CS411 Project 2 - FPGA

KAIST School of Computing

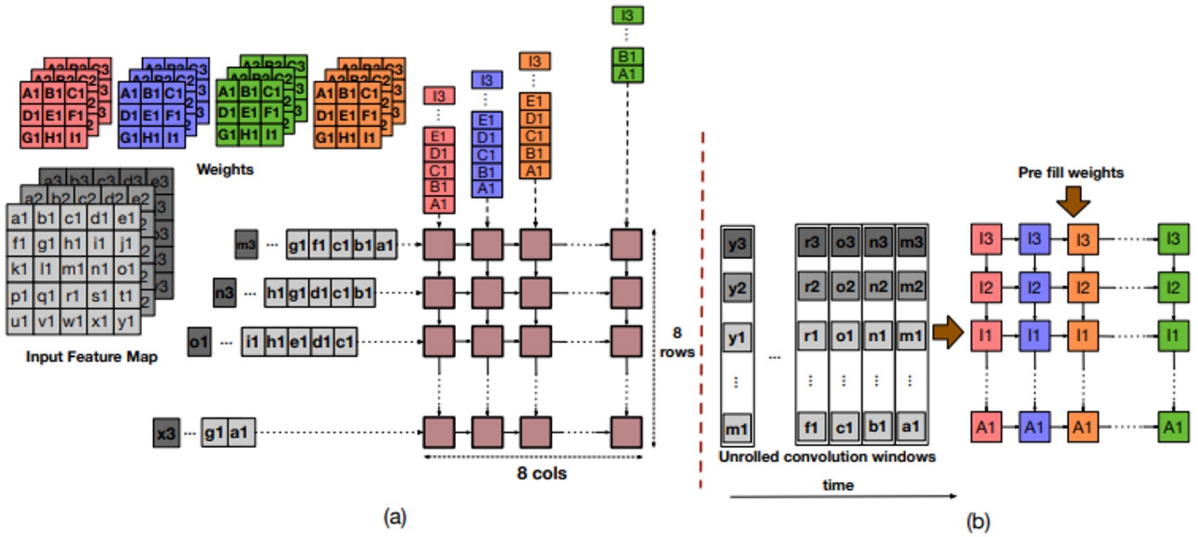
CASYS | KAIST Computer Architecture & System Lab

Contents

Systolic Array Operation

Systolic Array Operation

- **Operation mode {Weight Stationary, Output Stationary}**



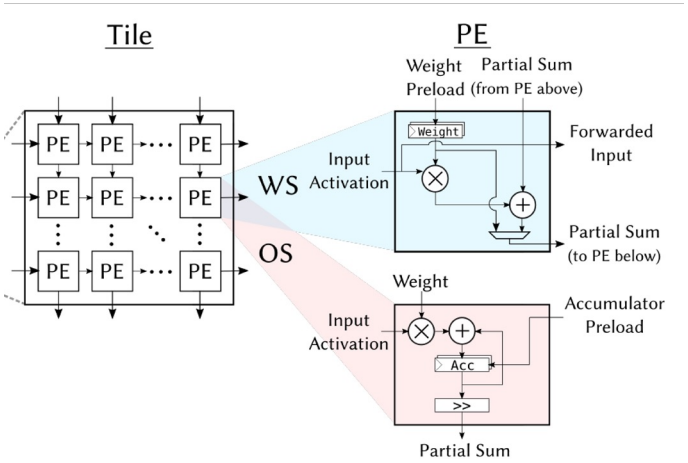
Samajdar, Ananda, et al. "Scale-sim: Systolic cnn accelerator simulator." *arXiv preprint arXiv:1811.02883* (2018). fig.2

Systolic Array Operation

- **Operation mode {Weight Stationary, Output Stationary}**

  PE and Systolic Array can operate in two modes {WS, OS}

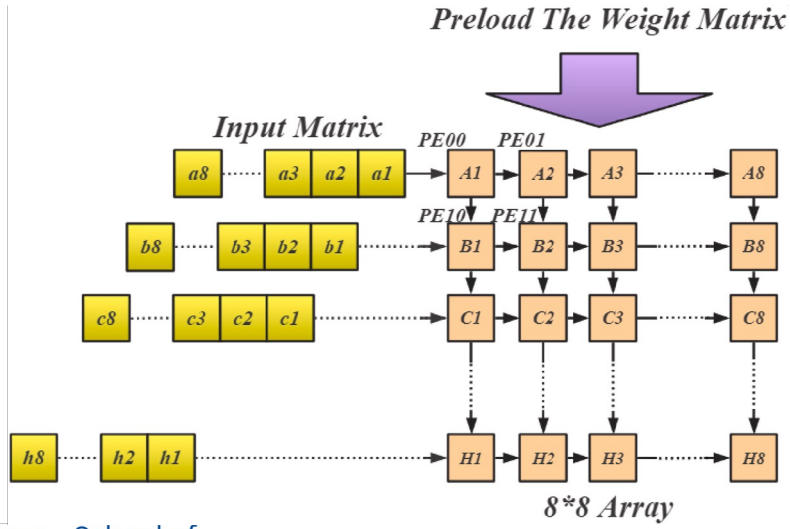
  The two modes can be switched using a control signal.

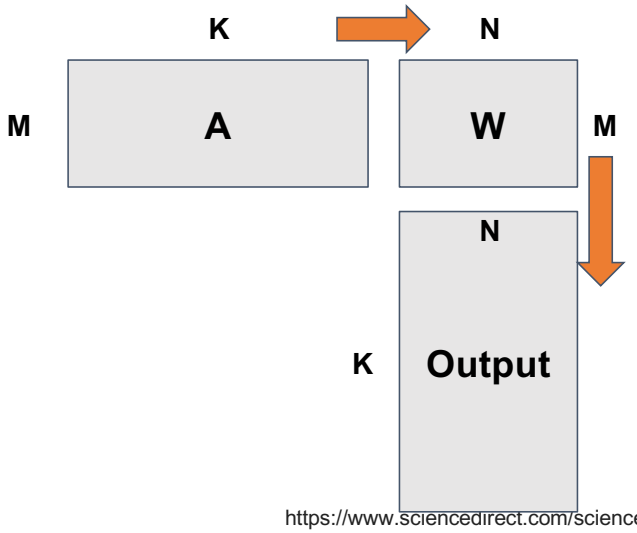


H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, Y. S. Shao, B. Nikoli´c, I. Stoica, and K. Asanovi´c. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. arXiv:1911.09925 [cs.DC], 2019.

Systolic Array Operation

- **Operation mode {Weight Stationary}**
  - Pre-load weight matrix into the systolic array.
  - Flow the activation matrix into the systolic array.
  - Accumate partial sums in the accumulator.
  - Store the accumulated result to RAM.

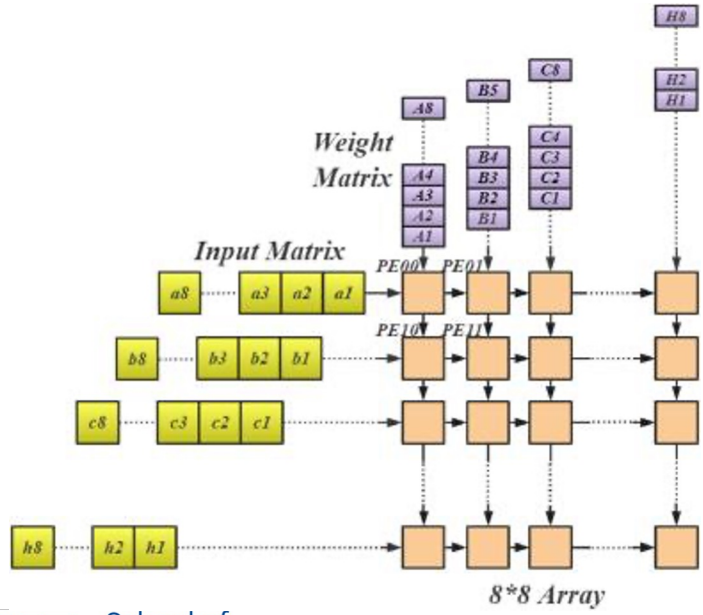


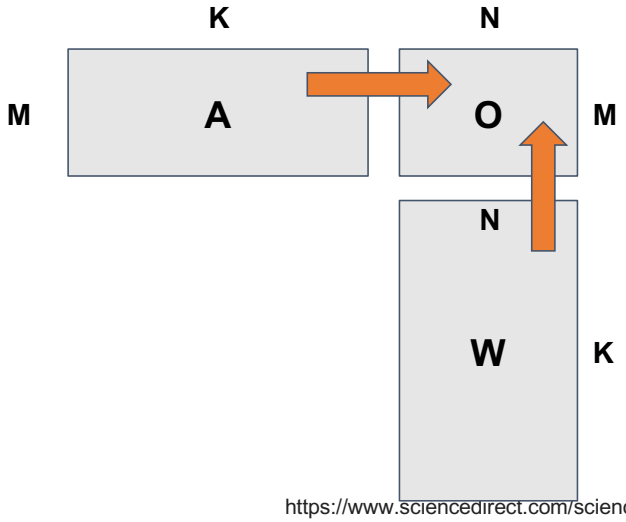


https://www.sciencedirect.com/science/article/pii/S0167926022000359

Systolic Array Operation

- **Operation mode {Output Stationary}**
  - Stream skewed activation and weight data into the systolic array
  - After all dot-product results are accumulated in the each PE, drain the results into RAM





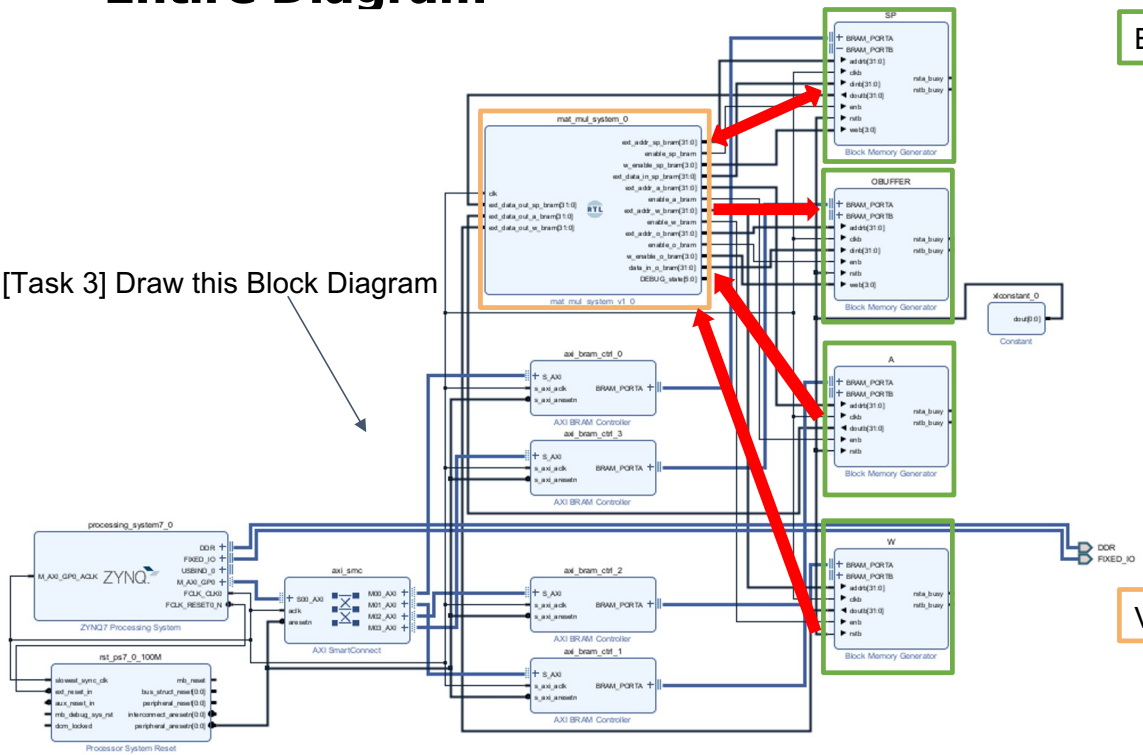
https://www.sciencedirect.com/science/article/pii/S0167926022000359

Systolic Array Modules

Systolic Array Modules

- **Entire Diagram**

[Task 3] Draw this Block Diagram



BRAM

- A/W_Bram
  - store activation/weight and feed data to ram
- O_Bram
  - store result (from systolic array)
- SP_Bram (you must follow this rule)
  - addr 0 : start signal (1: start, 0: end)
  - addr 4 : mode (1: OS, 0: WS)
  - addr 8 : M
  - addr 12: K
  - addr 16: N
  - addr 100: end signal

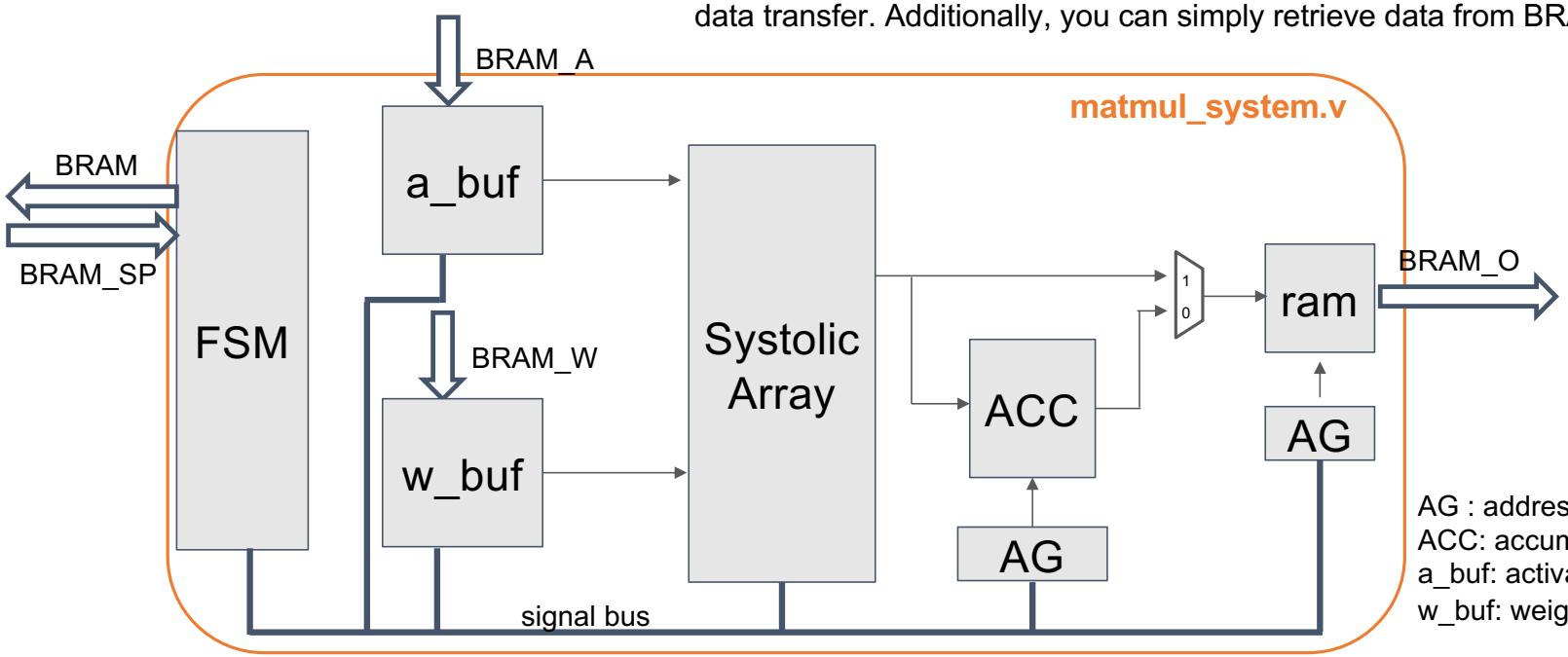
Verilog Module (Your Task 1,2)

Describe “matmul_system” in the next pages.
In verilog coding, your goal is making matmul.v

Systolic Array Modules

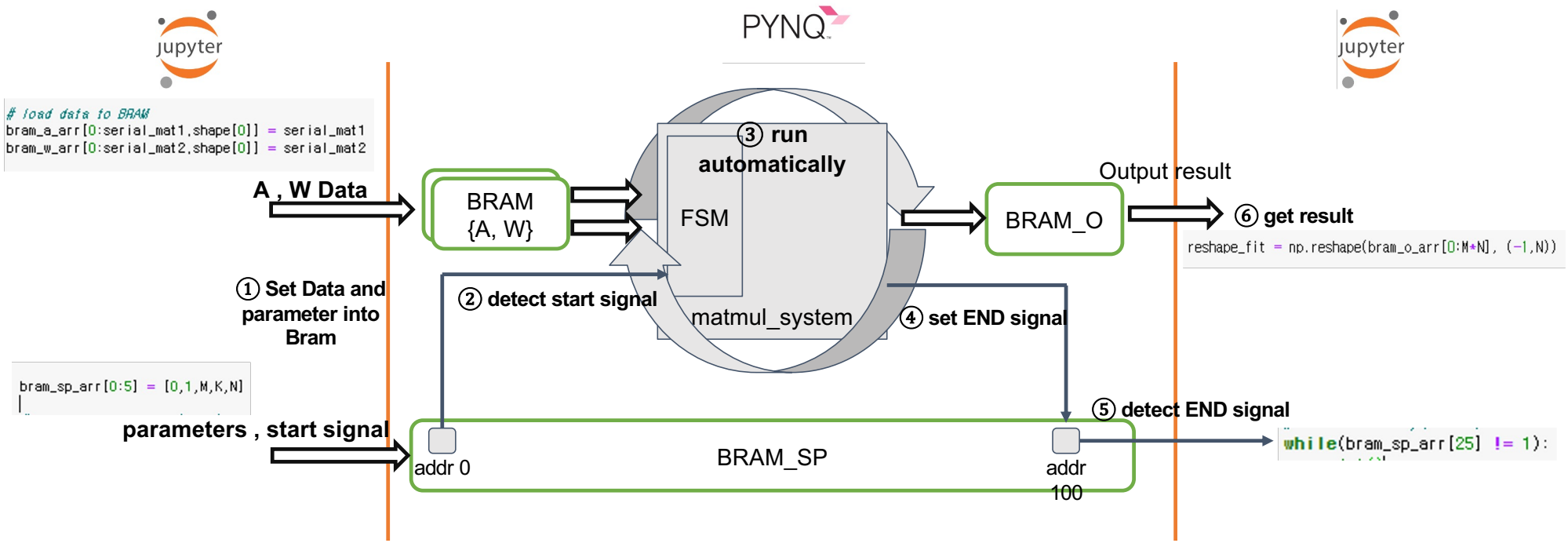
- **Example Diagram**

Ideally, we would create an AXI interface for direct data transfer from DRAM to RAM using DMA.
However, since we haven't covered that yet, we're using BRAM to handle data transfer. Additionally, you can simply retrieve data from BRAM.



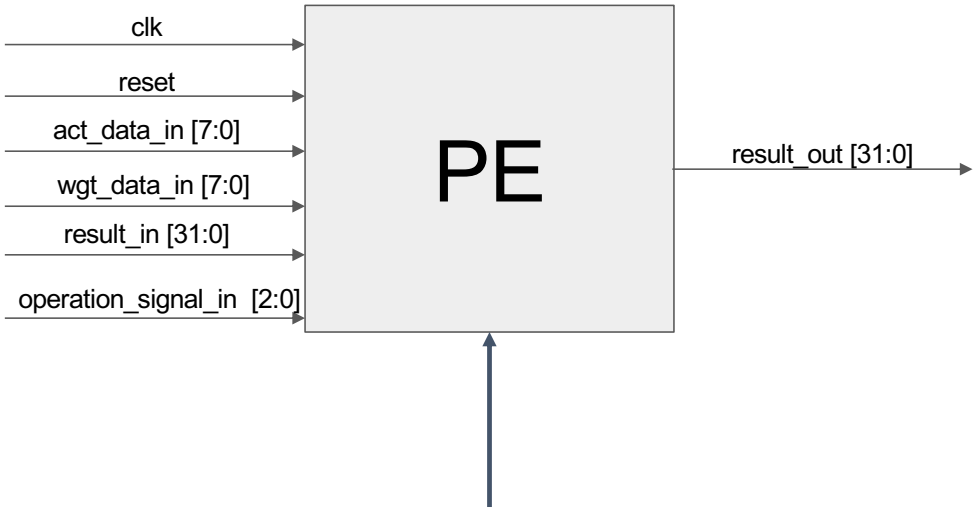
AG : address generator
ACC: accumulator
a_buf: activation buffer
w_buf: weight buffer

Systolic Array Modules



Systolic Array Modules

- **PE**

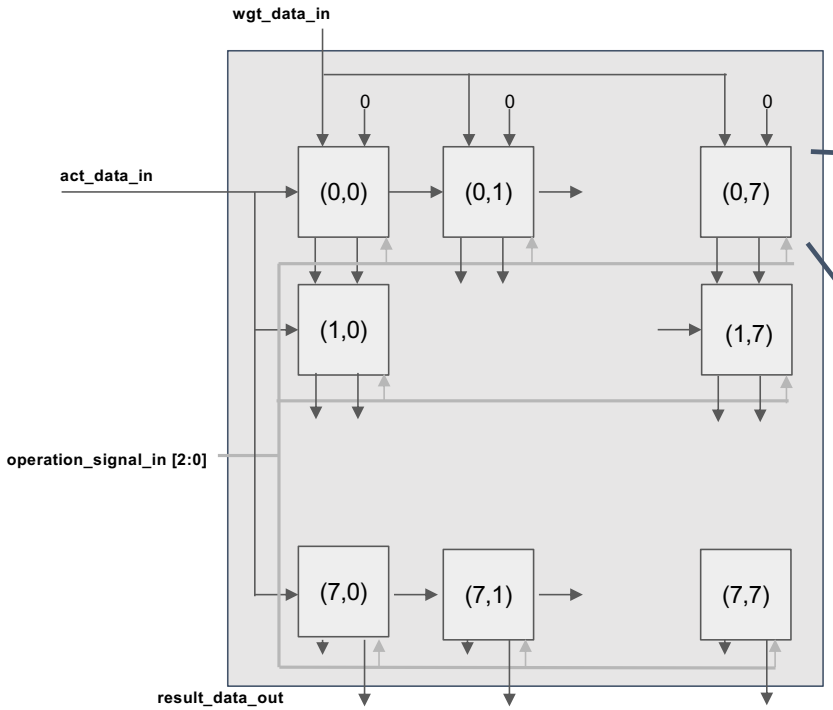


" Design your PE !! "

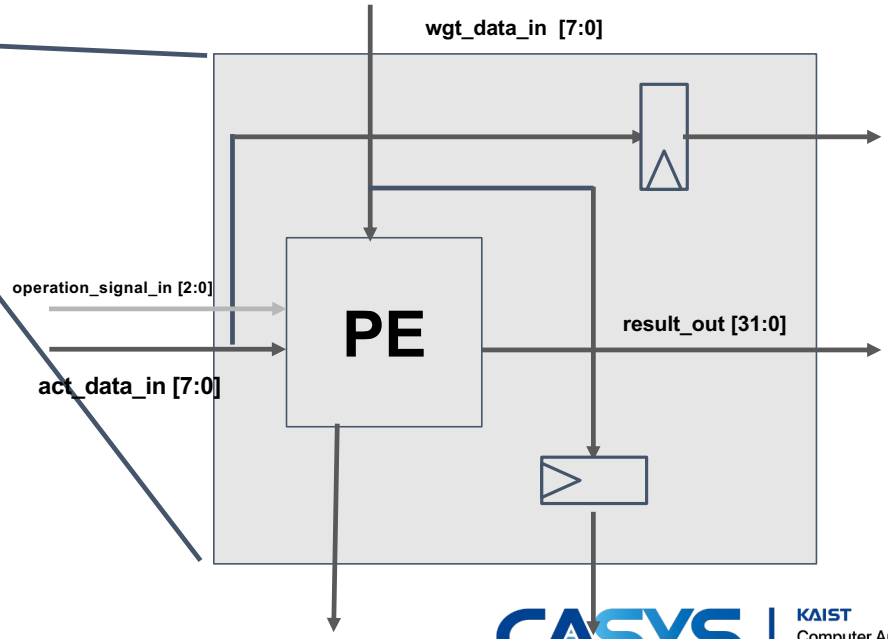
- **I/O ports are fixed**
  - act_data_in
    INT8 activation data port
  - wgt_data_in
    INT8 weight data port
  - result_in
    INT32
  - result_out
    INT32
  - operation_signal_in
    control signal for PE data flow
    {WS : weight preload, activation flow}
    {OS : data flow, drain result}

Systolic Array Modules

- **SYSTOLIC ARRAY (8x8 example)**



- **Operation**
  - mode : {OS, WS}
  - input data stream every one cycle.



Systolic Array Modules

- **Index/Address Generator (optional)**
  - This module serves the function of determining where to store incoming data in RAM and the accumulator, as well as indicating which data stream out to system.
  - The input is provided by the enable signal sent from the FSM. When the enable signal is activated, it automatically generates the addresses required for the operation of the systolic array. The output consists of address and the enable signal for RAM and the accumulator.

Bram Example + Block Diagram

Bram Example

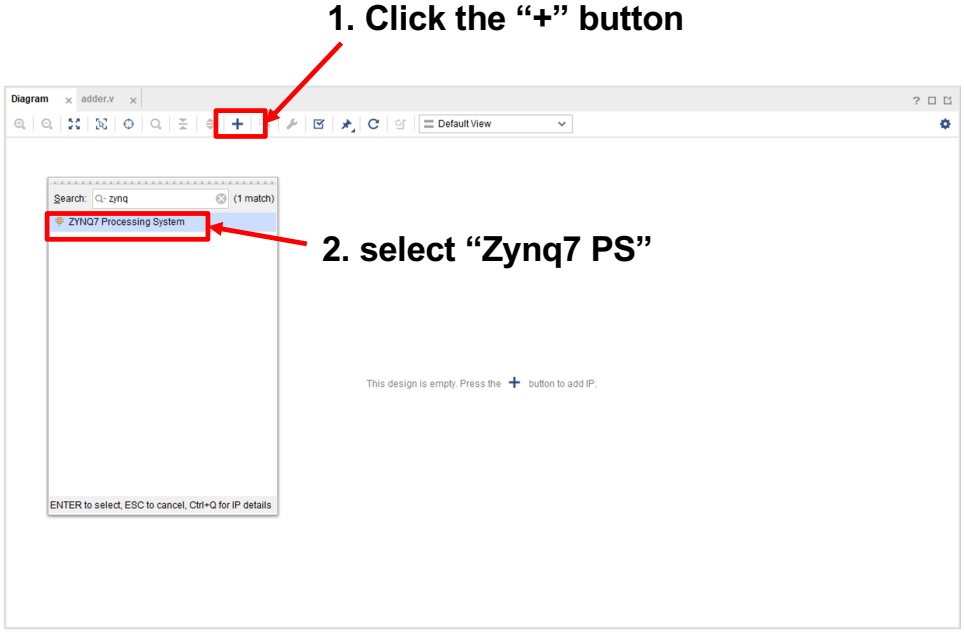
- Create the VIVADO project.

  This link explain how to create VIVADO project with PYNQ-Z2 Board

  https://discuss.pynq.io/t/tutorial-creating-a-hardware-design-for-pynq/145

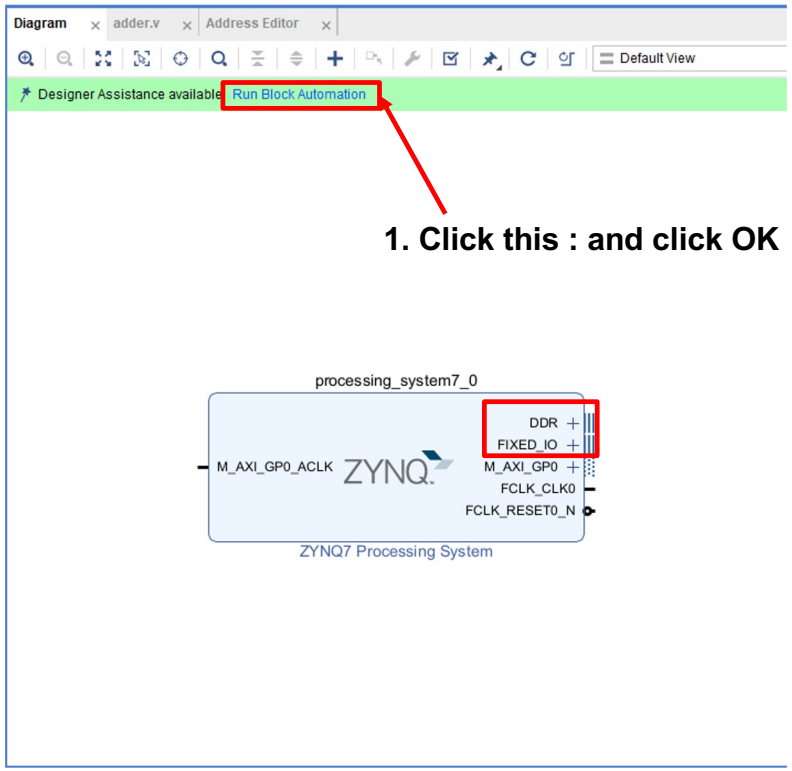
This example was executed on
the following version of VIVADO.

Bram Example



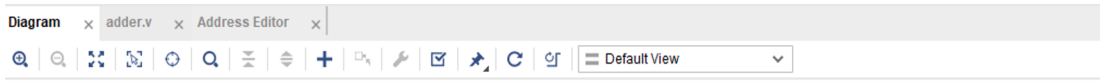
- After you make new project, build the Block Diagram
- Add Zynq7 PS that is one part of PYNQ-Z2 board.

Bram Example

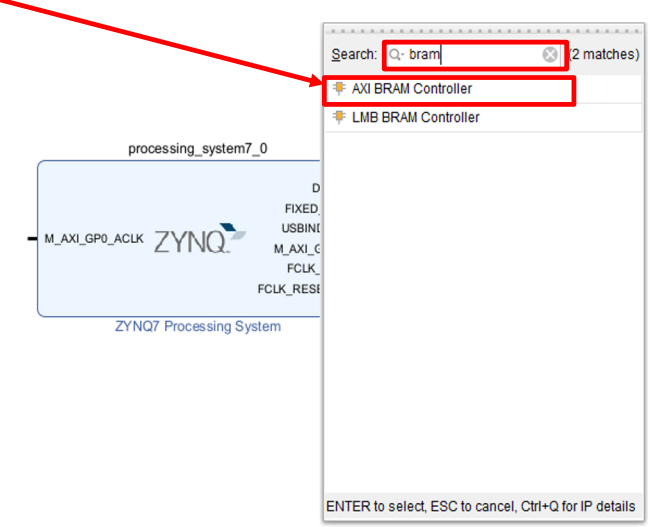


- **Run Block Automation**
  - Make DDR, FIXED_IO external port
  - There is no need to know detail about this.

Bram Example

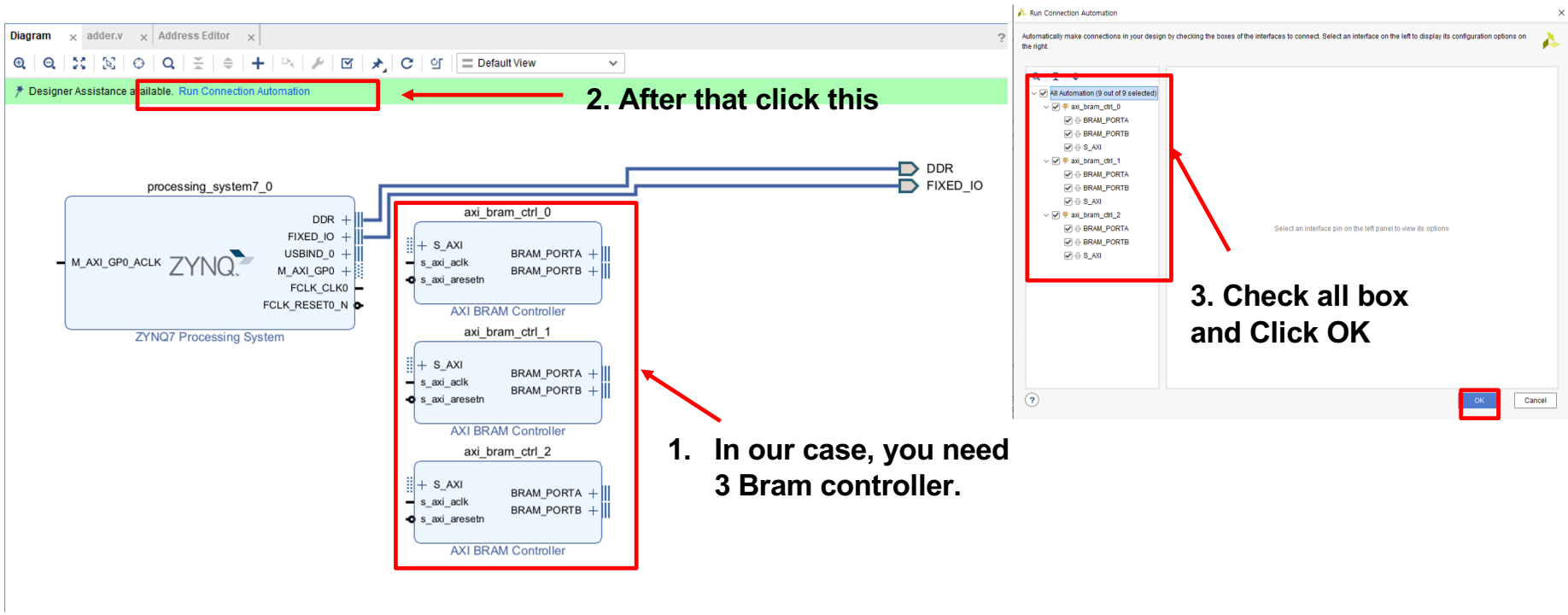


Search “AXI BRAM Controller” and click

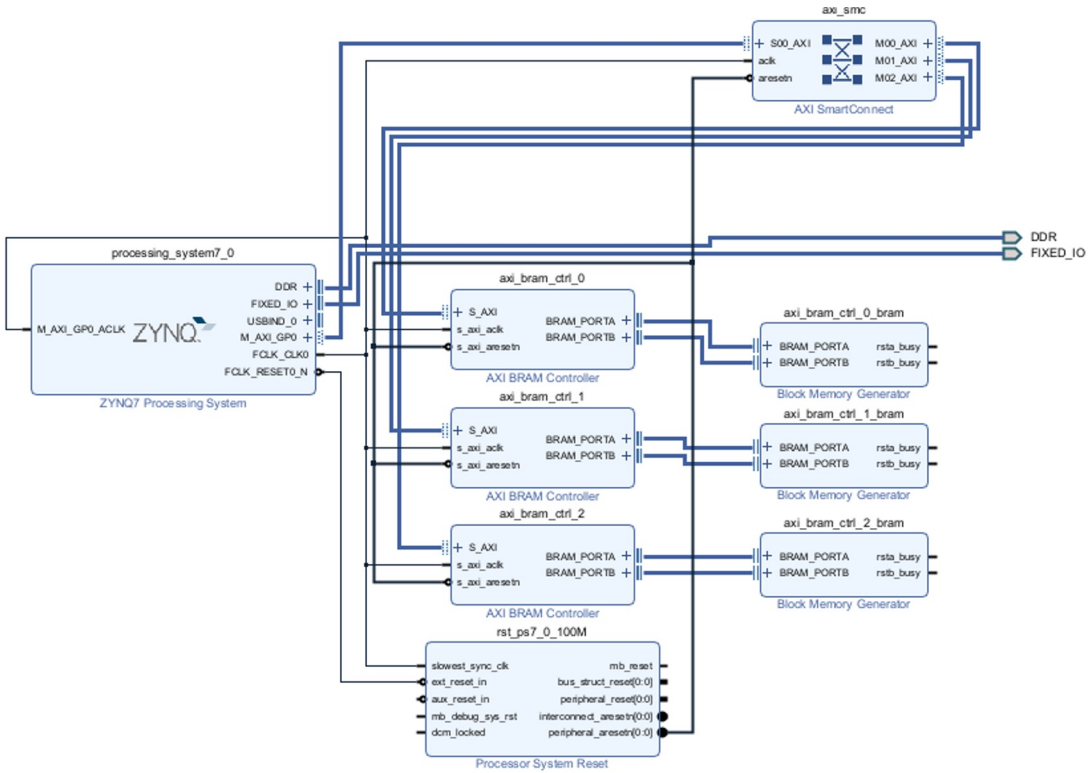


- **Add AXI BRAM Controller.**
  - This IP manages data transfer to/from the PS and stores/retrieves it from BRAM

Bram Example



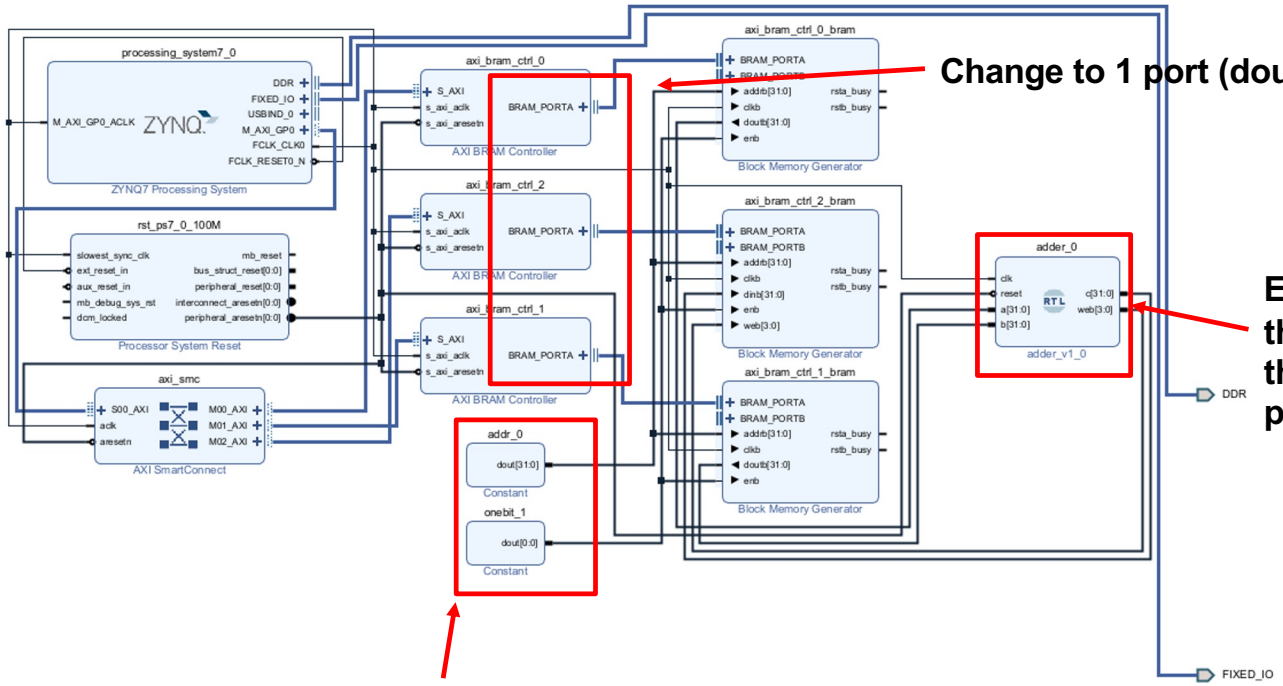
Bram Example



Then, you can see this scheme.

Bram Example

- **Entire Scheme**



Change to 1 port (double click the IP image)

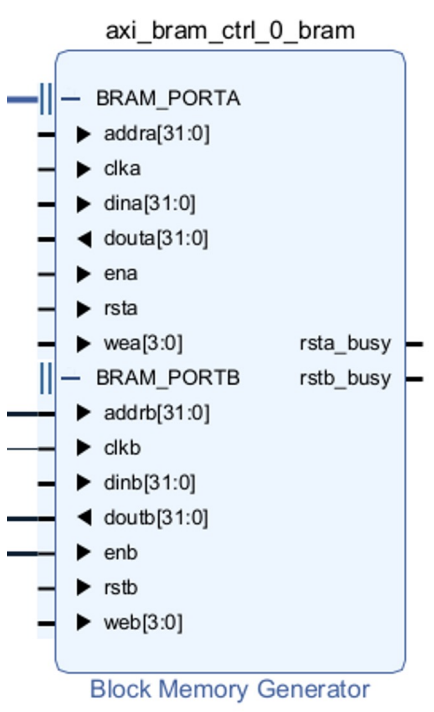
Establish a connection between the ‘adder module, as provided in the project file, and the BRAM port

Add IP “Constant”. If you double click the IP, you can modify constant value and its width.

Bram Example

Bram Port

- addr : address of Bram
- clk : clock signal
- din : data_in
- dout : data_out
- rst : reset
- wea[3:0] : write enable (1 bit → 1 byte)



(read)

addr = [address]

dout = (you can get data from here)

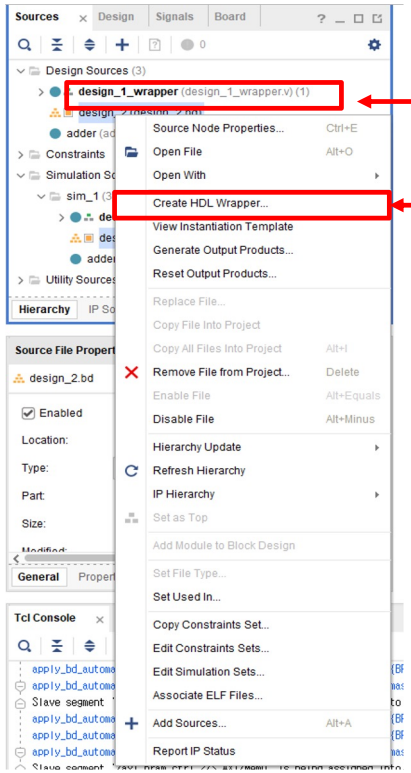
(write)

addr = [address]

wea = 4'b1111

din = [4 bytes DATA]

Bram Example

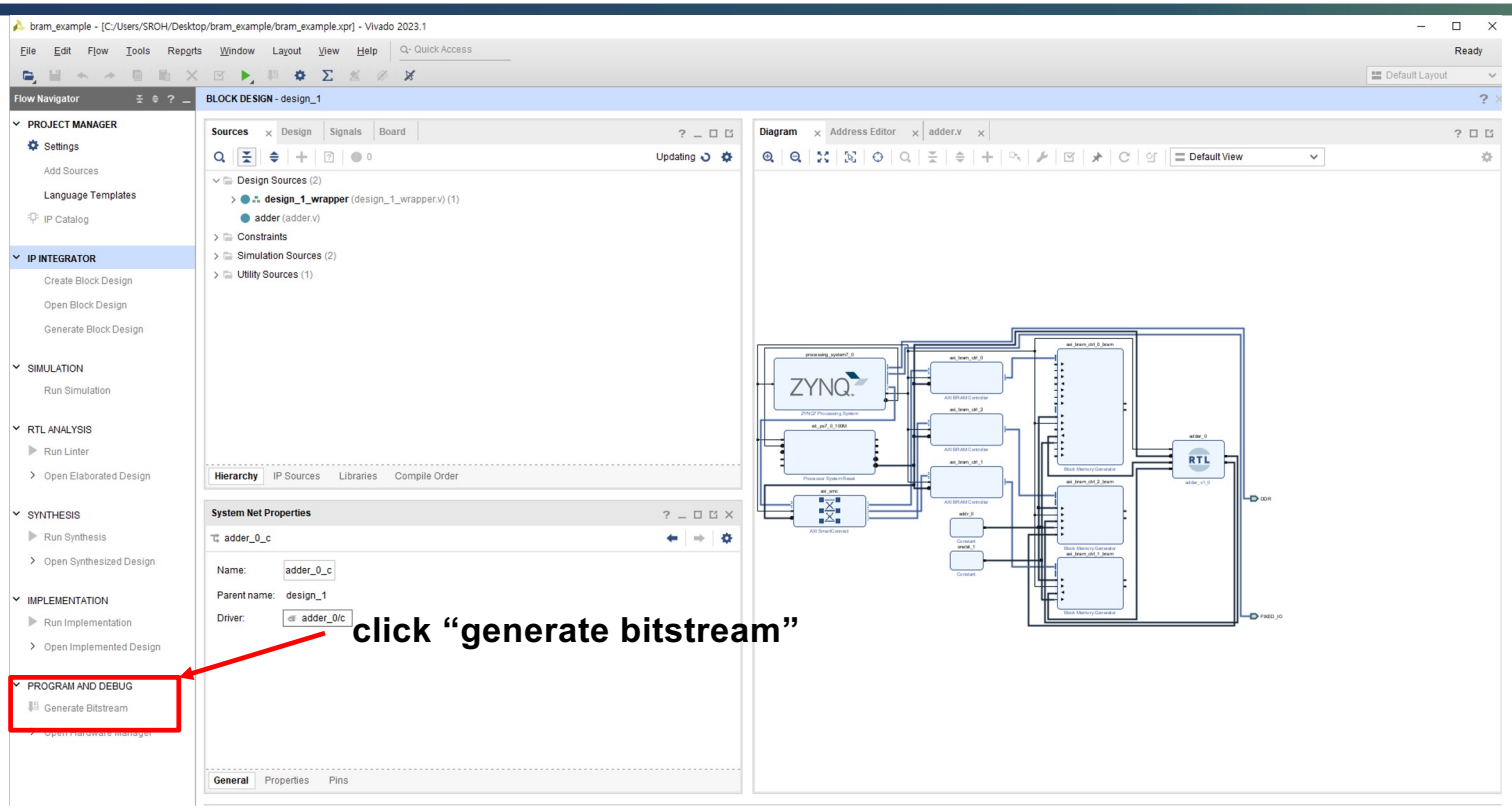


Make wrapper verilog file from your Block Diagram

2. you can make this verilog file : verilog file of Block Diagram

1. click

Bram Example



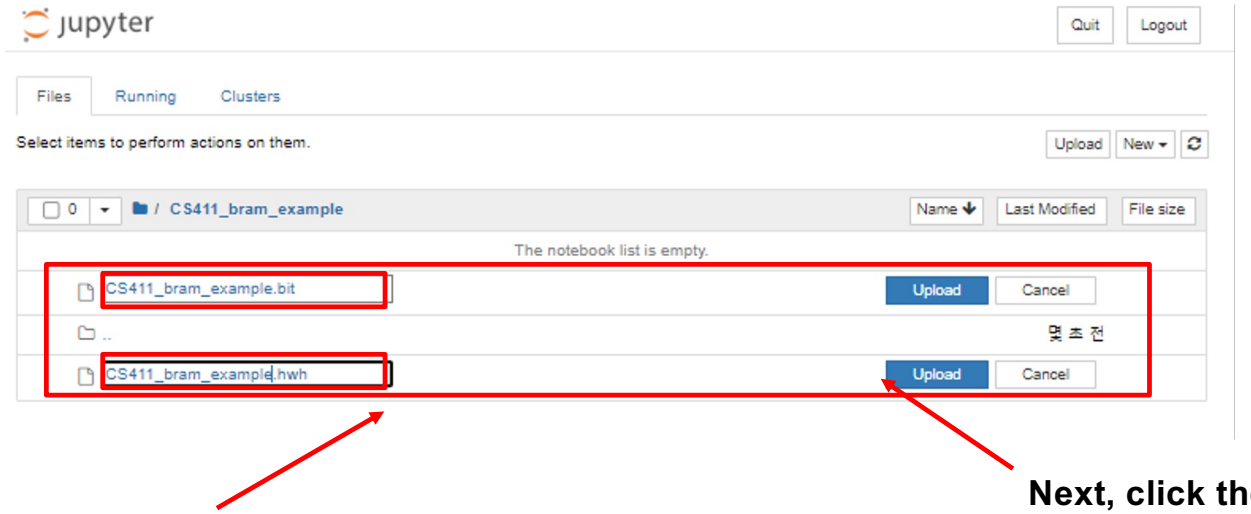
Bram Example

In VIVADO project folder, please search for the ‘bit stream file’ and ‘hwh file’ using the keywords ‘bit’ and ‘hwh.’

Once located, select these two files.

Next, open the Jupyter notebook and move these selected files into the Jupyter notebook folder.

Bram Example



Upload two file, and change file_name as you want.
In this case, name is "CS411_bram_example"

Next, click the Upload button

Bram Example

Plus, make new Notebook

Bram Example

```
In [1]: from pynq import Overlay
        ol = Overlay("CS411_bram_example.bit")

In [2]: ol?

In [3]: bram_in1 = ol.axi_bram_ctrl_0.mmio.array
        bram_in2 = ol.axi_bram_ctrl_1.mmio.array
        bram_out = ol.axi_bram_ctrl_2.mmio.array

In [4]: print(bram_in1[0:4])
        print(bram_in2[0:4])
        print(bram_out[0:4])
        [0 0 0 0]
        [0 0 0 0]
        [0 0 0 0]

In [5]: bram_in1[0]=5
        bram_in2[0]=6
```

your bitstream file name

axi_bram_ctrl_x come from here

You can read value from BRAM

You can write value into BRAM

ol? generate this

```
Type:            Overlay
String form:     <pynq.overlay.Overlay object at 0xb3899da8>
File:            /usr/local/share/pynq-venv/lib/python3.10/site-packages/pynq/overlay.py
Docstring:
Default documentation for overlay CS411_bram_example.bit. The following
attributes are available on this overlay:

IP Blocks
----------
processing_system7_0 : pynq.overlay.DefaultIP

Hierarchies
-----------
adder_0              : pynq.overlay.DefaultHierarchy

Interrupts
----------
None

GPIO Outputs
------------
None

Memories
------------
axi_bram_ctrl_0      : Memory
axi_bram_ctrl_1      : Memory
axi_bram_ctrl_2      : Memory
PSDDR                : Memory
```

Bram Example

```
In [5]: bram_in1[0]=5
        bram_in2[0]=6

In [6]: print(bram_out[0])
11
```

Our adder module read data from addr0 of bram_in1 and bram_in2, and store the result into addr0 of bram_out

https://www.youtube.com/watch?v=zbumITJQ2Z8

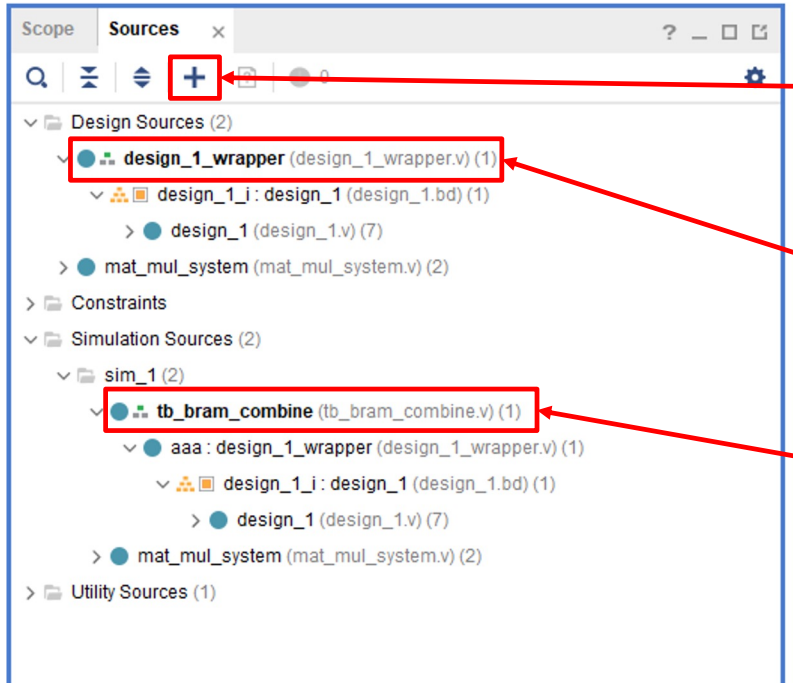
[Using PYNQ MMIO (Memory Mapped IO)]

“Now you can communicate with Bram.”

Vivado Simulation

Vivado Simulation

- **Set Design source file and Simulation source file**



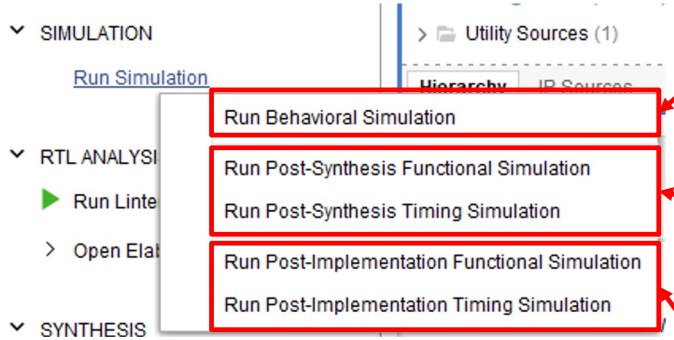
This button is for adding new source file (modules or testbenches.)

To set the proper file as the Design Source, right-click on it and select ‘Move to Design Source.’ This file will be used during the synthesis, implementation, and bitstream writing processes.

To set the proper file as the Simulation Source, right-click on it and select ‘Move to Simulation Source’

Vivado Simulation

- **Simulation**
  - prepare test bench file (.v)



(necessary)
You can run the simulation using your 'module.v' and 'testbench.v' files. The simulation is based on the RTL files.

(necessary)
You can run the simulation after synthesis. First, synthesize the module and prepare 'testbench.v.' The simulation will be based on the netlist, which may produce different results from the behavior simulation.

(not necessary)
You can run the simulation after implementation. Please note that if your module has more than 125 I/O ports. you may not be able to proceed with the implementation step.

Vivado Simulation

- **Behavior Simulation**
  - prepare test bench file (.v)

100ns/1ns is recommended
timescale: #1 = 100ns
simulate every 1ns

declare ports :
input → reg
output → wire

Make the instance of your module that you want to simulate.

```
1  `timescale 100ns/1ns
2
3  module tb_bram_combine ();
4    wire [5:0]DEBUG_STATE;
5    reg[31:0]a_addr;
6    reg [31:0]a_data;
7    reg [3:0]a_wen;
8    reg clk;
9    reg reset;
10   reg [31:0]o_addr;
11   wire [31:0]o_data_out;
12   reg [31:0]sp_addr;
13   reg [31:0]sp_data_in;
14   reg [3:0]sp_web;
15   reg [3:0]w_web;
16
17
18   design_1_wrapper aaa (
19     .clk(clk),
20     .reset(reset),
21     .a_addr(a_addr),
22     .a_data(a_data),
23     .a_wen(a_wen),
24     .sp_addr(sp_addr),
25     .sp_data_in(sp_data_in),
26     .sp_web(sp_web),
27     .w_web(w_web),
28     .STATE(DEBUG_STATE),
29     .O_BRAM_DATA(o_data_out),
30     .O_BRAM_ADDR(o_addr)
31   );
32
```

```
initial begin
    clk = 1;
    reset = 1;
    a_addr =0;
    a_data=0;
    a_wen=0;
    o_addr=0;
    sp_addr=0;
    sp_data_in=0;
    sp_web=0;
    w_web=0;
    #10
    ...
end

always #0.5 clk = !clk;
```

assign value to input port

Wait 100x10 ns (10 cycle)

Clock signal value change
every 0.5x100ns
T = 100ns = #1

Vivado Simulation

- **Post-synthesis Simulation**
  - You have to synthesize your module
  - This simulation is bases on the synthesis result
  - If you make changes to your Verilog code or block diagram, you'll need to synthesize it again.
  - Ensure you have the testbench file, which should be the same as the one used for behavioral simulation.
  - (Confirmed) The post-synthesis simulation result corresponds to the same operation as the behavior simulation.

About Verilog

Module

```
`timescale 100ns/1ns
module pe #(
    parameter integer   ACT_WIDTH       = 8,
    parameter integer   WGT_WIDTH       = 8,
    parameter integer   MULT_OUT_WIDTH  = ACT_WIDTH + WGT_WIDTH,
    parameter integer   PE_OUT_WIDTH    = 32,
    parameter integer   OP_SIG_WIDTH    = 3
) (
    input wire                                  clk,
    input wire                                  reset,
    input wire [ OP_SIG_WIDTH   -1 : 0]         operation_signal_in,

    input wire [ ACT_WIDTH      -1 : 0]         act_data_in,
    input wire [ WGT_WIDTH      -1 : 0]         wgt_data_in,
    input wire [ PE_OUT_WIDTH   -1 : 0]         result_in,

    output wire [ PE_OUT_WIDTH   -1 : 0]        result_out

);


// MAKE YOUR LOGIC HERE

endmodule
```

module name

parameters: you can declare them here and use them when you build logic

Here, You can declare I/O ports.

In this part, you can build your logic.

You can declare wires and regs, and build combinational or sequential logics.

(we recommend drawing diagram before writing code)

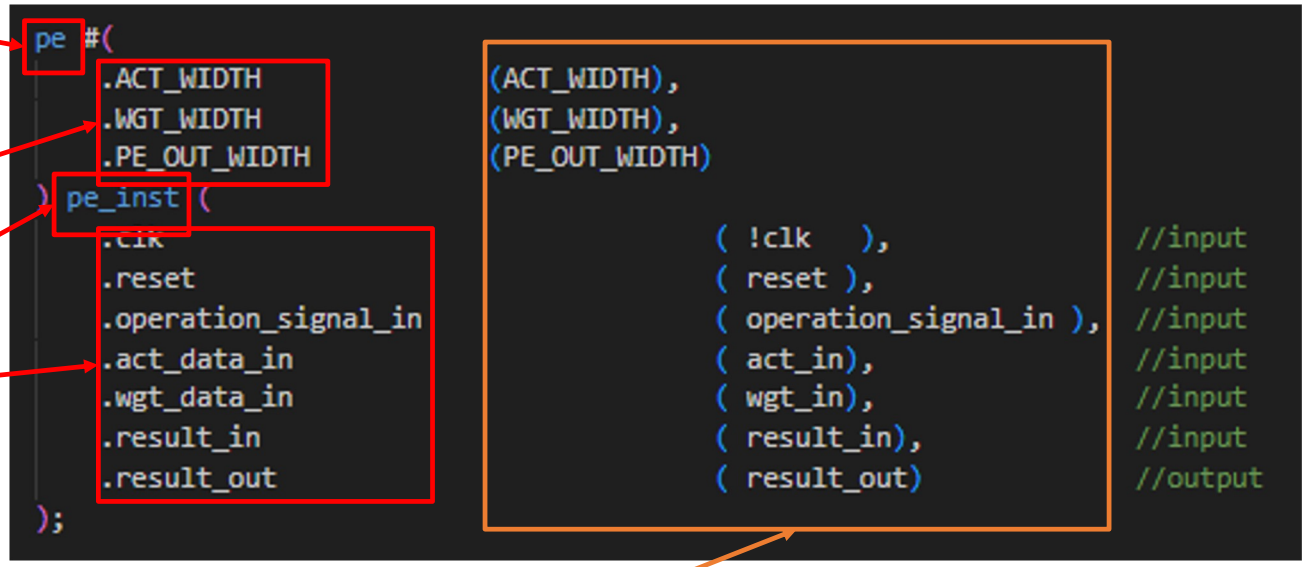
How to make instance of module

The name of module

Name of parameter (same as names in module declaration)

The name of instance

Name of I/O (same as names in module declaration)



In items enclosed within parentheses represent parameters, as well as wires or registers that have been declared at the location where this instance is utilized.

always @

```
reg [ WIDTH - 1 : 0] reg_inner;

always @(posedge clk)
begin
    if(reset)
        reg_inner <= 'b0;
    else if (enable)
        reg_inner <= in;
end
```

This example is simple register has reset and enable option.

Meaning of code: Every rising edge of clock, if reset is 1, then write 0 into reg_inner, and if reset is 0 and enable is 1, then write 'in' into reg_inner

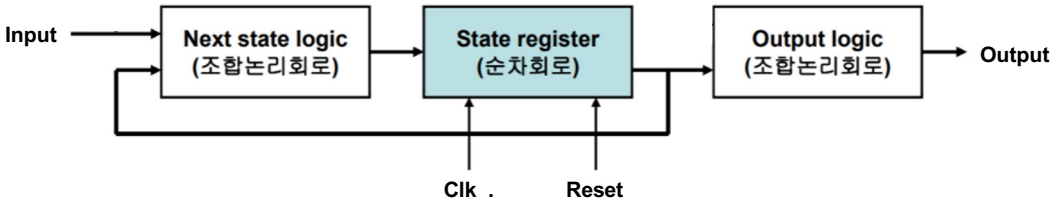
FSM

- **FSM(Finite State Machine)**
  - Input + Current_state ⇒ Next_state

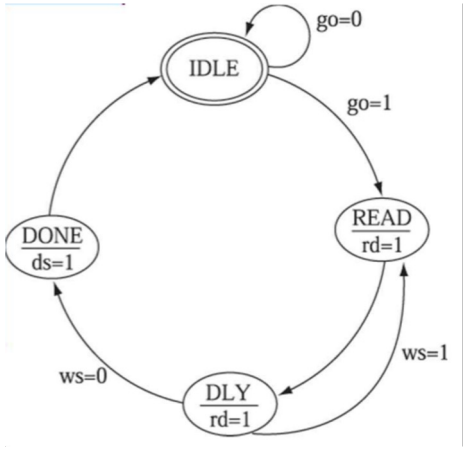
- **You can represent STATE in various ways !**

STATE	binary encoding	Gray encoding	Johnson encoding	One-hot encoding
0	00	00	000	0001
1	01	01	001	0010
2	10	11	011	0100
3	11	10	111	1000

FSM (example)



example code ⇒




```
module fsm_ex1(clk,rst_n, go, ws, rd, ds);
    input clk, rst_n, go, ws;
    output rd, ds;
    parameter IDLE = 2'b00, READ = 2'b01, DLY = 2'b10, DONE = 2'b11;
    reg [1:0] state , next ;

    // always block for state register
    always@(posedge clk or negedge rst_n)begin
        if(!rst_n) state <= IDLE;
        else state <= next;
    end

    // always combinational block to determine next state
    always@(*)begin
        next = 2'bx;
        case(state)
            IDLE : if(go) next = READ; else next = IDLE;
            READ : next = DLY;
            DLY : if(!ws) next = DONE; else next = READ;
            DONE : next =IDLE;
        endcase
    end

    //assign output
    assign rd = ((state==READ) || (state==DLY));
    assign ds = (state == DONE);
endmodule
```

https://returnclass.tistory.com/197

FSM

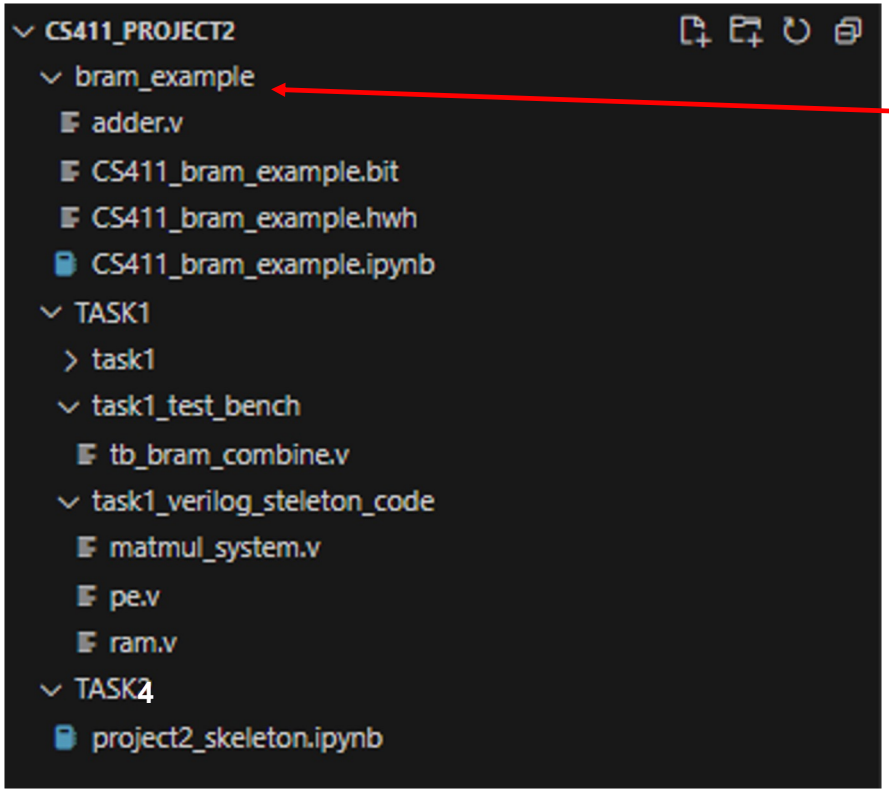
If you are not familiar with FSM.
The following slides will be helpful.

https://inst.eecs.berkeley.edu/~eecs151/sp22/files/verilog/verilog_fsm.pdf

or Try searching on Google

Project Folder

BRAM_example



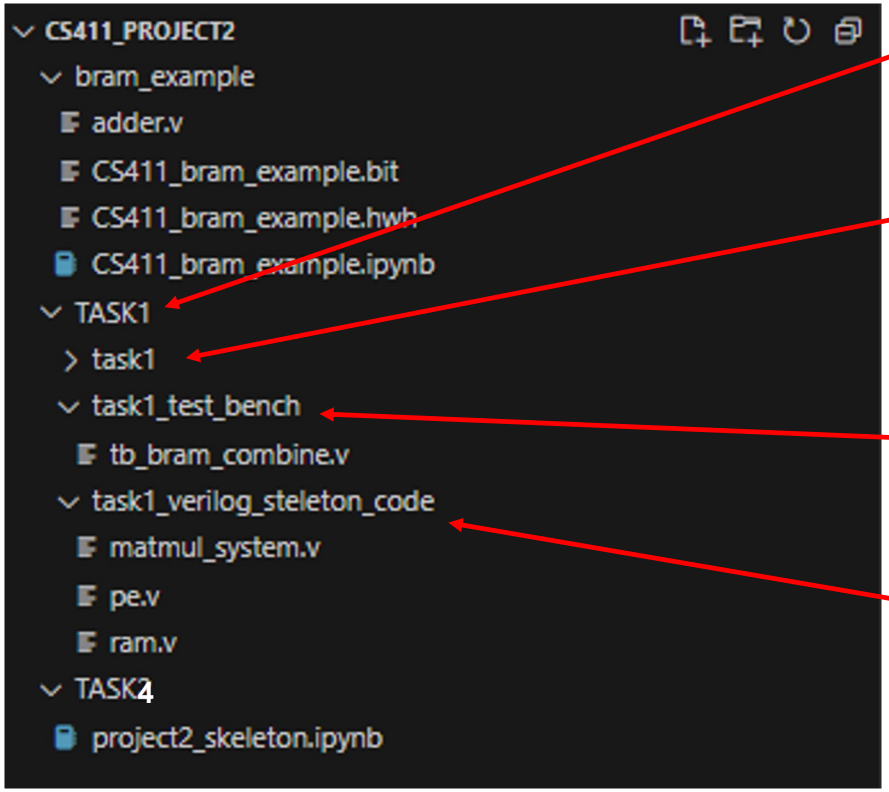
That is for BRAM_Example and includes the following components:

“adder.v” : This module is utilized in the bram_example.

“.hwh” “.bit” files : In your example, you need to generate a bitstream file that matches functionality if this file.

“.ipynb” file : You should use this file to verify functionality.

TASK 1



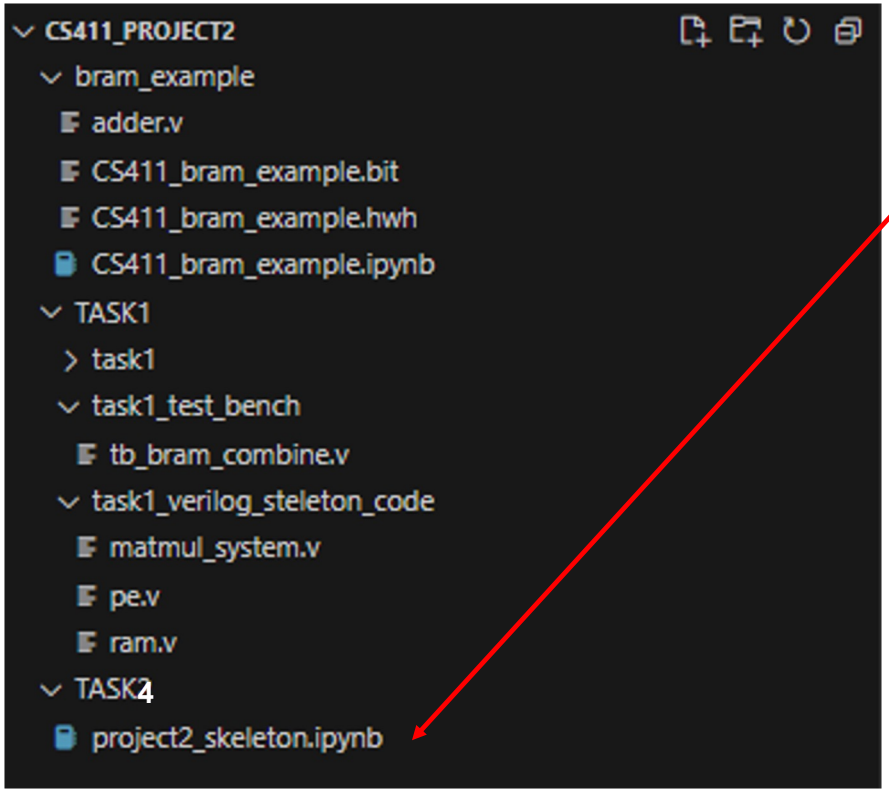
This is for TASK 1 ~ TASK 2

This folder contains a VIVADO project file (.xpr) specifically designed for running the "bram_combine"test. You can open the VIVADO project by using this file.

There is a test bench file available. You can use this file to perform simulation.

You can find skeleton codes.
You need to write additional module.v files to develop your matmul_system more easily.

TASK 4



This is the Jupyter notebook for TASK 4. Please ensure to relocate it to your Jupyter notebook folder. Within this notebook, you are tasked with creating the 'matmul_WS' and 'matmul_OS' functions. You may refer the example provided in the bram_example case

Submission

Score

- **TASK 0: Design Document (30%)**
- **TASK 1,2: Verilog test bench (tb_bram_combine.v) {Behavior, Post-synthesis func + timing} (25%, 15%)**
- **TASK 3,4: Test functionality in jupyter notebook (30%)**

(Extensible: recommendation)

- While it's not part of the scoring criteria, for Project 3, you do have the flexibility to adjust the size of the systolic array from 8x8 to 16x16 or 32x32 as needed. Modulating the systolic array size using parameters can indeed help reduce the workload and make the code more adaptable to different requirements.

Reference

1. PYNQ HW Setting

https://blog.umer-farooq.com/a-pynq-z2-guide-for-absolute-dummies-part-i-fun-with-leds-and-switches-47dd76abf9a9

- https://pynq.readthedocs.io/en/latest/getting_started/pynq_z2_setup.html

Submission

- **Due date : 2023.11.09**

- **Team_{Team_number}_project2 (.zip file)**
  - **Document**
  - **your all verilog file (.v)**
  - **VIVADO project folder**
  - **PYNQ jupyter notebook folder (.jpynb + .bit + .hwh)**