

**DESIGN AND IMPLEMENTATION OF A SPI CONTROLLER FOR ZIGBEE
MODULE
BY
YONG MIN AN**

**A REPORT
SUBMITTED TO
Universiti Tunku Abdul Rahman
in partial fulfillment of the requirements
for the degree of
BACHELOR OF INFORMATION TECHNOLOGY (HONS)
COMPUTER ENGINEERING
Faculty of Information and Communication Technology
(Kampar Campus)**

JAN 2020

REPORT STATUS DECLARATION FORM

Title: DESIGN AND IMPLEMENTATION OF A SPI CONTROLLER
FOR ZIGBEE MODULE

Academic Session: JAN 2020

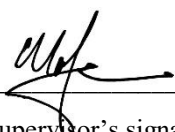
I YONG MIN AN
(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in
Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

YONG MIN AN
(Author's signature)


(Supervisor's signature)

Address:
No 2359, Seksyen 2/10 Bandar
Barat, Cambridge, 31900
Kampar, Ipoh

MOK KAI MING
Supervisor's name

Date: 24/4/2020

Date: 24/4/2020

**DESIGN AND IMPLEMENTATION OF A SPI CONTROLLER FOR ZIGBEE
MODULE**

**BY
YONG MIN AN**

**A REPORT
SUBMITTED TO
Universiti Tunku Abdul Rahman
in partial fulfillment of the requirements
for the degree of
BACHELOR OF INFORMATION TECHNOLOGY (HONS)
COMPUTER ENGINEERING
Faculty of Information and Communication Technology
(Kampar Campus)**

JAN 2020

DECLARATION OF ORIGINALITY

I declare that this report entitled “**DESIGN AND IMPLEMENTATION OF A SPI CONTROLLER FOR ZIGBEE MODULE**” is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature : _____ YONG MIN AN _____

Name : _____ YONG MIN AN _____

Date : _____ 24/4/2020 _____

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest appreciation to my project supervisor, Mr Mok Kai Ming for his constant supervision, constructive suggestions, as well as invaluable guidance and encouragement in helping me to complete this project. I am really thankful to him because I came to know and learn a lot of new things during the project development. Again, thank you for precious your time and patience along the duration of my project.

Apart from that, I would also like to express my special gratitude and thanks to my beloved family members, especially my parents who have been giving me endless care and support, both spiritually and financially over the years. Thank you for always supporting me in all the good and bad times throughout my life.

Last but not least, I would like to extend my sincere thanks to all my friends and seniors who have willingly helped me out with their abilities, providing necessary and useful information about the project. All the advises and suggestions had contributed to the completion of this project. Thank you for all of your generous helps and kindness.

ABSTRACT

This project is about the 4-wire Serial Peripheral Interface (SPI) controller unit design and implementation for academic purpose. The development of this project will begin with the design of the SPI controller unit. The RTL design flow will be used throughout the project development and the micro-architectural level design will be focused more as the SPI controller to be designed is in the unit level. The internal blocks of the SPI controller unit will be modeled by using Verilog HDL before they are integrated into unit level. The specifications of the SPI controller unit and its internal block will be functionally verified by writing testbenches in Verilog HDL.

After the SPI controller unit has been functionally verified, it will be integrated into the existing RISC32 pipelined processor developed in UTAR. This involves the development of the interface between the SPI controller and the RISC32 based on I/O memory mapping technique. Moving on, an Interrupt Service Routine (ISR) will be specifically developed and implemented on the RISC32 for handling the data received by the SPI controller. A MIPS test program will also be written to test the correctness of the ISR functionalities.

Lastly, it will be synthesized on the Field Programmable Gate Array (FPGA) technology and further interfaced with CC2420 RF transceiver in this project for wireless data communication. The CC2420 will be configured as the slave device whereas the SPI controller unit will be used as the master device. Data communication between the SPI controller unit in the RISC32 pipelined processor and the CC2420 RF transceiver is performed via a simple 4-wire SPI compatible interface (MOSI, MISO, SCLK and SS pin). In short, a piece of software, stimulation result and hardware are expected to be delivered at the end of the project.

TABLE OF CONTENTS

TITLE PAGE	i
DECLARATION OF ORIGINALITY	ii
ACKNOWLEDGEMENTS.....	iii
ABSTRACT	iv
TABLE OF CONTENTS	v
LIST OF FIGURES.....	xi
LIST OF TABLES.....	xvii
LIST OF ABBREVIATIONS.....	xix
CHAPTER 1: INTRODUCTION	1
1.1 Background Information	1
1.1.1 MIPS.....	1
1.1.2 Bus.....	1
1.1.3 SPI	2
1.1.4 Zigbee	3
1.2 Motivation	3
1.3 Problem Statement	5
1.4 Project Scope	5
1.5 Project Objectives	6
1.6 Impact, Significance, and Contribution.....	6
1.7 Report Organization	7
CHAPTER 2: LITERATURE REVIEW.....	9
2.1 Overview of 4-wire SPI Protocol.....	9
2.1.1 Detailed Pin Description in 4-wire SPI Protocol	9
2.1.2 Transfer Modes in 4-wire SPI Protocol.....	10
2.1.3 Timing Diagram in 4-wire SPI Protocol.....	11
2.1.4 Working Principal of 4-wire SPI Protocol.....	13
2.2 Overview of 3-wire SPI Protocol.....	16
2.2.1 Detailed Pin Description of 3-wire SPI Protocol	16
2.2.2 Transfer Modes of 3-wire SPI Protocol.....	17
2.2.3 Timing Diagram of 3-wire SPI Protocol	18

2.2.4 Working Principal of 3-wire SPI Protocol.....	18
2.3 SPI Controller	20
2.3.1 SPI Controller from Motorola Inc.....	20
2.3.2 SPI Controller Designed by Kiat Wei Pau.....	22
2.4 Memory-mapped I/O.....	23
CHAPTER 3: PROPOSED METHOD/APPROACH.....	25
3.1 Methodologies and General Work Procedures	25
3.1.1 RTL Design Flow.....	25
3.1.2 Micro-architecture Specification.....	26
3.1.3 RTL Modeling and Verification.....	26
3.1.4 Logic Synthesis for FPGA.....	27
3.2 Design Tools.....	27
3.2.1 ModelSim PE Student Edition 10.4a.....	28
3.2.2 Xilinx Vivado Design Suite	28
3.2.3 PCSpim.....	29
3.3 Technologies Involved	29
3.3.1 Field Programmable Gate Array (FPGA).....	29
3.3.2 Zigbee RF Transceiver	30
3.4 Implementation Issues and Challenges	30
3.5 Timeline.....	31
3.5.1 Gantt Chart for Project 1.....	31
3.5.2 Gantt Chart for Project 2.....	31
CHAPTER 4: SYSTEM SPECIFICATION.....	32
4.1 System Overview of the RISC32 Pipeline Processor	32
4.1.1 RISC32 Pipeline Processor Architecture.....	32
4.1.2 Functional View of the RISC32 Pipeline Processor	33
4.1.3 Memory Map of the RISC32 Pipeline Processor	34
4.2 Chip Interface of the RISC32 Pipeline Processor.....	36
4.3 Input Pin Description of the RISC32 Pipeline Processor	36
4.4 Output Pin Description of the RISC32 Pipeline Processor	37
4.5 Input Output Pin Description of the RISC32 Pipeline Processor	37

CHAPTER 5: MICRO-ARCHITECTURE SPECIFICATION	38
5.1 SPI Controller Unit	38
5.1.1 Functionality/Feature of the SPI Controller Unit.....	38
5.1.2 Operating Procedure (External Operation).....	39
5.1.3 Unit Interface of the SPI Controller Unit	41
5.1.4 Input Pin Description of the SPI Controller Unit.....	42
5.1.5 Output Pin Description of the SPI Controller Unit	43
5.1.6 Input Output Pin Description of the SPI Controller Unit	43
5.1.7 Internal Operation of the SPI Controller Unit.....	44
5.1.8 Design Partitioning of the SPI Controller Unit.....	45
5.1.9 Micro-Architecture of the SPI Controller Unit (Block Level).....	48
5.2 SPI Transmitter Block.....	50
5.2.1 Functionality/Feature of the SPI Transmitter Block	50
5.2.2 Block Interface of the SPI Transmitter Block.....	50
5.2.3 Input Pin Description of the SPI Transmitter Block	50
5.2.4 Output Pin Description of the SPI Transmitter Block.....	52
5.2.5 Finite State Machine of the SPI Transmitter Block	53
5.3 SPI Receiver Block	54
5.3.1 Functionality/Feature of the SPI Receiver Block.....	54
5.3.2 Block Interface of the SPI Receiver Block.....	54
5.3.3 Input Pin Description of the SPI Receiver Block.....	54
5.3.4 Output Pin Description of the SPI Receiver Block	55
5.3.5 Finite State Machine of the SPI Receiver Block.....	56
5.4 SPI Clock Generator Block	57
5.4.1 Functionality/Feature of the SPI Clock Generator Block.....	57
5.4.2 Block Interface of the SPI Clock Generator Block	57
5.4.3 Input Pin Description of the SPI Clock Generator Block.....	57
5.4.4 Output Pin Description of the SPI Clock Generator Block	58
5.5 16-deep Asynchronous FIFO Block	59
5.5.1 Functionality/Feature of the 16-deep Asynchronous FIFO Block.....	59
5.5.2 Block Interface of the 16-deep Asynchronous FIFO Block	59
5.5.3 Input Pin Description of the 16-deep Asynchronous FIFO Block.....	60
5.5.4 Output Pin Description of the 16-deep Asynchronous FIFO Block ...	60
5.5.5 Schematic and Block Diagram of the 16-deep Asynchronous FIFO block	61

5.6 2-deep FIFO Synchronizer Block	62
5.6.1 Functionality/Feature of the 2-deep FIFO Synchronizer Block.....	62
5.6.2 Block Interface of the 2-deep FIFO Synchronizer Block.....	62
5.6.3 Input Pin Description of the 2-deep FIFO Synchronizer Block.....	63
5.6.4 Output Pin Description of the 2-deep FIFO Synchronizer Block.....	63
5.6.5 Schematic and Block Diagram of the 2-deep FIFO Synchronizer Block	65
5.7 Register Set of SPI Controller Unit.....	66
5.7.1 SPI Configuration Register (SPICR).....	66
5.7.2 SPI Status Register (SPISR)	67
5.7.3 SPI Transmitter Data register (SPITDR).....	69
5.8.4 SPI Receiver Data register (SPIRDR).....	70
CHAPTER 6: FIRMWARE DEVELOPMENT	71
6.1 Exception Handler of the RISC32 Pipeline Processor	71
6.2 Interrupt Service Routine (ISR) of the SPI Controller Unit.....	72
CHAPTER 7: VERIFICATION SPECIFICATION AND STIMULATION RESULT	74
7.1 Test Plan for SPI Controller Unit's Functional Test.....	74
7.2 Stimulation Results of the SPI Controller Unit's Functional Test.....	83
7.2.1 Test Case #1: System Reset	83
7.2.2 Test Case #2: Write Operation on SPISR.....	83
7.2.3 Test Case #3: Write Operation on SPICR	84
7.2.4 Test Case #4: Transmitter Buffer Empty Interrupt Support	85
7.2.5 Test Case #5: Push One 8-bit Data into the TX_buffer16x8.....	85
7.2.6 Test Case #6: Mode 0 Serial Data Communication	86
7.2.7 Test Case #7: Receiver Buffer Full Interrupt Support After Receiving A 1-byte Data (RXFM = 0).....	87
7.2.8 Test Case #8: Pop 1-byte of Received Data from the RX_buffer16x887	
7.2.9 Test Case #9: Receiver Buffer Full Interrupt Support After Receiving 16x1-byte Data (RXFM = 1).....	88
7.2.10 Test Case #10: Pop 16 Number of 1-byte Data from the RX_buffer16x8.....	89
7.2.11 Test Case #11: Mode 1 Serial Data Communication	90
7.2.12 Test Case #12: Mode 2 Serial Data Communication	90

7.2.13 Test Case #13: Mode 3 Serial Data Communication	91
7.2.14 Test Case #14: Selectable Transmission Speed (Baud Rate).....	92
7.2.15 Test Case #15: Mode Fault Error Interrupt Support.....	93
7.4 Test Plan for SPI Controller Unit’s Integration Test with RISC32	94
7.5 MIPS Test Program for c_risc_dut in Integration Test	99
7.6 MIPS Test Program for c_risc_client in Integration Test	102
7.7 Stimulation Results of the SPI Controller Unit’s Integration Test with RISC32	105
7.7.1 Test Case #1: System Reset	105
7.7.2 Test Case #2: Transmitter Buffer Empty Interrupt Support	105
7.7.3 Test Case #3: Mode 0 Serial Data Communication	110
7.7.4 Test Case #4: Receiver Buffer Full Interrupt Support.....	112
7.7.5 Test Case #5: Mode 1 Serial Data Communication	116
7.7.6 Test Case #6: Mode 2 Serial Data Communication	117
7.7.7 Test Case #7: Mode 3 Serial Data Communication	118
7.7.8 Test Case #8: Mode Fault Error Interrupt Support	119
CHAPTER 8: SYNTHESIS AND IMPLEMENTATION	122
8.1 FPGA Resources Utilization of the Synthesized SPI Controller Unit	122
8.2 Timing Analysis.....	123
8.2.1 Timing Analysis of the On-board SPI Controller Unit	123
8.2.2 Timing Analysis of the RISC32 with the SPI Controller Unit	124
8.3 Proposed Hardware Implementation.....	125
8.4 Proposed Software Implementation	130
8.4.1 Flowchart of the Hardware/Software Behaviors in c_risc_dut.....	132
8.4.2 Flowchart of the Hardware/Software Behaviors in c_risc_client	134
CHAPTER 9: CONCLUSION AND FUTURE WORK	136
9.1 Conclusion.....	136
9.2 Future Work.....	137
BIBLIOGRAPHY.....	138
APPENDIX A: TIMING DIAGRAM.....	A-1
A.1 Timing diagram of different SPI’s Transfer Modes.....	A-1

APPENDIX B: TESTBENCH.....	B-1
B.1 Testbench for SPI Controller Unit’s Functional Test.....	B-1
B.2 Testbench for SPI Controller Unit’s Integration Test with RISC32	B-10
APPENDIX C: CC2420.....	C-1
C.1 Pin Assignment of the CC2420.....	C-1
C.2 Overview of the External Components Used with the CC2420	C-3
C.3 Bill of Materials for the Application Circuits	C-3
POSTER	D-1
PLAGIARISM CHECK RESULT.....	E-1
CHECKLIST	F-1

LIST OF FIGURES

Figure Number	Title	Page
Figure 1.1.1.1	Conventional pipeline execution representation.	1
Figure 1.1.2.1	An overview of various type of buses in the computer system.	2
Figure 2.1.3.1	Timing diagram for mode 0 serial data communication in the 4-wire SPI protocol.	11
Figure 2.1.3.2	Timing diagram for mode 1 serial data communication in the 4-wire SPI protocol.	12
Figure 2.1.3.3	Timing diagram for mode 2 serial data communication in the 4-wire SPI protocol.	12
Figure 2.1.3.4	Timing diagram for mode 3 serial data communication in the 4-wire SPI protocol.	13
Figure 2.1.4.1	An overview of the block diagram connection between a master device and a slave device in 4-wire SPI protocol.	14
Figure 2.1.4.2	An overview of the block diagram connection between a single master and multiple slave devices using independent slave configuration.	15
Figure 2.1.4.3	An overview of the block diagram connection between a single master device and multiple slave devices using Daisy-chain configuration.	15
Figure 2.2.3.1	Timing diagram for mode 0, 1, 2, and 3 serial data communication in the 3-wire SPI protocol.	18
Figure 2.2.4.1	An overview of the block diagram connection between a master device and a slave device in the 3-wire SPI protocol.	19
Figure 2.3.1.1	An overview on the Motorola Inc's SPI controller.	20
Figure 2.3.2.1	An overview of the SPI controller designed by Kiat Wei Pau	23
Figure 3.1.1.1	The RTL design flow used for developing the SPI controller unit is provided. The arrows indicate process or work flow (not data flow).	25
Figure 3.3.1.1	The top view of the Nexys 4 DDR (XC7A100T)	29

Figure 3.3.2.1	The top view of the CC2420 from the Texas Instruments company.	30
Figure 3.5.1.1.	Gantt chart for Project 1	31
Figure 3.5.2.1	Gantt chart for Project 2	31
Figure 4.1.1.1	An overview on the architecture of the RISC32 pipeline processor.	33
Figure 4.1.2.1	The functional view of the RISC32 pipeline processor.	34
Figure 4.1.3.1	Memory map of the RISC32 pipeline processor.	35
Figure 4.2.1	Chip interface of the RISC32 pipeline processor.	36
Figure 5.1.3.1	SPI controller unit interface.	41
Figure 5.1.8.1	Block-level partitioning of the SPI controller unit.	46
Figure 5.1.9.1	Simplified micro-architecture of the SPI controller unit.	48
Figure 5.1.9.3	Datapath of the SPI controller unit	49
Figure 5.2.2.1	Block interface of the SPI transmitter block.	50
Figure 5.2.5.1	Finite state machine of the SPI transmitter block.	53
Figure 5.3.2.1	Block interface of the SPI receiver block.	54
Figure 5.3.5.1	Finite state machine of the SPI receiver block.	56
Figure 5.4.2.1	Block interface of the SPI clock generator block.	57
Figure 5.5.2.1	Block interface of 16-deep asynchronous FIFO block.	59
Figure 5.5.5.1	Schematic and block diagram of the 16-deep asynchronous FIFO design with asynchronous comparisons.	61
Figure 5.6.2.1	Block interface of 2-deep FIFO synchronizer block	62
Figure 5.6.5.1	Schematic and block diagram of the 2-deep FIFO synchronizer block	65
Figure 5.7.1	Address of the special-purpose registers in virtual memory	66
Figure 5.8.1.1	SPI Configuration Register (SPICR)	66
Figure 5.8.2.1	SPI Status Register (SPISR)	68
Figure 5.8.3.1	SPI Transmitter Data Register (SPITDR)	69
Figure 5.8.4.1	SPI Receiver Data Register (RDR)	70
Figure 7.1.1	The connection mechanism of the DUT_MASTER and the DUT_SLAVE for SPI controller unit's functional verification.	74
Figure 7.2.1.1	Stimulation result for test case #1 using ModelSim stimulator.	83

Figure 7.2.2.1	Stimulation result for test case #2 using ModelSim stimulator.	83
Figure 7.2.3.1	Stimulation result for test case #3 using ModelSim stimulator.	84
Figure 7.2.4.1	Stimulation result for test case #4 using ModelSim stimulator.	85
Figure 7.2.5.1	Stimulation result for test case #5 using ModelSim stimulator.	85
Figure 7.2.6.1	Stimulation result for test case #6 using ModelSim stimulator.	86
Figure 7.2.7.1	Stimulation result for test case #7 using ModelSim stimulator.	87
Figure 7.2.8.1	Stimulation result for test case #8 using ModelSim stimulator.	87
Figure 7.2.9.1	Stimulation result for test case #9 using ModelSim stimulator.	88
Figure 7.2.9.2	Stimulation result for test case #9 using ModelSim stimulator (cont'd).	88
Figure 7.2.10.1	Stimulation result for test case #10 using ModelSim stimulator.	89
Figure 7.2.11.1	Stimulation result for test case #11 using ModelSim stimulator	90
Figure 7.2.12.1	Stimulation result for test case #12 using ModelSim stimulator	90
Figure 7.2.13.1	Stimulation result for test case #13 using ModelSim stimulator	91
Figure 7.2.14.1	Stimulation result for test case #14 with SCLK clock signal is 4 times slower than the I/O clock of the DUY_MASTER.	92
Figure 7.2.14.2	Stimulation result for test case #14 with SCLK clock signal is 8 times slower than the I/O clock of the DUY_MASTER.	92
Figure 7.2.14.3	Stimulation result for test case #14 with SCLK clock signal is 16 times slower than the I/O clock of the DUY_MASTER.	92
Figure 7.2.15.1	Stimulation result for test case #15 using ModelSim stimulator.	93
Figure 7.3.1	The connection mechanism of the c_risc_dut, c_risc_client, SPI_flash_dut, and SPI_flash_client for SPI controller unit's integration test with RISC32.	94
Figure 7.6.1.1	Stimulation result for test case #1 using Vivado stimulator.	105
Figure 7.6.2.1	Stimulation result of c_risc_dut for test case #2 using Vivado stimulator.	105
Figure 7.6.2.2	Stimulation result of c_risc_dut for test case #2 using Vivado stimulator (cont'd).	106

Figure 7.6.2.3	Stimulation result of c_risc_dut for test case #2 using Vivado stimulator (cont'd).	106
Figure 7.6.2.4	Stimulation result of c_risc_dut for test case #2 using Vivado stimulator (cont'd).	106
Figure 7.6.2.5	Stimulation result of c_risc_client for test case #2 using Vivado stimulator.	108
Figure 7.6.2.6	Stimulation result of c_risc_client for test case #2 using Vivado stimulator (cont'd).	108
Figure 7.6.2.7	Stimulation result of c_risc_client for test case #2 using Vivado stimulator (cont'd).	108
Figure 7.6.2.8	Stimulation result of c_risc_client for test case #2 using Vivado stimulator (cont'd).	109
Figure 7.6.3.1	Stimulation result for test case #3 using Vivado stimulator.	110
Figure 7.6.4.1	Stimulation result of c_risc_dut for test case #4 using Vivado stimulator.	112
Figure 7.6.4.2	Stimulation result of c_risc_dut for test case #4 using Vivado stimulator (cont'd).	112
Figure 7.6.4.3	Stimulation result of c_risc_dut for test case #4 using Vivado stimulator (cont'd).	112
Figure 7.6.4.4	Stimulation result of c_risc_dut for test case #4 using Vivado stimulator.	113
Figure 7.6.4.5	Stimulation result of c_risc_client for test case #4 using Vivado stimulator.	114
Figure 7.6.4.6	Stimulation result of c_risc_client for test case #4 using Vivado stimulator (cont'd).	114
Figure 7.6.4.7	Stimulation result of c_risc_client for test case #4 using Vivado stimulator (cont'd).	114
Figure 7.6.4.8	Stimulation result of c_risc_client for test case #4 using Vivado stimulator (cont'd).	115
Figure 7.6.5.1	Stimulation result for test case #5 using Vivado stimulator.	116
Figure 7.6.6.1	Stimulation result for test case #6 using Vivado stimulator.	117
Figure 7.6.7.1	Stimulation result for test case #7 using Vivado stimulator.	118
Figure 7.6.8.1	Stimulation result for test case #8 using Vivado stimulator.	119

Figure 7.6.8.2	Stimulation result of c_risc_dut for test case #8 using Vivado stimulator.	119
Figure 7.6.8.3	Stimulation result of c_risc_dut for test case #8 using Vivado stimulator (cont'd).	119
Figure 7.6.8.4	Stimulation result of c_risc_dut for test case #8 using Vivado stimulator (cont'd).	120
Figure 7.6.8.5	Stimulation result of c_risc_dut for test case #8 using Vivado stimulator (cont'd).	120
Figure 8.1.1	Resource utilization report of the synthesized SPI controller unit on the Nexys 4 DDR (XC7A100T) board.	122
Figure 8.1.2	Resource utilization summary of the synthesized SPI controller unit on the Nexys 4 DDR (XC7A100T) board.	122
Figure 8.2.1.1	Timing report of the on-board SPI controller unit.	123
Figure 8.2.1.2	Timing report of the on-board SPI controller unit.	124
Figure 8.2.2.1	Design timing summary of the entire RISC32 pipeline processor.	125
Figure 8.2.2.2	Top 10 paths in the RISC32 pipeline processor that have the largest total data path delay.	125
Figure 8.3.1	Detailed descriptions about the connection mechanism of the RISC32 pipeline processor with the CC2420 transceiver by using only few external components.	127
Figure 8.3.2	Connection mechanism of the c_risc_dut with the CC2420 transceiver for wireless communication.	128
Figure 8.3.3	Connection mechanism of the c_risc_client with the CC2420 transceiver for wireless communication	129
Figure 8.4.1	Expected wireless communication between the Zigbee end device and the Zigbee coordinator in this project	131
Figure 8.4.1.1	Flowchart of hardware/software behaviors in c_risc_dut for on-board testing.	132
Figure 8.4.1.2	Flowchart of hardware/software behaviors in c_risc_dut for on-board testing (cont'd).	133
Figure 8.4.2.1	Flowchart of hardware/software behaviors in c_risc_client for on-board testing.	134

Figure 8.4.2.2	Flowchart of hardware/software behaviors in c_risc_client for on-board testing (cont'd).	135
Figure A.1.1	Timing diagram for mode 0 serial data communication.	A-1
Figure A.1.2	Timing diagram for mode 1 serial data communication.	A-1
Figure A.1.3	Timing diagram for mode 2 serial data communication.	A-1
Figure A.1.4	Timing diagram for mode 3 serial data communication.	A-1
Figure C.1.1	Top view of the CC2420 pinout	C-1
Figure C.1.2	Pin description of the CC2420.	C-1
Figure C.1.3	Pin description of the CC2420 (cont'd).	C-2
Figure C.2.1	Description of the external components used with the CC2420.	C-3
Figure C.3.1	List of materials for the application circuit.	C-3

LIST OF TABLES

Table Number	Title	Page
Table 2.1.1.1	Functional descriptions of the standard 4-wire SPI's external pins.	9
Table 2.1.2.1	Functional descriptions of the Clock Polarity and Clock Phase parameters based on Motorola Inc.'s SPI Block Guide V03.06.	10
Table 2.1.2.2	SPI transfer mode information based on Motorola Inc.'s SPI Block Guide V03.06.	11
Table 2.2.1.1	Functional description of the 3-wire SPI's external pins.	17
Table 2.3.2.1	Normal mode and bidirectional mode in Motorola Inc.'s SPI controller.	21
Table 2.4.1	Three general types of special-purpose register used in MMIO	24
Table 3.2.1	Comparison among 3 different Verilog stimulators.	27
Table 4.1.1.1	Specification of the RISC32 pipeline processor.	33
Table 4.1.3.1	Memory map description of the RISC32 pipeline processor.	35
Table 4.3.1	Input pin description of the RISC32 pipeline processor.	36
Table 4.4.1	Output pin description of the RISC32 pipeline processor.	37
Table 4.5.1	Input output pin description of the RISC32 pipeline processor.	37
Table 5.1.1.1	Pin direction of the SPI standard pins when it is set as a master or slave device.	38
Table 5.1.4.1	Input pin description of the SPI controller unit.	42
Table 5.1.5.1	Output pin description of the SPI controller unit.	43
Table 5.1.6.1	Input output pin description of the SPI controller unit.	43
Table 5.1.7.1	Functional description of the SPI controller's write operation.	44
Table 5.1.7.2	Functional description of the SPI controller's read operation.	44
Table 5.1.8.1	Functional description of each SPI internal block.	46
Table 5.2.3.1	Input pin description of the SPI transmitter block.	50
Table 5.2.4.1	Output pin description of the SPI transmitter block.	52

Table 5.2.5.1	State description of the SPI transmitter block.	53
Table 5.3.3.1	Input pin description of the SPI receiver block.	54
Table 5.3.4.1	Output pin description of the SPI receiver block.	55
Table 5.3.5.1	State description of the SPI receiver block.	56
Table 5.4.3.1	Input pin description of the SPI clock generator block.	57
Table 5.4.4.1	Output pin description of the SPI clock generator block.	58
Table 5.5.3.1	Input pin description of the 16-deep asynchronous FIFO block.	60
Table 5.5.4.1	Output pin description of the 16-deep asynchronous FIFO block.	60
Table 5.6.3.1	Input pin description of the 2-deep FIFO synchronizer block.	63
Table 5.6.4.1	Output pin description of the 2-deep FIFO synchronizer block.	63
Table 7.1.1	Instance name of each SPI controller unit and its internal blocks that are being used in the test plans, testbenches, flowcharts and stimulation.	75
Table 7.1.2	Test plan for the SPI controller unit's functional verification.	75
Table 7.3.1	Test plan for the SPI controller unit's integration test with RISC32 pipeline processor.	95
Table 7.4.1	MIPS test program for c_risc_dut in integration test	99
Table 7.5.1	MIPS test program for c_risc_client in integration test	102

LIST OF ABBREVIATIONS

<i>ADC</i>	Analog-to-digital Converter
<i>CDC</i>	Clock Domain Crossing
<i>CISC</i>	Complex Instruction Set Computer
<i>CPHA</i>	Clock Phase
<i>CPO</i>	Coprocessor 0
<i>CPOL</i>	Clock Polarity
<i>CPU</i>	Central Processing Unit
<i>DAC</i>	Digital-to-analog Converter
<i>DMA</i>	Direct Memory Access
<i>DSSS</i>	Direct Sequence Spread Spectrum
<i>EDA</i>	Electronic Design Automation
<i>EX</i>	Execute
<i>FPGA</i>	Field Programmable Gate Array
<i>FSM</i>	Finite State Machine
<i>GPIO</i>	General-Purpose Input Output
<i>GPR</i>	General-Purpose Register
<i>HDL</i>	Hardware Description Language
<i>IC</i>	Integrated Circuit
<i>ID</i>	Instruction Decode and Operand Fetch
<i>IF</i>	Instruction Fetch
<i>ISA</i>	Instruction Set Architecture
<i>ISR</i>	Interrupt Service Routine
<i>MAC</i>	Media Access Control
<i>MCO</i>	Movement Control Order
<i>MEM</i>	Memory Access
<i>M2M</i>	Machine-to-machine
<i>MIPS</i>	Microprocessor without Interlocked Pipeline Stages
<i>MOMI</i>	Master In Master Out
<i>MOSI</i>	Master Out Serial In
<i>MISO</i>	Master In Serial Out

<i>PCB</i>	Printed Circuit Board
<i>RAM</i>	Random Access Board
<i>RAW</i>	Read-After-Write
<i>RF</i>	Radio Frequency
<i>RISC</i>	Reduced Instruction Set Computer
<i>ROM</i>	Read Only Memory
<i>RSR</i>	Receiver Shift Register
<i>RTL</i>	Register Transfer Level
<i>RX</i>	Receive
<i>SCLK</i>	Serial Clock
<i>SISO</i>	Slave In Slave Out
<i>SS</i>	Slave Select
<i>SPI</i>	Serial Peripheral Interface
<i>SPICR</i>	SPI Configuration Register
<i>SPIRDR</i>	SPI Receiver Data Register
<i>SPISR</i>	SPI Status Register
<i>SPITDR</i>	SPI Transmitter Data Register
<i>TSR</i>	Transmitter Shift Register
<i>TX</i>	Transmit
<i>UART</i>	Universal Asynchronous Receiver-Transmitter
<i>USB</i>	Universal Serial Bus
<i>VHDL</i>	VHSIC Hardware Description Language
<i>VLSI</i>	Very Large-Scale Integration
<i>WB</i>	Write Back
<i>WNS</i>	Worse Negative Slack

Chapter 1: Introduction

1.1 Background Information

An overview of the project fields that matter is provided in the following sections to help identify and understand some facts or knowledge related to this project.

1.1.1 MIPS

MIPS is the abbreviation of Microprocessor without Interlocked Pipeline Stages. It is a Reduced Instruction Set Computer (RISC) architecture developed by MIPS Technologies (UKEssays, 2018). As opposed to the complex instruction set and large number of addressing modes used in Complex Instruction Set Computer (CISC) architecture, it uses simplified instruction sets and few addressing modes. As a result, the hardware becomes less complex, faster, easier to build and test. However, it executes the instruction in one cycle at the cost of increasing the number of instructions used per program. To improve the throughput and reduce the average execution time per instruction, it overlaps multiple instructions in a pipeline fashion as shown in Figure 1.1.1.1. On the other hand, MIPS architecture, after years of development, now supports 64-bit addressing and operation as well as high performance floating point which made it popular in the embedded systems implementation such as routers, residential gateways and video game consoles.

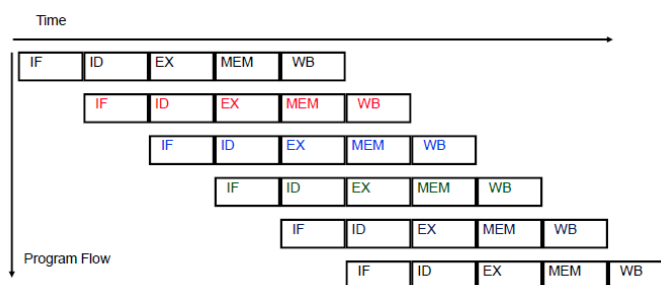


Figure 1.1.1.1: Conventional pipeline execution representation

1.1.2 Bus

In a computer system, a bus is a transmission path that interconnects various components such as Central Processing Unit (CPU), Direct Memory Access (DMA)

controller, memory, I/O devices and so on. Typically, there are three types of buses that carry information from place to place in the computer system. These buses include:

- address bus that carries a unique address information to a given device in order to be recognized by the CPU.
- data bus which gets data from or sends data to the device
- control bus which provides read or write signal to the device to indicate whether the CPU is asking for information or sending it information.

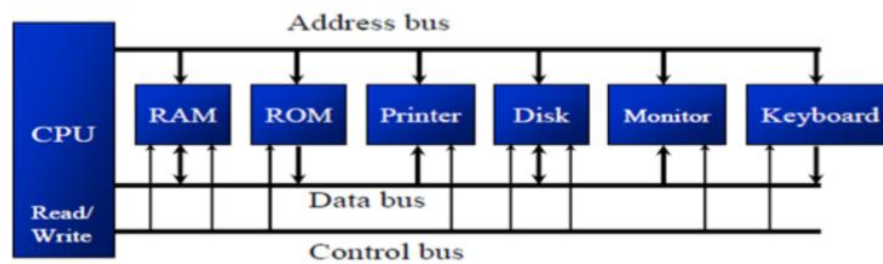


Figure 1.1.2.1: An overview of various type of buses in the computer system

1.1.3 SPI

Serial Peripheral Interface (SPI) is one of the communication protocols that provides a fast-synchronous serial communication between microcontroller and peripherals or between multiple microcontrollers with on-board peripherals (Anusha, 2017). It was developed with the intention to replace parallel interface so that the routing of the parallel bus around PCB can be avoided as well as to provide high speed data transfer between devices (Choudhury et al., 2014). Motorola Inc. was the first company that developed SPI to connect its first 68000 based microcontroller unit to other peripherals in the late 1970 (Choudhury et al., 2014). It has been later adopted by others in the industry and become a popular communication protocol due to its simplicity of interfacing and its full duplex characteristics for data communication (Choudhury et al., 2014). Over the years, SPI has been used in many kinds of applications and it is suitable for those applications that involve the transfer of data streams. For instance, it has been used to communicate with a variety of peripherals such as sensors, analog-to-digital converter (ADC), digital-to-analog converter (DAC), UART, USB, EEPROM and so on (Polytechnic Hub, 2017).

1.1.4 Zigbee

Zigbee is a standard wireless technology that has been developed for low-cost, low-power consumption wireless machine-to-machine (M2M) and Internet of Things (IoT) network (Linda, 2017). Since the Zigbee do not have any built-in microcontroller or processor, so they cannot manage the received or sent data (Priya, n.d.). In other words, they can simply transfer the information that they receive only (Priya, n.d.). However, they can be interfaced with other microcontrollers or processors such as Arduino, Raspberry Pi or PC via serial interface in order to manage the received or sent data (Priya, n.d.).

Furthermore, it works on the IEEE 802.15.4 specification and its WPANS can operate on 868 MHz, 900 MHz and 2.4 GHz frequencies (Linda, 2017). The IEEE 802.15.4 defines the physical and Media Access Control (MAC) layers for handling the devices at low rate (Elprocus, n.d.). The major applications of the Zigbee technology focus on sensor and automatic control area such as industrial automation, home automation, remote control and monitoring systems (Elprocus, n.d.). Because of the advantages of the Zigbee technology like low cost, low power consumption, and its topologies, it is therefore more suitable to be used for those applications mentioned above when compared to Bluetooth, Wi-Fi, and other short-range communication technologies (Elprocus, n.d.).

1.2 Motivation

A 32-bit RISC pipeline microprocessor has been developed in the Faculty of Information and Communication Technology (FICT) of Universiti Tunku Abdul Rahman (UTAR) by using Verilog Hardware Description Language (HDL). The project is based on the Reduced Instruction Set Computing (RISC) architecture. The motivations to initiate the project are due to the following reasons:

- Microprocessors have been designed by Microchip design companies as their Intellectual Property (IP) for commercial purposes. Generally, these microprocessor IP encompasses information about the complete design process, which includes the modeling, the verification and also the physical design of an integrated circuit. These IPs are qualified as the trade secrets of a company in

which they are protected by its holder. So, they are definitely not available in the market at a user-friendly price or without cost for research purposes.

- Several freely available microprocessor cores could be found over the Internet and majority of them are available at OpenCores. However, the MIPS Instruction Set Architecture (ISA) is not implemented entirely on those processors and they often lack of well-developed documentation. Because of these issues, it makes them not suitable for reuse and customization.
- The verification specifications for a freely available RISC microprocessor core are usually incomplete and not well constructed. The lack of well-developed verification specification can cause the subsequent verification process of a RISC microprocessor core to be slow-going. Eventually, it might slow down the overall design process.
- The physical design phase of these microprocessor cores will also be inevitably affected due to the lack of good verification specifications. In order for the physical design phase to be carried out smoothly, a design needs to be first functionally verified. This is in light of the fact that the physical design process will have to be repeated whenever the front-end design needs to change.

The RISC32 project that has been initiated in UTAR aims to deliver solutions to all of the issues mentioned above by creating a 32-bit RISC core-based development environment for assisting the research works in the area of soft-core as well as the application specific hardware modeling. Up to date, the RISC32 project that was initiated in UTAR has completed the CPU designs that supports basic instructions similar to MIPS instructions. The system control coprocessor that is the Coprocessor 0 (CP0) is also available to interface with I/O devices and handle interrupts.

With the completion of the RISC32 project, a RF transceiver module which is the Zigbee module will then be added to extend the research into cognitive radio area which requires modification to the I/O controllers and firmware of the RISC32 microprocessor. In this RISC32 project, several units based on the MIPS architecture have been divided. This project is one of those units for making wireless communication across the network possible in the RISC32 processor. With the available microarchitecture design

developed in the UTAR FICT, we can easily gain the software or firmware flexibility advantage without having to rely and wait for third party community to develop for us.

1.3 Problem Statement

As mentioned earlier, the MIPS ISA compatible pipeline processor which includes the Central Processing Unit (CPU), Coprocessor 0 (CP0), basic memory, flash controller, UART controller, SPI controller, GPIO controller and so on has been developed and functionally verified. However, the existing SPI controller architecture and its Interrupt Service Routine (ISR) are not fully workable after integrating it into the RISC32 pipeline processor. So, the previously developed SPI controller architecture and its ISR need to be revised. Further design work on the SPI controller unit needs to be continued in order for it to function normally with the processor. On top of that, there is also a lack of comprehensive documentation of the SPI controller unit such as verification specification, verification methodology, testbench and so on. The lack of well-developed verification specifications of the SPI controller unit can have the direct effect on the physical design phase because a design needs to be verified for its complete functionalities so that the subsequent physical design process can be carried out smoothly and easily. Otherwise, the physical design process would have to be carried out repeatedly whenever the front-end design is required to change because of the serious functionality failure.

1.4 Project Scope

The project scope will mainly focus on designing and integrating the SPI controller unit into the existing RISC32 pipeline processor. The specifications of the SPI controller unit and its internal block will be functionally verified by developing testbenches.

In addition, an Interrupt Service Routine (ISR) for handling all the interrupt requests generated by the SPI controller will be developed and then integrated into the existing exception handler of the RISC32 pipeline processor. Some MIPS test programs will also be written to test the SPI controller's functions after integration as well as to verify the correctness of the ISR execution.

Moving on, the developed SPI controller unit will be synthesized on the Field Programmable Gate Array (FPGA) technology and further interfaced with the Zigbee module which is the CC2420 RF transceiver in this project for wireless data communication.

Lastly, a comprehensive documentation on this project will be developed and maintained. In short, a piece of software, stimulation result and hardware are expected to be delivered at the end of the project.

1.5 Project Objectives

The objectives of this project are:

- To develop a SPI controller. This involves the micro-architecture modelling and verification of the SPI controller using Verilog language.
- To integrate the SPI controller into the RISC32. This involves the development of the interface between the SPI controller and the RISC32 based on I/O memory mapping technique. An Interrupt Service Routine (ISR) specifically for the SPI controller unit will also be developed in MIPS assembly language and integrated into the exception handler.
- On-board testing with Zigbee module. This involves the synthesis of the RISC32 onto an FPGA board. A Zigbee module will be connected to the SPI controller ports for final testing to demonstrate the transfer of data between two FPGA boards via the Zigbee modules.

1.6 Impact, Significance, and Contribution

After this project is done, it can provide a complete RISC microprocessor core-based development environment and proper interfacing system for connecting the SPI controller unit to the microprocessor as well as the Zigbee module. The development environment attributes to the availability of the following:

- A well-developed design documentation of the chip specification, the architecture specification as well as the micro-architecture specification.
- A fully functional interfacing system between the CPU and the SPI controller unit in the form of synthesis-ready RTL that is written in Verilog.

- A proper verification specification of the SPI controller unit. The verification specification contains the suitable verification methodology, verification techniques, test plan, testbench architecture and so on.
- A complete physical design in FPGA technology with documented timing and resources usage information.

This project can contribute to develop an environment that mentioned above by providing support to the hardware modeling research work. By having well-developed basic RISC RTL model, the verification environment, as well as the design documents, researchers will be able to develop their own research specific RTL models as part of the MIPS environment and can quickly verify the models to obtain results. As a result, the research works could be done more easily and rapidly.

1.7 Report Organization

This report consists of 9 chapters and the details of the project are shown in the following chapters.

In Chapter 1, some background information that matters is given, followed by the motivation of this project, the problem statement, project scope and objective in order to help readers to understand some facts or knowledge related to this project.

In Chapter 2, a literature review on two types of SPI protocols, the design of various type of SPI controller unit, and memory-mapped I/O technique has been highlighted and compared.

In Chapter 3, the methodologies and general work procedure for modeling, verifying, and synthesizing the SPI controller unit has been discussed. Moreover, it also discusses about the appropriate design tools that can help automate the design work, the technologies involved, the implementation issues and challenges, as well as the timeline of this project.

In Chapter 4, it discusses about the system overview of the RISC32 pipeline processor that will be used. The architecture, memory map, chip interface, and pin description of the processor used are stated in detail in Chapter 4.

In Chapter 5, it shows the full information about the micro-architecture specification of the designed SPI controller unit. It also gives an overview about each of the internal block in the SPI controller unit in terms of their functionality, block interface, pin description, and so on.

In Chapter 6, the exception handler of the RISC32 pipeline processor is briefly discussed, followed by the explanation of the Interrupt Service Routine (ISR) developed for the SPI controller unit.

In Chapter 7, it discusses about how the designed SPI controller unit is functionally verified, both as a single unit and in a whole system. All of the related verification specifications, test plans, testbenches, test programs, and stimulation results can be found in Chapter 7.

In Chapter 8, it analyses the synthesized SPI controller unit in terms of FPGA resource utilization and timing requirement. In addition, the solution for hardware and software implementation to achieve data transfer between two FPGA boards via the Zigbee modules is also proposed here.

In chapter 9, it concludes the overall project development, highlighting what have been achieved in the project. Furthermore, the future work that can be made to this project is also discussed here.

Chapter 2: Literature Review

2.1 Overview of 4-wire SPI Protocol

2.1.1 Detailed Pin Description in 4-wire SPI Protocol

The standard 4-wire SPI consists of 4 external pins, typically called Master Out Serial In (MOSI), Master In Serial Out (MISO), Serial Clock (SCLK), and Slave Select (SS). A detailed functional description of each pin is provided in Table 2.1.1.1.

Pin Name (Typical)	Pin Type	Functional Description
SCLK	Input Output	<ul style="list-style-type: none"> • It is used to output a clock signal generated by master to all the slave(s). • It is used for synchronizing the data transfer taking place across different devices. • It is only active during a data transfer and is tri-stated at any other time.
SS	Input Output	<ul style="list-style-type: none"> • It is an active low pin used by a master to select which slave. • Each slave has its own unique SS pin. • It must go low before a data transfer begins and must stay low during the process. Otherwise, the data transfer will be aborted.
MOSI	Input Output	<ul style="list-style-type: none"> • It is a unidirectional pin used to transfer serial data from the master to the slave. • When a device is configured as a master, serial data is sent through this pin. • When a device is configured as a slave, serial data is received through this pin. • It is only active during a data transfer and is tri-stated at any other time.

MISO	Input Output	<ul style="list-style-type: none"> • It is a unidirectional pin used to transfer serial data from the slave to the master. • When a device is configured as a slave and it is selected (the slave's SS pin goes low), serial data is sent through this pin. • When a device is configured as a slave and it is not selected, the slave will drive this pin to high impedance. • When a device is configured as a master, serial data is received through this pin. • It is only active during a data transfer and is tri-stated at any other time.
------	-----------------	---

Table 2.1.1.1: Functional descriptions of the standard 4-wire SPI's external pins

2.1.2 Transfer Modes in 4-wire SPI Protocol

Normally, a SPI peripheral can support up to 4 transfer modes (mode 0, 1, 2, and 3) which provide great flexibility in communication between master and slave(s). These 4 transfer modes have four different clocking configurations which is defined by a pair of parameters called Clock Polarity (CPOL) and Clock Phase (CPHA) (Anusha, 2017). The definition of the two parameters are given in Table 2.1.2.1.

Parameter	Value	Functional Description
Clock Polarity (CPOL)	CPOL = 0	Active-low clock is selected. SCLK is high in idle state.
	CPOL = 1	Active-high clock is selected. SCLK is low in idle state.
Clock Phase (CPHA)	CPHA = 0	Data sampling occurs at odd edges (1, 3, 5, ...,15) of the SCLK clock.
	CPHA = 1	Data sampling occurs at even edges (2, 4, 6, ..., 16) of the SCLK clock.

Table 2.1.2.1: Functional description of the Clock Polarity and Clock Phase parameters based on Motorola Inc.'s SPI Block Guide V03.06.

As shown in Table 2.1.2.2, each of the available modes has its own definition on the SCLK signal that determines what is the steady level (that is high or low) when the clock is not active as well as which SCLK edge is used for toggling the data and sampling the data (Leens, 2009, pp. 8-13). Therefore, in order for a communication to be possible, the master/slave pair must use the same set of parameters which include the SCLK frequency, CPOL and CPHA (Leens, 2009, pp. 8-13).

Mode	CPOL	CPHA	SCLK transmission edge	SCLK sample edge	SCLK idle stage
0	0	0	One half clock cycle before the rising edge	Rising edge	Low
1	0	1	Rising edge	Falling edge	Low
2	1	0	One half clock cycle before the falling edge	Falling edge	High
3	1	1	Falling edge	Rising edge	High

Table 2.1.2.2: SPI transfer mode information based on Motorola Inc.'s SPI Block Guide V03.06.

2.1.3 Timing Diagram in 4-wire SPI Protocol

In any of the transfer mode, the SS signal must go low before a data transfer begins and must stay low during the process. Otherwise, the data transfer will be aborted. If the 4-wire SPI device is set to operate in mode 0 (where CPOL = 0 and CPHA = 0), then it will transmit data (master → MOSI → slave, slave → MISO → master, simultaneously) one-half cycle before the rising edge and sample data on rising edge of the SCLK signal. Commonly, it is the mode 0 that is used for SPI bus communication (CORELIS, n.d.). Refer to Figure 2.1.3.1 to see the timing diagram for mode 0 serial data communication.

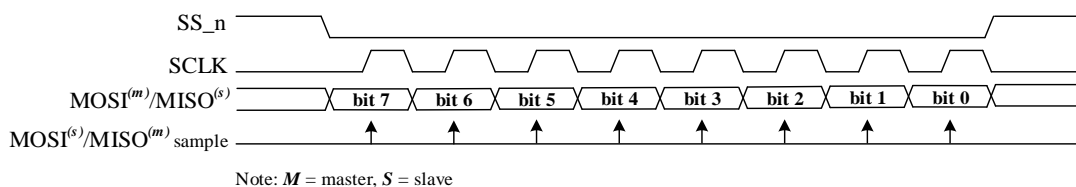


Figure 2.1.3.1: Timing diagram for mode 0 serial data communication in the 4-wire SPI protocol.

On the other hand, if the 4-wire SPI device is set to operate in mode 1 (where CPOL = 0 and CPHA = 1), then it will transmit data (master → MOSI → slave, slave → MISO → master, simultaneously) on rising edge and sample data on falling edge of the SCLK signal. Refer to Figure 2.1.3.2 to see the timing diagram for mode 1 serial data communication.

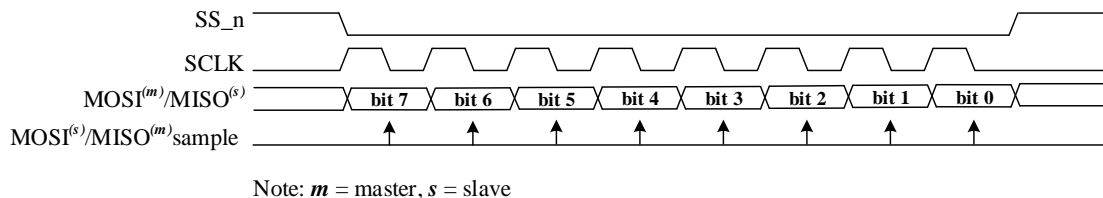


Figure 2.1.3.2: Timing diagram for mode 1 serial data communication in the 4-wire SPI protocol.

Apart from that, if the 4-wire SPI device is set to operate in mode 2 (where CPOL = 1 and CPHA = 0), then it will transmit data (master → MOSI → slave, slave → MISO → master, simultaneously) on one half clock cycle before the falling edge and sample data on falling edge of the SCLK signal. Refer to Figure 2.1.3.3 to see the timing diagram for mode 2 serial data communication.

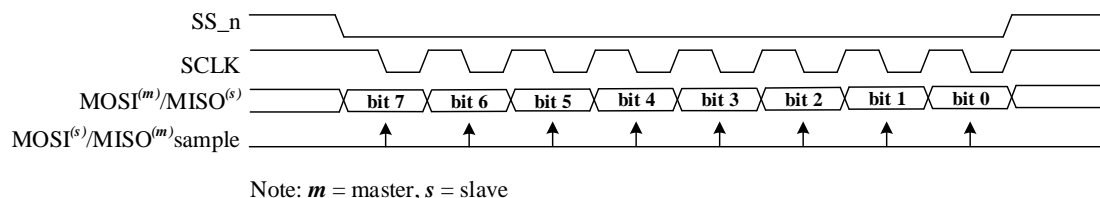


Figure 2.1.3.3: Timing diagram for mode 2 serial data communication in the 4-wire SPI protocol.

Lastly, if the 4-wire SPI device is set to operate in mode 3 (where CPOL = 1 and CPHA = 1), then it will transmit data (master → MOSI → slave, slave → MISO → master, simultaneously) on falling edge and sample data on rising edge of the SCLK signal. Refer to Figure 2.1.3.4 to see the timing diagram for mode 3 serial data communication.

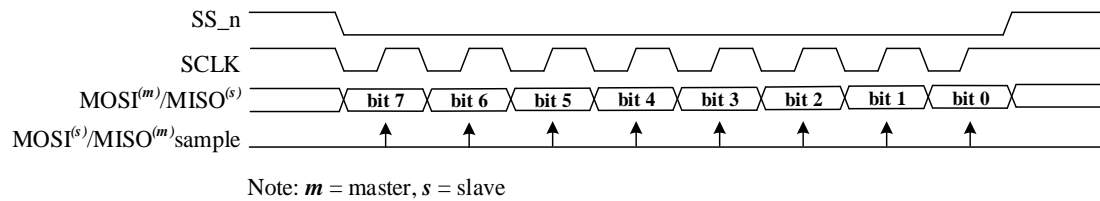


Figure 2.1.3.4: Timing diagram for mode 3 serial data communication in the 4-wire SPI protocol.

2.1.4 Working Principal of 4-wire SPI Protocol

The working of this 4-wire SPI is based on the contents of an eight-bit serial shift register in both of the master and the slave. In this SPI protocol, the master is always in control and initiate the communication. When the master wants to send data to a slave or request data from it, it will first select the particular slave by pulling the SS pin of the slave to low and it then activates the clock signal at a clock frequency that is usable by the master and the slave (Leens, 2009, pp. 8-13). In order for the communication to be possible, the master and slave must first agree on certain synchronization protocol. Meaning, they need to synchronize to the same clock and also operate in the same transfer mode (that is having the same set of CPOL and CPHA value) to ensure a valid data exchange. If multiple slaves are configured in different transfer modes, then the master will have to reconfigure itself whenever it wants to communicate with a different slave (Leens, 2009, pp. 8-13). Once they have set to follow the same synchronization protocol, the full-duplex data communication between the master and the slave can begin.

As the clock pulses are generated, the master transfers the data stored in its shift register serially to the slave via the MOSI pin. Similarly, the data contained in the slave's shift register is transferred back serially to the master's shift register via the MISO pin. For this case, the contents of the two shift registers get exchanged once a total of eight pulses of clock signals are generated. At the end, the master will pull the SS pin of the slave to high to complete the data transaction. An overview of the block diagram connection between a master device and a slave device in the 4-wire SPI protocol is provided in Figure 2.1.4.1 for better understanding.

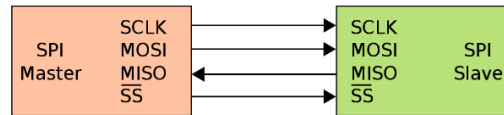


Figure 2.1.4.1: An overview of the block diagram connection between a master and a slave in the 4-wire SPI protocol.

On the other hand, it is also a single-master communication protocol in which only one master can exist in the connection at a time to initiate all the communications with slaves (Anusha, 2017). If more than one master is trying to drive the MOSI and SCLK simultaneously (with any attempt to pull low the SS pin), a mode fault error will occur (Motorola Inc, 2013).

Besides setting up to operate with a single master and a single slave (See Figure 2.1.4.1), SPI can also be set up with multiple slaves controlled by a single master. Commonly, there are two types of configuration used to connect multiple slaves to the master. They are independent slave configuration and Daisy-chain configuration.

In independent slave configuration, a master can have $(3+N)$ -wire serial interface where N is the total number of slaves connected to a single master on the bus. As indicated in Figure 2.1.4.2, a master needs to allocate an independent SS pin to each of its slave so that they can be addressed individually. In order to talk to a particular slave, the master needs to pull the desired slave's SS pin to low and keep the rest of them high. The advantage of this configuration is that it allows the connection of SPI devices operating in different transfer modes and/or baud rate as it controls each of the slaves separately. However, as the number of slaves increases in the system, the number of the independent SS pin needed by the master also increases and the board layout of the system become more complicated (Dhaker, 2018). Therefore, this method is simple to implement only when there are very few slaves connected to a single master.

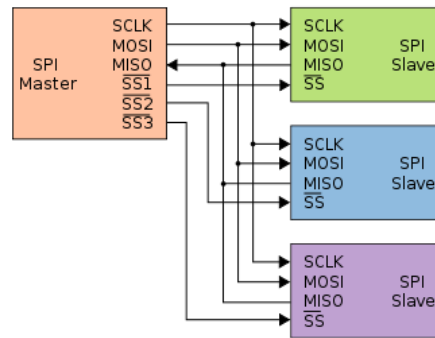


Figure 2.1.4.2: An overview of the block diagram connection between a single master device and multiple slave devices using independent slave configuration.

In Daisy-chain configuration, a common SS pin of the master is shared among all the slaves. Only the first slave in the chain receives the input data directly from the master while the rest of the slaves in the chain receive their input data from the output pin of the preceding slave. In Figure 2.1.4.3, the data shifted out of the master is connected directly to the first slave, and then out of the first slave into the second, and so on until the last slave in the series. In order for this scheme to work successfully, each of the slave needs to synchronize to the same clock as well as operate in the same transfer mode. The advantage of using Daisy-chain configuration is that it helps to save the number of SS pin needed on the master device. However, the speed of data transfer will be reduced significantly as the number of slave devices increases

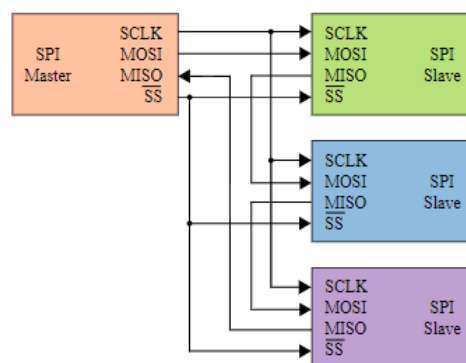


Figure 2.1.4.3: An overview of the block diagram connection between a single master device and multiple slave devices using Daisy-chain configuration.

Additionally, SPI does not use any slave acknowledge mechanism in its communication protocol to confirm receipt of data as well as offers no flow control (Leens, 2009, pp. 8-13). Furthermore, SPI neither specify any maximum data transfer rate (normally ranging up to several megabits per second) nor any addressing scheme in the protocol (Leens, 2009, pp. 8-13).

The 4-wire SPI protocol defined by Motorola Inc. has become a popular communication protocol and widely de facto in the industry due to its simplicity in interfacing with at least 4 wires only for data communication purpose between electronic devices. In addition, it becomes the current practice because it provides a good support for communication with low-speed devices by having full duplex capability. Meaning, it can transmit and receive data simultaneously, therefore resulting in a good data transfer performance and high throughput speed of over 10Mb/s (Leens, 2009, pp. 8-13). However, according to Tuan et al. (2017), the silicon cost and power consumption of the 4-wire SPI are the major issues in VLSI technology, such as the package size of IC and the quality of pad even though numerous transmission wires for data communication have been simplified.

2.2 Overview of 3-wire SPI Protocol

2.2.1 Detailed Pin Description of 3-wire SPI Protocol

Besides the standard 4-wire SPI implementation, SPI can also be designed to have 3 external pins only, namely Serial Data Input Output (SDIO), Serial Clock (SCLK), and Slave Select (SS). The bidirectional MOSI and MISO serial pin are now combined to a single bidirectional serial pin called SDI/SDO. A detailed functional description of each pin is provided in Table 2.2.1.1.

Pin Name (Typical)	Pin Type	Functional Description
SCLK	Input Output	<ul style="list-style-type: none"> It is used to output a clock signal generated by master to all the slave(s). It is used for synchronizing the data transfer taking place across different devices.

		<ul style="list-style-type: none"> • It is only active during a data transfer and is tri-stated at any other time.
SS	Input Output	<ul style="list-style-type: none"> • It is an active low pin used by a master to select which slave to initiate the communication with the master. • Each slave has its own unique SS pin. • When the SS pin of the slave goes low, the corresponding slave is selected. Otherwise, it is not selected. • It must go low before a data transfer begins and must stay low during the process. Otherwise, the data transfer will be aborted.
SDIO	Input Output	<ul style="list-style-type: none"> • It is a unidirectional pin used to transfer serial data from the master to the slave and vice versa. • When a device is configured as a master, serial data is sent and receive through this pin. • When a device is configured as a slave and it is selected, serial data is sent and receive through this pin. • When a device is configured as a slave and it is not selected, the slave will drive this pin to high impedance. • It is only active during a data transfer and is tri-stated at any other time.

Table 2.2.1.1: Functional descriptions of the 3-wire SPI's external pins

2.2.2 Transfer Modes of 3-wire SPI Protocol

The 3-wire SPI can also support up to 4 transfer modes (mode 0, 1, 2, and 3) in fulfilling different serial communication requirements of the connected peripherals. Similarly, the SPI transmission mode used can be defined by a pair of parameters called Clock Polarity (CPOL) and Clock Phase (CPHA) (Anusha, 2017). Refer to Table 2.1.2.1 to understand the definition of these two parameters. Apart from that, the 3-wire SPI also

applies the same SCLK definition on each of the available modes and the SPI transfer mode information can be found in Table 2.1.2.2.

2.2.3 Timing Diagram of 3-wire SPI Protocol

As mentioned earlier, the 3-wire SPI applies the same SCLK definition as what the 4-wire SPI uses. Therefore, they are having the same transfer mode information. In any of the transfer mode, the SS signal must go low before a data transfer begins and must stay low during the process. Otherwise, the data transfer will be aborted. If the 3-wire SPI device is set to operate in mode 0 (where CPOL = 0 and CPHA = 0), then it will transmit data one-half cycle before the rising edge and sample data on rising edge of the SCLK signal. On the other hand, if mode 1 (where CPOL = 0 and CPHA = 1) is selected, then it will transmit data on rising edge and sample data on falling edge of the SCLK signal. Apart from that, if it is set to operate in mode 2 (where CPOL = 1 and CPHA = 0), then it will transmit data on one half clock cycle before the falling edge and sample data on falling edge of the SCLK signal. Lastly, if mode 3 (where CPOL = 1 and CPHA = 1) is set, then it will transmit data on falling edge and sample data on rising edge of the SCLK signal. Refer to Figure 2.2.3.1 below to see the full timing diagram for serial data communication in all the transfer modes.

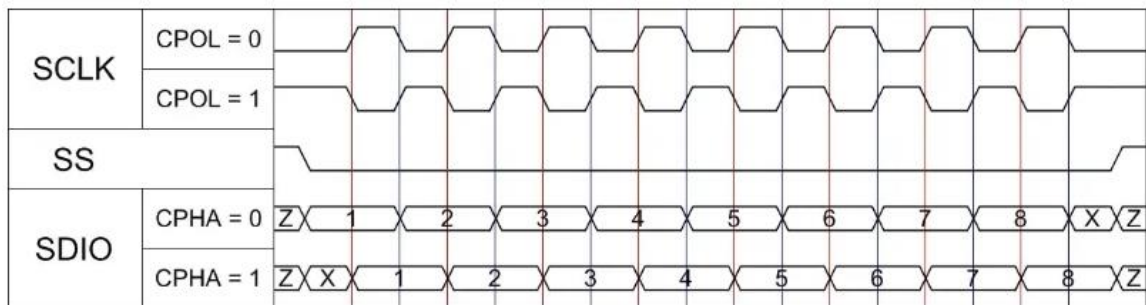


Figure 2.2.3.1: Timing diagram for mode 0, 1, 2 and 3 serial data communication in the 3-wire SPI protocol.

2.2.4 Working Principal of 3-wire SPI Protocol

The principle of the 3-wire SPI protocol is very similar to the 4-wire SPI protocol (Tuan et al, 2017). However, in 3-wire SPI protocol, there is only one serial bidirectional data line used for both input and output instead of having separate data input and data output

pin as shown in the 4-wire SPI type. And this has greatly affected the way SPIs communicate with each other.

The working of this 3-wire SPI is based on the contents of an eight-bit serial shift register in both of the master and the slave. In this SPI protocol, the master initiates the communication by first pulling the SS pin of the particular slave to low and then driving the clock signal at a clock frequency that is usable by the master and the slave. Once they have set to follow the same synchronization protocol, a valid data communication between the master and the slave can then begin.

As the clock pulses are generated, the master will first send a fixed-length command over the SDIO line. If it is a write command, then the master will continue to transmit the data stored in its shift register serially to the slave via the SDIO pin. If it is a read command, then the selected slave will transmit back the data contained in its shift register serially to the master's shift register via the SDIO pin as well. At the end, the SS pin of the slave will be de-asserted by the master in order to complete the data transaction. An overview of the block diagram connection between a master device and a slave device in 3-wire SPI protocol is provided in Figure 2.2.4.1 for better understanding.

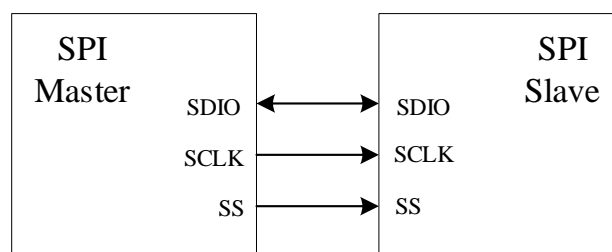


Figure 2.2.4.1: An overview of the block diagram connection between a master device and a slave device in the 3-wire SPI protocol.

Since it uses only one bidirectional pin for I/O, so it can minimize the silicon area and achieve cost-efficient (Tuan et al, 2017). However, it can only achieve half-duplex transmission in which either data transmission or receiving can occur at one time. Consequently, it will result in a slower throughput speed and lower data transfer performance.

2.3 SPI Controller

2.3.1 SPI Controller from Motorola Inc.

Based on the Motorola Inc's SPI specifications, version V03.06 that were revised on February 2003, the designs are the general-purpose solutions which offer viable ways to control SPI bus and highly flexible to suit any particular needs. The SPI controller designed by the Motorola Inc. has the following distinctive features:

- Have 4-wire SPI interfaces
- Have selectable serial clock frequency/baud rate
- Have 4 transfer modes with programmable clock phase and clock polarity
- Support master mode and slave mode
- Have bidirectional mode
- Have one double-buffered data register
- Have SPIF interrupt flag, SPI transmit empty interrupt flag, mode fault error interrupt capability
- Provide low power mode options

The SPI controller developed by Motorola Inc. consists of 4 pins, namely MOSI, MISO, SCLK and SS pin and it can support up to 4 transmission modes (mode 0, 1, 2, and 3). An overview on the Motorola's SPI controller architecture which contains the status register, control register, data register, shifter logic, baud rate generator, master/slave control logic and port control logic is given in Figure 2.3.1.1 below.

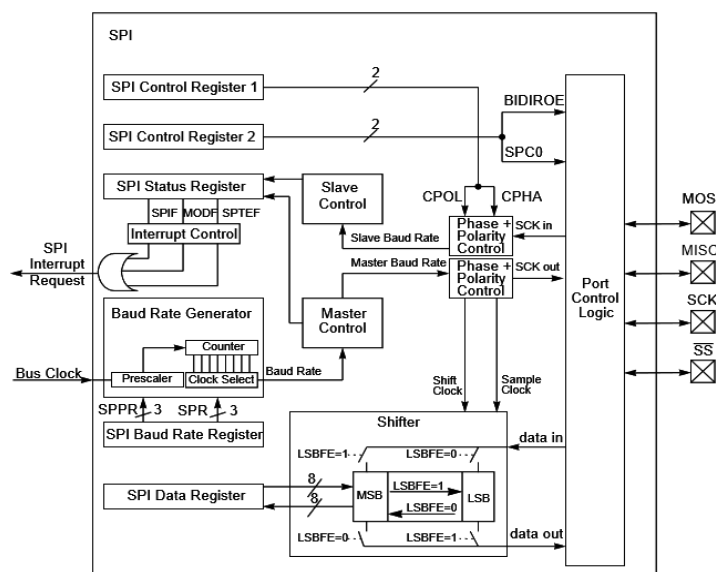


Figure 2.3.1.1: An overview on the Motorola Inc's SPI controller.

This SPI controller provides low power mode options that include run mode, wait mode, and stop mode. In run mode, the SPI system is in an active state and it operates normally. In wait mode, the SPI operation is in a configurable low power mode, which can be controlled by the setting of the SPICR2 register. In stop mode, it is inactive in order to save the power consumption.

Apart from that, the SPI controller designed by Motorola Inc. does not only have full-duplex capability, but also support half-duplex serial data communication. As shown in Table 2.3.1.1, it can have two modes, namely normal mode and bidirectional mode for interfacing with external devices. In normal mode, the SPI controller utilizes all of its 4 external pins to perform the full-duplex data communication that has been illustrated earlier. However, in bidirectional mode, only one serial data pin is used to interface with external devices. So, only half-duplex data communication is performed in this mode. When configured in bidirectional mode, the MOSI pin acts as the serial data I/O (MOMI) pin in master device whereas the MISO pin of the slave device becomes the serial data I/O pin (SISO) pin. The MISO pin in master mode and MOSI pin in slave mode are not used by the SPI controller in this bidirectional mode.

When SPE = 1	Master Mode, MSTR = 1	Slave Mode, MSTR = 0
Normal Mode SPC0 = 0		
Bidirectional Mode SPC0 = 1		

Table 2.3.1.1: Normal mode and bidirectional mode in Motorola Inc's SPI controller.

Typically, full-duplex data communication is favorable over half-duplex one because half-duplex data communication often results in a slower throughput speed and lower data transfer performance. Although the SPI controller designed by the Motorola Inc. provides a great flexibility in interfacing with external devices by having two modes, it actually makes the design more complicated to implement if compared to the one with a single mode only.

From Figure 2.3.1.1, it is noticeable that only one 8-bit SPI data register is being shared in the design. Meaning, it can either be used as a SPI receiver data register for read or a SPI transmitter data register for write at a given time only. Whenever the data transfer has completed, a read operation on this SPI data register must be first performed to release the register before any write operation on it, thus making simultaneous reading and writing become impossible. Moreover, only one 8-bit data can be buffered temporarily in it after each data transfer. So, more CPU's immediate attention may be required on it in order to release the register from time to time as it cannot hold large amount of data temporarily with the limited storage capability.

2.3.2 SPI Controller Designed by Kiat Wei Pau

The SPI controller developed by Kiat Wei Pau is a 4-wire SPI controller that contains all the MISO, MOSI, SCLK and SS pins in order to interface with external devices (Kiat, 2018). In short, it has the following features.

- Have 4-wire SPI interfaces
- Apply Wishbone interface connection
- Have 16 selectable serial clock frequency/ baud rate
- Have 4 transfer mode with programmable clock phase and clock polarity
- Support master mode only
- Have separate transmitter and receiver data register
- Have receiver buffer full, transmitter buffer empty, and mode fault error interrupt capability

The discussed SPI controller consists of a clock generator block, input output control block, receiver block, and transmitter block. On top of that, 4 registers which include SPI Configuration Register (SPICR), SPI Status Register (SPISR), SPI Receiver Data Register (SPIRDR), and SPI Transmitter Data Register (SPITDR) are available for user to access whereas the 2 shift registers, namely Transmitter Shift Register (TSR) and Receiver Shift Register (RSR) are used for parallel-to-serial and serial-to-parallel data conversion respectively (Kiat, 2018). The internal connection of the SPI controller that has been developed is given in Figure 2.3.2.1.

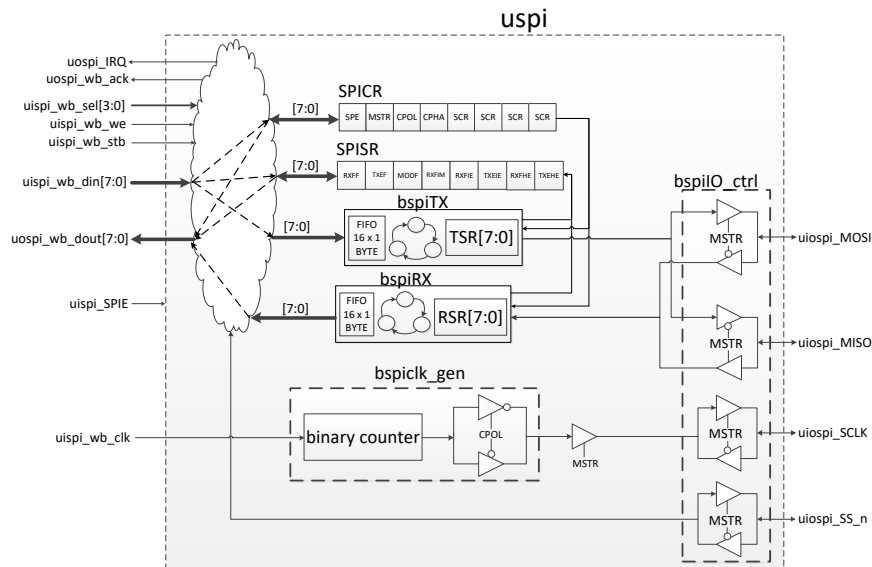


Figure 2.3.2.1: An overview of the SPI controller designed by Kiat Wei Pau.

When compared to the SPI controller from Motorola Inc., this SPI controller supports only full-duplex data communication between devices. The design is less complex and it uses lesser special-purpose registers for configuration and status monitoring. Moreover, it uses separate data registers to hold the data bytes, thus making simultaneous reading and writing become possible. To further illustrate on this point, the transmitter data register (SPITDR) is used to hold the data byte to be transmitted whereas the receiver data register (SPIRDRD) is used to store the received data byte from the other side. On top of that, it uses a 16 entries deep FIFO memory as the read/write buffer. So, by having larger data buffer capacity, less CPU's immediate response will be needed and the CPU can focus on executing its core task. However, it can only operate correctly in master mode and it also does not provide any low power mode options for power saving.

2.4 Memory-mapped I/O

Memory-map I/O (MMIO) is one of the general methods for assembly language program to address an I/O device. It is the I/O scheme where portions of address space are allocated to I/O devices, and reads and writes to those addresses are interpreted as commands to the I/O device (Patterson & Hennessy, 2005). With MMIO. CPU views

an I/O device as a set of special-purpose registers. Table 2.4.1 discusses the three general types of the special-purpose registers used in MMIO.

Register Type	Description
Status register	<ul style="list-style-type: none"> • Used to provide status information about the I/O device. • Often can be read only.
Configuration/Control register	<ul style="list-style-type: none"> • Used to configure and control the I/O device. • Both readable and writable.
Data register	<ul style="list-style-type: none"> • Used to read data from or send data to the I/O device. • Both readable and writable.

Table 2.4.1: Three general types of special-purpose registers used in MMIO.

By using MMIO method, the addresses of the registers in each of the I/O devices are assigned in a dedicated portion of the kernel's virtual address space. Each of the registers in the I/O controller must have a fixed and unique memory address within the mentioned address space in order for the CPU to access the specific register easily.

The benefits of using MMIO is that it keeps the instructions set small by adhering the design principles of MIPS, that is keeping the hardware simple via regularity (Langer, 2016). No new dedicated instructions are required in MMIO to simply read or write those special addresses because it allows the normal load and store instructions to be used for referencing, manipulating, and controlling both memory and I/O devices. The memory address that is being used will determine which type of device (memory or I/O device) to be accessed.

Chapter 3: Proposed Method/Approach

3.1 Methodologies and General Work Procedures

In the design process for digital system, there are 3 types of design methodologies available, namely top-down design methodology, bottom-up design methodology and mixed design methodology. In this project, the top-down design methodology will be used for designing and developing the SPI controller unit. In top-down design methodology, the top-level representation of a unit is first defined, followed by the lower-level representations based on several important criteria such as functionality, speed, silicon area and power consumption.

3.1.1 RTL Design Flow

The RTL design flow provided in Figure 3.1.1.1 below will be used throughout the project. In the RTL design flow, the micro-architectural level design will be focused more in this project because the SPI controller to be designed is in the unit level. A SPI controller that uses the 4-wire industry-standard SPI protocol will be designed for the Zigbee module in this project as it has better data transfer performance and higher throughput speed.

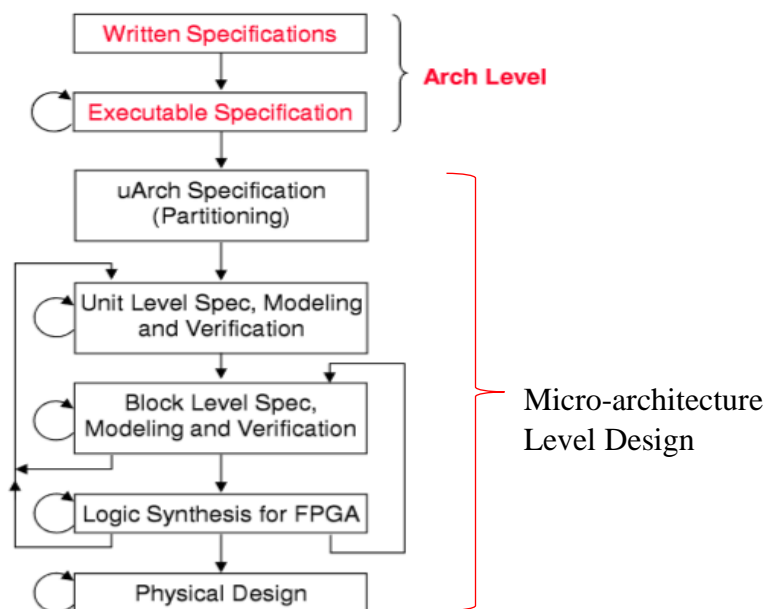


Figure 3.1.1.1: The RTL design flow used for developing the SPI controller unit is provided. The arrows indicate process or work flow (not data flow).

3.1.2 Micro-architecture Specification

Micro-architecture specification will describe the internal design of the SPI controller unit. The internal design of the SPI controller unit will be described with detailed and design-specific technical information in order for RTL coding to begin. In this project, the unit level of the SPI controller will include the following information:

- Functionality/feature description
- Interfaces and I/O pin description
- Functional partitioning into blocks and inter-blocks signaling
 - If the blocks are too complex to be coded, then further partition them into sub-blocks
- Test plan (focus on functional test)

Meanwhile, the block level of the SPI controller will have the following details:

- Functionality/feature description
- Interface and I/O pin description
- Internal operation: function table, FSM, and etc.
- Schematic and block diagram
- Test plan (focus on functional test)

3.1.3 RTL Modeling and Verification

With the development of the micro-architecture specification, the RTL coding on the SPI controller can begin. After coding, the RTL models are verified for functional correctness at each level. To further illustrate on this point, each block (RTL model) are verified before they are integrated into unit level. During the development of the project, if the design of the SPI controller unit does not meet all of the specified functional requirements, then the design flow would need to be repeated. After all the RTL models have successfully met the specified functional requirements, then logic synthesis will be carried out on the targeted technology which is the FPGA technology in this project.

3.1.4 Logic Synthesis for FPGA

After the SPI controller unit has been functionally verified, the model is said to be ready for logic synthesis. Logic synthesis is the process of converting RTL codes into an optimized gate level representation (a netlist). Based on the logic synthesis result, the gate level netlist is verified again for functional correctness. If it can successfully meet all the necessary specifications, the gate level netlist is now ready for physical design. However, if it cannot meet the required specifications, depending on the severity, corrections need to be made accordingly to the gate level netlist, the RTL models or the architecture.

3.2 Design Tools

Each stage of the design jobs requires the use of appropriate design tools to help automate the design work. Hence, there exist Electronic Design Automation (EDA) tools for design work at each particular level of abstraction. Since the RTL model of the SPI controller unit is designed by using Verilog hardware description language (HDL), thus a Verilog simulator is definitely needed to emulate the Verilog HDL. Some of the simulators are as shown in Table 3.2.1 below and several comparisons have been made among all of them.




Simulator	Incisive Enterprise Simulator	ModelSim	VCS
Company			
Language Supported	<ul style="list-style-type: none"> • VHDL-2002 • V2001 • SV2005 	<ul style="list-style-type: none"> • VHDL-2002 • V2001 • SV2005 	<ul style="list-style-type: none"> • VHDL-2002 • V2001 • SV2005
Platform supported	<ul style="list-style-type: none"> • Sun-solaris • Linux 	<ul style="list-style-type: none"> • Windows XP/Vista/7 • Linux 	Linux
Availability for free?	✗	✓ (SE edition only)	✗

Table 3.2.1: Comparison among 3 different Verilog simulators

Based on the comparison above, it is clear that the ModelSim from Mentor Graphic is the best choice among others to be used as the design tool for this project as they offer a free license for Student Edition version. Even though there is certain degree of limitations on the ModelSim Student Edition version, it is adequate to be used for this project. In addition, it supports Microsoft Windows platform as well. Although the other two simulators can also offer great features for Verilog stimulation, the price are too expensive (\$25,000 - \$100,000) and certainly not affordable to be used in this project.

As for the synthesis tools, there are a lot of logic synthesis tools targeting FPGA. Those logic synthesis tools include Quartus by Altera, Synplify by Synopsys, Vivado Design Suite by Xilinx, Encounter RTL Compiler by Cadence Design System, and so on. Among all the available logic synthesis tools, the Xilinx Vivado Design Suite is selected for this project as it is able to support the FPGA that we have in UTAR and it is already freely available in UTAR.

3.2.1 ModelSim PE Student Edition 10.4a

ModelSim from Mentor Graphic is the industry-leading simulation and debugging environment for HDL-based design in which its license can be obtained freely. The student version of the ModelSim is used for Verilog design stimulation instead of the full version because the features provided in the student edition are already adequate to be used for this project. Furthermore, both the Verilog and VHDL languages are supported by this ModelSim simulator. This simulator can also provide syntax error checking and waveform simulation which play an important part in developing the project. The timing diagrams and the waveforms are very useful in verifying the model functionalities after writing a program called testbench.

3.2.2 Xilinx Vivado Design Suite

Vivado Design Suite is a software suite designed by Xilinx. This software is designed for synthesis and analysis of HDL designs which enables developers to synthesize their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer easily. On top of that, it is a good design environment for FPGA products from Xilinx but it cannot be

used with those FPGA products from other vendors. The FPGA products that are supported by Xilinx Vivado Design Suite include Spartan FPGA, Virtex FPGA, Coolrunner, XC9500 Series CPLD and so on.

3.2.3 PCSpim

PCSpim is the Window version of spim. It is a software stimulator that loads and executes assembly language program for the MIPS RISC architecture. Besides, it also provides a simple assembler, debugger and a simple set of operating services. Hence, it is used in this project for developing the MIPS test program in order to verify the functional correctness of the ISR.

3.3 Technologies Involved

3.3.1 Field Programmable Gate Array (FPGA)

As mentioned earlier, the logic synthesis of the SPI controller unit will be eventually carried out on the FPGA technology. The FPGA technology is actually an integrated circuit (IC) that is programmable in the field after manufacture. FPGAs have been used widely by engineers in the design of specialized integrated circuits that can be later produced hard-wired in large quantities for distribution to computer manufacturers and end users. It is selected for prototype development in this project due to its benefits of cost efficiency, high flexibility and good scalability when compared to the other technologies. In this project, the FPGA development board used is the Xilinx Artix-7 XC7A100T FPGA chip on Digilent Nexys 4 DDR board and it is shown in Figure 3.3.1.1.

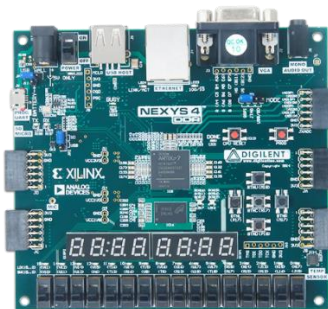


Figure 3.3.1.1: The top view of the Nexys 4 DDR (XC7A100T)

3.3.2 Zigbee RF Transceiver

Using the Zigbee communication system is less costly and simpler when compared to other short-range wireless sensor nodes like Bluetooth and Wi-Fi (Elprocus, n.d.). As presented in Figure 3.3.2.1, the Zigbee module used in this project would be the CC2420 transceiver product from the Texas Instruments company. The CC2420 transceiver is a true single-chip 2.4 GHz IEEE 802.15.4 compliant RF transceiver that is designed for low power and low voltage wireless application (Texas Instrument, 2013). It is a low-cost and highly integrated solution for robust wireless communication in the 2.4 GHz unlicensed ISM band. Besides, it has a digital direct sequence spread spectrum baseband modem that can provide a spreading gain of 9 dB and an effective data rate of 250 kbps (Texas Instrument, 2013). Most importantly, the configuration interface and transmit/receive FIFOs of the CC2420 can be accessed via a 4-wire SPI interface with serial clock. Typically, only reference crystal and a minimized number of passives are needed for the operation of the CC2420. So, it can be used together with a microcontroller and very few external passive components.



Figure 3.3.2.1: The top view of CC2420 from the Texas Instruments company.

3.4 Implementation Issues and Challenges

Multiple asynchronous clock domains have been employed in the RISC32 system. For instance, the designed SPI controller unit uses an I/O clock frequency of 10 MHz that is much slower than the 50 MHz CPU clock frequency for its internal operation. Since the clock signals of different clock domain are independent in general, passing signal and data safely from the fast clock domain into the slow clock domain can be a challenging task. This is on account of the fact that if the transition of the CDC signal happens too near to the active edge of the receiving clock, it may lead to setup or hold time violation of the flip flop, causing the output of this flip flop to be at an unknown

logic value for some duration of time. The undesired metastability problem is said to be occurred in the receiving system. Apart from that, a signal or data sending from the fast clock domain might change values twice or more times before it can be sampled by the receiving system that is running on a slower clock frequency. In short, serious design failures can happen due to the clock domain crossing error. Hence, proper synchronization is necessary when passing signals and data between clock domains and when receiving asynchronous inputs.

3.5 Timeline

3.5.1 Gantt Chart for Project 1

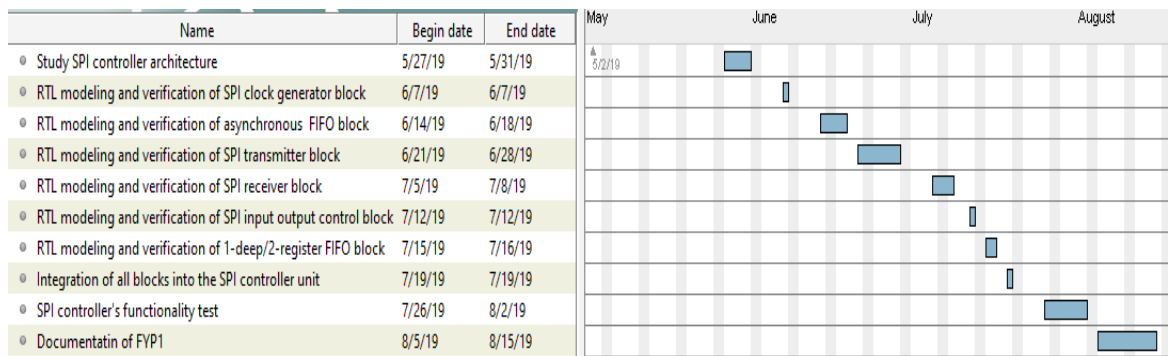


Figure 3.5.1.1: Gantt chart for Project 1

3.5.2 Gantt Chart for Project 2

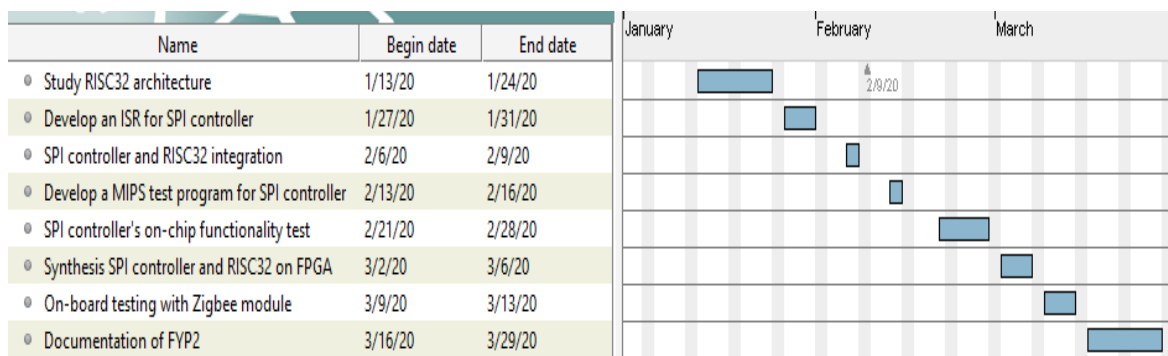


Figure 3.5.2.1: Gantt chart for Project 2

Chapter 4: System Specification

4.1 System Overview of the RISC32 Pipeline Processor

Since the selected CC2420 transceiver does not have any built-in microcontroller or processor with them, hence, it is proposed to interface the Zigbee module with the serial interface of the existing RISC32 pipeline processor which is the SPI in this project because of its convenience connection mechanism and full duplex capabilities. It is the RISC32 pipeline processor developed in the UTAR FICT that is being used because it can provide the software or firmware flexibility advantage for the SPI controller front-end design (modeling and verification).

4.1.1 RISC32 Pipeline Processor Architecture

The developed RISC32 pipeline processor is a 32-bit pipeline processor that consists of 3 major components that include Central Processing Unit (CPU), memory system and I/O system (Kiat, 2018). The developed CPU is said to be compatible to the 5-stage 32-bit MIPS Instruction Set Architecture (ISA) and it can support up to 49 instructions, covering arithmetic, logical, data transfer, program control, and system instruction classes. In addition, the memory system developed in this processor has a 2-level memory hierarchy with the first level consists of cache, Boot ROM as well as Data and Stack RAM whereas the second level contains a Flash memory. On the other hand, the I/O system of this processor contains GPIO controller, SPI controller, UART controller, Priority Interrupt controller and General-Purpose Register (GPR) unit. In addition, it also has a branch predictor that helps to improve the performance of the RISC32 processor in running program in terms of the number of clock cycle spent. An architectural overview on the RISC32 pipeline processor that has been developed is shown in Figure 4.1.1.1. On the other hand, the detailed specification of the RISC32 pipeline processor is also provided in Table 4.1.1.1.

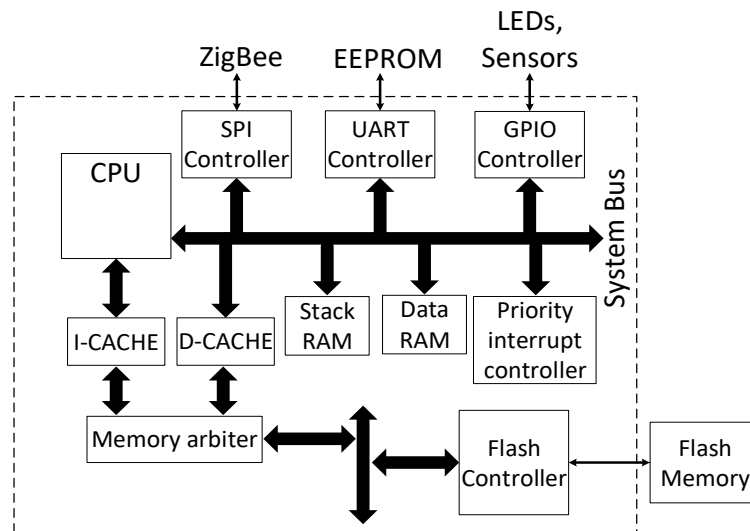


Figure 4.1.1.1: An overview on the architecture of the RISC32 pipeline processor.

		Pipeline
Frequency (MHz)		50
Instruction's cycle		5, overlapping
Branch predictor		64 entries 4 ways associative
Common features (Static Region)	Memory	4kBytes boot ROM, 128kBytes user access flash, 8kBytes RAM (Data & Stack), 1kBytes i-cache, 32Bytes d-cache, 512Bytes Memory Mapped I/O Register
	Communication interface	UART, SPI, 32 GPIO pins
Partial Bitstream start address		0x00A8 0000
Bitstream size		1,404,992 bits / 43906 words
FPGA board		Nexys 4 DDR (XC7A100T)
FPGA Resources (Overall)	LUT	8266
	LUTRAM	315
	FF	5643
	BRAM	3.50
	IO	46
	BUFG	1

Table 4.1.1.1: Specification of the RISC32 pipeline processor

4.1.2 Functional View of the RISC32 Pipeline Processor

The RISC32 pipeline processor that has been developed consists of 5 hardware stages which include Instruction Fetch (IF), Instruction Decode and Operand Fetch (ID), Execution (EX), Memory Access (MEM), and Write Back (WB) stages. Different hardware components are allocated in each of these pipeline stages. Therefore, every

instruction will need 5 clock cycles to run through all the 5 stages in order to complete its execution. Since the data hazard issue due to the Read-After-Write (RAW) data dependencies always exist in a pipeline processor, additional circuitries such as the forwarding and interlock block are built for solving the data hazards during the program execution (Kiat, 2018). The functional view of the 5-stage RISC32 pipeline processor is shown in Figure 4.1.2.1.

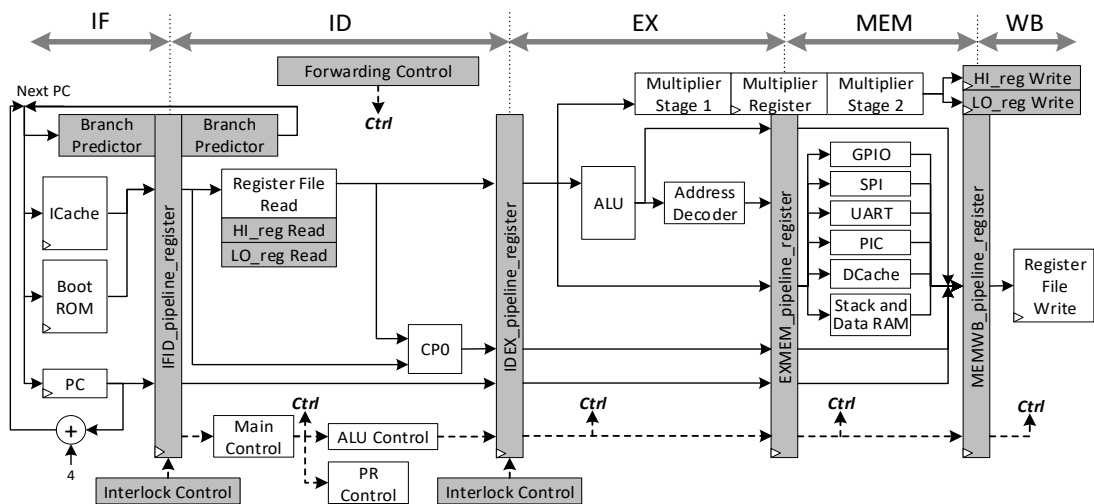


Figure 4.1.2.1: The functional view of the RISC32 pipeline processor.

4.1.3 Memory Map of the RISC32 Pipeline Processor

This RISC32 pipeline processor implements the MIPS memory space in two ways, that is by having virtual and physical addresses (Kiat, 2018). The virtual addresses are mainly used to access program instruction and data whereas the physical addresses are used to allocate physical memory such as Flash memory, Data and Stack RAM, boot ROM and I/O registers. The memory map used in the RISC32 pipeline processor is presented in Figure 4.1.3.1 and the purposes of various memory allocation are discussed in Table 4.1.3.1.

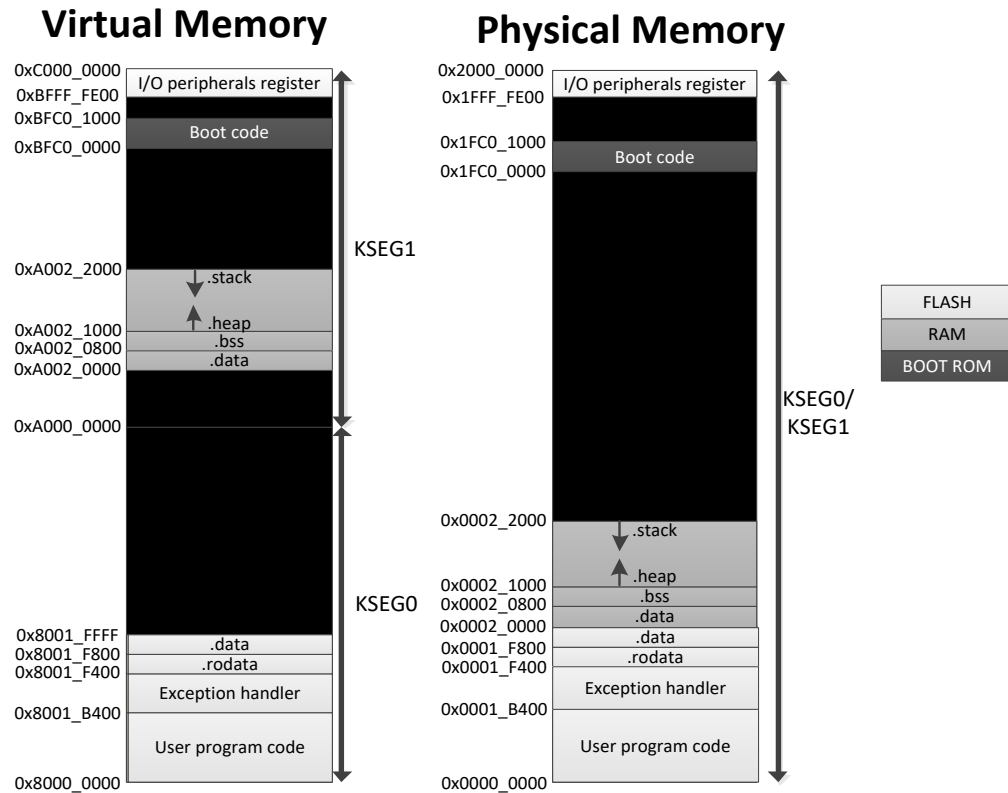


Figure 4.1.3.1: Memory map of the RISC32 pipeline processor.

Memory Usage	Description	Memory Size
I/O peripheral register	Used as the memory-mapped registers for I/O peripheral controllers.	512 bytes
Boot code	Used to store bootloader program code for initial system configuration when powered on.	4k bytes
Stack	Used by procedure during execution to store register values.	8k bytes
Heap	Used to hold variables declared dynamically.	
Exception handler	Used to store the exception handler codes.	16k bytes
User program code	Used to store user program codes	128k bytes

Table 4.1.3.1: Memory map description of the RISC32 pipeline processor.

4.2 Chip Interface of the RISC32 Pipeline Processor

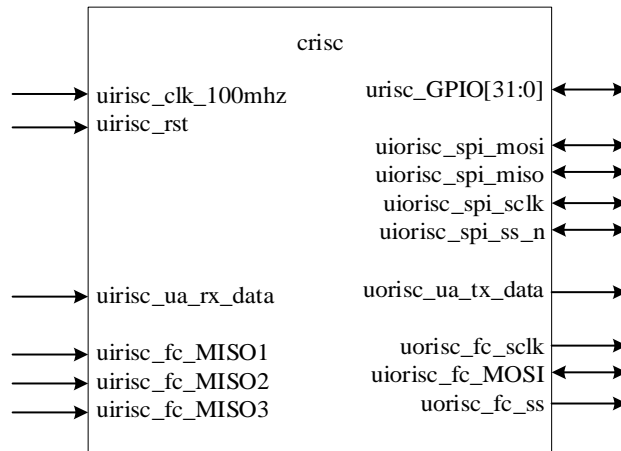


Figure 4.2.1: Chip interface of the RISC32 pipeline processor.

4.3 Input Pin Description of the RISC32 Pipeline Processor

Pin name: uirisc_clk_100mhz	Pin class: Global
Source → Destination: External → crisc	
Pin function: To provide a reference signal to synchronize all other signals in a system	
Pin name: uirisc_rst	Pin class: Global
Source → Destination: External → crisc	
Pin function: To reset the whole MIPS ISA compatible pipeline processor	
Pin name: uirisc_ua_rx_data	Pin class: Data
Source → Destination: External device’s UART unit → crisc	
Pin function: UART standard pin – Receive Serial Data	
Pin name: uirisc_fc_MISO1	Pin class: Data
Source → Destination: Flash memory → crisc	
Pin function: SPI protocol serial input pin	
Pin name: uirisc_fc_MISO2	Pin class: Data
Source → Destination: Flash memory → crisc	
Pin function: SPI protocol serial input pin	
Pin name: uirisc_fc_MISO3	Pin class: Data
Source → Destination: Flash memory → crisc	
Pin function: SPI protocol serial input pin	

Table 4.3.1: Input pin description of the RISC32 pipeline processor.

4.4 Output Pin Description of the RISC32 Pipeline Processor

Pin name: uorisc_ua_tx_data	Pin class: Data
Source → Destination: crisc → External device's UART unit	
Pin function: UART standard pin – Transmit Serial Data	
Pin name: uorisc_fc_sclk	Pin class: Data
Source → Destination: crisc → Flash memory	
Pin function: SPI protocol Serial Clock signal	
Pin name: uorisc_fc_ss	Pin class: Control
Source → Destination: crisc → Flash memory	
Pin function: SPI protocol Slave Select	

Table 4.4.1: Output pin description of the RISC32 pipeline processor.

4.5 Input Output Pin Description of the RISC32 Pipeline Processor

Pin name: urisc_GPIO[31:0]	Pin class: Data
Source → Destination: crisc ↔ External device (LEDs, switch, etc)	
Pin function: 32 GPIO pins	
Pin name: uiorisc_spi_mosi	Pin class: Data
Source → Destination: crisc ↔ External device's SPI unit	
Pin function: SPI standard pin – Master out Serial In (MOSI)	
If the crisc is configured as a master, then uiorisc_spi_mosi will become an output, else otherwise.	
Pin name: uiorisc_spi_miso	Pin class: Data
Source → Destination: crisc ↔ External device's SPI unit	
Pin function: SPI standard pin – Master In Serial Out (MISO)	
If the crisc is configured as a master, then uiorisc_spi_miso will become an input, else otherwise.	
Pin name: uiorisc_spi_sclk	Pin class: Control
Source → Destination: crisc ↔ External device's SPI unit	
Pin function: SPI standard pin – SPI Serial Clock signal for data synchronization across devices.	
If the crisc is configured as a master, then uiorisc_spi_clk will become an output, else otherwise.	
Pin name: uiorisc_spi_ss_n	Pin class: Control
Source → Destination: crisc ↔ External device's SPI unit	
Pin function: SPI standard pin – SPI Slave Select control signal.	
If the crisc is configured as a master, then uiorisc_spi_ss_n will become an output, else otherwise.	
Pin name: uiorisc_fc_MOSI	Pin class: Data
Source → Destination: crisc ↔ Flash memory	
Pin function: SPI protocol serial input output pin	

Table 4.5.1: Input output pin description of the RISC32 pipeline processor.

Chapter 5: Micro-Architecture Specification

5.1 SPI Controller Unit

5.1.1 Functionality/Feature of the SPI Controller Unit

The SPI controller is a controller unit that uses the 4-wire industry-standard SPI protocol to handle the data exchange between two SPI-interface devices. The details of the standard 4-wire SPI protocol have been fully discussed in Section 2.1. Currently, this SPI controller is mainly used to provide fast synchronous serial communication between a master device and a slave device. Additional address decoder will need to be added if it wants to communicate with multiple slaves in future. The designed SPI controller unit can transmit and receive 8-bit data simultaneously and correctly between the master device and the slave device in 4 different transfer modes as shown in Appendix A. This SPI controller can operate as a master or a slave device at a given time and Table 5.1.1.1 will describe the direction of the SPI standard pins (MOSI, MISO, SCLK, an SS pin) in each of the operation modes.

SPI standard pin	Master SPI	Slave SPI
MOSI	Output	Input
MISO	Input	Output
SCLK	Output	Input
SS	Output	Input

Table 5.1.1.1: Pin direction of the SPI standard pins when it is set as a master or slave device.

In short, a summary of the designed SPI controller unit's features is given in below.

- Have easy configuration interface
 - 4-wire SPI interfaces, which include MISO, MOSI, SCLK and SS pin.
- Have 16 selectable serial clock frequency/ baud rate
- Have 4 transfer modes with programmable clock phase and clock polarity
 - Mode 0 (CPOL = 0, CPHA = 0)
 - Mode 1 (CPOL = 0, CPHA = 1)
 - Mode 2 (CPOL = 1, CPHA = 0)
 - Mode 3 (CPOL = 1, CPHA = 1)
- Have 2 operation modes

- Master mode
- Slave mode
- Have separate transmitter and receiver data register (16x1-byte FIFO)
 - Each of the transmitter FIFO element is a SPI Transmitter Data Register.
 - Each of the receiver FIFO element is a SPI Receiver Data Register
- Support full-duplex synchronous serial data transfer
 - Serial data transmission and receiving can take place simultaneously.
 - 8 SCLK pulses are required when transmitting and receiving an 8-bit data.
- Provide 3 types of interrupts
 - Receiver buffer full interrupt
 - Transmitter buffer empty interrupt
 - Mode fault error interrupt

5.1.2 Operating Procedure (External Operation)

The details of the procedure for CPU to operate the designed SPI controller is provided in below.

1. CPU supplies a global clock signal to the SPI controller for clock reference.
2. CPU resets the SPI controller in order to initialize all of its registers from an unknown value to the initialized value, reset all of its FIFOs' pointers, as well as reset all of its FSMs to the idle state.
3. CPU stores one or more 8-bit data that is/are to be transmitted into the SPI Transmitter Data Register (SPITDR) by using the 0xbfff_fe26 address value. Writing to the SPITDR is actually writing to the 16-deep transmitter FIFO.
4. CPU configures the setting of the SPISR accordingly for status monitoring, interrupt enable controlling, and FSM stall controlling by using the 0xbfff_fe25 address value (Refer to Section 5.8 for SPISR's full information).
5. CPU configures the setting of the SPICR accordingly for activating the SPI controller, selecting the desired operation mode (master or slave), selecting the transfer mode (mode 0, 1, 2 or 3), and selecting the suitable baud rate by using the 0xbfff_fe24 address value (Refer to Section 5.8 for SPICR's full information).

- a. Once the settings on the SPISR and the SPICR have been configured correctly, the SPI controller is said to be ready to perform the full-duplex data communication with another SPI-compatible device.
 - b. The SPI controller starts to go through the respective FSMs to transmit and receive data simultaneously from the other side until all the pending transfers have completed.
6. Depending on the SPISR configuration for the application, CPU will serve the SPI controller in different ways.
- a. If interrupt is disabled, CPU will need to use the polling method to determine when it is ready to read the received data from the SPIRDR, load new data into the SPITDR or disable the SPI controller. The SPI controller simply puts the information in the SPISR, and the CPU must come and get the information. The status flag(s) such as RXDF, TXEF and MODF flag in the SPISR will have to be checked periodically by using instructions. When the status flag(s) is/are asserted, the CPU performs the service accordingly. The status flag(s) of the SPISR will be cleared automatically once the CPU finished the service. After that, the CPU can move on to perform other tasks.
 - b. If interrupt is enabled, the CPU can perform its normal tasks until it is being noticed. No extra instructions are needed to monitor the status flags in the SPISR. Whenever the SPI controller needs the CPU's immediate attention, it will notify the CPU by sending it an interrupt signal. Depending on the types of the interrupt requests generated by the SPI controller, the CPU will take the appropriate actions as defined in the SPI's Interrupt Service Routine (ISR), that is to read the received data from the SPIRDR or disable the SPI controller. Reading from the SPIRDR is actually reading from the 16-deep receiver FIFO. After finishing the ISR, the CPU returns to the place where it was interrupted and resumes the normal program execution. The status flag(s) of the SPISR will be cleared automatically once the CPU finished the service.

If the CPU wants to reconfigure the setting(s) after going through step 1 to 6, it is advisable to first de-activate the SPI controller to ensure a smooth configuration and operation. Do not perform the reconfiguration while the SPI controller is performing data transfer with other devices in both operation modes.

- a. Firstly, reset the SPE control bit in the SPICR to de-activate the SPI controller as well as to reset all of its FSMs to the initial state.
- b. Perform new setting on the SPI controller.
 - i. Repeat step 4 to 6 if only want to reconfigure the setting of the SPISR.
 - ii. Repeat step 5 to 6 if only want to reconfigure the setting of the SPICR (for new operation mode, new transfer mode and/or new SPI baud rate).
 - iii. Repeat step 4 to 6 if want to reconfigure the settings of both SPISR and SPICR.

5.1.3 Unit Interface of the SPI Controller Unit

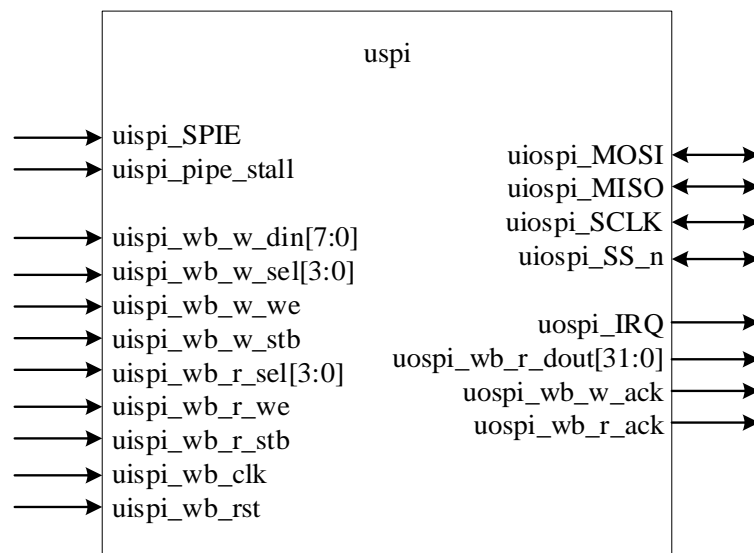


Figure 5.1.3.1: SPI controller unit interface.

5.1.4 Input Pin Description of the SPI Controller Unit

Pin name: uispi_SPIE	Pin class: Data
Source → Destination: Priority interrupt controller unit → SPI controller unit	
Pin function: To allow the SPI to interrupt	
1: enable SPI global interrupt	
0: disable SPI global interrupt	
Pin name: uispi_pipe_stall	Pin class: Control
Source → Destination: Priority interrupt controller unit → SPI controller unit	
Pin function: To stall the SPI controller unit	
1: stall the SPI controller unit	
0: do not stall the SPI controller unit	
Pin name: uispi_wb_w_din[7:0]	Pin class: Data
Source → Destination: Datapath unit → SPI controller unit	
Pin function: Wishbone standard data input bus (for write operation)	
Pin name: uispi_wb_w_sel[3:0]	Pin class: Control
Source → Destination: Address decoder block → SPI controller unit	
Pin function: Wishbone standard byte select signal (for write operation)	
Pin name: uispi_wb_w_we	Pin class: Control
Source → Destination: Address decoder block → SPI controller unit	
Pin function: Wishbone standard write enable signal – indicate current bus cycle for write	
1: write cycle – write to SPI controller	
Pin name: uispi_wb_w_stb	Pin class: Control
Source → Destination: Address decoder block → SPI controller unit	
Pin function: Wishbone standard strobe signal (for write operation) – indicate valid data transfer cycle	
1: activate SPI controller for write access	
0: de-activate SPI controller for write access	
Pin name: uispi_wb_r_sel[3:0]	Pin class: Control
Source → Destination: Address decoder block → SPI controller unit	
Pin function: Wishbone standard byte select signal – data granularity control	
1111: word select	
1100: upper half-word selected	
0011: lower half-word selected	
1000: 4th byte selected	
0100: 3rd byte selected	
0010: 2nd byte selected	
0001: 1st byte selected	
Pin name: uispi_wb_r_we	Pin class: Control
Source → Destination: Address decoder block → SPI controller unit	
Pin function: Wishbone standard read enable signal – indicate current bus cycle for read	
0: read cycle – read from SPI controller	
Pin name: uispi_wb_r_stb	Pin class: Control
Source → Destination: Address decoder block → SPI controller unit	
Pin function: Wishbone standard strobe signal (for read operation) – indicate valid data transfer cycle	
1: activate SPI controller for read access	
0: de-activate SPI controller for read access	

Pin name: uispi_wb_clk	Pin class: Global
Source → Destination: Global clock → SPI controller unit	
Pin function: Global clock	
Pin name: uispi_wb_rst	Pin class: Global
Source → Destination: Global reset → SPI controller unit	
Pin function: Global reset	
1: reset	
0: no reset is required	

Table 5.1.4.1: Input pin description of the SPI controller unit

5.1.5 Output Pin Description of the SPI Controller Unit

Pin name: uospi_IRQ	Pin class: Control
Source → Destination: SPI controller unit → CP0 block & Priority interrupt controller unit	
Pin function: To request an interrupt (The uispi_SPIE must be pulled high before an interrupt can be sent)	
1: request to interrupt	
0: no interrupt request	
Pin name: uospi_wb_r_dout[31:0]	Pin class: Data
Source → Destination: SPI controller unit → Datapath unit	
Pin function: Wishbone standard data output bus	
Pin name: uospi_wb_w_ack	Pin class: Status
Source → Destination: SPI controller unit → Datapath unit	
Pin function: Wishbone standard acknowledge signal (for write operation)	
1: normal bus cycle termination	
0: no bus cycle termination	
Pin name: uospi_wb_r_ack	Pin class: Status
Source → Destination: SPI controller unit → Datapath unit	
Pin function: Wishbone standard acknowledge signal (for read operation)	
1: normal bus cycle termination	
0: no bus cycle termination	

Table 5.1.5.1: Output pin description of the SPI controller unit.

5.1.6 Input Output Pin Description of the SPI Controller Unit

Pin name: uiospi_MOSI	Pin class: Data
Source → Destination: SPI controller unit ↔ External device's SPI unit	
Pin function: SPI standard pin – Master Out Serial In If the SPI controller unit is configured as a master, then uiospi_MOSI will become an output, else otherwise.	
Pin name: uiospi_MISO	Pin class: Data
Source → Destination: SPI controller unit ↔ External device's SPI unit	
Pin function: SPI standard pin – Master In Serial Out If the SPI controller unit is configured as a master, then uiospi_MISO will become an input, else otherwise.	

<p>Pin name: uiospi_SCLK Pin class: Control</p> <p>Source → Destination: SPI controller unit ↔ External device's SPI unit</p> <p>Pin function: SPI standard pin – Serial Clock</p> <p>It is the clock signal for data synchronization across devices. If the SPI controller unit is configured as a master, then uiospi_SCLK will become an output, else otherwise.</p>
<p>Pin name: uiospi_SS_n Pin class: Control</p> <p>Source → Destination: SPI controller unit ↔ External device's SPI unit</p> <p>Pin function: SPI standard pin – Slave Select</p> <p>If the SPI controller unit is configured as a master, then uiospi_SS_n will become an output, else otherwise.</p>

Table 5.1.6.1: Input output pin description of the SPI controller unit.

5.1.7 Internal Operation of the SPI Controller Unit

uspi_wb_w_stb	uispi_wb_w_we	uspi_stall_reg	uispi_wb_w_sel [3:0]	Function
1	1	0	0001	Enable write operation to SPICR
1	1	0	0010	Enable write operation to SPISR
1	1	0	0100	Enable write operation to SPITDR

Table 5.1.7.1: Functional description of the SPI controller's write operation.

uspi_wb_r_stb	uispi_wb_r_we	uispi_wb_r_sel [3:0]	Function
1	0	0001	Enable read operation to SPICR
1	0	0010	Enable read operation to SPISR
1	0	1000	Enable read operation to SPIRDR
1	0	0011	Enable read operation to SPISR and SPICR

1	0	1100	Enable read operation to SPIRDR
1	0	1111	Enable read operation on SPIRDR, SPISR, and SPICR.

Table 5.1.7.2: Functional description of the SPI controller's read operation.

5.1.8 Design Partitioning of the SPI Controller Unit

The SPI controller unit developed in this project consists of several internal blocks that work together so that the data communication between two SPI-interface devices can be performed correctly when using the SPI protocol. This SPI controller unit consists of one SPI transmitter block, one SPI receiver block, one SPICR FIFO block, one SPISR FIFO block, and one SPI clock generator block. In addition, it also consists of 4 special-purpose registers for users to access (read or write). These special registers are the SPI Configuration Register (SPICR) for configuration setting, the SPI Status Register (SPISR) for status monitoring purpose, the SPI Transmitter Data Register (SPITDR) for holding the data to be transmitted, and the SPI Receiver Data Register (SPIRDR) for holding the received data. Writing to the SPITDR is actually writing to a 16x1-byte transmitter FIFO block whereas reading from the SPIRDR is actually reading from the 16x1-byte receiver FIFO block. All of these I/O peripheral registers are memory-mapped and have their own addresses so that the CPU can read or write the specific registers easily. The details of these 4 memory-mapped I/O peripheral registers are discussed in Section 5.7 respectively. An overview of the block-level partitioning of the SPI controller unit is provided in Figure 5.1.8.1 whereas the details of its internal blocks are discussed in Table 5.1.8.1.

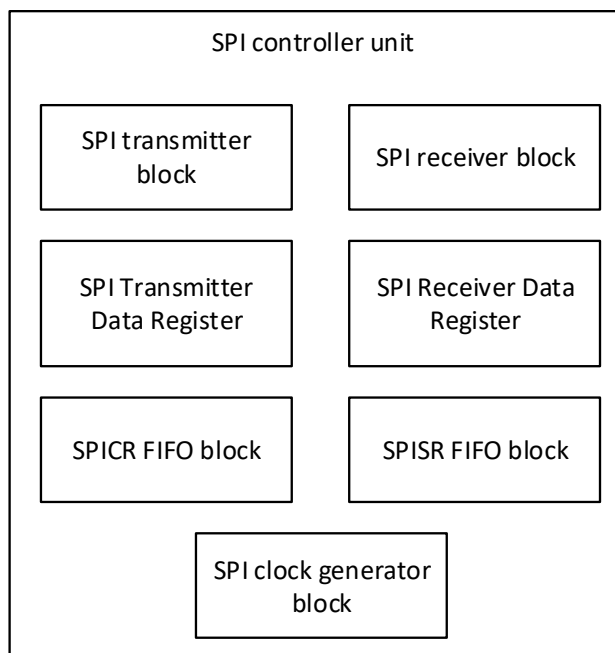


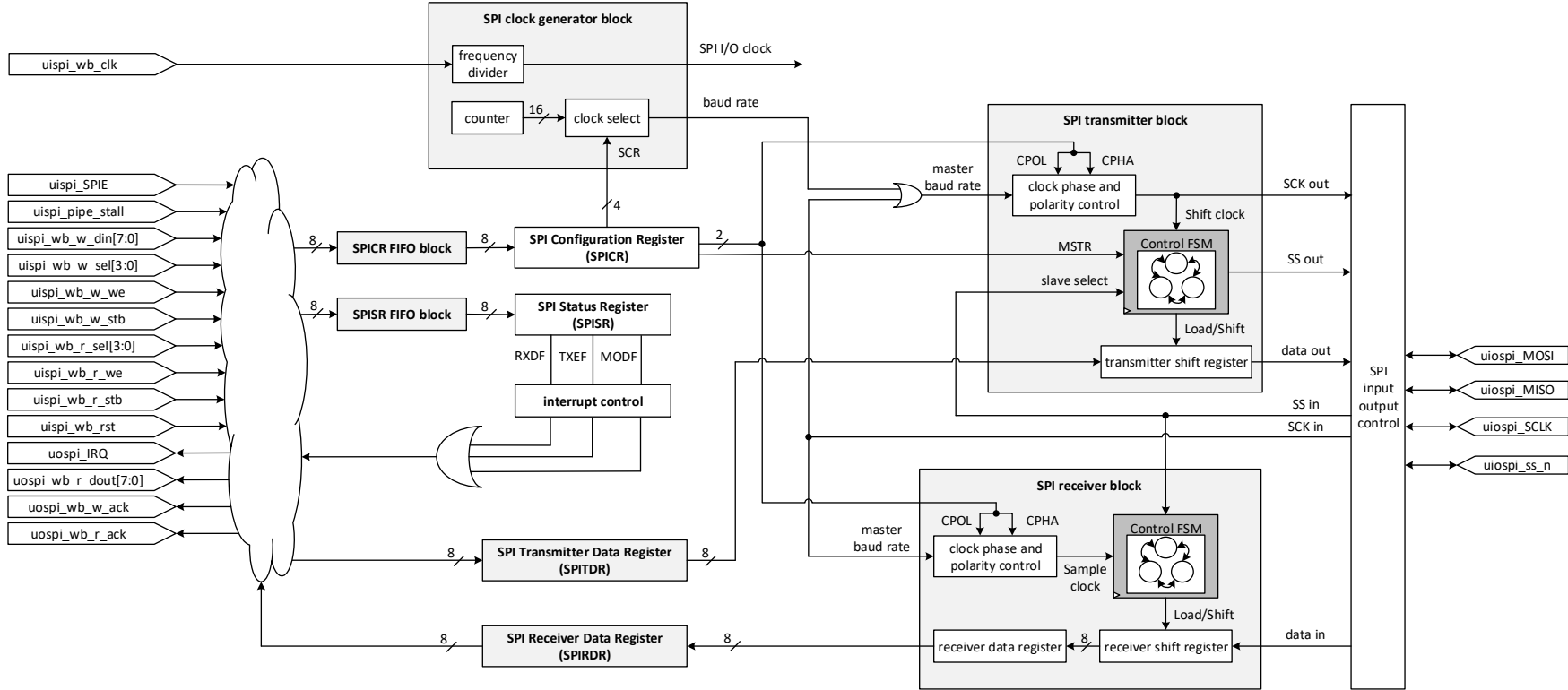
Figure 5.1.8.1: Block-level partitioning of the SPI controller unit.

Internal Block	Function
SPI transmitter block	To handle the serial data transmission of the SPI controller unit.
SPI receiver block	To handle the serial data receiving of the SPI controller.
SPI Transmitter Data Register (a 16-deep asynchronous FIFO block)	<ul style="list-style-type: none"> To hold the data that will be transmitted to another SPI-compatible device. To pass multiple data bits safely across CDC boundaries.
SPI Receiver Data Register (a 16-deep asynchronous FIFO block)	<ul style="list-style-type: none"> To hold the data received from another SPI-compatible device. To pass multiple data bits safely across CDC boundaries.
SPICR FIFO block (a 2-deep FIFO synchronizer block)	To safely handle the passing of multi-bit control signal from one clock domain to a new clock domain.

SPI SR FIFO block (a 2-deep FIFO synchronizer block)	To safely handle the passing of multi-bit control signal from one clock domain to a new clock domain.
SPI clock generator block	<ul style="list-style-type: none">• To generate the I/O clock frequency for SPI internal operation.• To generate 16 transmission speed / baud rates.

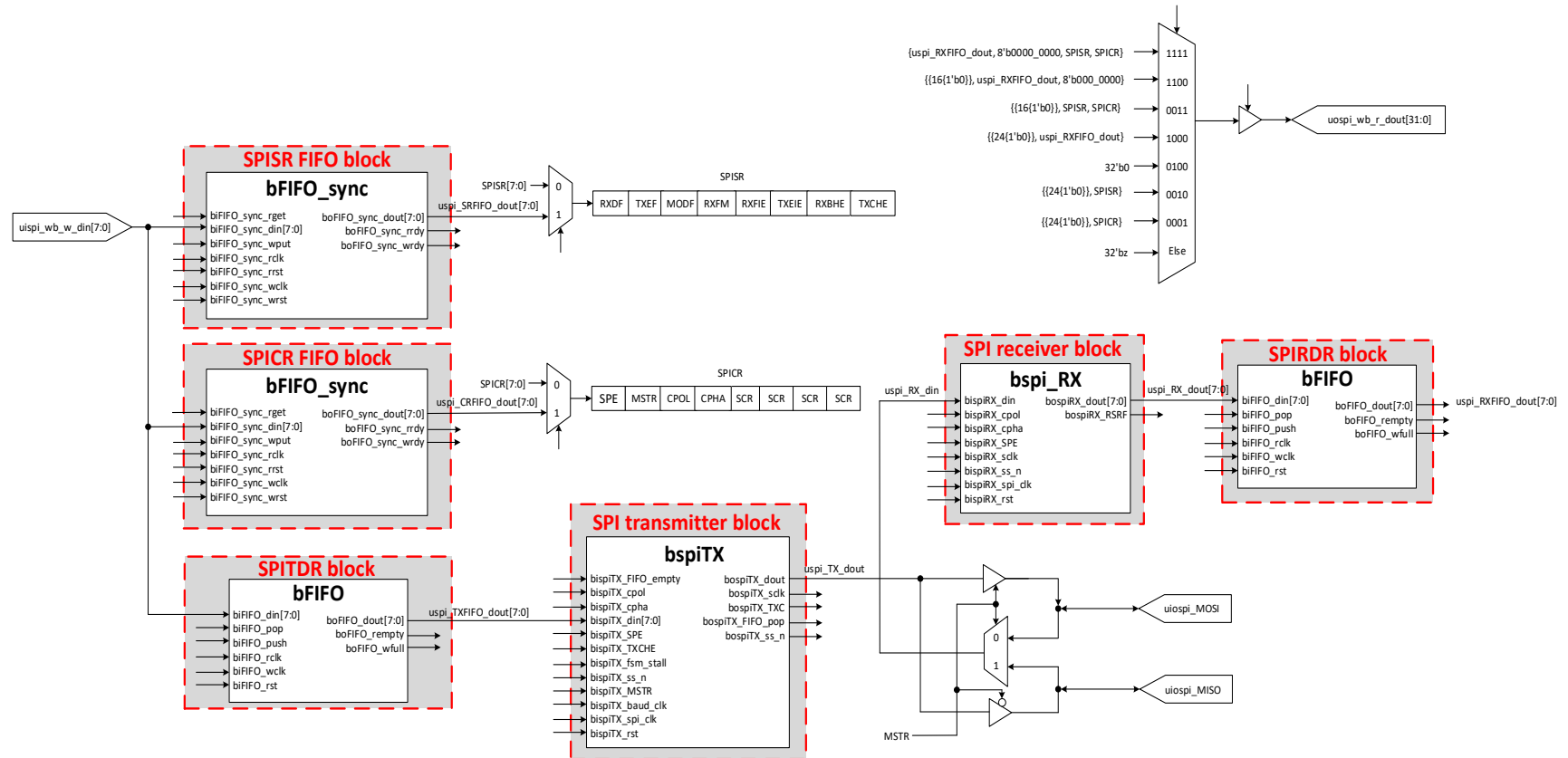
Table 5.1.8.1: Functional description of each SPI internal block

5.1.9 Micro-Architecture of the SPI Controller Unit (Block Level)



Note: The shaded areas indicate the internal blocks of the designed SPI controller

Figure 5.1.9.1: Simplified micro-architecture of the SPI controller unit.



Note: The shaded areas indicate the internal blocks of the designed SPI controller

Figure 5.1.9.2: Datapath of the SPI controller unit.

5.2 SPI Transmitter Block

5.2.1 Functionality/Feature of the SPI Transmitter Block

The SPI transmitter block is used to handle the serial data transmission of the SPI controller unit. It uses an 8-bit transmitter shift register (TSR) to store data loaded from the SPITDR block. As the baud clock pulses are generated, it transmits the data stored in the transmitter shift register (TSR) serially to another SPI device via one of the SPI standard pins, namely MISO or MOSI. The designed SPI transmitter block can perform the serial data transmission between SPI devices as shown in Appendix A correctly in all of the 4 transfer modes (mode 0, 1, 2, 3) regardless of whether it is configured as a master or a slave.

5.2.2 Block Interface of the SPI Transmitter Block

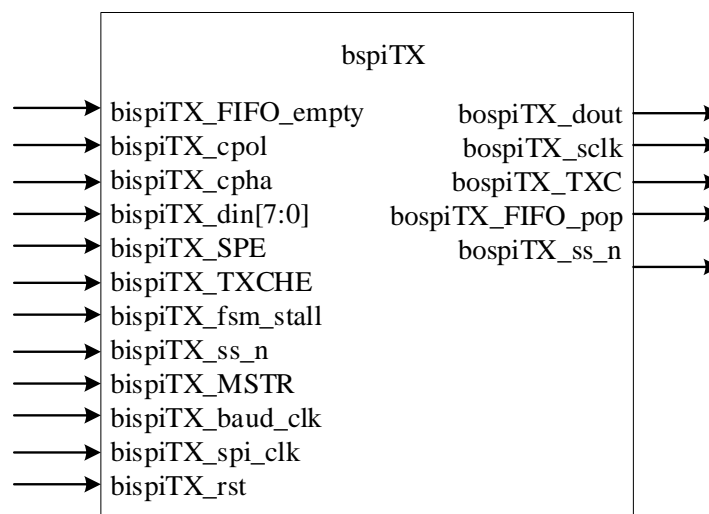


Figure 5.2.2.1: Block interface of the SPI transmitter block.

5.2.3 Input Pin Description of the SPI Transmitter Block

Pin name: bispiTX_FIFO_empty	Pin class: Status
Source → Destination: SPITDR block → SPI transmitter block	
Pin function:	
1: SPITDR block is empty	
0: SPITDR block is not empty	
Pin name: bispiTX_cpol	Pin class: Control
Source → Destination: SPICR → SPI transmitter block	
Pin function: Clock polarity	

Pin name: bispiTX_cpha	Pin class: Control
Source → Destination: SPICR → SPI transmitter block	
Pin function: Clock phase	
Pin name: bispiTX_din[7:0]	Pin class: Data
Source → Destination: SPICR → SPI transmitter block	
Pin function: 8-bit data input bus	
Pin name: bispiTX_SPE	Pin class: Control
Source → Destination: SPICR → SPI transmitter block	
Pin function: SPI controller enable control signal	
1: enable SPI transmitter block	
0: disable SPI transmitter block	
Pin name: bispiTX_TXCHE	Pin class: Control
Source → Destination: SPISR → SPI transmitter block	
Pin function: Transmit complete halt enable signal	
1: halt SPI transmitter's FSM state when complete transmission	
0: continue SPI transmitter's FSM state when complete transmission	
Pin name: bispiTX_fsm_stall	Pin class: Control
Source → Destination: SPI controller unit → SPI transmitter block	
Pin function: SPI transmitter stall control signal	
1: stall SPI transmitter's FSM state	
0: SPI transmitter's FSM state run normally	
Pin name: bispiTX_ss_n	Pin class: Control
Source → Destination: SPI controller unit → SPI transmitter block	
Pin function: Slave select input signal	
1: selected by master to communicate with	
0: not selected by master to communicate with/when it is configured as master	
Pin name: bispiTX_MSTR	Pin class: Control
Source → Destination: SPICR → SPI transmitter block	
Pin function: Master/Slave mode	
1: SPI is in master mode	
0: SPI is in slave mode	
Pin name: bispiTX_baud_clk	Pin class: Control
Source → Destination: SPI clock generator block → SPI transmitter block	
Pin function: Data synchronization clock source	
Pin name: bispiTX_spi_clk	Pin class: Control
Source → Destination: SPI clock generator block → SPI transmitter block	
Pin function: SPI I/O clock	
Pin name: bispiTX_rst	Pin class: Global
Source → Destination: Global reset → SPI transmitter block	
Pin function: Global reset	
1: reset	
0: no reset is required	

Table 5.2.3.1: Input pin description of the SPI transmitter block.

5.2.4 Output Pin Description of the SPI Transmitter Block

Pin name: bospiTX_dout	Pin class: Data
Source → Destination: SPI transmitter block → SPI controller unit	
Pin function: Serial data output	
Pin name: bospiTX_sclk	Pin class: Control
Source → Destination: SPI transmitter block → SPI controller unit	
Pin function: Data synchronization clock source	
Pin name: bospiTX_TXC	Pin class: Status
Source → Destination: SPI transmitter block → SPI controller unit	
Pin function:	
1: complete transmission of one data byte	
0: transmission of one data byte is not complete	
Pin name: bospiTX_FIFO_pop	Pin class: Control
Source → Destination: SPI transmitter block → SPITDR	
Pin function: To pop one byte of data from the SPITDR block	
Pin name: bospiTX_ss_n	Pin class: Control
Source → Destination: SPI transmitter block → SPI controller unit	
Pin function: Serial data valid control	
1: disable serial data communication	
0: enable serial data communication	

Table 5.2.4.1: Output pin description of the SPI transmitter block.

5.2.5 Finite State Machine of the SPI Transmitter Block

The SPI transmitter block that has been developed has a built-in FSM that is used to switched between different states. The change of state can be done based on the internal events. The details of the SPI transmitter's FSM are illustrated in Figure 5.2.6.1.

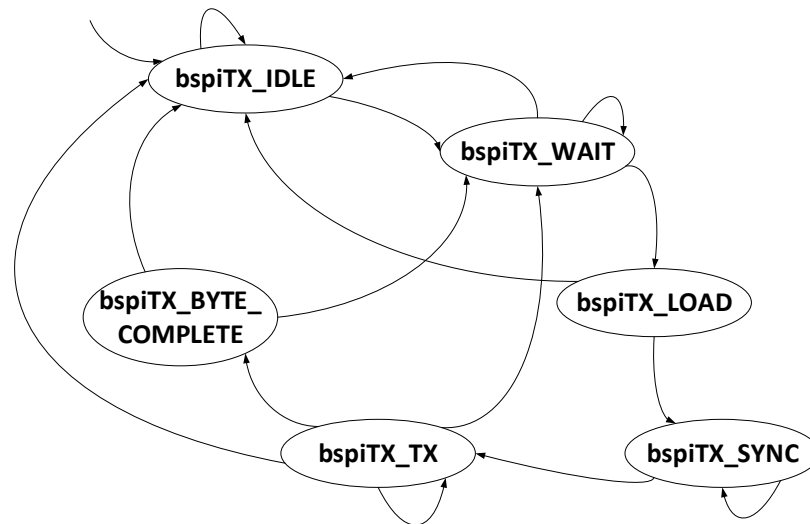


Figure 5.2.5.1: Finite State Machine of the SPI transmitter block.

State Name	Description
bspiTX_IDLE	No operation
bspiTX_WAIT	Wait state when FSM halt
bspiTX_LOAD	Load data to transmitter shift register (TSR)
bspiTX_SYNC	SCLK synchronizing
bspiTX_TX	Data transmission is in progress
bspiTX_BYTE_COMPLETE	One-byte data transmission is complete

Table 5.2.5.1: State description of the SPI transmitter block.

5.3 SPI Receiver Block

5.3.1 Functionality/Feature of the SPI Receiver Block

The SPI receiver block is responsible to handle the serial data receiving of the SPI controller. It uses an 8-bit receiver shift register (RSR) to receive each bit serially from another SPI device. After 8 baud clock cycles are generated by the master, the process of data exchange between the master and the slave is said to be completed. Thus, the receiver shift register (RSR) will contain the 8-bit received data. The data on the shift register (RSR) is transferred to the receiver data register (RDR) before it is later pushed into the SPIRDR block. The designed SPI receiver block can perform the serial data receiving between SPI devices in all of the 4 transfer modes (mode 0, 1, 2 and 3) as shown in Appendix A correctly regardless of whether it is configured as a master or a slave.

5.3.2 Block Interface of the SPI Receiver Block

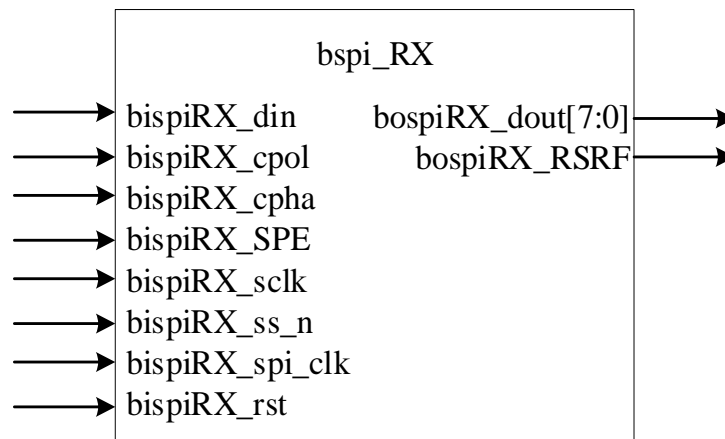


Figure 5.3.2.1: Block interface of the SPI receiver block.

5.3.3 Input Pin Description of the SPI Receiver Block

Pin name: bspiRX_din	Pin class: Data
Source → Destination: SPI controller unit → SPI receiver block	
Pin function: Serial data input	
Pin name: bspiRX_cpol	Pin class: Control
Source → Destination: SPICR → SPI receiver block	
Pin function: Clock polarity	

Pin name: bispiRX_cpha	Pin class: Control
Source → Destination: SPICR → SPI receiver block	
Pin function: Clock phase	
Pin name: bispiRX_SPE	Pin class: Control
Source → Destination: SPICR → SPI receiver block	
Pin function: SPI controller enable control signal	
1: enable SPI receiver	
0: disable SPI receiver	
Pin name: bispiRX_sclk	Pin class: Control
Source → Destination: SPI controller unit → SPI receiver block	
Pin function: Data synchronization clock source	
Pin name: bispiRX_ss_n	Pin class: Control
Source → Destination: SPI controller unit → SPI receiver block	
Pin function: Serial data valid control	
1: disable serial data communication	
0: enable serial data communication	
Pin name: bispiRX_spi_clk	Pin class: Control
Source → Destination: SPI clock generator block → SPI receiver block	
Pin function: SPI I/O clock	
Pin name: bispiRX_rst	Pin class: Global
Source → Destination: Global reset → SPI receiver block	
Pin function: Global reset	
1: reset	
0: no reset is required	

Table 5.3.3.1: Input pin description of the SPI receiver block.

5.3.4 Output Pin Description of the SPI Receiver Block

Pin name: bospiRX_dout[7:0]	Pin class: Data
Source → Destination: SPI receiver block → SPIRDR block	
Pin function: 8-bit data output bus	
Pin name: bospi_RSUF	Pin class: Status
Source → Destination: SPI receiver block → SPIRDR block	
Pin function:	
1: RSR is full	
0: RSR is not full	

Table 5.3.4.1: Output pin description of the SPI receiver block.

5.3.5 Finite State Machine of the SPI Receiver Block

The SPI receiver block that has been developed has a built-in FSM that is used to switched between different states. The change of state can be done based on the internal events. The details of the SPI receiver's FSM are illustrated in Figure 5.3.6.1.

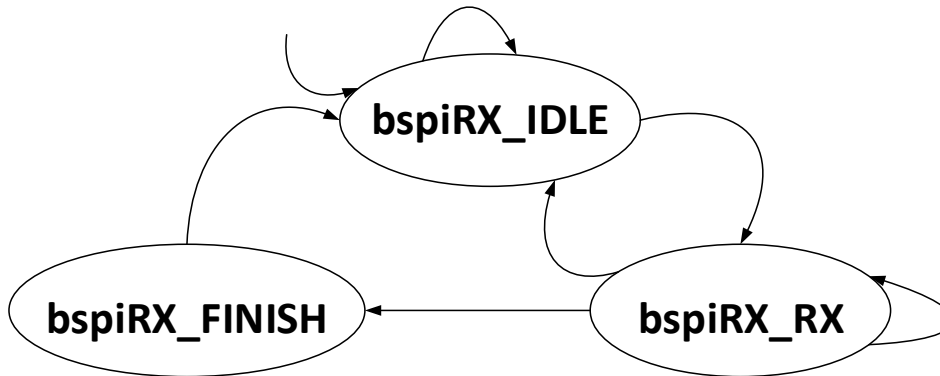


Figure 5.3.5.1: Finite State Machine of the SPI receiver block.

State Name	Description
bspiRX_IDLE	No operation
bspiRX_RX	Data receiving is in progress
bspiRX_FINISH	Data receiving is complete

Table 5.3.5.1: State description of the SPI receiver block.

5.4 SPI Clock Generator Block

5.4.1 Functionality/Feature of the SPI Clock Generator Block

The SPI clock generator block is used to generate the 10 MHz SPI I/O clock frequency for SPI internal operation. Besides, it is also a user-configurable clock divider. It is able to generate 16 transmission speed (or baud rate), ranging from 152 Hz to 5 MHz. Only the baud rate generated by the master device will be used for synchronizing the serial data transfer taking place across different SPI-compatible devices. The last four bits in the SPICR (which is the SPICR[3:0]) will control the divisor to the SPI I/O clock and determine the baud rate generation. The baud rate can be calculated by using the equation provided in below.

$$\text{baud rate (Hz)} = \frac{\text{SPI I/O clock(Hz)}}{2 \times 2^{\text{SPICR}[3:0]}}$$

5.4.2 Block Interface of the SPI Clock Generator Block

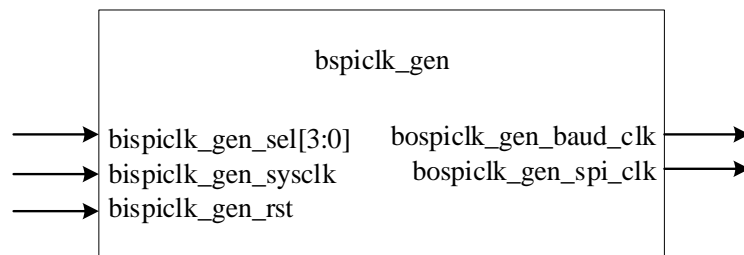


Figure 5.4.2.1: Block interface of the SPI clock generator block.

5.4.3 Input Pin Description of the SPI Clock Generator Block

Pin name: bspicl_gen_sel[3:0]	Pin class: Control
Source → Destination: SPI controller unit → SPI clock generator block	
Pin function: To select 1 out of 16 transmission speed or baud rate	
Pin name: bspicl_gen_sysclk	Pin class: Global
Source → Destination: Global → SPI clock generator block	
Pin function: Global clock	
Pin name: bspicl_gen_rst	Pin class: Global
Source → Destination: Global reset → SPI clock generator block	
Pin function: Global reset	
1: reset	
0: no reset is required	

Table 5.4.3.1: Input pin description of the SPI clock generator block.

5.4.4 Output Pin Description of the SPI Clock Generator Block

Pin name: bospiclk_gen_baud_clk	Pin class: Control
Source → Destination: SPI clock generator block → SPI transmitter block	
Pin function: To output the selected baud rate	
Pin name: bospiclk_gen_spi_clk	Pin class: Control
Source → Destination: SPI clock generator block → All SPI blocks and registers	
Pin function: To output the generated SPI I/O clock	

Table 5.4.4.1: Output pin description of the SPI clock generator block.

5.5 16-deep Asynchronous FIFO Block

5.5.1 Functionality/Feature of the 16-deep Asynchronous FIFO Block

One of the FIFO designs used in this project is the asynchronous FIFO design with asynchronous pointer comparisons. An asynchronous FIFO refers to a FIFO design where data values are written sequentially into a FIFO buffer using one clock domain, and the data values are sequentially read from the same FIFO buffer using another clock domain, where the two clock domains are asynchronous to each other (Cummings & Alfke, 2002). Basically, it is used to safely handle the passing of multi-bit data (randomly changing signals) from one clock domain to a new clock domain as the use of synchronizer to handle the passing of these type of data is generally unacceptable. The FIFO used in this project is a 16 entries deep FIFO memory with each data entry is of 8-bit size. It uses circular memory with two pointers to simulate the infinite big memory needed. As a result, multiple data bytes can be stored sequentially in it as long as the total number of data bytes does not exceed 16, which greatly reduces the CPU's workload to move the data and allow continuous data transfer in the background. Apart from that, it has implemented full-removal and empty-removal using a "pessimistic" method. Meaning, the "full" and "empty" statuses are both asserted exactly on time but removed late. In this project, it is used as the SPITDR and SPIRDR respectively.

5.5.2 Block Interface of the 16-deep Asynchronous FIFO Block

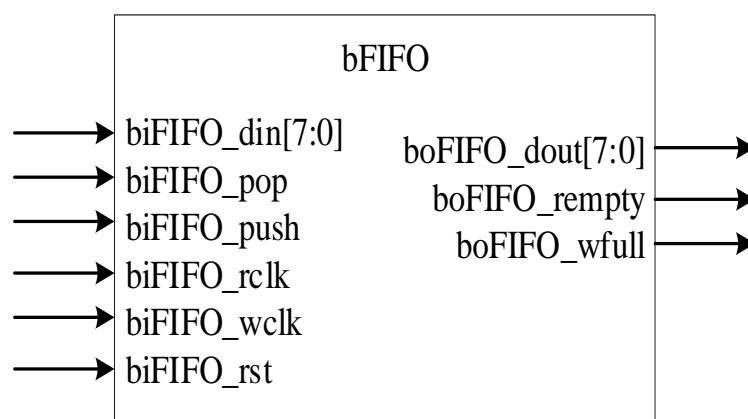


Figure 5.5.2.1: Block interface of the 16-deep asynchronous FIFO block.

5.5.3 Input Pin Description of the 16-deep Asynchronous FIFO Block

Pin name: biFIFO_din[7:0]	Pin class: Data
Source → Destination: SPI controller unit → Asynchronous FIFO block SPI receiver block → Asynchronous FIFO block	
Pin function: 8-bit input data bus	
Pin name: biFIFO_pop	Pin class: Control
Source → Destination: SPI controller unit → Asynchronous FIFO block SPI transmitter block → Asynchronous FIFO block	
Pin function: To pop one data byte from the asynchronous FIFO block 1: pop one data byte from the FIFO 0: no popping of one data byte from the FIFO	
Pin name: biFIFO_push	Pin class: Control
Source → Destination: SPI controller unit → Asynchronous FIFO block SPI receiver block → Asynchronous FIFO block	
Pin function: To push one data byte into the asynchronous FIFO block 1: push one data byte into the FIFO 0: no pushing of one data byte into the FIFO	
Pin name: biFIFO_rclk	Pin class: Control
Source → Destination: SPI controller unit → Asynchronous FIFO block SPI clock generator block → Asynchronous FIFO block	
Pin function: Read clock signal	
Pin name: biFIFO_wclk	Pin class: Control
Source → Destination: SPI controller unit → Asynchronous FIFO block SPI clock generator block → Asynchronous FIFO block	
Pin function: Write clock signal	
Pin name: biFIFO_rst	Pin class: Global
Source → Destination: Global reset → Asynchronous FIFO block	
Pin function: Global reset	

Table 5.5.3.1: Input pin description of the 16-deep asynchronous FIFO block.

5.5.4 Output Pin Description of the 16-deep Asynchronous FIFO Block

Pin name: boFIFO_dout[7:0]	Pin class: Data
Source → Destination: Asynchronous FIFO block → SPI transmitter block Asynchronous FIFO block → SPI controller unit	
Pin function: 8-bit output data bus	
Pin name: boFIFO_rempy	Pin class: Status
Source → Destination: Asynchronous FIFO block → SPI transmitter block Asynchronous FIFO block → SPI controller unit	
Pin function: 1: the FIFO is empty 0: the FIFO is not empty	
Pin name: boFIFO_wfull	Pin class: Status
Source → Destination: Asynchronous FIFO block → SPI controller unit	
Pin function: 1: the FIFO is full 0: the FIFO is not full	

Table 5.5.4.1: Output pin description of the asynchronous FIFO block.

5.5.5 Schematic and Block Diagram of the 16-deep Asynchronous FIFO block

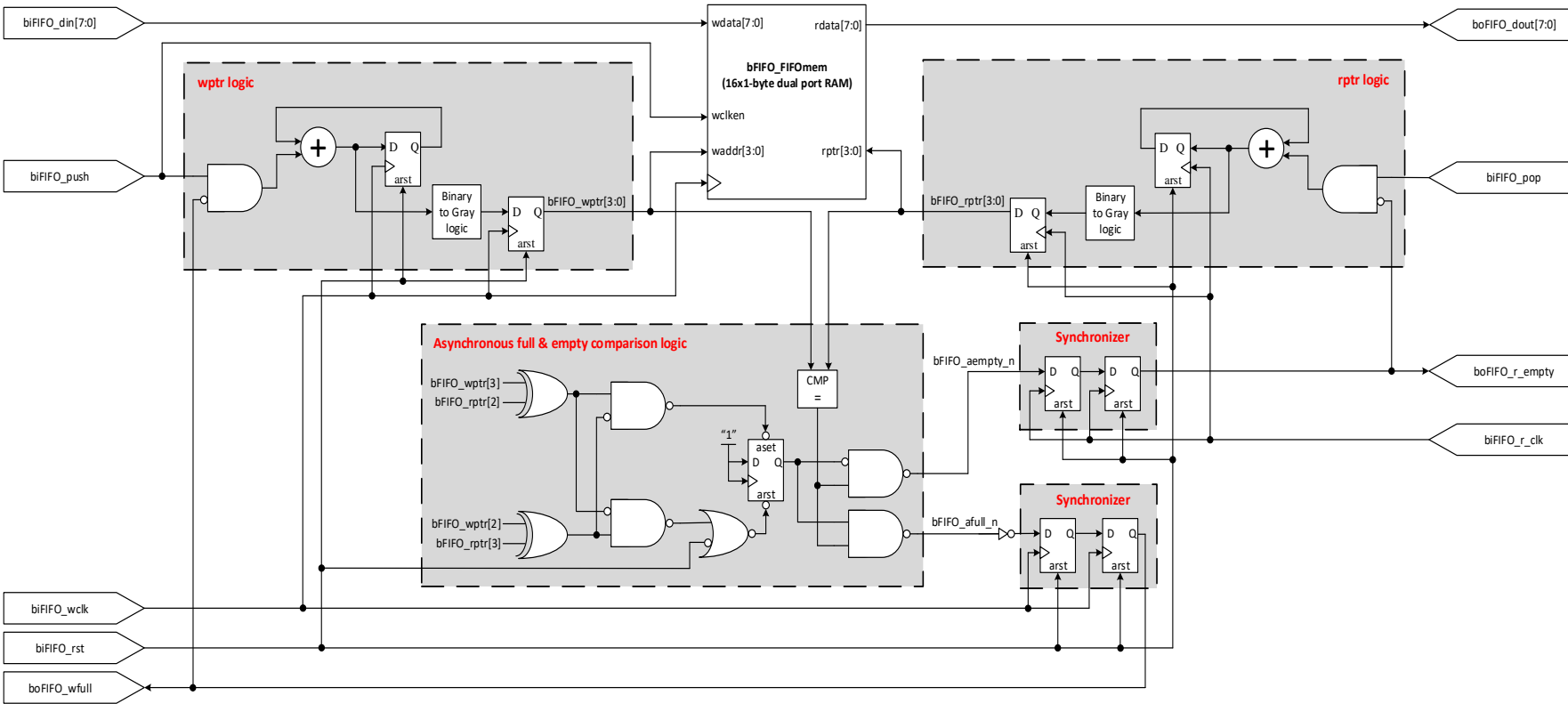


Figure 5.5.5.1: Schematic and block diagram of the 16-deep asynchronous FIFO design with asynchronous pointer comparisons

5.6 2-deep FIFO Synchronizer Block

5.6.1 Functionality/Feature of the 2-deep FIFO Synchronizer Block

The 2-deep FIFO synchronizer block is another variation on passing multiple control and data bits safely across CDC boundaries. It allows the CPU to buffer the multi-bit control signal at its own speed, thus reducing the timing requirement. It is a 2-deep dual port FIFO memory as it is built by using only two registers. Each data entry is of 8-bit size. Similarly, it also uses circular memory with two pointers to simulate the infinite big memory needed. On the other hand, an inverted not-full condition is used in this FIFO design to indicate that the FIFO is ready to receive a control byte. On the other hand, in order to indicate that the FIFO has a data or control byte that is ready to be read, an inverted not empty condition is applied. In this project, it is used as the SPICR FIFO and SPISR FIFO block.

5.6.2 Block Interface of the 2-deep FIFO Synchronizer Block

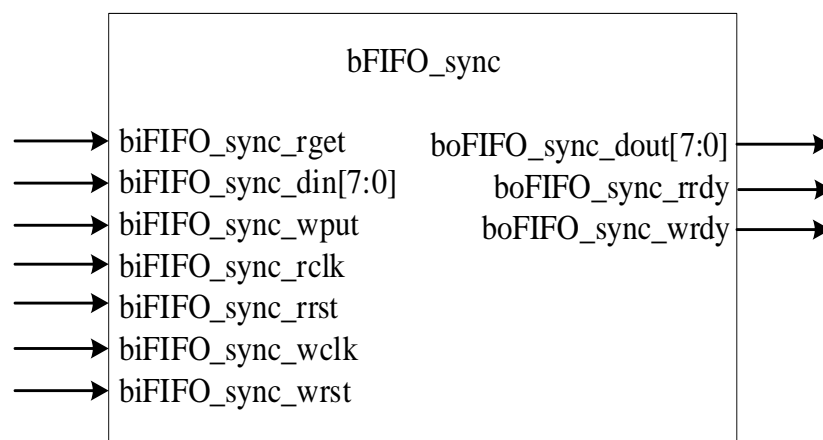


Figure 5.6.2.1: Block interface of the 2-deep FIFO synchronizer block.

5.6.3 Input Pin Description of the 2-deep FIFO Synchronizer Block

Pin name: biFIFO_sync_rget	Pin class: Control
Source → Destination: 2-deep FIFO synchronizer block → 2-deep FIFO synchronizer block	
Pin function: To get one data byte from the FIFO block 1: pop one data byte from the FIFO 0: no popping of one data byte from the FIFO	
Pin name: biFIFO_sync_din[7:0]	Pin class: Data
Source → Destination: SPI controller unit → 2-deep FIFO synchronizer block	
Pin function: 8-bit input data bus	
Pin name: biFIFO_sync_wput	Pin class: Control
Source → Destination: SPI controller unit → 2-deep FIFO synchronizer block	
Pin function: To push one data byte into the FIFO block 1: push one data byte into the FIFO 0: no pushing of one data byte into the FIFO	
Pin name: biFIFO_sync_rclk	Pin class: Control
Source → Destination: SPI clock generator block → 2-deep FIFO synchronizer block	
Pin function: Read clock signal	
Pin name: biFIFO_sync_rrst	Pin class: Global
Source → Destination: Global reset → 2-deep FIFO synchronizer block	
Pin function: Global reset 1: reset 0: no reset is required	
Pin name: biFIFO_sync_wclk	Pin class: Global
Source → Destination: Global clock → 2-deep FIFO synchronizer block	
Pin function: Global clock	
Pin name: biFIFO_sync_wrst	Pin class: Global
Source → Destination: Global reset → 2-deep FIFO synchronizer block	
Pin function: Global reset 1: reset 0: no reset is required	

Table 5.6.3.1: Input pin description of the 2-deep FIFO synchronizer block

5.6.4 Output Pin Description of the 2-deep FIFO Synchronizer Block

Pin name: biFIFO_sync_dout[7:0]	Pin class: Data
Source → Destination: 2-deep FIFO synchronizer block → SPICR 2-deep FIFO synchronizer block → SPISR	
Pin function: 8-bit output data bus	
Pin name: biFIFO_sync_rrdy	Pin class: Status
Source → Destination: 2-deep FIFO synchronizer block → 2-deep FIFO synchronizer block	
Pin function: 1: the data in the FIFO is ready to be read 0: the data in the FIFO is not ready to be read	
Pin name: biFIFO_sync_wrdy	Pin class: Status

<p>Source → Destination: 2-deep FIFO synchronizer block → SPI controller unit</p> <p>Pin function:</p> <p>1: FIFO is ready to receive a data</p> <p>0: FIFO is not ready to receive a data</p>

Table 5.6.4.1: Output pin description of the 2-deep FIFO synchronizer block.

5.6.5 Schematic and Block Diagram of the 2-deep FIFO Synchronizer Block

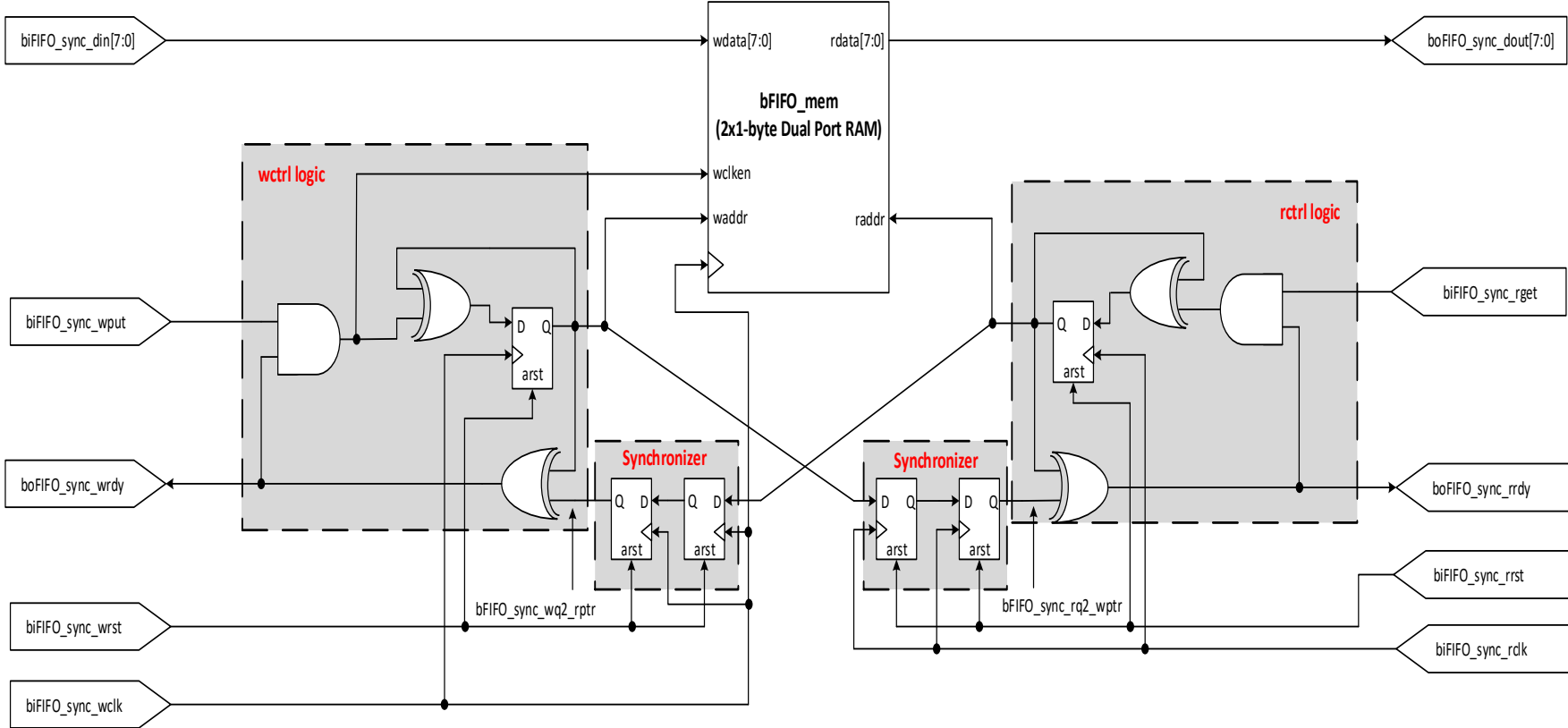


Figure 5.6.5.1: Schematic and block diagram of the 2-deep FIFO synchronizer block

5.7 Register Set of SPI Controller Unit

Four special-purpose registers are used to allow data communication between the CPU and the SPI controller unit. All these special registers are memory-mapped and user-accessible by using the normal load and store instructions.

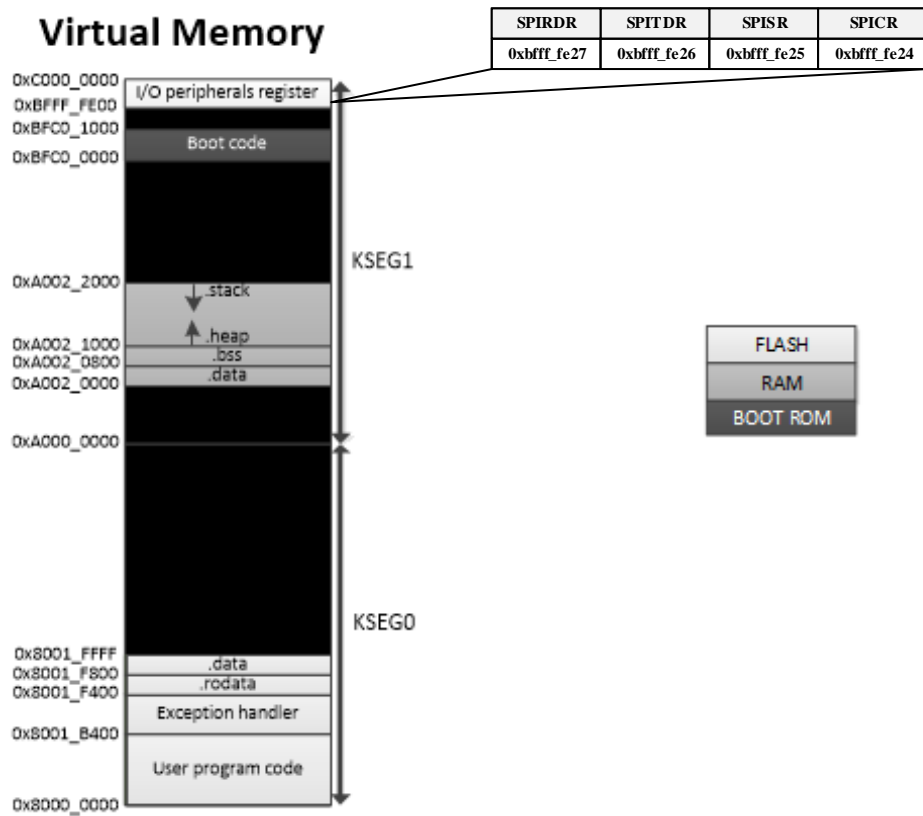


Figure 5.7.1: Address of the special-purpose registers in virtual memory.

5.7.1 SPI Configuration Register (SPICR)

Type: read/write

Width: 8 bits

Address: 0xbfff_fe24

Function: To configure the setting of the SPI controller unit.

SPE	MSTR	CPOL	CPHA	SCR	SCR	SCR	SCR
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

Figure 5.7.1.1: SPI Configuration Register (SPICR)

a) SPE - SPI enable control

It is used to deactivate the SPI controller when it is not in use. To have better control on power consumption, the SPI controller is recommended to be deactivated when it is not in use.

- SPE = 0: Deactivate SPI controller.
- SPE = 1: Activate SPI controller.

b) MSTR - Master/Slave device

- MSTR = 0: Set as slave device.
- MSTR = 1: Set as master device.

c) CPOL - Clock Polarity

- CPOL = 0: Active-high clock is selected. SCLK is low in idle state.
- CPOL = 1: Active-low clock is selected. SCLK is high in idle state.

d) CPHA - Clock Phase

- CPHA = 0: Data sampling occurs at odd edges (1, 3, 5, ..., 15) of the SCLK clock.
- CPHA = 1: Data sampling occurs at even edges (2, 4, 6, ..., 16) of the SCLK clock.

e) SCR [3:0] - SPI Baud Rate (Given the SPI I/O clock speed is 10 MHz)

- 0000: 5 MHz
- 0001: 2.5 MHz
- ...
- 1110: 305 Hz
- 1111: 152 Hz

5.7.2 SPI Status Register (SPISR)

Type: read/write

Width: 8 bits

Address: 0xbfff_fe25

Function: To configure the setting of the SPI controller unit and for status monitoring purpose.

RXDF	TXEF	MODF	RXFM	RXFIE	TXEIE	RXBHE	TXCHE
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

Figure 5.7.2.1: SPI Status Register (SPISR)

a) RXDF: Receive Done Flag

When this bit is set by the SPI controller unit, it indicates that 1-byte or 16-byte of data have been completely received. It is used in conjunction with the RXFM bit in the SPISR to determine if the received data is 1-byte (when RXFM = 0) or 16-byte (when RXFM = 1) when the FIFO is full.

b) TXEF: Transmit FIFO Empty flag

- TXEF = 0: SPI Transmitter FIFO block is not empty.
- TXEF = 1: SPI Transmitter FIFO block is empty.

c) MODF: Mode fault error

When the SPI controller unit is configured as a master device, the uiospi_SS_n pin must be pulled high by the master device. If there exist two or more master devices in the same connection, any attempt to pull low the uiospi_SS_n pin will trigger the mode fault error. This is to avoid damage to the hardware.

- MODF = 0: No mode fault error occurs.
- MODF = 1: Mode fault error occurs.

d) RXFM: Receive FIFO Full Mode.

It is part of the SPICR and it is placed in SPISR to avoid creating longer bytes of SPICR.

- RXFM = 0: 1-byte of data is expected to be read by CPU.
- RXFM = 1: 16-byte of data (FIFO full) is expected to be read by CPU.

e) RXFIE: Receive Complete Interrupt enable

It is part of the SPICR and it is placed in SPISR to avoid creating longer bytes of SPICR. It can only be used if and only if the SPIE bit (the SPI global interrupt enable pin) is set to high. This bit is used for interrupt enable control (to select interrupt method instead of polling) after data has been completely received (as indicated by the RXDF bit in SPISR).

- RXFIE = 0: Disable Receive Complete Interrupt.
- RXFIE = 1: Enable Receive Complete Interrupt.

f) TXEIE: Transmit FIFO Empty Interrupt Enable

It is part of the SPICR and it is placed in SPISR to avoid creating longer bytes of SPICR. It can only be used if and only if the SPIE bit (SPI global interrupt enable bit) is set to high. This bit is used for interrupt enable control (to select interrupt method instead of polling) when the SPI transmitter FIFO is empty (as indicated by the TXEF bit in SPISR).

- TXEIE = 0: Disable Transmit Enable Interrupt.
- TXEIE = 1: Enable Transmit Enable Interrupt.

g) RXBHE: Receive Byte Halt enable

It is part of the SPICR and it is placed in SPISR to avoid creating longer bytes of SPICR.

- RXBHE = 0: Disable SPI transmitter's FSM stall.
- RXBHE = 1: Enable SPI transmitter's FSM stall when one byte of data is received.

h) TXCHE: Transmit FIFO Complete Halt enable

It is part of SPICR and it is placed in SPISR to avoid creating longer bytes of SPICR.

- TXCHE = 0: Continue SPI transmitter's FSM state when complete transmission.
- TXCHE = 1: Halt SPI transmitter's FSM state when complete transmission.

5.7.3 SPI Transmitter Data register (SPITDR)

Type: read/write

Width: 8 bits

Address: 0xbfff_fe26

Function: To hold the data that will be transmitted to another SPI-compatible device.

Z	Z	Z	Z	Z	Z	Z	Z
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

Figure 5.8.3.1: SPI Transmitter Data Register (SPITDR)

5.8.4 SPI Receiver Data register (SPIRDR)

Type: read/write

Width: 8 bits

Address: 0xbfff_fe27

Function: To hold the data received from another SPI-compatible device.

Z	Z	Z	Z	Z	Z	Z	Z
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

Figure 5.7.4.1: SPI Receiver Data Register (SPIRDR)

Chapter 6: Firmware Development

6.1 Exception Handler of the RISC32 Pipeline Processor

Exception is the unexpected or unscheduled event (internally or externally caused) that can change the normal flow of the instruction execution. In order to handle the unexpected events within the processor, an exception handler has already been created. The exception handler is able to handle unexpected events such as interrupt, address error trap on data load or instruction fetch, address error trap on data store, bus error on data load or store, bus error on instruction fetch, Syscall trap, breakpoint trap, undefined instruction trap, and arithmetic overflow trap. Upon detecting an exception signal, the CPU suspends its current program execution, saves the address of the next instruction (PC) for return purpose, and then jumps to the exception handler at 0x8000_b400 for handling the exception. After performing whatever actions that are required because of the exception, the CPU returns to the place where it was interrupted and resumes the normal program execution. A pseudocode that describes the existing exception handler of the RISC32 pipeline processor is given in below for better understanding.

```

BEGIN
    Push the current state of the user program to stack
    Push the current CP0's status register value to stack
    Push the current CP0's cause register value to stack
    Clear the exception level in the CP0's status register
    Decode the exception code in the CP0's cause register
    CASEOF exception code
        0: Branch to the exception routine of Interrupt
        4: Branch to the exception routine of Address Error Trap LOAD
        5: Branch to the exception routine of Address Error Tap STORE
        6: Branch to the exception routine of Bus Error on IF Trap
        7: Branch to the exception routine of Bus Error on LOAD/STORE Tap
        8: Branch to the exception routine of Syscall
        9: Branch to the exception routine of Breakpoint Trap
        10: Branch to the exception routine of Reserved/Undefined Instruction
        12: Branch to the exception routine of Arithmetic Overflow
    ENDCASE
    Set the exception level in the CP0's status register
    Pop the previous state of user program from stack
    Clear the exception code in the CP0's cause register
    Clear the interrupt priority level in the CP0's cause register
    Return to user program based on the address value in the CP0's EPC register
END

```

6.2 Interrupt Service Routine (ISR) of the SPI Controller Unit

An interrupt is an external event that interrupts the CPU to inform it that a device needs its service. In this project, the device that interrupts the CPU will be the SPI controller unit. Since interrupt is asynchronous to the program execution, the CPU will simply suspend the normal instruction execution and then resume to the place where it was interrupted after finishing the execution of the corresponding Interrupt Service Routine (ISR).

In this project, an ISR specifically for the SPI controller unit is developed by using MIPS assembly language and subsequently integrated into the existing exception handler. This ISR will be invoked by the CPU to handle different types of interrupt requests generated by the SPI controller unit. The generated interrupt request(s) could be mode fault error interrupt, transmitter buffer empty interrupt and/or receiver buffer full interrupt. Table 6.2.1 describes the actions that the CPU performs when the corresponding SPI controller's interrupt request is generated.

Types of interrupt	Actions to be taken
Mode fault error interrupt	The CPU will deactivate the SPI controller unit and force it into the idle state. The transmission will be aborted if a transmission is in progress when the mode fault error occurs.
Transmitter buffer empty interrupt	When there are no more data bytes to be transmitted, the CPU will deactivate the SPI controller unit to reduce power consumption. Eventually, the SPI controller is forced into the idle state.
Receiver buffer full interrupt	The CPU will load the received data bytes from the receiver buffer to process.

Below shows the pseudocode of the developed ISR for handling different types of interrupt requests from the SPI controller unit.

```

BEGIN
    Load the SPISR value
    Check the mode fault error (MODF) interrupt status from the SPISR
    Check the transmitter buffer empty (TXEF) interrupt status from the SPISR
    IF the MODF or the TXEF interrupt occurs THEN
        Deactivate the SPI controller
  
```

```
ENDIF
Check the receiver buffer full (RXDF) interrupt status from the SPISR
IF the RXDF interrupt does not occur THEN
    Return to the main exception handler

ELSE
    BEGIN
        Load the received data bytes from the receiver buffer
        Return to the main exception handler
    END
END
```

Chapter 7: Verification Specification and Stimulation Result

The test cycle in this project consists of two stages. During the first stage, the SPI controller unit is tested and verified individually for functional correctness before any integration. Once it has passed the individual test, it will be tested again as a whole in the second stage for its complete functionality.

7.1 Test Plan for SPI Controller Unit’s Functional Test

Before the designed SPI controller unit can be integrated into the RISC32 pipeline processor, it is important to first verify the functional behaviors of the SPI controller unit. Thus, a test plan detailing the functional testing objective, scope, approach, expected output, and final test status is constructed and presented in Table 7.1.2. A testbench based on the test plan is written in Verilog HDL and can be found in Appendix B.1.

As shown in Figure 7.1.1, the DUT_MASTER used in this verification test represents the master device whereas the DUT_SLAVE used represents the slave device. Same type of SPI controller unit is used throughout the verification process but with different operation modes (master mode or slave mode) to prove that the designed SPI controller unit can function correctly in both operation modes.

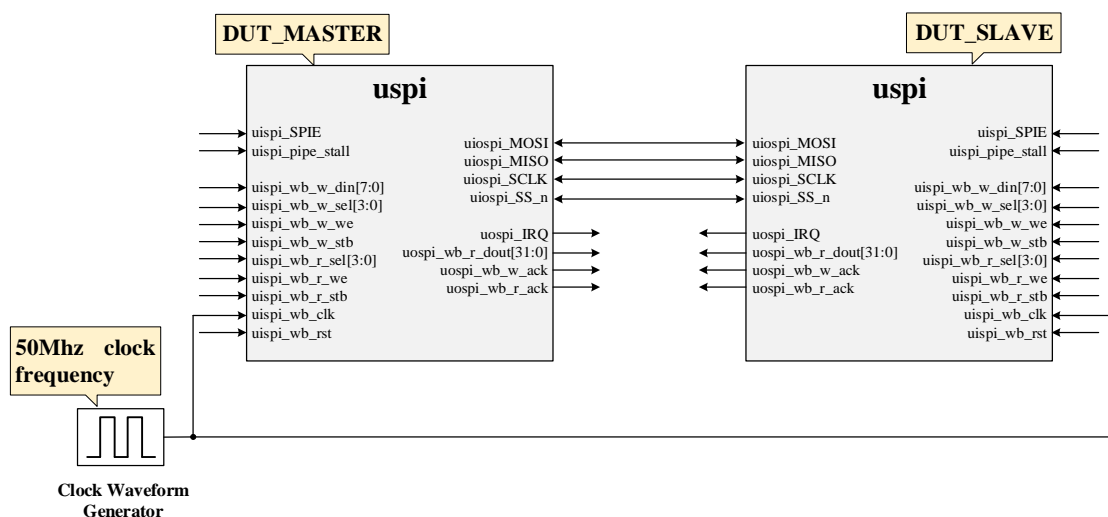


Figure 7.1.1: The connection mechanism of the DUT_MASTER and the DUT_SLAVE for SPI controller unit’s functional verification.

Unit/Block Used in Stimulation	Instance Name Used in Stimulation
SPI controller unit (configured as master)	DUT_MASTER
SPI controller unit (configured as slave)	DUT_SLAVE
SPI transmitter block	bspITX
SPI receiver block	bspIRX
SPI Transmitter Data Register	TX_buffer16x8
SPI Receiver Data Register	RX_buffer16x8
SPICR FIFO block	SPICR_buffer2x8
SPISR FIFO block	SPISR_buffer2x8
SPI clock generator block	bspicl_gen

Table 7.1.1 Instance name of each SPI controller unit and its internal blocks that are being used in the test plans, testbenches, flowcharts and stimulation.

Test	Expected Output	Status
<p>Test Case #1: System Reset</p> <p>Function to be tested</p> <ul style="list-style-type: none"> Able to reset the whole SPI controller unit <p>Procedure</p> <ol style="list-style-type: none"> Reset both devices. 	<ul style="list-style-type: none"> tb_w_uiospi_SS_n = 1'bz tb_w_uiospi_SCLK = 1'bz tb_w_uiospi_MOSI = 1'bz tb_w_uiospi_MISO = 1'bz tb_w_uospi_IRQ_master = 1'b0 tb_w_uospi_wb_w_ack_master = 1'b0 tb_w_uospi_wb_r_ack_master = 1'b0 tb_w_uospi_wb_r_dout_master = 32'hz SPICR of DUT_MASTER = 8'b0000_0000 SPISR of DUT_MASTER = 8'b0100_0000 tb_w_uospi_IRQ_slave = 1'b0 tb_w_uospi_wb_w_ack_slave = 1'b0 tb_w_uospi_wb_r_ack_slave = 1'b0 tb_w_uospi_wb_r_dout_slave = 32'hz SPICR of DUT_SLAVE = 8'b0000_0000 	Pass

	<ul style="list-style-type: none"> • SPISR of DUT_SLAVE = 8'b0100_0000 	
<p>Test Case #2: Write operation on SPISR</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Write operation on SPISR in both master mode and slave mode <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable write access on both devices. 2. Enable write operation on both SPISR. 3. Configure both SPISR with 8'b0000_1111. 4. Disable write operation on both SPISR. 	<ul style="list-style-type: none"> • SPISR of DUT_MASTER = 8'b0100_1111 • SPISR of DUT_SLAVE = 8'b0100_1111 	Pass
<p>Test case #3: Write operation on SPICR</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Write operation on SPICR in both master mode and slave mode <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable write operation on both SPICR. 2. Configure DUT_MASTER's SPICR with 8'b1100_0000. 3. Configure DUT_SLAVE's SPICR with 8'b1000_0000. 4. Disable write operation on both SPICR. 5. Enable read operation on both SPISR and SPICR of the devices. 	<ul style="list-style-type: none"> • SPICR of DUT_MASTER = 8'b1100_0000 • SPICR of DUT_SLAVE = 8'b1000_0000 	Pass
<p>Test Case #4: Transmitter buffer empty interrupt support</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Transmitter buffer empty interrupt support in both master mode and slave mode <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Hold for 20 clock cycles 2. Disable read operation on both SPISR and SPICR of the devices. 	<ul style="list-style-type: none"> • SPISR of DUT_MASTER = 8'b0100_1111 • tb_w_uospi_IRQ_master = 1'b1 • tb_w_uospi_wb_r_dout_master = 32'h0000_4fc0 • SPISR of DUT_SLAVE = 8'b0100_1111 • tb_w_uospi_IRQ_slave = 1'b1 • tb_w_uospi_wb_r_dout_slave = 32'h0000_4f80 	Pass

<p>Test Case #5: Push one 8-bit data into the TX_buffer16x8</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Able to load data into the TX_buffer16x8 in both master mode and slave <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable write operation on both TX_buffer16x8. 2. Load 8'b1010_1010 to DUT_MASTER's TX_buffer16x8. 3. Store 8'b0101_0101 to DUT_SLAVE's TX_buffer16x8. 4. Disable write operation on both TX_buffer16x8. 5. Enable read operation on both SPISR. 6. Hold for 20 clock cycles. 7. Disable read operation on both SPISR. 	<ul style="list-style-type: none"> • bFIFO_FIFOmem[0] of DUT_MASTER = 8'b1010_1010 • SPISR of DUT_MASTER = 8'b0000_1111 • tb_w_uospi_IRQ_master = 1'b0 • bFIFO_FIFOmem[0] of DUT_SLAVE = 8'b0101_0101 • SPISR of DUT_SLAVE = 8'b0000_1111 • tb_w_uospi_IRQ_slave = 1'b0 	Pass
<p>Test case #6: Mode 0 serial data communication</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Able to transmit and receive data simultaneously in both master mode and slave mode when mode 0 is used. <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable write operation on both TX_buffer16x8. 2. Store 8'b1010_1010 to DUT_MASTER's TX_buffer16x8. 3. Store 8'b0101_0101 to DUT_SLAVE's TX_buffer16x8. 4. Disable write operation on both TX_buffer16x8. 5. Hold for 100 clock cycles. 	<ul style="list-style-type: none"> • tb_w_uiospi_SS_n = 1'b0 • 8 baud clock cycles should appear on the tb_w_uiospi_SCLK • There should be only one bit of data on the tb_w_uiospi_MOSI for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b1010_1010 • There should be only one bit of data on the tb_w_uiospi_MISO for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b0101_0101 • Each of these data bit is transmitted serially at one half clock cycle before the rising edge of the tb_w_uiospi_SCLK clock. • tb_w_uospi_IRQ_master = 1'b1 	Pass

	<ul style="list-style-type: none"> • <code>tb_w_uospi_IRQ_slave = 1'b1</code> 	
<p>Test case 7: Receiver buffer full interrupt support after receiving a 1-byte data (RXFM = 0)</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Receiver buffer full interrupt support in both master mode and slave mode when RXFM = 0 <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Hold for 15 clock cycles. 	<ul style="list-style-type: none"> • SPISR of DUT_MASTER = <code>8'b1000_1111</code> • <code>tb_w_uospi_IRQ_master = 1'b1</code> • SPISR of DUT_SLAVE = <code>8'b1000_1111</code> • <code>tb_w_uospi_IRQ_slave = 1'b1</code> 	Pass
<p>Test case 8: Pop 1-byte of received data from the RX_buffer16x8</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Pop 1-byte of received data from the RX_buffer16x8 in both master mode and slave mode <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable read operation on both RX_buffer16x8. 2. Disable read operation on RX_buffer16x8 3. Enable read operation on both SPISR and SPICR of the devices. 4. Hold for 5 clock cycles. 5. Disable read operation on both SPISR and SPICR of the devices. 	<ul style="list-style-type: none"> • <code>tb_w_uospi_wb_r_dout_master = 32'h00000055</code> • SPISR of DUT_MASTER = <code>8'b0000_1111</code> • <code>tb_w_uospi_IRQ_master = 1'b0</code> • <code>tb_w_uospi_wb_r_dout_slave = 32'h000000aa</code> • SPISR of DUT_SLAVE = <code>8'b0000_1111</code> • <code>tb_w_uospi_IRQ_slave = 1'b0</code> 	Pass
<p>Test case #9: Receiver buffer full interrupt support after receiving 16x1-byte data (RXFM = 1)</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Receiver buffer full interrupt support in both master mode and slave mode when RXFM = 1 <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable write operation on both SPICR. 2. Configure DUT_MASTER's SPICR with <code>8'b0100_0000</code>. 3. Configure DUT_SLAVE's SPICR with <code>8'b0000_0000</code>. 4. Disable write operation on both SPICR and enable write operation on both SPISR. 	<ul style="list-style-type: none"> • SPISR of DUT_MASTER = <code>8'b1000_1111</code> • <code>tb_w_uospi_IRQ_master = 1'b1</code> • SPISR of DUT_SLAVE = <code>8'b1000_1111</code> • <code>tb_w_uospi_IRQ_slave = 1'b1</code> • All the received data should be stored correctly on the RX_buffer16x8 of both devices. 	Pass

<ol style="list-style-type: none"> 5. Configure both SPISR with 8'b0001_1111. 6. Disable write operation on both SPISR and enable write operation on both TX_buffer16x8. 7. Store 16-1byte data into both TX_buffer16x8. 8. Disable write operation on TX_buffer16x8 and enable write operation on SPICR. 9. Configure DUT_MASTER's SPICR with 8'b1100_0000. 10. Configure DUT_SLAVE's SPICR with 8'b1000_0000. 11. Disable write operation on both SPICR. 12. Enable read operation on SPISR and SPICR of the devices. 13. Hold for 1640 clock cycles. 14. Enable write operation on both TX_buffer16x8. 15. Store one 8-bit data into both TX_buffer16x8. 16. Disable write operation on both TX_buffer16x8. 		
<p>Test case 10: Pop 16 number of 1-byte data from the RX_buffer16x8</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Able to pop data from the RX_buffer16x8 in both master mode and slave mode <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable the read operation on both RX_buffer16x8. 2. Hold for 18 clock cycles. 3. Disable the read operation on both RX_buffer16x8. 	<ul style="list-style-type: none"> • tb_w_uospi_IRQ_master = 1'b0 • SPISR of DUT_MASTER = 8'b0001_1111 • tb_w_uospi_IRQ_slave = 1'b0 • SPISR of DUT_SLAVE = 8'b0001_1111 • All the received data should be loaded correctly on the tb_w_uospi_wb_r_dout_master and the tb_w_uospi_wb_r_dout_slave. 	Pass
<p>Test case 11: Mode 1 serial data communication</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Able to transmit and receive data simultaneously in both master mode and slave mode when mode 1 is used. 	<ul style="list-style-type: none"> • tb_w_uiospi_SS_n = 1'b0 • 8 baud clock cycles should appear on the tb_w_uiospi_SCLK. • There should be only one bit of data on the tb_w_uiospi_MOSI for every baud clock cycles. The expected sequence of 	Pass

<p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable write operation on both SPICR. 2. Configure DUT_MASTER's SPICR with 8'b0101_0000. 3. Configure DUT_SLAVE's SPICR with 8'b0001_0000. 4. Disable write operation on SPICR and enable write operation on SPISR. 5. Configure both SPISR with 8'b0000_1111. 6. Disable write operation on both SPISR and enable write operation on both TX_buffer16x8. 7. Store 16x1-byte data into both TX_buffer16x8. 8. Disable write operation on both TX_buffer16x8 and enable write operation on both SPICR. 9. Configure DUT_MASTER's SPICR with 8'b1101_0000. 10. Configure DUT_SLAVE's SPICR with 8'b1001_0000. 11. Disable write operation on both SPICR. 12. Enable read operation on SPISR and SPICR of the devices. 13. Hold for 140 clock cycles 14. Disable the read operation on SPISR and SPICR of the devices. 	<p>data to be transmitted on this pin is 8'b1010_1010.</p> <ul style="list-style-type: none"> • There should be only one bit of data on the tb_w_uiospi_MISO for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b0101_0101. • Each of these data bit is transmitted serially at the rising edge of the tb_w_uiospi_SCLK clock. • tb_w_uospi_IRQ_master = 1'b1 • tb_w_uospi_IRQ_slave = 1'b1 	
<p>Test case #12: Mode 2 serial data communication</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Able to transmit and receive data simultaneously in both master mode and slave mode when mode 2 is used. <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable write operation on both SPICR. 2. Configure DUT_MASTER's SPICR with 8'b1110_0000. 3. Configure DUT_SLAVE's SPICR with 8'b1010_0000. 4. Disable the write operation on both SPICR. 5. Enable the read operation on both RX_buffer16x8. 	<ul style="list-style-type: none"> • tb_w_uiospi_SS_n = 1'b0 • 8 baud clock cycles should appear on the tb_w_uiospi_SCLK. • There should be only one bit of data on the tb_w_uiospi_MOSI for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b0101_0101. • There should be only one bit of data on the tb_w_uiospi_MISO for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b1010_1010. 	Pass

<ol style="list-style-type: none"> 6. Disable the read operation on both RX_buffer16x8 and enable the read operation on the SPISR and SPICR. 7. Hold for 100 clock cycles 8. Disable the read operation on SPISR and SPICR of the devices. 	<ul style="list-style-type: none"> • Each of these data bit is transmitted serially at one half clock cycle before the falling edge of the tb_w_uiospi_SCLK clock. • tb_w_uospi_IRQ_master = 1'b1 • tb_w_uospi_IRQ_slave = 1'b1 	
<p>Test case #13: Mode 3 serial data communication</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Able to transmit and receive data simultaneously in both master mode and slave mode when mode 3 is used. <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable write operation on both SPICR. 2. Configure DUT_MASTER's SPICR with 8'b1111_0000. 3. Configure DUT_SLAVE's SPICR with 8'b1011_0000. 4. Disable the write operation on both SPICR. 5. Enable the read operation on both RX_buffer16x8. 6. Disable the read operation on both RX_buffer16x8 and enable the read operation on the SPISR and SPICR. 7. Hold for 110 clock cycles. 8. Disable the read operation on SPISR and SPICR of the devices. 	<ul style="list-style-type: none"> • tb_w_uiospi_SS_n = 1'b0 • 8 baud clock cycles should appear on the tb_w_uiospi_SCLK. • There should be only one bit of data on the tb_w_uiospi_MOSI for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b1010_1010. • There should be only one bit of data on the tb_w_uiospi_MISO for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b0101_0101. • Each of these data bit is transmitted serially at the falling edge of the tb_w_uiospi_SCLK clock. • tb_w_uospi_IRQ_master = 1'b1 • tb_w_uospi_IRQ_slave = 1'b1 	Pass
<p>Test case #14: Selectable transmission speed (baud rate)</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Support selectable transmission speed (baud rate) <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable write operation on both SPICR. 2. Configure DUT_MASTER's SPICR with 8'b1111_0001. 3. Configure DUT_SLAVE's SPICR with 8'b1011_0001. 	<ul style="list-style-type: none"> • Firstly, the uiospi_SCLK is expected to be 4 times slower than the bosp_i_gen_spi_clk of DUT_MASTER. • Next, the uiospi_SCLK is expected to be 8 times slower than the bosp_i_gen_spi_clk of DUT_MASTER. • Lastly, the uiospi_SCLK is expected to be 16 times slower than the 	Pass

<ol style="list-style-type: none"> 4. Disable the write operation on both SPICR. 5. Enable the read operation on both RX_buffer16x8. 6. Disable the read operation on both RX_buffer16x8 and enable the read operation on the SPISR and SPICR. 7. Hold for 220 clock cycles. 8. Disable the read operation on SPISR and SPICR of the devices. 9. Repeat step 1 to 8 by replacing some values in the above steps. That is, assign 8'b1111_0010 to the DUT_MASTER's SPICR in step 2 and 8'b1011_0010 to the DUT_SLAVE's SPICR in step 3, and hold 450 clock cycles in step 7. 10. Enable write operation in both SPISR. 11. Configure both SPISR with 8'b0000_0011. 12. Repeat step 1 to 8 by replacing some values in the above steps. That is, assign 8'b1111_0011 to the DUT_MASTER's SPICR in step 2 and 8'b1011_0011 to the DUT_SLAVE's SPICR in step 3, and hold 660 clock cycles in step 7. 13. Enable read operation on SPISR and SPICR of the devices. 	<p>bospi_gen_spi_clk of DUT_MASTER.</p>	
<p>Test case #15: Mode Fault Error Interrupt Support</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Mode fault error support in master mode <p><u>Procedure</u></p> <ol style="list-style-type: none"> 1. Enable write operation on DUT_SLAVE's SPICR. 2. Configure DUT_SLAVE's SPICR with 8'b1111_0011. 3. Disable write operation on DUT_SLAVE's SPICR. 4. Enable read operation on DUT_SLAVE's RX_buffer16x8. 5. Hold for 120 clock cycles. 	<ul style="list-style-type: none"> • tb_w_uospi_IRQ_master = 1'b1 • SPISR of DUT_MASTER = 8'b1010_0011 • tb_w_uospi_IRQ_slave = 1'b0 • SPISR of DUT_SLAVE = 8'b0000_0011 	<p>Pass</p>

Table 7.1.2: Test plan for the SPI controller unit's functional verification

7.2 Stimulation Results of the SPI Controller Unit's Functional Test

7.2.1 Test Case #1: System Reset

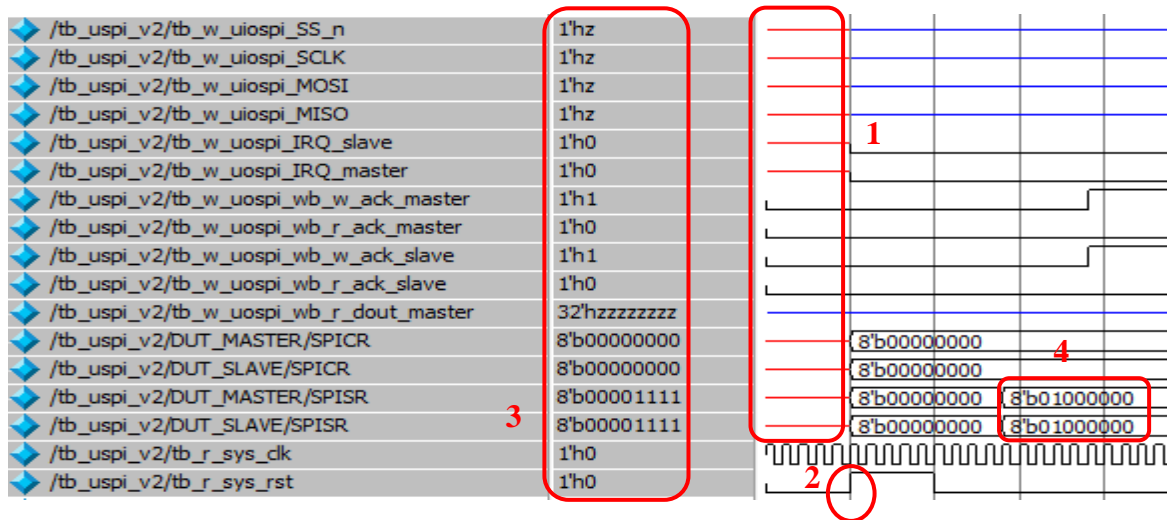


Figure 7.2.1.1: Stimulation result for test case #1 using ModelSim simulator.

1. Before the reset signal is asserted, most of the signals have an unknown value.
2. The reset signal is asserted in order to initialize all the system registers, FSM state, and read/write pointers.
3. All of the output signals and registers are set to their respective default value.
4. The bit 6 (TXEF) of the SPISR in both of the DUT_MASTER and the DUT_SLAVE is set to 1 by itself respectively when the reset signal is de-asserted as the TX_buffer16x8 block of each device is initially empty. At the end, both SPISRs hold the value of 8'b0100_0000 (0x40) respectively.

7.2.2 Test Case #2: Write Operation on SPISR

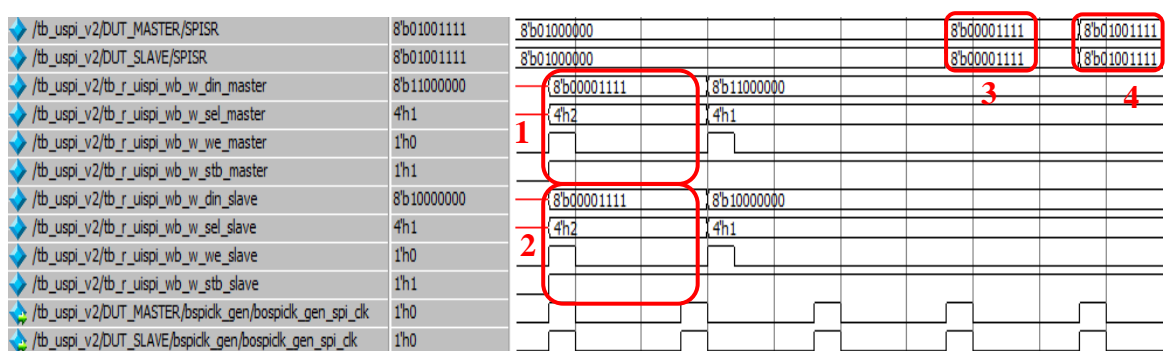


Figure 7.2.2.1 Stimulation result for test case #2 using ModelSim simulator.

1. Write operation on the SPISR of the DUT_MASTER is activated and an input data of 8'b0000_1111 (0x0f) is sent to it.

7.2.4 Test Case #4: Transmitter Buffer Empty Interrupt Support

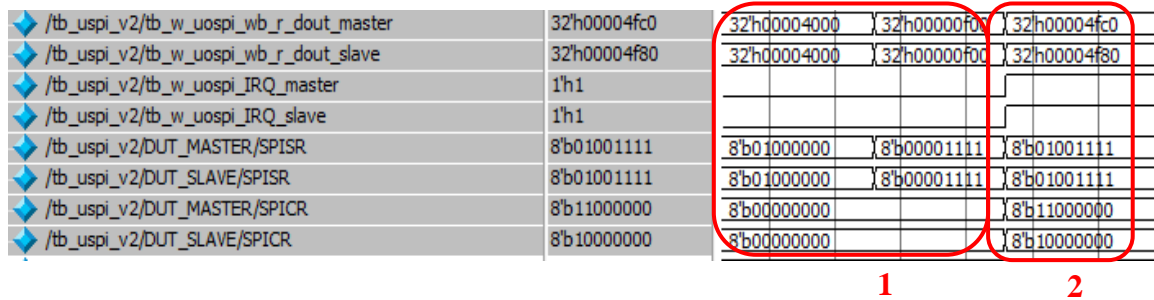


Figure 7.2.4.1: Stimulation result for test case #4 using ModelSim simulator.

1. Even though the bit 7 (TXEF) of both SPISR is asserted respectively (which means the TX_buffer16x8 is now empty), no interrupt request is being generated from both of the DUT_MASTER and the DUT_SLAVE. This is because both of the SPI controllers are initially de-activated (because SPE = 0) and their transmitter buffer empty interrupts are not enabled (because TXEIE = 0).
2. When all the conditions are met (TXEF = 1, SPE = 1, TXEIE = 1), both of the DUT_MASTER and the DUT_SLAVE will generate an interrupt request to notify the CPU for service.

7.2.5 Test Case #5: Push One 8-bit Data into the TX_buffer16x8

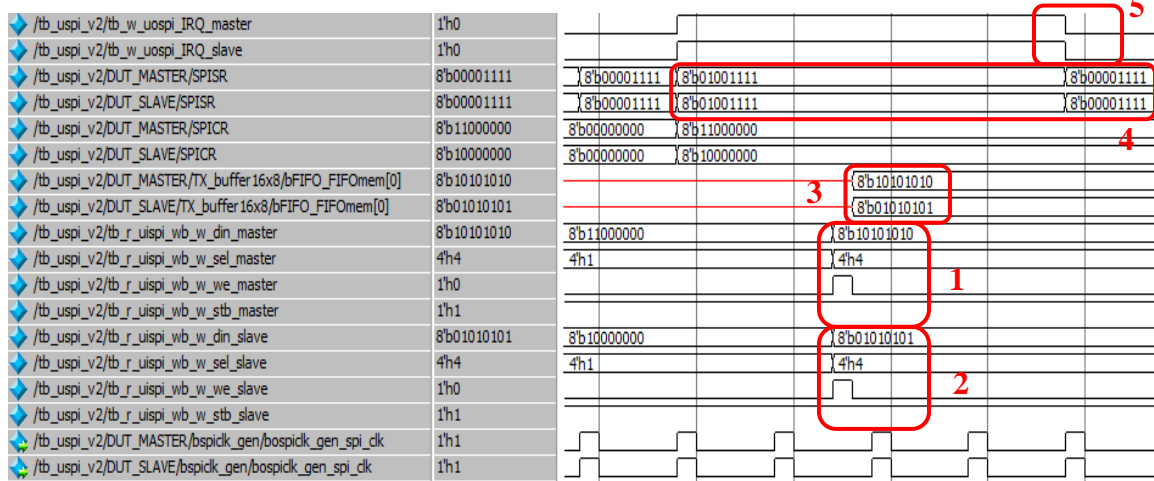


Figure 7.2.5.1: Stimulation result for test case #5 using ModelSim simulator.

1. Write operation on the TX_buffer16x8 of the DUT_MASTER is activated and an input data of 8'b1010_1010 (0xaa) is sent to it.
2. Write operation on the SPITDR of the DUT_SLAVE is activated and an input data of 8'b0101_0101 (0x55) is sent to it.

3. The data value of 8'b1010_1010 (0xaa) is stored into the TX_buffer16x8.bFIFO_FIFOmem[0] of the DUT_MASTER whereas the data value of 8'b0101_0101 (0x55) is stored into the TX_buffer16x8.bFIFO_FIFOmem[0] of the DUT_SLAVE.
4. Bit 6 (TXEF) of the SPISR in both of the DUT_MASTER and DUT_SLAVE is initially one. However, it changed to zero after the corresponding input data is successfully stored into the TX_buffer16x8 of the DUT_MASTER and DUT_SLAVE respectively. At the end, both SPISRs hold the value of 8'b0000_1111 (0x0f) respectively.
5. No interrupt request is generated from the DUT_MASTER and also the DUT_SLAVE as the TX_buffer16x8 of each device has been filled with one 8-bit data and is not empty now.

7.2.6 Test Case #6: Mode 0 Serial Data Communication

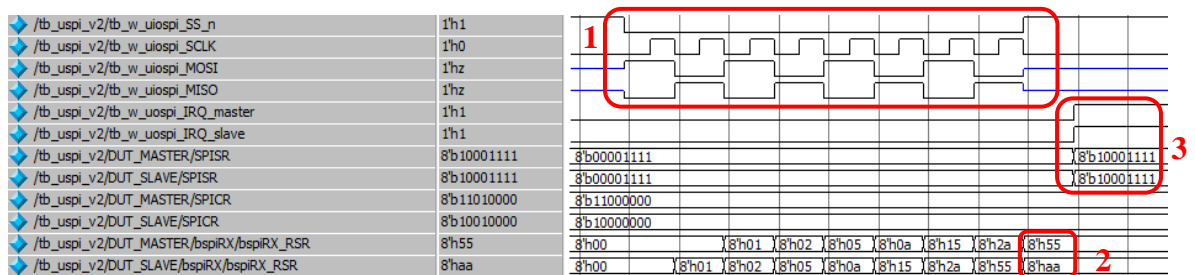


Figure 7.2.6.1: Stimulation result for test case #6 using ModelSim stimulator

1. The SS pin goes low for 8 SCLK clock cycles. Meanwhile, the data value of 8'b1010_1010 (0xaa) can be successfully transmitted by the DUT_MASTER via its MOSI pin. Similarly, the data value of 8'b0101_0101 (0x55) can also be successfully transmitted by the DUT_SLAVE via its MISO pin. Each of these data bit is transmitted serially at one half clock cycle before the rising edge of the SCLK clock.
2. The data on the MOSI and MISO line get exchange after 8 SCLK clock cycles. The data value of 8'b0101_0101 (0x55) is stored into the receiver shift register (RSR) of the DUT_MASTER whereas the data value of 8'b1010_1010 (0xaa) is stored into receiver shift register (RSR) of the DUT_SLAVE.

- The interrupt request from both of the DUT_MASTER and the DUT_SLAVE is asserted respectively after they have received one 8-bit data from each other (See test case 7).

7.2.7 Test Case #7: Receiver Buffer Full Interrupt Support After Receiving A 1-byte Data (RXFM = 0)

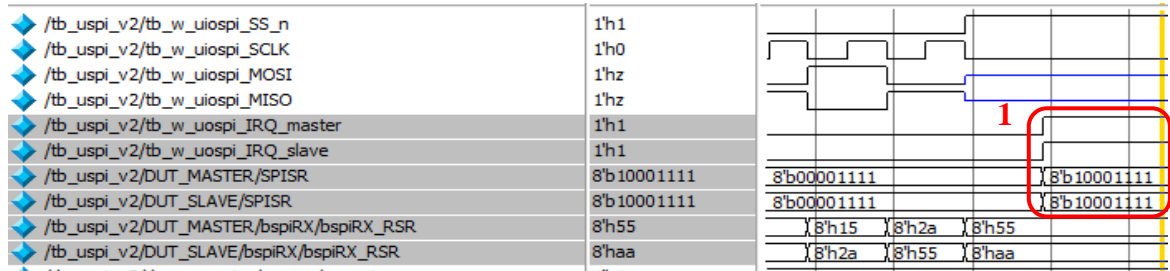


Figure 7.2.7.1: Stimulation result of test case #7 using ModelSim stimulator.

- Since the bit 4 (RXFM) of the SPISR is set to 0 in both of the DUT_MASTER and DUT_SLAVE, it indicates that only 1-byte of data is expected to be read by CPU. Hence, the interrupt request from both of the DUT_MASTER and DUT_SLAVE is asserted when the bit 7 (RXDF) of the SPISR become 1 (indicates that the 1-byte of data has been completely received). At the end, both SPISR's hold the value of 8'b1000_1111 (0x8f) respectively.

7.2.8 Test Case #8: Pop 1-byte of Received Data from the RX_buffer16x8

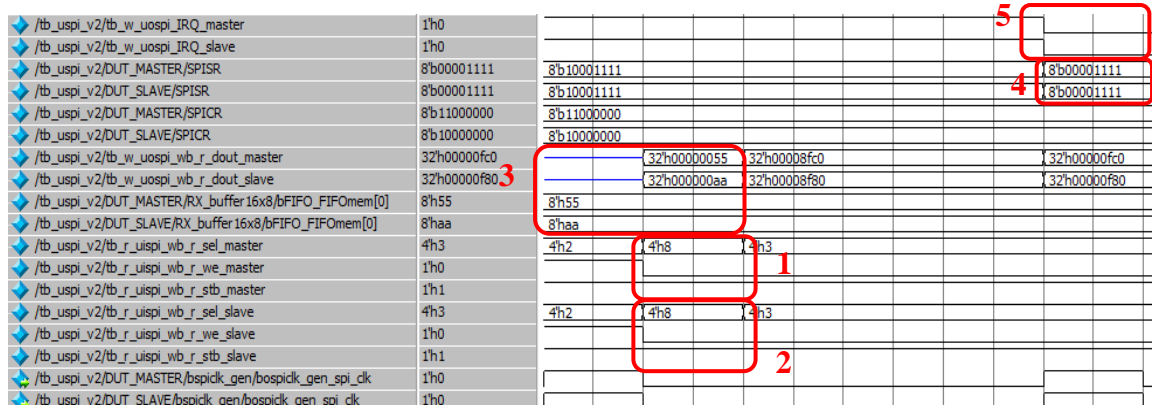


Figure 7.2.8.1: Stimulation result for test case #8 using ModelSim stimulator.

- Read operation on the RX_buffer16x8 of the DUT_MASTER is activated.
- Read operation on the RX_buffer16x8 of the DUT_SLAVE is activated.

3. The data value of 32'h0000_0055 (0x0000_0055) stored in the RX_buffer16x8.FIFOmem[0] can be read by the DUT_MASTER. On the other hand, the DUT_SLAVE can also read the received data whose value is 32'h0000_000aa (0x0000_000aa) from its RX_buffer16x8.FIFOmem[0].
4. The bit 7 (RXDF) of the SPISR for both of the DUT_MASTER and the DUT_SLAVE is set to 0 respectively because the 1-byte of received data has been read from the RX_buffer16x8. At the end, both SPISRs hold the value of 8'b0001_1111 (0x1f) respectively.
5. The de-assertion of bit 7 in SPISR causes the interrupt request in both of the devices to be de-activated.

7.2.9 Test Case #9: Receiver Buffer Full Interrupt Support After Receiving 16x1-byte Data (RXFM = 1)

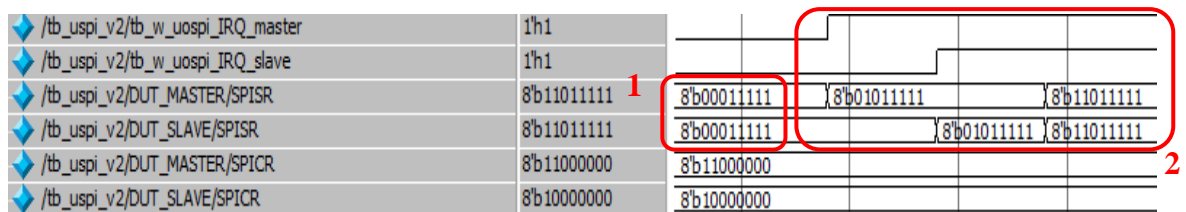


Figure 7.2.9.1: Stimulation result for test case #9 using ModelSim simulator.

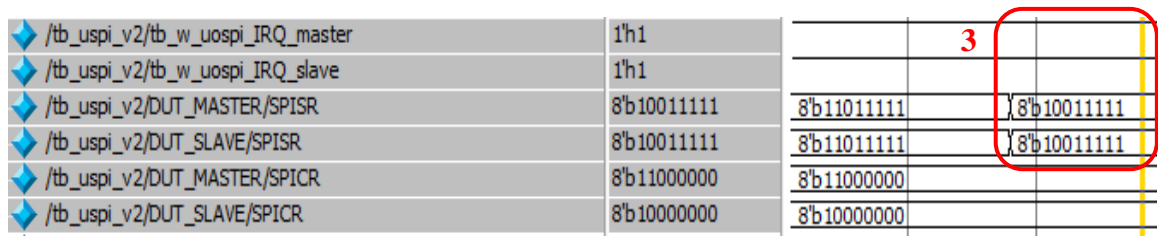


Figure 7.2.9.2: Stimulation result for test case #9 using ModelSim simulator (cont'd).

1. Since the bit 4 (RXFM) of the SPISR is set to 1 in both of the DUT_MASTER and the DUT_SLAVE, it indicates that 16 number of 8-bit data are expected to be read by the CPU. Moreover, both of the transmit buffer empty and received buffer full interrupt have also been enabled.
2. So, there are two reasons why the interrupt requests from both of the devices happen here:
 - a. Their TX_buffer16x8 become empty (TXEF = 1) after sending 16 number of 8-bit data (See test case 4).

- b. Their RX_buffer16x8 become full (RXDF = 1) after receiving 16 number of 8-bit data.
3. A new data is then inserted into their TX_buffer16x8 to disable the transmitter buffer empty interrupt. At the end, the interrupt requests from both of the DUT_MASTER and the DUT_SLAVE remain asserted, indicating that the SPI controller can issue an interrupt to alert the CPU after receiving 16 number of 1-bit data.

7.2.10 Test Case #10: Pop 16 Number of 1-byte Data from the RX_buffer16x8

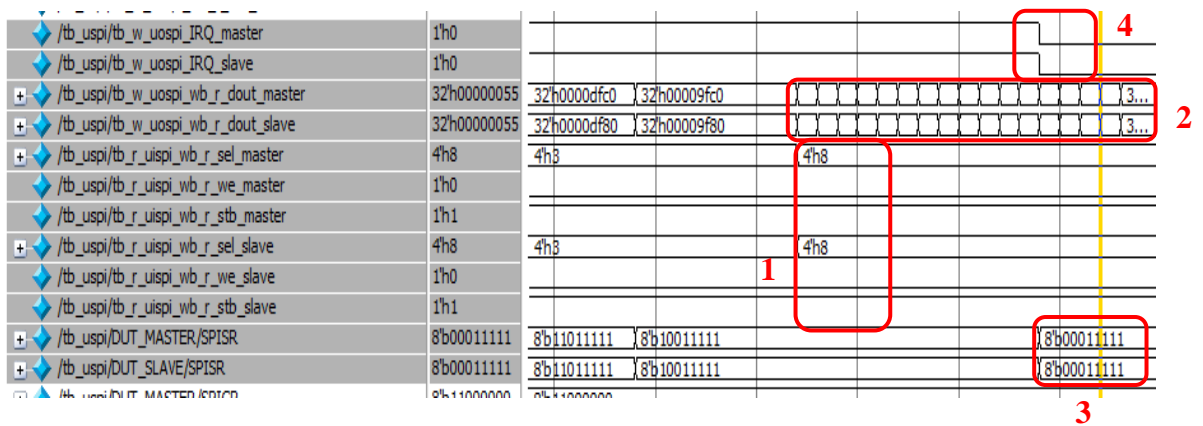


Figure 7.2.10.1: Stimulation result for test case #10 using ModelSim stimulator.

1. Read operation on the RX_buffer16x8 of the DUT_MASTER and the DUT_SLAVE is activated respectively.
2. The 16 number of 1-byte data that were stored in the RX_buffer16x8 of the DUT_MASTER and the DUT_SLAVE can be successfully read by them.
3. The bit 7 (RXDF) of the SPIISR for both of the DUT_MASTER and the DUT_SLAVE is set to 0 respectively because the RX_buffer16x8s in both devices are no longer full with data. At the end, both SPIISRs hold the value of 8'b0001_1111 (0x1f) respectively.
4. The de-assertion of bit 7 in SPIISR causes the interrupt request in both of the devices to be de-activated.

successfully transmitted by the DUT_SLAVE via its MISO pin. Each of these data bit is transmitted serially at one half clock cycle before the falling edge of the SCLK clock.

2. The data on the MOSI and MISO line get exchange after 8 SCLK clock cycles. The data value of 8'b1010_1010 (0xaa) is stored into the receiver shift register (RSR) of the DUT_MASTER whereas the data value of 8'b0101_0101 (0x55) is stored into receiver shift register (RSR) of the DUT_SLAVE.
3. The interrupt request from both of the DUT_MASTER and the DUT_SLAVE is asserted respectively after they have received one 8-bit data from each other (See test case 7).

7.2.13 Test Case #13: Mode 3 Serial Data Communication

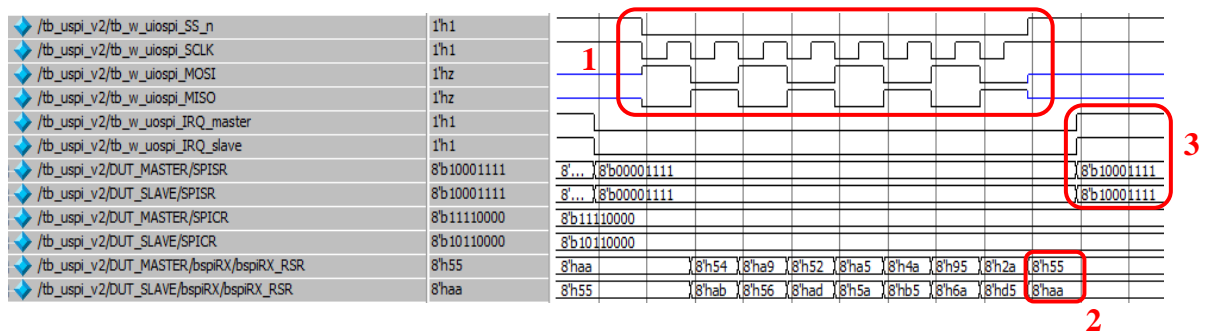


Figure 7.2.13.1: Stimulation result for test case #13 using ModelSim stimulator.

1. The SS pin goes low for 8 SCLK clock cycles. Meanwhile, the data value of 8'b1010_1010 (0xaa) can be successfully transmitted by the DUT_MASTER via its MOSI pin. Similarly, the data value of 8'b0101_0101 (0x55) can also be successfully transmitted by the DUT_SLAVE via its MISO pin. Each of these data bit is transmitted serially at the falling edge of the SCLK clock.
2. The data on the MOSI and MISO line get exchange after 8 SCLK clock cycles. The data value of 8'b0101_0101 (0x55) is stored into the receiver shift register (RSR) of the DUT_MASTER whereas the data value of 8'b1010_1010 (0xaa) is stored into receiver shift register (RSR) of the DUT_SLAVE.
3. The interrupt request from both of the DUT_MASTER and the DUT_SLAVE is asserted respectively after they have received one 8-bit data from each other (See test case 7).

7.2.14 Test Case #14: Selectable Transmission Speed (Baud Rate)

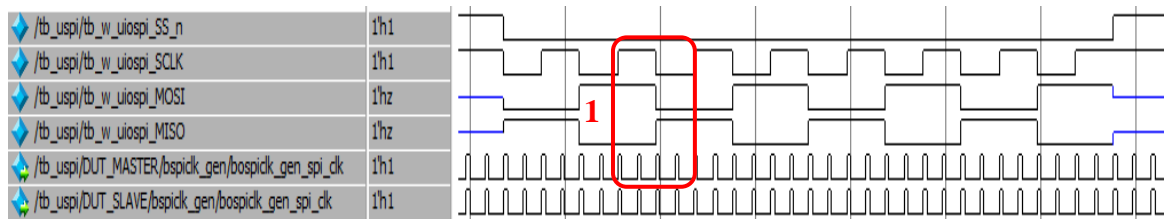


Figure 7.2.14.1: Stimulation result for test case #14 with SCLK clock signal is 4 times slower than the I/O clock of the DUT_MASTER.

1. The SCLK clock signal can be configured to be 4 times slower than the I/O clock of the DUT_MASTER.

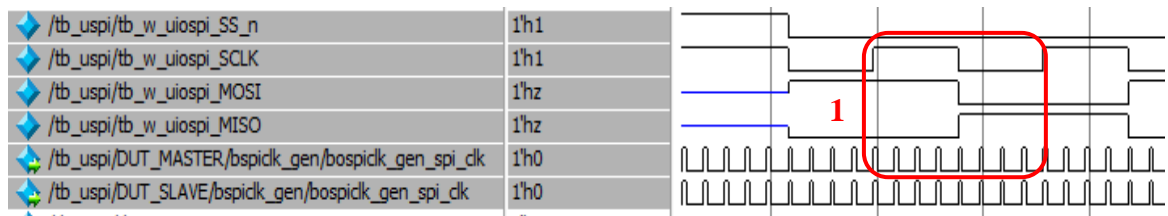


Figure 7.2.14.2: Stimulation result for test case #14 with SCLK clock signal is 8 times slower than the I/O clock of the DUT_MASTER.

1. The SCLK clock signal can be configured to be 8 times slower than the I/O clock of the DUT_MASTER.

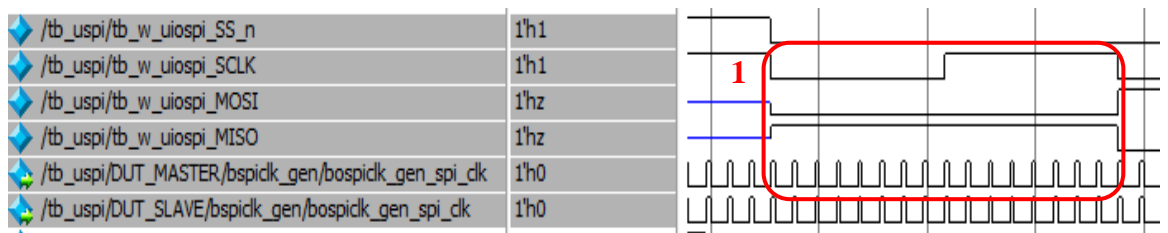


Figure 7.2.14.3: Stimulation result for test case #14 with SCLK clock signal is 16 times slower than the I/O clock of the DUT_MASTER.

1. The SCLK clock signal can be configured to be 16 times slower than the I/O clock of the DUT_MASTER.

7.2.15 Test Case #15: Mode Fault Error Interrupt Support

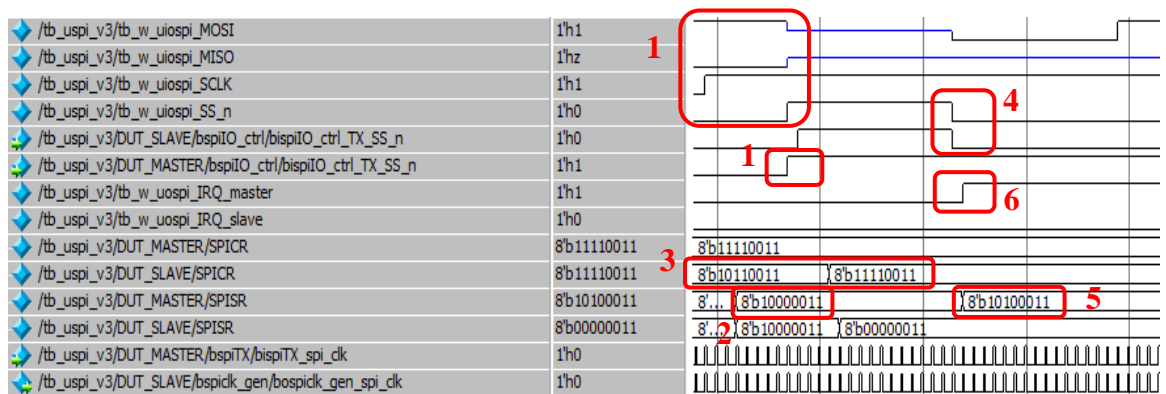


Figure 7.2.15.1: Stimulation result for test case #15 using ModelSim stimulator.

1. The contents of the two shift registers get exchanged once a total of eight pulses of clock signals are generated. When the data transaction is completed, the DUT_MASTER will de-activate the SCLK signal and pull its SS pin to high.
2. The DUT_MASTER's transmit buffer empty and received full interrupt supports are both disabled for this test case, leaving the mode fault error interrupt support being activated only. Meaning, the interrupt request from the DUT_MASTER will be generated if and only if a mode fault error is detected.
3. The DUT_SLAVE is reconfigured to act as a master. Since there are two master devices in the same connection, any attempt to pull the SS pin to low will trigger the mode fault error.
4. The newly configured master device attempts to initiate the communication with the DUT_MASTER by pulling the SS pin to low.
5. Once the mode fault error is successfully detected, the bit 5 (MODF) of the DUT_MASTER's SPISR will be set to 1.
6. An interrupt request corresponding to the mode fault error detection is immediately issued by the DUT_MASTER in order to alert the CPU to take action.

7.3 Test Plan for SPI Controller Unit's Integration Test with RISC32

Once the developed SPI controller has met its functional specifications, it is then ready to be integrated into the existing RISC32. In order to do this, the interface connection between the SPI controller unit and the RISC32 is developed based on the I/O memory mapping technique as this technique keeps the instructions set small. After that, a test plan is created for verifying the behaviors of the integrated SPI controller unit and the RISC32.

As indicated in Figure 7.3.1. it is the RISC32 discussed in Chapter 4 that is being used as the `c_risc_dut` and the `c_risc_client` respectively during the verification process. On the other hand, each of the RISC32s has its own flash memory (a non-volatile memory) where it can get its program codes to execute upon reset. For your information, all the written programs such as the MIPS test programs and the exception handler programs are first stored into the flash memory before the verification test starts.

In this test, the SPI controller unit that has been integrated into the `c_risc_dut` is configured as a master device whereas the SPI controller unit which is integrated in the `c_risc_client` is set as a slave device. Same type of SPI controller unit is used throughout the verification process but with different operation modes (master mode or slave mode) to prove that the designed SPI controller unit can function correctly in both operation modes. In this test phase, a testbench has been specifically developed based on the constructed test plan and Appendix B.2 provides the full information about this testbench.

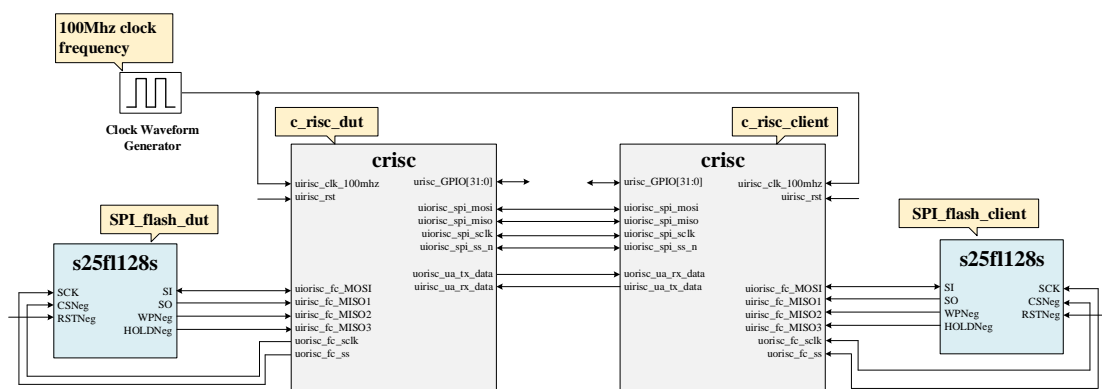


Figure 7.3.1: The connection mechanism of the `c_risc_dut`, `c_risc_client`, `SPI_flash_dut`, and `SPI_flash_client` for SPI controller unit's integration test with RISC32.

Test	Expected Output	Status
<p>Test Case #1: System Reset</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> Reset the whole RISC32 (including SPI controller unit) <p><u>Procedure</u></p> <ol style="list-style-type: none"> Reset both devices. 	<ul style="list-style-type: none"> • uiospi_MOSI of c_risc_dut = 1'b0 • uiospi_MISO of c_risc_dut = 1'b1 • uiospi_SCLK of c_risc_dut = 1'b0 • uiospi_SS_n of c_risc_dut = 1'b1 • uospi_IRQ of c_risc_dut = 1'b0 • uospi_wb_r_dout of c_risc_dut = 32'hz • SPICR of c_risc_dut = 8'h00 • SPISR of c_risc_dut = 8'h00 • uodp_if_pseudo_pc of c_risc_dut = 8'hbfc0_0000 <ul style="list-style-type: none"> • uiospi_MOSI of c_risc_client = 1'b0 • uiospi_MISO of c_risc_client = 1'b1 • uiospi_SCLK of c_risc_client = 1'b0 • uiospi_SS_n of c_risc_client = 1'b1 • uospi_IRQ of c_risc_client = 1'b0 • uospi_wb_r_dout of c_risc_client = 32'hz • SPICR of c_risc_client = 8'h00 • SPISR of c_risc_client = 8'h00 • uodp_if_pseudo_pc of c_risc_client = 8'hbfc0_0000 	Pass
<p>Test Case #2: Transmitter buffer empty interrupt support</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> Transmitter buffer empty interrupt support in both master mode and slave mode <p><u>Procedure for c_risc_dut</u></p> <ol style="list-style-type: none"> Store a value of 0x08 to PICMASK. Store a value of 0x05 to both SPISR Store a value of 0xc0 to SPICR. <p><u>Procedure for c_risc_client</u></p> <ol style="list-style-type: none"> Store a value of 0x08 to PICMASK. Store a value of 0x05 to both SPISR Store a value of 0x80 to SPICR. 	<ul style="list-style-type: none"> • PICMASK of c_risc_dut = 8'h08 • PICSTAT of c_risc_dut = 8'h03 • SPISR of c_risc_dut = 8'h45 • SPICR of c_risc_dut = 8'hc0 • uospi_IRQ of c_risc_dut = 1'b1 • uodp_if_pc of c_risc_dut = 8'h8001_b400 (Jump to the exception handler address in the next clock cycle) • bcpc_cause[6:2] of c_risc_dut = 6'b00_0000 • bcpc_epc of c_risc_dut = bicpc0_if_pc of c_risc_dut <ul style="list-style-type: none"> • PICMASK of c_risc_client = 8'h08 • PICSTAT of c_risc_client = 8'h03 • SPISR of c_risc_client = 8'h45 • SPICR of c_risc_client = 8'h80 • uospi_IRQ of c_risc_client = 1'b1 • uodp_if_pc of c_risc_client = 8'h8001_b400 (Jump to the exception handler address in the next clock cycle) • bcpc_cause[6:2] of c_risc_client = 6'b00_0000 	Pass

	<ul style="list-style-type: none"> • bcpo_epc of c_risc_client = bicp0_if_pc of c_risc_client 	
<p>Test Case #3: Mode 0 Serial Data Communication</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Able to transmit and receive data simultaneously in both master mode and slave mode when mode 0 is used. <p><u>Procedure for c_risc_dut</u></p> <ol style="list-style-type: none"> 1. Store 0xaa to SPITDR. 2. Store 0x0b to SPISR 3. Store 0xc0 to SPICR <p><u>Procedure for c_risc_client</u></p> <ol style="list-style-type: none"> 1. Store 0x55 to SPITDR. 2. Store 0x0b to SPISR 3. Store 0x81 to SPICR 	<ul style="list-style-type: none"> • uiospi_SS_n = 1'b0 • 8 baud clock cycles should appear on the uiospi_SCLK. • There should be only one bit of data on the uiospi_MOSI for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b1010_1010. • There should be only one bit of data on the uiospi_MISO for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b0101_0101. • Each of these data bit is transmitted serially at one half clock cycle before the rising edge of the uiospi_SCLK clock. • uospi_IRQ of c_risc_dut = 1'b1 • uospi_IRQ of c_risc_client = 1'b1 	Pass
<p>Test Case #4: Receiver buffer full interrupt support</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Receiver buffer full interrupt support in both master mode and slave mode. 	<ul style="list-style-type: none"> • PICMASK of c_risc_dut = 8'h08 • PICSTAT of c_risc_dut = 8'h03 • SPISR of c_risc_dut = 8'hcb • SPICR of c_risc_dut = 8'hc0 • uospi_IRQ of c_risc_dut = 1'b1 • uodp_if_pc of c_risc_dut = 8'h8001_b400 (Jump to the exception handler address in the next clock cycle) • bcpo_cause[6:2] of c_risc_dut = 6'b00_0000 • bcpo_epc of c_risc_dut = bicp0_if_pc of c_risc_dut • PICMASK of c_risc_client = 8'h08 • PICSTAT of c_risc_client = 8'h03 • SPISR of c_risc_client = 8'hcb • SPICR of c_risc_client = 8'h81 • uospi_IRQ of c_risc_client = 1'b1 • uodp_if_pc of c_risc_client = 8'h8001_b400 (Jump to the exception handler address in the next clock cycle) • bcpo_cause[6:2] of c_risc_client = 6'b00_0000 • bcpo_epc of c_risc_client = bicp0_if_pc of c_risc_client 	Pass

<p>Test Case #5: Mode 1 Serial Data Communication</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Able to transmit and receive data simultaneously in both master mode and slave mode when mode 1 is used. <p><u>Procedure for c_risc_dut</u></p> <ol style="list-style-type: none"> 1. Store 0x55 to SPITDR. 2. Store 0x0f to SPISR 3. Store 0xd1 to SPICR <p><u>Procedure for c_risc_client</u></p> <ol style="list-style-type: none"> 1. Store 0xaa to SPITDR. 2. Store 0x0f to SPISR 3. Store 0x91 to SPICR 	<ul style="list-style-type: none"> • uiospi_SS_n = 1'b0 • 8 baud clock cycles should appear on the uiospi_SCLK. • There should be only one bit of data on the uiospi_MOSI for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b0101_0101. • There should be only one bit of data on the uiospi_MISO for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b1010_1010. • Each of these data bit is transmitted serially at the rising edge of the uiospi_SCLK clock. • uospi_IRQ of c_risc_dut = 1'b1 • uospi_IRQ of c_risc_client = 1'b1 	Pass
<p>Test Case #6: Mode 2 Serial Data Communication</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Able to transmit and receive data simultaneously in both master mode and slave mode when mode 2 is used. <p><u>Procedure for c_risc_dut</u></p> <ol style="list-style-type: none"> 1. Store 0xaa to SPITDR. 2. Store 0x0f to SPISR 3. Store 0xe2 to SPICR <p><u>Procedure for c_risc_client</u></p> <ol style="list-style-type: none"> 1. Store 0xaa to SPITDR. 2. Store 0x0f to SPISR 3. Store 0xa1 to SPICR 	<ul style="list-style-type: none"> • uiospi_SS_n = 1'b0 • 8 baud clock cycles should appear on the uiospi_SCLK. • There should be only one bit of data on the uiospi_MOSI for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b1010_1010. • There should be only one bit of data on the uiospi_MISO for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b0101_0101. • Each of these data bit is transmitted serially at one half clock cycle before the falling edge of the uiospi_SCLK clock. • uospi_IRQ of c_risc_dut = 1'b1 • uospi_IRQ of c_risc_client = 1'b1 	Pass

<p>Test Case #7: Mode 3 Serial Data Communication</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Able to transmit and receive data simultaneously in both master mode and slave mode when mode 3 is used. <p><u>Procedure for c_risc_dut</u></p> <ol style="list-style-type: none"> 1. Store 0x55 to SPITDR. 2. Store 0x0b to SPISR 3. Store 0xf3 to SPICR <p><u>Procedure for c_risc_client</u></p> <ol style="list-style-type: none"> 1. Store 0xaa to SPITDR. 2. Store 0x0f to SPISR 3. Store 0xb1 to SPICR 	<ul style="list-style-type: none"> • uiospi_SS_n = 1'b0 • 8 baud clock cycles should appear on the uiospi_SCLK. • There should be only one bit of data on the uiospi_MOSI for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b0101_0101. • There should be only one bit of data on the uiospi_MISO for every baud clock cycles. The expected sequence of data to be transmitted on this pin is 8'b1010_1010. • Each of these data bit is transmitted serially at the falling edge of the uiospi_SCLK clock. • uospi_IRQ of c_risc_dut = 1'b1 • uospi_IRQ of c_risc_client = 1'b1 	Pass
<p>Test Case #8: Mode fault error interrupt support</p> <p><u>Function to be tested</u></p> <ul style="list-style-type: none"> • Mode fault error interrupt in master mode <p><u>Procedure for c_risc_client</u></p> <ol style="list-style-type: none"> 1. Store 0x55 to SPITDR. 2. Store 0xf1 to SPICR 	<ul style="list-style-type: none"> • PICMASK of c_risc_dut = 8'h08 • PICSTAT of c_risc_dut = 8'h03 • SPISR of c_risc_dut = 8'h6b • SPICR of c_risc_dut = 8'hf3 • uospi_IRQ of c_risc_dut = 1'b1 • uodp_if_pc of c_risc_dut = 8'h8001_b400 (Jump to the exception handler address in the next clock cycle) • bcpo_cause[6:2] of c_risc_dut = 6'b00_0000 • bcpo_epc of c_risc_dut = bicp0_if_pc of c_risc_dut 	Pass

Table 7.3.1: Test plan for the SPI controller unit's integration test with RISC32 pipeline processor.

7.4 MIPS Test Program for c_risc_dut in Integration Test

Label	Instruction	Comment
setting:	lui \$s0, 0xbfff	
	ori \$s0, \$s0, 0xfe00	\$s0 = 0xbfff_fe00 (I/O peripheral address)
	addi \$t0, \$zero, 0x08	\$t0[7:0] = 8'b0000_1000
	sb \$t0, 34(\$s0)	PICMASK = 0x08 => enable SPI controller's interrupt. Interrupt method instead of the polling method will be used throughout this test program for testing the ISR execution of the SPI controller unit.
Test Case #2		
TXEF_int:	addi \$t0, \$zero, 0x05	\$t0[7:0] = 8'b0000_0101
	sb \$t0, 37(\$s0)	SPISR = 0x05 => enable transmitter buffer empty interrupt, disable receiver full buffer interrupt, halt TX FSM when TX_buffer16x8 is empty
	addi \$t0, \$zero, 0xc0	\$t0[7:0] = 8'b1100_0000
	sb\$t0, 36(\$s0)	SPICR = 0xc0 => activate the SPI controller, set as master, use mode 0, use baud rate = (SPI i/o clock)/2
	jal program_code	Whenever the transmitter buffer is detected to be empty (TXEF bit = 1), the SPI controller unit will interrupt the CPU for service even though the CPU is busy executing its normal task.
Test Case #3 and #4		
mode_0:	addi \$t0, \$zero, 0xaa	\$t0[7:0] = 8'b1010_1010
	sb \$t0, 38(\$s0)	TX_buffer16x8.bFIFO_FIFOmem[0] = 0xaa
	addi \$t0, \$zero, 0x0b	\$t0[7:0] = 8'b0000_1011
	sb \$t0, 37(\$s0)	SPISR = 0x0b => disable transmitter buffer empty interrupt, enable receiver buffer full interrupt, halt TX FSM when TX_buffer16x8 is empty or when one byte of data is received
	addi \$t0, \$zero, 0xc0	\$t0[7:0] = 8'b1100_0000
	sb \$t0, 36(\$s0)	SPICR = 0xc0 => activate the SPI controller, set as master, use mode 0, use baud rate = (SPI i/o clock)/2
	jal program_code	Whenever the receiver buffer is detected to be full (RXDF bit = 1), the SPI controller unit will interrupt the CPU for service even though the CPU is busy executing its normal task.
Test Case #5		

mode_1:	addi \$t0, \$zero, 0x55	\$t0[7:0] = 8'b0101_0101
	sb \$t0, 38(\$s0)	TX_buffer16x8.bFIFO_FIFOmem[1] = 0x55
	addi \$t0, \$zer0, 0x0f	\$t0[7:0] = 8'b0000_1111
	sb \$t0, 37(\$s0)	SPISR = 0x0f => enable transmitter buffer empty interrupt, enable receiver buffer full interrupt, halt TX FSM when TX_buffer16x8 is empty or when one byte of data is received
	addi \$t0, \$zero, 0xd1	\$t0[7:0] = 8'b1101_0001
	sb \$t0, 36(\$s0)	SPICR = 0xd1 => activate the SPI controller, set as master, use mode 1, use baud rate = (SPI i/o clock)/4
	jal program_code	Whenever the transmitter buffer is empty (TXEF = 1) or the receiver buffer is full (RXDF = 1), the SPI controller unit will interrupt the CPU for service even though the CPU is busy executing its normal task.
Test Case #6		
mode_2	addi \$t0, \$zero, 0xaa	\$t0[7:0] = 8'b1010_1010
	sb \$t0, 38(\$s0)	TX_buffer16x8.bFIFO_FIFOmem[3] = 0xaa
	addi \$t0, \$zer0, 0x0f	\$t0[7:0] = 8'b0000_1111
	sb \$t0, 37(\$s0)	SPISR = 0x0f => enable transmitter buffer empty interrupt, enable receiver buffer full interrupt, halt TX FSM when TX_buffer16x8 is empty or when one byte of data is received
	addi \$t0, \$zero, 0xe2	\$t0[7:0] = 8'b1110_0010
	sb \$t0, 36(\$s0)	SPICR = 0xe2 => activate the SPI controller, set as master, use mode 2, use baud rate = (SPI i/o clock)/8
	jal program_code	Whenever the transmitter buffer is empty (TXEF = 1) or the receiver buffer is full (RXDF = 1), the SPI controller unit will interrupt the CPU for service even though the CPU is busy executing its normal task.
Test Case #7		
mode_3:	addi \$t0, \$zero, 0x55	\$t0[7:0] = 8'b0101_0101
	sb \$t0, 38(\$s0)	TX_buffer16x8.bFIFO_FIFOmem[2] = 0x55
	addi \$t0, \$zer0, 0x0b	\$t0[7:0] = 8'b0000_1011
	sb \$t0, 37(\$s0)	SPISR = 0x0f => disable transmitter buffer empty interrupt, enable receiver buffer full interrupt, halt TX FSM when TX_buffer16x8 is empty or when one byte of data is received

	addi \$t0, \$zero, 0xf3	\$t0[7:0] = 8'b1111_0011
	sb \$t0, 36(\$s0)	SPICR = 0xd1 => activate the SPI controller, set as master, use mode 3, use baud rate = (SPI i/o clock)/16
	jal program_code	Whenever the receiver buffer is full (RXDF = 1), the SPI controller unit will interrupt the CPU for service even though the CPU is busy executing its normal task.
Test Case #8		
MODF_int:	jal program_code	Only one master can exist in the connection at a time to initiate all the communications with slaves. Since there are two master devices in the same connection, any attempt from a master device to pull the SS pin to low will trigger the mode fault (MODF) error in other master device. Whenever the MODF error is detected, the SPI controller unit will interrupt the CPU for service even though the CPU is busy executing its normal task.
	j exit	
Function		
program_code:	addi \$t1, \$zero, 300	\$t1 = 300
loop_1	addi \$t1, \$t1, -1	\$t1 = \$t1 -1
	slt \$t2, \$t1, \$zero	\$t2 = 1 if \$t1 < 0, \$t2 = 0 if \$t1 >= 0
	beq \$t2, \$zero, loop_1	branch to loop_1 if \$t2=0
	jr \$ra	This program_code function represents a simple user program which is just a loop to decrease the value from 300 to 0. It will be executed whenever it is called in order to test if the integrated SPI controller is able to interrupt the CPU for service while the CPU is executing its normal task.
exit:	nop	

Table 7.4.1: MIPS test program for c_risc_dut in integration test.

7.5 MIPS Test Program for c_risc_client in Integration Test

Label	Instruction	Comment
setting:	lui \$s0, 0xbfff	
	ori \$s0, \$s0, 0xfe00	\$s0 = 0xbfff_fe00 (I/O peripheral address)
	addi \$t0, \$zero, 0x08	\$t0[7:0] = 8'b0000_1000
	sb \$t0, 34(\$s0)	PICMASK = 0x08 => enable SPI controller's interrupt. Interrupt method instead of the polling method will be used throughout this test program for testing the ISR execution of the SPI controller unit.
Test case #2		
TXEF_int:	addi \$t0, \$zer0, 0x05	\$t0[7:0] = 8'b0000_0101
	sb \$t0, 37(\$s0)	SPISR = 0x05 => enable transmitter buffer empty interrupt, disable receiver full buffer interrupt, halt TX FSM when TX_buffer16x8 is empty
	addi \$t0, \$zero, 0x80	\$t0[7:0] = 8'b1000_0000
	sb\$t0, 36(\$s0)	SPICR = 0xc0 => activate the SPI controller, set as slave, use mode 0, use baud rate = (SPI i/o clock)/2
	jal program_code	Whenever the transmitter buffer is detected to be empty (TXEF bit = 1), the SPI controller unit will interrupt the CPU for service even though the CPU is busy executing its normal task.
Test case #3 and #4		
mode_0:	addi \$t0, \$zero, 0x55	\$t0[7:0] = 8'b0101_0101
	sb \$t0, 38(\$s0)	TX_buffer16x8.bFIFO_FIFOmem[0] = 0x55
	addi \$t0, \$zero, 0x0b	\$t0[7:0] = 8'b0000_1011
	sb \$t0, 37(\$s0)	SPISR = 0x0b => disable transmitter buffer empty interrupt, enable receiver buffer full interrupt, halt TX FSM when TX_buffer16x8 is empty or when one byte of data is received
	addi \$t0, \$zero, 0x81	\$t0[7:0] = 8'b1000_0001
	sb \$t0, 36(\$s0)	SPICR = 0xc0 => activate the SPI controller, set as slave, use mode 0, use baud rate = (SPI i/o clock)/4
	jal check_RXDF	Whenever the receiver buffer is detected to be empty (RXDF bit = 1), the SPI controller unit will interrupt the CPU for service.
Test case #5		
mode_1:	addi \$t0, \$zero, 0xaa	\$t0[7:0] = 8'b1010_1010
	sb \$t0, 38(\$s0)	TX_buffer16x8.bFIFO_FIFOmem[1] = 0xaa
	addi \$t0, \$zer0, 0x0f	\$t0[7:0] = 8'b0000_1111
	sb \$t0, 37(\$s0)	SPISR = 0x0f => enable transmitter buffer empty interrupt, enable receiver buffer full interrupt, halt TX FSM when TX_buffer16x8 is empty or when one byte of data is received
	addi \$t0, \$zero, 0x91	\$t0[7:0] = 8'b1001_0001

	sb \$t0, 36(\$s0)	SPICR = 0xd1 => activate the SPI controller, set as slave, use mode 1, use baud rate = (SPI i/o clock)/4
	jal check_RXDF	Whenever the transmitter buffer is empty (TXEF = 1) or the receiver buffer is full (RXDF = 1), the SPI controller unit will interrupt the CPU for service even though the CPU is busy executing its normal task.
Test case #6		
mode_2	addi \$t0, \$zero, 0x55	\$t0[7:0] = 8'b0101_0101
	sb \$t0, 38(\$s0)	TX_buffer16x8.bFIFO_FIFOmem[3] = 0x55
	addi \$t0, \$zer0, 0x0f	\$t0[7:0] = 8'b0000_1111
	sb \$t0, 37(\$s0)	SPISR = 0x0f => enable transmitter buffer empty interrupt, enable receiver buffer full interrupt, halt TX FSM when TX_buffer16x8 is empty or when one byte of data is received
	addi \$t0, \$zero, 0xa1	\$t0[7:0] = 8'b1010_0001
	sb \$t0, 36(\$s0)	SPICR = 0xe2 => activate the SPI controller, set as slave, use mode 2, use baud rate = (SPI i/o clock)/4
	jal check_RXDF	Whenever the transmitter buffer is empty (TXEF = 1) or the receiver buffer is full (RXDF = 1), the SPI controller unit will interrupt the CPU for service even though the CPU is busy executing its normal task.
Test case #7		
mode_3:	addi \$t0, \$zero, 0xaa	\$t0[7:0] = 8'b1010_1010
	sb \$t0, 38(\$s0)	TX_buffer16x8.bFIFO_FIFOmem[2] = 0x55
	addi \$t0, \$zer0, 0x0f	\$t0[7:0] = 8'b0000_1111
	sb \$t0, 37(\$s0)	SPISR = 0x0f => enable transmitter buffer empty interrupt, enable receiver buffer full interrupt, halt TX FSM when TX_buffer16x8 is empty or when one byte of data is received
	addi \$t0, \$zero, 0xb1	\$t0[7:0] = 8'b1011_0001
	sb \$t0, 36(\$s0)	SPICR = 0xd1 => activate the SPI controller, set as slave, use mode 3, use baud rate = (SPI i/o clock)/4
	jal check_RXDF	Whenever the transmitter buffer is empty (TXEF = 1) or the receiver buffer is full (RXDF = 1), the SPI controller unit will interrupt the CPU for service even though the CPU is busy executing its normal task.
Test case #8		
MODF_int:	addi \$t0, \$zero, 0x55	\$t0[7:0] = 8'b0101_0101
	sb \$t0, 38(\$s0)	TX_buffer16x8.bFIFO_FIFOmem[6] = 0x55
	addi \$t0, \$zero, 0xf1	\$t0[7:0] = 8'b1111_0001
	sb \$t0, 36(\$s0)	SPICR = 0xf1 => activate the SPI controller, set as master, use mode 3, use baud rate = (SPI i/o clock)/4

	jal_program_code	In test case 8, this SPI controller unit is reconfigured to act as a master device instead of the slave device. It will then initiate a communication to another master device in the c_risc_dut by pulling the shared SS line to low.
	j exit	
Function #1		
program_code:	addi \$t1, \$zero, 100	\$t1 = 100
loop_1	addi \$t1, \$t1, -1	\$t1 = \$t1 -1
	slt \$t2, \$t1, \$zero	\$t2 = 1 if \$t1 < 0, \$t2 = 0 if \$t1 >= 0
	beq \$t2, \$zero, loop_1	branch to loop_1 if \$t2=0
	jr \$ra	This program_code function represents a simple user program which is just a loop to decrease the value from 100 to 0. A smaller value is used here to make sure the slave is ready before the master initiate a communication with it in test case #3 and #4. It will be executed whenever it is called in order to test if the integrated SPI controller is able to interrupt the CPU for service while the CPU is executing its normal task.
Function #2		
check_RXDF:	lbu \$t1, 37(\$s0)	\$t1 = SPISR
	srl \$t1, \$t1, 7	\$t1 = SPISR[7] = RXDF
	beq \$t1, \$zero, check_RXDF	branch to check_RXDF if \$t1 = 0
exit:	nop	Before a new transfer mode can be used, need to make sure that the SPI controller has used the current transfer mode to perform the data transmission and sampling. So, this check_RXDF function will do this job by constantly checking the RXDF bit in the SPISR. When a data transfer occurs and a data byte has been successfully received, the RXDF will be asserted. After successfully testing the current mode for data transfer, the program can then continue to test a new transfer mode.

Table 7.5.1: MIPS test program for c_risc_client in integration test

7.6 Stimulation Results of the SPI Controller Unit's Integration Test with RISC32

7.6.1 Test Case #1: System Reset

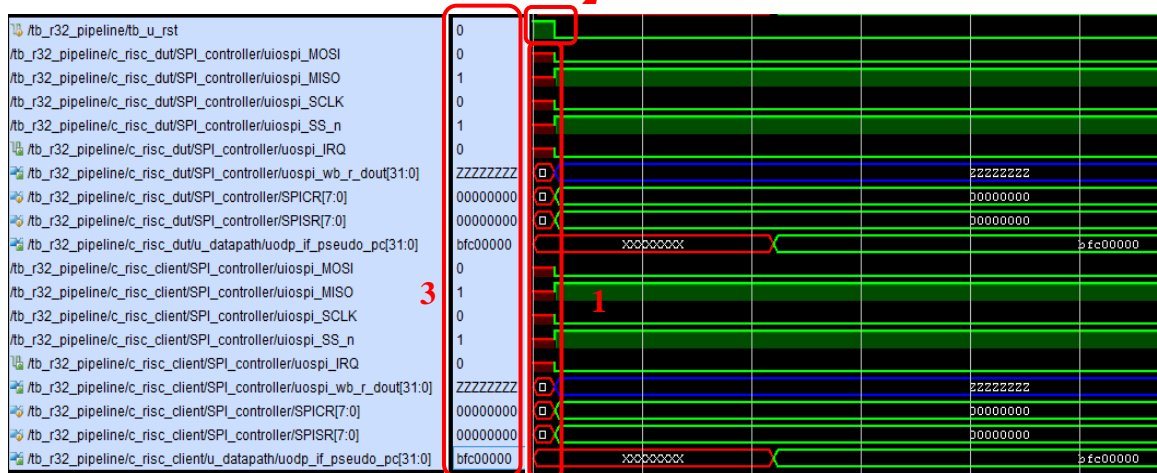


Figure 7.6.1.1: Stimulation result for test case #1 using Vivado stimulator.

1. Before the reset signal is asserted, most of the signals have an unknown value.
2. The reset signal is asserted in order to initialize all the system registers, FSM state, read/write pointer, program counter (PC) and etc.
3. All of the output signals and registers are set to their respective default values.

The program counter of both RISC32s is set to point at the memory address of 0xbfc0_000 (which is the beginning address of the boot ROM) for executing the bootloader program.

7.6.2 Test Case #2: Transmitter Buffer Empty Interrupt Support

Stimulation Result of c_risc_dut:

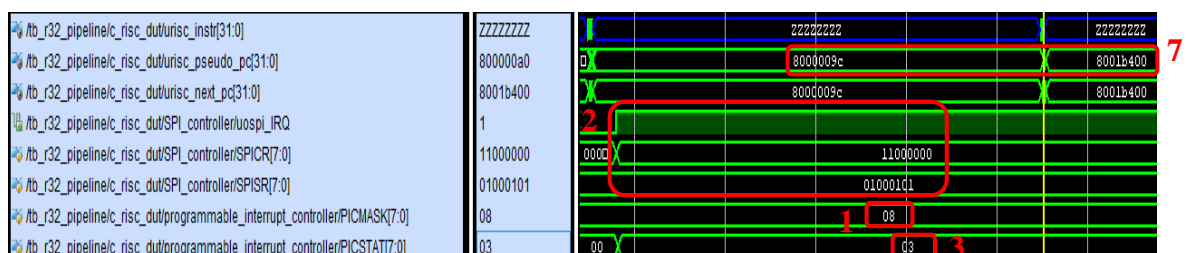


Figure 7.6.2.1: Stimulation result of c_risc_dut for test case #2 using Vivado stimulator.

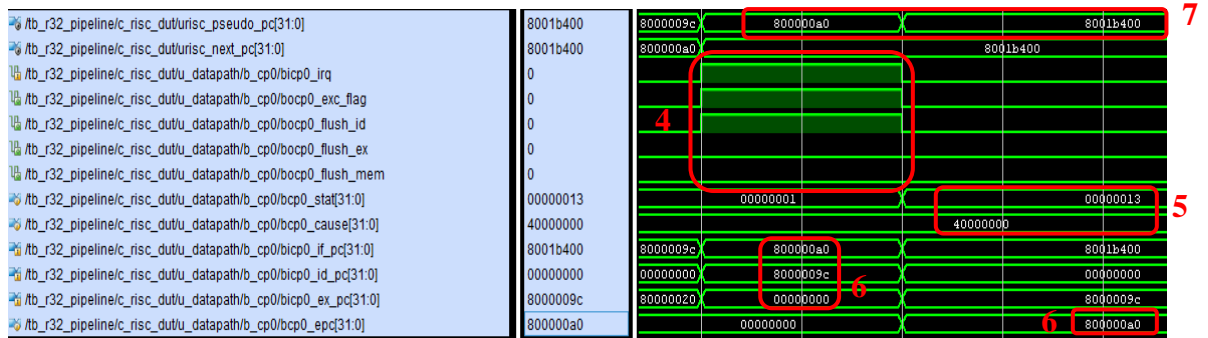


Figure 7.6.2.2: Stimulation result of c_risc_dut for test case #2 using Vivado stimulator (cont'd).

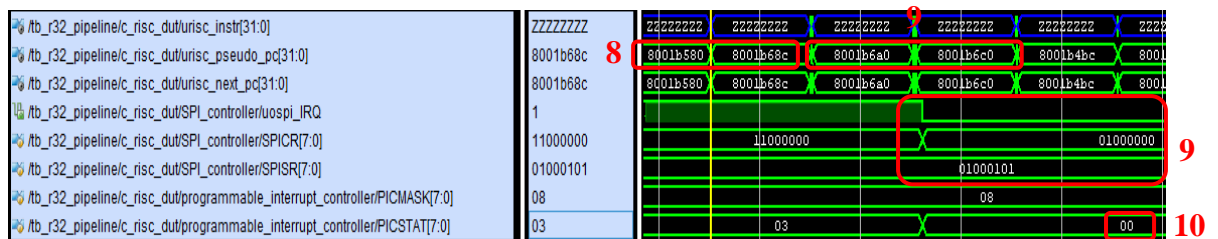


Figure 7.6.2.3: Stimulation result of c_risc_dut for test case #2 using Vivado stimulator (cont'd).

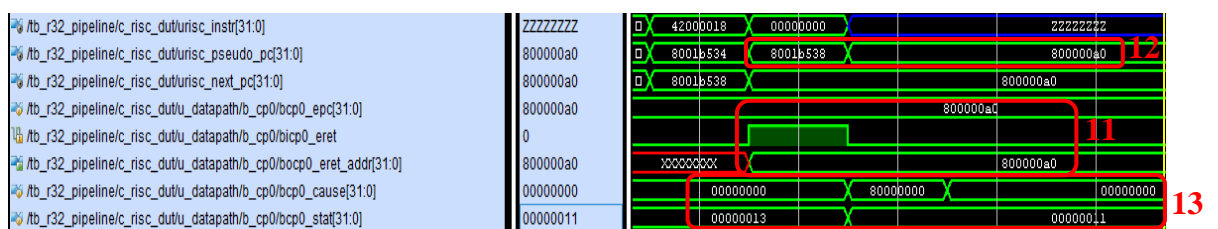


Figure 7.6.2.4: Stimulation result of c_risc_dut for test case #2 using Vivado stimulator (cont'd).

1. With the value of 0x08 being set in the PICMASK, the SPI I/O interrupt in c_risc_dut is enabled.
2. Initially, the TX_buffer16x8 of the SPI controller is empty. Upon activation, the on-board SPI controller is configured to enable the transmitter buffer empty interrupt only (by setting SPISR[2] = 1). Thus, when the bit 6 (TXEF) of the SPISR goes high, the SPI controller will initiate an interrupt request (IRQ) to notify the CPU for service. As a result, the uospi_IRQ status flag goes high.
3. Whenever an I/O interrupt is detected, the PICSTAT of the c_risc_dut will be updated with a value corresponding to the IRQ source. The value for SPI's IRQ is 0x03. The exception handler will then compare this PICSTAT value to

identify which I/O to serve and which Interrupt Service Routine (ISR) to jump to.

4. When the SPI's IRQ occurs, the CPO hardware will raise the `bocp0_exc_flag` and set to flush the IF/ID pipeline register.
5. The `bcp0_stat[1]` (which is the CP0's status register) is set to 1 in order to disable further exception from occurring whereas the `bcp0_cause[6:2]` (which is the CP0's cause register) remain unchanged because the exception code for I/O interrupt is 0.
6. The CP0 will also load the `bicp0_if_pc` (which is the IF stage's PC) into the `bcp0_epc` (which is the CP0's EPC register) for return purpose after executing the exception handler program.
7. The `c_risc_dut` will then jump to the exception handler address of `0x8001_b400` in the next clock cycle and start servicing the SPI controller request.
8. After decoding the PICSTAT value to figure out which ISR to jump to, the `c_risc_dut` will then branch to the ISR of the respective interrupt source and execute the ISR. For SPI, the starting address of its ISR is `0x8001_b68c`.
9. For handling the transmitter buffer empty interrupt, the SPI controller will need to be disabled. At the end, the SPICR holds the value of `8'b0100_0000 (0x40)`. After the `c_risc_dut` successfully disables the SPI controller, the `uospi_IRQ` from the SPI controller is also removed and no more interrupt request is triggered from the SPI controller.
10. When the interrupt request is successfully handled, the PICSTAT value that stores the I/O interrupt source information will be changed from `0x03` to `0x00` value.
11. After completing the SPI's ISR, the `c_risc_dut` will branch to the address of `0x8001_b4bc` and pop all the saved contents from stack before returning to the user program. When the last instruction which is the `eret` instruction is being executed (`bicp0_eret = 1`), it will load the saved return address value from `bcp0_epc` (which is the CP0's EPC register) to the `bocp0_eret_addr` for jumping back to the user program in the next clock cycle.
12. Upon completing the last instruction in the exception handler program, the `c_risc_dut` returns to the place where it was interrupted and starts to execute from that address.

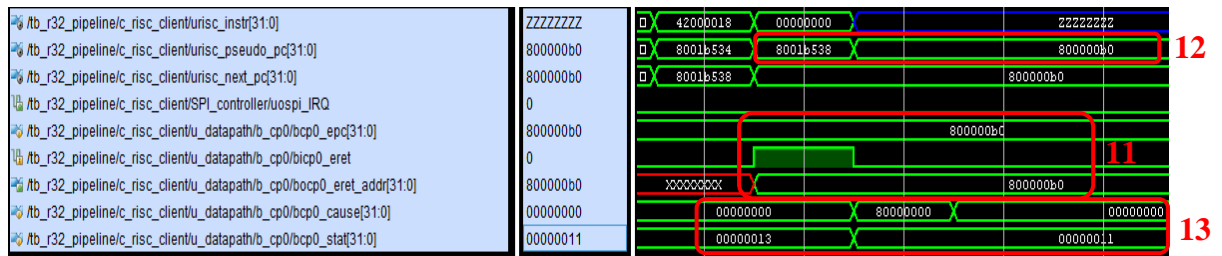


Figure 7.6.2.8: Stimulation result of `c_risc_client` for test case #2 using Vivado stimulator (cont'd).

1. With the value of 0x08 being set in the PICMASK, the SPI I/O interrupt in `c_risc_client` is enabled.
2. Initially, the TX_buffer16x8 of the SPI controller is empty. Upon activation, the on-board SPI controller is configured to enable the transmitter buffer empty interrupt (by setting `SPISR[2] = 1`) only. Thus, when the bit 6 (TXEF) of the SPISR goes high, the SPI controller will initiate an interrupt request (IRQ) to notify the CPU for service. As a result, the `uospi_IRQ` status flag goes high.
3. Whenever an I/O interrupt is detected, the PICSTAT of the `c_risc_client` will be updated with a value corresponding to the IRQ source. The value for SPI's IRQ is 0x03. The exception handler will then compare this PICSTAT value to identify which I/O to serve and which Interrupt Service Routine (ISR) to jump.
4. When the SPI's IRQ occurs, the CPO hardware will raise the `bcp0_exc_flag` and set to flush the IF/ID pipeline register.
5. The `bcp0_stat[1]` (which is the CP0's status register) is set to 1 in order to disable further exception from occurring whereas the `bcp0_cause[6:2]` (which is the CP0's cause register) remain unchanged because the exception code for I/O interrupt is 0.
6. The CP0 will also load the `bicp0_if_pc` (which is the IF stage's PC) into the `bcp0_epc` (which is the CP0's EPC register) for return purpose after executing the exception handler program.
7. The `c_risc_client` will then jump to the exception handler address of 0x8001_b400 in the next clock cycle and start servicing the SPI controller request.
8. After decoding the PICSTAT value to figure out which ISR to jump to, the `c_risc_client` will then branch to the ISR of the respective interrupt source and execute the ISR. For SPI, the starting address of its ISR is 0x8001_b68c.

9. For handling the transmitter buffer empty interrupt, the SPI controller will need to be disabled. At the end, the SPICR holds the value of $8'b0100_0000$ (0x40). After the `c_risc_client` successfully disables the SPI controller, the `uospi_IRQ` from the SPI controller is also removed and no more interrupt request is triggered from the SPI controller.
10. When the interrupt request is successfully handled, the PICSTAT value that stores the I/O interrupt source information will be changed from 0x03 to 0x00 value.
11. After completing the SPI's ISR, the `c_risc_client` will branch to the address of `0x8001_b4bc` and pop all the saved contents from stack before returning to the user program. When the last instruction which is the `eret` instruction is being executed (`bicp0_eret = 1`), it will load the saved return address value from `bcp0_epc` (which is the CP0's EPC register) to the `bocp0_eret_addr` for jumping back to the user program in the next clock cycle.
12. Upon completing the last instruction in the exception handler program, the `c_risc_dut` returns to the place where it was interrupted and starts to execute from that address.
13. The `bcp0_stat[1]` (which is the CP0's status register) is reset to 0 in order to allow further exception from occurring whereas the `bcp0_cause` (which is the CP0's cause register) is cleared in order to remove the exception code stored in it.

7.6.3 Test Case #3: Mode 0 Serial Data Communication

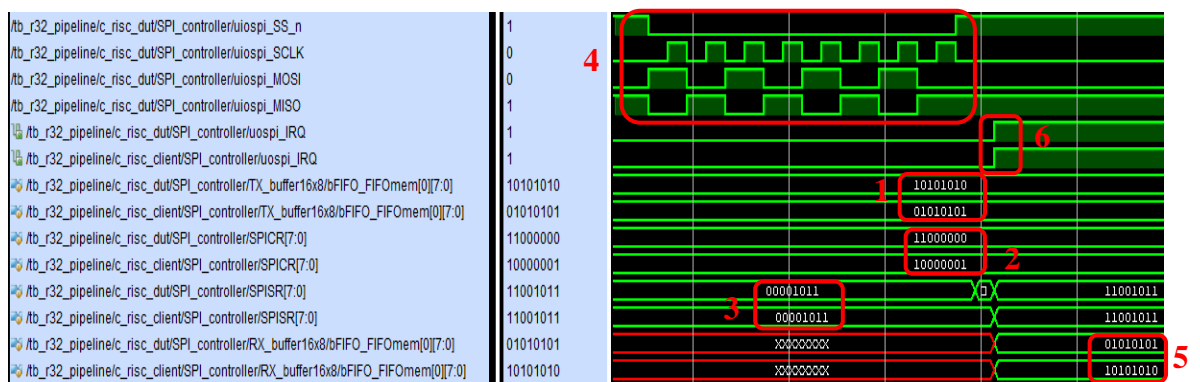


Figure 7.6.3.1: Stimulation result for test case #3 using Vivado stimulator.

1. Before the data exchange begins, the `c_risc_dut` has the data value of `8'b1010_1010 (0xaa)` in its `TX_buffer16x8.bFIFO_FIFOmem[0]` whereas the `TX_buffer16x8.bFIFO_FIFOmem[0]` of the `c_risc_client` holds the data value of `8'b0101_0101 (0x55)`.
2. Same transfer mode, which is mode 0 has been set by both devices for communication.
3. Initially, both of them have disabled the transmitter buffer empty interrupt and enabled the received buffer full interrupt only (for test case #4).
4. The SS pin goes low for 8 SCLK clock cycles. Meanwhile, the data value of `8'b1010_1010 (0xaa)` can be successfully transmitted by the `c_risc_dut` via its MOSI pin. Similarly, the data value of `8'b0101_0101 (0x55)` can also be successfully transmitted by the `c_risc_client` via its MISO pin. Each of these data bit is transmitted serially at one half clock cycle before the rising edge of the SCLK clock.
5. The data on the MOSI and MISO line get exchange after 8 SCLK clock cycles. The data value of `8'b0101_0101 (0x55)` is stored into the `RX_buffer16x8.bFIFO_FIFOmem[0]` of the `c_risc_dut` whereas the data value of `8'b1010_1010 (0xaa)` is stored into `RX_buffer16x8.bFIFO_FIFOmem[0]` of the `DUT_SLAVE`.
6. Both of the devices generate an interrupt request upon completing the data transaction (See test case #4).

7.6.4 Test Case #4: Receiver Buffer Full Interrupt Support

Stimulation Result of c_risc_dut:

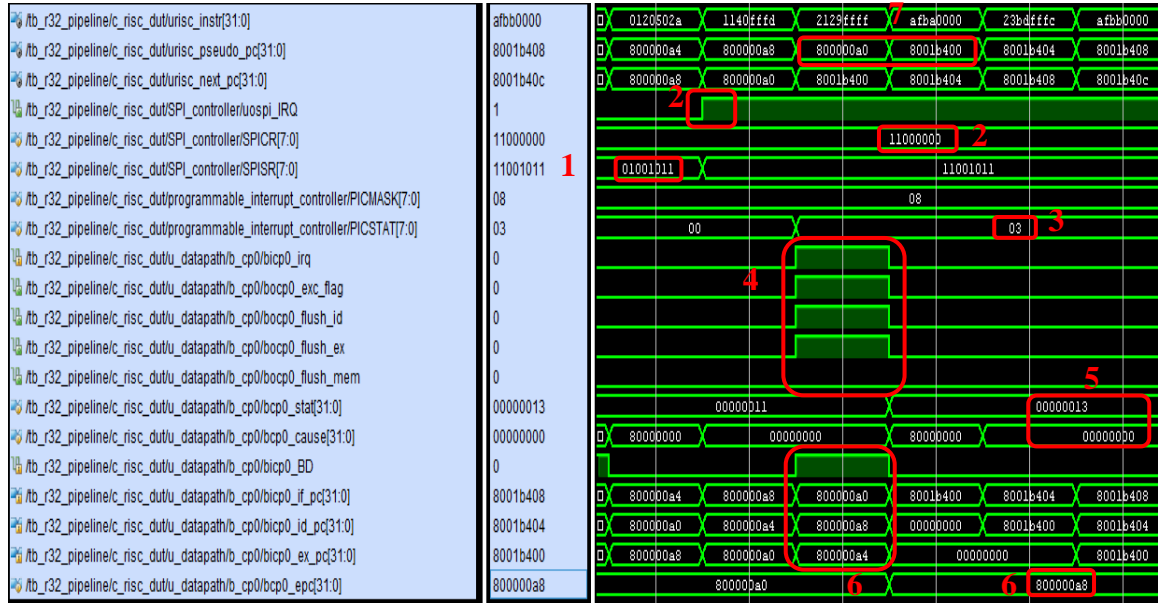


Figure 7.6.4.1: Stimulation result of c_risc_dut for test case #4 using Vivado stimulator.

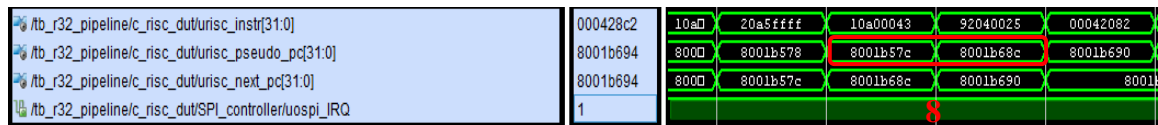


Figure 7.6.4.2: Stimulation result of c_risc_dut for test case #4 using Vivado stimulator (cont'd).

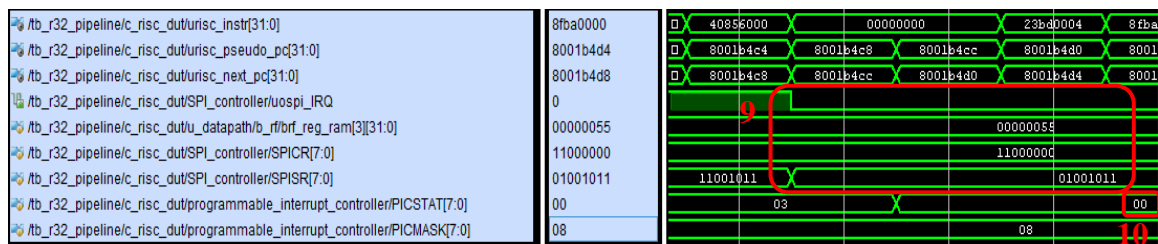


Figure 7.6.4.3: Stimulation result of c_risc_dut for test case #4 using Vivado stimulator (cont'd).

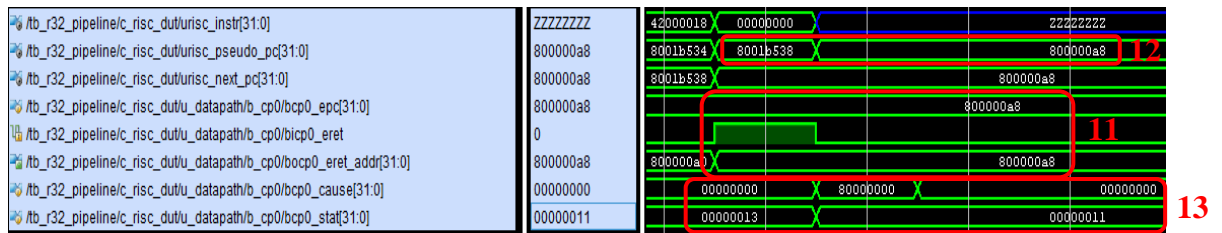


Figure 7.6.4.4: Stimulation result of `c_risc_dut` for test case #4 using Vivado stimulator (cont'd).

1. Since the bit 4 (RXFM) of the SPISR is set to 0, it indicates that only 1-byte of data is expected to be read by CPU.
2. The interrupt request is asserted when the bit 7 (RXDF) of the SPISR become 1 (indicates that the 1-byte of data has been completely received). At the end, the SPISR holds the value of $8'b1100_1011$ (0xcb).
4. When the SPI's IRQ occurs, the CP0 hardware will raise the `bocp0_exc_flag`. Besides, the IF/ID as well as the ID/EX pipeline register are set to be flushed because the exception occurs at the branch delay slot.
6. Since the exception occurs at the branch delay slot, the CP0 loads the `bicp0_id_pc` (which is the ID stage's PC) into the `bcp0_epc` (which is the CP0's EPC register) for return purpose after executing the exception handler program.
9. In response to the receiver buffer full interrupt request, the `c_risc_dut` will handle it by loading the received data into the \$v1 register. After the `c_risc_dut` successfully loads the data from the `RX_buffer16x8` into the \$v1 register, the `uospi_IRQ` from the SPI controller is then removed and no more interrupt request is triggered by the SPI controller. At the end, the \$v1 will hold the received 8-bit data whose value is 0x55.

For the remaining numbers, refer to the stimulation results part of the `c_risc_dut` in test case 2.

Stimulation Result of c_risc_client:

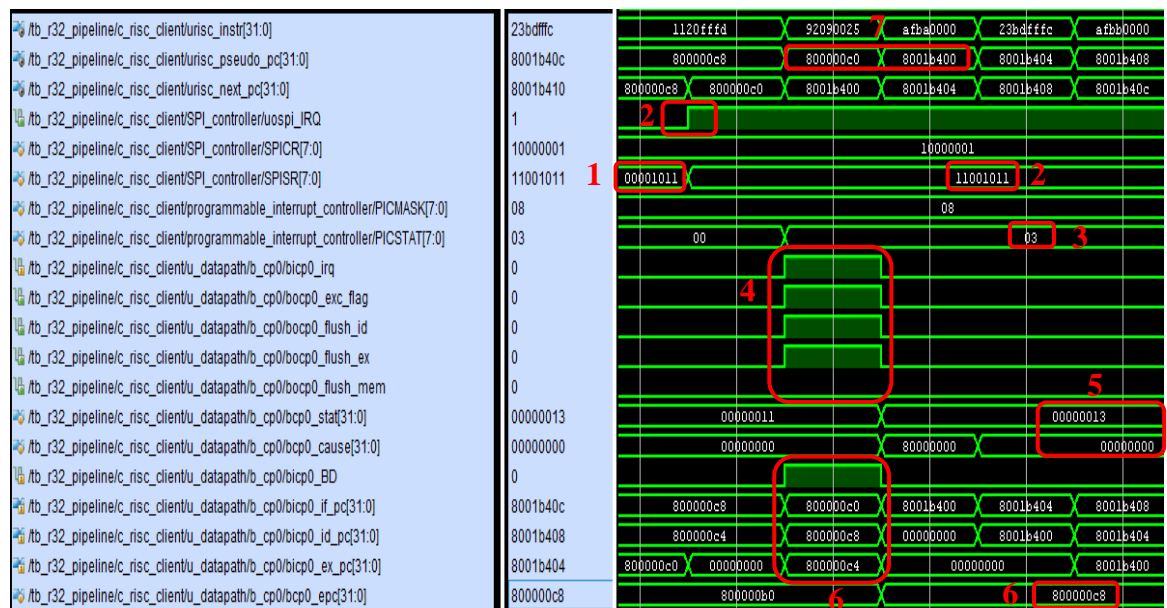


Figure 7.6.4.5: Stimulation result of c_risc_client for test case #4 using Vivado stimulator.



Figure 7.6.4.6: Stimulation result of c_risc_client for test case #4 using Vivado stimulator (cont'd).

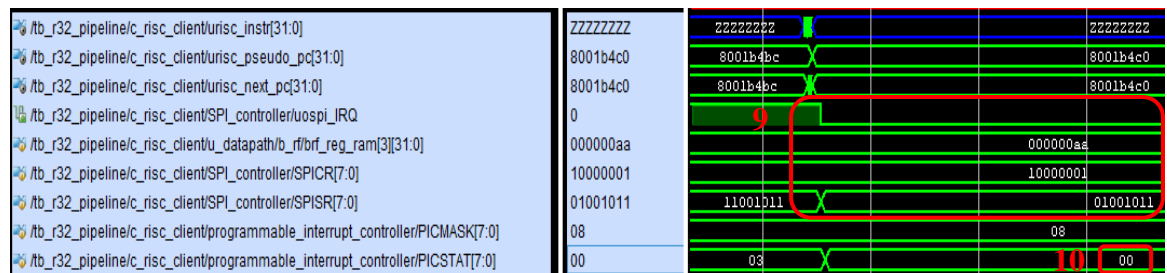


Figure 7.6.4.7: Stimulation result of c_risc_client for test case #4 using Vivado stimulator (cont'd).

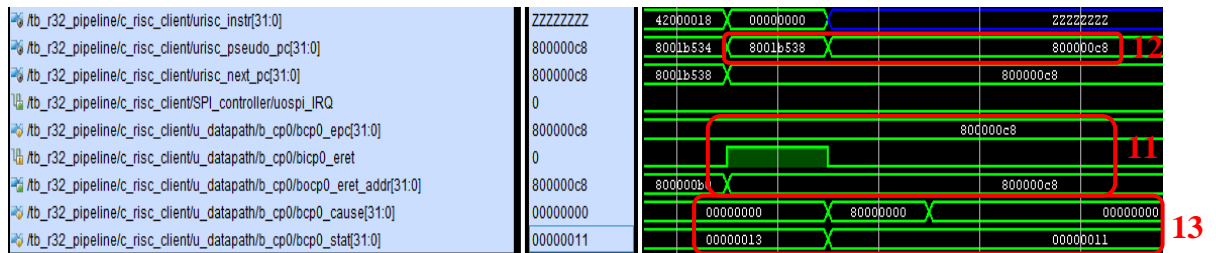


Figure 7.6.4.8: Stimulation result of `c_risc_client` for test case #4 using Vivado stimulator (cont'd).

1. Since the bit 4 (RXFM) of the SPISR is set to 0, it indicates that only 1-byte of data is expected to be read by CPU.
2. The interrupt request is asserted when the bit 7 (RXDF) of the SPISR become 1 (indicates that the 1-byte of data has been completely received). At the end, the SPISR holds the value of $8'b1100_1011$ (0xcb).
4. When the SPI's IRQ occurs, the CP0 hardware will raise the `bcp0_exc_flag`. Besides, the IF/ID as well as the ID/EX pipeline register are set to be flushed because the exception occurs at the branch delay slot.
6. Since the exception occurs at the branch delay slot, the CP0 loads the `bcp0_id_pc` (which is the ID stage's PC) into the `bcp0_epc` (which is the CP0's EPC register) for return purpose after executing the exception handler program.
9. In response to the receiver buffer full interrupt request, the `c_risc_client` will handle it by loading the received data into the `$v1` register. After the `c_risc_client` successfully loads the data from the `RX_buffer16x8` into the `$v1` register, the `uospi_IRQ` from the SPI controller is then removed and no more interrupt request is triggered by the SPI controller. At the end, the `$v1` will hold the received 8-bit data whose value is 0xaa.

For the remaining numbers, refer to the stimulation results part of the `c_risc_client` in test case 2.

7.6.5 Test Case #5: Mode 1 Serial Data Communication

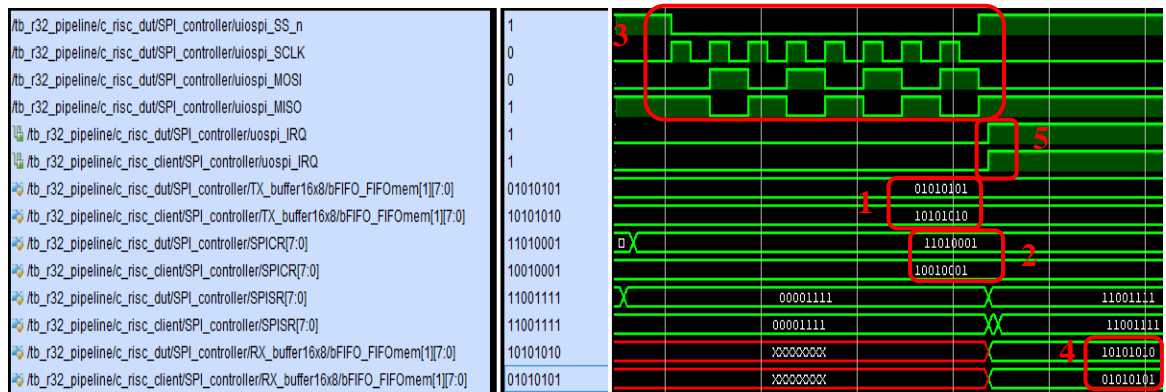


Figure 7.6.5.1: Stimulation result for test case #5 using Vivado stimulator.

1. Before the data exchange begins, the c_risc_dut has the data value of 8'b0101_0101 (0x55) in its TX_buffer16x8.bFIFO_FIFOmem[1] whereas the TX_buffer16x8.bFIFO_FIFOmem[1] of the c_risc_client holds the data value of 8'b1010_1010 (0xaa).
2. Same transfer mode, which is mode 1 has been set by both devices for communication.
3. The SS pin goes low for 8 SCLK clock cycles. Meanwhile, the data value of 8'b0101_0101 (0x55) can be successfully transmitted by the c_risc_dut via its MOSI pin. Similarly, the data value of 8'b1010_1010 (0xaa) can also be successfully transmitted by the c_risc_client via its MISO pin. Each of these data bit is transmitted serially at the rising edge of the SCLK clock.
4. The data on the MOSI and MISO line get exchange after 8 SCLK clock cycles. The data value of 8'b1010_1010 (0xaa) is stored into the RX_buffer16x8.bFIFO_FIFOmem[1] of the c_risc_dut whereas the data value of 8'b0101_0101 (0x55) is stored into RX_buffer16x8.bFIFO_FIFOmem[1] of the DUT_SLAVE.
5. Both of the devices generate an interrupt request upon completing the data transaction.

7.6.6 Test Case #6: Mode 2 Serial Data Communication

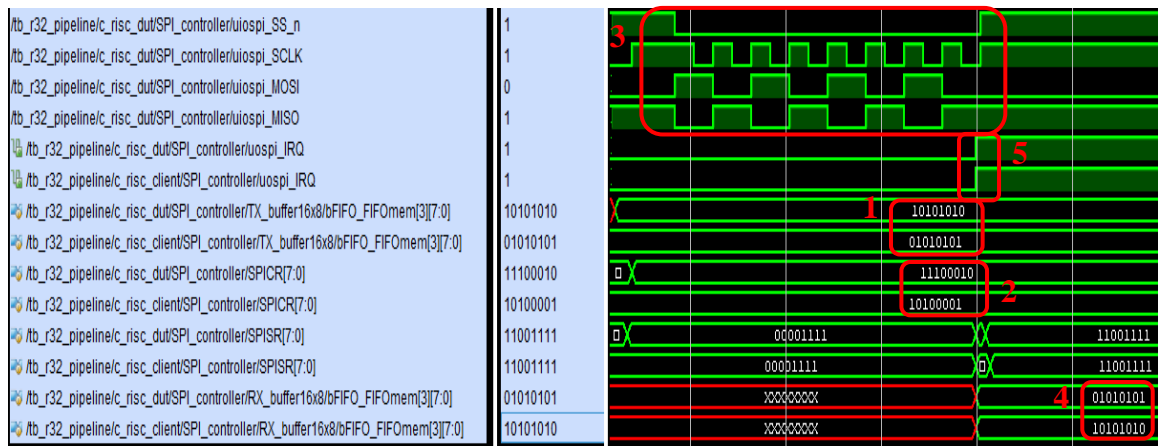


Figure 7.6.6.1: Stimulation result for test case #6 using Vivado stimulator.

1. Before the data exchange begins, the `c_risc_dut` has the data value of `8'b1010_1010` (0xaa) in its `TX_buffer16x8.bFIFO_FIFOmem[3]` whereas the `TX_buffer16x8.bFIFO_FIFOmem[3]` of the `c_risc_client` holds the data value of `8'b0101_0101` (0x55).
2. Same transfer mode, which is mode 2 has been set by both devices for communication.
3. The SS pin goes low for 8 SCLK clock cycles. Meanwhile, the data value of `8'b1010_1010` (0xaa) can be successfully transmitted by the `c_risc_dut` via its MOSI pin. Similarly, the data value of `8'b0101_0101` (0x55) can also be successfully transmitted by the `c_risc_client` via its MISO pin. Each of these data bit is transmitted serially at one half clock cycle before the falling edge of the SCLK clock.
4. The data on the MOSI and MISO line get exchange after 8 SCLK clock cycles. The data value of `8'b0101_0101` (0x55) is stored into the `RX_buffer16x8.bFIFO_FIFOmem[3]` of the `c_risc_dut` whereas the data value of `8'b1010_1010` (0xaa) is stored into `RX_buffer16x8.bFIFO_FIFOmem[3]` of the `DUT_SLAVE`.
5. Both of the devices generate an interrupt request upon completing the data transaction.

7.6.7 Test Case #7: Mode 3 Serial Data Communication

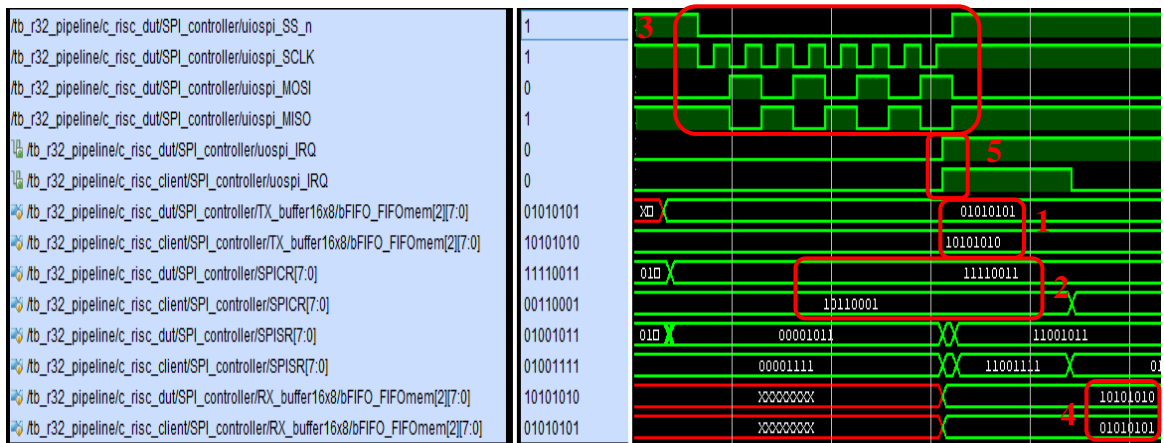


Figure 7.6.7.1: Stimulation result for test case #7 using Vivado stimulator.

1. Before the data exchange begins, the `c_risc_dut` has the data value of `8'b0101_0101` (0x55) in its `TX_buffer16x8.bFIFO_FIFOmem[2]` whereas the `TX_buffer16x8.bFIFO_FIFOmem[2]` of the `c_risc_client` holds the data value of `8'b1010_1010` (0xaa).
2. Same transfer mode, which is mode 3 has been set by both devices for communication.
3. The SS pin goes low for 8 SCLK clock cycles. Meanwhile, the data value of `8'b0101_0101` (0x55) can be successfully transmitted by the `c_risc_dut` via its MOSI pin. Similarly, the data value of `8'b1010_1010` (0xaa) can also be successfully transmitted by the `c_risc_client` via its MISO pin. Each of these data bit is transmitted serially at the falling edge of the SCLK clock.
4. The data on the MOSI and MISO line get exchange after 8 SCLK clock cycles. The data value of `8'b1010_1010` (0xaa) is stored into the `RX_buffer16x8.bFIFO_FIFOmem[2]` of the `c_risc_dut` whereas the data value of `8'b0101_0101` (0x55) is stored into `RX_buffer16x8.bFIFO_FIFOmem[2]` of the `DUT_SLAVE`.
5. Both of the devices generate an interrupt request upon completing the data transaction.

7.6.8 Test Case #8: Mode Fault Error Interrupt Support

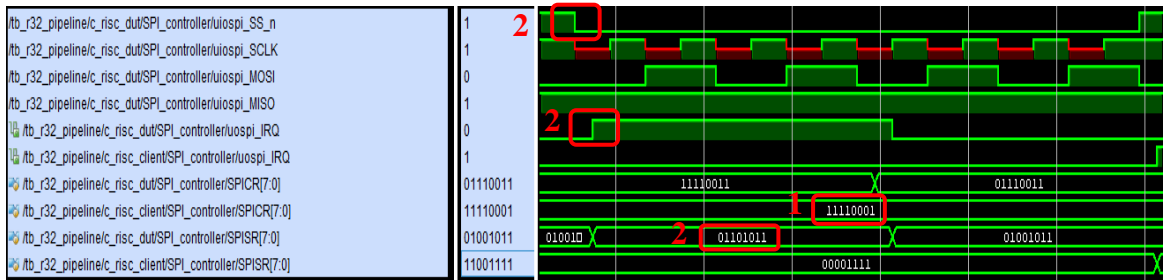


Figure 7.6.8.1: Stimulation result for test case #8 using Vivado stimulator.

Stimulation Result of c_risc_dut:

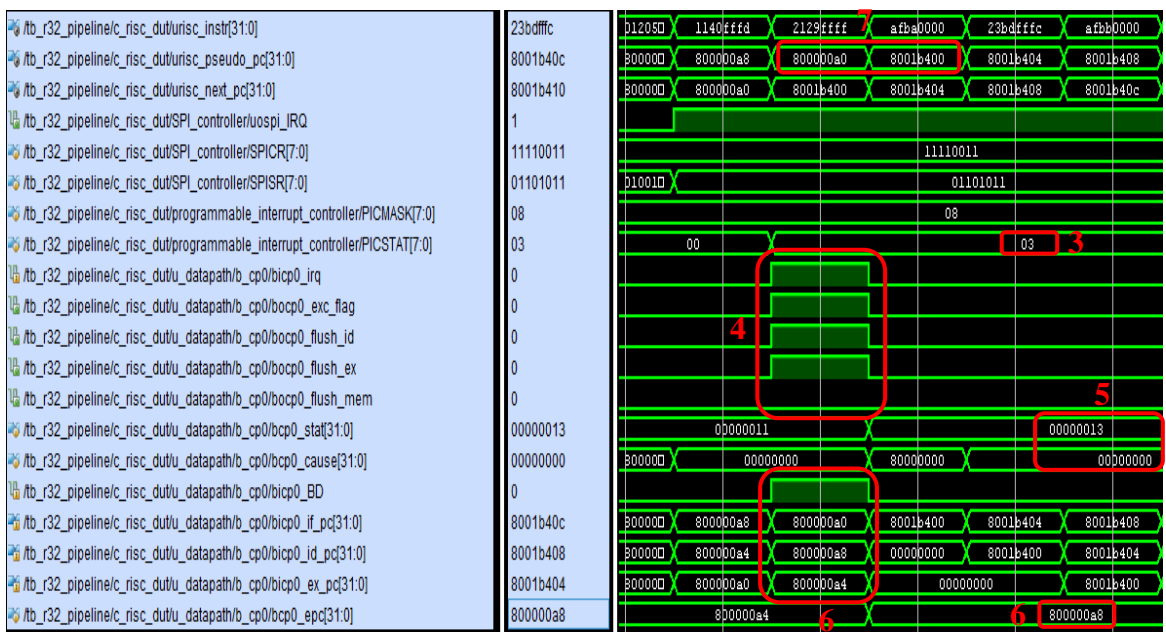


Figure 7.6.8.2: Stimulation result of c_risc_dut for test case #8 using Vivado stimulator.

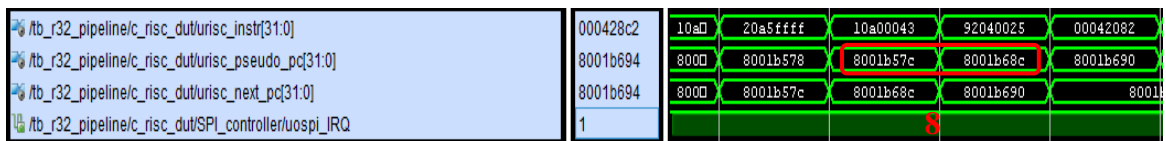


Figure 7.6.8.3: Stimulation result of c_risc_dut for test case #8 using Vivado stimulator

(cont'd)

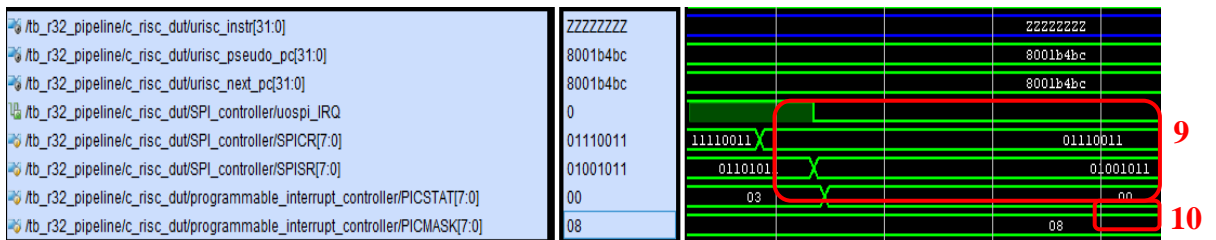


Figure 7.6.8.4: Stimulation result of c_risc_dut for test case #8 using Vivado stimulator (cont'd).

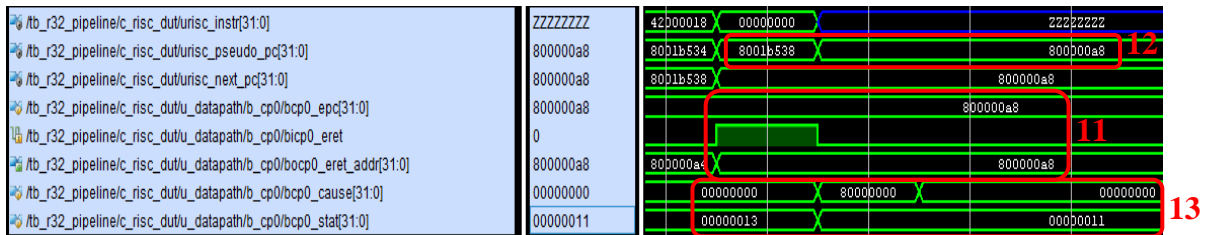


Figure 7.6.8.5: Stimulation result of c_risc_dut for test case #8 using Vivado stimulator (cont'd).

1. In this case, the c_risc_client is reconfigured to act as a master. Since there are two master devices in the same connection, any attempt to pull the SS pin to low will trigger the mode fault error.
2. The newly configured master device, namely c_risc_client attempts to initiate the communication with the c_risc_dut by pulling the SS pin to low. Once the mode fault error is successfully detected, the bit 5 (MODF) of the c_risc_dut's SPIISR will be set to 1. An interrupt request corresponding to the mode fault error detection is immediately issued by the SPI controller in order to alert the c_risc_dut to take action.
4. When the SPI's IRQ occurs, the CP0 hardware will raise the bocp0_exc_flag. Besides, the IF/ID as well as the ID/EX pipeline register are set to be flushed because the exception occurs at the branch delay slot.
6. Since the exception occurs at the branch delay slot, the CP0 loads the bicp0_id_pc (which is the ID stage's PC) into the bcp0_epc (which is the CP0's EPC register) for return purpose after executing the exception handler program.
9. In response to the mode fault error interrupt request, the c_risc_dut will handle it by disabling the SPI controller. At the end, the SPICR holds the value of 8'b0111_0011 (0x73). After the c_risc_dut successfully disables the SPI

controller, the uospi_IRQ from the SPI controller is also removed and no more interrupt request is triggered from the SPI controller.

For the remaining numbers, refer to the stimulation results part of the c_risc_dut in test case 2.

Chapter 8: Synthesis and Implementation

8.1 FPGA Resources Utilization of the Synthesized SPI Controller Unit

After the successful behavioral stimulation of the SPI controller unit, it is then ready for logic synthesis and implementation. The FPGA development board used in this project is the Xilinx Artix-7 XC7A100T FPGA chip on Digilent Nexys 4 DDR board. The resources utilization information of the synthesized SPI controller unit on the selected FPGA board is shown in below.

Name	^1	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Bonded IOB (210)	BUFGCTRL (32)
uspi_v2		183	137	2	1	63	2
bspick_gen (bspick_gen)		23	19	2	1	0	0
bspIRX (bspIRX)		8	23	0	0	0	0
bspITX (bspITX)		42	15	0	0	0	0
RX_buffer16x8 (bFIFO_v2)		52	21	0	0	0	0
SPICR_buffer2x8 (bFIFO_sync)		14	6	0	0	0	0
SPIISR_buffer2x8 (bFIFO_sync_0)		14	6	0	0	0	0
TX_buffer16x8 (bFIFO_v2_1)		26	21	0	0	0	0

Figure 8.1.1: Resource utilization report of the synthesized SPI controller unit on the Nexys 4 DDR (XC7A100T) board.

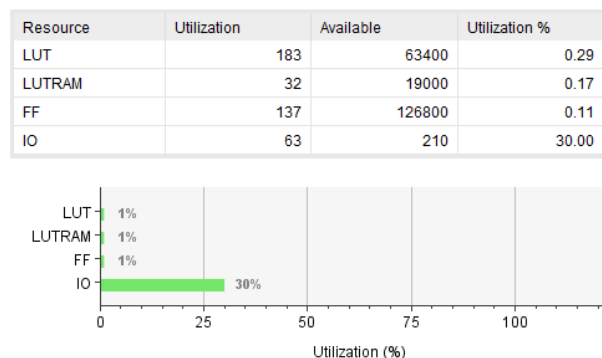


Figure 8.1.2: Resource utilization summary of the synthesized SPI controller unit on the Nexys 4 DDR (XC7A100T) board.

8.2 Timing Analysis

8.2.1 Timing Analysis of the On-board SPI Controller Unit

Since the SPI controller unit is operating at 10Mhz clock frequency, so it is important to make sure that the largest data path delay within it must not exceed $\frac{1}{10\text{Mhz}} = 100\text{ns}$ clock period. In order to obtain its maximum data path delay, static timing analysis is performed on the on-board SPI controller unit after it had been successfully implemented on the selected FPGA board. During the timing analysis, a timing constraint/requirement of 20ns clock period with 50% duty cycle is used because the RISC32 pipeline processor is running at 50Mhz clock frequency. At the end, the maximum data path delay found within the SPI controller unit is about 12.607ns, which is below the 100ns clock period. This indicates that the designed SPI controller unit can operate safely when the 10Mhz clock frequency is used in it. A timing report describing the largest data path delay together with the source and destination of the path is provided in below.

Timing Report

```

Slack (MET) :          7.130ns  (required time - arrival time)
Source:          SPI_controller/RX_buffer16x8/bFIFO_rptr_reg[0]/C
                 (rising edge-triggered cell FDCE clocked by clk_BUFR_BUFG {rise@0.000ns fall@10.000ns period=20.000ns})
Destination:    SPI_controller/TX_buffer16x8/bFIFO_FIFOMem_reg_0_15_0_5/RAMB/I
                 (rising edge-triggered cell RAMD32 clocked by clk_BUFR_BUFG {rise@0.000ns fall@10.000ns period=20.000ns})
Path Group:     clk_BUFR_BUFG
Path Type:      Setup (Max at Slow Process Corner)
Requirement:    20.000ns  (clk_BUFR_BUFG rise@20.000ns - clk_BUFR_BUFG rise@0.000ns)
Data Path Delay: 12.607ns  (logic= 2.210ns (17.530%) route 10.397ns (82.470%))
Logic Levels:   11  (LUT2=1 LUT3=3 LUT5=1 LUT6=5 RAMD32=1)
Clock Path Skew: -0.043ns  (DCD - SCD + CPR)
Destination Clock Delay (DCD): 7.814ns = ( 27.814 - 20.000 )
Source Clock Delay (SCD):      8.683ns
Clock Pessimism Removal (CPR): 0.826ns
Clock Uncertainty: 0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
Total System Jitter (TSJ):     0.071ns
Total Input Jitter (TIJ):      0.000ns
Discrete Jitter (DJ):          0.000ns
Phase Error (PE):              0.000ns

```

Location	Delay type	Incr(ns)	Path(ns)	Netlist Resource(s)
(clock clk_BUFR_BUFG rise edge)				
		0.000	0.000	r
E3	net (fo=0)	0.000	0.000	r uirisc_clk_100mhz (IN)
		0.000	0.000	r uirisc_clk_100mhz
E3	IBUF (Prop_ibuf_I_0)	1.482	1.482	r uirisc_clk_100mhz_IBUF_inst/I
E3	net (fo=1, routed)	3.705	5.187	r uirisc_clk_100mhz_IBUF_inst/O
				r uirisc_clk_100mhz_IBUF
BUFR_X0Y7	BUFR (Prop_bufn_I_0)	0.982	6.169	r BUFR_inst/I
BUFR_X0Y7	net (fo=1, routed)	0.797	6.966	r BUFR_inst/O
				r clk_BUFR_BUFG
BUFGCTRL_X0Y0	BUFG (Prop_bufg_I_0)	0.096	7.062	r BUFG_inst/I
BUFGCTRL_X0Y0	net (fo=3528, routed)	1.621	8.683	r BUFG_inst/O
SLICE_X15Y118	FDCE			r SPI_controller/RX_buffer16x8/urisc_clk
SLICE_X15Y118	FDCE (Prop_fdce_C_Q)	0.419	9.102	r SPI_controller/RX_buffer16x8/bFIFO_rptr_reg[0]/C
SLICE_X14Y118	net (fo=13, routed)	1.162	10.264	r SPI_controller/RX_buffer16x8/bFIFO_FIFOMem_reg_0_15_0_5/ADDRB0
SLICE_X14Y118	RAMD32 (Prop_ramd32_RADR0_0)			r SPI_controller/RX_buffer16x8/bFIFO_FIFOMem_reg_0_15_0_5/RAMB/RADR0
	net (fo=3, routed)	0.327	10.591	r SPI_controller/RX_buffer16x8/bFIFO_FIFOMem_reg_0_15_0_5/RAMB/O
		0.698	11.289	r u_datapath/b_rf/boFIFO_dout[2]
SLICE_X15Y115	LUT3 (Prop_lut3_I0_0)	0.348	11.637	r u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_24_29_i_95/I0
SLICE_X15Y115	net (fo=1, routed)	1.093	12.730	r u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_24_29_i_95/O
SLICE_X6Y113	LUT6 (Prop_lut6_I4_0)	0.124	12.854	r u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_24_29_i_55/I4
SLICE_X6Y113	net (fo=1, routed)	0.941	13.795	r u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_24_29_i_55/O
SLICE_X4Y110	LUT3 (Prop_lut3_I1_0)	0.124	13.919	r u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_24_29_i_33/I1
SLICE_X4Y110	net (fo=1, routed)	0.433	14.353	r u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_24_29_i_33/O
SLICE_X4Y110	LUT2 (Prop_lut2_I0_0)	0.124	14.477	r u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_24_29_i_20/I0
SLICE_X4Y110	net (fo=4, routed)	0.917	15.393	r u_datapath/b_rf/urisc_loaded_data[26]
SLICE_X6Y106				r u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_6_11_i_28/I5

Figure 8.2.1.1: Timing report of the on-board SPI controller unit.

SLICE_X6Y106	LUT6 (Prop_lut6_I5_0) net (fo=1, routed)	0.124 0.528	15.517 16.046	r	u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_6_11_i_28/0 u_datapath/b_rf/data1_0[10]
SLICE_X8Y106	LUT6 (Prop_lut6_I2_0) net (fo=1, routed)	0.124 0.407	16.170 16.577	r	u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_6_11_i_20/I2 u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_6_11_i_20/0
SLICE_X11Y105	LUT6 (Prop_lut6_I4_0) net (fo=4, routed)	0.124 0.931	16.701 17.632	r	u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_6_11_i_11/I4 u_datapath/b_rf/brf_reg_ram_reg_r1_0_31_6_11_i_11/0 u_datapath/data4[10]
SLICE_X30Y103	LUT3 (Prop_lut3_I0_0) net (fo=3, routed)	0.124 1.268	17.756 19.024	r	u_datapath/udp_fw_data_mem_rt32[10]_i_1/I0 u_datapath/udp_fw_data_mem_rt32[10]_i_1/0 u_datapath/udp_fw_data_ex_rt32[10]
SLICE_X5Y101	LUT6 (Prop_lut6_I3_0) net (fo=1, routed)	0.124 0.545	19.148 19.693	r	u_datapath/bFIFO_mem_reg_0_1_0_5_i_19/I3 u_datapath/bFIFO_mem_reg_0_1_0_5_i_19/0 u_datapath/bFIFO_mem_reg_0_1_0_5_i_19_n_2
SLICE_X5Y103	LUT5 (Prop_lut5_I0_0) net (fo=29, routed)	0.124 1.473	19.817 21.290	r	u_datapath/bFIFO_mem_reg_0_1_0_5_i_5/I0 u_datapath/bFIFO_mem_reg_0_1_0_5_i_5/0 SPI_controller/TX_buffer16x8/bFIFO_FIFOmem_reg_0_15_0_5/DI00
SLICE_X34Y115	RAMD32			r	SPI_controller/TX_buffer16x8/bFIFO_FIFOmem_reg_0_15_0_5/RAMB/I

	(clock clk_BUFR_BUFGB rise edge)				
		20.000	20.000	r	
E3	net (fo=0)	0.000	20.000	r	uirisc_clk_100mhz (IN)
E3	IBUF (Prop_ibuf_I_0) net (fo=1, routed)	1.411 3.143	21.411 24.554	r	uirisc_clk_100mhz_IBUF_inst/I uirisc_clk_100mhz_IBUF_inst/0 uirisc_clk_100mhz_IBUF
BUFR_X0Y7	BUFR (Prop_bufn_I_0) net (fo=1, routed)	0.918 0.751	25.472 26.223	r	BUFR_inst/I BUFR_inst/0 clk_BUFR_BUFGB
BUFGCTRL_X0Y0	BUFG (Prop_bufg_I_0) net (fo=3528, routed)	0.091 1.500	26.314 27.814	r	BUFG_inst/I BUFG_inst/0 SPI_controller/TX_buffer16x8/bFIFO_FIFOmem_reg_0_15_0_5/WCLK
SLICE_X34Y115	RAMD32 clock pessimism clock uncertainty	0.826 -0.035	28.640 28.604	r	SPI_controller/TX_buffer16x8/bFIFO_FIFOmem_reg_0_15_0_5/RAMB/CLK
SLICE_X34Y115	RAMD32 (Setup_ramd32_CLK_I)	-0.185	28.419	r	SPI_controller/TX_buffer16x8/bFIFO_FIFOmem_reg_0_15_0_5/RAMB

	required time		28.419		
	arrival time		-21.290		

	slack		7.130		

Figure 8.2.1.2: Timing report of the on-board SPI controller unit (cont'd).

8.2.2 Timing Analysis of the RISC32 with the SPI Controller Unit

As mentioned previously, the RISC32 pipeline processor is currently running at 50Mhz clock frequency. Thus, it is crucial to ensure that the RISC32 can still operating at the same clock frequency even after the SPI controller unit has been integrated into it. To check this, static timing analysis is conducted on the entire RISC32 to obtain the maximum data path delay. Same period constraint of 20ns is applied here.

From Figure 8.2.2.1, the worst negative slack (WNS) is calculated to be 2.056ns (> 0) which shows that it has spare time after meeting the timing requirement. $WNS = 2.056ns$ indicates that it takes only approximate 17.944ns (20ns-2.056ns, where 20ns is the time period of 1 clock cycle for 50Mhz clock frequency that is specified in the constraints file) to complete the execution. Therefore, the maximum possible frequency that can be used is $\frac{1}{17.944ns} \approx 55Mhz$. In other words, the RISC32 with the on-board SPI controller unit can still run at 50Mhz in terms of implementation. This is further supported with the result shown in Figure 8.2.2.2. As we can observe from Figure 8.2.2.2, the largest data path delay of the RISC32 is only about 17.731ns, which does not exceed the 20ns clock period. In short, the integrated SPI controller unit has minimum impact on the RISC32 pipeline processor in terms of timing requirement.

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 2.056 ns	Worst Hold Slack (WHS): 0.026 ns	Worst Pulse Width Slack (WPWS): 8.334 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 9419	Total Number of Endpoints: 9419	Total Number of Endpoints: 3530

All user specified timing constraints are met.

Figure 8.2.2.1: Design timing summary of the entire RISC32 pipeline processor.

Name	Slack	Levels	High Fanout	From	To	Total Delay
Path 1	2.056	17	79	u_datapath/udp_mem_alb_out_reg[5]/C	ADC/UADC_CREG_reg[1][28]/D	17.731
Path 2	2.057	17	79	u_datapath/udp_mem_alb_out_reg[5]/C	ADC/UADC_CREG_reg[2][28]/D	17.710
Path 3	2.138	18	79	u_datapath/udp_mem_alb_out_reg[5]/C	data_ram/luram_...reg_0/DIAD[1]	17.533
Path 4	2.194	17	79	u_datapath/udp_mem_alb_out_reg[5]/C	ADC/UADC_CREG_reg[4][28]/D	17.613
Path 5	2.219	17	79	u_datapath/udp_mem_alb_out_reg[5]/C	ADC/UADC_CREG_reg[1][24]/D	17.562
Path 6	2.235	17	79	u_datapath/udp_mem_alb_out_reg[5]/C	ADC/UADC_CREG_reg[3][28]/D	17.558
Path 7	2.249	17	79	u_datapath/udp_mem_alb_out_reg[5]/C	ADC/UADC_CREG_reg[5][28]/D	17.557
Path 8	2.276	16	79	u_datapath/udp_mem_alb_out_reg[5]/C	dcache/cache_ce...reg/DIAD[12]	17.452
Path 9	2.320	17	79	u_datapath/udp_mem_alb_out_reg[5]/C	ADC/UADC_CREG_reg[2][24]/D	17.464
Path 10	2.352	17	79	u_datapath/udp_mem_alb_out_reg[5]/C	ADC/UADC_CREG_reg[1][31]/D	17.418

Figure 8.2.2.2: Top 10 paths in the RISC32 pipeline processor that have the largest total data path delay.

8.3 Proposed Hardware Implementation

Only a few external components are required to build up the typical transceiver circuit for wireless communication in this project and they are the antenna, bias resistors, decoupling capacitors, inductors and reference crystal. The overview of the external components used can be found in Appendix C.2 and C.3 respectively. Proper power supply must be used for the error-free performance of the CC2420 transceiver. Thus, a suggested power supply of 3.3 V is used as its voltage regulator power supply input in the circuit. In addition, it is also connected directly to the RISC32 pipeline processor via the SPI bus and several GPIO pins. The transceiver circuit is designed and followed closely based on the reference circuit provided in its datasheet in order to obtain the optimum performance of the Zigbee module. Figure 8.3.1. gives the full information about how to interface the CC2420 transceiver with the RISC32 pipeline processor.

In this project, there will be only one SPI master and one SPI slave in one node. At the end, there will be two nodes used in this project because a single Zigbee module is meaningless and a pair of Zigbee modules are always required in order to communicate with each other wirelessly. In each node, the RISC32 pipeline processor acts as the SPI master, providing the serial clock to initiate communication with the CC2420 transceiver that acts as the SPI slave. The Zigbee module is eventually monitored, controlled and programmed by the RISC32 pipeline processor via the high-speed SPI bus and the connected GPIO pins. The S25FL128S SPI serial flash memory is also available in the connection for program storage purpose. Moreover, a simple active-high LED circuit that consist of LED1, LED2, LED3 and LED4 is setup and driven by the RISC32 pipeline processor for debugging purpose. Upon requested, data received by the CC2420 module can be read by the RISC32 pipeline processor via the SPI bus. After that, the data can be displayed on a PC by using the UART serial communication. In order to do this, the RISC32 pipeline processor needs to be connected to the UART interface of a PC that has the PuTTY software.

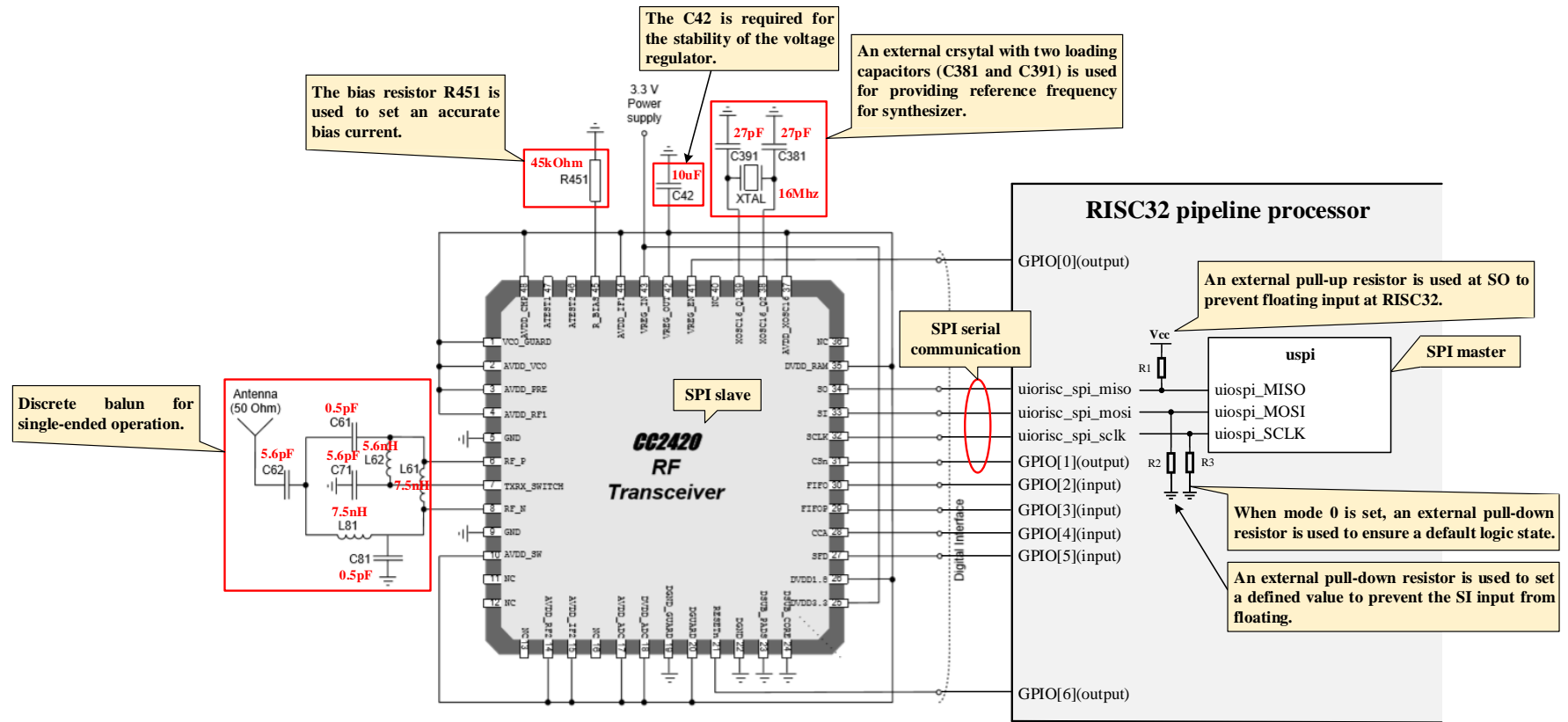


Figure 8.3.1: Detailed descriptions about the connection mechanism of the RISC32 pipeline processor with the CC2420 transceiver by using only few external components.

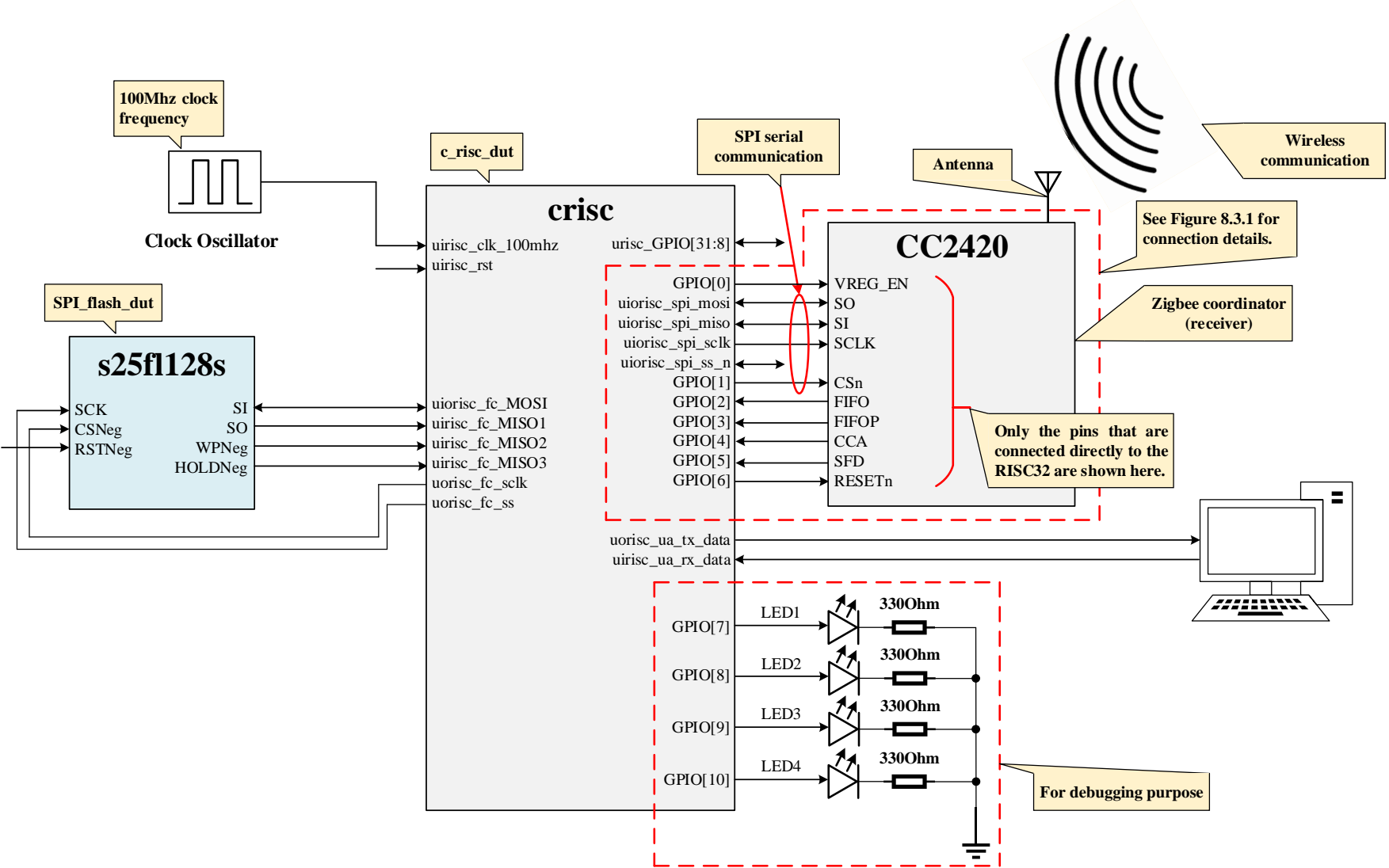


Figure 8.3.2: Connection mechanism of the c_risc_dut with the CC2420 transceiver for wireless communication.

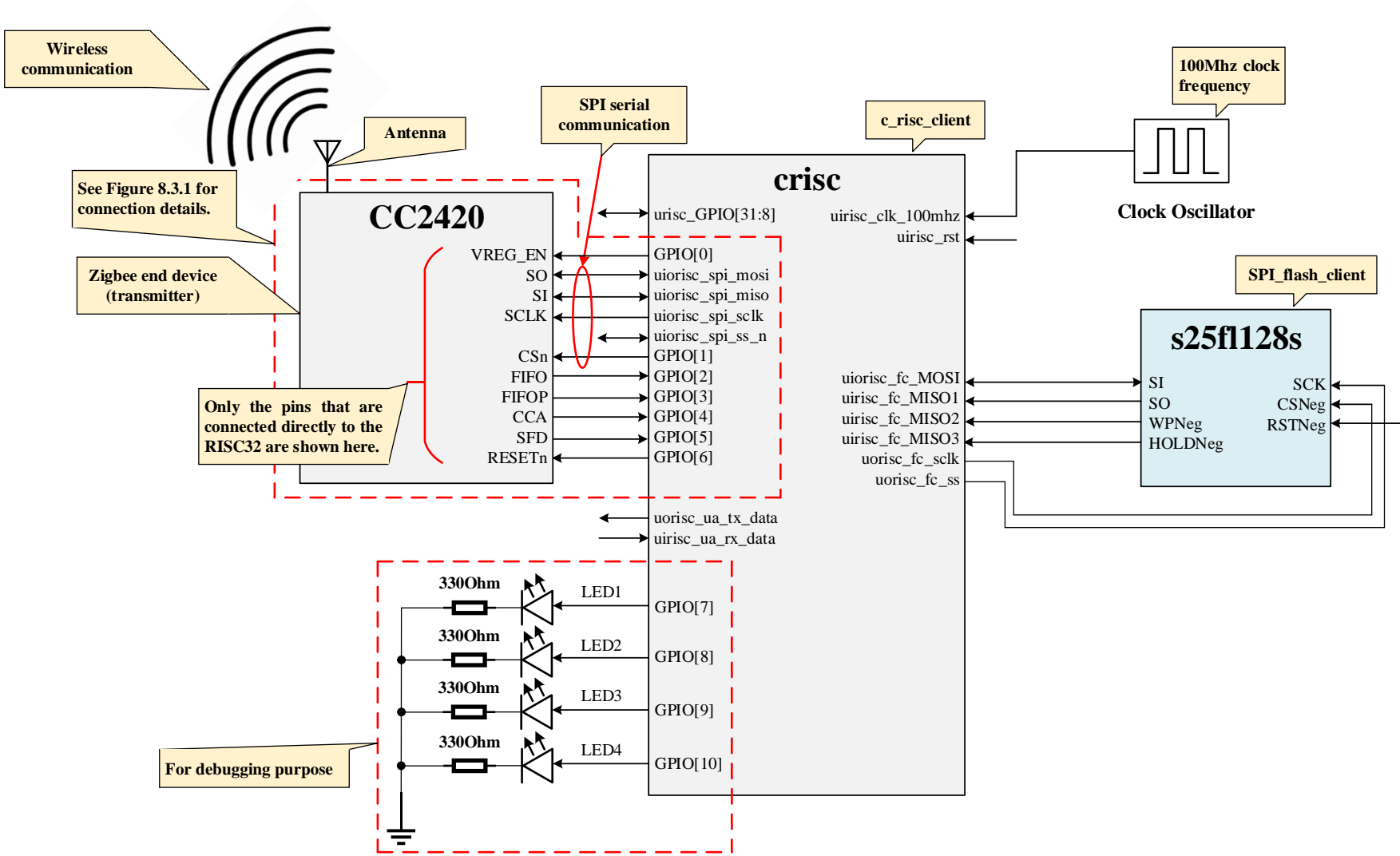


Figure 8.3.3: Connection mechanism of the `c_risc_client` with the CC2420 transceiver for wireless communication.

8.4 Proposed Software Implementation

After setting up the necessary connections and circuits, the CC2420 transceiver is first initialized by using the connected SPI bus and GPIO pins. By using SPI serial communication, the RISC32 pipeline processor can send and read data from the CC2420 transceiver simultaneously through the SPI interface.

First of all, activate the CC2420's voltage regulator for 1.8V power supply and wait around 1ms before proceeding to ensure the voltage regulator has powered up. Upon activated, reset the CC2420 transceiver and then issue the SXOSCON command strobe for activating its crystal oscillator. Once the crystal oscillator of the CC2420 transceiver is running, all of its FIFO/RAM can be accessed. Finally, configure both CC2420 transceivers by programming their configuration registers and FIFO/RAM respectively.

As stated in the Zigbee protocol, in a network, one device needs to act as the coordinator and the rest can be either routers or end devices. Therefore, one of the C2420 transceiver is configured as the coordinator whereas the other is used as the end device. Point-to-point network is used as both Zigbee modules will transmit and receive data wirelessly from each other only. In order for data to transmit and receive from one CC2420 transceiver to another, they need to be properly configured on the same network and using the same frequency channel. In this project, the PAN id is manually programmed to be 1 and both Zigbee modules are set to operate in channel 12.

After completing the initialization, both CC2420 transceivers are ready to be used. Besides configuring the C2420 transceiver, the RISC32 pipeline processor also performs the frame-transfer operations. The processor of the Zigbee end device transmits data frame serially to the Zigbee end device through the SPI bus. The Zigbee end device will first check its CCA signal for congestion avoidance and do not transmit unless the channel is clear. Once the CCA signal goes high, it will then transmit the data wirelessly to the Zigbee coordinator and wait for acknowledgement.

On the other hand, the Zigbee coordinator continuously looks for any wireless radio message sent to it. It will only receive the data frame if and only if the address recognition is successful. After that, it will pass the received data frame serially to the processor interfaced to it and automatically issue an acknowledgement of the data frame back to the Zigbee end device, confirming the successful frame reception. Figure 8.4.1 depicts the expected wireless communication between both CC2420 transceivers. In

addition, Section 8.4.1 and 8.4.2 provide the full descriptions of the MIPS program for `c_risc_dut` (interfaced with the Zigbee coordinator) and `c_risc_client` (interfaced with the Zigbee end device) in terms of flowchart.

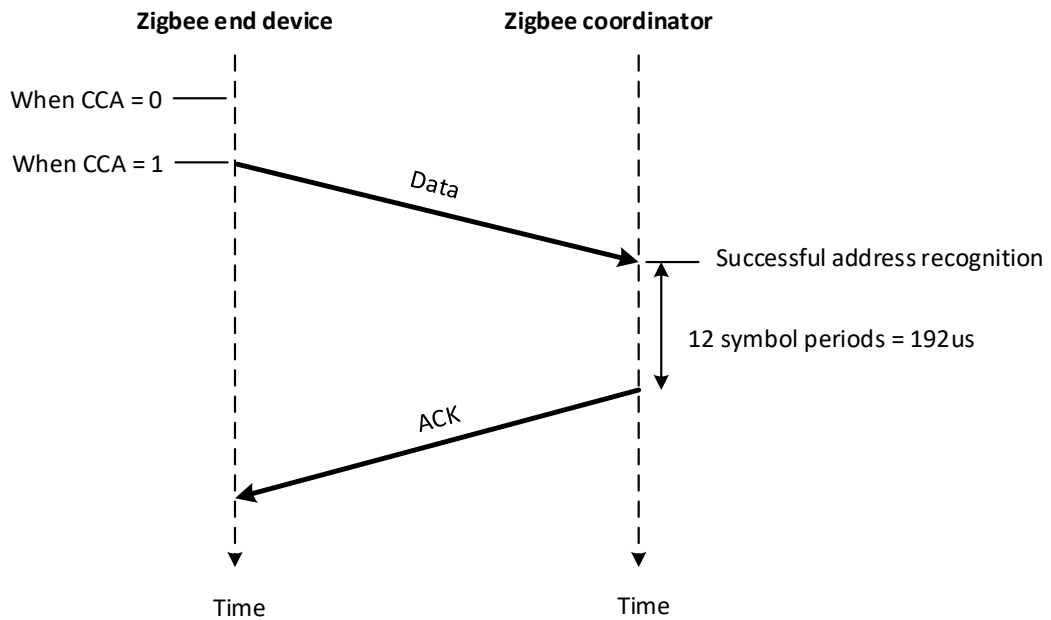


Figure 8.4.1: Expected wireless communication between the Zigbee end device and the Zigbee coordinator in this project.

8.4.1 Flowchart of the Hardware/Software Behaviors in `c_risc_dut`

A MIPS program for `c_risc_dut` in terms of hardware implementation has been developed and the flowchart below provides the full information about the hardware/software behaviors in `c_risc_dut` for on-board testing.

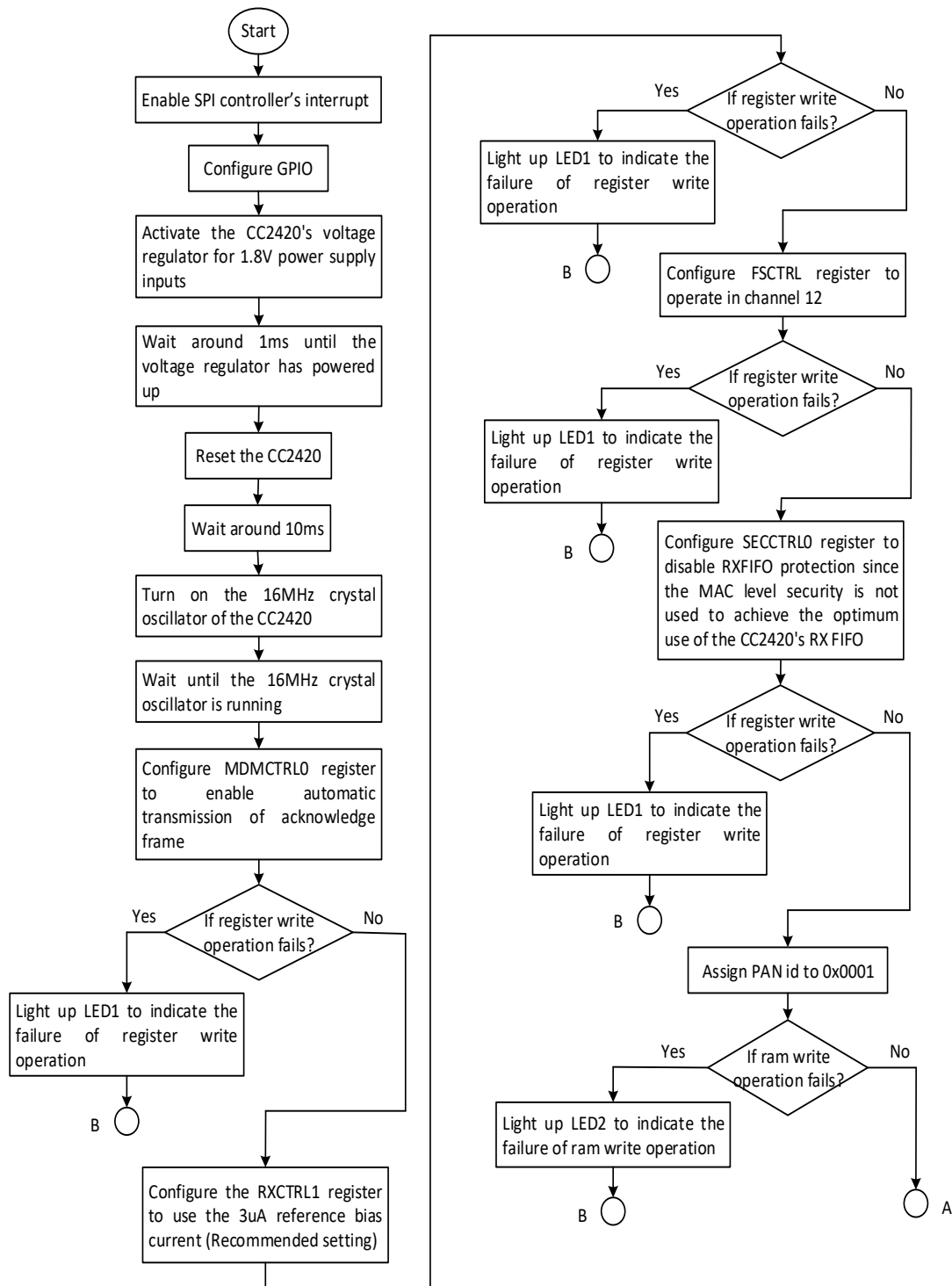


Figure 8.4.1.1: Flowchart of the hardware/software behaviors in `c_risc_dut` for on-board testing.

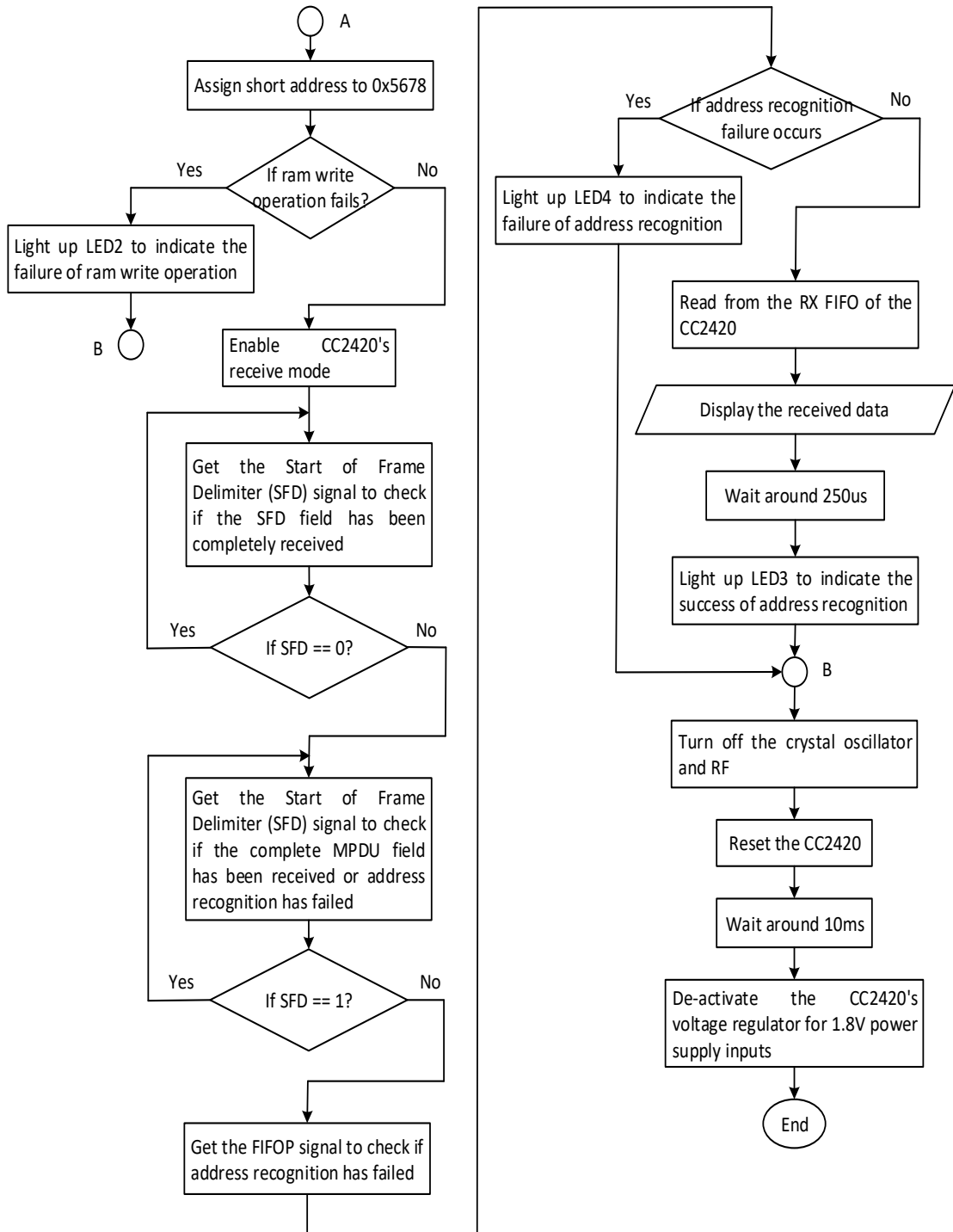


Figure 8.4.1.2: Flowchart of the hardware/software behaviors in c_risc_dut for on-board testing. (cont'd).

8.4.2 Flowchart of the Hardware/Software Behaviors in `c_risc_client`

A MIPS program for `c_risc_client` in terms of hardware implementation has been developed and the flowchart below provides the full information about the hardware and software behaviors in for on-board testing.

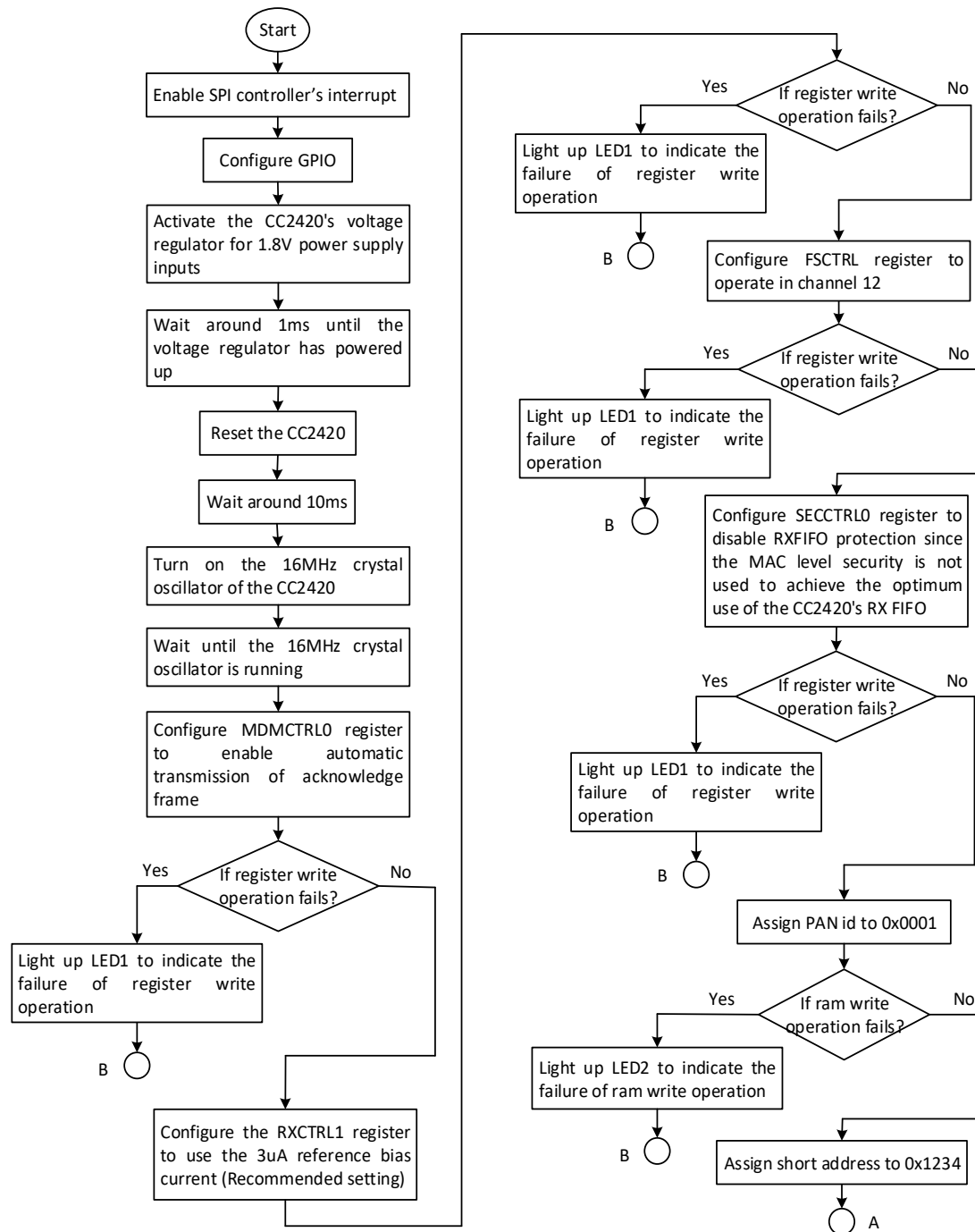


Figure 8.4.2.1: Flowchart of the hardware/software behaviors in `c_risc_client` for on-board testing.

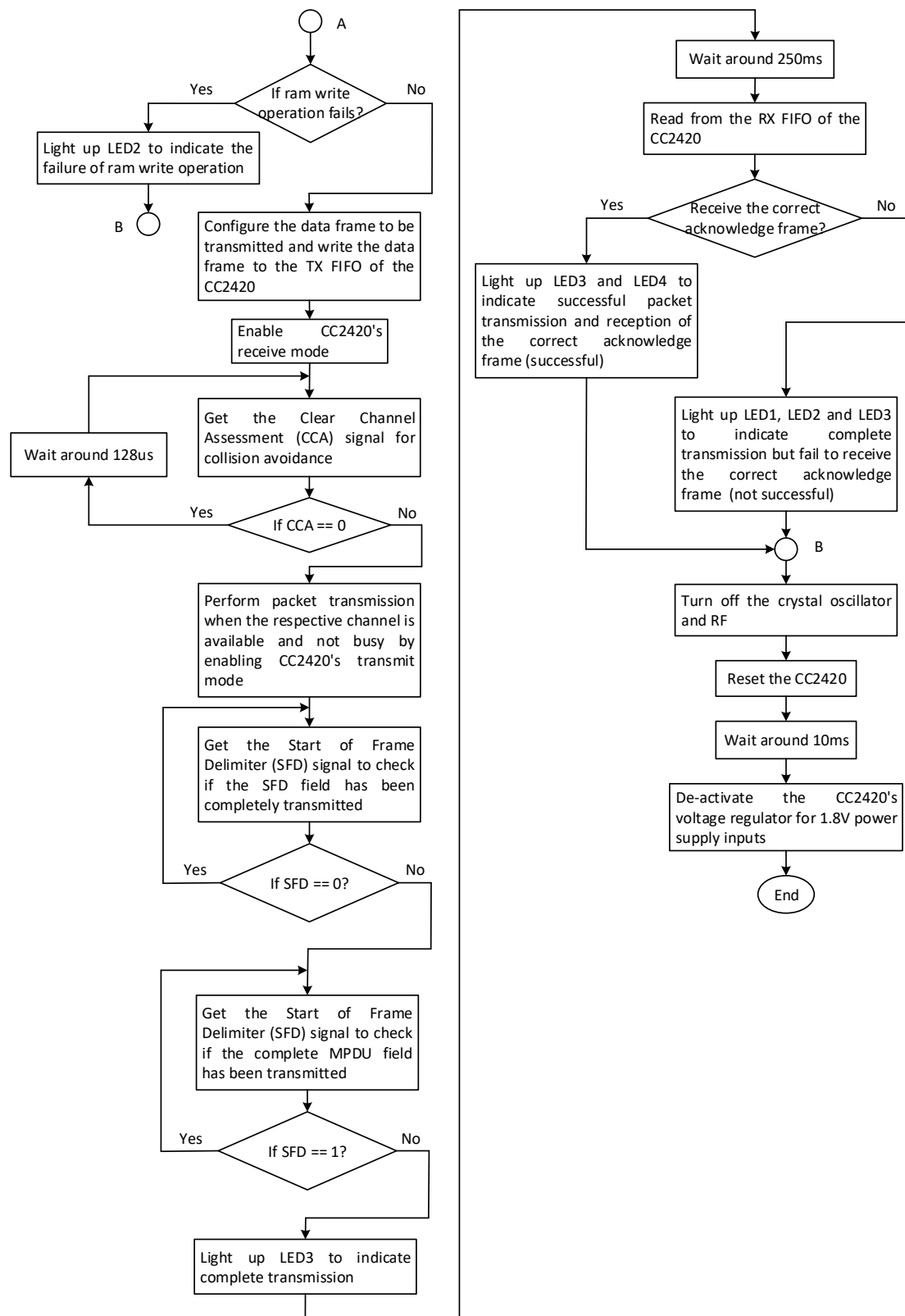


Figure 8.4.2.2: Flowchart of the hardware/software behaviors in c_risc_client for on-board testing (cont'd).

Chapter 9: Conclusion and Future Work

9.1 Conclusion

The first two objectives of this project have been achieved. The previously developed SPI controller unit has been revised and further enhanced. It can now perform full-duplex data communication correctly with another SPI-interface device in all of the 4 transfer modes (mode 0, 1, 2, 3) in both master and slave operations. The micro-architecture specification of the designed SPI controller unit and its internal blocks have been presented and they can be found in Chapter 5. With the availability of well-developed designed documents, the research works can now be done easier and speed up significantly as a SPI controller that meet the standard SPI protocol can now be built easily.

Besides, it has also been successfully integrated into the RISC32 pipeline processor by using the I/O memory mapping technique. It can function well with the RISC32 pipeline processor and vice versa across the CDC boundaries regardless of interrupt or polling method is being used. Chapter 6 provides the full information about the Interrupt Service Routine (ISR) that is developed specifically for the SPI controller unit. In addition, the designed SPI controller has been functionally proven and is able to meet all of the specified functional requirements, either as a single unit or in a whole system. A comprehensive documentation about the verification specifications, test plans, test programs and testbenches of the SPI controller unit has been well-developed and maintained. All of the stimulation results can be found in Chapter 7.

However, in response to the COVID-19 pandemic in the country, the university campus is closed temporarily during the implementation of MCO. As a result, the third objective of this project is only partially completed as the physical design (which is the on-board testing with Zigbee module) cannot be completed due to the inaccessibility of the required lab hardware resources and equipment. By using the Vivado Design Suite tools, the RISC32 pipeline processor can be stimulated and synthesized into the Digilent Nexys 4 DDR (XC7A100T) board. At the end, the designed SPI controller unit is fully synthesizable and can operate safely at its 10Mhz I/O clock. Moreover, it has minimum impact on the integrated RISC32 pipeline processor in terms of timing requirement. The information about FPGA resource utilization and timing requirement are available in Chapter 8.

9.2 Future Work

In future, the on-board testing with Zigbee module may need to be carried out. By using the proposed solutions for hardware and software implementation presented in Chapter 8, a final testing can be performed easily for demonstrating the transfer of data between two FPGA boards via the CC2420 transceivers.

Moreover, the designed SPI controller unit can also be further enhanced by having two modes, namely normal mode and bidirectional mode so that it can communicate with 3-wire and 4-wire SPI devices in future. Four external pins will be used in the normal mode to perform the full-duplex data communication whereas three external pins will be used in the bidirectional mode for half-duplex data communication. This special feature could allow the SPI-equipped processor to interface more flexibly with all types of SPI devices in the market.

Bibliography

- Anusha 2017, *Basics of Serial Peripheral Interface (SPI)*. Available from: <<https://www.electronicshub.org/basics-serial-peripheral-interface-spi/>>. [Accessed on 10th March 2019].
- Choudhury, S, Singh, GK, & Mehra, RM 2014, *Design and Verification Serial Peripheral Interface (SPI) Protocol for Low Power Applications*. Available from: <<http://www.rroj.com/open-access/design-and-verification-serial-peripheralinterface-spi-protocol-for-low-powerapplications.pdf> >. [Accessed on 1st April 2019].
- Cummings, CE 2002, *Stimulation and Synthesis Techniques for Asynchronous FIFO Design*. <http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf>. [Accessed on 10th March 2019].
- Cummings, CE 2008, *Clock Domain Crossing (CDC) Design & Verification Technique Using System Verilog*. <http://www.sunburst-design.com/papers/CummingsSNUG2008Boston_CDC.pdf> [Accessed on 14th March 2019].
- Cummings, CE, & Alfke, P 2002, *Stimulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons*. <http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf>. [Accessed on 10th March 2019].
- CORELIS n.d., *SPI Tutorial*. Available from: <<https://www.corelis.com/education/tutorials/spi-tutorial/> >. [Accessed on 20th February 2020].
- Patterson, DA & Hennessy, JL 2005, *Computer Organization and Design: The Hardware/Software Interface (3th edition)*. San Francisco: Morgan Kaufmann Publishers.
- De la Piedra, A, Braeken, A, & Touhafi, A 2012, ‘Sensor Systems Based on FPGAs and Their Applications’, A Survey. *Sensors*.12(9), pp.12235-12264. Available from: <<https://doi.org/10.3390/s120912235>>. [Accessed on 31st March 2019].

Bibliography

- Dhaker, P 2018, *Introduction to SPI Interface*. Available from <<https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html>>. [Accessed on 20th February 2020].
- Elprocus n.d., *Zigbee Wireless Technology Architecture and Applications*. Available from: <<https://www.elprocus.com/what-is-zigbee-technology-architecture-and-its-applications/>>. [Accessed on 10 March 2019].
- Goh, JS 2019, *The development of an exception scheme for 5-stage pipeline RISC processor*. Available from <http://eprints.utar.edu.my/3434/1/fyp_CT_2019_GJS_1503470.pdf>. [Accessed on 1st January 2020].
- Kiat, WP 2018, *The Design of an FPGA-based Processor with Reconfigurable Processor Execution Structure for Internet of Things (IoT) applications*. Available from: <<http://eprints.utar.edu.my/3146/>> [Accessed on 31st March 2019].
- Leens, F 2009, 'An introduction to I2C and SPI protocols', *IEEE Instrumentation & Measurement Magazine*, 12(1), pp.8-13. Available from: <<https://ieeexplore.ieee.org/document/4762946>>. [Accessed on 31st March 2019].
- Linda, R 2017, *Zigbee*. Available from: <<https://internetofthingsagenda.techtarget.com/definition/ZigBee>>. [Accessed on 10th March 2019].
- Kaneria, D 2014, *Serial Peripheral Interface (SPI)*. Available from: <<https://www.slideshare.net/DhavalKaneria/serial-peripheral-interfacespi>>. [Accessed on 20th February 2020].
- Laner, M 2016, *Memory Mapped I/O, Polling, DMA*. Available from <<http://www.cim.mcgill.ca/~langer/273/20-notes.pdf> >. [Accessed from 25th March 2020].
- Larson, S 2019, *SPI 3-Wire master (VHDL)*. Available from <[https://www.digikey.com/eewiki/pages/viewpage.action?pageId=27754638#SPI-3-WireMaster\(VHDL\)-Clocking](https://www.digikey.com/eewiki/pages/viewpage.action?pageId=27754638#SPI-3-WireMaster(VHDL)-Clocking)>. [Accessed form 20th February 2020].

Bibliography

- Mok, KM 2015, Digital Systems Designs, lecture notes distributed in Faculty of Information and Communication Technology at Universiti Tunku Abdul Rahman.
- Mok, KM 2015, Computer Organization and Architecture, lecture notes distributed in Faculty of Information and Communication Technology at Universiti Tunku Abdul Rahman.
- Motorola Inc. 2003, *SPI Block Guide V03.06*. Available from : <http://www.cse.chalmers.se/~svenk/mikrodatorsystem/HC12/reference_manuals/S12SPIV3.pdf>. [Accessed on 10th March 2019].
- Oudhida, AK, Berrabdo, ML, Liacha, R, Tiar, K & Alhoumays, YN 2010, 'Design and Test of General-Purpose SPI Master/Slave IPs on OPB Bus', *2010 7th International Multi-Conference on Systems, Signals and Devices*, pp. 1-6. Available from: <<https://ieeexplore.ieee.org/document/5585592> >. [Accessed on 3rd April 2019].
- Polytechnic Hub 2017, *Application of serial peripheral interface (SPI)*. Available from: <<https://www.polytechnichub.com/applications-serial-peripheral-interface-spi/>>. [Accessed on 1st April 2019].
- Priya n.d., *IoT Communication between two devices over Zigbee Protocol: IOT Part 37*. Available from: <<https://www.engineersgarage.com/Contribution/D2D-IoT-Communication-Zigbee-Protocol>>. [Accessed on 12th March 2019].
- Sporre, K 2018, *IoT Development with Wireless Communications: Getting Started*. Available from: <<https://www.digi.com/blog/iot-development-and-wireless-communication/>>. [Accessed on 3rd April 2019].
- Texas Instrument 2013, *2.4 GHz IEEE 802.15.4 / Zigbee-ready RF Transceiver*. Available from: <<https://www.ti.com/lit/ds/symlink/cc2420.pdf>>. [Accessed on 3rd April 2019]
- Tuan, MC, Chen, SL, Lai, YK, Chen, CC, & Lee, HY 2017, *A 3-wire SPI Protocol Chip Design with Application-Specific Integrated Circuit (ASIC) and FPGA Verification*. Available from:

Bibliography

<https://avestia.com/EECSS2017_Proceedings/files/paper/EEE/EEE_110.pdf>. [Accessed on 1st April 2019].

UKEssays 2018, *Microprocessor Without Interlocked Pipeline Stages Computer Science Essay*. Available from: <<https://www.ukessays.com/essays/computer-science/microprocessor-without-interlocked-pipeline-stages-computer-science-essay.php?vref=1>>. [Accessed on 9th March 2019].

Appendix A: Timing Diagram

A.1 Timing diagram of different SPI's Transfer Modes

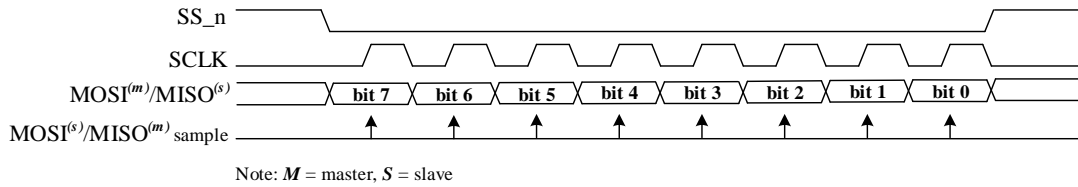


Figure A.1.1: Timing diagram for mode 0 serial data communication.

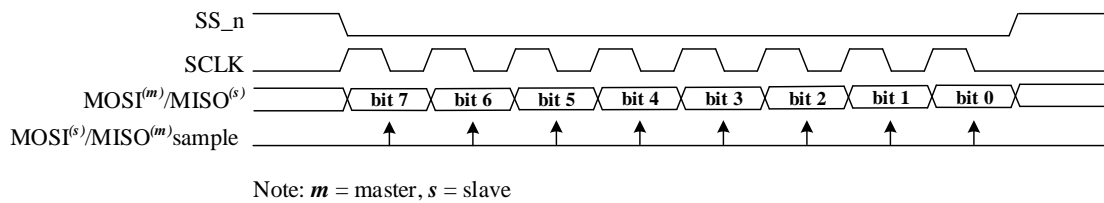


Figure A.1.2: Timing diagram for mode 1 serial data communication.

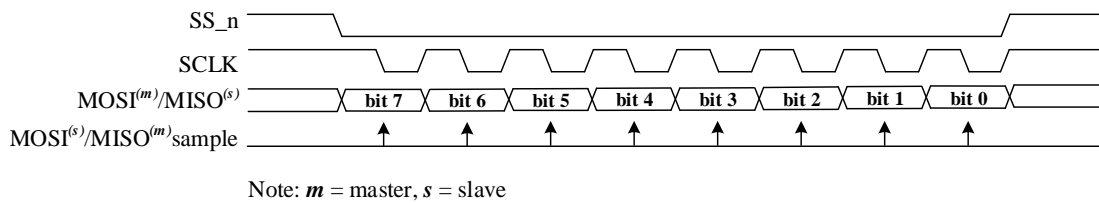


Figure A.1.3: Timing diagram for mode 2 serial data communication.

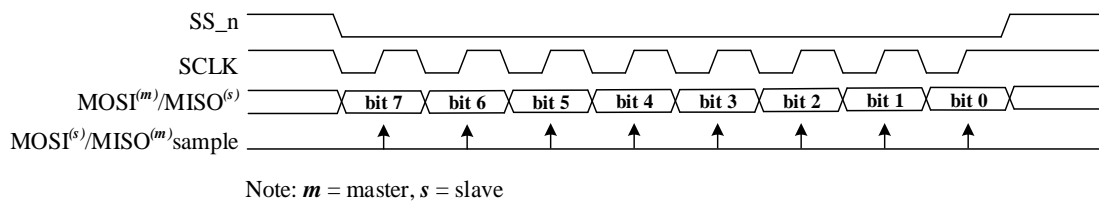


Figure A.1.4: Timing diagram for mode 3 serial data communication.

Appendix B: Testbench

B.1 Testbench for SPI Controller Unit's Functional Test

```

`default_nettype none//to catch typing errors due to misspelled of signal names

`ifdef MODEL_TECH
    `include "../util/macro.v"
`else
    `include "../util/macro.v"
`endif

//define WORD_NB    32 (defined in macro.v)
//define BYTE_NB    8 (defined in macro.v)

module tb_uspi_v3
();
//Declarations of the connections to the DUT_MASTER outputs
wire          tb_w_uiospi_MOSI;
wire          tb_w_uiospi_MISO;
wire          tb_w_uiospi_SCLK;
wire          tb_w_uiospi_SS_n;
wire          tb_w_uospi_IRQ_master;
wire [^WORD_NB - 1 : 0] tb_w_uospi_wb_r_dout_master;
wire          tb_w_uospi_wb_w_ack_master;
wire          tb_w_uospi_wb_r_ack_master;
//Declarations of the drivers to the DUT_MASTER inputs
reg           tb_r_uispi_SPIE_master;
reg           tb_r_uispi_pipe_stall_master;
reg [^BYTE_NB - 1 : 0] tb_r_uispi_wb_w_din_master;
reg [3:0]     tb_r_uispi_wb_w_sel_master;
reg           tb_r_uispi_wb_w_we_master;
reg           tb_r_uispi_wb_w_stb_master;
reg [3:0]     tb_r_uispi_wb_r_sel_master;
reg           tb_r_uispi_wb_r_we_master;
reg           tb_r_uispi_wb_r_stb_master;

//Declarations of the connections to the DUT_SLAVE outputs
wire          tb_w_uospi_IRQ_slave;
wire [^WORD_NB - 1 : 0] tb_w_uospi_wb_r_dout_slave;
wire          tb_w_uospi_wb_w_ack_slave;
wire          tb_w_uospi_wb_r_ack_slave;
//Declarations of the drivers to the DUT_SLAVE inputs
reg           tb_r_uispi_SPIE_slave;
reg           tb_r_uispi_pipe_stall_slave;
reg [^BYTE_NB - 1 : 0] tb_r_uispi_wb_w_din_slave;
reg [3:0]     tb_r_uispi_wb_w_sel_slave;
reg           tb_r_uispi_wb_w_we_slave;

```

```

reg                tb_r_uispi_wb_w_stb_slave;
reg [3:0]          tb_r_uispi_wb_r_sel_slave;
reg                tb_r_uispi_wb_r_we_slave;
reg                tb_r_uispi_wb_r_stb_slave;

//Declaration of the drivers to the sys and rst of both modules
reg tb_r_sys_clk;
reg tb_r_sys_rst;

//Module instantiation
uspi_v2
DUT_MASTER
(.uiospi_MOSI(tb_w_uiospi_MOSI),
.uiospi_MISO(tb_w_uiospi_MISO),
.uiospi_SCLK(tb_w_uiospi_SCLK),
.uiospi_SS_n(tb_w_uiospi_SS_n),
.uospi_IRQ(tb_w_uospi_IRQ_master),
.uospi_wb_r_dout(tb_w_uospi_wb_r_dout_master),
.uospi_wb_w_ack(tb_w_uospi_wb_w_ack_master),
.uospi_wb_r_ack(tb_w_uospi_wb_r_ack_master),
.uispi_SPIE(tb_r_uispi_SPIE_master),
.uispi_pipe_stall(tb_r_uispi_pipe_stall_master),
.uispi_wb_w_din(tb_r_uispi_wb_w_din_master),
.uispi_wb_w_sel(tb_r_uispi_wb_w_sel_master),
.uispi_wb_w_we(tb_r_uispi_wb_w_we_master),
.uispi_wb_w_stb(tb_r_uispi_wb_w_stb_master),
.uispi_wb_r_sel(tb_r_uispi_wb_r_sel_master),
.uispi_wb_r_we(tb_r_uispi_wb_r_we_master),
.uispi_wb_r_stb(tb_r_uispi_wb_r_stb_master),
.uispi_wb_clk(tb_r_sys_clk),
.uispi_wb_rst(tb_r_sys_rst));

uspi_v2
DUT_SLAVE
(.uiospi_MOSI(tb_w_uiospi_MOSI),
.uiospi_MISO(tb_w_uiospi_MISO),
.uiospi_SCLK(tb_w_uiospi_SCLK),
.uiospi_SS_n(tb_w_uiospi_SS_n),
.uospi_IRQ(tb_w_uospi_IRQ_slave),
.uospi_wb_r_dout(tb_w_uospi_wb_r_dout_slave),
.uospi_wb_w_ack(tb_w_uospi_wb_w_ack_slave),
.uospi_wb_r_ack(tb_w_uospi_wb_r_ack_slave),
.uispi_SPIE(tb_r_uispi_SPIE_slave),
.uispi_pipe_stall(tb_r_uispi_pipe_stall_slave),
.uispi_wb_w_din(tb_r_uispi_wb_w_din_slave),
.uispi_wb_w_sel(tb_r_uispi_wb_w_sel_slave),
.uispi_wb_w_we(tb_r_uispi_wb_w_we_slave),

```

```

.uispi_wb_w_stb(tb_r_uispi_wb_w_stb_slave),
.uispi_wb_r_sel(tb_r_uispi_wb_r_sel_slave),
.uispi_wb_r_we(tb_r_uispi_wb_r_we_slave),
.uispi_wb_r_stb(tb_r_uispi_wb_r_stb_slave),
.uispi_wb_clk(tb_r_sys_clk),
.uispi_wb_rst(tb_r_sys_rst));

//Clock waveform generation for DUT_MASTER
initial tb_r_sys_clk <= 1'b1;
always #10 tb_r_sys_clk = ~tb_r_sys_clk;

//Test pattern generation
initial begin
  //Sim time = 0
  //Signals initialization for DUT_MASTER
  tb_r_uispi_SPIE_master <= 1'b1;//enable global interrupt
  tb_r_uispi_pipe_stall_master <= 1'b0;//do not stall the master
  tb_r_uispi_wb_w_stb_master <= 1'b0;//disable the write access on master
  tb_r_uispi_wb_w_we_master <= 1'b0;//disable the write operation on master
  tb_r_uispi_wb_r_stb_master <= 1'b0;//disable the read access on master
  tb_r_uispi_wb_r_we_master <= 1'b1;//disable the read operation on master

  //Signals initialization for DUT_SLAVE
  tb_r_uispi_SPIE_slave <= 1'b1;//enable global interrupt
  tb_r_uispi_pipe_stall_slave <= 1'b0;//do not stall the slave
  tb_r_uispi_wb_w_stb_slave <= 1'b0;//disable the write access on slave
  tb_r_uispi_wb_w_we_slave <= 1'b0;//disable the write operation on slave
  tb_r_uispi_wb_r_stb_slave <= 1'b0;//disable the read access on slave
  tb_r_uispi_wb_r_we_slave <= 1'b1;//disable the read operation on slave

  #####
  //Test case 1: System reset
  repeat(5)@(posedge tb_r_sys_clk) tb_r_sys_rst <= 1'b0;
  repeat(5)@(posedge tb_r_sys_clk) tb_r_sys_rst <= 1'b1;
  repeat(10)@(posedge tb_r_sys_clk) tb_r_sys_rst <= 1'b0;

  #####
  //Test case 2: Write operation on SPISR
  tb_r_uispi_wb_w_stb_master <= 1'b1;//enable the write access on master
  tb_r_uispi_wb_w_we_master <= 1'b1;//enable the write operation on SPI master
  tb_r_uispi_wb_w_sel_master <= 4'b0010;//enable write operation on SPISR
  tb_r_uispi_wb_w_din_master <= 8'b0000_1111;//8'h0F
  tb_r_uispi_wb_w_stb_slave <= 1'b1;//enable write access on slave
  tb_r_uispi_wb_w_we_slave <= 1'b1;//enable write operation on slave
  tb_r_uispi_wb_w_sel_slave <= 4'b0010;//enable write operation on SPISR
  tb_r_uispi_wb_w_din_slave <= 8'b0000_1111;//8'h0F

  @(posedge tb_r_sys_clk);

```

```

tb_r_uispi_wb_w_we_master <= 1'b0;//disable the write operation on master
tb_r_uispi_wb_w_we_slave <= 1'b0;//disable the write operation on slave
repeat(5)@(posedge tb_r_sys_clk);

#####
//Test case 3: Write operation on SPICR
//Mode 0 is selected
tb_r_uispi_wb_w_we_master <= 1'b1;//enable the write operation on master
tb_r_uispi_wb_w_sel_master <= 4'b0001;//enable write operation on SPICR
tb_r_uispi_wb_w_din_master <= 8'b1100_0000;//8'hC0

//Mode 0 is selected
tb_r_uispi_wb_w_we_slave <= 1'b1;//enable write operation on slave
tb_r_uispi_wb_w_sel_slave <= 4'b0001;//enable write operation on SPICR
tb_r_uispi_wb_w_din_slave <= 8'b1000_0000;//8'h80

@(posedge tb_r_sys_clk);
tb_r_uispi_wb_w_we_master <= 1'b0;//disable write operation on master
tb_r_uispi_wb_w_we_slave <= 1'b0;//disable write operation on slave

@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_stb_master <= 1'b1;//enable the read access on master
tb_r_uispi_wb_r_we_master <= 1'b0;//enable the read operation on master
tb_r_uispi_wb_r_sel_master <= 4'b0011;//read SPISR and SPICR content
tb_r_uispi_wb_r_stb_slave <= 1'b1;//enable the read access on slave
tb_r_uispi_wb_r_we_slave <= 1'b0;//enable the read operation on slave
tb_r_uispi_wb_r_sel_slave <= 4'b0011;//read SPISR and SPICR content

#####
//Test case 4: Transmitter buffer empty interrupt support
repeat(20)@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_we_master <= 1'b1;//disable the read operation on master
tb_r_uispi_wb_r_we_slave <= 1'b1;//disable the read operation on slave

#####
//Test case 5: Push one 8-bit data into the bFIFO_FIFOreg of the TX_buffer16x8
//Load data 8'b1010_1010 into the TX_buffer16x8 of the DUT_MASTER (master)
tb_r_uispi_wb_w_we_master <= 1'b1;//enable the write operation on master
tb_r_uispi_wb_w_sel_master <= 4'b0100;//enable the write operation on
//TX_buffer16x8
tb_r_uispi_wb_w_din_master <= 8'b1010_1010;//8'hAA

//Load data 8'b0101_0101 into the TX_buffer16x8 of the DUT_SLAVE (slave)
tb_r_uispi_wb_w_we_slave <= 1'b1;//enable the write operation on slave
tb_r_uispi_wb_w_sel_slave <= 4'b0100;//enable the write operation on
//TX_buffer16x8
tb_r_uispi_wb_w_din_slave <= 8'b0101_0101;//8'h55

```



```

@(posedge tb_r_sys_clk);
tb_r_uispi_wb_w_we_master <= 1'b0;//disable the write operation on master
tb_r_uispi_wb_w_we_slave <= 1'b0;//disable the write operation on slave

@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_we_master <= 1'b0;//enable the read operation on master
tb_r_uispi_wb_r_sel_master <= 4'b0010;//read SPISR content
tb_r_uispi_wb_r_we_slave <= 1'b0;//enable the read operation slave
tb_r_uispi_wb_r_sel_slave <= 4'b0010;//read SPISR content

repeat(20)@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_we_master <= 1'b1;//disable the read operation on master
tb_r_uispi_wb_r_we_slave <= 1'b1;//disable the read operation on master

#####
//Test case 6: Mode 0 serial data communication
//Load data 8'b1010_1010 into bFIFO TX_buffer16x8 of the DUT_MASTER
(master)
tb_r_uispi_wb_w_we_master <= 1'b1;//enable the write operation on master
tb_r_uispi_wb_w_sel_master <= 4'b0100;//enable the write operation on
//TX_buffer16x8
tb_r_uispi_wb_w_din_master <= 8'b1010_1010;//8'hAA
//Load data 8'b0101_0101 into bFIFO TX_buffer16x8 of the DUT_SLAVE (slave)
tb_r_uispi_wb_w_we_slave <= 1'b1;//enable the write operation on slave
tb_r_uispi_wb_w_sel_slave <= 4'b0100;//enable the write operation on
//TX_buffer16x8
tb_r_uispi_wb_w_din_slave <= 8'b0101_0101;//8'h55
@(posedge tb_r_sys_clk);
tb_r_uispi_wb_w_we_master <= 1'b0;//disable the write operation on master
tb_r_uispi_wb_w_we_slave <= 1'b0;//disable the write operation on slave
repeat(100)@(posedge tb_r_sys_clk);

#####
//Test case 7: Received buffer full interrupt support after receiving a 1-byte data
//(RXFM = 0)
repeat(15)@(posedge tb_r_sys_clk);

#####
//Test case 8: Pop 1-byte of received data from the RX_buffer16x8
tb_r_uispi_wb_r_we_master <= 1'b0;//enable the read operation on master
tb_r_uispi_wb_r_sel_master <= 4'b1000;//pop 1-byte of data from the
//RX_buffer16x8
tb_r_uispi_wb_r_we_slave <= 1'b0;//enable the read operation on slave
tb_r_uispi_wb_r_sel_slave <= 4'b1000;//pop 1-byte of data from the RX_buffer16x8
@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_sel_master <= 4'b0011;//read SPISR and SPICR content
tb_r_uispi_wb_r_sel_slave <= 4'b0011;//read SPISR and SPICR content

```

```

repeat(5)@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_we_master <= 1'b1;//disable the read operation on master
tb_r_uispi_wb_r_we_slave <= 1'b1;//disable the read operation on slave

#####
//Test case 9: Received buffer full interrupt support after receiving 16 x 1-byte data
//(RXFM = 1)
//Configure the SPICR first to de-activate/stop the data communication
tb_r_uispi_wb_w_we_master <= 1'b1;//enable the write operation on master
tb_r_uispi_wb_w_sel_master <= 4'b0001;//enable the write operation on SPICR
tb_r_uispi_wb_w_din_master <= 8'b0100_0000;//8'h40
//Configure the SPICR first to de-activate/stop the data communication
tb_r_uispi_wb_w_we_slave <= 1'b1;//enable the write operation on master
tb_r_uispi_wb_w_sel_slave <= 4'b0001;//enable the write operation on SPICR
tb_r_uispi_wb_w_din_slave <= 8'b0000_0000;//8'h00

@(posedge tb_r_sys_clk);
//Configure the SPISR
tb_r_uispi_wb_w_sel_master <= 4'b0010;//enable the write operation on SPISR
tb_r_uispi_wb_w_sel_slave <= 4'b0010;//enable the write operation on SPISR
tb_r_uispi_wb_w_din_master <= 8'b0001_1111;//8'h1F
tb_r_uispi_wb_w_din_slave <= 8'b0001_1111;//8'h1F

//Load 16x1-byte data into bFIFO TX_buffer16x8 of the DUT_MASTER (master) for
//serial 1-byte data transmission
//to another SPI device (slave)
@(posedge tb_r_sys_clk);
tb_r_uispi_wb_w_din_master <= 8'b1010_1010;//8'hAA
tb_r_uispi_wb_w_sel_master <= 4'b0100;//enable the write operation on
//TX_buffer16x8
tb_r_uispi_wb_w_din_slave <= 8'b0101_0101;//8'h55
tb_r_uispi_wb_w_sel_slave <= 4'b0100;//enable the write operation on
TX_buffer16x8
repeat(16) begin
  @(posedge tb_r_sys_clk);
  tb_r_uispi_wb_w_din_master <= ~tb_r_uispi_wb_w_din_master;
  tb_r_uispi_wb_w_din_slave <= ~tb_r_uispi_wb_w_din_slave;
end

@(posedge tb_r_sys_clk);
//Configure the SPICR again to activate the master and the slave
tb_r_uispi_wb_w_sel_master <= 4'b0001;//enable the write operation on SPICR
tb_r_uispi_wb_w_din_master <= 8'b1100_0000;//8'hC0
tb_r_uispi_wb_w_sel_slave <= 4'b0001;//enable the write operation on SPICR
tb_r_uispi_wb_w_din_slave <= 8'b1000_0000;//8'h80

@(posedge tb_r_sys_clk);

```

```

tb_r_uispi_wb_w_we_master <= 1'b0;//disable the write operation on master
tb_r_uispi_wb_w_we_slave <= 1'b0;//disable the write operation on slave

@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_we_master <= 1'b0;//enable the read operation on master
tb_r_uispi_wb_r_sel_master <= 4'b0011;//read SPISR and SPICR content
tb_r_uispi_wb_r_we_slave <= 1'b0;//enable the read operation on slave
tb_r_uispi_wb_r_sel_slave <= 4'b0011;//read SPISR and SPICR content

//data communication between the master and the slave begins
//transmit 16x1-byte of data
//receive 16x1-byte of data
repeat(1640)@(posedge tb_r_sys_clk);
tb_r_uispi_wb_w_we_master <= 1'b1;//enable the write operation on master
tb_r_uispi_wb_w_sel_master <= 4'b0100;//enable the write operation on
//TX_buffer16x8
tb_r_uispi_wb_w_din_master <= 8'b1001_1010;//8'h9A
tb_r_uispi_wb_w_we_slave <= 1'b1;//enable the write operation on slave
tb_r_uispi_wb_w_sel_slave <= 4'b0100;//enable the write the operation on
//TX_buffer16x8
tb_r_uispi_wb_w_din_slave <= 8'b1010_1001;//8'hA9
@(posedge tb_r_sys_clk);
tb_r_uispi_wb_w_we_master <= 1'b0;//disable the write operation on master
tb_r_uispi_wb_w_we_slave <= 1'b0;//disable the write operation on slave
repeat(20)@(posedge tb_r_sys_clk);

#####
//Test case 10: Pop 16 number of 1-byte data from the RX_buffer16x8
tb_r_uispi_wb_r_we_master <= 1'b0;//enable the read operation on master
tb_r_uispi_wb_r_sel_master <= 4'b1000;//read RX_buffer16x8 content
tb_r_uispi_wb_r_we_slave <= 1'b0;//enable the read operation on slave
tb_r_uispi_wb_r_sel_slave <= 4'b1000;//read RX_buffer16x8 content
repeat(18)@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_we_master <= 1'b1;//disable the read operation on master
tb_r_uispi_wb_r_we_slave <= 1'b1;//disable the read operation on slave

#####
//Test case 11: Mode 1 serial data communication
tb_r_uispi_wb_w_we_master <= 1'b1;//enable the write operation on master
//Configure the SPICR first to de-activate/stop the data communication
tb_r_uispi_wb_w_sel_master <= 4'b0001;//enable the write operation on master
tb_r_uispi_wb_w_din_master <= 8'b0101_0000;//8'h50
tb_r_uispi_wb_w_we_slave <= 1'b1;//enable the write operation on slave
tb_r_uispi_wb_w_sel_slave <= 4'b0001;//enable the write operation on slave
tb_r_uispi_wb_w_din_slave <= 8'b0001_0000;//8'h10

@(posedge tb_r_sys_clk);
//Configure the SPISR

```

```

repeat(30)@(posedge tb_r_sys_clk);
tb_r_uispi_wb_w_we_master <= 1'b0;//disable the write operation on master
tb_r_uispi_wb_w_we_slave <= 1'b0;//disable the write operation on slave
@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_we_master <= 1'b0;//enable the read operation on master
tb_r_uispi_wb_r_sel_master <= 4'b1000;//read RX_buffer16x8 content
tb_r_uispi_wb_r_we_slave <= 1'b0;//enable the read operation on slave
tb_r_uispi_wb_r_sel_slave <= 4'b1000;//read RX_buffer16x8 content

@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_sel_master <= 4'b0011;//read SPISR and SPICR content
tb_r_uispi_wb_r_sel_slave <= 4'b0011;//read SPISR and SPICR content

//data communication between the master and the slave are happening
//transmit 1-byte of data
//receive 1-byte of data
repeat(450)@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_we_master <= 1'b1;//de-activate the read enable signal
tb_r_uispi_wb_r_we_slave <= 1'b1;//de-activate the read enable signal
tb_r_uispi_wb_w_we_master <= 1'b1;//enable the write operation on master
tb_r_uispi_wb_w_we_slave <= 1'b1;//enable the write operation on slave

//To disable the transmit buffer empty and received buffer full interrupt
tb_r_uispi_wb_w_sel_master <= 4'b0010;//enable the write operation on SPISR
tb_r_uispi_wb_w_din_master <= 8'b0000_0011;
tb_r_uispi_wb_w_sel_slave <= 4'b0010;//enable the write operation on SPISR
tb_r_uispi_wb_w_din_slave <= 8'b0000_0011;
repeat(5)@(posedge tb_r_sys_clk);

tb_r_uispi_wb_w_sel_master <= 4'b0001;//enable the write operation on SPICR
tb_r_uispi_wb_w_din_master <= 8'b1111_0011;
tb_r_uispi_wb_w_sel_slave <= 4'b0001;//enable the write operation on SPICR
tb_r_uispi_wb_w_din_slave <= 8'b1011_0011;

// Pop 1-byte of data from the RX_buffer16x8
repeat(30)@(posedge tb_r_sys_clk);
tb_r_uispi_wb_w_we_master <= 1'b0;//disable the write operation on master
tb_r_uispi_wb_w_we_slave <= 1'b0;//disable the write operation on slave

@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_we_master <= 1'b0;//enable the read operation on master
tb_r_uispi_wb_r_sel_master <= 4'b1000;//read RX_buffer16x8 content
tb_r_uispi_wb_r_we_slave <= 1'b0;//enable the read operation on slave
tb_r_uispi_wb_r_sel_slave <= 4'b1000;//read RX_buffer16x8 content

@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_sel_master <= 4'b0011;//read SPISR and SPICR content

```

```

tb_r_uispi_wb_r_sel_slave <= 4'b0011;//read SPISR and SPICR content
//data communication between the master and the slave are happening
//transmit 1-byte of data
//receive 1-byte of data
repeat(660)@(posedge tb_r_sys_clk);

@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_we_master <= 1'b0;//enable the read operation on master
tb_r_uispi_wb_r_sel_master <= 4'b0011;//read SPISR and SPICR content
tb_r_uispi_wb_r_we_slave <= 1'b0;//enable the read operation on slave
tb_r_uispi_wb_r_sel_slave <= 4'b0011;//read SPISR and SPICR content
repeat(15)@(posedge tb_r_sys_clk);

#####
//Test case 15: Mode fault error interrupt
//Mode fault error occurs when more than one master are trying to drive the shared
//line
//Firstly, reconfigure the slave device to act as a master
tb_r_uispi_wb_w_we_slave <= 1'b1;//enable the write operation on slave
tb_r_uispi_wb_w_sel_slave <= 4'b0001;//enable the write operation on SPICR
tb_r_uispi_wb_w_din_slave <= 8'b1111_0011;

repeat(10)@(posedge tb_r_sys_clk);
tb_r_uispi_wb_r_we_slave <= 1'b0;//enable the read operation on the newly
//configured master
tb_r_uispi_wb_r_sel_slave <= 4'b1000;//read the RX_buffer16x8 content
repeat(120)@(posedge tb_r_sys_clk);

//Give output some time to settle down
repeat(120)@(posedge tb_r_sys_clk);
//To stop stimulation
$stop;
end
endmodule

```

B.2 Testbench for SPI Controller Unit's Integration Test with RISC32

```

`timescale 1ns / 1ps
`default_nettype none
`define demo005_SPI 1
`ifdef demo005_SPI
    `define TEST_CODE_PATH_DUT "demo005_SPI_mem_03program.txt"
    `define EXC_HANDLER_DUT "new_exc_handler_dut_2.txt"
    `define TEST_CODE_PATH_CLIENT
"demo101_pending_for_int_mem_03program.txt"
    `define EXC_HANDLER_CLIENT "new_exc_handler_dut_2.txt"
`endif

module tb_r32_pipeline();
reg        tb_u_clk;
reg        tb_u_rst;
wire       tb_u_spi_mosi;
wire       tb_u_spi_miso;
wire       tb_u_spi_sclk;
wire       tb_u_spi_ss_n;

wire       tb_u_fc_sclk_dut;
wire       tb_u_fc_ss_dut;
wire       tb_u_fc_MOSI_dut;
wire       tb_u_fc_MISO1_dut;
wire       tb_u_fc_MISO2_dut;
wire       tb_u_fc_MISO3_dut;
wire       tb_ua_tx_rx_dut;
wire       tb_ua_RTS_dut, tb_ua_CTS_dut;
wire [31:0] tb_u_GPIO_dut;

wire       tb_u_fc_sclk_client;
wire       tb_u_fc_ss_client;
wire       tb_u_fc_MOSI_client;
wire       tb_u_fc_MISO1_client;
wire       tb_u_fc_MISO2_client;
wire       tb_u_fc_MISO3_client;
wire       tb_ua_tx_rx_client;
wire       tb_ua_RTS_client, tb_ua_CTS_client;
wire [31:0] tb_u_GPIO_client;

//***** INSTANTIATION *****/
crisc c_risc_dut(
//===== INPUT =====
//GPIO
.urisc_GPIO(tb_u_GPIO_dut),
//SPI controller
.uiorisc_spi_mosi(tb_u_spi_mosi),

```

```

.uiorisc_spi_miso(tb_u_spi_miso),
.uiorisc_spi_sclk(tb_u_spi_sclk),
.uiorisc_spi_ss_n(tb_u_spi_ss_n),

//UART controller
.uorisc_ua_tx_data(tb_ua_tx_rx_dut),
.uirisc_ua_rx_data(tb_ua_tx_rx_client),

//FLASH controller
.uorisc_fc_sclk(tb_u_fc_sclk_dut),
.uiorisc_fc_MOSI(tb_u_fc_MOSI_dut),
.uirisc_fc_MISO1(tb_u_fc_MISO1_dut),
.uirisc_fc_MISO2(tb_u_fc_MISO2_dut),
.uirisc_fc_MISO3(tb_u_fc_MISO3_dut),
.uorisc_fc_ss(tb_u_fc_ss_dut),

// System signal
.uirisc_clk_100mhz(tb_u_clk),
.uirisc_rst(tb_u_rst));

s25fl128s SPI_flash_dut(
.SI(tb_u_fc_MOSI_dut), //IO0
.SO(tb_u_fc_MISO1_dut), //IO1
.SCK(tb_u_fc_sclk_dut),
.CSNeg(tb_u_fc_ss_dut),
.RSTNeg(tb_u_rst),
.WPNeg(tb_u_fc_MISO2_dut), //IO2
.HOLDNeg(tb_u_fc_MISO3_dut));

crisc c_risc_client(
//===== INPUT =====
//GPIO
.uirisc_GPIO(tb_u_GPIO_client),
//SPI controller
.uiorisc_spi_mosi(tb_u_spi_mosi),
.uiorisc_spi_miso(tb_u_spi_miso),
.uiorisc_spi_sclk(tb_u_spi_sclk),
.uiorisc_spi_ss_n(tb_u_spi_ss_n),

//UART controller
.uorisc_ua_tx_data(tb_ua_tx_rx_client),
.uirisc_ua_rx_data(tb_ua_tx_rx_dut),

//FLASH controller
.uorisc_fc_sclk(tb_u_fc_sclk_client),
.uiorisc_fc_MOSI(tb_u_fc_MOSI_client),

```

```

.uirisc_fc_MISO1(tb_u_fc_MISO1_client),
.uirisc_fc_MISO2(tb_u_fc_MISO2_client),
.uirisc_fc_MISO3(tb_u_fc_MISO3_client),
.uorisc_fc_ss(tb_u_fc_ss_client),

// System signal
.uirisc_clk_100mhz(tb_u_clk),
.uirisc_rst(tb_u_rst));

s25fl128s SPI_flash_client(
.SI(tb_u_fc_MOSI_client), //IO0
.SO(tb_u_fc_MISO1_client), //IO1
.SCK(tb_u_fc_sclk_client),
.CSNeg(tb_u_fc_ss_client),
.RSTNeg(tb_u_rst),
.WPNeg(tb_u_fc_MISO2_client), //IO2
.HOLDNeg(tb_u_fc_MISO3_client));

assign tb_ua_CTS_dut = tb_ua_RTS_client;
assign tb_ua_CTS_client = tb_ua_RTS_dut;

//*****Clock*****
initial tb_u_clk = 1'b1;
always #5 tb_u_clk =~ tb_u_clk;
initial begin
$readmemh(`EXC_HANDLER_CLIENT, tb_r32_pipeline.SPI_flash_client.Mem);
$readmemh(`TEST_CODE_PATH_CLIENT, tb_r32_pipeline.SPI_flash_client.Mem);
$readmemh(`EXC_HANDLER_DUT, tb_r32_pipeline.SPI_flash_dut.Mem);
$readmemh(`TEST_CODE_PATH_DUT, tb_r32_pipeline.SPI_flash_dut.Mem);
tb_u_rst = 1'b1;
repeat(1)@(posedge tb_u_clk);
tb_u_rst = 1'b0;
repeat(30000)@(posedge tb_u_clk);
tb_u_rst = 1'b1;
repeat(1200000)@(posedge tb_r32_pipeline.c_risc_dut.urisc_clk);
end
endmodule

```


Appendix C: CC2420

C.1 Pin Assignment of the CC2420

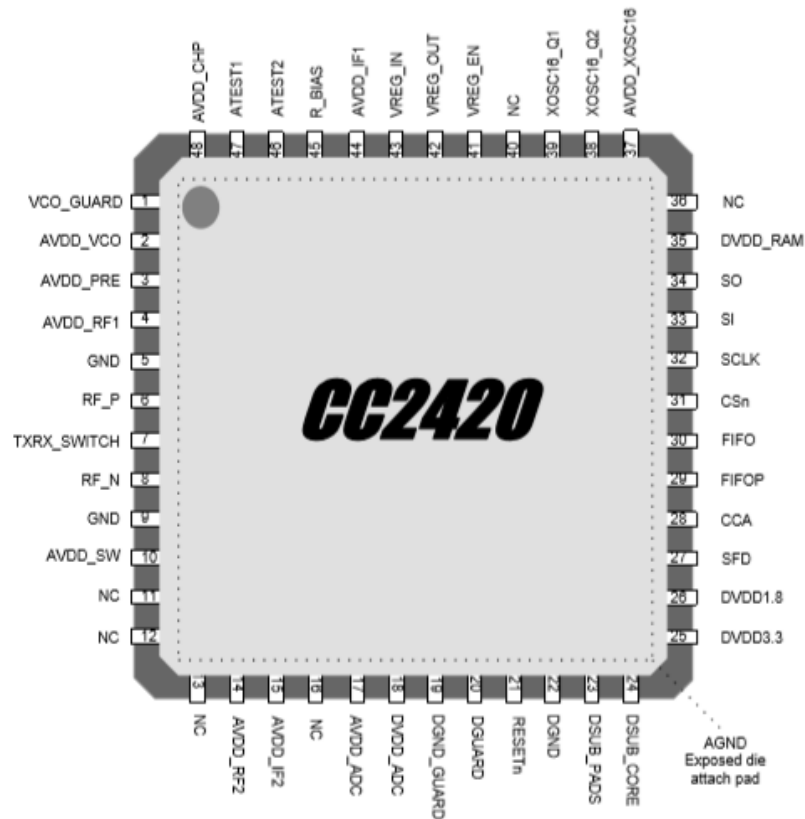


Figure C.1.1: Top view of the CC2420 pinout

Pin	Pin Name	Pin type	Pin Description
-	AGND	Ground (analog)	Exposed die attach pad. Must be connected to solid ground plane
1	VCO_GUARD	Power (analog)	Connection of guard ring for VCO (to AVDD) shielding
2	AVDD_VCO	Power (analog)	1.8 V Power supply for VCO
3	AVDD_PRE	Power (analog)	1.8 V Power supply for Prescaler
4	AVDD_RF1	Power (analog)	1.8 V Power supply for RF front-end
5	GND	Ground (analog)	Grounded pin for RF shielding
6	RF_P	RF I/O	Positive RF input/output signal to LNA/from PA in receive/transmit mode
7	TXRX_SWITCH	Power (analog)	Common supply connection for integrated RF front-end. Must be connected to RF_P and RF_N externally through a DC path
8	RF_N	RF I/O	Negative RF input/output signal to LNA/from PA in receive/transmit mode
9	GND	Ground (analog)	Grounded pin for RF shielding
10	AVDD_SW	Power (analog)	1.8 V Power supply for LNA / PA switch
11	NC	-	Not Connected
12	NC	-	Not Connected
13	NC	-	Not Connected
14	AVDD_RF2	Power (analog)	1.8 V Power supply for receive and transmit mixers

Figure C.1.2: Pin description of the CC2420.

Appendix

Pin	Pin Name	Pin type	Pin Description
15	AVDD_IF2	Power (analog)	1.8 V Power supply for transmit / receive IF chain
16	NC	-	Not Connected
17	AVDD_ADC	Power (analog)	1.8 V Power supply for analog parts of ADCs and DACs
18	DVDD_ADC	Power (digital)	1.8 V Power supply for digital parts of receive ADCs
19	DGND_GUARD	Ground (digital)	Ground connection for digital noise isolation
20	DGUARD	Power (digital)	1.8 V Power supply connection for digital noise isolation
21	RESETn	Digital Input	Asynchronous, active low digital reset
22	DGND	Ground (digital)	Ground connection for digital core and pads
23	DSUB_PADS	Ground (digital)	Substrate connection for digital pads
24	DSUB_CORE	Ground (digital)	Substrate connection for digital modules
25	DVDD3.3	Power (digital)	3.3 V Power supply for digital I/Os
26	DVDD1.8	Power (digital)	1.8 V Power supply for digital core
27	SFD	Digital output	SFD (Start of Frame Delimiter) / digital mux output
28	CCA	Digital output	CCA (Clear Channel Assessment) / digital mux output
29	FIFOP	Digital output	Active when number of bytes in FIFO exceeds threshold / serial RF clock output in test mode
30	FIFO	Digital I/O	Active when data in FIFO / serial RF data input / output in test mode
31	CSn	Digital input	SPI Chip select, active low
32	SCLK	Digital input	SPI Clock input, up to 10 MHz
33	SI	Digital input	SPI Slave Input. Sampled on the positive edge of SCLK
34	SO	Digital output (tristate)	SPI Slave Output. Updated on the negative edge of SCLK. Tristate when CSn high.
35	DVDD_RAM	Power (digital)	1.8 V Power supply for digital RAM
36	NC	-	Not Connected
37	AVDD_XOSC16	Power (analog)	1.8 V crystal oscillator power supply
38	XOSC16_Q2	Analog I/O	16 MHz Crystal oscillator pin 2
39	XOSC16_Q1	Analog I/O	16 MHz Crystal oscillator pin 1 or external clock input
40	NC	-	Not Connected
41	VREG_EN	Digital input	Voltage regulator enable, active high, held at VREG_IN voltage level when active. Note that VREG_EN is relative VREG_IN, not DVDD3.3.
42	VREG_OUT	Power output	Voltage regulator 1.8 V power supply output
43	VREG_IN	Power (analog)	Voltage regulator 2.1 to 3.6 V power supply input
44	AVDD_IF1	Power (analog)	1.8 V Power supply for transmit / receive IF chain
45	R_BIAS	Analog output	External precision resistor, 43 k Ω , \pm 1 %
46	ATEST2	Analog I/O	Analog test I/O for prototype and production testing
47	ATEST1	Analog I/O	Analog test I/O for prototype and production testing
48	AVDD_CHP	Power (analog)	1.8 V Power supply for phase detector and charge pump

Figure C.1.3: Pin description of the CC2420 (cont'd).

Note: *The exposed die attach pad must be connected to a solid ground plane as this is the main ground connection for the chip.*

C.2 Overview of the External Components Used with the CC2420


Ref	Description
C42	Voltage regulator load capacitance
C61	Balun and match
C62	DC block to antenna and match
C71	Front-end bias decoupling and match
C81	Balun and match
C381	16MHz crystal load capacitor, see page 53
C391	16MHz crystal load capacitor, see page 53
L61	DC bias and match
L62	DC bias and match
L71	DC bias and match
L81	Balun and match
R451	Precision resistor for current reference generator
XTAL	16MHz crystal, see page 53

Figure C.2.1: Description of the external components used with the CC2420.

C.3 List of Materials for the Application Circuits

Item	Single ended output, transmission line balun	Single ended output, discrete balun	Differential antenna
C42	10 μ F, $0.5\Omega < \text{ESR} < 5\Omega$	10 μ F, $0.5\Omega < \text{ESR} < 5\Omega$	10 μ F, $0.5\Omega < \text{ESR} < 5\Omega$
C61	Not used	0.5 pF, +/- 0.25pF, NP0, 0402	Not used
C62	Not used	5.6 pF, +/- 0.25pF, NP0, 0402	Not used
C71	Not used	5.6 pF, 10%, X5R, 0402	Not used
C81	5.6 pF, +/- 0.25pF, NP0, 0402	0.5 pF, +/- 0.25pF, NP0, 0402	Not used
C381	27 pF, 5%, NP0, 0402	27 pF, 5%, NP0, 0402	27 pF, 5%, NP0, 0402
C391	27 pF, 5%, NP0, 0402	27 pF, 5%, NP0, 0402	27 pF, 5%, NP0, 0402
L61	8.2 nH, 5%, Monolithic/multilayer, 0402	7.5 nH, 5%, Monolithic/multilayer, 0402	27 nH, 5%, Monolithic/multilayer, 0402
L62	Not used	5.6 nH, 5%, Monolithic/multilayer, 0402	Not used
L71	22 nH, 5%, Monolithic/multilayer, 0402	Not used	12 nH, 5%, Monolithic/multilayer, 0402
L81	1.8 nH, +/- 0.3nH, Monolithic/multilayer, 0402	7.5 nH, 5%, Monolithic/multilayer, 0402	Not used
R451	43 k Ω , 1%, 0402	43 k Ω , 1%, 0402	43 k Ω , 1%, 0402
XTAL	16 MHz crystal, 16 pF load (C_L), ESR < 60 Ω	16 MHz crystal, 16 pF load (C_L), ESR < 60 Ω	16 MHz crystal, 16 pF load (C_L), ESR < 60 Ω

Figure C.3.1: List of materials for the application circuit.



UNIVERSITI TUNKU ABDUL RAHMAN
FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

DESIGN AND IMPLEMENTATION OF A SPI CONTROLLER FOR ZIGBEE MODULE

Prepared by Yong Min An
 Supervised by Mr Mok Kai Ming

INTRODUCTION

In recent years, the wireless sensor networks (WSNs) have grown considerably and they have a potential in different applications, including health, environment, military and so on. Zigbee module is one of the most widely used transceiver for wireless communication because of its advantages such as low cost, low power consumption, and low data rate. In this project, the design and implementation of the 4-wire SPI controller unit for Zigbee module are presented by using Verilog HDL as using full-duplex SPI is suitable for those applications that involve the transfer of data stream.

METHOD

Top-down design methodology is used in the design process for digital system.

uArch Specification (Partitioning)

↓

Unit/Block Level Spec, Modeling, & Verification

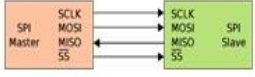
↓

Logic Synthesis in FPGA

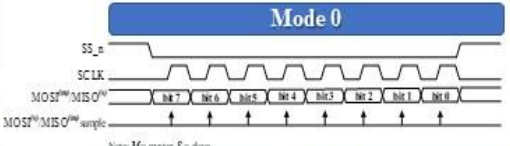
↓

Physical Design & Implementation

RESULTS

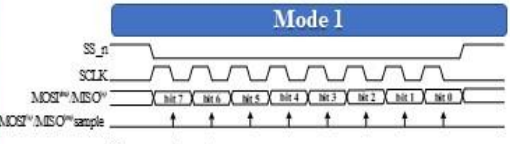


Mode 0



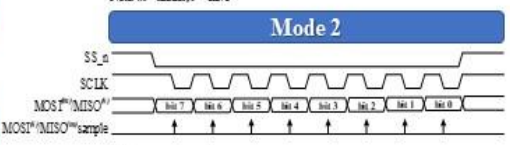
Note: M = master, S = slave

Mode 1



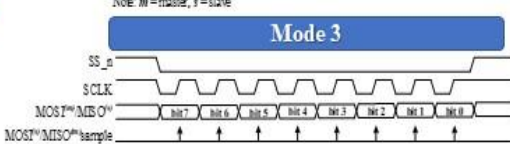
Note: m = master, s = slave

Mode 2



Note: m = master, s = slave

Mode 3




Note: m = master, s = slave

DISCUSSION

Features:

- Support full-duplex synchronous serial data transfer in all 4 transfer modes.
- Provide 16 selectable serial clock frequency/baud rate.
- Have separate 16-deep FIFOs to reduce CPU's workload to move data.
- Provide 3 types of interrupts (RXF, TXEF, and MODF).
- Fully synthesizable in FPGA technology.



CONCLUSIONS

- The SPI controller unit's design has been revised, functionally verified and further enhanced.
- It can work well with the integrated RISC32 pipeline processor across the CDC boundaries.
- It is synthesizable and has minimum timing impact on the integrated processor.

Plagiarism Check Result

DESIGN AND IMPLEMENTATION OF A SPI CONTROLLER FOR ZIGBEE MODULE

ORIGINALITY REPORT

7 %	6 %	3 %	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	eprints.utar.edu.my Internet Source	3 %
2	users.ece.utexas.edu Internet Source	<1 %
3	Dwaraka N Oruganti, Siva S Yellampalli. "Design of a power efficient SPI interface", 2014 International Conference on Advances in Electronics Computers and Communications, 2014 Publication	<1 %
4	www.melexis.com Internet Source	<1 %
5	www.rfworld.org Internet Source	<1 %
6	Anand N, , George Joseph, Suwin Sam Oommen, and R Dhanabal. "Design and implementation of a high speed Serial Peripheral Interface", 2014 International	<1 %

Conference on Advances in Electrical
Engineering (ICAEE), 2014.

Publication

7	sites.google.com Internet Source	<1%
8	www.farnell.com Internet Source	<1%
9	cache.freescale.com Internet Source	<1%
10	cegt201.bradley.edu Internet Source	<1%
11	avestia.com Internet Source	<1%
12	www.ti.com Internet Source	<1%
13	Wei-Pau Kiat, Kai-Ming Mok, Wai-Kong Lee, Hock-Guan Goh, Ramachandra Achar. "An energy efficient FPGA partial reconfiguration based micro-architectural technique for IoT applications", Microprocessors and Microsystems, 2020 Publication	<1%
14	www.flexhdr.org Internet Source	<1%
15	Bertozzi, Davide, Alessandro Strano, Daniele	

	Ludovici, Vasileios Pavlidis, Federico Angiolini, and Milos Krstic. "The Synchronization Challenge", Chapman & Hall/CRC Computational Science, 2010.	<1%
	Publication	
16	Zhaoxiang Zong. "Pin multiplexing optimization in FPGA prototyping system", 2017 4th International Conference on Systems and Informatics (ICSAI), 2017	<1%
	Publication	
17	www.ijert.org	<1%
	Internet Source	
18	pt.scribd.com	<1%
	Internet Source	
19	fongelectronics.blogspot.com	<1%
	Internet Source	
20	"Microcontrollers in Practice", Springer Science and Business Media LLC, 2005	<1%
	Publication	
21	www.ember.net	<1%
	Internet Source	
22	www.slideshare.net	<1%
	Internet Source	
23	P. Rajashekar Reddy, P. Sreekanth, K. Arun Kumar. "Serial Peripheral Interface-Master	<1%

Universal Verification Component using UVM",
International Journal of Advanced Scientific
Technologies in Engineering and Management
Sciences, 2017

Publication

-
- | | | |
|----|---|-----|
| 24 | Heiser, Jay, Steve Stanek, Sasan Hamidi, Ben Rothke, Paul Lambert, Ralph Spencer Poore, James Tiller, Ronald Gove, and Mark Edmead. "Methods of Attacking and Defending Cryptosystems", Information Security Management Handbook on CD-ROM 2006 Edition, 2006.
Publication | <1% |
| 25 | hurtle.land
Internet Source | <1% |
| 26 | www.bartleby.com
Internet Source | <1% |
| 27 | www.elprocus.com
Internet Source | <1% |
| 28 | W. J. Buchanan. "The Handbook of Data Communications and Networks", Springer Nature, 2004
Publication | <1% |
| 29 | xplorestaging.ieee.org
Internet Source | <1% |
| 30 | "IEEE-ICDCS conference proceeding", 2012 | |

	International Conference on Devices Circuits and Systems (ICDCS), 03/2012 Publication	<1%
31	www.embedded.com Internet Source	<1%
32	www.eecs.berkeley.edu Internet Source	<1%
33	Oudjida, A.K., M.L. Berrandjia, A. Liacha, R. Tiar, K. Tahraoui, and Y.N. Alhoumays. "Design and test of general-purpose SPI Master/Slave IPs on OPB bus", 2010 7th International Multi-Conference on Systems Signals and Devices, 2010. Publication	<1%
34	docplayer.net Internet Source	<1%
35	Zhaoqing Wang, Harry H. Cheng, Stephen S. Nestinger, Benjamin D. Shaw, Joe Palen. "Real-Time Architecture for an Electro-Mech-Optical System for Detection of Vehicles on Highway", Volume 4: 24th Computers and Information in Engineering Conference, 2004 Publication	<1%
36	Jiayi Qiang, Yong Gu, Guochu Chen. "FPGA Implementation of SPI Bus Communication Based on State Machine Method", Journal of	<1%

Physics: Conference Series, 2020

Publication

37

link.springer.com

Internet Source

<1%

Exclude quotes On

Exclude matches < 20 words

Exclude bibliography On

Universiti Tunku Abdul Rahman			
Form Title : Supervisor's Comments on Originality Report Generated by Turnitin for Submission of Final Year Project Report (for Undergraduate Programmes)			
Form Number: FM-IAD-005	Rev No.: 0	Effective Date: 01/10/2013	Page No.: 1 of 1



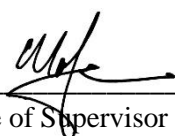
FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Full Name(s) of Candidate(s)	Yong Min An
ID Number(s)	16ACB01733
Programme / Course	Bachelor of Information Technology (Honours) Computer Engineering
Title of Final Year Project	Design and Implementation of a SPI Controller for Zigbee Module

Similarity	Supervisor's Comments (Compulsory if parameters of originality exceeds the limits approved by UTAR)
Overall similarity index: <u>7</u> % Similarity by source Internet Sources: <u>6</u> % Publications: <u>3</u> % Student Papers: <u>0</u> %	Overall similarity is within range.
Number of individual sources listed of more than 3% similarity: <u>0</u>	
Parameters of originality required and limits approved by UTAR are as Follows: (i) Overall similarity index is 20% and below, and (ii) Matching of individual sources listed must be less than 3% each, and (iii) Matching texts in continuous block must not exceed 8 words <i>Note: Parameters (i) – (ii) shall exclude quotes, bibliography and text matches which are less than 8 words.</i>	

Note Supervisor/Candidate(s) is/are required to provide softcopy of full set of the originality report to Faculty/Institute

Based on the above results, I hereby declare that I am satisfied with the originality of the Final Year Project Report submitted by my student(s) as named above.



Signature of Supervisor

Name: MOK KAI MING

Date: 24/4/2020

Signature of Co-Supervisor

Name: _____

Date: _____



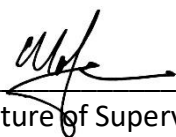
UNIVERSITI TUNKU ABDUL RAHMAN
FACULTY OF INFORMATION & COMMUNICATION
TECHNOLOGY (KAMPAR CAMPUS)

CHECKLIST FOR FYP2 THESIS SUBMISSION

Student Id	16ACB01733
Student Name	Yong Min An
Supervisor Name	Mok Kai Ming

TICK (✓)	DOCUMENT ITEMS
	Your report must include all the items below. Put a tick on the left column after you have checked your report with respect to the corresponding item.
✓	Front Cover
✓	Signed Report Status Declaration Form
✓	Title Page
✓	Signed form of the Declaration of Originality
✓	Acknowledgement
✓	Abstract
✓	Table of Contents
✓	List of Figures (if applicable)
✓	List of Tables (if applicable)
✓	List of Symbols (if applicable)
✓	List of Abbreviations (if applicable)
✓	Chapters / Content
✓	Bibliography (or References)
✓	All references in bibliography are cited in the thesis, especially in the chapter of literature review
✓	Appendices (if applicable)
✓	Poster
✓	Signed Turnitin Report (Plagiarism Check Result - Form Number: FM-IAD-005)

*Include this form (checklist) in the thesis (Bind together as the last page)

<p>I, the author, have checked and confirmed all the items listed in the table are included in my report.</p> <p style="text-align: center;">_____ YONG MIN AN (Signature of Student) Date: 24/4/2020</p>	<p>Supervisor verification. Report with incorrect format can get 5 mark (1 grade) reduction.</p> <p style="text-align: center;">  _____ (Signature of Supervisor) Date: 24/4/2020</p>
---	--