A SPECIAL PURPOSE PROCESSOR FOR IC TESTING AND SPEED CHARACTERIZATION

BY

AMRAN AL-AGHBARI

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

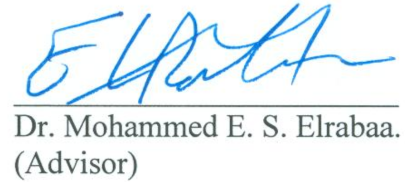
In

COMPUTER ENGINEERING

December, 2012

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN- 31261, SAUDI ARABIA
DEANSHIP OF GRADUATE STUDIES

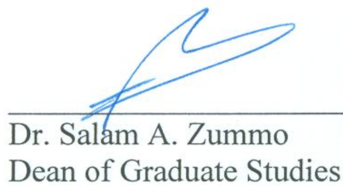
This thesis, written by **AMRAN ABDULRAHMAN ABDULWALI AL-AGHBARI** under the direction his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING.**



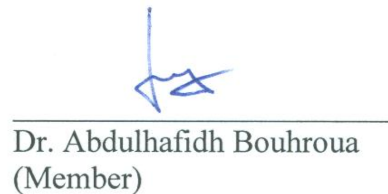
Dr. Mohammed E. S. Elrabaa.
(Advisor)

Dr. Basem AL-Madani
Department Chairman

Dr. Aiman Helmi El-Maleh
(Member)



Dr. Salam A. Zummo
Dean of Graduate Studies



Dr. Abdulhafidh Bouhroua
(Member)

17/3/13
Date:

© AMRAN ABDULRAHMAN ABDULWALI AL-AGHBARI
2012

Dedication

To my beloved parents, brothers and sisters all of them who I live for and think of. To my wonderful wife who are behind my happiness and success. To my teachers and doctors who taught me with good emotions and feeling that they care of me. To my friends who support me and are being happy when they feel me on a right way. To all who respect the truth and the goodness and follow the truth once they know it.

ACKNOWLEDGMENTS

All thanks and praise are due to my God, Allah (azzawajal) who created me, blessed me with health, as well as provided me with patience and ambition to achieve this research.

I acknowledge deeply my thesis advisor Dr. Mohammed Elrabaa who guided me in this research and I actually learned from him many things in this specific field and in computer engineering in general.

I acknowledge my university King Fahd University for Petroleum and Minerals (KFUPM). I acknowledge the Computer Engineering Department (COE). I acknowledge the faculties who taught me, helped me and were very very cooperative and interactive. I acknowledge all of them for pushing me to be at my current scientific level.

I acknowledge King Abdulaziz City for Science and Technology (KACST) for their supporting for the project related to the thesis and for providing us with all needed devices.

I acknowledge Saudi Arabia Kingdom for providing me an MS scholarship and for the facilitations on different fields and providing us with an excellent learning environment.

TABLE OF CONTENTS

DEDICATION III

ACKNOWLEDGMENTS IV

TABLE OF CONTENTS V

LIST OF TABLES XI

LIST OF FIGURES XII

LIST OF ABBREVIATIONS XVII

ABSTRACT XIX

ملخص الرسالة XX

CHAPTER 1 INTRODUCTION 1

1.1 Circuit IPs 2

1.2 IC Testing 2

Design for Testability (DFT) 3

Testing Principle 3

Characterization Process 4

1.3 Thesis Organization 5

CHAPTER 2 LITERATURE REVIEW 6

2.1 Digital circuit prototyping 6

Virtual Prototyping 6

Physical Prototyping 7

BIST-based test processors 7

Software-based testing 10

Low-cost FPGA-Based testers 11

2.2 Multi-cycle processors 13

CHAPTER 3 OVERVIEW OF THE PROPOSED TEST AND CHARACTERIZATION PLATFORM 14

3.1 The TACP Support circuitry (TSC) 15

TSC Fixed interface 17

3.1.1 The Configurable Clock Generator 18

3.1.2 The Frequency Measuring Circuit (FMC) 19

3.1.3 The Clock Selection and Application Circuit (CSaAC) ... 20
3.1.4 The Port Selection Block ... 22
3.1.5 The Test Application/Result Ports (TAP/TRP) ... 23
3.2 User Interface Software ... 25
Interface Protocol ... 26
3.3 Test and Characterizing Processor (TACP) ... 27
Processor ... 28
Memories ... 29
User Communication Unit ... 30
CHAPTER 4 DESIGN OF THE TEST AND CHARACTERIZATION PROCESSOR ... 31
4.1 Instruction Design and Microinstructions ... 31
4.1.1 SendSelectionMask instruction design ... 33
4.1.2 SendTestData instruction design ... 34
4.1.3 ReadResult instruction design ... 36
4.1.4 Compare instruction design ... 37
4.2 TACP Top Level Design ... 38
4.3 User Communication Unit ... 39
4.3.1 UART module ... 40
4.3.2 Communication flags ... 42
4.3.3 Previous received byte register ... 44
4.3.4 Transmitted byte multiplexers ... 44
4.3.5 Receiving state machine (rx_FSM) ... 45
4.3.6 Transmitting state machine (tx_FSM) ... 47
4.3.7 Break point register ... 49
4.3.8 Receiving counter ... 49
4.3.9 Transmitting counter ... 50
4.3.10 Communication Error Flag ... 51
4.3.11 PCWrite circuitry ... 51
4.3.12 PCRead circuitry ... 52
4.3.13 DCWrite circuitry ... 53
4.3.14 RCRead circuitry ... 54

4.3.15 DCRead circuitry 54
4.3.16 RCWrite circuitry 55
4.4 Memories 55
4.5 Memory Multiplexer 56
IsProcessing Circuitry 57
4.6 TACP Processor 57
4.6.1 The sequencer 60
4.6.2 The control store 61
4.6.3 Two previous parameter registers 64
4.6.4 Port selection mask circuitry 65
4.6.5 Selection mask shift register SM 66
4.6.6 Test data shift register TD 67
4.6.7 Test result shift register TR 67
4.6.8 Frequency register FR 68
4.6.9 Frequency control word register CW 68
4.6.10 Instruction Register IR 69
4.6.11 General counter CR 70
4.6.12 Word counter WC 70
4.6.13 User counter register UC 71
4.6.14 Stack pointer SP 72
4.6.15 Flags 73
4.6.16 Memory addressing circuitry 74
4.6.17 Test result memory writing circuitry 75
4.6.18 Push circuitry 76
4.6.19 Enumerate multiplexer 76
CHAPTER 5 TEST AND CHARACTERIZATION PROCESSOR IMPLEMENTATION 78
5.1 User Interface Implementation 79
Writing Programs 79
Executing Programs 80
Memory Interface 82
5.2 The Instruction Builder Software 83

5.3 TSC Prototyping 85
5.3.1 Clock gating 86
5.3.2 Clock multiplexing 87
5.3.3 Emulating the Configurable Clock Generator (CCG) 89
5.4 IPs Under Test (IUTs) 91
1st IUT: 4-bit Combinitional Adder 92
2nd IUT: 8-bit pipelined Adder 93
3rd IUT: s820 benchmark with scan chain is inserted 93
4th IUT: s820 benchmark with scan chain is inserted and flip-flops are doubled. 94
CHAPTER 6 TEST RESULTS 96
6.1 Maximum Frequency Test 96
6.2 Testing the 1st IUT: the 4bit Combinational Adder 98
6.3 Testing the 2nd IUT: the 8-bit Pipelined Adder 101
6.4 Complete Testing and Characterizing Program 103
Testing and characterizing the 3rd IUT 104
Testing and characterizing the 4rd IUT 108
6.5 Testing of the Loop Back from the chip 111
CONCLUSION AND FUTURE WORK 114
CHAPTER 7 APPENDIX 115
A. Instruction-Set List with their microinstructions 115
A.1. fetch 115
A.2. SendSelectionMask 116
A.3. SendTestData 116
A.4. ReadResult 117
A.5. ApplyAndCapture 118
A.6. Compare 118
A.7. Load_DCRead 119
A.8. Load_RCRead 119
A.9. Load_RCWrite 119
A.10. ResetCompareFlag 119
A.11. JCompareCorrect 120

A.12. JCompareError 120
A.13. SetFrequencyControlWord 120
A.14. SendFrequencyControlWord 120
A.15. MeasureFrequency 121
A.16. ReadFrequencyRegister 121
A.17. INC_CW 122
A.18. DEC_CW 122
A.19. SetHFClock 122
A.20. ResetHFClock 122
A.21. Load_UserCounter_value 122
A.22. Load_UserCounter_Mem 123
A.23. Store_UserCounter 123
A.24. INC_UserCounter 124
A.25. DEC_UserCounter 124
A.26. JNZ 124
A.27. JZ 124
A.28. Jump 125
A.29. Call 125
A.30. Return 125
A.31. NOP 125
A.32. Stop 126
A.33. ClearTestDataRegister 126
B. Instruction Builder Tutorial 127
1. Starting empty project 127
2. Add signals 127
3. Add instructions 128
4. Add microinstructions 128
5. Write data path code 129
6. Generate microcode 130
7. Export instructions 131
8. Generate test benches 131
C. User Interface Tutorials 134

Writing Programs tab 134
Executing Programs tab 135
Memory windows 137
Test-data memory window 137
Test-result memory window 138
Importing test vectors to memory window 140
REFERENCES 141
VITA 147

LIST OF TABLES

Table 3.1: Communication protocol - the available user commands with their codes. 27

Table 4.1: Instructions and their opcodes 32

Table 4.2: Communication protocol – UART communication speed. 40

Table 4.3: Communication protocol - the available user commands with their codes. 45

Table 4.4: Sequencer to data path signals (micro-instructions signals)............................ 62

Table 4.5: Selecting port examples.. 65

Table 5.1: The generated clock frequencies and their control words using the DCMs in the prototyped chip. 91

LIST OF FIGURES

Figure 1.1 : Principle of testing with ATEs: apply test patterns, capture responses and compare them with expected ones.. 4

Figure 1.2 : Characterization process: test the IP under different frequencies to find out the maximum. 5

Figure 2.1 : Basic BIST Architecture Block Diagram. 8

Figure 2.2 : A 4-bit linear feedback shift register (LFSR) which is used as a test pattern generator (TPG). 9

Figure 3.1 : The Proposed Platform: PC, Test Processor on FPGA board, and Support Circuitry On Chip. 15

Figure 3.2 : Block diagram of the TACP Support Circuitry (TSC) to be placed on the prototype chip. 16

Figure 3.3 : The fixed interface between TACP and TSC. 17

Figure 3.4 : The configurable clock generator. 19

Figure 3.5 : The frequency measuring Circuit (FCM). 20

Figure 3.6 : The state diagram of the control unit of the frequency measuring circuit (FMC). 21

Figure 3.7 : The Clock Frequency Control Register. 21

Figure 3.8 : The Clock Selection and Application Circuit. 22

Figure 3.9 : Logic Simulation Results for the CSaAC [1]. 22

Figure 3.10 : The Port Selection Circuitry. 23

Figure 3.11 : Test application port (TAP) and test results port (TRP). 24

Figure 3.12 : Scan test application/result ports. 25

Figure 3.13 : Packet type list. Each packet starts with flags defining a command and determines packet size. 26

Figure 3.14 : TACP main components. 28

Figure 3.15 : The three memories. Each memory has four inputs and four outputs. 30

Figure 4.1 : Four IUTs, each has three ports, each port has a serial number. 33

Figure 4.2 : SendSelectionMask microinstructions and flow chart. 34

Figure 4.3 : SendTestData microinstructions and flow chart. 35

Figure 4.4 : Simulation of the instruction SendTestData. 36

Figure 4.5 : ReadResult microinstructions and flow chart. 37

Figure 4.6 : Compare microinstructions and flow chart. 38
Figure 4.7 : TACP design top view and its subcomponents. 39
Figure 4.8 : Communication protocol connected to UART module. 40
Figure 4.9 : The UART module input/output diagram. 41
Figure 4.10 : Decoding the received type-byte to eighteen flags. 42
Figure 4.11 : Combining previous received byte with the current received byte to form a 16-bit word. 44
Figure 4.12 : tx_byte multiplexers. 44
Figure 4.13 : Receiving state machine FSM diagram. 46
Figure 4.14 : Receiving state machine circuitry. 47
Figure 4.15 : transmitting state machine sends test result and register values to the user.48
Figure 4.16 : transmitting state machine circuitry and the transmit signal. 48
Figure 4.17 : Break point register circuitry and BreakF flag. 49
Figure 4.18 : The receiving counter and its circuitry. 50
Figure 4.19 : The transmitting counter and its circuitry. 50
Figure 4.20 : Error flag circuitry. 51
Figure 4.21: PCWrite circuitry in the communication protoocol. 52
Figure 4.22 : PCRead circuitry in the communication protoocol. 53
Figure 4.23 : DCWrite circuitry in the communication protoocol. 53
Figure 4.24 : RCRead circuitry in the communication protoocol. 54
Figure 4.25 : DCRead circuitry in the communication protoocol. 55
Figure 4.26 : RCWrite circuitry in the communication protocol. 55
Figure 4.27 : Memories and address registers. Each memory has four inputs and four outputs. Each memory has two address registers; one write/read register and one read only register. 56
Figure 4.28 : Memory multiplexer circuitry manage memory access between the data path and the protocol. 56
Figure 4.29 : Memory multiplexer circuitry manage memory access from the TACP data path and the communication protocol to the memories. 57
Figure 4.30 : Processor top diagram. 58
Figure 4.31 : Processor components: sequencer, control store, and data path. 58
Figure 4.32 : The sequencer and the control store. 61

Figure 4.33 : Control store entry consists of selection, status signal, and branch address. 62

Figure 4.34 : Combining previous and current byte to form a 16-bit word for each port on instruction memory. 64

Figure 4.35 : CR down-counter with the 16 bit port selection mask generation circuitry and the CR_IsZero flag. 65

Figure 4.36 : 8-bit selection mask register shifts in the returned-back port selection from the chip. 66

Figure 4.37 : 8-bit test data register and its circuitry. 67

Figure 4.38 : 8-bit test results register and its circuitry. 68

Figure 4.39 : Frequency register circuitry. 68

Figure 4.40 : 16-bit control word register and its circuitry. 69

Figure 4.41 : Six bit instruction register. 69

Figure 4.42 : The 32 bit general counter CR and its circuitry. 70

Figure 4.43 : 4-bit Word Counter WC and its circuitry. 71

Figure 4.44 : User-counter circuitry. 72

Figure 4.45 : Stack pointer circuitry. 72

Figure 4.46 : Zero flag circuitry for the user counter register. 73

Figure 4.47 : Zero flag circuitry for the user counter register. 73

Figure 4.48 : Zero flags circuitry of the general counters. 74

Figure 4.49 : Clock selection Flag circuitry. 74

Figure 4.50 : Compare error flag circuitry. 74

Figure 4.51 : Addressing circuitry in the TACP data path. 75

Figure 4.52 : Test-result memory data-in port and write enable circuitry. 75

Figure 4.53 : Push circuitry generates Stack_in bus which is connected to the data-in port in the instruction memory. 76

Figure 4.54 : 5-bit multiplexer selects one byte at a time to be send as a response to the user interface. 77

Figure 5.1: The Implemented test & characterization platform; the host PC running the user interface tool, an FPGA board for the TACP connected to the PC, and another FPGA board containing the TSC and 4 CUTs and connected to the TACP FPGA. 79

Figure 5.2 : User interface to write and download programs to the TACP FPGA. 80

Figure 5.3 : User interface thet executes the program and tracks register contentss on the TACP FPGA. 81

Figure 5.4 : User interface to display memory contenets. 82

Figure 5.5 : Eight signal types in microcode. 84

Figure 5.6 : TACP Processor uses the microcode archeticture....................................... 85

Figure 5.7 : Replace gated clock signal by using enable signal. gated_clock = CLK && Enable 86

Figure 5.8 : Replace gated clock signal by using FPGA clock tri-state buffer. gated_clock = $Port_0$ && CLK_out 87

Figure 5.9 : BUFGCE simulation – dedicated clock signals tri-state with no pulse lose. 87

Figure 5.10 : The BUFGMUX clock multiplexer simulation... 88

Figure 5.11 : The ASIC version of clock selection and application circuit (CSaAC). The circuitry has a clock multiplexer and a clock gating.. 89

Figure 5.12 : The FPGA implementation of clock selection and application circuit CSaAC with the implementation of four CUTs clock gating. All gated clocks are replaced by FPGA clock buffers BUFGMUX and BUFGCE. The critical path has four level of clock gating. 89

Figure 5.13 : Emulating the DCO using eight DCMs. The DCO is combined with the four phase divider. The frequency is chosen by the 6-bit control word register. 90

Figure 5.14 : Illustration of assigning the ten port selection bits to the four IUTs.......... 91

Figure 5.15 : 1st IUT: 4-bit combinitional adder.. 92

Figure 5.16 : 2nd IUT: 8-bit piplined adder.. 93

Figure 5.17 : 3rd IUT: s820 benchmark with scan chain is inserted. 94

Figure 5.18 : 4th IUT: s820 benchmark with scan chain is inserted and flip-flops are doubled... 95

Figure 6.1 : Code and flow chart of the test program that changes frequency within a loop. 97

Figure 6.2 : Execution snapshot of the test program that changes frequency within a loop. 98

Figure 6.3 : Program execution window snapshot. The program tested a compinational 4-bit adder. 100

Figure 6.4 : Memory windows snapshot after executing the test program of the compinational 4-bit adder. The three windows show the test data, the test result and the comparision.. 101

Figure 6.5 : Program execution window snapshot. The program tested a compinational 4-bit adder. 102

Figure 6.6 : Program execution window snapshot. The program tested a compinational 4-bit adder. 103

Figure 6.7 : A complete test program and its flow chart for the S820S benchmark IUT. 104

Figure 6.8 : IUT3 testing: Snapshots for the user interface with four instances of the memory viewer after executing the IUT 3 test program. At this point, not all test results matches the expected. The comparison window shows some non-zero values where a difference exists.This indicates that the chip cannot handle the currrent frequency. 107

Figure 6.9 : IUT 4 testing: Snapshots for the user interface with four instances of the memory viewer after executing the test program. At this point, not all test results matches the expected. The comparison window shows some non-zero values where a difference exists.This indicates that the chip cannot handle the currrent frequency. 110

Figure 6.10 : Loop back testing; test-data is sent to the TSC and received back. The TD register gets back the returned test-data (TD = 0x22).. 112

Figure 6.11 : Test program for the selection mask SM register. SM contains the looped back selection mask that was sent (SM = 0x60).. 113

Figure 7.1 : Starting Instruction builder with an empty project. 127

Figure 7.2 : Add & edit signals and determine the signal type. 128

Figure 7.3 : Add & edit instruction and define its parameters sizes and names. 128

Figure 7.4 : Writing microinstruction and defining their cycles. 129

Figure 7.5 : Writing verilog code for data path component. 130

Figure 7.6 : Generating verilog code for microcode. 131

Figure 7.7 : Writing programs window... 134

Figure 7.8 : Executing programs window. 136

Figure 7.9 : Test-data memory window. 138

Figure 7.10 : Comparing test-results with expected results. 139

Figure 7.11 : Comparing results sorted by the third column to rise up vector caused error. 139

Figure 7.12 : Import test-vector dialog window and test-vectors file snapshot. 140

LIST OF ABBREVIATIONS

ASIC	:	Application Specific Integrated Circuit.
ASIP	:	Application Specific Instruction-set Processor.
ATE	:	Automatic Test Equipment
ATPG	:	Automatic Test Pattern Generator
BIST	:	Built-In Self-Test
BRAM	:	Block RAM
CAD	:	Computer Aided Design
CCG	:	Configurable Clock Generator.
CPI	:	Cycle Per Instruction or Clocks Per Instructions.
CPU	:	Central Processing Unit
CSaAC	:	Clock Selection and Application Circuitry.
DP	:	Data Path.
DCM	:	Digital Clock Manager
DUT	:	Device Under Test
EDA	:	Electronic Design Automation
FPGA	:	Field Programmable Gate Array
FMC	:	Frequency Measurement Circuitry.
FSM	:	Finite State Machine
GUI	:	Graphical User Interface
HW	:	Hardware
IC	:	Integrated Circuit
I/O	:	Input/Output
IP	:	Intellectual Property
IUT	:	IP Under Test
LUT	:	Look Up Table
µ-Address	:	Micro-Address
µ-Operation	:	Micro-Operation
opcode	:	Operation Code

RAM	: Random Access Memory
ROM	: Read-Only Memory
SoC	: System on Chip
SW	: Software
TACP	: Test And Characterization Processor
TSC	: Test Support Circuitry
UCU	: User Communication Unit.
USB	: Universal Serial Bus

ABSTRACT

Full Name : AMRAN ABDULRAHMAN ABDULWALI AL-AGHBARI

Thesis Title : A Special Purpose Processor for IC Testing and Speed Characterization

Major Field : Computer Engineering

Date of Degree : December, 2012

Conventionally, IC testing and speed characterization is carried out using very expensive Automatic Test Equipments (ATEs). Built-in-self-test (BIST) techniques can also be used as a low-cost solution for at-speed testing. However, BIST may require some modification of the circuit under test (CUT) to coup with the pseudo random nature of the test vectors (what is known as test points insertion). Also, speed characterization can't be directly carried out by BIST. Other low-cost testing and speed characterization methods are needed especially for developers of circuit IPs in small companies and universities. In this thesis, a special purpose test and characterization processor (TACP) for IC testing and speed characterization has been developed, implemented and tested. The processor utilizes specially developed test support circuitry (TSC) which is fabricated on the chip containing the IPs under test. The TSC, in coordination with the off-chip stand-alone TACP processor, receives test data serially, re-format them, apply them to IPs under test, reformat the test results and send it serially to the test processor. The TSC also include a configurable clock generator which is controlled by the TACP. By controlling the testing frequency and test patterns application, the IPs can be characterized to find their maximum frequency of operation. A proof-of-concept implementation was realized using two FPGA boards; one for the processor and the other to emulate the chip that contains IPs and on-chip circuitry. Also, a complete user interface tool has been developed allowing the user to write, load and administer his/her test program, download test data and receive the test results through a standard PC.

ملخص الرسالة

الاسم الكامل : عمران عبد الرحمن عبد الولي الأغبري

عنوان الرسالة : تصميم معالج خاص باختبار الدوائر المتكاملة و توصيف سرعتها

التخصص : هندسة حاسوب

تاريخ الدرجة العلمية : ديسمبر 2012م

يتم اختبار و توصيف سرعة الدوائر الإلكترونية الرقمية IC بطرق تقليدية تعتمد على استخدام أجهزة اختبار أوتوماتيكية مكلفة جداً (ATEs) Automatic Test Equipments. يمكن أيضا استخدام دوائر الفحص التلقائي (BIST) كحلول قليلة التكلفة لتقوم باختبارات السرعة العالية. لكن استخدامها يتطلب تعديلات في الدائرة المراد اختبارها لتتلائم مع طريقة التوليد العشوائي لسلاسل الاختبار (و تعرف أيضا بإضافة نقاط أختبار). كما أنها غير مصممة لاستخدامها مباشرة من أجل توصيف السرعة. هناك حاجة لطرق أخرى أقل كلفة لاختبار و توصيف سرعة الدوائر، خاصة لمبتكري الدوائر الرقمية المبتكرة Circuit IPs في الشركات الصغيرة و الجامعات. في هذه الرسالة، تم تصميم و تطبيق و اختبار معالج خاص باختبار و توصيف الدوائر الإلكترونية A Special Purpose Test and Characterization Processor (TACP). المعالج يستخدم دائرة إلكترونية أخرى مساندة صممت خصيصا لتسهيل عمله، بحيث يتم دمجها و تصنيعها في رقاقة إلكترونية مع الدوائر الرقمية المطلوب إختبارها و توصيف سرعتها. الدائرة المساندة – بالتنسيق مع المعالج المستقل (الغير مصنوع معها في نفس الرقاقة الإلكترونية)– تستلم بيانات الإختبار و تعيد ترتيبها ثم تطبقها على الدائرة المطلوب إختبارهاز و بعد ذلك تقوم بترتيب النتائج ثم ترسلها للمعالج بالطريقة التسلسلية Serially. الدائرة المساندة فيها أيضا ترددات قابل للضبط (CCG) Configurable Clock Generator يتحكم به المعالج. فعندما يقوم المعالج بتطبيق بيانات الإختبار مع التحكم بالتردد يمكن توصيف الدائرة و إيجاد أعلى سرعة يمكن أن تعمل عليه. تم إثبات هذا المفهوم بتطبيقه عمليا باستخدام الدوائر الرقمية القابلة للبرمجة Field Programmable Gate Array (FPGA)، حيث تم تصميم المعالج على واحدة منها و تم عمل محاكاة للدائرة المساندة مع بعض الدوائر للإختبار في واحدة أخرى. كما تم تصميم برنامج حاسوبي متكامل يسمح للمستخدم بكتابة برنامج إختبار و إدخال بيانات الإختبار و قراءة النتائج عبر الحاسوب.

CHAPTER 1

INTRODUCTION

Developers of circuit intellectual properties (IPs) in universities and small companies need to silicon-prove their IPs. Unfortunately, automatic test equipments (ATEs) that can handle Giga-Hertz testing are very expensive making them beyond the reach of many universities and IP developers in small companies. ATEs are best suited for testing thousands of chips of the same design, however, they are not practical for developers who prototype and verify only several number of different circuit IPs.

In this thesis, a special purpose processor that can test and characterize prototypes of circuit IPs has been developed. These IPs are fabricated along with a special test support circuitry (TSC) on the same chip. The processor is a part of the low-cost testing and characterizing platform introduced in [1]. The processor is designed and implemented on an FPGA board. To verify the processor's operation, the rest of the platform has also been implemented. The support circuitry with some IPs are emulated on another FPGA board. A graphical user interface tool was implemented to write programs and control executing them on the processor. Many successful programs were successfully run.

1.1 Circuit IPs

Circuit intellectual property is a reusable unit of logic, cell, or chip layout design. It is also called IP for simplicity. It is used as building blocks within larger designs. IPs are licensed either as soft IPs which are a synthesizable hardware-description language modules or as hard IPs which are layout macros [2].

IP-based design promise large productivity gains. Many IPs are used and integrated with other circuitries to work together as a single system. IP-based design has a very short time-to-market development cycle because it reuses existing IPs to build larger designs. IPs have rapidly become the cornerstone of the SoC industry [3, 4, 5]. In SoC, pre-designed and pre-verified hardware and software blocks can be combined on chips for many different applications.

Many researchers in universities and small companies are developing new IPs but they face a huge problem when they try to market them. Unfortunately, IPs cannot be marketed unless silicon-proven with specific performance numbers (Maximum frequency of operation, maximum throughput, maximum latency, average power, etc.). Moreover, fabricating an IP prototype is relatively cheap, testing and characterizing it on the other hand could be very costly. Developing a cost-effective solution would enable circuit designers to prototype, test and characterize their IPs at the operational speeds.

1.2 IC Testing

Testing is a manufacturing step that ensures that each of the fabricated physical devices (integrated circuits or ICs), has no manufacturing defect(s). Testing also characterizes the fabricated ICs by determining their maximum operating frequency (called speed or frequency binning).

Verification, on the other hand, is a predictable testing that comes before fabrication in all phases of the IC design flow used to prove the correctness of the design. Functional simulation and timing simulation are examples of verification methods.

Design for Testability (DFT)

To facilitate the test process, modification on the synthesized design is suggested to get a testable design. Design for testability (DFT) methods were developed and became a standard phase after the synthesis phase in the IC design flow. In this phase, all storage elements inside the IC are replaced with scan cells which are connected and forms multiple shift registers (i.e. scan chains). Thus, the IC can be set directly into a specific state by shifting in stimulus to all storage elements. Additional test points may be inserted to improve the observeability and controllability of the design in case a pseudo random pattern generator is used to generate the test vectors (as the case with BIST).

Testing Principle

Figure 1.1 illustrates the basic principle of digital testing with ATEs. Test patterns are applied to the IC, and then test responses are captured and compared with stored expected responses. The circuit is considered good if the responses match. The quality of the tested circuit will depend upon the thoroughness of the test vectors that are usually generated using Automatic Test Pattern Generation (ATPG) techniques. The test vector itself usually has two parts; the first is applied to the IC inputs, and the other part is shifted into the scan chain of the IC under test to change its storage elements values and therefore force the IC to a specific state.

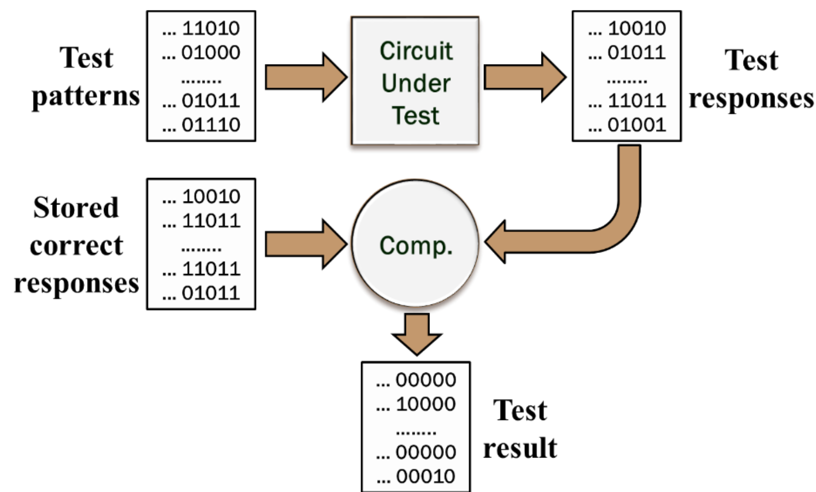


Figure 1.1 : Principle of testing with ATEs: apply test patterns, capture responses and compare them with expected ones..

Characterization Process

Characterization is to determine the exact limits of device operating values: What is the maximum frequency the design can operate on with no errors? How much power does it consume? In this thesis, the main concern is speed characterization. To do that, the IC clock frequency is set initially at minimum value, and then at-speed testing of the IC is administered by applying stimuli and comparing the test results with expected ones. If the test result is OK, the frequency is increased and the test is done again and again. The test continues until reaching the maximum frequency or getting a difference between the test result and the expected results. Figure 1.2 illustrates this process.

In at-speed testing, part of the stimulus is shifted into the scan chain while the rest is used as primary inputs. Thus, the stimulus length is equal to the scan chain length plus the number of primary inputs while the result vector length is equal to the scan chain length plus the number of primary outputs. Testing is done by applying two clock cycles at a specific frequency. The first

clock pulse results in new stimulus which is applied with the next clock pulse. This way a transition delay fault can be discovered which indicates that the frequency has to be decremented.

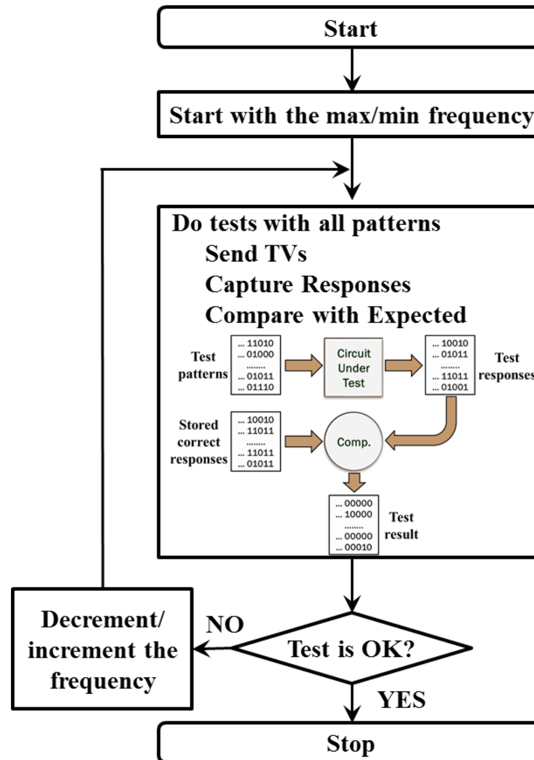


Figure 1.2 : Characterization process: test the IP under different frequencies to find out the maximum.

1.3 Thesis Organization

The next chapter contains a literature survey on test and characterization methods and multi-cycle processors. The platform overview is explained in chapter three. Chapter four contains the complete design of the TACP and its components in details. The implementation of the platform is presented in chapter four with a discussion about the ASIC emulation problems. Experimental results are presented and discussed in chapter six followed by conclusions and references. Several appendices that summarize the TACP instruction set and provide user tutorials on the different software tools developed are provided at the end.

CHAPTER 2

LITERATURE REVIEW

This chapter includes literature survey about IC test processors and other testing and characterizing methods. The survey shows the contribution of the testing platform on the IC testing and characterizing field. The last section in the chapter introduces the multi-cycle processor architecture that are needed to build the testing processor.

2.1 Digital circuit prototyping

Developers of circuit IPs need to prove the functional correctness of their IPs and to characterize their performance (speed and power). There are two main methods for verifying new circuit IPs functionality and performance; simulation-based verification with very detailed process and device models in what is called virtual prototyping, and through physical prototyping by either using FPGA implementation or via fabrication with a silicon foundry.

Virtual Prototyping

One way to prove the correctness of an IP is virtual prototyping. Virtual prototyping tools attempt to capture the effects of all physical parameters (process and otherwise) through modeling. Virtual prototypes are used to faithfully represent the ‘‘product-to-be’’, so as to be able to simulate

its features, performances, functionality and usage before the real product is actually built [6]. Virtual prototyping is just simulation-based verification software that is more accurate than traditional simulations. The existing virtual prototype software costs high and is not practical for testing circuit IPs. Furthermore, it is still a simulation that cannot be compared with a silicon-proven chip.

Physical Prototyping

Chip fabrication is the most trusted and accepted method of verification, since it reveals the actual performance of the circuit being prototyped. Fabricated chips would achieve the highest performance but they would require very expensive automatic testing equipments (ATEs) to test and characterize their performance at their operational speeds (called at-speed testing).

Automatic test equipments (ATEs) are standalone devices that can be used to test digital designs. They have many advantages such as digital and analog test capability, high-current pin protections and high-speed test execution. They also has disadvantages such as they are very expensive and require an accurate setup. Agilent, Advantest and Teradyne are example of companies that provide these machines. The ATEs mainly detects failures due to manufacturing defects, aging, environment effects and others [7] and helps manufacturers to maintain their manufacturing tools. They are not practical for prototyping IPs of universities researchers and small companies because of their high cost.

BIST-based test processors

Built-in self-test (BIST) is the primary test methodology which reduces dependency on external Automatic Test Equipment (ATE). It is a circuitry that is designed and integrated on the chip with

the circuit under test (CUT). It has many components as shown in the simple block diagram in Figure 2.1. The test pattern generator (TPG) generates test patterns to be applied to the circuit under test (CUT). The analyzer retrieves the responses, updates responses signature and compares the signature with a good CUT signature to detect fault.

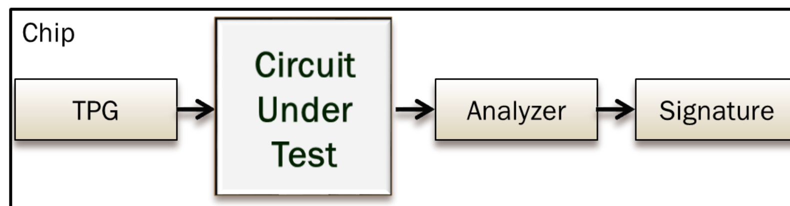


Figure 2.1 : Basic BIST Architecture Block Diagram.

Test pattern generators (TPG) is the main component that affect the test process. It could be deterministic or pseudorandom (i.e. requires a seed to start the pattern random generation). In the deterministic way a ROM could be used to store good test vectors that covers most faults. These good vector usually are generated using automatic test pattern generator (ATPG). However, this is too expensive in the chip area. Another way is to use a counter on the circuit input to generate all permutations. This is not practical if inputs number is large [8]. For pseudorandom TPGs, Linear feedback shift register (LFSR) is a well-known example of TPGs that needs a small hardware. Figure 2.2 show a 4-bit LFSR and the sequences that it can generate. It randomly generates all 4-bit permutations (except the sequence that has all zeros). Another successful TPG idea is to use LFSR with a small ROM that stores some test patterns that are not covered by LFSR. This called the mixed-mode testing in which the pseudo-random testing is followed by a deterministic testing approach. General-LFSR is presented and well explained in [9] to be used instead of LFSR. GLFSR is the general form of LFSR, MISR. It can generate higher randomness test vectors so the fault can be discovered with fewer patterns. Using GLFSR for a mixed-mode testing approach is presented in [10] to investigate its performance. It starts by pseudo-random test while a controller

counts the generated test vectors. When the counter reaches a predefined number, it starts the deterministic test.

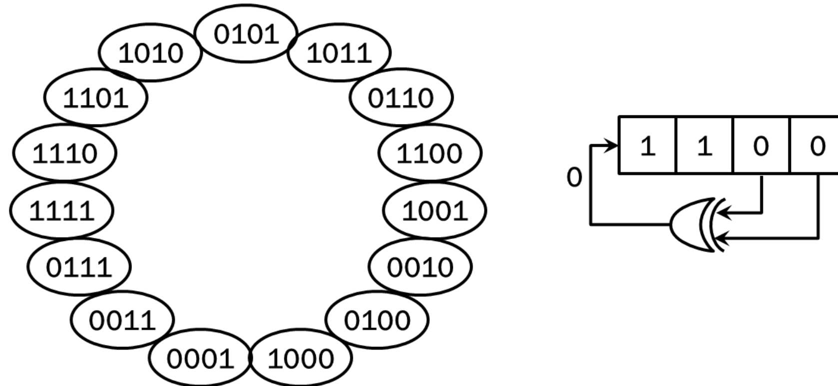


Figure 2.2 : A 4-bit linear feedback shift register (LFSR) which is used as a test pattern generator (TPG).

A BIST-Based test processor is presented in [11] that contains linear feedback shift register (LFSR), signature analyzer and RAMs. The LFSR is programmable and can set user seed for every test set. The processor uses LFSR to generate random numbers and apply them to the circuit under test (CUT). Then, it compress the responses to generate the signature and store them in its RAM to be sent later to computer to compare them with a signature of a good CUT. Other RAMs is used to store the seeds, test length and the polynomial.

It is not practical to test the circuit using all 2n combinations. Many researches are done on selecting the best seeds that can cover most faults. Test length affects the testing time. High fault coverage cannot be achieved within an acceptable test length. Reseeding is a technique which has been proposed to solve this problem. A heuristic approach is presented in [12] that come of a small leads to very small number of seeds, short test sequences and almost complete fault coverage. Based on that approach, [13] proposes and simulate an external test processor architecture.

In general, BIST test quality depends on signature analysis that can detect more than 99% of faults. This coverage percentage decreases with the increasing of the complexity of the design.

BIST-based test processors achieve fault coverage for memory cores better than complex design such as microprocessors and IP cores. BIST also adds an overhead area for each CUT since it is included in the chip that contains the CUT. Some researchers proposed efficient utilization of area by using one BIST to test multiple components in SoC [14]. They used a microcode-based controller to control one BIST to test multiple RAM cores for SoC system.

In summary, BIST is a good random test method that require a reasonable area. It can reach to 100% fault coverage for some designs and more than 90% in average but also requires modification of the CUT in order to achieve the high fault coverage.

Software-based testing

Software-based self-testing strategy is a proposed for complex designs that cannot be tested perfectly using BIST techniques such as system-on-chip (SoC). System-on-chip consists of many heterogeneous embedded modules such as RAMs, processors, IPs, etc. There is a need for special test processor designed on the chip to test all its components for these reasons; some of these components could be black boxes and not designed for testability. In addition, the controllability and observability become more limited with the increase of the complexity of the design. Also, most of SoC components are not connected to the ASIC pins and cannot be tested by external testers.

Trying to utilize BIST-based testing for SoC, researchers in [15] suggest using BIST-based testing strategy for testing processor IPs by generating random instructions. This way, they achieve a good fault coverage with a minimum area and without the need for scan chain insertion into the processor under test.

Researchers in [16] remark the fact that most SoC has at least one processor core. They suggest utilizing existing processors and use a subset of their instruction set for testing purposes. They also suggest mutual self-test of processors that can do hardware- and software-based test strategy in which the following possibilities are considered; one processor is made active and tests the other passive processor at the logic block level via scan-chains(hardware-based testing). Then both processors are made active one of them test the other using valid op-code, valid data and functional inputs (software-based testing). Also the processor can provide active March test for a memory block. In addition, one of the test processors can work as a watchdog that monitor the chip correctness at normal operation.

In most cases, the test processor in SoC generates the test vectors. Some researchers suggest connecting the SoC to large external RAM that holds the test program, data and expected responses [17]. This way the RAM could be considered as an external ATE but the test process is controlled by the chip.

The test processor in [18] is a 16-bit RISC processor and supported by a scan controller that is connected to all components scan chains in the chip. It can support bus tests, functional tests, scan testing and act as a watchdog in normal operation.

Low-cost FPGA-Based testers

FPGA is also used in many testing platforms to present a standalone low-cost tester. An FPGA holds the tester, software to control the tester and the chip under test which is usually put on a daughter board that is connected to the FPGA board. This platform is good for functional testing. It can also do at-speed testing but at low speeds (not more than few hundred MHz) [19, 20, 21, 22, 23]. FPGAs are also used as a verification method to prototype ASIC designs [21].

An FPGA-based functional tester to test SoC is presented in [22] . An automatic software tool in a host PC prepares a compressed test set and a decompression logic. Then, the compressed test set is downloaded into SRAM on the FPGA-board and the decompression logic is downloaded into the FPGA. The FPGA reads, decompresses the test set from the RAM, sends them to the DUT and captures the responses.

SRAM testing platform is presented in [20]. It is consist of FPGA board (i.e. Xilinx Virtex 4) connected to slave board accommodates SRAM under test. The FPGA executes a special March-C testing algorithm using a Microblaze™ micro-processor that could detect specific SRAM faults.

FPGA-based test platform is presented in [19]. It is consist of FPGA board (Xilinx Spartan 3) connected to slave board that accommodates the DUT. The platform uses three SRAMs to store timing data, test patterns and responses. There are a PC software to read timing data and test files and send them to SRAMs. A state machine on the FPGA manages all operations.

In the work presented in [23], a new multiplier architecture is designed, implemented, fabricated and then used as a DUT for the presented FPGA-based testing platform.

An on-chip and at-speed tester for memories is presented in [24]. The presented patent platform can do at-speed testing and characterizing of multiple memories. It consists of two parts; the centralized flow controller and the localized signal generator. The centralized flow controller consists of memory, processor and a user interface. The localized signal generator is to be included with the memory under test in an integrated circuit. It has also a clock generator, characterization circuit and a phase lock loop (PLL). The memory stores some memory test algorithms and the test program that use these algorithms. The testing starts by receiving an execute signal indicating the memory type and the storing test operations.

That platform is excellent since it can do testing and characterizing. However, it is dedicated for memories only and cannot be generalized because it depends on specific stored testing algorithms.

2.2 Multi-cycle processors

Processor architecture can be single-cycle in which each instruction is executed in one cycle (i.e. Clock Per Instruction (CPI) = 1) or multi-cycle in which the instruction is executed in multiple cycles (i.e. CPI > 1). Multi-cycle architecture is suitable for our work because it can deal with variable data size. Instructions of variable data size are needed to send test data, receive test results and compare results.

Microcode is a simple well-known processor architecture that allows multi-cycle instructions. It has the ability to add and remove instructions with relatively less effort. It consists of three parts; data path, sequencer and control store. The sequencer is the control unit that fetches low-level microinstructions from a control store and derives the appropriate control signals as well as micro-program sequencing information from each microinstruction. The data path is controlled by these control signals. Control store is a ROM and stores microinstructions of all instructions. Each entry reflects all the signal values at specific clock. In the data path all operations and data manipulations are performed. It may contain registers, shifters, ALUs, or any combinational and sequential circuits. Data path is controlled by control signals coming from the selected entry of the control store. A good view of microcode history is presented in [25]. It discusses the evolution of microcode from its introduction to its decline and to its likely resurgence in custom computing machines and reconfigurable computing.

CHAPTER 3

OVERVIEW OF THE PROPOSED TEST AND CHARACTERIZATION PLATFORM

This chapter gives an overview of the targeted test and characterization platform [1] and describes its components in details. Figure 3.1 shows the general architecture of the test and characterization platform. Unlike many previous techniques which either use a test circuit that is entirely on-chip with the device under test or entirely off the DUT's chip, the new method uses a hybrid approach. Also, unlike the approach in [20] where voltage and clock controllers are integrated on the DUT's chip while the test controller could be off-chip, this method provides a general way for applying stimuli and capturing results with fixed interfaces (i.e. the same test controller can be used to test and characterize any circuit). Also, unlike the approach in [20] no BIST circuitry is required. The test controller (TACP) can be implemented on an ASIC or a Field-Programmable Gate Array (FPGA). The TACP could be interfaced to a PC for receiving test instructions and data and sending the test results. The TACP's on-chip support circuitry provides the fixed interface (Figure 3.3) to the TACP and the controlled clock source for the IUTs. All interfaces use serial data communications to save I/O pins [1].

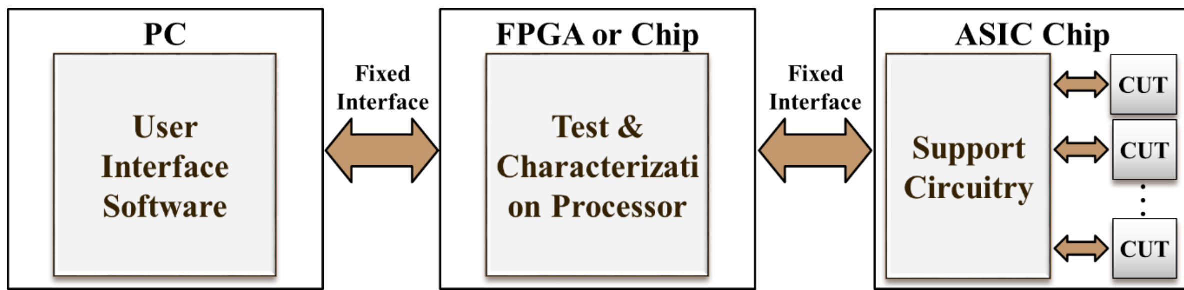


Figure 3.1 : The Proposed Platform: PC, Test Processor on FPGA board, and Support Circuitry On Chip.

3.1 The TACP Support circuitry (TSC)

The TACP support circuitry (TSC), shown in Figure 3.2, performs the following functions:

- **Port Selection**: The proposed method supports testing and characterization of unlimited number of IPs on the prototype chip. Each IP could also have several input/output ports for different purposes (functional I/Os and scan I/Os). The TSC provides a mean to select a specific port to apply/receive test data to/from.
- **Serial-to-Parallel and Parallel-to-Serial data conversion (SERDES):** To have fixed logic interfaces between the TACP and the prototype chip all data communications are serial. As such, the TSC converts the received serial test data to parallel data to be applied to the IUT. It also converts back the captured test results from parallel form to serial form.
- **Controlled Clock Source:** All data transfer between the TACP and the prototype chip and functional characterization is carried out using the TACP relatively low frequency clock to ease the design of the interface. For speed characterization, a high speed digitally controlled oscillator is provided as part of the TSC. The user can increase/decrease this oscillator frequency and use it for at speed testing of his/her IP(s).

Figure 3.2 shows a block diagram of the TSC. The main components are the configurable clock generator, the port selection block, test application ports (TAPs), and test result ports (TRPs).

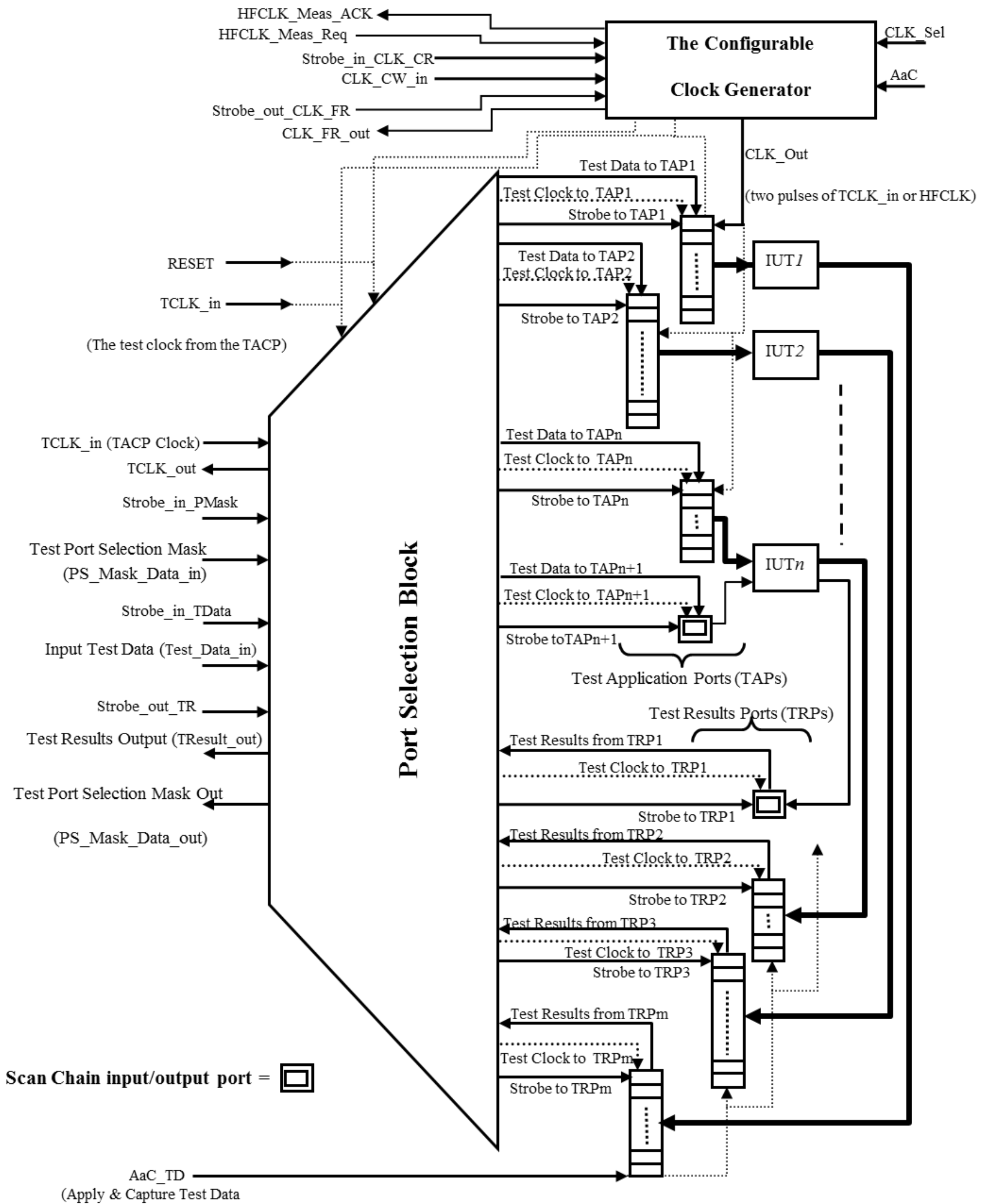


Figure 3.2 : Block diagram of the TACP Support Circuitry (TSC) to be placed on the prototype chip.

TSC Fixed interface

Figure 3.3 shows the interface between the TACP and the prototype chip. This interface is fixed and will not change with any chip being tested or characterized. Whatever the number of IPs to be tested and whatever the number of inputs each IP has, the interface is fixed and does not change. The interface has twenty pins as depicted in Figure 3.3. Data is moved serially. The transition of the test data happens while the strobe signal is high which works as a shift signal for the serial data.

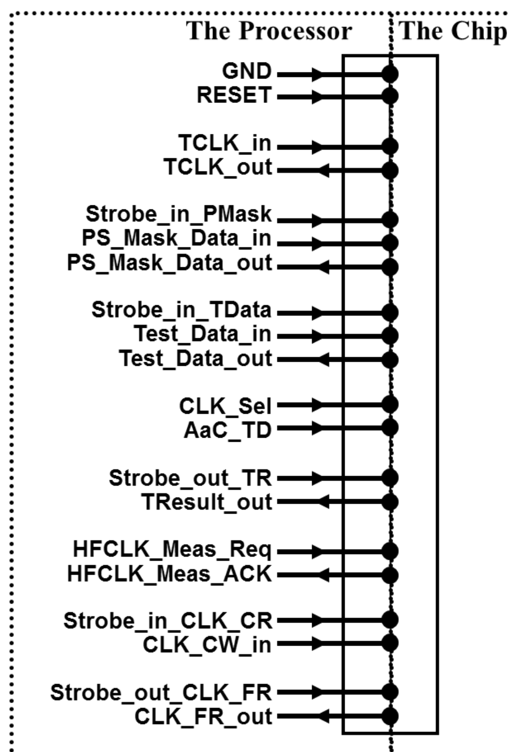


Figure 3.3 : The fixed interface between TACP and TSC.

- **TCLK_in**: The processor clock operates the TSC to synchronize it with the processor.
- **TCLK_out**: The same processor clock loops back for de-skewing purpose.
- **Strobe_in_PMask**: Strobe signal for scanning in the port selection bits.
- **PS_Mask_Data_in**: Port selection input stream.
- **PS_Mask_Data_out**: Port selection output stream used for loop back testing purposes.
- **Strobe_in_TData**: Strobe signal for scanning in test data.
- **Test_Data_in**: Test data input stream.
- **Test_Data_out**: Test data output stream used for loop back testing purposes.

- **CLK_Sel**: Selects the clock source for testing; either the TACP TCLK or the on-chip HFCLK.
- **AaC_TD**: Apply-and-capture signal that prompt the TSC to apply two cycles of the selected clock to the selected IUT and capture the result.
- **Strobe_out_TR**: Strobe to read out test result.
- **TResult_out**: Test result output stream.
- **HFCLK_Meas_Req**: A request to measure the selected frequency on the chip.
- **HFCLK_Meas_ACK**: An acknowledgement indicates finishing the frequency measurement process.
- **Strobe_in_CLK_CR**: Strobe to input the control word of the on-chip clock generator.
- **CLK_CW_in**: Control word input stream.
- **Strobe_out_CLK_FR**: Strobe to read out the measured frequency register.
- **CLK_FR_out**: Measured frequency register output stream.

3.1.1 The Configurable Clock Generator

As mentioned before, the regular test clock is coming from the TACP which is off-chip. This clock is kept at a moderate frequency (50~100 MHz). Hence no special high-frequency transceivers or signal traces are required. This eases the design of the interface and keeps its cost to a minimum. At the same time this clock is adequate for scanning in/out the test data/results and performing functional characterization of the IUTs. Frequency characterization, however, requires a clock source that can be configured to produce a high-frequency clock. This configurable source is placed on the prototype chip and dubbed the Configurable Clock Generator. This generator, as illustrated in Figure 3.4, is made up of a frequency measuring circuit (FMC), Figure 3.5 and Figure 3.6, a clock frequency control register, Figure 3.7, and a clock selection and application circuit, Figure 3.8.

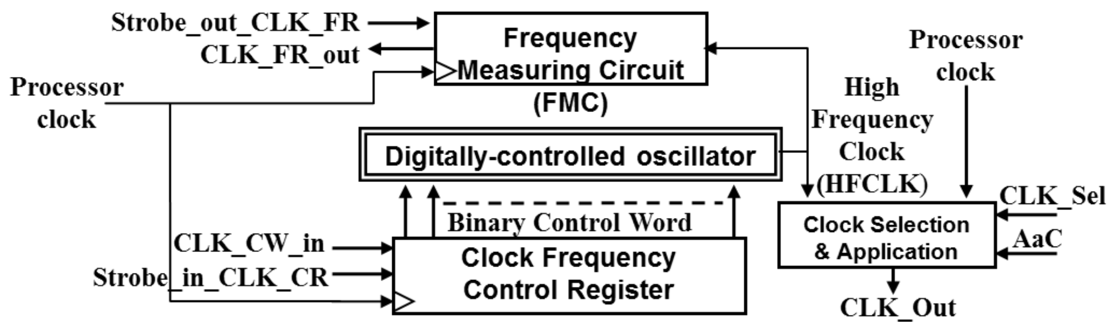


Figure 3.4 : The configurable clock generator.

3.1.2 The Frequency Measuring Circuit (FMC)

The FMC, simply counts the number of high-frequency clock cycles within a certain period and puts the result in a shift register that would be shifted out by the TACP using the Strobe_out_CLK_FR strobe signal and through the CLK_FR_out pin. The measurement period is specified by the TACP as the difference between activating the measurement request (HFCLK_Meas_Req) and deactivating the request. When the FMC is done it activates the acknowledgement signal (HFCLK_Meas_ACK) which remains high till a new measurement request is received. The detailed design of the FCM including its controller's state diagram and its operation is shown in Figure 3.5 and Figure 3.6. The user can control the accuracy of the measurement by having a longer measurement period. To get the frequency the following formula is used:

$$Measured\ Frequency = \frac{FR\ \times TACP\ processor\ frequency}{Request\ period\ length\ (Cycles)}$$

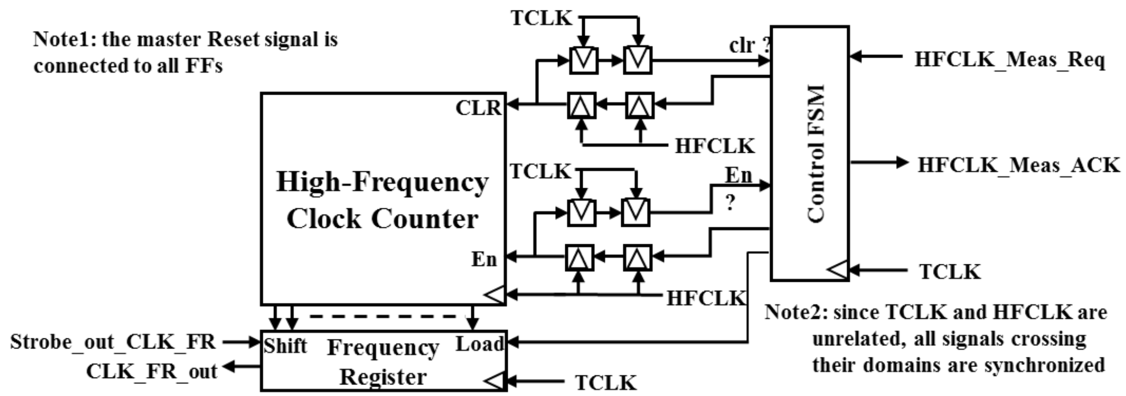


Figure 3.5 : The frequency measuring Circuit (FCM).

3.1.3 The Clock Selection and Application Circuit (CSaAC)

The clock selection and application circuit (CSaAC), Figure 3.7, is responsible for selecting the required test clock (based on the CLK_Sel input signal from the TACP) and applying exactly two pulses of that clock to the selected TAP/TRP ports (in response to a strobe on the AaC input). The TACP triggers the CSaAC by setting the AaC signal to high for at least two cycles of the selected clock (Sel_CLK). The CSaAC will produce exactly two pluses of the selected clock for each AaC pulse, but in order for this circuit to fire again, the AaC signal must be reset for at least two cycles of the selected clock. The clock gating circuit ensures that the two pulses applied are complete with no glitches by enabling the output clock when the selected clock is low. The only constraint for this circuit is that the sum of the clock inverter delay, the FF's clock to Q delay and the clock-gating AND gate delay is less than the width of the negative pulse of the selected clock. Also, due to the required synchronization of the AaC input with the selected clock (3 FF synchronizer is used), the output clock pulses will have a latency of 3 cycles of the selected clock. The TACP takes care of all these issues by applying the AaC signal for two TCLK_in cycles (TCLK_in frequency is always ≤ than the selected clock frequency) and then resetting it for two more cycles before setting it again (in case of successive apply and capture commands).

Figure 3.9 shows logic simulation results of the CSaAC with unit gate delays. Figure 3.9 (a) shows how the circuit functions correctly when the AaC pulse is at least two cycles of Sel_CLK and the so is the reset time in between AaC pulses. When the AaC pulse is less than two cycles or the reset time in between pulses is less than two cycles, the circuit fails, as shown in Figure 3.9 (b) and Figure 3.9 (c), respectively.

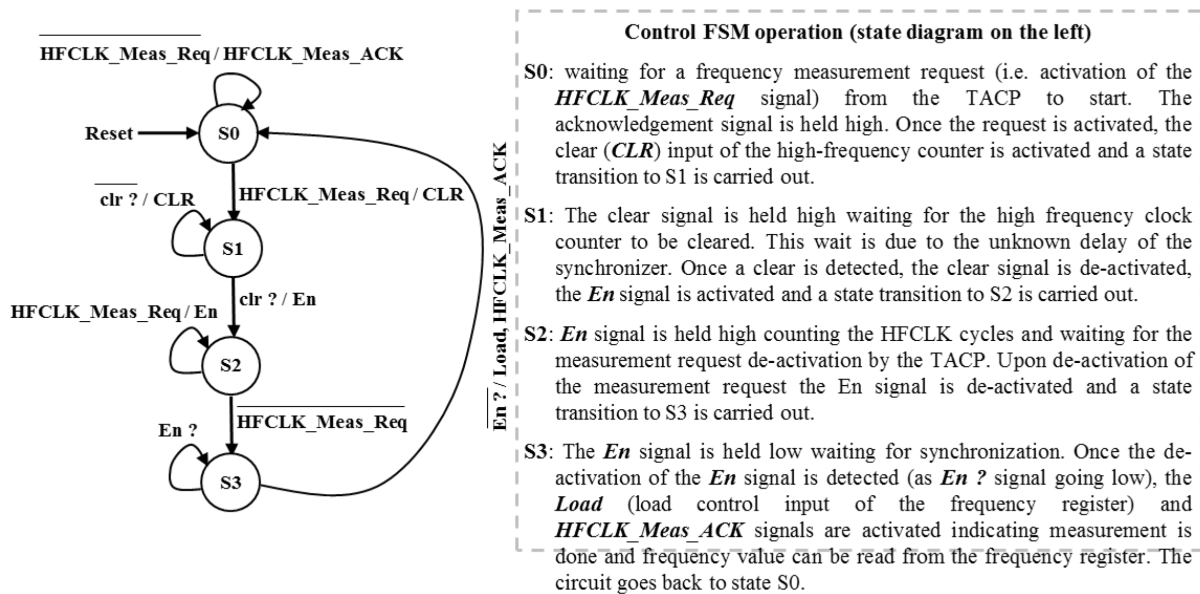


Figure 3.6 : The state diagram of the control unit of the frequency measuring circuit (FMC).

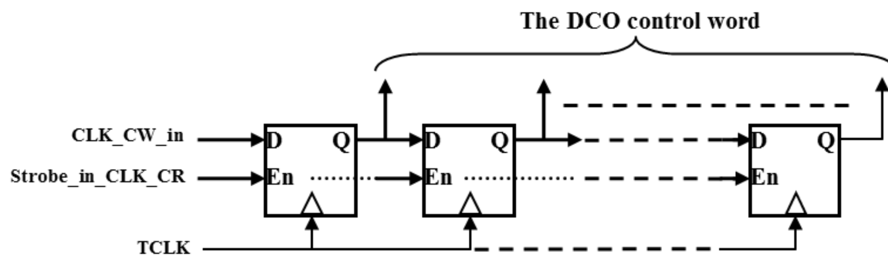


Figure 3.7 : The Clock Frequency Control Register.

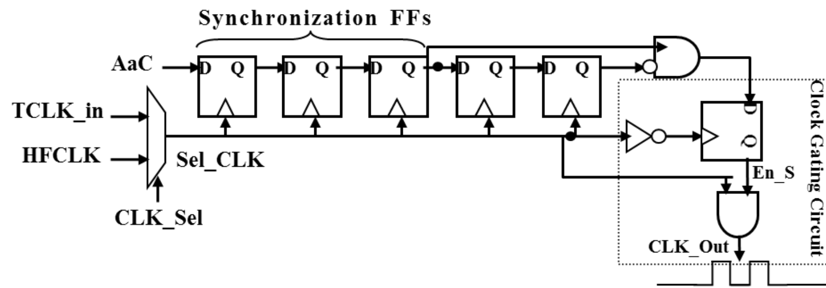


Figure 3.8 : The Clock Selection and Application Circuit.

3.1.4 The Port Selection Block

This block is responsible for selecting a specific test application/test result port to deliver the strobes, test clock and input test data to or receive test results from. The user can select a single input/output port or two ports (one input and one output). To make this block general yet with a fixed interface to the TACP, it is made up by cascading a basic cell as shown in Figure 3.10. The selection mask is loaded serially through the PS_Mask_Data_in input using the Strobe_in_PMask strobe signal. The TACP supports variable length selection mask (up to 216 bits). The port selection mask is also read out through PS_Mask_Data_out for testing the selection chain.

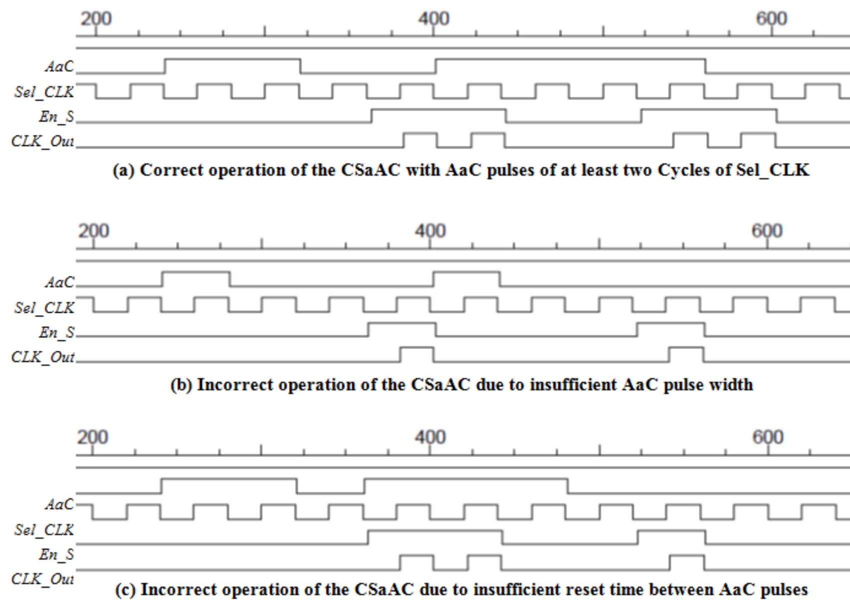


Figure 3.9 : Logic Simulation Results for the CSaAC [1].

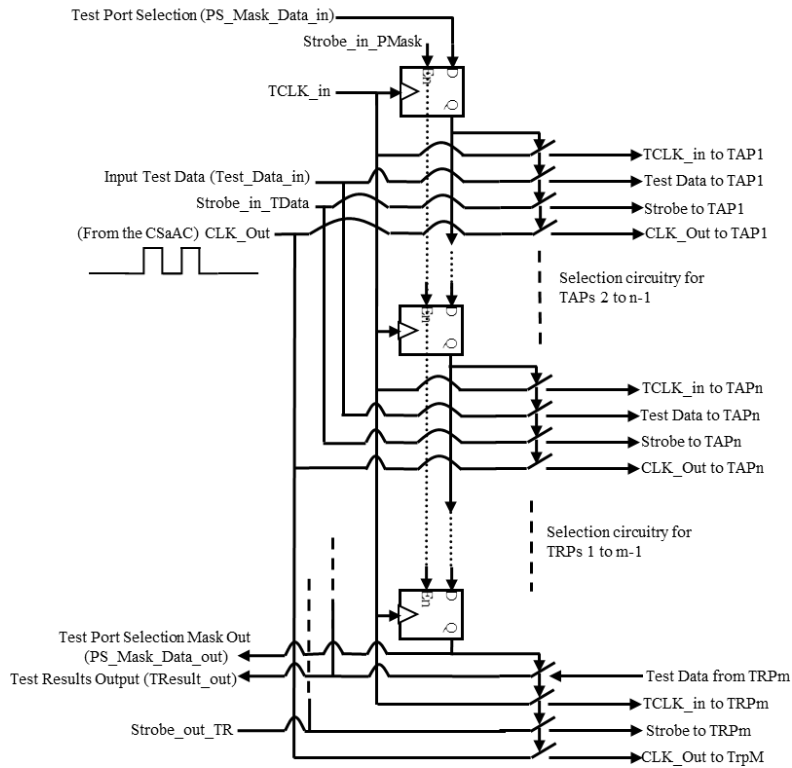
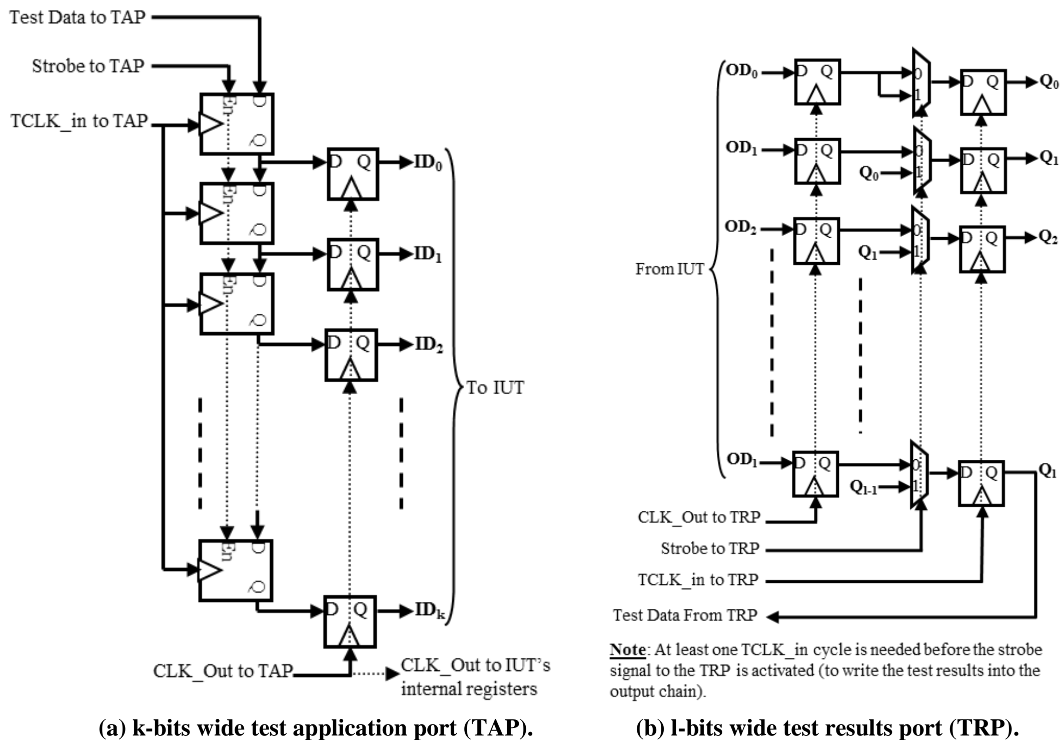


Figure 3.10 : The Port Selection Circuitry.

3.1.5 The Test Application/Result Ports (TAP/TRP)

There are two types of test application/result ports as was illustrated in Figure 3.2. The first type, shown in Figure 3.11 (a) and Figure 3.11 (b), are used for applying and capturing primary inputs/outputs of an IUT. These are similar to boundary scan ports and are made of shift registers for scanning in/out the test data/results and parallel-load registers for applying/capturing the test data/results. As Figure 3.11 shows, each TAP (or TRP) is made of a cascaded number of identical cells equal to the port's data width. The shift registers use the TCLK_in and the application/capture registers use the selected apply and capture clock (CLK_Out). The CLK_Out clock is also used for the IUT's internal registers. For the TRP, the TACP needs to apply at least one TCLK_in cycle (to load the test results into the shift register) before activating the Strobe_out_TR signal to read out the results.

IP designers may also need to use full-scan designs in addition to/or instead of boundary-scan. This requires making all or part of the internal Flip Flops scanable (forming one long scan chain). Such scan chains could be used for debugging/diagnostics of an IUT internal circuitry or to fully test a sequential circuit which is difficult to do using only primary inputs/outputs. Special TAP/TRP scan ports were developed for scan chain inputs/outputs of IUTs, as shown in Figure 3.12. These ports have to be used (i.e. selected) in pairs where data is shifted through the chain when either the Strobe_in_TR or the Strobe_out_TR signals is activated. The TCLK_in, Scan_En and CLK_Out signals are made available for the internal scan FFs of the IUT. Regular TAP/TRP ports are used for non-scan primary inputs and outputs of the IUT. The TACP instructions support shifting test data in, shifting test results out, or simultaneous shifting in and out of test data and results, respectively.



(a) k-bits wide test application port (TAP). **(b) l-bits wide test results port (TRP).**

Figure 3.11 : Test application port (TAP) and test results port (TRP).

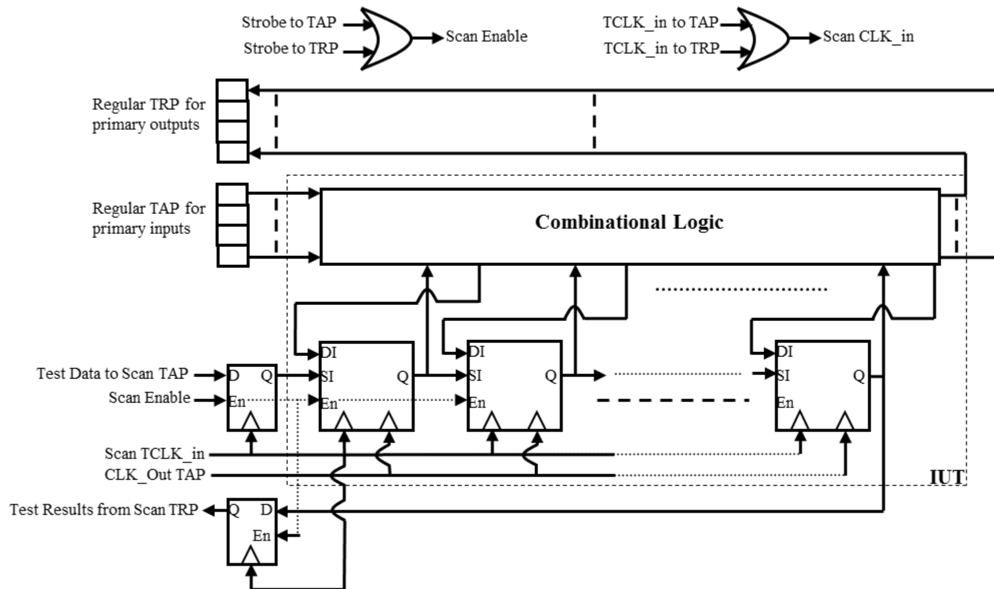


Figure 3.12 : Scan test application/result ports.

3.2 User Interface Software

This software enables the user to fully control the testing process. It provides the user with complete interfaces for writing and editing programs, downloading program and test data to memories, uploading programs, test data and test results from memories, reading register contents, sending control signals to reset registers, reset program execution, set a break point, edit the value of an address registers, start or stop running the current program.

Any communication media can be used; Ethernet, USB or serial port (i.e. UART port). The user interface and the test processor can are communicating through a serial cable. A communication protocol is proposed to be implemented in both sides. The protocol take care of downloading and uploading from memories. It also forwards control signals from the user to the processor. It gives a high level of abstraction to facilitate the processor design and the user interface design.

Interface Protocol

The user interface has to implement the user interface protocol which will be also implemented on TACP (a hardware version). The protocol defines 19 commands as listed and described in Table 3.1. They are categorized into four groups; loading registers commands, downloading to memories commands, uploading form memories requests and control commands. The communication unit is the packet. The user sends variable-size packets each packet starts with type byte that defines the packet type and length. A graphical representation of the implemented protocol structure is depicted in Figure 3.13.

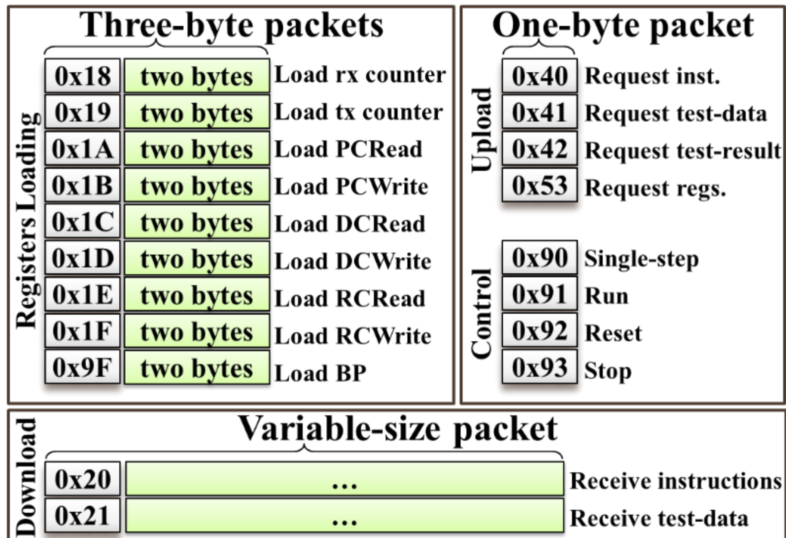


Figure 3.13 : Packet type list. Each packet starts with flags defining a command and determines packet size.

Table 3.1: Communication protocol - the available user commands with their codes.

Class	No.	Command	Description
Loading	1	Load rx counter	Set a new value to the receiving counter.
	2	Load tx counter	Set a new value to the transmitting counter.
	3	Load PCRead	Set the instruction memory reading address register.
	4	Load PCWrite	Set the writing address of the instruction memory.
	5	Load DCRead	Set the reading address of the test-data memory.
	6	Load DCWrite	Set the writing address of the test-data memory.
	7	Load RCRead	Set the reading address of the test-result memory.
	8	Load RCWrite	Set the writing address of the test-result memory.
	9	Load BP	Set new value to the break point register.
Download	10	Receive inst.	Receives instructions and stores them in the instruction memory. The count is determined by the receiving counter.
	11	Receive test data	Receives test-data and stores them in test-data memory. Received data size is determined by the receiving counter.
Upload	12	Request inst.	Requests to read data from instruction memory. Data size is determined by the transmitting counter.
	13	Request test data	Requests to read data from test-data memory. Data size is determined by the transmitting counter.
	14	Request test result	Requests to read data from test-result memory. Data size is determined by the transmitting counter.
	15	Request registers	Requests reading all register contents.
Control	16	Single Step	Sends a single-step control signal to execute one instruction.
	17	Run	Sends a run control signal to execute the rest of the program.
	18	Reset	Sends a reset control signal that clear all processor registers.
	19	Stop	Sends a stop control signal to stop the program execution.

3.3 Test and Characterizing Processor (TACP)

This Thesis focuses on the test and characterization processor (TACP). The test processor executes the program written and downloaded by the user through the user interface software. It is supported with memories and user communication unit. The processor can select an IP on the chip, send test data to IP inputs or scan chain, apply test data and read the captured results back from IP outputs on the chip. It can also set, decrease or increase the frequency of testing clock on the chip.

It has the ability to compare the results with the expected results and store the comparison in memory.

TACP consists of the processor, user communication unit, memories and memory multiplexer. Figure 3.14 shows the TACP which is a special purpose processor associated with memories and a user communication unit. It runs programs from its memory and produces the appropriate test signals to TSC on the chip. The user communication unit enables the user to access memories and read register contents.

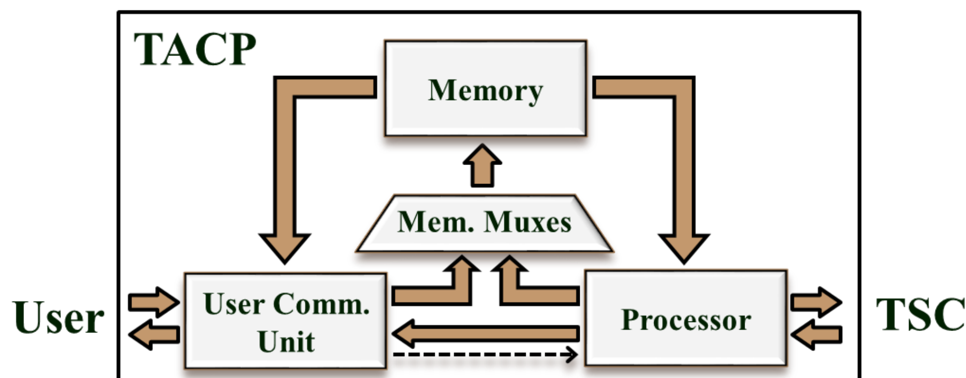


Figure 3.14 : TACP main components.

Processor

The processor is a microcode architecture that consists of sequencer and data path. The sequencer is the control unit of the processor that sequences the micro instruction execution. It only has five components; address register, comparator, incrementer and two multiplexers.

The control store is a ROM that stores all control signals for all execution cycles. Each entry contains the states of all data path control signals and info about the next address. In each clock cycle, the sequencer selects on entry that controls the data path and determine the next micro instruction address.

The data path contains all components that is needed to execute the instructions and controlled by the control signals from the control store. The data path components are as follows:

- **Test data shift register (TD)**: It shifts the test data out to the TSC. At the same time it shifts the old test data in that are coming from the TSC (returned back).
- **Test result shift register (TR)**: It shifts the test result in from the TSC.
- **Selection mask shift register (SM)**: It shifts in the old selection mask coming from the TSC. SM and TD are used as a loop back checkers.
- **Measured frequency shift register (FR)**: It shifts in the corresponding FR in the TSC. Its value indicates the measured frequency.
- **Frequency control word shift register (CW)**: It shifts out the frequency control word to the corresponding FR in the TSC.
- **User counter (UC)**: it is a 32-bit register that can be decremented, incremented, loaded from memory, stored in memory or loaded by immediate value. It is assigned with a zero flag so the user can use JZ and JNZ instructions that branches according to that flag. The ability to load store this register make it possible to use it for multiple counters.
- **Addressing multiplexers**: These multiplexers controls the new value that should be stored in address registers. An address register can be loaded with immediate value,or incremented.
- **Enumerating multiplexer**: The user can read the contents of some registers by sending a request containing the register number. The enumerating multiplexer outputs the selected register value as a response to the user request.
- **Instruction Register**: Holds the current executing instruction opcode.
- **Stack register**: The processor supports calling subroutines. It uses the bottom of the instruction memory as a stack to push/pop program counter.
- **Break point register**: The program execution stop if the program counter match the break point register.
- Flags.

Memories

The platform also contains memories to store instruction, data and results. To facilitate memory management, three independent memories are suggested as shown in Figure 3.15. Each memory has two 16-bit address ports one of them is a writing port.

- **Instruction memory**: It stores the program and the stack.
- **Test data memory**: It is used for test data and expected result.
- **Test result memory**: It is used for the test result and comparison result.

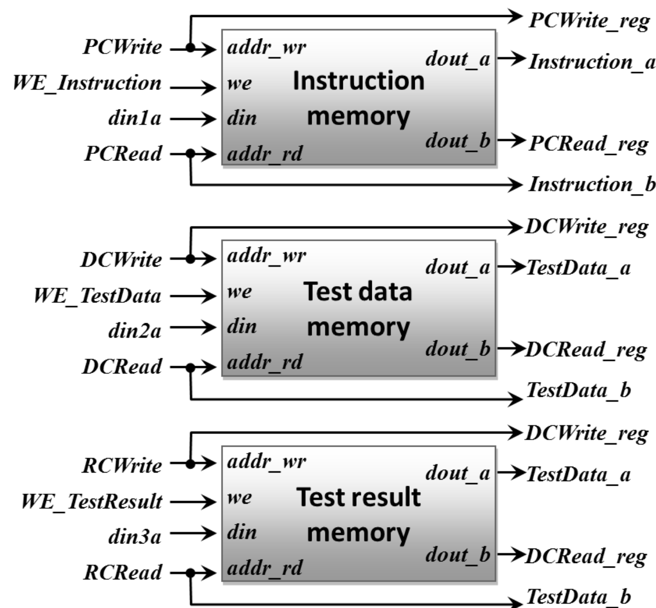


Figure 3.15 : The three memories. Each memory has four inputs and four outputs.

User Communication Unit

User communication unit implements a hardware version of the user interface protocol that has the 19 commands mentioned earlier in Table 3.1. A graphical representation of the implemented protocol structure is depicted in Figure 3.13. The user communication unit has these components:

- **Communication media unit**: This unit deals with the communication physical details and totally depends on the communication media type. For each communication media (i.e. UART, USB, Ethernet, etc.) there is a different version. This unit simply converts signals (bits) to bytes and vice versa to facilitate the protocol implementation that only deals with packets and bytes.
- **Receiving Transmitting state machines**: The protocol is designed in hardware as two finite state machines (FSMs).
- **Receiving counter:** The receiving counter is used with variable length packets to count down received instructions and data.
- **Transmitting counters**: The transmitting counter is used to count the data sent to the user.
- **Addressing multiplexers**: Accessing memories requires incrementing address registers. The user can also set a direct value to any address register. These multiplexers manage modifying the address registers.
- **Flags**: Flags supports the state machine by storing the encoding of the current user command.

CHAPTER 4

DESIGN OF THE TEST AND CHARACTERIZATION PROCESSOR

In this chapter, the implemented design of test and characterization processor (TACP) is illustrated in details. The chapter starts by discussing some main instructions design. Then the TACP top level diagram is illustrated showing its components. Then each component has its own block and subcomponents illustrated in details and so on. The block diagram of each component is viewed and followed by its signals definitions. Each signal appears in this chapter has its unique name so signals can be matched between figures. If two signal appears in many figures and have exactly the same name that means these signals are connected together. The last section in this chapter illustrates the design of the four main instructions. The explanations are supported by flowchart and micro instructions.

4.1 Instruction Design and Microinstructions

Thirty three instructions are designed. They can be distinguished into five groups. They are listed in Table 4.1 with their names and opcodes. Appendix A, page 115 has the detailed descriptions of all instructions, their parameters and their micro-instruction

Table 4.1: Instructions and their opcodes

	Testing Instructions	opcode		Branch Instructions	opcode
1	SendSelectionMask	28	19	JCompareError	08
2	SendTestData	29	20	JCompareCorrect	09
3	ReadResult	23	21	Jump	11
4	ApplyAndCapture	01	22	Call	02
5	Load_DCRead	13	23	Return	26
6	Load_RCRead	15			
7	Load_RCWrite	16		Counter Instructions	opcode
8	ClearTestDataRegister	35	24	Load_UserCounter_value	18
9	ResetCompareFlag	24	25	Load_UserCounter_Mem	17
10	Compare	03	26	Store_UserCounter	33
			27	INC_UserCounter	07
	Frequency Instructions	opcode	28	DEC_UserCounter	05
11	SetFrequencyControlWord	30	29	JNZ	10
12	SendFrequencyControlWord	27	30	JZ	12
13	MeasureFrequency	20			
14	ReadFrequencyRegister	22		Others Instructions	opcode
15	INC_CW	06	31	Stop	32
16	DEC_CW	04	32	Nop	21
17	SetHFClock	31	33	Fetch	00
18	ResetHFClock	25			

- **Testing instructions**: The processor needs instructions to send test vectors to the chip, apply them and read test result back. This involves memory addressing instructions to control reading and storing location in memory.
- **Frequency control instructions**: The processor needs instructions to control the frequency on the chip, measure, increase, decrease it and read the measured frequency from the chip.
- **Counter instructions**: The processor needs counters to do loops. For instance, it is required to repeat the test with different frequencies until getting compare error. Only one counter register is designed. However, its value can be stored in temporary memory and reloaded again. This allows using the counter register for multiple times in the same program.
- **Branching instructions**: the processor design allows loops, calling subroutines and conditional/unconditional branching. It uses the instruction memory as stack starting from the bottom.
- **Other instructions**: such as stop executing the program, NOP.

The most important instructions are explained in the following sub sections.

4.1.1 SendSelectionMask instruction design

The on-chip circuitry has multiple IUTs. Each IUT has multiple ports. Each port has an enable bit in the selection register that can be set using this instruction. The circuit decodes the selected port number. SendSelectionMask instruction decodes a selected port number. It has two zero-based parameters, the selection window length and the selected port number. For example, suppose we have for IUTs each IUT has three ports and the ports are serialized as depicted in Figure 4.1. SendSelectionMask B, 3 selects port number four by sending a 12-bit stream 0000000000100, SendSelectionMask B, 7 selects port eight 000010000000 and so on.

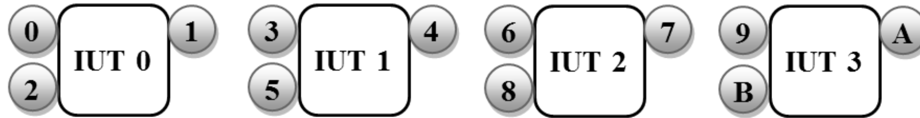


Figure 4.1 : Four IUTs, each has three ports, each port has a serial number.

To select more than one port at a time, the instruction has to be called multiple times with different window lengths. SendSelectionMask B, 7 followed by SendSelectionMask 6, 0 results in sending 0000001_000010000000. The latest sent 12-bits are 0000001_00001 which means selecting ports 0 and 6.

The microinstruction and flowchart of the instruction is show in Figure 4.2. The instruction starts by loading instruction parameters. It increments the program counter PC twice to load two bytes from instruction memory to the lower part of the 32-bit register CR. Then it loads another two bytes to the higher part of CR register. It then increment the program counter PC for the fifth time to have the second parameter, the selected port number. After then the instruction loops and decrements CR until it reach zero. The selection mask circuitry outputs one when CR equals the selected port number, otherwise it outputs zero.

SendSelectionMask microinstructructions:

```
1C:0 Increment_PC
1C:1 Increment_PC
1C:1 Load_CR_Low_Instruction2
1C:2 Increment_PC
1C:3 Increment_PC
1C:3 Load_CR_High_Instruction2
1C:4 Increment_PC
1C:5 Decrement_CR
1C:5 Strobe_in_PMask
1C:5 if not (CR_IsZero)  Branch 1C:5
1C:6 Increment_PC
```

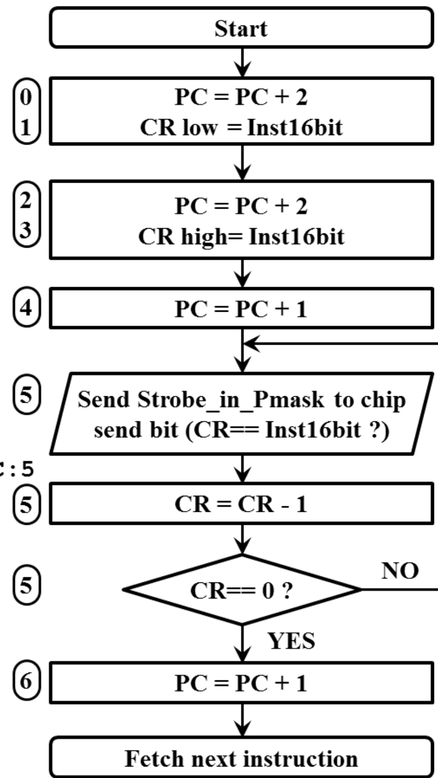


Figure 4.2 : SendSelectionMask microinstructions and flow chart.

4.1.2 SendTestData instruction design

SendTestData instruction reads data from test data memory and sends them as one serial bitstream with strobe signal to TSC. Load_DCRead instruction needs to be invoked before start sending to determines data address. It has two parameters, the number of bytes and the number of bits minus three. It always sends three bits at least. For example, SendTestData 0, 3 load one byte from test data memory sends six bits, SendTestData 5, 5 load 6 bytes and send eight bits from each, SendTestData 3, 0 load 4 bytes and sends six bits from each.

Figure 4.3 shows the micro instructions and flowchart of this SendTestData associated by clock number to show the concurrent microinstructions. It starts by loading four bytes to CR, then it loads TD register, set number of bits in WC register, and increments DCRead. There are two nested

loops, the inner loop sends bits to TSC while the outer loop load bytes from memory. The algorithm is designed to generate a continuous bitstream with no stops between bytes.

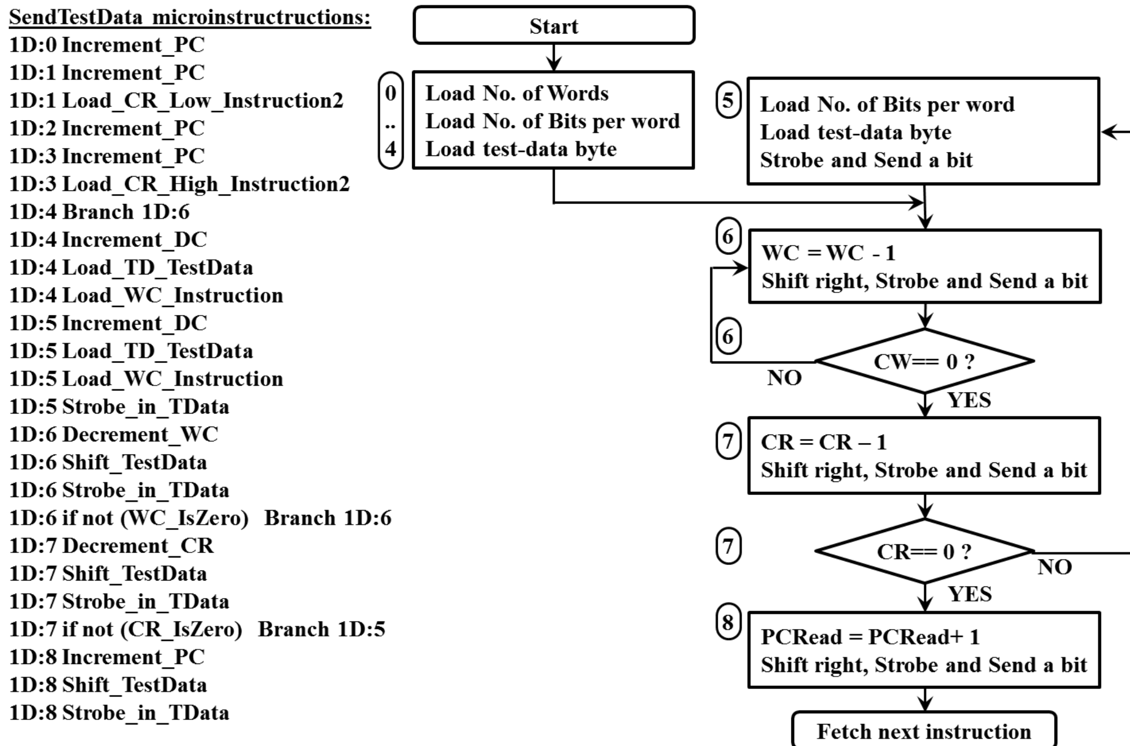
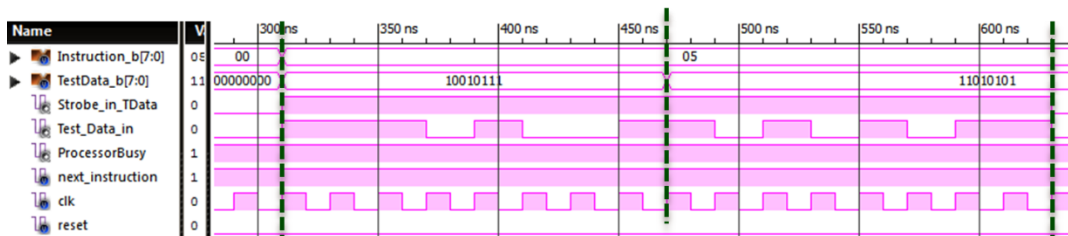
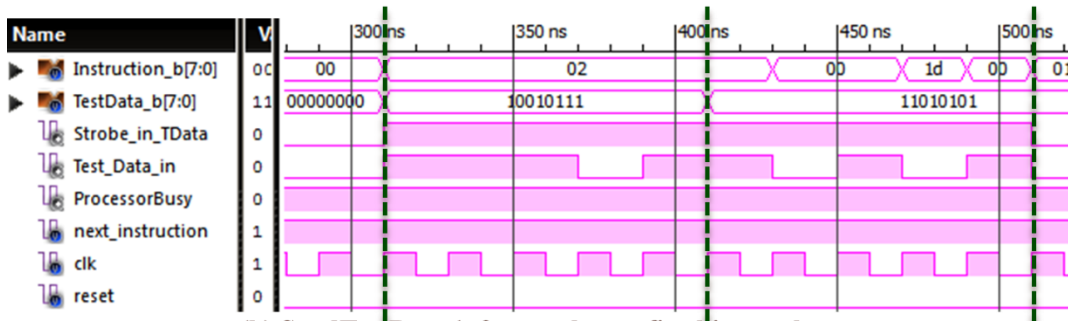


Figure 4.3 : SendTestData microinstructions and flow chart.

Figure 4.4-a shows a simulation of calling SendDataInstruction with the parameters 1 and 5. This means to read two bytes from the memory and to send eight bits from each byte. The simulation shows that the strobe stay high for sixteen clock cycles. The test data (i.e. 97 and D5) are sent in this period through the Test_Data_in output pin. Figure 4.4-b shows another implementation of the instruction with the parameters 1 and 2. This will read two bytes from the memory and send five bits from each byte.



(a) SendTestData 1, 5 – two bytes, eight bits per byte



(b) SendTestData 1, 2 – two bytes, five bits per byte

Figure 4.4 : Simulation of the instruction SendTestData

4.1.3 ReadResult instruction design

ReadResult instruction sends strobe signal to TSC and receives the test result. It stores the received bytes in test result memory. Load_RCWrite instruction needs to be invoked to determine the memory location. It has two parameters the number of bytes to be stored in memory and the number of bits per byte. For example ReadResult 3, 4 reads 20 bits and stores them as four bytes in test result memory four bits in each.

Figure 4.5 below shows microinstructions and flowchart of ReadResult. It starts by loading CR with 32 bit value to be used by the outer loop and indicates the number of bytes. The inner loop sends strobe and shifts in test results coming from TSC to TR register. WC register is used as number of bits and a counter for the inner loop.

Micro instructructions:

```
17:0 Increment_PC
17:1 Increment_PC
17:1 Load_CR_Low_Instruction2
17:2 Increment_PC
17:3 ClearTR
17:3 Increment_PC
17:3 Load_CR_High_Instruction2
17:4 Load_WC_Instruction
17:4 Strobe_out_TR
17:5 Decrement_WC
17:5 Strobe_out_TR
17:5 if not (WC_IsZero)  Branch 17:5
17:6 Decrement_CR
17:6 Increment_RCWrite
17:6 Load_WC_Instruction
17:6 Store_TestResults_TR
17:6 Strobe_out_TR
17:6 if not (CR_IsZero)  Branch 17:5
17:7 Increment_PC
```

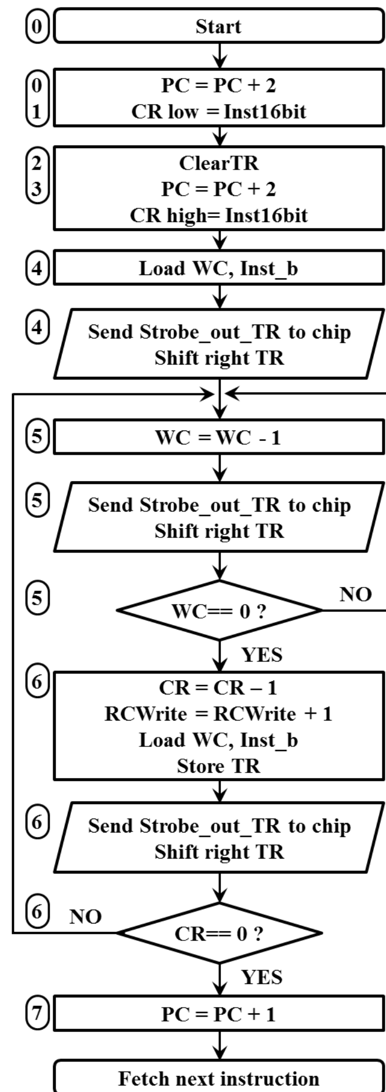


Figure 4.5 : ReadResult microinstructions and flow chart.

4.1.4 Compare instruction design

Compare instruction compares (i.e. XORing) between test data memory block and test result memory block and stores results in test result memory. The addresses needs to be determined by invoking instructions Load_DCRead, LoadRCRead, and Load_RCWrite. The instruction affects CF flags. It makes CF high when it detects a discrepancy. Figure 4.6 shows microinstructions and

flowchart. The instruction keeps incrementing addresses and stores the compare results into test result memory.

microinstructructions:

```
03:0 Increment_PC
03:1 Increment_PC
03:1 Load_CR_Low_Instruction2
03:2 Increment_PC
03:3 Increment_PC
03:3 Load_CR_High_Instruction2
03:4 Decrement_CR
03:4 Increment_DC
03:4 Increment_RCRead
03:4 Increment_RCWrite
03:4 Store_TestResults_Compare
03:4 if not (CR_IsZero)  Branch 03:4
```

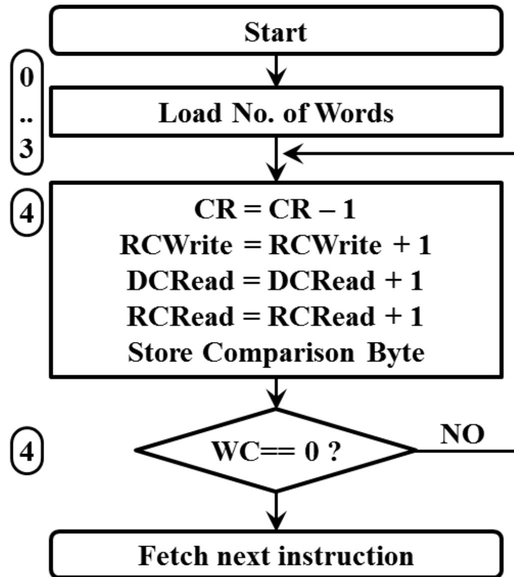


Figure 4.6 : Compare microinstructions and flow chart.

4.2 TACP Top Level Design

The TACP black box is depicted in Figure 4.7 showing pin names that represents the interfaces with the user interface and with the test support circuitry (TSC). TACP is connected to the user interface through two serial pins rx and tx. It is then connected to the TSC through 19 pins (5 inputs and 14 outputs). In addition, it may be connected to additional external control signals such as reset, run and single-step that the user can access them directly if needed.

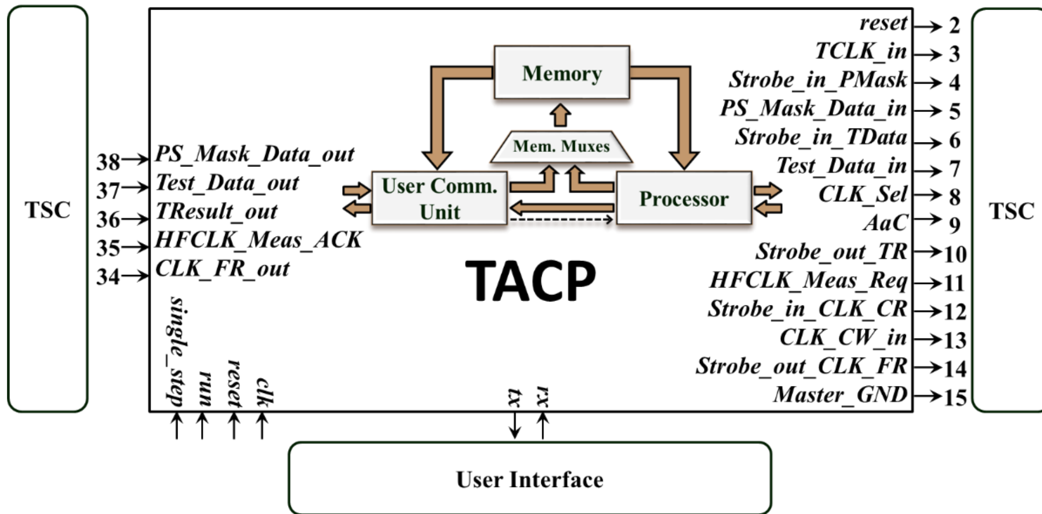


Figure 4.7 : TACP design top view and its subcomponents.

4.3 User Communication Unit

The communication protocol defines interchanging information rules between the user and the TACP processor. The communication unit implements the protocol at the TACP side while the user interface software implements the protocol at the user side. There are nineteen commands the user could issue each has its unique byte code, as listed and described in Table 3.1.

The user communication unit is attached with UART module that deals with physical communication issues through the COM serial port. The communication protocol circuitries include two state machines to control the receiving and transmitting operations with the user, 16-bit down-counter for receiving, 16-bit down-counter for transmitting, memory addressing multiplexing circuitries, break point register, previous-byte register and flags. The communication unit receives and responds to user commands through the UART module, Figure 4.8. It can reach memory, registers and other components.

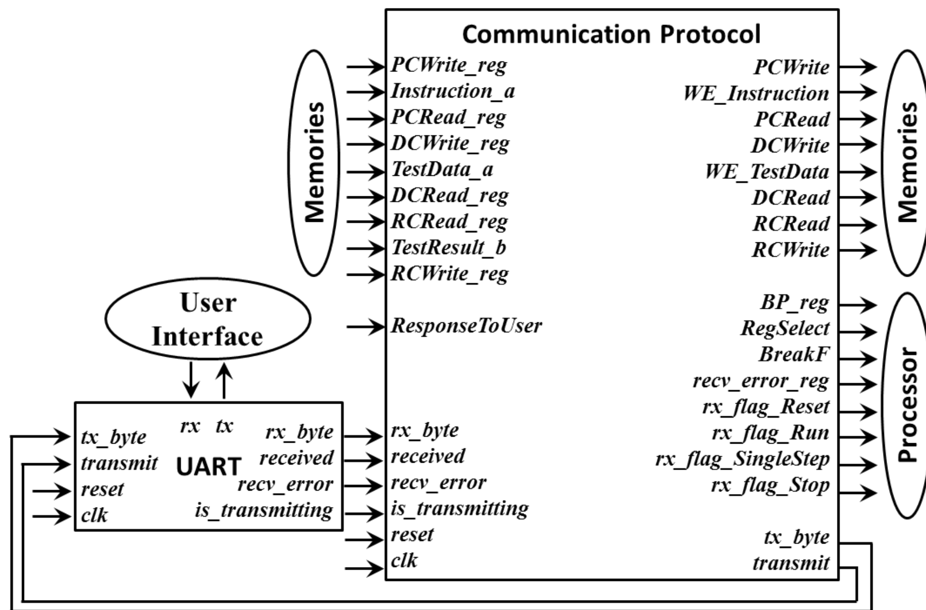


Figure 4.8 : Communication protocol connected to UART module.

4.3.1 UART module

The UART module does the physical communication between the user and the processor. It uses COM port for communication. The communication speed is fixed and can be controlled by setting the CLOCK_DIVIDE parameter according to Table 4.2 below. The table comes from the equation $\text{CLOCK_DIVIDE} = \frac{\text{clock rate}}{4\times\text{baud rate}} = \frac{50*10^6}{4\times\text{baud rate}}$. The chosen baud rates are the standard baud rates of the COM port. The implemented version works at 57600 baud rate which has the least error margin as it is shown in Table 4.2, row 7.

Table 4.2: Communication protocol – UART communication speed.

	CLOCK_DIVIDE	Baud	Error
1	10417	1200	0.667
2	5208	2400	0.333
3	2604	4800	0.167
4	1302	9600	0.083
5	651	19200	0.041
6	325	38400	0.520
7	217	57600	0.013
8	108	115200	0.506

UART module is located at the middle between the user and the processor. It does the physical communication so it corresponds to the COM driver on the user PC.

UART module has two state machines for receiving and transmitting bits with the user side. It receives bytes from the COM port serially and sends them with a receive strobe with each byte to the communication protocol. It also takes bytes from the communication protocol and sends them serially while rising a transmission busy signal. Figure 4.8 shows a black box of UART module that uses the 15-pin serial communication port.

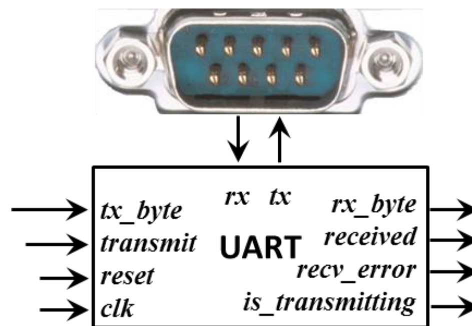


Figure 4.9 : The UART module input/output diagram.

- **rx:** the received bit from the serial port.
- **tx:** the transmitted bit to the serial port.
- **tx_byte:** Byte to be transmitted to the user interface through the UART module.
- **transmit:** The communication protocol module has to set this signal high when it needs to transmit a byte to the user interface.
- **rx_byte:** The received byte from the user interface through the UART module.
- **received:** The UART module sets this signal high whenever it finishes receiving a byte.
- **rcv_error:** This bit indicates if there is an error on receiving the current byte.
- **is_transmitting:** The UART module sets this signal high when it is transmitting a byte to indicate that the transmitting line is busy. When the protocol intends to send a byte, it has to wait until is_transmitting becomes low.
- **clk:** it has to be 50 MHz clock signals so the UART module can work as illustrated in section 4.3.1, page 40 and Table 4.2, page 40.
- **reset:** resets the transmitting/receiving state machines of the UART module.

4.3.2 Communication flags

The receiving state machine starts its work by receiving one of the known type-byte. Each type-byte represents a user command and has its unique control bits as shown in Table 4.3. When the receiving state machine receives a valid type-byte, its bits are decoded to eighteen flags that controls the transmitting and receiving state machines, Figure 4.10.

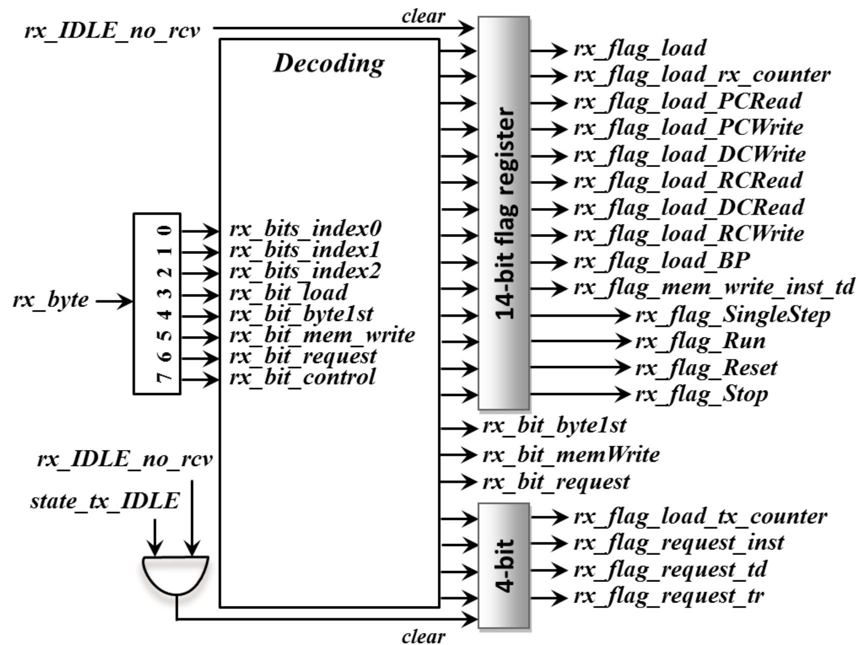


Figure 4.10 : Decoding the received type-byte to eighteen flags.

The flags decoder has three inputs rx_byte, rx_IDLE_no_rcv and state_tx_IDLE:

- **rx_IDLE_no_rcv**: indicates that the the receiving state machine is at state IDLE and no byte is received. It is used as a clear signal for all flags.
- **state_tx_IDLE:** indicates that the transmitting FMS is at IDLE state.
- **rx_byte:** The received byte from the user interface through the UART module. It has the following bits which are mentioned in Table 4.3.
    - **rx_bits_index**: three bits reserved for indexing as shown in Table 4.3.
    - **rx_bit_load**: this bit indicates that the user is sending two bytes to be loaded to one of the nine registers (i.e. rx counter, tx counter, BP, PCRead, DCRead, RCRead, PCWrite, DCWrite or RCWrite) according to Table 4.3.
    - **rx_bit_control**: this bit indicates that the user is sending a control signal. There are four control signals as shown in Table 4.3: single-step, run, reset and stop.

- **rx_bit_byte1st**: this bit set the receiving FSM to state_byte1st which indicates receiving the first byte of a register contents. It is also used if a control signal will be received.
- **rx_bit_mem_write**: this bit the moves the receiving state machine to state_LOOP. It indicates that the user is writing data either to instruction memory or to test-data memory.
- **rx_bit_request**: this bit indicates that the user is requesting on of the four available requests shown in Table 4.3 (i.e. reading data from one of the three memories or reading the contents of the enumerated registers).

The output of the circuitry is decoded flags which are explained in the following:

- **rx_flag_load:** This flag indicates that the user is sending two bytes to be loaded to one of the nine registers (i.e. rx counter, tx counter, BP, PCRead, DCRead, RCRead, PCWrite, DCWrite or RCWrite) according to Table 4.3.
- **rx_flag_load_rx_counter:** the current context is loading new value to rx counter.
- **rx_flag_load_tx_counter:** the current context is loading new value to tx counter.
- **rx_flag_load_PCRead:** the current context is loading new value to PCRead, the instruction memory reading address register.
- **rx_flag_load_PCWrite:** the current context is loading new value to PCWrite, the instruction memory writing address register.
- **rx_flag_load_DCWrite:** the current context is loading new value to DCWrite, the test-data memory writing address register.
- **rx_flag_load_RCRead:** the current context is loading new value to RCRead, the test-result memory reading address register.
- **rx_flag_load_DCRead:** the current context is loading new value to DCRead, the test-data memory reading address register.
- **rx_flag_load_RCWrite:** the current context is loading new value to RCWrite, the test-result memory writing address register.
- **rx_flag_load_BP:** the current context is loading new value to BP_reg, the break point register.
- **rx_flag_request_inst:** the user requests reading from instruction memory.
- **rx_flag_request_td:** the user requests reading from test-data memory.
- **rx_flag_request_tr:** the user requests reading from test-result memory.
- **rx_flag_mem_write_inst_td:** if it is high indicates that the user requests writing test-data. If it is low, indicates that the user requests writing instructions.
- **rx_flag_SingleStep:** a control flag which is raised for one clock cycle only. It asks the processor to execute the current instruction.

- **rx_flag_Run:** a control flag which is raised for one clock cycle only. It sets the RunF flag which give the processor a green light to continue executing all program instructions.
- **rx_flag_Reset:** This control flag send a reset signal to all registers in the processor.
- **rx_flag_Stop:** This control signal resets the RunF flag and therefore stop running the current program. It starts the single-step mode in which the user controls when to execute the next instruction.

4.3.3 Previous received byte register

Prev_rx_byte register saves the current received byte. It is used to combine two consecutive received bytes and forms a 16-bit word as shown in Figure 4.11. This 16-bit word can be loaded at once to any 16-bit register. It is useful for receiving addresses and loading counters.

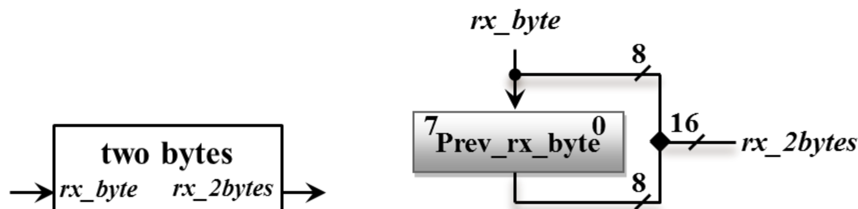


Figure 4.11 : Combining previous received byte with the current received byte to form a 16-bit word.

- **rx_2bytes:** the latest received two bytes.
- **rx_bytes:** the latest received byte.

4.3.4 Transmitted byte multiplexers

Three multiplexers to determine which data line is connected to the user response byte. User can request one of the four available requests; reading data from one of the three memories or reading the contents of the enumerated registers.

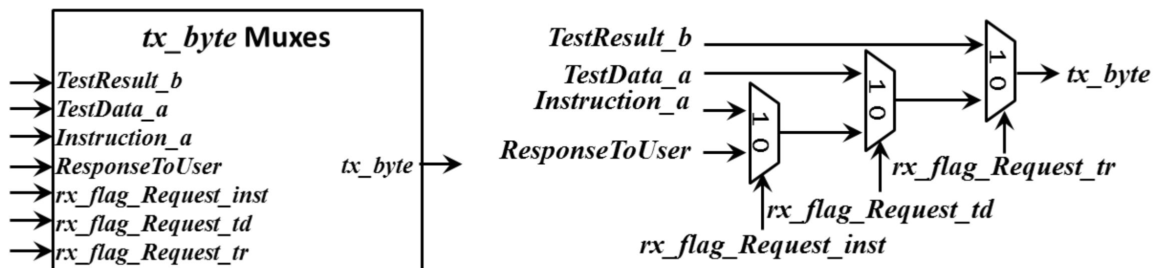


Figure 4.12 : tx_byte multiplexers.

- **TestResult_b:** the 2nd data port of test result memory accessed by the address port RCRead.
- **TestData_a:** the 1st data port of test data memory accessed by the address port DCWrite.
- **Instruction_a:** the 1st data port of instruction memory accessed by address port PCWrite.
- **rx_flag_Request_inst**, **rx_flag_Request_td** , **rx_flag_Request_tr:** explained in page 43.
- **tx_byte:** Byte to be transmitted to the user interface through the UART module.

4.3.5 Receiving state machine (rx_FSM)

The receiving state machine receives and implements the user commands stated in Table 4.3. Command codes are designed in such a way to facilitate the hardware design of the transmitting and receiving state machine.

Table 4.3: Communication protocol - the available user commands with their codes.

No.	class	Command	code	Code	control	request	mem_write	byte1st	load	index [2:0]
1	Loading	Load rx counter	18	00011000				1	1	000
2		Load tx counter	19	00011001				1	1	001
3		Load PCRead	1A	00011010				1	1	010
4		Load PCWrite	1B	00011011				1	1	011
5		Load DCRead	1C	00011100				1	1	100
6		Load DCWrite	1D	00011101				1	1	101
7		Load RCRead	1E	00011110				1	1	110
8		Load RCWrite	1F	00011111				1	1	111
9		Load BP	9F	10011111	1			1	1	111
10	Down load	Receive inst.	20	00100000			1			000
11		Receive test data	21	00100001			1			001
12	Upload	Request inst.	40	01000000		1				000
13		Request test data	41	01000001		1				001
14		Request test result	42	01000010		1				010
15		Request registers	53	01010011		1		1		011
16	Control	Single Step	90	10010000	1			1		000
17		Run	91	10010001	1			1		001
18		Reset	92	10010010	1			1		010
19		Stop	93	10010011	1			1		011

Figure 4.13 shows its FSM diagram. It starts in the IDLE state waiting for receiving the type-byte from the UART module. If the type-byte indicates a memory write, it goes to LOOP state, starts receiving, storing bytes and decrements the receiving counter until it reaches zero. If the type-byte indicates a control signal or loading register, it goes to the 1st BYTE state. On this state, if the type-byte indicates a control signal, it returns to the IDLE state directly. If it is a loading command, it waits for receiving a byte and goes to the 2nd BYTE to receive another byte to form a complete 16-bit word. It then loads the 16-bit value to the register specified by the type-byte and return to the IDLE state. Finally, if the type-byte indicates a request for reading memories or registers, the receiving state machine stay in IDLE state and trigger the transmitting state machine to handle the request. The receiving state machine is a 2-bit state machine. Its circuitry is shown Figure 4.14.

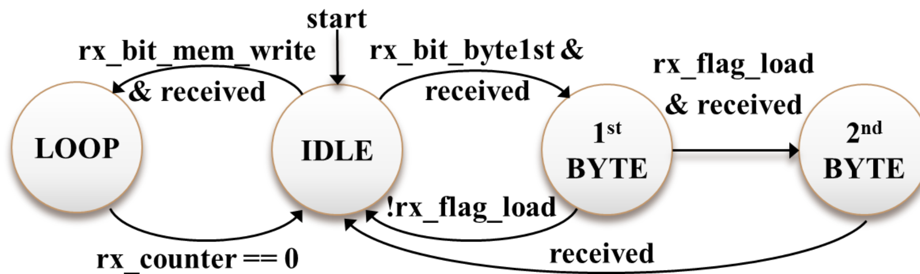


Figure 4.13 : Receiving state machine FSM diagram.

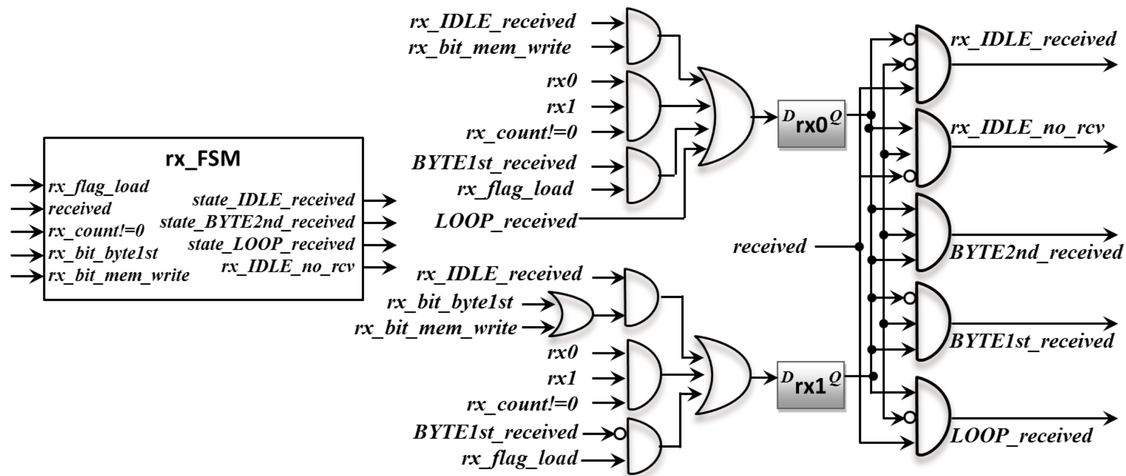


Figure 4.14 : Receiving state machine circuitry.

- **rx0 and rx1:** These are two flip-flops to store the FSM current state.
- **rx_flag_load:** This flag indicates that the current context is loading to a register.
- **received:** indicates that a byte has been received by the UART module.
- **rx_count!=0**: indicates that the receiving counter has a non-zero value.
- **rx_bit_byte1st**: this bit indicates whether the current context is receiving a control signal or loading the lower byte to a 16-bit register.
- **rx_bit_mem_write**: indicates that the current context is loading to memory.
- **rx_IDLE_no_rcv**: the receiving state machine is at state IDLE and no byte is received now.
- **rx_IDLE_received**: the receiving state machine is at state IDLE and a new byte is received now.
- **state_BYTE1st_received:** the state machine is at state BYTE1st and a new byte is received.
- **state_BYTE2nd_received:** FSM is at state BYTE2nd and a new byte is received.
- **state_LOOP_received:** the state machine is at state LOOP and a new byte is received.

4.3.6 Transmitting state machine (tx_FSM)

The transmitting state machine responds to the user requests and sends him the requested information. There are four types of requests. This includes three read requests from the three memories. The fourth one is a request to send all enumerated register and flag values to the user. The transmitting state machine is triggered by the receiving state machine. It has three states. It starts with the IDLE state and waits until it is triggered and receives a request, then it moves to the BUSY state. Then, the state machine loops in the BUSY state waiting until the transmission line

become free, then it moves to the TRANSMIT state. In the transmit state it decrements the transmitting counter and loops while the transmitting line is free. When the state machine reaches the TRANSMIT state, a byte will be transmitted, the transmitting line will become busy again and the state machine will return beck to the BUSY state. At any state, if the transmitting counter reaches zero it return directly to the IDLE state.

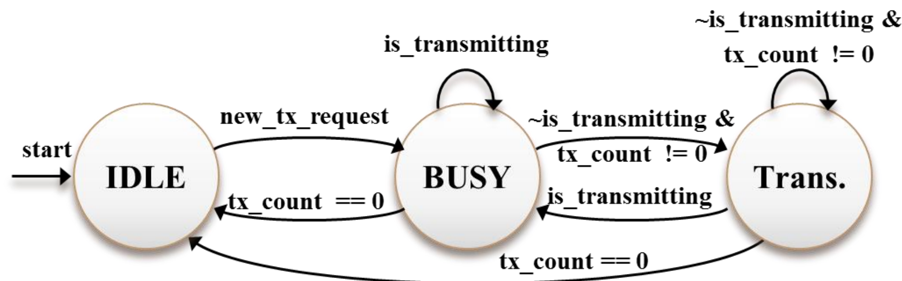


Figure 4.15 : transmitting state machine sends test result and register values to the user.

The transmitting state machine is a 2-bit FSM. Its circuitry is shown in Figure 4.16.

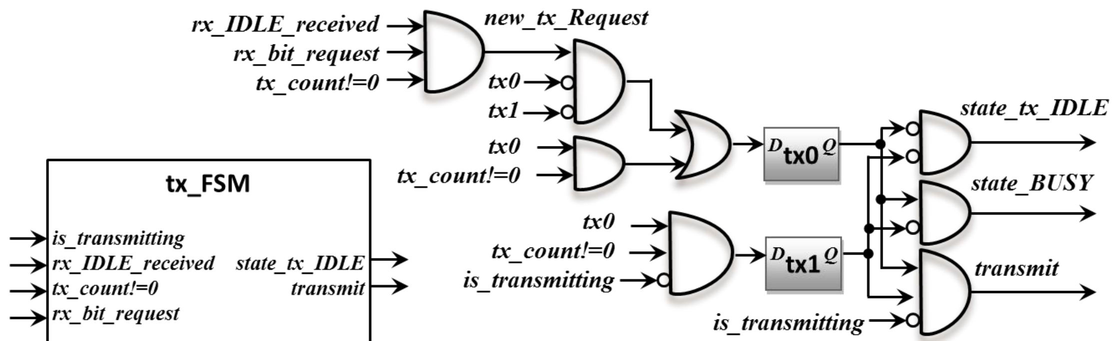


Figure 4.16 : transmitting state machine circuitry and the transmit signal.

- **is_transmitting**: indicates whether the transmitting line is busy or not.
- **rx_IDLE_received:** the receiving state machine is at IDLE state and a new byte is received.
- **tx_count!=0**: indicates that the transmitting counter has a non-zero value.
- **rx_bit_request**: this bit indicates that the user is requesting.
- **new_tx_request**: start the transmitting process if it is not busy with a previous request.
- **state_tx_IDLE:** indicates that the transmitting state machine is at IDLE state.
- **state_BUSY:** indicates that the transmitting state machine is at BUSY state.
- **transmit:** indicates that there is a byte ready to be transmitted to the user interface.

4.3.7 Break point register

The break point register enables the user to stop running the program at specific point. It stores the instruction address where the program execution has to stop. The break point register circuitry includes a comparator between the break point register and the program counter, Figure 4.17. The comparator result named BreakF, is ORed later with the stop control signal to the processor to stop the program execution.

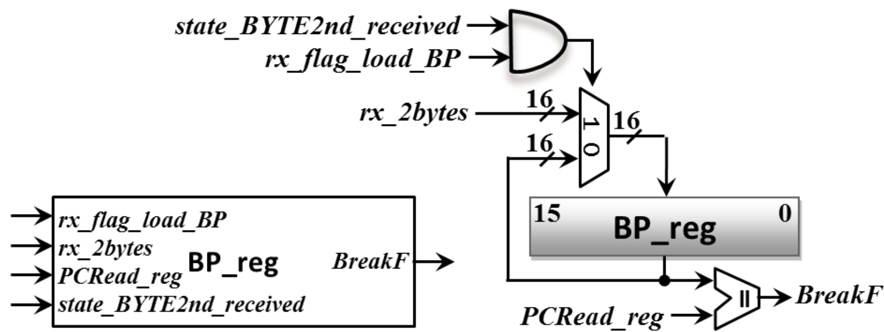


Figure 4.17 : Break point register circuitry and BreakF flag.

- **rx_flag_load_BP:** indicates that the context is loading to the break point register.
- **rx_2bytes:** 16-bit value to be loaded to the break point register.
- **state_BYTE2nd_received**: indicates that the 16-bit value is received and ready.
- **PCRead_reg**: the program counter.
- **BreakF**: the break point flag, it is the comparator result between BP and PCRead.

4.3.8 Receiving counter

The receiving counter is a down counter used for downloading instructions or test-data from the user interface to the corresponding memory. The user has to initialize the counter value before the downloading request to determine the number of bytes to be downloaded. The counter circuitry contains a zero flag that indicates whether the counter reaches zero, Figure 4.18.

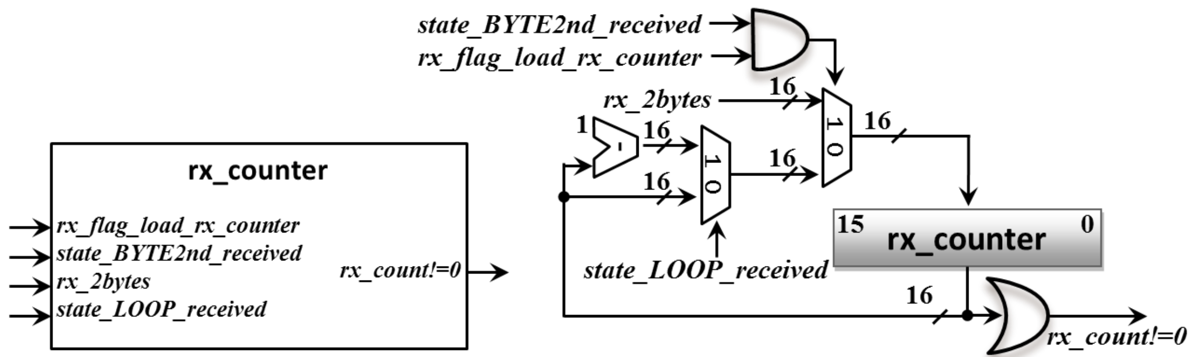


Figure 4.18 : The receiving counter and its circuitry.

- **rx_flag_load_rx_counter**: is loading value to rx counter.
- **state_BYTE2nd_received**: indicates that 16-bit value has been recieved.
- **rx_2bytes**: the latest received two bytes.
- **state_LOOP_received:** the receiving context is downloading to memory and a new byte is received.
- **rx_count!=0**: indicates that the receiving counter has a non-zero value.

4.3.9 Transmitting counter

This down counter is used by the transmitter to upload a specific number of bytes to the user. The user has to set its value and also to initialize the memory address before the uploading request. The counter is decremented and the address is incremented automatically with each sent byte. The circuitry also contains a zero flag for the counter, Figure 4.19.

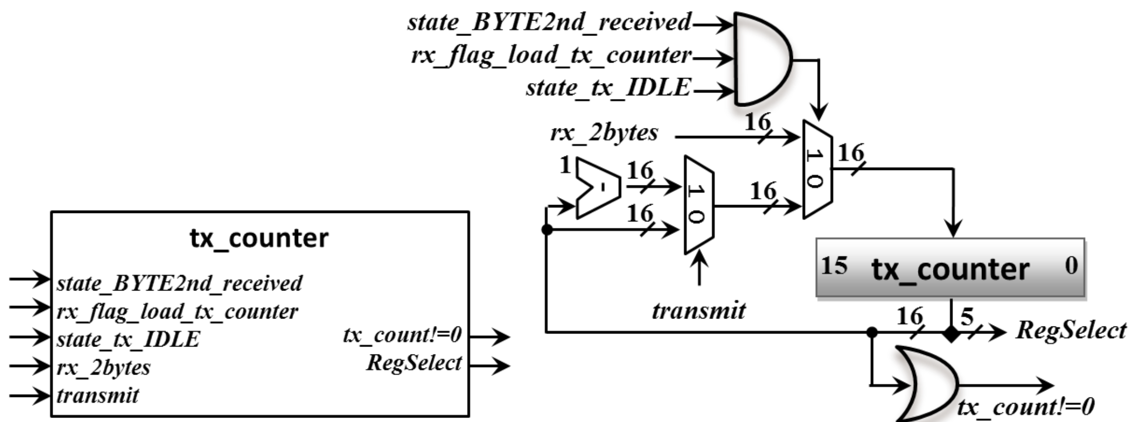


Figure 4.19 : The transmitting counter and its circuitry.

- **state_BYTE2nd_received**: indicates that the receiving FSM has received 16-bit.
- **rx_flag_load_tx_counter**: indicates that the user requests loading value to tx counter.
- **state_tx_IDLE:** indicates that the transmitting FMS is at IDLE state.
- **rx_2bytes**: the latest received two bytes.
- **transmit**: rises when the user is requesting and the transmitter is ready for transmission.
- **tx_count!=0**: indicates that the transmitting counter has a non-zero value.

4.3.10 Communication Error Flag

Error flag is a communication flag that accumulates the UART module receiving errors. It reflects communication error for each sending operation. It is cleared with each new user command, Figure 4.20.

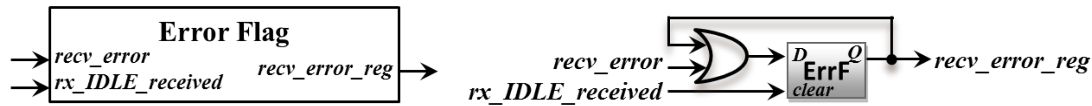


Figure 4.20 : Error flag circuitry.

- **recv_error:** indicates if there is an error on receiving the current byte. This signal is coming from the UART module.
- **rx_IDLE_received:** the idle transmitting state clear the flag.
- **recv_error_reg**: the accumulated recv_error for a complete command.

4.3.11 PCWrite circuitry

PCWrite is the first instruction memory address port which is a write/read port. PCWrite circuitry allows the user to load to the PCWrite register. The register is also incremented when the user reads/writes from instruction memory. Write enable signal is also raised when writing to instruction memory. The circuitry is shown in Figure 4.21.

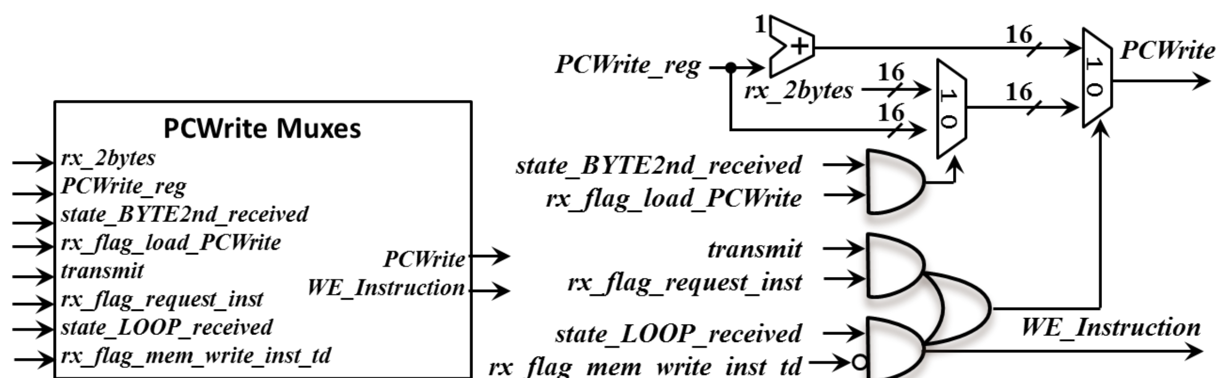


Figure 4.21: PCWrite circuitry in the communication protoocol.

- **PCWrite_reg**: The old value of the PCWrite register.
- **rx_2bytes**: the latest received two bytes.
- **state_BYTE2nd_received**: indicates that the receiving FSM has completed receiving 16-bit.
- **rx_flag_load_ PCWrite**: indicates that the user requests loading value to PCWrite.
- **transmit**: indicates that the UART module starts transmitting a byte to the user.
- **rx_flag_request_inst:** indicates that the current context is reading from instruction memory.
- **state_LOOP_received:** the state machine is at state LOOP and a new byte is received.
- **rx_flag_mem_write_inst_td:** if it is low, the writing bytes will be to instruction memory.
- **PCWrite**: The new value to be stored in the PCWrite register.
- **WE_Instruction**: the write-enable pin of the instruction memory.

4.3.12 PCRead circuitry

PCRead is the second test result memory address port which is a read only port. The PCRead is the program counter register. PCRead circuitry in the protocol can load new value to the register. This will enable the user to put an arbitrary value directly to the PCRead register to set the program execution point or even to restart the program. The circuitry is shown in Figure 4.22.

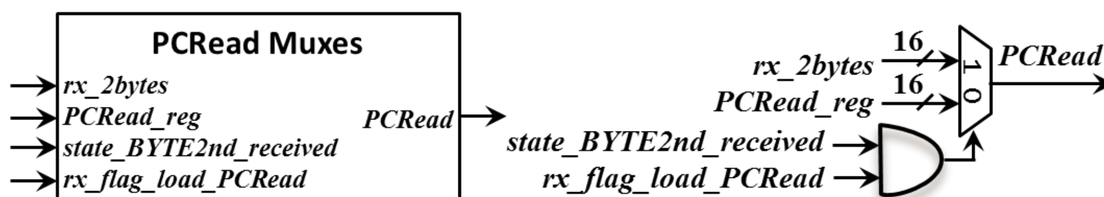


Figure 4.22 : PCRead circuitry in the communication protoocol.

- **rx_2bytes**: the latest received two bytes.
- **PCRead_reg**: The old PCRead register value.
- **state_BYTE2nd_received**: rises once the receive-state-machine receives 16-bit value.
- **rx_flag_load_PCRead**: indicates that the user requests loading new value to PCRead.
- **PCRead**: The new value of PCRead register.

4.3.13 DCWrite circuitry

DCWrite is the first test data memory address port which is a write/read port. Using the protocol, DCWrite can be incremented or loaded. It is loaded by the user but incremented when the user read/write from test-data memory. Write enable signal also raised when the user write to test-data memory. The circuitry is shown in Figure 4.23.

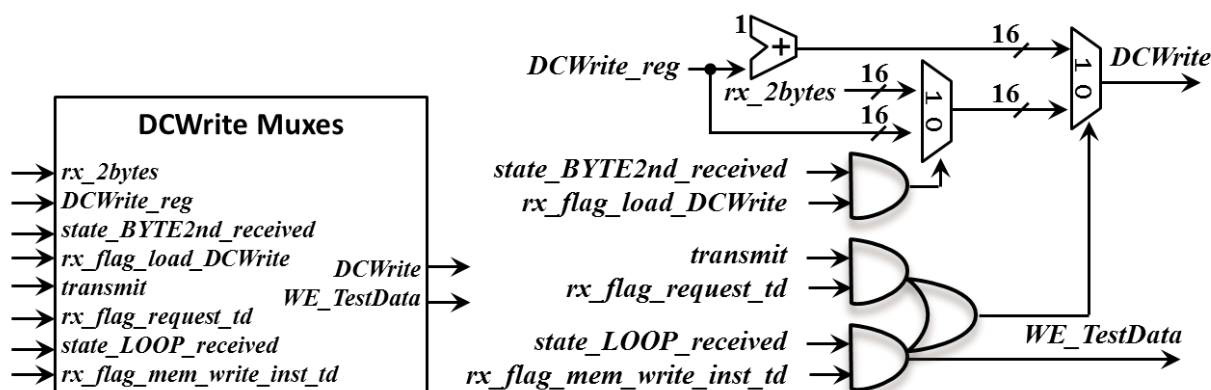


Figure 4.23 : DCWrite circuitry in the communication protoocol.

- **rx_2bytes**: the latest received two bytes.
- **DCWrite_reg**: The old DCWrite register value.
- **state_BYTE2nd_received**: rises once the receive-state-machine receives 16-bit value.
- **rx_flag_load_ DCWrite**: indicates that the user requests loading value to DCWrite.
- **transmit**: indicates that the UART module starts transmitting a byte to the user.
- **rx_flag_request_td:** indicates that the current context is reading from test data memory.

- **state_LOOP_received:** the state machine is at state LOOP and a new byte is received.
- **rx_flag_mem_write_inst_td:** if it is high, the writing bytes will be to test data memory.
- **DCWrite**: The new DCWrite register value.
- **rx_flag_request_td**: a flag raises when the user requests reading test-data.
- **WE_TestData**: set the write enable pin of the test-data memory.

4.3.14 RCRead circuitry

RCRead is the second test result memory address port which is a read only port. The user can read from test-result momory and RCRead register holds the reading address. The user can load this register using the protocol before requesting test result. When the protocol start transmitting the test result, RCRead is incremented with each sent byte. The RCRead circuitry is shown in Figure 4.24.

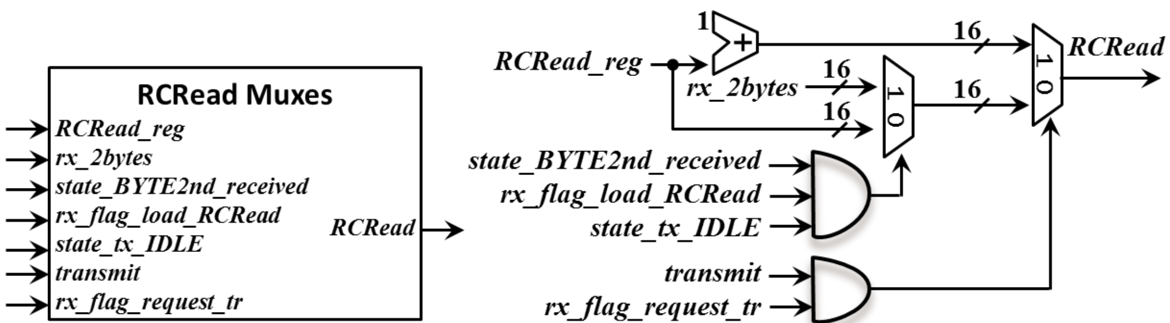


Figure 4.24 : RCRead circuitry in the communication protoocol.

- **RCRead_reg**: The old RCRead register value.
- **rx_2bytes**: the latest received two bytes.
- **state_tx_IDLE:** indicates that the transmitting FMS is at IDLE state.
- **transmit**: indicates that the UART module starts transmitting a byte to the user.
- **rx_flag_request_tr**: a flag raises when the user requests reading test result.
- **RCRead**: The new RCRead register value.

4.3.15 DCRead circuitry

DCRead is the second test data memory address port which is a read only port. PCRead circuitry allows only loading new values to PCRead, Figure 4.25.

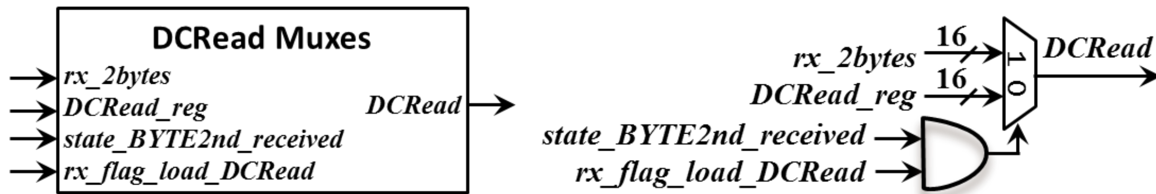


Figure 4.25 : DCRead circuitry in the communication protoocol.

- **rx_2bytes**: the latest received two bytes.
- **DCRead_reg**: The old DCRead register value.
- **state_BYTE2nd_received**: rises once the receive-state-machine receives 16-bit value.
- **rx_flag_load_DCRead**: a flag raises when the user requests loading new value to DCRead.
- **DCRead**: The new DCRead register value.

4.3.16 RCWrite circuitry

RCWrite is the first test result memory address port which is a write/read port. RCWrite circuitry allows just loading new values to RCWrite, Figure 4.26.

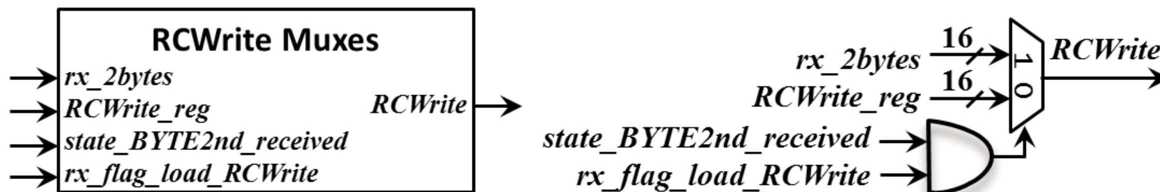


Figure 4.26 : RCWrite circuitry in the communication protocol.

- **rx_2bytes**: the latest received two bytes.
- **RCWrite_reg**: The old RCWrite register value.
- **state_BYTE2nd_received**: rises once the receive-state-machine receives 16-bit value.
- **rx_flag_load_RCWrite**: a flag raises when the user requests loading new value to RCWrite.
- **RCRead**: The new RCWrite register value.

4.4 Memories

There are three identical dual port RAMs: instruction memory, test data memory, and test result memory. Each is connected with two address registers; one write/read register and one read register. There is one write enable signal for each memory, Figure 4.27. Each memory has four inputs, four outputs, clock and reset signal.

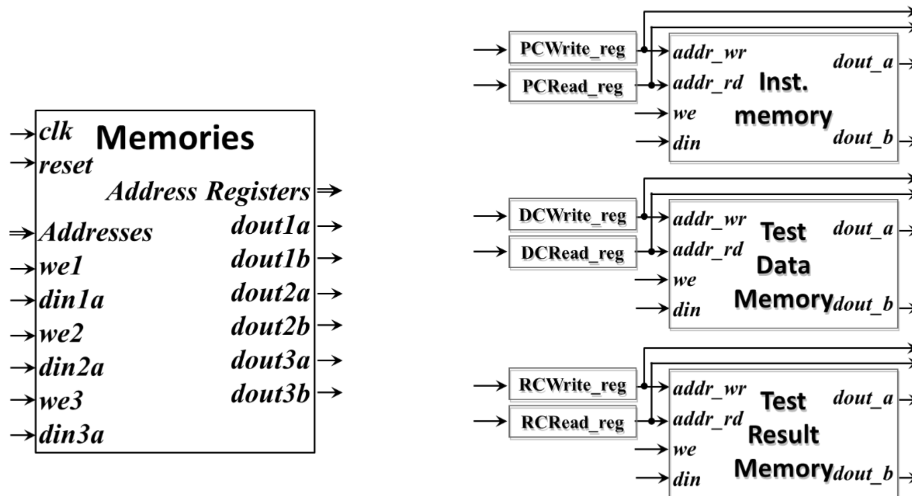


Figure 4.27 : Memories and address registers. Each memory has four inputs and four outputs. Each memory has two address registers; one write/read register and one read only register.

4.5 Memory Multiplexer

The TACP processor and the communication protocol both are accessing the memories. Obviously, some multiplexers are needed for the shared address ports, write enable, and data-in ports. Those needed multiplexers are shown in Figure 4.28.

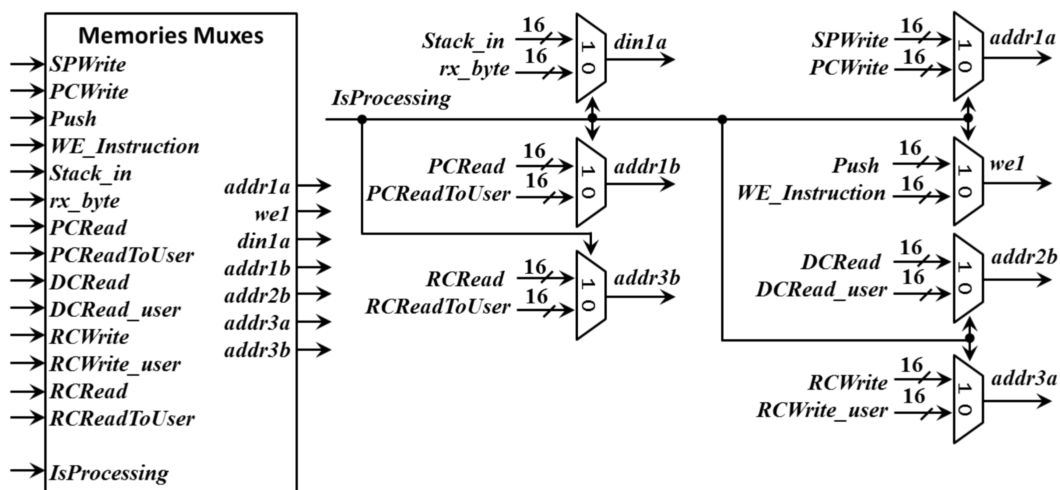


Figure 4.28 : Memory multiplexer circuitry manage memory access between the data path and the protocol.

IsProcessing Circuitry

The memories multiplexers are controlled by IsProcessing signal. When IsProcessing is high, it allows the processor to access memory. When it is low, it allows the protocol to access memory. This signal ensures that the processor is not executing an instruction when the protocol accessing memory, Figure 4.29.

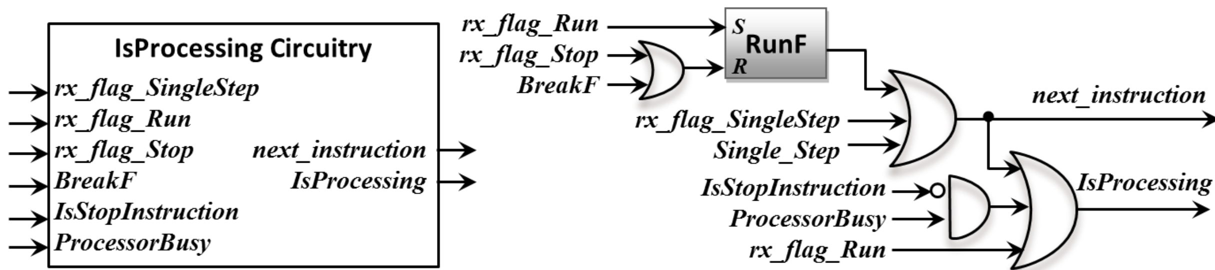


Figure 4.29 : Memory multiplexer circuitry manage memory access from the TACP data path and the communication protocol to the memories.

4.6 TACP Processor

The processor connected to memory to read instructions and test data, store test results, and modify address register contents, Figure 4.30. It is connected to the user communication unit to receive user control signals and requests for reading register contents and responds to them. The processor is connected to TSC to do testing and read the results.

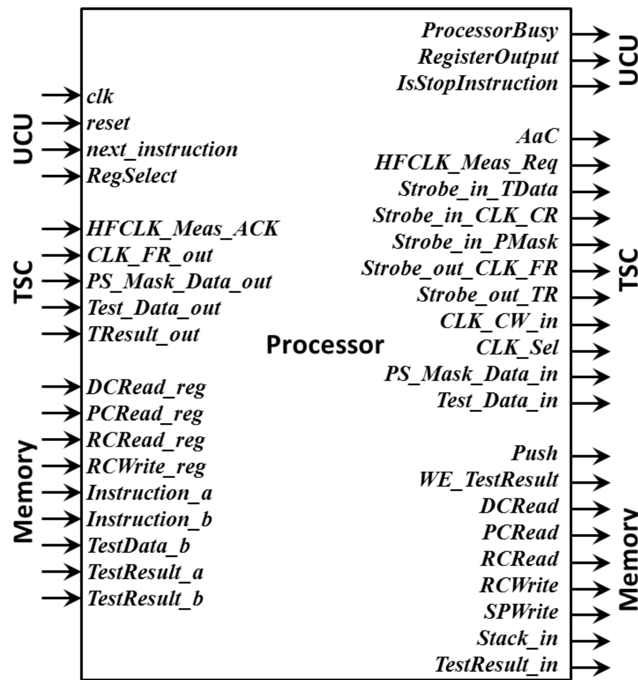


Figure 4.30 : Processor top diagram.

The processor is designed using microcode architecture which consists of a sequencer, control store, and data path, Figure 4.31.

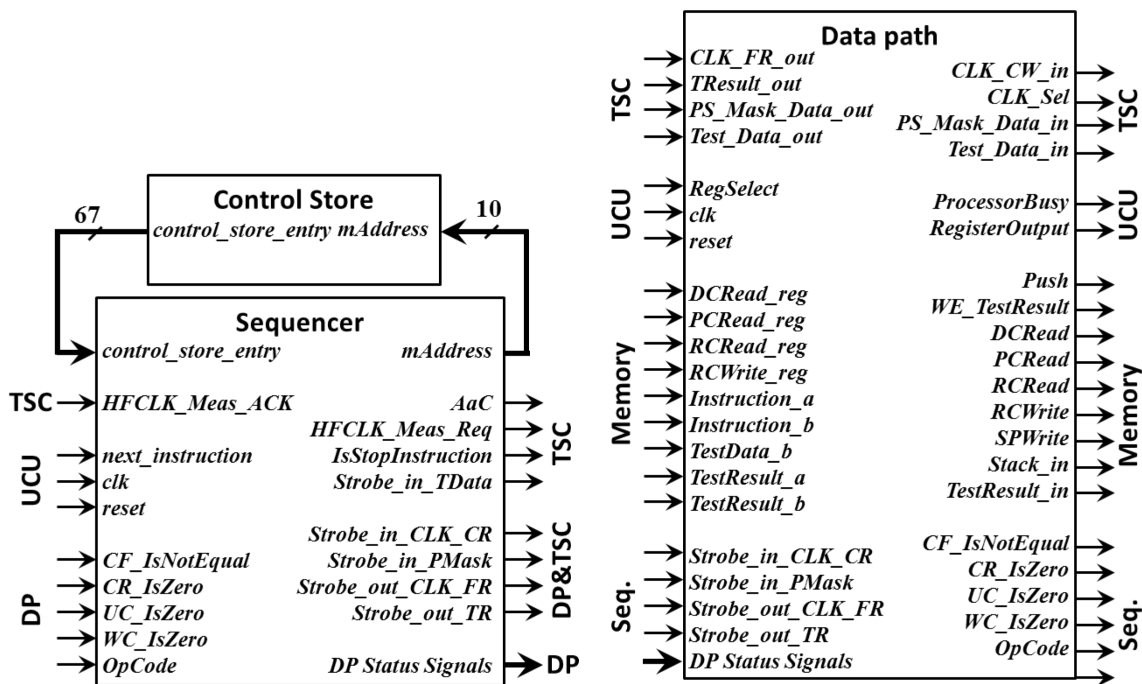


Figure 4.31 : Processor components: sequencer, control store, and data path.

- **AaC**: apply and capture signal.
- **CF_IsNotEqual**: control flag connected to the sequencer reflects the compare flag status.
- **CLK_CW_in**: control word shift register output. This signal is used for shifting out the control word register to the chip. It is used with the associated strobe signal Strobe_out_CLK_CR.
- **CLK_FR_out**: frequency register bit coming from the chip. It has to be shifted to FR register when Strobe_out_CLK_FRis high.
- **CLK_Sel**: a signal is sent to the chip to indicate that the testing mode is at-speed testing.
- **control_store_entry**: each entry represents all microinstructions to be executed in the current clock cycle. It consists of selection, control signals and branch address.
- **CR_IsZero**: control signal connected to the sequencer reflects the CR counter zero flag.
- **DCRead**: data counter to be connected to the second address port of the test data memory.
- DCRead_reg, PCRead_reg, RCRead_reg, RCWrite_reg: address ports of memories.
- **HFCLK_Meas_ACK**: the acknowledge signal is coming from the chip to indicate that the frequency measuring circuit is ready to start measuring.
- **HFCLK_Meas_Req**: request to the chip to start measuring the chip frequency.
- Instruction_a, Instruction_b, TestData_b, TestResult_a, TestResult_b: data out ports of memories.
- **IsStopInstruction**: indicates that the processor is currently executing the stop instruction.
- **mAddress**: the current micro address which is the control store ROM address.
- **Master_GND**: a ground signal is sent from TACP to the chip.
- **OpCode**: control signal reflects the current instruction opcode. It is connected to IR register.
- **PCRead**: program counter to be connected to the second instruction memory address port.
- **ProcessorBusy**: indicates whether the processor is currently executing an instruction. This signal is used to prevent contention on writing to memories by the processor and the communication unit.
- **PS_Mask_Data_in**: a signal is used for sending port selection mask to the chip. It is used with the associated strobe signal Strobe_in_PMask.
- **PS_Mask_Data_out**: return back bit from the chip used for verification. When the port selection mask is sent using the PS_Mask_Data_in, the old port selection mask is returned back through this pin.
- **Push**: indicates that the processor is writing to the stack in the instruction memory. This signal is used to prevent contention between the processor and the communication unit.
- **RCRead**: result counter to be connected to the second test result memory address port.
- **RCWrite**: result counter to be connected to the first address port of the test result memory.
- **RegisterOutput**: one byte gets one of data path registers according to RegSelect. This bus signals is used to read data path info to the user.

- **reset**: the processor reset signal is sent to the chip.
- **RegSelect**: determines which register the user is currently read.
- **SPWrite**: stack pointer to be connected to the first address port of the instruction memory.
- **Stack_in**: pushed stack data. To be connected to data in port of the instruction memory.
- **Strobe_in_CLK_CR**: indicates shifting out the control word register.
- **Strobe_in_PMask**: indicates shifting out the port selection mask.
- **Strobe_in_TData**: indicates shifting out the test data register.
- **Strobe_out_CLK_FR**: indicates shifting in the frequency register.
- **Strobe_out_TR**: indicates shifting in the test result.
- **Test_Data_in**: a signal is used for shifting out the test data register to the chip. It is used with the associated strobe signal Strobe_in_TData.
- **Test_Data_out**: return back bit from the chip used for verification. When the test data is sent using the Test_Data_in, the old test data is returned back through this pin.
- **TestResult_in**: To be connected to data in port of the test result memory.
- **TResult_out**: the test result bit coming from the chip. It is associated with the strobe Strobe_out_TR
- **UC_IsZero**: control signal connected to the sequencer reflects the user counter zero flag.
- **WC_IsZero**: control signal connected to the sequencer reflects the WC counter zero flag.
- **WE_TestResult**: the write enable pin of the test result memory.
- **DP Status Signals**: a collection of 45 signals that are coming from the sequencer to control the data path. those signals are as shown in Table 4.4.

4.6.1 The sequencer

Sequencer and control store are shown on Figure 4.32. Sequencer has five components:

- Micro-address incrementer (variable size = a bits).
- Micro-address multiplexer (fixed size = 2-input multiplexer).
- Micro-address register (variable size = a bits).
- Selection 0 comparator (variable size = a bits).
- Control multiplexer (variable size = $log_2 d$ bits, d = number of control signals).

Unlike other components in the sequencer, the control multiplexer size changes according to the number of control signals. This number comes from the number of signals that causes conditional branching and used as micro-instructions. Clearly, it changes when editing the

instruction set. Special algorithm is developed to generate the HDL code of the control multiplexer. Other sequencer components are parameterizable and their HDL code could be fixed. Figure 4.29 shows the sequencer and its control multiplexer that has a variable size.

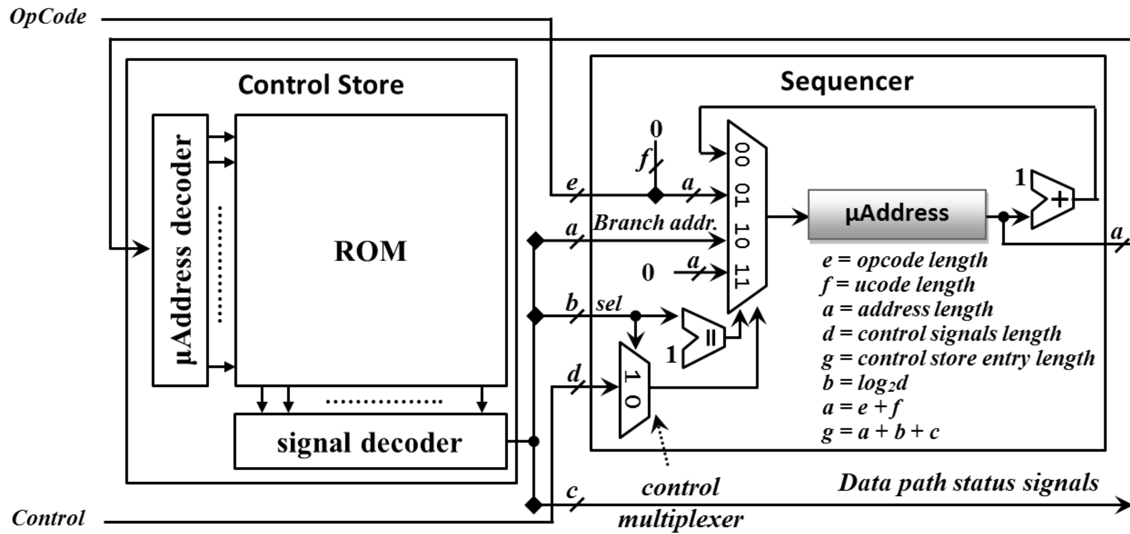


Figure 4.32 : The sequencer and the control store.

4.6.2 The control store

The control store consists of ROM and two decoders, Figure 4.32. The ROM width, height, and contents vary with each edition of the instruction set. Each instruction is consists of one or more control store entries. Each entry represents all micro-instructions that should be executed in the same clock cycle. The entry consists of the control multiplexer selection bits followed by the status signals to be sent to the data path, and ends by the branch address which consists of opcode and micro-code, Figure 4.33. the sequencer to datapath signals are listed in Table 4.4.

0001_0000001000000000000000_00000_00000_0000

selection *status signals* *opcode_ucode*

branch address

Figure 4.33 : Control store entry consists of selection, status signal, and branch address.

Table 4.4: Sequencer to data path signals (micro-instructions signals)

No.	Signal name	Description
1	ClearTD	Clear test data counter
2	ClearTR	Clear test result counter
3	Decrement_CR	Decrement the general counter CR.
4	Decrement_WC	Decrement the word counter WC.
5	DEC_CW	Decrement frequency control word CW register.
6	DEC_SP	Decrement the stack pointer SP.
7	DEC_UC	Decrement the user counter UC.
8	Increment_DC	Increment the test data counter, i.e. the DCRead register.
9	Increment_PC	Increment the program counter, i.e. the PCRead register.
10	Increment_RCRead	Increment the second test result address register, i.e. the RCRead register.
11	Increment_RCWrite	Increment the first test result address register, i.e. the RCWrite register.
12	INC_CW	Increment frequency control word CW register.
13	INC_UC	Increment the user counter UC.
14	Load_CR_High_Instruction2	Load 16 bits to the highest part of the general register.
15	Load_CR_Low_Instruction2	Load 16 bits to the lowest part of the general register.
16	Load_CW_Instruction2	Load 16 bits to the frequency control word regiser CW.
17	Load_DC_Instruction2	Load 16 bits to the test data counter, i.e. the DCRead register.
18	Load_IR_Instruction	Load 8 bits from instruction memory to the instruction register IR.
19	Load_PC_Instruction2	Load 16 bits to the program counter register IR from instruction memory.
20	Load_RCRead_Instruction2	Load 16 bits to the second test result memory address register.
21	Load_RCWrite_Instruction2	Load 16 bits to the first test result memory address register.
22	Load_TD_TestData	Copy a byte from test data memory to test data register TD.

23	Load_UC_High	Load 16 bits to the highest part of the user counter.
24	Load_UC_Low	Load 16 bits to the lowest part of the user counter.
25	Load_UC_TR1	Copy a byte from the test result memory to the first part of the user counter.
26	Load_UC_TR2	Copy a byte from the test result memory to the second part of the user counter.
27	Load_UC_TR3	Copy a byte from the test result memory to the third part of the user counter.
28	Load_UC_TR4	Copy a byte from the test result memory to the fourth part of the user counter.
29	Load_WC_Instruction	Load 8 bits from instruction memory to word counter WC.
30	Pop1	Pops 8 bits from the stack to the PrevParam_b register.
31	Pop_PC2	Pops 8 bits from the stack and combine them with PrevParam_b then copy the formed 16 bits to the program counter PCRead.
32	Push_PC1	Push the lowest part of the program counter PCRead to the stack.
33	Push_PC2	Push the highest part of the program counter PCRead to the stack.
34	ResetBusy	Reset the processor busy flag.
35	ResetCF	Reset the compare flag CF.
36	ResetHFClock	Reset the high frequency clock selection flag SF.
37	SetBusy	Set the processor busy flag.
38	SetHFClock	Set the high frequency clock selection flag SF.
39	Shift_TestData	Shift the test data register TD out to the chip.
40	Store_TestResults_Compare	Store the result of comparing test result with expected results.
41	Store_TestResults_TR	Store the test result register TR in the test result memory.
42	Store_UC1	Store the first byte of the user counter UC in the test result memory.
43	Store_UC2	Store the second byte of the user counter UC in the test result memory.
44	Store_UC3	Store the third byte of the user counter UC in the test result memory.
45	Store_UC4	Store the fourth byte of the user counter UC in the test result memory.

The following subsections are the data path components.

4.6.3 Two previous parameter registers

Instruction memory has two data-out ports Instruction_a and Instruction_b. PrevParam_a and PrevParam_b registers are used to store the latest read value of these ports respectively. This way, it is possible to have a 16-bit word by combining the current data-out (Instruction_a or Instruction_b as a higher part) with the previously read byte (PrevParam_a or PrevParam_b as a lower part). The PrevParam_a does not change its value until there is an increment on the program counter. This way port-selection circuitry can use Instruction_a_16 in its comparison loop while Increment_PC prevent PrevParam_a against taking Instruction_a value before completing the loop. As the program counter is incremented, we get 16-bit word from each instruction memory port as shown in Figure 4.34.

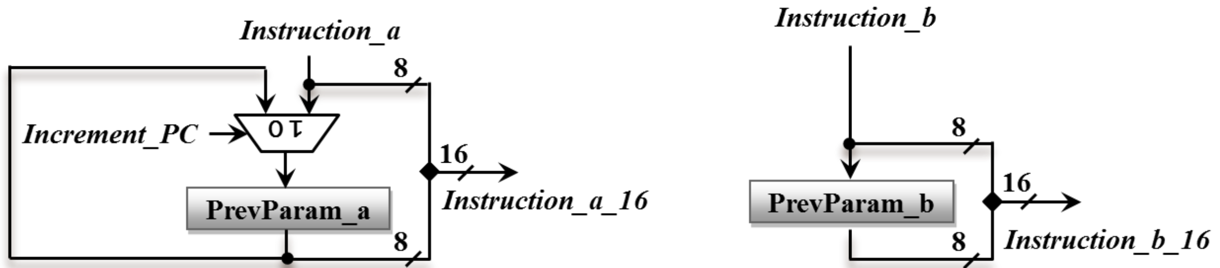


Figure 4.34 : Combining previous and current byte to form a 16-bit word for each port on instruction memory.

- **Increment_PC**: load a new value only with the incrementing the program counter. This is important prevent overwriting PrevParam_a since it will be used in a loop to send port selection mask.
- **Instruction_a**: the first instruction memory data out port.
- **Instruction_a_16**: two bytes which are loaded from the stack in the instruction memory.
- **Instruction_b**: the second instruction memory data out port.
- **Instruction_b_16**: the last two bytes which are loaded from the instruction memory.

4.6.4 Port selection mask circuitry

The port selection mask is a binary string to be sent to the selection register in the support circuitry on the chip. As shown in Figure 4.35 the port selection mask circuitry is a comparing circuit that compares the lower 16-bit part of CR with the port number.

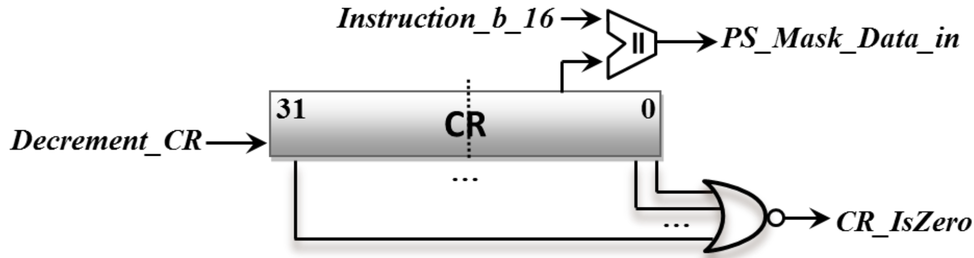


Figure 4.35 : CR down-counter with the 16 bit port selection mask generation circuitry and the CR_IsZero flag.

- **PS_Mask_Data_in**: a signal is used for sending port selection mask to the chip. It is used with the associated strobe signal Strobe_in_PMask.
- **CR_IsZero**: control signal connected to the sequencer reflects the CR counter zero flag.
- **Decrement_CR**: Decrement the general counter CR.
- **Instruction_b_16**: the last two bytes which are loaded from the instruction memory.

This circuitry works as a decoder that decodes the port number. At first, the number of ports has to be loaded into the general counter CR. Then, CR is decremented until it reaches zero. By comparing the value of CR with the 16-bit parameter, a bit stream is generated whose length is equal to the number of ports and contains all zeroes except one position. Table 4.5 below shows examples of some mask strings and the needed CR and parameter value to generate it.

Table 4.5: Selecting port examples

Parameter	CR	Output
0	16	0000000000000001
1	16	0000000000000010
2	8	00000100
3	6	001000
15	16	1000000000000000

To select more than one port at a time, the operation has to be done multiple times. For example, to get the bit stream “0001000100”: set parameter = 2 and CR = 6. This will generate “000100”. Then set parameter = 0 and CR = 4 to generate “0001”. By sending those two strings to the selection mask it is possible to activate more than one port simultaneously.

4.6.5 Selection mask shift register SM

It shifts in the selection mask that is returned back from the chip. It is one-word length (i.e. byte) register whose only use is for verification. The register is cleared when the sending selection mask starts (i.e when CR is loaded). The port selection strobe is also used here as a shift signal. So, the register has two signals that are coming from the chip which are the strobe and the returned back mask bit. Figure 4.36 shows the SM circuitry.

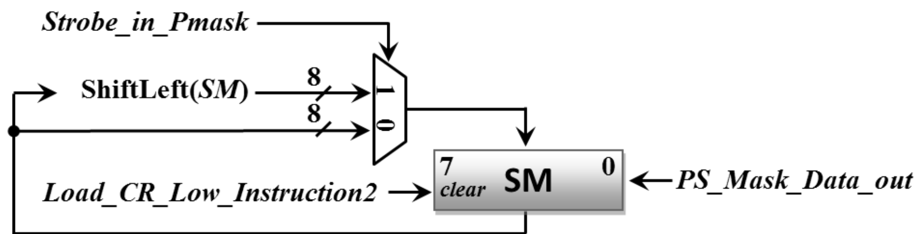


Figure 4.36 : 8-bit selection mask register shifts in the returned-back port selection from the chip.

- **Strobe_in_Pmask**: the strobe that is sent to the chip to shift in the port selection mask.
- **PS_Mask_Data_out**: the returned back bit from the chip. PS_Mask_data_in, PS_Mask_data_out are two signals that are connected to the selection shift register on the chip as a shift data in and out respectively.
- **Load_CR_Low_Instruction2**: load two bytes from instruction memory to the lower part of the counter register CR. This signal is used here as a clear signal since it is called on the beginning of each sending port selection mask operation.
- **ShiftLeft(SM)**: the SM register after shifting left and including PS_Mask_Data_out signal.

4.6.6 Test data shift register TD

It loads a byte from the test data memory then shifts it out to the chip bit by bit. At the same time it shifts in the returned back test data from the chip. Figure 4.37 shows the register TD and its circuitry.

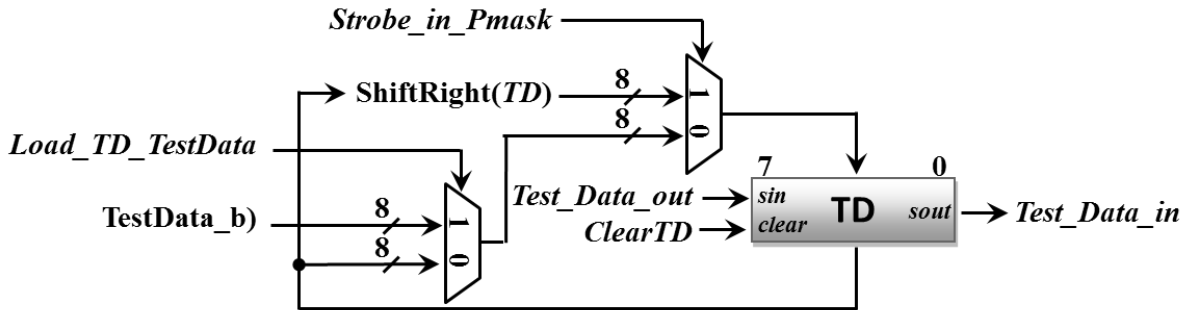


Figure 4.37 : 8-bit test data register and its circuitry.

- **Shift_TestData**: if it is high the TD contents will be shifted right and Test_Data_out bit will be shited into the register.
- **Load_TD_TestData**: is used to load from test data memory to the TD register.
- **TestData_b**: the second test-data memory data port. This value will be loaded into TD if Load_TD_TestData is high.
- **ClearTD**: is used to clear the register before each shifting operation. This is important when it is needed to shift less than 8 bits.
- **Test_Data_in**: the bit to be sent to the chip.
- **Test_Data_out**: the returned back bit from the chip.

4.6.7 Test result shift register TR

It shifts in the test result bits that are coming from the chip. Then its content is stored in the test result memory. Its circuitry is shown in Figure 4.38 below.

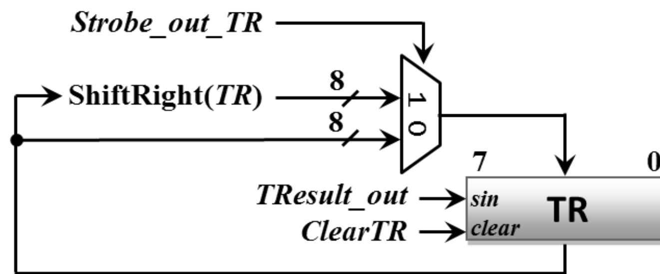


Figure 4.38 : 8-bit test results register and its circuitry.

- **Strobe_out_TR**: a signal to start shifting in the test result.
- **TResult_out**: test result bit received from the chip.
- **ClearTR**: clear TR register. It is important since the number of shifted bits is arbitrary.

4.6.8 Frequency register FR

This frequency register shifts in the frequency register that contains the measured frequency on the chip, Figure 4.39.

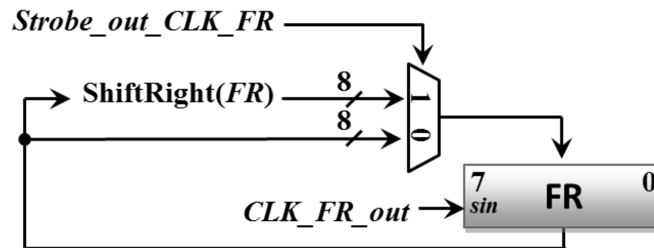


Figure 4.39 : Frequency register circuitry.

- **Strobe_out_CLK_FR**: shifts in the frequency register from the chip.
- **CLK_FR_out**: the frequency bit coming from the chip.
- **ShiftLeft(FR)**: the FR register after shifting left and including CLK_FR_out signal.

4.6.9 Frequency control word register CW

The control word register determines the at-speed clock frequency. The register can be loaded with a 16-bit immediate value and then incremented or decremented as shown in Figure 4.40. Its content has to be sent to the chip by rotating it left.

- **Load_CW_Instruction2**: load two instruction memory bytes to the CW register.
- **INC_CW**: Increment frequency control word CW register.

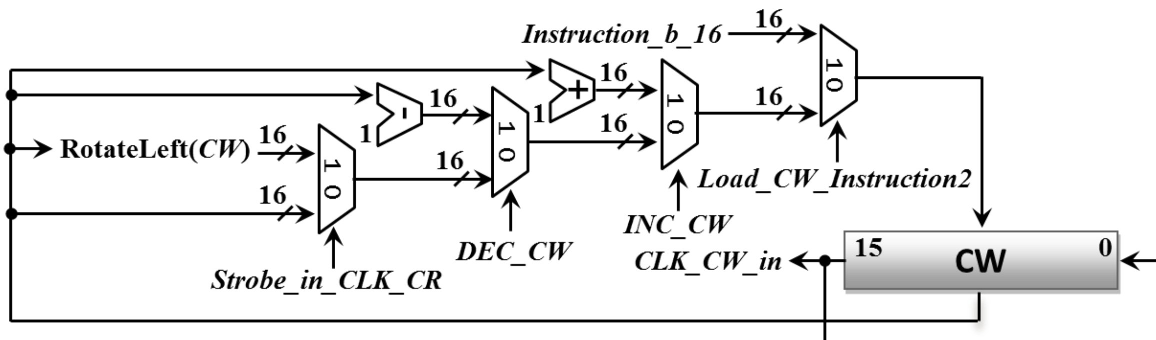


Figure 4.40 : 16-bit control word register and its circuitry.

- **DEC_CW**: decrement frequency control word CW register.
- **Strobe_in_CLK_CR**: a shift signal rotates the control word register.
- **Instruction_b_16**: the last two bytes which are loaded from the instruction memory.
- **RotateLeft(CW)**: the CW register after rotated left.
- **CLK_CW_in**: control word shift register output. This signal is used for shifting out the control word register to the chip. It is used with the associated strobe signal Strobe_out_CLK_CR.

4.6.10 Instruction Register IR

The instruction register contains the opcode of the instruction under execution. Its circuitry has only one signal to control the loading. The register takes new value on the fetch stage of the instruction execution, Figure 4.41.

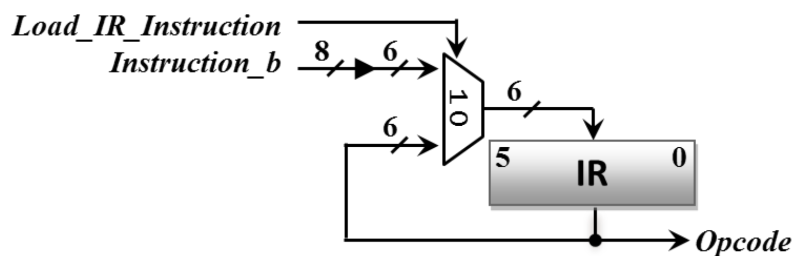


Figure 4.41 : Six bit instruction register.

- **Instruction_b**: the second instruction memory data out port.
- **Load_IR_Instruction**: Load 8 bits from instruction memory to the instruction register IR.
- **OpCode**: control signal reflects the current instruction opcode. It is connected to IR register.

4.6.11 General counter CR

It is a general 32-bit down counter for internal use. Instructions mainly use it to count reading/writing words from/to memory, Figure 4.42. This counter has to be loaded with two separate operation each operation increments the program counter PC by two.

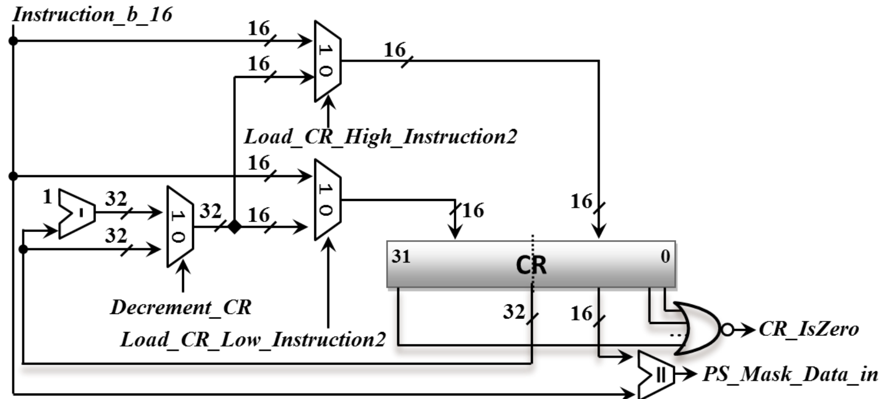


Figure 4.42 : The 32 bit general counter CR and its circuitry.

- **Instruction_b_16**: the last two bytes which are loaded from the instruction memory.
- **Decrement_CR**: decrements the 32-bit counter by one.
- **Load_CR_Low_Instruction2**: loads the parameter to the lower 16-bit part.
- **Load_CR_High_Instruction2**: loads the parameter to the higher 16-bit part.
- **PS_Mask_Data_in**: a signal is used for sending port selection mask to the chip. It is used with the associated strobe signal Strobe_in_PMask.
- **CR_IsZero**: control signal connected to the sequencer reflects the CR counter zero flag.

4.6.12 Word counter WC

It is a general 4-bit down counter for internal use. It is used to count bits when shifting in or out a register. It is very helpful to do shifting by arbitrary number, Figure 4.43.

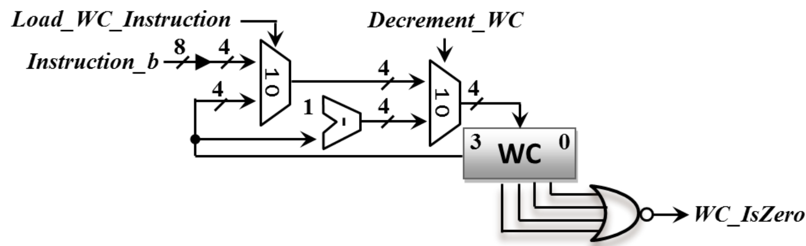


Figure 4.43 : 4-bit Word Counter WC and its circuitry.

- **Load_WC_instruction**: load four bits from the instruction memory to the WC counter.
- **Decrement_WC**: decrement the WC counter.
- **Instruction_b**: the instruction parameter coming from the second instruction memory port.
- **WC_IsZero**: control signal connected to the sequencer reflects the WC counter zero flag.

4.6.13 User counter register UC

A 32-bit up-down counter is designed for the user. It can be loaded with immediate value or it can be loaded/stored in the test result memory. This enables the user to have multiple counters in the memory and switch between them. It is associated with a zero flag **ZF** (combinational flag) and JZ and JNZ instructions. So the user can make conditional branching according to its value.

As shown in Figure 4.44, four signals are used to load from test result memory, **Load_UC_TR1**, **2**, **3**, and **4**. Two signal are used to load immediate value coming from instruction memory, **Load_UC_High** and **Load_UC_Low**. The test result memory is chosen for load/store the counter value because it has the least addressing multiplexers.

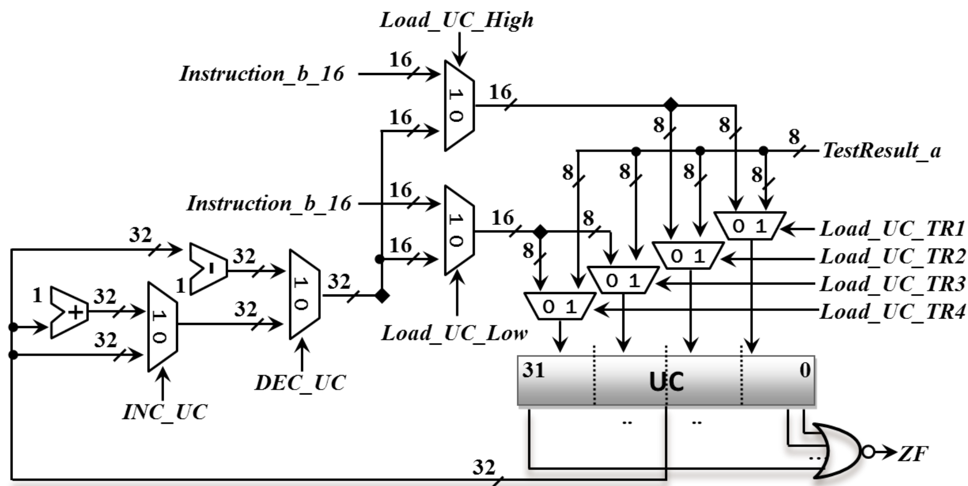


Figure 4.44 : User-counter circuitry.

4.6.14 Stack pointer SP

Stack pointer is a 16-bit address register and points to the top of the stack. The instruction memory is used as a stack memory for calling subroutines starting from the highest address location. The stack pointer is initialized with zero and incremented with each push operation and incremented with each pop operation.

When the call instruction is invoked, it decrement SP twice and store two byte of PC. When return instruction is invoked, it send two signals Pop1 and Pop_PC2 to retrieve the PC value and increment SP twice, Figure 4.45.

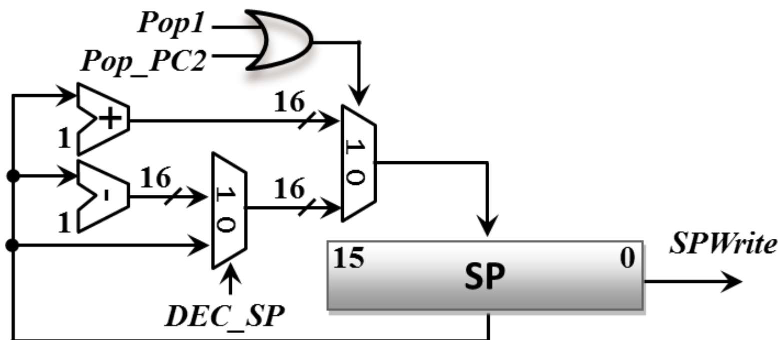


Figure 4.45 : Stack pointer circuitry.

- **Pop1**: increment SP by one.

- **Pop_PC2**: increment SP by one and copy the read address “Instruction_a_16” to the program counter PC.
- **DEC_SP**: decrement SP by one.
- **SPWrite**: stack pointer to be connected to the first address port of the instruction memory.

4.6.15 Flags

There are seven flags. Some of them are storage elements like CF, SF, ErrF and RunF. The rest are combinational circuits outputs like ZF, CR_IsZero and WC_IsZero.

a. **Communication Error Flag ErrF:** Its value become high if there is an error on the last transmission operation between the PC and the Processor. This register is part of the communication protocol module not the TACP data path module.
b. **Run Flag RunF:** indicates whether the current mode is single step mode or normal mode. In the normal mode, its value is high. This register is part of the communication protocol not the TACP data path module, Figure 4.46.

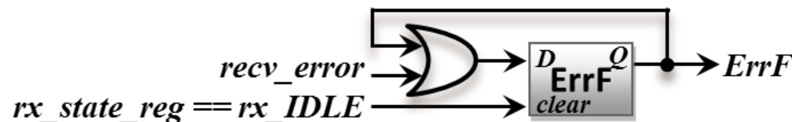


Figure 4.46 : Zero flag circuitry for the user counter register.

c. **User Counter Flag ZF:** Its value indicates whether the contents of the user counter register is zero or not. Since the user can use more than one counter using the load/store instructions, this flag always reflects the zero flag of the current counter because it is a combinational circuit, Figure 4.47.

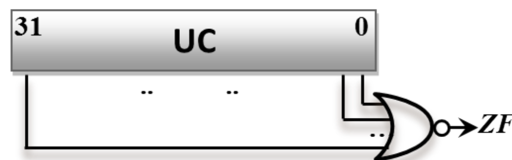


Figure 4.47 : Zero flag circuitry for the user counter register.

d. **CR_IsZero, WC_IsZero:** Internal flags that are used in looping inside the instruction. They reflects whether CR or WC is zero or not respectively, Figure 4.48.

Figure 4.48 : Zero flags circuitry of the general counters.

e. **Clock Selection Flag SF:** It is a set-reset flip-flop used to switch between the high speed and the low speed clock in the chip. Its value indicates which clock should be selected and applied to the IP under test when the ApplyAndCapture signal is high, Figure 4.49.

Figure 4.49 : Clock selection Flag circuitry.

f. **Compare error flag CF:**It became high if there is an error resulted in comparing the test results with the expected results. The compare instruction compares one word from the test data memory with one word from the test result memory. Its value is changed with the storing of the comparison result in the memory. Its circuitry is shown in Figure 4.50.

Figure 4.50 : Compare error flag circuitry.

4.6.16 Memory addressing circuitry

There are three dual port memories associated with six address registers. The data path needs to modify some of those address registers. It needs to modify the program counter and the stack pointer, increment the test data reading address and increment the test result writing address. The needed circuitry to do this is shown in Figure 4.51.

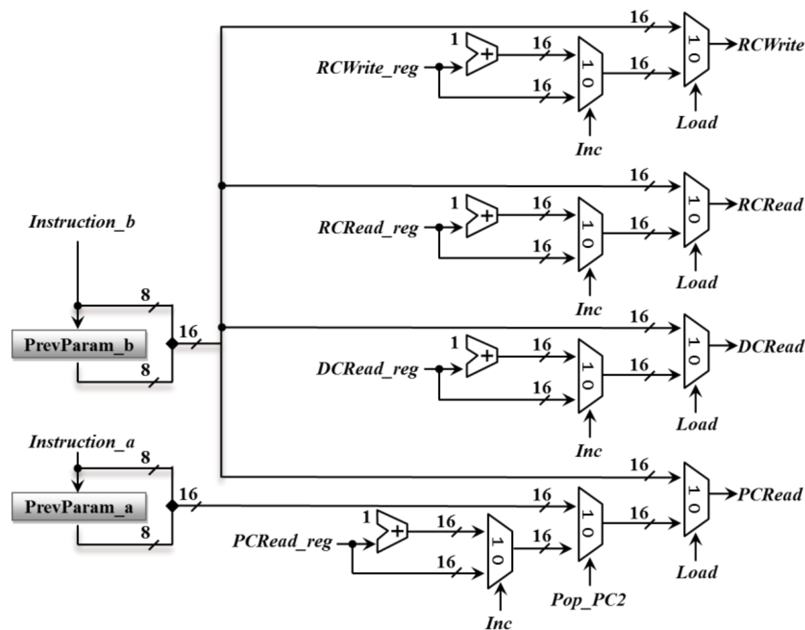


Figure 4.51 : Addressing circuitry in the TACP data path.

4.6.17 Test result memory writing circuitry

Unlike the communication protocol module, the TACP data path can write to the test result memory. It may store the test result TR register, the comparison result between test result and expected result, or the user counter. Thus the test-result memory data-in port needed circuitry as shown in Figure 4.52 consists mainly of multiplexers. Notice that the user counter UC needs four multiplexers since it is 32-bit up-down counter.

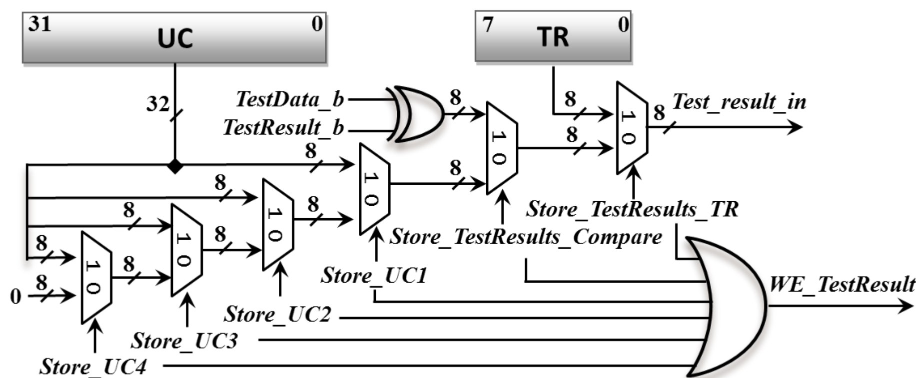


Figure 4.52 : Test-result memory data-in port and write enable circuitry.

- **TestData_b**: the second data out ports of test data memory.
- **TestResult_b**: the second data out ports of test result memory.
- **Store_TestResults_Compare**: Store the result of comparing test result with expected results.
- **Store_TestResults_TR**: Store the test result register TR in the test result memory.
- **Store_UC1**: Store the first byte of the user counter UC in the test result memory.
- **Store_UC2**: Store the second byte of the user counter UC in the test result memory.
- **Store_UC3**: Store the third byte of the user counter UC in the test result memory.
- **Store_UC4**: Store the fourth byte of the user counter UC in the test result memory.
- **TestResult_in**: To be connected to data in port of the test result memory.
- **WE_TestResult**: the write enable pin of the test result memory.

4.6.18 Push circuitry

The TACP processor uses a stack located at the bottom of the instruction memory. When a CALL instruction is invoked, the program counter has to be incremented and stored in the stack. The push circuitry contains an incrementer and a multiplexer as shown in Figure 4.53. The Stack_in bus will be multiplexed with other bus coming from the communication protocol module as it also need to write to instruction memory.

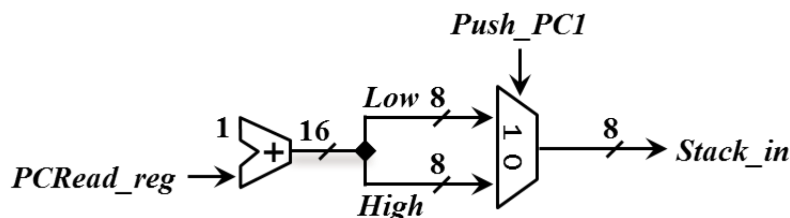


Figure 4.53 : Push circuitry generates Stack_in bus which is connected to the data-in port in the instruction memory.

4.6.19 Enumerate multiplexer

When the user requests info, the protocol responses by sending the TACP data path register contents, the memories addresses and data-out, and other registers. Enumerate multiplexer is a five-bit multiplexer that selects one byte at a time to be send to the user. As shown in Figure 4.54,

the multiplexer inputs are the processor data path info, memories address and data ports and the break point register. The five-bit selection RegSelect is five bits of the transmitting counter in the communication protocol. When the user ask for info, the transmitting state machine starts decrementing the transmitting counter and therefore sending one byte each time. Hence, the user has to set the value 24 in the transmitting counter to fetch all info at once.

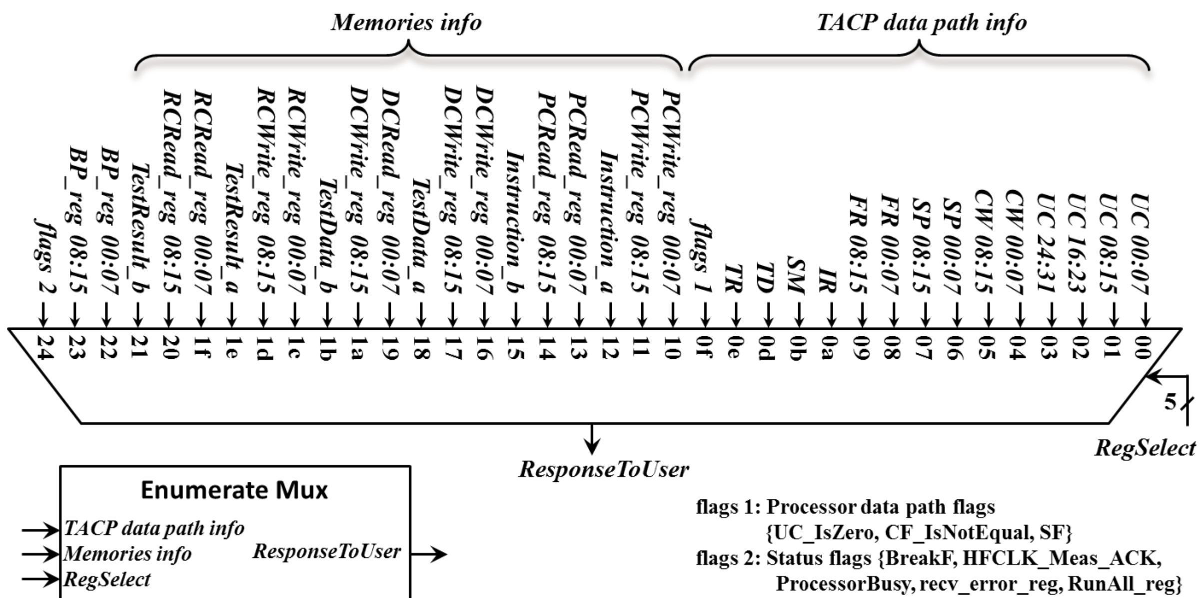


Figure 4.54 : 5-bit multiplexer selects one byte at a time to be send as a response to the user interface.

CHAPTER 5

TEST AND CHARACTERIZATION PROCESSOR IMPLEMENTATION

This chapter introduces the implementation of the test and characterization processor (TACP) and its integration with the whole system. The graphical user interface is implemented on a PC using a high level programming language. The TACP is implemented on one FPGA while the test support circuitry (TSC) with four circuits-under-test (CUTs) are emulated in another FPGA board. The PC is connected to the TACP FPGA and the two FPGAs are connected through a 20-pin cable forming the fixed interface. Xilinx ISE 14.2 is used to synthesize, simulate and download designs to FPGAs. The system works properly and many test programs are executed and will be discussed in the next chapter.

The chapter also introduces the instruction builder software that helps on editing the microcode instructions. Finally, the chapter discusses the problems faced when prototyping the TSC.

Figure 5.1 shows a snapshot of the platform implementation. The PC is connected through a serial communication cable to the TACP FPGA which in turn is connected to the prototyped TSC FPGA.

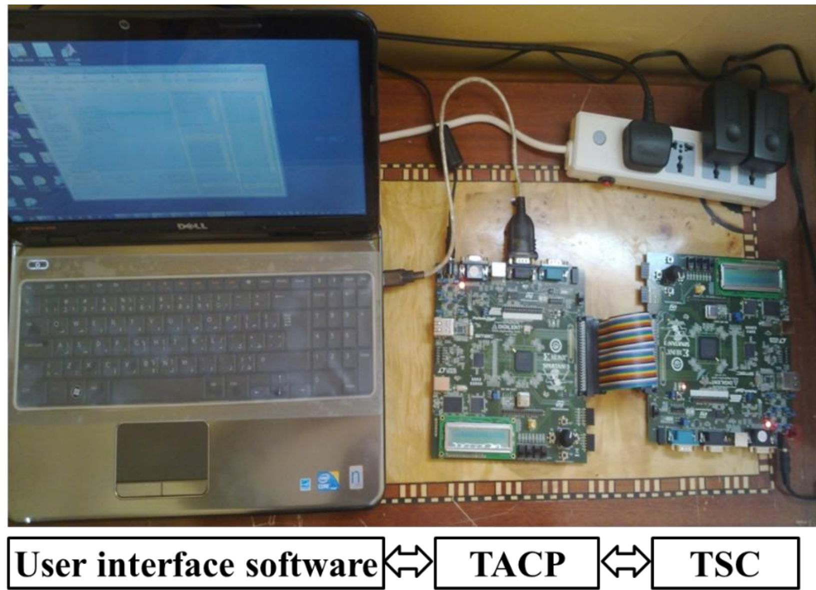


Figure 5.1: The Implemented test & characterization platform; the host PC running the user interface tool, an FPGA board for the TACP connected to the PC, and another FPGA board containing the TSC and 4 CUTs and connected to the TACP FPGA.

5.1 User Interface Implementation

Complete user interface software is designed using C#.Net programming language. It consists mainly of three interfaces; writing programs interface, executing programs interface and multiple memory interfaces. The software also implements the user communication protocol and has assembler and disassembler. It has a wizard to import test data vectors from text files generated by automatic test pattern generators (ATPGs). Appendix C includes a tutorial on this software.

Writing Programs

The software contains a dedicated interface to write programs as shown in Figure 5.2. The user can write programs by selecting instructions from the instruction list that are imported from the instruction builder software. The interface helps the user to set instruction parameters, move or delete program lines. It can also generate the assembly code of the program and uses the interface protocol to download the program to the instruction memory on the TACP FPGA.

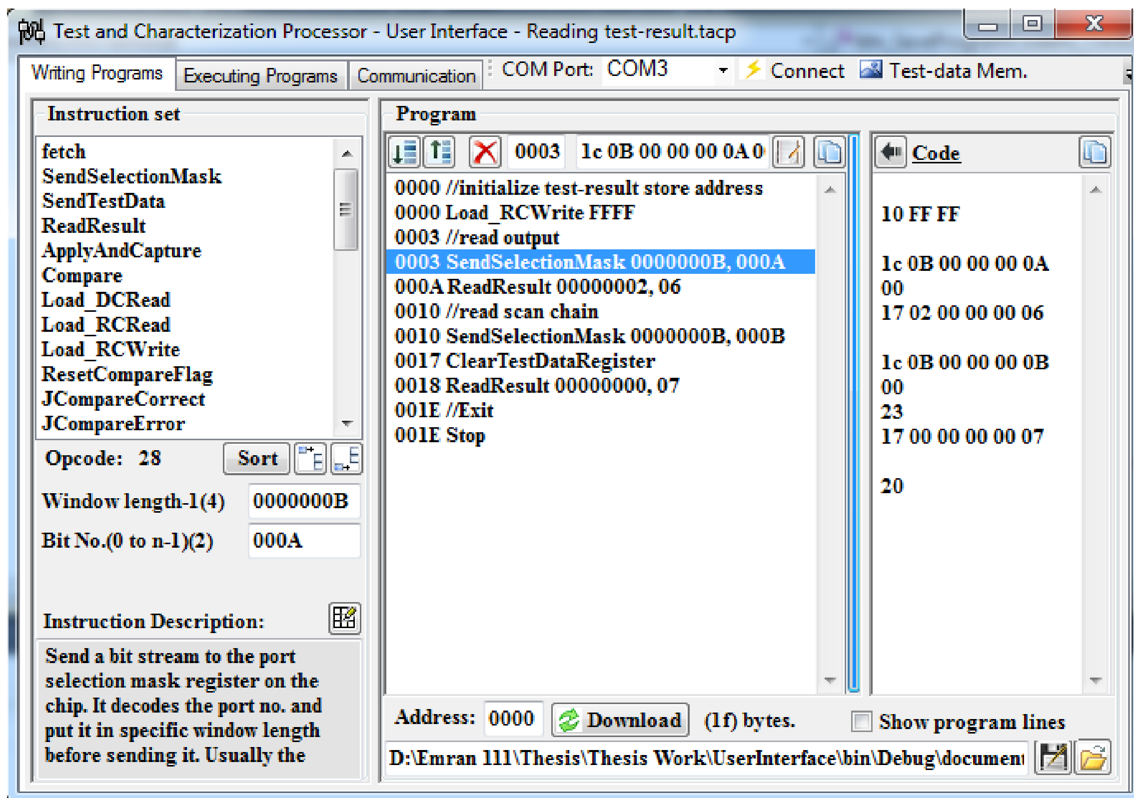


Figure 5.2 : User interface to write and download programs to the TACP FPGA.

Executing Programs

The software contains a dedicated interface to execute and track the execution of the program as shown in Figure 5.3. The user starts by uploading the current program on the TACP memory on the TACP FPGA. Then the execution starts with single mode or batch mode. The user can set a break point by selecting a program line and press “set break point” button. To change the current execution to a specific line the user double clicks that line. The user is also allowed to set values to any of the six address registers. The TACP registers are displayed and updated manually or automatically after each execution request.

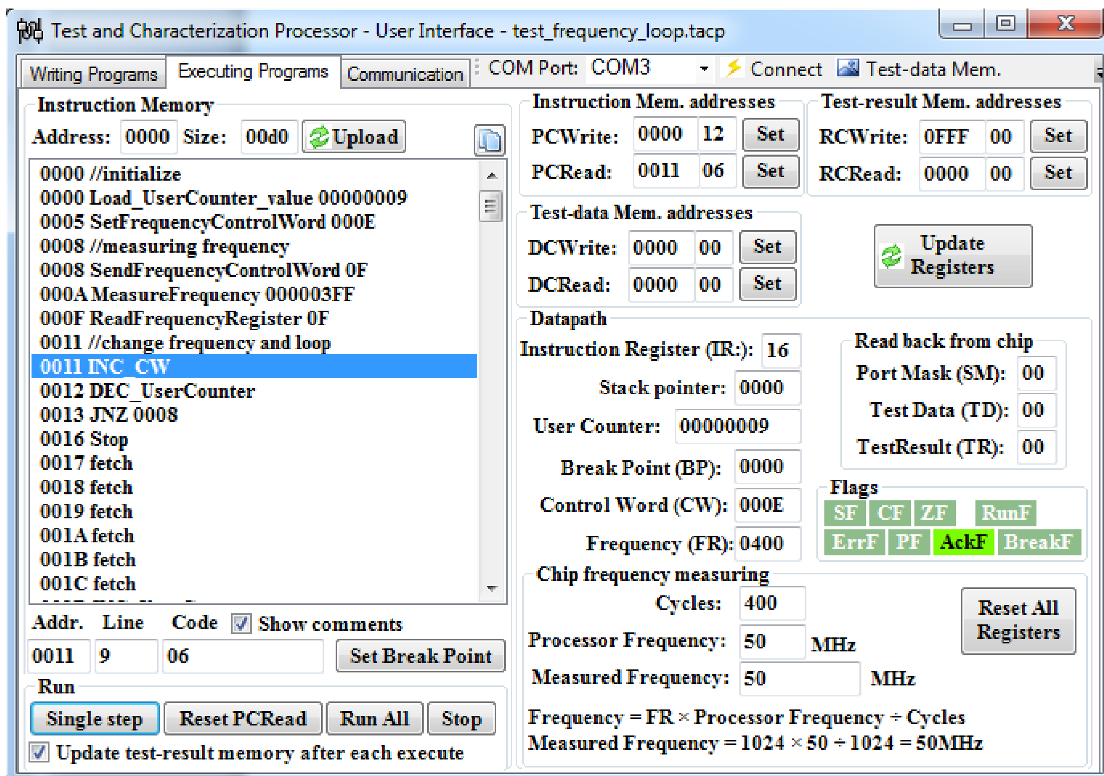


Figure 5.3 : User interface thet executes the program and tracks register contentss on the TACP FPGA.

- **PCRead**: The program counter.
- **PCWrite**: The instruction download address.
- **DCRead**: The test data address to be used when sending data to the TSC.
- **DCWrite**: The test data download address.
- **RCRead**: The upload test result
- **RCWrite**: The test result address to be used when storing results coming from the TSC.
- **IR**: The instruction registers shows the executed instruction opcode.
- **Stack pointer**: The stack address to be used with subroutines.
- **User Counter**: The user counter used to do iterations.
- **BP**: the break point register used to stop the execution when its value equals PCRead.
- **CW**: The frequency control word register value on the TACP that supposed to be sent to the corresponding CW register on the TSC.
- **FR**: The frequency register value on the TACP that supposed to be read from the corresponding FR register on the TSC.
- **Cycles**: The user has to write this value manually. It has to match the number of cycles used with the MeasureFrequency instruction.
- **Processor Frequency**: The TACP frequency. The user has to write this value manually.
- **Measured Frequency**: a calculated value represents the selected frequency on the chip.

- **SM**: The selection mask returned back from the chip.
- **TD**: The test data register value represents the returned back data from the chip.
- **TR**: The latest read test result byte.

Memory Interface

The software contains a dedicated interface to interact with the TACP memories on the TACP FPGA as shown in Figure 5.3. The interface allows the user to download to or upload from the TACP memories. The interface allows the user to write data or import them from file. The user can upload a memory block from the corresponding memory in the TACP FPGA by specifying its location and size. He also can download to any specific memory location. The memory interface can be customized to view the contents one test vector for each row. It views memory contents in table with each row represents a test vector and each column represents byte. Test vector importing wizard has been designed. It can also import test data from test files generated by automatic test pattern generation (ATPG). The user can specify a memory location and number of bytes to be uploaded. The interface memory can be updated manually or automatically after each execution.

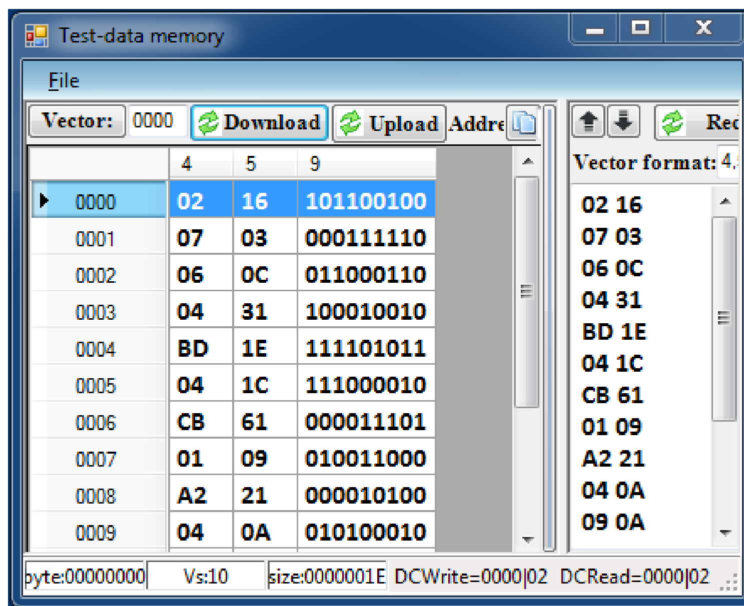


Figure 5.4 : User interface to display memory contenets.

5.2 The Instruction Builder Software

The instruction builder software is designed to automate editing and building the processor in the TACP. It mainly rebuild the sequencer and the control store after each editing on the instructions. Instruction builder software is a graphical user interface that helps on writing and editing microcode instructions. It simplifies editing instructions by automating the processor Verilog code writing when any change on the instruction set is required. It shortens the time need to design and verify the modified version of the instruction set. It also updates and exports the instruction list which is used by the user interface software to write programs.

The sequencer and the control store are the control unit of the microcode architecture. To modify instructions, very accurate modifications needed to be done on these components. Instruction builder software helps to automate this to avoid mistakes and to shorten modification time. The most important job is summarized in these points:

- **Building the sequencer control multiplexer**: The instruction builder has an enumerating algorithm that takes care of the sequencer control signals, enumerates them and produces the control multiplexer Verilog lines. Doing the same work manually after each instruction design modification is a tedious work.
- **Building the control store**: Each control store entry consists of the control multiplexer selection bits, control signals for the data path and next address bits. Adjusting these bits after each instruction modification is anther tedious work. The instruction builder constructs each entry of the control store and builds the control store Verilog code.
- **Export instructions to the assembler**: after instruction modification, the instruction metadata can be exported which contains: instruction name, code and parameter count, names and sizes.

The instruction builder requires the user to enter all needed microcode signals and categorizes them by defining the source-to-destination type (i.e. sequencer to data path, data path to external,

etc.). Then it uses these information with the micro-instructions to write the microcode Verilog code automatically. Appendix B, shows a tutorial of this software.

Figure 5.5 shows a diagram explaining signal types.

1. Sequencer to data path
2. Data path to sequencer
3. Sequencer to external
4. External to sequencer
5. External to data path
6. Data path to external
7. Sequencer to data path and external
8. External to sequencer and data path

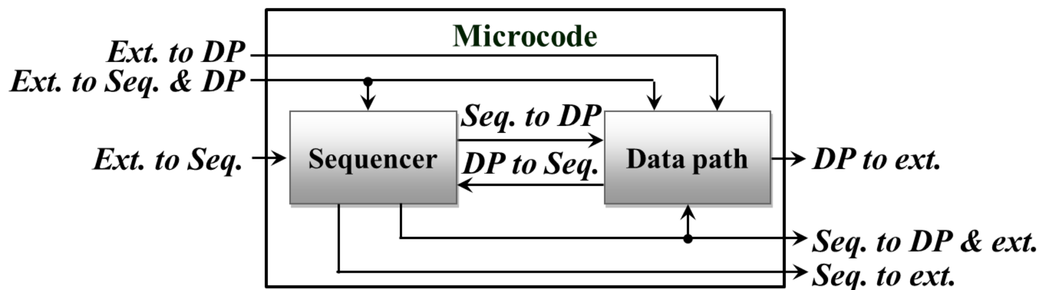


Figure 5.5 : Eight signal types in microcode.

The processor is a microcode architecture that consists of sequencer and data path as depicted in Figure 5.6. The sequencer is the control unit of the processor that sequences the micro instruction execution. It only has five components; address register, comparator, incrementer and two multiplexers.

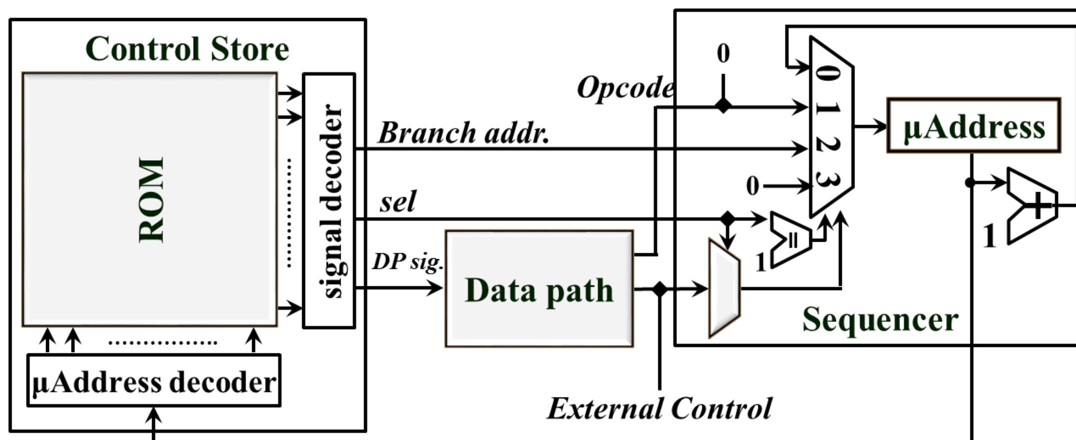


Figure 5.6 : TACP Processor uses the microcode archeticture.

The control store is a ROM that stores all control signals for all execution cycles. Each entry contains the states of all data path control signals and info about the next address. In each clock cycle, the sequencer selects on entry that controls the data path and determine the next micro instruction address.

5.3 TSC Prototyping

Usually, FPGA boards are used for prototyping. Designs are tested and emulated using FPGA boards before sending for fabrication to get an ASIC chips. FPGAs do not have any basic pure logic gates such as AND, OR and NOT available to the designer. Rather, it utilizes the look-up-tables (LUTs) to emulate all combinational functions existing in the design. LUTs make it difficult to do timing analysis and estimation for the prototyped ASIC. For example, all two-input functions will be emulated by identical two-input LUTs. Although functions may vary in their complexity level and delays, LUTs make complicated functions take the same delay simple functions take.

While LUTs may unify functions delay, place-and-route (PAR) phase becomes critical in FPGA design flow. PAR could assign different paths with different delays for the design. Hence, the same design could give different results with different place-and-route algorithms. It happens many

times with a proved and well-simulated design that a slight change on the design could corrupt it and gives strange result. This error rises only because of changing routes from one implementation to another. For instance, it happened that a program which run perfectly as one shot from start to the end, gives different results when we set a break point and run the program in two batches. For all of these timing problems, we searched for solutions that can make the prototyping possible and can came up with a good working processor.

To emulate the test support circuitry (TSC), some clock-related issues such as clock gating and clock multiplexing has to be manipulated. In addition, the TSC has a configurable clock generator that the user can modify its frequency. An alternative circuitry has to be designed to emulate it. The following subsection addresses the emulating problems and the solution we choice to overcome them.

5.3.1 Clock gating

Clock gating is used to reduce dynamic power dissipation since it switches of some parts of the design. The ASIC version of the TSC uses gated clocks to enable and disable the shift registers that surrounds each CUT. Since gated clock is synthesized using look-up-table in FPGAs, the signal will not be a clock anymore and the clock will cause timing problems. Gated clocks could not be emulated this way. One way to overcome this problem is to use the enable signal as shown in Figure 5.7. This solution is used for all shift registers.

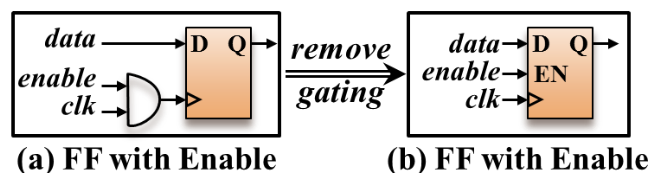


Figure 5.7 : Replace gated clock signal by using enable signal.
gated_clock = CLK && Enable

This solution is not enough to solve all clock gating issues. Some CUTs may not have enable signal. Another solution is to use a dedicated clock tri-state buffer from the FPGA resurces. In our FPGA prototyping using Xilinx Spartan 3A, we used BUFGCE. This solution is used with the clock signals that are feeding CUT and some shift registers and ANDed with the corresponding port selection signal.

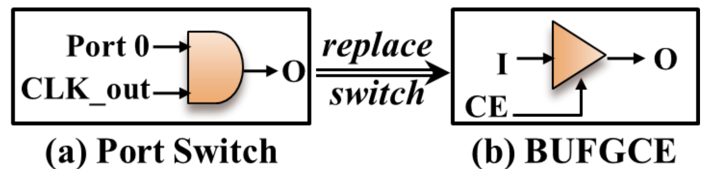


Figure 5.8 : Replace gated clock signal by using FPGA clock tri-state buffer.
gated_clock = Port$_0$ && CLK_out

Figure 5.9 shows a simulation that explains how BUFGCE passes the clock when its chip enable (CE) is high [26]. It is simply work as an AND gate.

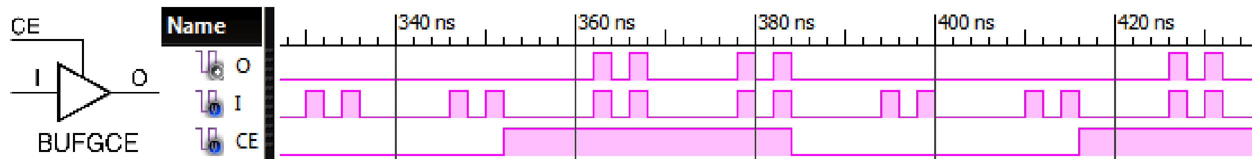


Figure 5.9 : BUFGCE simulation – dedicated clock signals tri-state with no pulse lose.

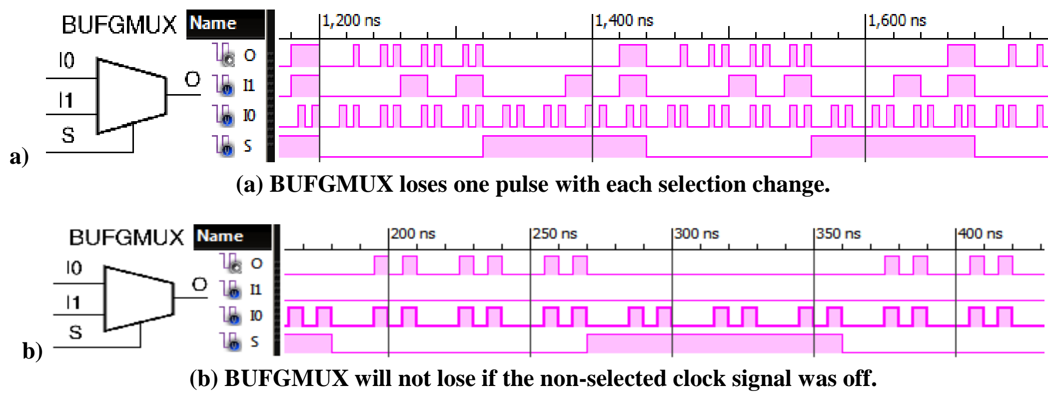
5.3.2 Clock multiplexing

Another issue on ASIC emulation is the multiplexing between two clock signals. The TSC needs to allow the user to select between the high frequency clock to do at-speed testing and the TACP clock to do testing using the TACP clock or to fill the scan chain. The chosen solution for clock multiplexing is to use the FPGA clock multiplexers that are designed for the digital clock managers (DCMs). In our FPGA prototyping using Xilinx Spartan 3A, we used BUFGMUX.

The BUFGMUX has a bad consequence on the resulting clock signal. To multiplex between two clock signals, the BUFGMUX loses the first pulse after each change in its selection.

Farthermore, the multiplexer will not lose any pulses if the previously selected clock signal was off. This is illustrated in the simulation in Figure 5.10 (a). The BUFGMUX loses the first clock pulses with each change on its clock selection as depicted in Figure 5.10 (b).

.a. This losing will not happen if the other clock input was kept high or low during its last selection periods.



(a) BUFGMUX loses one pulse with each selection change.

(b) BUFGMUX will not lose if the non-selected clock signal was off.

Figure 5.10 : The BUFGMUX clock multiplexer simulation.

The behavior of the BUFGMUX has a great impact on the ApplayAndCapture instruction. Since the clock selection and application circuitry (CSaAC) has to generate two consecutive clock pulses, the first call of ApplayAndCapture instruction produces one clock pulse while the next call will produce two. Figure 5.11 shows the CSaAC which uses a multiplexer to select between high frequency clock (HFCLK) and processor clock (TCLK) and an AND gate for its clock gating circuitry. The circuit output the CLK_out clock signal that will be ANDed later for with suitable port selection pin to feed a CUT.

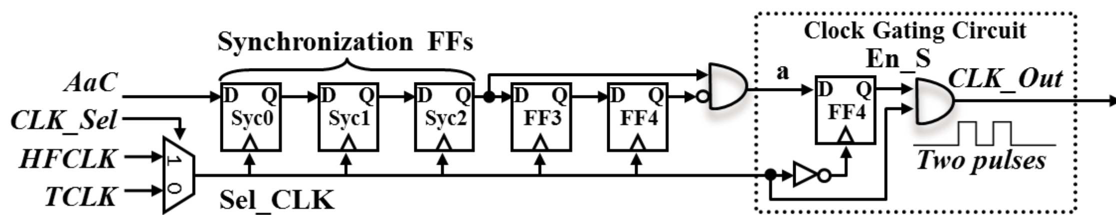


Figure 5.11 : The ASIC version of clock selection and application circuit (CSaAC). The circuitry has a clock multiplexer and a clock gating.

The FPGA implementation of CSaAC along with four CUTs clock gating circuitries are shown in Figure 5.12. BUFGMUX is used as a multiplexer to select one of the two clocks (TACP's clock or the HFCLK). BUFGCE is used to replace clock gating in five locations as depicted in the figure. The CLK_IUT signal feeds directly the corresponding IUT and some of its shift registers. If the CUT has additional port for scan chain, additional clock multiplexer or clock tri-state buffer may be needed that may be controlled by the scan_enable signal.

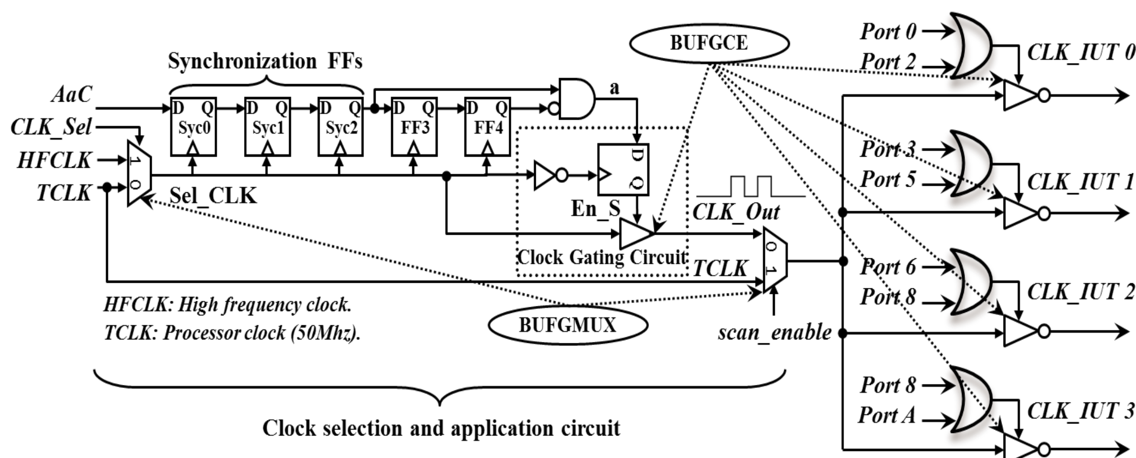


Figure 5.12 : The FPGA implementation of clock selection and application circuit CSaAC with the implementation of four CUTs clock gating. All gated clocks are replaced by FPGA clock buffers BUFGMUX and BUFGCE. The critical path has four level of clock gating.

5.3.3 Emulating the Configurable Clock Generator (CCG)

The TSC has a digitally controlled oscillator (DCO) within the configurable cock generator (CCG). Obviously, ring oscillators cannot be implemented on FPGAs since FPGAs use LUTs.

Each FPGA board has many digital clock managers (DCMs). Most Xilinx FPGA boards has eight DCMs. The DCM can be configured to a specific frequency within the range ~4 MHz-333MHz. Unfortunately, DCM cannot be configured at run-time (unless reconfigurable computing is used).

To emulate the DCO, eight DCMs are configured to generate different clock frequencies. These clock signals are multiplexed using three-bit multiplexer. The resulted clock signals is fed into the divider that has four dividing phase and the resulted frequencies with the original one enter another three-bit multiplexer as depicted in Figure 5.13. This way we get six-bit control word.

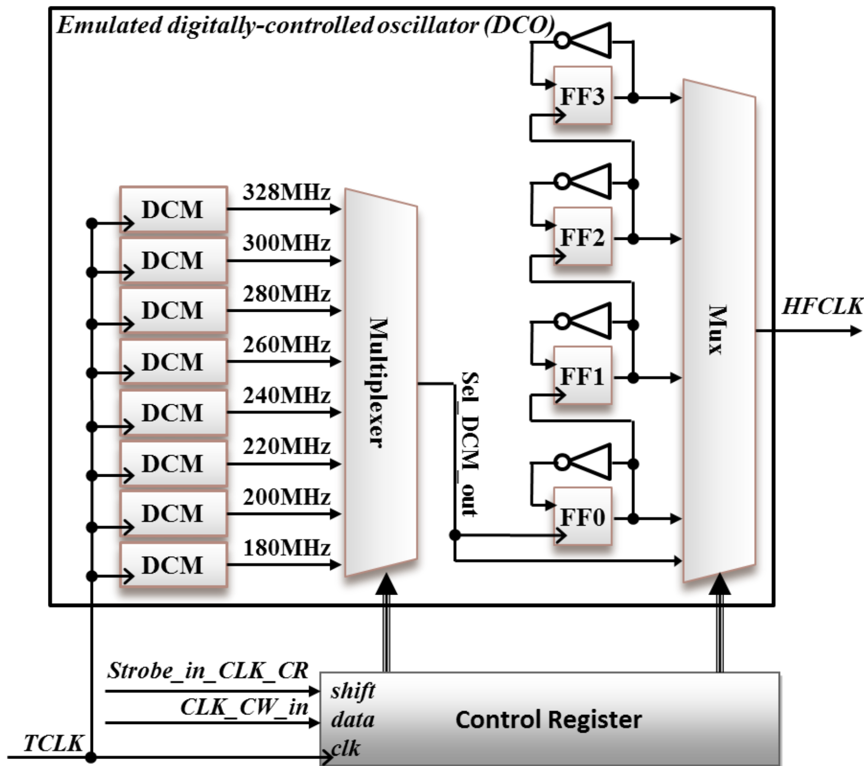


Figure 5.13 : Emulating the DCO using eight DCMs. The DCO is combined with the four phase divider. The frequency is chosen by the 6-bit control word register.

The circuitry is designed in such a way to make the control word value is proportional to the frequency. Table 5.1 shows the resulting frequencies as a function of the six-bit control word. A total of 40 different frequencies ranged between 11MHz and 325MHz could be obtained with this method with control word ranged between 0x38 and 0x1F (in six bits). .

Table 5.1: The generated clock frequencies and their control words using the DCMs in the prototyped chip.

Control Word	Freq. (MHz)	Control Word	Freq. (MHz)	Control Word	Freq. (MHz)	Control Word	Freq. (MHz)
111_000	325	000_010	140	001_100	60	010_110	25
111_001	300	000_011	130	001_101	55	010_111	22.5
111_010	280	000_100	120	001_110	50	011_000	20.3125
111_011	260	000_101	110	001_111	45	011_001	18.75
111_100	240	000_110	100	010_000	40.625	011_010	17.5
111_101	220	000_111	90	010_001	37.5	011_011	16.25
111_110	200	001_000	81.25	010_010	35	011_100	15
111_111	180	001_001	75	010_011	32.5	011_101	13.75
000_000	162.5	001_010	70	010_100	30	011_110	12.5
000_001	150	001_011	65	010_101	27.5	011_111	11.25

5.4 IPs Under Test (IUTs)

In our emulated chip, four circuits are included as a CUTs; a four-bit adder, an eight-bit pipelined adder and two instances s820 which is one of the ISCAS-89 benchmarks. Scan chains are inserted into s820 circuit which has five flip-flops. In one of these benchmarks, the Flip-flops are doubled. The design has ten ports as shown in Figure 5.14. The first and the second IUTs are assigned with two ports for each; one input port and one output port. The third and the fourth IUTs are assigned with three ports for each; one input port, one output port and one scan chain port.

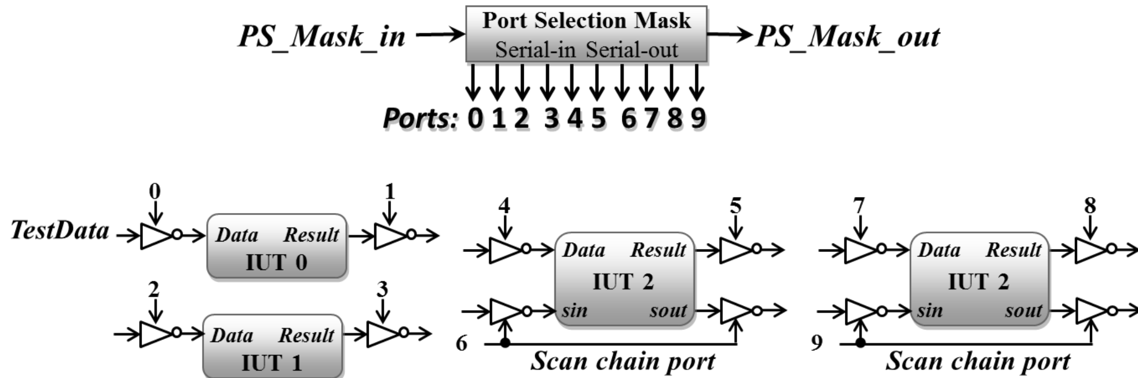


Figure 5.14 : Illustration of assigning the ten port selection bits to the four IUTs.

Each IUT is surrounded by four registers; TAP, APP, CAP and TRP. The user select an IUT and send test data serially to be shifted into the corresponding TAP shift register. Then the data has to be moved to the AAP register which is connected to the IUT inputs. The CAP register is connected to the IUT outputs to capture the result. The TRP shift register reads the data from the CAP register and send them serially to the user. The processor clock (TCLK) drives the TAP and the TRP shift registers operates by to synchronize the shifting with the processor while the application clock (CLK_out) drives the APP and the CAP registers.

1st IUT: 4-bit Combinitional Adder

A combinational 4-bit adder has two 4-bit inputs, carry-in bit, carry-out bit and 4-bit output. The inputs are connected to the 9-bit APP register while the outputs are connected to the 5-bit CAP register as depicted in Figure 5.15.

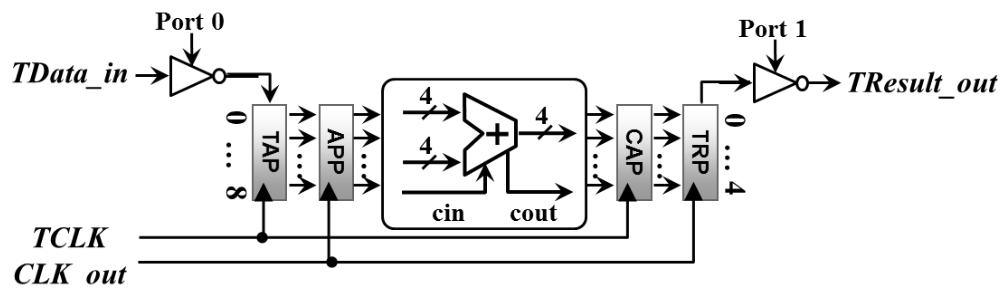


Figure 5.15 : 1st IUT: 4-bit combinitional adder.

- **TData_in**: The received test data bit from the TACP processor.
- **TResult_out**: The test result bit of the IUT.
- **TCLK**: The TACP clock.
- **CLK_out**: The clock out signal of the CSaAC circuit that generates two clock pulses after each applay-and-capture signal sent from TACP.
- **Port 0**: Port selection bit indicates that test data will be shifted into IUT0 input port.
- **Port 1**: Port selection bit indicates that test data will be shifted into IUT0 output port.

2nd IUT: 8-bit pipelined Adder

A pipelined 8-bit adder has two 8-bit inputs, carry-in bit, carry-out bit and 8-bit output. The inputs are connected to the 17-bit APP register while the outputs are connected to the 9-bit CAP register as depicted in Figure 5.16.

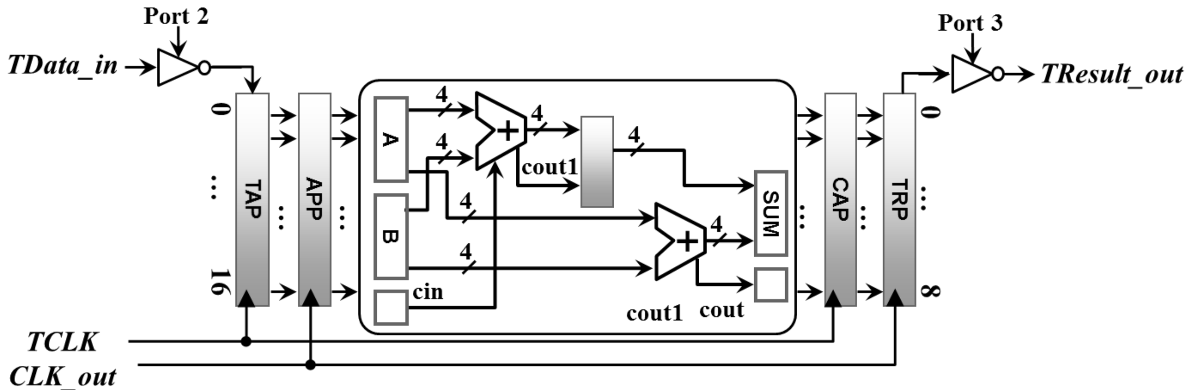


Figure 5.16 : 2nd IUT: 8-bit piplined adder.

- **TData_in**: The received test data bit from the TACP processor.
- **TResult_out**: The test result bit of the IUT.
- **TCLK**: The TACP clock.
- **CLK_out**: The clock out signal of the CSaAC circuit that generates two clock pulses after each applay-and-capture signal sent from TACP.
- **Port 2**: Port selection bit indicates that test data will be shifted into IUT1 input port.
- **Port 3**: Port selection bit indicates that test data will be shifted into IUT1 output port.

3rd IUT: s820 benchmark with scan chain is inserted

The third IUT is s820 which is one of the ISCAS89 benchmarks. It is a sequential circuit that contains five flip-flops. Before using it, a scan chain is inserted by multiplexing the five flip-flops inputs. The circuit has 18 inputs which are connected to the APP register and 19 outputs which are connected to the CAP register as depicted in Figure 5.17. ATPG is used to generate test vectors with their expected results. Each 23-bit test vector starts by the scan chain five bits followed by the 17 inputs.

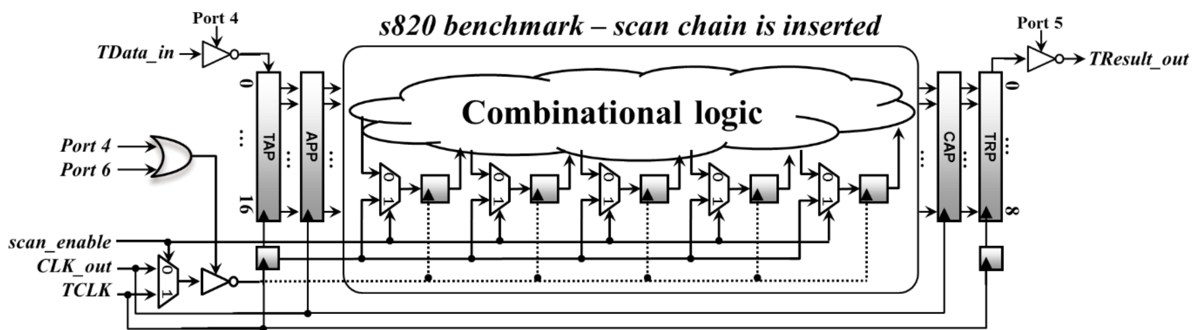


Figure 5.17 : 3rd IUT: s820 benchmark with scan chain is inserted.

- **TData_in**: The received test data bit from the TACP processor.
- **TResult_out**: The test result bit of the IUT.
- **TCLK**: The TACP clock.
- **CLK_out**: The clock out signal of the CSaAC circuit that generates two clock pulses after each applay-and-capture signal sent from TACP.
- **scan_enable**: turn on the scan chain to be filled with test data.
- **Port 4**: Port selection bit indicates that test data will be shifted into IUT3 input port.
- **Port 5**: Port selection bit indicates that test data will be shifted into IUT3 output port.
- **Port 6**: Port selection bit indicates that the scan chain of the IUT3 is selected.

4th IUT: s820 benchmark with scan chain is inserted and flip-flops are doubled.

The scan chain is duplicated in the s820 circuit to make the fourth IUT. Now, there are two columns of the flip-flops as depicted in Figure 5.18. The second row stores the previous state of the first row. One part of the test vector is made ready on the circuit inputs and the other part is scanned into the first row. Since there are two clock pulses with each apply-and-capture instruction, the next state will be captured in the scan chain and the current state outputs will be captured on the CAP register. So, the scan chain with the CAP register will be shifted out to be matched with the expected results.

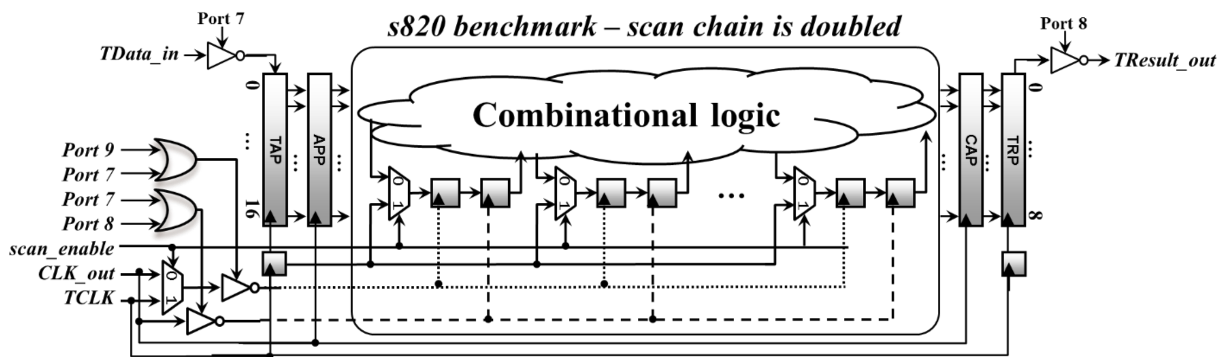


Figure 5.18 : 4th IUT: s820 benchmark with scan chain is inserted and flip-flops are doubled.

- **TData_in**: The received test data bit from the TACP processor.
- **TResult_out**: The test result bit of the IUT.
- **TCLK**: The TACP clock.
- **CLK_out**: The clock out signal of the CSaAC circuit that generates two clock pulses after each applay-and-capture signal sent from TACP.
- **scan_enable**: turn on the scan chain to be filled with test data.
- **Port 7**: Port selection bit indicates that test data will be shifted into IUT4 input port.
- **Port 8**: Port selection bit indicates that test data will be shifted into IUT4 output port.
- **Port 9**: Port selection bit indicates that the scan chain of the IUT4 is selected.

CHAPTER 6

TEST RESULTS

Numerous successful tests were conducted using the implemented FPGA prototype of the complete test and characterization platform. Many of these tests are discussed in this chapter and supported with snapshots in detail to illustrate the capabilities of the platform. Some results shown in this chapter are a little bit different from the ASIC version due to the prototyping issues mentioned in section 5.3. In the following subsections, a small characterization program is explained. After that, there are four programs to test the four IPs under test (IUTs) that shows that the platform is implemented well.

6.1 Maximum Frequency Test

The objective of this test is to determine the maximum possible speed of an IUT (i.e. the maximum frequency it can operate at). This is done by a test program that changes the chip frequency within a loop as shown in Figure 6.1. It starts with a specific frequency specified by initializing the control word to a specific value. The frequency is decremented by incrementing the control word. The loop lasts for nine times. The program starts with setting the number of loops in the counter and setting an initial control word. It ends with decrementing the counter and loop if it does not reach zero. Inside the loop, the processor sends the frequency, measures it and reads the

measured frequency register back to the processor. IP testing has to be done here. If the testing fails then the control-word is incremented and loops again.

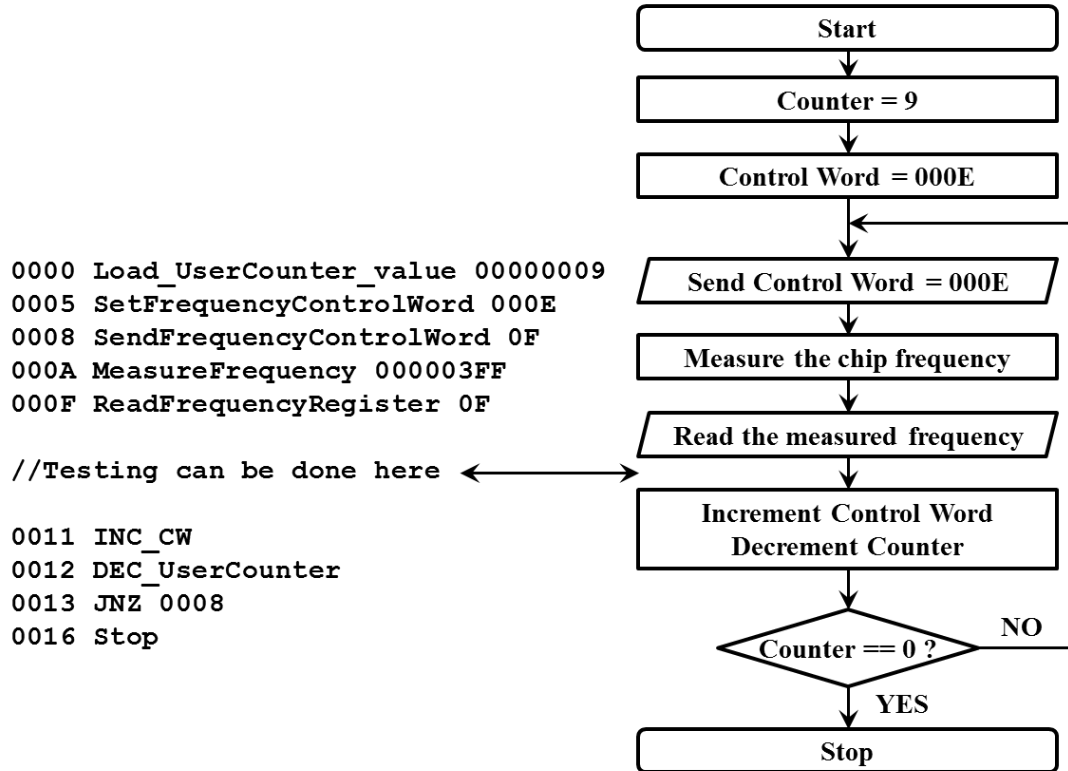


Figure 6.1 : Code and flow chart of the test program that changes frequency within a loop.

Figure 6.2 below shows a snapshot of the execution of the program. For example, take the second execution of the loop. The user counter contains the value is 7 and CW contains 10. The measured frequency is calculated from this formula:

$$\text{Measured Frequency} = \frac{\text{FR} \times \text{Processor Frequency}}{\text{No. of cycles}}$$

The read frequency register on FR is 0340 which is 832 in decimal. The processor frequency is 50 MHz. The number of cycles is (400) which is 1024 in decimal. So, the measured frequency = $\frac{832 \times 50\text{MHz}}{1024}$ = 40.625MHz which is similar to exact value in Table 5.1. Potential error can be measured from the same formula $\frac{1 \times 50\text{MHz}}{1024}$ = 0.05. This error ratio can be minimized if needed by

increasing the No. of cycles. This experiment showed the correct operation of the platform and the accuracy of the frequency measurement scheme.

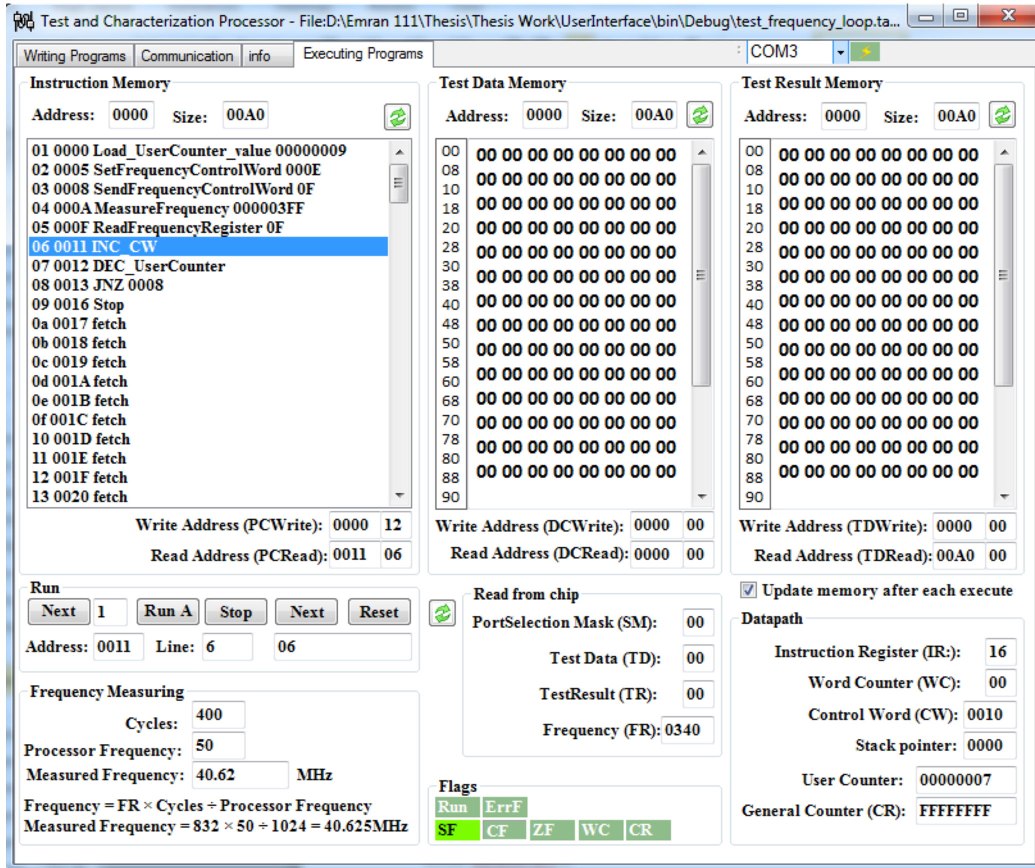


Figure 6.2 : Execution snapshot of the test program that changes frequency within a loop.

6.2 Testing the 1st IUT: the 4bit Combinational Adder

The presented program here is used to test IUT0, the combinational adder. To start, the user writes the program and downloads it to the instruction memory and downloads the test data followed by the expected results to the test data memory. The program sends the ten test-vectors, receives the test-results and compares them with expected-results.

- **The test data (10 vectors, 20 bytes)**: 02, 16, 07, 03, 06, 0C, 04, 31, BD, 1E, 04, 1C, CB, 61, 01, 09, A2, 21, 04, 0A.
- **The expected results (10 vectors, 10 bytes)**: 09, 0A, 12, 06, 1C, 11, 0C, 0A, 03, 0E. That is: 2+6+1=09, 7+3+0=0A, 6+C+0=12 and so on.

- **Examples**: $A + B + cin = \{cout, Sum\}$
  2+6+1=09, 7+3+0=0A, 6+C+0=12 and so on.
- **The program (15 instructions, 58 bytes)**:

```
0000 //Initialize addresses
0000 Load_DCRead 0000
0003 Load_RCWrite FFFF
0006 Load_UserCounter_value 0000000A
000B //Start Test Process Loop
000B SendSelectionMask 00000009, 0000
0012 SendSelectionMask 00000000, 0000
0019 SendTestData 00000000, 01
001F SendTestData 00000000, 02
0025 ApplyAndCapture
0026 ReadResult 00000000, 06
002C DEC_UserCounter
002D JNZ 0019
0030 //Compare All
0030 ResetCompareFlag
0031 Load_RCRead 0000
0034 Compare 00000009
0039 Stop
```

The program starts by setting the test-data reading-address register to 0 and the writing address test results to 0xFFFF. The user counter is loaded by 10 which is the number of test vectors.

After that, the program loads a byte from test data memory and sends four bits from it. It loads another byte and sends five bits from it. This makes 9-bit test data which represents A, B and cin inputs for the adder. After that, apply-and capture is sent to the circuit and the result is read back to be stored in the test result memory as one byte. The addition result is the first 5 bits from the stored byte which represents sum and cout. The process is iterated ten times for ten test vectors. At

the end the program compares the ten-byte expected results with the 10-byte results, update the compare flag (CF) and save the comparison on the test result memory.

Figure 6.3 shows a snapshot of the program after finishing the program execution and reach the stop instruction. CF is off which means that the test result matches the expected result.

Figure 6.4 shows three instances of memory viewer. On the left the test data memory shows the ten test vectors which are twenty bytes. The test result is shown in the middle window. The rightmost window shows that the comparison result in zeroes for all ten comparison which means that all bytes matches the expected results.

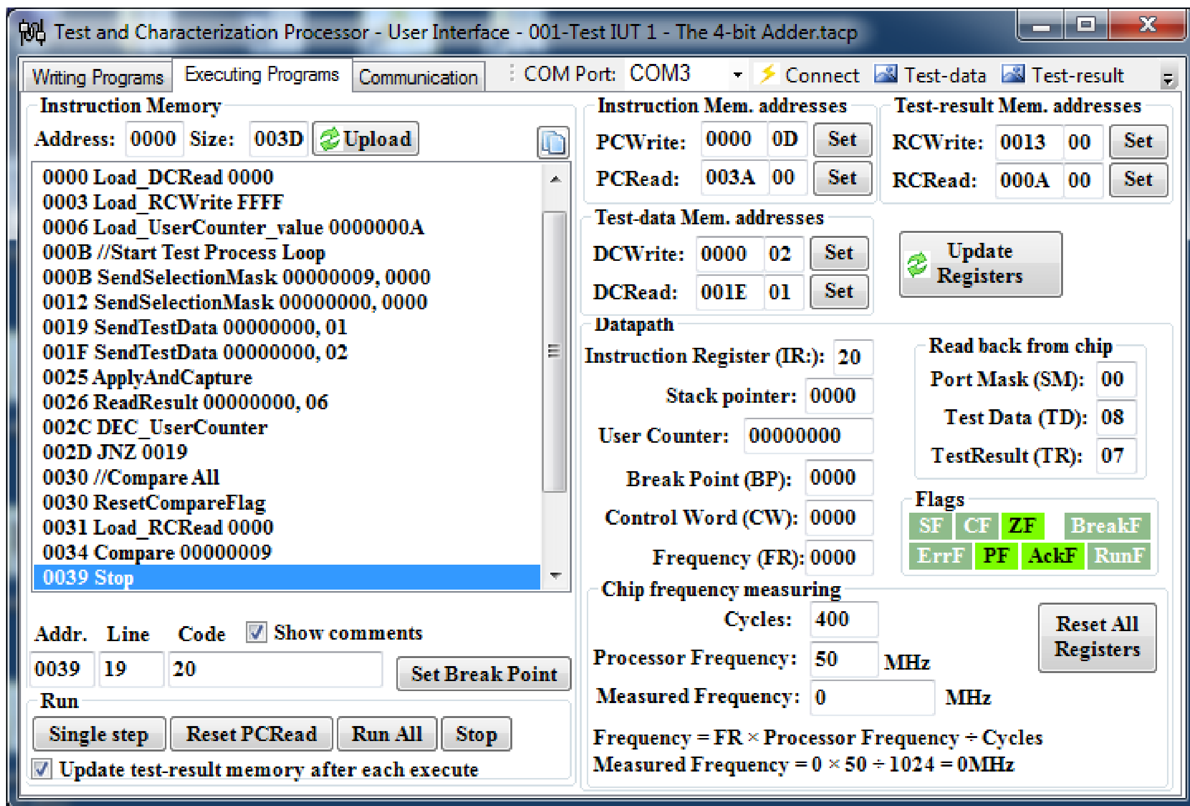


Figure 6.3 : Program execution window snapshot. The program tested a compinational 4-bit adder.

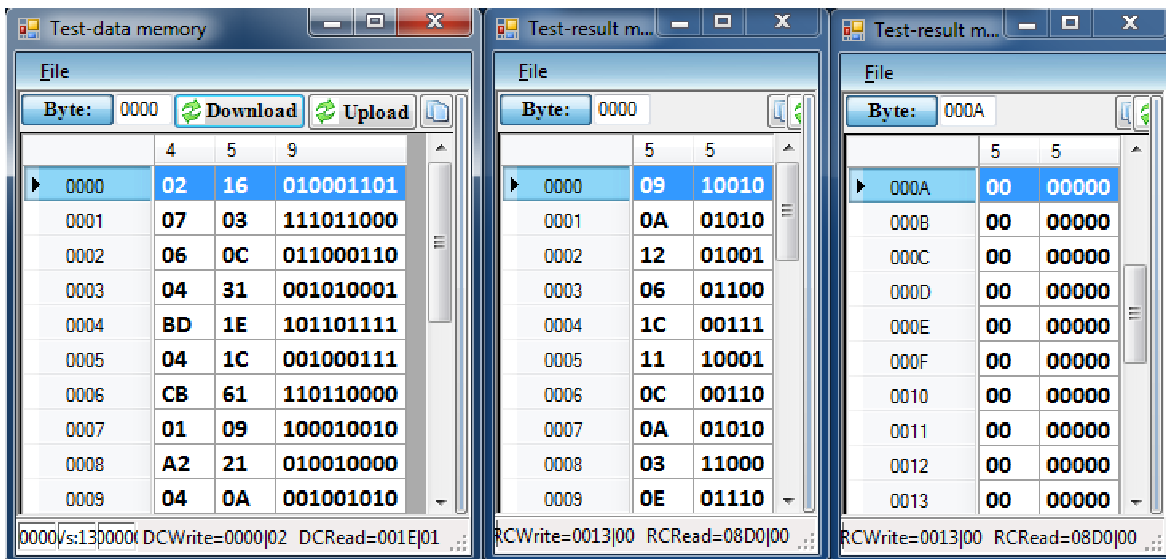


Figure 6.4 : Memory windows snapshot after executing the test program of the compinational 4-bit adder. The three windows show the test data, the test result and the comparision.

6.3 Testing the 2nd IUT: the 8-bit Pipelined Adder

In pipelined circuits multiple clock cycles is required to reach the desired result. In the pipelined adder test program, two (or more) apply-and-capture instructions are needed to empty the pipeline and get the correct summation result.

- **The test data (10 vectors, 30 bytes)**: 7C, 01, 1D, A3, 61, 02, 19, 15, 3C, 57, 16, 8B, 06, 0D, 3D, 01, 03, 00, 11, 76, 07, 0E, 33, 2E, B3, 55, 03, 0C, 96, 43
- **The expected results (10 vectors, 20 bytes)**: 4E, 01, C4, 00, DF, 00, 0D, 01, E4, 00, 04, 00, 87, 00, F1, 00, E8, 00, 42, 00
- **Examples**: *A + B + cin = cout Sum*
  7C+D1+1=01 4E, A3+21+0=00 C4, 19+C5+1=00 DF, 57+B6+0 = 01 0D and so on.
- **The program (17 instructions, 65 bytes)**:

```
0000 //Initialize addresses
0000 Load_DCRead 0000
0003 Load_RCWrite FFFF
0006 Load_UserCounter_value 0000000A
000B //Start Test Process Loop
000B SendSelectionMask 00000009, 0000
0012 SendSelectionMask 00000002, 0002
0019 SendTestData 00000000, 05
001F SendTestData 00000000, 01
0025 SendTestData 00000000, 02
```

002B ApplyAndCapture
002C ApplyAndCapture
002D ReadResult 00000001, 06
0033 DEC_UserCounter
0034 JNZ 0019
0037 //Compare All
0037 ResetCompareFlag
0038 Load_RCRead 0000
003B Compare 00000009
0040 Stop

Figure 6.5 shows a snapshot of the program after finishing the program execution and reach the stop instruction. CF is off which means that the test result matches the expected result.

Figure 6.6 shows two instances of memory viewer. The left window shows the test data memory which consists of 10 vectors, 3 bytes for each. The rightmost window shows that the test results which consists of 10 vectors, 2 bytes for each.

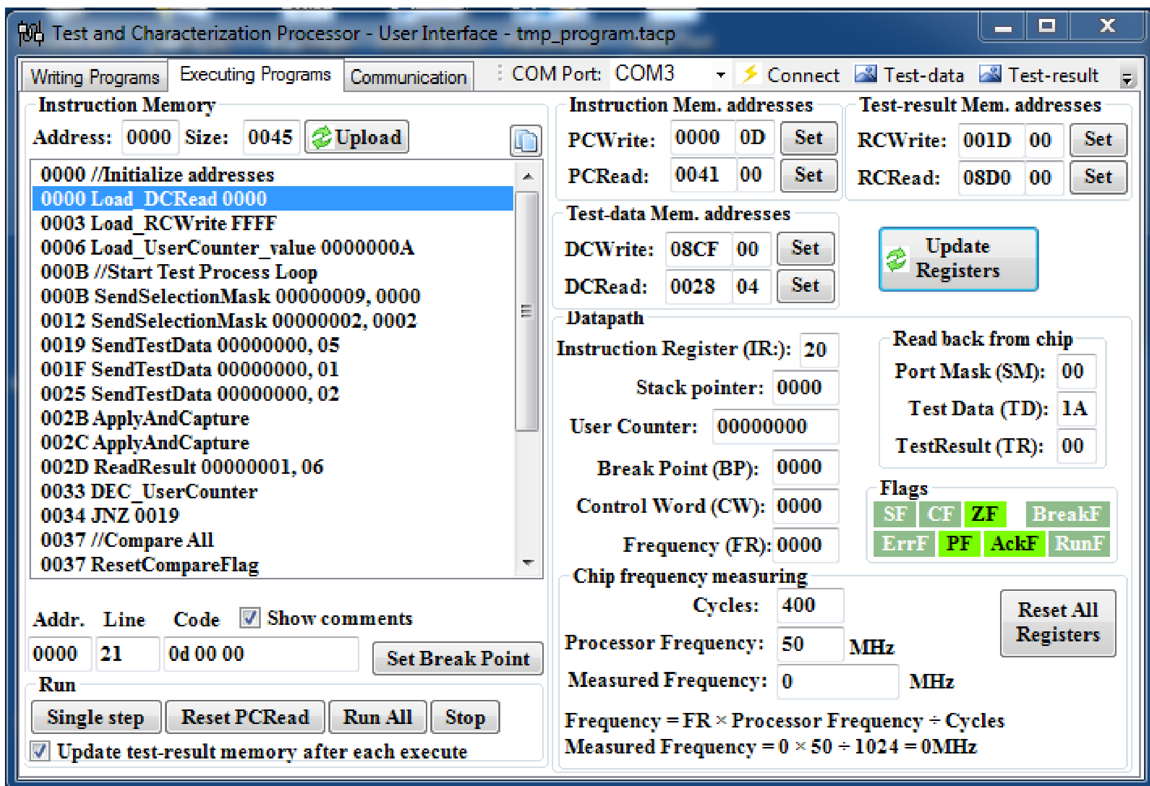


Figure 6.5 : Program execution window snapshot. The program tested a compinational 4-bit adder.

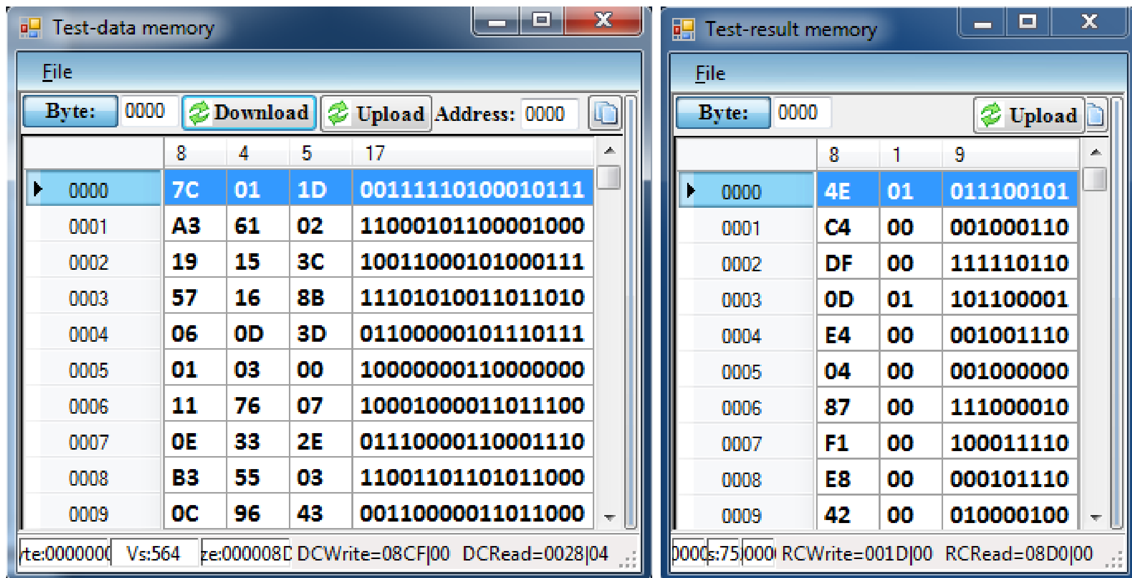


Figure 6.6 : Program execution window snapshot. The program tested a compinational 4-bit adder.

6.4 Complete Testing and Characterizing Program

The program algorithm is illustrated in Figure 6.7. The program starts by initializing the control word and set the at-speed mode (select the high frequency clock). Then it starts two loops using the user counter. It stores the user counter in the test result memory temporary to be able to use it as two counters for the two loops. The inner loop is responsible for sending the 282 test vectors to the TSC and reads the result back. The outer loop has two jobs. First, it is responsible of changing the control word and sending it to the TSC to change the frequency of the high clock. Second, it compares the test results and store the comparison into memory. If the comparison results in non-zero value the comparison flag is turned on the execution get out of the outer loop. Otherwise the outer loop continues until reaching the maximum number of iteration.

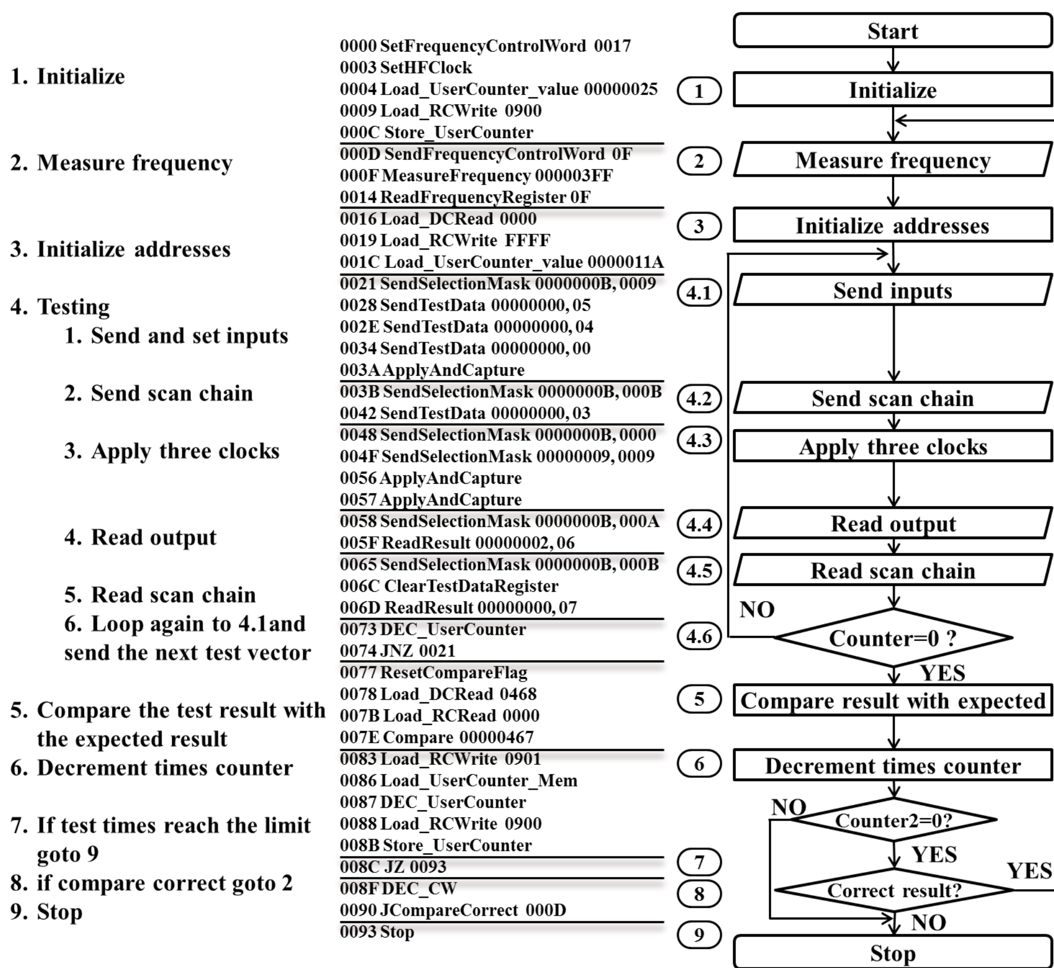


Figure 6.7 : A complete test program and its flow chart for the S820S benchmark IUT.

IUT 3 and IUT 4 have scan chains. The program sends the input data at first then it send apply-and-capture signal to move the input data from the TAP to the CAP to make the circuit inputs ready. After that, the program fills the scan chain. At this point, the IUT inputs and scan chain are ready with the test vector and apply-and-capture can be sent.

Reading the result also consists of two phases, reading the output and reading the scan chain.

Testing and characterizing the 3rd IUT

In a normal ASIC test it has to be one apply-and-capture signal however, in FPGA prototype, the first apply-and-capture always results in one clock pulse while the followed ones produce two

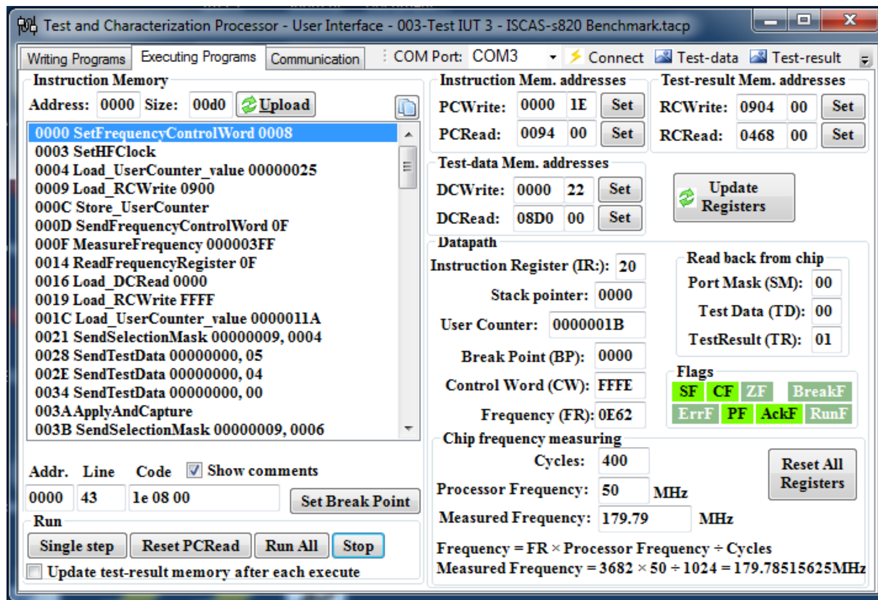
pulses due to the prototyping issues mentioned in section 5.3. Hence, the test result in this program resulted by applying three clocks to the IUT 3. The associated expected results also are generated using a simulator and taken after applying three clock pulses.

- **The test data (282 vectors, 1128 bytes)**: 22, 76, 07, 03, 66, 5C, 04, 01, …
- **The expected results (282 vectors, 1128 bytes)**: 00, 60, 00, 00, 00, 60, 00, 00, …
- **The program (43 instructions, 148 bytes)**:

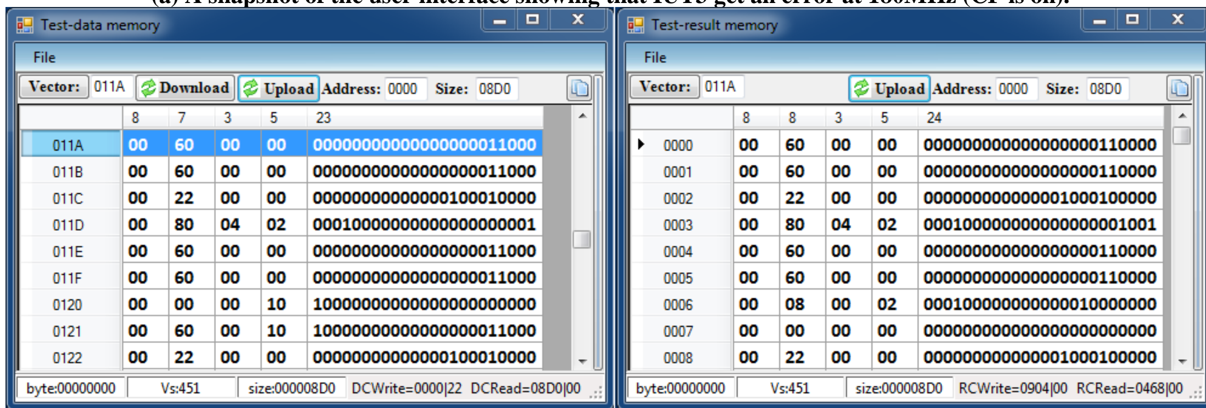
```
0000 SetFrequencyControlWord 0008
0003 SetHFClock
0004 Load_UserCounter_value 00000025
0009 Load_RCWrite 0900
000C Store_UserCounter
000D SendFrequencyControlWord 0F
000F MeasureFrequency 000003FF
0014 ReadFrequencyRegister 0F
0016 Load_DCRead 0000
0019 Load_RCWrite FFFF
001C Load_UserCounter_value 0000011A
0021 SendSelectionMask 00000009, 0004
0028 SendTestData 00000000, 05
002E SendTestData 00000000, 04
0034 SendTestData 00000000, 00
003A ApplyAndCapture
003B SendSelectionMask 00000009, 0006
0042 SendTestData 00000000, 03
0048 SendSelectionMask 00000009, 0000
004F SendSelectionMask 00000004, 0004
0056 ApplyAndCapture
0057 ApplyAndCapture
0058 SendSelectionMask 00000009, 0005
005F ReadResult 00000002, 06
0065 SendSelectionMask 00000009, 0006
006C ClearTestDataRegister
006D ReadResult 00000000, 07
0073 DEC_UserCounter
0074 JNZ 0021
0077 ResetCompareFlag
0078 Load_DCRead 0468
007B Load_RCRead 0000
007E Compare 00000467
0083 Load_RCWrite 0901
```

```
0086 Load_UserCounter_Mem
0087 DEC_UserCounter
0088 Load_RCWrite 0900
008B Store_UserCounter
008C JZ 0093
008F DEC_CW
0090 JCompareCorrect 000D
0093 Stop
```

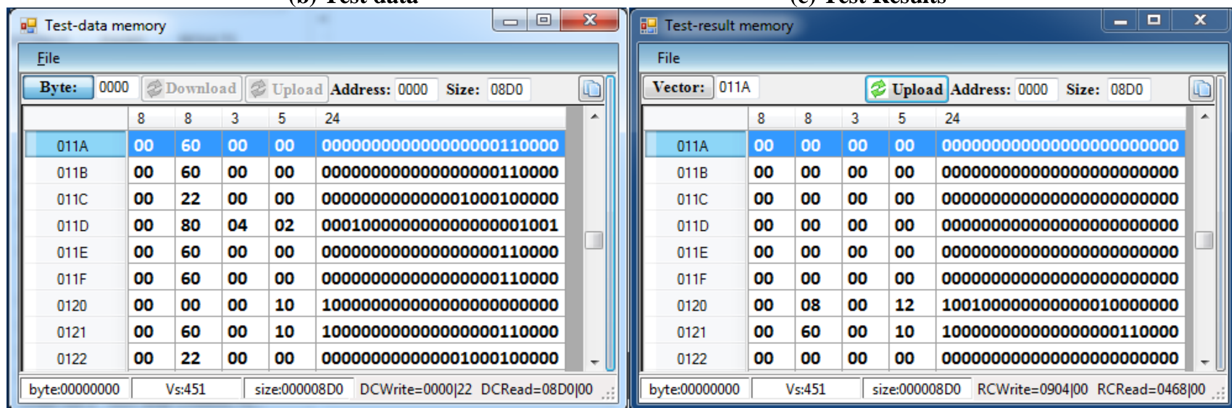
Figure 6.8 shows a snap shot of the user interface tool after executing the program with four snapshots from memories for the test data, test results, expected results and comparison. The latest test was conducted using a 180 MHz. The CF flag indicates that the comparison detect a discrepancy between results and expected results at this speed and the comparison window shows where the discrepancy exists.



(a) A snapshot of the user interface showing that IUT3 get an error at 180MHz (CF is on).



(b) Test data **(c) Test Results**



(d) Expected Results **(e) Comparison**

Figure 6.8 : IUT3 testing: Snapshots for the user interface with four instances of the memory viewer after executing the IUT 3 test program. At this point, not all test results matches the expected. The comparison window shows some non-zero values where a difference exists.This indicates that the chip cannot handle the currrent frequency.

Testing and characterizing the 4rd IUT

In a normal ASIC test it has to be one apply-and-capture signal however, in FPGA prototype, the first apply-and-capture always results in one clock pulse while the followed ones produce two pulses due to the prototyping issues mentioned in section 5.3. Hence, the test result in this program resulted by applying three clocks to the IUT 4. Two pulses will generate the results and the third one will produce the result after applying two pulses. The associated expected results also are generated using a simulator and taken after applying two clock pulses.

- **The test data (282 vectors, 1128 bytes)**: 22, 76, 07, 03, 66, 5C, 04, 01, …
- **The expected results (282 vectors, 1128 bytes)**: 00, 60, 00, 00, 00, 60, 00, 00, …
- **The program (43 instructions, 148 bytes)**:

```
0000 //initialize
0000 SetFrequencyControlWord 0016
0003 SetHFClock
0004 Load_UserCounter_value 00000025
0009 Load_RCWrite 0900
000C Store_UserCounter
000D //Measure frequency
000D SendFrequencyControlWord 0F
000F MeasureFrequency 000003FF
0014 ReadFrequencyRegister 0F
0016 //Initialize addresses
0016 Load_DCRead 0000
0019 Load_RCWrite FFFF
001C Load_UserCounter_value 0000011A
0021 //Send inputs
0021 SendSelectionMask 00000009, 0007
0028 SendTestData 00000000, 05
002E SendTestData 00000000, 04
0034 SendTestData 00000000, 00
003A ApplyAndCapture
003B //Send scan chain
003B SendSelectionMask 00000009, 0009
0042 SendTestData 00000000, 03
0048 //Apply three clocks
0048 SendSelectionMask 00000009, 0000
004F SendSelectionMask 00000007, 0007
```

```
0056 ApplyAndCapture
0057 ApplyAndCapture
0058 //Read output
0058 SendSelectionMask 00000009, 0008
005F ReadResult 00000002, 06
0065 //Read scan chain
0065 SendSelectionMask 00000009, 0009
006C ClearTestDataRegister
006D ReadResult 00000000, 07
0073 //Loop for the next test vector
0073 DEC_UserCounter
0074 JNZ 0021
0077 //Compare test result with expected
0077 ResetCompareFlag
0078 Load_DCRead 0468
007B Load_RCRead 0000
007E Compare 00000467
0083 //Decrement times counter
0083 Load_RCWrite 0901
0086 Load_UserCounter_Mem
0087 DEC_UserCounter
0088 Load_RCWrite 0900
008B Store_UserCounter
008C //if time reach maximum then exit
008C JZ 0093
008F //if correct result change frequency and loop
008F DEC_CW
0090 JCompareCorrect 000D
0093 //Exit
0093 Stop
```

(a) A snapshot of the user interface showing that IUT3 get an error at 180MHz (CF is on).

(b) Test data

(c) Test Results

(d) Expected Results

(e) Comparison

Figure 6.9 : IUT 4 testing: Snapshots for the user interface with four instances of the memory viewer after executing the test program. At this point, not all test results matches the expected. The comparison window shows some non-zero values where a difference exists.This indicates that the chip cannot handle the currrent frequency.

6.5 Testing of the Loop Back from the chip

The developed platform allows testing of the **TSC** circuitry itself by scanning in and out test data and port selection masks. Figure 6.10 below shows a small program execution that sends test data to the chip. The purpose of this program is to see the returned back data in the TD register. The test data memory is initially loaded with the test data memory with the data shown in Figure 6.10 which starts with 22 at location 0 in the test data memory. The program, starting with address 0, it selects the tenth port which represents the input port for the fourth IUT. Then it sends 8 bits from the 1st byte 0x22, then sends 7 bits from the second byte 0x76, then sends three bits from the third byte 0x07 then 8 bits from the forth byte 0x03. In each time, the test data is sent to the **TSC** and is automatically looped back to the **TACP**. The total number of sent bits is 26 which are 01000100_0110111_111_11000000. Since the number of inputs in IUT4 is 18 bits, the first 8 bits will be returned back to the processor (value of 22). As can be seen in the user interface (Figure 6.10), TD register contains 22 and this is correct.

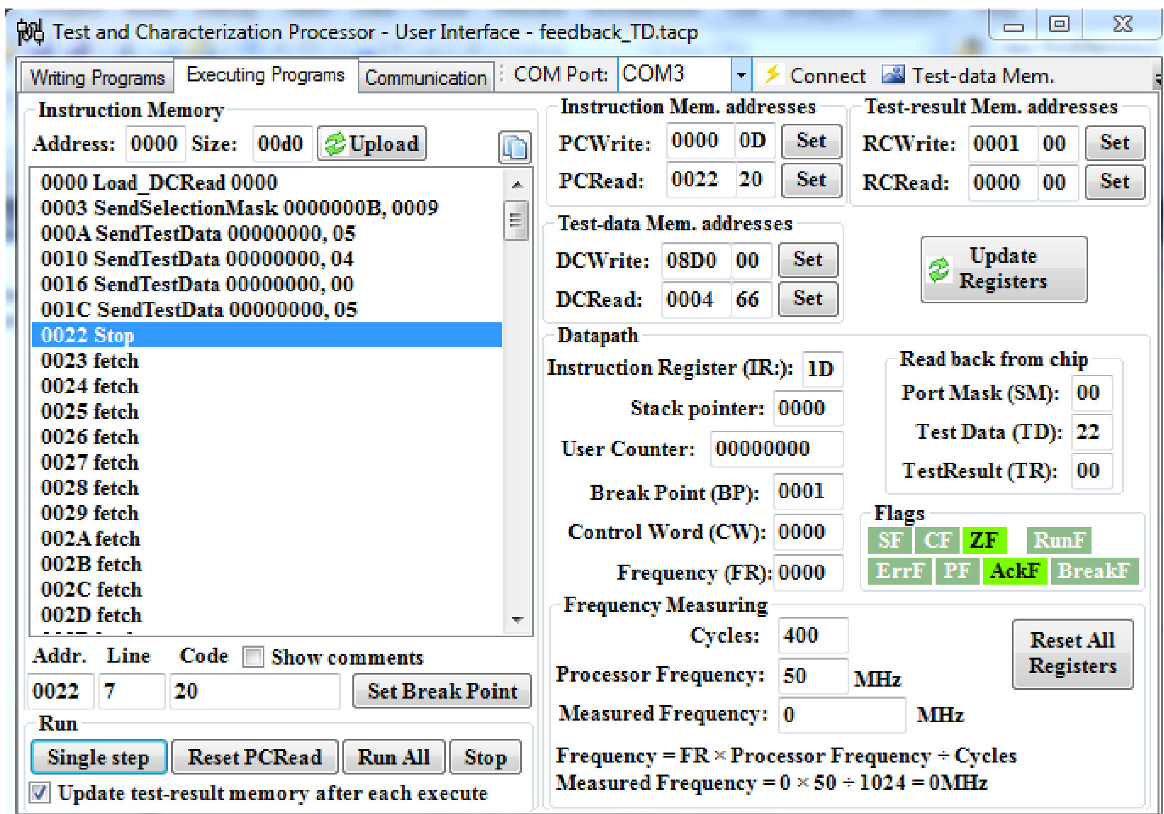


Figure 6.10 : Loop back testing; test-data is sent to the TSC and received back. The TD register gets back the returned test-data (TD = 0x22).

Another test program was run to check the selection mask register SM, Figure 6.11. As can be seen from the figure, the **SM** register contains 60 which is **0110_0000_0000**. This means that, out of the available twelve ports, ports number nine and ten had been selected. To achieve this, the test program first sends the 12-bit sequence **1000_0000_0000** using the following instruction:

0000 SendSelectionMask 0000000B, 0000

Then it sends the following 10-bit sequence **0000_0000_01** using the following instruction:

0007 SendSelectionMask 00000009, 0009

Hence the value of the 12-bit selection-mask register SM becomes **0000_0000_0110**. Finally, the byte **0110_0000** is shifted out to the SM register using the instruction:

000E SendSelectionMask 00000007, 0000

Thus we get the value 60 for SM register in the processor as it shown in Figure 6.11 below.

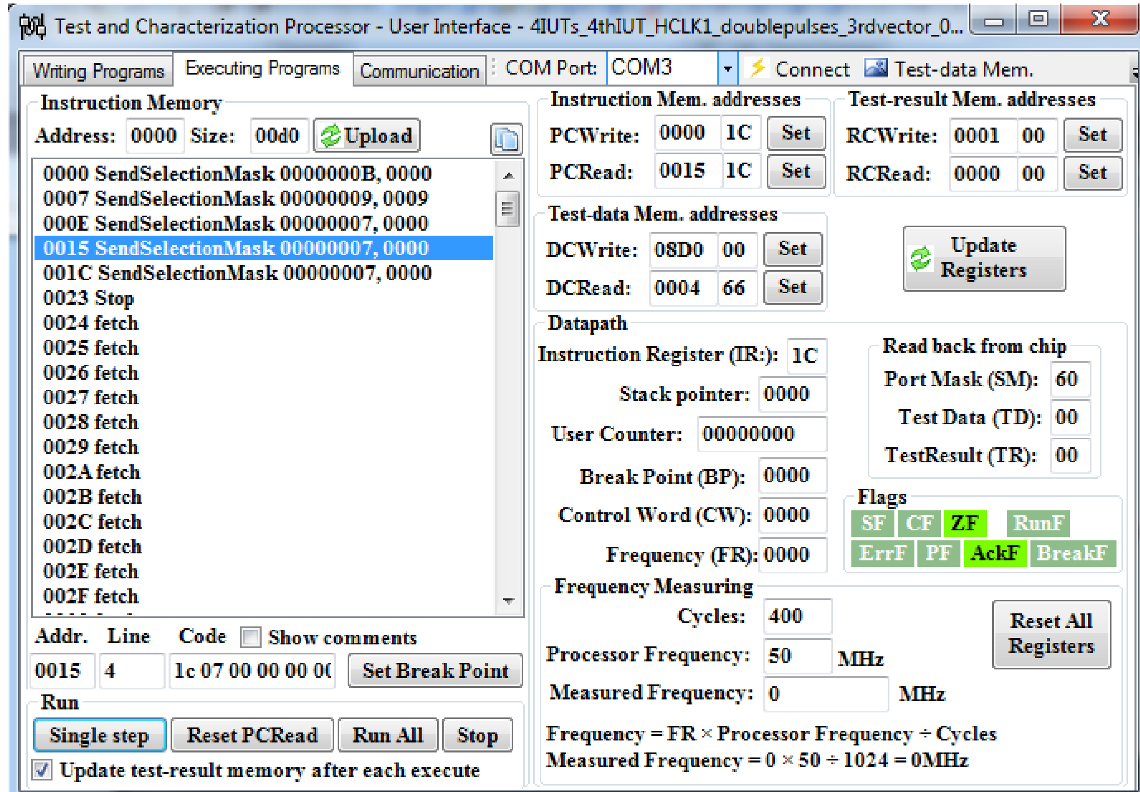


Figure 6.11 : Test program for the selection mask SM register. SM contains the looped back selection mask that was sent (SM = 0x60)..

CONCLUSION AND FUTURE WORK

Test and characterization processor is designed and implemented on FPGA boards. TSC with some IPs were emulated on another FPGA board. Graphical user interface is designed to effectively use the processor. The system is successfully tested running many programs on it. Then all circuitries are designed. Implementation is done on two Spartan 3A FPGA boards. Graphical user interface is designed and tested. Comprehensive test programs are written and run on the processor. Test results show that the processor works as planned.

CHAPTER 7

APPENDIX

A. Instruction-Set List with their microinstructions

Explaining and designing of some instructions is stated in section 4.6. In this section the complete list of the thirty three designed instructions and their microinstructions is listed.

A.1. fetch

It is the default instruction that exists at address 0 so, the execution will start with it. It selects the next instruction to be execute and branch to it by loading the instruction register (IR) address. Each instruction branches to address 0 after finishing its execution. That is, fetch instruction is executed before each instruction. The fetch instruction also holds the execution if the next_instruction flag is off which enable running the program in a single step mode. It also set the processor busy flag on while the processor executes instructions.

<table>
<tr><th>Opcode</th><th>Instruction name</th><th>Parameters</th></tr>
<tr><td>00</td><td>feach</td><td>no parameters</td></tr>
<tr><td colspan="3">micro instructions:
00:0 NOP
00:0 ResetBusy
00:1 if not (next_instruction) Branch 00:0</td></tr>
</table>

```
00:2 Load_IR_Instruction
00:2 SetBusy
00:3 Increment_PC
00:3 NOP
```

A.2. SendSelectionMask

Send a bit stream to the port selection mask register on the chip. It decodes the port no. and put it in specific window length before sending it. Usually the window length is the total number of ports.

Opcode	Instruction name	Parameters	Size (bytes)
1C	SendSelectionMask	Window length-1	4
		Bit No.(0 to n-1)	2

```
micro instructions:
    1C:0 Increment_PC
    1C:1 Increment_PC
    1C:1 Load_CR_Low_Instruction2
    1C:2 Increment_PC
    1C:3 Increment_PC
    1C:3 Load_CR_High_Instruction2
    1C:4 Increment_PC
    1C:5 Decrement_CR
    1C:5 Strobe_in_PMask
    1C:5 if not (CR_IsZero)  Branch 1C:5
    1C:6 Increment_PC
```

A.3. SendTestData

Send test data from the address specified by DCRead register.

Opcode	Instruction name	Parameters	Size (bytes)
1D	SendTestData	No. of words-1	4
		Bits per word-3	1

```
micro instructions:
    1D:0 Increment_PC
    1D:1 Increment_PC
    1D:1 Load_CR_Low_Instruction2
    1D:2 Increment_PC
    1D:3 Increment_PC
    1D:3 Load_CR_High_Instruction2
    1D:4 Branch 1D:6
```

```
    1D:4 Increment_DC
    1D:4 Load_TD_TestData
    1D:4 Load_WC_Instruction
    1D:5 Increment_DC
    1D:5 Load_TD_TestData
    1D:5 Load_WC_Instruction
    1D:5 Strobe_in_TData
    1D:6 Decrement_WC
    1D:6 Shift_TestData
    1D:6 Strobe_in_TData
    1D:6 if not (WC_IsZero)  Branch 1D:6
    1D:7 Decrement_CR
    1D:7 Shift_TestData
    1D:7 Strobe_in_TData
    1D:7 if not (CR_IsZero)  Branch 1D:5
    1D:8 Increment_PC
    1D:8 Shift_TestData
    1D:8 Strobe_in_TData
```

A.4. ReadResult

Read results back from the chip and strore them in test result memory in the address specified by RCWrite register.

Opcode	Instruction name	Parameters	Size (bytes)
17	ReadResult	No. of Words-1	4
		Bits pre word-2	1

```
micro instructions:
    17:0 Increment_PC
    17:1 Increment_PC
    17:1 Load_CR_Low_Instruction2
    17:2 Increment_PC
    17:3 ClearTR
    17:3 Increment_PC
    17:3 Load_CR_High_Instruction2
    17:4 Load_WC_Instruction
    17:4 Strobe_out_TR
    17:5 Decrement_WC
    17:5 Strobe_out_TR
    17:5 if not (WC_IsZero)  Branch 17:5
    17:6 Decrement_CR
    17:6 Increment_RCWrite
    17:6 Load_WC_Instruction
```

17:6 Store_TestResults_TR 17:6 Strobe_out_TR 17:6 if not (CR_IsZero) Branch 17:5 17:7 Increment_PC

A.5. ApplyAndCapture

Send an apply-and-capture signal and leave two clock cycles before and after sending.

<table>
<tr><td>Opcode</td><td>Instruction name</td><td>Parameters</td></tr>
<tr><td>01</td><td>ApplyAndCapture</td><td>no parameters</td></tr>
<tr><td colspan="3">micro instructions:
01:0 NOP
01:1 NOP
01:2 NOP
01:3 AaC
01:4 AaC
01:5 AaC
01:6 AaC
01:6 NOP
01:7 NOP
01:8 NOP</td></tr>
</table>

A.6. Compare

Compare (xor) between two memory locations and store the comparision result on another location. The addresses has to be set before calling this instruction using these instructions: Load_DCRead: the expected result location. LoadRCWrite: location to save comparison results. LoadRCRead: the stored test results location.

<table>
<tr><td>Opcode</td><td>Instruction name</td><td>Parameters</td><td>Size (bytes)</td></tr>
<tr><td>03</td><td>Compare</td><td>No. of words-1</td><td>4</td></tr>
<tr><td colspan="4">micro instructions:
03:0 Increment_PC
03:1 Increment_PC
03:1 Load_CR_Low_Instruction2
03:2 Increment_PC
03:3 Increment_PC
03:3 Load_CR_High_Instruction2
03:4 Decrement_CR
03:4 Increment_DC</td></tr>
</table>

03:4 Increment_RCRead 03:4 Increment_RCWrite 03:4 Store_TestResults_Compare 03:4 if not (CR_IsZero) Branch 03:4

A.7. Load_DCRead

Set a value to the test-data memory reading address register (b).

Opcode	Instruction name	Parameters	Size (bytes)
0D	Load_DCRead	Data reading address	2
micro instructions: 0D:0 Increment_PC 0D:1 Increment_PC 0D:1 Load_DC_Instruction2			

A.8. Load_RCRead

Set a value to the test-result memory reading address register (b)

Opcode	Instruction name	Parameters	Size (bytes)
0F	Load_RCRead	Result reading address	2
micro instructions: 0F:0 Increment_PC 0F:1 Increment_PC 0F:1 Load_RCRead_Instruction2			

A.9. Load_RCWrite

Set a value to the test-result memory writing address register (a)

Opcode	Instruction name	Parameters	Size (bytes)
10	Load_RCWrite	Result writing address	2
micro instructions: 10:0 Increment_PC 10:1 Increment_PC 10:1 Load_RCWrite_Instruction2			

A.10. ResetCompareFlag

Clear the comparing result flag (CF)

Opcode	Instruction name	Parameters
18	ResetCompareFlag	no parameters

micro instructions:
18:0 ResetCF

A.11. JCompareCorrect

Jump if compare flag (CF) is not zero

Opcode	Instruction name	Parameters	Size (bytes)
08	JCompareCorrect	Branching address	2
micro instructions: 08:0 Increment_PC 08:0 if not (CF_IsNotEqual) Branch 08:2 08:1 Branch 00:0 08:1 Increment_PC 08:2 Load_PC_Instruction2			

A.12. JCompareError

Jump if compare flag (CF) is zero

Opcode	Instruction name	Parameters	Size (bytes)
09	JCompareError	Branching address	2
micro instructions: 09:0 Increment_PC 09:0 if (CF_IsNotEqual) Branch 09:2 09:1 Branch 00:0 09:1 Increment_PC 09:2 Load_PC_Instruction2			

A.13. SetFrequencyControlWord

Set a value in the frequency control word register (CW) to be shifted out later to the chip.

Opcode	Instruction name	Parameters	Size (bytes)
1E	SetFrequencyControlWord	Control word	2
micro instructions: 1E:0 Increment_PC 1E:1 Increment_PC 1E:1 Load_CW_Instruction2			

A.14. SendFrequencyControlWord

Shift out the frequency control word register (CW) to the chip

Opcode	Instruction name	Parameters	Size (bytes)
1B	SendFrequencyControlWord	Control word length-1	1
micro instructions: 1B:0 Increment_PC 1B:0 Load_WC_Instruction 1B:1 Decrement_WC 1B:1 Strobe_in_CLK_CR 1B:1 if not (WC_IsZero) Branch 1B:1			

A.15. MeasureFrequency

Run the frequency measuring algorithm to measure chip high frequency clock

Opcode	Instruction name	Parameters	Size (bytes)
14	MeasureFrequency	No. of cycles-1	4
micro instructions: 14:0 Increment_PC 14:1 Increment_PC 14:1 Load_CR_Low_Instruction2 14:2 Increment_PC 14:3 Increment_PC 14:3 Load_CR_High_Instruction2 14:4 if not (HFCLK_Meas_ACK) Branch 14:4 14:5 Decrement_CR 14:5 HFCLK_Meas_Req 14:5 if not (CR_IsZero) Branch 14:5 14:6 if not (HFCLK_Meas_ACK) Branch 14:6			

A.16. ReadFrequencyRegister

Copy the chip measured-frequency register (FR) to the precessor frequency register (FR), It shifts right the two registers simultanously.

Opcode	Instruction name	Parameters	Size (bytes)
16	ReadFrequencyRegister	Word length-1	1
micro instructions: 16:0 Increment_PC 16:0 Load_WC_Instruction 16:1 Decrement_WC 16:1 Strobe_out_CLK_FR 16:1 if not (WC_IsZero) Branch 16:1			

A.17. INC_CW

Increment the frequency control word register (CW) to be sent to the chip later

Opcode	Instruction name	Parameters
06	INC_CW	no parameters
micro instructions: 06:0 INC_CW		

A.18. DEC_CW

Decrement the frequency control word register (CW) to be sent to the chip later

Opcode	Instruction name	Parameters
04	DEC_CW	no parameters
micro instructions: 04:0 DEC_CW		

A.19. SetHFClock

Set the high frequency clock flag (SF).

Opcode	Instruction name	Parameters
1F	SetHFClock	no parameters
micro instructions: 1F:0 SetHFClock		

A.20. ResetHFClock

Reset the high frequency clock flag (SF).

Opcode	Instruction name	Parameters
19	ResetCompareFlag	no parameters
micro instructions: 19:0 ResetHFClock		

A.21. Load_UserCounter_value

Load an immediate value to the user counter register (UC).

Opcode	Instruction name	Parameters	Size (bytes)
12	Load_UserCounter_value	Immediate value	4
micro instructions:			

12:0 Increment_PC 12:1 Increment_PC 12:1 Load_UC_Low 12:2 Increment_PC 12:3 Increment_PC 12:3 Load_UC_High

A.22. Load_UserCounter_Mem

Load a value from the test-result memory to the user counter register (CU). Load_RCWrite instruction has to be called before this instruction to set the loading address.

Opcode	Instruction name	Parameters
11	Load_UserCounter_Mem	no parameters
micro instructions: 11:0 Increment_RCWrite 11:0 Load_UC_TR1 11:1 Increment_RCWrite 11:1 Load_UC_TR2 11:2 Increment_RCWrite 11:2 Load_UC_TR3 11:3 Increment_RCWrite 11:3 Load_UC_TR4		

A.23. Store_UserCounter

Store the user counter register (UC) value into test-result memory. Load_RCWrite instruction has to be called before this instruction to set the Storing address.

Opcode	Instruction name	Parameters
21	Store_UserCounter	no parameters
micro instructions: 21:0 Increment_RCWrite 21:0 Store_UC1 21:1 Increment_RCWrite 21:1 Store_UC2 21:2 Increment_RCWrite 21:2 Store_UC3 21:3 Increment_RCWrite 21:3 Store_UC4		

A.24. INC_UserCounter

Incerment the user counter register (UC).

Opcode	Instruction name	Parameters
07	INC_UserCounter	no parameters
micro instructions: 07:0 INC_UC		

A.25. DEC_UserCounter

Decerment the user counter register (UC).

Opcode	Instruction name	Parameters
05	DEC_UserCounter	no parameters
micro instructions: 05:0 DEC_UC		

A.26. JNZ

Jump if the user counter (UC) register is not zero.

Opcode	Instruction name	Parameters	Size (bytes)
0A	JNZ	Branch address	2
micro instructions: 0A:0 Increment_PC 0A:0 if not (UC_IsZero) Branch 0A:2 0A:1 Branch 00:0 0A:1 Increment_PC 0A:2 Load_PC_Instruction2			

A.27. JZ

Jump if the user counter register (CU) is zero.

Opcode	Instruction name	Parameters	Size (bytes)
0C	JZ	Branch address	2
micro instructions: 0C:0 Increment_PC 0C:0 if (UC_IsZero) Branch 0C:2 0C:1 Branch 00:0 0C:1 Increment_PC 0C:2 Load_PC_Instruction2			

A.28. Jump

Do unconditional branching.

Opcode	Instruction name	Parameters	Size (bytes)
0B	Jump	Branch address	2
micro instructions: 0B:0 Increment_PC 0B:1 Load_PC_Instruction2			

A.29. Call

Push the program counter (i.e. PCRead register) to the stack, then branch.

Opcode	Instruction name	Parameters	Size (bytes)
02	Call	Branch address	2
micro instructions: 02:0 DEC_SP 02:0 Increment_PC 02:1 DEC_SP 02:1 Push_PC2 02:2 Load_PC_Instruction2 02:2 Push_PC1			

A.30. Return

Pop the program counter (i.e. PCRead register) from the stack. It does branch because the program counter will change.

Opcode	Instruction name	Parameters
1A	Return	no parameters
micro instructions: 1A:0 Pop1 1A:1 NOP 1A:2 Pop_PC2 1A:3 NOP		

A.31. NOP

Do nothing. Just waste processor cycles. It is useful for doing delay loop or delay between instructions.

Opcode	Instruction name	Parameters
15	NOP	no parameters
micro instructions: 15:0 NOP		

A.32. Stop

Stop running the program by looping to the same location.

Opcode	Instruction name	Parameters
20	Stop	no parameters
micro instructions: 20:0 Branch 20:0		

20:0 IsStopInstruction

A.33. ClearTestDataRegister

Clear the test data register (TD).

Opcode	Instruction name	Parameters
23	ClearTestDataRegister	no parameters
micro instructions: 23:0 ClearTD		

B. Instruction Builder Tutorial

This appendix presents a tutorial for designing instructions and generating the resulted processor HDL code.

1. Starting empty project

All signals and instruction data are stored in the file “TACPDatabase.bin”. By deleting this file we start a new empty project. Figure 7.1 shows a snapshot of instruction builder software after deleting the file. There is also another text file “DatapathBody.txt” that stores the data path code.

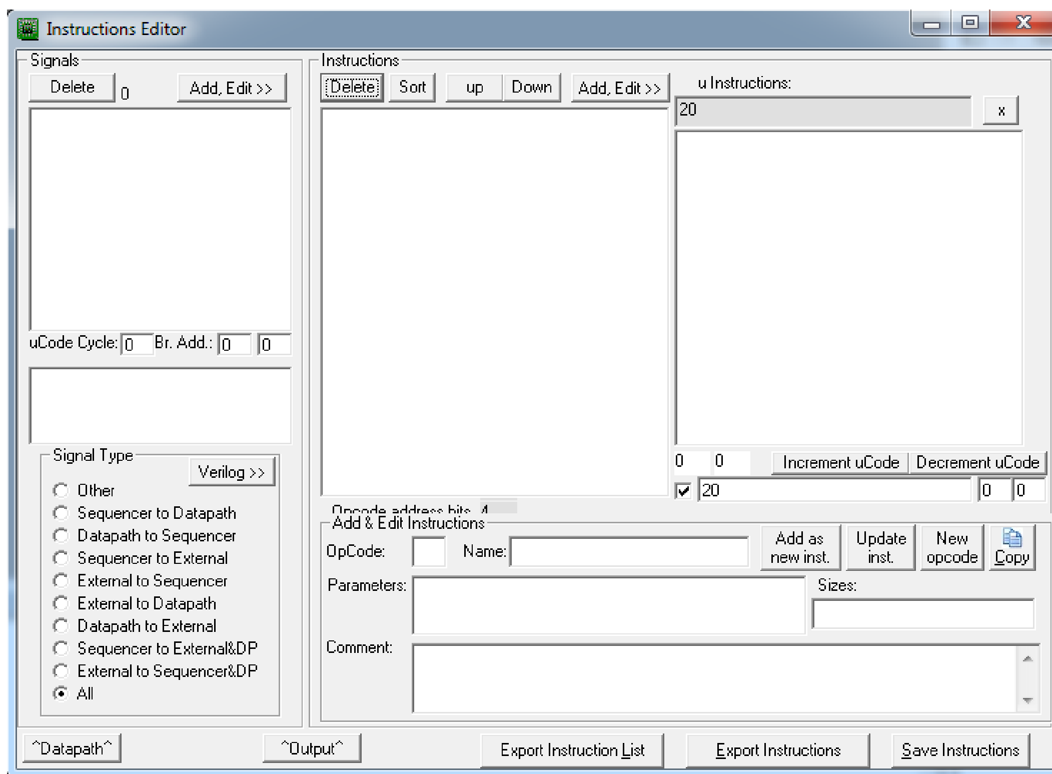


Figure 7.1 : Starting Instruction builder with an empty project.

2. Add signals

On the left there is signals list. To add signals, press the button “Add, Edit >>” then write the signal unique name and a description to be included as a comment in the Verilog file.

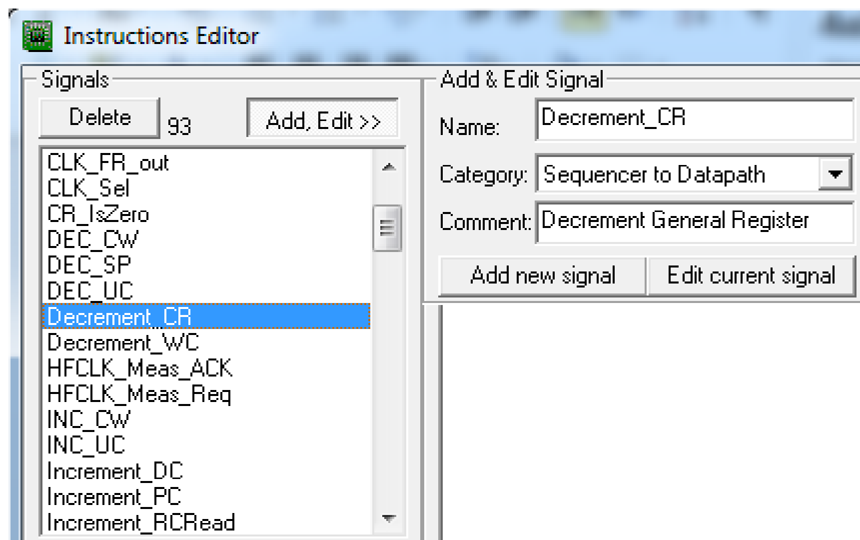


Figure 7.2 : Add & edit signals and determine the signal type.

It is also important to set the signal type from the type list mentioned earlier and showed in Figure 5.5.

3. Add instructions

Instruction is defined by its opcode and name. The user could also define parameters. Parameters should be written in parameter-text-box separated by comma and their sizes in sizes-text-box.

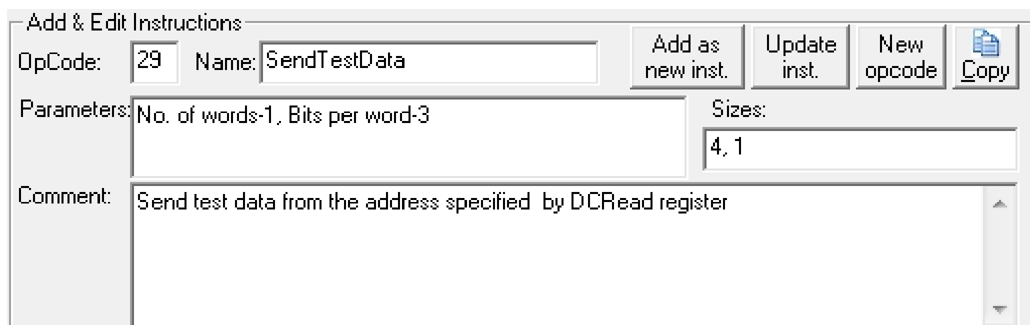


Figure 7.3 : Add & edit instruction and define its parameters sizes and names.

4. Add microinstructions

Each instruction could have any number of microinstructions. For the selected instruction, microinstructions can be added by double clicking any signal from the signal list.

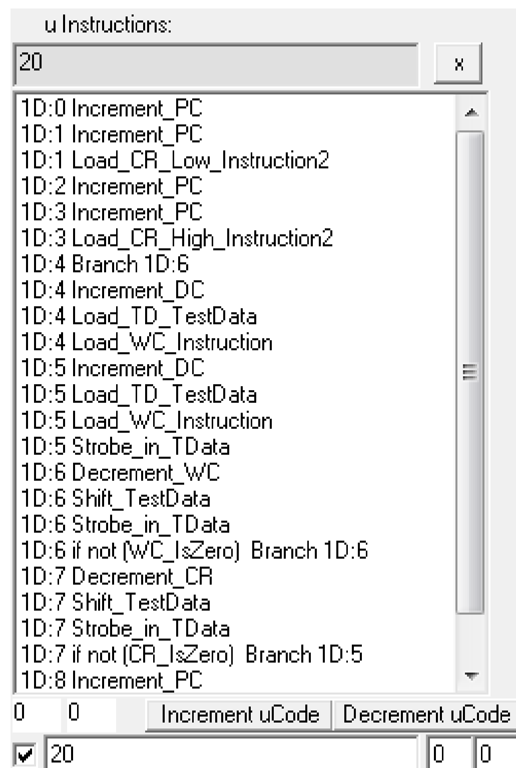


Figure 7.4 : Writing microinstruction and defining their cycles.

There are two types of microinstructions:

- Normal microinstructions: like the sequencer to data path or to external signals.
- Conditional microinstructions: any signal goes to the sequencer. These signals control the sequencer.

Microinstruction consists of clock number, signal name, and branch address if it is conditional. If some microinstructions have the same clock cycle, that means they will be executed in the same clock.

There are buttons for incrementing and decrementing clock cycle for the selected microinstruction(s).

5. Write data path code

The last step in designing the microcode is to write the data path code. Data path button exists in the right bottom corner of the instruction builder window.

The user can put any needed data path components and define their wires and registers. he should not write the data path interface. He also should not redefine the signals that are defined in the signal list.

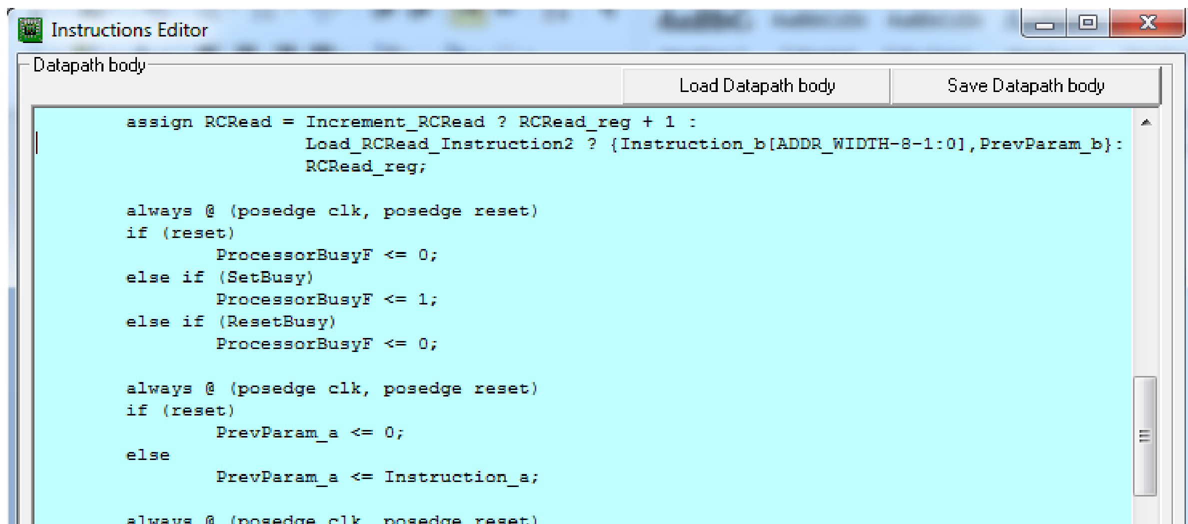


Figure 7.5 : Writing verilog code for data path component.

6. Generate microcode

The software has many options. It can generate data path module verilog code, sequencer module verilog code or the whole microcode verilog code that include sequencer code, data path code, and sequencer and data path instantiations.

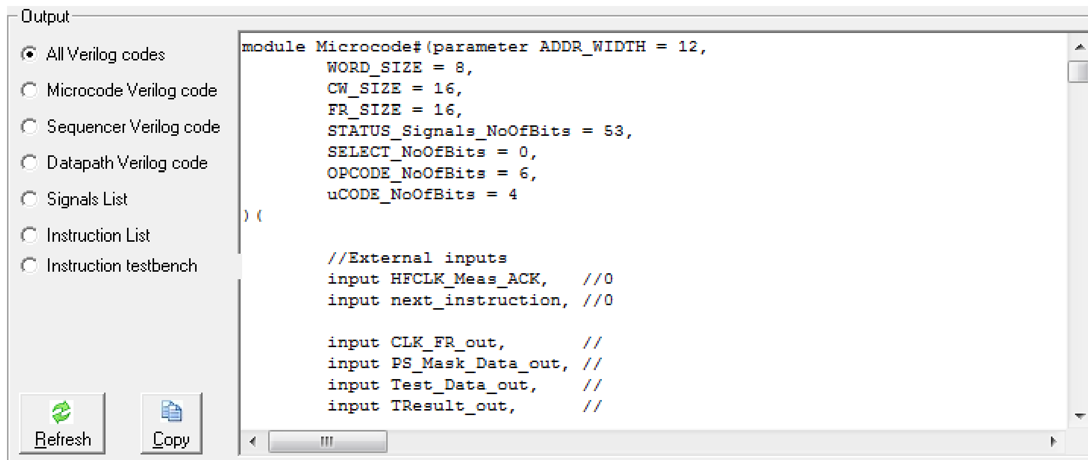


Figure 7.6 : Generating verilog code for microcode.

In addition, the software can report signals grouped as their types. It can report instructions with their microinstructions.

7. Export instructions

It is possible to have the instructions in a text file. The software reports instructions with their parameters and sizes. This text file can be read by software that uses the instructions to build programs.

8. Generate test benches

The software can translate the microinstructions into testbench format as shown in this table. This table is a translation of one instruction (i.e. SendTestData).

Instruction microinstruction	Testbench

SendTestData
1D:0 Increment_PC
1D:1 Increment_PC
1D:1
Load_CR_Low_Instruction2
1D:2 Increment_PC
1D:3 Increment_PC
1D:3
Load_CR_High_Instruction2
1D:4 Branch 1D:6
	//Testbench:
	//Code: 1D
	//Mnemonic: SendTestData
	//Parameters: No. of words-1=4, Bits per word-3=1
	//for a given CR=3 and WC=2
	#20;//1D:0 Increment_PC
	Increment_PC = 1'b1;
	
	#20;//1D:1 Increment_PC, Load_CR_Low_Instruction2
	Load_CR_Low_Instruction2 = 1'b1;

1D:4 Increment_DC
1D:4 Load_TD_TestData
1D:4 Load_WC_Instruction
1D:5 Increment_DC
1D:5 Load_TD_TestData
1D:5 Load_WC_Instruction
1D:5 Strobe_in_TData
1D:6 Decrement_WC
1D:6 Shift_TestData
1D:6 Strobe_in_TData
1D:6 if not (WC_IsZero) Branch 1D:6
1D:7 Decrement_CR
1D:7 Shift_TestData
1D:7 Strobe_in_TData
1D:7 if not (CR_IsZero) Branch 1D:5
1D:8 Increment_PC
 	
1D:8 Shift_TestData
1D:8 Strobe_in_TData
	#20;//1D:2 Increment_PC
	Load_CR_Low_Instruction2 = 1'b0;
	
	#20;//1D:3 Increment_PC, Load_CR_High_Instruction2
	Load_CR_High_Instruction2 = 1'b1;
	
	#20;//1D:4 Branch 1D:6, Increment_DC, Load_TD_TestData, Load_WC_Instruction
	Increment_PC = 1'b0;
	Load_CR_High_Instruction2 = 1'b0;
	Increment_DC = 1'b1;
	Load_TD_TestData = 1'b1;
	Load_WC_Instruction = 1'b1;
	
	#20;//1D:6 Decrement_WC, Shift_TestData, Strobe_in_TData, if not (WC_IsZero) Branch 1D:6
	Increment_DC = 1'b0;
	Load_TD_TestData = 1'b0;
	Load_WC_Instruction = 1'b0;
	Decrement_WC = 1'b1;
	Shift_TestData = 1'b1;
	#20;//1D:6 Decrement_WC, Shift_TestData, Strobe_in_TData, if not (WC_IsZero) Branch 1D:6
	Increment_DC = 1'b0;
	Load_TD_TestData = 1'b0;
	Load_WC_Instruction = 1'b0;
	Decrement_WC = 1'b1;
	Shift_TestData = 1'b1;
	#20;//1D:7 Decrement_CR, Shift_TestData, Strobe_in_TData, if not (CR_IsZero) Branch 1D:5
	Decrement_WC = 1'b0;
	Decrement_CR = 1'b1;
	#20;//1D:5 Increment_DC, Load_TD_TestData, Load_WC_Instruction, Strobe_in_TData
	Strobe_in_TData = 1'b1;
	#20;//1D:6 Decrement_WC, Shift_TestData, Strobe_in_TData, if not (WC_IsZero) Branch 1D:6
	Increment_DC = 1'b0;
	Load_TD_TestData = 1'b0;
	Load_WC_Instruction = 1'b0;
	Decrement_WC = 1'b1;
	Shift_TestData = 1'b1;
	#20;//1D:6 Decrement_WC, Shift_TestData, Strobe_in_TData, if not (WC_IsZero) Branch 1D:6

	Increment_DC = 1'b0;
	Load_TD_TestData = 1'b0;
	Load_WC_Instruction = 1'b0;
	Decrement_WC = 1'b1;
	Shift_TestData = 1'b1;
	
	#20;//1D:7 Decrement_CR, Shift_TestData, Strobe_in_TData, if not (CR_IsZero) Branch 1D:5
	Decrement_WC = 1'b0;
	Decrement_CR = 1'b1;
	
	#20;//1D:5 Increment_DC, Load_TD_TestData, Load_WC_Instruction, Strobe_in_TData
	Strobe_in_TData = 1'b1;
	#20;//1D:6 Decrement_WC, Shift_TestData, Strobe_in_TData, if not (WC_IsZero) Branch 1D:6
	Increment_DC = 1'b0;
	Load_TD_TestData = 1'b0;
	Load_WC_Instruction = 1'b0;
	Decrement_WC = 1'b1;
	Shift_TestData = 1'b1;
	
	#20;//1D:6 Decrement_WC, Shift_TestData, Strobe_in_TData, if not (WC_IsZero) Branch 1D:6
	Increment_DC = 1'b0;
	Load_TD_TestData = 1'b0;
	Load_WC_Instruction = 1'b0;
	Decrement_WC = 1'b1;
	Shift_TestData = 1'b1;
	#20;//1D:7 Decrement_CR, Shift_TestData, Strobe_in_TData, if not (CR_IsZero) Branch 1D:5
	Decrement_WC = 1'b0;
	Decrement_CR = 1'b1;
	
	#20;//1D:8 Increment_PC, Shift_TestData, Strobe_in_TData
	Decrement_CR = 1'b0;
	Increment_PC = 1'b1;
	#20;//1D:9
	Increment_PC = 1'b0;
	Shift_TestData = 1'b0;
	Strobe_in_TData = 1'b0;

C. User Interface Tutorials

The program has two parts: writing program tab and executing program tab.

Writing Programs tab

The writing program tab is shown in Figure below. To write a program there is an instruction list to select instruction. Below the instruction list there are some fields to set or edit the instruction parameter. Instruction description also viewed there. The instructions could be sorted alphabetically by pressing the sort button.

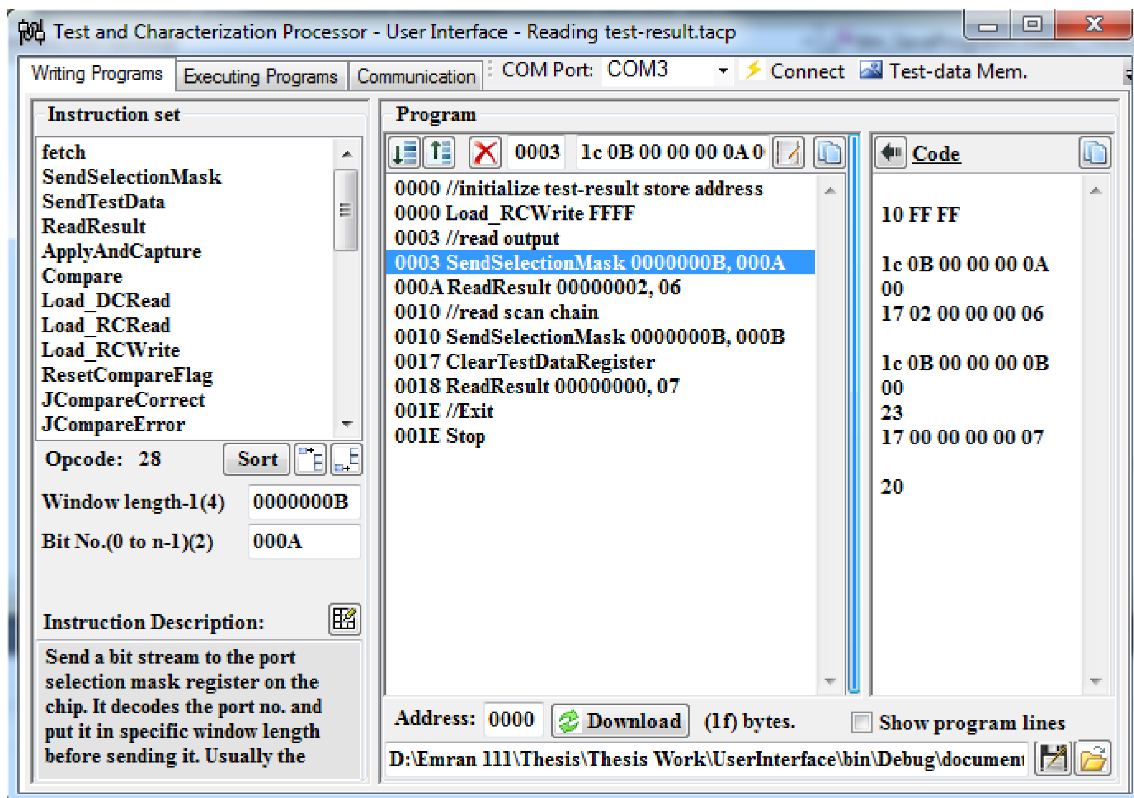


Figure 7.7 : Writing programs window.

The program is written by selecting instructions and adding them to the program list. In this list it is possible to edit instruction parameter, select instruction or instructions and move them up or down or delete instructions. Beside the program list there are the program code written in

hexadecimal values. The code can be edited directly and then transformed to instruction list rather than editing the program list. It is also possible to copy another program code and paste it there.

Comments can be added as separate lines. The show program lines checkbox can show or hide program lines. The program can be saved as a text file contains the hexadecimal code. The program comments are saved in a separate file with the same name and with additional "_cmt" extension.

Finally for this tab, the program can be downloaded to the instruction memory to a specific address. The connection had to be established.

Executing Programs tab

The Executing program tab is shown in Figure 7.8 below. In the beginning, the program is uploaded from the instruction memory. The uploaded program is listed on the left side of the window.

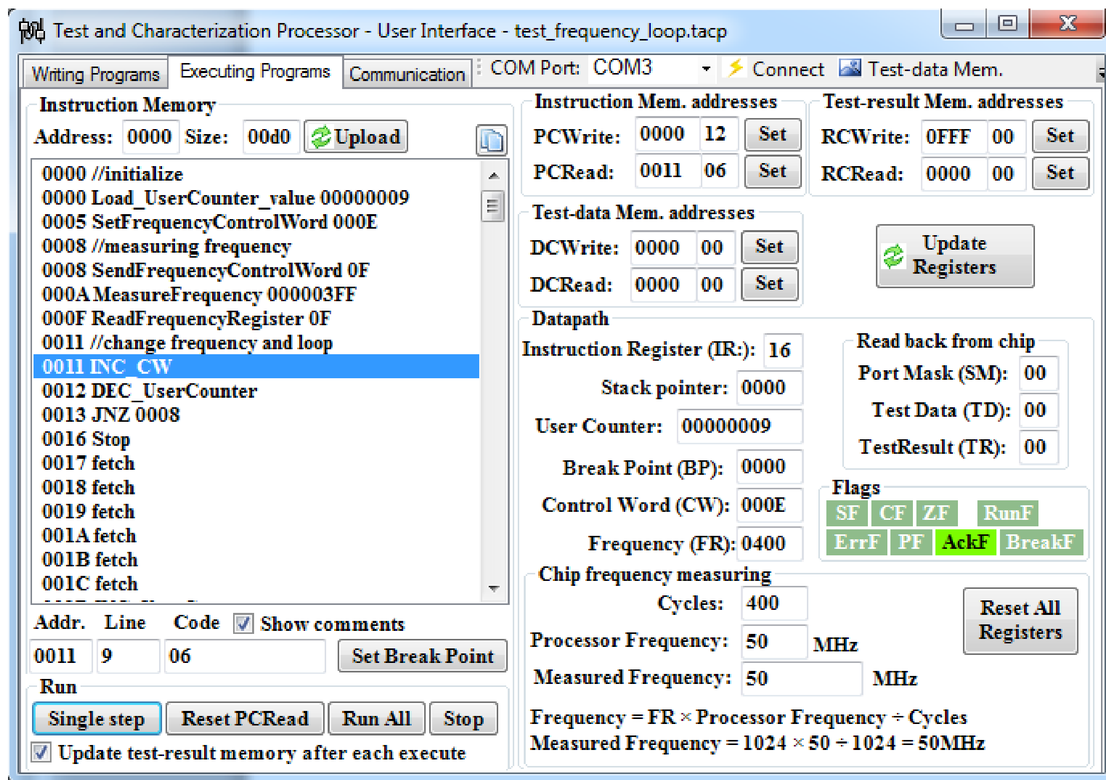


Figure 7.8 : Executing programs window.

The upload address and the number of bytes to upload have to be determined before uploading. Comments can be added from the current comment file to the uploaded program by checking the show-comments checkbox.

Below the uploaded program, there are execution control buttons. Set-break-point button copies the selected instruction address to BP register. Single-step button executes only one instruction. Run-all button executes all instruction. The execution stops when it reaches the stop instruction, BP register equals PCRead register or the user presses stop button. Reset PCRead button set the PCRead register to zero which means to reset the execution.

On the right side of this window, processor register contents are displayed. Memory addresses six address registers are displayed and assigned with a “set” buttons that enables the user to set a value to the register manually. The chip frequency is calculated according to FR register value.

The user can set the processor frequency and the number of cycles used by the MeasureFrequency instruction. Update-Registers button reload all register value from the processor. Reset-All-Registers button sends a reset signals to the processor and reload the register values.

Memory windows

In the main windows there are four buttons to launch different memory windows. The memory window has a memory table reflects the memory contents. It is shown in Figure 7.9 below and Figure 7.10 below. Each row in the table represents a vector (i.e. test vector, result vector, etc.). The first column in the table is the address column shows the memory address or the vector number. The middle columns represent the memory contents one byte for each cell. The last column is the vector column which is formed be combining the other columns together. Column headers represent how many bits should be taken from each byte to form the test vector. For example, the test vector header in Figure 7.9 below is 23 which is the summation of (8+7+3+5) in the other columns. The user should take care of these numbers of bits when writing programs to send those test vectors.

Finally, using the file menu, the memory contents can be saved to or read from files.

Test-data memory window

This window is used to download or upload test-data and expected results to the processor test-data memory. The user can specify a memory location and number of bytes to upload. The user also can move directly to a specific test vector or a specific memory location. The current write and read addresses of the test data memory is shown in the status bar and updated after each

execution. This window is also used to show the expected result. A snapshot of test-data memory window is shown in Figure 7.9 below.

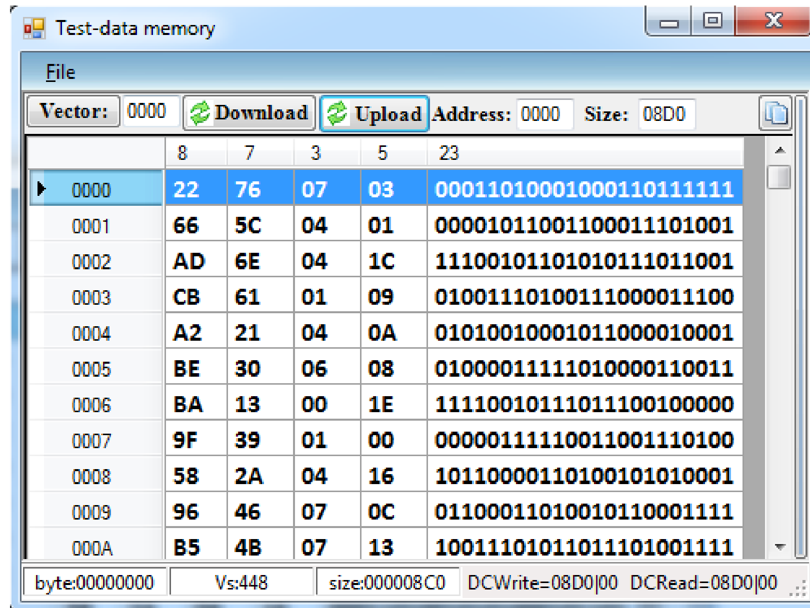


Figure 7.9 : Test-data memory window.

Test-result memory window

This window is used to upload test-result and the comparison result stored in the test-data memory. A snapshot of this window is shown Figure 7.10 below. The memory contents can be compared

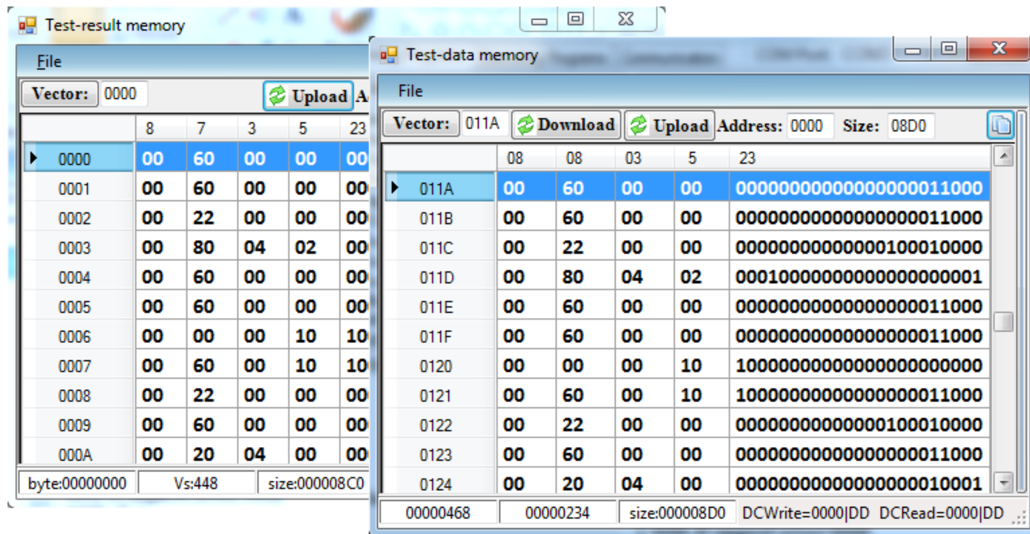


Figure 7.10 : Comparing test-results with expected results.

Compare instruction can compare between test-data and test-result memory and store the comparison result in test-data memory. Figure 7.11 below shows the comparison data generated by the compare instruction. The memory is sorted by the third column which brings up five errors rows. This indicates that vectors 234, 67, 189, 146 and 220 (i.e. EA, 43, BD, 92 and DC) resulted in errors.

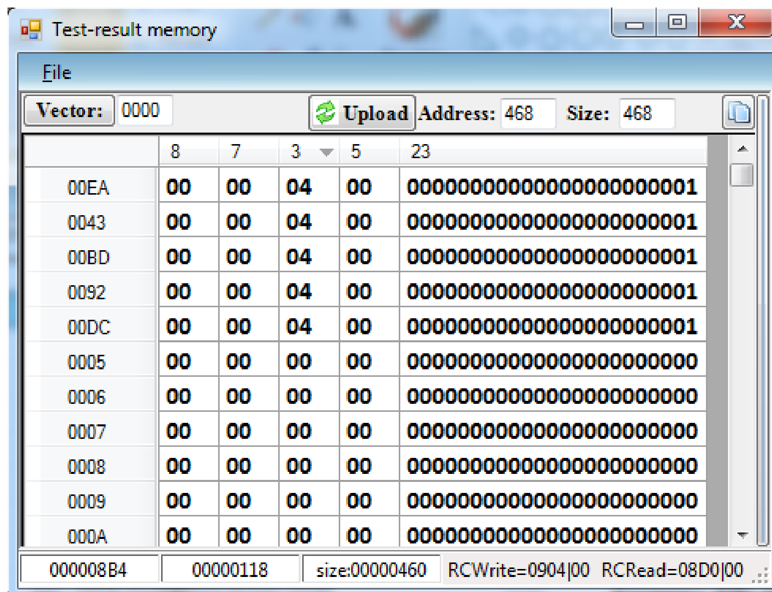


Figure 7.11 : Comparing results sorted by the third column to rise up vector caused error.

Importing test vectors to memory window

Usually, test-vectors and expected-results are presented in a binary format in a text file similar to that in Figure 7.12 below. These vectors are needed to be converted to bytes to be downloaded to the processor memory. Import-test-vectors window can read those files and extract test vectors and expected results.

Figure 7.12 : Import test-vector dialog window and test-vectors file snapshot.

The import window gives flexibility to the user. The user can specify what column in the text file has the test-vectors, how many scan bits per vector and whether those bits are on the right or the left of the test vector. The user also can specify whether the test vectors or scan chains will be read from right to left or left to right.

For example, let's take the first test vector 00011010001000110111111. In the import window, scan bits equals 5 and the 'scan bits before data bits' is checked. Hence,

Scan bits: 00011.

Data bits: 010001000110111111.

'Data bits left to right' is checked. Hence, data bits: 111111011000100010.

18-bit data vector will be divided to three bytes (3-7-8) as follows: 111_1110110_00100010

At the end we came up with these four bytes: 22 76 07 03 and the number of bits per byte are 8-7-3-5 which is shown in column headers in Figure 7.9 above.

REFERENCES

[1] M. Elrabaa, "Method for Digital Integrated Circuits Testing and Characterization". Saudi arabia Patent US Patent application number 13/471346, 14 May 2012.

[2] "wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Semiconductor_intellectual_property_core. [Accessed 2012].

[3] Z. Nozica, "Semiconductor intellectual property and system-on-chip for communications the future for small companies," in *Telecommunications, 2003. ConTEL 2003. Proceedings of the 7th International Conference on*, 2003.

[4] K. S. Yeo, K. T. Ng, Z. H. Kong and T. B. Y. Dang, "Importance of Intellectual Property Rights for Integrated Circuits," in *Intellectual Property for Integrated Circuits*, J. Ross Publishing, 2010, pp. 19-33.

[5] R. Saleh, SteveWilton, S. Mirabbasi, AlanHu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu and A. Ivanov, "System-on-Chip: Reuse and Integration," in *Proceedings of the IEEE*, 2006.

[6] M. Bordegoni and C. Rizzi, Innovation in Product Design From CAD to Virtual Prototyping, London: Springer, 2011.

[7] G. Jervan, "High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems," Linköpings university, Linköping, 2002.

[8] M. L. Bushnell and V. D. Agrawal, "Chapter 15: BUILT-IN SELF-TEST," in *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*, Norwell, Massachusetts USA, Kluwer Acadimic Publishers, 2004, pp. 489-548.

[9] D. K. Pradhan and M. Chatterjee, "GLFSR—A New Test Pattern Generator for Built-in-Self-Test," *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS,* vol. 18, no. 2, pp. 238-247, 1999.

[10] M. Kabir and L. Ali, "Design of GLFSR based test processor chip," *Research and Development (SCOReD), 2009 IEEE Student Conference on ,* pp. 234-237, 2009.

[11] M. L. Ali, Z. M. Darus and M. A. M. Ali, "Test Processor ASIC Design," *Semiconductor Electronics, 1996. ICSE '96. Proceedings., 1996 IEEE International Conference on ,* pp. 261-265, 1996.

[12] E. Kalligeros, X. Kavousianos, D. Bakalis and D. Nikolos, "An Efficient Seeds Selection Method for LFSR-based Test-per-clock BIST," in *Proceedings of the International Symposium on Quality Electronic Design (ISQED'02)*, San Jose, CA, USA, 2002.

[13] M. Ali, S. Islam and M. Ali, "Test processor chip design with complete simulation result including reseeding technique," *Semiconductor Electronics, 2002. Proceedings. ICSE 2002. IEEE International Conference on ,* pp. 218-221, 2002.

[14] W. Ying and W. Hong, "The testing of multiple RAM Cores in Soc system," *Solid-State and Integrated Circuit Technology, 2006. ICSICT '06. 8th International Conference on ,* pp. 2148 - 2150, 2006.

[15] K. Batcher and C. Papachristou, "Instruction Randomization Self Test For Processor Cores," *VLSI Test Symposium, 1999. Proceedings. 17th IEEE ,* pp. 34-40, 1999.

[16] C. Galke, M. Pflanz and H. T. Vierhaus, "A Test Processor Concept for Systems-on-a-Chip," *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on,* pp. 210 - 212, 2002.

[17] M. Benabdenbi, A. Greiner, F. Pecheux, E. Viaud and M. Tuna, "STEPS: experimenting a new software-based strategy for testing SoCs containing P1500-compliant IP cores," *IEEE,* vol. 1, pp. 712 - 713, 2004.

[18] R. Frost, D. Rudolph, C. Galke, R. Kothe and H. T. Vierhaus, "A Configurable Modular Test Processor and Scan Controller Architecture," *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International,* 2007.

[19] L. Mostardini, L. Bacciarelli, L. Fanucci, L. Bertini, M. Tonarelli, A. Giambastiani and M. D. Marinis, "FPGA-based Low-cost System for Automatic Tests on Digital Circuits," *Electronics, Circuits and Systems, 2007. ICECS 2007. 14th IEEE International Conference on ,* pp. 911-914, 2007.

[20] S. D. Carlo, P. Prinetto, A. Scionti, J. Figueras, S. Manich and R. Rodriguez-Monta˜n´es, "A Low-Cost FPGA-Based Test and Diagnosis Architecture for SRAMs," *Advances in System*

Testing and Validation Lifecycle, 2009. VALID '09. First International Conference on , pp. 141-146, 2009.

[21] D. Markovic, C. Chang, B. Richards, H. So, B. Nikolic and R. Brodersen, "ASIC Design and Verification in an FPGA Environment," *Custom Integrated Circuits Conference, 2007. CICC '07. IEEE ,* pp. 737 - 740 , 2007.

[22] L. Ciganda, F. Abate, P. Bernardi, M. Bruno and M. Reorda, "An enhanced FPGA-based low-cost tester platform exploiting effective test data compression for SoCs," *Design and Diagnostics of Electronic Circuits & Systems, 2009. DDECS '09. 12th International Symposium on ,* pp. 258-263, 2009.

[23] L. L. d. Oliveira, J. B. d. S. Martins and A. L. Aita, "A low-price platform to test digital integrated circuits using FPGA," *Circuits and Systems, 2005. 48th Midwest Symposium on,* vol. 2, pp. 1127-1130, 2005.

[24] S. Bahl and B. Singh, "On-Chip and At-Speed Tester for Testing and Characterization of Different Types of Memories". United States Patent US 7,353,442 B2, 1 April 2008.

[25] S. Vassiliadis, S. Wong and S. Cotofana, "MICROCODE PROCESSING: POSITIONING AND DIRECTIONS," *IEEE Micro,* vol. 23, no. 4, pp. 21-31, 2003.

[26] Xilinx, "Spartan-3 Generation FPGA User Guide - Extended Spartan-3A, Spartan-3E, and Spartan-3 FPGA Families," 13 June 2011. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug331.pdf.

[27] J. Shen and J. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," *Test Conference, 1998. Proceedings., International ,* pp. 990-999 , 1998.

[28] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on ,* pp. 369-380, 2001.

[29] A. Krstic, W.-C. Lai, K.-T. Cheng, L. Chen and S. Dey, "Embedded software-based self-test for programmable core-based designs," *Design & Test of Computers, IEEE ,* vol. 19, pp. 18 - 27 , 2002.

[30] D. Keezer and Q. Zhou, "Test support processors for enhanced testability of high performance circuits," *Test Conference, 1999. Proceedings. International ,* pp. 801 - 809, 1999.

[31] J. Davis and D. Keezer, "Multi-Purpose Digital Test Core Utilizing Programmable Logic," *Test Conference, 2002. Proceedings. International,* pp. 438-445, 2002.

[32] D. Keezer, C. Gray, A. Majid and N. Taher, "Low-Cost Multi-Gigahertz Test Systems Using CMOS FPGAs and PECL," *Design, Automation and Test in Europe, 2005. Proceedings ,* vol. 1, pp. 152 - 157, 2005.

[33] "http://www.teradyne.com/J750/".

[34] V. Vorisek, T. Koch and H. Fischer, "At-speed testing of SOC ICs," *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings ,* vol. 3, pp. 120 - 125, 2004.

[35] S. Zeidler, C. Wolf, M. Krsti´c, F. Vater and R. Kraemer, "Design of a Test Processor for Asynchronous Chip Test," *Test Symposium (ATS), 2011 20th Asian ,* pp. 244-250, 2011.

[36] M. Abramovici, M. A. Breuer and A. D. Friedman, "Chapter 2 Modeling," in *Digital System Testing and Testable Design*, Piscataway, IEEE Press Marketing, 1990, p. 652.

[37] M. Abramovici, M. A. Breuer and A. D. Friedman, Digital System Testing and Testable Design, Piscataway: IEEE Pres Marketing, 1990.

[38] M. L. Bushnell and V. D. Agrawal, Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits, KLUWER ACADEMIC PUBLISHERS, 2000.

VITA

Name : Amran Abdulrahman Abdulwali Al-aghbari

Nationality : Yemeni

Date of Birth : 29/12/1978

Email : emranemran@hotmail.com

Permanent Address : Taiz University - Taiz - Yemen

Present Address : King Fahd University for Petroleum and Minerals –Dhahran – Saudi Arabia.

Tel. : +966 508052122

: +967 715081959

Academic Background : Bachelor of Computer Science - Sana'a University – Yemen - June 2004.