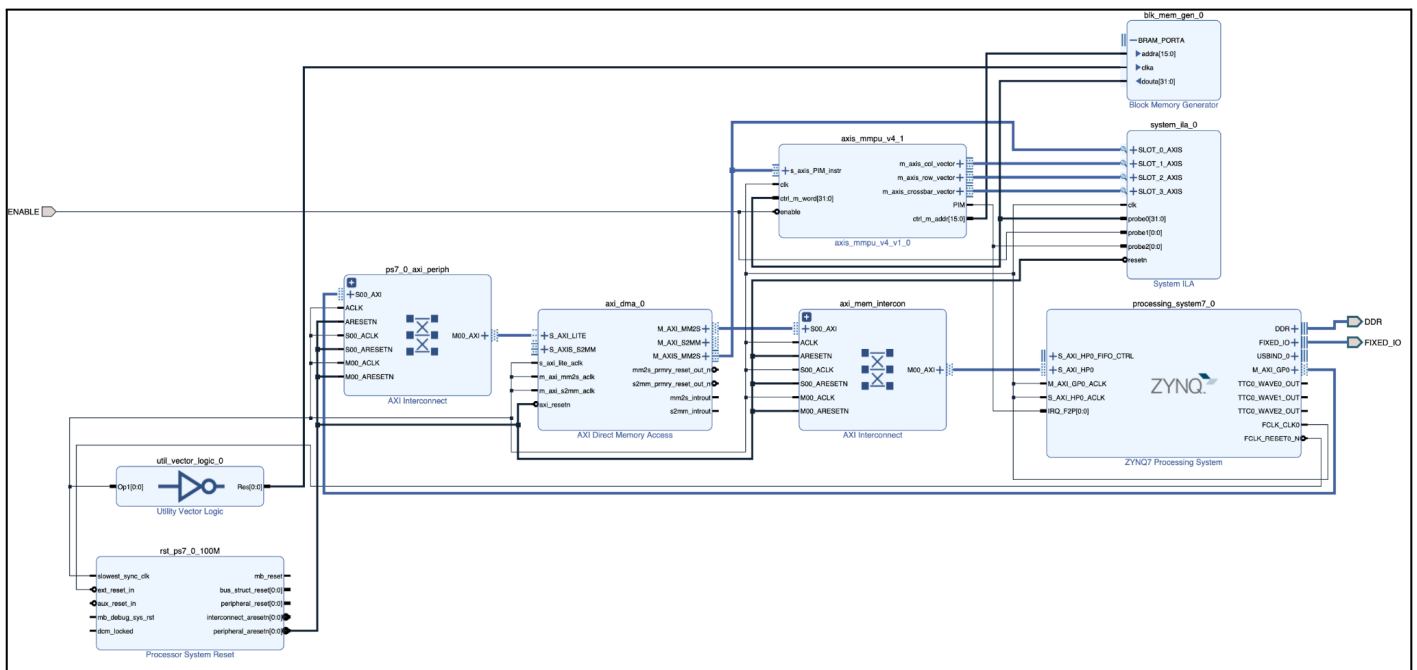


# MICROCODE BASED CONTROLLER FOR MEMRISTIVE MEMORY PROCESSING UNIT (mMPPU)

## USER MANUAL



By Arjun Tyagi  
Supervised by Rotem Ben-Hur



ARCHITECTURES  
SYSTEMS  
INTELLIGENT COMPUTING  
INTEGRATED CIRCUITS

# INTRODUCTION

Modern computers adopt the Von Neumann architecture, separating data storage (memory) and processing (CPU). A significant issue with Von Neumann architecture is the "Memory Wall," which refers to a growing disparity between the CPU and memory outside the CPU chip due to the channel's constrained communication bandwidth. With data-intensive applications becoming increasingly popular, processing-in-memory (PIM)-based systems are receiving greater attention. The Memristive Memory Processing Unit (mMPU) enables true in-memory processing based on a unit that stores and processes data using the same cells. The mMPU is enabled by memristors that are naturally utilized as memory. They can also perform logical operations using a technology we developed called Memristor Aided Logic (MAGIC), forming the foundation of an mMPU.

In order to fully utilize the mMPU, we developed a microcode based controller that determines the sequence of voltages that must be applied at the mMPU in order to perform the required PIM instructions. Furthermore, a system incorporating CPU and the proposed mMPU Controller was designed to enable the user to send in PIM instructions directly from the CPU and observe the output of the mMPU Controller. Lastly, the output of the mMPU Controller was applied to a python-based mMPU simulator to verify if the sequence of voltages from mMPU Controller produced the desired result .

Note: Please follow all the steps as mentioned in the user manual. Skipping a step might result in incorrect operation or errors.

# PREREQUISITE

Before following the manual, please download the repository '*mMPU\_Controller\_&\_System\_Design*' from the following GitHub link: [https://github.com/Arjun-Tyagi25/mMPU\\_Controller\\_and\\_System\\_Design.git](https://github.com/Arjun-Tyagi25/mMPU_Controller_and_System_Design.git) & extract it.

The extracted folder contains the following files:

- 1. mMPU\_Controller**

This folder contains the Vivado Project & IP core of the mMPU Controller.

- 2. mMPU\_System\_Design**

This folder contains the system design incorporating CPU, mMPU Controller & various other components.

- 3. mMPU\_CPU\_Source\_Code.c**

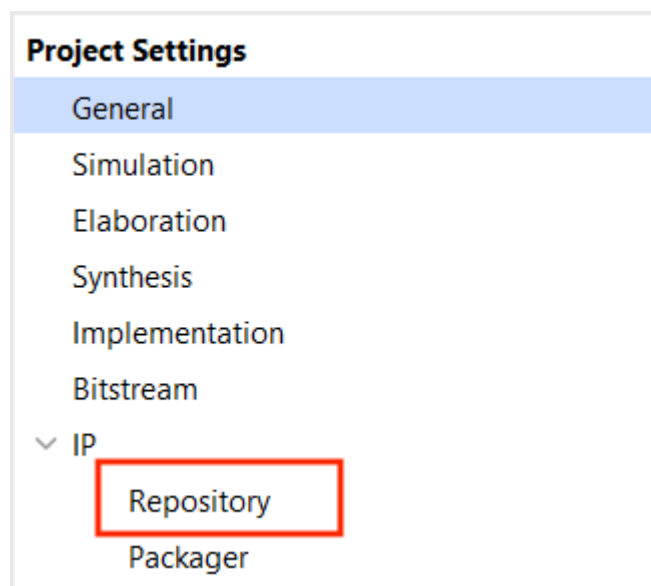
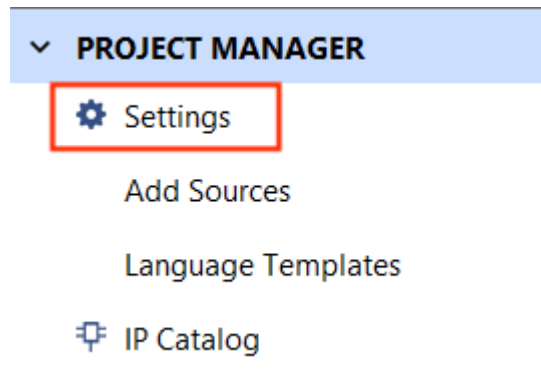
This is the source code for the CPU that we will use for the system design.

## 1. Opening the design

Open the mMPU\_System\_Design.xpr Vivado project inside the mMPU\_System\_Design folder, preferably in version Vivado 2021.2. If a newer version is being used, Vivado will automatically prompt you to update the project to the newer version.

## 2. Adding the mMPU Controller IP to the project

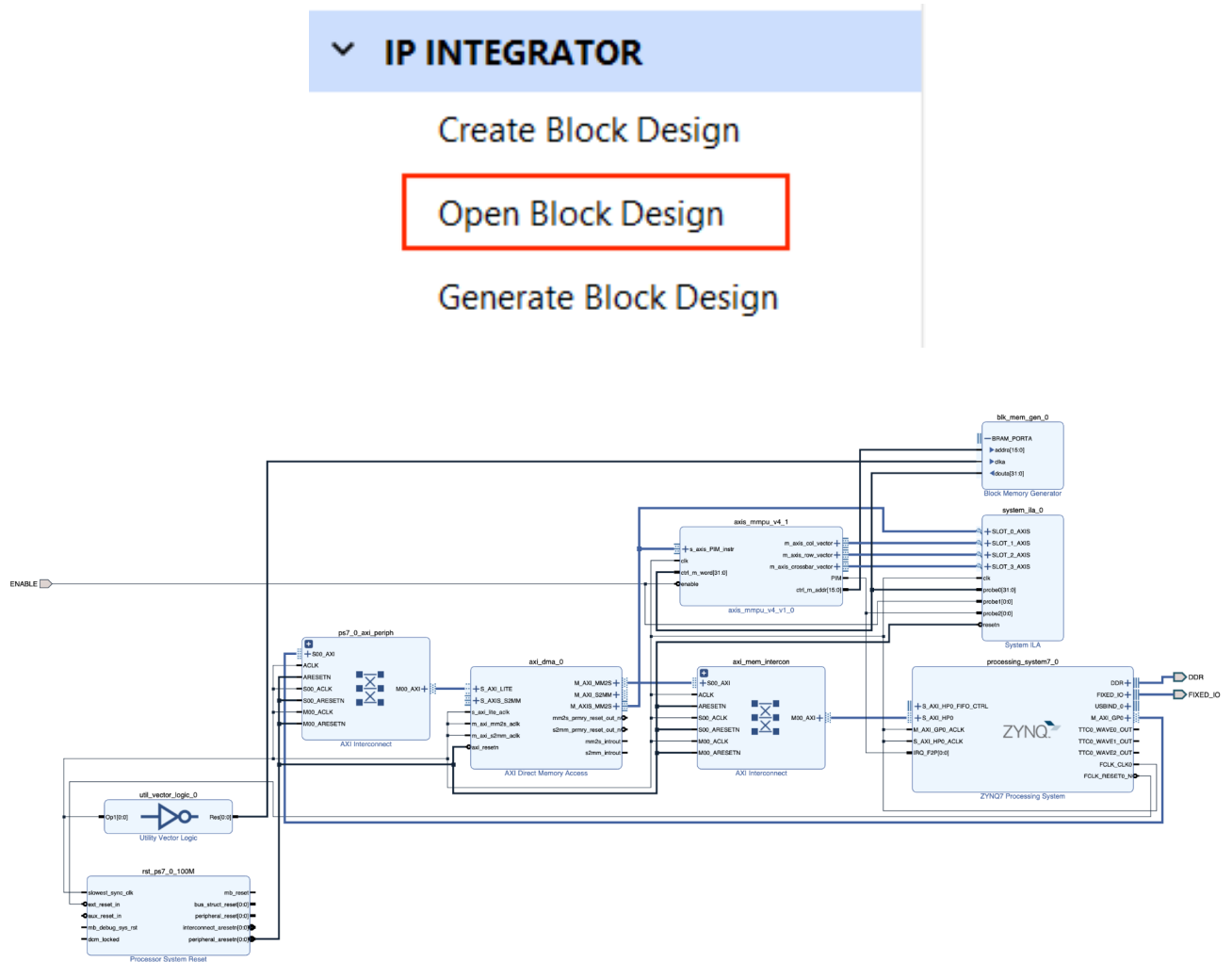
Click on Settings in the Project Manager. In the Settings dialog box, click on IP → Repository as shown below. Click on Add button and select the mMPU\_Controller folder that you extracted. Vivado will automatically detect the mMPU Controller IP inside the folder. Click Apply & OK.



### 3. Viewing Block Design

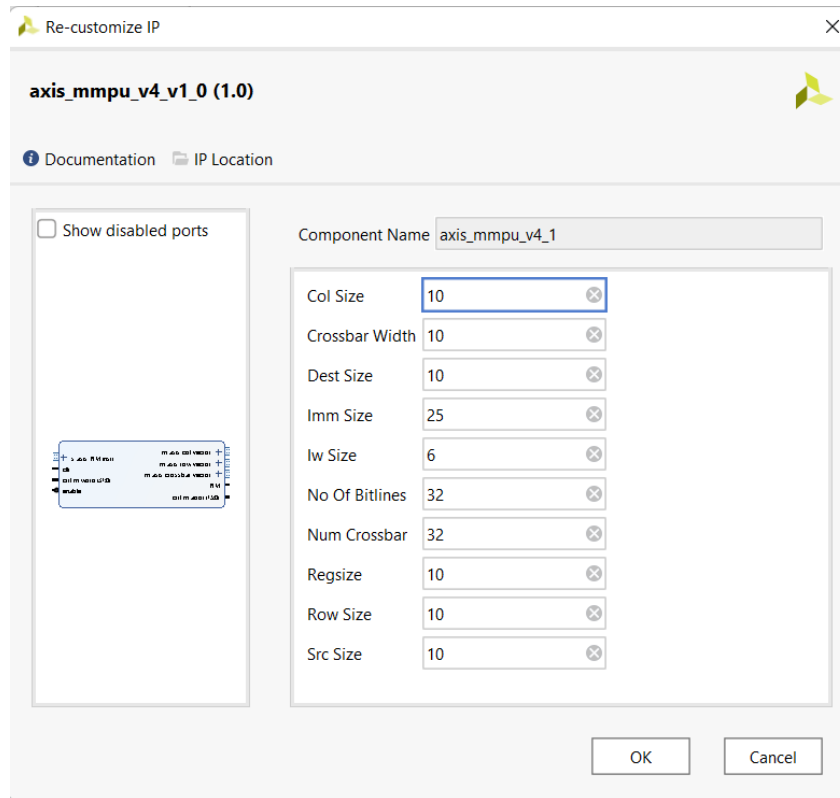
Once the project is open, click on IP Integrator → Open Block Design from the Flow Navigator panel on the left.

The block design connecting various components like the CPU and the mMPU Controller should now be open.



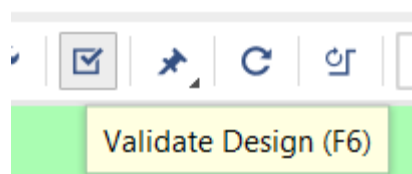
### 4. Changing mMPU Controller Parameters

If you desire, you can change the various parameters of mMPU Controller like number of bitlines, number of crossbar arrays, etc as shown below.

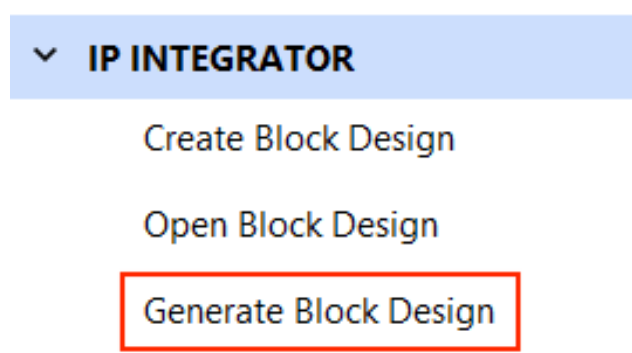


## 5. Validating & Generating the Block Design

Once you have made the changes, click on Validate Design (or press F6) as shown below.

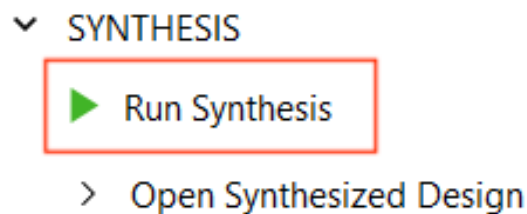


Once the design is validated, click on Generate Block Design from the Flow Navigator as shown below.



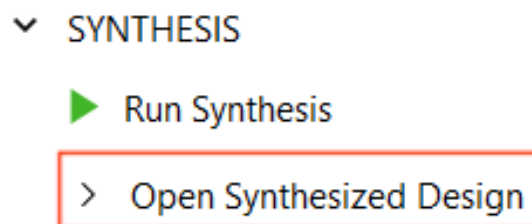
## 6. Synthesizing the Design

Click on Synthesis → Run Synthesis option in the Flow Navigator as shown below.



## 7. Mapping Enable Input to a DIP Switch on the FPGA

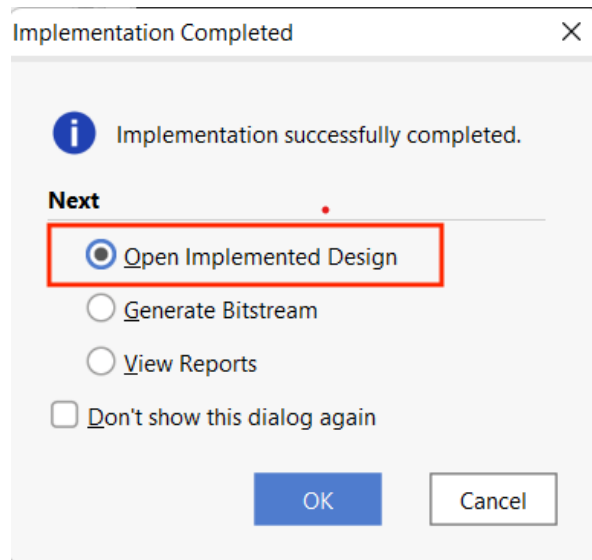
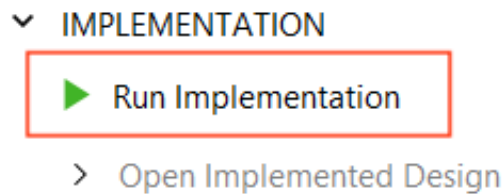
Once the synthesis is complete, click on Open Synthesized Design & then click on I/O Ports. As shown below, map the ENABLE input of the design to one of the DIP switches present on the development board. In our case, we map it to F21, which is in turn connected to a DIP switch.



Tcl Console	Messages	Log	Reports	Design Runs	IP Status	Package Pins	I/O Ports											
?														□	□			
Q														≡	⚙	+	⌕	⚙
Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	SI						
All ports (131)																		
DATA.ENABLE_54576 (1)	IN					✓	35	LVC MOS18	1.800									
Scalar ports (1)																		
ENABLE	IN				F21	✓	35	LVC MOS18	1.800									
DDR_54576 (71)	INOUT					✓	502	(Multiple)*	1.500	(Multiple)			⌕					
FIXED_IO_54576 (59)	INOUT					✓	(Multiple)	(Multiple)*	(Multiple)	(Multiple)	(Multiple)		⌕					
Scalar ports (0)																		

## 8. Implementing Design

Once the ENABLE input is mapped, click on Implementation → Run Implementation option to start the process. After that open the implemented design by clicking on Open Implemented Design.



## 9. Saving Bitstream, Debug Probe & XSA

Once the implemented design is opened, type in the following TCL command in the TCL console:

*write\_bitstream \*Path where you want to save the bitstream file\*.bit*

*write\_debug\_probes \*Path where you want to save the debug probe file\*.ltx*

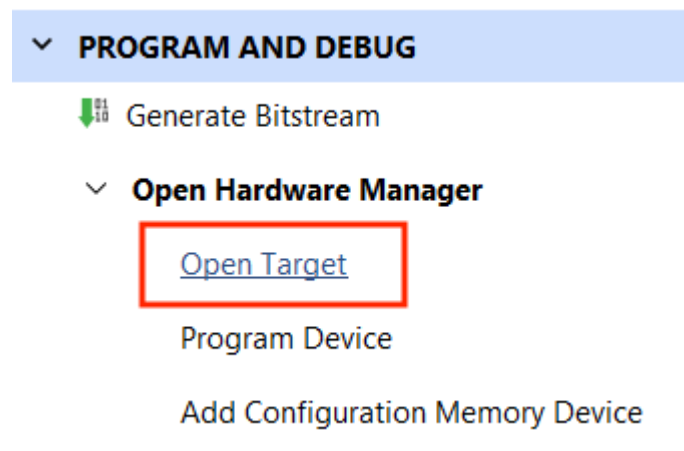
*write\_hw\_platform -fixed -force -file \*Path where you want to save the XSA file\*.xsa*

The XSA file essentially exports the necessary information about the design which will be used by the CPU for controlling the whole system.



## 10. Connecting to the FPGA

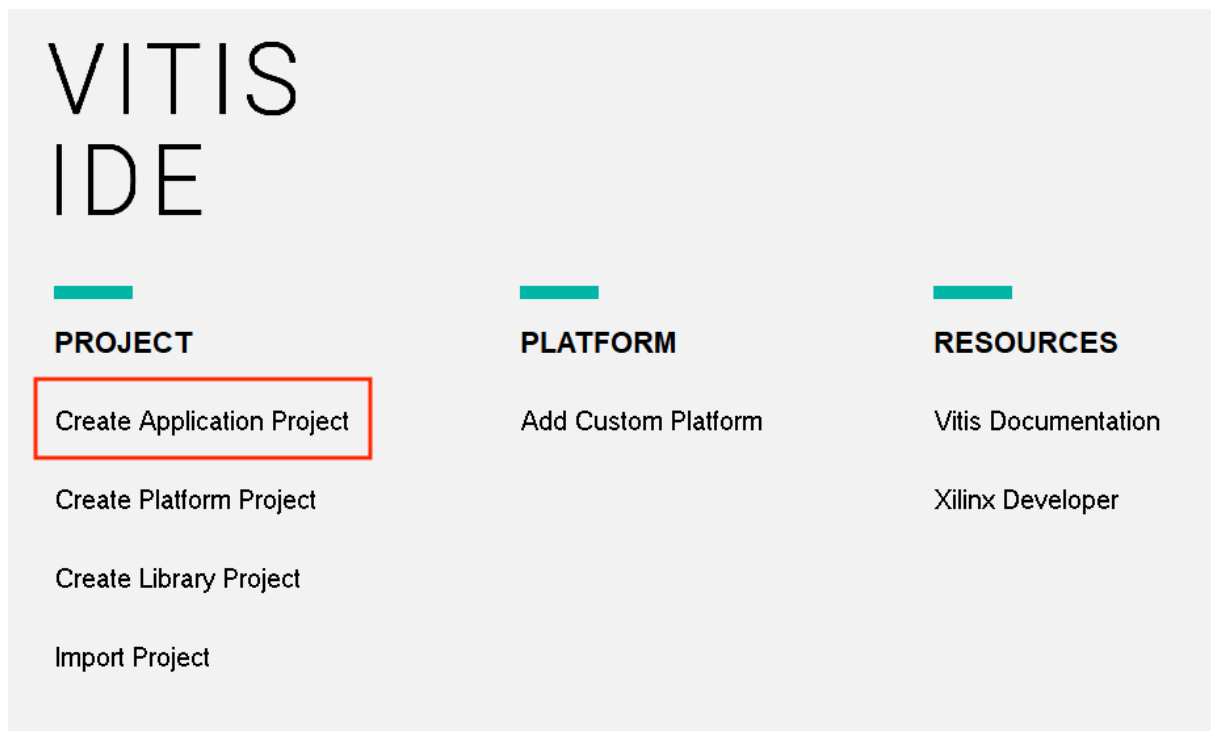
Connect the development board to your computer. In Vivado, click on 'Open Target' in the Flow Navigator window.



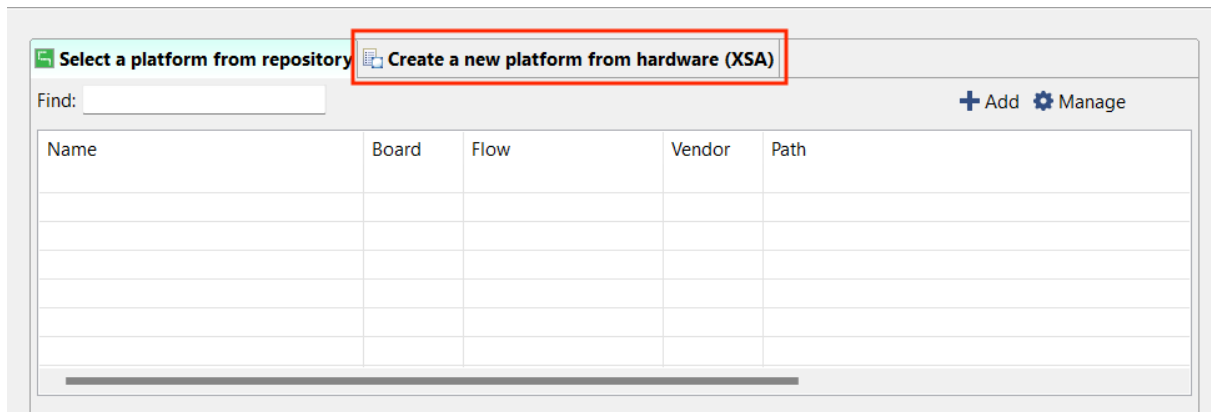
Vivado will now try to detect your development board and connect to it.

## 11. Launching Vitis

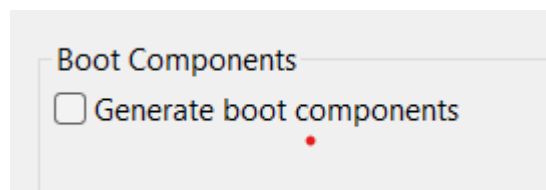
Open the Vitis software provided by Xilinx. Create a folder & choose that as the workspace. After that, click on Create Application Project as shown below.



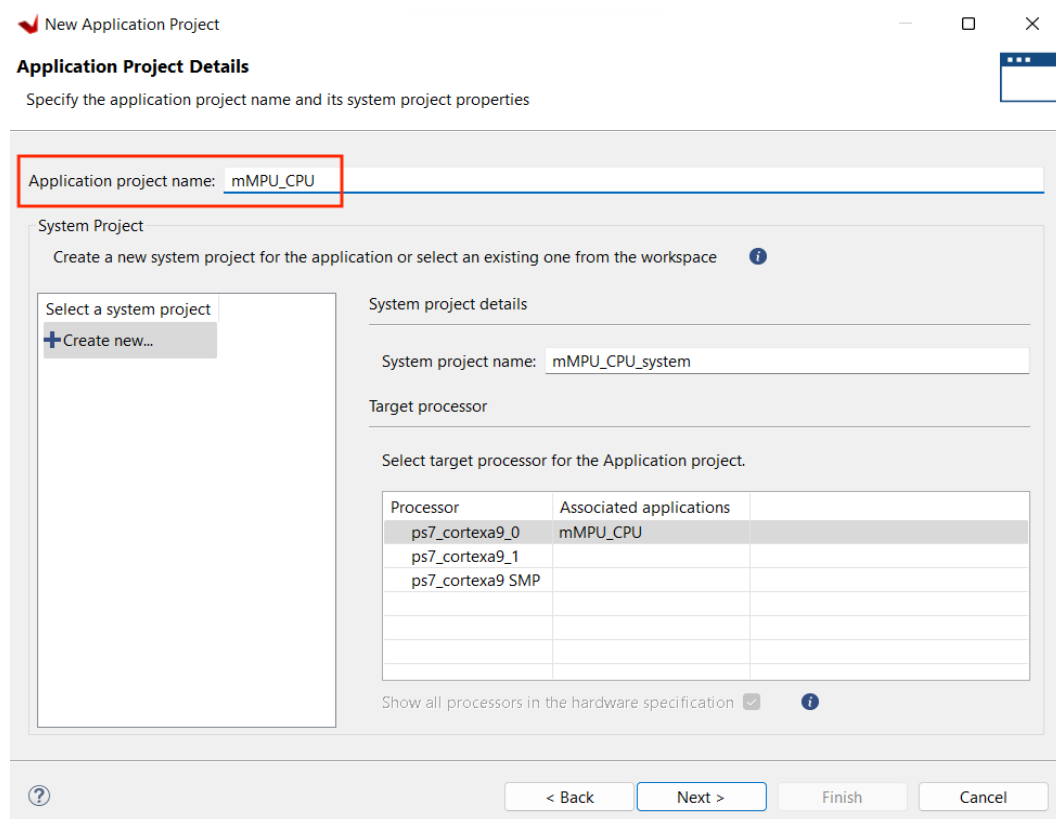
Then click on 'Create a new platform from hardware (XSA)' as highlighted in red below.



Browse & import the XSA file you saved earlier. Once the file is imported, uncheck the box to 'Generate Boot Components'.



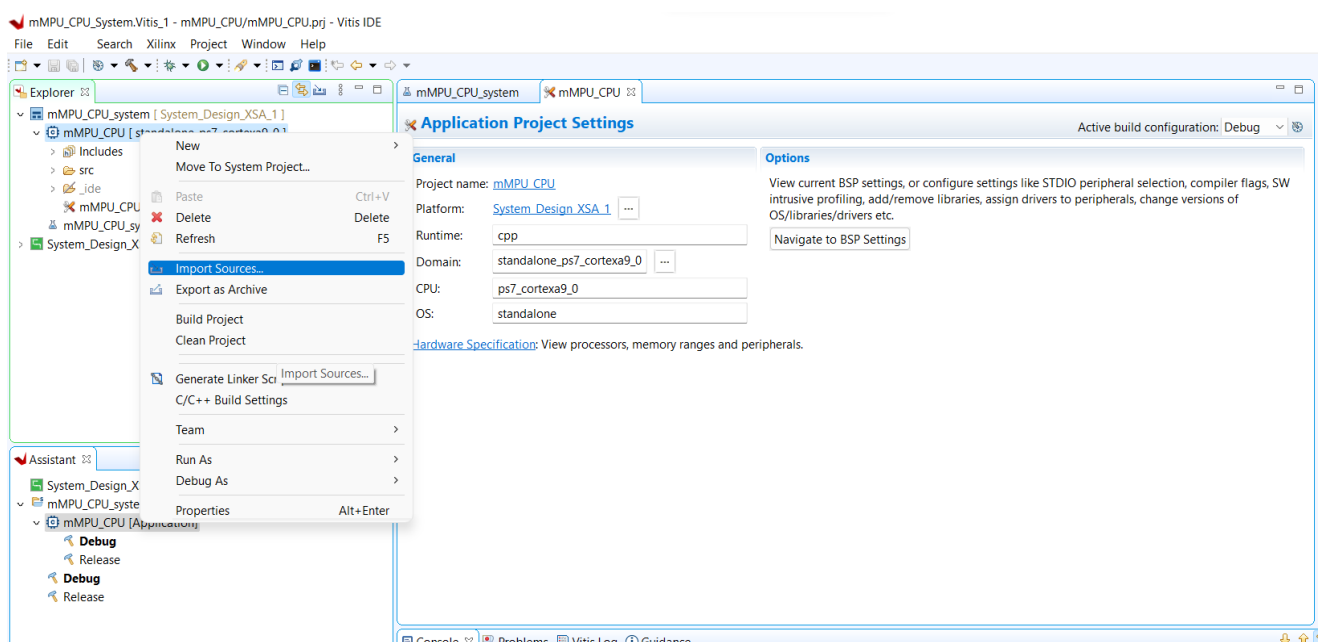
After that, give the application project a name.



Lastly, choose the template as 'Empty Application (C)' and click Finish.

## 12. Importing C code into the Project

Once the project is created, in the Explorer window, right click on the project name that you gave & click on 'Import Sources' as shown below.

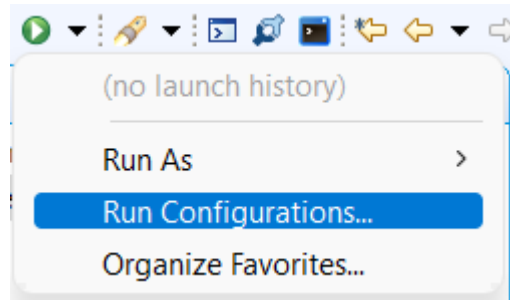


Click the folder where you extracted the zip file. The source code is located inside the folder. After selecting the folder, select the 'mMPU\_CPU\_Source\_Code.c' file.

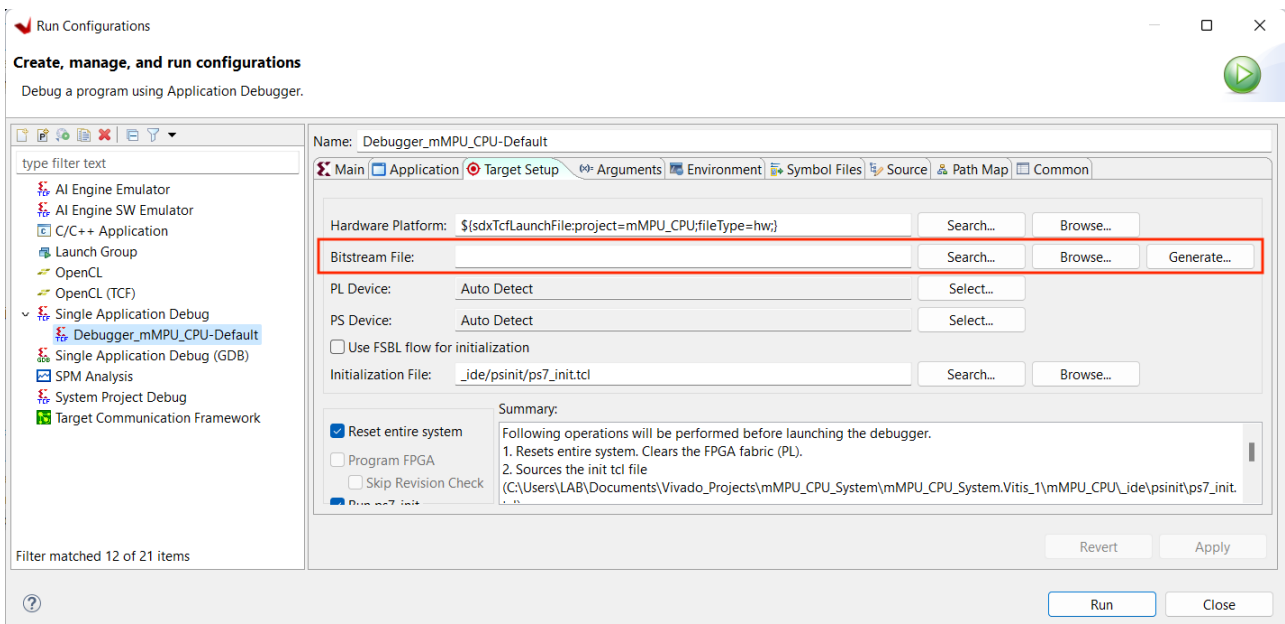
## 13. Inputting the PIM Instruction to Simulate

Once the code is imported, open the C file. In line 20, an array called PIM has been created. Enter the PIM instructions you want to simulate as binary 64-bit elements as shown below. For this example, we use the ADD\_CO instruction. In order to generate the 64-bit PIM instructions, please refer to Appendix A.





Select 'Single Application Debug' from the menu on the left & then click on 'Target Setup' tab. In the Bitstream field, click on Browse and choose the bitstream that you generated from Vivado.



## 15. Running the C code

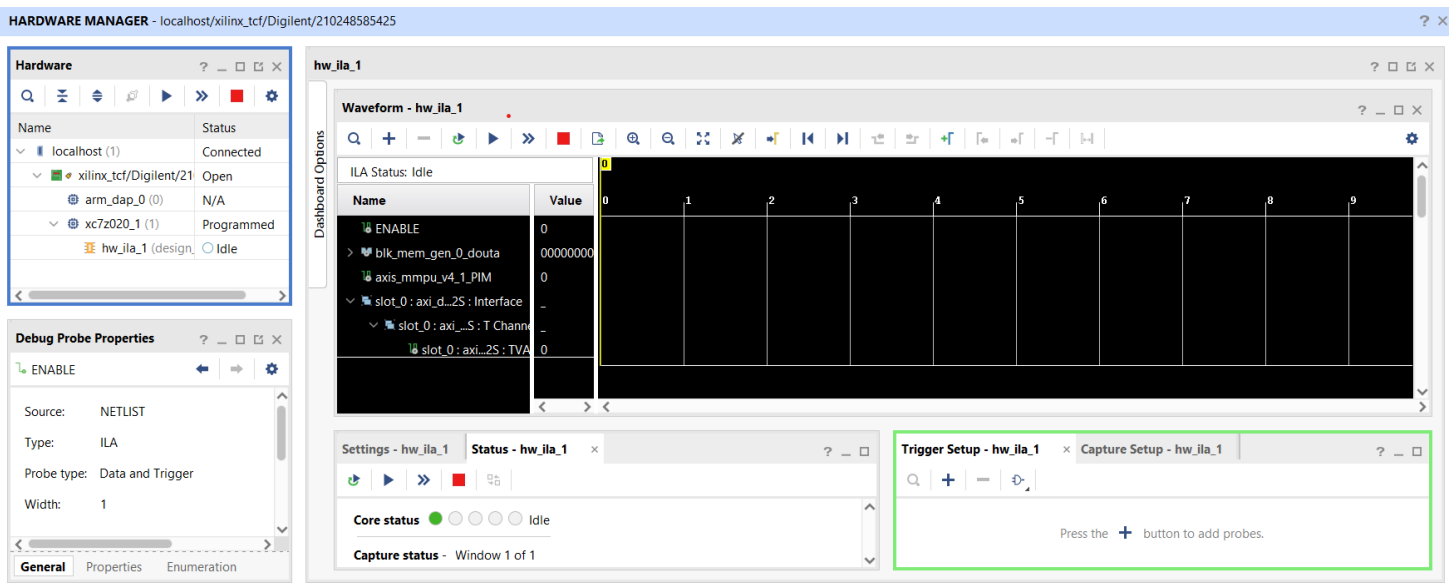
Once the connection is established, click on the 'Run' button as shown above. Since we selected the bitstream file in Vitis, the software will automatically program the FPGA as well before running the C code. On the bottom right corner, you can view the progress bar. Ensure that before clicking on run, the DIP switch used in the design is set to OFF state.

## 16. Setting up the Trigger for ILA

Once the progress bar in Vitis reaches 100%, go back to Vivado and click on 'Refresh Device' as shown below.

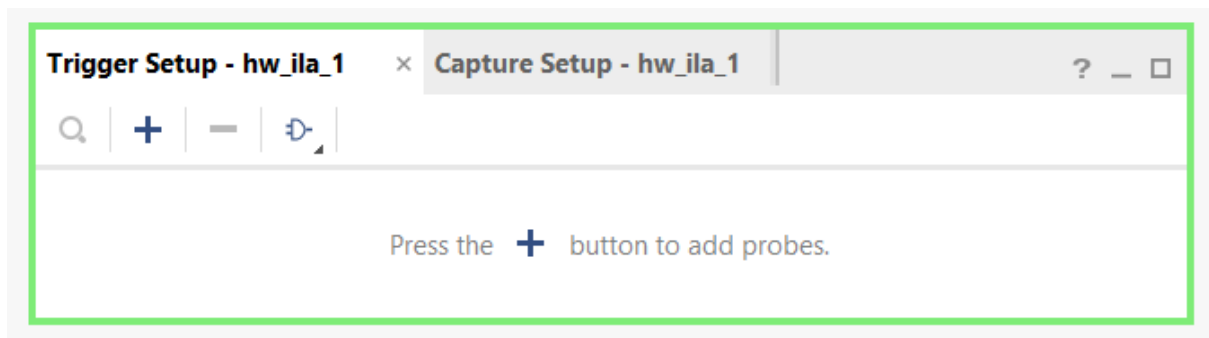


Vivado will refresh the connection to your development board and since we already programmed it using Vitis, you will get a window like this.

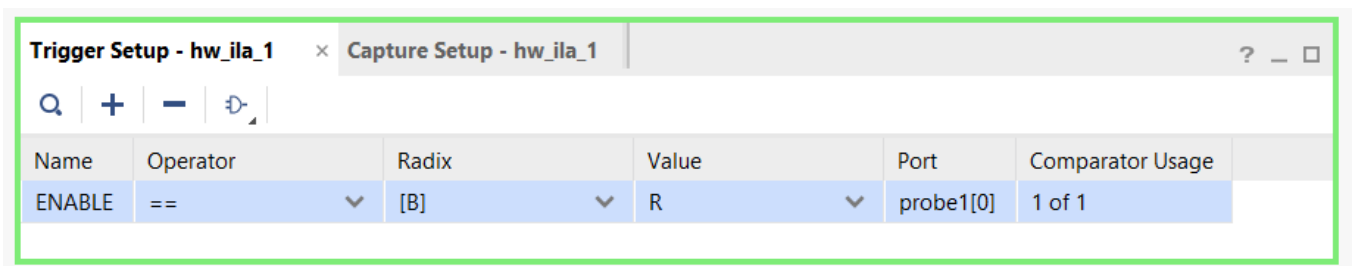


You can see that currently, we do not have any waveform on the ILA window as we haven't yet set a trigger condition for the ILA.

In the Trigger Setup window in the bottom right corner, click on the + button and choose ENABLE. Then, change the value to R which signifies the rising edge.



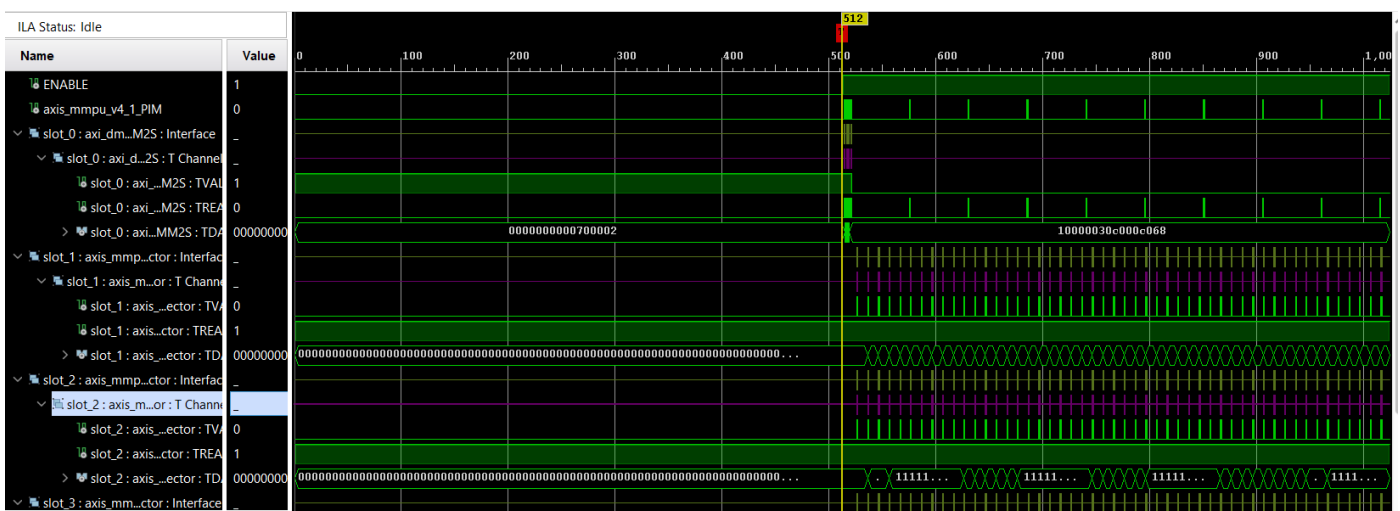
You should finally have a trigger like this in the Trigger Setup window.



## 17. Viewing the Waveform from the FPGA

Once the trigger is set, click on the 'Run trigger for this ILA core' button. This would activate the trigger condition that we just set. You would see the status of ILA changing from idle to 'Waiting for Trigger (512 out of 1024 samples)'. This essentially means that the ILA core is now waiting for the trigger condition to be met & since the trigger location is set to 512, half of the ILA is filled i.e. 512 out of 1024 samples.

On the development board, change the DIP switch from OFF state to ON state. You would see that the ILA instantly captures the trigger & displays the waveform as shown below.

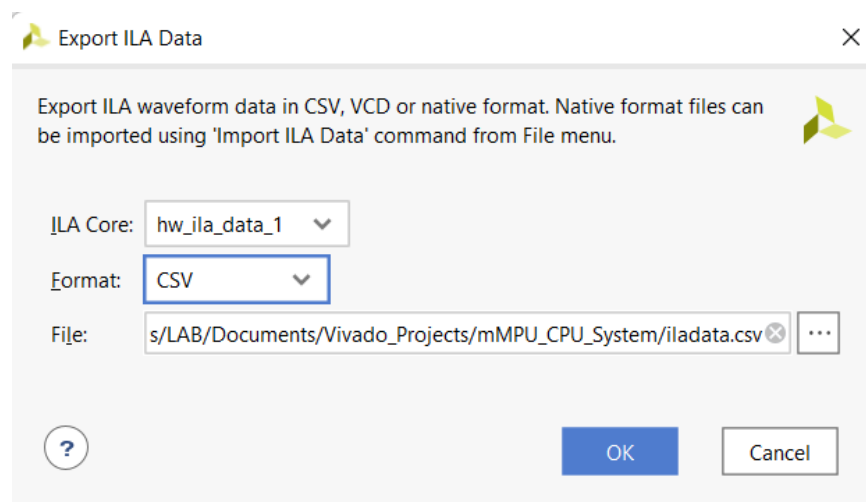


## 18. Changing Radix

Right-click on TDATA of slot\_0, slot\_1, slot\_2 & slot\_3 and change the radix from hexadecimal to binary. This is required since when we export the data in the next step, we want the PIM instruction, column, row & crossbar vectors to be exported in binary format so that we can import them directly into the mMPU Python Simulator. Hence, this step is required.

## 19. Exporting the ILA Waveform

On the ILA window, click on 'Export ILA waveform data'. This would prompt you to export the waveform in one of the supported formats. In the format option, choose CSV and save the file to a location of your choice.



## 20. The mMPU Python Simulator

The mMPU Python Simulator will allow you to check whether the sequence of column, row & crossbar vector generator by the system design performs the desired operation or not.

The mMPU Python Simulator is implemented in Google Colab, although you can download the .py file and use it in an IDE of your choice.

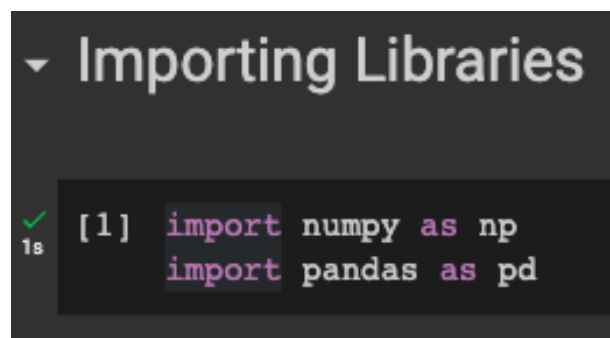


The simulator is broken down into 6 sections with the following heading:

1. Importing Libraries
2. Defining Variables
3. mMPU Content before Performing Instructions
4. Simulating mMPU
5. mMPU Content after Performing Instructions
6. Verifying the Contents of mMPU

#### **a. Importing Libraries**

This cell basically imports the necessary python libraries i.e. Numpy and Pandas.



```
▼ Importing Libraries

[1]: import numpy as np
     import pandas as pd
```

#### **b. Defining Variables**

This cell defines the necessary variables. Depending upon the mMPU controller parameters, you can vary the number of wordlines, bitlines & crossbar arrays. Then the CSV file that we exported from the ILA is imported. Ensure that the name of the file is the same as what you uploaded. From the CSV file, the row, column and crossbar vector are extracted and stored as a list. The instructions vector is also extracted from the CSV file. Lastly, we declare a numpy array of the size (wordlines, bitlines, crossbar\_arrays) and initialize them with random 0s and 1s. This would act as our mMPU.

## ▾ Defining Variables

```
[2] wordlines      = 32
    bitlines       = 32
    crossbar_arrays = 32

df = pd.read_csv("/content/iladata_OR.csv", dtype = str)
df = df.drop(df.index[0])

col_vector      = df[df.columns[11]].values.tolist()
row_vector      = df[df.columns[15]].values.tolist()
crossbar_vector = df[df.columns[19]].values.tolist()
instructions    = list(df[df.columns[6]].unique())

mMPU = np.random.randint(0,2,(wordlines,bitlines,crossbar_arrays))
```

### c. mMPU Content before Performing Instructions

This cell basically displays the content of mMPU before we perform the instructions. This is useful in case the size of the array is small. You can skip running this cell if the mMPU size is large.

## ▾ mMPU Content before Performing Instructions

```
✓ 0s [3] for array in range(crossbar_arrays):
      print(np.matrix(mMPU[:, :, array]))
```

### d. Simulating mMPU

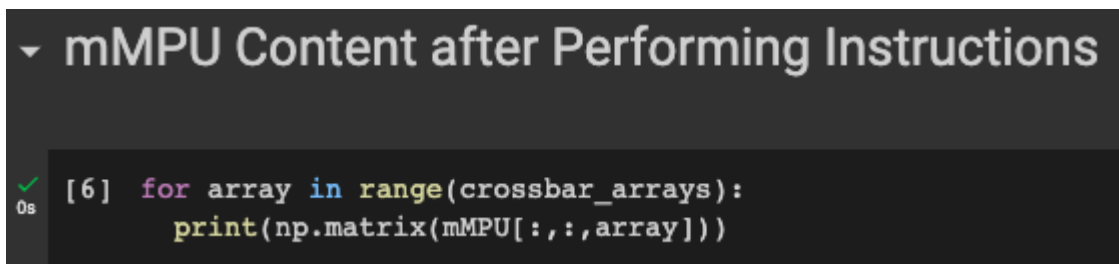
This cell is the main cell which essentially simulates the mMPU and performs instructions depending on the extracted row, column and crossbar vectors.

We first define a NOR function which we would use in case we want to perform NOR operations between various cells.

The simulator would then go over each of the row, column and crossbar vectors and determine the operation depending upon the voltage applied. The operations can range from writing 0/1 to a cell to performing MAGIC NOR between various cells.

### e. mMPU Content after Performing Instructions

This cell basically displays the content of mMPU after we perform the instructions. This is useful in case the size of the array is small & for manually verifying the content by going over each bit of the mMPU. However, as the size increases, it becomes cumbersome and time-consuming to manually verify whether the correct instructions were executed or not. Therefore, you can skip running this cell if the mMPU size is large.



```
0s [6] for array in range(crossbar_arrays):  
      print(np.matrix(mMPU[:, :, array]))
```

### f. Verifying the Contents of mMPU

In order to verify the content of mMPU with a large size, the last cell performs this function. Using the instructions extracted from the CSV file, the cell determines which instructions were performed and then verifies the content of the mMPU according to that. Right now, the code can verify 6 instructions which are AND, OR, XOR, ADD\_CO, SUB\_BO & ANDI. Support for further instructions from the PIM ISA will be added in the future.

Assuming we ran the simulator for ADD\_CO instruction, upon running this cell, you should either see an output stating that 'Addition operation was correct for Row = *row number* at Crossbar Array = *crossbar array number*' or 'Addition operation was wrong for Row = *row number* at Crossbar Array = *crossbar array number*'. This would verify that the sequence of voltages generated by the mMPU controller in the system design indeed performed the instructions correctly.

# APPENDIX A

The python script is a multi-utility tool that will allow us to define and construct many preliminary steps before carrying out simulations and eventual hardware implementation. The figure below specifies the files present in the mMPU\_uCode\_Controller repository. The script was run on Pycharm IDE. (Can use any other IDE of your choice).

This PC > Desktop > Thesis_Work > Microcoded_mMPU_controller_v1 > 04_Python > mMPU_uCode_Controller				
Name	^	Date modified	Type	Size
.idea		05-07-2022 00:46	File folder	
venv		05-07-2022 00:47	File folder	
instructions		04-07-2022 14:45	Text Document	2 KB
main		04-07-2022 14:45	Python File	35 KB
microinstructions		17-02-2022 18:18	Text Document	1 KB
Microinstructions_binary		04-07-2022 14:46	Text Document	2,177 KB
PIM_instructions_binary		03-07-2022 15:33	Text Document	1 KB

```
Welcome to Microcode mMPU Controller
```

```
Choose one of the following :
```

- 1 - Display PIM instructions.
- 2 - Add new PIM instructions.
- 3 - Edit Binary Instruction file.
- 4 - Display Microinstructions.
- 5 - Edit Microinstruction Memory.
- 6 - Display Microinstruction Memory.
- 7 - End.

```
Choose one of the following :
```

On running the script, you are presented with the above 6 options:

1. Display PIM instructions – Presents all the instructions present in the *instructions.txt* file.
2. Add new PIM instructions – To add new user defined instructions into the pre-existing *instructions.txt* file.
3. Edit Binary Instruction file – To generate a binary version/file of PIM instructions to be sent to the controller.
4. Display Microinstructions – To display predefined microinstructions that the control can work with.
5. Edit Microinstruction Memory – To edit the corresponding microinstruction sequence of a PIM instruction.
6. Display Microinstruction Memory - Presents the microinstructions stored in the *Microinstructions\_binary.txt* file.

## 1. Display PIM instructions

The below table specifies the available PIM instructions along with their opcode, function code, mapped location, etc.

	Type	Instruction	Opcode	Function code	Jump_Ld	Location
0	0	set	0000	0		0
1	1	reset	0001	0		1
2	2	set cmask	0010	0		2
3	3	set mask	0011	0		3
4	4	set temp	0100	0		4
5	5	lui	0101	0		5
6	6	pim ld	0110	0		6
7	6	pim st	0110	1		7
8	6	lb	0110	2		8
9	6	lh	0110	3		9
10	6	lw	0110	4		10
11	6	ld	0110	5		11
12	6	sb	0110	6		12
13	6	sh	0110	7		13
14	6	sw	0110	8		14
15	6	sd	0110	9		15
16	7	hcopy	0111	0		38
17	7	vcopy	0111	1		39
18	7	not	0111	2		40
19	7	hcopy neg	0111	3		41
20	7	vcopy neg	0111	4		42
21	7	and all	0111	5		43
22	7	or all	0111	6		44

## 2. Add new PIM instructions

You can add new PIM instructions of your choice using this option limited to L/S, V, VV, I Type instructions. After providing the name for your instruction, enter the starting and the ending location of your microcode sequence that will be edited in the control memory.

```
Choose one of the following :
2
Instruction Types

6 - L/S Type
7 - V Type
8 - VV Type
9 - I Type

Enter type of instructions (6-9) : 7
Enter instruction name : Instruction1
Enter starting microinstruction location: 800
Enter microinstruction end location: 810
```

The following are the contents of a) *instructions.txt* (left), b) *microinstructions\_binary.txt* (right) after runtime. The jump\_id instructions are automatically added at the next available free location for the same instruction type.

```
7,or all,0111,6,44,,
7,Instruction1,0111,7,45,800,810
8,and,1000,0,54,151,158
```

45	00000000000000000000000000000000
46	000000110010000000111000000011010
47	00000000000000000000000000000000

## 3. Edit Binary Instruction file

This step is important to generate a PIM instruction file, to be used for Vivado Simulations and the subsequent FPGA device



Choose one of the following :

4

	Type	microInstruction	Opcode
0	0	ubr	0000
1	0	mov	0001
2	0	sri	0010
3	0	ret	1000
4	0	bnz	1001
5	0	bz	1010
6	0	bl	1011
7	0	jump	1100
8	0	jump ld	1101
9	1	NOT	00
10	1	NOR	01
11	1	SET	10
12	1	RESET	11

## 5. Edit Microinstruction Memory

This step allows us to edit the contents of a particular PIM instruction, stored in the control memory (*Microinstructions\_binary.txt*). Choose the required index number of the PIM instruction. This provides the starting address of the microinstructions sequence and the number of microinstructions to be added. (Derived from details provided in *instructions.txt*).



Choose one of the following :

5

Do you want to add more microinstructions to the list? (y/n)

y

	Type	Instruction	Opcode	Function code	Jump_Ld	Location
0	0	set	0000	0		0
1	1	reset	0001	0		1
2	2	set cmask	0010	0		2
3	3	set mask	0011	0		3
4	4	set temp	0100	0		4
5	5	lui	0101	0		5

Select the instruction index to edit its microinstruction sequence

23

Microcode starting address - 800

Number of Microinstructions to be added - 11

	microInstruction
0	ubr
1	mov
2	sri
3	ret
4	bnz
5	bz
6	bl
7	jump
8	jump ld
9	NOT
10	NOR
11	SET
12	RESET

The prompt then asks the user to choose the type of microinstruction and the supporting fields required to construct the binary version of the same. The program is continued in a loop until the last microinstruction is added to the *Microinstructions\_binary.txt* file.

```

Choose one of the following microinstructions : 5
Branch_addr (absolute address) : 822
FLAG Types

0 - Col
1 - Ziw1
2 - Ziw2
3 - Zimm
4 - Co
5 - Call
6 - Sign
flag (0-7) : 1

microInstruction
0          ubr
1          mov
2          sri
3          ret
4          bnz

```

## 6. Display Microinstruction Memory

After entering the index of PIM instruction, it displays the micro-coded sequence of that particular instruction in a binary format.

```

Enter the index of instruction for microinstruction display: 39
['000000001011100010000100000010100']
['0000000010110110000000110000010100']
['000000000110000000011000000000101']
['00000000000000000000100000001001']
['0000000000000000000000000000101']
['000000000000000000001100000000001']
['000000001011011010000000000011000']
['0000000000000000000000000000111']
['0000000000000000000000110100000']
['000000001011101000000010000010100']
['0000000000000000000000000000100']
['000000001011001010000000000011000']
['0000000000000000000000000000111']
['0000000000000000000000110100000']
['000000001011100010000010000010010']
['000000000000000000000000000010000']

```

## REFERENCES

- S. Kvatinsky et al., "MAGIC—Memristor-Aided Logic," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 61, no. 11, pp. 895-899, Nov. 2014, doi: 10.1109/TCSII.2014.2357292.
- R. B. Hur and S. Kvatinsky, "Memory Processing Unit for in-memory processing," 2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH), Beijing, China, 2016, pp. 171-172, doi: 10.1145/2950067.2950086.