Industrial Training Report

submitted to

Bhilai Institute of Technology, Durg

for

Summer Training

on

Controller Design for QSPI NOR Flash

On-Site Internship

at

Space Applications Centre, Ahmedabad
Indian Space Research Organization (ISRO)

by

ADHIRAJ ROY CHOWDHURY

B.Tech Electronics & Telecommunication Engineering

Semester: VII

University Roll Number: 300102819006

Enrollment No: BH2854

DEPARTMENT OF ELECTRONICS & TELECOMMUNICATION ENGINEERING
BHILAI INSTITUTE OF TECHNOLOGY, BHILAI HOUSE, DURG (C.G.) -491001, INDIA

SESSION 2022-2023

D E C L A R A T I O N

I sincerely declare that:

1. I am the sole writer of this report
2. The detail of training and experience contained in this Industrial Training report describe my involvement as a trainee in the field of ET &T engineering.
3. All the information contained in this report is based on my field experience during training.

Adhiraj Roy Chowdhury

300102819006

A C K N O W L E D G E M E N T

‘Nothing has been achieved without constant inspiration’. so, at the completion of this project, I would like to dedicate my hardwork to the unending enthusiasm of my guide Shri Jimit J. Gadhia without his guidance it might have been a non-entity.

My deep and sincere thanks to Mr. Anirban Paul and Mrs. Suchi Khare for their propositions and ideas bestowed during this project.

My sincere thanks to Indian Space Research Organization (ISRO) and the Scientific Research and Training Division of Space Applications Centre, Ahmedabad for giving me the opportunity of learning and doing the Project.

Finally, I thank one and all associated with this project.

भारत सरकार
अंतरिक्ष विभाग
अंतरिक्ष उपयोग केन्द्र
आंबावाड़ी विस्तार डाक घर,
अहमदाबाद-380 015. (भारत)
दूरभाष : +91-79-26913050, 26913060
वेबसाईट : www.sac.isro.gov.in/www.sac.gov.in

Government of India
Department of Space
SPACE APPLICATIONS CENTRE
Ambawadi Vistar P.O.
Ahmedabad - 380 015. (INDIA)
Telephone : +91-79-26913050, 26913060
website : www.sac.isro.gov.in/www.sac.gov.in

Scientific Research and Training Division (SRTD)
Research, Outreach and Training Coordination Group (RTCG)
Management and Information Systems Area (MISA)

CERTIFICATE

This is to certify that **Mr. Adhiraj Roy Chowdhury**, a student of B.Tech (Electronics and Telecommunications Engineering) of Bhilai Institute of Technology, Durg, Chhattisgarh has completed a 6 weeks (1st May 2022 to 16th June 2022.) Project on **"Controller design for QSPI NOR Flash"** under the supervision of Mr. Jimit J. Gadhia, Scientist-SF, SCPD/SEG/SEDA, Space Applications Centre (ISRO), Ahmedabad. The research work was carried out through Scientific Research and Training Division of Space Applications Centre (ISRO) Ahmedabad.

डॉ सर्वेधर व्यास / Dr. S P Vyas
प्रधान, वैज्ञानिक अनुसन्धान एवं प्रशिक्षण प्रभाग
Head, Scientific Research and Training Division
एसआरटीडी-आरटीसीजी सीएमए/ SRTD-RTCG-MISA
अंतरिक्ष उपयोग केंद्र (इसरो)
Space Applications Centre(ISRO)
अंतरिक्ष विभाग / Department of Space
भारत सरकार / Government of India
अहमदाबाद / Ahmedabad - 380015

I N D E X

A B S T R A C T

Flash memory, whether it is NOR or NAND in structure, is a non-volatile memory that is used to replace traditional EEPROM and hard disks for its low cost and versatility. Because of the difference in the structure of interconnection of the memory cells, NAND Flash is known for its compact size and high speeds for page accesses, while the NOR Flash is known for its random-access capability. NOR Flash is mainly used for code storage and execution.

A memory controller is needed to perform read/ write/ erase operations on the memory.

This project involves designing a NOR FLASH controller for a given QSPI NOR Flash memory device. The main objective of this project was to go through the entire cycle of design for implementation in FPGA.

Programming Language used - VHDL
Tools used - Questasim 10.1d, Microsemi Libero SOC v11.9
Devices used - RT4G150_ES, Dell Precision Tower 7810 (My workstation), 3DFS256M04VS2801 (NOR Flash memory used in this project)

CHAPTER – 1

I N T R O D U C T I O N

The most relevant phenomenon of this past decade in the field of semiconductor memories has been the explosive growth of them. Flash memory market, driven by cellular phones and other types of electronic portable equipment. Moreover, in the coming years, portable systems will demand even more non-volatile memories, either with high density and very high writing throughput for data storage application or with fast random access for code execution in place.

Basically, two types of flash storages are available NOR based and NAND based flash memory. Further, there are two types of NOR and NAND flash memories: serial NOR, serial NAND and parallel NOR, parallel NAND. For reliable code storage such as booting code, application code, operating system NOR based flash storage is preferred. The NAND type is preferred for its greater data storage ability.

A Flash controller communicates with flash memory. To operate the flash memory various signals are needed to be generated. The Flash controller has to handle all these signals with specified timing parameters to perform operation on flash memory. Multiple types of Serial NOR flash memories are available in market. We will be designing a controller for a serial NOR flash memory, part number - 3DFS256M04VS2801 from Ms 3D plus.

This report focuses on the implementation of a memory controller for serial NOR flash memory and discusses the various steps involves in its design and implementation in FPGA. The Project report is organized in the following way:

Chapter 2 gives a detail on Flash Memory, discussing its types and architectures. Chapter 3 deals with the memory device we are using, 3DFS256M04VS2801, describing the timing requirements for the various commands. In Chapter 4, I have tried to explain the processes in FPGA Designing. A brief description of the FPGA Device (RT4G150_ES) used in this project has been provided too, along with an introduction to VHDL programming. Chapter 5 entirely deals with the coding and the designing of the controller device, after which, the simulation outputs have been discussed in Chapter 6. The report is concluded with the Concluding Remarks in Chapter 7 and the Bibliography at the end.

CHAPTER – 2

F L A S H M E M O R Y & S P I P R O T O C O L

The term Memory is used to describe a system with the ability to store digital information. The term semiconductor memory refers to systems that are implemented using integrated circuit technology. These types of systems store the digital information using transistors, fuses, and/or capacitors on a single semiconductor substrate. Memory is a key element in the design of modern digital electronics. Memory allows the retention of information that can be used by the electronic system. This information may be computer files, but it can also be digital photographs, home videos, movies, music, and other personal and commercial contents.

MEMORY CLASSIFICATION:

Memory is classified into two categories - Non-Volatile is used to describe memory that holds information when the power is removed, while the term Volatile is used to describe memory that loses its information when power is removed. Historically, volatile memory is able to run at faster speeds compared to non-volatile memory, so it is used as the primary storage mechanism while a digital system is running. Non-volatile memory is necessary in order to hold critical operation information for a digital system such as start-up instructions, operations systems, and applications.

Memory can also be classified into two categories with respect to how data is accessed.

RANDOM ACCESS MEMORY (RAM) describes memory in which any location in the system can be accessed at any time. The opposite of this is Sequential access memory, in which not all address locations are immediately available. An example of a sequential access memory system is a tape drive. Most semiconductor memory in modern systems is random access.

The three types of semiconductor memories generally used in process execution are *SRAM* (static random-access memory), *DRAM* (dynamic random-access memory) and *NOR* (refers to “neither or” a logical operation) memory.

SRAM – stands for Static RAM. Transistors are used to store information in SRAM. bits are stored in voltage form. SRAM retains its contents as long as electrical power is applied to the chip. If the power is turned off or lost temporarily, its contents will be lost forever. It is faster than DRAM, but more expensive and bulkier, having six transistors in each cell. For these reasons, SRAM is generally

used only as a data cache within a CPU itself or as RAM in very high-end server systems.

DRAM – stands for Dynamic RAM. It is the most common type of RAM used in computers. DRAM consists of a transistor and a capacitor in each cell. The data is stored in capacitors in the form of electric energy. Capacitors that store data in DRAM gradually discharge energy, no energy means the data has been lost. DRAM, has an extremely short data lifetime-typically about four milliseconds even when power is applied constantly. So, a periodic refresh of power is required in order to function. DRAM is called dynamic as a constant change or action, i.e., refreshing is needed to keep the data intact. It is used to implement main memory. Even though DRAM is slower than SRAM and requires the overhead of a refresh circuit, it is still much cheaper and takes about one quarter of the space of SRAM.

NOR is a type of flash memory. Flash memory is non-volatile, which means that it does not need power to maintain the information stored in the chip. Flash memory offers slower read times compared to volatile DRAM memory. Reading from NOR flash is similar to reading from random-access memory, provided the address and data bus are mapped correctly. Because of this, most microprocessors can use NOR flash memory as execute-in-place (XIP) memory, meaning that programs stored in NOR flash can be executed directly without the need to copy them into RAM. The capability of acting as a random-access ROM (read-only memory) allows NOR to provide a low-power, low-cost option to DRAM or SRAM in consumer products

Read-only memory (ROM) is a device that cannot be written to during normal operation. This type of memory is useful for holding critical system information or programs that should not be altered while the system is running. Read/write memory refers to memory that can be read and written to during normal operation and is used to hold temporary data and variables.

ROM is further classified into 4 types- MROM, PROM, EPROM, EEPROM. A brief about all these types is given below-

PROM – stands for Programmable Read-Only Memory. It can be programmed by the user, and once programmed, the data and instructions in it cannot be changed.

MROM – Mask ROM. It is a kind of read-only memory, that is masked off at the time of production. Like other types of ROM, mask ROM cannot enable the user to change the data stored in it. If it can, the process would be difficult or slow.

EPROM – Erasable Programmable ROM. EPROM is reprogrammable. To erase data from it, expose it to ultraviolet light. To reprogram it, erase all the previous data.

EEPROM - Electrically Erasable Programmable ROM. In EEPROM, the data can be erased by applying an electric field, with no need for ultraviolet light. We can erase only portions of the chip.

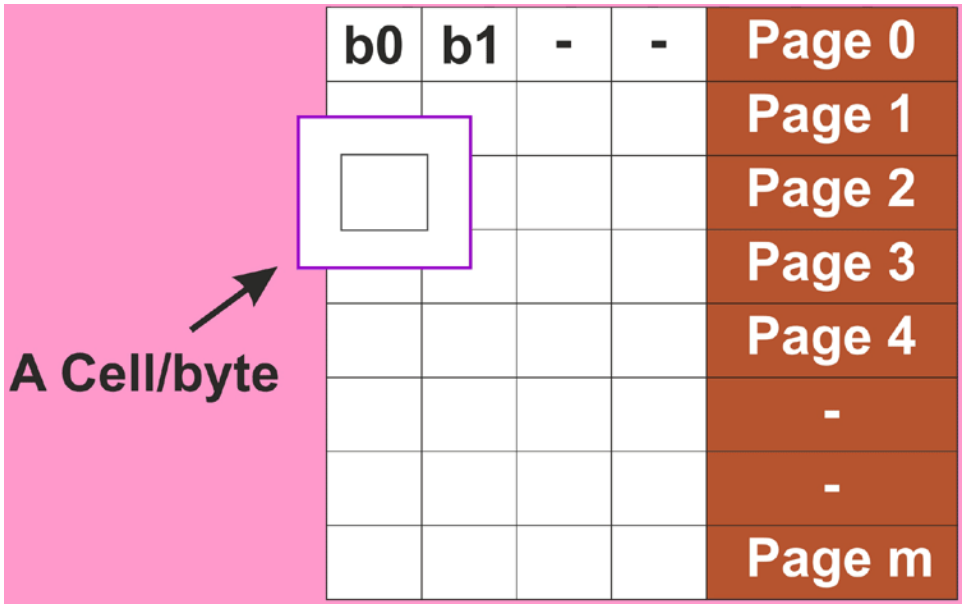
EEPROM AND FLASH MEMORY

One of the drawbacks of EEPROM was that the circuitry that provided the capability to program and erase individual bits also added to the size of each individual storage element. FLASH EEPROM was a technology that attempted to improve the density of floating-gate memory by programming and erasing in large groups of data, known as blocks. This allowed the individual storage cells to shrink and provided higher-density memory parts. This new architecture was called NAND FLASH and provided faster write and erase times coupled with higher-density storage elements. The limitation of NAND FLASH was that reading and writing could only be accomplished in a block-by-block basis. This characteristic precluded the use of NAND FLASH for run-time variables and data storage but was well suited for streaming applications such as audio/video and program loading. As NAND FLASH technology advanced, the block size began to shrink, and the software adapted to accommodate the block-by-block data access. This expanded the applications that NAND FLASH.

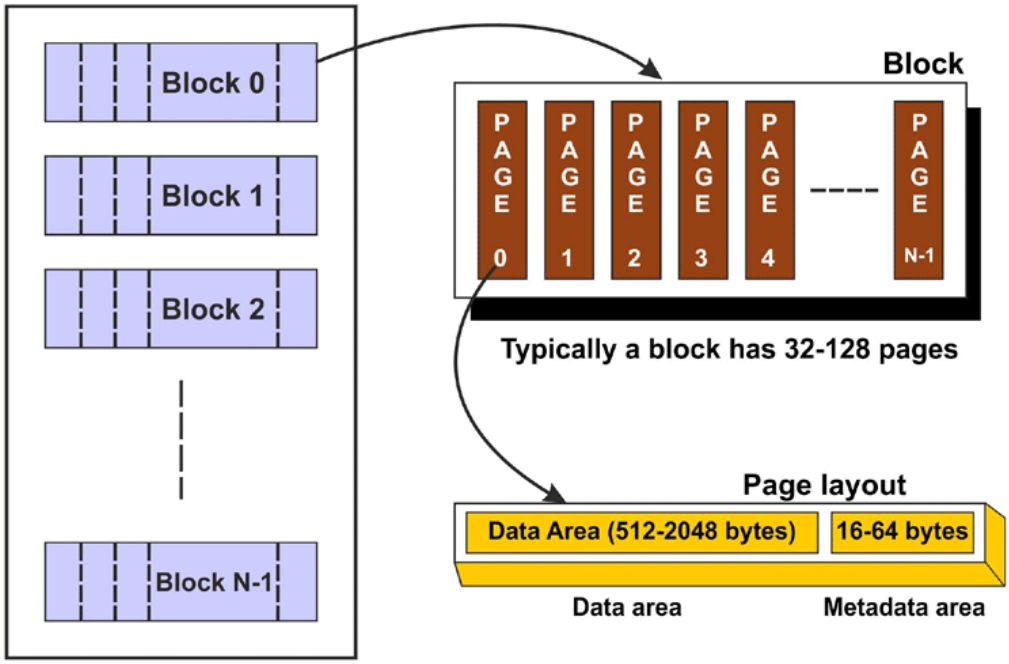
In order to provide individual word access, NOR FLASH was introduced. In NOR FLASH, circuitry is added to provide individual access to data words. This architecture provided faster read times than NAND FLASH, but the additional circuitry causes the write and erase times to be slower and the individual storage cell size to be larger. NOR FLASH is considered random access memory, while NAND FLASH is typically not. All FLASH memory is non-volatile and read/write.

READ, WRITE AND ERASE IN EEPROM AND FLASH –

In EEPROM, bytes are arranged into pages and the read, write and erase commands can be performed at the byte level. This eases the write operation, as one can simply overwrite the new data to the required address. The image below shows its architecture –



However, in FLASH MEMORY, memory information is stacked in blocks. Then each block is stored in multiple pages. In flash memory, due to its block-like architecture, erase can only be done at the block level, and read or write can be done at block, page, or byte-level based upon the type of gate used (NOR & NAND). To reprogram the flash memory, first, the entire section containing that location must be erased. The figure below shows the architecture –



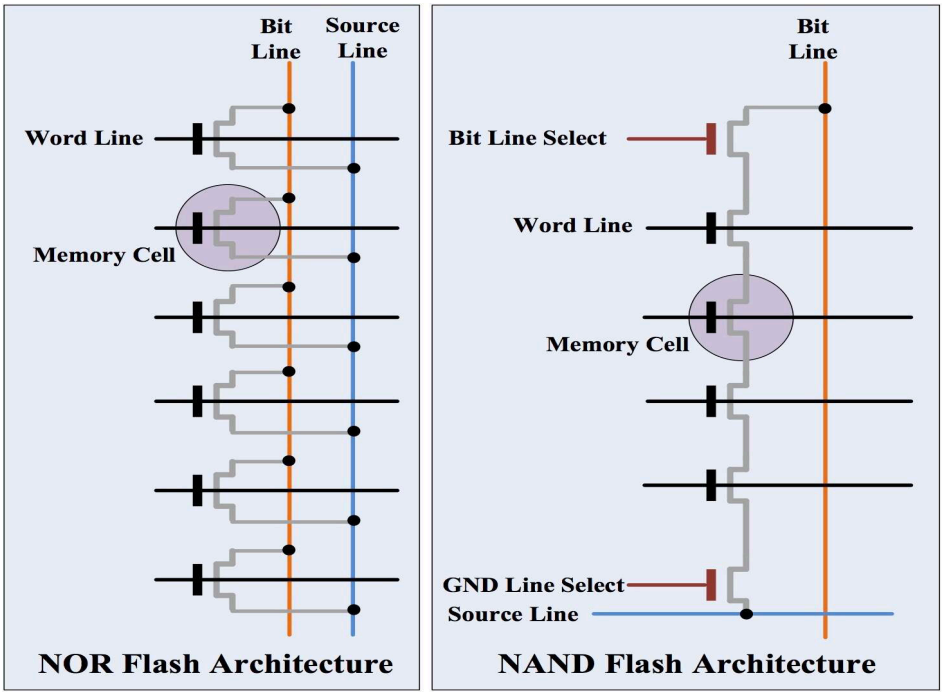
To sum up,

1) Write cycles are faster in Flash than EEPROM.
2) Read cycles are faster in EEPROM than in Flash
3) Erase cycles are faster in Flash than EEPROM.
4) Flash memories can hold large amounts of data to the scale of gigabytes and more, while regular EEPROM memory chips are usually used to store only small amounts of data.
5) Flash is available bigger (by up to 10x), cheaper, and faster (>10x for SPI versions)
6) EEPROM advantage (besides byte erase) is that it endures more write/erase cycles by 10x.
7) Flash memory devices are generally I2C or SPI protocol based.

TYPES OF FLASH MEMORY:

Based on the type of logic gate each type utilizes flash memories are categorized into 2 types:

1) NOR Flash memory: have faster read speed than NAND and can read and edit more precisely, but it comes at a higher price per byte. Mostly used for code execution purposes.
2) NAND Flash memory: have slower read speed and can only access memory in blocks rather than bytes, but it is cheaper than NOR. It works well for storing large and frequently-updated files.



NAND FLASH:

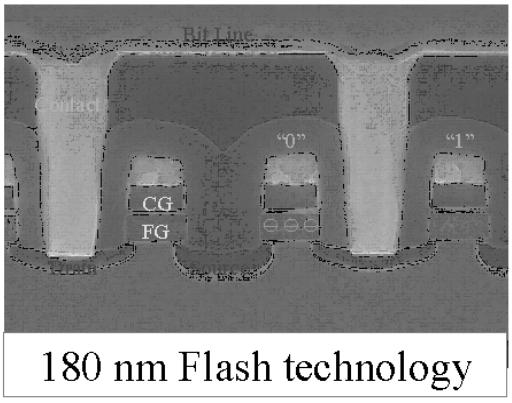
Memory cells in NAND Flash are organized in series. This series array is called NAND String and can contain 32 to 64 memory cells. There are two selection transistors that connect each NAND String to Source Line and Bit Line. A Word Line (WL) connects control gates of all memory cells in a row. Cells that are connected by a single Word Line form a memory Page, which is the smallest unit that can be programmed. Multiple Word Lines form a Block in a Flash memory, which is the smallest unit to be erased i.e., any erase operation has to span an entire Block. Each of the cells store either a single bit of information in case of Single Level Cell (SLC) technology or multiple bits of information in case of Multi-Level Cell (MLC) Flash technology. NAND Flash memory allows for higher storage density compared to other kinds of Flash memory, e.g., NOR Flash memory. They have faster program times and faster erase times than NOR types. However, since they are arranged in series random access is not available. They are good for high volume storage applications, but since they do not provide random access, they are not used in cases where execute-in-place functionality is required.

NOR FLASH:

NOR flash is known for its random-access capabilities, which means it is capable of accessing data in any order and does not require following a sequence of storage locations. NOR Flash is ideal for lower-density, high-speed read applications, which are mostly read only, often referred to as code-storage applications. In NOR flash, each cell has one end connected to source line, and the other end connected directly to a bit line. Because of this, the system is able to access individual memory cells.

Reading performance: Since the cells are connected in parallel making them directly accessible, NOR Flash have shorter read times than its NAND counterpart.

Write and Erase performances: due to NOR Flash having a larger cell size (when compared to NAND type) and its more complicated erasure process, it cannot write and erase as quickly as compared to NAND.



NOR Flash cross-section

SPI PROTOCOL:

The Serial Peripheral Interface (SPI) is a synchronous serial communication interface specification used for short-distance communication, primarily in embedded systems.

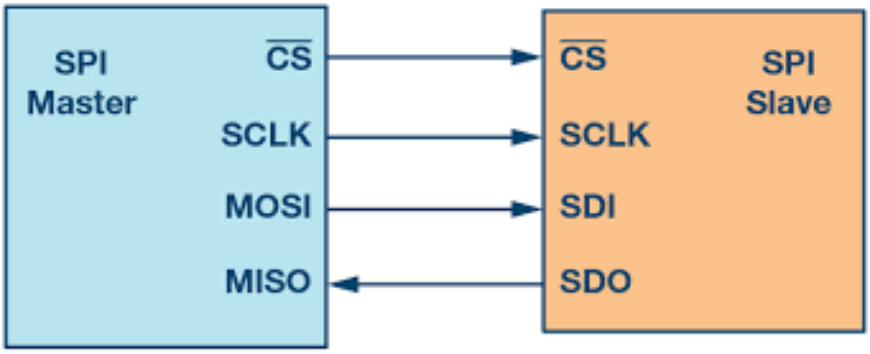
SPI Simple connection between master and slave:

SPI Devices communicate in full duplex mode using a master-slave architecture with a single master. The master device originates the frame for reading and writing. Multiple slave-devices are supported through selection with individual slave select (SS), sometimes called chip select (CS) line.

The SPI bus specifies 4 logic signals:

- SCLK: Serial Clock (output from master)
- MOSI: Master Out Slave In (data output from master)
- MISO: Master In Slave Out (data output from slave)
- SS or CS: Slave Select or Chip Select (often active low, output from master)

.



Quad SPI has been explained in Chapter 3

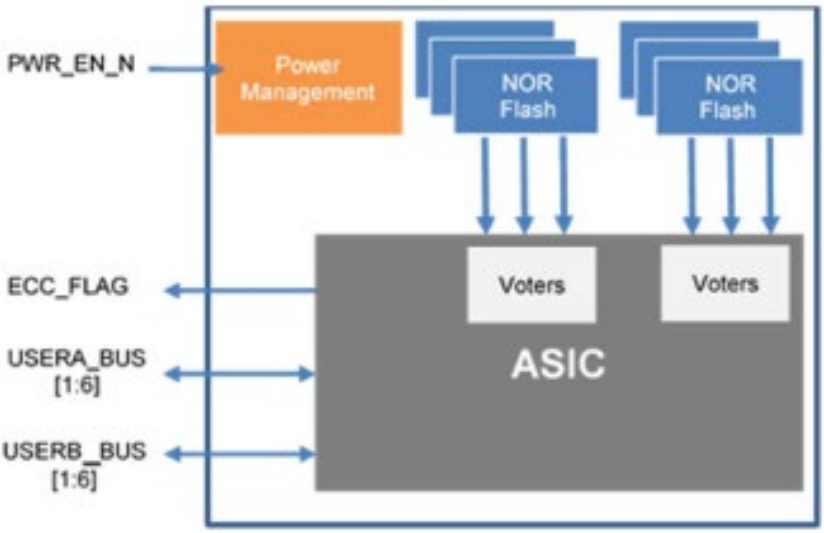
CHAPTER – 3

3D+ QSPI NOR FLASH MEMORY

SYSTEM HARDWARE:

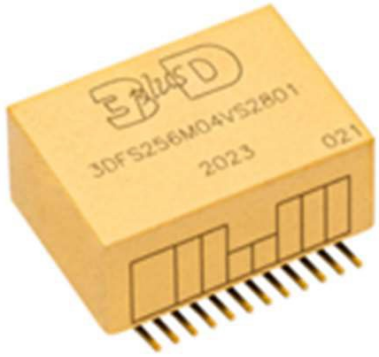
The NOR Flash memory controller is developed considering Ms 3D plus make 256 Mbit QSPI NOR Flash memory. The part number is 3DFS256M04VS2801. The memory device is designed for fault tolerance, and contains three 256 Mbit NOR Flash memories for redundancy. Each logic block is implemented three times with a 2-out-of-3 voter at the output. Thus, the correct logic value is available even if there is an upset bit in one location. Triple Modular Redundancy (TMR) majority voting function ensures Single Event Upset (SEU) immunity. Each 256 Mbit memory is made with two banks of 128 Mbit. Along with the three memories, the module also contains a Power management block and an ASIC. The power management block is used to control the supply voltage of the device. The ASIC provides a bridge between the QSPI buses and the QSPI memories and the TMR voting logic. It integrates transceivers connecting the memories and two distinct QSPI interfaces. QSPI is controller extension to SPI bus. It stands for Queued Serial Peripheral Interface. It uses data queue with pointers which allow data transfers without any CPU. In addition it has wrap-around mode which allows continuous transfer of data to/from queue without the need of CPU.

The block diagram of the module is as shown below-



KEY FEATURES

- 256 Mbit density
- Single supply voltage: 3.3 V
- Supports standard SPI, Dual, QSPI modes
- Supports only 16 bits word access (read
- and write) at even address
- Two QSPI user buses
- Up to 50 MHz in Fast Read Mode
- 100,000 + erase/program cycles
- More than 20-year data retention
- Low Instruction Overhead Operations
- Continuous Read 8-byte burst
- Available Temperature Range: 0°C to +70°C, -40°C to +85°C, -55°C to +105°C
- Package: SOP 24 -0.65mm pitch

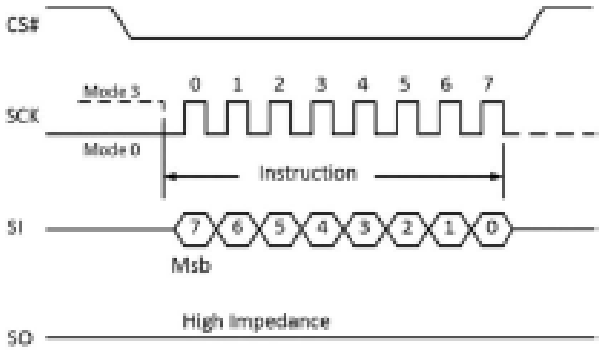


TMR:

TMR stands for Triple Modular Redundancy. The Triple Modular Redundancy technology allows protection of the functionality of FPGAs against single event upsets (SEUs). Each logic block is implemented three times with a 2-out-of-3 voter at the output. Thus, the correct logic value is available even if there is an upset bit in one location.

MODE 0 (0,0) AND MODE 3 (1,1):

The NOR Flash supports two SPI modes: Mode 0 (0, 0) and Mode 3 (1, 1). The difference between these two modes is the clock polarity. The serial clock is idle at “0” (SCK=0) for Mode 0 and the clock is idle at “1” (SCK=1) for Mode 3.



MEMORY DEFINITIONS:

PIN #	SIGNAL NAME	PIN #	SIGNAL NAME
1	GND	13	GND
2	USERA_SCK	14	ECC_FLAG
3	USERA_CE_N	15	VCC
4	VCC	16	USERB_IO3
5	USERA_IO0	17	USERB_IO2
6	USERA_IO1	18	GND
7	GND	19	USERB_IO1
8	USERA_IO2	20	USERB_IO0
9	USERA_IO3	21	VCC
10	VCC	22	USERB_SCK
11	PWR_EN_N	23	USERB_CE_N
12	GND	24	GND

Table 2: Pinout

SIGNAL NAME	DESCRIPTION	COMMENTS
USERy_CE_N(1)	Chip Enable	Enable or disable all memories in the module. Active Low.
PWR_EN_N	Power Enable	When PWR EN N is low, the QSPI NOR Flash memories are powered with V_{CC} and the various data and commands are transmitted to the memories selected with CE_N pin. When PWR EN N is high, the supply of the powered memories is cut-off, and their input are in High-Z state
ECC_FLAG	Radiation failure notification	Output
USERy_IO0(1)	Data Input used to write to the device on the rising edge of the clock (SCK).	USERy_IO0 is a bidirectional IO to write instructions, addresses or data to the device on the rising edge of the clock and read data or status from the device on the falling edge of SCK. In Quad SPI, USERy_IO1 is only an output.
USERy_IO1(1)	Data Output used to read data or status of the device on the falling edge of SCK	
USERy_IO2(1) (WP#)	Data Input/Output	The WP# pin is used in conjunction with SRWD bit to protect the Status Register. For more details, please refer to section 5.3.1 In Quad SPI mode (QE bit set at "1"), this pin is used as USERy_IO2.
USERy_IO3(1) (HOLD#)	Data Input/Output	The HOLD# pin allows the device to be paused while it is selected. It pauses serial communication by the master device without resetting the serial sequence. The HOLD# pin is active low. When HOLD# is in a low state and USERy_CE_N is low, the USERy_IO1 pin will be at high impedance. Device operation can resume when HOLD# pin is brought to a high state. In Quad SPI mode (QE bit set at "1"), this pin is used as USERy_IO3.
USERy_SCK(1)	Synchronized clock	
VCC(2)	Voltage Supply	
GND(2)	Ground	

The memory pinout and pin descriptions are as shown above.

USERA and USERB in the snaps above denote the 2 SPI Buses that can interact with the memory, i.e., 2 distinct users can access the memory. Memory access by two distinct users shall be managed through the USERy_CE_N pin. However, at a time only one of the CE_N pin shall be activated (set low), and in such cases, USERA will have the priority to interact with the memory.

FLASH OPERATIONS:

Here is a list of Flash operations supported by the memory module: -

SPI COMMAND	OPCODE
Write status register	01h
Page Program	02h
Read Page	03h
Write disable	04h
Read status register	05h
Write enable	06h
Fast Read	0Bh
Fast Read, 32-bit address	0Ch
Page Program 32 bits	12h
Read 32 bits	13h
Exit 4 bytes mode	29h/E9h
Quad Page Program	32h
Quad Page Program 32 bits	34h
Dual Output Fast Read	3Bh
Dual Output Fast Read, 32-bit address	3Ch
Reset enable	66h
Quad Output Fast Read	6Bh
Quad Output Fast Read, 32-bit address	6Ch
Reset	99h
Read JEDEC ID	9Fh
Enter 4 bytes mode	B7h
Chip Erase	C7h/60h
Block Erase	D8h
Block Erase 32 bits	DCh

MEMORY CONFIGURATION:

In the memory module 3DFS256M04VS2801, each 256Mbit bank is divided into uniformly sized 128/64 Kbyte blocks. These blocks are further divided into 8Kbyte sectors. The reason to do this is to enable faster erase cycles when the block size is smaller. However, if the block size is too small, then the die area and the memory cost will increase.

The Memory map is as shown below: -

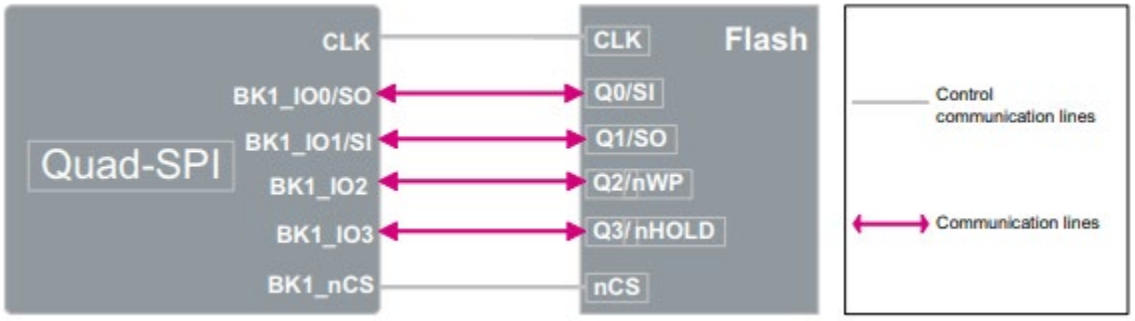
BANK/EMBEDDED DEVICE DENSITY	BLOCK # (128 KBYTE)	BLOCK # (64 KBYTE)	SECTOR #	SECTOR SIZE (KBYTE)	ADDRESS RANGE (HEXA)
256 Mbit	Block 0	Block 0	Sector 0	8	0000 0000-0000 1FFE
			:	:	:
		Block 1	:	:	:
			Sector 15	8	0001 E000-0001 FFFE
	Block 1	Block 2	Sector 16	8	0002 0000-0002 1FFE
			:	:	:
		Block 3	:	:	:
			Sector 31	8	0003 E000-0003 FFFE
	Block 2	Block 4	Sector 32	8	0004 0000-0004 1FFE
			:	:	:
		Block 5	:	:	:
			Sector 47	8	0005 E000-0005 FFFE
	:	:	:	:	:
	Block 63	Block 126	Sector 1008	8	007E 0000-007E 1FFE
			:	:	:
		Block 127	:	:	:
			Sector 1023	8	007F E000-007F FFFE
	:	:	:	:	:

	Block 127	Block 254	Sector 2032	8	00FE 0000-00FE 1FFE
			:	:	:
		Block 255	:	:	:
			Sector 2047	8	00FF E000-00FF FFFE
	:	:	:	:	:
	Block 254	Block 508	Sector 4064	8	01FC 0000-01FC 1FFE
			:	:	:
		Block 509	:	:	:
			Sector 4079	8	01FD E000-01FD FFFE
	Block 255	Block 510	Sector 4080	8	01FE 0000-01FE 1FFE
			:	:	:
		Block 511	:	:	:
			Sector 4095	8	01FFE000-01FF FFFE

Because it's a NOR flash, we will have to erase every sector before we write it.

QSPI:

The memory module 3DFS256M04VS2801 is a QSPI NOR Flash memory. QSPI stands for Queued Serial Peripheral Interface. It is also abbreviated as Quad-SPI.



QSPI is a serial communication interface. It is useful in applications that involve a lot of memory-intensive data like multi-media and on-chip memory is not enough. It can also be used to store code externally and it has the ability to make the external memory behave as fast as the internal memory through some special mechanisms.

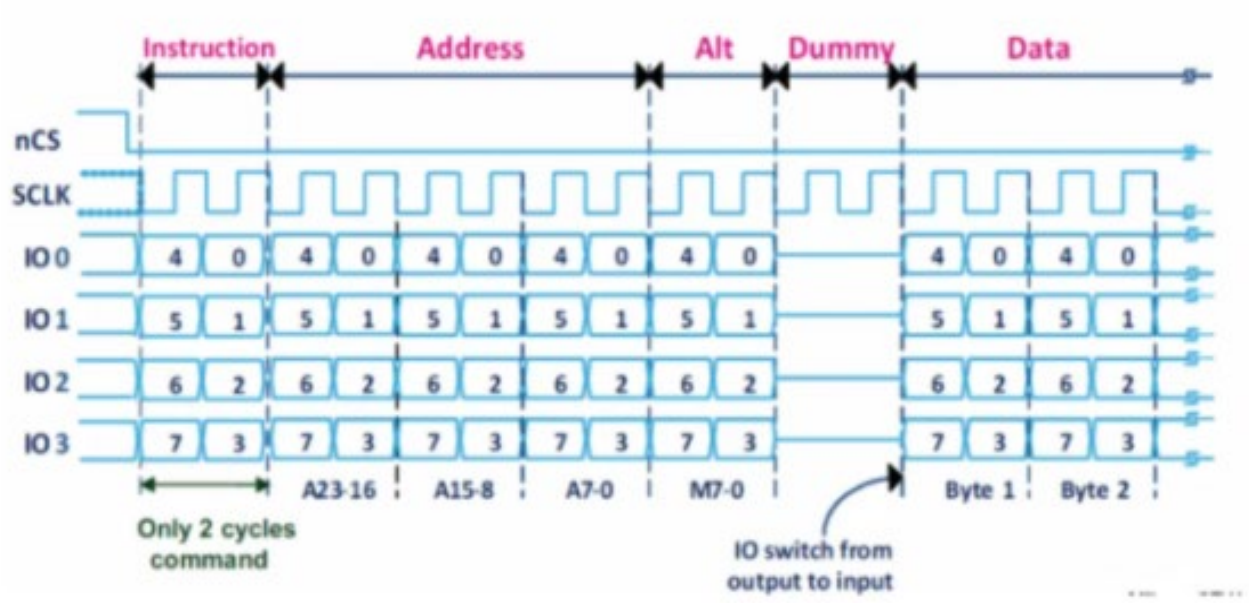
QSPI uses 4 data lines namely; I0, I1, I2 and I3.

Working:

First stage, the instruction is sent over the IO lines followed by the address and the Alt field which can be implemented the way the flash memory wants it to be.

Second stage, for a short period of two clock cycles, the transmission is paused to allow for changing the direction of the I/O line as shown above.

Third stage, the data line is sent from the flash device to the microcontroller. Here, 4 bits are transferred in every clock cycle. The bit order is IO0 sends bit 0, IO1 sends bit 1 and so on in the first clock cycle and bits 4,5,6 and 7 are sent out in the second clock cycle. Thereby transmitting the entire byte in just two clock cycles.



RECOMMENDED OPERATING CONDITIONS:

The operating conditions define the conditions for actual device operation and should be observed to ensure that a device operates properly. Given below are the physical parameters that the chip (3DFS256M04VS2801) can operate within.

CHARACTERISTICS	SYMBOL	MIN	TYP	MAX	UNIT
Supply Voltage	V_{CC}	3.0	3.3	3.6	V
Frequency for Normal Mode	f	-	-	20	MHz
Frequency for Fast Read Mode	f	-	-	50	MHz
Input High Voltage	V_{IH}	0.7 x V_{CC}	-	V_{CC} + 0.3	V
Input Low Voltage	V_{IL}	-0.5	-	0.8	V
Thermal Resistance Junction to Case	$R_{TH(J-C)}$ (Note 1)	-	-	15	°C/W
Thermal Resistance Junction to Ambiant with underfill	$R_{TH(J-A)\_underfill}$ (Note 2)	-	-	27	°C/W

ABSOLUTE MAXIMUM RATINGS:

Maximum ratings are the extreme limits to which the chip can be exposed for a limited amount of time without permanently damaging it. Exposure to absolute maximum ratings for prolonged periods of time may affect the reliability of the device. Given below are maximum ratings of 3DFS256M04VS2801.

CHARACTERISTICS	SYMBOL	MAXIMUM RATINGS	UNIT	REMARKS
Power Supply	V_{CC}	0 to +5.0	V	(Note 2)
Input Voltage with Respect to GND on all Pins	V_T	-0.5 to V_{CC} + 0.5	V	(Note 2)
All Output Voltage with Respect to Ground	-	-0.5 to V_{CC} + 0.5	V	(Note 2)
Storage Temperature Range	T_{STG}	-55 to +150	°C	-
Power Dissipation	P_{DMAX}	0.7	W	-
Maximum Junction Temperature	T_{JMAX}	+150	°C	-
Body Temperature (short exposure only)	T_{BODY}	+215	°C	Measured at module side level (exposure < 60s)

DC PARAMETERS:

The dc parameters represent internal errors that occur because of mismatches between devices and components inside the memory. These errors are always present from the time the power is turned on (i.e., before, during and after any input signal is applied), and they determine how precisely the output matches the ideal memory model. Thus, the precision of the memory is determined by the magnitude of the dc errors.

PARAMETER	SYMBOL	TEST CONDITIONS	MIN	MAX	UNIT
Active Current NORD	I_{CC1}	$V_{CC} = V_{CCMAX} = 3.6$ V f = 20 MHz	-	100	mA
Active Current FAST READ	I_{CC1_FRD}	$V_{CC} = V_{CCMAX} = 3.6$ V f = 50 MHz	-	100	mA
Program Current	I_{CC2}		-	125	mA
Sector Erase Current	I_{CC4}		2	150	mA
Chip Erase Current	I_{CC5}		-	125	mA
V_{CC} Standby Current	I_{SB1}	$V_{CC} = V_{CCMAX} = 3.6$ V PWR_EN_N = 0 V USERy_CE_N = 0.95*V_{CC}	-	10	mA
Deep Power Down Current	$I_{SB\_DeepPower}$	$V_{CC} = V_{CCMAX} = 3.6$ V PWR_EN_N = 0 V USERy_CE_N = 0 V Flash NOR Memories in Deep Power Mode	-	10	mA
Input Leakage Current Low for each input on USERy_BUS except USERy_CE_N	I_{LIL}	$V_{CC} = V_{CCMAX} = 3.6$ V $V_{IN} = 0$ V	-1	+1	µA
Input Leakage Current Low for each USERy_CE_N	I_{LIL}		-50	+50	µA
Input Leakage Current Low for PWR_EN_N	$I_{LIL\_PWR\_EN}$	$V_{CC} = V_{CCMAX} = 3.6$ V $V_{IN} = 0$ V	-50	+50	µA
Input Leakage Current High for each input on USERy_BUS except USERy_CE_N	I_{LIH}	$V_{CC} = V_{CCMAX} = 3.6$ V $V_{IN} = V_{CCMAX}$	-1	+1	µA
Input Leakage Current High for each USERy_CE_N	I_{LIH}		-50	+50	µA
Input Leakage Current High for PWR_EN_N	$I_{LIH\_PWR\_EN}$	$V_{CC} = V_{CCMAX} = 3.6$ V $V_{IN} = V_{CCMAX}$	-4	+4	mA
Output Low Voltage (SO0, SO1, SO2 and SO3) for each USERy_BUS	V_{OL}	$V_{CC} = V_{CCMIN} = 3.0$ V $I_{OL} = 100$ µA	-	0.2	V
Output High Voltage (SO0, SO1, SO2 and SO3) for each USERy_BUS	V_{OH}	$V_{CC} = V_{CCMIN} = 3.0$V $I_{OH} = -100$ µA	V_{CC} – 0.2 V	-	V

AC PARAMETERS:

AC parameters are those speed quantities like access, hold and setup times. They depend on internal processes and indicate the efficiency of a chip. The measured values of these parameters show the timing conditions of a chip which is working correctly. If all AC-parameters of a chip meet the specification, the chip passes the test and can be classified into a group which indicates its speed. Below are the AC parameters of the 3D+ Memory module.

PARAMETER	SYMBOL	TEST CONDITIONS	MIN	MAX	UNIT
Clock frequency for normal read mode	f	--		20	MHz
Clock frequency for all other modes	f	--		50	MHz
Clock rise time	t_{CLCH} (1)		0.1		V/ns
Clock fall time	t_{CHCL} (1)		0.1		V/ns
Clock high time	t_{CKH} (1)		0.45 x 1/fmax		ns
Clock low time	t_{CKL} (1)		0.45 x 1/fmax		ns
USERy_CE_N high time	t_{CEH} (1)		7		ns
USERy_CE_N hold time	t_{CH} (1)		8		ns
Output Valid NOR	t_V	$V_{CC} = V_{CCMIN}$ $V_{IL} = 0$ V $V_{IH} = 0.95 \times V_{CC}$ $V_{OL} = 0.5 \times V_{CC}$ $V_{OH} = 0.5 \times V_{CC}$ f = 20 MHz		23	ns
Output Valid FRD	$t_{V\_fastread}$			8	ns
USERy_CE_N setup time	t_{CS}		6		ns
Output Hold Time	t_{OH}		2	-	ns
Sector Erase Time (8 KByte)	t_{ECS}		-	300	ms
Block Erase Time (64 KByte)	t_{ECB64}			0.5	s
Block Erase Time (128 KByte)	t_{ECB128}			1	s
Page program time	t_{PP}		-	0.8	ms

Output disable time	t_{DIS}	V_{OL} = 0.5 x V_{CC} – 0.2 V V_{OH} = 0.5 x V_{CC} + 0.2 V	-	30	ns
Data in setup time	t_{DS} (1)		2		ns
Data in hold time	t_{DH} (1)		7		ns
Chip Erase Time	t_{EC_2} (1)			90	s
Write status register time	t_W (1)			15	ms
Reset recovery time	t_{RST} (1)			100	µs
Memory readiness time	$t_{Ready\_"0"}$ (1)	PWR_EN_N toggling (1 to 0)		15	ms
	$t_{Ready\_"HZ"}$ (1)	PWR_EN_N in HZ		15	ms
CE_N disabled to power supply off	t_{OFF} (1)		2.5		ms

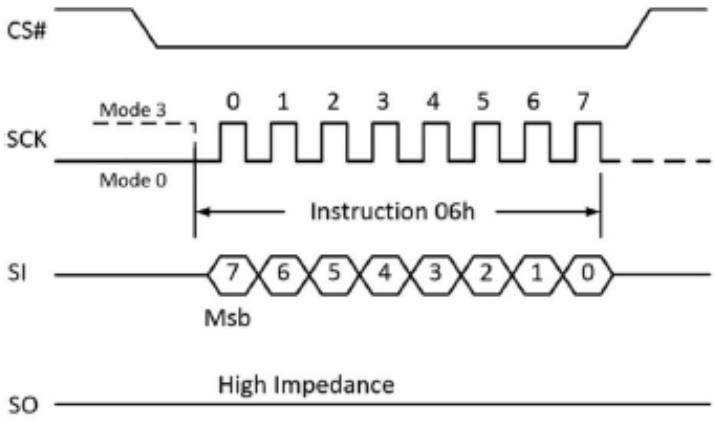
WRITING OPERATIONS TO THE FLASH:

Programming data requires two commands: Write Enable (WREN), and Page Program (PP or QPP). The Page Program command accepts from 1 byte up to 512 consecutive bytes of data (page) to be programmed in one operation. Programming means that bits can either be left at 1 or programmed from 1 to 0. Changing bits from 0 to 1 requires an erase operation.

1. Write Enable

   Opcode: 06h

   The Write Enable (WREN) instruction is used to set the Write Enable Latch (WEL) bit of the Status Register to 1. The WEL bit is reset to the write-protected state after power-up. The WEL bit must be write enabled before any write, program, and erase commands. CE_N must be driven into the logic HIGH state after the eighth bit of the instruction byte has been latched in on SI. Without CE_N (CS#) being driven to the logic HIGH state after the eighth bit of the instruction byte has been latched in on SI, the write enable operation will not be executed. The WEL bit will be reset to the write-protected state automatically upon completion of a write operation. The WREN instruction is required before any above operation is executed.



2. Page Program (02h) & Page Program 32 bits (12h)

   The Page Program (PP) commands allows bytes to be programmed in the memory (changing bits from 1 to 0). Before the Page Program (PP) commands can be accepted by the device, a Write Enable (WREN) command must be issued and decoded by the device.

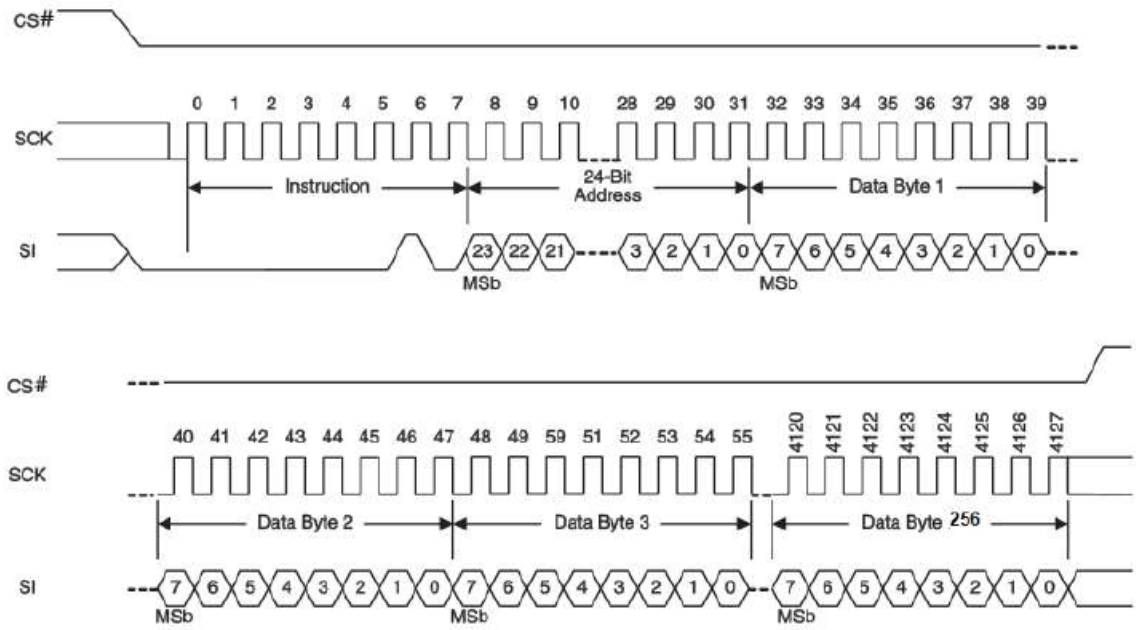
   After the Write Enable (WREN) command has been decoded successfully, the device sets the Write Enable Latch (WEL) in the Status Register to enable any write operations. The instruction

   - 02h is followed by a 3-byte address (A23-A0) or a 4 byte address (A31-A0) when "enter 4 bytes mode" is activated
   - 12h is followed by a 4-byte address (A31-A0)

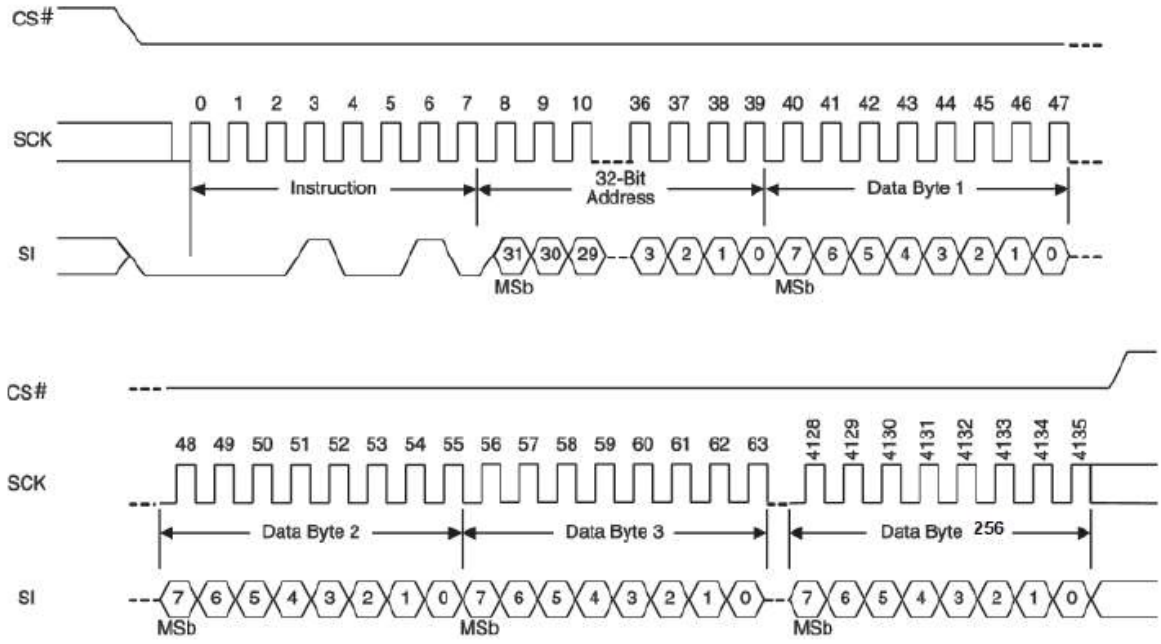
   and at least one data byte on SI. Up to a page can be provided on SI after the 3-byte address with instruction 02h or 4-byte address with instruction 12h has been provided.

   Program operation will start immediately after the CE_N (CS#) is brought high, otherwise the PP instruction will not be executed.

Page Program (02h) Sequence:



Page Program 32 bits (12h) sequence:



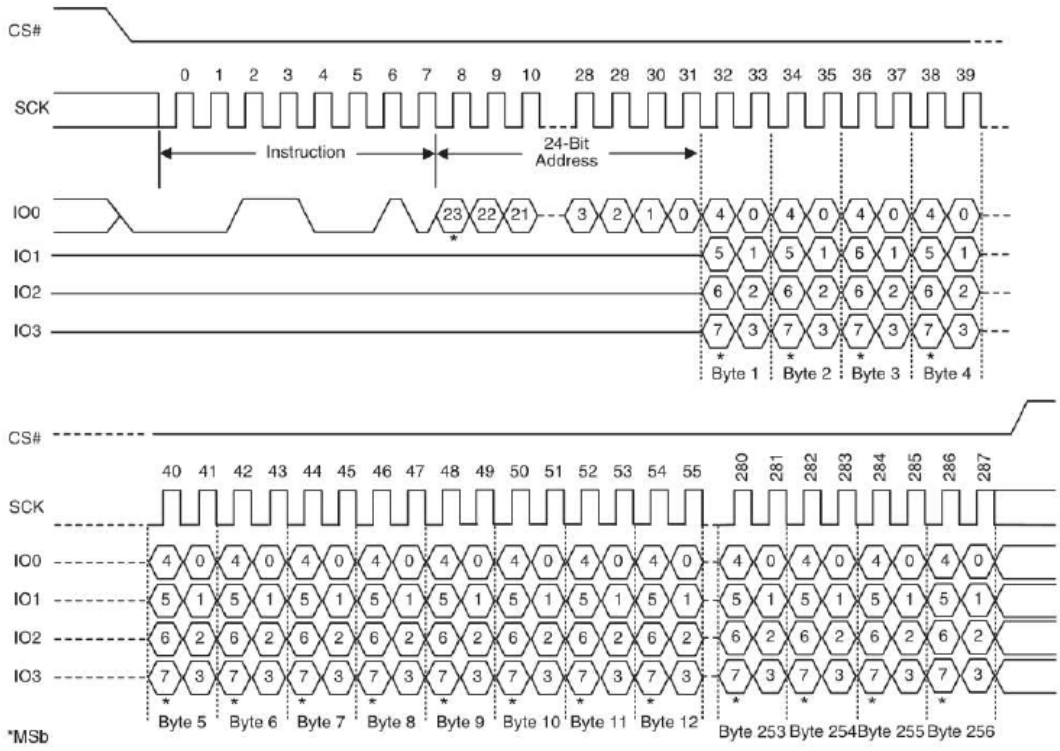
3. Quad Page Program (32h) & QPP 32 bits (34h)

The Quad Page Program (QPP) command allows bytes to be programmed in the memory (changing bits from 1 to 0). The QPP command allows up to a page size of data (512 bytes) to be loaded into the Page Buffer using four signals: IO0-IO3. QPP can improve performance for PROM Programmer and applications that have slower clock speeds (< 12 MHz) by loading 4 bits of data per clock cycle. The QE bit in the Status Register must be set to "1" and a Write Enable command must be executed before the device will accept the QPP command (WEL=1). The instruction

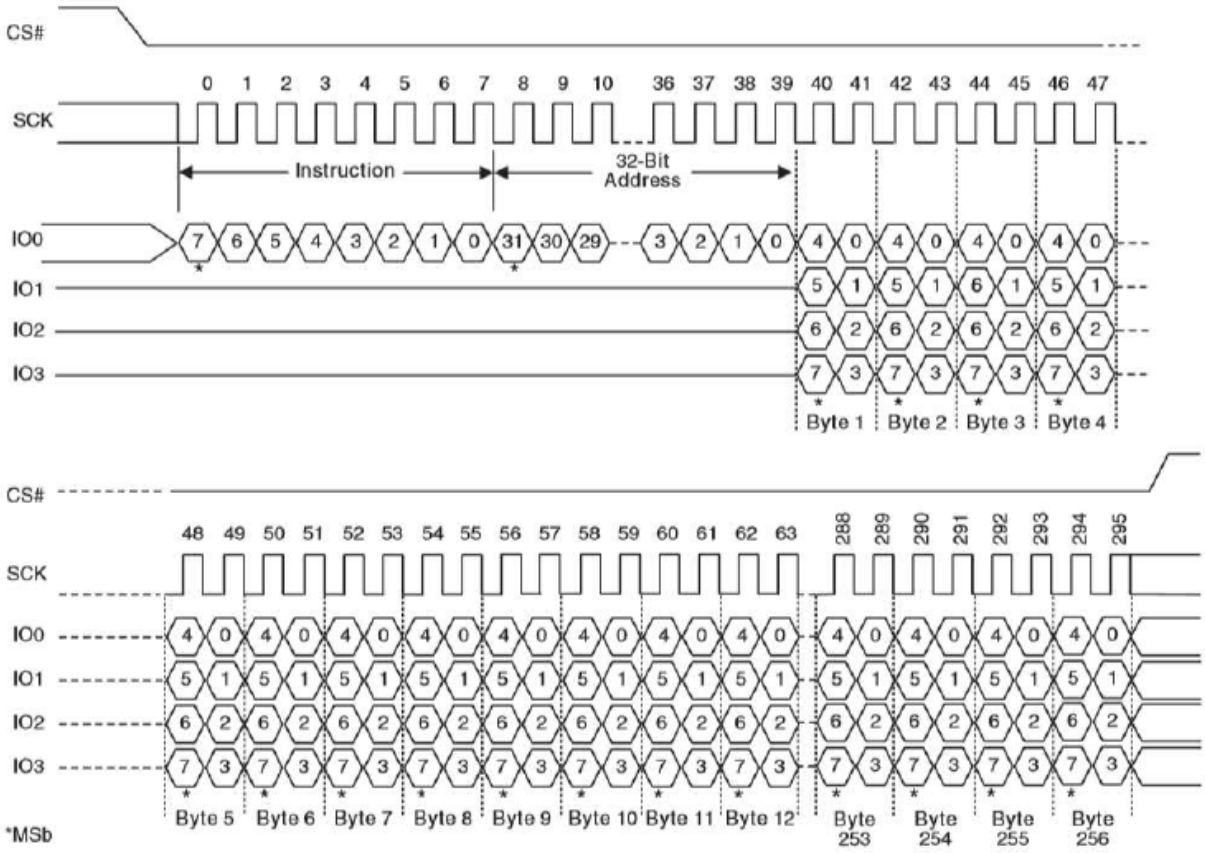
- 32h is followed by a 3-byte address (A23-A0) or a 4-byte address (A31-A0) when "enter 4 bytes mode" is activated.
- 34h is followed by a 4-byte address (A31-A0).

and at least one data byte, into the IO signals. Data must be programmed at previously erased (FFh) memory locations. The programming page is aligned on the page size address boundary. It is possible to program from one bit up to a page size in each Page programming operation.

Quad Page Program (32h) sequence:



QPP 32 bits (34h) sequence:



READING OPERATIONS FROM THE FLASH:

1. READ PAGE (03h) and READ PAGE 32 bits (13h)

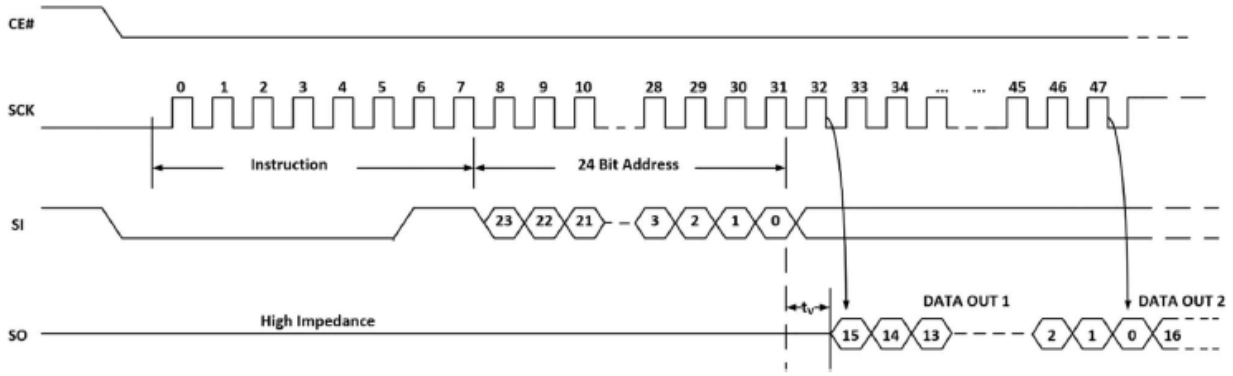
   The Read Page (NORD) instruction is used to read memory contents of the device at a maximum frequency of 20 MHz. The instruction is transmitted via the SI line.

   - 03h is followed by a 3-byte address (A23-A0), or a 4-byte address (A31-A0) when "enter 4 bytes mode" is activated.
   - 13h is followed by a 4-byte address (A31-A0).

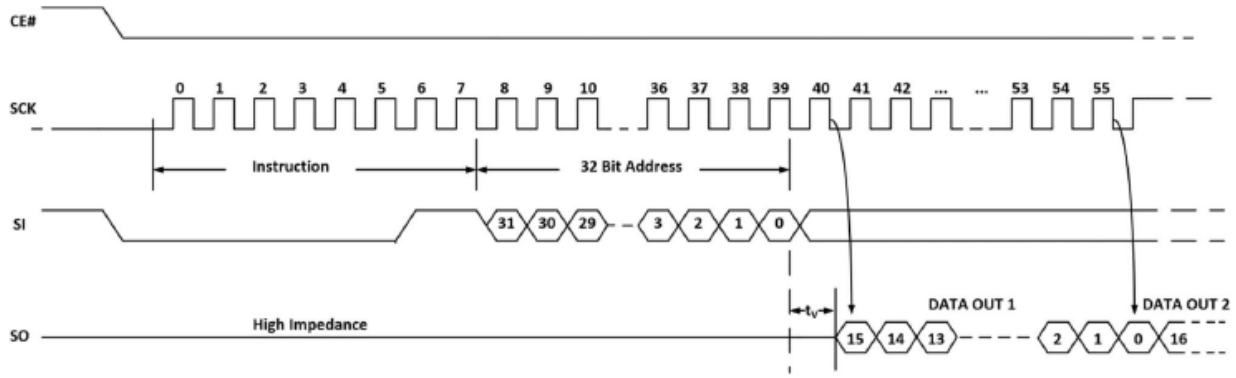
   Then the memory contents, at the address given, are shifted out on SO. The address can start at any even address location of the memory array. The address is automatically incremented to the next higher address in sequential order after each byte of data is shifted out. The entire memory can therefore be read out with one single read instruction and address 000000h provided. When the highest address is reached, the address counter will wrap around and roll back to 000000h, allowing the read sequence to be continued indefinitely.

The read operation can be terminated at any time by driving CE_N (CS#) high (VIH) after the data comes out. It is prohibited to issue a Read Page instruction while an Erase, Program or Write operation is in process (BUSY = 1).

READ PAGE (03h) sequence:



READ PAGE 32 bits (13h) sequence:



2. QUAD OUTPUT FAST READ (6Bh) & QOFR 32 bits (6Ch)

The Quad Output Fast Read (FRQO) instruction is used to read memory data on four output pins each at up to a 50 MHz clock. A Quad Enable (QE) bit of Status Register must be set to "1" before sending the Fast Read Quad Output instruction. The instruction –

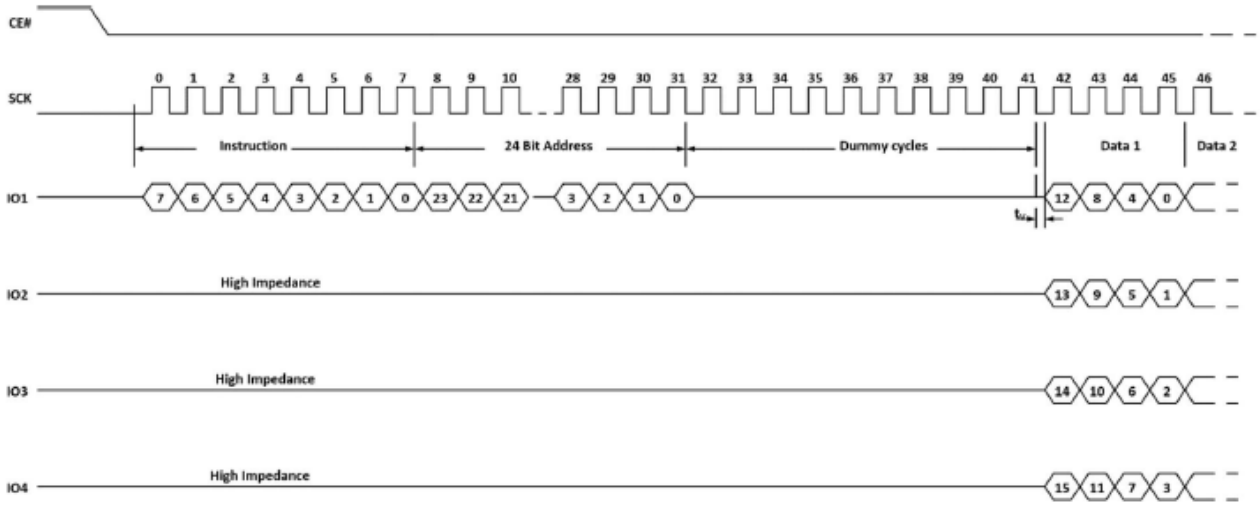
- 6Bh is followed by a 3-byte address (A23-A0) or a 4-byte address (A31-A0) when “enter 4 bytes mode” is activated
- 6Ch is followed by a 4-byte address (A31-A0).

Then the memory contents, at the address given, is shifted out four bits at a time through IO0-IO3. Each nibble (4 bits) is shifted out at the SCK frequency by the falling edge of the

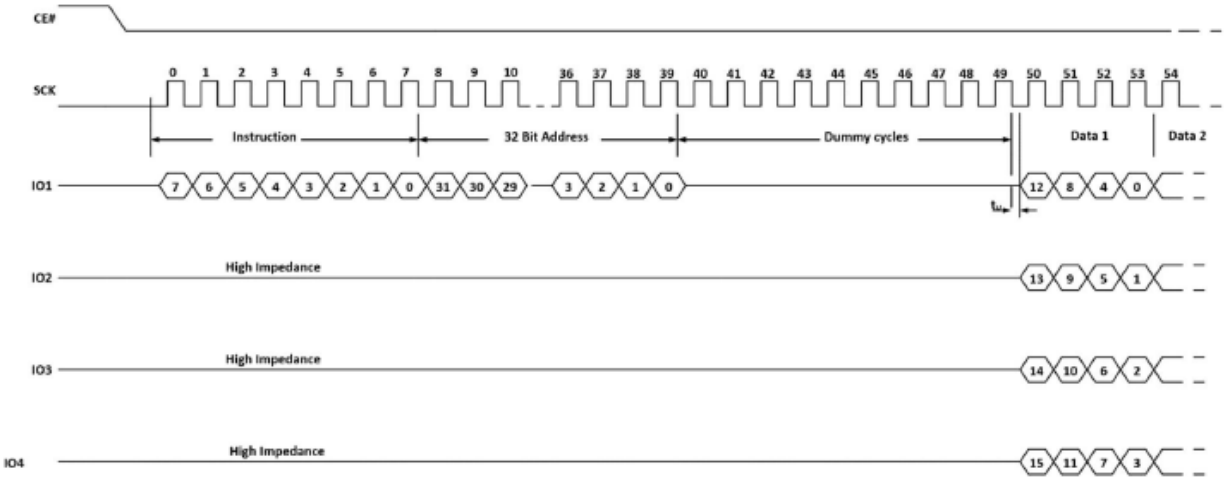
SCK signal. For Quad Output Read Mode, there are ten dummy cycles required after the last address bit is shifted into SI before data begins shifting out of IO0-IO3. This latency period (i.e., dummy cycles) allows the device's internal circuitry enough time to set up for the initial address. The entire memory can therefore be read out with one single read instruction and address 000000h provided. When the highest address is reached, the address counter will wrap around and roll back to 000000h, allowing the read sequence to be continued indefinitely.

FRQO instruction is terminated by driving CE_N (CE#) high (VIH).

QUAD OUTPUT FAST READ (6Bh) sequence:



QOFR 32 bits (6Ch) sequence:



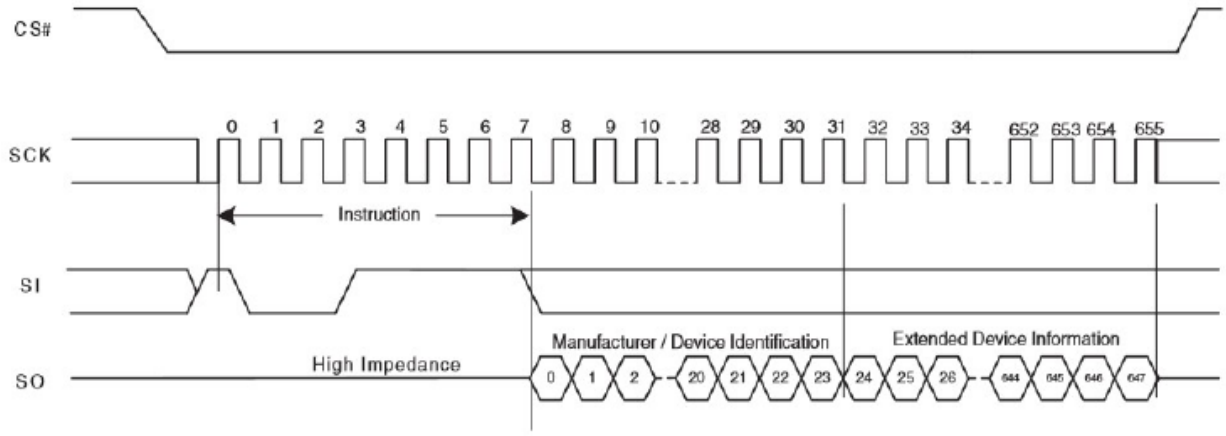
3. READ JEDEC ID

The Read JEDEC ID (RDID) instruction allows the user to read the Manufacturer and product ID of devices. It is prohibited to issue a RDID command while a program, erase, or write cycle is in progress.

MODULE	DENSITY	JEDEC ID
3DFS256M04VS2801	256 Mbit	0x9D6019

Product Identification Table

Sequence:

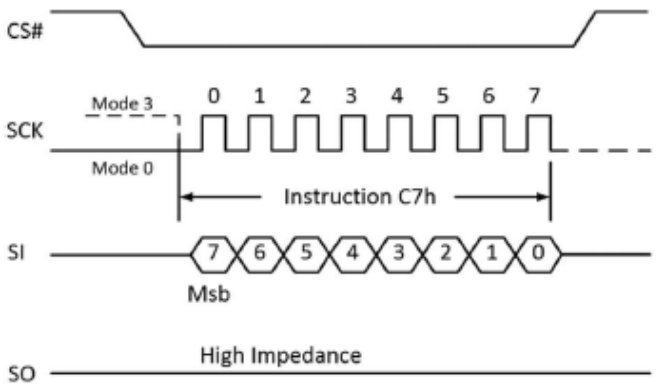


ERASE OPERATIONS OF THE FLASH:

In NOR flash, we will have to erase every sector before we write it. What happens in Erase is that, all of the sector become 1s and when you are writing its just like an XOR operation being performed on each cell.

1. CHIP ERASE (C7h/ 60h)

A Chip Erase (CER) instruction erases the entire memory array (all bytes are FFh). Before the execution of CER instruction, the Write Enable Latch (WEL) must be set via a Write Enable (WREN) instruction. The WEL is automatically reset after completion of a chip erase operation. The CER instruction code is input via the SI. CE_N (CS#) must be driven into the logic HIGH state after the eighth bit of the instruction byte has been latched in on SI. This will initiate the erase cycle.



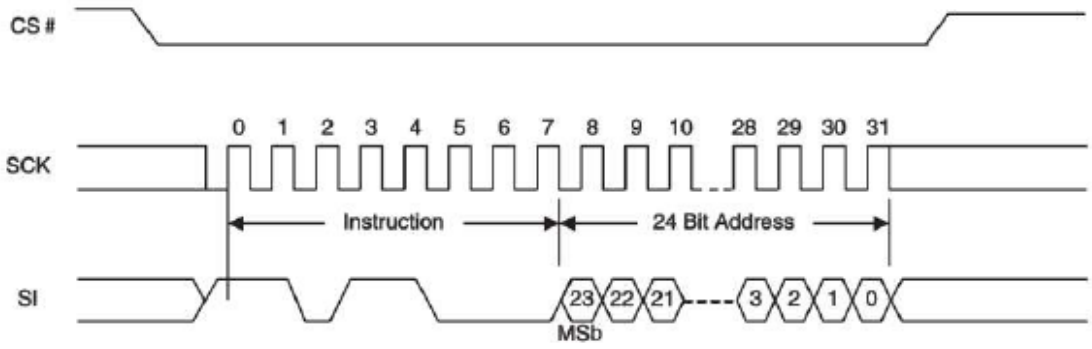
2. Block Erase (D8h) and Block Erase 32 bits (DCh)

A Block Erase (BER) instruction erases a 128 Kbyte block. Before the execution of a BER instruction, the Write Enable Latch (WEL) must be set via a Write Enable (WREN) instruction. The WEL is reset automatically after the completion of a block erase operation. The instruction

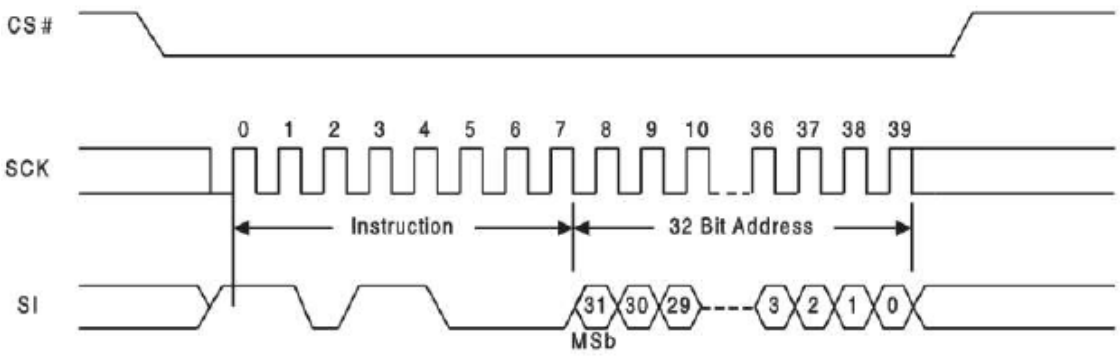
- D8h is followed by a 3-byte address (A23-A0), or a 4-byte address (A31-A0) when "enter 4 bytes mode" is activated
- DCh is followed by a 4-byte address (A31-A0).

CE_N (CS#) must be driven into the logic HIGH state after the twenty-fourth or thirty-second bit of address has been latched in on SI. This will initiate the erase cycle

Block Erase sequence



Block Erase 32 bits (DCh) sequence



CHAPTER - 4

F P G A D E S I G N & V H D L

Field Programmable Gate Array (FPGA) is an integrated circuit that consists of internal hardware blocks with user-programmable interconnects to customize operation for a specific application. The interconnects can readily be reprogrammed, allowing an FPGA to accommodate changes to a design or even support a new application during the lifetime of the part.

A basic FPGA architecture consists of thousands of fundamental elements called configurable logic blocks (CLBs) surrounded by a system of programmable interconnects, called a fabric, that routes signals between CLBs. Input/output (I/O) blocks interface between the FPGA and external devices.

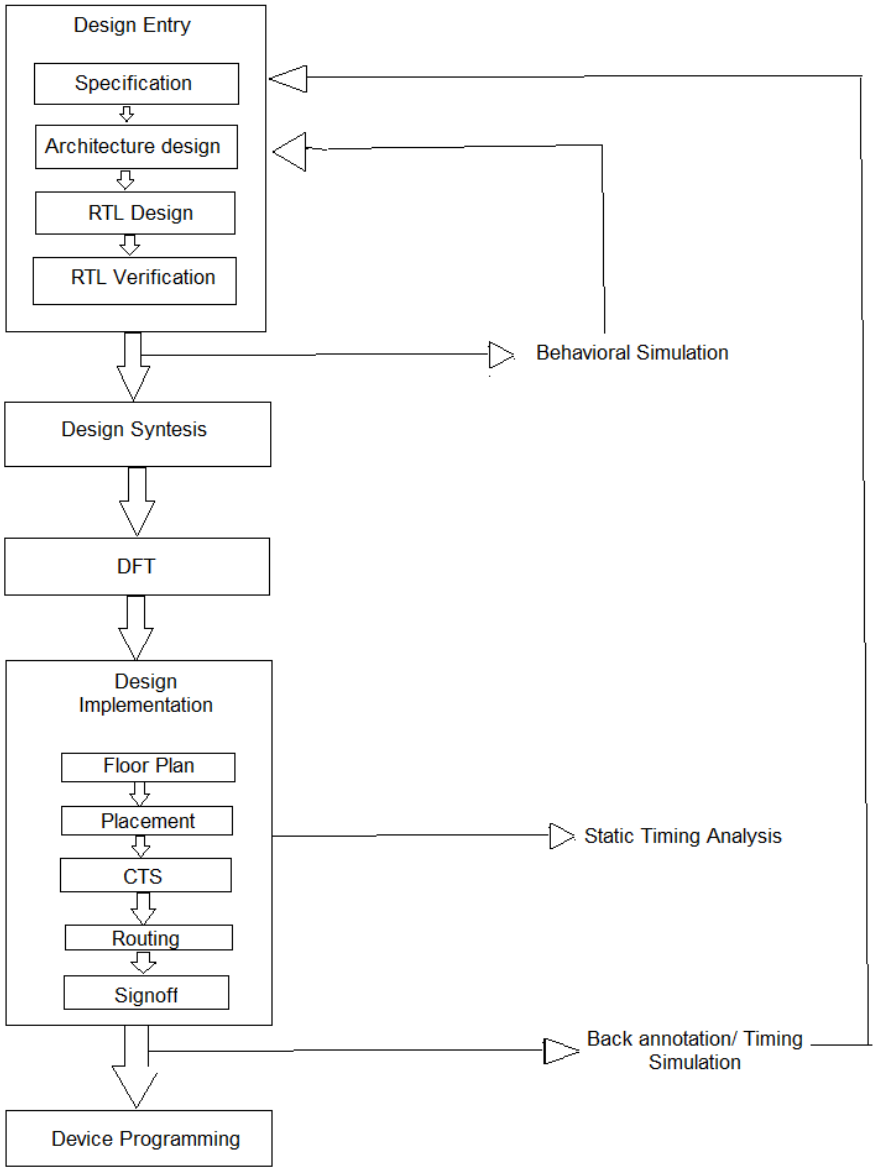
An FPGA-based design begins by defining the required computing tasks in the development tool, then compiling them into a configuration file that contains information on how to hook up the CLBs and other modules. The process is similar to a software development cycle except that the goal is to architect the hardware itself rather than a set of instructions to run on a predefined hardware platform. For this a hardware description language (HDL) such as VHDL (where V stands for VHSIC, Very High-Speed Integrated Circuit) or Verilog is used to design the FPGA configuration.

Once the FPGA design has been created and verified using HDL, the compiler takes the text-based file and generates a configuration file that contains information on how the components should be wired together. One challenge with an HDL approach is that configuring an FPGA requires both coding skills and a detailed knowledge of the underlying hardware.

Choosing an FPGA: Following criteria are considered when choosing a FPGA –

1. Speed of FPGA
2. Size of the FPGA
3. Number of Available IO's (Differential and Single Ended)
4. Number of Internal memories (RAMs)
5. DSP Slices (Complex Multipliers)
6. Connectivity Ports (Serial Interfaces, Memory Interfaces)

Below is a Flowchart showing FPGA Design flow –



System specification: In this stage, the features information which is being expected from the customer is collected.

Architecture Design: With the help of the specification sheet the target IC's architecture is decided and a layout for same is created.

RTL Design: RTL stands for Register Transfer Level. In this stage, the above designed architecture is converted into Verilog or VHDL code which describes how data is transformed as it is passed from register to register.

RTL Verification: After the RTL design step, by applying test cases, now we verify the design for logical correctness and ensure no major timing errors occurs. A test bench file may be used here for verification.

Design Synthesis: It is the process of converting the RTL code into a gate-level netlist (or, a list of wires). Steps performed in synthesis are –

1) The RTL code is converted into Boolean expression.
2) The Boolean expression is optimized by SOP and POS optimization methods.
3) The gate-level netlist is generated.
4) Synthesis tools ensure that the gate-level netlist meets timing, area, and power specifications.

A gate-level netlist is a description of entire chip in terms of logic gates and their interconnections.

DFT: Design for testability is a technique which facilitates a design to become testable after production. It consists of IC design techniques that add testability features to a hardware product design. The added features make it easier to develop and apply manufacturing tests to the designed hardware. The purpose of manufacturing tests is to validate that the product hardware contains no manufacturing defects that could adversely affect the product's correct functioning.
This is very different from functional testing, which attempts to validate that the circuit under test functions according to its functional specification.

Floorplan: Here we will specify the size of the die/block and creates wire tracks for placement of standard cells and we will determine the placement of macros, physical cells (tap, filler, end cap) and pre-placement cells like spare cells, power straps and i/o placement. we will also specify the i/o, pin/pad placement information.

Placement: In Placement stage based on user preference as timing aware, congestion aware, clock gate aware PNR tool will place all the standard cells in the design. Placement will be done in two steps.

Global placement: - All the standard cells will be placed inside the core area roughly.

Detailed placement: - In this stage, all the standard cells will be placed in site rows (legalize placement).

In the placement stage, we check the congestion value by GRC map.

Clock Tree Synthesis: In this stage, we will build the clock tree by using inverters and buffers. In the chip clock signal is essential to the flip flops, to give the clock signal from clock source with minimum skew, we will build the clock tree structure. It is the process of balancing the clock skew and minimizing insertion delay to meet timing.

Routing: Before the routing stage, PNR tool will perform trail routing. In trail, the route tool will use the same track for two (or) more wires which we call as congestion. In this stage, the PNR tool we perform detail routing. Routing is divided into two parts -

1)global routing

2)detailed routing.

In global routing, the tool will divide the entire core into tiles and will assign tracks. Before the detailed routing, all are the logical connections. In detailed routing, the physical connections are done.

Sign off: After the routing, the physical layout of the chip is completed. In signoff stage analysis that are done are -

1) STA
2) Physical verification
3) IR drop analysis

TIMING ANALYSIS:

Timing Analysis is a methodical analysis of a digital circuit to determine if the timing constraints imposed by components or interfaces are met. Timing Analysis is an integral part of the Design Flow. Anything else can be compromised but not Timing It is helpful when –

1. We want to verify whether our circuit meets all its timing requirements.
2. During designing there is a trade-off between speed, area, power, and runtime according to the constraints set by the designer (There are 3 types of design constraints – Timing, Power, Area). However, a chip must meet the timing constraints to operate at the intended clock rate, so timing is the most important design constraint.
3. We want to make sure that circuit is properly designed and can work properly for all combinations of components over the entire specified operating environment, every time.

There are 2 types of Timing Analysis

1. Static Timing Analysis
2. Dynamic Timing Analysis

Static Timing Analysis:

Static timing analysis is a method of validating the timing performance of a design by checking all possible paths for timing violations under worst-case conditions. It checks static delay requirements of the circuit without any input or output vectors. STA considers the worst possible delay through each logic element, but not the logical operation of the circuit.

In comparison to circuit simulation, static timing analysis is –

a) Faster - It is faster because it does not need to simulate multiple test vectors.
b) More Thorough - It is more thorough because it checks the worst-case timing for all possible logic conditions, not just those sensitized by a particular set of test vectors.

In static timing analysis, the word static alludes to the fact that this timing analysis is carried out in an input-independent manner. It locates the worst-case delay of the circuit over all possible input combinations. There are huge numbers of logic paths inside a chip of complex design. The advantage of STA is that it performs timing analysis on all possible paths (whether they are real or potential false paths). However, it is worth noting that STA is not suitable for all design styles. It has proven efficient only for fully synchronous designs. Since most of the chip design is synchronous, it has become a mainstay of chip design over the last few decades.

The Way STA is performed on a given circuit:

- Design is broken down into sets of timing paths.
- Calculates the signal propagation delay along each path
- And checks for violations of timing constraints inside the design and at the input/output interface.

The STA tool analyzes ALL paths from each and every startpoint to each and every endpoint and compares it against the constraint that (should) exist for that path. All paths should be constrained, most paths are constrained by the definition of the period of the clock, and the timing characteristics of the primary inputs and outputs of the circuit. Further, there are different types of timing paths-

- Data Path
- Clock Path
- Clock Gating path
- Asynchronous path

Dynamic Timing Analysis:

Dynamic timing analysis verifies circuit timing by applying test vectors to the circuit. This approach is an extension of simulation and ensures that circuit timing is tested in its functional context. This method reports timing errors that functionally exist in the circuit and avoids reporting errors that occur in unused circuit paths. Dynamic timing analysis has to be accomplished and functionality of the design must be cleared before the design is subjected to Static Timing Analysis (STA). Dynamic Timing Analysis (DTA) and STA are not alternatives to each other. Quality of the DTA increases with the increase of input test

vectors. Increased test vectors increase simulation time. Dynamic timing analysis can be used for synchronous as well as asynchronous designs. One such dynamic timing analysis is the min-max analysis method. However, Min-max simulation is not currently used in the industry. Instead, either functional simulation with timing (timing simulation) or formal verification method is typically used to verify complex IC designs

A major issue with dynamic timing analysis is the incomplete coverage. It may only check circuitry that is exercised by test stimulus, which may leave critical paths untested, and timing problems undiscovered. It is also not path oriented. Since DTA reports errors on a certain pin at a certain time, the user must trace through the schematic to locate the path that caused the problem (difficult for large designs).

Advantages:
1. Extends coverage of circuit simulation (edges to region).
2. Evaluates worst-case timing using both min. and max. delay values for components.
3. Uses the same test stimulus as logic simulation.
4. Does not report false errors.

Disadvantages:
1. It is not complete.
2. It is not path oriented.
3. It is slower than logic simulation and may require additional test stimulus.
4. It requires functional behavioral models.

Example of DTA tools are Modelsim (from mentor Graphics), VCS (from Synopsys). DTA is also carried out on post layout netlist to verify that functionality of the design has not changed. Test vectors remain same for both.

BACK ANNOTATION:

This term is in general used in connection to netlist simulations and STA where the propagation delay(s) through each cell in the netlist is overridden by the delay value(s) specified in a special file called sdf (synopsys delay format) file. The process of putting delays from a given source for the cells in a netlist during netlist simulation is called Back Annotation. Normally the values of the delays corresponding to each cell in the netlist would come from the simulation library, i.e., VHDL model of library cells. But those delays are not the actual delays of cells, as each of them is instantiated in a netlist in different surroundings, different physical locations, different loads, different fan in. The delay of two similar cells in the netlist at two different physical locations in a chip can be significantly different depending upon above said factors. Therefore in order to have actual delays for the cells in your netlist, an SDF is written out, by a EDA tool can be a synthesis

tool or a layout tool etc. which contains the delays of each instance of each library cell in the netlist, under the circumstances the cell is in.

During simulations or Static Timing Analysis, each cell in the netlist gets its corresponding delay read, or more technically 'annotated' from the SDF file. SDF file contains the delay value of each timing arc corresponding to each cell in the netlist. These delay values in the SDF file are extracted under a given conditions of the netlist. It may be noted that the SDF corresponds to just an after synthesis netlist, with wire loads estimated according to some wire load model, or it may be that the SDF corresponds to a netlist which has been laid out, with actual position of cell, actual load on the cell, actual metal wires connected to the cells.

VHDL:

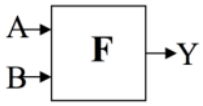
Short for **VHSIC** (Very High-Speed Integrated Circuit) **Hardware Description Language**. It is a programming language used to describe, simulate and to create integrated circuits. It is mainly used to discover the faults in the design (modelling and simulating) before implementing it in the hardware (gates and wires). VHDL is an IEEE standard, powerful language with numerous language constructs that are capable of describing very complex behaviour.

Features of VHDL:

1) Design entry can be done in coded form, schematic and state diagram form.
2) It is a case-insensitive language.
3) VHDL permits concurrent and sequential statements. All statements in the architecture are concurrent. A process statement is a concurrent statement meaning that A number of processes may run at the same simulated time. Sequential statements are inside process, specifying the step-by-step behavior of the process.
4) It is an event driven language, i.e., whenever an event occurs on a signal, all statements that have a reference to that signal will get executed.
5) VHDL is a platform independent language
6) Supports both synchronous and asynchronous timing models.
7) It supports both predefined and user defined data types
8) Supports all CAD tools.
9) Supports code sharing and code reusability through the use of packages, user defined libraries, functions and procedures.
10) In VHDL, design verification can be done at 3 levels: compilation and simulation, synthesis and post synthesis level.
11) A digital system can be modelled using concurrent (data flow), behavioural (sequential) and structural modelling styles.

Basic elements of VHDL:

1) Entity: used to specify the input and output ports of the circuit. An Entity usually has one or more ports that can be inputs, outputs, input-outputs or buffer.




```
entity F is
 generic (N: natural := 8);
 port (A, B: in bit_vector (N-1 downto 0); Y: out bit);
end F;
```

Fig: Example entity declaration

2) Architecture: An architecture is the actual description of the design, which is used to describe how the circuit operates by including a set of inner signals, functions, procedures, functions, etc.

VHDL Objects:

1) Constants: Objects that have an initial value that is assigned before the simulation. This value shall never be modified during the synthesis or the operation of the circuit. They can be declared before the begin of an architecture, and/or before the begin of a process.

2) Variables: Objects that take a single value that can change during the simulation/execution by means of an assignment statement.
3) Signal: Objects that represent memory elements or connections between subcircuits. Contrasting to constants and variables, signals can be synthesized, i.e., the signal in the source code can be physically translated into a memory element (flip-flop, register, etc.) in the final circuit.

RT4G150-ES FPGA FEATURES:

The device features reprogrammable flash configuration, and offers reprogrammable flash technology for immunity to radiation-induced configuration upsets in radiation environments. RT4G150-ES supports space applications requiring up to 150,000 logic elements and up to 300 MHz of system performance.

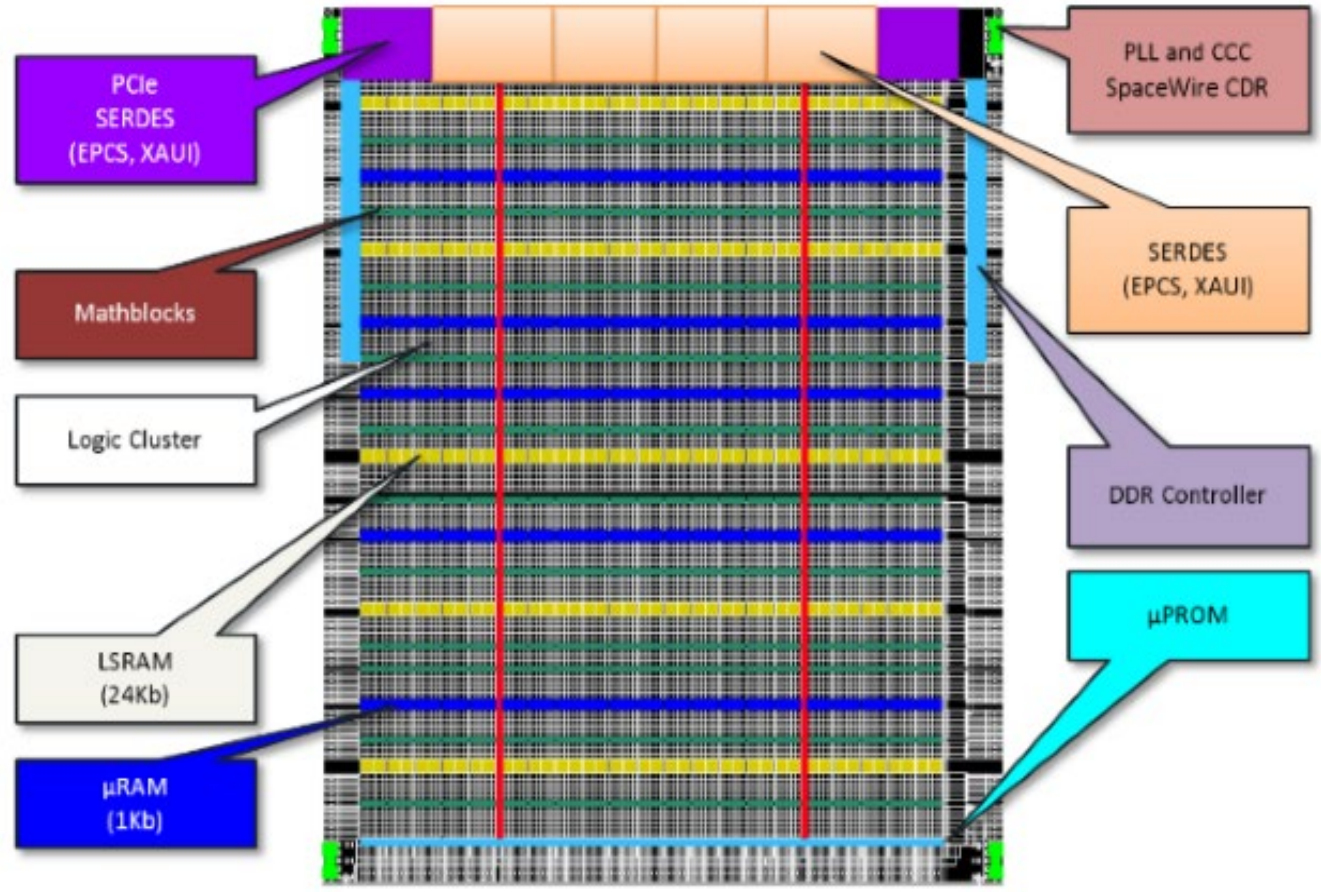
Uses –

Typical uses for RT4G150-ES include remote sensing space payloads like radar, imaging and spectrometry in civilian, scientific and commercial applications for weather forecasting and climate research, land use, astronomy and astrophysics, planetary exploration, and Earth sciences.

Features-

Includes 24 serial transceivers, with operation from 1 to 3.125 gigabits per second; 16 SEU- and SET-protected SpaceWire clock and data recovery circuits; 462 SEU- and SET-protected multiply-accumulate math blocks; more than 5 megabits of on-board SEU-protected SRAM; ingle event latch-up (SEL) and configuration memory upset immunity; and total ionizing dose (TID) beyond 100 kilorads.

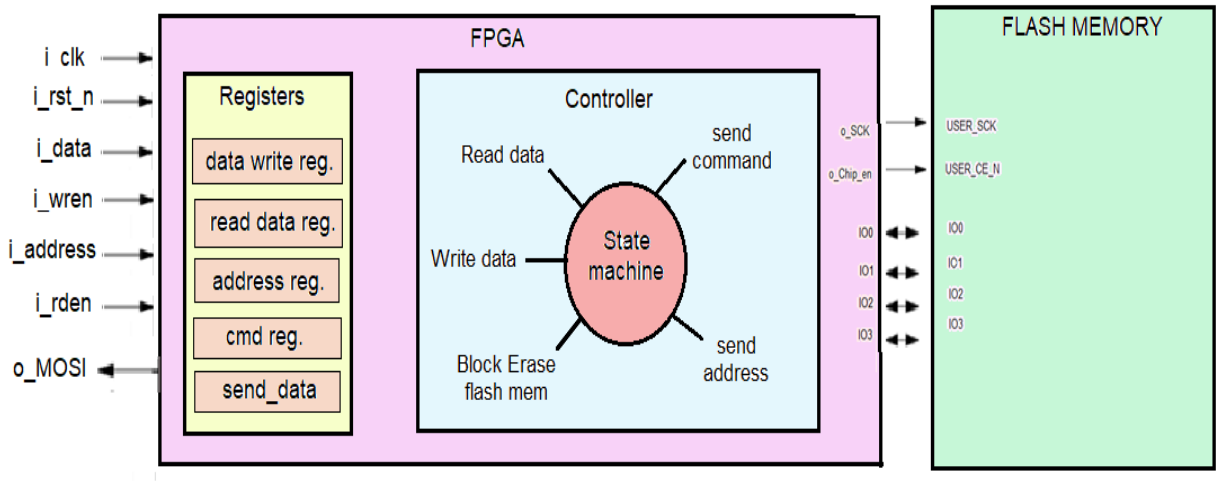
Its Architecture is shown below:



CHAPTER - 5

N O R F L A S H C O N T R O L L E R D E S I G N

When designing the Controller, we will follow the principle "Divide and Conquer" in order to use a modular design that consists of smaller, more manageable blocks, some of which can be re-used. The figure below shows the different blocks (modules) that make up the memory controller. Function of each block is explained further.



Registers Module:

The Registers module contains 5 registers as shown above. The ‘data write register’ stores data values input by the user, before writing it to the memory. Similarly, address and command registers store the address and command resp., input by the user. The ‘read data register’ will store the data read from the memory by the controller, before displaying to the user through o_MOSI line. The send data register is just for signaling to start sending the data to memory.

REGISTER NAMES	REG. ADDRESS	DESCRIPTION	INITIAL VALUES
Data write reg.	0x00	Bits (31:0) – data write to the memory	0x00000000
Read data reg.	0x04	Bits (31:0) – Data reads from the memory	0x00000000
Address reg.	0x08	Bits (23:0) – stores address of flash mem.	0x00000000

Command reg.	0x10	Bits (7:0) – command data	0x00000000
Send data reg.	0x14	Bit 0 – ‘1’ Send data to flash mem.	0x00000000

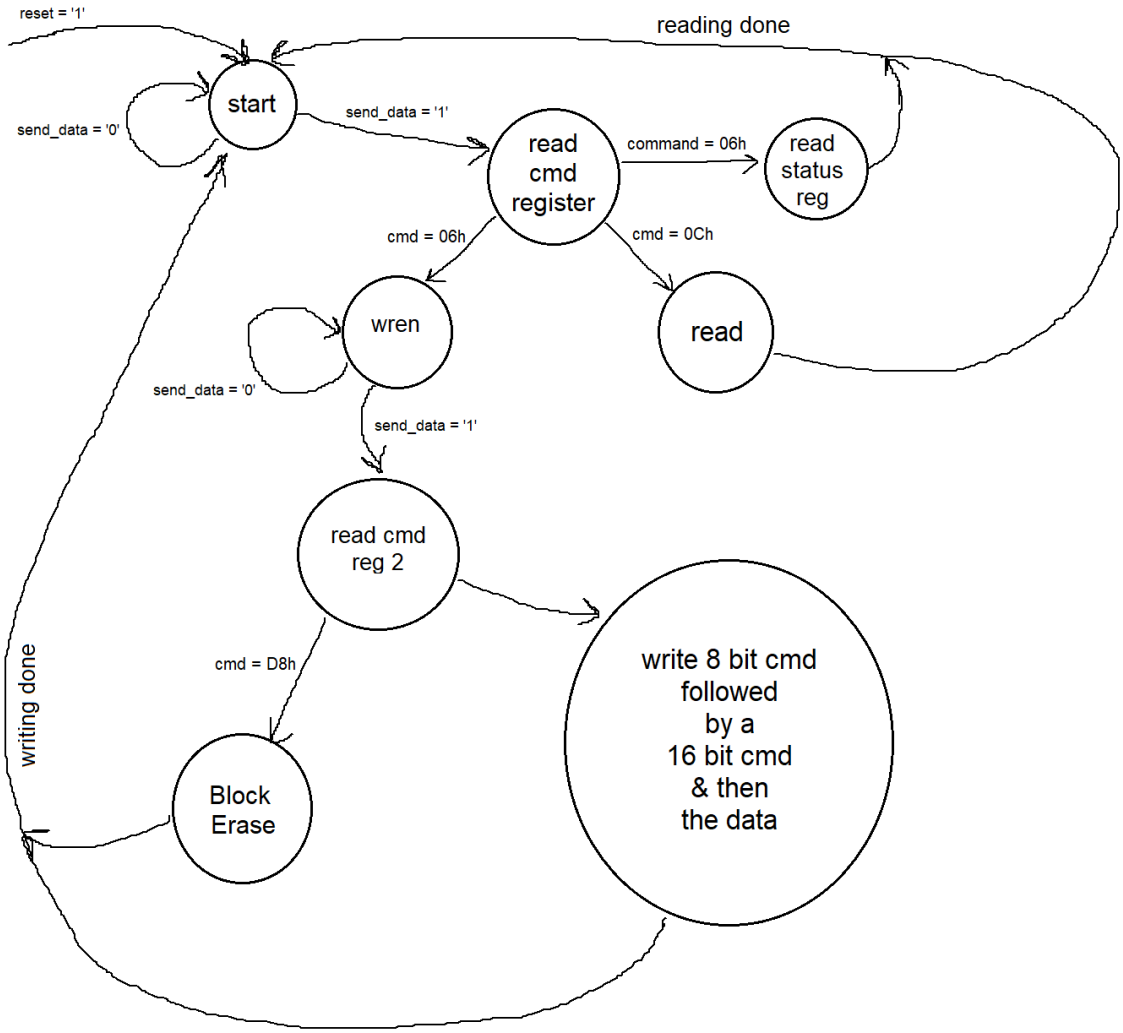
VHDL Code of ‘Registers’ module:

```
----------------------------------------------------------------------------------------------------
-- Company                      : Space Applications Centre, ISRO
-- Engineer                     : Adhiraj Roy Chowdhury
-- Create Date                  : 16:35:00 10/05/2022
-- Target Devices               : FPGA:RT4G150_ES, Controller for 3DFS256M04VS2801 NOR Flash memory

---- Registers
----------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity Registers_file is
        port(
                clock          :in std_logic;
                reset          :in std_logic;
                i_data         :in  std_logic_vector(31 downto 0);
                o_MOSI         :out std_logic_vector(31 downto 0);
                i_address      :in  std_logic_vector(23  downto 0);
                i_wren         :in  std_logic;
                i_rden         :in  std_logic;
                data_write     :out std_logic_vector(31 downto 0);
                data_read      :in  std_logic_vector(31 downto 0);
                address_data   :out std_logic_vector(31 downto 0);
                command_data   :out std_logic_vector(31 downto 0);
                send_data      :out std_logic_vector(31 downto 0)

        );
end Registers_file;

architecture registers_file_arch of registers_file is

signal data_write_sig      :std_logic_vector(31 downto 0);
signal data_read_sig       :std_logic_vector(31 downto 0);
signal address_data_sig    :std_logic_vector(31 downto 0);
signal command_data_sig    :std_logic_vector(31 downto 0);
signal send_data_sig       :std_logic_vector(31 downto 0);

begin
```

```
data_write <= data_write_sig;
data_read_sig <= data_read;
address_data <= address_data_sig;
command_data <= command_data_sig;
send_data <= send_data_sig;

process(reset,clock,i_rden,data_write_sig,data_read_sig,address_data_sig,command_data_sig,send_data_sig) --read
registers
begin
   if(reset = '0') then
            o_MOSI <= (others=>'0');
         elsif(rising_edge(clock)) then
            if(i_rden = '1') then
                  case i_address is
                              when x"000000" => o_MOSI <= data_write_sig  ;
                              when x"000004" => o_MOSI <= data_read_sig   ;
                              when x"000008" => o_MOSI <= address_data_sig;
                              when x"00000C" => o_MOSI <= command_data_sig;
                              when x"000010" => o_MOSI <= send_data_sig   ;

                              when others=> o_MOSI <= x"12345678";
                        end case;
                  end if;
         end if;
end process;

process(reset,clock,i_wren) --write registers
begin
   if(reset = '0') then
            data_write_sig   <= (others=>'0');
            address_data_sig <= (others=>'0');
            command_data_sig <= (others=>'0');
            send_data_sig    <= (others=>'0');


         elsif(rising_edge(clock)) then
            if(i_wren = '1') then
                  case i_address is
                              when x"000000" => data_write_sig   <= i_data;
                              when x"000008" => address_data_sig <= i_data;
                              when x"00000C" => command_data_sig <= i_data;
                              when x"000010" => send_data_sig    <= i_data;

                              when others=> null;
                        end case;
                  end if;
         end if;
end process;
end registers_file_arch;
```

Controller Module:

The Controller module is described by a state machine, which takes care of all the processes: Read, Write, Erase, etc.



MAIN STATE MACHINE

The state machine describing the different processes is shown above. A State Machine, also known as Finite State Machine consists of a set of states, a start state, an input, and a transition function that maps input and current states to a next state. The Machine begins in the start state with an input. It changes to new states depending on the transition function. The transition function depends on current states and inputs. The output of the machine depends on input and/or current state.

The various SPI commands supported by this controller are given in the table below:

COMMAND	CMD. DESCRIPTION
12h	Page Program
34h	Quad Page Program
0Bh	Read Page
6Bh	Quad Output Fast Read
99h	Reset
D8h	Block Erase
06h	Write Enable
05h	Read Status register

The frame format for each command shall be –

Command encoding bits	Address bits	Dummy Cycles	Data bits

In reference to above table, there are address, dummy and data cycles for each command. These address and data cycles, other associated signals are generated at the output of the controller. Which address – data cycles are generated for which commands is specified in data sheet and also discussed in Chapter 3. According to which these address – data cycles need to be generated, then and then only the flash memory will go into the desired mode, otherwise not.

The dummy cycles allow the device internal circuits additional time for accessing the initial address location. During the dummy cycles the data value on SO (Serial Out pin) is “don’t care” and must be high impedance. Then the memory contents, at the address given, are shifted out on SO.

VHDL Code for Controller Module:

```
----------------------------------------------------------------------------------------------------
-- Company                      : Space Applications Centre, ISRO
-- Engineer                     : Adhiraj Roy Chowdhury
-- Create Date                  : 16:35:00 10/05/2022
-- Target Devices               : FPGA:RT4G150_ES, Controller for 3DFS256M04VS2801 NOR Flash memory

-- Controller
----------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity Qspi_controller is
```

```
        port(
                clock           :in  std_logic;                 --100MHZ
                reset           :in  std_logic;
                qspi_io_in      :in  std_logic_vector(3 downto 0);
                qspi_io_out     :out  std_logic_vector(3 downto 0);
                qspi_out_en     :out  std_logic;
                qspi_clk_out    :out  std_logic;                --25MHZ
                qspi_cs         :out  std_logic;
                qspi_init_state  :out  std_logic;               -- 1: init mode, 0: quad mode
                qspi_dummy_cycle :out std_logic;                -- 1: dummy cycles, 0: no dummy
                data_write      :in  std_logic_vector(31 downto 0);
                data_read       :out std_logic_vector(31 downto 0);
                address_data    :in  std_logic_vector(31 downto 0);
                command_data    :in  std_logic_vector(31 downto 0);
                send_data       :in  std_logic_vector(31 downto 0)

        );
end Qspi_controller;

architecture arch of Qspi_controller is

-----Signals-----
signal qspi_clock               :std_logic;
signal qspi_clock_cnt           :std_logic_vector(1 downto 0);
signal clock_en                 :std_logic;
signal clock_en_cnt             :std_logic_vector(1 downto 0);
signal clock_en_rising_edge     :std_logic;
signal clock_en_cnt_rising_edge  :std_logic_vector(1 downto 0);

-- read signals
signal qspi_io_in_sig  :std_logic_vector(31 downto 0);
signal data_in_counter :std_logic_vector(3 downto 0);
signal data_read_sig   :std_logic_vector(31 downto 0);

-- command
signal initiate_cmd        :std_logic;
signal start_cmd_out       :std_logic;
signal send_quad_cmd_done  :std_logic;
signal command_data_sig    :std_logic_vector(7 downto 0);
signal command_counter     :std_logic_vector(1 downto 0);
signal qspi_dq_cmd_out     :std_logic_vector(3 downto 0);

-- address
signal initiate_add        :std_logic;
signal start_add_out       :std_logic;
signal send_quad_add_done  :std_logic;
signal address_data_sig    :std_logic_vector(23 downto 0);
signal address_counter     :std_logic_vector(2 downto 0);
signal qspi_dq_add_out     :std_logic_vector(3 downto 0);

-- data
signal initiate_data       :std_logic;
signal start_data_out      :std_logic;
signal send_quad_data_done  :std_logic;
signal output_data_sig     :std_logic_vector(31 downto 0);
signal data_out_counter    :std_logic_vector(3 downto 0);
signal qspi_dq_data_out    :std_logic_vector(3 downto 0);
```

```
signal initiate_nonvol       :std_logic;
signal start_nonvol_out      :std_logic;
signal send_quad_nonvol_done  :std_logic;
signal output_nonvol_sig     :std_logic_vector(15 downto 0);
signal nonvol_out_counter    :std_logic_vector(2 downto 0);
signal qspi_dq_nonvol_out    :std_logic_vector(3 downto 0);

type main_state_m is (initial,read_command_data_register1,read_status_register,fast_read,write_enable,
                read_command_data_register2,write_data_to_the_qspi,write_16bits_command,write_8bits_command,
                                                sector_erase);
signal main_sm:main_state_m;

signal done_read :std_logic;
signal done_write :std_logic;

type initialization_state_m is
(initial,send_command_we,wait_before_next_cmd,send_cmd_nonvol,send_single_data,done);
signal initialization_sm:initialization_state_m;

signal wait_counter          :std_logic_vector(7 downto 0);
signal command_we_singel     :std_logic_vector(7 downto 0);
signal counter_data          :std_logic_vector(7 downto 0);
signal qspi_singel_initial   :std_logic;
signal nonvol_reg_cmd        :std_logic_vector(7 downto 0);
signal nonvol_reg_singel     :std_logic_vector(15 downto 0);
signal initialization_sm_done  :std_logic;

-- state machines

type Read_status_reg_state_m is (initial,send_q_cmd_pr,read_data_pr,write_data_to_read_reg);
signal Read_status_reg_sm:Read_status_reg_state_m;

signal initiate_read_status    :std_logic;

type Fast_read_state_m is (initial,initiate_send_q_cmd_pr,initiate_send_q_addr_pr,wait_10_dummy_cycle,
                initiate_read_d_pr,write_data_to_read_reg);
signal Fast_read_sm:Fast_read_state_m;

signal dummy_10_cycles_counter   :std_logic_vector(3 downto 0);
signal Dummy_cycles_done         :std_logic;

type write_d_to_qspi_state_m is (initial,initiate_send_q_cmd_pr,initiate_send_q_addr_pr,initiate_send_q_data_pr);
signal write_d_to_qspi_sm:write_d_to_qspi_state_m;

type write_16bits_command_state_m is (initial,initiate_send_q_cmd_pr,initiate_send_nonvol_d_pr);
signal write_16bits_command_sm:write_16bits_command_state_m;

type write_8bits_command_state_m is (initial,initiate_send_q_cmd_pr,initiate_send_q_data_pr);
signal write_8bits_command_sm:write_8bits_command_state_m;

signal initiate_8bits_data           :std_logic;

type sector_erase_state_m is (initial,initiate_send_q_cmd_pr,initiate_send_q_addr_pr);
signal sector_erase_sm:sector_erase_state_m;

begin
qspi_clk_out <= qspi_clock;
```

```
process (clock, reset) is
begin
   if (reset = '0') then
      clock_en <= '0';
                  clock_en_cnt <= "00";
         elsif(falling_edge (clock)) then
            if (clock_en_cnt < "11") then
        clock_en_cnt <= clock_en_cnt +'1';
                  clock_en <= '0';
               else
                  clock_en_cnt <= clock_en_cnt +'1';
                  clock_en <= '1';
               end if;
         end if;
end process;

process (clock, reset) is
begin
   if (reset = '0') then
      clock_en_rising_edge <= '0';
                  clock_en_cnt_rising_edge <= "00";
         elsif(falling_edge(clock)) then
            if (clock_en_cnt_rising_edge = "01") then
        clock_en_cnt_rising_edge <= clock_en_cnt_rising_edge +'1';
                  clock_en_rising_edge <= '1';
               else
                  clock_en_cnt_rising_edge <= clock_en_cnt_rising_edge +'1';
                  clock_en_rising_edge <= '0';
               end if;
         end if;
end process;

process (clock, reset) is
begin
   if (reset = '0') then
      qspi_clock <= '0';                  --25Mhz
                  qspi_clock_cnt <= "00";
         elsif(falling_edge(clock)) then
            if (qspi_clock_cnt < "10") then
        qspi_clock_cnt <= qspi_clock_cnt +'1';
                  qspi_clock <= '0';
               elsif (qspi_clock_cnt = "10" or qspi_clock_cnt = "11") then
                  qspi_clock_cnt <= qspi_clock_cnt +'1';
                  qspi_clock <= '1';
               else
                  qspi_clock <= qspi_clock;
                        qspi_clock_cnt <= qspi_clock_cnt;
               end if;
         end if;
end process;

-- Read process
process (clock, reset) is
begin
   if (reset = '0') then
            qspi_io_in_sig <= (others=>'0');
```

```
            data_in_counter <= (others=>'0');
        elsif(falling_edge(clock)) then
            if (clock_en_rising_edge = '1') then
                if (Dummy_cycles_done = '1' and data_in_counter < X"8") then
                        data_in_counter <= data_in_counter + '1';
                                if (data_in_counter = X"0") then
                                        qspi_io_in_sig(31 downto 28) <= qspi_io_in(3 downto 0);
                                elsif (data_in_counter = X"1") then
                                        qspi_io_in_sig(27 downto 24) <= qspi_io_in(3 downto 0);
                                elsif (data_in_counter = X"2") then
                                        qspi_io_in_sig(23 downto 20) <= qspi_io_in(3 downto 0);
                                elsif (data_in_counter = X"3") then
                                        qspi_io_in_sig(19 downto 16) <= qspi_io_in(3 downto 0);
                                elsif (data_in_counter = X"4") then
                                        qspi_io_in_sig(15 downto 12) <= qspi_io_in(3 downto 0);
                                elsif (data_in_counter = X"5") then
                                        qspi_io_in_sig(11 downto 8) <= qspi_io_in(3 downto 0);
                                elsif (data_in_counter = X"6") then
                                        qspi_io_in_sig(7 downto 4) <= qspi_io_in(3 downto 0);
                                else
                                        qspi_io_in_sig(3 downto 0) <= qspi_io_in(3 downto 0);
                                end if;
                            elsif (initiate_read_status = '1' and data_in_counter < X"2") then
                        data_in_counter <= data_in_counter + '1';
                                if (data_in_counter = X"0") then
                                        qspi_io_in_sig(7 downto 4) <= qspi_io_in(3 downto 0);
                                else
                                        qspi_io_in_sig(3 downto 0) <= qspi_io_in(3 downto 0);
                                end if;

                    else
                        data_in_counter <= (others=>'0');
                        qspi_io_in_sig <= qspi_io_in_sig;
                    end if;
                end if;
        end if;
end process;

-- send command process
process (clock, reset) is
begin
   if (reset = '0') then
                    command_data_sig <= X"00";          -- 8 bit command
                    command_counter <= "00";
                    start_cmd_out <= '0';
                    send_quad_cmd_done <= '0';
                    qspi_dq_cmd_out <= (others=> '0');
        elsif(falling_edge(clock)) then
            if (clock_en = '1') then
       if (initiate_cmd = '1') then
                                if (command_counter = "00") then
                                   command_data_sig <= command_data(7 downto 0);
                                        qspi_dq_cmd_out <= (others=> '0');
                                        start_cmd_out <= '1';
                                        send_quad_cmd_done <= '0';
                                        command_counter <= command_counter + '1';
                                elsif (command_counter = "01") then
```

```
                                    qspi_dq_cmd_out <= command_data_sig(7 downto 4);
                                    start_cmd_out <= '1';
                                        send_quad_cmd_done <= '0';
                                        command_counter <= command_counter + '1';
                                elsif (command_counter = "10") then
                                    qspi_dq_cmd_out <= command_data_sig(3 downto 0);
                                    start_cmd_out <= '0';
                                        send_quad_cmd_done <= '1';
                                        command_counter <= command_counter + '1';
                                else
                                    qspi_dq_cmd_out <= qspi_dq_cmd_out;
                                command_data_sig <= command_data_sig;
                                        command_counter <= command_counter;
                                        start_cmd_out <= start_cmd_out;
                                        send_quad_cmd_done <= send_quad_cmd_done;
                                end if;
                            else
                                qspi_dq_cmd_out <= qspi_dq_cmd_out;
                                command_data_sig <= command_data_sig;
                                        command_counter <= "00";
                                        start_cmd_out <= '0';
                                        send_quad_cmd_done <= '0';
                            end if;
                    end if;
            end if;
end process;


--Send address process
process (clock, reset) is
begin
   if (reset = '0') then
                    address_data_sig <= X"000000";  --24 bit address
                    address_counter <= "000";
                    start_add_out <= '0';
                    send_quad_add_done <= '0';
                    qspi_dq_add_out <= (others=> '0');
            elsif(falling_edge(clock)) then
                if (clock_en = '1') then
           if (initiate_add = '1') then
                                    address_counter <= address_counter + '1';
                                    if (address_counter = "000") then
                                        address_data_sig <= address_data(23 downto 0);
                                            qspi_dq_add_out <= (others=> '0');
                                            start_add_out <= '1';
                                            send_quad_add_done <= '0';
                                    elsif (address_counter >= "001" and address_counter <= "110" ) then
                                        qspi_dq_add_out <= address_data_sig(23 downto 20);
                                            address_data_sig <= address_data_sig(19 downto 0)&address_data_sig(23
downto 20);
                                        if (address_counter = "110") then
                                            start_add_out <= '0';
                                                send_quad_add_done <= '1';
                                            else
                                                        start_add_out <= '1';
                                                send_quad_add_done <= '0';
                                            end if;
```

```
                                    else
                                        qspi_dq_add_out <= qspi_dq_add_out;
                                address_data_sig <= address_data_sig;
                                    address_counter <= address_counter;
                                    start_add_out <= start_add_out;
                                    send_quad_add_done <= send_quad_add_done;
                                end if;
                            else
                               qspi_dq_add_out <= qspi_dq_add_out;
                               address_data_sig <= address_data_sig;
                                    address_counter <= "000";
                                    start_add_out <= '0';
                                    send_quad_add_done <= '0';
                            end if;
                        end if;
            end if;
end process;

--Send Quad data process
process (clock, reset) is
begin
   if (reset = '0') then
                        output_data_sig <= X"00000000";  --32bit data
                        data_out_counter <= "0000";
                        start_data_out <= '0';
                        send_quad_data_done <= '0';
                        qspi_dq_data_out <= (others=> '0');
              elsif(falling_edge(clock)) then
                 if (clock_en = '1') then
            if (initiate_data = '1') then
                                    data_out_counter <= data_out_counter + '1';
                                    if (data_out_counter = "0000") then
                                       output_data_sig <= data_write(31 downto 0);
                                            qspi_dq_data_out <= (others=> '0');
                                            start_data_out <= '1';
                                            send_quad_data_done <= '0';
                                    elsif (data_out_counter >= "0001" and data_out_counter <= "1000" ) then
                                       qspi_dq_data_out <= output_data_sig(31 downto 28);
                                            output_data_sig <= output_data_sig(27 downto 0)&output_data_sig(31
downto 28);
                                       if (data_out_counter = "1000") then
                                          start_data_out <= '0';
                                               send_quad_data_done <= '1';
                                            else
                                                    start_data_out <= '1';
                                               send_quad_data_done <= '0';
                                            end if;
                                    else
                                       qspi_dq_data_out <= qspi_dq_data_out;
                                output_data_sig <= output_data_sig;
                                       data_out_counter <= data_out_counter;
                                       start_data_out <= start_data_out;
                                       send_quad_data_done <= send_quad_data_done;
                                    end if;

                                elsif(initiate_8bits_data = '1')then
                                       data_out_counter <= data_out_counter + '1';
```

```
                                    if (data_out_counter = "0000") then
                                       output_data_sig <= data_write(31 downto 0);
                                             qspi_dq_data_out <= (others=> '0');
                                             start_data_out <= '1';
                                             send_quad_data_done <= '0';
                                    elsif (data_out_counter >= "0001" and data_out_counter <= "0010" ) then
                                       qspi_dq_data_out <= output_data_sig(7 downto 4);
                                             output_data_sig <= output_data_sig(27 downto 0)&output_data_sig(31
downto 28);
                                       if (data_out_counter = "0010") then
                                          start_data_out <= '0';
                                                send_quad_data_done <= '1';
                                             else
                                                      start_data_out <= '1';
                                                send_quad_data_done <= '0';
                                             end if;
                                    else
                                       qspi_dq_data_out <= qspi_dq_data_out;
                                 output_data_sig <= output_data_sig;
                                       data_out_counter <= data_out_counter;
                                       start_data_out <= start_data_out;
                                       send_quad_data_done <= send_quad_data_done;
                                    end if;

                              else
                                 qspi_dq_data_out <= qspi_dq_data_out;
                                 output_data_sig <= output_data_sig;
                                       data_out_counter <= "0000";
                                       start_data_out <= '0';
                                       send_quad_data_done <= '0';
                              end if;
                        end if;
               end if;
end process;


--Send Quad nonvolatile data process (16 bits)
process (clock, reset) is
begin
   if (reset = '0') then
                        output_nonvol_sig <= X"0000";  --16bit data nonvolatile register
                        nonvol_out_counter <= "000";
                        start_nonvol_out <= '0';
                        send_quad_nonvol_done <= '0';
                        qspi_dq_nonvol_out <= (others=> '0');
            elsif(falling_edge(clock)) then
               if (clock_en = '1') then
           if (initiate_nonvol = '1') then
                                    nonvol_out_counter <= nonvol_out_counter + '1';
                                    if (nonvol_out_counter = "000") then
                                       output_nonvol_sig <= data_write(15 downto 0);
                                             qspi_dq_nonvol_out <= (others=> '0');
                                             start_nonvol_out <= '1';
                                             send_quad_nonvol_done <= '0';
                                    elsif (nonvol_out_counter >= "001" and nonvol_out_counter <= "100" ) then
                                       if (nonvol_out_counter = "001") then
                                                qspi_dq_nonvol_out <= output_nonvol_sig(7 downto 4); --LSB
```

```
                                    elsif(nonvol_out_counter = "010") then
                                       qspi_dq_nonvol_out <= output_nonvol_sig(3 downto 0);
                                    elsif(nonvol_out_counter = "011") then
                                          qspi_dq_nonvol_out <= output_nonvol_sig(15 downto 12); --
MSB
                                    elsif(nonvol_out_counter = "100") then
                                          qspi_dq_nonvol_out <= output_nonvol_sig(11 downto 8);
                                          start_nonvol_out <= '0';
                                       send_quad_nonvol_done <= '1';
                                    else
                                          qspi_dq_nonvol_out <= qspi_dq_nonvol_out;
                                          start_nonvol_out <= '1';
                                       send_quad_nonvol_done <= '0';
                                    end if;
                                 else
                                    qspi_dq_nonvol_out <= qspi_dq_nonvol_out;
                              output_nonvol_sig <= output_nonvol_sig;
                                    nonvol_out_counter <= nonvol_out_counter;
                                    start_nonvol_out <= start_nonvol_out;
                                    send_quad_nonvol_done <= send_quad_nonvol_done;
                                 end if;
                           else
                              qspi_dq_nonvol_out <= qspi_dq_nonvol_out;
                              output_nonvol_sig <= output_nonvol_sig;
                                    nonvol_out_counter <= "000";
                                    start_nonvol_out <= '0';
                                    send_quad_nonvol_done <= '0';
                           end if;
                     end if;
            end if;
end process;


--main state machine
process (clock, reset) is
begin
   if (reset = '0') then
      main_sm <= initial;
          elsif(falling_edge(clock)) then
             if (clock_en = '1') then
         case main_sm is
                                 when initial =>
                                    if (Send_data(0) = '1' and initialization_sm_done = '1') then
                                          main_sm <= read_command_data_register1;
                                    else
                                          main_sm <= initial;
                                    end if;

                                 when read_command_data_register1 =>
                                    if (command_data = X"05" or command_data = X"70") then
                                          main_sm <= read_status_register;
                                    elsif (command_data = X"0B") then
                                       main_sm <= fast_read;
                                    elsif (command_data = X"06") then
                                       main_sm <= write_enable;
                                    else
                                          main_sm <= read_command_data_register1;
```

```
                end if;

            when read_status_register =>
                if (done_read = '1') then
                        main_sm <= initial;
                else
                        main_sm <= read_status_register;
                end if;

            when fast_read =>
                if (fast_read_sm = write_data_to_read_reg) then
                        main_sm <= initial;
                else
                        main_sm <= fast_read;
                end if;

            when write_enable =>
               if (Send_data(0) = '1' and send_quad_cmd_done = '1') then
                        main_sm <= read_command_data_register2;
                else
                        main_sm <= write_enable;
                end if;

            when read_command_data_register2 =>
                if (command_data = X"02" and send_data(0) = '1') then
                        main_sm <= write_data_to_the_qspi;
                elsif (command_data = X"B1" and send_data(0) = '1') then
                   main_sm <= write_16bits_command;
                elsif ((command_data = X"61" or command_data = X"81") and
send_data(0) = '1') then
                   main_sm <= write_8bits_command;
                elsif (command_data = X"D8" and send_data(0) = '1') then
                   main_sm <= sector_erase;
                else
                        main_sm <= read_command_data_register2;
                end if;

            when write_data_to_the_qspi =>
                if (done_write = '1') then
                        main_sm <= initial;
                else
                        main_sm <= write_data_to_the_qspi;
                end if;
            when write_16bits_command =>
                if (done_write = '1') then
                        main_sm <= initial;
                else
                        main_sm <= write_16bits_command;
                end if;
            when write_8bits_command =>
                if (done_write = '1') then
                        main_sm <= initial;
                else
                        main_sm <= write_8bits_command;
                end if;
            when sector_erase =>
                if (done_write = '1') then
```

```
                                        main_sm <= initial;
                                    else
                                        main_sm <= sector_erase;
                                    end if;

                                when others => main_sm <= initial;

                            end case;
                        end if;
                end if;
end process;

data_read <= data_read_sig;
--main state machine
process (clock, reset) is
begin
   if (reset = '0') then
      done_read <= '0';
                        done_write <= '0';
                        initiate_add <= '0';
                        initiate_cmd <= '0';
                        initiate_data <= '0';
                        initiate_nonvol <= '0';
                        initiate_8bits_data <= '0';
                        qspi_io_out(3 downto 0) <= "0000";
                        qspi_cs <= '1';
                        qspi_out_en <= '1'; --write
                        qspi_init_state <= '1'; -- init mode
                        qspi_dummy_cycle <= '0';
                        data_read_sig <= X"00000000";
          elsif(falling_edge(clock)) then
             if (clock_en = '1') then
          case main_sm is
                                    when initial =>
                                       done_read <= '0';
                            done_write <= '0';
                            initiate_add <= '0';
                            initiate_cmd <= '0';
                            initiate_data <= '0';
                            initiate_nonvol <= '0';
                                        initiate_8bits_data <= '0';
                                        qspi_out_en <= '1'; --write
                                        qspi_dummy_cycle <= '0';
                                        data_read_sig <= data_read_sig;
                                    if ((initialization_sm = send_command_we) and (counter_data >= X"01" and
counter_data <= X"09")) then
                                        qspi_io_out(0) <= qspi_singel_initial;
                                        qspi_io_out(2 downto 1) <= "00";
                                        qspi_io_out(3) <= '1';
                                        qspi_init_state <= '1'; -- init mode
                                        qspi_cs <= '0';
                                    elsif ((initialization_sm = send_cmd_nonvol) and (counter_data >= X"01" and
counter_data <= X"08")) then
                                        qspi_io_out(0) <= qspi_singel_initial;
                                        qspi_io_out(2 downto 1) <= "00";
                                        qspi_io_out(3) <= '1';
                                        qspi_init_state <= '1'; -- init mode
```

```
                qspi_cs <= '0';
        elsif ((initialization_sm = send_single_data) and (counter_data >= X"09" and
counter_data <= X"18")) then
                qspi_io_out(0) <= qspi_singel_initial;
                qspi_io_out(2 downto 1) <= "00";
                qspi_io_out(3) <= '1';
                qspi_init_state <= '1'; -- init mode
                qspi_cs <= '0';
        else
           qspi_io_out <= "0000";
                qspi_init_state <= '0'; -- quad mode
           qspi_cs <= '1';
        end if;

        when read_command_data_register1 =>
           qspi_io_out <= "0000";
           qspi_cs <= '1';
           qspi_out_en <= '1'; --write

        when read_status_register =>
                if (Read_status_reg_sm = initial) then
                        qspi_io_out <= "0000";
              qspi_cs <= '1';
              qspi_out_en <= '1'; --write
                        initiate_cmd <= '1';
                elsif (Read_status_reg_sm = send_q_cmd_pr) then
                   qspi_io_out <= qspi_dq_cmd_out;
              qspi_out_en <= '1'; --write
                        initiate_cmd <= '1';
                        if (command_counter >= "10") then
                                qspi_cs <= '0';
                        else
                                qspi_cs <= '1';
                        end if;
                elsif (Read_status_reg_sm = read_data_pr) then
                   initiate_cmd <= '0';
                   qspi_io_out <= "0000";
              qspi_cs <= '0';
              qspi_out_en <= '0'; --read
                        if (data_in_counter = X"02") then
                                done_read <= '1';
                                qspi_cs <= '1';
                        end if;
                elsif (Read_status_reg_sm = write_data_to_read_reg) then
                   data_read_sig <= X"000000"&qspi_io_in_sig(7 downto 0);
                        qspi_io_out <= "0000";
              qspi_cs <= '1';
              qspi_out_en <= '1'; --write
                        done_read <= '1';
                else
                   qspi_io_out <= "0000";
              qspi_cs <= '1';
              qspi_out_en <= '1'; --write
                end if;

        when fast_read =>
                if (Fast_read_sm = initial) then
```

```
                qspi_io_out <= "0000";
        qspi_cs <= '1';
        qspi_out_en <= '1'; --write
                initiate_cmd <= '1';
            elsif (Fast_read_sm = initiate_send_q_cmd_pr) then
                qspi_io_out <= qspi_dq_cmd_out;
        qspi_out_en <= '1'; --write
                if (command_counter >= "01") then
                        initiate_add <= '1';
                else
                   initiate_add <= '0';
                end if;
                if (command_counter >= "10") then
                        qspi_cs <= '0';
                else
                        qspi_cs <= '1';
                end if;
            elsif (Fast_read_sm = initiate_send_q_addr_pr) then
                qspi_io_out <= qspi_dq_add_out;
        qspi_cs <= '0';
        qspi_out_en <= '1'; --write
                initiate_cmd <= '0';
            elsif (Fast_read_sm = wait_10_dummy_cycle) then
                initiate_add <= '0';
                qspi_io_out <= "0000";
        qspi_cs <= '0';
                qspi_dummy_cycle <= '1';

                if (dummy_10_cycles_counter >= X"A") then
                        qspi_out_en <= '0'; --read
                else
                        qspi_out_en <= '1'; --write
                end if;
            elsif (Fast_read_sm = initiate_read_d_pr) then
                qspi_io_out <= "0000";
        qspi_cs <= '0';
        qspi_out_en <= '0'; --read

            elsif (Fast_read_sm = write_data_to_read_reg) then
                data_read_sig <= qspi_io_in_sig;
                qspi_io_out <= "0000";
        qspi_cs <= '1';
        qspi_out_en <= '1'; --write
                done_read <= '1';
            else
                qspi_io_out <= "0000";
        qspi_cs <= '1';
        qspi_out_en <= '1'; --write
            end if;

    when write_enable =>
            if (send_data(0) = '1' and send_quad_cmd_done = '1') then
                initiate_cmd <= '0';
                qspi_io_out <= qspi_dq_cmd_out;
                qspi_cs <= '0';
            else
                initiate_cmd <= '1';
```

```
                qspi_out_en <= '1'; --write
                if (command_counter = "10") then
                        qspi_io_out <= qspi_dq_cmd_out;
                        qspi_cs <= '0';
                else
                        qspi_io_out <= "0000";
                        qspi_cs <= '1';
                end if;
        end if;

when read_command_data_register2 =>
        qspi_io_out <= "0000";
   qspi_cs <= '1';
   qspi_out_en <= '1'; --write

when write_data_to_the_qspi =>
        if (write_d_to_qspi_sm = initial) then
                qspi_io_out <= "0000";
      qspi_cs <= '1';
      qspi_out_en <= '1'; --write
                initiate_cmd <= '1';
                done_write <= '0';
        elsif (write_d_to_qspi_sm = initiate_send_q_cmd_pr) then
      qspi_out_en <= '1'; --write
          qspi_io_out <= qspi_dq_cmd_out;
                if (command_counter >= "01") then
                        initiate_add <= '1';
                else
                        initiate_add <= '0';
                end if;
                if (command_counter >= "10") then
                        qspi_cs <= '0';
                else
                        qspi_cs <= '1';
                end if;
        elsif (write_d_to_qspi_sm = initiate_send_q_addr_pr) then
           qspi_io_out <= qspi_dq_add_out;
      qspi_cs <= '0';
      qspi_out_en <= '1'; --write
                initiate_cmd <= '0';
                if (address_counter >= "101") then
                        initiate_data <= '1';
                else
                   initiate_data <= '0';
                end if;
        elsif (write_d_to_qspi_sm = initiate_send_q_data_pr) then
                qspi_io_out <= qspi_dq_data_out;
      qspi_cs <= '0';
      qspi_out_en <= '1'; --write
                initiate_add <= '0';
                if (data_out_counter = "1001") then
                        initiate_data <= '0';
                        done_write <= '1';
                else
                        initiate_data <= '1';
                end if;
        else
```

```
                qspi_io_out <= "0000";
        qspi_cs <= '1';
        qspi_out_en <= '1'; --write
            end if;

when write_16bits_command =>
            if (write_16bits_command_sm = initial) then
                        qspi_io_out <= "0000";
        qspi_cs <= '1';
        qspi_out_en <= '1'; --write
                        initiate_cmd <= '1';
                        done_write <= '0';
            elsif (write_16bits_command_sm = initiate_send_q_cmd_pr) then
                qspi_io_out <= qspi_dq_cmd_out;
        qspi_out_en <= '1'; --write
                        if (command_counter >= "01") then
                                initiate_nonvol <= '1';
                        else
                           initiate_nonvol <= '0';
                        end if;
                        if (command_counter >= "10") then
                                qspi_cs <= '0';
                        else
                                qspi_cs <= '1';
                        end if;
            elsif (write_16bits_command_sm = initiate_send_nonvol_d_pr) then
                        qspi_io_out <= qspi_dq_nonvol_out;
        qspi_out_en <= '1'; --write
                        initiate_cmd <= '0';
                        qspi_cs <= '0';
                        if (nonvol_out_counter = "101") then
                                initiate_nonvol <= '0';
                                done_write <= '1';
                        else
                                initiate_nonvol <= '1';
                        end if;
            else
                qspi_io_out <= "0000";
        qspi_cs <= '1';
        qspi_out_en <= '1'; --write
            end if;

when write_8bits_command =>
            if (write_8bits_command_sm = initial) then
                        qspi_io_out <= "0000";
        qspi_cs <= '1';
        qspi_out_en <= '1'; --write
                        initiate_cmd <= '1';
                        done_write <= '0';
            elsif (write_8bits_command_sm = initiate_send_q_cmd_pr) then
                qspi_io_out <= qspi_dq_cmd_out;
        qspi_out_en <= '1'; --write
                        if (command_counter >= "01") then
                                initiate_8bits_data <= '1';
                        else
                           initiate_8bits_data <= '0';
                        end if;
```

```
                    if (command_counter >= "10") then
                            qspi_cs <= '0';
                    else
                            qspi_cs <= '1';
                    end if;
            elsif (write_8bits_command_sm = initiate_send_q_data_pr) then
                    qspi_io_out <= qspi_dq_data_out;
        qspi_out_en <= '1'; --write
                    initiate_cmd <= '0';
                    qspi_cs <= '0';
                    if (data_out_counter = "0011") then
                            initiate_8bits_data <= '0';
                            done_write <= '1';
                    else

                            initiate_8bits_data <= '1';
                    end if;
            else
               qspi_io_out <= "0000";
        qspi_cs <= '1';
        qspi_out_en <= '1'; --write
            end if;

when sector_erase =>
            if (sector_erase_sm = initial) then
                    qspi_io_out <= "0000";
        qspi_cs <= '1';
        qspi_out_en <= '1'; --write
                    initiate_cmd <= '1';
                    done_write <= '0';
            elsif (sector_erase_sm = initiate_send_q_cmd_pr) then
               qspi_io_out <= qspi_dq_cmd_out;
        qspi_out_en <= '1'; --write
                    if (command_counter >= "01") then
                            initiate_add <= '1';
                    else
                       initiate_add <= '0';
                    end if;
                    if (command_counter >= "10") then
                            qspi_cs <= '0';
                    else
                            qspi_cs <= '1';
                    end if;
            elsif (sector_erase_sm = initiate_send_q_addr_pr) then
                    qspi_io_out <= qspi_dq_add_out;
        qspi_out_en <= '1'; --write
                    initiate_cmd <= '0';
                    qspi_cs <= '0';
                    if (address_counter >= "111") then
                            initiate_add <= '0';
                            done_write <= '1';
                    else
                            initiate_add <= '1';
                    end if;
            else
               qspi_io_out <= "0000";
        qspi_cs <= '1';
```

```
                                        qspi_out_en <= '1'; --write
                                            end if;

                                        when others =>  qspi_io_out <= "0000";
                                                                    qspi_cs <= '1';
                                                                    qspi_out_en <= '1'; --write

                                end case;
                        end if;
                end if;
end process;

--initialization state machine
process (clock, reset) is
begin
   if (reset = '0') then
                initialization_sm <= initial;
      wait_counter <= X"00";
                        counter_data <= X"00";
                        qspi_singel_initial <= '0';
                        command_we_singel <= X"06"; --write enable command
                        nonvol_reg_cmd <= X"B1";
                        nonvol_reg_singel <= X"F7AF";
                initialization_sm_done <= '0';

            elsif(falling_edge(clock)) then
                if (clock_en = '1') then
           case initialization_sm is
                                        when initial =>
               if (wait_counter >= X"DF" and wait_counter < X"FF") then

                                                        initialization_sm <= initial;
                                                wait_counter <= wait_counter + '1';
               elsif (wait_counter = X"FF") then

                                                wait_counter <= X"00";
                                                initialization_sm <= send_command_we;
                                            else
                                                initialization_sm <= initial;
                                                wait_counter <= wait_counter + '1';

                                            end if;

                                        when send_command_we =>
                                                if (counter_data = X"09") then
                                                    initialization_sm <= wait_before_next_cmd;
                                                        counter_data <= X"00";
                                                        qspi_singel_initial <= '0';
                                                else
                                                    qspi_singel_initial <= command_we_singel(7);
                                                        command_we_singel <= command_we_singel(6 downto 0)&
command_we_singel(7);
                                                        counter_data <= counter_data + '1';
                                                        initialization_sm <= send_command_we;
                                                end if;

                                        when wait_before_next_cmd =>
```

```
                                    qspi_singel_initial <= '0';
            if (wait_counter = X"B0") then
                                        wait_counter <= X"00";
                                        initialization_sm <= send_cmd_nonvol;
                                    else
                                        initialization_sm <= wait_before_next_cmd;
                                        wait_counter <= wait_counter + '1';
                                    end if;

                                when send_cmd_nonvol =>
                                    if (counter_data = X"08") then
                                        initialization_sm <= send_single_data;
                                            counter_data <= counter_data + '1';
                                            qspi_singel_initial <= nonvol_reg_singel(15);
                                            nonvol_reg_singel <= nonvol_reg_singel(14 downto 0)&
nonvol_reg_singel(15);
                                    else -- counter_data < X"09"
                                        qspi_singel_initial <= nonvol_reg_cmd(7);
                                            nonvol_reg_cmd <= nonvol_reg_cmd(6 downto 0)&
nonvol_reg_cmd(7);
                                            counter_data <= counter_data + '1';
                                            initialization_sm <= send_cmd_nonvol;
                                    end if;

                                when send_single_data =>
                                    if (counter_data = X"18") then
                                        initialization_sm <= done;
                                            counter_data <= X"00";
                                            qspi_singel_initial <= '0';
                                    else -- counter_data < X"10"
                                        qspi_singel_initial <= nonvol_reg_singel(15);
                                            nonvol_reg_singel <= nonvol_reg_singel(14 downto 0)&
nonvol_reg_singel(15);
                                            counter_data <= counter_data + '1';
                                            initialization_sm <= send_single_data;
                                    end if;

                                when done =>

                                    if (wait_counter >= X"00" and wait_counter <= X"0F") then
                                            wait_counter <= wait_counter + '1';

                                            qspi_singel_initial <= '0';
                                            initialization_sm <= done;
                                            initialization_sm_done <= '0';
                                    elsif (wait_counter > X"0F" and wait_counter <= X"B0") then
                                            wait_counter <= wait_counter + '1';

                                            qspi_singel_initial <= '0';
                                            initialization_sm <= done;
                                            initialization_sm_done <= '0';
                                    else
                                            wait_counter <= wait_counter;

              initialization_sm <= done;
                                            qspi_singel_initial <= '0';
                                            initialization_sm_done <= '1';
```

```
                                        end if;
                                    when others => initialization_sm <= done;

                                end case;
                        end if;
                end if;
end process;

--Read status register state machine
process (clock, reset) is
begin
   if (reset = '0') then
                Read_status_reg_sm <= initial;
      initiate_read_status <= '0';
           elsif(falling_edge(clock)) then
              if (clock_en = '1') then
         case Read_status_reg_sm is
                                    when initial =>
                                        if (main_sm = read_status_register) then
                                                Read_status_reg_sm <= send_q_cmd_pr;
                                      else
                                                Read_status_reg_sm <= initial;
                                        end if;

                                    when send_q_cmd_pr =>
                                        if (command_counter = "11") then
                                                Read_status_reg_sm <= read_data_pr;
                                      else
                                                Read_status_reg_sm <= send_q_cmd_pr;
                                        end if;

                                    when read_data_pr =>
                                        if (data_in_counter = X"2") then
                                                Read_status_reg_sm <= write_data_to_read_reg;
                                                initiate_read_status <= '0';
                                      else
                                         initiate_read_status <= '1';
                                                Read_status_reg_sm <= read_data_pr;
                                        end if;

                                    when write_data_to_read_reg =>
              Read_status_reg_sm <= initial;

                                    when others => Read_status_reg_sm <= initial;

                                end case;
                        end if;
                end if;
end process;

--Fast Read state machine
process (clock, reset) is
begin
   if (reset = '0') then
                Fast_read_sm <= initial;
                        Dummy_cycles_done <= '0';
                        dummy_10_cycles_counter <= "0000";
```

```
        elsif(falling_edge(clock)) then
            if (clock_en = '1') then
        case Fast_read_sm is
                                        when initial =>
                                                Dummy_cycles_done <= '0';
                                                dummy_10_cycles_counter <= "0000";
                                                if (main_sm = fast_read) then
                                                        Fast_read_sm <= initiate_send_q_cmd_pr;
                                                else
                                                   Fast_read_sm <= initial;
                                                end if;

                                        when initiate_send_q_cmd_pr =>
                                                if (command_counter = "11") then
                                                        Fast_read_sm <= initiate_send_q_addr_pr;
                                           else
                                                        Fast_read_sm <= initiate_send_q_cmd_pr;
                                                end if;

                                        when initiate_send_q_addr_pr =>
                                                if (address_counter = "111") then
                                                   Fast_read_sm <= wait_10_dummy_cycle;
                                                else
                                                   Fast_read_sm <= initiate_send_q_addr_pr;
                                                end if;

                                        when wait_10_dummy_cycle =>
                                                dummy_10_cycles_counter <= dummy_10_cycles_counter + '1';
                                                if (dummy_10_cycles_counter = "1010") then
                                                        Fast_read_sm <= initiate_read_d_pr;
                                                        Dummy_cycles_done <= '1';
                                                else
                                                        Fast_read_sm <= wait_10_dummy_cycle;
                                                        Dummy_cycles_done <= '0';
                                                end if;
                                        when initiate_read_d_pr =>
                                                Dummy_cycles_done <= '1';
                                                if (data_in_counter = X"7") then
                                                   Fast_read_sm <= write_data_to_read_reg;
                                                else
                                                   Fast_read_sm <= initiate_read_d_pr;
                                                end if;

                                        when write_data_to_read_reg =>
                                                Dummy_cycles_done <= '0';
                                                Fast_read_sm <= initial;

                                        when others => Fast_read_sm <= initial;

                                end case;
                        end if;
                end if;
end process;

--Write quad data to the qspi state machine

process (clock, reset) is
```

```
begin
   if (reset = '0') then
            write_d_to_qspi_sm <= initial;

          elsif(falling_edge(clock)) then
             if (clock_en = '1') then
         case write_d_to_qspi_sm is
                                        when initial =>
                                                if (main_sm = write_data_to_the_qspi) then
                                                        write_d_to_qspi_sm <= initiate_send_q_cmd_pr;
                                                else
                                                   write_d_to_qspi_sm <= initial;
                                                end if;

                                        when initiate_send_q_cmd_pr =>
                                                if (command_counter = "11") then
                                                        write_d_to_qspi_sm <= initiate_send_q_addr_pr;
                                           else
                                                        write_d_to_qspi_sm <= initiate_send_q_cmd_pr;
                                                end if;

                                        when initiate_send_q_addr_pr =>
                                                if (address_counter = "111") then
                                                   write_d_to_qspi_sm <= initiate_send_q_data_pr;
                                                else
                                                   write_d_to_qspi_sm <= initiate_send_q_addr_pr;
                                                end if;

                                        when initiate_send_q_data_pr =>
                                                if (data_out_counter = "1001") then
                                                   write_d_to_qspi_sm <= initial;
                                                else
                                                   write_d_to_qspi_sm <= initiate_send_q_data_pr;
                                                end if;

                                        when others => write_d_to_qspi_sm <= initial;

                                end case;
                        end if;
             end if;
end process;

--write 16 bits to qspi state machine
process (clock, reset) is
begin
   if (reset = '0') then
            write_16bits_command_sm <= initial;
          elsif(falling_edge(clock)) then
             if (clock_en = '1') then
         case write_16bits_command_sm is
                                        when initial =>
                                                if (main_sm = write_16bits_command) then
                                                        write_16bits_command_sm <= initiate_send_q_cmd_pr;
                                                else
                                                   write_16bits_command_sm <= initial;
                                                end if;
```

```
                                        when initiate_send_q_cmd_pr =>
                                                if (command_counter = "11") then
                                                        write_16bits_command_sm <= initiate_send_nonvol_d_pr;
                                                  else
                                                        write_16bits_command_sm <= initiate_send_q_cmd_pr;
                                                end if;

                                        when initiate_send_nonvol_d_pr =>
                                                if (nonvol_out_counter = "101") then
                                                   write_16bits_command_sm <= initial;
                                                else
                                                   write_16bits_command_sm <= initiate_send_nonvol_d_pr;
                                                end if;

                                        when others => write_16bits_command_sm <= initial;
                                end case;
                        end if;
                end if;
end process;

--write 8 bits to qspi state machine
process (clock, reset) is
begin
   if (reset = '0') then
                write_8bits_command_sm <= initial;
           elsif(falling_edge(clock)) then
                if (clock_en = '1') then
          case write_8bits_command_sm is
                                        when initial =>
                                                if (main_sm = write_8bits_command) then
                                                        write_8bits_command_sm <= initiate_send_q_cmd_pr;
                                                else
                                                   write_8bits_command_sm <= initial;
                                                end if;

                                        when initiate_send_q_cmd_pr =>
                                                if (command_counter = "11") then
                                                        write_8bits_command_sm <= initiate_send_q_data_pr;
                                                  else
                                                        write_8bits_command_sm <= initiate_send_q_cmd_pr;
                                                end if;

                                        when initiate_send_q_data_pr =>
                                                if (data_out_counter = "0011") then
                                                   write_8bits_command_sm <= initial;
                                                else
                                                   write_8bits_command_sm <= initiate_send_q_data_pr;
                                                end if;

                                        when others => write_8bits_command_sm <= initial;

                                end case;
                        end if;
                end if;
end process;

--sector erase state machine
```

```
process (clock, reset) is
begin
   if (reset = '0') then
            sector_erase_sm <= initial;
         elsif(falling_edge(clock)) then
            if (clock_en = '1') then
        case sector_erase_sm is
                                    when initial =>
                                        if (main_sm = sector_erase) then
                                            sector_erase_sm <= initiate_send_q_cmd_pr;
                                        else
                                           sector_erase_sm <= initial;
                                        end if;

                                    when initiate_send_q_cmd_pr =>
                                        if (command_counter = "11") then
                                            sector_erase_sm <= initiate_send_q_addr_pr;
                                      else
                                            sector_erase_sm <= initiate_send_q_cmd_pr;
                                        end if;

                                    when initiate_send_q_addr_pr =>
                                        if (address_counter = "111") then
                                           sector_erase_sm <= initial;
                                        else
                                           sector_erase_sm <= initiate_send_q_addr_pr;
                                        end if;

                                    when others => sector_erase_sm <= initial;

                            end case;
                    end if;
            end if;
end process;
end arch;
```

VHDL Code for Top Module:

```
----------------------------------------------------------------------------------------------------
-- Company                      : Space Applications Centre, ISRO
-- Engineer                     : Adhiraj Roy Chowdhury
-- Create Date                  : 16:35:00 10/05/2022
-- Target Devices               : FPGA:RT4G150_ES, Controller for 3DFS256M04VS2801 NOR Flash memory
----------------------------------------------------------------------------------------------------

library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_unsigned.all;

Entity Flash_mem_top is                         -- this is our top module
        port(

        --clock and reset from the user/testbench
```

```
        i_clk          :IN std_logic;                          --100MHZ
    i_rst_n        :IN std_logic;

        --flash mem pins
        o_SCK          :OUT std_logic;
    o_Chip_en      :OUT std_logic;
    io             :INOUT std_logic_vector(3 downto 0);

        --User/TB register file pins
        i_data         :IN  std_logic_vector(31 downto 0);
        o_MOSI         :OUT std_logic_vector(31 downto 0);
        i_address      :IN  std_logic_vector(23  downto 0);
        i_wren         :IN  std_logic;
        i_rden         :IN  std_logic
);
end entity;

Architecture Flash_mem_top_arch of Flash_mem_top is

--components
component Qspi_controller
        port(
                clock              :in  std_logic;                    --100MHZ
                reset           :in  std_logic;
                qspi_io_in      :in   std_logic_vector(3 downto 0);
                qspi_io_out      :out  std_logic_vector(3 downto 0);
                qspi_out_en      :out  std_logic;
                qspi_clk_out     :out  std_logic;                    --25MHZ
                qspi_cs         :out  std_logic;
                qspi_init_state  :out  std_logic;                    -- 1: init mode, 0: quad mode
                qspi_dummy_cycle :out std_logic;                     -- 1: dummy cycles, 0: no dummy
                data_write       :in  std_logic_vector(31 downto 0);
                data_read        :out std_logic_vector(31 downto 0);
                address_data     :in  std_logic_vector(31 downto 0);
                command_data     :in  std_logic_vector(31 downto 0);
                send_data        :in  std_logic_vector(31 downto 0)

        );
end component;

component Registers_file
        port(
                clock           :in std_logic;
                reset         :in std_logic;
            i_data        :in  std_logic_vector(31 downto 0);
            o_MOSI        :out std_logic_vector(31 downto 0);
            i_address     :in  std_logic_vector(23  downto 0);
            i_wren        :in  std_logic;
            i_rden        :in  std_logic;
                data_write    :out std_logic_vector(31 downto 0);
                data_read     :in  std_logic_vector(31 downto 0);
                address_data  :out std_logic_vector(31 downto 0);
                command_data  :out std_logic_vector(31 downto 0);
                send_data     :out std_logic_vector(31 downto 0)

        );
end component;
```

```
--- flash signals
signal o_Chip_en_sig                    :std_logic;
signal qspi_io_in_sig       :std_logic_vector(3 downto 0);
signal qspi_io_out_sig      :std_logic_vector(3 downto 0);
signal qspi_out_en_sig      :std_logic;
signal qspi_init_state_sig  :std_logic; -- 1: init mode, 0: quad mode
signal qspi_dummy_cycle_sig :std_logic; -- 1: dummy cycles, 0: no dummy

--register file signals
signal data_write_sig       :std_logic_vector(31 downto 0);
signal data_read_sig        :std_logic_vector(31 downto 0);
signal address_data_sig     :std_logic_vector(31 downto 0);
signal command_data_sig     :std_logic_vector(31 downto 0);
signal send_data_sig        :std_logic_vector(31 downto 0);

begin

o_Chip_en <= o_Chip_en_sig;
io(3 downto 0) <= qspi_io_out_sig when ((qspi_out_en_sig = '1') and (o_Chip_en_sig = '0') and (qspi_init_state_sig =
'0')and (qspi_dummy_cycle_sig = '0'))
            else "1ZZ"&qspi_io_out_sig(0) when ((qspi_out_en_sig = '1') and (o_Chip_en_sig = '0') and
(qspi_init_state_sig = '1')and (qspi_dummy_cycle_sig = '0'))
                                        else "ZZZZ";
qspi_io_in_sig<= io(3 downto 0) when ((qspi_out_en_sig = '0') and (o_Chip_en_sig = '0')) else "0000";


Qspi_controller_top: Qspi_controller
        port map(
                clock              => i_clk,
                reset           => i_rst_n,
                qspi_io_in       => qspi_io_in_sig,
                qspi_io_out      => qspi_io_out_sig,
                qspi_out_en      => qspi_out_en_sig,
                qspi_clk_out     => o_SCK,
                qspi_cs          => o_Chip_en_sig,
                qspi_init_state  => qspi_init_state_sig,
                qspi_dummy_cycle => qspi_dummy_cycle_sig,
                data_write       => data_write_sig,
                data_read        => data_read_sig,
                address_data     => address_data_sig,
                command_data     => command_data_sig,
                send_data        => send_data_sig

        );

Registers_file_top: Registers_file
        port map(
                clock => i_clk,
                reset => i_rst_n,
           i_data => i_data,
           o_MOSI => o_MOSI,
           i_address => i_address,
           i_wren => i_wren,
           i_rden => i_rden,
                data_write => data_write_sig,
                data_read => data_read_sig,
```

```
                address_data => address_data_sig,
                command_data => command_data_sig,
                send_data => send_data_sig

        );

end Flash_mem_top_arch;
```

CHAPTER - 6

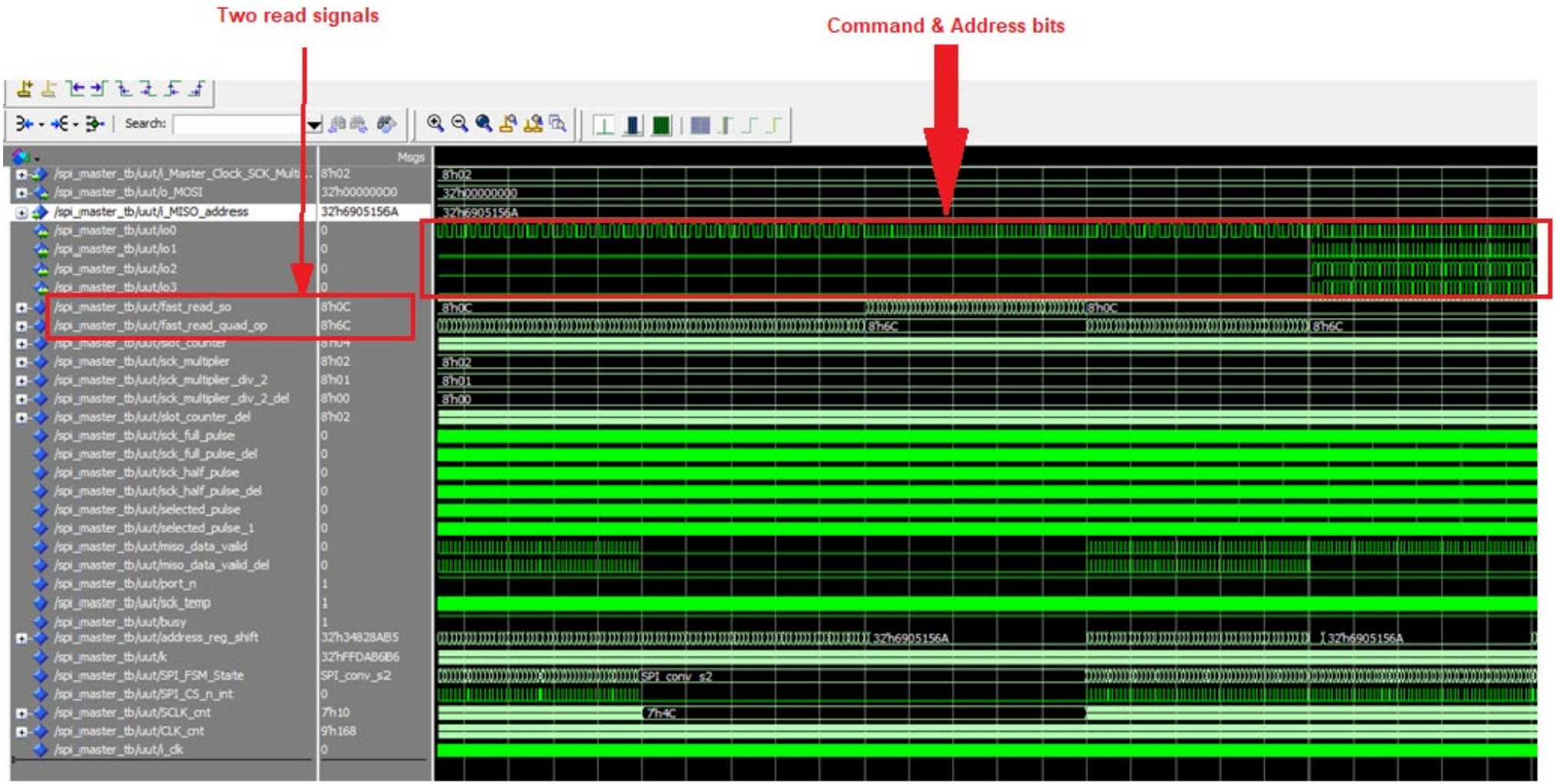
S I M U L A T I O N R E S U L T S

Pre-Synthesis (Behavioral) simulation:

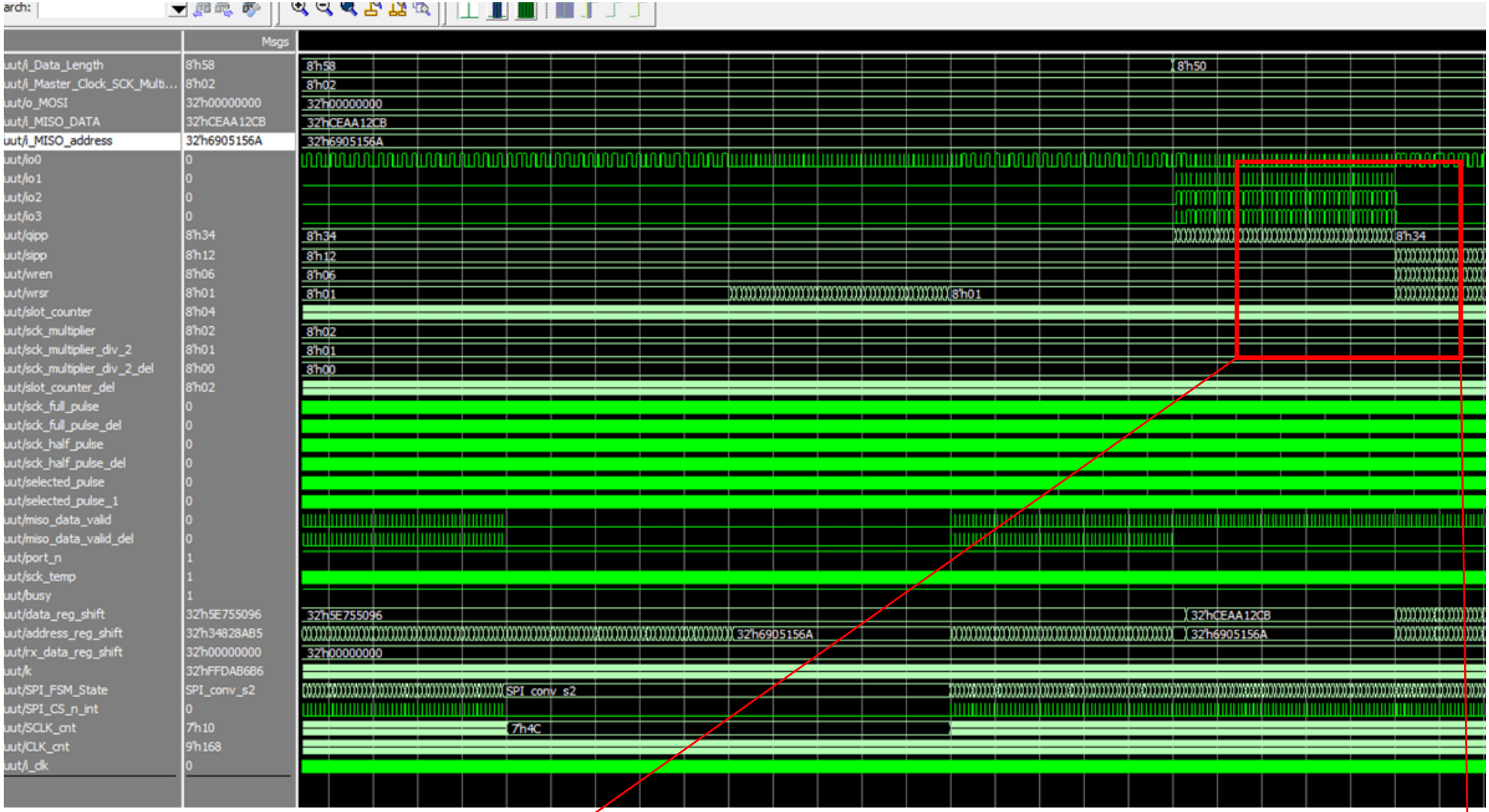
Software used: Questasim 10.1d

Simulation is the process of verifying the functionality and timing of a design against its original specifications. The purpose of pre-synthesis or 'Behavioral' simulation is to verify the logical functionality of the design, without worrying about the specific timing details of a particular implementation. This saves a lot of time in the functional debugging cycle, since you don't need to wait for the synthesis process to complete before simulating again after making a change.

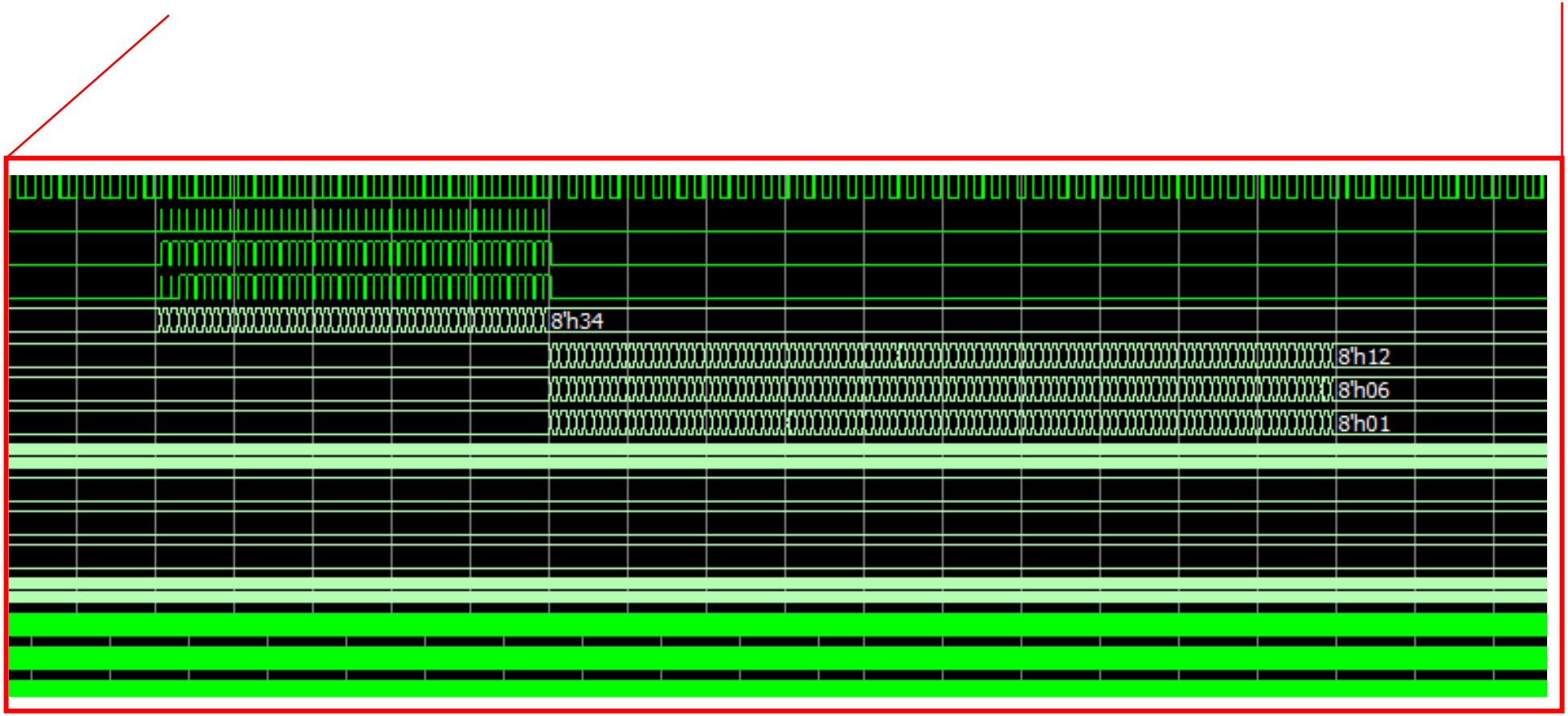
The READ and WRITE simulations are shown next.



READ SIMULATION



WRITE SIMULATION



Zoomed - In View of Write Simulation

Code Coverage results:

Software used: Questasim 10.1d

Code coverage measures how much code of the Design Under Test (DUT) is checked by the testbench, providing information about dead code in the design and holes in test suits. Code coverage also helps improve the predictability and quality of the test environment.

There are different categories of code coverage.

- Statement coverage
- Block coverage
- Conditional/Expression coverage
- Branch/Decision coverage
- Toggle coverage
- FSM coverage

Statement Coverage /Line Coverage:
This gives an indication of how many statements (lines) are covered in the simulation, by excluding lines like module, end module, comments, timescale etc.

Block coverage:
A group of statements which are in the begin-end or if-else or case or wait or while loop or for loop etc. is called a block. Block coverage gives the indication that whether these blocks are covered in simulation or not.

Conditional/Expression Coverage:
This gives an indication how well variables and expressions (with logical operators) in conditional statements are evaluated.

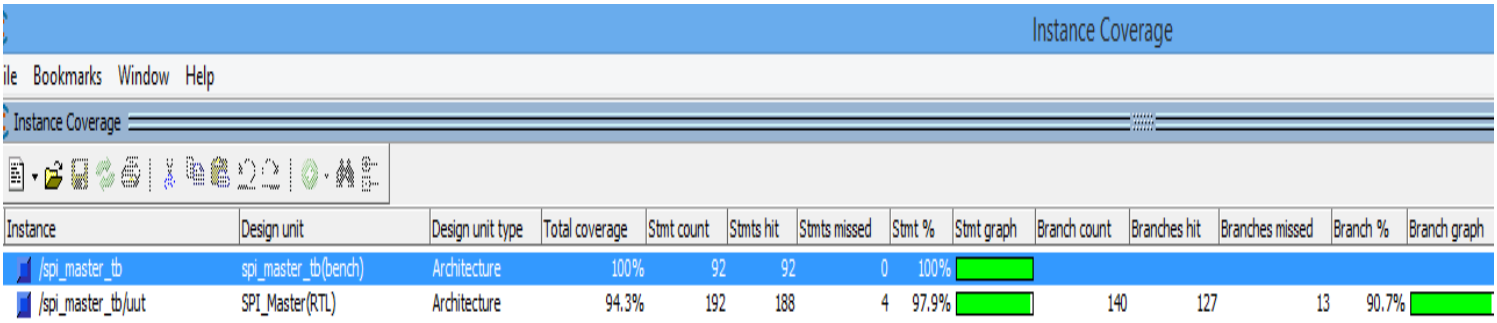
Branch/Decision Coverage:
In Branch coverage or Decision coverage reports, conditions like if-else, case and the ternary operator (?:) statements are evaluated in both true and false cases.

Toggle Coverage:
Toggle coverage gives a report that how many times signals and ports are toggled during a simulation run. It also measures activity in the design, such as unused signals or signals that remain constant or less value changes.

State/FSM Coverage:
FSM coverage reports, whether the simulation run could reach all of the states and cover all possible transitions or arcs in a given state machine.

Below is a snapshot of the code coverage result:

Statements covered = 97.9%
Branches covered = 90.7%



Instance	Design unit	Design unit type	Total coverage	Stmt count	Stmts hit	Stmts missed	Stmt %	Stmt graph	Branch count	Branches hit	Branches missed	Branch %	Branch graph
/spi_master_tb	spi_master_tb(bench)	Architecture	100%	92	92	0	100%						
/spi_master_tb/uut	SPI_Master(RTL)	Architecture	94.3%	192	188	4	97.9%		140	127	13	90.7%	

A coverage of 90% and above indicates the testbench is highly productive.

Code Coverage depends on two main things – design complexity and the time available [number of days/weeks/months allocated for simulation].

The Behavioral simulation and Coverage tests were done before Synthesizing our design. Next, Design Synthesis was done. For synthesizing and further performing timing analyses, LiberoSOC software was used. The reason behind using LiberoSOC was that it has a support for RTG4 family of FPGA devices – the kind of device on which my code will be programmed.

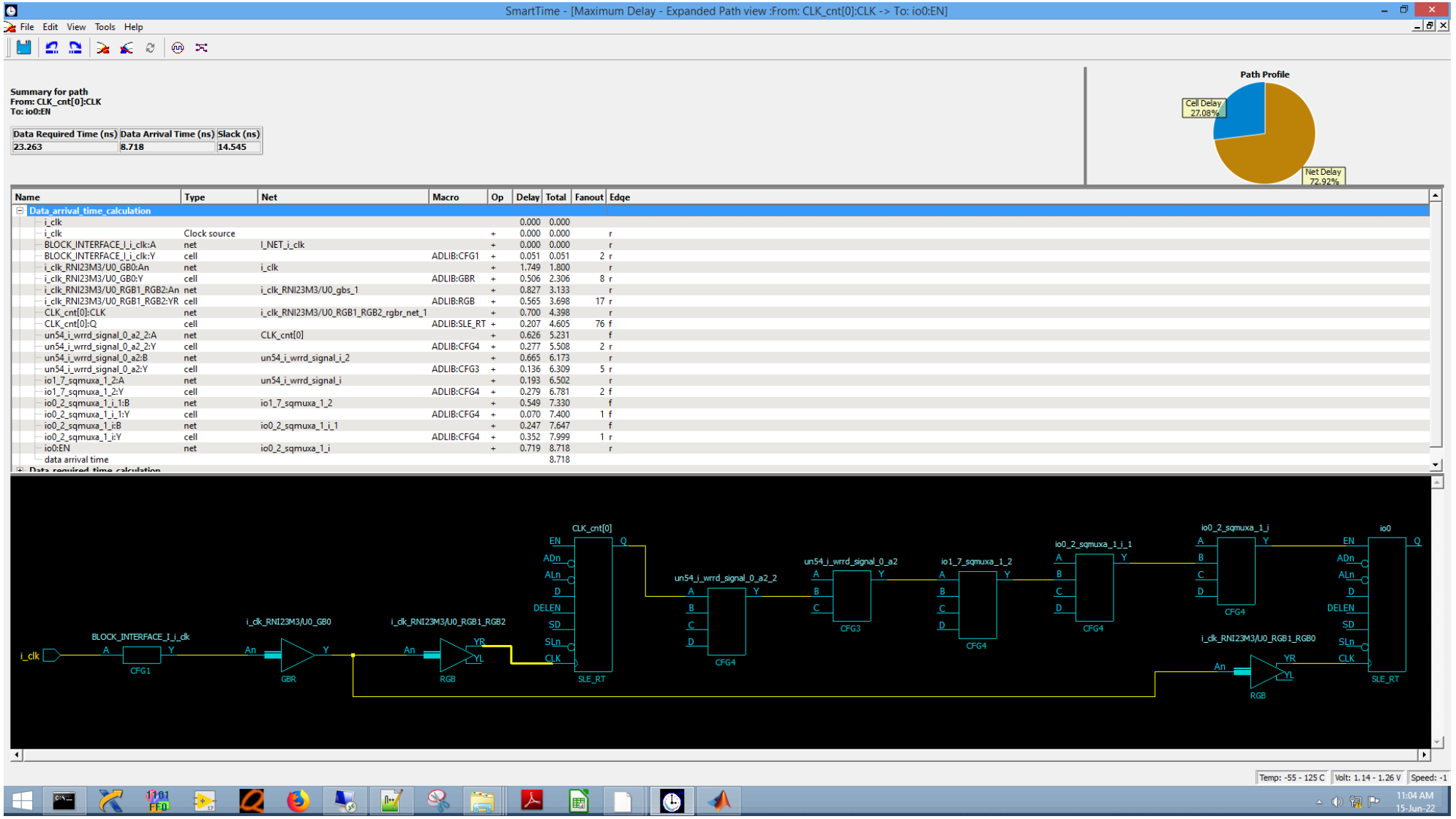
For Synthesis part, used 'Synplify Pro AE' tool of the software to generate an EDIF netlist. The design was then re- verified post- synthesis.

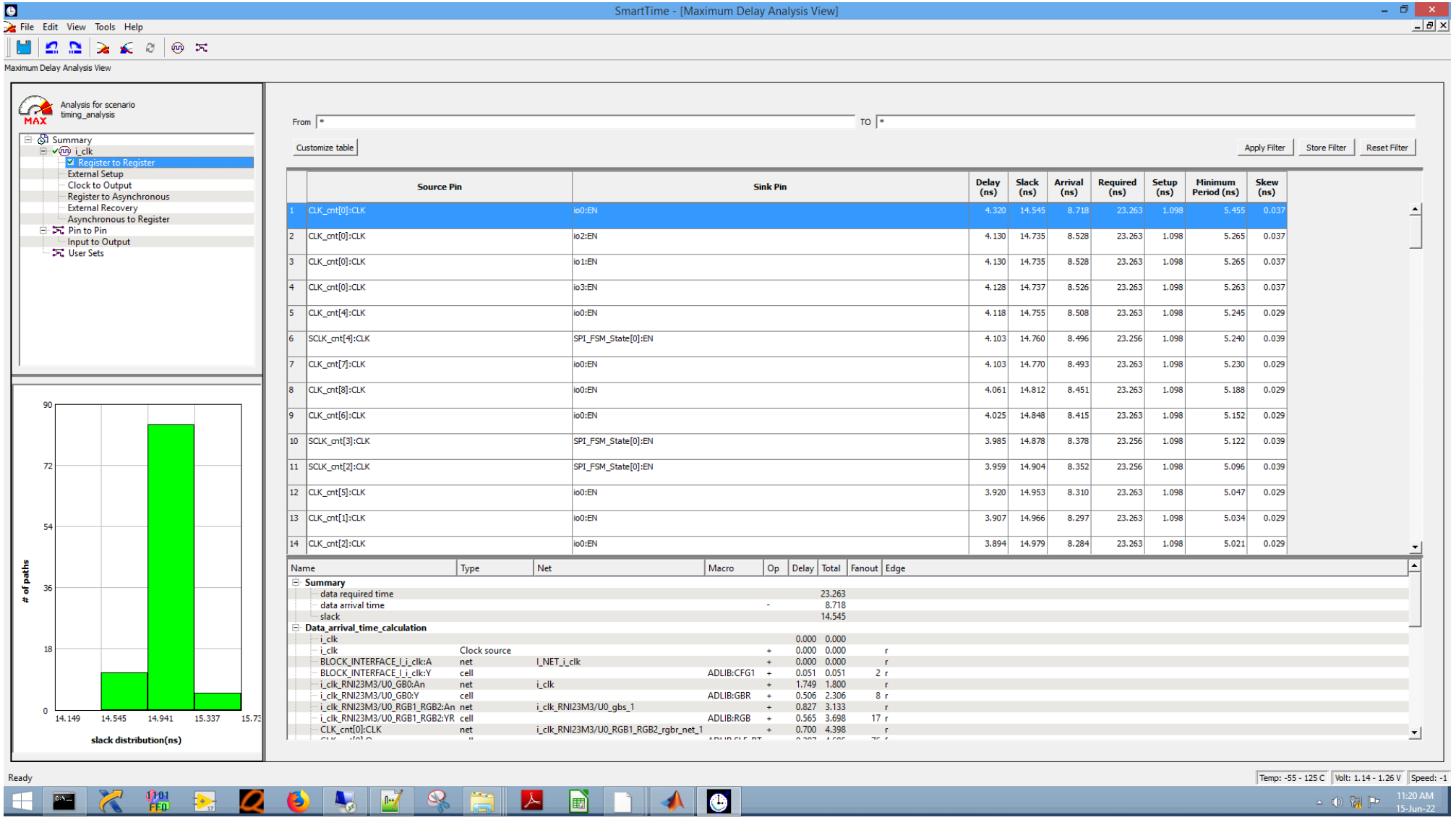
After verifying that the design works, in the next step implemented the design using the Actel Designer software (this software is combined in **Libero SoC Design** Flow window). The Designer software automatically places and routes the design and returned timing information. Further, the 'SmartTime' tool was used to perform Static Timing Analysis on the design as shown below:

Static Timing Analysis:

Software Used: LiberoSOC v11.9 for Device: RT4G150_ES

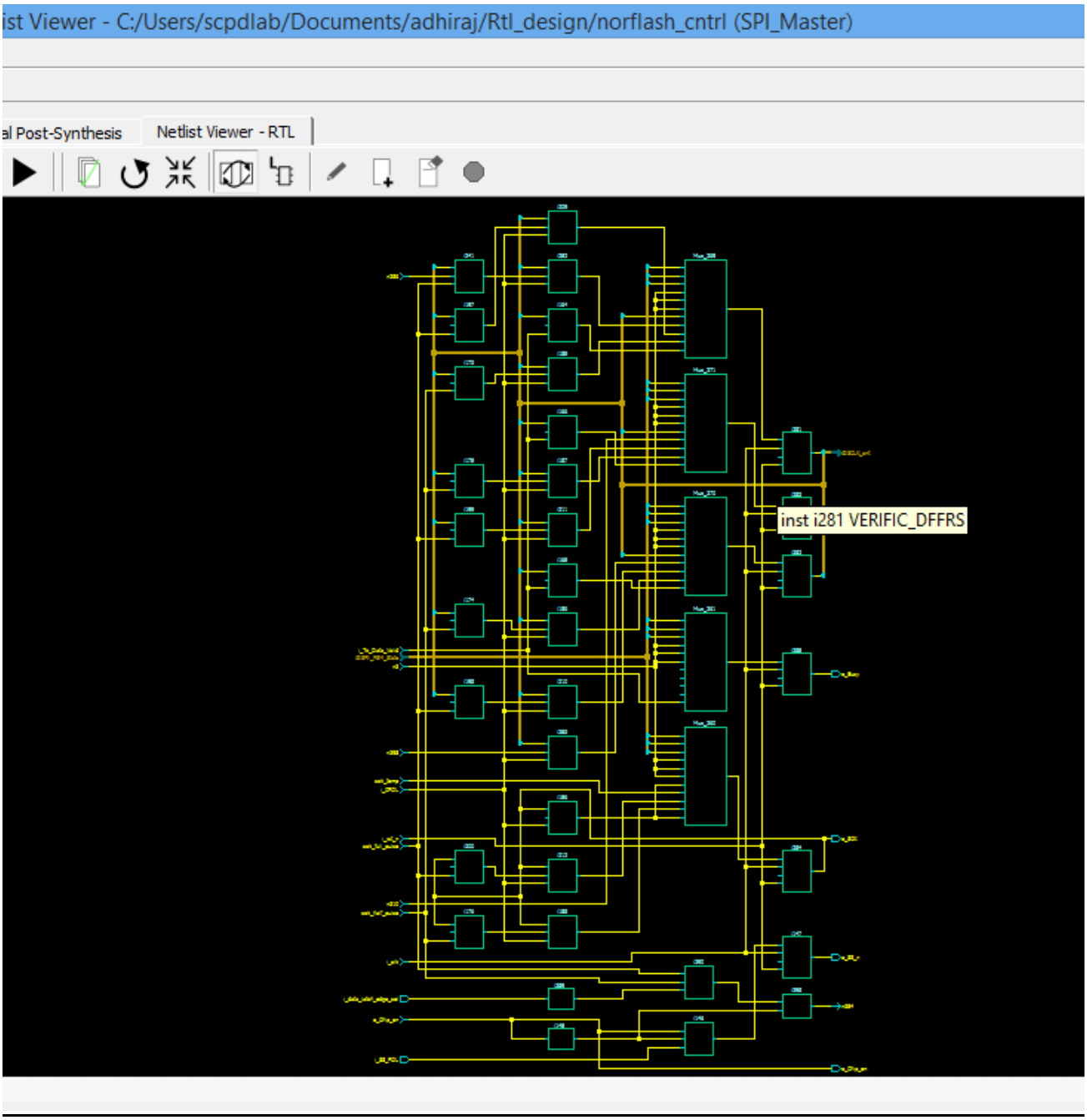
The SmartTime shows the delays taken by various paths. Fanout refers to the maximum number of output signals that are fed by the output equations of a logic cell.


SmartTime - [Maximum Delay - Expanded Path view :From: CLK_cnt[0]:CLK -> To: io0:EN]
File Edit View Tools Help
Summary for path
From: CLK_cnt[0]:CLK
To: io0:EN
Data Required Time (ns) 23.263
Data Arrival Time (ns) 8.718
Slack (ns) 14.545
Path Profile
Cell Delay 27.08%
Net Delay 72.92%
Name Type Net Macro Op Delay Total Fanout Edge
Data_arrival_time_calculation
i_clk
i_clk Clock source 0.000 0.000 r
BLOCK_INTERFACE_I_i_clk:A net I_NET_i_clk + 0.000 0.000 r
BLOCK_INTERFACE_I_i_clk:Y cell ADLIB:CFG1 + 0.051 0.051 2 r
i_clk_RNI23M3/U0_GB0:An net i_clk + 1.749 1.800 r
i_clk_RNI23M3/U0_GB0:Y cell ADLIB:GBR + 0.506 2.306 8 r
i_clk_RNI23M3/U0_RGB1_RGB2:An net i_clk_RNI23M3/U0_gbs_1 + 0.827 3.133 r
i_clk_RNI23M3/U0_RGB1_RGB2:YR cell ADLIB:RGB + 0.565 3.698 17 r
CLK_cnt[0]:CLK net i_clk_RNI23M3/U0_RGB1_RGB2_rgbr_net_1 + 0.700 4.398 r
CLK_cnt[0]:Q cell ADLIB:SLE_RT + 0.207 4.605 76 f
un54_i_wrrd_signal_0_a2_2:A net CLK_cnt[0] + 0.626 5.231 f
un54_i_wrrd_signal_0_a2_2:Y cell ADLIB:CFG4 + 0.277 5.508 2 r
un54_i_wrrd_signal_0_a2:B net un54_i_wrrd_signal_i_2 + 0.665 6.173 r
un54_i_wrrd_signal_0_a2:Y cell ADLIB:CFG3 + 0.136 6.309 5 r
io1_7_sqmuxa_1_2:A net un54_i_wrrd_signal_i + 0.193 6.502 r
io1_7_sqmuxa_1_2:Y cell ADLIB:CFG4 + 0.279 6.781 2 f
io0_2_sqmuxa_1_i_1:B net io1_7_sqmuxa_1_2 + 0.549 7.330 f
io0_2_sqmuxa_1_i_1:Y cell ADLIB:CFG4 + 0.070 7.400 1 f
io0_2_sqmuxa_1_i:B net io0_2_sqmuxa_1_i_1 + 0.247 7.647 f
io0_2_sqmuxa_1_i:Y cell ADLIB:CFG4 + 0.352 7.999 1 r
io0:EN net io0_2_sqmuxa_1_i + 0.719 8.718 r
data arrival time 8.718
Data required time calculation
i_clk
BLOCK_INTERFACE_I_i_clk
CFG1
i_clk_RNI23M3/U0_GB0
GBR
i_clk_RNI23M3/U0_RGB1_RGB2
RGB
CLK_cnt[0]
SLE_RT
un54_i_wrrd_signal_0_a2_2
CFG4
un54_i_wrrd_signal_0_a2
CFG3
io1_7_sqmuxa_1_2
CFG4
io0_2_sqmuxa_1_i_1
CFG4
io0_2_sqmuxa_1_i
CFG4
i_clk_RNI23M3/U0_RGB1_RGB0
RGB
io0
SLE_RT
Temp: -55 - 125 C
Volt: 1.14 - 1.26 V
Speed: -1
11:04 AM
15-Jun-22


SmartTime - [Maximum Delay Analysis View]
File Edit View Tools Help
Maximum Delay Analysis View
Analysis for scenario
timing_analysis
MAX
Summary
i_clk
Register to Register
External Setup
Clock to Output
Register to Asynchronous
External Recovery
Asynchronous to Register
Pin to Pin
Input to Output
User Sets
From *
TO *
Customize table
Apply Filter
Store Filter
Reset Filter
Source Pin
Sink Pin
Delay (ns)
Slack (ns)
Arrival (ns)
Required (ns)
Setup (ns)
Minimum Period (ns)
Skew (ns)
1 CLK_cnt[0]:CLK io0:EN 4.320 14.545 8.718 23.263 1.098 5.455 0.037
2 CLK_cnt[0]:CLK io2:EN 4.130 14.735 8.528 23.263 1.098 5.265 0.037
3 CLK_cnt[0]:CLK io1:EN 4.130 14.735 8.528 23.263 1.098 5.265 0.037
4 CLK_cnt[0]:CLK io3:EN 4.128 14.737 8.526 23.263 1.098 5.263 0.037
5 CLK_cnt[4]:CLK io0:EN 4.118 14.755 8.508 23.263 1.098 5.245 0.029
6 SCLK_cnt[4]:CLK SPI_FSM_State[0]:EN 4.103 14.760 8.496 23.256 1.098 5.240 0.039
7 CLK_cnt[7]:CLK io0:EN 4.103 14.770 8.493 23.263 1.098 5.230 0.029
8 CLK_cnt[8]:CLK io0:EN 4.061 14.812 8.451 23.263 1.098 5.188 0.029
9 CLK_cnt[6]:CLK io0:EN 4.025 14.848 8.415 23.263 1.098 5.152 0.029
10 SCLK_cnt[3]:CLK SPI_FSM_State[0]:EN 3.985 14.878 8.378 23.256 1.098 5.122 0.039
11 SCLK_cnt[2]:CLK SPI_FSM_State[0]:EN 3.959 14.904 8.352 23.256 1.098 5.096 0.039
12 CLK_cnt[5]:CLK io0:EN 3.920 14.953 8.310 23.263 1.098 5.047 0.029
13 CLK_cnt[1]:CLK io0:EN 3.907 14.966 8.297 23.263 1.098 5.034 0.029
14 CLK_cnt[2]:CLK io0:EN 3.894 14.979 8.284 23.263 1.098 5.021 0.029
Name Type Net Macro Op Delay Total Fanout Edge
Summary
data required time 23.263
data arrival time - 8.718
slack 14.545
Data_arrival_time_calculation
i_clk 0.000 0.000
i_clk Clock source + 0.000 0.000 r
BLOCK_INTERFACE_I_i_clk:A net I_NET_i_clk + 0.000 0.000 r
BLOCK_INTERFACE_I_i_clk:Y cell ADLIB:CFG1 + 0.051 0.051 2 r
i_clk_RNI23M3/U0_GB0:An net i_clk + 1.749 1.800 r
i_clk_RNI23M3/U0_GB0:Y cell ADLIB:GBR + 0.506 2.306 8 r
i_clk_RNI23M3/U0_RGB1_RGB2:An net i_clk_RNI23M3/U0_gbs_1 + 0.827 3.133 r
i_clk_RNI23M3/U0_RGB1_RGB2:YR cell ADLIB:RGB + 0.565 3.698 17 r
CLK_cnt[0]:CLK net i_clk_RNI23M3/U0_RGB1_RGB2_rgbr_net_1 + 0.700 4.398 r
# of paths
90
72
54
36
18
0
14.149
14.545
14.941
15.337
slack distribution(ns)
Ready
Temp: -55 - 125 C
Volt: 1.14 - 1.26 V
Speed: -1
11:20 AM
15-Jun-22

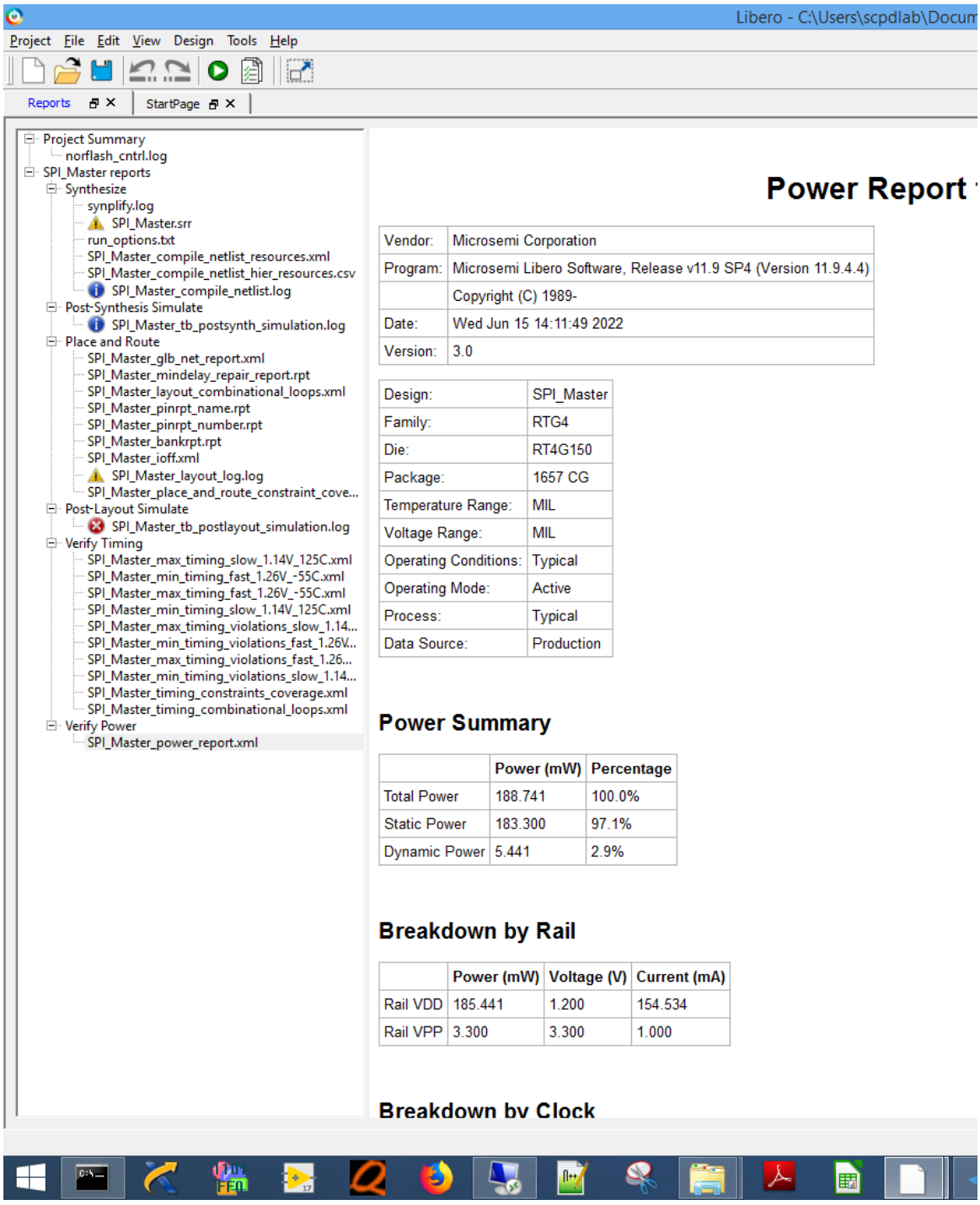
Schematic Viewer:

Using NetlistViewer tool of the software (LiberoSOC), the electronic components inside our design and their design can be viewed.



Power Analysis:

Software Used: LiberoSOC v11.9



At last, a bitstream programming file is generated by the software which gets programmed into the FPGA.

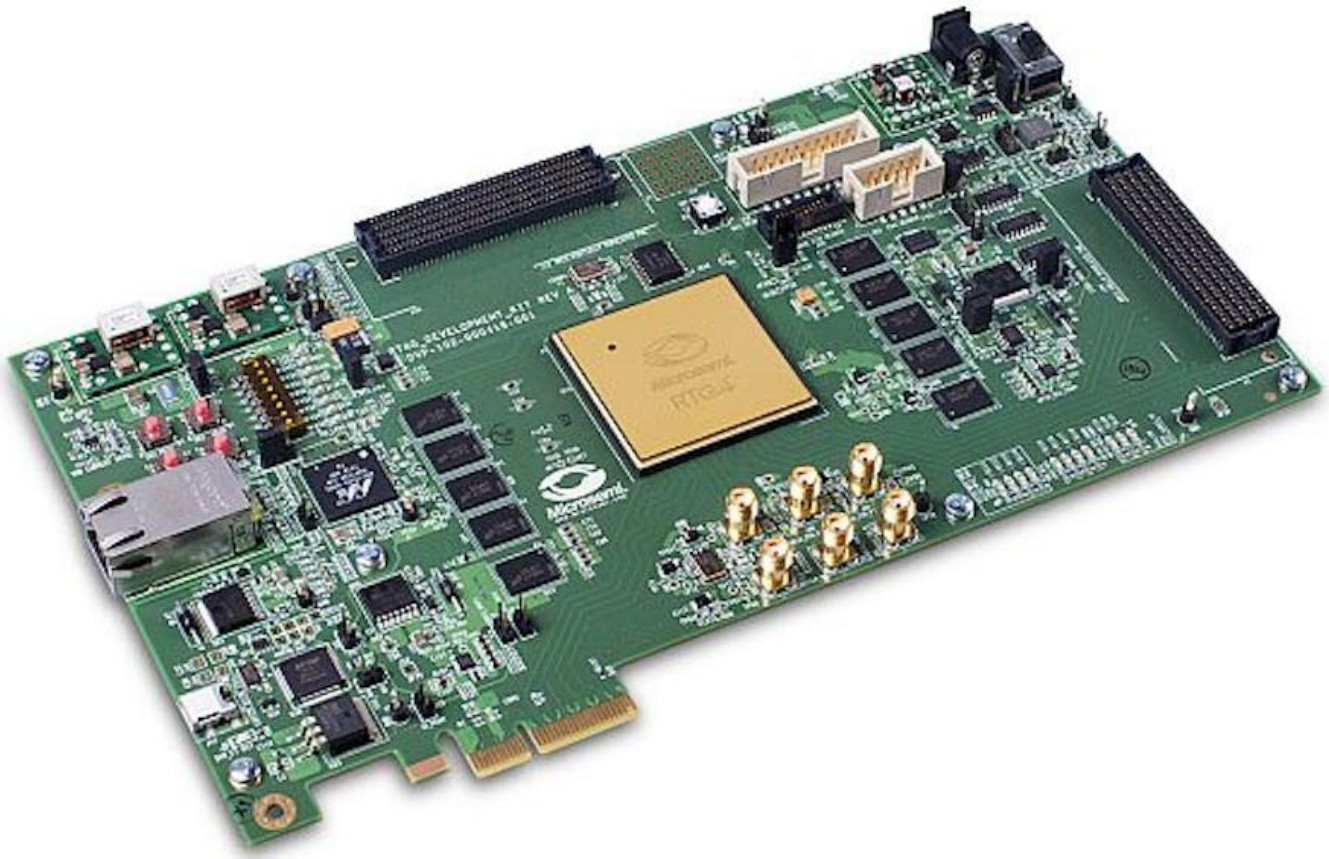


Photo of the RTG4 FPGA board.

CHAPTER - 7

C O N C L U D I N G R E M A R K S

In this project, a QSPI NOR Flash Controller has been designed for the NOR Flash memory device 3DFS256M04VS2801. The Controller has been implemented using VHDL and tested.
The Controller has been observed to follow correctly the specifications given. Softwares used in this project are QuestaSim 10.1d and Modelsim Libero v11.9.

Through this project, my VHDL coding skills have improved. Exposure to all the processes helped me in understanding the design flow in a better way. Also got hands on experience in various tools and softwares.

B I B L I O G R A P H Y

1) VHDL. Programming by Example. 4th Ed - Douglas Perry
2) Introduction to logic circuits & logic design with VHDL. 2nd Ed - Brock J. Lamare's
3) Toshiba - NAND vs. NOR Flash Memory Technology Overview
4) Data sheet of QSPI NOR Flash 3DFS256M04VS2801
5) Data sheet of RTG4 FPGA