ECE495 Final Project (Fall 2015)

Floating Point CORDIC Based Power Operation

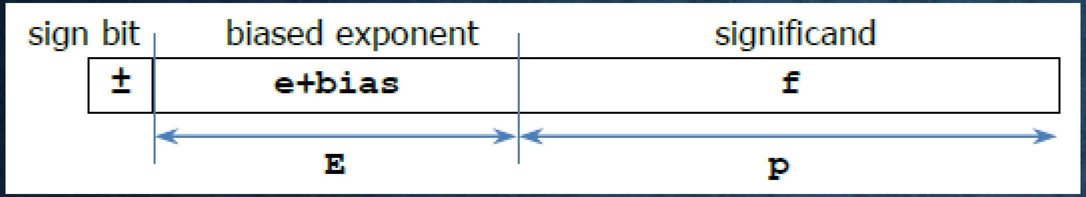
Kazumi Malhan

Padmaja AVL

OUTLINE

- Floating Point Format
- Extended Hyperbolic CORDIC
- Power Operation
- Interface with FIFO
- Implementation of System
- SD Card
- Test Result
- Timing Issue
- Demonstration
- Q & A

SUPPORTED FLOATING POINT FORMAT



	32 bit (Single)	64 bit (Double)
Ordinary Number	-	-
Min	2^{-126}	2^{-1022}
Max	$(2-2^{-23})\times2^{127}$	$(2-2^{-52})\times2^{1023}$
Exponent bits E	8	11
Range of e	[-126, 127]	[-1022, 1023]
Bias	127	1023
Dynamic Range (dB)	759 dB	6153 dB
Significand range	$[1, 2-2^{-23}]$	$[1, 2-2^{-52}]$
Significand bits (p)	23	52

Some of the components supports 16 bit and 24 bit FP format, but not officially supported.

Zero, infinite, Not a Number is supported and follows similar to IEEE-754 Standard.

Note: Deformalized numbers are not supported.

EXPANDED HYPERBOLIC CORDIC

For i < 0

$$i \le 0: \begin{cases} x_{i+1} = x_i + \delta_i y_i (1 - 2^{i-2}) \\ y_{i+1} = y_i + \delta_i x_i (1 - 2^{i-2}) \\ z_{i+1} = z_i - \delta_i \theta_i, \theta_i = Tanh^{-1}(1 - 2^{i-2}) \end{cases}$$

M = 5 is chosen, -2 operation was done inside FSM counter. (counted from -7 to -2)

For i > 0

$$i > 0: \begin{cases} x_{i+1} = x_i + \delta_i y_i 2^{-i} \\ y_{i+1} = y_i + \delta_i x_i 2^{-i} \\ z_{i+1} = z_i - \delta_i \theta_i, \theta_i = Tanh^{-1}(2^{-i}) \end{cases}$$

N = 16. Inside FSM counter, included the code to generate indication when i = 4,13. FSM controlled the enable single to the counter to repeat iteration. Register is used to confirm two iteration is occurred.

i = 4, and 13 were repeated.

Delta

$$Rotation: \delta_i = -1 \; if \; z_i < 0; \; +1, otherwise$$
$$Vectoring: \delta_i = -1 \; if \; x_i y_i \ge 0; \; +1, otherwise$$

For vectoring mode, checked if x(i) and y(i) have same bit. If it is same, => positive.

General output

$$Rotation: \begin{cases} x_n = A_n(x_0 coshz_0 + y_0 sinhz_0) \\ y_n = A_n(x_0 coshz_0 + y_0 sinhz_0) \\ z_n = 0 \end{cases}$$

$$Vectoring: \begin{cases} x_n = A_n\sqrt{x_0^2 - y_0^2} \\ y_n = 0 \\ z_n = z_0 + tanh^{-1}(y_0/x_0) \end{cases}$$

$$A_n = \left(\prod_{i=-M}^{0}\sqrt{1-(1-2^{i-2})^2}\right)\prod_{i=1}^{N}\sqrt{1-2^{-2i}}$$

- ❑ An = 5.0382×10^{-4}
  M = 5, N = 16
- ❑ Vector mode:
  ln(x)/2 = $\tanh^{-1}$(x-1/x+1)
- ❑ Rotation mode:
  e^x = cosh(x) + sinh(x)

How to calculate x^y

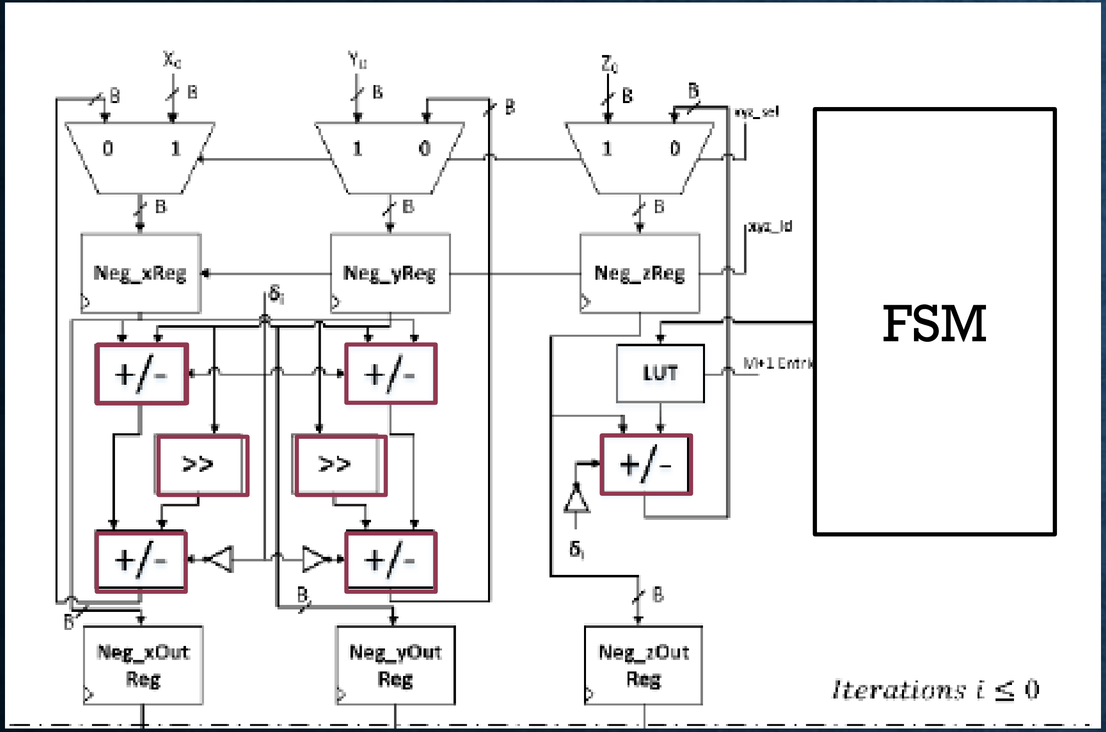
1) Using vectoring mode, provide $x_0 = x + 1$, $y_0 = x – 1$, $z_0 = 0$.
2) You get Zn = ln(x)/2
3) Multiply ln(x)/2 and 2. (Performed by bit shifting)
4) Multiply ln(x) and y.
5) Using rotation mode, provide $x_0 = y_0 = 1/An$, $z_0 = \ln(x)*y$.
6) You get Xn = $e^{y\ln x} = x^y$

Parameter to CORDIC

N = total number of bits
EXP = exponent bits
FR = fractional bits

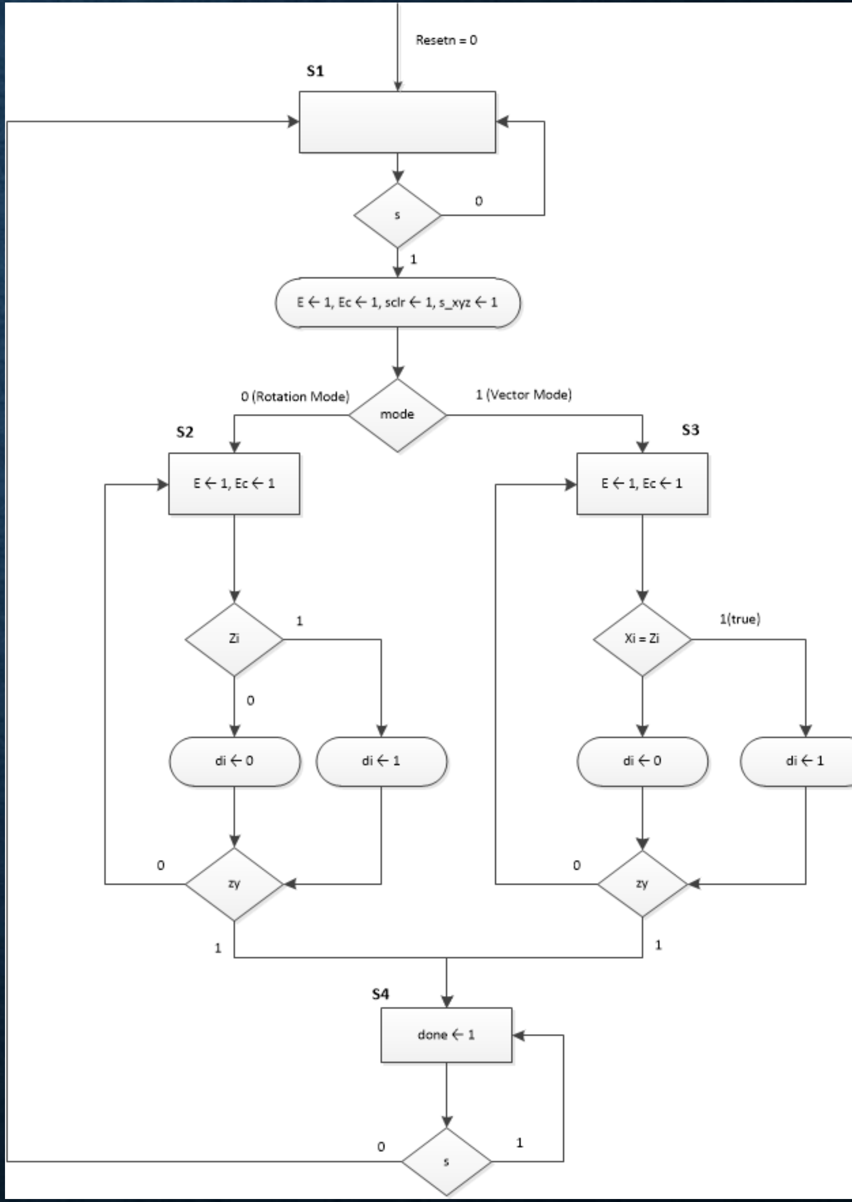
CORDIC is coded as parametrized to support any FP format. Just need to modify LUTs and some constant definitions

EXPANDED HYPERBOLIC CORDIC IMPLEMENTATION

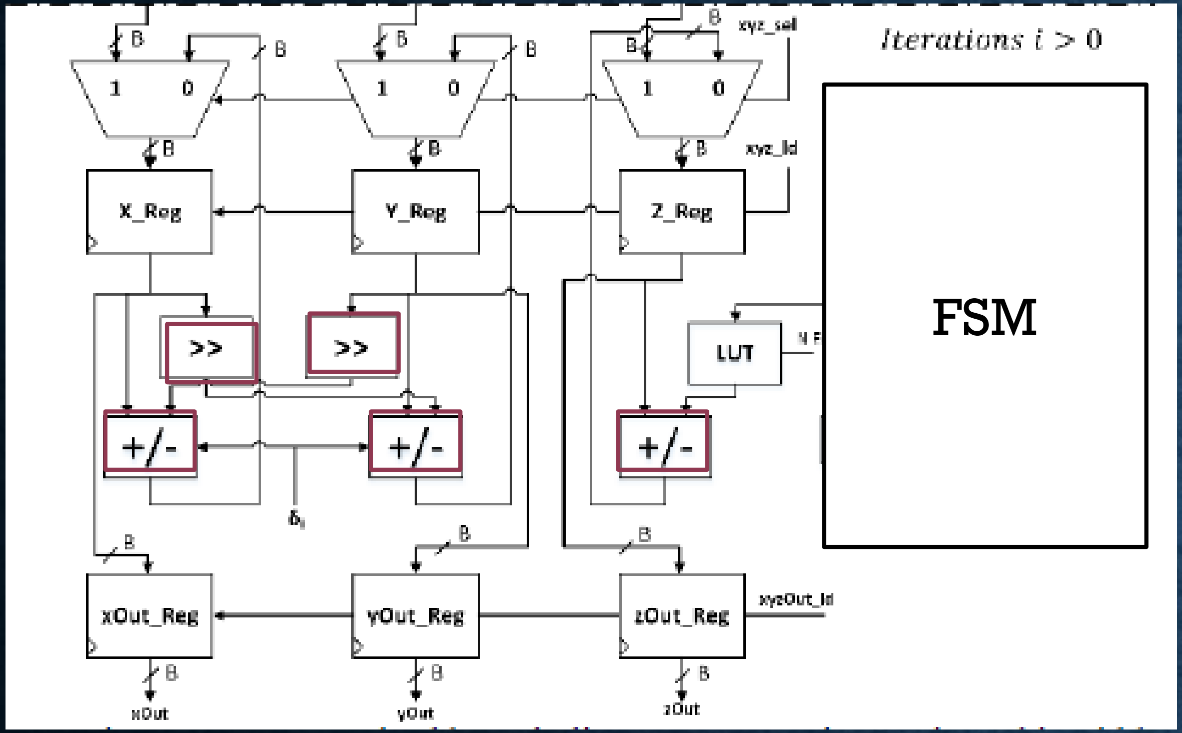


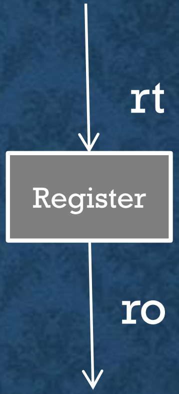
LUT uses "if (N = ?) generate" statement to output appropriate FP formatted numbers

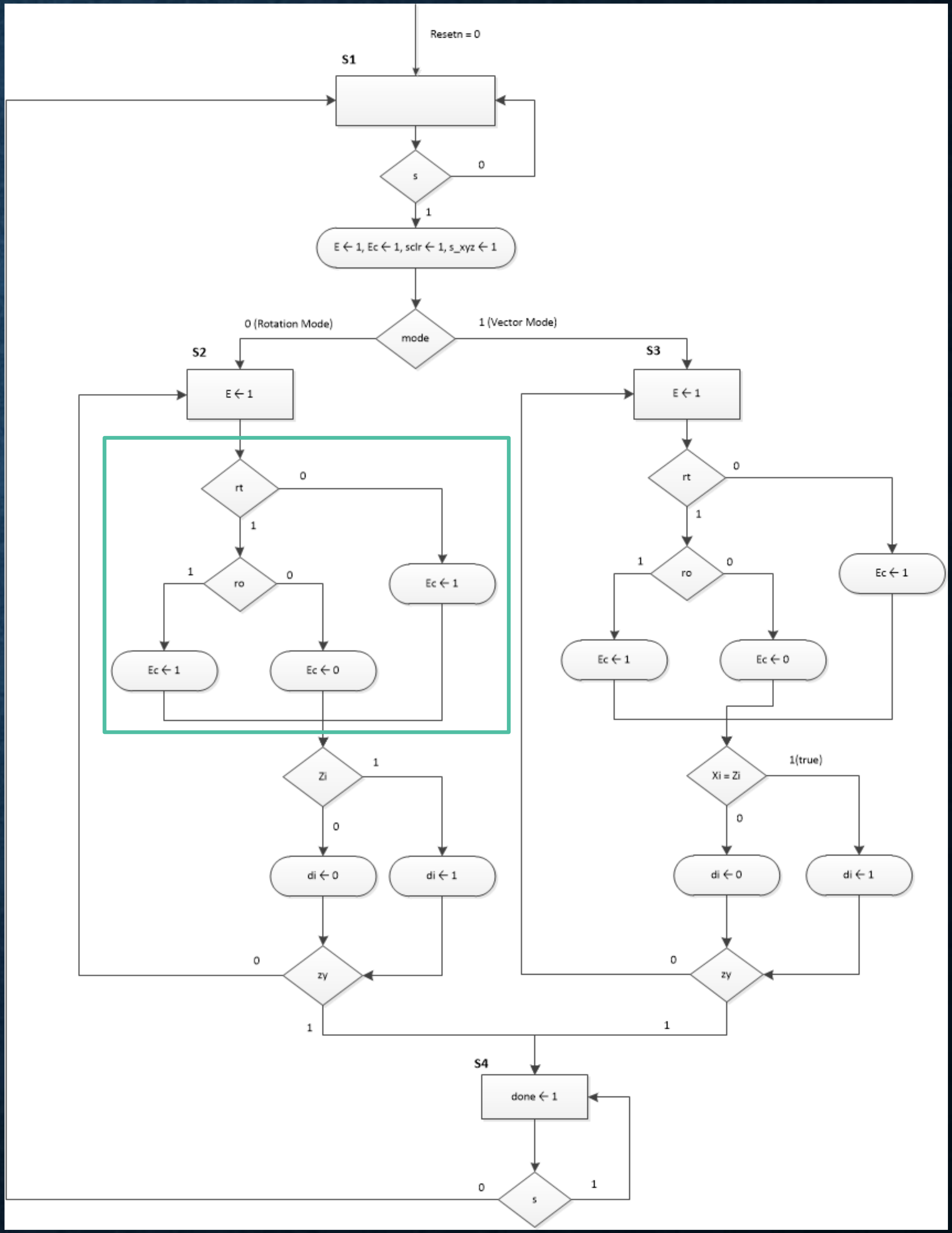
For I < 0 iteration



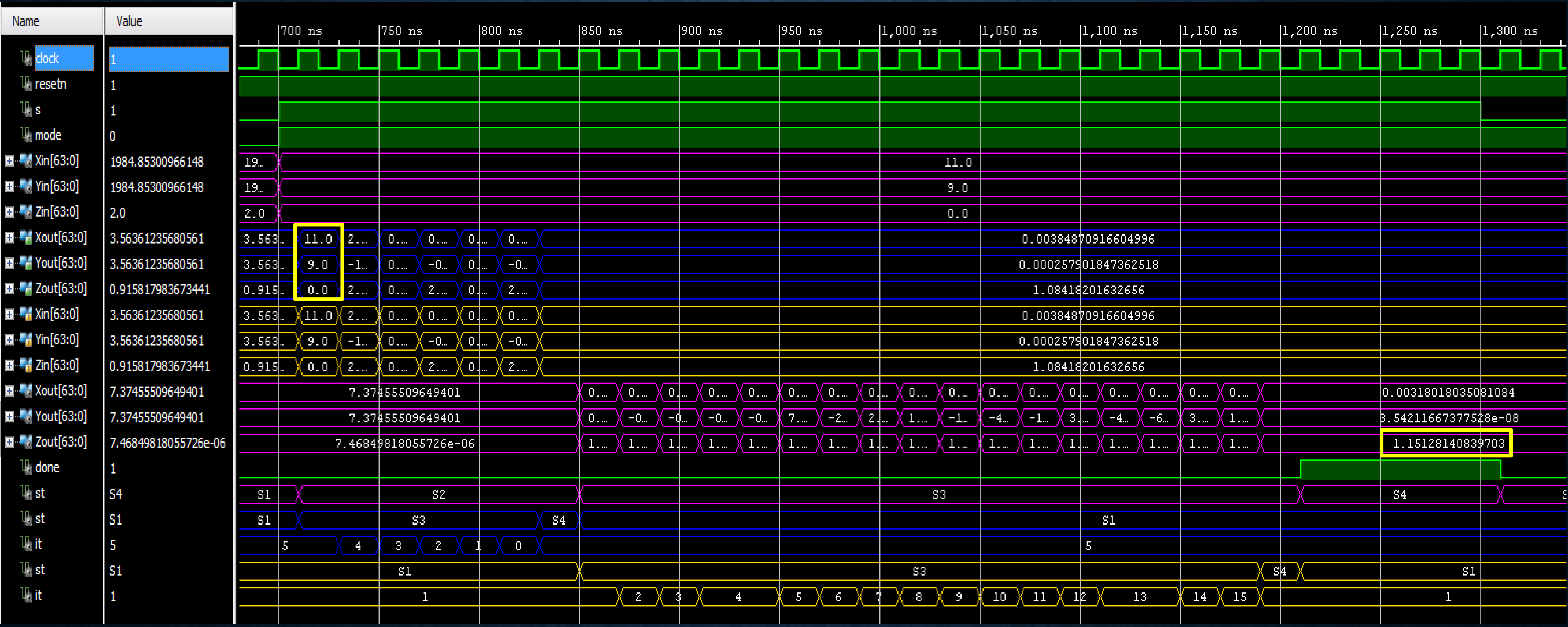
For I > 0 iteration







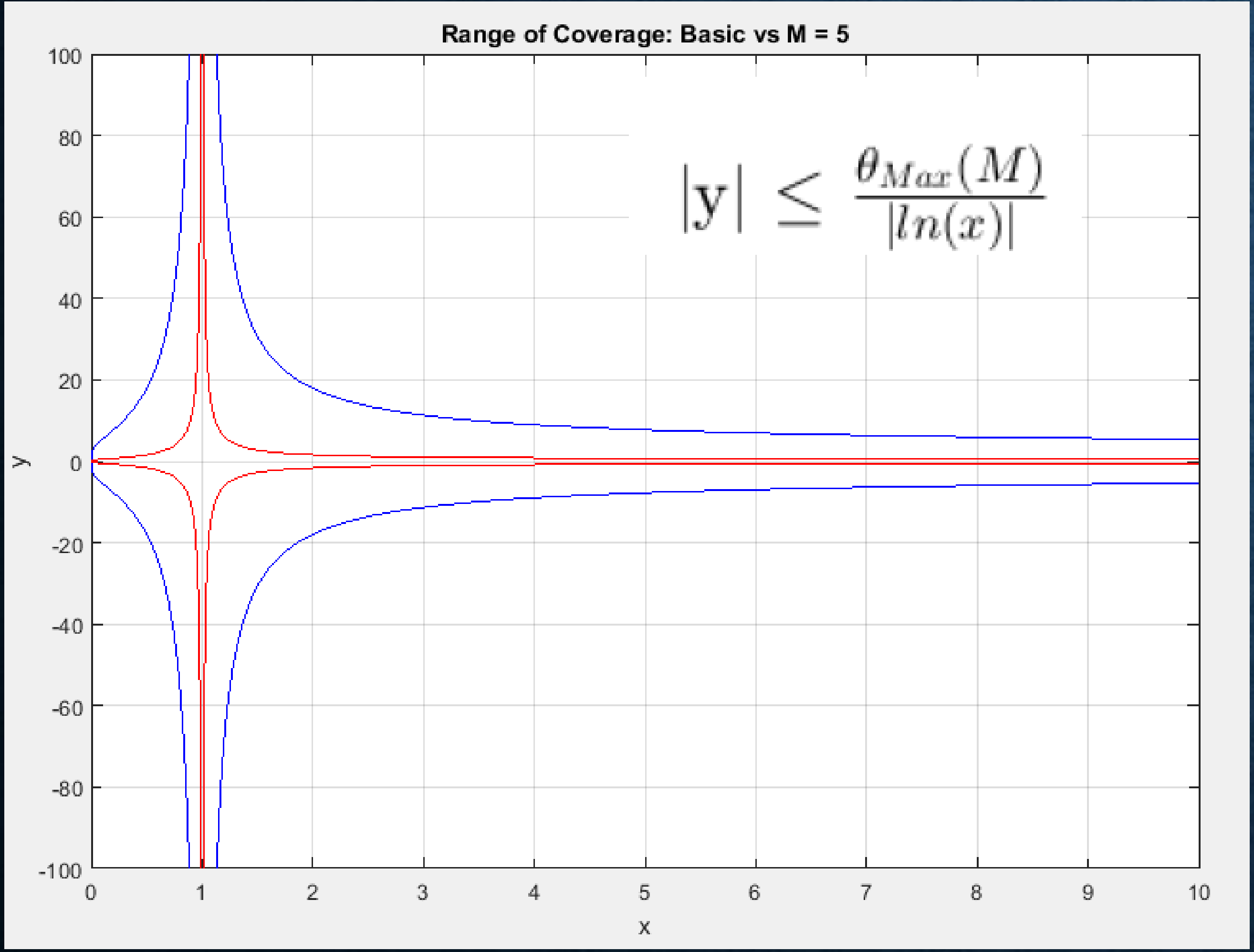
SIMULATION



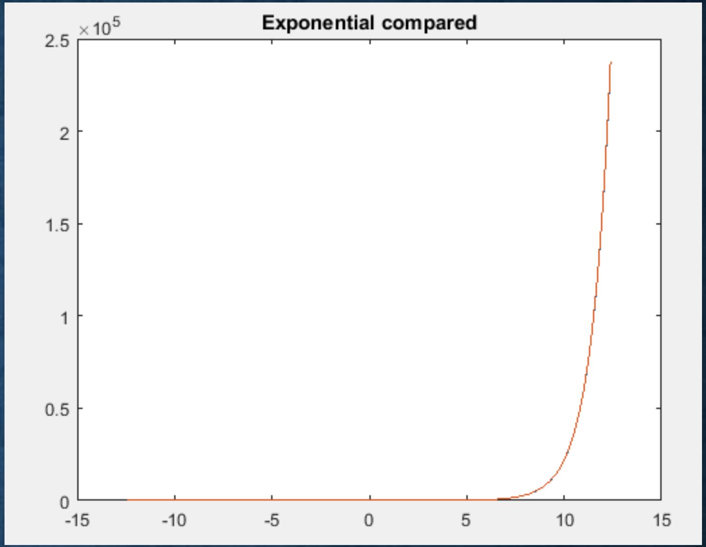
Name	Value
clock	1
resetn	1
s	1
mode	0
Xin[63:0]	1984.85300966148
Yin[63:0]	1984.85300966148
Zin[63:0]	2.0
Xout[63:0]	3.56361235680561
Yout[63:0]	3.56361235680561
Zout[63:0]	0.915817983673441
Xin[63:0]	3.56361235680561
Yin[63:0]	3.56361235680561
Zin[63:0]	0.915817983673441
Xout[63:0]	7.37455509649401
Yout[63:0]	7.37455509649401
Zout[63:0]	7.46849818055726e-06
done	1
st	S4
st	S1
it	5
st	S1
it	1

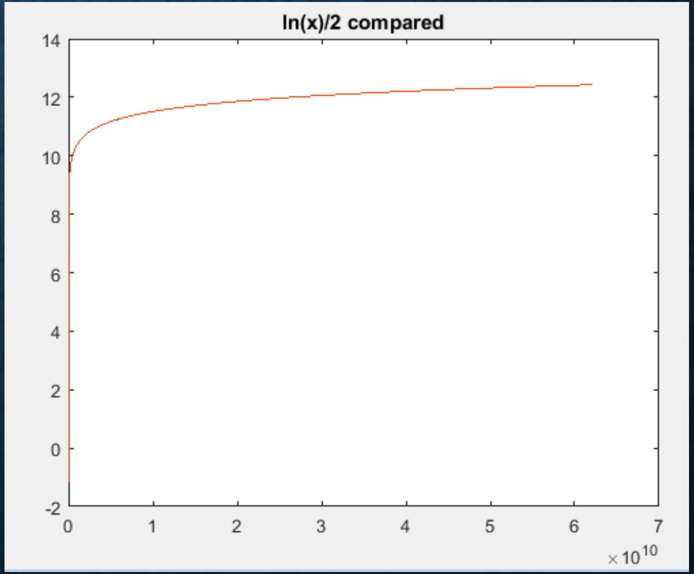
Calculator Result: ln(10)/2 = 1.15129

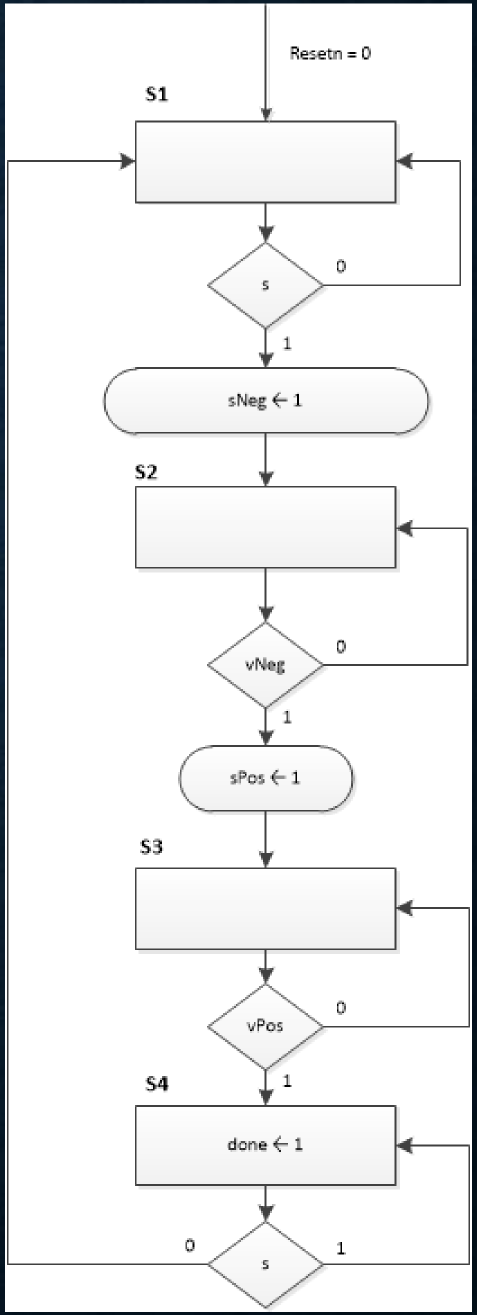
RANGE OF COVERAGE



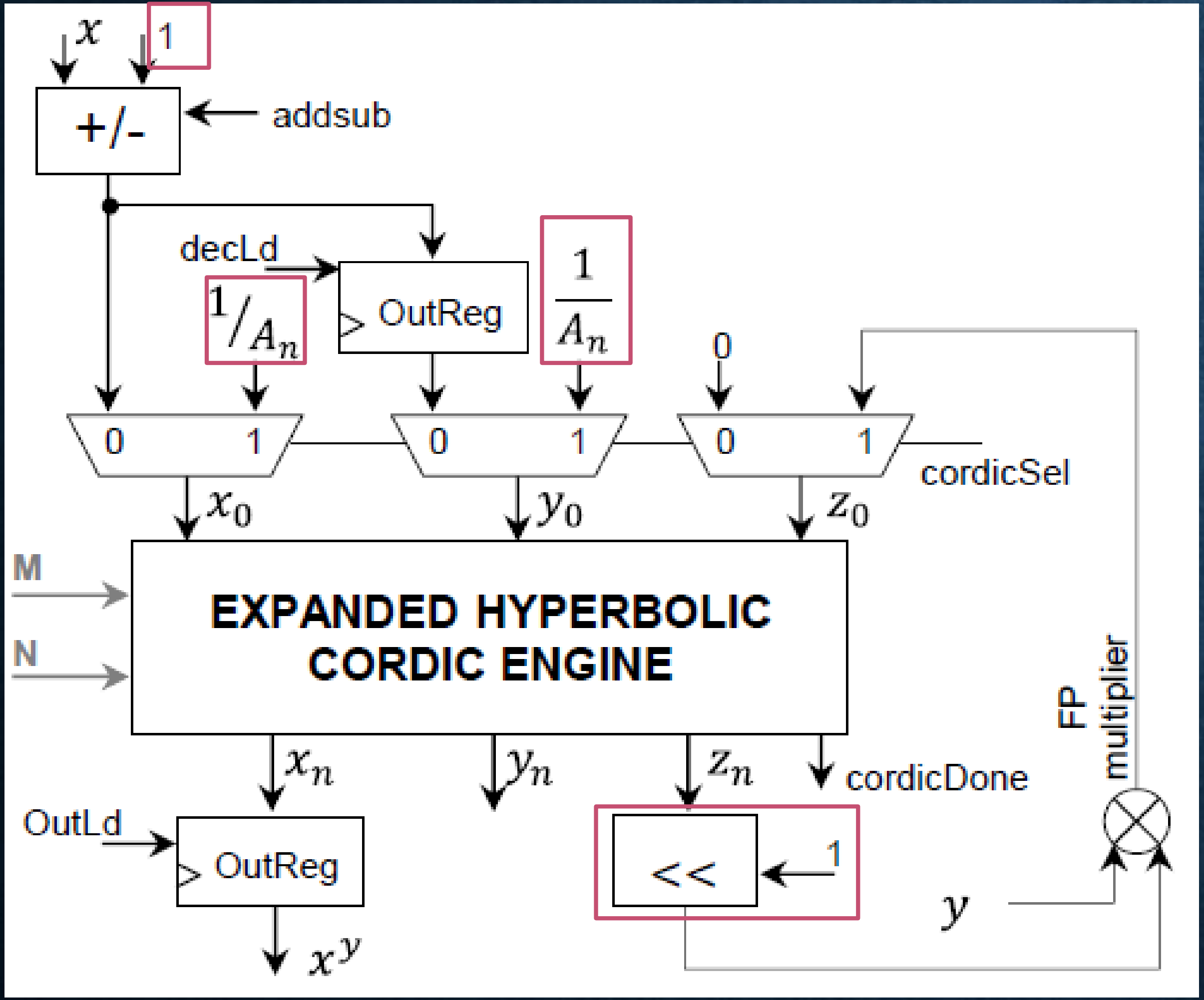
$$|y| \leq \frac{\theta_{Max}(M)}{|ln(x)|}$$

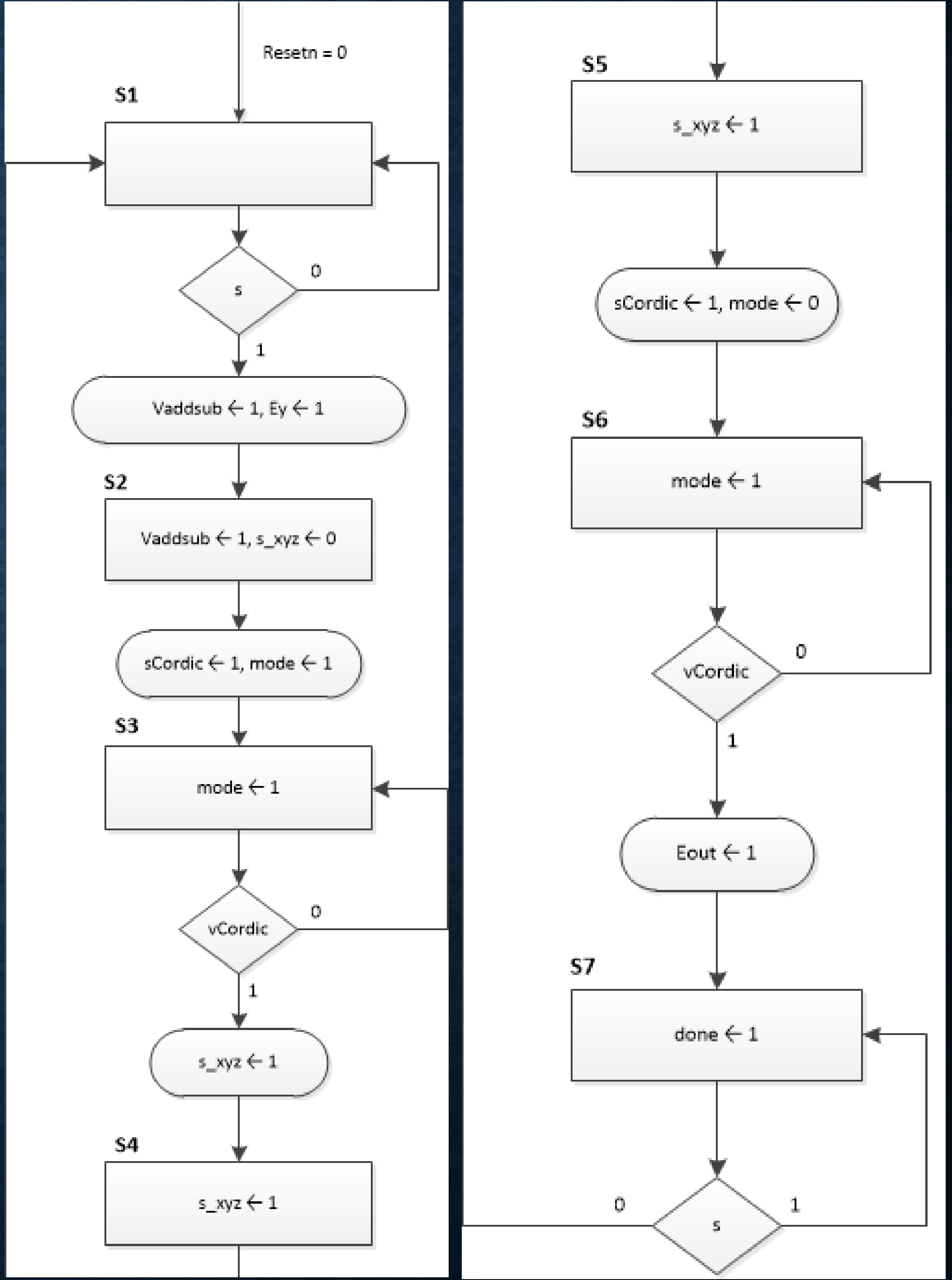




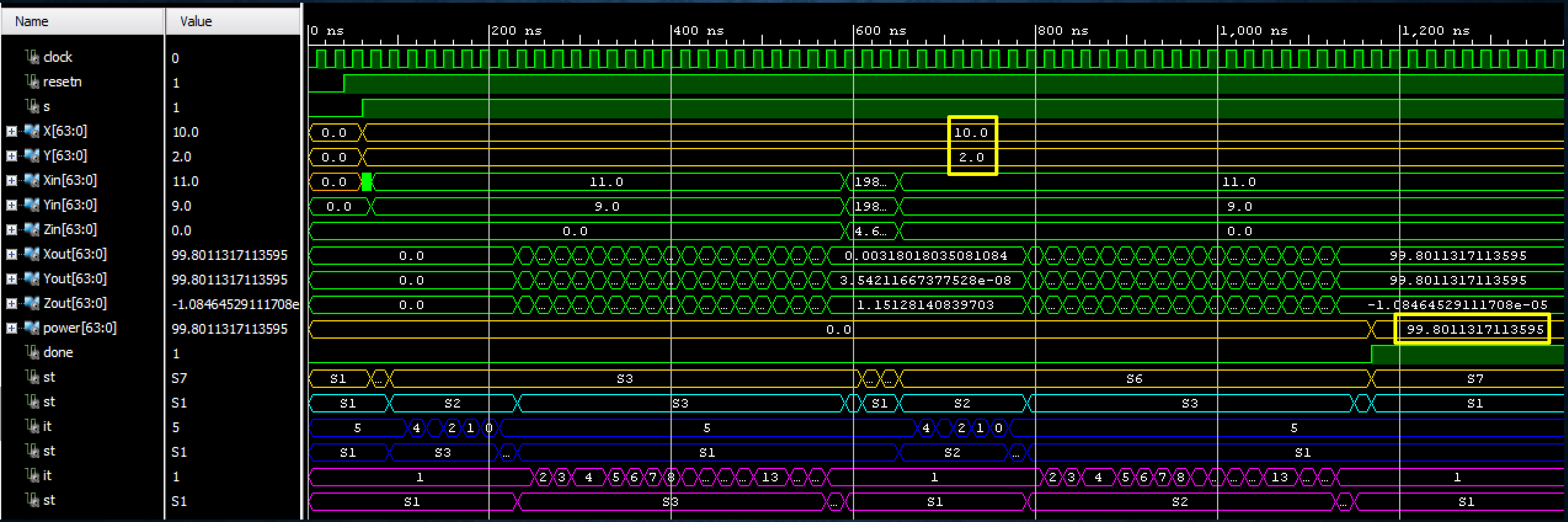


CORDIC BASED SYSTEM



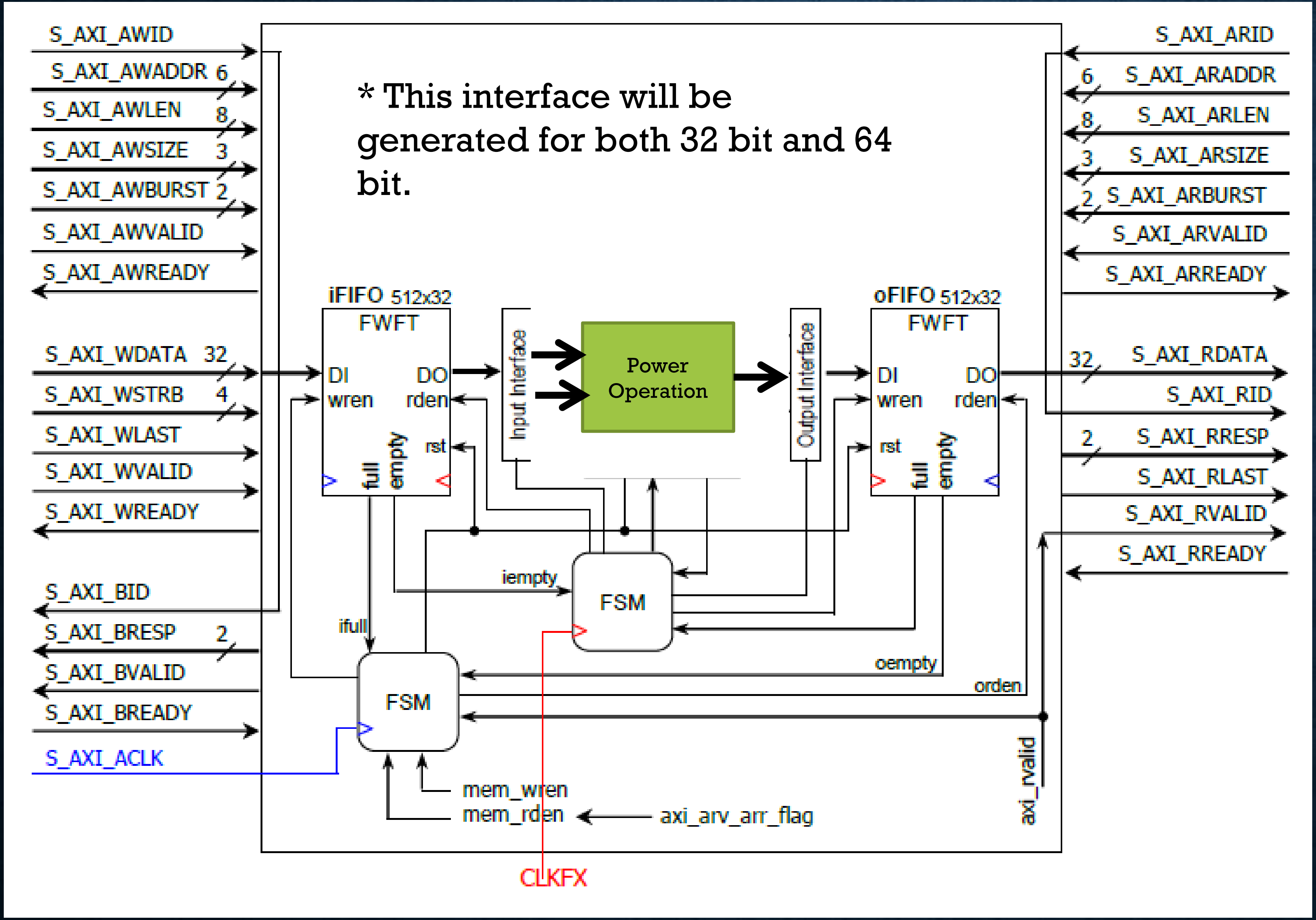


SIMULATION



Name	Value
clock	0
resetn	1
s	1
X[63:0]	10.0
Y[63:0]	2.0
Xin[63:0]	11.0
Yin[63:0]	9.0
Zin[63:0]	0.0
Xout[63:0]	99.8011317113595
Yout[63:0]	99.8011317113595
Zout[63:0]	-1.08464529111708e
power[63:0]	99.8011317113595
done	1
st	S7
st	S1
it	5
st	S1
it	1
st	S1

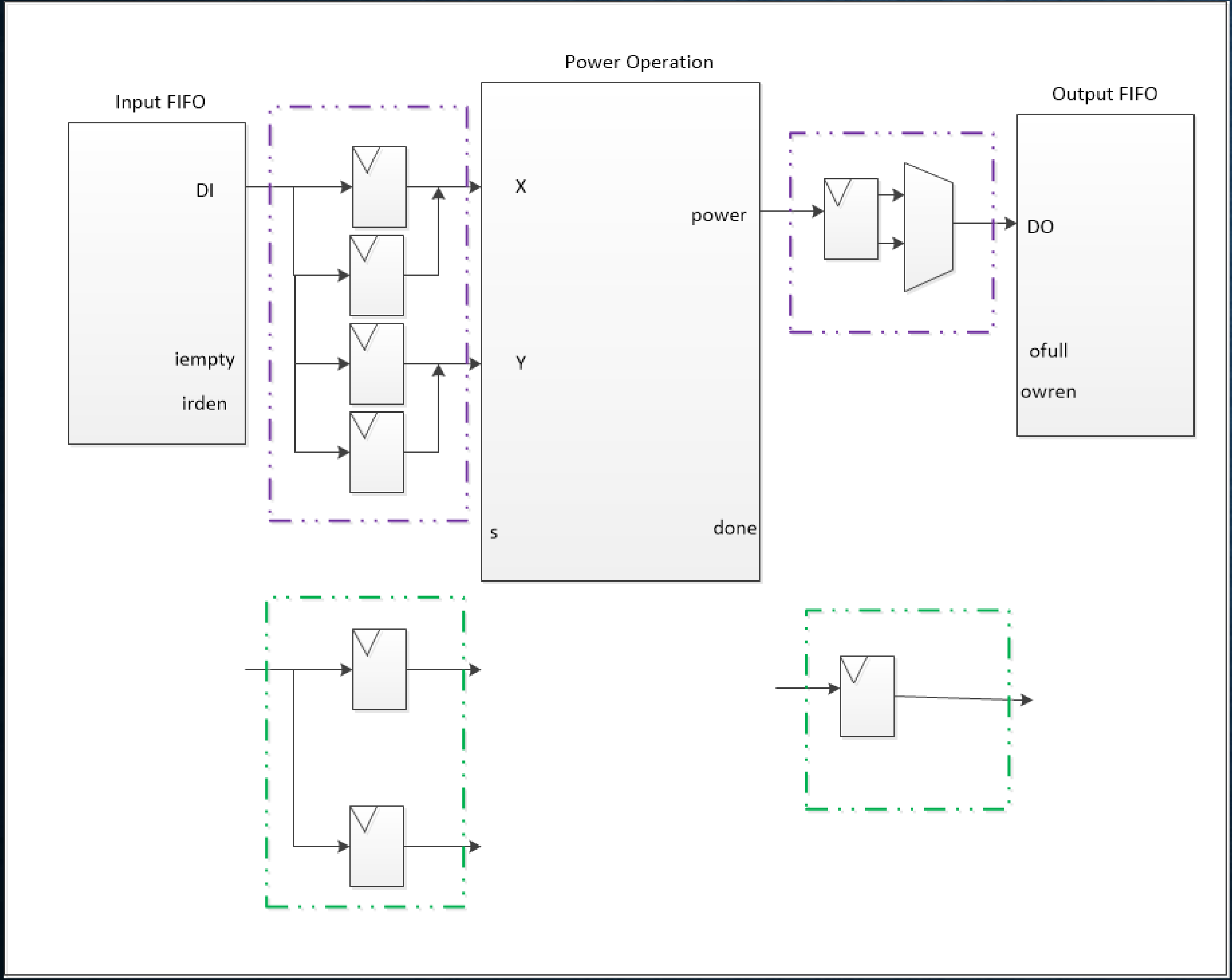
AXI4-FULL INTERFACE



* This interface will be generated for both 32 bit and 64 bit.

AXI4-Full interface was chosen as large amount of input (x,y) goes to input FIFO from SD card.

INTERFACE WITH FIFO



Depending on N = 64 bit or 32 bit, Separate circuit is created in-between FIFOs and Power block.

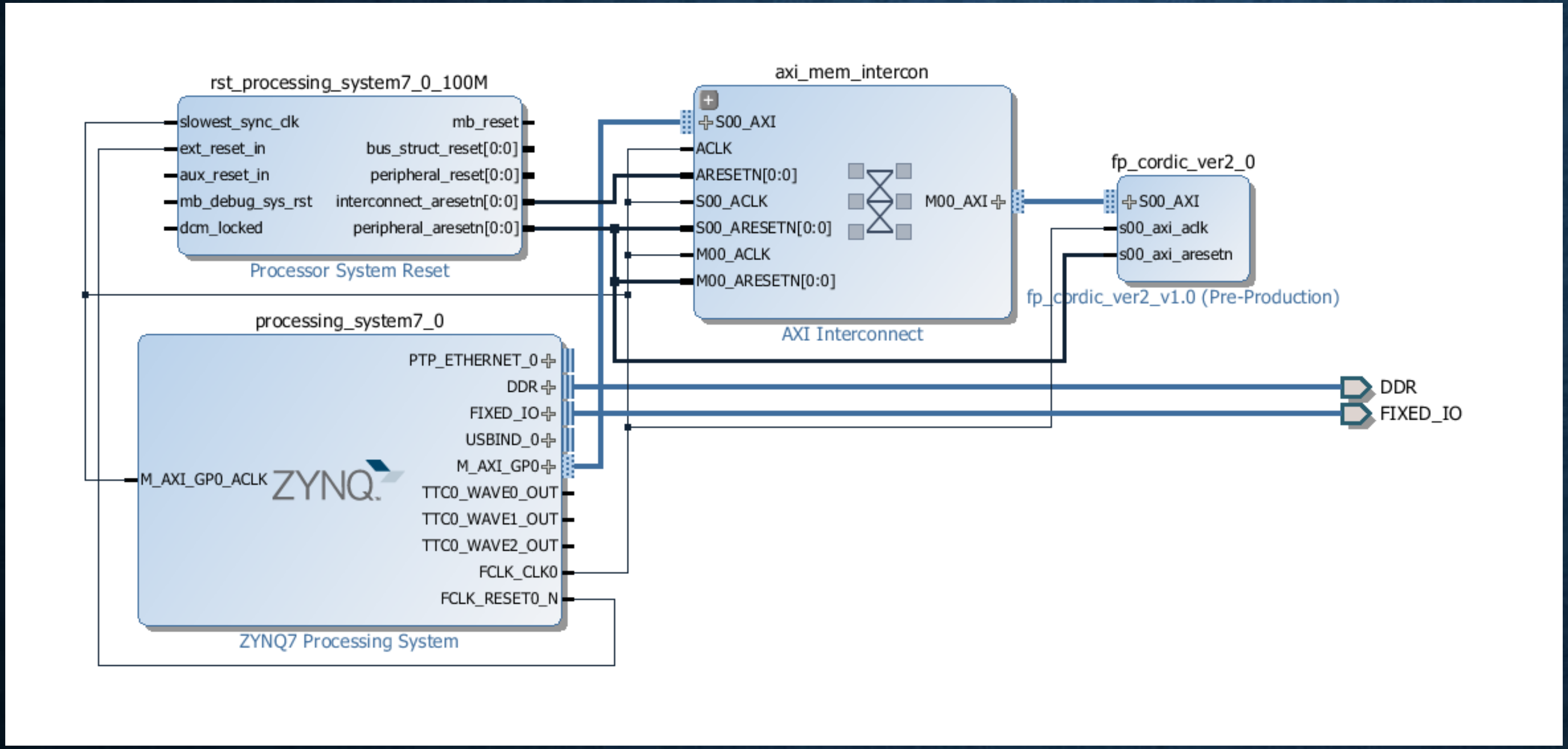
64 bit = Purple
32 bit = Green

Separate FSM controls the input interface and output interface.

Also, different FSM for 32 bit and 64 bit.

Every input has register to ensure the correct input because power block takes reads “y” after first cordic is done.

IMPLEMENTATION



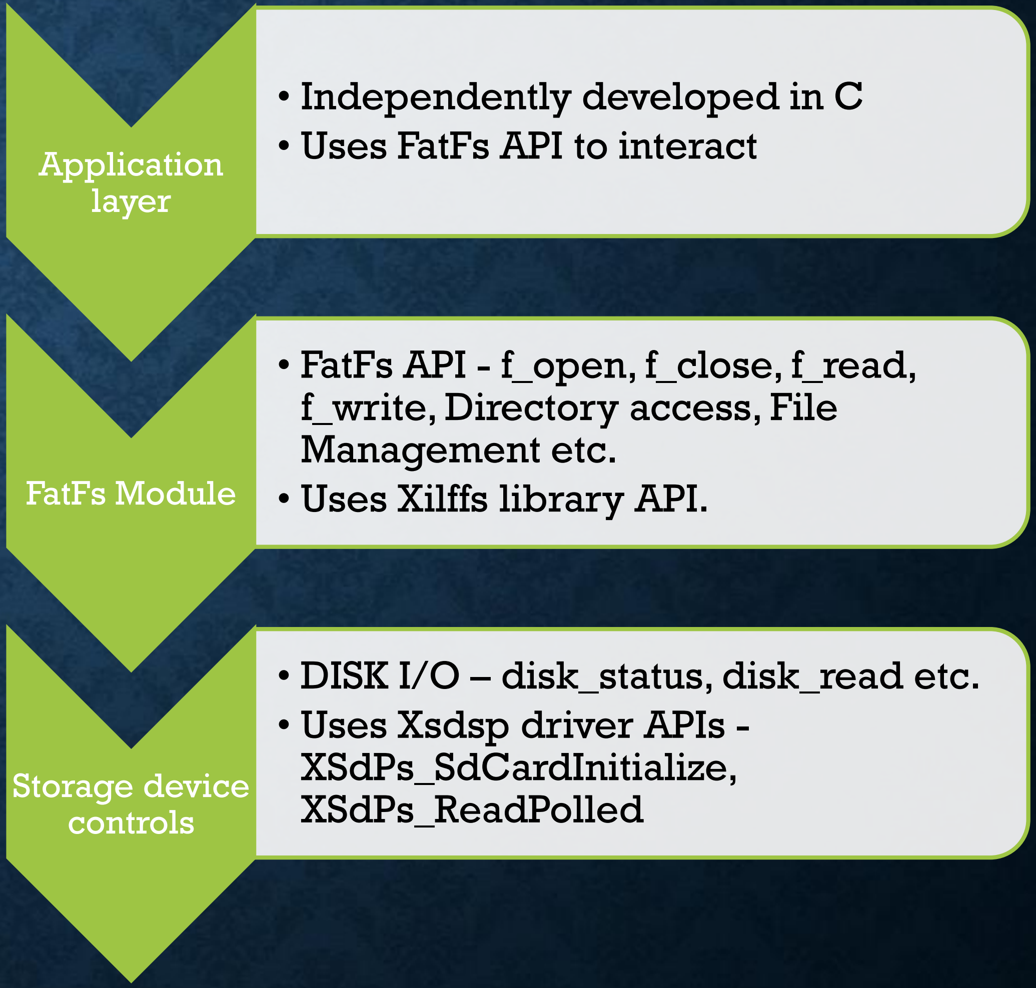
<u>Resource Usage</u>

64 Bit:
LUT => 9464 (18%)
FF => 895

32 Bit:
LUT => 4378 (8%)
FF => 502

SD CARD INTERFACE

- Uses the Xsdps libraries at driver level.
- This driver is used to initialize read from and write to the SD card.
- Data transfer: The SD card is put in transfer state to read from or write to it and works in polled mode using ADMA2. The default block size is 512 bytes.
- File system: The xilffs library is used to read/write files to SD.
- Application file and functions are completely developed independently and it supports read from a file in SD card repeatedly until end of the file and after manipulating the data, write back into SD card file in another format.



Application layer

- Independently developed in C
- Uses FatFs API to interact

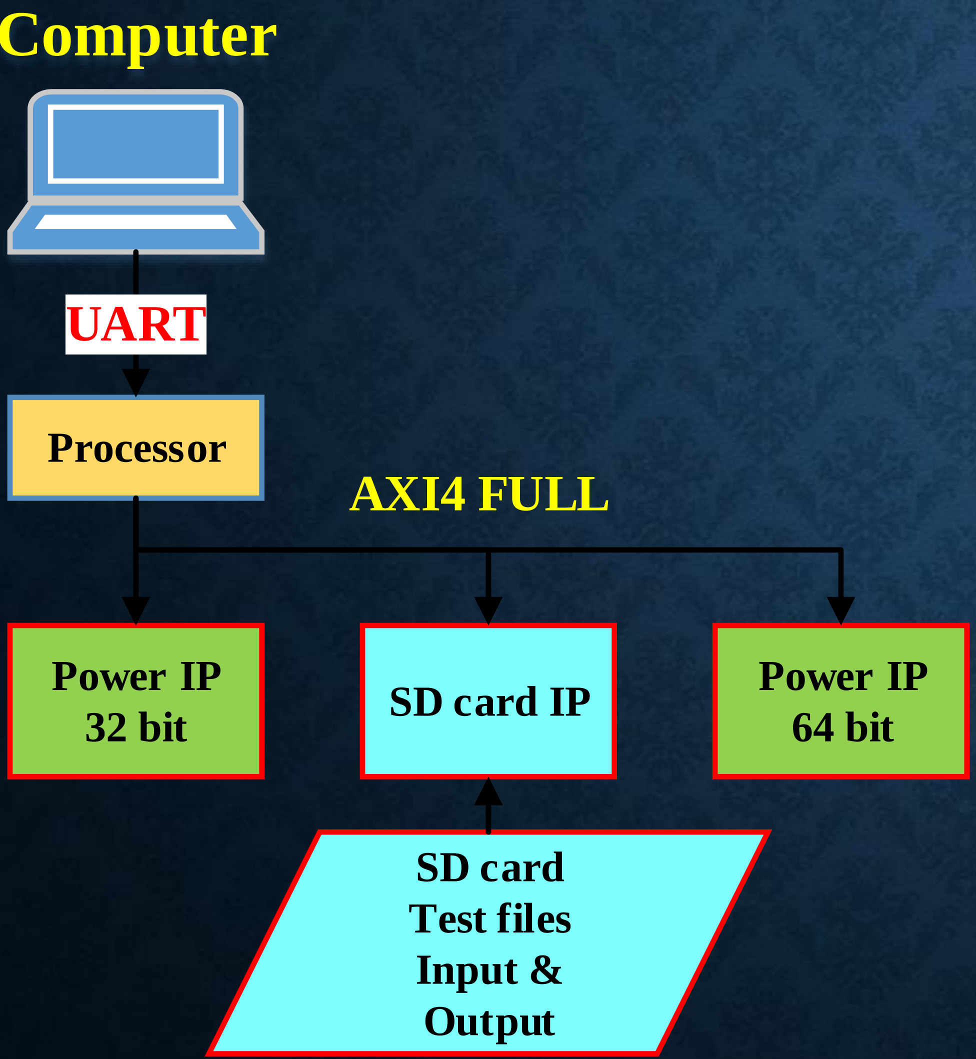
FatFs Module

- FatFs API - f_open, f_close, f_read, f_write, Directory access, File Management etc.
- Uses Xilffs library API.

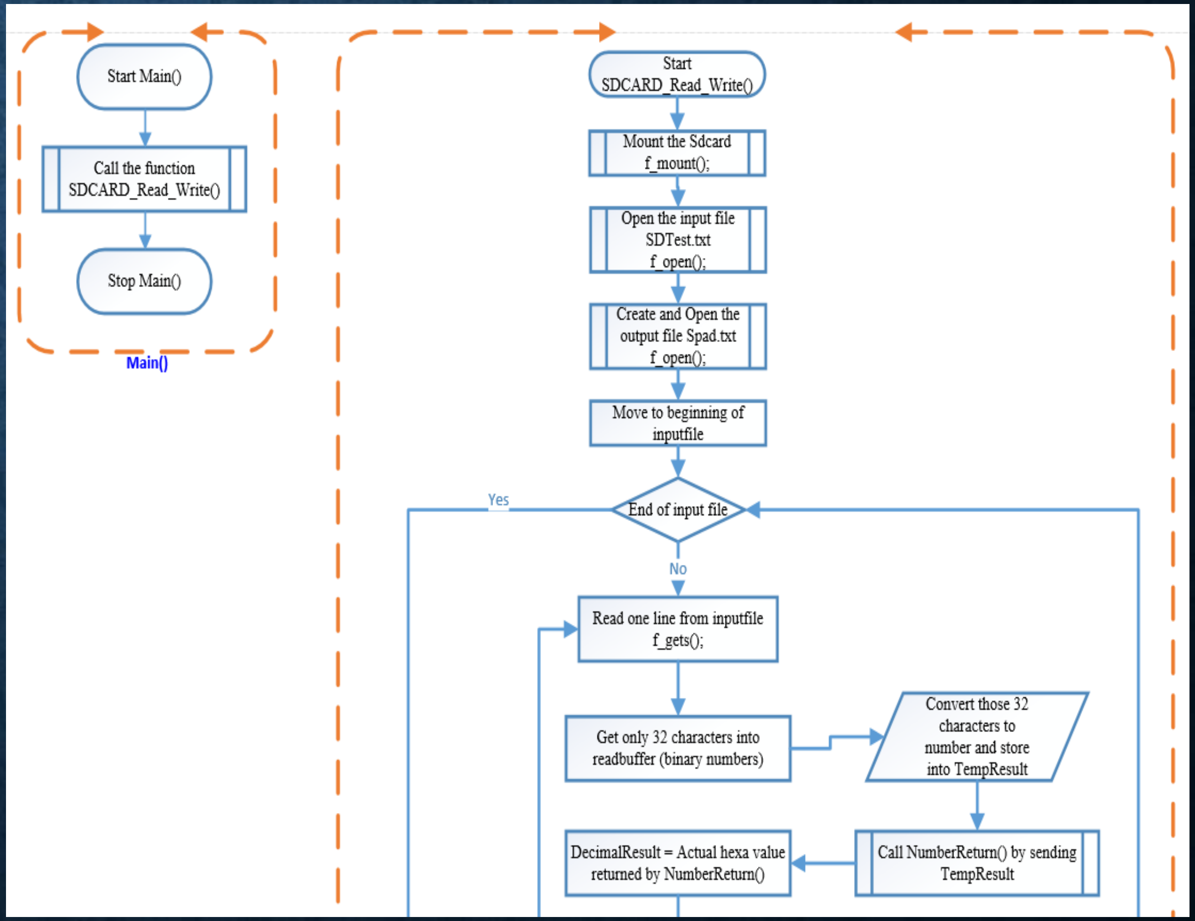
Storage device controls

- DISK I/O – disk_status, disk_read etc.
- Uses Xsdsp driver APIs - XSdPs_SdCardInitialize, XSdPs_ReadPolled

CONTROL LOGIC

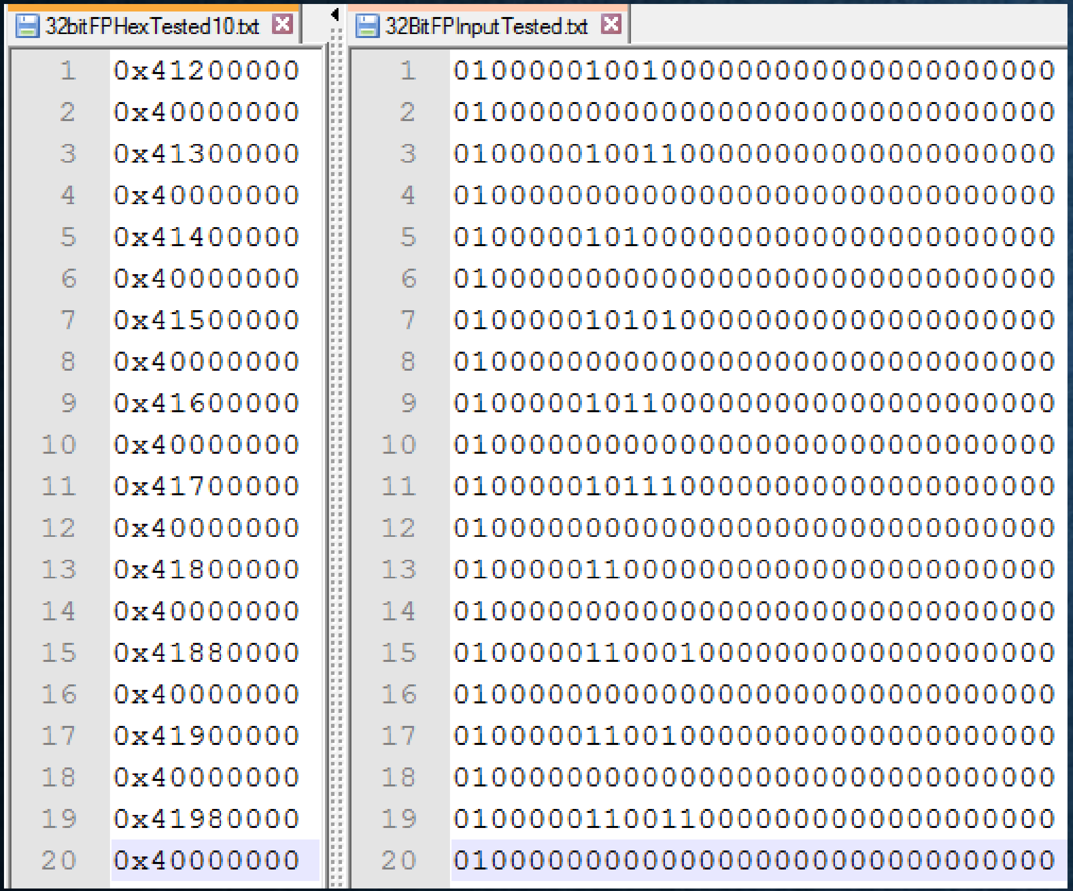


FlowChart


Start Main()
Call the function SDCARD_Read_Write()
Stop Main()
Main()
Start SDCARD_Read_Write()
Mount the Sdcard f_mount();
Open the input file SDTest.txt f_open();
Create and Open the output file Spad.txt f_open();
Move to beginning of inputfile
End of input file
Yes
No
Read one line from inputfile f_gets();
Get only 32 characters into readbuffer (binary numbers)
Convert those 32 characters to number and store into TempResult
Call NumberReturn() by sending TempResult
DecimalResult = Actual hexa value returned by NumberReturn()

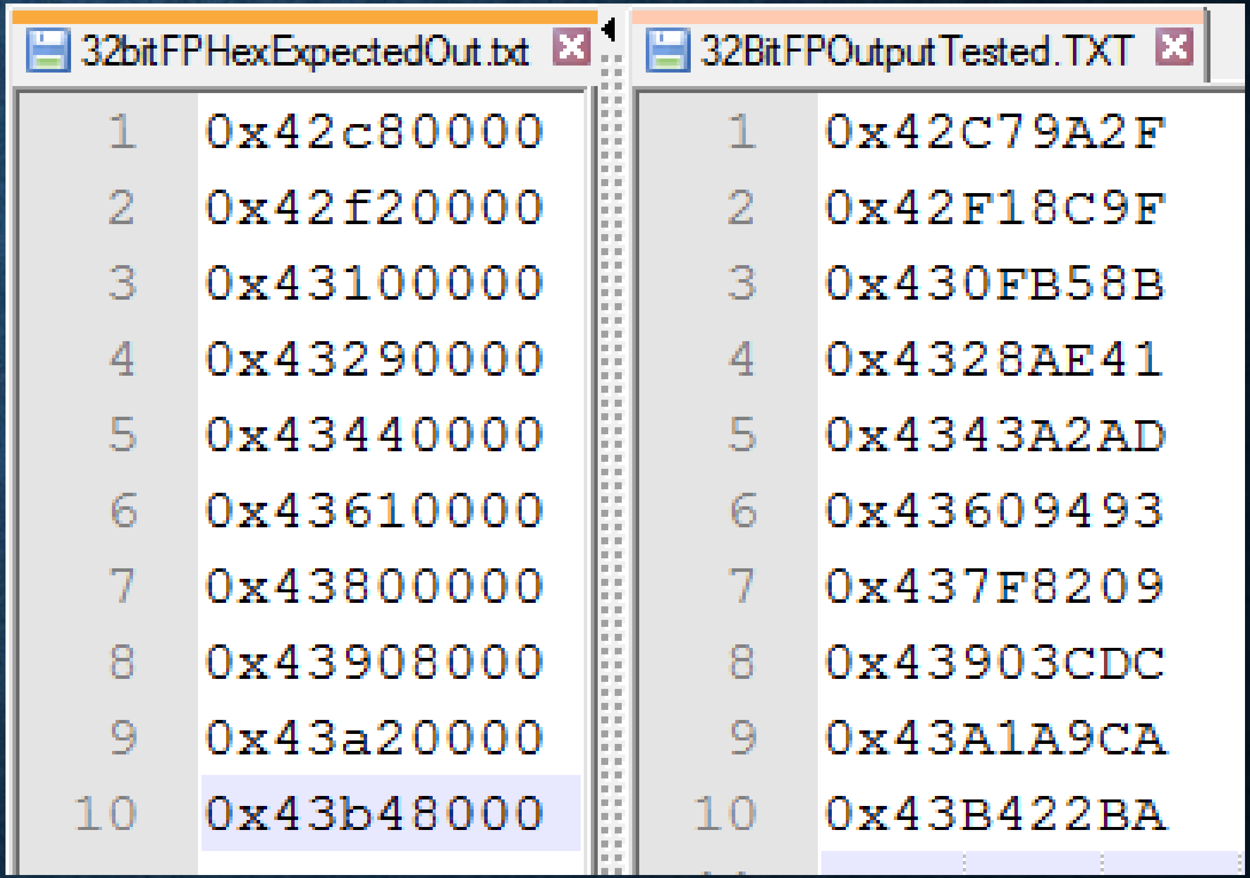
TEST RESULTS – 32 BIT

32 bit Input



```
32bitFPHexTested10.txt    32BitFPInputTested.txt
0x41200000                01000001001000000000000000000000
0x40000000                01000000000000000000000000000000
0x41300000                01000001001100000000000000000000
0x40000000                01000000000000000000000000000000
0x41400000                01000001010000000000000000000000
0x40000000                01000000000000000000000000000000
0x41500000                01000001010100000000000000000000
0x40000000                01000000000000000000000000000000
0x41600000                01000001011000000000000000000000
0x40000000                01000000000000000000000000000000
0x41700000                01000001011100000000000000000000
0x40000000                01000000000000000000000000000000
0x41800000                01000001100000000000000000000000
0x40000000                01000000000000000000000000000000
0x41880000                01000001100010000000000000000000
0x40000000                01000000000000000000000000000000
0x41900000                01000001100100000000000000000000
0x40000000                01000000000000000000000000000000
0x41980000                01000001100110000000000000000000
0x40000000                01000000000000000000000000000000
```

32 bit Output: Expected vs Actual



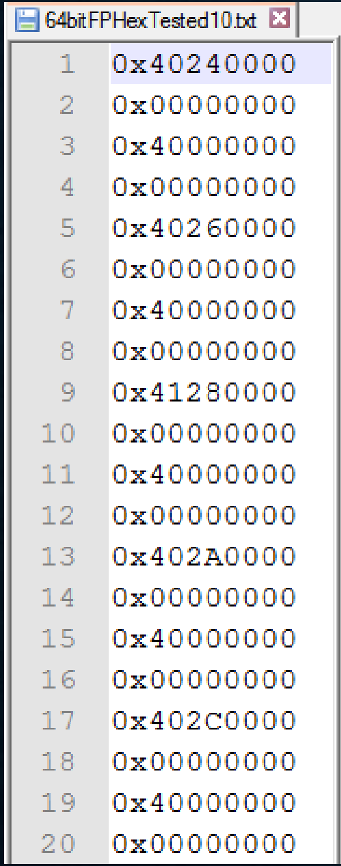
```
32bitFPHexExpectedOut.txt    32BitFPOutputTested.TXT
0x42c80000                   0x42C79A2F
0x42f20000                   0x42F18C9F
0x43100000                   0x430FB58B
0x43290000                   0x4328AE41
0x43440000                   0x4343A2AD
0x43610000                   0x43609493
0x43800000                   0x437F8209
0x43908000                   0x43903CDC
0x43a20000                   0x43A1A9CA
0x43b48000                   0x43B422BA
```

Input -> 1st value 10 in 32-bit IEEE 754 format is 0x 41200000
2nd value 2 in 32-bit IEEE 754 format is 0x40000000

Output -> expected value is 100 in 32-bit IEEE 754 format is 0x 42c800000
Actual value is 99.8 in 32-bit IEEE 754 format is 0x42c79A2F

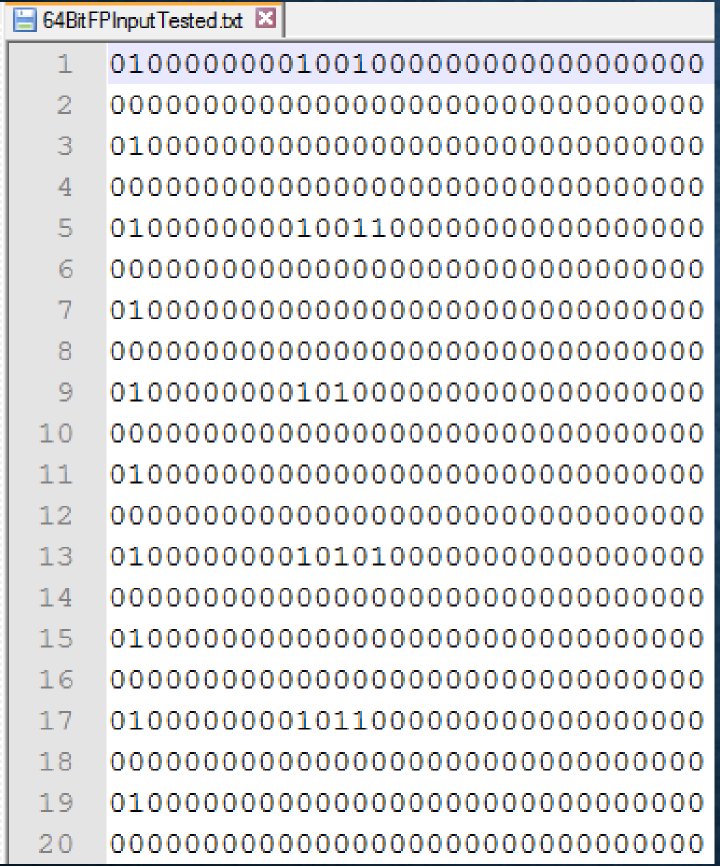
TEST RESULTS – 64 BIT

64 bit Input



64bitFPHexTested10.txt

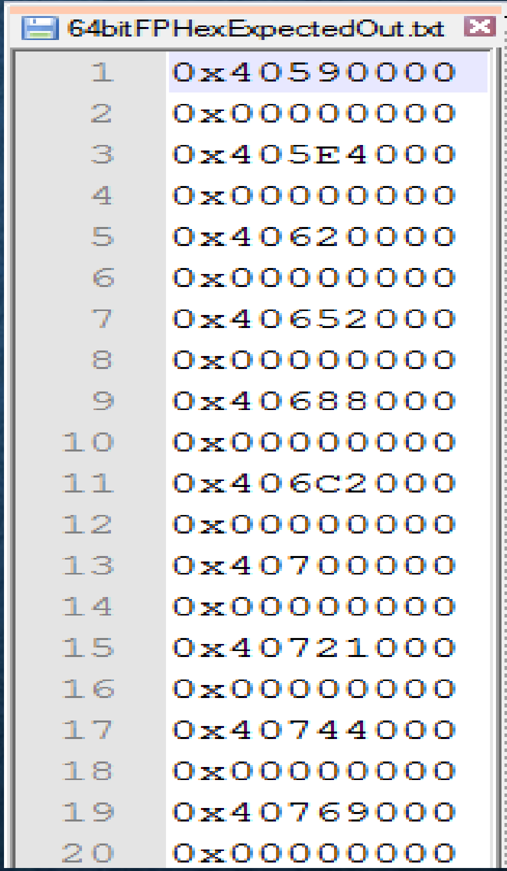
```
0x40240000
0x00000000
0x40000000
0x00000000
0x40260000
0x00000000
0x40000000
0x00000000
0x41280000
0x00000000
0x40000000
0x00000000
0x402A0000
0x00000000
0x40000000
0x00000000
0x402C0000
0x00000000
0x40000000
0x00000000
```



64BitFPInputTested.txt

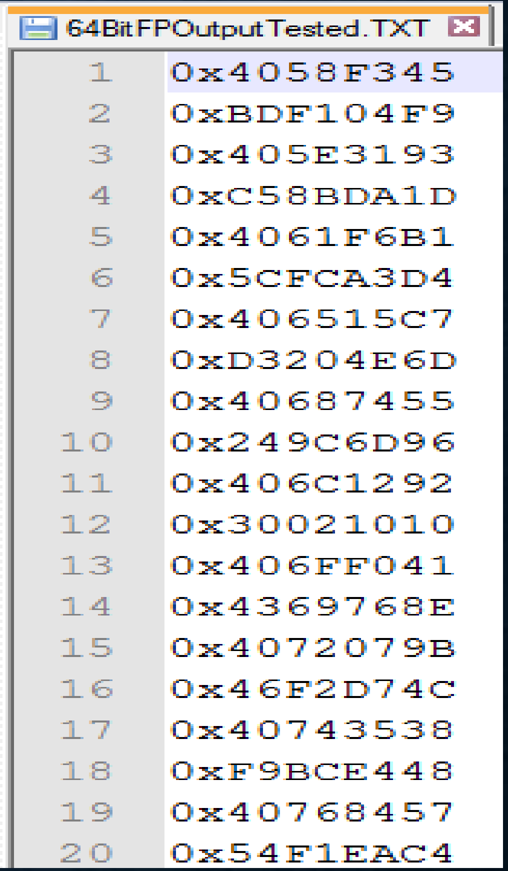
```
01000000001001000000000000000000
00000000000000000000000000000000
01000000000000000000000000000000
00000000000000000000000000000000
01000000001001100000000000000000
00000000000000000000000000000000
01000000000000000000000000000000
00000000000000000000000000000000
01000000001010000000000000000000
00000000000000000000000000000000
01000000000000000000000000000000
00000000000000000000000000000000
01000000001010100000000000000000
00000000000000000000000000000000
01000000000000000000000000000000
00000000000000000000000000000000
01000000001011000000000000000000
00000000000000000000000000000000
01000000000000000000000000000000
00000000000000000000000000000000
```

64 bit Output: Expected vs Actual



64bitFPHexExpectedOut.txt

```
0x40590000
0x00000000
0x405E4000
0x00000000
0x40620000
0x00000000
0x40652000
0x00000000
0x40688000
0x00000000
0x406C2000
0x00000000
0x40700000
0x00000000
0x40721000
0x00000000
0x40744000
0x00000000
0x40769000
0x00000000
```



64BitFPOutputTested.TXT

```
0x4058F345
0xBDF104F9
0x405E3193
0xC58BDA1D
0x4061F6B1
0x5CFCA3D4
0x406515C7
0xD3204E6D
0x40687455
0x249C6D96
0x406C1292
0x30021010
0x406FF041
0x4369768E
0x4072079B
0x46F2D74C
0x40743538
0xF9BCE448
0x40768457
0x54F1EAC4
```

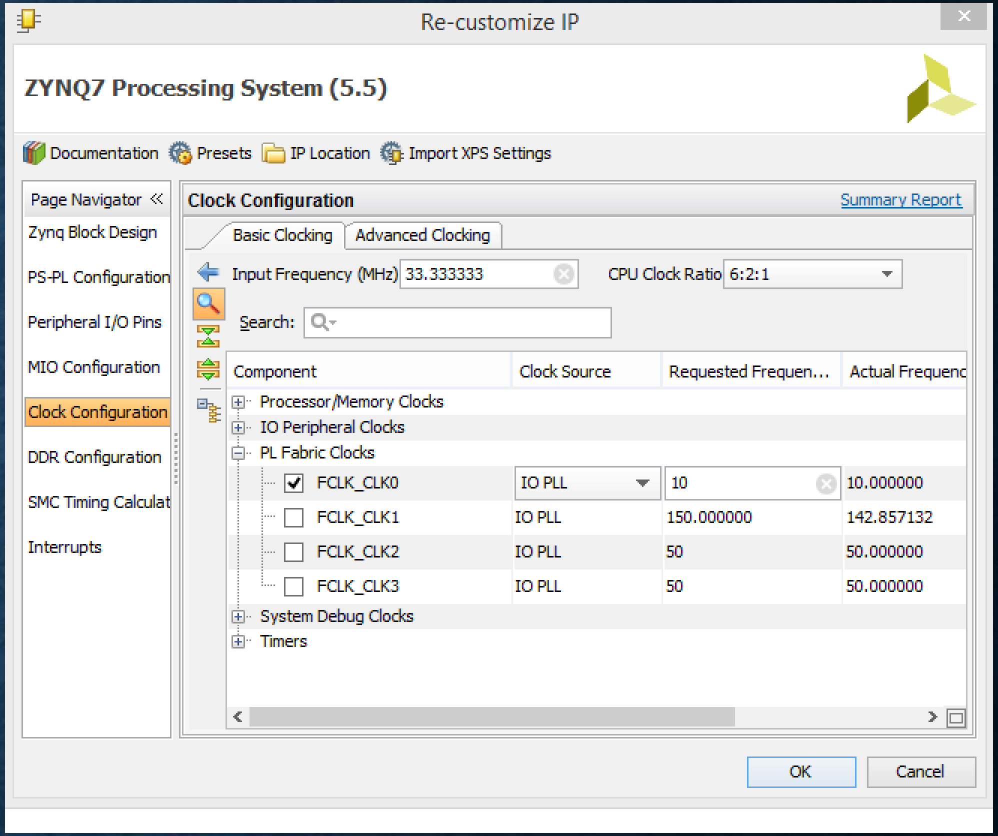
Input -> 1st and 2nd value 10 in 64-bit IEEE 754 format is 0x 40240000 00000000
3rd and 4th value 2 in 64-bit IEEE 754 format is 0x40000000 00000000

Output -> expected value is 100 in 64-bit IEEE 754 format is 0x 40590000 00000000
Actual value is 99.8 in 64-bit IEEE 754 format is 0x4058F345 BDF104F9

TIMING ISSUE

During the implementation, Vivado detected "timing violation" error.

- Frequency of AXI bus was originally 100 MHz.
- With help from Professor, reduced the frequency.
- Finally, settled at 10 MHz. (50MHz didn't work)



Re-customize IP

ZYNQ7 Processing System (5.5)

Clock Configuration

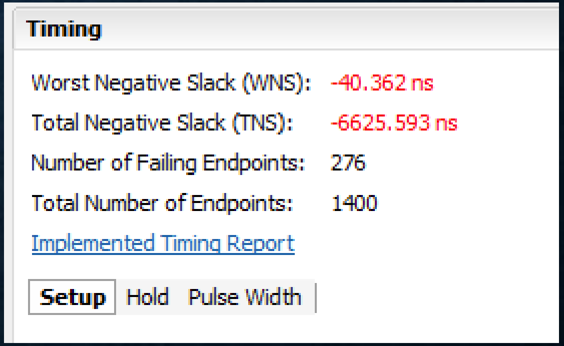
Input Frequency (MHz) 33.333333 CPU Clock Ratio 6:2:1

Search:

Component	Clock Source	Requested Frequen...	Actual Frequenc
Processor/Memory Clocks			
IO Peripheral Clocks			
PL Fabric Clocks			
✓ FCLK_CLK0	IO PLL	10	10.000000
FCLK_CLK1	IO PLL	150.000000	142.857132
FCLK_CLK2	IO PLL	50	50.000000
FCLK_CLK3	IO PLL	50	50.000000
System Debug Clocks			
Timers			

<u>Root of this problem</u>

- Long combinational logic.
- Specially, negative iteration of CORDIC uses two floating point adder during one clock cycle, propagation delay exceeded to clock frequency.



Timing

Worst Negative Slack (WNS):	-40.362 ns
Total Negative Slack (TNS):	-6625.593 ns
Number of Failing Endpoints:	276
Total Number of Endpoints:	1400

Implemented Timing Report

DEMONSTRATION

THANK YOU VERY MUCH

Any Questions?