EE-271 PROJECT

16-BIT PIPELINED

FLOATING POINT MAC UNIT

Project By

Group -13

Nidhu Narayanan: 015324660
Purav Bhatt: 015264041
Surbhi Sharda: 015968745

TABLE OF CONTENTS

ABSTRACT

The objective of this project is to implement a 3-stage Pipelined Floating-point MAC unit for AI engines using half-precision (i.e., 16 bit) IEEE 754 floating-point standard. We will implement the working of our project using a Finite state machine. We have 4 states (from S0 to S3) from which we will traverse from one to another using the push button and by reading the input given from the keypad. We have also provided a mechanism to reset the FSM state using an asynchronous reset controlled via the switch button. Whenever this switch button is turned to high, our FSM is reset and we traverse to S0. State S0 is the reset state. Traversing to S1 and S2, there we are providing input1 and input2 respectively using a 4x4 numerical keypad, and the value of the input provided is displayed on the 7-segment display of the DE10 Lite board. The movement from one state to another can be monitored on the I2C LCD connected with Arduino Uno. Lastly in state S4, the result after the calculation can be seen on the 7-segment display which is included in the FPGA DE10 lite board.

INTRODUCTION

The object of this project is to implement a floating-point MAC unit. Floating-point numbers are required for the representation of real numbers in binary format. Floating-point numbers are required for many engineering and technical calculations. Examples of some floating points numbers are 3.256, 2.1, and 0.0036. IEEE 754 standard is most commonly used for representing floating-point numbers. IEEE 754 is a predefined technical standard that is used to represent floating-point arithmetic. (IEEE) established this in 1985 and this has addressed many issues found in the diverse floating-point implementation that made challenging to use them portably as well as reliably. IEEE half-precision floating-point standard representation requires **10 fraction bits F, 5 exponent bits E, and 1 sign bit S**, with a total of 16 bits for each word.

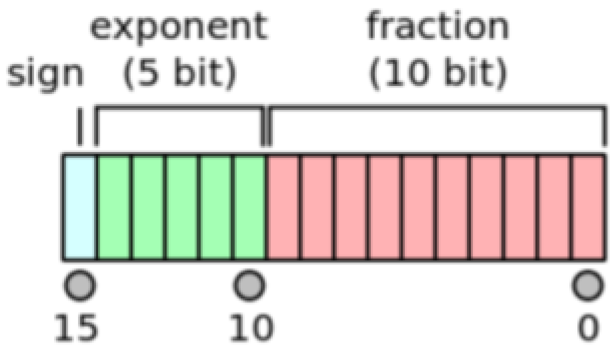
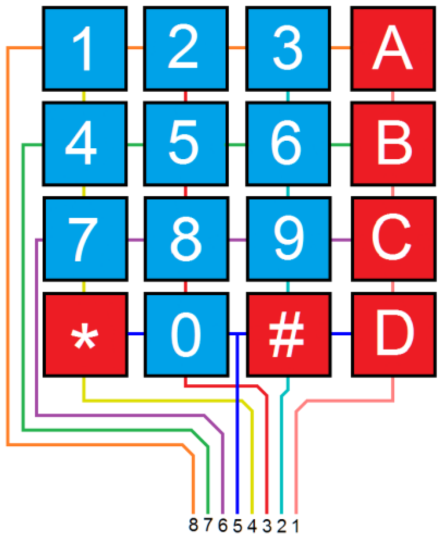


Figure 1: 16-bit floating-point representation

In this project we are implementing our design in three pipelining stages i.e., Multiplication, Align, addition, and normalizing so we can parallelly run more than one set of input/data at a particular time.

HARDWARE

- 4x4 Numerical Keypad [3]:

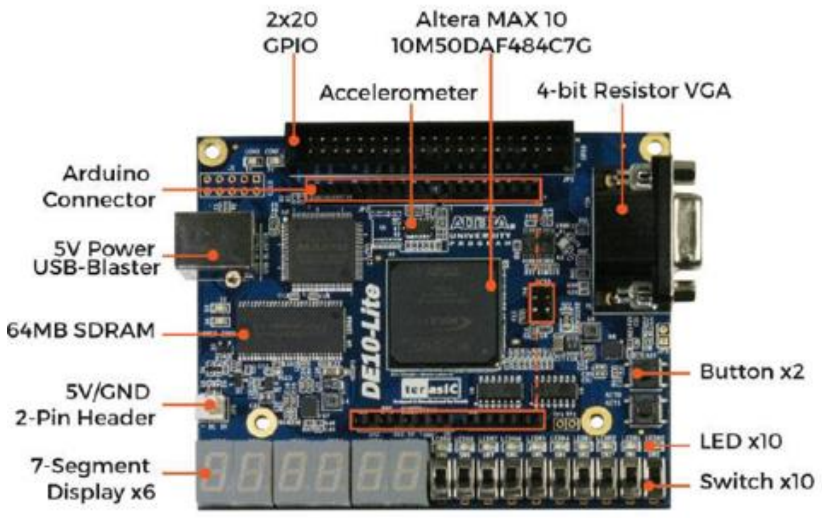


Keypad 4x4 is used for loading numeric into the microcontroller. It consists of 16 buttons arranged in a form of an array containing four lines and four columns. It is connected to the development system by a regular IDC 10 female connector plugged in some development system's port.

- Arduino Uno [2] :

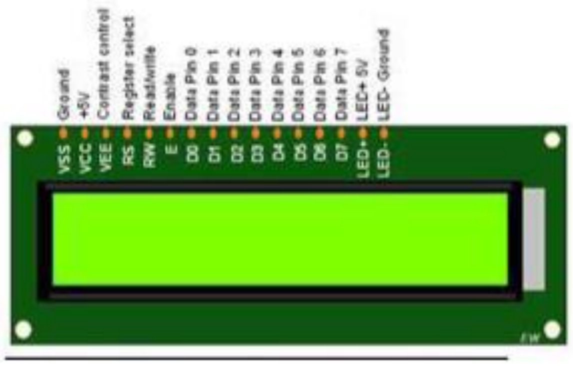
Arduino Uno is widely used a microcontroller development board that is based on 8-bit ATmega328P (datasheet). Also, to support the microcontroller it also consists of other components as a voltage regulator, crystal oscillator and serial communication, etc. to support the microcontroller. Arduino board has provided 14 digital input/output pins for I/O connections (6 can be used for PWM outputs), 6 for analog inputs i.e., a USB connection, A Power barrel jack, an ICSP header, and a reset button.

- DE-10 Lite FPGA Board [1]:



DE10-Lite is an economical FPGA board by Altera MAX 10. FPGA is known as a field-programmable gate array. This is a hardware circuit that can be programmed by a user for performing one or multiple logic operations. This board consists of multiple on-die analogs to digital converts, around 50k logic elements. It has an onboard SDRAM, acetometer, USB blaster, 2x20 GPIO connector, etc.

- LCD Display [4]:



LCD stands for Liquid Crystal Display. It is a 16x2 LCD which is a very commonly used module with other circuits for displaying output values. The LCD is a flat panel display and is operated using liquid crystals. An LCD can display 16 characters per line and there are 2 such lines.

SOFTWARE

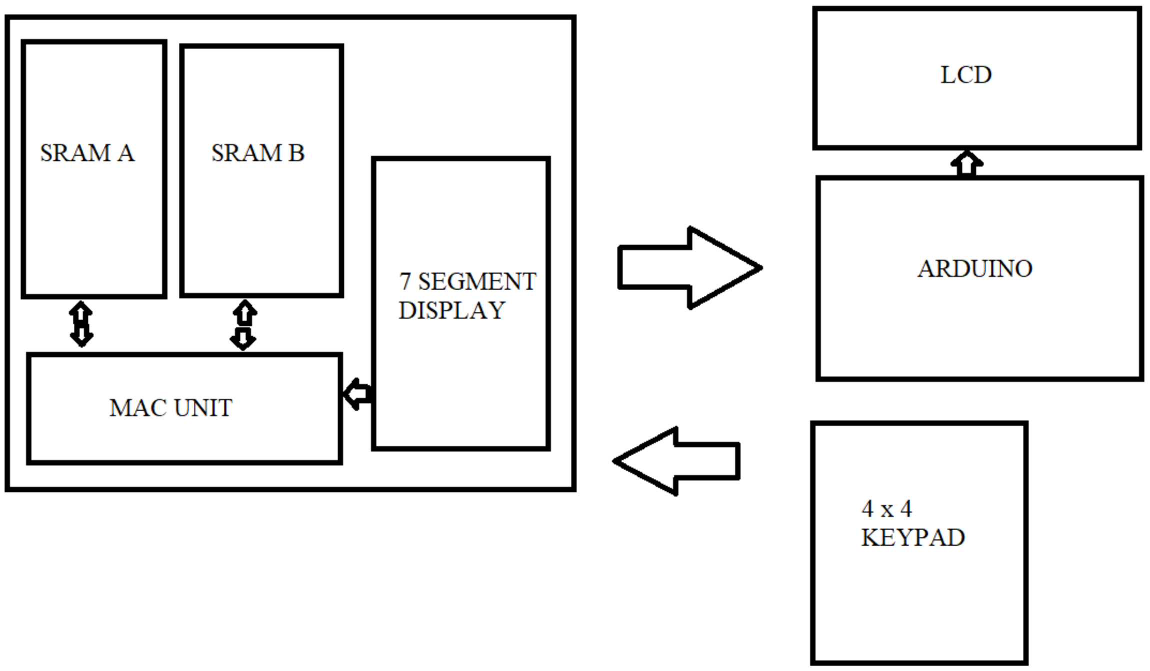
- Arduino IDE
  Arduino IDE (Integrated Development Environment) is open-source software that makes it simple to write a code and upload it to the board. This software environment contains a text editor where we can write the code, a toolbar, a message area, a text console, and a series of menus.

- Quartus Prime Lite Edition
  This is a software tool that helps in compilation and programming for a limited number of Intel FPGA boards.

- VCS Tool
  VCS is the tool provided by Synopsys which is used for functional verification solutions. VCS is used by the majority of the world's top semiconductor companies as the primary verification solution. VCS provides different features to achieve higher performance. VCS uses Fine-Grained Parallelism (FGP) technology with advantages provided by multicore processors which gives a high-speed performance

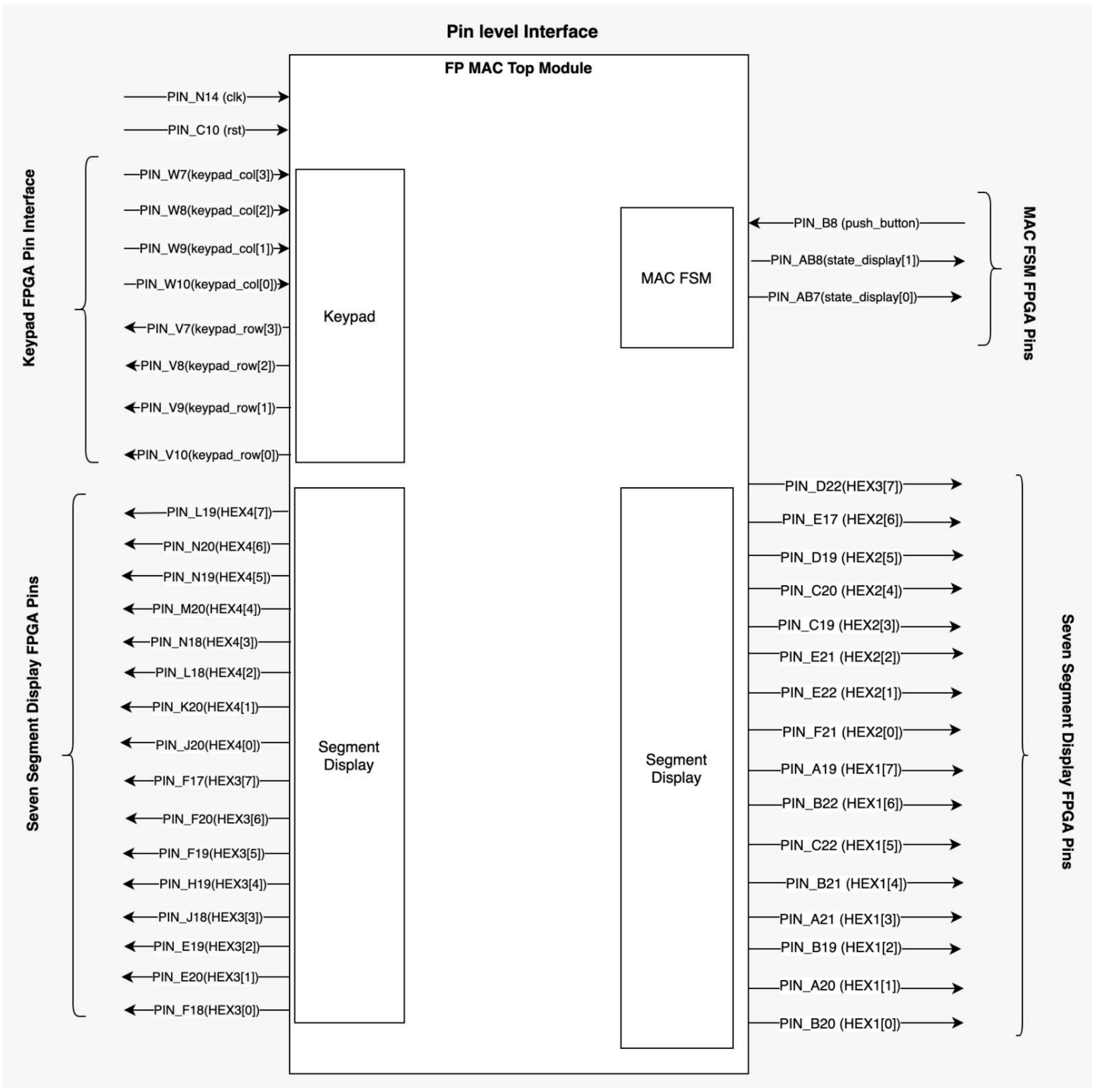
DESIGN OVERVIEW

The diagram below represents the bird's eye view of the project design. The design starts with the keypad. The user gives inputs through the 4x4 keypad. The custom-built SRAMs are filled with 8 inputs each. The MAC unit then takes every row of the SRAMs and multiples them while adding the previous result. The project uses 3-stage pipelining.



After the final addition, the result is displayed on the 7 segment displays on the board. While the data moves from one stage to another, the corresponding states are displayed on the LCD which is interfaced with Arduino. The arrows in the figure represent the direction flow of data. On-board GPIO pins, push-button, digital pins, and switch buttons are used for several purposes. The keypad is interfaced with the FPGA board via the GPIO pins. 8 jumper wires, 4 for rows, and 4 columns are used for the interface. The onboard switch button is used to reset the system. It is an active-high reset. The push-button is used to go to the next state. After the MAC output state, the system goes back to the reset state when pressing the push button. The Arduino is interfaced with the FPGA board by using digital pins. The pins give the status of the state of the system and depending on it, the LCDs the appropriate message of the screen. The LCD follows the I2C communication protocol and is interfaced with the GPIO pins of the Arduino. A potentiometer is used to change the color intensity of the LCD. On giving inputs through the keypad, the appropriate hex number is displayed on the onboard seven-segment displays. Moreover, to reset the SRAMs, the push button is pressed on the final state. Traversing through the states without giving inputs, resets the SRAMs.

FPGA SCHEMATIC



The above figure represents the pin-level interface of the modules used in the project. For the state transition push button, pin B8 is used. For resetting the system, pin C10 of the board is used. 50 MHz clock is used that is available from pin N 14. 8 GPIO pins are used for the keypad. 4 columns pins are W7, W8, W9, W10. Pins V7, V8, V9, and V10 are used to interface the rows. 4 seven segment displays are used. The first display uses B20, A20, B19, A21, B21, C22, B22, and pin A19. . Second display uses E22, E21, C19, C20, D19, E17, and pin F21. The third display uses F18, E20, E19, J18, H19, F19, F20, and pin F17. The last display uses L19, J20, K20, L18, N18, M20, N19, and pins N20. Pins AB7 and AB8 are used to interface with Arduino to display the current state of the system.

- Arduino Interface with FPGA

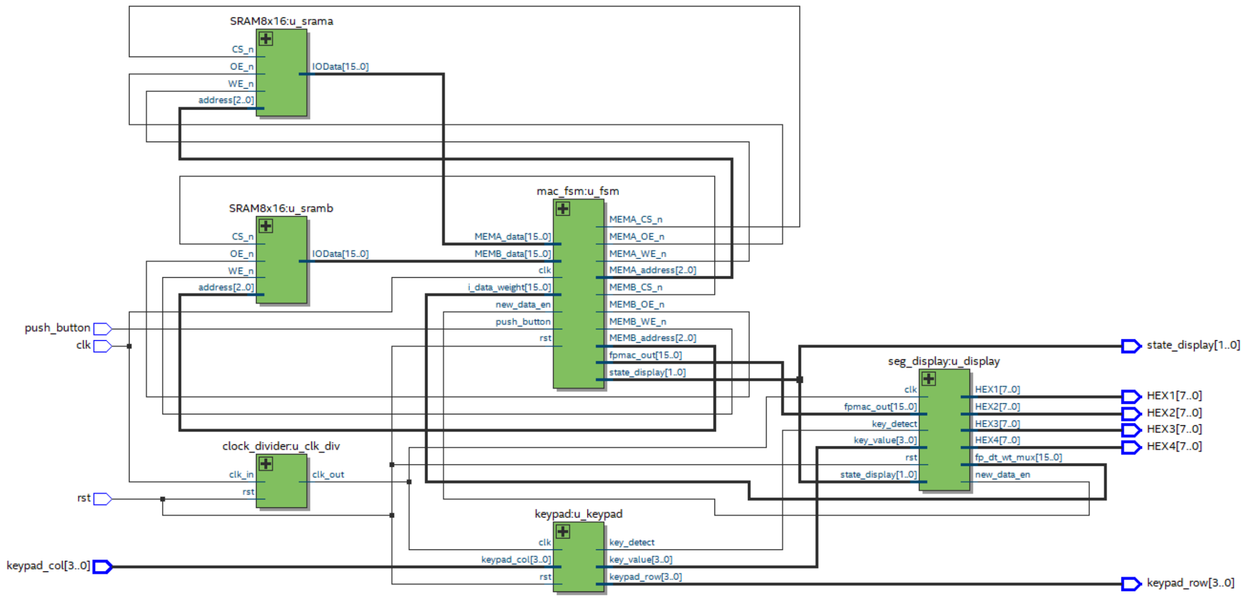
Schematic Signal Name	FPGA Pin No	Description	Signal in design
Arduino_IO3	PIN_AB8	Arduino IO3	state_display[1]
Arduino_IO2	PIN_AB7	Arduino IO2	state_display[0]

- Arduino Interface with LCD

Arduino Pin	LCD Pin	Description
SCL	SCL	The Serial Clock pin synchronizes the data transfer over the I2C bus.
SDA	SDA	The Serial Data pin is used for transmitting the data between LCD and Arduino.
5V	VCC	The VCC pin of LCD is connected to the +5V output of Arduino.
GND	GND	Ground pin.

IMPLEMENTATION

- Micro-Architecture Diagram



The floating-point MAC unit is implemented using seven modules as mentioned below. Top module, clock divider, keypad module, segment display, FSM, and MAC module, SRAM module (SRAM A and SRAM B instance)

- Functionality Description of Modules

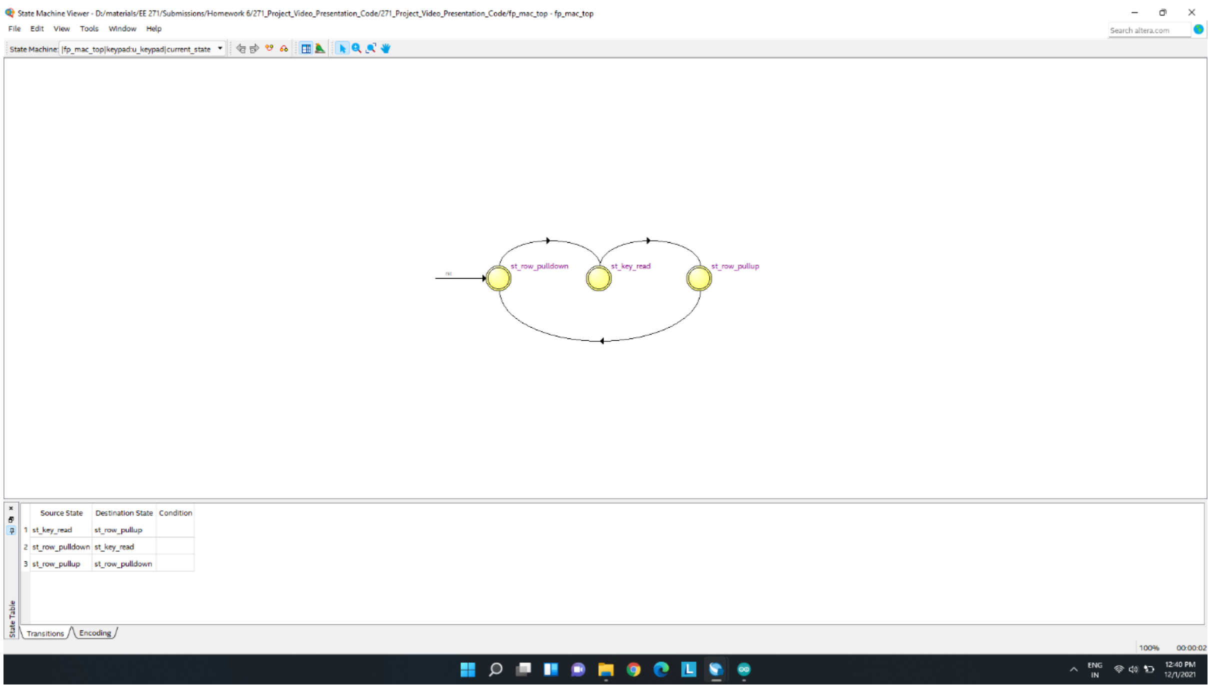
1) Top module: fp_mac_top.v

The top module contains the instantiation of each sub modules. This module interfaces with the FPGA through pins. The design operates on a 50 MHz frequency, and the FPGA switch is used for active high asynchronous reset. The MAC design uses the push button on FPGA to control the state transitions.

2) Clock divider module: clock_divider.v

The functionality of the clock divider is to create 1 KHZ clock to run the keypad from a 50 MHz clock.

3) Keypad module: keypad.v


State Machine Viewer - D:/materials/EE 271/Submissions/Homework 6/271_Project_Video_Presentation_Code/271_Project_Video_Presentation_Code/fp_mac_top - fp_mac_top
File Edit View Tools Window Help
State Machine: |fp_mac_top|keypad:u_keypad|current_state
st_row_pulldown
st_key_read
st_row_pullup
State Table
Transitions
Encoding

	Source State	Destination State	Condition
1	st_key_read	st_row_pullup	
2	st_row_pulldown	st_key_read	
3	st_row_pullup	st_row_pulldown	

The keypad module's functionality is to detect the 4x4 matrix keypad inputs. The module has three states as shown in the above diagram. The keypad rows are set as outputs. The first state is the row pull-up state. In this state, one of the rows is pulled low, and the other rows are high. The key reads the column's value (active low) in the second state. The keypad column input pin is assigned with pull up resistor. Due to the pull up resistor, the input column pin will read a high when the button is not pressed, and will read a low when it is pressed. The implementation logic has a keypad array that keeps track of the keys pressed and avoids debouncing the keys. The last row of the 4x4 matrix is considered a special case; the value 'E' is programmed using '#,' and F uses '*.' After reading the keys pressed in the selected row, FSM moves to the row's next state, the pull-up state.

4) Segment display module: seg_display.v

The functionality of the segment module is to decode the seven-segment value for the keypad inputs, MAC output, and provide the floating-point input to the MAC module. The floating-point input is generated using a shift register. The segment display module generates an enable signal when four keypad inputs are detected. This data enable signal acts as the floating-point input valid for the MAC unit. Only four-segment displays are used for the current MAC design. The segments used in the design are HEX5, HEX4, HEX3, HEX2. The segment display module output HEX4, HEX3, HEX2, and HEX1 are connected to the FPGA segment display HEX5, HEX4, HEX3, HEX2, respectively. When the system is reset using active high reset, the seven-segment displays 8. on all four displays. The display goes off when the system is out of reset. When the MAC FSM is in SRAM A state, the segment displays the 8 data inputs and the weight input when the FSM is in SRAM B state. The segment display will display the final accumulated output when the FSM is in the MAC calculation state.

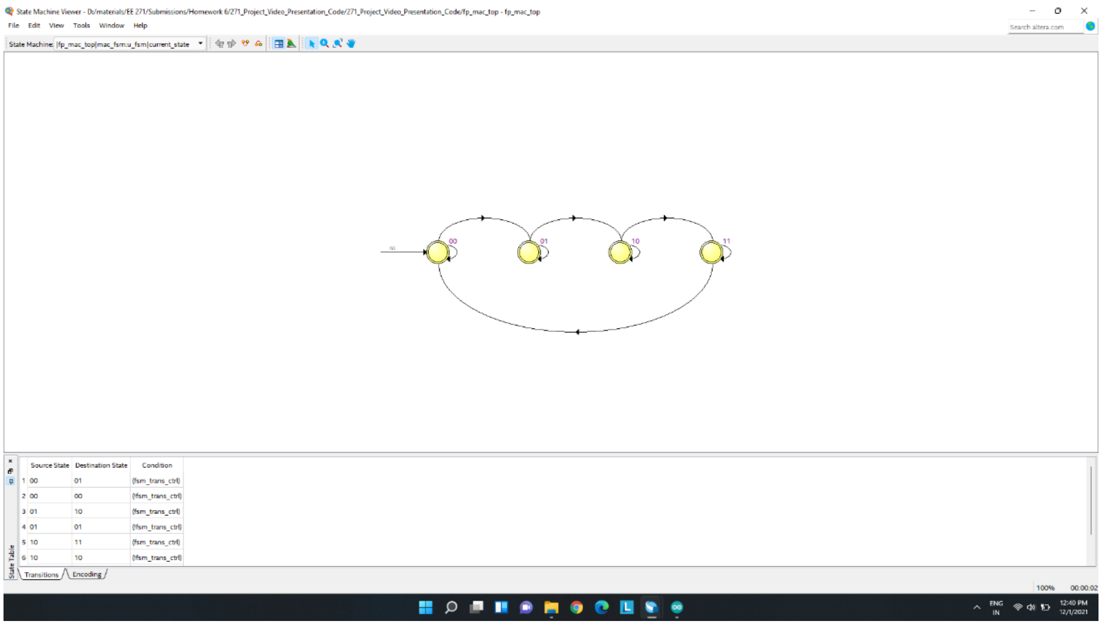
5) SRAM module: SRAM8x16.v

We have used an asynchronous SRAM in our current design. The depth of the SRAM is 8, and each location can store 16 bits. The address bus is 3 bits. The SRAM is a single port memory; hence, the bidirectional data bus. The SRAM has a chip select, write enable, and output enable control signals. All the control signals are active low. The SRAM supports asynchronous read and write. When the chip selects and writes enable is low, the SRAM will write. And when chip select and output enable is low, read will happen. If there is no read or write, the data is floating.

6) MAC FSM: mac_fsm.v

The MAC FSM module has two functionalities. One is the FSM control, and the other is the multiplication and accumulation calculation.

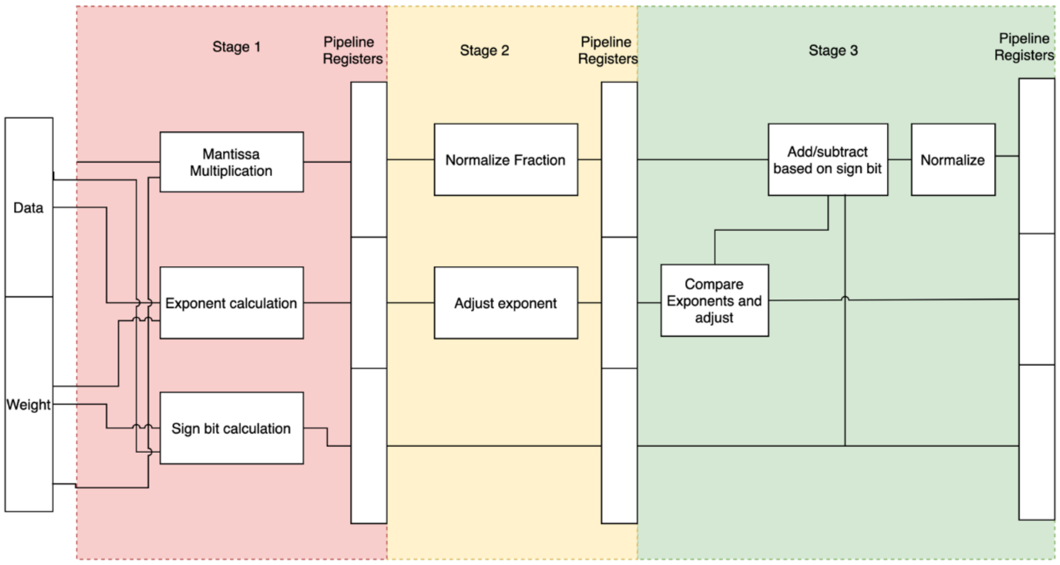
- System State Diagram:


State Machine Viewer - D:/materials/EE 271/Submissions/Homework 6/271_Project_Video_Presentation_Code/271_Project_Video_Presentation_Code/fp_mac_top - fp_mac_top
File
Edit
View
Tools
Window
Help
State Machine: |fp_mac_top|mac_fsm:u_fsm|current_state
00
01
10
11
State Table
Transitions
Encoding

	Source State	Destination State	Condition
1	00	01	(fsm_trans_ctrl)
2	00	00	(!fsm_trans_ctrl)
3	01	10	(fsm_trans_ctrl)
4	01	01	(!fsm_trans_ctrl)
5	10	11	(fsm_trans_ctrl)
6	10	10	(!fsm_trans_ctrl)

The above state diagram represents various stages of the system. On pushing the reset button, the system goes to state 00. On pressing the push button, the system moves to state 01. Here the SRAM A gets input from the keypad. Finishing that, the system goes to state 10. Here, the SRAM B gets inputs from the keypad. Next, the system goes to state 11. The FSM also generates a state signal to indicate the status of the FSM, which is used by the segment display (to print the values on display) and Arduino board to display the state information on LCD. The design uses a synchronizing logic to detect the falling edge of the push button signal and creates an internal pulse signal to do the state transition. The synchronization logic is also used for the data enable signal since it is generated from a slower clock.

- Pipeline diagram:

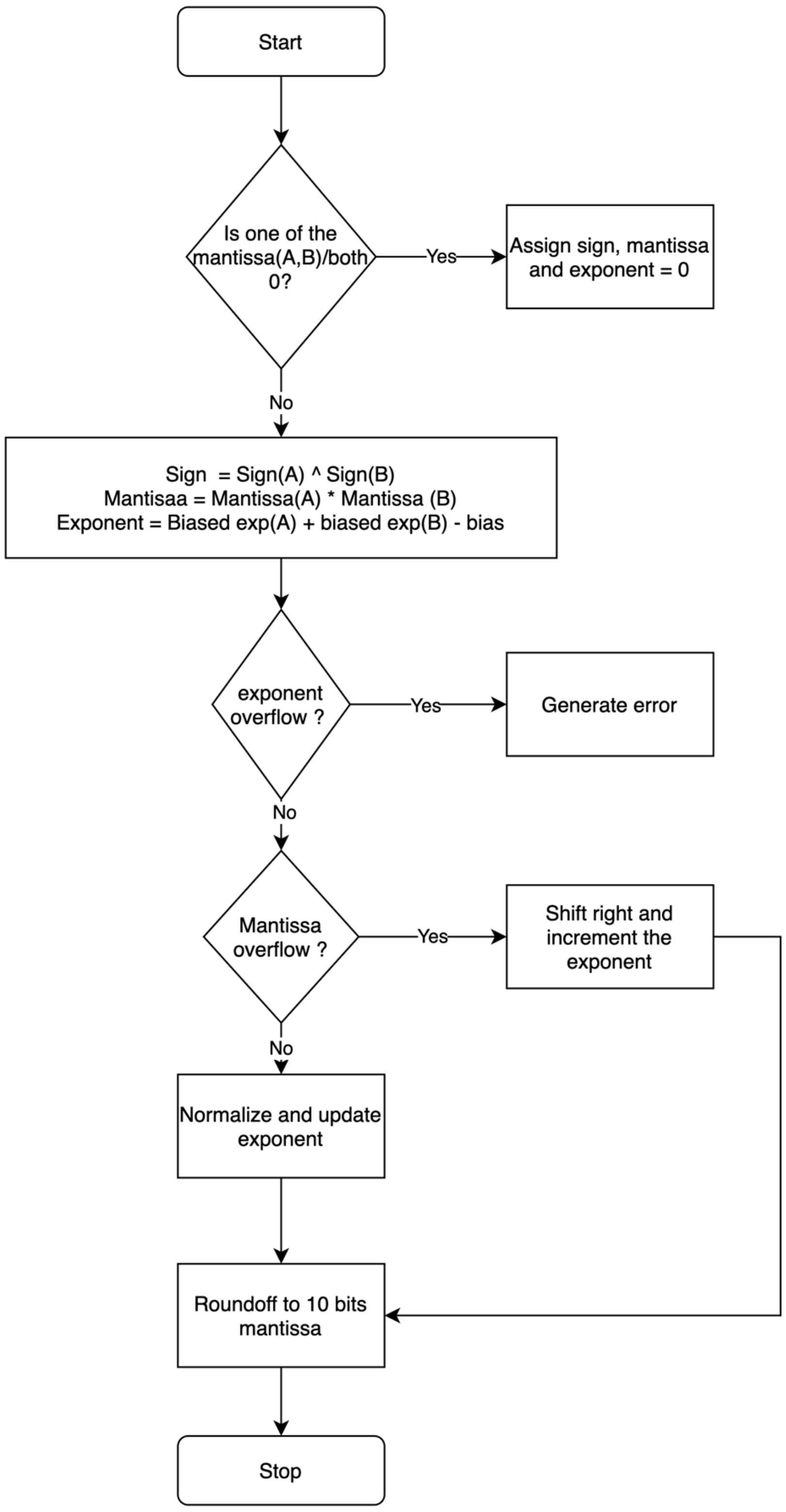


In the first stage, the mantissa part of data and weight is multiplied, including the hidden bit, the exponent is calculated, and the sign of the result is determined. The normalization for the multiplied result is done in the stage. In the normalization stage, the design checks for the exponent overflow. If the overflow is detected, an overflow error is detected. The fraction and exponents are adjusted in the case of fraction overflow. Accumulated addition and normalization are done in this stage.

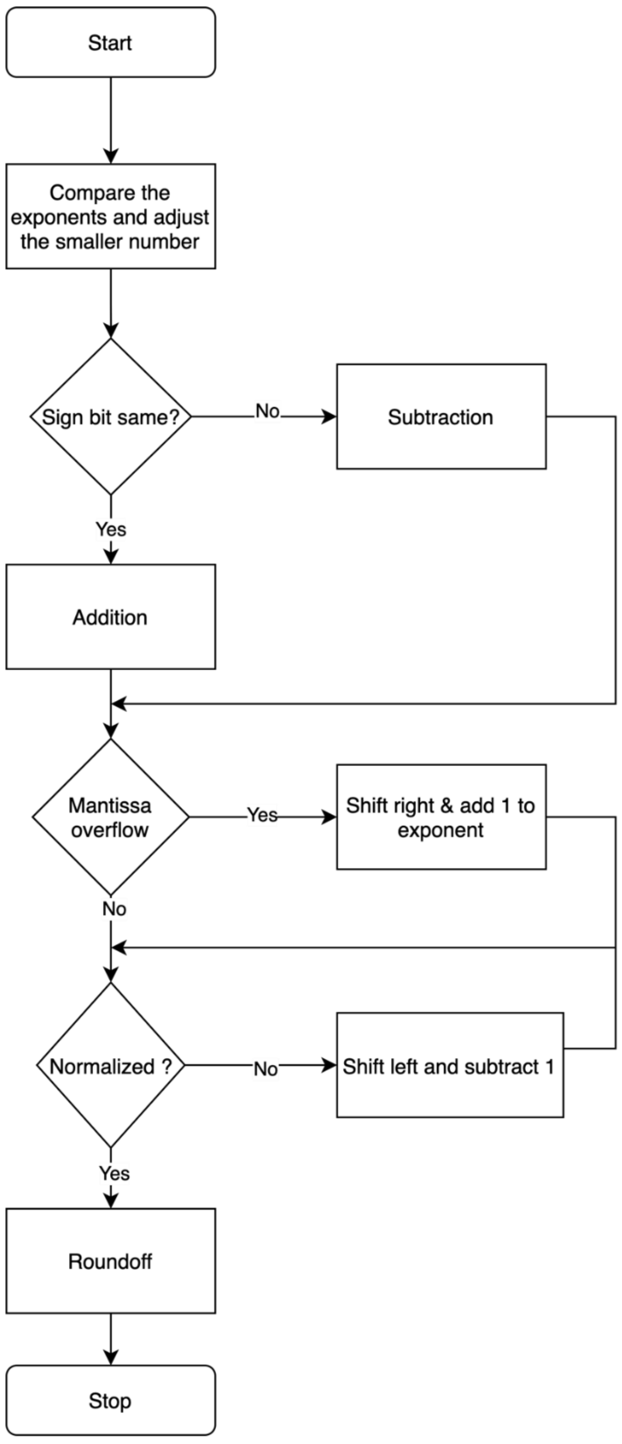
- Pipeline Stage Table:

Input	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
mema[0] memb[0]	MULT	NORM	ACC							
mema[1] memb[1]		MULT	NORM	ACC						
mema[2] memb[2]			MULT	NORM	ACC					
mema[3] memb[3]				MULT	NORM	ACC				
mema[4] memb[4]					MULT	NORM	ACC			
mema[5] memb[5]						MULT	NORM	ACC		
mema[6] memb[6]							MULT	NORM	ACC	
mema[7] memb[7]								MULT	NORM	ACC

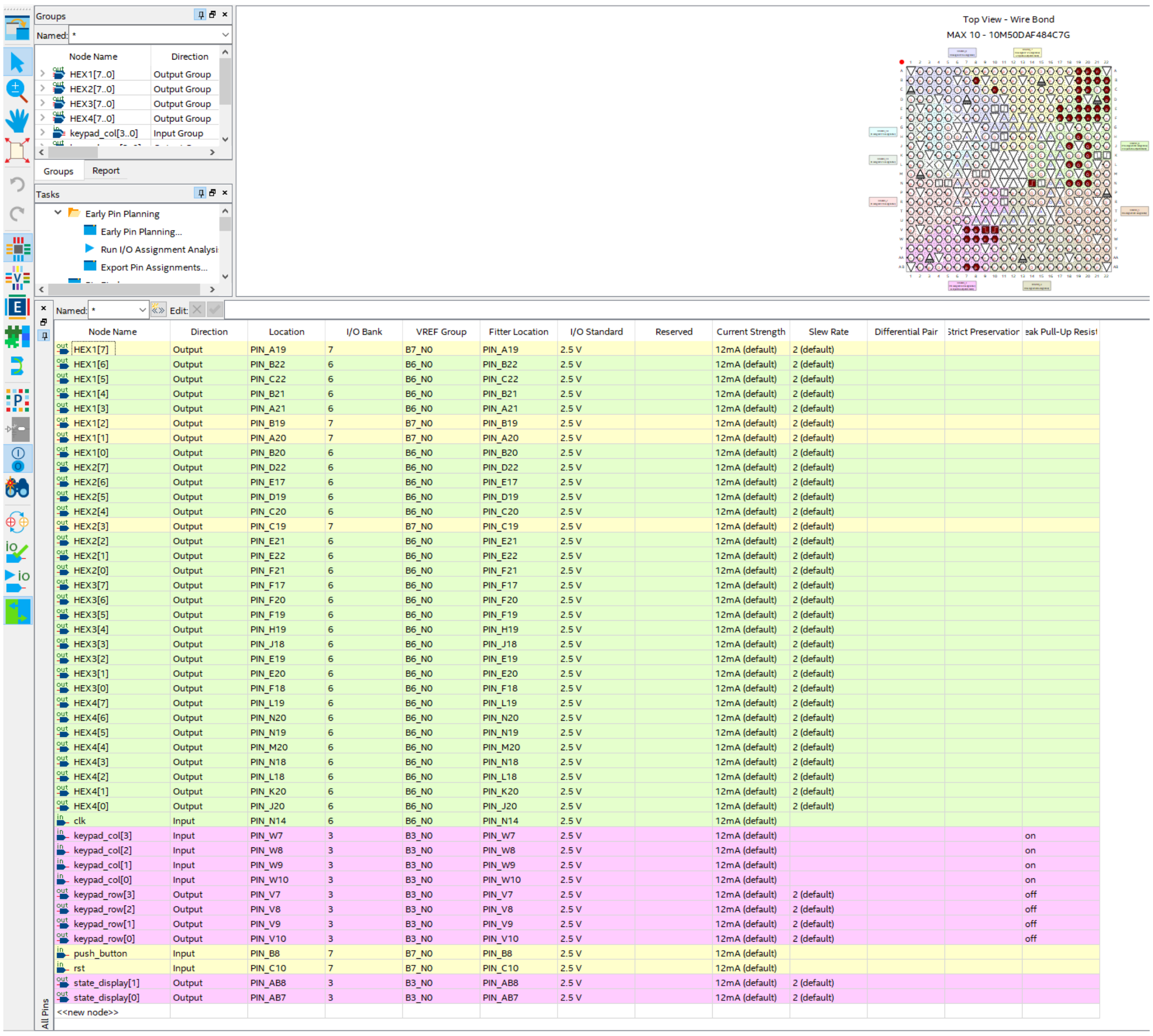
- Multiplication Flowchart:


Start
Is one of the mantissa(A,B)/both 0?
Yes
Assign sign, mantissa and exponent = 0
No
Sign = Sign(A) ^ Sign(B)
Mantisaa = Mantissa(A) * Mantissa (B)
Exponent = Biased exp(A) + biased exp(B) - bias
exponent overflow ?
Yes
Generate error
No
Mantissa overflow ?
Yes
Shift right and increment the exponent
No
Normalize and update exponent
Roundoff to 10 bits mantissa
Stop

- Addition Flowchart:


Start
Compare the exponents and adjust the smaller number
Sign bit same?
No
Subtraction
Yes
Addition
Mantissa overflow
Yes
Shift right & add 1 to exponent
No
Normalized ?
No
Shift left and subtract 1
Yes
Roundoff
Stop

- PIN Assignment:



Groups

Named: *

Node Name	Direction
HEX1[7..0]	Output Group
HEX2[7..0]	Output Group
HEX3[7..0]	Output Group
HEX4[7..0]	Output Group
keypad_col[3..0]	Input Group

Groups Report

Tasks

- Early Pin Planning
  - Early Pin Planning...
  - Run I/O Assignment Analysi
  - Export Pin Assignments...

Top View - Wire Bond

MAX 10 - 10M50DAF484C7G

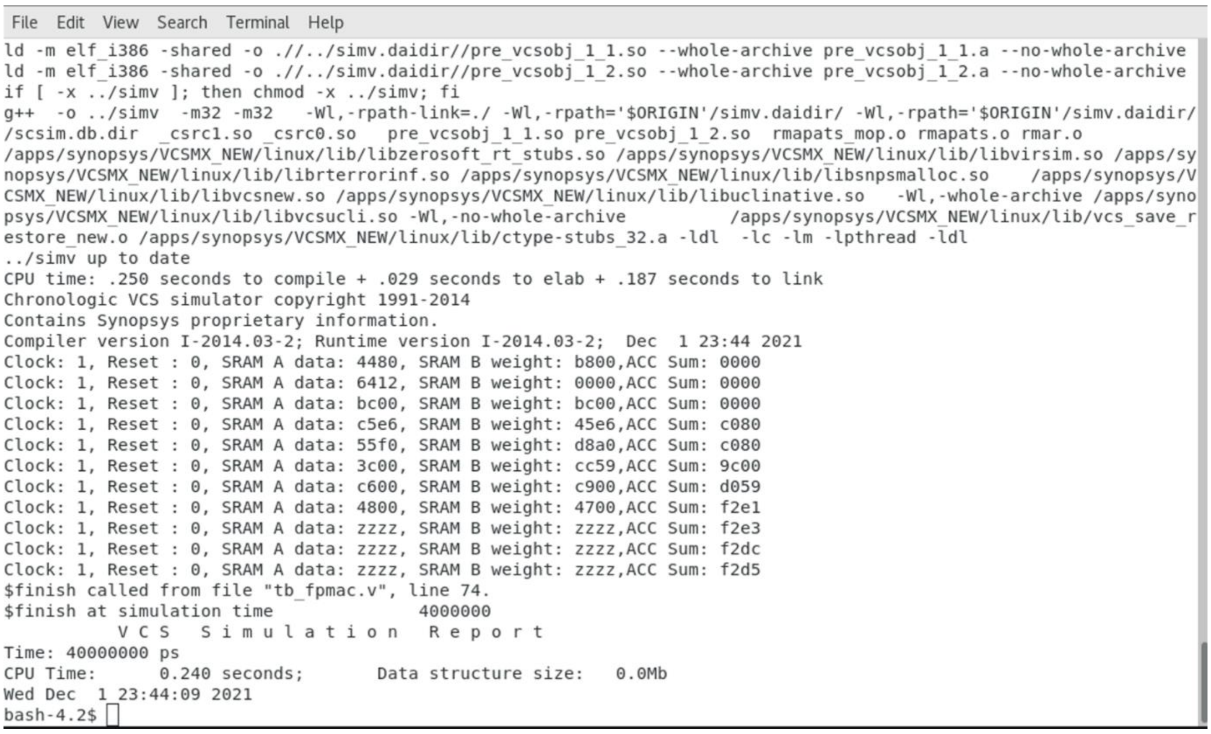
Named: * Edit:

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Strict Preservatior	eak Pull-Up Resist
HEX1[7]	Output	PIN_A19	7	B7_N0	PIN_A19	2.5 V		12mA (default)	2 (default)			
HEX1[6]	Output	PIN_B22	6	B6_N0	PIN_B22	2.5 V		12mA (default)	2 (default)			
HEX1[5]	Output	PIN_C22	6	B6_N0	PIN_C22	2.5 V		12mA (default)	2 (default)			
HEX1[4]	Output	PIN_B21	6	B6_N0	PIN_B21	2.5 V		12mA (default)	2 (default)			
HEX1[3]	Output	PIN_A21	6	B6_N0	PIN_A21	2.5 V		12mA (default)	2 (default)			
HEX1[2]	Output	PIN_B19	7	B7_N0	PIN_B19	2.5 V		12mA (default)	2 (default)			
HEX1[1]	Output	PIN_A20	7	B7_N0	PIN_A20	2.5 V		12mA (default)	2 (default)			
HEX1[0]	Output	PIN_B20	6	B6_N0	PIN_B20	2.5 V		12mA (default)	2 (default)			
HEX2[7]	Output	PIN_D22	6	B6_N0	PIN_D22	2.5 V		12mA (default)	2 (default)			
HEX2[6]	Output	PIN_E17	6	B6_N0	PIN_E17	2.5 V		12mA (default)	2 (default)			
HEX2[5]	Output	PIN_D19	6	B6_N0	PIN_D19	2.5 V		12mA (default)	2 (default)			
HEX2[4]	Output	PIN_C20	6	B6_N0	PIN_C20	2.5 V		12mA (default)	2 (default)			
HEX2[3]	Output	PIN_C19	7	B7_N0	PIN_C19	2.5 V		12mA (default)	2 (default)			
HEX2[2]	Output	PIN_E21	6	B6_N0	PIN_E21	2.5 V		12mA (default)	2 (default)			
HEX2[1]	Output	PIN_E22	6	B6_N0	PIN_E22	2.5 V		12mA (default)	2 (default)			
HEX2[0]	Output	PIN_F21	6	B6_N0	PIN_F21	2.5 V		12mA (default)	2 (default)			
HEX3[7]	Output	PIN_F17	6	B6_N0	PIN_F17	2.5 V		12mA (default)	2 (default)			
HEX3[6]	Output	PIN_F20	6	B6_N0	PIN_F20	2.5 V		12mA (default)	2 (default)			
HEX3[5]	Output	PIN_F19	6	B6_N0	PIN_F19	2.5 V		12mA (default)	2 (default)			
HEX3[4]	Output	PIN_H19	6	B6_N0	PIN_H19	2.5 V		12mA (default)	2 (default)			
HEX3[3]	Output	PIN_J18	6	B6_N0	PIN_J18	2.5 V		12mA (default)	2 (default)			
HEX3[2]	Output	PIN_E19	6	B6_N0	PIN_E19	2.5 V		12mA (default)	2 (default)			
HEX3[1]	Output	PIN_E20	6	B6_N0	PIN_E20	2.5 V		12mA (default)	2 (default)			
HEX3[0]	Output	PIN_F18	6	B6_N0	PIN_F18	2.5 V		12mA (default)	2 (default)			
HEX4[7]	Output	PIN_L19	6	B6_N0	PIN_L19	2.5 V		12mA (default)	2 (default)			
HEX4[6]	Output	PIN_N20	6	B6_N0	PIN_N20	2.5 V		12mA (default)	2 (default)			
HEX4[5]	Output	PIN_N19	6	B6_N0	PIN_N19	2.5 V		12mA (default)	2 (default)			
HEX4[4]	Output	PIN_M20	6	B6_N0	PIN_M20	2.5 V		12mA (default)	2 (default)			
HEX4[3]	Output	PIN_N18	6	B6_N0	PIN_N18	2.5 V		12mA (default)	2 (default)			
HEX4[2]	Output	PIN_L18	6	B6_N0	PIN_L18	2.5 V		12mA (default)	2 (default)			
HEX4[1]	Output	PIN_K20	6	B6_N0	PIN_K20	2.5 V		12mA (default)	2 (default)			
HEX4[0]	Output	PIN_J20	6	B6_N0	PIN_J20	2.5 V		12mA (default)	2 (default)			
clk	Input	PIN_N14	6	B6_N0	PIN_N14	2.5 V		12mA (default)				
keypad_col[3]	Input	PIN_W7	3	B3_N0	PIN_W7	2.5 V		12mA (default)				on
keypad_col[2]	Input	PIN_W8	3	B3_N0	PIN_W8	2.5 V		12mA (default)				on
keypad_col[1]	Input	PIN_W9	3	B3_N0	PIN_W9	2.5 V		12mA (default)				on
keypad_col[0]	Input	PIN_W10	3	B3_N0	PIN_W10	2.5 V		12mA (default)				on
keypad_row[3]	Output	PIN_V7	3	B3_N0	PIN_V7	2.5 V		12mA (default)	2 (default)			off
keypad_row[2]	Output	PIN_V8	3	B3_N0	PIN_V8	2.5 V		12mA (default)	2 (default)			off
keypad_row[1]	Output	PIN_V9	3	B3_N0	PIN_V9	2.5 V		12mA (default)	2 (default)			off
keypad_row[0]	Output	PIN_V10	3	B3_N0	PIN_V10	2.5 V		12mA (default)	2 (default)			off
push_button	Input	PIN_B8	7	B7_N0	PIN_B8	2.5 V		12mA (default)				
rst	Input	PIN_C10	7	B7_N0	PIN_C10	2.5 V		12mA (default)				
state_display[1]	Output	PIN_AB8	3	B3_N0	PIN_AB8	2.5 V		12mA (default)	2 (default)			
state_display[0]	Output	PIN_AB7	3	B3_N0	PIN_AB7	2.5 V		12mA (default)	2 (default)			
<<new node>>												

All Pins

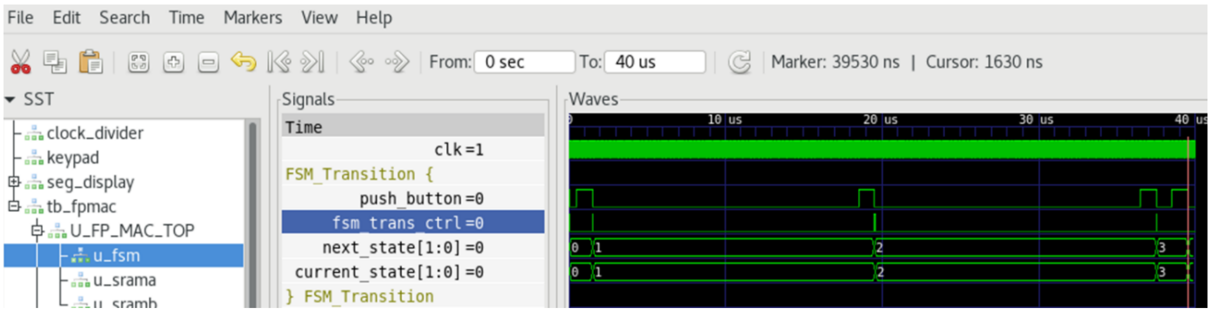
SIMULATION RESULT

The testbench tries to emulate the behavior of the push-button. TB uses two clocks to achieve that. The data and weight input is read from a file “data.txt” & “weight.txt”. Testbench code is available Annexure section.



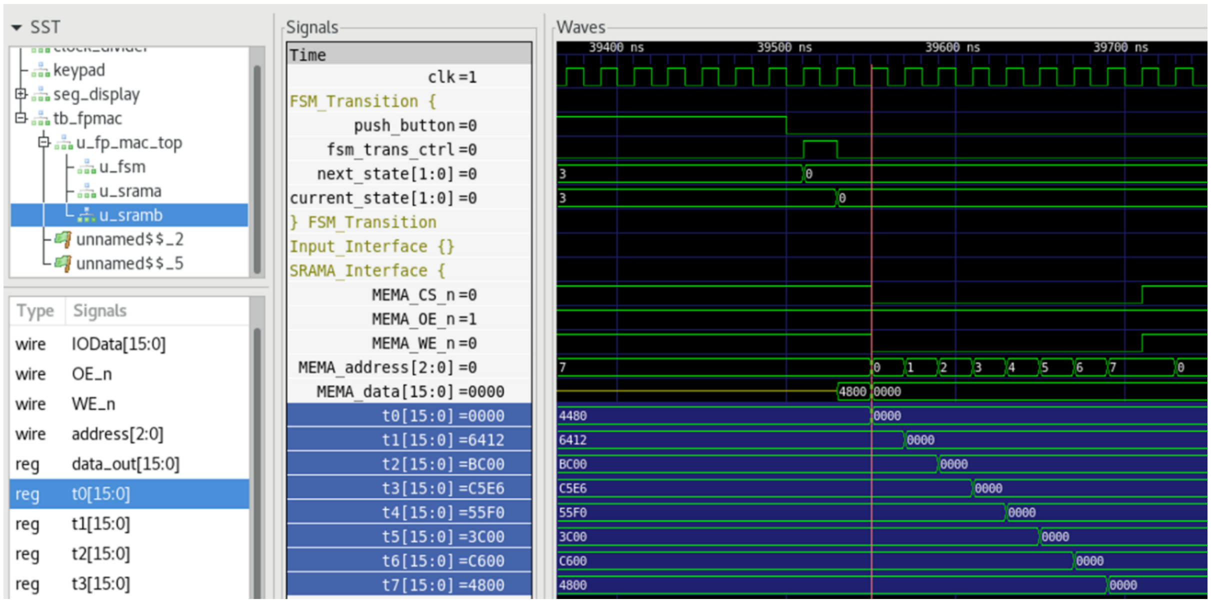
```
ld -m elf_i386 -shared -o .//../simv.daidir//pre_vcsobj_1_1.so --whole-archive pre_vcsobj_1_1.a --no-whole-archive
ld -m elf_i386 -shared -o .//../simv.daidir//pre_vcsobj_1_2.so --whole-archive pre_vcsobj_1_2.a --no-whole-archive
if [ -x ../simv ]; then chmod -x ../simv; fi
g++  -o ../simv  -m32 -m32   -Wl,-rpath-link=./ -Wl,-rpath='$ORIGIN'/simv.daidir/ -Wl,-rpath='$ORIGIN'/simv.daidir/
/scsim.db.dir  _csrc1.so _csrc0.so   pre_vcsobj_1_1.so pre_vcsobj_1_2.so  rmapats_mop.o rmapats.o rmar.o
/apps/synopsys/VCSMX_NEW/linux/lib/libzerosoft_rt_stubs.so /apps/synopsys/VCSMX_NEW/linux/lib/libvirsim.so /apps/sy
nopsys/VCSMX_NEW/linux/lib/librterrorinf.so /apps/synopsys/VCSMX_NEW/linux/lib/libsnpsmalloc.so    /apps/synopsys/V
CSMX_NEW/linux/lib/libvcsnew.so /apps/synopsys/VCSMX_NEW/linux/lib/libuclinative.so   -Wl,-whole-archive /apps/syno
psys/VCSMX_NEW/linux/lib/libvcsucli.so -Wl,-no-whole-archive         /apps/synopsys/VCSMX_NEW/linux/lib/vcs_save_r
estore_new.o /apps/synopsys/VCSMX_NEW/linux/lib/ctype-stubs_32.a -ldl  -lc -lm -lpthread -ldl
../simv up to date
CPU time: .250 seconds to compile + .029 seconds to elab + .187 seconds to link
Chronologic VCS simulator copyright 1991-2014
Contains Synopsys proprietary information.
Compiler version I-2014.03-2; Runtime version I-2014.03-2;  Dec  1 23:44 2021
Clock: 1, Reset : 0, SRAM A data: 4480, SRAM B weight: b800,ACC Sum: 0000
Clock: 1, Reset : 0, SRAM A data: 6412, SRAM B weight: 0000,ACC Sum: 0000
Clock: 1, Reset : 0, SRAM A data: bc00, SRAM B weight: bc00,ACC Sum: 0000
Clock: 1, Reset : 0, SRAM A data: c5e6, SRAM B weight: 45e6,ACC Sum: c080
Clock: 1, Reset : 0, SRAM A data: 55f0, SRAM B weight: d8a0,ACC Sum: c080
Clock: 1, Reset : 0, SRAM A data: 3c00, SRAM B weight: cc59,ACC Sum: 9c00
Clock: 1, Reset : 0, SRAM A data: c600, SRAM B weight: c900,ACC Sum: d059
Clock: 1, Reset : 0, SRAM A data: 4800, SRAM B weight: 4700,ACC Sum: f2e1
Clock: 1, Reset : 0, SRAM A data: zzzz, SRAM B weight: zzzz,ACC Sum: f2e3
Clock: 1, Reset : 0, SRAM A data: zzzz, SRAM B weight: zzzz,ACC Sum: f2dc
Clock: 1, Reset : 0, SRAM A data: zzzz, SRAM B weight: zzzz,ACC Sum: f2d5
$finish called from file "tb_fpmac.v", line 74.
$finish at simulation time            4000000
           V C S   S i m u l a t i o n   R e p o r t
Time: 40000000 ps
CPU Time:      0.240 seconds;       Data structure size:   0.0Mb
Wed Dec  1 23:44:09 2021
bash-4.2$
```

FSM Transition: In the simulation, the push_button signal emulates the behavior of push-button FPGA. fsm_trans_ctrl is a pulse signal generated by detecting the falling edge of the push_button. All state transitions 0-1-2-3-0 are in the waveform.

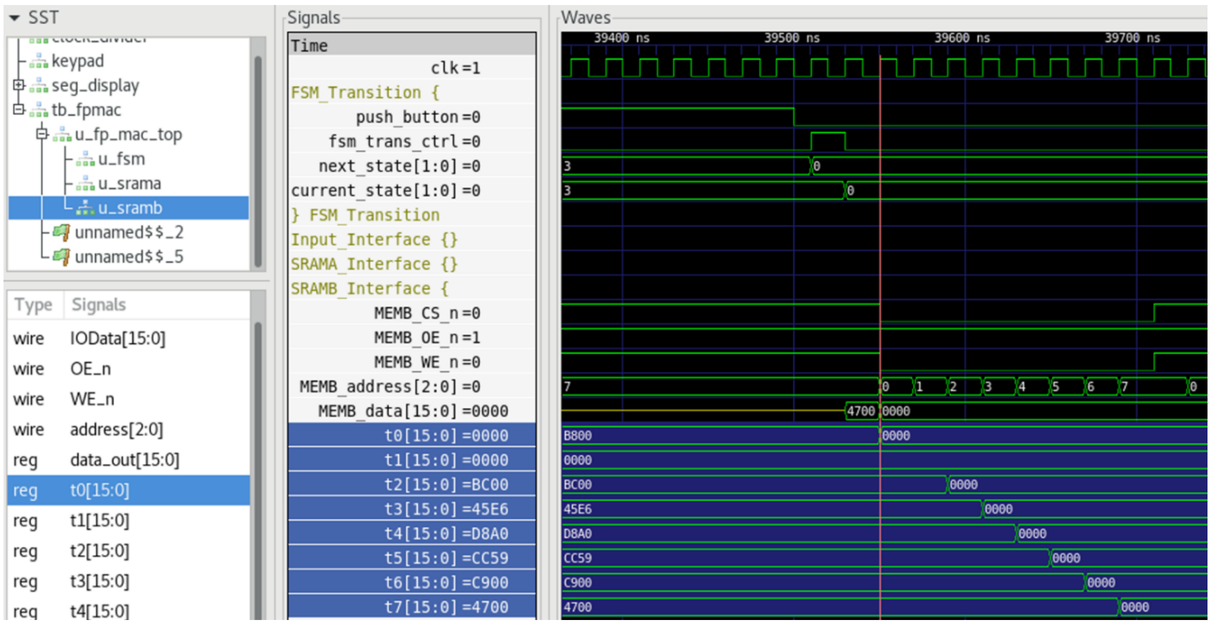


From: 0 sec To: 40 us Marker: 39530 ns | Cursor: 1630 ns

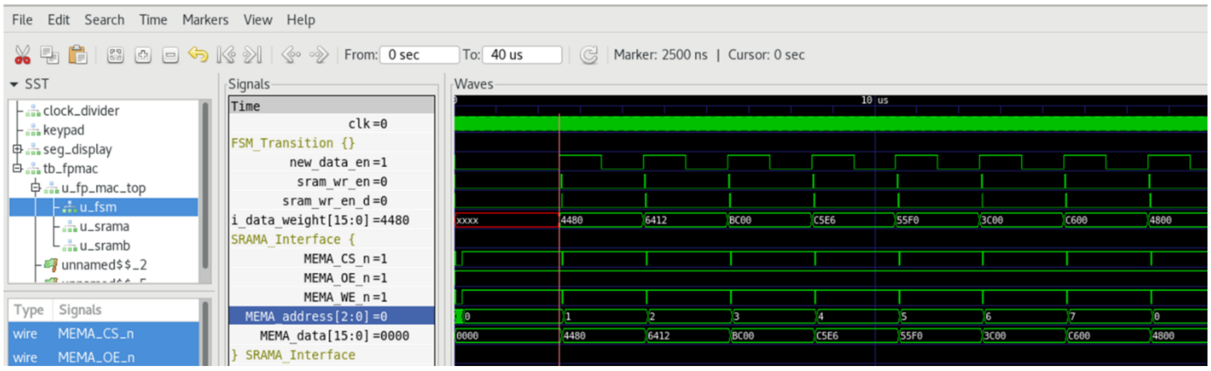
SRAM A Reset: Write 0s into SRAM A in the reset state t0 - t7 is debugged signals for SRAM.



SRAM B Reset: Write 0s into SRAM B in the reset state

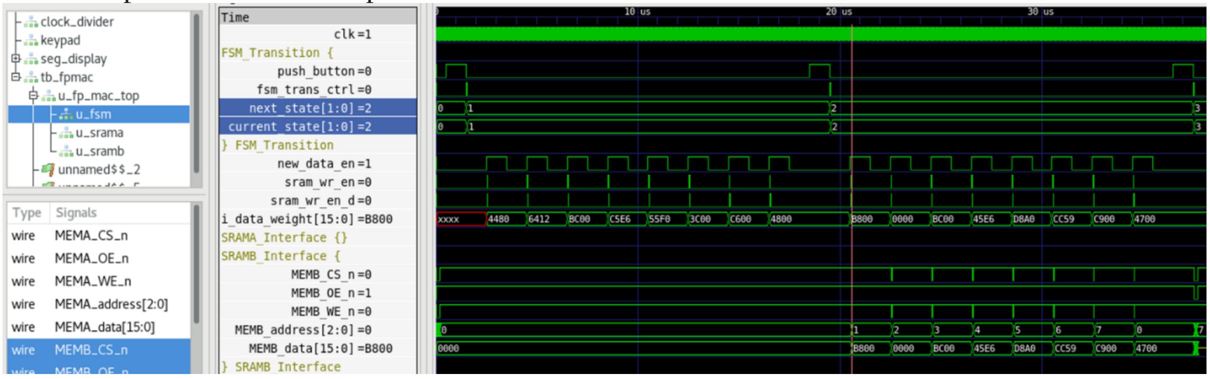


SRAM A Write: Write to SRAM A happens in state 1. i_data_weight is the floating-point input to the MAC unit; input comes with valid new_data_en. MAC units create a pulse sram_wr_en in sync with the 50MHz clock and use that pulse signal to write data into SRAM. SRAM A signals are represented with the prefix MEMA.



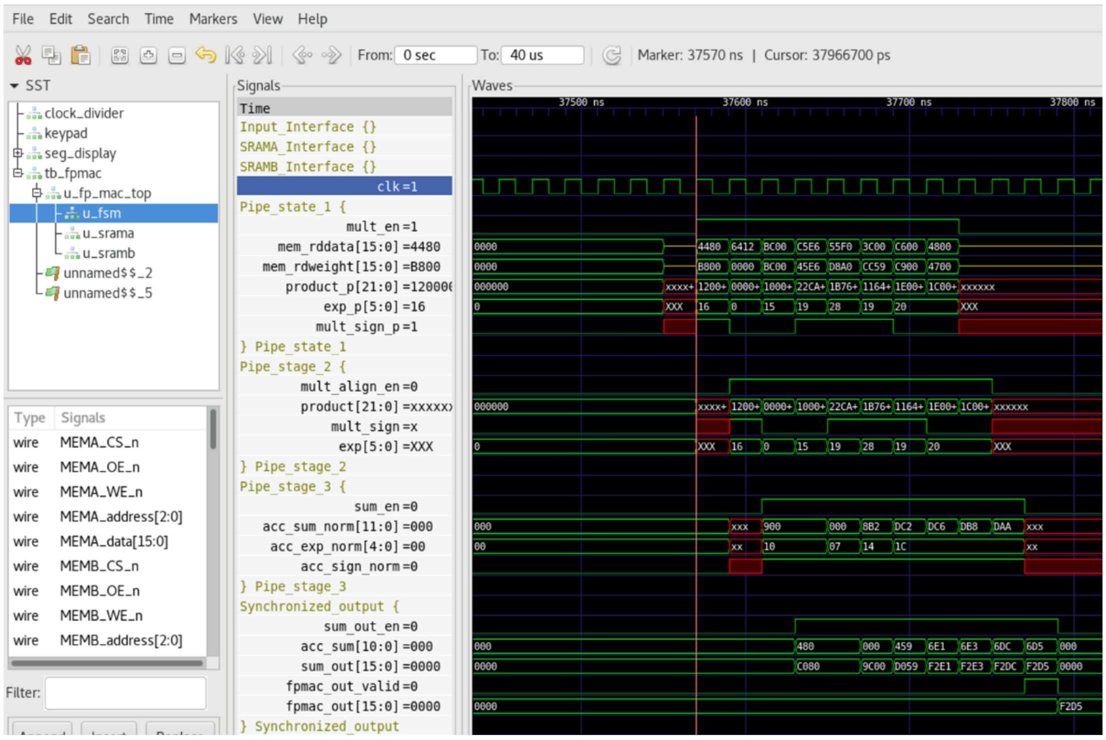
```
Time
clk=0
FSM_Transition {}
new_data_en=1
sram_wr_en=0
sram_wr_en_d=0
i_data_weight[15:0]=4480
SRAMA_Interface {
MEMA_CS_n=1
MEMA_OE_n=1
MEMA_WE_n=1
MEMA_address[2:0]=0
MEMA_data[15:0]=0000
} SRAMA_Interface
```

SRAM B Write: Write to SRAM B happens in state 2. SRAM B signals are represented with the prefix MEMB.

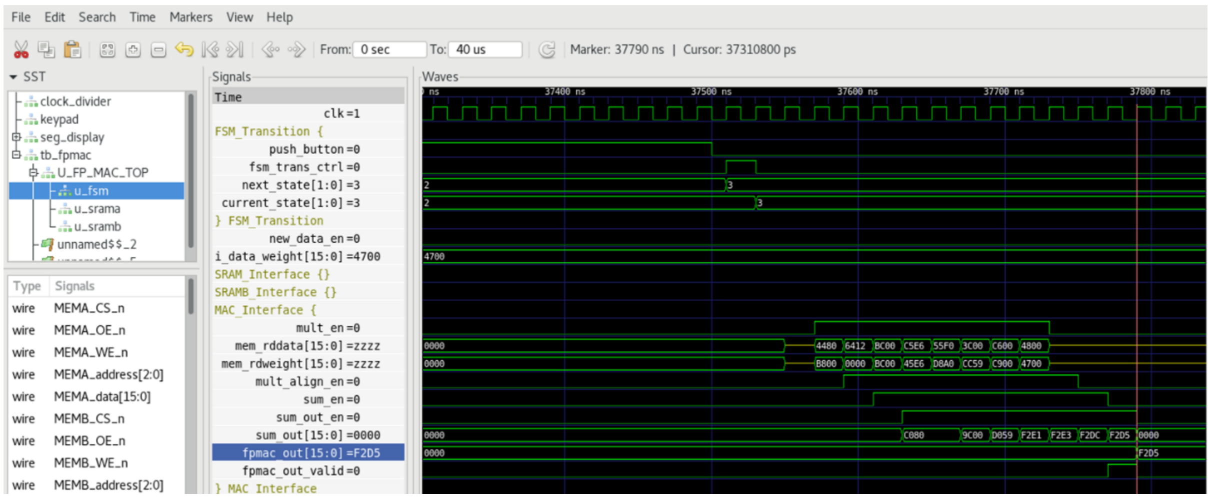


```
Time
clk=1
FSM_Transition {
push_button=0
fsm_trans_ctrl=0
next_state[1:0]=2
current_state[1:0]=2
} FSM_Transition
new_data_en=1
sram_wr_en=0
sram_wr_en_d=0
i_data_weight[15:0]=B800
SRAMA_Interface {}
SRAMB_Interface {
MEMB_CS_n=0
MEMB_OE_n=1
MEMB_WE_n=0
MEMB_address[2:0]=0
MEMB_data[15:0]=B800
} SRAMB_Interface
```

MAC Calculation:The x value in the waveform is due to the SRAM floating value z. However, these x's don't propagate to the output because they are gated using enable signals.



```
Time
Input_Interface {}
SRAMA_Interface {}
SRAMB_Interface {}
clk=1
Pipe_state_1 {
mult_en=1
mem_rddata[15:0]=4480
mem_rdweight[15:0]=B800
product_p[21:0]=120000
exp_p[5:0]=16
mult_sign_p=1
} Pipe_state_1
Pipe_stage_2 {
mult_align_en=0
product[21:0]=xxxxxx
mult_sign=x
exp[5:0]=XXX
} Pipe_stage_2
Pipe_stage_3 {
sum_en=0
acc_sum_norm[11:0]=000
acc_exp_norm[4:0]=00
acc_sign_norm=0
} Pipe_stage_3
Synchronized_output {
sum_out_en=0
acc_sum[10:0]=000
sum_out[15:0]=0000
fpmac_out_valid=0
fpmac_out[15:0]=0000
} Synchronized_output
```


File Edit Search Time Markers View Help
From: 0 sec To: 40 us
Marker: 37790 ns | Cursor: 37310800 ps
SST
clock_divider
keypad
seg_display
tb_fpmac
U_FP_MAC_TOP
u_fsm
u_srama
u_sramb
unnamed$$_2
Type Signals
wire MEMA_CS_n
wire MEMA_OE_n
wire MEMA_WE_n
wire MEMA_address[2:0]
wire MEMA_data[15:0]
wire MEMB_CS_n
wire MEMB_OE_n
wire MEMB_WE_n
wire MEMB_address[2:0]
Signals
Time
clk=1
FSM_Transition {
push_button=0
fsm_trans_ctrl=0
next_state[1:0]=3
current_state[1:0]=3
} FSM_Transition
new_data_en=0
i_data_weight[15:0]=4700
SRAM_Interface {}
SRAMB_Interface {}
MAC_Interface {
mult_en=0
mem_rddata[15:0]=zzzz
mem_rdweight[15:0]=zzzz
mult_align_en=0
sum_en=0
sum_out_en=0
sum_out[15:0]=0000
fpmac_out[15:0]=F2D5
fpmac_out_valid=0
} MAC Interface
Waves
37400 ns
37500 ns
37600 ns
37700 ns
37800 ns
2
3
2
3
4700
0000
4480
6412
BC00
C5E6
55F0
3C00
C600
4800
0000
B800
0000
BC00
45E6
D8A0
CC59
C900
4700
0000
C080
9C00
D059
F2E1
F2E3
F2DC
F2D5
0000
0000
F2D5

CONCLUSION

The floating-point MAC unit project helps to understand the concept of pipelining. As the system works on 50 MHz, the time period is 20 ns. This time period is sufficient enough to implement 3-stage pipelining for addition, multiplication, and normalization to operate on multiple inputs. Without the pipelining, the system would require a slower frequency clock to execute the same set of operations. Moreover, the project also highlights the solution to tackle the debouncing effect of the keypad.

CONTRIBUTION

Work	Team Member
Keypad	Surbhi
Clock Divider	Surbhi
SRAM	Surbhi
MAC	Nidhu
7 segment display	Purav
Arduino	Purav
FPGA Debugging	Nidhu
Testbench	Purav

REFERENCES

[1] DE10-Lite User Manual

[2] Guide to Arduino Uno: arduino guide

[3] 4x4 Matrix Membrane Keypad: keypad guide

[4] I2C Serial Interface 1602 LCD Module: LCD guide

[5] *Digital Systems Design Using Verilog* by Charles Roth, Lizy K. John, Byeong Kil Lee

ANNEXURE (CODE)

- **Top Module**

```
// Group 13
// File Name : fp_mac.v
// Functionality: Top level module for EE271 Flotiang MAC project

`include "mac_fsm.v"
`include "SRAM8x16.v"
`include "clock_divider.v"
`include "keypad.v"
`include "seg_display.v"

module fp_mac_top(
  input clk,
  input rst, // Active high
  input push_button, // state transtion control input, controlled by push button of FPGA
  input [3:0] keypad_col,
  output [3:0] keypad_row,
        output [1:0] state_display,
  output [7:0] HEX4,
  output [7:0] HEX3,
  output [7:0] HEX2,
  output [7:0] HEX1
);

wire MEMA_CS_n;
wire MEMA_WE_n;
wire MEMA_OE_n;
wire [2:0] MEMA_address;
wire [15:0] MEMA_data;

wire MEMB_CS_n;
wire MEMB_WE_n;
wire MEMB_OE_n;
wire [2:0] MEMB_address;
wire [15:0] MEMB_data;

wire keypad_clk;
wire key_detect;
wire [3:0] key_value;
wire new_data_en;
wire [15:0] fp_dt_wt_mux;
wire [15:0] fpmac_out;


// Clock Divider for Keypad
clock_divider u_clk_div(
   .clk_in (clk),
   .rst    (rst),
   .clk_out (keypad_clk)
);

// 4x4 Matrix Keypad
keypad u_keypad(
   .clk        (keypad_clk),
   .rst        (rst),
   .keypad_col (keypad_col),
   .keypad_row (keypad_row),
   .key_detect (key_detect),
   .key_value  (key_value)
);

// Seven Segment display
seg_display u_display(
   .clk           (keypad_clk),
   .rst           (rst),
   .state_display (state_display),
   .fpmac_out     (fpmac_out),
   .key_detect    (key_detect),
   .key_value     (key_value),
           .new_data_en   (new_data_en),
   .fp_dt_wt_mux  (fp_dt_wt_mux),
   .HEX4          (HEX4),
   .HEX3          (HEX3),
   .HEX2          (HEX2),
   .HEX1          (HEX1)
);

// Instantiate FSM Contorl
mac_fsm u_fsm(
   .clk           (clk),
   .rst           (rst),
   .push_button   (push_button),
           .new_data_en   (new_data_en),
   .i_data_weight (fp_dt_wt_mux),
   .state_display (state_display),
   .fpmac_out     (fpmac_out),
   .MEMA_CS_n     (MEMA_CS_n),
   .MEMA_WE_n     (MEMA_WE_n),
   .MEMA_OE_n     (MEMA_OE_n),
   .MEMA_address  (MEMA_address),
   .MEMA_data     (MEMA_data),
   .MEMB_CS_n     (MEMB_CS_n),
   .MEMB_WE_n     (MEMB_WE_n),
   .MEMB_OE_n     (MEMB_OE_n),
   .MEMB_address  (MEMB_address),
   .MEMB_data     (MEMB_data)
);

// Instantiate SRAM A for Data input
SRAM8x16 u_srama(
   .CS_n    (MEMA_CS_n),
   .WE_n    (MEMA_WE_n),
   .OE_n    (MEMA_OE_n),
   .address (MEMA_address),
   .IOData  (MEMA_data)
);

// Instantiate SRAM B for Weight input
SRAM8x16 u_sramb(
   .CS_n    (MEMB_CS_n),
   .WE_n    (MEMB_WE_n),
   .OE_n    (MEMB_OE_n),
   .address (MEMB_address),
   .IOData  (MEMB_data)
);


Endmodule
```

- **Keypad Interface**

```
  // Group 13
  module keypad(
    input clk,
    input rst,
    input [3:0] keypad_col,
    output reg [3:0] keypad_row,
    output reg key_detect,
    output reg [3:0] key_value
  );

  parameter st_row_pulldown = 0, st_key_read = 1, st_row_pullup = 2;
  reg [1:0] current_state;
  reg [1:0] row_idx;
  reg [15:0] keypad_array; // An array to to detect the keypad value

  integer i;
  always @ (posedge clk or posedge rst) begin
    if(rst) begin
      current_state <= st_row_pulldown;
      row_idx      <= 2'd0;
      keypad_row    <= 4'bzzzz; // No rows are selected
      key_detect    <= 1'b0;
      keypad_array  <= 16'hFFFF; // Key column Active low detect
      key_value     <= 4'd0;
    end
    else begin
      case(current_state)

        st_row_pulldown : begin
          keypad_row[row_idx] <= 1'b0; // Select one row(pulldown) at a time based on
  the index value
          current_state      <= st_key_read;
        end

        st_key_read : begin // Read key in the selected row
          if((keypad_row == 4'b1110) || (keypad_row == 4'b1101) || (keypad_row ==
  4'b1011))begin // First 3 rows of the keypad
            // 0th(1,2,3), 1st(4,5,6),2nd(7,8,9) and 3rd(A,B,C) columns
            for(i = 0; i <= 3; i = i + 1) begin
              if(~keypad_col[i]) begin

          if((keypad_array[3*row_idx+i+1] != keypad_col[i]) && (i < 3)) begin
                                                          key_detect <= 1'b1;
                                                          key_value  <= (3*row_idx
  + i + 1); // decode the key value
                                                  end
                                                  if((keypad_array[row_idx+10]  !=
  keypad_col[i]) && (i == 3)) begin
```

```
                                                        key_detect <= 1'b1;
                                                        key_value   <= row_idx +
10;
                                                end
            end
                                        if(i == 3) begin
                                                keypad_array[row_idx+10]      <=
keypad_col[i];
                                        end
                                        else begin
                                                keypad_array[3*row_idx+i+1]
<= keypad_col[i]; // Keypad detect, 1 : key not yet pressed, 0 : key pressed
                                        end
          end
        end

        if(keypad_row == 4'b0111) begin // last row, special case

                                keypad_array[15] <= keypad_col[0];
                                keypad_array[0]  <= keypad_col[1];
                                keypad_array[14] <= keypad_col[2];
                                keypad_array[13] <= keypad_col[3];

          if(~keypad_col[0] && keypad_array[15] != keypad_col[0]) begin // detect F
(*)
            key_detect <= 1'b1;
            key_value  <= 4'hF;
          end
          if(~keypad_col[1] && keypad_array[0] != keypad_col[1]) begin // detect 0
            key_detect <= 1'b1;
            key_value  <= 4'h0;
          end
          if(~keypad_col[2] && keypad_array[14] != keypad_col[2]) begin // detect E
(#)
            key_detect <= 1'b1;
            key_value  <= 4'hE;
          end
          if(~keypad_col[3] && keypad_array[13] != keypad_col[3]) begin // detect D
            key_detect <= 1'b1;
            key_value  <= 4'hD;
          end
        end

        current_state <= st_row_pullup;
```

```
      end

      st_row_pullup : begin
        current_state <= st_row_pulldown;
        row_idx       <= row_idx + 2'd1;
        keypad_row    <= 4'bzzzz;
        key_detect    <= 1'b0;
      end

      default: begin
        current_state <= st_row_pulldown;
      end
    endcase
  end
end

endmodule
```

- **Clock Divider**

```
// Group 13

module clock_divider(
  input clk_in, // 50MHz FPGA clock
  input rst, // Active high rst
  output reg clk_out // 1KHz clock for keypad
);


reg [24:0] count;
//reg [14:0] count;

always @(posedge clk_in or posedge rst) begin
  if(rst) begin
    count   <= 15'd0;
    clk_out <= 1'b0;
  end
  else if(count == 0) begin
    count <= 24999;
    clk_out <= ~clk_out;
  end
  else begin
    count <= count - 15'd1;
  end
end

endmodule

/*module clock_divider(
  input clk_in, // 50MHz FPGA clock
  input rst, // Active high rst
  output reg clk_out // 1KHz clock for keypad
);


//reg [24:0] count;
//reg [14:0] count;
parameter SIZE = 15;
parameter CLK_CNT = 24999;

reg [SIZE-1:0] count;

always @(posedge clk_in or posedge rst) begin
  if(rst) begin
    count   <= 'd0;
    clk_out <= 1'b0;
  end
  else if(count == 0) begin
    count <= CLK_CNT;
    clk_out <= ~clk_out;
  end
  else begin
    count <= count - 'd1;
  end
end

endmodule*/
```

- **Segment Display**

```
// Group 13
module seg_display(
  input clk,
  input rst,
  input [1:0] state_display,
        input [15:0] fpmac_out,
  input key_detect,
  input [3:0] key_value,
        output reg new_data_en,
  output reg [15:0] fp_dt_wt_mux,
  output reg [7:0] HEX4,
  output reg [7:0] HEX3,
  output reg [7:0] HEX2,
  output reg [7:0] HEX1
);

wire [7:0] hex0;
reg [7:0] hex1,hex2,hex3,hex4;
reg [15:0] half_fpnum;
reg [2:0] ip_count;
wire new_data_en_p;
wire [15:0] fpnum_p;

assign hex0 = seg_decode(key_value);

// Shift the hex and input values
always @ (posedge clk or posedge rst) begin
  if(rst) begin
    half_fpnum <= 16'd0;
    hex4  <= 8'hFF;
    hex3  <= 8'hFF;
    hex2  <= 8'hFF;
    hex1  <= 8'hFF;
  end
  else if(key_detect) begin
    half_fpnum[15:12] <= half_fpnum[11:8];
    half_fpnum[11:8]  <= half_fpnum[7:4];
    half_fpnum[7:4]   <= half_fpnum[3:0];
    half_fpnum[3:0]   <= key_value;

    hex4  <= hex3;
    hex3  <= hex2;
    hex2  <= hex1;
    hex1  <= hex0;
  end
end

always @ (posedge clk or posedge rst) begin
        if(rst) begin
                ip_count <= 3'd0;
        end
        else if(ip_count == 3'd4) begin
                ip_count <= 3'd0;
        end
        else if (key_detect) begin
                ip_count <= ip_count + 3'd1;
        end
end

assign new_data_en_p = (ip_count == 3'd4) ? 1'b1: 1'b0;
assign fpnum_p       = half_fpnum;

// Enable signal for keypad data, send it to MAC module
always @ (posedge clk or posedge rst) begin
        if(rst) begin
                new_data_en  <= 1'b0;
                fp_dt_wt_mux <= 16'd0;
        end
        else begin
                new_data_en  <= new_data_en_p;
                fp_dt_wt_mux <= fpnum_p;
        end
end

always @ (posedge clk or posedge rst) begin
  if(rst) begin
    HEX4 <= 8'b00000000; // display 8.(Including decimal point)
    HEX3 <= 8'b00000000; // display 8.
    HEX2 <= 8'b00000000; // display 8.
    HEX1 <= 8'b00000000; // display 8.
  end
  else if(state_display == 2'd0) begin // reset state
    HEX4 <= 8'b11111111; // display off
    HEX3 <= 8'b11111111; // display off
    HEX2 <= 8'b11111111; // display off
    HEX1 <= 8'b11111111; // display off
  end
  else if((state_display == 2'd1) ||
       (state_display == 2'd2)) begin // input 1 / input 2 state
    HEX4 <= hex4;
    HEX3 <= hex3;
    HEX2 <= hex2;
    HEX1 <= hex1;
  end
  else if(state_display == 2'd3) begin // MAC state
    HEX4 <= seg_decode(fpmac_out[15:12]);
    HEX3 <= seg_decode(fpmac_out[11:8]);
    HEX2 <= seg_decode(fpmac_out[7:4]);
    HEX1 <= seg_decode(fpmac_out[3:0]);
  end
end

// Segment display decode function for FP MAC SUM output

function [7:0] seg_decode(input [3:0] fpvalue);
 case(fpvalue)
    0:seg_decode  = 8'b11000000; // 0
    1:seg_decode  = 8'b11111001; // 1
    2:seg_decode  = 8'b10100100; // 2
    3:seg_decode  = 8'b10110000; // 3
    4:seg_decode  = 8'b10011001; // 4
    5:seg_decode  = 8'b10010010; // 5
    6:seg_decode  = 8'b10000010; // 6
    7:seg_decode  = 8'b11111000; // 7
    8:seg_decode  = 8'b10000000; // 8
    9:seg_decode  = 8'b10010000; // 9
    10:seg_decode = 8'b10001000; // A
    11:seg_decode = 8'b10000011; // b
    12:seg_decode = 8'b11000110; // C
    13:seg_decode = 8'b10100001; // d
    14:seg_decode = 8'b10000110; // E
    15:seg_decode = 8'b10001110; // F
  endcase

endfunction


endmodule
```

- **SRAMs**

```
// Group 13
// File Name : SRAM8x16.v, asynchronous read and write
// SRAM Depth 8, Width: 16

module SRAM8x16(CS_n,WE_n,OE_n,address,IOData);

parameter SADDR_WIDTH = 3;
parameter SDATA_WIDTH = 16;
parameter SRAM_DEPTH  = 1 << SADDR_WIDTH; // 8 locations

input CS_n;
input WE_n;
input OE_n;
input [SADDR_WIDTH-1:0] address;
inout [SDATA_WIDTH-1:0] IOData;

reg [SDATA_WIDTH-1:0] smem [SRAM_DEPTH-1:0];

reg [15:0] data_out;

// Write data into smem
always @ (*) begin
  if(~CS_n & ~WE_n) begin
    smem[address] = IOData;
  end
end

// Read data
assign IOData = (~CS_n & ~OE_n) ? smem[address] : 16'bzzzzzzzzzzzzzzzz;

// Debug logic
reg [15:0] t0,t1,t2,t3,t4,t5,t6,t7;

always @ (*) begin
t0 = smem[0];
t1 = smem[1];
t2 = smem[2];
t3 = smem[3];
t4 = smem[4];
t5 = smem[5];
t6 = smem[6];
t7 = smem[7];
end
// end of debug logic

endmodule
```

- **MAC unit**

```
// Group 13
// File name mac_fsm.v
// Functionality : FSM and control for MAC

module mac_fsm(
  input clk,
  input rst, // Active high
  input push_button,
        input new_data_en,
  input [15:0] i_data_weight, // Data or Weight Input

  output reg [1:0] state_display,
  output reg [15:0] fpmac_out,
  // SRAM A interface
  output  MEMA_CS_n,
  output  MEMA_WE_n,
  output  MEMA_OE_n,
  output  [2:0] MEMA_address,
  inout   [15:0] MEMA_data,
  // SRAM B interface
  output  MEMB_CS_n,
  output  MEMB_WE_n,
  output  MEMB_OE_n,
  output  [2:0] MEMB_address,
  inout  [15:0] MEMB_data
);

parameter st_reset = 0, st_smemwr_dt = 1, st_smemwr_wt = 2, st_mac = 3;

reg [1:0] state_display_p;
reg [1:0] current_state,next_state;

reg [3:0] sram_rst_cnt_p, sram_rst_cnt;

reg [2:0] srama_adrcnt_p,srama_adrcnt,srama_adrcnt_d;
reg [2:0] sramb_adrcnt_p,sramb_adrcnt,sramb_adrcnt_d;

reg smema_cs_n,smema_we_n,smema_oe_n;
reg [2:0] smema_addr;
reg [15:0] smema_data;

reg smemb_cs_n,smemb_we_n,smemb_oe_n;
reg [2:0] smemb_addr;
reg [15:0] smemb_data;

reg smema_cs_n_d,smema_we_n_d,smema_oe_n_d;
reg [2:0] smema_addr_d;
reg [15:0] smema_data_d;

reg smemb_cs_n_d,smemb_we_n_d,smemb_oe_n_d;
reg [2:0] smemb_addr_d;
reg [15:0] smemb_data_d;

reg [15:0] mem_rddata_p, mem_rdweight_p;
reg [15:0] mem_rddata, mem_rdweight;

reg [2:0] sram_rdcnt_p, sram_rdcnt, sram_rdcnt_d,sram_rdcnt_d1;
reg sram_wr_done_p, sram_wr_done;
reg sram_rd_done_p, sram_rd_done;
reg pipeline_en_p, pipeline_en;

reg push_button_d,fsm_trans_ctrl;
reg new_data_en_d,new_data_en_d1,sram_wr_en,sram_wr_en_d;

// MAC Signals
reg mult_sign_p,mult_sign,mult_sign_d;
reg [21:0] product_p,product;
reg [9:0] product_norm;
reg [5:0] exp_p,exp;
reg [4:0] exp_norm;
wire [16:0] fp_mult_out; // sign(1bit) , exponent(5bits), multiplied result(22bit,
including hidden 1)
wire [15:0] fp_mult_result;
reg mult_en, mult_align_en;

reg [4:0] acc_exp_p;
reg [4:0] acc_exp;
reg acc_sign_p,acc_sign;
wire exp_equal;
reg [10:0] fp_a_p,fp_b_p; // including hidden bit
wire [11:0] acc_sum_p;
reg [10:0] acc_sum;
reg [11:0] acc_sum_norm;
reg [4:0] acc_exp_norm;
reg acc_sign_norm;
reg acc_sum_ovf;
reg [4:0] exp_pos,exp_shift;
reg sum_en, sum_out_en;

reg [2:0] scnt;
wire [15:0] sum_out;
wire fpmac_out_valid;

always @ (*) begin

  next_state = current_state;

  state_display_p = 2'd0;

        sram_rst_cnt_p = sram_rst_cnt;
  srama_adrcnt_p = srama_adrcnt;
  sramb_adrcnt_p = sramb_adrcnt;

  // Memory interface signals
  smema_cs_n = 1'b1;
  smema_we_n = 1'b1;
  smema_oe_n = 1'b1;
  smema_addr = srama_adrcnt_d;
  smema_data = smema_data_d;

  smemb_cs_n = 1'b1;
  smemb_we_n = 1'b1;
  smemb_oe_n = 1'b1;
  smemb_addr = sramb_adrcnt_d;
  smemb_data = smemb_data_d;

  mem_rddata_p   = 16'd0;
  mem_rdweight_p = 16'd0;
  pipeline_en_p  = 1'b0;
        sram_rdcnt_p   = sram_rdcnt;
        sram_wr_done_p = sram_wr_done;
        sram_rd_done_p = sram_rd_done;

  case(current_state)
    st_reset : begin
      state_display_p = 2'd0;
```

```
                              sram_rst_cnt_p = sram_rst_cnt + 4'd1;

         if(~sram_wr_done) begin
                              // Reset(initialize) SRAM A with 0
                              smema_cs_n      = 1'b0;
                              smema_we_n      = 1'b0;
                              smema_addr      = srama_adrcnt;
                              smema_data      = 16'd0;
                              srama_adrcnt_p  = srama_adrcnt + 3'd1;

                              // Reset(initialize) SRAM B with 0
                              smemb_cs_n      = 1'b0;
                              smemb_we_n      = 1'b0;
                              smemb_addr      = sramb_adrcnt;
                              smemb_data      = 16'd0;
                              sramb_adrcnt_p  = sramb_adrcnt + 3'd1;
         sram_wr_done_p  = (srama_adrcnt == 3'd7) ? 1'b1 : 1'b0;
       end

         if(fsm_trans_ctrl) begin // Input control push button or switch
           next_state     = 2'b01;
           srama_adrcnt_p = 3'd0;
           sramb_adrcnt_p = 3'd0;
                                   sram_rst_cnt_p = 4'd0;
           sram_wr_done_p = 1'b0;
         end
       end

       st_smemwr_dt : begin // Need a control from keypad after the entire key is entered
         state_display_p = 2'd1;
                    if(sram_wr_en_d) begin
                                   smema_cs_n      = 1'b0;
                                   smema_we_n      = 1'b0;
                                   smema_addr      = srama_adrcnt;
                                   smema_data      = i_data_weight;
                                   srama_adrcnt_p  = srama_adrcnt + 3'd1;
                              end
         if(fsm_trans_ctrl) begin
           next_state     = 2'b10;
           srama_adrcnt_p = 3'd0;
         end
       end
```

```
      st_smemwr_wt : begin
        state_display_p = 2'd2;
                         if(sram_wr_en_d) begin
                                 smemb_cs_n      = 1'b0;
                                 smemb_we_n      = 1'b0;
                                 smemb_addr      = sramb_adrcnt;
                                 smemb_data      = i_data_weight;
                                 sramb_adrcnt_p = sramb_adrcnt + 3'd1;
                         end
        if(fsm_trans_ctrl) begin
          next_state     = 2'b11;
          sramb_adrcnt_p = 3'd0;
        end
      end

      st_mac : begin
        state_display_p = 2'd3;
        // Read SRAM A
        if(srama_adrcnt_d < 7) begin
                                 smema_cs_n      = 1'b0;
                                 smema_oe_n      = 1'b0;
                                 smema_addr      = srama_adrcnt;
                                 srama_adrcnt_p       =    (srama_adrcnt    ==    3'd7)    ?
srama_adrcnt : srama_adrcnt + 3'd1;
                         end

        mem_rddata_p    = MEMA_data;

        // Read SRAM B
        if(sramb_adrcnt_d < 7) begin
                                 smemb_cs_n      = 1'b0;
                                 smemb_oe_n      = 1'b0;
                                 smemb_addr      = sramb_adrcnt;
                                 sramb_adrcnt_p     =     (sramb_adrcnt     ==     3'd7)     ?
sramb_adrcnt :sramb_adrcnt + 3'd1;
                         end
        mem_rdweight_p  = MEMB_data;

                         sram_rdcnt_p         =   (sram_rdcnt   ==   3'd7)   ?   sram_rdcnt   :
sram_rdcnt + 3'd1;
        pipeline_en_p   = (sram_rdcnt_d == 3'd7) ? 1'b0 : 1'b1;

        if(fsm_trans_ctrl) begin
          next_state     = 2'b00;
```

```
                                        srama_adrcnt_p = 3'd0;
                                        sramb_adrcnt_p = 3'd0;
                                        sram_rdcnt_p   = 3'd0;
                                        sram_rd_done_p = 1'b0;
          end
        end

      endcase
    end

    always @(posedge clk or posedge rst) begin
      if(rst) begin
        current_state  <= st_reset;
        state_display  <= 2'd0;
                    sram_rst_cnt   <= 4'd0;
        srama_adrcnt   <= 3'd0;
        sramb_adrcnt   <= 3'd0;
                    srama_adrcnt_d <= 3'd0;
        sramb_adrcnt_d <= 3'd0;
                    sram_rdcnt     <= 3'd0;
                    sram_rdcnt_d   <= 3'd0;
                    sram_rdcnt_d1  <= 3'd0;
                    sram_wr_done   <= 1'b0;
                    sram_rd_done   <= 1'b0;
      end
      else begin
        current_state  <= next_state;
        state_display  <= state_display_p;
                    sram_rst_cnt   <= sram_rst_cnt_p;
        srama_adrcnt   <= srama_adrcnt_p;
        sramb_adrcnt   <= sramb_adrcnt_p;
        srama_adrcnt_d <= srama_adrcnt;
        sramb_adrcnt_d <= sramb_adrcnt;
                    sram_rdcnt     <= sram_rdcnt_p;
                    sram_rdcnt_d   <= sram_rdcnt;
                    sram_rdcnt_d1  <= sram_rdcnt_d;
                    sram_wr_done   <= sram_wr_done_p;
                    sram_rd_done   <= sram_wr_done_p;
      end
    end

    // Synchronize the asynchoronous key button input
    // Synchronize input data enable with 50mhz clock
    always @ (posedge clk or posedge rst) begin
```

```
        if(rst) begin
                push_button_d  <= 1'b0;
                new_data_en_d  <= 1'b0;
                new_data_en_d1 <= 1'b0;
                sram_wr_en_d   <= 1'b0;
        end
   else begin
                push_button_d  <= push_button;
                new_data_en_d  <= new_data_en;
                new_data_en_d1 <= new_data_en_d;
                sram_wr_en_d   <= sram_wr_en;
        end
end

//      Create a pulse signal for fsm state transition in sync with 50MHz clock
always @ (posedge clk or posedge rst) begin
        if(rst) begin
                fsm_trans_ctrl <= 1'b0;
        end
        else if(~push_button & push_button_d) begin
                fsm_trans_ctrl <= 1'b1;
        end
        else begin
                fsm_trans_ctrl <= 1'b0;
        end
end

// 50MHz Synchronized signal for SRAM control
always @ (posedge clk or posedge rst) begin
        if(rst) begin
                sram_wr_en <= 1'b0;
        end
        else if(new_data_en_d & ~new_data_en_d1) begin
                sram_wr_en <= 1'b1;
        end
        else begin
                sram_wr_en <= 1'b0;
        end
end

// Timing fixes
always @ (posedge clk or posedge rst) begin
        if(rst) begin
                smema_cs_n_d <= 1'b1;
```

```
            smema_we_n_d <= 1'b1;
            smema_oe_n_d <= 1'b1;
            smema_addr_d <= 3'd0;
            smema_data_d <= 16'd0;
            smemb_cs_n_d <= 1'b1;
            smemb_we_n_d <= 1'b1;
            smemb_oe_n_d <= 1'b1;
            smemb_addr_d <= 3'd0;
            smemb_data_d <= 16'd0;
        end
        else begin
            smema_cs_n_d <= smema_cs_n;
            smema_we_n_d <= smema_we_n;
            smema_oe_n_d <= smema_oe_n;
            smema_addr_d <= smema_addr;
            smema_data_d <= smema_data;

            smemb_cs_n_d <= smemb_cs_n;
            smemb_we_n_d <= smemb_we_n;
            smemb_oe_n_d <= smemb_oe_n;
            smemb_addr_d <= smemb_addr;
            smemb_data_d <= smemb_data;
        end
    end


    // SRAM A Interface
    assign MEMA_CS_n    = smema_cs_n_d;
    assign MEMA_WE_n    = smema_we_n_d;
    assign MEMA_OE_n    = smema_oe_n_d;
    assign MEMA_address = smema_addr_d;
    assign  MEMA_data        =  (current_state  ==  st_mac)  ?  16'bzzzzzzzzzzzzzzzz  :
    smema_data_d;

    // SRAM B Interface
    assign MEMB_CS_n    = smemb_cs_n_d;
    assign MEMB_WE_n    = smemb_we_n_d;
    assign MEMB_OE_n    = smemb_oe_n_d;
    assign MEMB_address = smemb_addr_d;
    assign  MEMB_data        =  (current_state  ==  st_mac)  ?  16'bzzzzzzzzzzzzzzzz  :
    smemb_data_d;

    // Pipeline Start
    always @ (posedge clk or posedge rst) begin
    if(rst) begin
      mem_rddata    <= 16'd0;
      mem_rdweight  <= 16'd0;
      pipeline_en   <= 1'b0;
    end
    else begin
      mem_rddata    <= mem_rddata_p;
      mem_rdweight  <= mem_rdweight_p;
      pipeline_en   <= pipeline_en_p;
    end
  end


  always @ (*) begin
    product_p   = 22'd0;
    exp_p       = 6'd0;
    mult_sign_p = 1'b0;
    // Check if one of the mantissa is 0, then product 0 adjust exponent
    if(~|mem_rddata || ~|mem_rdweight) begin
      product_p   = 22'd0;
      exp_p       = 6'd0;
      mult_sign_p = 1'b0;
    end
    else begin // A x B
      product_p   = ({1'b1,mem_rddata[9:0]} * {1'b1,mem_rdweight[9:0]});
      exp_p       = mem_rddata[14:10] + mem_rdweight[14:10] - 6'd15;
      mult_sign_p = mem_rddata[15] ^ mem_rdweight[15]; // Sign bit +ve(0), -ve(1)
    end
  end

  always @ (posedge clk or posedge rst) begin
    if(rst) begin
      mult_sign <= 1'b0;
      product   <= 22'd0;
      exp       <= 6'd0;
      mult_en   <= 1'b0;
    end
    else begin
      mult_sign <= mult_sign_p;
      product   <= product_p;
      exp       <= exp_p;
      mult_en   <= pipeline_en;
    end
  end
```

```
// Pipeline 2 : Normalize and Roundoff
// Check for exponent overflow/underflow
reg overflow_error,fract_overflow;
reg [4:0] exp_p1;
reg hidden_bit;

always @ (*) begin
  exp_p1        = 5'd0;
  overflow_error = 1'b0;
  fract_overflow = 1'b0;
  if(exp > 5'd31) begin
    overflow_error = 1'b1;
  end
  else begin
    exp_p1 = exp[4:0];
  end
  // check for fraction overflow
  if(product[21]) begin
    fract_overflow = 1'b1;
  end
end

always @ (posedge clk or posedge rst) begin
  if(rst) begin
    mult_sign_d   <= 1'b0;
    exp_norm      <= 5'd0;
    product_norm  <= 10'd0;
    mult_align_en <= 1'b0;
    sum_en        <= 1'b0;
    hidden_bit    <= 1'b0;
                sum_out_en    <= 1'b0;
  end
  else begin
    mult_sign_d   <= mult_sign;
    product_norm  <= fract_overflow ? product[20:11] : product[19:10]; // Roundoff
    exp_norm      <= fract_overflow ? exp_p1 + 5'd1 : exp_p1;
    mult_align_en <= mult_en;
    sum_en        <= mult_align_en;
    hidden_bit    <= (|exp_p1 & 1'b1);
                sum_out_en    <= sum_en;
  end
end
```

```
assign fp_mult_out = {mult_sign_d,exp_norm,hidden_bit,product_norm};
assign fp_mult_result = {mult_sign_d,exp_norm,product_norm};

// Pipeline 3 : Addition
// compare the exponents
// Check if exponents are equal
assign exp_equal = (acc_exp == exp_norm) ? 1'b1 : 1'b0;

// Align the smaller fraction
always @ (*) begin

  acc_exp_p  = 5'd0;
  fp_a_p     = 11'd0;
  fp_b_p     = 11'd0;
  acc_sign_p = 1'b0;

  if(exp_equal) begin
    acc_exp_p = exp_norm;
    if(acc_sum > fp_mult_out[10:0]) begin
      fp_a_p     = acc_sum;
      fp_b_p     = fp_mult_out[10:0];
      acc_sign_p = acc_sign;
    end
    else begin
      fp_a_p     = fp_mult_out[10:0];
      fp_b_p     = acc_sum;
      acc_sign_p = mult_sign_d;
    end
  end
  else if(acc_exp > exp_norm) begin
    acc_exp_p  = acc_exp;
    fp_a_p     = acc_sum;
    fp_b_p     = fp_mult_out[10:0] >> (acc_exp - exp_norm);
    acc_sign_p = acc_sign;
  end
  else begin
    acc_exp_p  = exp_norm;
    fp_a_p     = fp_mult_out[10:0];
    fp_b_p     = acc_sum >> (exp_norm - acc_exp);
    acc_sign_p = mult_sign_d;
  end
end

// Add or subtract based on the sign bit
```

```
assign acc_sum_p = (acc_sign ^ mult_sign_d) ? (fp_a_p - fp_b_p) : (fp_a_p + fp_b_p);

integer i;
// Normalize the result
always @ (*) begin
  exp_pos     = 5'd1;
  exp_shift   = 5'd2;
        acc_sum_ovf = 1'b0;
  if(acc_sum_p == 12'd0) begin
    acc_sum_norm  = 12'd0;
    acc_exp_norm  = 5'd0;
    acc_sign_norm = 1'b0;
  end
  else begin
    if(acc_sum_p[11:10] == 2'b00) begin // Normalize
      for(i = 10; i > 0 ; i= i-1) begin : NORM_LOOP // fraction part
        if(!acc_sum_p[i-1]) begin
          exp_pos = exp_pos + 5'd1;
          exp_shift = exp_shift + 5'd1;
        end
        else begin
          exp_pos = exp_pos;
                              disable NORM_LOOP;
        end
      end
    end
    else begin
              if(acc_sum_p[11:10] == 2'b01) begin // no Overflow
                      exp_shift = 5'd1;
                      exp_pos   = 5'd0;
                  end
                  else      begin      //if(acc_sum_p[11:10]      ==      2'b10)      ||
(acc_sum_p[11:10] == 2'b11) // overflow
                      exp_shift   = 5'd0;
                      exp_pos     = 5'd1;
                      acc_sum_ovf = 1'b1;
                  end
    end
    acc_sum_norm  = acc_sum_p << exp_shift;
    acc_sign_norm = acc_sign_p;
    acc_exp_norm  = acc_sum_ovf ? (acc_exp_p + exp_pos) : (acc_exp_p - exp_pos) ;
  end
end
```

```
always @ (posedge clk or posedge rst) begin
  if(rst) begin
    acc_sign  <= 1'b0;
    acc_exp   <= 5'd0;
    acc_sum   <= 11'd0;
  end
        else if (fpmac_out_valid) begin // Reset the accumulated sum value after one
NN layer calculation
    acc_sign  <= 1'b0;
    acc_exp   <= 5'd0;
    acc_sum   <= 11'd0;
  end
  else begin
    if(sum_en) begin
      acc_sign <= acc_sign_norm;
      acc_exp  <= acc_exp_norm;
      acc_sum  <= acc_sum_norm[11:1];
    end
  end
end

always @ (posedge clk or posedge rst) begin
  if(rst) begin
          scnt <= 3'd0;
        end
        else begin
                scnt <= sum_out_en ? scnt + 3'd1 : 3'd0;
        end
end

assign fpmac_out_valid = (scnt == 3'd7) ? 1'b1 : 1'b0;
assign sum_out = {acc_sign,acc_exp,acc_sum[9:0]};

// MAC output for Seven segment display
// Latch the current value till next MAC output
always @ (posedge clk or posedge rst) begin
        if(rst) begin
                fpmac_out <= 16'd0;
        end
        else begin
                fpmac_out <= fpmac_out_valid ? sum_out : fpmac_out;
        end
end
```

```
endmodule
```

- **LCD display**

```
// Group 13
// Arduino code for LCD display

// Include liquid crystal I2C library

#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x27,16,2); // Default address 0x27

void setup () {

  lcd.init(); // LCD initialization
  lcd.setBacklight(1);
  lcd.setContrast(128); // Set Contrast for better display
  lcd.clear();
  Serial.begin(9600);

  pinMode(3,INPUT); // set the direction as Input for digital pin 3
  pinMode(2,INPUT); // set the direction as Input for digital pin 2
}

void loop() {
  int pin3 = digitalRead(3); // Read the value of input pin 3, MSB bit of the state
  int pin2 = digitalRead(2); // Read the value of input pin 3, LSB bit of the state

  if((pin3 == 0) && (pin2 == 0)) { // State 0 : RESET state
    lcd.clear();
    lcd.setCursor(1,0);
    lcd.print("Group-13"); // First row

    lcd.setCursor(2,1);
    lcd.print("Reset");

    delay(1000);
  }

  else if((pin3 == 0) && (pin2 == 1)) { // State 1 : SRAM A Input state
    lcd.clear();
    lcd.setCursor(1,0);
    lcd.print("Group-13"); // First row
```

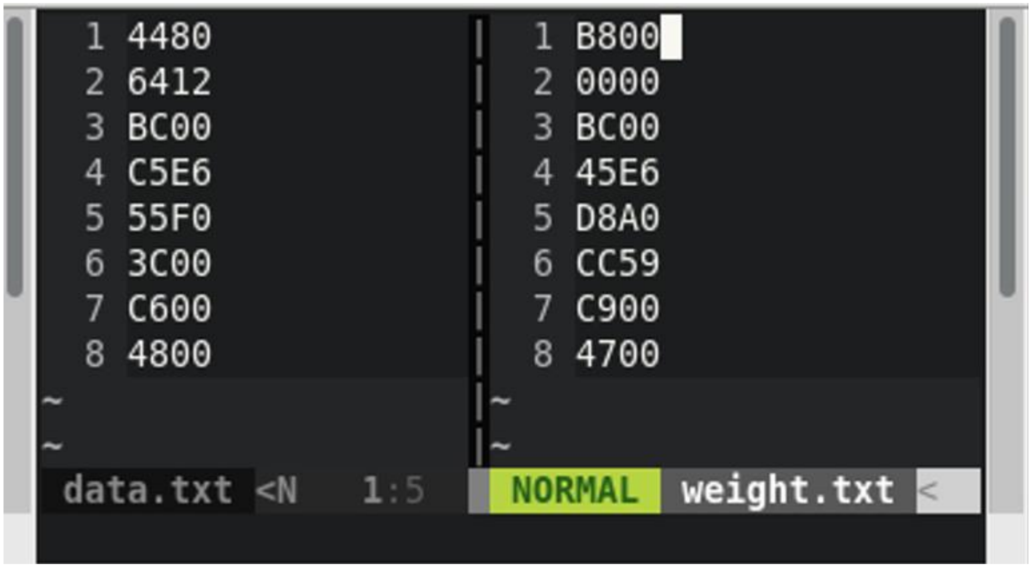
```
      lcd.setCursor(2,1);
      lcd.print("SRAM A Input");

      delay(1000);
    }

    else if((pin3 == 1) && (pin2 == 0)) { // State 2 : SRAM B Input state
      lcd.clear();
      lcd.setCursor(1,0);
      lcd.print("Group-13"); // First row

      lcd.setCursor(2,1);
      lcd.print("SRAM B Input");

      delay(1000);
    }
    else { //if((pin3 == 1) && (pin2 == 1)) { // State 3 : FP MAC Calc Input state
      lcd.clear();
      lcd.setCursor(1,0);
      lcd.print("Group-13"); // First row

      lcd.setCursor(2,1);
      lcd.print("FP MAC Calc");

      delay(1000);
    }
  }
```

ANNEXURE (Test Bench)

```
// Group 13
// EE271: MAC testbench
`timescale 1ns/10ps

module tb_fpmac();

reg clk, rst, in_ctrl,data_en;
integer read_datfile;
integer read_weightfile;
reg [15:0] in_data,in_weight;
wire [15:0] sum_out;
reg [1:0] state;

reg clk_1mhz;
fp_mac_top u_fp_mac_top(clk,rst,in_ctrl,data_en,in_data,state,sum_out);

always #10 clk = ~clk; // Clock period 20ns , f = 50 MHz
always #500 clk_1mhz = ~clk_1mhz; // Slower clock to emulate the keypad clock

initial begin
  clk_1mhz = 0;
end

initial begin
  read_datfile = $fopen("data.txt","r");
  read_weightfile = $fopen("weight.txt","r");
  clk = 0;
  rst = 1; // Asynchronous Active high reset
  in_ctrl = 0;
  data_en = 0;
  #15;
  rst = 0; // Reset release
  #150;
  @(posedge clk_1mhz)
  in_ctrl = 1;
  @(posedge clk_1mhz)

  in_ctrl = 0;
  #10;
  for (int i = 0 ; i < 8 ; i++) begin
    while(!$feof(read_datfile)) begin
      @(posedge clk_1mhz)
      data_en = 1;
      $fscanf(read_datfile,"%h\n",in_data);
      @(posedge clk_1mhz)
      data_en = 0;
    end
  end
  #10;
  @(posedge clk_1mhz)
  in_ctrl = 1;
  @(posedge clk_1mhz)
  in_ctrl = 0;
  #10;
  for (int i = 0 ; i < 8 ; i++) begin
    while(!$feof(read_weightfile)) begin
      @(posedge clk_1mhz)
      data_en = 1;
      $fscanf(read_weightfile,"%h\n",in_data);
      @(posedge clk_1mhz)
      data_en = 0;
    end
  end
  data_en = 0;
  #10;
  @(posedge clk_1mhz)
  in_ctrl = 1;
  @(posedge clk_1mhz)
  in_ctrl = 0;
  @(posedge clk_1mhz)
  in_ctrl = 1;
  @(posedge clk_1mhz)
  in_ctrl = 0;
  #500 $finish;
end
always @ (*) begin
  @(posedge clk)
  if(u_fp_mac_top.u_fsm.mult_en || u_fp_mac_top.u_fsm.sum_out_en) begin
    $display("Clock: %b, Reset : %b, SRAM A data: %h, SRAM B weight: %h,\
        ACC Sum: %h",clk,rst,u_fp_mac_top.u_fsm.mem_rddata,
        u_fp_mac_top.u_fsm.mem_rdweight,u_fp_mac_top.u_fsm.sum_out);
  end
end
```

```
initial begin
  $dumpfile("MAC.vcd");
  $dumpvars();
end
endmodule
```

Input files: data.txt and weight.txt



```
4480        B800
6412        0000
BC00        BC00
C5E6        45E6
55F0        D8A0
3C00        CC59
C600        C900
4800        4700
```