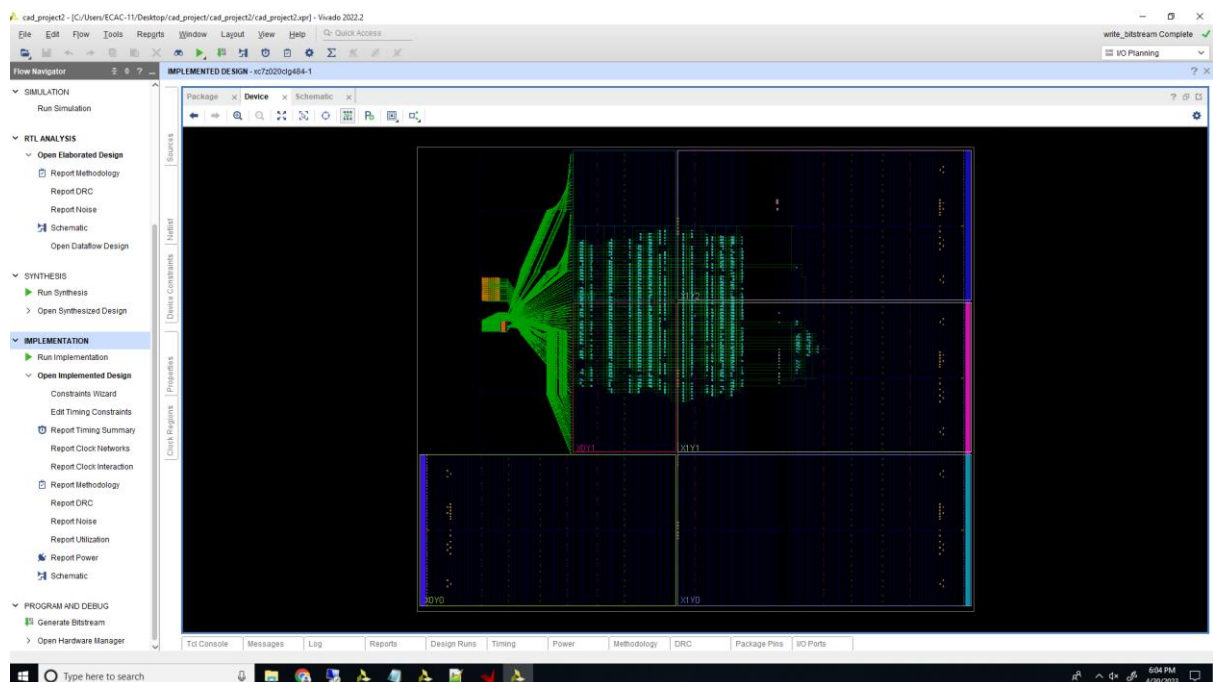
### Block Diagram



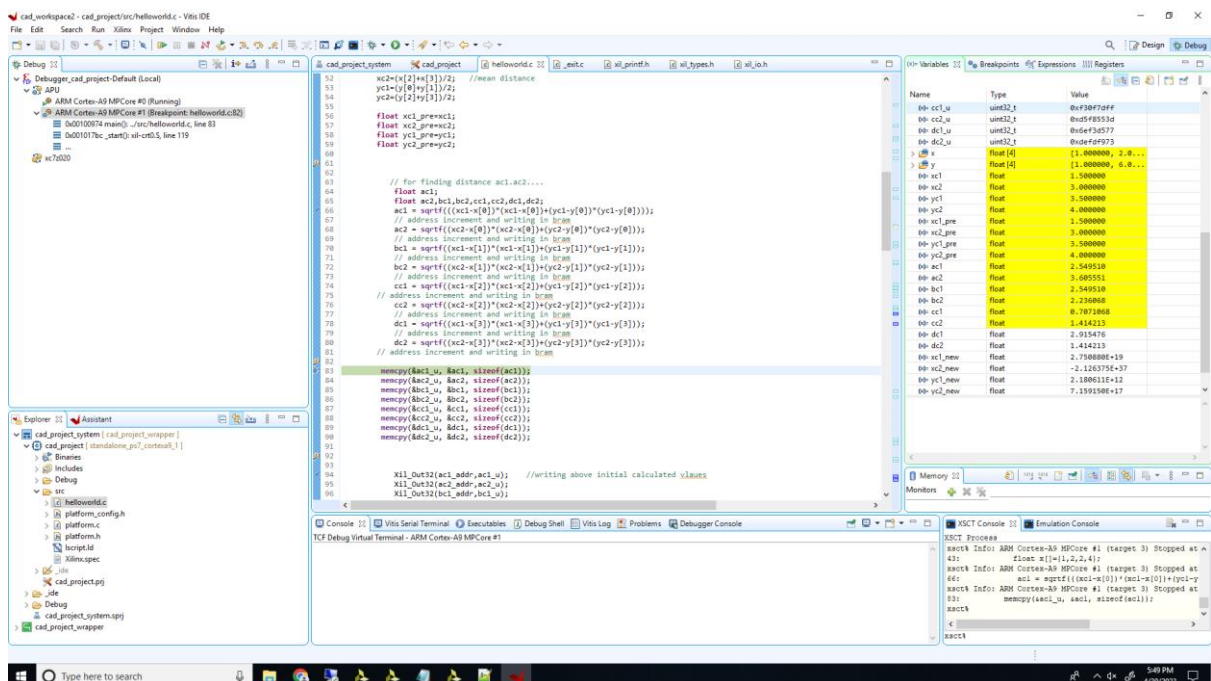
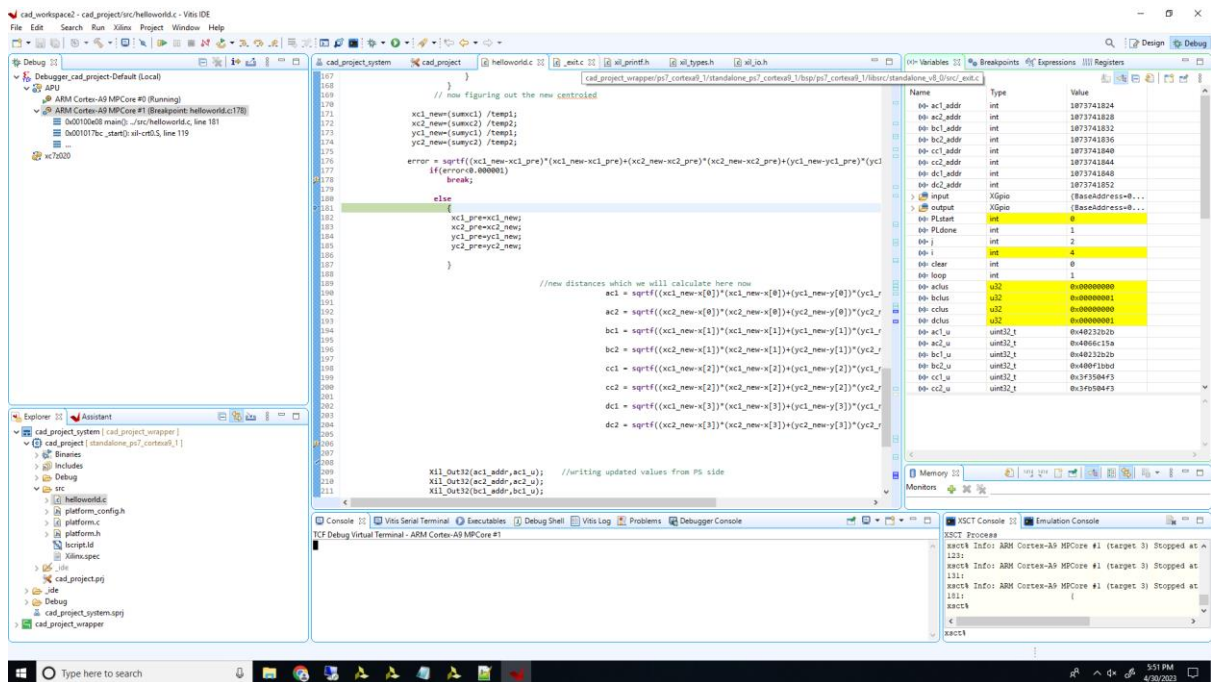
PL part

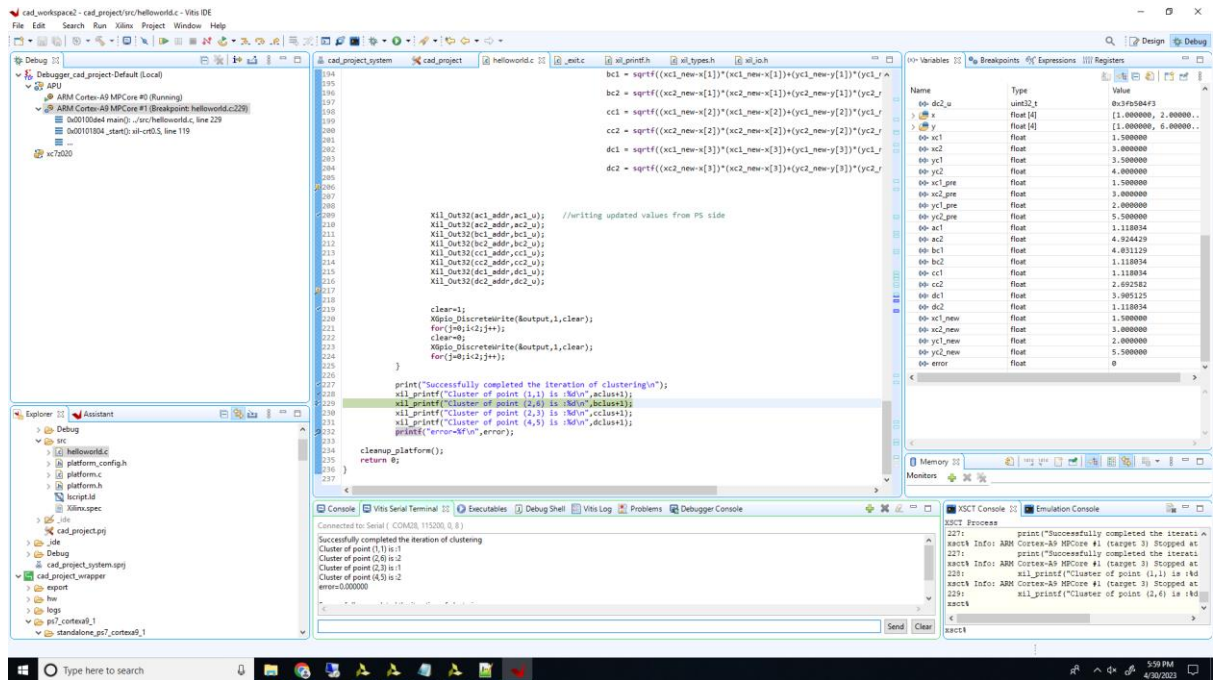


## Implemented Design Schematic



## Device Mapping



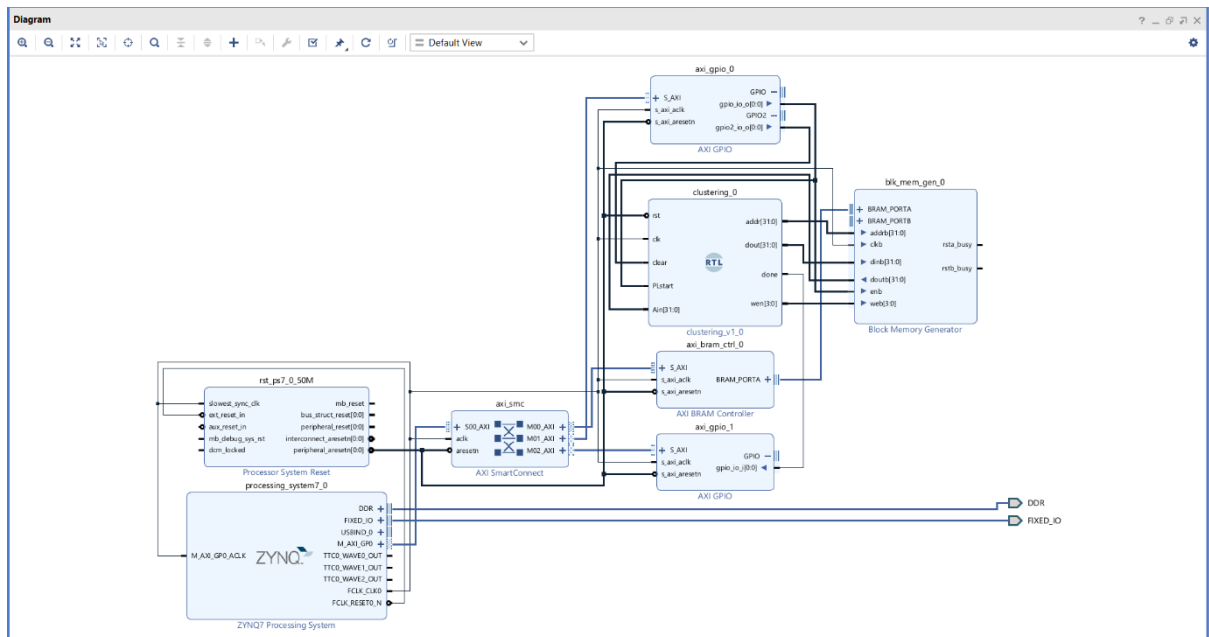


## Results

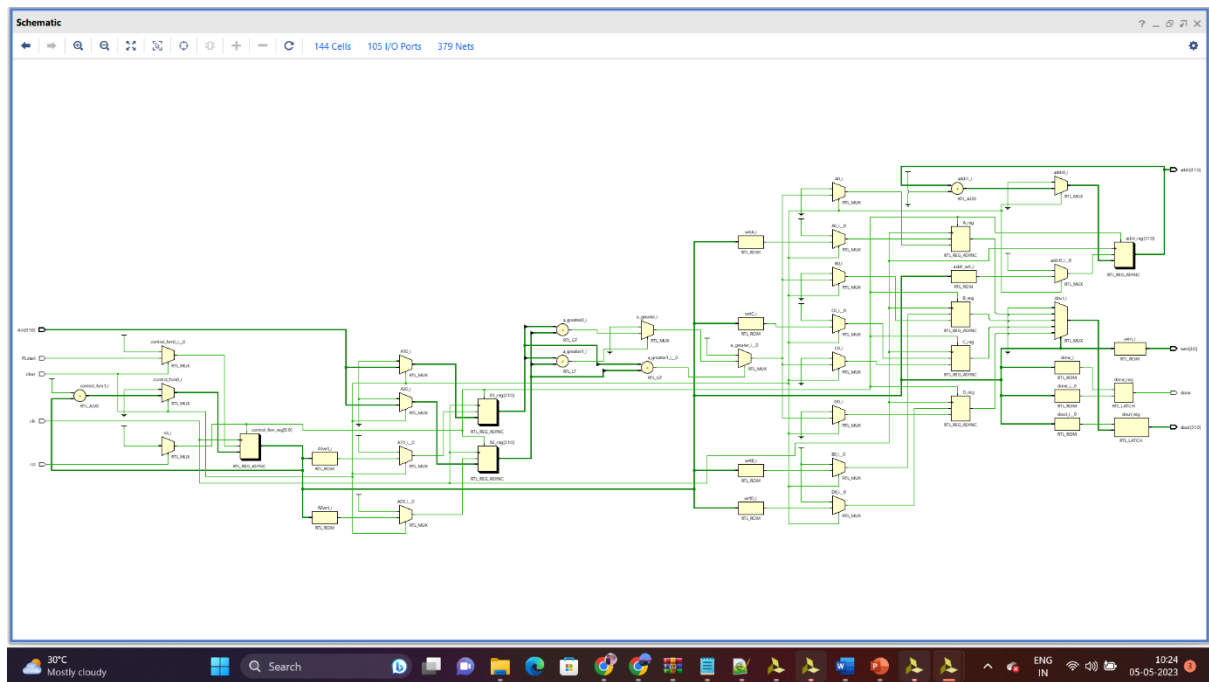
**Design 2:** We have implemented the comparison and fetching of data simultaneously in a single block to reduce the hardware resources. Here's how the process works:

1. Calculate the distances and fetch the data: In this design, the block calculates the distance between each data point and each centroid and fetches the distance data at the same time. This is done using a single block, which reduces the number of hardware resources required.
2. Compare distances and write to BRAM: After the distance data is fetched, the block compares the distances between each data point and each centroid. The block outputs the index of the nearest centroid, which is then written to BRAM.
3. Read comparison data from BRAM: After the comparison block has finished comparing distances and writing the results to BRAM, the PS reads the comparison data from the BRAM.
4. Calculate new cluster centroids based on the comparison data: The PS reads the comparison data from the BRAM and calculates the new cluster centroids based on the data.
5. Repeat steps 1-4 until convergence: The process of calculating distances, comparing distances, writing results to BRAM, reading comparison data from BRAM, and calculating new cluster centroids is repeated until the cluster centroids converge.

By combining the distance calculation and data fetching into a single block, we can reduce the number of hardware resources required, which can result in a more efficient implementation. However, this approach may also have some limitations, such as a reduced ability to parallelize the computation and the possibility of increased data transfer overhead.

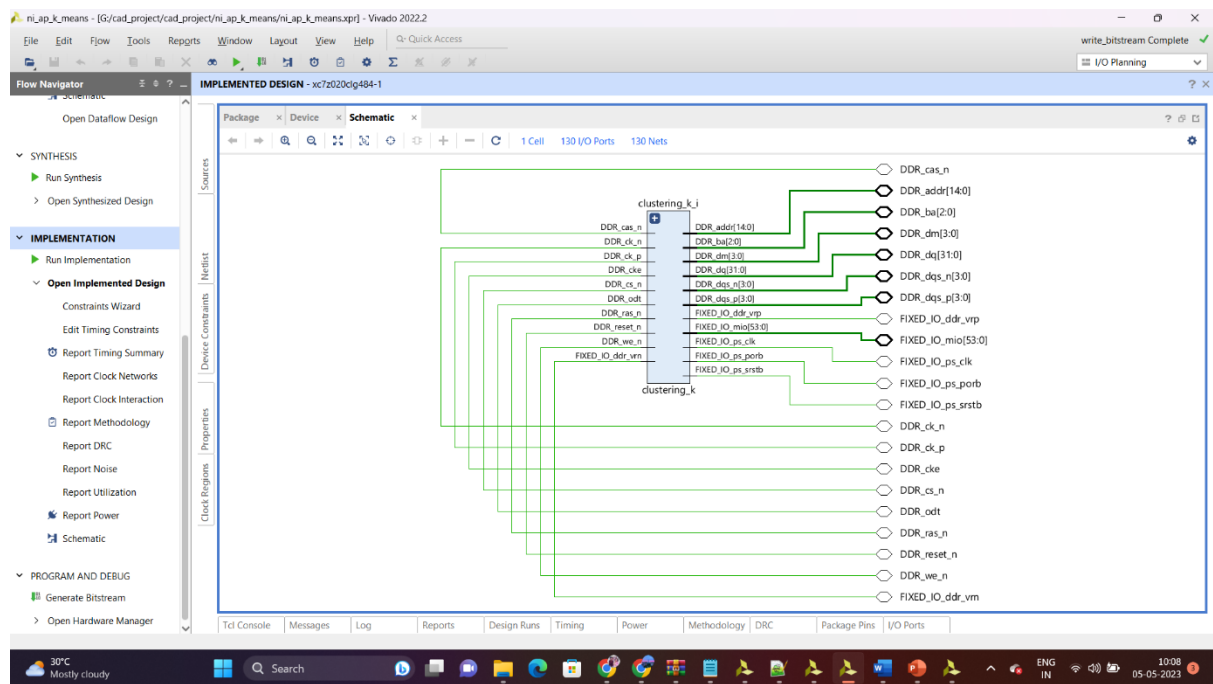


## Block design

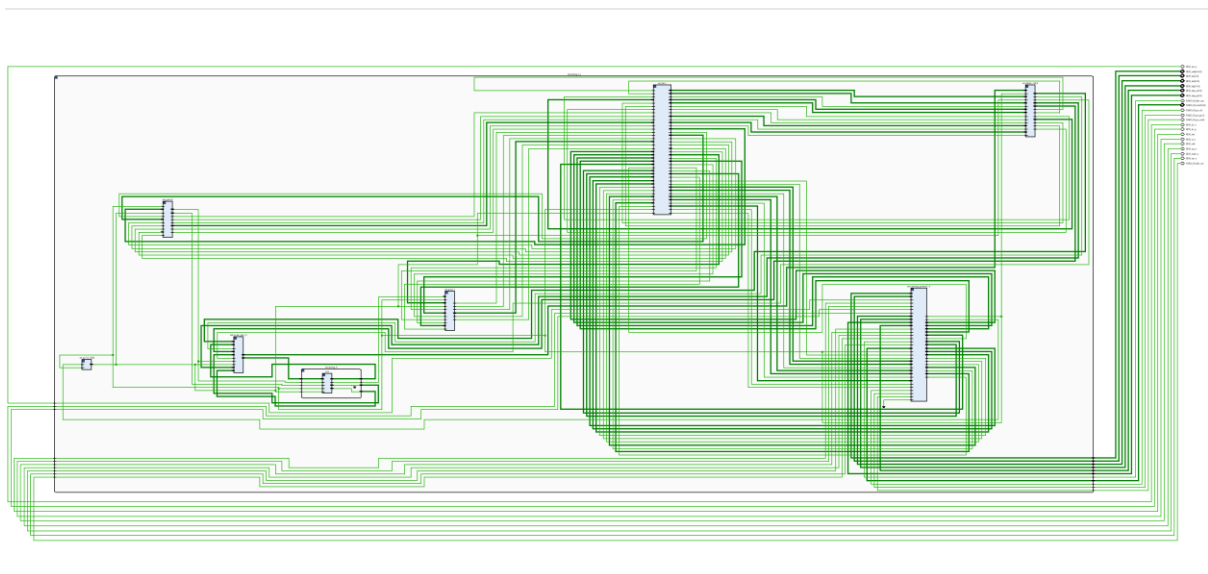


## PL part

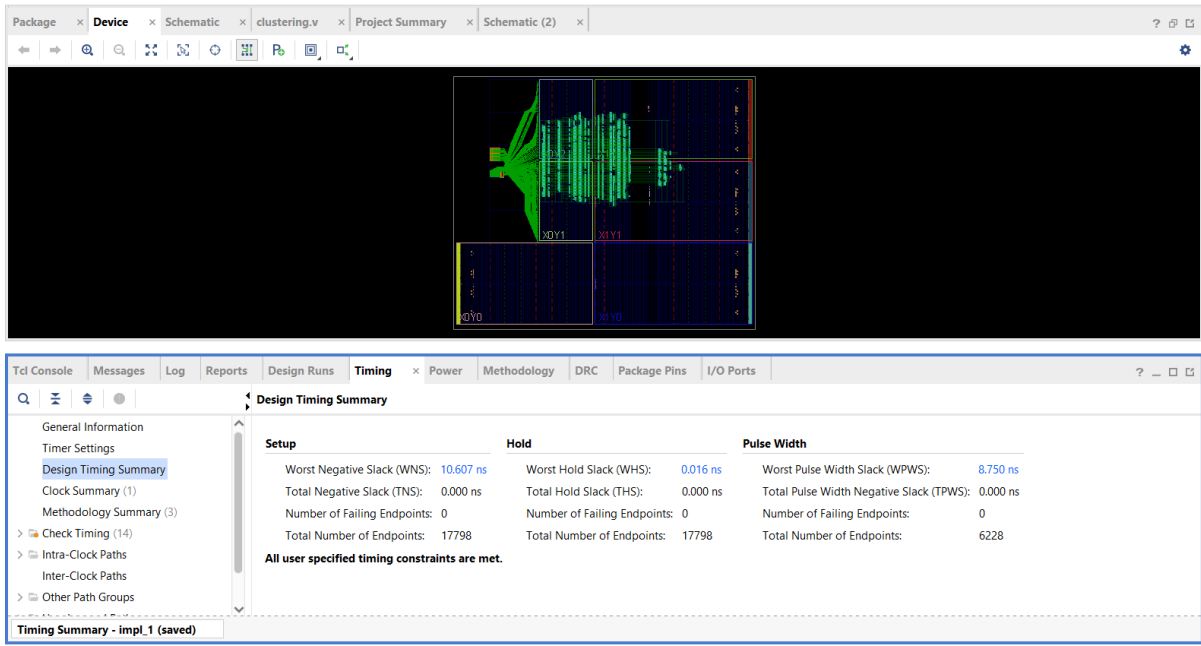




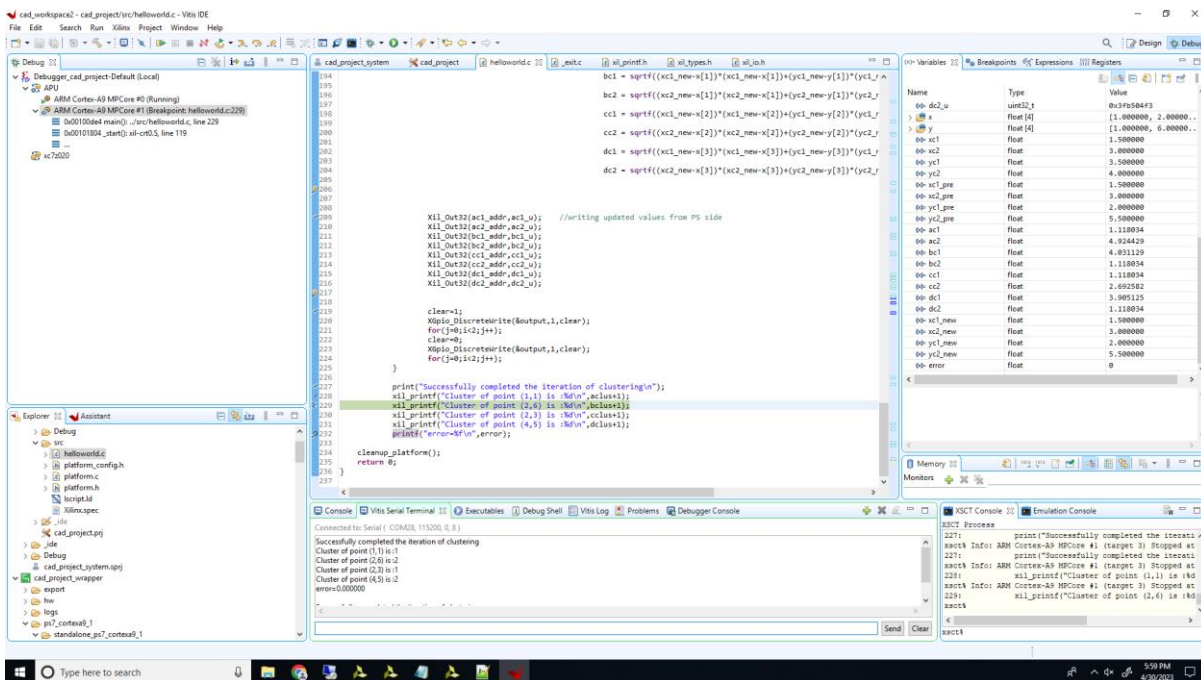
**Implemented Design Schematic**



**Implemented Design Schematic**



## Device Mapping



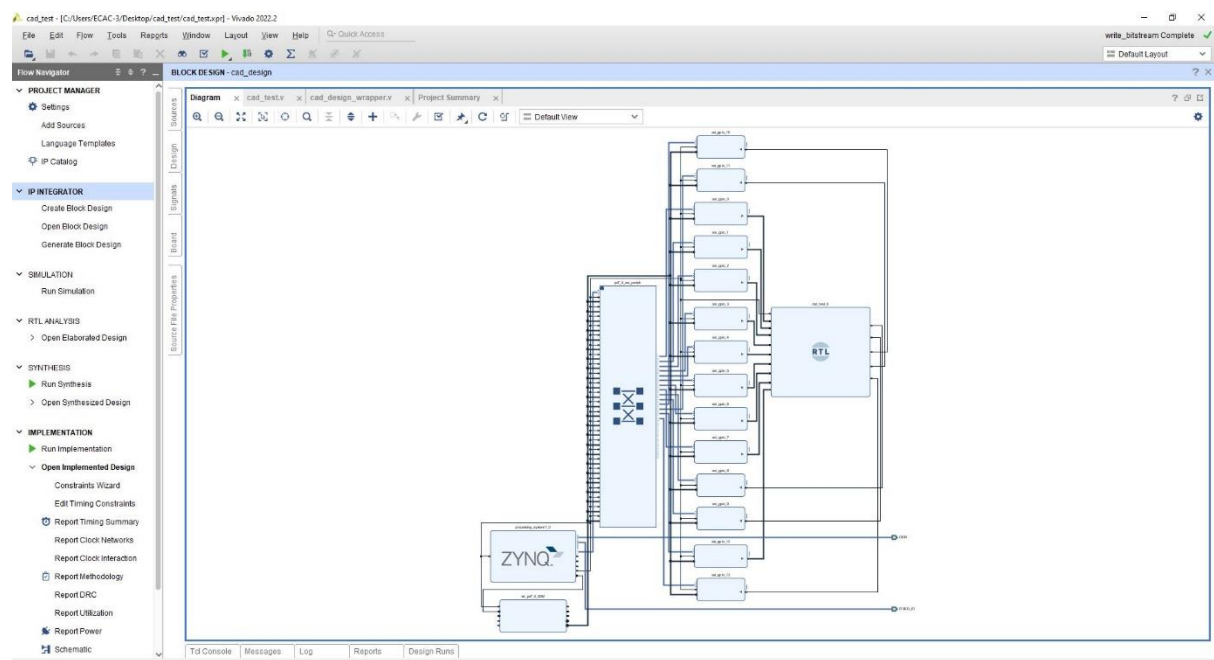
## Results

### Design 3:

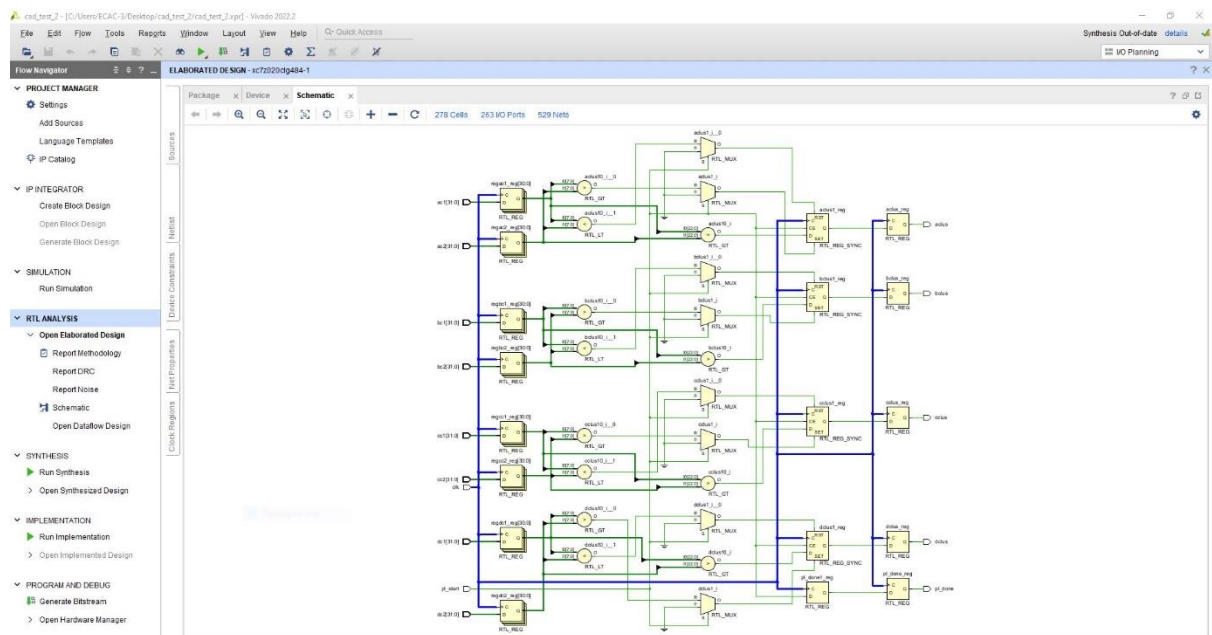
We have removed BRAM completely and used GPIO ports for communication between the PL and PS parts to speed up the process. Here's how the process works:

1. Calculate the distances and fetch the data: The PL calculates the distance between each data point and each centroid and fetches the distance data. This is done using a single block, which reduces the number of hardware resources required.
2. Compare distances and send the result to the PS: After the distance data is fetched, the PL block compares the distances between each data point and each centroid. The block outputs the index of the nearest centroid and sends the result to the PS using a GPIO port.
3. Read the comparison data in the PS: The PS reads the comparison data sent by the PL using the GPIO port.
4. Calculate new cluster centroids based on the comparison data: The PS calculates the new cluster centroids based on the data.
5. Repeat steps 1-4 until convergence: The process of calculating distances, comparing distances, and calculating new cluster centroids is repeated until the cluster centroids converge.

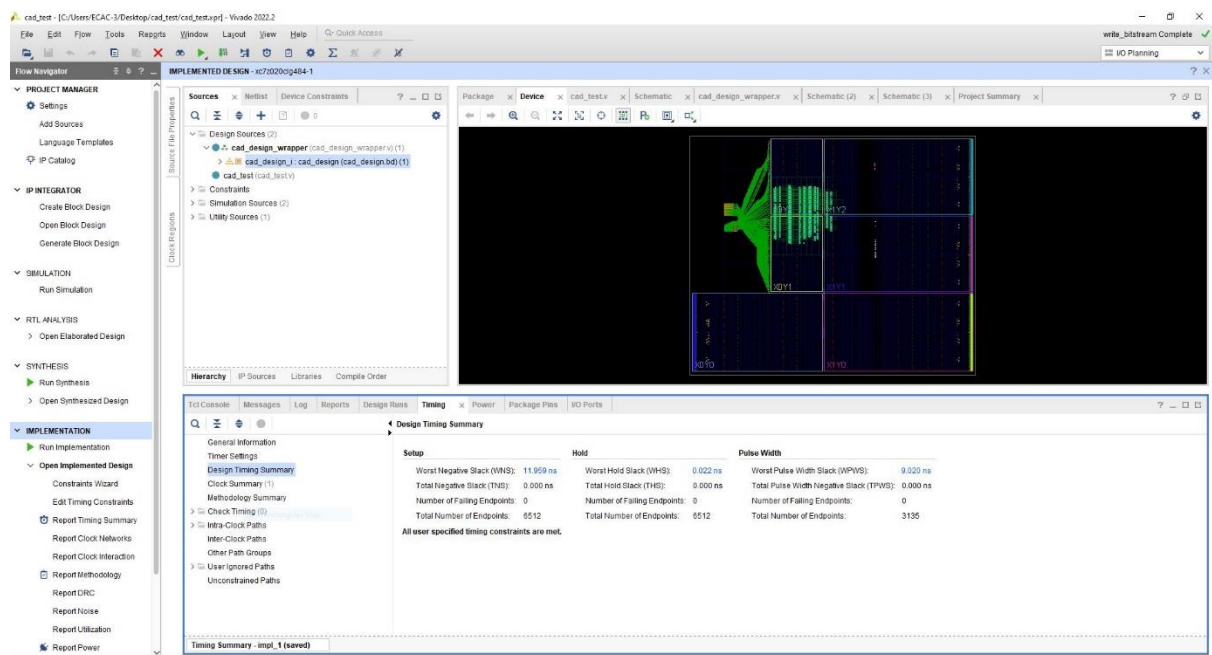
By removing BRAM and using GPIO ports for communication between the PL and PS parts, we can further reduce the hardware resources required and potentially speed up the process. However, this approach may also have some limitations, such as an increased overhead in data transfer. It is important to carefully evaluate the trade-offs between these different approaches and choose the one that is most suitable for our specific use case.



Block diagram



## PL part



## Device mapping



- Comparison: This design may be the most efficient in terms of hardware resources, but it may have less data transfer capability compared to Design 1 and Design 2.

Overall, the best design depends on the specific requirements of our use case. If we have limited hardware resources and are willing to work on smaller data set , Design 3 may be the most efficient approach. If we have more hardware resources available and want to maximize performance, Design 1 or Design 2 may be better. It is important to carefully evaluate the trade-offs between different design options and choose the one that best fits our specific needs.

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	5060	53200	9.51
LUTRAM	732	17400	4.21
FF	5272	106400	4.95
BRAM	2	140	1.43
BUFG	1	32	3.13

Design-1

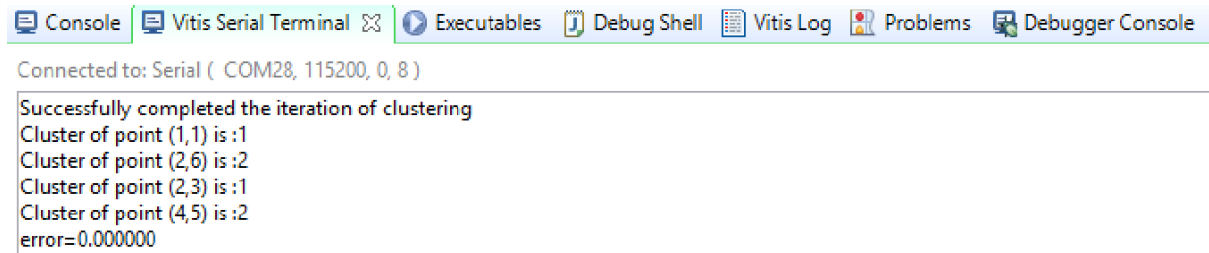
Resource	Utilization	Available	Utilization %
LUT	5000	53200	9.40
LUTRAM	732	17400	4.21
FF	5087	106400	4.78
BRAM	2	140	1.43
BUFG	1	32	3.13

Design-2

Resource	Utilization	Available	Utilization %
LUT	2216	53200	4.17
LUTRAM	83	17400	0.48
FF	3051	106400	2.87
BUFG	1	32	3.13

Design-3

## Results and Conclusion:



The screenshot shows the Vitis Serial Terminal window with the following text:

```
Connected to: Serial ( COM28, 115200, 0, 8 )  
Successfully completed the iteration of clustering  
Cluster of point (1,1) is :1  
Cluster of point (2,6) is :2  
Cluster of point (2,3) is :1  
Cluster of point (4,5) is :2  
error=0.000000
```

The results obtained in our debugging run matches the expected result of our dataset mentioned earlier.

Through this project we learned and successfully implemented a small unsupervised learning algorithm involving hardware software co-design, implemented on Vivado and Vitis IDE.

### Leaning outcome:

During the process of completing this project we learned to work in an integrated development environment by Xilinx Vitis and also learned to do better hardware designing on Vivado HDL platform.

We would also like to list few advantages of hardware software co-design realized during the process of completion of this project:

1. Improved performance: Hardware-software co-design allows for hardware and software to be optimized together, resulting in better performance than when designed separately.
2. Reduced costs: Co-design allows for better resource utilization, as hardware and software are designed to work together efficiently. This can lead to reduced costs in terms of hardware, software development, and system integration.
3. Faster time-to-market: Co-design allows for better collaboration between hardware and software teams, which can result in faster development times and faster time-to-market for products.
4. Improved reliability: Hardware-software co-design can lead to more reliable systems as hardware and software are designed to work together seamlessly.
5. Flexibility: Co-design allows for greater flexibility in the system design, as hardware and software can be designed to adapt to changing requirements and constraints.



## **B. MIN - ARCHITECTURE DESIGN - CISC PROCESSOR**

### **1. INTRODUCTION**

CISC Processors use variable size instructions and a variety of addressing modes to access data and do operations on them. This flexibility on the side of the programmer when implemented on hardware needs variable cycles for implementation. The various addressing modes in order to be implemented on the hardware need a controller design using micro-coded implementation. MIN Architecture is one such implementation which covers some basic set of instructions of CISC processors.

The instructions are firstly broken down into RTL steps. These are noted down as hardware flowcharts and then each RTL step is mapped to a micro-coded instruction.

These steps are implemented using the execution unit which takes in the micro-coded instruction as input giving us the control signals.

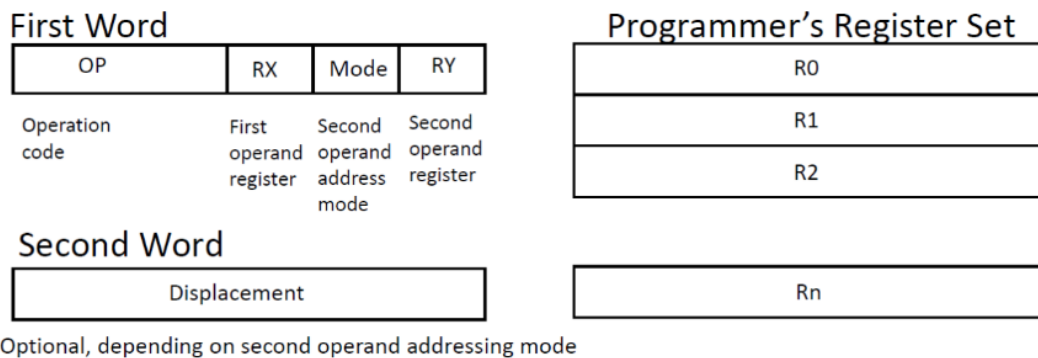
Each micro-coded instruction maps to a set of control bits.

The processor is designed using behavioral modeling and is simulated on Xilinx Vivado.

MIN instruction set format is:

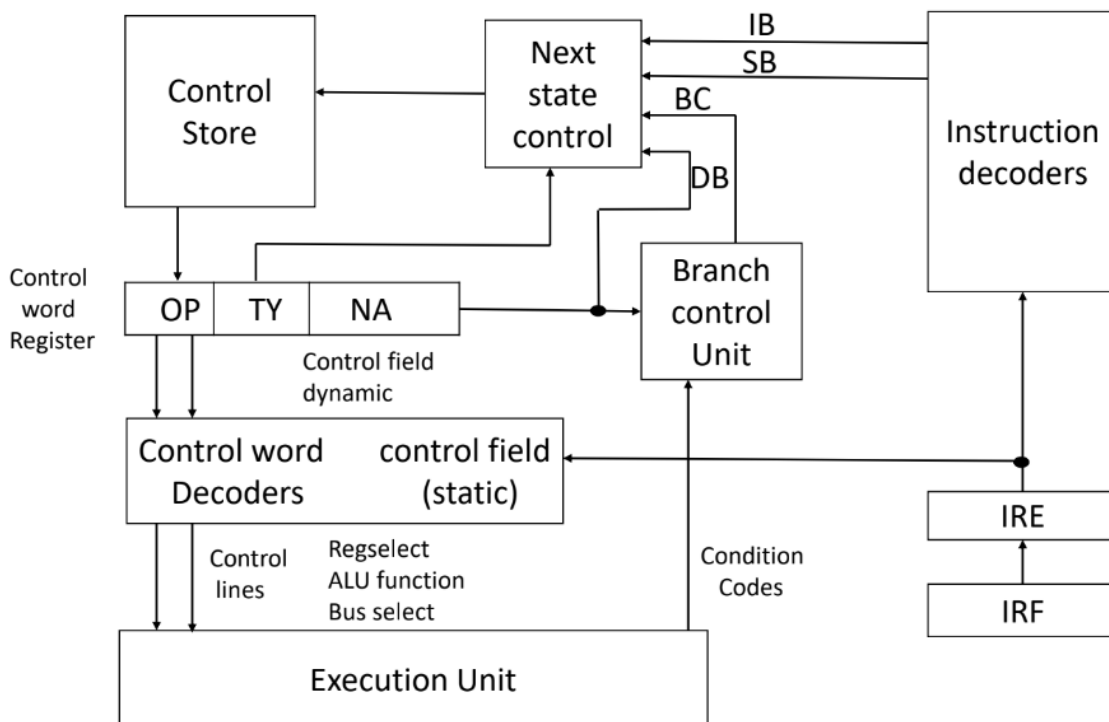
## MIN Instruction set format

- **Instruction Format:**



### MIN instruction format and register set

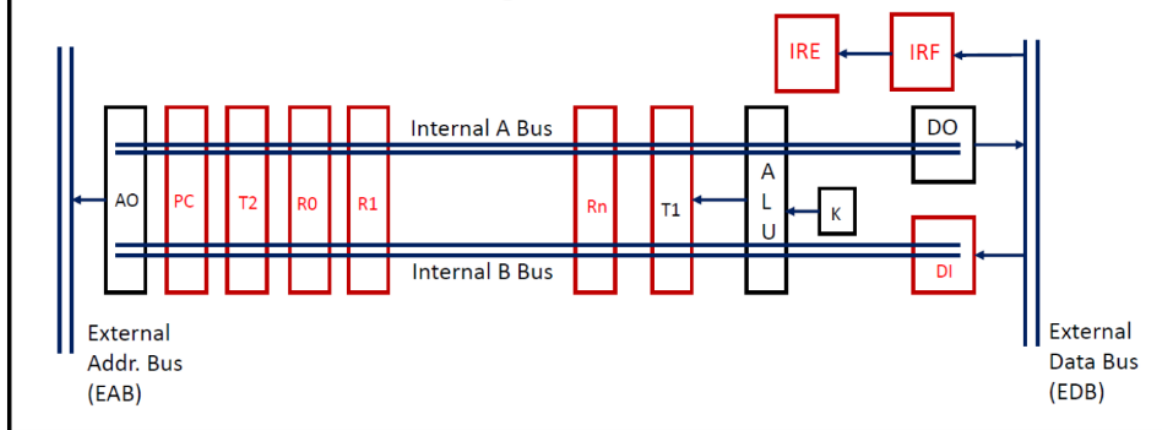
The microcoded control implementation block diagram for MIN architecture is shown:



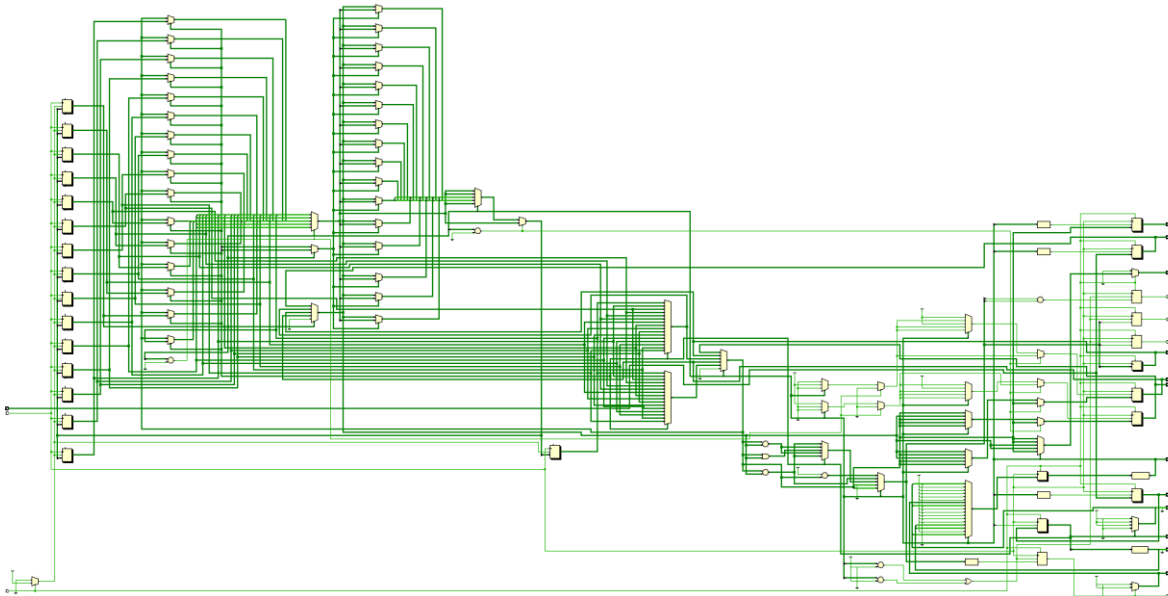
### MICROPROCESSOR BLOCK DIAGRAM(Microcoded Controller)

The execution unit block is implemented as:

• MIN execution unit block diagram:



The following implementation can be seen on Xilinx Vivado:



## C. VERILOG DESIGN AND SIMULATION

### Verilog Code

```
`timescale 1ns / 1ps
```

```
module cisc_proc(
    input clk,
    input rst,
    output reg [17:0] cntrl,
    input [15:0] edb,
    output reg signed [15:0] eab,
    //output reg signed [15:0] ao
    output reg zero,
    output reg overflow,
    output reg sign,
    output reg carry,
    output reg signed [15:0] pc,
    output reg signed [15:0] t1,
    output reg signed [15:0] t2,
    output reg signed [15:0] di,
    output reg signed [15:0] do,
    output reg signed [15:0] irf,
```

```

output reg signed [15:0] ire,
output reg [4:0] ib,
output reg [4:0] sb,
output reg [4:0] bc

);

// reg signed [15:0] ire;

reg [2:0] srcA,srcB;
reg [1:0] destA;
reg [2:0] destB;
reg signed [15:0] busA,busB;
//reg signed [15:0] pc,t1,t2,di,do,irf,ire;
reg signed [15:0] reg_file [0:15];
//reg zero,overflow,sign,carry;
reg signed [15:0] ao;

reg signed [16:0] aluout;
reg signed [15:0] alua,alub;

//initializing registers and ire - for simulation
initial
begin
reg_file[1]=16'd0;
reg_file[2]=16'd2;
#3 ire=16'b000001_0001_10_0010;
end

//simulation of the register value -displaying them
always@(reg_file[1],reg_file[2])
begin
$display("time =%0t, r1 = %h", $time, reg_file[1]);
$display("r2 = %h", reg_file[2]);
end

//assigning value to busA based on control signals
always@(pc,t1,t2,cntrl[17:15],ire) //include reg_file
begin
case(cntrl[17:15])
3'b011:busA=pc;
3'b101:busA=t1;
3'b010:busA=reg_file[ire[3:0]];
3'b110:busA=t2;
3'b001:busA=reg_file[ire[9:6]];
3'b000:busA=16'd0;
default: busA=16'd0;
endcase
end

//assigning value to bus B based on control signals
always@(di,t1,t2,cntrl[12:10]) //include reg_file in sensitivity list
begin
case(cntrl[12:10])
3'b111:busB=di;
3'b010:busB=reg_file[ire[3:0]];
3'b101:busB=t1;
3'b110:busB=t2;
3'b001:busB=reg_file[ire[9:6]];
3'b000:busB=16'd0;
default: busB=16'd0;
endcase
end

//assigning value to dest of bus A based on control signals
always@(posedge clk,posedge rst)
begin

```

```

if(rst==1)
begin
pc<=0;
t2<=0;
end
else
begin
//assigning destA based on control
if(cntrl[14:13]!=2'b00)
begin
case(cntrl[14:13])
2'b11:pc<=busA;
2'b01:t2<=busA;
2'b10:reg_file[ire[3:0]]<=busA;
// 2'b00:pc<=busA;
// default: ;
endcase
end
//assigning dest B based on control
if(cntrl[9:7]!=3'b000)
begin
case(cntrl[9:7])
3'b100:t2<=busB;
3'b011:pc<=busB;
3'b101:begin
t2<=busB;
reg_file[ire[9:6]]<=busB;
end
3'b110:begin
t2<=busB;
reg_file[ire[3:0]]<=busB;
end
3'b010: reg_file[ire[3:0]]<=busB;
3'b001: reg_file[ire[9:6]]<=busB;
// default:
endcase
end
end
end

//alu - alu functions controlled by control word
always@(busA,busB,cntrl[6:4])
begin
case(cntrl[6:4])
3'b001: aluout=busA+16'd1;
3'b010: aluout=busA+busB;
3'b110: begin
case(ire[15:10])
6'b001100: aluout=busA+busB;
6'b010100: aluout=busA-busB;
6'b011100: aluout=busA&busB;
6'b000101: aluout=busA-busB;
default: aluout=busA+busB;
endcase
end
3'b100: aluout=busA+16'd0;
3'b011: aluout=busA+-16'd1;
3'b000: aluout=busA+16'd0;
default: aluout=busA+16'd0;
endcase
end

//CC - setting the cconditional codes -zero, overflow, sign and carry based on the control word and aluout
always@(posedge clk,posedge rst)
begin
if(rst==1)
begin
zero<=1'b0;
overflow<=1'b0;
sign<=1'b0;

```

```

        carry<=1'b0;
    end
    else
    begin
    if((cntrl[6:4]==3'b110)|| (cntrl[6:4]==3'b100))
    begin

    if(aluout==16'd0)
    zero<=1'b1;
    else
    zero<=1'b0;

    if(aluout[16]!=aluout[15]) // (~s[7]&a[7]&b[7])(s[7]&(~a[7])&(~b[7]))
    overflow<=1'b1;
    else
    overflow<=1'b0;

    if(aluout[15]==1'b1)
    sign<=1'b1;
    else
    sign<=1'b0;

    if(aluout[16]==1'b1)
    carry<=1'b1;
    else
    carry<=1'b0;
    end
    end
    end

    //assigning value from external memory
    always@(posedge clk,posedge rst)
    begin
    if(rst==1)
    begin
    di<=16'd0;
    irf<=16'd0;
    do<=16'd0;
    end
    else
    begin
    case(cntrl[3:1])
    3'b001: di<=edb;
    3'b010: irf<=edb;
    3'b101: di<=edb;
    3'b111: do<=busA;          ///not writing to edb??
    default: ;
    endcase
    end
    end

    //pssing out the address to external memory
    always@(cntrl,busA,busB,rst)
    begin
    if(rst==1)
    begin
    ao<=0;
    end
    else
    begin
    case(cntrl[3:1])
    3'b001: begin
            ao<=busA;
            end
    3'b010: ao<=busA;
    3'b101: ao<=busB;
    3'b111: ao<=busB;
    3'b000: ao<=busA;
    default: ao<=busA;
    endcase
    end
    end

```

```
end
end
```

```
//passing a0 to external address bus
always@(ao)
begin
eab<=ao;
end
```

```
//assigning value to ire from irf
always@(posedge clk,posedge rst)
begin
if(rst==1)
begin
ire<=16'd0;
end
else if(cntrl[0]==1'b1)
begin
ire<=irf;
end
end
```

```
//assigning value to t1 from aluout
always@(posedge clk,posedge rst)
begin
if(rst==1)
begin
t1<=16'd0;
end
else
begin
t1<=aluout;
end
end
```

```
/////////////////////////////////CONTROL/////////////////////////////////
```

```
reg [4:0] cntrl_addr;
//reg [1:0] next_state;
//reg [4:0] ib,sb,bc;
```

```
//control memory - we have used a rom based implementation
```

```
always@(cntrl_addr)
begin
case(cntrl_addr)
5'd1:cntrl<=18'b001_00_010_000_110_000_0; //oprr1
5'd2:cntrl<=18'b011_00_101_010_001_010_0; //oprr2
5'd3:cntrl<=18'b011_00_000_000_001_001_0; //abdm1
5'd4:cntrl<=18'b101_11_000_000_000_000_0; //abdm2
5'd5:cntrl<=18'b010_00_111_000_010_000_0; //abdm3
5'd6:cntrl<=18'b101_01_000_000_000_001_0; //abdm4
5'd7:cntrl<=18'b000_00_010_100_000_101_0; //adrm1
5'd8:cntrl<=18'b011_00_111_101_001_010_0; //ldrm1
5'd9:cntrl<=18'b110_00_101_011_100_000_1; //ldrm2
5'd10:cntrl<=18'b001_00_110_000_100_111_0; //strm1
5'd11:cntrl<=18'b011_00_111_100_001_010_0; //test1
5'd12:cntrl<=18'b001_00_111_000_110_000_0; //oprm1
5'd13:cntrl<=18'b101_00_110_000_000_111_0; //oprm2
5'd14:cntrl<=18'b010_00_000_000_001_010_0; //brzz1
5'd15:cntrl<=18'b000_00_101_011_000_000_1; //brzz2
5'd16:cntrl<=18'b011_00_000_000_001_010_0; //brzz3
default: cntrl<=18'd0;
endcase
end
```

```
//instr decoder - gives ib and sb based on opcode
always@(ire)
```



```

begin
if(ire[15:10]==000101)
ib<=5'd14;
else
begin
case(ire[5:4])
2'b00:ib<=5'd1;//reg_direct
2'b01:ib<=5'd7;//reg_indirect
2'b10:ib<=5'd3;//base_plus_displacement
default:ib<=5'd0;
endcase
end
case(ire[15:10])
6'b000_001:sb<=5'd8;//load
6'b000_010:sb<=5'd10;//store
6'b000_011:sb<=5'd11;//test
6'b001_100:sb<=5'd12;//add
6'b010_100:sb<=5'd12;//sub
6'b011_100:sb<=5'd12;//and
//6'b000_101:sb<=5'd12;//bz
default:sb<=5'd0;
endcase
end

//next state control - gives which control word should be used next- ib, sb, bc or direct branch
always@(posedge clk,posedge rst)
begin
if(rst==1)
cntrl_addr<=5'd0;
else
begin
case(cntrl)
//give ib,sb or direct branch or branch control? how to do branch?
18'd0:cntrl_addr<=ib;
18'b001_00_010_000_110_000_0:cntrl_addr<=5'd2; //oprr1
18'b011_00_101_010_001_010_0:cntrl_addr<=5'd15; //oprr2
18'b011_00_000_000_001_001_0:cntrl_addr<=5'd4; //abdm1
18'b101_11_000_000_000_000_0:cntrl_addr<=5'd5; //abdm2
18'b010_00_111_000_010_000_0:cntrl_addr<=5'd6; //abdm3
18'b101_01_000_000_000_001_0:cntrl_addr<=sb; //abdm4
18'b000_00_010_100_000_101_0:cntrl_addr<=sb; //adrm1
18'b011_00_111_101_001_010_0:cntrl_addr<=5'd9; //ldrm1
18'b110_00_101_011_100_000_1:cntrl_addr<=ib; //ldrm2
18'b001_00_110_000_100_111_0:cntrl_addr<=5'd16; //strm1
18'b011_00_111_100_001_010_0:cntrl_addr<=5'd9; //test1
18'b001_00_111_000_110_000_0:cntrl_addr<=5'd13; //oprml
18'b101_00_110_000_000_111_0:cntrl_addr<=5'd16; //oprml2
18'b010_00_000_000_001_010_0:cntrl_addr<=bc; //brzz1
18'b000_00_101_011_000_000_1:cntrl_addr<=ib; //brzz2
18'b011_00_000_000_001_010_0:cntrl_addr<=5'd15; //brzz3
default:cntrl_addr<=5'd0;
endcase
end
end

//branch control unit
always@(zero)
begin
if(zero==1)
bc<=5'd15;
else
bc<=5'd16;
end

endmodule

```

# Testbench

```
`timescale 1ns / 1ps

module tb;

reg signed [15:0] mem [0:15];
reg clk,rst;
reg signed [15:0] edb;
wire signed [15:0] eab;
wire signed [17:0] cntrl;
wire signed [15:0] pc,t1,t2,di,do,irf,ire;
wire zero,sign,carry,overflow;

wire [4:0] ib,sb,bc;
initial
begin
clk=0;
forever #5 clk=~clk;
end

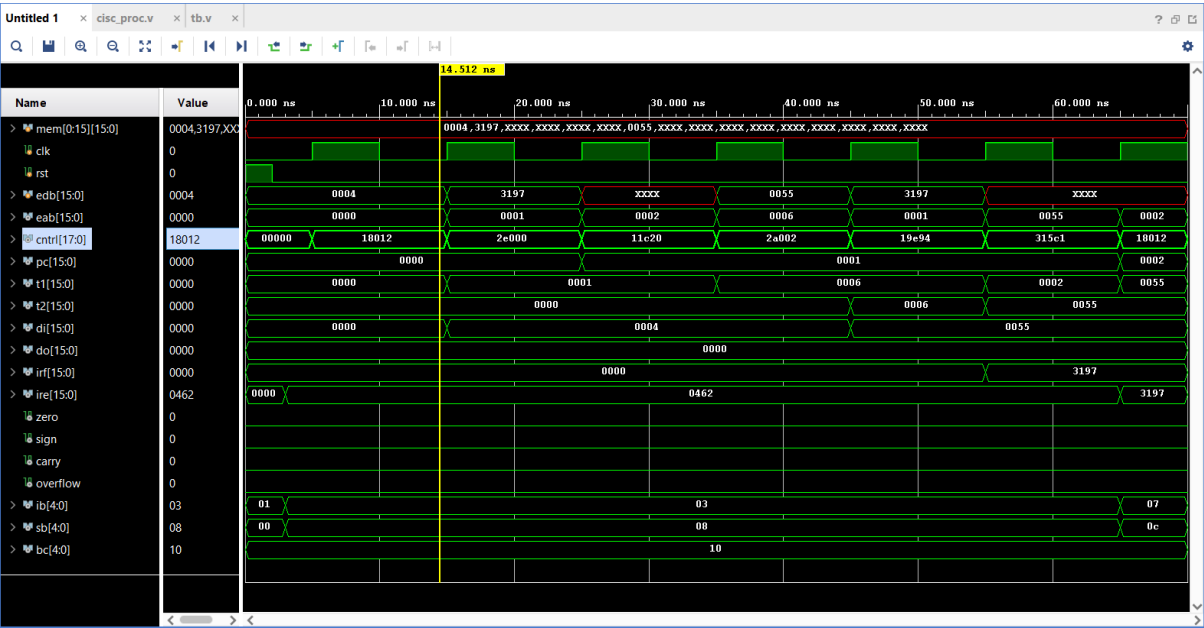
initial
begin
mem[0]=16'd4;
mem[1]=16'h3197;
mem[6]=16'h0055;
end

always@(cab)
begin
edb<=mem[eab];
end
cisc_proc DUT(clk,rst,cntrl,edb,eab,zero,overflow,sign,carry,pc,t1,t2,di,do,irf,ire,ib,sb,bc);

initial
begin
rst=1'b1;
#2 rst=1'b0;

#68 $finish;
end
endmodule
```

# Simulation Result



## D. RTL Compiler

- RTL Compiler is a tool that is commonly used in digital design to synthesize register transfer level (RTL) code to gate-level netlists. The RTL code is a high-level description of the digital circuit behavior that is expressed in a hardware description language (HDL) such as Verilog or VHDL.
- The RTL Compiler takes the RTL code as input and performs a series of optimizations to produce an optimized gate-level netlist that can be used to generate the physical layout of the circuit. These optimizations include logic optimization, mapping to a library of standard cells, timing optimization, and placement and routing.
- The output of the RTL Compiler is a gate-level netlist, which is a description of the circuit in terms of gates, flip-flops, and interconnects. This netlist can then be used by other tools in the design flow, such as the place and route tool, to produce the final physical layout of the circuit.
- The use of RTL Compiler is an important step in the digital design flow, as it allows designers to optimize their RTL code for area, power, and timing. By using a tool such as RTL Compiler, designers can achieve better performance and lower power consumption in their digital circuits.

## E. RTL Compiler

- Area report

```
=====
Generated by:      Encounter(R) RTL Compiler v07.20-s009_1
Generated on:      Apr 30 2023 03:10:26 PM
Module:            synth_cisc
Technology library: fsd0k_a_generic_core_1d0vtc 2007Q2v1.3
Operating conditions: _nominal_ (balanced_tree)
Wireload mode:     enclosed
Area mode:         timing library
=====

-----
Instance          Cells  Cell Area  Net Area  Wireload
-----
synth_cisc         1907    17124      0      enG10K (S)
  final_adder_mux_aluout_153_10  136      691      0      enG5K (S)

(S) = wireload was automatically selected
|
```

- Gate Area Report

```

=====
Generated by:      Encounter(R) RTL Compiler v07.20-s009_1
Generated on:      Apr 30 2023 03:10:27 PM
Module:            synth_cisc
Technology library: fsd0k_a_generic_core_1d0vtc 2007Q2v1.3
Operating conditions: _nominal_ (balanced_tree)
Wireload mode:     enclosed
Area mode:         timing library
=====

```

Gate	Instances	Area	Library
-----			
AN2B1RLX1	2	12.000	fsd0k_a_generic_core_1d0vtc
AN2B1RLXLP	15	75.000	fsd0k_a_generic_core_1d0vtc
AN2RLX1	50	250.000	fsd0k_a_generic_core_1d0vtc
AN2RLX2	23	161.000	fsd0k_a_generic_core_1d0vtc
AN2RLX3	5	40.000	fsd0k_a_generic_core_1d0vtc
AN2RLX4	3	39.000	fsd0k_a_generic_core_1d0vtc
AN3B2RLX1	2	14.000	fsd0k_a_generic_core_1d0vtc
AN3RLX1	10	70.000	fsd0k_a_generic_core_1d0vtc
AN3RLX2	2	16.000	fsd0k_a_generic_core_1d0vtc
AN3RLX4	1	16.000	fsd0k_a_generic_core_1d0vtc
AN4B1RLX1	1	8.000	fsd0k_a_generic_core_1d0vtc
AN4B1RLXLP	5	40.000	fsd0k_a_generic_core_1d0vtc
AN4RLX1	33	330.000	fsd0k_a_generic_core_1d0vtc
A0112RLX1	1	9.000	fsd0k_a_generic_core_1d0vtc
A012RLX1	25	175.000	fsd0k_a_generic_core_1d0vtc
A022RLXLP	16	144.000	fsd0k_a_generic_core_1d0vtc
A01122RLX1	2	18.000	fsd0k_a_generic_core_1d0vtc
A0112RLX1	17	102.000	fsd0k_a_generic_core_1d0vtc
-----			
OR2B1RLXLP	11	55.000	fsd0k_a_generic_core_1d0vtc
OR2RLX1	4	20.000	fsd0k_a_generic_core_1d0vtc
OR2RLX2	1	7.000	fsd0k_a_generic_core_1d0vtc
OR3B2RLX1	4	28.000	fsd0k_a_generic_core_1d0vtc
OR3RLX1	1	7.000	fsd0k_a_generic_core_1d0vtc
OR4B1RLX1	2	16.000	fsd0k_a_generic_core_1d0vtc
OR4B2RLX1	4	32.000	fsd0k_a_generic_core_1d0vtc
QDFERBRLX1	16	400.000	fsd0k_a_generic_core_1d0vtc
QDFFRBRLX1	19	342.000	fsd0k_a_generic_core_1d0vtc
QDFFRBRLX2	1	21.000	fsd0k_a_generic_core_1d0vtc
QDFFRBRLX3	2	46.000	fsd0k_a_generic_core_1d0vtc
QDFZRBRLX1	54	1350.000	fsd0k_a_generic_core_1d0vtc
QDFZRBRLX2	6	162.000	fsd0k_a_generic_core_1d0vtc
QDFZRBRLX3	4	116.000	fsd0k_a_generic_core_1d0vtc
QDFZRLX1	256	5888.000	fsd0k_a_generic_core_1d0vtc
XNR2RLX1	21	210.000	fsd0k_a_generic_core_1d0vtc
XNR2RLX2	1	16.000	fsd0k_a_generic_core_1d0vtc
XOR2RLX1	18	180.000	fsd0k_a_generic_core_1d0vtc
-----			
total	1907	17124.000	

Type	Instances	Area	Area %
-----			
sequential	358	8325.000	48.6
inverter	222	764.000	4.5
logic	1327	8035.000	46.9
-----			
total	1907	17124.000	100.0

- Power Report

```

=====
Generated by:      Encounter(R) RTL Compiler v07.20-s009_1
Generated on:      Apr 30 2023  03:10:27 PM
Module:           synth_cisc
Technology library: fsd0k_a_generic_core_1d0vtc 2007Q2v1.3
Operating conditions: _nominal_ (balanced_tree)
Wireload mode:    enclosed
Area mode:        timing library
=====

          Instance          Leakage    Dynamic    Total
                           Cells Power(nW) Power(nW) Power(nW)
-----
synth_cisc                1907    53.238 1216325.159 1216378.398
  final_adde..aluout_153_10  136     2.355   8923.043   8925.398

```

- Timing report

### Path 1

```

final_adder_mux_aluout_153_10/Z[13]
t1_reg[13]/D      QDFFRBRLX1      +0      3257
t1_reg[13]/CK      setup      0 +240      3496 R
-----
(clock clock)      capture      3500 R
-----
Timing slack :      4ps
Start-point : cntrl_addr_reg[3]/CK
End-point : t1_reg[13]/D

```

### Path 2

```

final_adder_mux_aluout_153_10/Z[14]
t1_reg[14]/D      QDFFRBRLX1      +0      3257
t1_reg[14]/CK      setup      0 +240      3497 R
-----
(clock clock)      capture      3500 R
-----
Timing slack :      5ps
Start-point : ire_reg[6]/CK
End-point : t1_reg[14]/D

```

## Path 3

```
final_adder_mux_aluout_153_10/Z[15]
t1_reg[15]/D      QDFFRBRLX1          +0      3260
t1_reg[15]/CK      setup              0 +240    3499 R
-----
(clock clock)      capture              3500 R
-----
Timing slack :      5ps
Start-point : ire_reg[1]/CK
End-point : t1_reg[15]/D
```

## Path 4

```
final_adder_mux_aluout_153_10/Z[12]
t1_reg[12]/D      QDFFRBRLX1          +0      3260
t1_reg[12]/CK      setup              0 +240    3500 R
-----
(clock clock)      capture              3500 R
-----
Timing slack :      4ps
Start-point : ire_reg[6]/CK
End-point : t1_reg[12]/D
```

- Report Summary

```
=====
Generated by:      Encounter(R) RTL Compiler v07.20-s009_1
Generated on:      Apr 30 2023 03:10:26 PM
Module:           synth_cisc
Technology library: fsd0k_a_generic_core_1d0vtc 2007Q2v1.3
Operating conditions: _nominal_ (balanced_tree)
Wireload mode:    enclosed
Area mode:        timing library
=====

Timing
-----
Tracing clock networks.
Levelizing the circuit.
Computing delays.
Computing arrivals and requireds.

Slack   Endpoint   Cost Group
-----
+4ps t1_reg[12]/D default
```

```

      Area
      ----|

Instance   Cells   Cell Area   Net Area   Wireload
-----
synth_cisc   1907       17124         0   enG10K (S)

(S) = wireload was automatically selected

Design Rule Check
-----
      Initializing DRC engine.

Max_transition design rule: no violations.

Max_capacitance design rule: no violations.

Max_fanout design rule: no violations.

```

## F. SOC Encounter

- It is a complete end-to-end physical design tool from Cadence Design Systems that is used to design and implement complex system-on-chip (SoC) designs. It is a powerful tool that supports all aspects of physical design, including floorplanning, placement, routing, and verification.
- It provides a comprehensive set of features for physical design, including optimization engines, detailed analysis tools, and automated scripting capabilities. The tool is designed to handle large-scale designs and can optimize the design for various goals, such as power, performance, and area.
- The tool is integrated with other tools in the Cadence design flow, such as the RTL Compiler and Innovus Implementation System, which enables designers to use the output of these tools as input to SOC Encounter. This tight integration allows for a smooth design flow and reduces the time-to-market for complex SoC designs.
- It is widely used in the semiconductor industry for designing high-performance, low-power SoCs for a range of applications, such as mobile devices, networking equipment, and automotive systems. It is known for its scalability, speed, and accuracy, which make it an ideal tool for designing complex SoCs with millions of gates.



- Placement check

Congestion distribution:

Remain	cntH		cntV	
3:	0	0.00%	9	0.16%
4:	0	0.00%	33	0.60%
5:	5512	100.00%	5470	99.24%

Total length: 4.293e+04um, number of vias: 14104  
M1(H) length: 1.490e+00um, number of vias: 7003  
M2(V) length: 1.667e+04um, number of vias: 6400  
M3(H) length: 1.929e+04um, number of vias: 670  
M4(V) length: 6.511e+03um, number of vias: 31  
M5(H) length: 4.630e+02um, number of vias: 0  
M6(V) length: 0.000e+00um

Peak Memory Usage was 376.6M  
\*\*\* Finished trialRoute (cpu=0:00:00.2 mem=376.6M) \*\*\*

CongRepair Bin Grid Size (width, height) = ( default\_value , default\_value )  
Trial Route Overflow 0.000000(H) 0.000000(V).  
Skipped repairing congestion.  
\*\*\* Finishing placeDesign default flow \*\*\*  
\*\*\*\* Total cpu 0:0:8  
\*\*\*\* Total real time 0:0:8  
\*\*placeDesign ... cpu = 0: 0: 8, real = 0: 0: 8, mem = 376.6M \*\*  
encounter 1> Begin checking placement ... (start mem=376.6M, init mem=376.6M)  
\*info: Placed = 1907  
\*info: Unplaced = 0  
Placement Density:70.64%(13425/19004)

- Timing Report (pre CTS)

timeDesign Summary						
Setup mode	all	reg2reg	in2reg	reg2out	in2out	clkgate
WNS (ns):	-5.421	-3.065	-5.421	N/A	N/A	N/A
TNS (ns):	-2219.3	-1393.5	-1464.6	N/A	N/A	N/A
Violating Paths:	748	646	358	N/A	N/A	N/A
All Paths:	1116	982	390	N/A	N/A	N/A

DRVs	Real		Total	
	Nr nets(terms)	Worst Vio	Nr nets(terms)	
max_cap	1 (1)	-0.030	1 (1)	
max_tran	1 (106)	-0.301	1 (106)	
max_fanout	0 (0)	0	0 (0)	
max_length	0 (0)	0	0 (0)	

Density: 70.644%  
Routing Overflow: 0.00% H and 0.00% V

Reported timing to dir timingReports  
Total CPU time: 0.68 sec  
Total Real time: 1.0 sec  
Total Memory Usage: 388.886719 Mbytes  
encounter 1> ☐

- Optimizing Design

```
-----
optDesign Final Summary
-----

-----+-----+-----+-----+-----+-----+-----+
| Setup mode | all | reg2reg | in2reg | reg2out | in2out | clkgate |
|-----+-----+-----+-----+-----+-----+-----+
| WNS (ns): | -1.321 | -1.321 | 0.058 | N/A | N/A | N/A |
| TNS (ns): | -672.138 | -672.138 | 0.000 | N/A | N/A | N/A |
| Violating Paths: | 646 | 646 | 0 | N/A | N/A | N/A |
| All Paths: | 1116 | 982 | 390 | N/A | N/A | N/A |
|-----+-----+-----+-----+-----+-----+-----+

-----+-----+-----+-----+-----+
| DRVs | Real | Total |
|-----+-----+-----+-----+-----+
| | Nr nets(terms) | Worst Vio | Nr nets(terms) |
|-----+-----+-----+-----+-----+
| max_cap | 0 (0) | 0.000 | 0 (0) |
| max_tran | 0 (0) | 0.000 | 0 (0) |
| max_fanout | 0 (0) | 0 | 0 (0) |
| max_length | 0 (0) | 0 | 0 (0) |
|-----+-----+-----+-----+-----+

Density: 95.144%
Routing Overflow: 0.00% H and 0.00% V
-----
**optDesign ... cpu = 0:06:09, real = 0:06:09, mem = 478.9M, totSessionCpu=0:19:32 **
*** Finished optDesign ***
encounter 1> 
```

- Timing report (After Optimization)

```
-----
timeDesign Summary
-----

-----+-----+-----+-----+-----+-----+-----+
| Setup mode | all | reg2reg | in2reg | reg2out | in2out | clkgate |
|-----+-----+-----+-----+-----+-----+-----+
| WNS (ns): | -1.321 | -1.321 | 0.058 | N/A | N/A | N/A |
| TNS (ns): | -672.138 | -672.138 | 0.000 | N/A | N/A | N/A |
| Violating Paths: | 646 | 646 | 0 | N/A | N/A | N/A |
| All Paths: | 1116 | 982 | 390 | N/A | N/A | N/A |
|-----+-----+-----+-----+-----+-----+-----+

-----+-----+-----+-----+-----+
| DRVs | Real | Total |
|-----+-----+-----+-----+-----+
| | Nr nets(terms) | Worst Vio | Nr nets(terms) |
|-----+-----+-----+-----+-----+
| max_cap | 0 (0) | 0.000 | 0 (0) |
| max_tran | 0 (0) | 0.000 | 0 (0) |
| max_fanout | 0 (0) | 0 | 0 (0) |
| max_length | 0 (0) | 0 | 0 (0) |
|-----+-----+-----+-----+-----+

Density: 95.144%
Routing Overflow: 0.00% H and 0.00% V
-----
Reported timing to dir timingReports
Total CPU time: 0.77 sec
Total Real time: 1.0 sec
Total Memory Usage: 476.9375 Mbytes
encounter 1> 
```

- Timing report (post CTS)

```
-----
timeDesign Summary
-----

+-----+-----+-----+-----+-----+-----+-----+
| Setup mode | all | reg2reg | in2reg | reg2out | in2out | clkgate |
+-----+-----+-----+-----+-----+-----+-----+
| WNS (ns): | -1.376 | -1.376 | 0.268 | N/A | N/A | N/A |
| TNS (ns): | -682.175 | -682.175 | 0.000 | N/A | N/A | N/A |
| Violating Paths: | 646 | 646 | 0 | N/A | N/A | N/A |
| All Paths: | 1116 | 982 | 390 | N/A | N/A | N/A |
+-----+-----+-----+-----+-----+-----+-----+

+-----+-----+-----+-----+
| | Real | Total |
| DRVs | Nr nets(terms) | Worst Vio | Nr nets(terms) |
+-----+-----+-----+-----+
| max_cap | 0 (0) | 0.000 | 0 (0) |
| max_tran | 0 (0) | 0.000 | 0 (0) |
| max_fanout | 0 (0) | 0 | 0 (0) |
| max_length | 0 (0) | 0 | 0 (0) |
+-----+-----+-----+-----+

Density: 95.875%
Routing Overflow: 0.00% H and 0.00% V
-----
Reported timing to dir timingReports
Total CPU time: 0.75 sec
Total Real time: 0.0 sec
Total Memory Usage: 545.90625 Mbytes
encounter 1> 
```

- Verifying connectivity

```
***** Start: VERIFY CONNECTIVITY *****
Start Time: Tue May 2 17:23:34 2023

Design Name: synth_cisc
Database Units: 1000
Design Boundary: (0.0000, 0.0000) (152.7450, 144.4400)
Error Limit = 1000; Warning Limit = 50
Check all nets

Begin Summary
  Found no problems or warnings.
End Summary

End Time: Tue May 2 17:23:34 2023
Time Elapsed: 0:00:00.0

***** End: VERIFY CONNECTIVITY *****
  Verification Complete : 0 Viols. 0 Wrngs.
  (CPU Time: 0:00:00.1 MEM: 0.000M)

encounter 1> 
```

- Power Analysis

```
File Edit View Search Terminal Help
* report_power
*
-----
Total Power
-----
Total Internal Power: 1.59654332 61.5327%
Total Switching Power: 0.99709400 38.4292%
Total Leakage Power: 0.00098697 0.0380%
Total Power: 2.59462430
-----

Group Internal Power Switching Power Leakage Power Total Power Percentage (%)
-----
Sequential 0.8953 0.09933 0.0002161 0.9949 38.34
Macro 0 0 0 0 0
IO 0 0 0 0 0
Combinational 0.659 0.7233 0.0007562 1.383 53.31
Clock (Combinational) 0.04217 0.1745 1.469e-05 0.2167 8.351
Clock (Sequential) 0 0 0 0 0
Total 1.597 0.9971 0.000987 2.595 100
-----

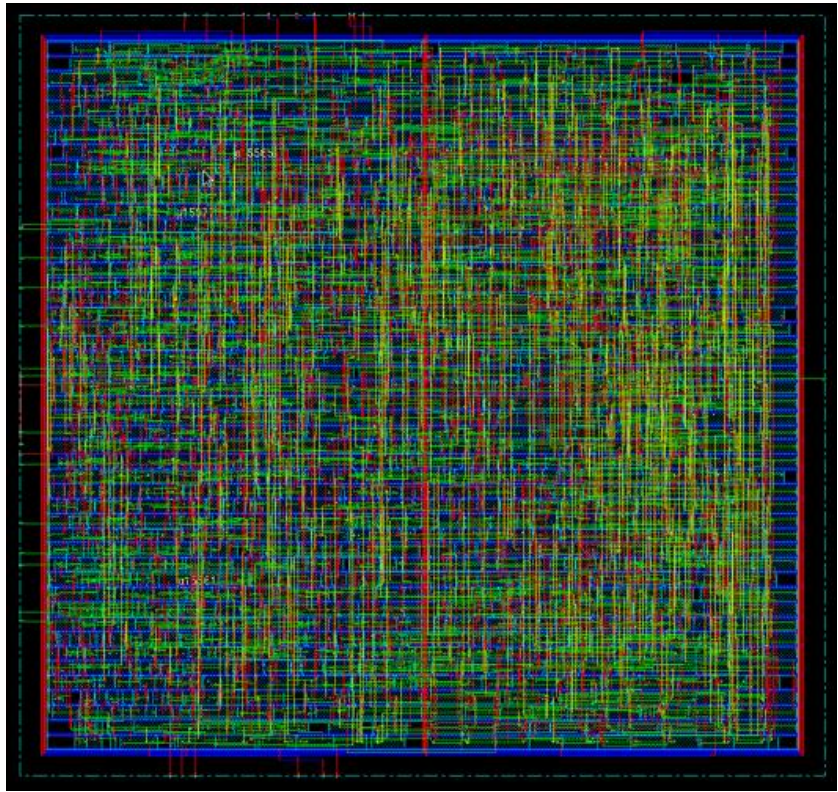
Rail Voltage Internal Power Switching Power Leakage Power Total Power Percentage (%)
-----
Default 0.9 1.597 0.9971 0.000987 2.595 100
-----

Clock Internal Power Switching Power Leakage Power Total Power Percentage (%)
-----
clock 0.04217 0.1745 1.469e-05 0.2167 8.351
Total 0.04217 0.1745 1.469e-05 0.2167 8.351
-----

* Power Distribution Summary:
* Highest Average Power: clock_L1_I0 (INVRLX20): 0.04064
* Highest Leakage Power: clock_L1_I0 (INVRLX20): 2.996e-06
* Total Cap: 2.07294e-11 F
* Total instances in design: 2223
* Total instances in design with no power: 0
* Total instances in design with no activity: 0
* Total Fillers and Decap: 0
-----

report_power consumed time (real time) 00:00:00 : peak memory (835M)
1
encounter 5> □
```

- Final Layout



## **REFERENCES**

- XLINX official Website
- verilog-hdl-samir-palnitkar-2nd-edition
- handbook-of-hardware-software-codesign-soonhoi-ha
- <https://xilinxprod-catalog.netexam.com/Certification/47226/designing-with-versal-ai-engine-2-graph-programming-with-ai-engine-kernels>
- <https://www.abebbooks.fr/9780792376446/Advanced-ASIC-Chip-Synthesis-Using-0792376447/plp>