COL215 - Hardware Assignment 3

Matrix multiplication

Submission Deadline: 12th November 2022

1 Introduction

In this assignment, your task is to implement matrix multiplication on basys 3 board.

This assignment involves the design and integration of memories, registers and multiplier-accumulator components (MAC) in your design.

Please note that the diagrams in this document are for the illustration and explanation of the problem statement and do not represent the exact design to be implemented. The design choices can vary for each group and should be carefully explained in the report of the assignment.

2 Problem Description

Given two input matrices of size 128x128 and 8-bit unsigned integer, design a core in VHDL which perform matrix multiplication given input matrices. An overview of the problem is given in figure 1

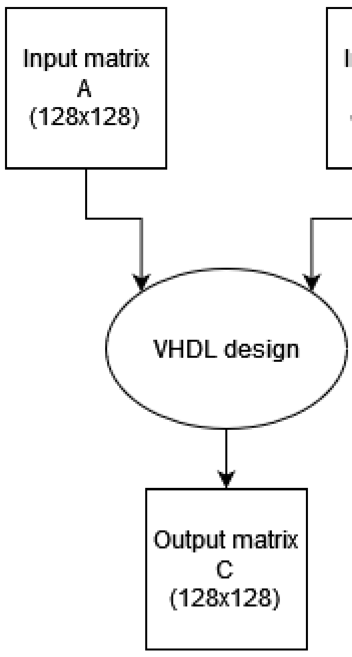


Figure 1: Overview of the problem: matmut

2.1 Operations performed in a matmut

The matrix-matrix multiplication operation can be divided into separate vector-matrix multiplication (which we will be doing in this assignment) as shown in Figure 2.

The overall design would consist of the following:

1. A Multiplier-Accumulator block (MAC): to perform vector-matrix multiplication
2. A Read-Write Memory (RWM, more popularly known as RAM or Random-Access Memory) - You will be using this memory to store the output.
3. A Read-Only Memory (ROM) - You will be using this memory to store the input matrices

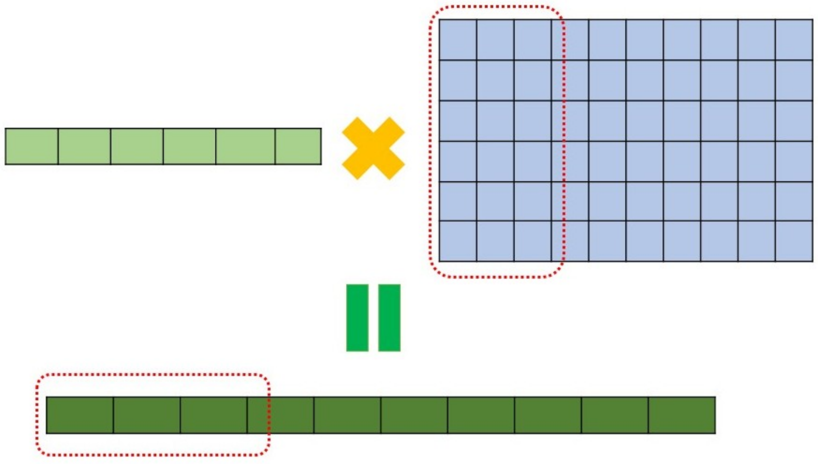


Figure 2: Vector-Matrix multiplication

4. Registers - You will be using these to store temporary results across clock cycles
5. An overall circuit that stitches all modules together to perform matrix multiplication.

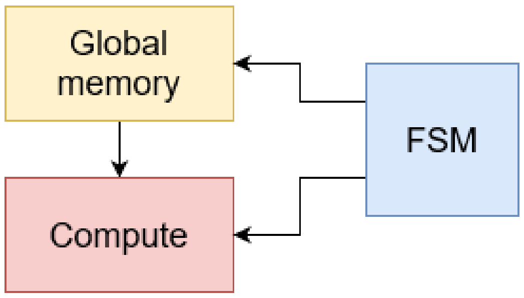


Figure 3: Overview of the design implementation

Figure 3 shows a brief overview of the hardware implementation with a controller (Finite State Machine – FSM) and datapath. The datapath includes reading the inputs values from a global memory (designed as RAM and ROM) and passing it over to the compute unit for computation.

The FSM controls the reading of inputs from global to local registers and compute unit. Figure 4 shows an overview of the FSM and datapath taking an example of one set of inputs.

2.2 Multiply-Accumulate block (MAC)

The first component to be designed in this assignment is a MAC unit that has a 8x8 multiplier, a register and a 16-bit adder to keep accumulating the products. Figure 5 shows the high-level overview of a multiplier-accumulate unit with the required input and output ports. For performing matrix multiplication, you will read out two 8 bit input values from the memory, perform the multiplication using multiplier. The multiplier will generate a 16-bit output which will be sent to the adder for accumulation. Once all the input values from row and column have been multiplied and accumulated, the final output will be stored in the RAM.

cntrl signal can be used to tell the unit whether it is the first product of the compute or not. If it is the first product, assign sum to the first product. After that the product needs to be accumulated with the previous sum. The other signals are self-explanatory.

2.3 Memories

The design requires memories to store the inputs and the output matrices. We will store the input matrices in a read-only memory (ROM) the outputs matrices into a random-access memory (RAM).

2.3.1 Read-Only Memory or ROM

This memory will be used to store the following:

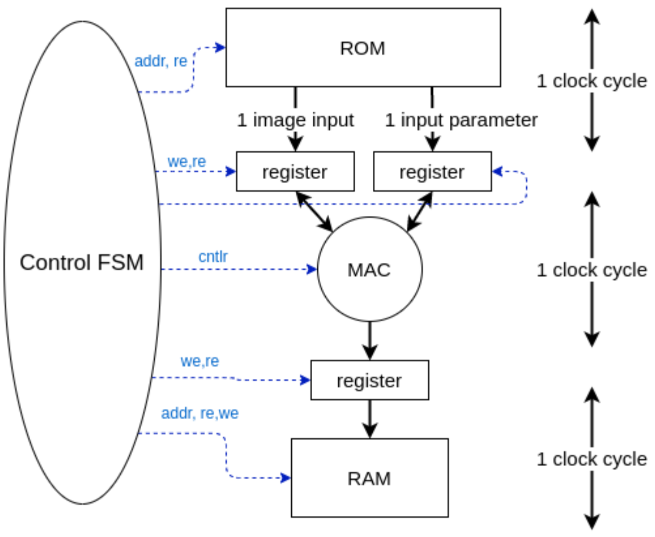


Figure 4: Overview of the FSM and datapath for one set of inputs from one layer to generate one partial output

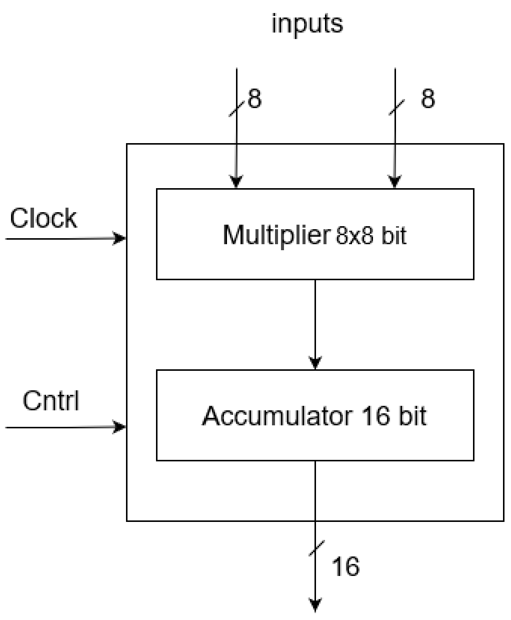


Figure 5: Multiplier-Accumulate Unit

- Input matrix: 128x128 image with each value as an 8-bit unsigned integer. For storing the input we need 16,384 words, 8 bits wide. These will be stored at address 0 (0000_{16}) onwards. The address width of the memory should be at least 14 bits.

2.3.2 Random-Access Memory or Read-Write Memory or RAM

This memory will be used to store the final output matrix of size 128x128, with each value as 16-bit unsigned integer.

2.3.3 Registers

Registers are sequential elements (set of flip-flops) that can be used to store data at clock edges. The signal description is given in Figure 6. You can use these registers as per your design requirements. It has two input flags:

- Read enable 're' signal to read the data stored in the register.
- Write enable 'we' signal to write the data in the register.

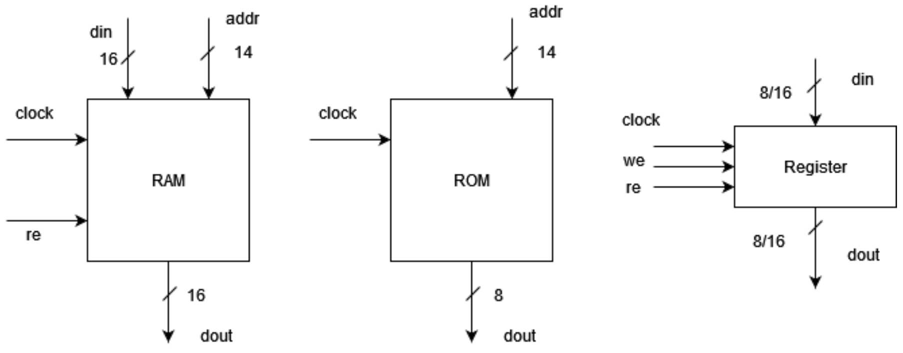


Figure 6: RAM, ROM and Registers block diagram

2.4 Control Path

The control path is basically an FSM in your design which is supposed to do all the sequencing of the data required starting from generating addresses of the data in the order in which you want to read the data, generating the cycle-wise signals to be sent to the datapath to control the modules in the design.

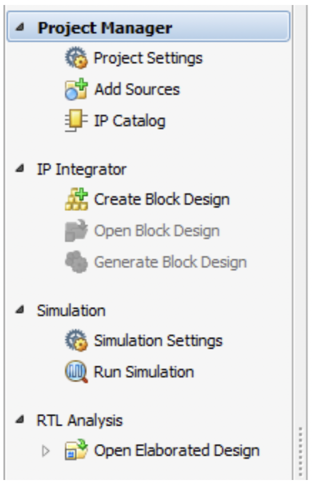
2.5 Output

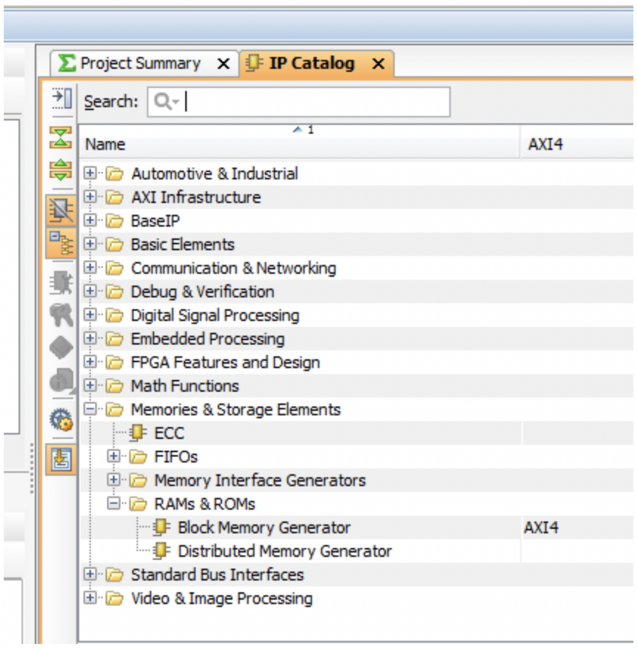
- You need to verify the output using simulation and show waveforms in your report.
- You will be given a special input matrix during the demo. You will be loading that input matrix into the memory. You will be given 2/4 random matrix indices and you will have to display the hexadecimal value of the output matrix at that index on the 7-segment display on the board.

2.6 Generate the memory block using vivado IP catalog

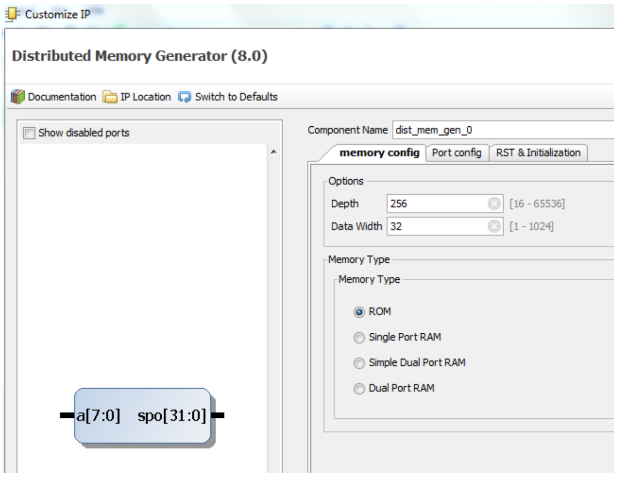
We would be using Vivado's memory generator to design program and data memories. Instructions are as follows:

- In Vivado, go to the Project Manager window on the left side of the screen which looks like the screen-shot shown on right.
- Open IP catalog. You can also open IP Catalog through a drop down menu after clicking on Window in the main menu on the top of the screen. "IP" (Intellectual Property) refers to pre-designed modules available for use. The IP catalog lists various IPs category-wise.
- Select the category **Memories & Storage Elements** and sub-category **RAMs & ROMs**, as shown below.





- Now choose **Distributed Memory Generator**, to create a ROM / RAM as per your design requirement. This generator creates memory modules of specified sizes using LUTs of FPGA. The reason for using the Distributed Memory Generator here is that **Block Memory Generator** creates a memory with an address register since they use FPGA BRAMs, and are mostly preferred when we want large on-chip memories.
  The distributed RAMs are faster and more efficient because they use the LUTs in contrast to FPGA BRAMs.
  The Distributed Memory Generator opens a window with three tabs. In the **memory config tab**, specify memory size and type. The screen-shot shown below is for a ROM with 256 words, each of size 32 bits. You should change these parameters based on the input matrix size that is ROM with 16,384 words with size of 8 bits.


Customize IP

Distributed Memory Generator (8.0)

Component Name: dist_mem_gen_0

memory config

Options
Depth: 256 [16 - 65536]
Data Width: 32 [1 - 1024]

Memory Type
[✓ ROM] Single Port RAM Simple Dual Port RAM Dual Port RAM

- In the **RST & Initialization** tab, specify name of the file that defines the ROM contents. A sample file named *sample.coe* is available on moodle.
  Now instead of passing the inputs from the test bench or using "force value" construct, please initialize the inputs using a *coe* file and pass input to the Design Under Test (DUT) by connecting it to the memory module.
  Once you synthesize and implement your design, you will see a change in the number of LUTs used in your design.

- Use the following test bench to simulate the memory IP block in vivado. The test bench is an illustration of how to import the generated memory block into your VHDL design file.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.numeric_std.all;


ENTITY tb_ROM_block IS
END tb_ROM_block;
ARCHITECTURE behavior OF tb_ROM_block IS
    COMPONENT blk_mem_gen_0
    PORT(
        clka : IN STD_LOGIC;
        addra : IN STD_LOGIC_VECTOR(13 DOWNTO 0);
        douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
    END COMPONENT;
  --Inputs
   signal clock : std_logic := '0';
   signal rdaddress : std_logic_vector(13 downto 0) := (others => '0');
  --Outputs
   signal data : std_logic_vector(7 downto 0) := (others => '0');

   -- Clock period definitions
   constant clock_period : time := 10 ns;

   signal i: integer;
BEGIN
 -- Read image in VHDL
   uut: blk_mem_gen_0  PORT MAP (
          clka => clock,
          douta => data,
          addra => rdaddress
        );

   -- Clock process definitions
   clock_process :process
   begin
      clock <= '0';
      wait for clock_period/2;
      clock <= '1';
      wait for clock_period/2;
   end process;
   -- Stimulus process
   stim_proc: process
   begin
      for i in 0 to 16383 loop
        rdaddress <= std_logic_vector(to_unsigned(i, 14));
      wait for 20 ns;
      end loop;
      wait;
   end process;

END;
```

2.7 Creating finite state machine in VHDL

For the control path, you will implement an FSM to control the MAC unit and perform data transfer between ROM/RAM and the local registers. In this section, an illustration is given on how to implement FSM in VHDL.

An example FSM shown in figure 7, this is the FSM for the programmable adder/subtractor as discussed

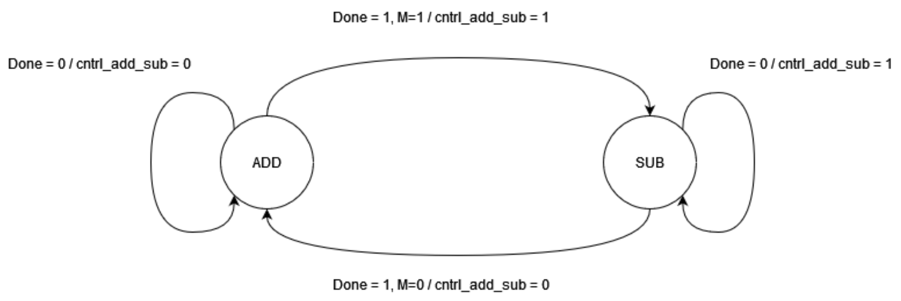

Done = 1, M=1 / cntrl_add_sub = 1
Done = 0 / cntrl_add_sub = 0
Done = 0 / cntrl_add_sub = 1
ADD
SUB
Done = 1, M=0 / cntrl_add_sub = 0

Figure 7: FSM for adder/subtractor

in the lecture. It consists of 2 states ADD and SUB, along with input flags M (to change the operation) and DONE (to know when the current operation is completed) and output flag cntrl_add_sub to set the operation in the programmable adder/subtractor.

The FSM in VHDL will be made up of two parts:

1. Sequential block: In this block, the state is updated on the clock edge or initialized to a default state if the reset flag is set HIGH.
2. Combinational Block: It consists of the logic to update the next state based on the flag values and the current state.

The following is the VHDL template for implementing an FSM. You can see that the state of the system is captured in the Sequential block and updated every positive clock edge, and the work performed in one clock cycle is indicated/controlled in the Combinational block.

```
entity FSM IS
    Port ( M : in STD_LOGIC;
           Done : in STD_LOGIC;
           clk    : in  STD_LOGIC;
           reset : in STD_LOGIC;
           cntrl_add_sub : out STD_LOGIC;
        );
end FSM;
architecture machine of FSM IS
    type state_type is (ADD, SUB);
    signal   cur_state          : state_type := ADD;
    signal   next_state          : state_type := ADD;
begin
   --Sequential block
   process (clk, reset)
   begin
      if (reset = '1') then
         cur_state <= ADD;
      elsif (clk'EVENT AND clk = '1') then
         cur_state <= next_state;
      end if;
    end process;
    --Combinational block
    process (cur_state, M, Done)
    begin
        next_state <= cur_state;

        case cur_state is
            when ADD =>
                if Done = '1' and M = '1' then
                    next_state <= SUB;
                    cntrl_add_sub <= '1';
                elsif Done = '0' then
                    next_state <= ADD;
                    cntrl_add_sub <= '0';
                end if;
            when SUB =>
                if Done = '1' and M = '0' then
                    next_state <= ADD;
                    cntrl_add_sub <= '0';
                elsif Done = '0' then
                    next_state <= SUB;
                    cntrl_add_sub <= '1';
                end if;
        end case;
    end process;
end machine;
```

3 Submission and Demo Instructions

Since this assignment will span over 2 weeks, you will be evaluated each week in the lab depending on your progress. We have broken down the design into blocks and you need to spend time accordingly to complete this assignment.

1. **Week 1: Design and test all the sub-components of the datapath in the assignment. For this week, we are giving you a sample COE file to get your memory block working.**
2. **Week 2: Design the control path and integrate it with the data path. Test the design**
3. You are required to submit a zip file containing the following on Gradescope:
   - VHDL files for all the designed modules.
   - Simulated waveforms with test cases of each sub-component and complete design.
   - A short report (1-2 pages) explaining your approach. Include block diagram depicting the modules.
4. Post your doubts in HW Assignment 3 thread on Piazza.
5. Be ready with your design before the lab session. Validate the design through simulations. During the lab session, perform further validation of the design by downloading it into the FPGA board.

4 Resources references

- IEEE document: `https://ieeexplore.ieee.org/document/8938196`
- Basys 3 board reference manual: `https://digilent.com/reference/_media/basys3:basys3_rm.pdf`
- Online VHDL simulator: `https://www.edaplayground.com/x/A4`