NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

TRIPLE MODULAR REDUNDANCY (TMR) IN A CONFIGURABLE FAULT-TOLERANT PROCESSOR (CFTP) FOR SPACE APPLICATIONS

by

Rong Yuan

December 2003

Thesis Co-Advisors: Herschel H. Loomis, Jr.
Alan A. Ross

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Triple Modular Redundancy (TMR) in a Configurable Fault–Tolerant Processor (CFTP) for Space Applications

5. FUNDING NUMBERS

6. AUTHOR(S) Yuan, Rong

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Without the protection of atmosphere, space systems have to mitigate radiation effects. Several different technologies are used to deal with different radiation effects in order to keep the space device work properly. One of the radiation effects called Single Event Upset (SEU) can change the state of a component or data on the bus. A single error is possible to cause a system failure if it is not corrected.

Besides error correction, a space system also needs the flexibility to be modified or upgraded easily. Consequently, the idea of having a TMR design instantiated in an FPGA to construct a Configurable Fault-Tolerant Processor (CFTP) developed. The TMR, which runs one program in three identical soft-core processors with voters, is a scheme used to mitigate an SEU. The full design of TMR running in an FPGA functions as a System-On-a-Chip (SOC). Both soft-core processor and FPGA offer the CFTP a great flexibility to be reconfigured.

A complete TMR design includes some fundamental components besides processors and voters such as the *Reconiler*, *Interrupt,* and *Error Syndrome Storage Device (ESSD)*. These components have their unique function in the TMR design. They are created and simulated. Factors that affect test bench-settings like processor pipelining are important to always keep in mind. A component is designed to implement proper functions first. Then it is revised to work with the processor and memory. The full design for the TMR in this thesis proves its ability to detect and correct an SEU. The follow-on research suggested is to improve the efficiency and performance of this design.

14. SUBJECT TERMS Single Event Upset, SEU, Configurable Fault-Tolerant Processor, CFTP, TMR, FPGA, System-On-a-Chip, SOC, Reconciler, Interrupt and Error Syndrome Storage Device, ESSD

15. NUMBER OF PAGES 285

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

TRIPLE MODULAR REDUNDANCY (TMR) IN A CONFIGURABLE FAULT TOLERANT PROCESSOR (CFTP) FOR SPACE APPLICATIONS

Rong Yuan
1st Lieutenant, Taiwan Air Force
B.S., Chinese Air Force Academy, 1998

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2003

Author: Rong Yuan

Approved by: Herschel H. Loomis, Jr.
Thesis Co-Advisor

Alan A. Ross
Thesis Co-Advisor

John P. Powers
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Without the protection of atmosphere, space systems have to mitigate radiation effects. Several different technologies are used to deal with different radiation effects in order to keep the space device work properly. One of the radiation effects called Single Event Upset (SEU) can change the state of a component or data on the bus. A single error is possible to cause a system failure if it is not corrected.

Besides error correction, a space system also needs the flexibility to be modified or upgraded easily. Consequently, the idea of having a TMR design instantiated in an FPGA to construct a Configurable Fault-Tolerant Processor (CFTP) developed. The TMR, which runs one program in three identical soft-core processors with voters, is a scheme used to mitigate an SEU. The full design of TMR running in an FPGA functions as a System-On-a-Chip (SOC). Both soft-core processor and FPGA offer the CFTP a great flexibility to be reconfigured.

A complete TMR design includes some fundamental components besides processors and voters such as the *Reconiler*, *Interrupt,* and *Error Syndrome Storage Device (ESSD)*. These components have their unique function in the TMR design. They are created and simulated. Factors that affect test bench-settings like processor pipelining are important to always keep in mind. A component is designed to implement proper functions first. Then it is revised to work with the processor and memory. The full design for the TMR in this thesis proves its ability to detect and correct an SEU. The follow-on research suggested is to improve the efficiency and performance of this design.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank all the professors, technicians, and students of the Naval Postgraduate School who made this research possible. Many of them may not realize the magnitude of their efforts, but the author does.

Special thanks are owed to these individuals for their assistance:

To Professors Loomis and Ross for your support and patience with me throughout this research. Your knowledge and experience always inspires my invention.

To Professor Butler for providing good foundations in microprocessor systems.

To Professor Durren for the assistance in understanding VHDL code.

To Dr. Kenneth Clark for your assistance in understanding the function of the KDLX and always answering my questions in no time.

To Captain Charles Hulme for your friendship, for helping me get through many courses, and for standing my horrible English so long.

To David Rigmaiden for helping me set up the Xilinx software.

To Major Dean Ebert and Lieutenant Steven Johnson for your efforts on your thesis works which are great reference for any further research.

And most importantly, I wish to thank my family. Your support and encouragement let me study carefree overseas.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Space systems suffer radiation effects in space. These radiation effects occur randomly and are hard to predict. The combination of effects can destroy a system or make it functionless. Therefore, different methods are presented to protect space devices such as radiation hardened or fault tolerant systems. Space systems are usually tested and simulated several times before launching in order to minimize the probability of losing control of it after launch.

The Single Event Upset (SEU) is a radiation effect which causes a bit flipping in a device. This effect is not strong enough to destory a system but may cause a series of errors that finally make the system unusable. This error should be corrected in time and Triple Modular Redundancy (TMR) is one of the schemes to mitigate this problem.

The TMR design selected for the CFTP is to instantiate three soft-core processors with some other components into a fault tolerant Field Programmable Gate Array (FPGA). The FPGA is easily reconfigured and the soft-core processor has great flexibility to be programmed or modified. Those features give a TMR design the ability to be maintained and upgraded. The processor chosen for TMR design is a 16-bit Reduced Instruction Set Computer (RISC) processor named KDLX. It is a 5-stage pipelined processor with Harvard architecture. The pipeline affects the settings of a test bench and the influence is discussed in this thesis. A full simulation for all instructions is introduced to help understand functions of different operation codes.

To stop an error being propagated, the TMR has to correct the error once it is detected. Three processors in TMR should always execute the same instruction and all actions should be identical. Any inconsistency found among these three processors will be considered as an error. Then the TMR needs to have a function to stall the current operation and correct errors in processors. For error detection and correction, the following four major components are designed: majority bit voter, *Reconciler*, *Interrupt,* and *Error Syndrome Storage Device (ESSD)*.

Voters are connected at output pins or buses of processors. Therefore all output signals are voted. The majority bit voter takes two out of three identical signals as the output signal and reports the occurrence of an error if one of the three is different. The voter is able to correct an error immediately and indicate where the error is. Construction of three processors with voters called the TMR Assembly.

Due to different architectures between the processor and memory, a *Reconciler* is responsible for coordinating the difference between these two architectures. The solution is to run memory twice as fast as the processor and let the *Reconciler* route data of memory. The memory acts as an instruction memory at the first half of processor clock cycle and acts as a data memory at the other half cycle. Thus, the processor thinks it is connected with two different memories. The *Reconciler* in TMR for this thesis is purely a reconciler and does nothing directly related with error detection or correction. This purity makes it independent of other components.

When an error is detected by voters, the *Interrupt* starts the Interrupt Service Routine (ISR). In order to store and read properly, this component has to run as fast as the *Reconciler*. The *Interrupt* replaces the current instruction on the bus with a TRAP instruction when an error occurs. This TRAP instruction will be fetched by all processors and executed. The ISR is a special program designed to correct inconsistency of contents in registers between three processors. At the end of ISR, the *Interrupt* injects a Jump instruction into instruction bus and leads processors back to the normal operation.

The *ESSD* latches some specific data from the buses when an error occurs. These specific data are called the error syndrome, which is unique for one specific error. Error syndromes are very useful for health checking or error debugging to a system. In order to latch data at the correct timing, the *ESSD* has to run as fast as the *Reconciler* (or *Interrupt*). The *ESSD* does not pass its data to the *Reconciler* when storing. Instead, it takes over the whole memory and saves error syndromes while the processors are deliberately stalled.

The full design consolidates all components to construct a complete TMR design. The design was simulated and its function was proved in this thesis. This premiere design gives a big picture of how errors are detected and corrected. Furthermore, interaction between different components is one of the important concepts to learn. The full design has four different clocks. The *Reconciler*, *Interrupt* and *ESSD* are using the same clock speed since none of them needs the signal from another. The other three clocks are KDLX clock, memory clock and one special clock for the latch.

For further research, extra circuits or components are needed to improve the ability of error correction on different components. Considering an error generated in the *Reconciler*, the error may never be found and data stored to memory is always wrong. Reinforcing reliability of some components is something that needs to be considered. The current design may be modified to meet the requirements of advanced functions. Finally, searching for a better processor to enhance the performance is required as well. Commercial processors usually come with a software package and have better customer support. OpenCores that people share to the public are free but a user needs to have backgrounds of coding in order to realize the core.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

An electronic device in space environment suffers an extreme challenge to its reliability due to the lack of atmosphere and huge temperature variation. Without protection of atmosphere, a space system is exposed in a very unique circumstance which contains cosmic rays (85% protons, 14% alpha particles and 1% heavy Nuclie), solar events (X-rays, heavy ions and protons) and trapped radiation (electrons and protons trapped in magnetic field of earth, called Van Allen Belt). Thus, radiation effects on a space electronic system become one of the most important issues that need to be solved. Those effects include Total Dose Effects and Single Event Effects.

A number of methods have been presented to mitigate radiation effects. Using soft-core Triple Modular Redundancy (TMR) on a Field Programmable Gate Array (FPGA) provides a practical solution to Single Event Effects which is low cost and offers flexibility to be reconfigured and easily developed. The Configurable Fault-Tolerant Processor (CFTP) is a system based on this concept utilizing Commercial-Off-the-Shelf (COTS) technology and features of TMR soft-core microprocessors on FPGAs as a System-On-a-Chip (SOC).

A. RADIATION EFFECTS

Radiation effects on a space system vary depending on different altitude, location and solar events. For example, the inner Van Allen Belt, from 650 km to 6300 km above Earth's surface, is composed mostly of protons about 10 to 15 MeV (1 MeV = 10^6 eV, 1 electronvolt ≈ 1.6×10^{-19} J). As a satellite travels in Low-Earth Orbit (LEO), from 160 to 6000 km, it will have many chances to be affected by protons. The scheme used to solve radiation problems on this satellite must be different from the one that travels in geostationary orbit, whose altitude is 35,780 km. Since a satellite in geostationary orbit has almost no protection by Earth, it needs to be more radiation-hardened (RADHARD) or radiation-tolerant. Major effects caused by radiation are Total Dose Effects and Single Event Effects (SEE) including Single Event Phenomenon (SEP), Single Event Upset (SEU), Single Event Latchup (SEL) and Single Event Burnout (SEB) [1].

1. Total Dose Effects

Total Dose Effects refer to total radioactive particles that a device accumulates over its lifetime. This accumulation degrades the performance until the device becomes totally useless. The general solution to mitigate these effects so far is using radiation-hardening or shielding techniques, but such methods can only extend the end of life of the chip, not totally eliminate this problem.

2. Single Event Phenomenon (SEP)

Single Event Phenomenon is the situation where a transistor resets to its original state due to the particle passing through. This causes unpredictable results and may or may not affect operation of a system.

3. Single Event Upset (SEU)

Single Event Upset is a logical bit changing because of the radiation. A bit flipping may cause a chain reaction and consequently result in an unrecoverable error of a system. TMR is a mitigation scheme using three identical processors to run a same instruction set and voting all results to detect and correct such an error.

4. Single Event Latchup (SEL) and Single Event Burnout (SEB)

Single Event Latchup occurs when a parasitic transistor is formed by a spurious current spike like heavy cosmic ray [2]. This puts a circuit into a high-operating-current mode that has to be cleared by power off-on reset. Hard errors can drag the bus voltage down or even burn out the circuit. This is called Single Event Burnout.

Some techniques used to mitigate radiation effects are shown in Table 1.

Radiation Effects	Mitigation Techniques
Total Dose	Radiation-Hardening Silicon-On-Sapphire Silicon-On-Insulator Thin-Gate-Oxide Shielding
Single Event Latchup (SEL)	Radiation Hardening Guard Rings
Single Event Upset (SEU)	Quadded Logic Software Fault Tolerance Tripple Modular Redundancy

Table 1. Radiation Effects and Mitigation (From Ref. [1].)

B. FIELD PROGRAMMABLE GATE ARRAY (FPGA)

Sequential programmable devices are composed of gates and flip-flops and are able to perform a variety of functions. Three major types of sequential programmable devices are the Sequential (or simple) Programmable Logic Device (SPLD), the Complex Programmable Logic Device (CPLD) and the Field Programmable Gate Array (FPGA). A SPLD which integrates the AND-OR array and flip-flops is the smallest and the cheapest form of programmable logic. A CPLD is similar to a SPLD except that it is a collection of individual PLDs. Interconnections between PLDs are programmable as well. A typical CPLD is equal to 2 to 64 SPLDs. An FPGA consists of logic cells surrounded by a ring of programmable I/O blocks. Each cell is able to implement a logic function which is done by programming and all interconnections between cells are also programmable.

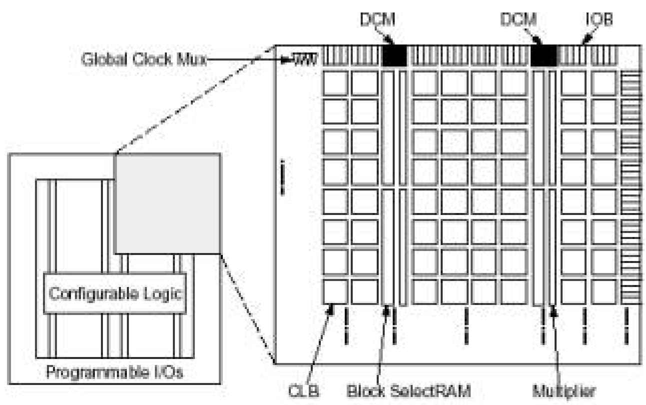


Figure 1. Composition of FPGA (From Ref. [3].)

Unlike the FPGA, PLDs need to be physically removed from a system and reprogrammed by specific methods. This disadvantage makes a space system made of these devices almost impossible to be modified or upgraded. Programmed circuits can be easily instantiated on a FPGA without any specific requirements. This feature reduces time-to-market of a product as well. Comparing with other device, FPGAs are less power consuming, less expensive, have large-scale advantages of programmable logic and high flexibility [4].

The FPGA selected for CFTP is the Virtex XCV800, a member in Virtex FPGA family of Xilinx[1]. Table 2 shows the specification of some Virtex family members. A

[1] Xilinx is a registered trademark of Xilinx Corporation.

CLB is a Configuration Logic Block which can be configured to represent any 4-input switching function to define a design. CLBs are also connected to each other by programming as part of the design process. A design can be parsed to multiple CLBs for full implementation if it is too large to fit into a single CLB [5].

Device	System Gates	CLB Array	Logic Cells	Maximum Available I/O	Block RAM Bits	Maximum SelectRAM+™ Bits
XCV50	57,906	16x24	1,728	180	32,768	24,576
XCV100	108,904	20x30	2,700	180	40,960	38,400
XCV150	164,674	24x36	3,888	260	49,152	55,296
XCV200	236,666	28x42	5,292	284	57,344	75,264
XCV300	322,970	32x48	6,912	316	65,536	98,304
XCV400	468,252	40x60	10,800	404	81,920	153,600
XCV600	661,111	48x72	15,552	512	98,304	221,184
XCV800	888,439	56x84	21,168	512	114,688	301,056
XCV1000	1,124,022	64x96	27,648	512	131,072	393,216

Table 2. Virtex FPGA family members (From Ref. [6].)

One of the reasons for choosing this FPGA was because its pin configuration is a flat-pack. This type of interface is spaceflight certified and has been used in space for years. Some of the newest and largest FPGAs nowadays are using ball grid array (BGA) connections which are not only difficult to be attached to a printed circuit board, but also not qualified for space applications [5].

C. SOFT-CORE PROCESSORS

A soft-core processor is a set of source codes expressed in hardware description language (HDL) which express the behavior of a real processor. It is a synthesizable HDL design and has no explicit hardware realization. This type provides great flexibility but has limitation of performance and predictability. A hard-core processor, on the other hand, provides high performance but is not flexible.

Since a soft-core processor can be easily instantiated on a FPGA, a designer has a wide range of selections and combinations. A soft core can be optimized for different FPGA sizes and characteristics to improve performance, giving the most cost-efficient solution for target applications. A hard core which has specific function blocks needs to work with special FPGA device. The need for these specific FPGAs is limited; therefore they do not have the large-scale manufacturing benefits which forces vendors to support few FPGA packages. Another disadvantage of using a hard core is if a problem is found in one version, all specific FPGAs supporting that version have to be revised. Hard cores are good for big and commonly used functions like a RAM [4].

The soft-core processor chosen for this iteration of the CFTP is a 16-bit Reduced Instruction Set (RISC) KDLX processor. The DLX processor is coded in HDL and described in Hennessy and Patterson's *Computer Architecture: A Quantitative Approach* [7]. The KDLX processor is a revision of DLX processor by Dr. Kenneth Clark that was used on complex digital systems to predict SEU tolerance as described in his dissertation [8]. Therefore, one of the reasons to use this processor is that it had been designed and tested.

D. TRIPLE MODULAR REDUNDANCY (TMR)

Once a system is launched to space, it is hard and expensive to maintain it. In order to correct errors caused by radiation, different ways have been presented and actually used in space. Using RADHARD devices or fault-tolerant designs are the most common ways. TMR is one of the solutions to make a circuit be able to tolerate occurrence of an error and correct it. This is done by software so it is simple and low-cost. Taking advantage of the FPGA, the TMR instantiated inside becomes easily modified and upgraded in the future.

Basically, a TMR system is composed of three identical devices and voting logic as shown in Figure 2. The voting logic is a majority voter which takes the majority of the inputs to be the output value. Since Devices B and C are replication of Device A and they all accept the same input value, the outputs of A, B and C should be consistent in theory. Due to radiation effects in space, one of these three devices may have an error inside and generate a different output. This inconsistency will be caught and corrected by voting logic. Thus, the voted output is always a correct value under the assumption of a single error.

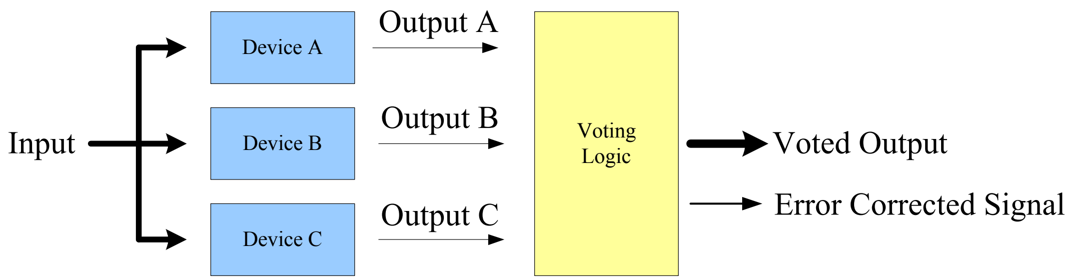


Figure 2. Basic TMR Concept (After Ref. [1].)

When the TMR concept is applied to a microprocessor, it is illustrated in Figure 3. All output signals of the CPU are voted; therefore no error should exist at outputs of voters. Any error that occurs represents that one of the CPUs has an error inside. If that error is not corrected by some way, it may result in more errors and finally become unrecoverable. Thus, the Error Encoder in Figure 3 is a device that will analyze error signals offered by voters and find out which CPU generates the error. Once the faulty CPU is identified, some extra circuits will interrupt all three processors and correct that error. When a simple circuit acting as a system is instantiated on a chip (e.g., FPGA), it is called a system on a chip (SOC). Recall that a soft core is not efficient for complex functions; therefore the memory block in Figure 3 is an external chip.

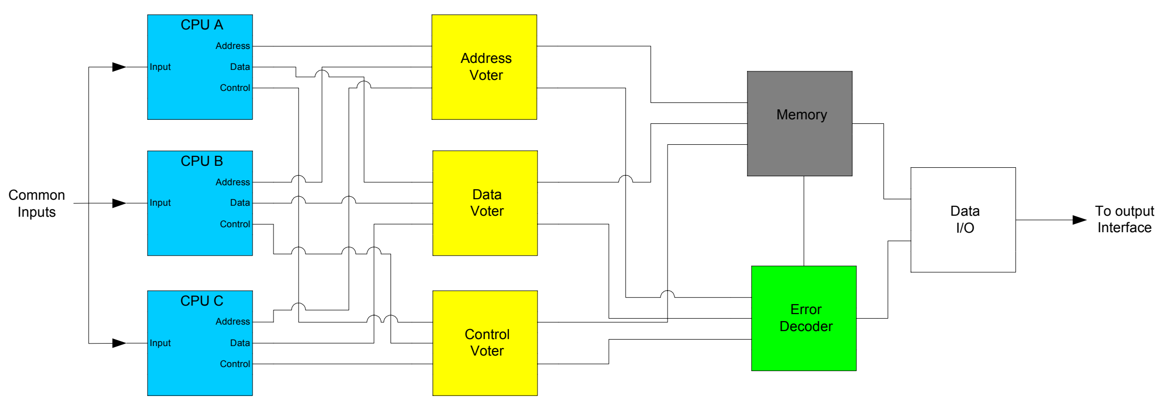


Figure 3. Microprocessor TMR Concept

The CFTP implements these basic ideas. The circuits to do interruption and correct an error are quite complicated. All concepts for constructing a complete TMR design will be explained in the rest of chapters.

E. ORGANIZATION

Chapter II reviews previous theses and gives other information related to the CFTP. Chapter III describes the testing environment and introduces the software used in the thesis. Chapter IV discusses the function and features of the KDLX. Simulations of all instructions for the KDLX are shown in this chapter. Chapter V goes over the design of voter logic in previos theses then constructs the TMR Assembly and simulates it. Chapter VI describes the *Reconciler* used to coordinate different architectures in this design. Chapter VII is a description of the *Interrupt* module designed for correcting errors in the registers. Chapter VIII shows the simulation of the full design without any circuitry to handle the reporting of errors. This chapter explains the function of the ISR and how different components work together. Chapter IX introduces the component used to store necessary data for future analysis when an error occurs. This component is *Error Syndrome Storage Device* and its function of the full design is verified in this chapter. Chapter X contains conclusions and topics for follow-on research.

F. ADDITIONAL DOCUMENTATION

Appendix A contaions all schematics, test benches, and simulation results discussed in this thesis. Some the the figures are zoomed in to provide better views of the small numbers on the buses. Appendix B is the description of the whole instruction set for the KDLX. Appendix C contains VHDL codes for all components designed in this thesis. The VHDL files for the KDLX processor are also included.

G. CHAPTER SUMMARY

This chapter has given fundamental understanding of radiation effects, FPGA and soft-core processors. The general concept of a TMR design has been introduced as well. Previous thesis work of CFTP will be reviewed in next chapter and the TMR technique for correcting an error will also be described. Reading old thesis work is always a good starting point of learning. Experience will be shared and direction for following research will be pointed out.

THIS PAGE INTETIONALLY LEFT BLANK

II. TMR REVIEW IN PREVIOUS WORK

To construct a CFTP design is a really complex work and needs a significant amount of time to finish. In order to have a flawless design, lots of conditions need to be considered and all problems should be solved in a reasonable way. Selecting components may take few days or months depending on how much data or information is collected. Decisions may still be changed at the last minute due to some unpredictable situations or inevitable factors. Any change in the final design on a component sometimes will cause a series of modifications to others. It is obvious that building a fully-functional CFTP does take much effort and designers have to really understand how circuits relate each other in order to revise or debug it. Unfortunately, graduate students at Naval Postgraduate School only stay a short amount of time. A big design like CFTP is chopped into several segments and assigned to different students. In this time constraints, students not only need to realize what previous students have done but also take up a design in progress. Most of the time, students picking up the segments do not have a chance to learn directly from students who have worked on this design before. Thus, the thesis becomes an important interface of experience inheritance between generations of students.

A. LASHOMB'S DESIGN

Peter A. LaShomb [1] expressed many concepts in both TMR design and FPGA selection. Traditional solutions for radiation effects were introduced including hardware redundancy, like Quadded Logic, and software improvement for fault tolerance, like time redundancy or software redundancy. In the TMR section, RADHARD and COTS were compared in availability, performance and cost. Potential benefits of those two were clearly described as well. The processor used in his TMR design was KCPSM, an 8-bit microcontroller. It was free downloaded from Xilinx's website and served as a readily available test-case processor while waiting availability of other high performance processors. Constructing and testing of the TMR were done on Xilinx Foundation series software which was available at Naval Postgraduate School (NPS). Voters and an error encoder were designed and explained in detail. Other issues including interrupt routine and memory/error controller were left as follow-on research.

In the FPGA section, different FPGAs were compared in a number of aspects. Five major parameters for choosing a good FPGA were gate count, availability of hardware and software, packages (flat-pack vs. ball-grid-array), re-programmablility and radiation tolerance. The Xilinx XCV800 was chosen as the candidate at that time for future implementation.

B. EBERT'S RESEARCH

A complete CFTP conceptual design presented was in Dean A. Ebert's thesis [9]. For hardware considerations, his thesis discussed why specific components were chosen and how chips communicated in an integrated circuit. More detail and realistic concepts about FPGA and CFTP configurations were described than before and chips were selected based on a number of space-environment considerations. Discussion of system memory was important and first described in this thesis. Memory configuration controller, functional logic and glue logic were also new ideas never talked about in previous work. The TMR circuitry was not one of the main topics in his research, but from his work one can visualize the external connections of the FPGA and understand the role of TMR in the CFTP process. Figure 4 illustrates the layout of the board he developed.

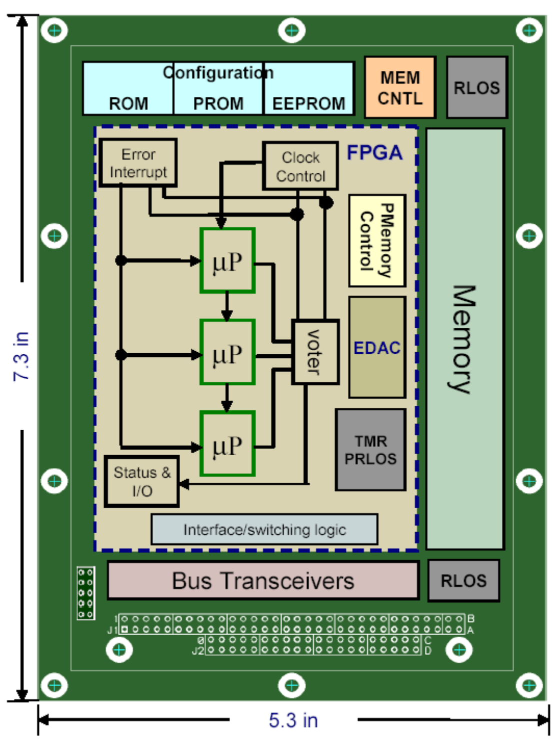


Figure 4. CFTP Conceptual Diagram (From Ref. [9].)

The CFTP will be launched into LEO orbit on two satellites, NPSAT-1 and MidSTAR-1, in 2006. How the Department of Defense and Navy Space Experiment Review Board (SERB) and the Space Test Program (STP) Office were involved with these two satellites was described in his thesis. Other documents related to design descriptions and requirements of the STP office were attached as appendixes as well.

C. JOHNSON'S IMPLEMENTATION

Steven A. Johnson [5] focused his work on TMR design. The essential components to make a circuit be fault-tolerant were identified. Circuits designed in Lashomb's thesis could not be used due to different design architecture and the significant upgrade of computer-aided-design software employed. Basic concepts for constructing a TMR circuit were still the same, but implemented in a different way.

KDLX, a 16-bit processor, better than 8-bit KCPSM processor, was the processor used in Johnson's research. His design consisted of *tmra*, *Interrup*, *Error Syndrome Storage Device (ESSD)* and *Reconciler*. The block named *tmra* consists of three KDLX processors and six voters. All processor output signals have to be voted. *Interrup* was compiled in a state diagram and used to trigger the interrupt service routine to correct an error inside the KDLX. *ESSD* was used to save the error syndrome in order to offer a log file for analysis. The KDLX is a Harvard architecture device which has two address buses and two data buses, a set of address and data bus for instruction memory and another set for data memory. The off-chip memory for the CFTP is Von Neumann architecture. The Von Neumann architecture has only one address bus and one data bus. Due to this difference, a *Reconciler* was designed to coordinate different timing constraints in order to make a proper read and write on memory. The difference between Harvard and Von Neumann architecture will be explained again while introducing KDLX in Chapter IV.

Johnson's full design schematic is shown in Figure 5. The memory is external to FPGA and it should be connected to *Reconciler* located at the top left corner. Normally, *tmra* communicates with *Reconciler* in order to access memory. Meanwhile, the syndrome data is latched into *ESSD* regardless of an error occurring or not. When an error occurs, a signal will be sent to *Interrup* and starts the Interrupt Service Routine (ISR). At this moment, KDLX is stalled and *ESSD* saves the error syndrome to memory through *Reconciler*. Then *Interrup* generates a TRAP instruction to KDLX and leads the whole circuit into an error correction condition. When KDLX sees the TRAP instruction, it jumps to a specific memory location and the program counter value before the jump is saved in an interrupt address register (IAR), a special register inside KDLX. In the error correction condition, the contents of all registers inside KDLX are saved to memory through voters. Then, each register is reloaded from memory. The purpose for doing this step is to correct any inconsistencies of the registers in all three KDLX processors. Since all contents have to pass voters while saving, any error inside any register will be corrected.

The last instruction in ISR is Return From Exception (RFE). This instruction indicates the end of ISR and the program counter saved in IAR will be loaded back to the KDLX. The logic gate set at the bottom in Figure 5 is a simple encoder of the RFE instruction which tells *Interrup* to stop the ISR. Finally, the whole circuit goes back to its normal operation.

This circuit primitively illustrated the complexity of the design and was built based on theory. Simulations and timing problems were left as follow-on research. It was proved on software that with such huge circuit built inside, the XCV800 FPGA still had a plenty of space and I/O blocks available.

D. CHAPTER SUMMARY

This chapter introduces work done by previous graduate students to give a direction where other resources are. This thesis mainly focuses on the TMR design and follows concepts in Lashomb and Johnson's research. The primitive design has been done and general concepts have been given. The *Interrup* takes over the whole circuit when an error occurs. Specific locations in memory are reserved for ISR and storing error syndromes. No other instructions should be able to access these locations.

In the next chapter, the testing environment and ISE software are introduced. Developing a consistent testing environment is important in order to have the right comparison. A description of software tools is also often useful information for a reader. This helps people understand more about simulation.

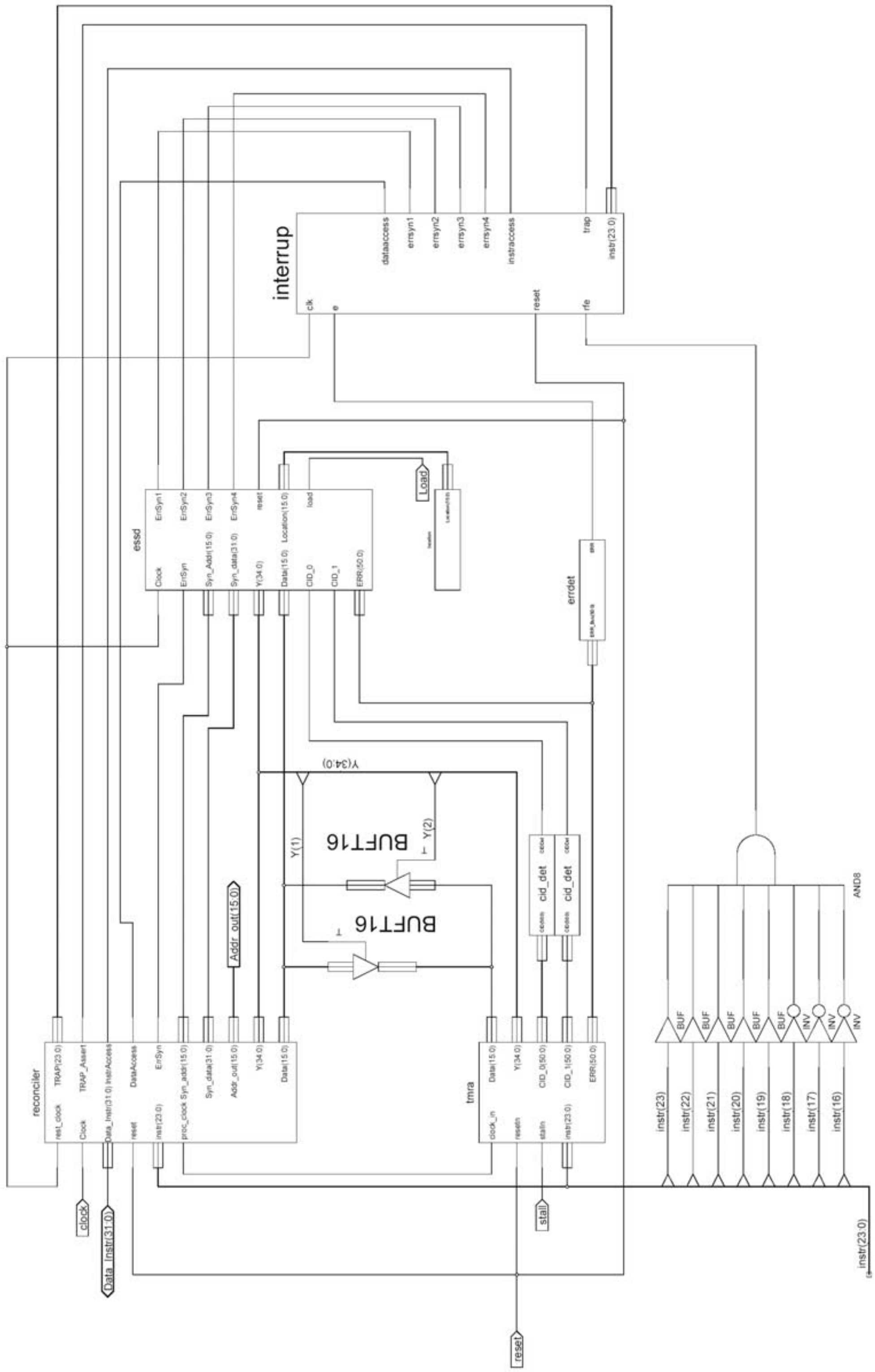


Figure 5. Full TMR Design Schematic (From Ref. [5].)

THIS PAGE INTENTIONALLY LEFT BLANK

III. TESTING ENVIRONMENT AND ISE SOFTWARE

It is hard to build a circuit without simulating it since that is the cheapest and fastest way to verify if a design works or not. The software used for simulation and the one used for constructing circuits do not need to be made by the same company. Different programs may use different ways to compile code or run simulations. A circuit built via some specific functions offered in one program may not fit into other programs. Therefore, a designer using programs made by different persons or companies sometimes face the problem of incompatibility. This issue can be solved if a package of service is bought. Generally speaking, products made by the same company are more compatible with each other and it is easier for that company to provide complete customer services.

Simulation is a very important component of design. A good design without a proper simulation may have degraded performance or efficiency. Sometimes inaccurate simulation results can mislead a designer into modifying something which is not supposed to be modified. A good simulation result could not only prove one's design but also help others understand the concept one embodies in a design. In terms of thesis research, simulation helps the designer and others to verify the design without spending too much time. Follow-on students can simply rerun the program and prove the consistency.

All settings of test benches for simulations will be offered in this thesis. This kind of information is usually not available on a lot of testing or simulation. Providing the simulation result without providing parameters means that others may not be able to understand the testing backgrounds and may prevent people from building an identical test bench. This is not important for a reader on the web, but it is important for a graduate student working on a thesis. First, a program sometimes crashes and files will be lost for some reasons which means someone may never get the same simulation outputs. Second, a modified circuit sometimes needs a new test bench for it. Without those parameters, simulation will be done under different testing environments and performance improvement may not be proved.

A. COMPUTER SPECIFICATIONS

System performance is often an important factor for testing. Running a program on a slow machine takes longer time than on a fast machine but the program result should be the same. When considering timing issues, performance of a system can be an important role. A slow computer basically cannot handle large amount of data and sometimes forces a user to reboot. As the TMR design gets more complicated, simulation will take longer for sure. The speed of how many data per second that a system can handle may affect the accuracy of simulation. Specifications of testing environment are always stated in a lot of computer magazines especially when testing a new hardware performance. The TMR design so far is not so complicated that it needs a high performance computer to simulate it. The information offered in Table 3 can be used as a reference in future thesis work.

Model	IBM ThinkPad A31 (2652Q5U)
Processor	Pentium®[2] 4 2.0 GHz
Memory	1 GB PC2100 DDR SDRAM
Hard Drive	40 GB 4200 RPM
Operating System	Windows 2000 Professional
OS version	5.0 Service Pack 3
Video Card	Mobility Radeon 7500 AGP

Table 3. Computer Specifications for Simulation

B. XILINX ISE SOFTWARE

The software used for constructing TMR design is a package called ISE made by Xilinx®[3], one of the largest FPGA manufactures in the world. This software is available at NPS and is used in labs for some courses. Students who want to do FPGA design should have basic understanding of this program. In order to do this research, it was necessary to learn about ISE and its associated simulator from the Xilinx website [10], an in-depth tutorial [11] or personal experience.

[2] Pentium is a registered trademark of Intel Corporation.

[3] Xilinx is a registered trademark of Xilinx Corporation.

ISE 5.2.03i was the version used for this thesis. Project Navigator was the overall controller of the ISE design system. The other important program used in this thesis called ModelSim®[4] is a powerful simulation tool. Its full version name is ModelSim XE II 5.6e. Logos of Project Navigator and ModelSim are shown in Figures 6 and 7.

Figure 6. Xilinx ISE Project Navigator Logo

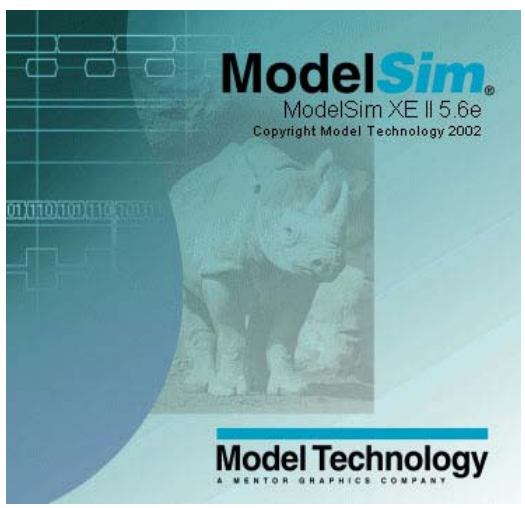


Figure 7. Xilinx ISE ModelSim Logo

The FPGA selected for CFTP was a Xilinx Virtex XCV800 hq 240 with speed grade of –4. This is an FPGA with 800 gate equivalents, in a package with 240 pins. Thus using ISE to develop and simulate the TMR design should be able to achieve the best design and the most realistic simulation of any other programs.

While this research was being performed, Xilinx released a new version of ISE 6.1i to its customers. Xilinx has warned that loading a project made in an old version of

[4] ModelSim is a registered trademark of Mentor Graphics Corporation.

ISE into ISE 6.1i will make an unrecoverable change and the project can no longer be read by older ISE software. Since a lot of simulations have been done at this moment and in order to keep the consistency of all testing environment, simulation on the latest version is left as a part of future work.

C. CHAPTER SUMMARY

This chapter summarized hardware and software information along with simulation environment. Simulation may look different in different software versions and sometimes new error will be generated. Undiscovered errors or potential defects of a design may be pointed out in the new version software. Sometimes the difference between new and old program is described in the user guide or on company's website. It is good to know primary evolution on new software and expect changes on old design. Work becomes efficient if one can exploit a program's features and functions.

Components in TMR design will be introduced in following chapters. Before constructing a full design, each circuit is built and tested. Therefore, simulation results will be used to explain how a circuit functions.

IV. KDLX INTRODUCTION

The KDLX, a 16-bit processor, is the kernel of this TMR design. Each component in the design is connected with a KDLX processor and tested as the final procedure. The KDLX is the soft-core processor to be used for each of the three processors in the design of the TMR system as shown in Figure 3. Due to the features of the KDLX pipeline and wiring delays, a circuit that works in a test bench by itself sometimes does not work with a KDLX. Knowing KDLX helps a designer foresee problems when building a circuit with it. Therefore, understanding KDLX is the first step for constructing a TMR design.

A. INSIDE KDLX

The KDLX is coded in VHDL, VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. It is composed of two top-level blocks, *core* and *IO_Pads*, as shown in Figure 8. The *core* and *IO_Pads* are names of blocks; *core1* and *IO_Pads1* are local block names representing *core* and *IO_Pads*, respectively, in the VHDL file called "dlx.vhd". The word KDLX at the top right corner is the name of the outer block. Numbers next to input and output pins represent the width of the bus. Words in bright green are local signals and none of the interconnections between these local pins are accessible from the outside (e.g., the connection between *In_Data* on *IO_Pads1* and *Input_data* on *core1*). All pins on the left side are input signals and all pins on the right side are output signals, except the *Data* bus. Controlled by *IO_Pads1*, the data bus on KDLX is bi-directional. It sends out data when writing to memory and stays high impedance otherwise. High impedance allows other devices connected on the data bus to drive the bus, but data will not be accepted by KDLX at this moment even if it flows inbound. The dash line in sky blue inside *IO_Pads1* is an internal connection. This internal connection functions only when input signal *Out_En_n* is low.

Notice that most input and output pins of KDLX are the same as *core1*. The function of *IO_Pads1* is to interface the external bi-directional data bus to input data and output data buses on *core1*. To understand KDLX better, the *core* needs to be explored.

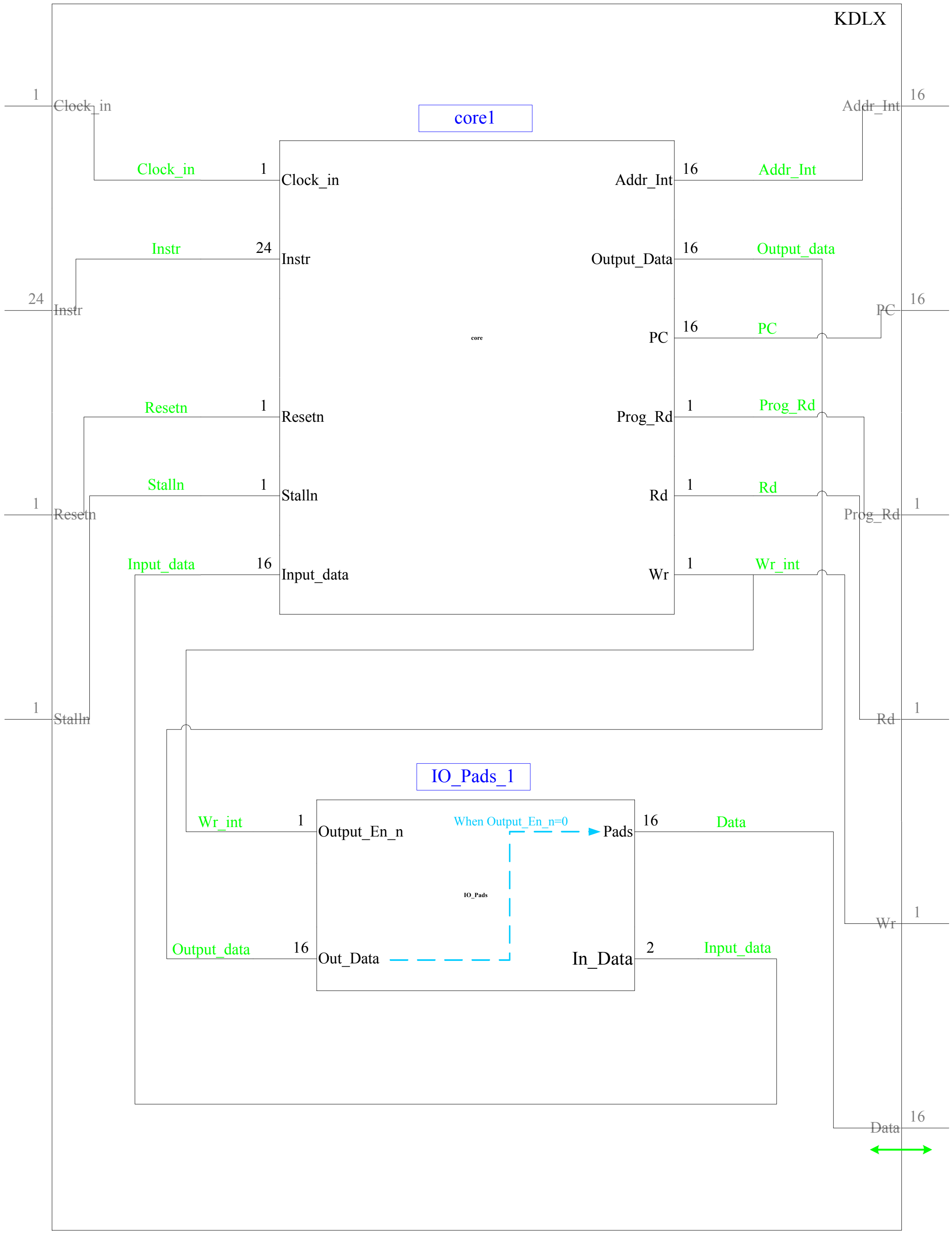


Figure 8. Inside KDLX

Major functional blocks are all inside *core* and are shown in Figure 9. These blocks are *zero_test*, *pipeline*, *regfile*, *pc_control*, *rw_control*, *alu*, *word_reg_single*, *word_mux3* and *word_mux4*.

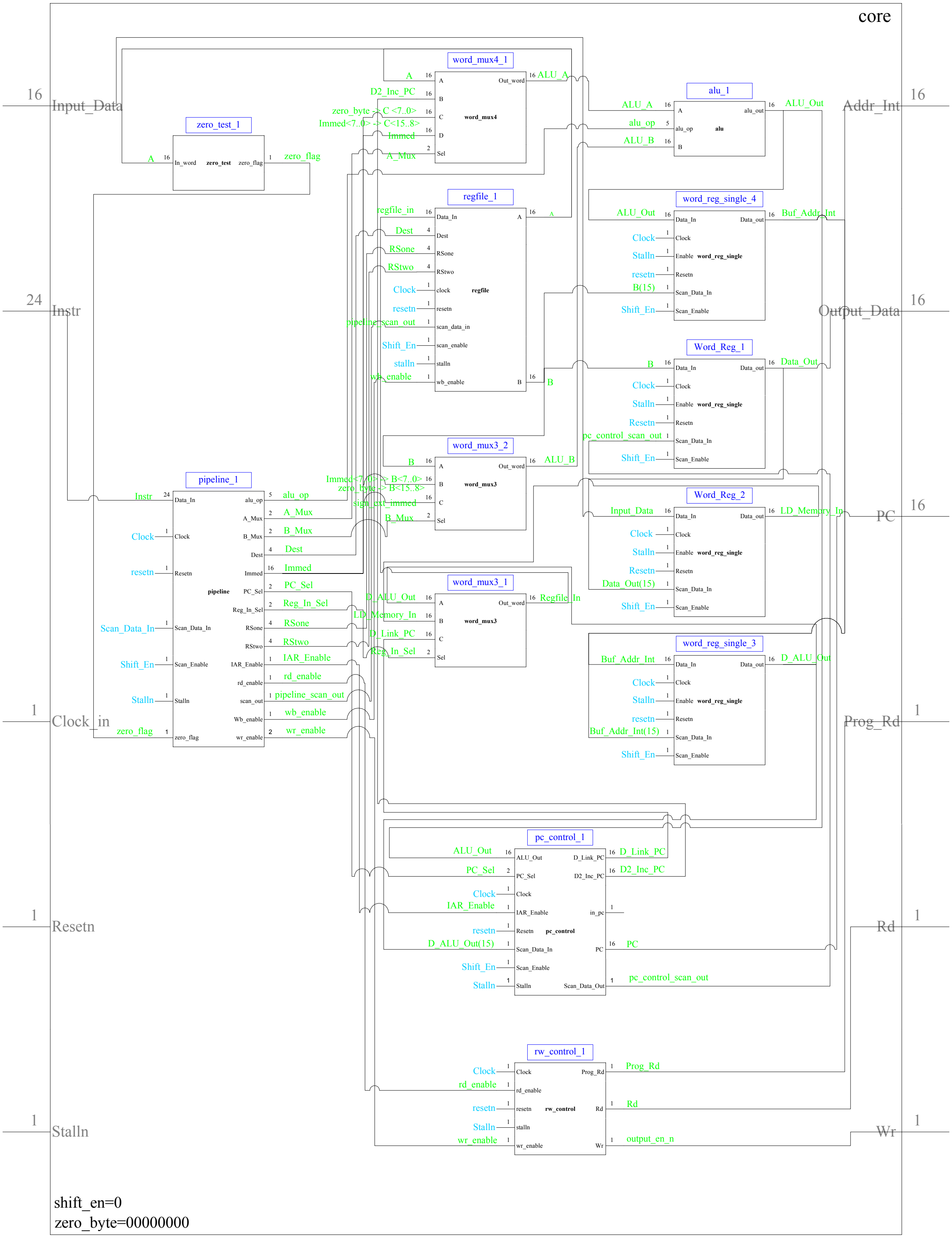


Figure 9. Inside *core*

The local block name used in the file "core.vhd" is boxed at the top of each function block. Words in bright green are still local signals and those in sky blue represent global signals **only** within the *core*. They are considered global signals because most blocks have these signals and they all receive the same value. For instance, all blocks receive zero when signal *resetn* is low. When the global signals *Shift_En* is low, local block *pipeline_1* may invert this signal to high internally and use it to trigger other functions. Therefore, *Shift_En* low in the *core* does not mean this signal is low inside *pipeline_1*. That is why global signals are used for the *core* only.

The detailed functioning of each block is described in KDLX's VHDL code. Figures 8 and 9 are plotted directly from the original VHDL code to illustrate how these components connect. Functions of important components like *alu*, *regfile*, *pc_control*, *rw_control* and *pipeline* are briefed here. Simulation of KDLX later will verify these functions.

1. Function of *alu*

This block is able to do addition, logic computation, and barrel shifting. Subtraction can be achieved by adding a positive number with a negative number. KDLX uses 2's complement arithmetic to do calculation. A simple 8-bit 2's complement number table is shown in Table 4.

Binary number	Equivalent Decimal number
1 1 1 1 1 1 1 1	127
.	.
.	.
.	.
0 0 0 0 0 0 1 1	3
0 0 0 0 0 0 1 0	2
0 0 0 0 0 0 0 1	1
0 0 0 0 0 0 0 0	0
1 1 1 1 1 1 1 1	-1
1 1 1 1 1 1 1 0	-2
1 1 1 1 1 1 0 1	-3
1 1 1 1 1 0 1 1	-4
.	.
.	.
.	.
1 0 0 0 0 0 0 0	-128

Table 4. 2's Complement Numbers

Logic computation includes logic AND, OR and XOR functions. KDLX allows a user to do logic computation between contents of two registers or the contents of a register and an immediate value.

A built-in barrel shifter gives KDLX the ability to do logic or arithmetic shifting.

2. Function of *regfile*

All 15 registers of KDLX are in this block. The inbound data bus is connected to all registers and an enable bus is used to control which register is being written. Two big muxes, *MUXA* and *MUXB*, route the output of a selected register to the outbound data bus.

3. Function of *pc_control*

The program counter sends the address to the instruction memory in order to fetch an instruction for next step. The *pc_control* assumes an important role while executing a Branch, Jump or TRAP instruction. For some instructions like Jump and Link, *pc_control* will save the return address of the instruction that comes after the next 2 instructions. This is because KDLX is pipelined, and, therefore, two instructions after the Jump will be executed before the jump occurs. The return address is saved in register 15. Since no instruction in KDLX is able to read the return address in register 15 directly, another circuit needs to be constructed in order to jump back to where the Jump and Link instruction left off.

Another important component in *pc_control* is the interrupt address register (IAR) which has been mentioned in Johnson's implementation. IAR is a register not accessible for a user. This special register is merely used to save the return address of the TRAP instruction. When the TRAP instruction is executed, the return address (which is the address right after the next 2 instructions) is saved into the IAR. After this, the program counter jumps to another memory location and start reading another set of instructions. Another instruction named Return From Exception (RFE) will be at the end of the instruction set. RFE will read the IAR and jump back to the memory location indicated. The jump, branch and trap implementations will be discussed again while simulating KDLX in this chapter.

4. Function of ***rw_control***

Obviously this is where KDLX controls read, write and program read signals for the memory modules that are attached to it. An important point here is that the KDLX read and write signals are active low. This means these two signals are activated at the falling edge of clock.

5. Function of ***pipeline***

Inheriting the nature of DLX, the KDLX is a five-stage pipelined processor, i.e., Fetch, Decode, Execute, Memory and Write Back. At the Decode stage, signals used to select registers in *regfile* are assigned. At the Execute stage, eight instructions are specific monitored. These eight instructions are Jump, Jump and Link, Branch if Equal Zero, Branch if Not Equal Zero, RFE, TRAP, Jump Register and Jump Register and Link. At the Memory stage, the signals are generated to allow the KDLX to read from or write to memory. The last stage, Write Back stage, allows most of the instructions to write to registers except some specific ones.

6. KDLX Summary

Thankfully, the ISE software has the ability to transfer VHDL code to a schematic so the user has an option to study a circuit without understanding VHDL code. The Schematic is more graphical than code and allows people to physically see how circuit is wired. The schematic symbol of KDLX is shown in Figure 10.

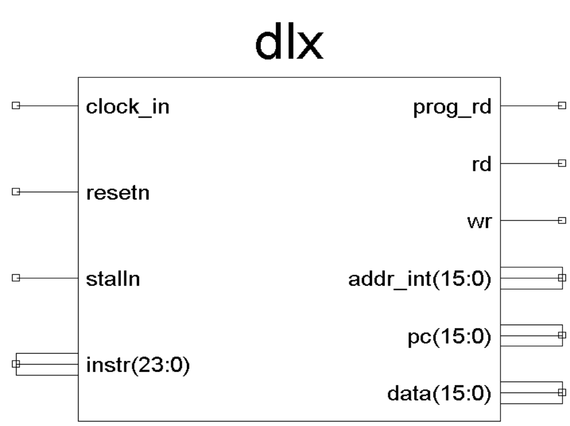


Figure 10. Schematic Symbol of KDLX

a. *Inputs and Outputs*

As mentioned earlier, KDLX has four inputs, five outputs and one bi–direction bus. Four inputs are three 1-bit pins, i.e., *clock_in*, *resetn* and *stalln*, and one 24-bit instruction bus. Five outputs are three 1-bit pins, i.e., *prog_rd*, *rd* and *wr*, and two 16-bit buses, i.e., *addr_int(15:0)* and *pc(15:0)*. The only bi-directional bus is a 16-bit data bus. Functions of these pins are listed in Table 5.

Symbol	Signal Name	Function
clock_in	Clock input	
resetn	Reset	Reset KDLX when low. All register contents are cleared.
stalln	Stall	Stall KDLX when low. Stall everything including data in pipeline stage.
instr(23:0)	Instruction Bus	Receive instructions sent from instruction memory.
prog_rd	Program Read	
rd	Read	Read data from data memory when low.
wr	Write	Write data to data memory when low.
addr_int(15:0)	Data Address	Send data address to data memory.
pc(15:0)	Program Counter	Send instruction address to instruction memory.
data(15:0)	Data Bus	Receive data from data memory or send data out to data memory.

Table 5. Function of Pins on KDLX

b. *Harvard Architecture and Von Neumann Architecture*

KDLX is a Harvard architecture device that has a pair of address and data buses for instruction memory and another pair for data memory. Figure 11 illustrates the concept of this architecture. The device at the center sends the address of instruction to an instruction memory. Then the instruction memory on the left will send an instruction back to the device. If the instruction received is to read or write data to data memory, the device at the center will send a data address to the data memory at the right side to indicated the memory location it wants to read or write. If the device wants to read, the data bus will be driven by data memory and data is sent from data memory to the device. If the device wants to write, the data bus will be driven by the device and data is sent from the device to data memory.

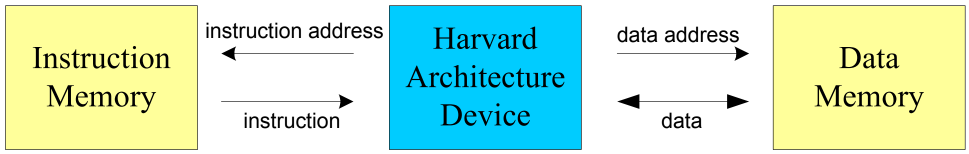


Figure 11. Harvard Architecture

By applying the same concept to KDLX, a picture like Figure 12 is understandable.

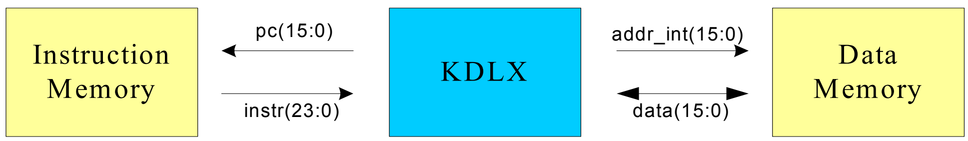


Figure 12. KDLX Connections with Two Memories

The Von Neumann architecture, on the other hand, has only one address bus and one data bus. A single memory is used in this architecture. A processor using Von Neumann architecture has less timing issues that need to be solved with memory since they are the same architecture. A Harvard-architecture processor, e.g., KDLX, needs to deal with possible timing mismatches with memory if only one memory is available. In the CFTP design, only one memory is available for the TMR circuit thus it is an instruction memory and a data memory as well. Recall that a component in Johnson's implementation (called *Reconciler*) is such a device used to integrate these two different architectures.

In order to consolidate a four-bus processor with a two-bus memory, the memory has to run in double speed to support two accesses per clock cycle. Figure 13 shows how KDLX communicates with only one memory.

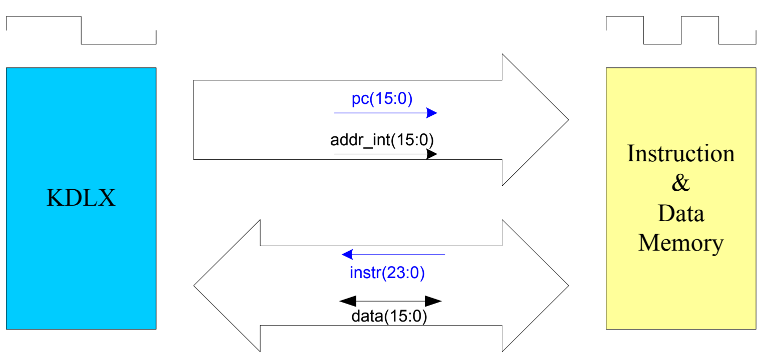


Figure 13. KDLX with One Memory

Since KDLX is a pipelined processor, it needs to be able to read or write data at the time it fetches an instruction. Both of these events can happen in one KDLX clock cycle. If the memory is twice as fast as the KDLX, it is able to deal with instruction at the first memory clock cycle and deal with data at the second memory clock cycle. In Figure 13, *pc(15:0)* and *instr(23:0)* are done in the first memory clock cycle; *addr_int(15:0)* and *data(15:0)* are done in the second memory clock cycle. The memory used here needs to be a 24-bit memory due to the width of instruction bus. Because the KDLX data bus is only 16-bits wide, only the lower 16-bit data will be accepted and the rest are buffered out.

B. PIPELINE CONCEPTS

The KDLX is a five-stage pipelined processor. These five stages are Fetch, Decode, Execute, Memory (Mem) and Write Back (WB). When doing a write, data is written to a register at the third clock cycle, i.e., the Execute stage. Therefore, a destination register used in one instruction is not available until 2 clock cycles later. This concept has significant impacts when creating a test bench. Figure 14 shows the pipeline execution of KDLX in normal operation.

Instruction number	Clock cycle 1	2	3	4	5	6	7	8	9
Instruction 1	Fetch	Decode	Execute	Mem	WB				
Instruction 2		Fetch	Decode	Execute	Mem	WB			
Instruction 3			Fetch	Decode	Execute	Mem	WB		
Instruction 4				Fetch	Decode	Execute	Mem	WB	
Instruction 5					Fetch	Decode	Execute	Mem	WB

Figure 14. Pipeline Execution in KDLX

In Figure 14, if Instruction 1 is loading data from the memory to register 3 (for example), the action to load register 3 starts at clock 3 and ends at clock 5 which means register 3 should not be accessed as a source register in Instruction 2, 3 and 4. Failing to do so, Instruction 2, 3 and 4 will either fetch a wrong value or unidentified data. Thus a new value of register 3 is only available for an instruction equivalent to or later than Instruction 5.

C. MEMORY IN SIMULATION

All components generated for TMR design were simulated with KDLX and memory as the final step. The ISE software has several different kinds of RAM or ROM in schematics for users to choose. A designer can also construct a memory via VHDL code. Another function called the CORE generator (Coregen) is a graphical interactive design tool in ISE software to help a user design a module. Due to its simplicity, memory used in this thesis was generated from Coregen.

A 24-bit memory with its simulation result is shown in Appendix A, section A. In order to explain, a copy of this simulation was made and labeled as Figure 15.

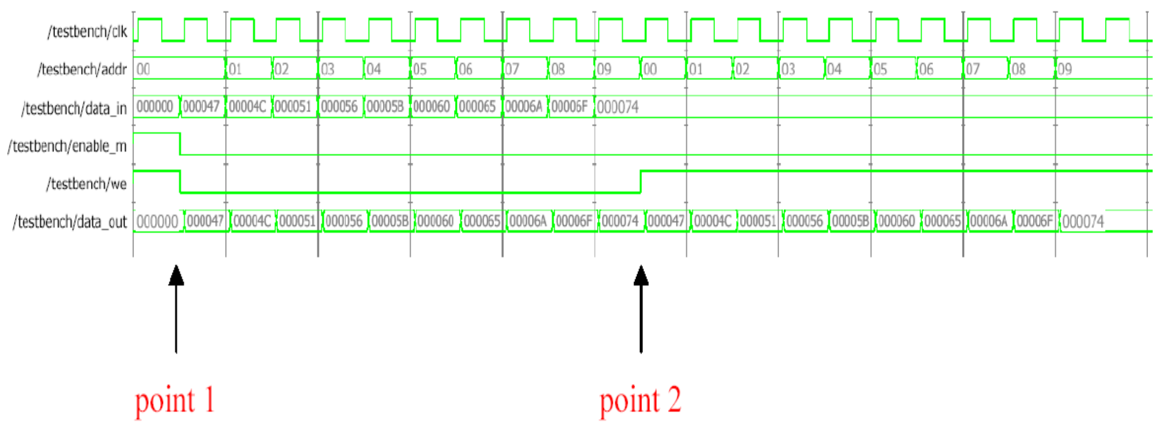


Figure 15. 24-bit Memory Simulation Result

Values on the address bus and input data bus are assigned in the test bench. In this simulation, memory is being written at point 1. The first value (i.e., 000047_{16}) is written into memory location 00_{16} and the second value (i.e., $00004C_{16}$) is written into memory location 01_{16} and so on. At point 2, memory starts being read and all values are output as originally initiated. One of the features of this memory is that data sent to *data_in* bus for writing comes out at the *data_out* bus. A designer can monitor the data written into memory from here. The write enable signal of this memory is active low; therefore it reads when this signal is high.

Memory used in simulation can be a RAM or ROM. A ROM is used as an instruction memory which is not allowed to be written. A RAM can be initialized by writing it before using it, but a ROM cannot since it does not have a write enable pin. Thus, a ROM needs to be pre-configured. In the ISE software, a user needs to generate a *coe* file and load it before a memory is generated in Coregen.

Memory offered in ISE software is not a real Von Neumann architecture since it has separate buses for data input and output. For simplicity, the TMR design in this thesis uses this kind of memory. Further modification is needed when a real Von Neumann architecture memory is available.

D. KDLX SIMULATION WITHOUT MEMORY

Operation codes (Opcodes) for the instruction set are described in Appendix B. This appendix includes all instructions that can be implemented in KDLX. Simulation of all instructions is one of the best ways to understand how KDLX functions. Before doing that, a simple simulation on KDLX itself is shown in Appendix A, section B. Figure 16 is a copy of this simulation result for explanation. All registers in the KDLX are initialized to the value 0000_{16} and register 0 is always zero.

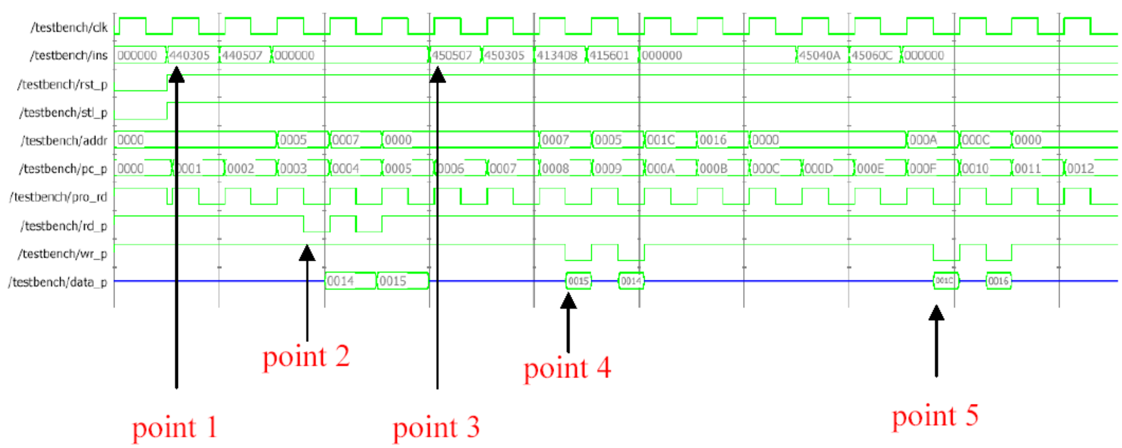


Figure 16. KDLX Simulation

In Figure 16, the first instruction at point 1 represents *loading the value at memory location [(register 0)+05] into register 3*. One can find a read signal becomes low at point 2. Comparing the timing here with Figure 14, it is proved that the action on the register occurs at Execute clock cycle. Since two values, 0014_{16} and 0015_{16}, are already available on the bus, KDLX loads these two data into register 3 and register 5, respectively. Recall that the pipeline features discussed in Figure 14, the new content of register 5 is not available at any clock cycle before point 3. Using register 5 anywhere before point 3 will use the old value in register 5 which is 0000_{16} in this case. In this simulation, three NOP are inserted before using register 5.

At point 3, instruction 450507_{16} stands for *storing the content of register 5 to the memory location [(register 0)+07]*. Again, the action starts at point 4 which is the Execute cycle for this instruction and the value loaded before shows up on the data bus. Since the data bus is high impedance at this clock cycle, the KDLX is able to drive the bus and output data. Without a high impedance, the KDLX is not able to use the bus because it assumes someone is using it. By checking the address bus of the KDLX simulation, one can find how the instruction and address correspond with each other.

The two instructions following the store instructions are 413408_{16} and 415601_{16}. These add immediate values to register 3 and 5, respectively, thus the data inside register 3 and 5 changes. This can be seen at point 5 when these two register contents are stored again.

For the rest of this thesis, we will use assembly language mnemonics to refer to instructions. For example, a register is represented by R. Thus, R0 stands for register 0 and R1 means register 1. Instead of a long explanation of each instruction, the operation symbol will also be used in following contents. An instruction like 440305_{16} will be represented as LW R3←Mem(R0+05). The symbols and expressions are defined in Appendix B.

E. KDLX SIMULATION WITH MEMORY

There are a total of 42 instructions for KDLX. Understanding these instructions is necessary to generate a test bench for the TMR processor. Utilizing different combinations of instructions can also help a designer use a short test bench to achieve the same goal of simulation. Instead of loading a large number of instructions into instruction memory before testing, pre-configured memory is used. Simply by selecting a different memory file, the same test bench can be used to test different instruction set; otherwise, several test benches are needed for different instruction set.

Instead of testing all instructions in one huge test bench, the 42 instructions were separated into four different instruction sets. Instruction set 1 and 2 test arithmetic and logic functions. Instruction set 3 and 4 test Jump, Branch and TRAP functions.

The schematic designed for this testing is shown in Figure 17. Memory at left side is a ROM used as instruction memory. The other one at right side is data memory which is a RAM. The *addr_box* contains only buffers used to truncate the width of the address bus since the memory address for this design is only 8-bits wide. Data memory is pre-configured with 0003_{16} since some numbers need to be loaded into registers at the beginning of simulation.

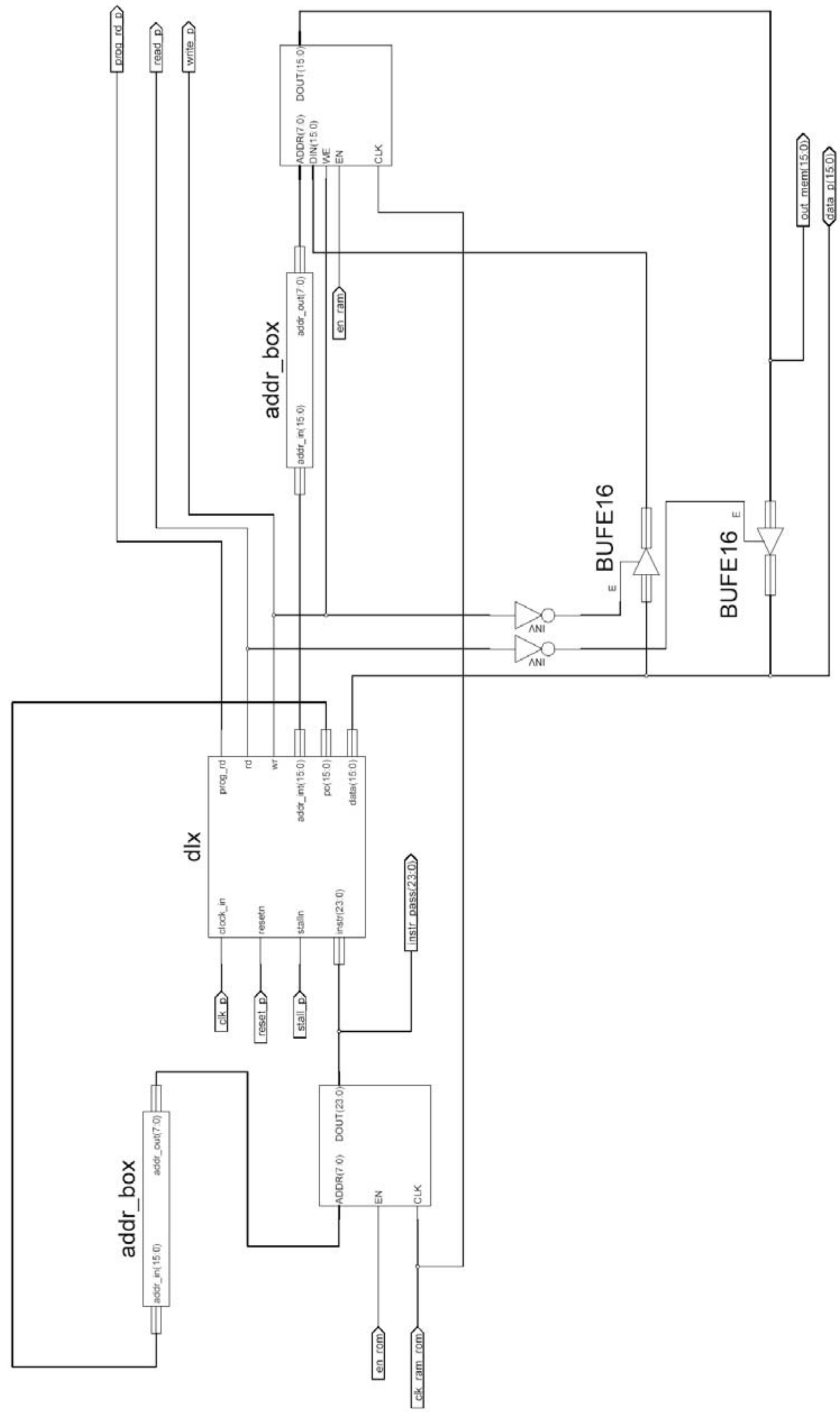


Figure 17. KLXD with Instruction and Data Memory

The write signal on KDLX is connected directly to data memory in order to be able to write memory. Since KDLX uses a bi-directional data bus, buffers with enable pin are needed to control the direction of data flow. Read and write signals are used to enable or disable these buffers. Extra output buses are added for monitor purposes. All test benches and simulation results are in Appendix A, section C.

1. Implementation Table of Instruction Set 1

An implementation table is generated as Table 6. Constructing such an instruction test bench can take a lot of time since instructions need to be rearranged and simulation results need to be checked. Instructions tested in each set are not many, but a number of loading and storing instructions are needed to check the data. All numbers in Table 6 are hexadecimal and R0 is always zero.

Instruction (operation symbol)		Opcode	Value through Data Bus
LW	R1←Mem(R0+03)	440103	
SW	R1→Mem(R0+08)	450108	0003
LW	R2←Mem(R0+04)	440204	
SW	R2→Mem(R0+09)	450209	0003
ADD	R1+R2→R3	011320	
SW	R3→Mem(R0+0D)	45030D	0006
ADDI	R1+ext(F9)→R4	4114F9	
SW	R4→Mem(R0+0E)	45040E	FFFC
ADDUI	R1+(0A) →R5	21150A	
SW	R5→Mem(R0+0F)	45050F	000D
AND	R1•R3→R6	091630	
SW	R6→Mem(R0+10)	450610	0002
ANDI	R4•(FD)→R7	2947FD	
SW	R7→Mem(R0+11)	450711	00FC
LHI	R8←FF||$(0)^8$	0808FF	
SW	R8→Mem(R0+12)	450812	FF00
OR	R1+R3→R9	0A1930	
SW	R9→Mem(R0+13)	450913	0007
ORI	R1+(F0)→R10	2A1AF0	
SW	R10→Mem(R0+14)	450A14	00F3
SEQ	R1=R2→R11=1	181B20	
SW	R11→Mem(R0+15)	450B15	0001
SEQ	R1≠R3→R12=0	181C30	
SW	R12→Mem(R0+16)	450C16	0000
SEQI	R1=(0003)→R13=1	581D03	
SW	R13→Mem(R0+17)	450D17	0001
SEQI	R1≠(0004)→R14=0	581E04	

Instruction (operation symbol)		Opcode	Value through Data Bus
SW	R14→Mem(R0+18)	450E18	0000
SLL	$R4^{\leftarrow R2=(0003)}$→R15	114F20	
SW	R15→Mem(R0+19)	450F19	FFE0
SLLI	$R4^{\leftarrow (0005)}$→R3	514305	
SW	R3→Mem(R0+1A)	45031A	FF80
SRA	$R4^{\rightarrow R1=(0003)}$→R5	134510	
SW	R5→Mem(R0+1B)	45051B	FFFF
SRLI	$R4^{\rightarrow (0003)}$→R6	524603	
SW	R6→Mem(R0+1C)	45061C	1FFF
SUBI	R8−ext(7B)→R7	43877B	
SW	R7→Mem(R0+1D)	45071D	FE85
XOR	R9⊕R10→R11	0B9BA0	
SW	R11→Mem(R0+1E)	450B1E	00F4

Table 6. Instruction Set 1

There are four sections in this map. Instructions for loading or computing data are implemented first in each section. Instructions for storing are used for checking data and are implemented later. The third column lists all Opcodes for implementing and the fourth column shows all data that should come out on the data bus.

2. Simulation Result of Instruction Set 1

To see the difference with the simulation of KDLX only, part of the simulation results is shown in Figure 18.

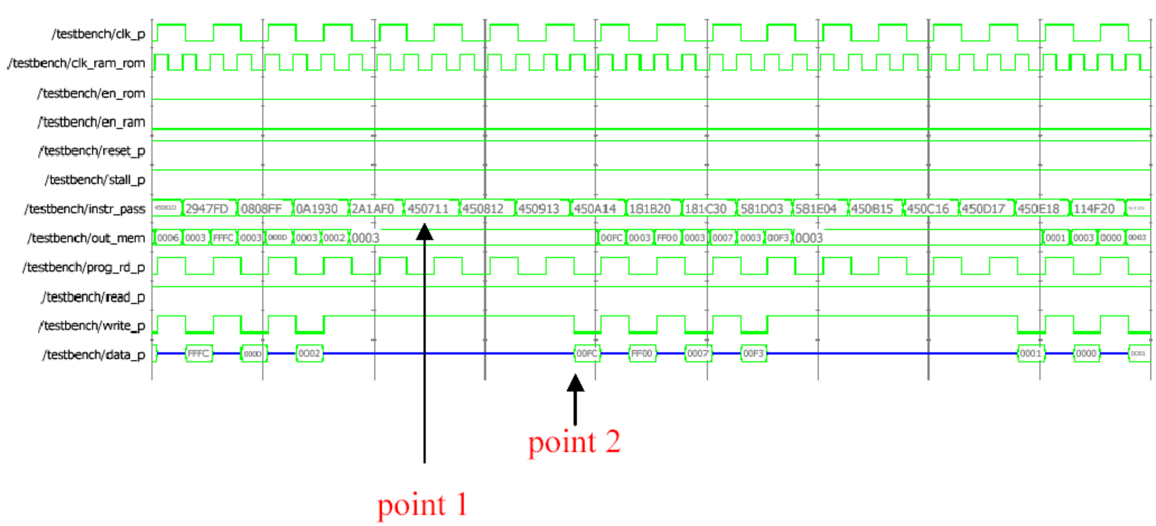


Figure 18. Simulation of KDLX with Memory

In order to make sure that the memory is stable before KDLX is going to use it, the memory clock cycle is doubled. The instruction memory will be ready before KDLX reads the instruction. The data memory will write data in a very short time and always be ready to be read by the KDLX.

Comparing timing before and after KDLX connects with the memory, a delay of the read and write operation can be found. In Figure 18, the instruction at point 1 does not start the write until point 2. Without the memory, this signal should be about one-half clock cycle earlier than point 2. This difference is due to the timing delays from the connecting memory. The fourth cycle of the KDLX clock is Mem which means that the KDLX is accessing memory at this time.

Another delay shows on instruction fetching. (Recall the schematic in Figure 17.) The program counter of KDLX sends out an instruction address to the instruction memory. Then the instruction memory reads the program counter and sends out an instruction to KDLX. This delay makes each instruction in Figure 18 start at the falling edge of clock. This is not like the instruction in Figure 16 which starts at the rising edge. The same delay happens when KDLX reads from or writes to the data memory.

The pipeline feature can also be seen in Figure 18. While KDLX is still sending out data, it is simultaneously fetching a new instruction.

An alternative way to check the simulation result is to construct tables for memories and registers as shown in Table 7. The instruction memory is pre-configured as the first table at the left. The second table shows how the contents of registers change in the simulation. The third table at the right expresses values in different locations after the simulation is done. Blank areas in data memory will contain the default value 0003_{16}.

In the instruction memory, a series of store instructions is used to check the contents in registers. A series of load instructions is used to check the contents in the memory locations. The first six Opcodes implement the instructions in section 1 of Table 6. Then the Opcodes from memory locations 08 to 10 execute the instructions in section 2 of Table 6. All instructions for loading and computation are executed before storing to memory. The instruction sequence in Table 6 is used to track which part of the instructions are checked when storing.

Instruction Mem			
00		2D	45071D
01	440103	2E	450B1E
02	440204	2F	000000
03	000000	30	000000
04	000000	31	000000
05	450108	32	450101
06	450209	33	450201
07	000000	34	450301
08	011320	35	450401
09	4114F9	36	450501
0A	21150A	37	450601
0B	000000	38	450701
0C	091630	39	450801
0D	45030D	3A	450901
0E	45040E	3B	450A01
0F	45050F	3C	450B01
10	450610	3D	450C01
11	2947FD	3E	450D01
12	0808FF	3F	450E01
13	0A1930	40	450F01
14	2A1AF0	41	000000
15	450711	42	000000
16	450812	43	000000
17	450913	44	44010D
18	450A14	45	44020E
19	181B20	46	44030F
1A	181C30	47	440410
1B	581D03	48	440511
1C	581E04	49	440612
1D	450B15	4A	440713
1E	450C16	4B	440814
1F	450D17	4C	440915
20	450E18	4D	440A16
21	114F20	4E	440B17
22	514305	4F	440C18
23	134510	50	440D19
24	524603	51	440E1A
25	450F19	52	440F1B
26	45031A	53	44011C
27	45051B	54	44021D
28	45061C	55	44031E
29	43877B	56	000000
2A	0B9BA0	57	000000
2B	000000	58	000000
2C	000000	59	000000

Register		
00		
01	0003	
02	0003	
03	~~0006~~	FF80
04	FFFC	
05	~~000D~~	FFFF
06	~~0002~~	1FFF
07	~~00FC~~	FE85
08	FF00	
09	0007	
10	00F3	
11	~~0001~~	00F4
12	0000	
13	0001	
14	0000	
15	FFE0	

Data Mem	
00	
01	
02	
03	
04	
05	
06	
07	
08	0003
09	0003
0A	
0B	
0C	
0D	0006
0E	FFFC
0F	000D
10	0002
11	00FC
12	FF00
13	0007
14	00F3
15	0001
16	0000
17	0001
18	0000
19	FFE0
1A	FF80
1B	FFFF
1C	1FFFF
1D	FE85
1E	00F4
1F	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
2A	

Table 7. Tables of Registers and Memories in Simulation 1

The Opcode, $4114F9_{16}$, at memory location 09_{16} implements ADDI R1+ext(F9)→R4. The original value of R1 is 0003_{16} which equals to 3_{10}. Since KDLX uses 2's complement numbers, the sign extension value of $F9_{16}$ is $FFF9_{16}$ which is (–7) in decimal. The sum of 3_{10} and $(-7)_{10}$ is $(-4)_{10}$. Convert $(-4)_{10}$ to a binary number and do 2's complement, the result in hexadecimal is $FFFC_{16}$. This agrees with the value in data memory location $0E_{16}$.

3. Implementation Table of Instruction Set 2

The rest of the instructions (not including Jump and Branch) are listed in Table 8. This table only shows the instructions that were tested in this thesis. The table does not include the instructions for configuring memory contents. This will be explained further in the simulation section of this chapter.

Instruction (operation symbol)		Opcode	Expected Value
SGE	R1>R3→R13=1	191D30	
SW	R13→Mem(R0+1F)	450D1F	0001
SGE	R15>R14→R9=0	19F9E0	
SW	R9→Mem(R0+20)	450920	0000
SGEI	R15≥ext(E8)→R10=0	59FAE8	
SW	R10→Mem(R0+21)	450A21	0000
SGEI	R15≥ext(E0) →R11=1	59FBE0	
SW	R11→Mem(R0+22)	450B22	0001
SGT	R4>R15→R6=1	1A46F0	
SW	R6→Mem(R0+23)	450623	0001
SGT	R15>R4→R7=0	1AF740	
SW	R7→Mem(R0+24)	450724	0000
SGTI	R15>ext(FF)→R8=0	5AF8FF	
SW	R8→Mem(R0+25)	450825	0000
SGTI	R15>ext(87)→R9=1	5AF987	
SW	R9→Mem(R0+26)	450926	0001
SLE	R1=R2→R10=1	1B1A20	
SW	R10→Mem(R0+27)	450A27	0001
SLE	R1<R13→R11=0	1B1BD0	
SW	R11→Mem(R0+28)	450B28	0000
SLEI	R1≤ext(03)→R12=1	5B1C03	
SW	R12→Mem(R0+29)	450C29	0001
SLEI	R1≤ext(02)→R13=0	5B1D02	
SW	R13→Mem(R0+2A)	450D2A	0000
SLT	R15<R1→R6=1	1CF610	
SW	R6→Mem(R0+01)	450601	0001
SLT	R1<R15→R7=0	1C16F0	

Instruction (operation symbol)		Opcode	Expected Value
SW	R7→Mem(R0+02)	450702	0000
SLTI	R1<ext(0D)→R8=1	5C180D	
SW	R8→Mem(R0+03)	450803	0001
SLTI	R1<ext(01)→R9=0	5C1901	
SW	R9→Mem(R0+04)	450904	0000
SNE	R1≠R2→R10=0	1D1A20	
SW	R10→Mem(R0+05)	450A05	0000
SNE	R1≠R15→R11=1	1D1BF0	
SW	R11→Mem(R0+06)	450B06	0001
SNEI	R1≠ext(03)→R12=1	581C03	
SW	R12→Mem(R0+07)	450C07	0001
SNEI	R15≠ext(E1)→R13=0	58FDE1	
SW	R13→Mem(R0+08)	450D08	0000
SRAI	$R3^{\rightarrow(0006)}\rightarrow R6$	533606	
SW	R6→Mem(R0+09)	450609	FFFE
SRL	$R3^{\rightarrow R2=(0003)}\rightarrow R7$	123720	
SW	R7→Mem(R0+0A)	45070A	1FF0
XORI	R15⊕(8A)→R8	2BF88A	
SW	R8→Mem(R0+0B)	45080B	FF6A
SUBUI	R3–(80)→R9	233980	
SW	R9→Mem(R0+0C)	45090C	FF00
SUB	R1–R3→R14	031E30	
SW	R14→Mem(R0+0D)	450E0D	0083

Table 8. Instruction Set 2

4. Simulation Result of Instruction Set 2

The complete table set that shows all values inside memories and registers for this simulation is shown in Table 9. In the instruction memory part of the table, the instructions shown in Table 8 actually start at memory location $2A_{16}$. Instructions before this point are used to generate the same register values used in instruction set 1. The first column of Table 9 shows values that are identical to the final results in Table 7.

The registers change many times during this simulation, but the table only shows the initial and final values. The first column as described in the last paragraph is the starting data for instruction set 2. The second column lists all final values in registers.

This simulation uses different data memory locations than instruction set 1. This provides a boundary test for memory while testing KDLX.

This instruction set demonstrates most of the possible comparisons between registers or of a register with an immediate value. Since the KDLX uses 2's complement values, 0003_{16} is obviously greater than $FF80_{16}$. Logical operations like ANDI, ORI, and XORI do not use sign extension on an immediate value.

Instruction Mem			
00		30	450A21
01	410103	31	450B22
02	410203	32	1A46F0
03	0803FF	33	1AF740
04	0804FF	34	5AF8FF
05	0805FF	35	5AF987
06	08061F	36	450623
07	410380	37	450724
08	4104FC	38	450825
09	4105FF	39	450926
0A	2166FF	3A	1B1A20
0B	0807FE	3B	1B1BD0
0C	0808FF	3C	5B1C03
0D	080FFF	3D	5B1D02
0E	210AF3	3E	450A27
0F	217785	3F	450B28
10	210BF4	40	450C29
11	410907	41	450D2A
12	410D01	42	1CF610
13	410E00	43	1C17F0
14	410C00	44	5C180D
15	410FE0	45	5C1901
16	000000	46	450601
17	000000	47	450702
18	450100	48	450803
19	450200	49	450904
1A	450300	4A	1D1A20
1B	450400	4B	1D1BF0
1C	450500	4C	581C03
1D	450600	4D	58FDE1
1E	450700	4E	450A05
1F	450800	4F	450B06
20	450900	50	450C07
21	450A00	51	450D08
22	450B00	52	533603
23	450C00	53	123720
24	450D00	54	2BF88A
25	450E00	55	233980
26	450F00	56	031E30
27	000000	57	450609
28	000000	58	45070A
29	000000	59	45080B
2A	191D30	5A	45090C
2B	19F9E0	5B	450E0D
2C	59FAE8	5C	000000
2D	59FBE0	5D	000000
2E	450D1F	5E	000000
2F	450920	5F	000000

Register		
00		
01	0003	0003
02	0003	0003
03	FF80	FF80
04	FFFC	FFFC
05	FFFF	FFFF
06	1FFF	FFFE
07	FE85	1FF0
08	FF00	FF6A
09	0007	FF00
10	00F3	0000
11	00F4	0001
12	0000	0001
13	0001	0000
14	0000	0083
15	FFE0	FFE0

Data Mem	
00	
01	0001
02	0000
03	0001
04	0000
05	0000
06	0001
07	0001
08	0000
09	FFFE
0A	1FF0
0B	FF6A
0C	FF00
0D	0083
0E	
0F	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
1A	
1B	
1C	
1D	
1E	
1F	0001
20	0000
21	0000
22	0001
23	0001
24	0000
25	0000
26	0001
27	0001
28	0000
29	0001
2A	0000

Table 9. Tables of Registers and Memories in Simulation 2

5. Implementation Table of Instruction Set 3

This instruction set starts by testing the Jump and Branch instructions. The complete implementation is listed in Table 10. There are no divisions in this table and the sequence of execution is from top to bottom. If an instruction jumps to the wrong memory location, one or all contents of the target registers will not agree with the expected value shown here.

	Instruction (operation symbol)	Opcode	Expected Value
LW	R1←Mem(R0+03)	410103	
LW	R2←Mem(R0+04)	410204	
LW	R3←Mem(R0+00)	410300	
LW	R4←Mem(R0+06)	410406	
BNEZ	R1≠0→Prog_Addr←(05)+1+ext(04) Note: PC=05 and (05)+1+ext(04)=0A	C01004	
BEQZ	R3=0→Prog_Addr←(0A)+1+ext(04) Note: PC=0A and (0A)+1+ext(04)=0F	C13004	
ADDI	R0+ext(25)→R5	410525	
J	(0020)→Prog_Addr	C80020	
JAL	(0014)→Prog_Addr ; (23)→R15 Note:(23) is return address	E80014	
ADDI	R0+ext(8A)→R6	41068A	
ADDI	R0+ext(40)→R7	410740	
ADD	R1+R2→R8	011820	
ADD	R1+R4→R9	011940	
SW	R15→Mem(R0+01)	450F01	0023
JALR	R5→Prog_Addr ; (1D)→R15 Noter:(1D) is return address	685000	
J	(0030)→Prog_Addr	C80030	
SW	R5→Mem(R0+02)	450502	0025
SW	R6→Mem(R0+03)	450603	FF8A
SW	R7→Mem(R0+04)	450704	0040
SW	R8→Mem(R0+05)	450805	0007
SW	R9→Mem(R0+06)	450906	0009
SW	R15→Mem(R0+07)	450F07	001D
JR	R7→Prog_Addr	487000	
SW	R2→Mem(R0+08)	450208	0004

Table 10. Instruction Set 3

6. Simulation Result of Instruction Set 3

For Jump and Branch instructions, the sequence of instructions in memory is not the sequence of implementation. This can be easily understood by looking at Table 11.

The black arrows represent the normal sequence of operation. The blue dash lines stand for Jump or Branch instructions without link, and the blue solid lines stand for Jump and Link or Branch and Link.

The first branch occurs at memory location 05_{16}. Since the program counter at that point is 05_{16}, it branches to memory location $0A_{16}$ with a given immediate value 04_{16}. The action of branching occurs two clocks later due to pipelining, so the instructions at memory location 06_{16} and 07_{16} are fetched before the sequence branches to the new address.

At memory location $0A_{16}$, another branch instruction is executed. It branches to another memory location, $0F_{16}$. Because the Opcode 410525_{16} is fetched before the branch occurs, an immediate value is added into R5. This can be checked in the register table or in data memory location 02_{16} where Opcode 450502_{16} loads data to.

Opcode $E80014_{16}$ is a Jump and Link instruction. It jumps to address 14_{16} and save address 23_{16} into R15. There is no doubt that address 23_{16} is where the jump occurs, not address 20_{16}, 21_{16} or 22_{16}. In each case, the two instructions following Jump and Link are fetched before the jump instruction is executed.

The instruction at memory location $1A_{16}$ is Jump Register and Link. This allows KDLX to read the address it wishes to jump to directly from its internal register. Suppose one register is reserved for a special purpose and it contains a special memory location. Then KDLX can always jump to that specific memory location by simply reading the contents of that register without any extra instructions needing to be implemented.

Instruction Mem	
00	
01	410103
02	410204
03	410300
04	410406
05	C01004
06	000000
07	000000
08	
09	
0A	C13004
0B	410525
0C	000000
0D	
0E	
0F	C80020
10	000000
11	000000
12	
13	
14	011820
15	011940
16	450F01
17	000000
18	000000
19	000000
1A	685000
1B	000000
1C	000000
1D	
1E	
1F	
20	E80014
21	41068A
22	410740
23	
24	
25	C80030
26	000000
27	000000
28	
29	
2A	
30	450502
31	450603
32	450704
33	450805
34	450906
35	450F07
36	487000
37	000000
38	000000
39	
⋮	
40	450208
41	000000
42	000000
43	000000

1D→R15

23→R15

Register	
00	
01	0003
02	0004
03	0000
04	0006
05	0025
06	FF8A
07	0040
08	0007
09	0009
10	
11	
12	
13	
14	
15	

Data Mem	
00	
01	0023
02	0025
03	FF8A
04	0040
05	0007
06	0009
07	001D
08	0004
09	
0A	
0B	
0C	
0D	
0E	
0F	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
1A	
1B	
1C	
1D	
1E	
1F	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
2A	

Table 11. Tables of Registers and Memories in Simulation 3

7. Implementation Table of Instruction Set 4

This instruction set contains one of the most complicated instructions in the TMR design, which is the TRAP instruction. The TRAP instruction acts as Jump and Link or Branch and Link. The difference is that it saves its return address into the IAR, not into R15. The IAR is a specific register mentioned earlier when introducing the *pc_control* inside KDLX. Storing the return address into the IAR not only saves a register but also guarantees the integrity since it is only accessible for the TRAP instruction.

Another feature of the TRAP instruction is that it owns an instruction called Return from Exception (RFE). The RFE, Opcode $F80000_{16}$, only reads the content of IAR and jumps to that address. Since the IAR always contains the return address of the TRAP instruction, the RFE instruction only works with the TRAP instruction.

Instruction set 4 for testing the TRAP instruction is shown in Table 12.

Instruction	(operation symbol)	Opcode	Expected Value
ADDI	R0+ext(04)→R1	410104	
ADDI	R0+ext(07)→R2	410207	
TRAP	(0020)→Prog_Addr ; (06)→IAR Note: (06) is return address	280020	
ADDI	R0+ext(09)→R3	410309	
ADDI	R0+ext(15)→R4	410415	
ADDI	R0+ext(0A)→R7	41070A	
ADDI	R0+ext(11)→R8	410811	
ADDI	R0+ext(C2)→R10	410AC2	
RFE	(06)→Prog_Addr Note: (06) is IAR	F80000	
J	(0011)→Prog_Addr	C80011	
SW	R1→Mem(R0+01)	450101	0004
SW	R2→Mem(R0+02)	450202	0007
SW	R3→Mem(R0+03)	450303	0009
SW	R4→Mem(R0+04)	450404	0015
SW	R7→Mem(R0+07)	450707	000A
SW	R8→Mem(R0+08)	450808	0011
SW	R10→Mem(R0+0A)	450A0A	FFC2

Table 12. Instruction Set 4

8. Simulation Result of Instruction Set 4

The features of the TRAP instruction are shown in Table 13. When fetching the TRAP instruction at memory location 03_{16}, KDLX stores the return address 06_{16} to the IAR. Two clock cycles later in the TRAP, the program counter changes to 20_{16} and reads the instruction at that address. After implementing a few instructions, the KDLX sees the Opcode $F80000_{16}$ and retrieves address 06_{16} for the return. The content at location 06 is a Jump instruction. Therefore, the KDLX jumps again to memory location 11_{16}.

Some important features can be found in this implementation. First, the TRAP occurs exactly after 2 clock cycles; otherwise the Opcode $C80011_{16}$ will be fetched. Second, the IAR is not directly addressable, so using Opcode $F80000_{16}$ is the only way to verify the content of the IAR. Third, instruction set 4 can be an infinite loop if the test bench never stops. After jumping to memory location 11_{16}, the program counter keeps counting in order to read instructions. If no other signal stops the KDLX, it will read Opcode $F80000_{16}$ again. This retrieves the IAR and jumps back to memory location 06_{16}. The Opcode $C80011_{16}$ will lead KDLX to jumping to address 11_{16} then to keep on reading instructions until it hits $F80000_{16}$ again. This loop can be observed in the full simulation result for instruction set 4 in Appendix A, section C.

06→IAR

Instruction Mem	
00	
01	410104
02	410207
03	280020
04	410309
05	410415
06	C80011
07	000000
08	000000
09	
0A	
0B	
0C	
0D	
0E	
0F	
10	
11	450101
12	450202
13	450303
14	450404
15	450707
16	450808
17	450A0A
18	000000
19	000000
1A	000000
1B	
1C	
1D	
1E	
1F	
20	41070A
21	410811
22	410AC2
23	000000
24	000000
25	000000
26	F80000
27	000000
28	000000
29	
2A	

Register	
00	
01	0004
02	0007
03	0009
04	0015
05	
06	
07	000A
08	0011
09	
10	FFC2
11	
12	
13	
14	
15	

Data Mem	
00	
01	0004
02	0007
03	0009
04	0015
05	
06	
07	000A
08	0011
09	
0A	FFC2
0B	
0C	
0D	
0E	
0F	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
1A	
1B	
1C	
1D	
1E	
1F	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
2A	

Table 13. Tables of Registers and Memories in Simulation 4

F. CHAPTER SUMMARY

This chapter introduced several important components inside KDLX and discussed pipeline concepts. Drawing a schematic from VHDL code is a good way to understand KDLX.

The simulation of KDLX with and without memory illustrated the concept of the pipeline and developed ideas on how to organize a test bench. Most of the tables necessary for simulation purposes were generated in this chapter. Having the tables generated before constructing a test bench helps a designer to understand what the goal is and how to achieve it. Tables created by the simulation gives a designer a big picture on how things interact with each other. Sometimes things are hard to say but easy to see.

The TMR Assembly is designed in the next chapter. The function of the voter and how it corrects an error will be explained. Then we will combine three KDLX processors with voters to form a TMR Assembly. Important simulation concepts will be reviewed as well.

THIS PAGE INTENTIONALLY LEFT BLANK

V. TMR ASSEMBLY

The TMR Assembly is composed of three KDLX processors with voters on all outputs. All of the KDLX instructions have been tested in the simulation described in the previous chapter and the fundamental concept of KDLX has been established. The next step is to realize the function of a voter.

A voter is constructed by some simple logic gates and is able to find an error when inputs are not consistent. Since the CFTP will be operating in a relatively benign LEO orbit, the TMR design does not have to deal with too many errors per unit time. The assumption of the TMR design is that we will not see identical errors on two processors at the same time. The voters pass the majority vote so, if the errors are identical, they will not be detected (and will, in fact, be turned into truth.)

A. 1-BIT VOTER

The CFTP is designed to be fault tolerant by software. Its circuit needs to be able to detect an error and correct the error by itself. In order to achieve that, the concept of a voter is generated.

The function of a 1-bit voter has been introduced in Lashomb's thesis [1]. This section reviews the basic concepts and then starts constructing the TMR Assembly. Figure 19 shows what a 1-bit voter looks like. It is a simple circuit consisting of only AND and OR gates.

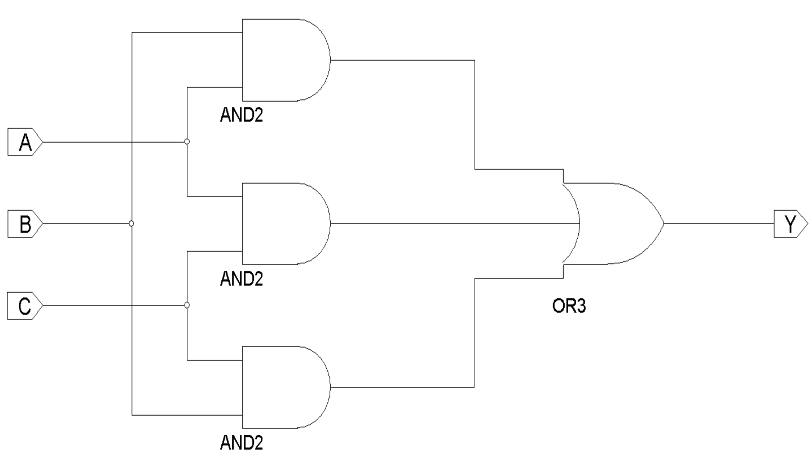


Figure 19. 1-Bit Majority Voter (After Ref. [1].)

The voter function is more obvious in the truth table shown in Table 14. This voter always selects the majority of identical bits as its output bit. If two or more inputs are incorrect, the voter output will also be incorrect. The ability to detect and correct two or more errors in a voter is not vital for a system (e.g., the CFTP) in LEO orbit.

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 14. Truth Table of A 1-Bit Voter (From Ref. [1].)

Assuming a single error, the output is always correct, but we cannot tell if there has been an error just by looking at this output. Therefore, some extra gates are added to report the occurrence of an error. Figure 20 shows a voter with error detection and Table 15 is its truth table.

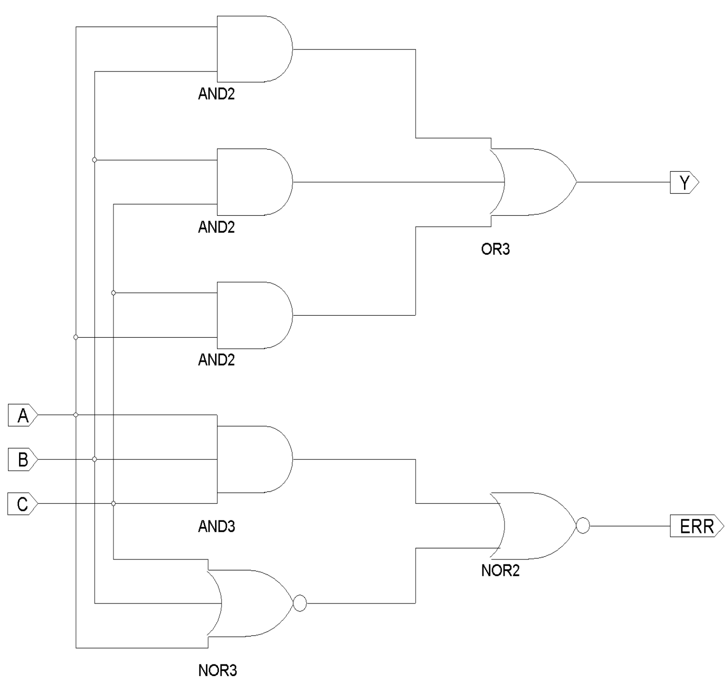


Figure 20. Voter with Error Detection (After Ref. [1].)

A	B	C	Y	ERR
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

Table 15. Truth Table of Voter with Error Detection (From Ref. [1].)

The error detection, *ERR*, is 1 when one of the inputs is not identical with the rest. When the CFTP is in space, it is possible to have an SEU on the voter itself. A bit flip may cause the voter output to be incorrect. Say the second column of Table 15 has a bit flipping on *A*. This flipping makes 1 become the majority bit and output *Y* will give a 1 not a 0. Since a voter is used to catch and correct an error, it is not pleasant if it has an error itself. Thus, some reliability is needed for the voter. A voter with added reliability is shown in Figure 21.

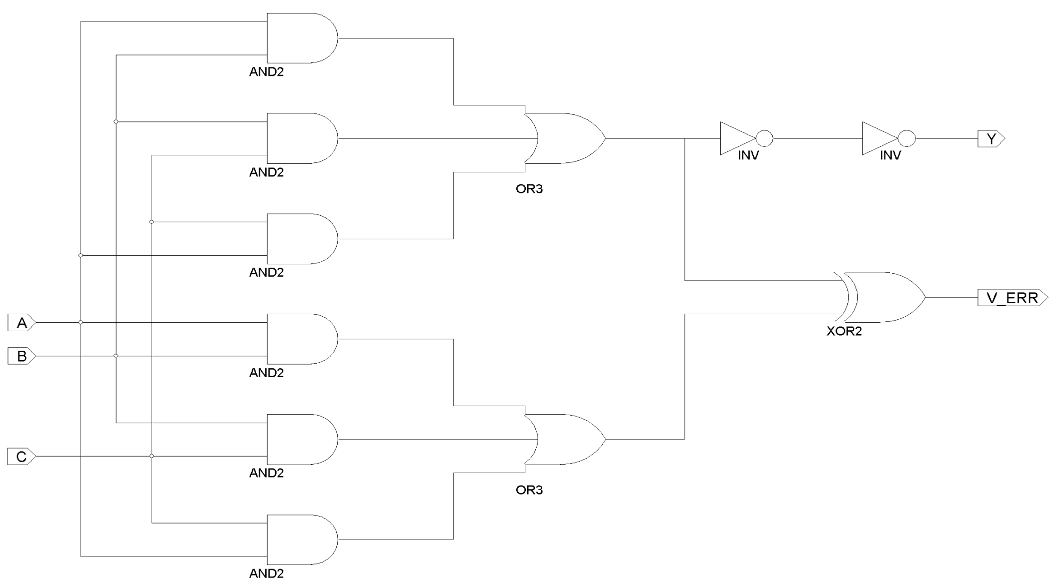


Figure 21. Voter with Added Reliability (After Ref. [1].)

This version is built by duplicating the original part of the voter and XORing the two parts to generate a voter error detection, *V_ERR*. If the voter errors, the outputs of the two OR3 in Figure 21 will not agree with each other, and *V_ERR* becomes 1. Table 16 is the truth table of this circuit.

A	B	C	Y	V_ERR
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Table 16. Truth Table of Voter with Added reliability (From Ref. [1].)

The last step is to collect all of these pieces to construct a complete single-bit voter. As introduced earlier, a voter with error detection is able to correct the error and tell the user an error has occurred. For the TMR design, knowing the existence of an error is not good enough since the error also has to be corrected. In order to correct the error, the faulty input may needs to be identified. With all these considerations, a complete circuit is generated as shown in Figure 22. The truth table for this circuit is Table 17.

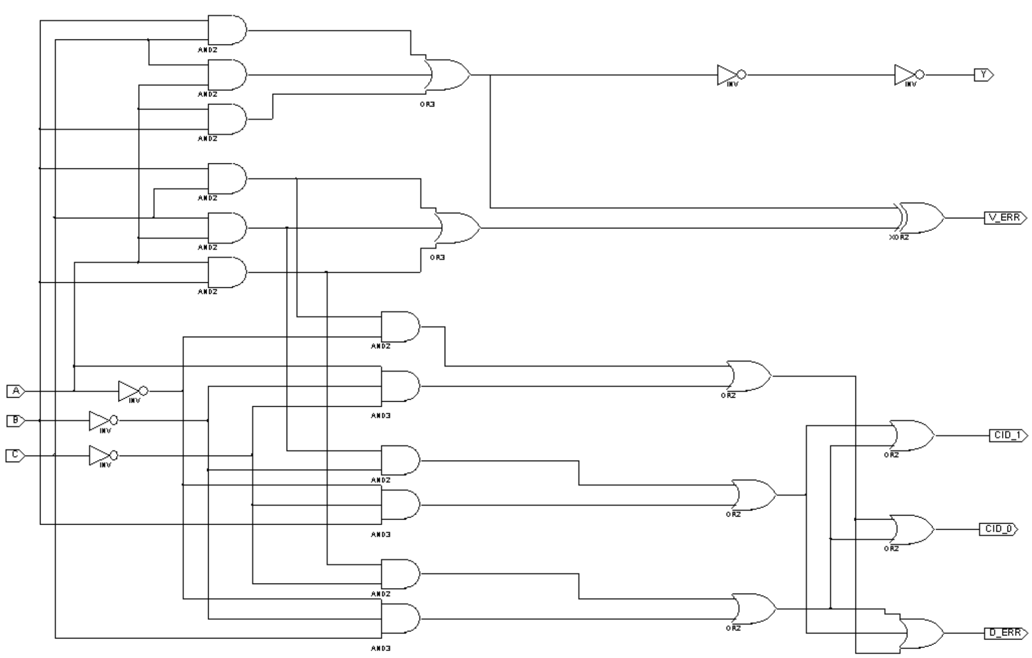


Figure 22. Complete Majority Voter (After Ref. [1].)

A	B	C	Y	V_ERR	D_ERR	CID_1	CID_0
0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	1
0	1	0	0	0	1	1	0
0	1	1	1	0	1	0	1
1	0	0	0	0	1	0	1
1	0	1	1	0	1	1	0
1	1	0	1	0	1	1	1
1	1	1	1	0	0	0	0

Table 17. Truth Table of Complete Majority Voter (From Ref. [1].)

New signals *CID_0* and *CID_1* are used to identify the faulty input, with *CID_0* representing the least significant bit. Using the third row of the table as an example, the voter should be able to capture the error and identify the faulty input pin. The output signal *Y* is a 0 and *D_ERR*, error detection, reports a 1. This indicates that one of input signals is not consistent and the correct input signal is 0. Furthermore, *CID_1* and *CID_0* show 1 and 0, respectively, which means the second processor is faulty. Since *Y* is 0 and the second input is faulty, it can be concluded that input *B* has an error and its value is 1.

The schematic of the complete majority voter built in ISE is shown in Figure 23. All input and output pins are 1-bit wide.

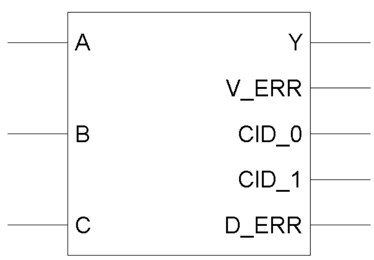


Figure 23. Schematic Symbol of 1-Bit Majority Voter

B. 16-BIT VOTER

Since KDLX has 16-bit output buses, 16-bit voters are needed in order to vote every bit on these buses. A 16-bit voter is simply composed of sixteen 1-bit voters as shown in Figure 24. All voters vote in parallel and produce five output buses for five different signals, *Y*, *V_ERR*, *CID_0*, *CID_1*, and *D_ERR*.

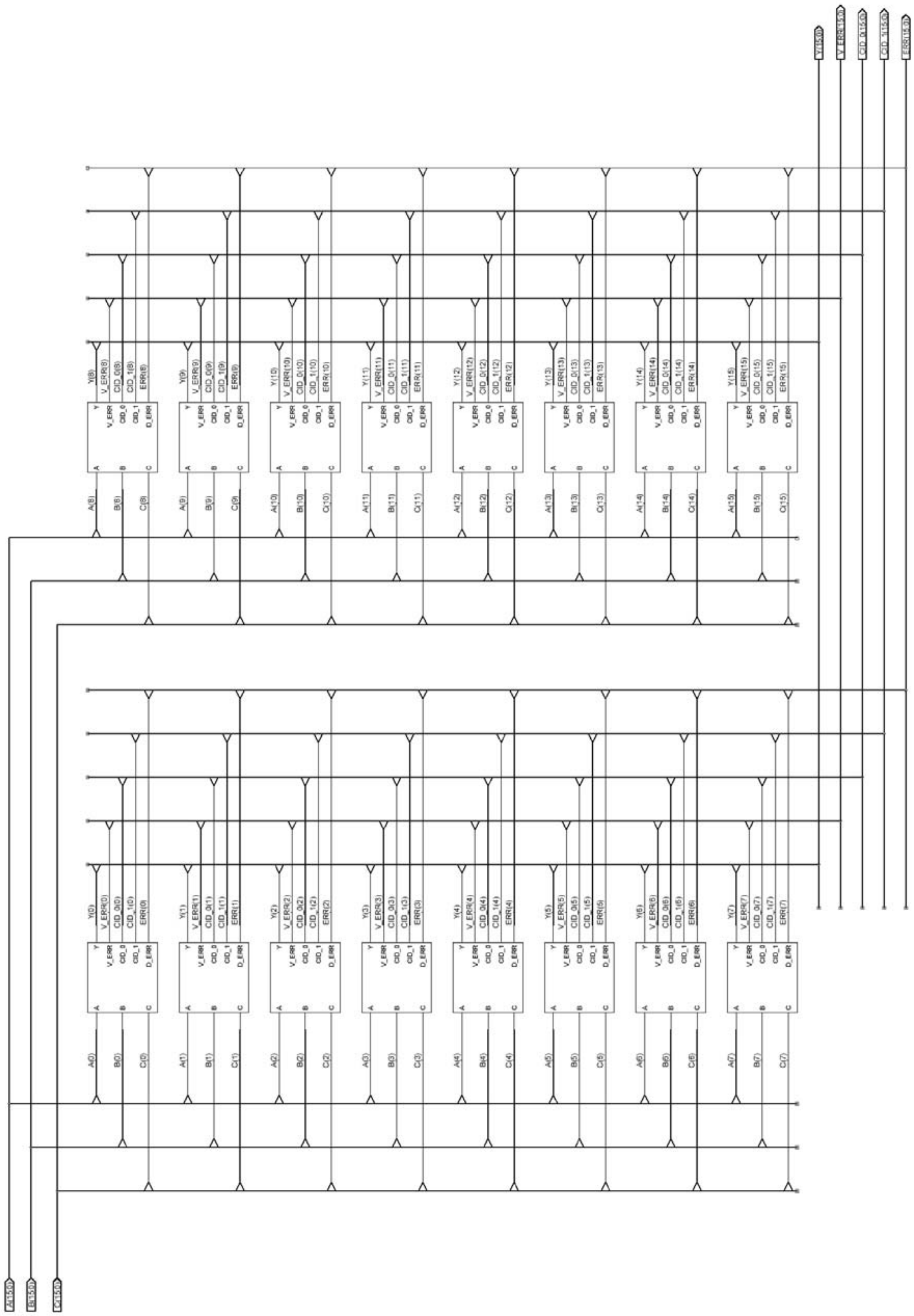


Figure 24. Sixteen 1-Bit Voters

Figure 25 is the schematic symbol used in ISE. The signal name *D_ERR* is changed to *ERR* in order to simplify the notation.

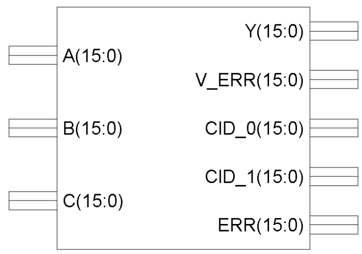


Figure 25. Schematic Symbol of 16-Bit Voter

The voter performs an important role in TMR. It is the device to catch and report errors. The CFTP in space can have an SEU occur anywhere in the FPGA. If the error is caught by the voter, it will be corrected. If the voter votes incorrectly, it will be caught by the voter error detection circuitry. The problem becomes more complicated if an error occurs on the voter error detection. If the voter voted wrong but the error detection did not catch it, the error may propagate through the system and corrupt the data. A new circuit can be added to detect error detection, but adding gates increases the probability of an error and also increases the complexity. Making a voter that has acceptable reliability without increasing the probability of an SEU too much is difficult.

C. TMR ASSEMBLY WITHOUT MEMORY

The concept of the TMR is to triplicate processors and vote all output signals to get correct values. An even number of processors cannot use majority voters. Five or more processors will increase the circuit size dramatically. As described earlier, this increases the probability of having an error by SEU. The usual compromise is to use three processors. The TMR does not increase circuitry too much and its efficiency has been proved in some existing space systems.

In this section, several different architectures will be discussed, which is a good chance to show how things change when different components are used. Important learning points will be provided at the end of this chapter.

1. Schematic and Simulation 1

Figure 26 is the first design of the TMR Assembly for this thesis. Important signals are indicated with arrows. The three big blocks at the left side are KDLX processors. The sequence from top to bottom is processor A, B and C. The 24-bit instruction input buses are *instr_a(23:0)*, *instr_b(23:0)*, and *instr_c(23:0)*, respectively.

Voters are connected at the outputs of the processors. All of the outputs are voted. The first three voters at the top are 1-bit voters for control signals and the other three are 16-bit voters for buses. The voter at the top is the voter for the program read signal. The read signals for the instruction fetch of all three processors are connected to this voter to be voted. The second one is the voter for data read signals and the third one is for data write signals. The three 16-bit voters are for the address, the program counter, and the data bus, respectively.

The outputs of each voter are collected to a bus. Therefore, there are four buses on the right side. One data bus is at the output of the data voter, named *data_p(15:0)*. Since each bus on the right side collects the outputs of six voters, each bus is 51-bits wide.

Because the data memory used in the ISE has separate buses for the input and the output, *data_p(15:0)* is generated as a write bus and *data_m(15:0)* is generated as a read bus. The read and write signals are active low. Thus, inverters are used to enable buffers. Without a buffer for isolation, data injected at *data_m(15:0)* will be voted and sent out to *data_p(15:0)* which may cause a bus conflict.

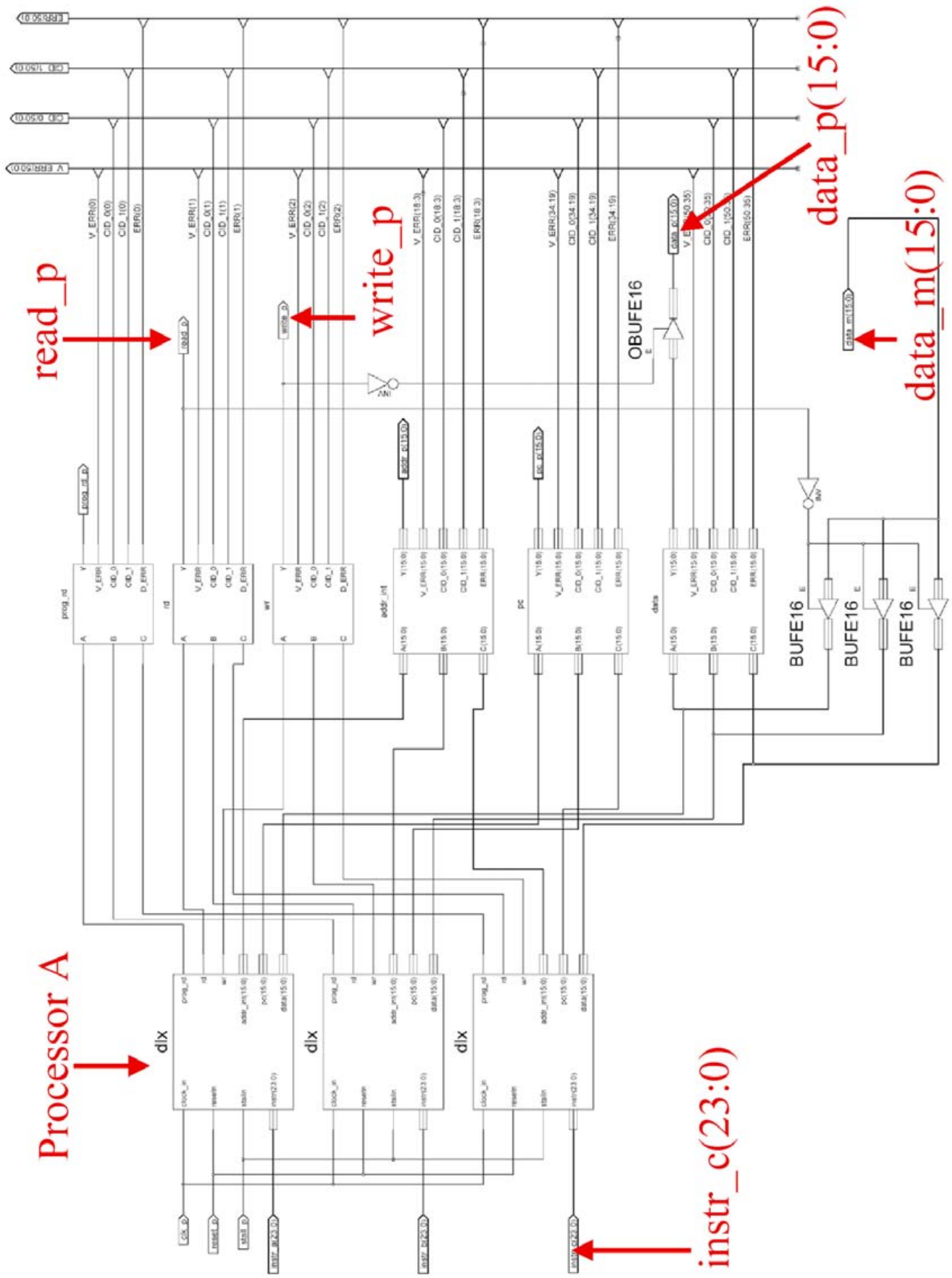


Figure 26. TMR Assembly

This design so far provides everything needed for a TMR processor based on the theory described in section B. The next step was to put it on a simulation test bench and run it. The time constraints are 50 ns for clock high and low time and 10 ns for setup and hold time. Since only one clock is used in this simulation, the time constraints are trivial. The simulation results are shown in Figures 27 and 28.

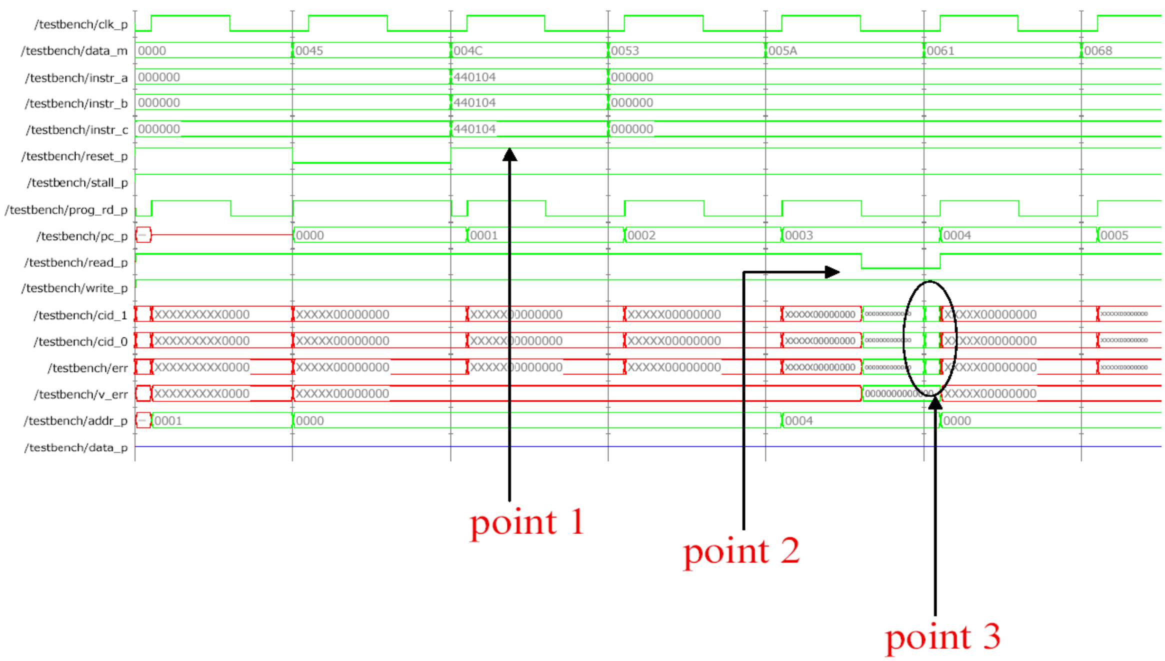


Figure 27. TMR Assembly Simulation 1-1

In Figure 27, the *data_m* bus offers a series of data regardless of whether the instruction needs it or not. All instruction buses (i.e., *instr_a*, *instr_b* and *instr_c*) have the same instruction at the same time. The first instruction, LW R1←Mem(R0+04), is fetched at point 1. It is not executed until point 2. Since the read signal goes low at point 2, it is reasonable to say it loads data $005A_{16}$. Signals *cid_0*, *cid_1* and *err* all report zero because all instructions are consistent. Notice that the data on the *data_m* bus changes while *read_p* is still low. A clipping occurs at point 3.

In Figure 28, another instruction, SW R1→Mem(R0+02), is fetched. Since R1 had already fetched data at point 2, here we expected to see $005A_{16}$ on the *data_p* bus. Unfortunately this is not the case at point 5. The simulation tells us that KDLX has the read signal active low, but it actually reads data at the rising edge. In this simulation, it read 0061_{16} at point 3 not $005A_{16}$, as desired.

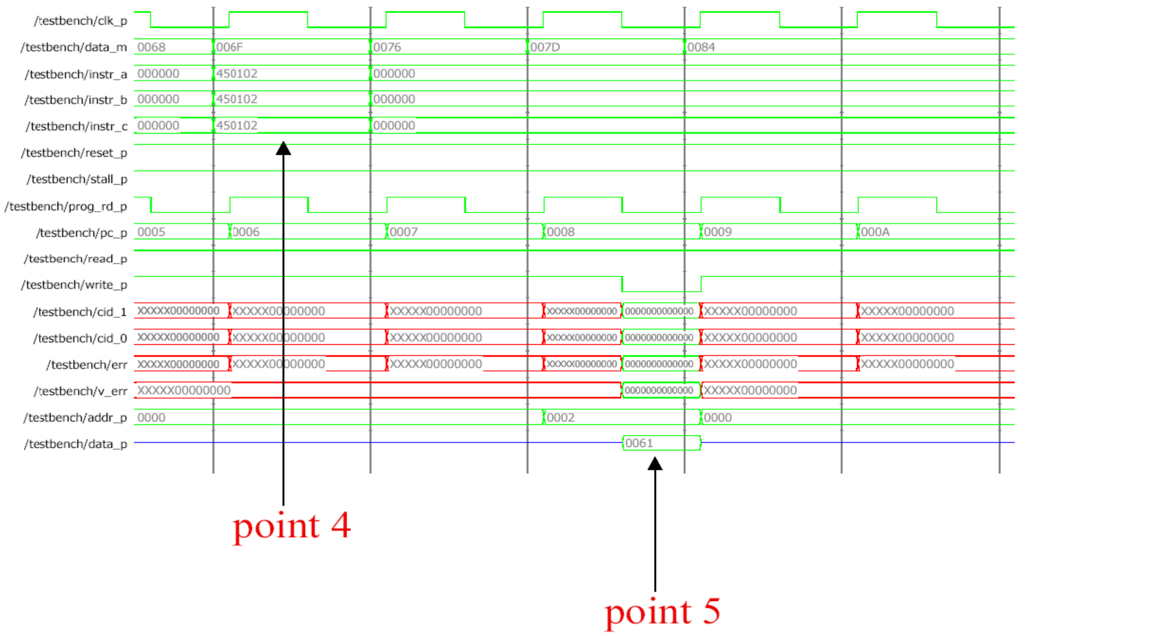


Figure 28. TMR Assembly Simulation 1-2

Since the processor reads at the rising edge, the circuit must be able to keep the data stable to that point. The simulation in Figure 27 shows that $005A_{16}$ stays for most of the duration while *read_p* is low. However, the bus changes to 0061_{16} at the last instant, which is not a desirable situation. Thus the next step is to modify the circuit to make the data stable through the rising edge of *read_p*. Figure 29 is the modified design.

2. Schematic and Simulation 2

A 16-bit latch is added to keep the input data stable. With this latch, the input data only changes when the read signal changes which should in theory, provide a perfect timing match. Simulations of this modified TMR Assembly are shown in Figures 30 and 31.

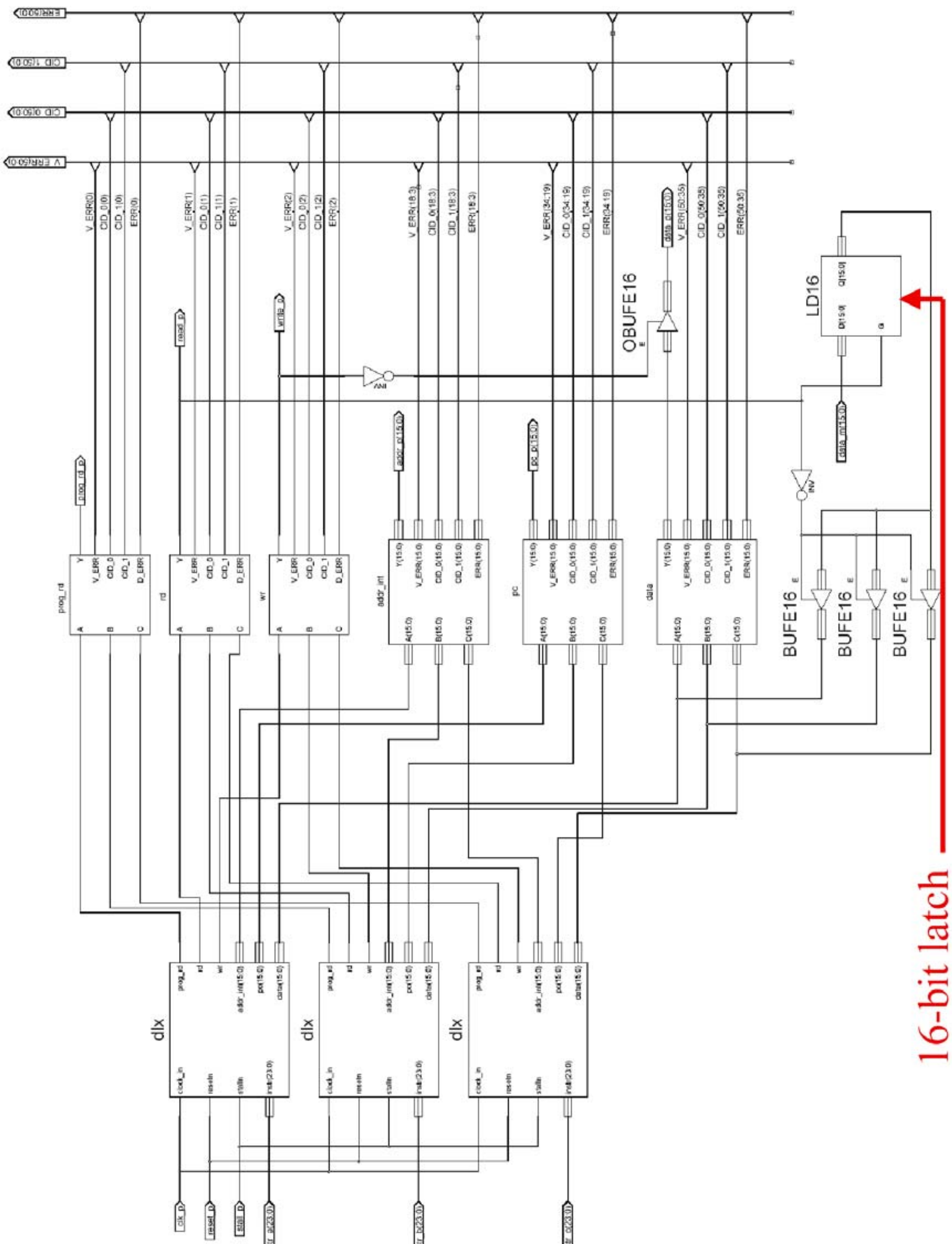


Figure 29. Modified TMR Assembly

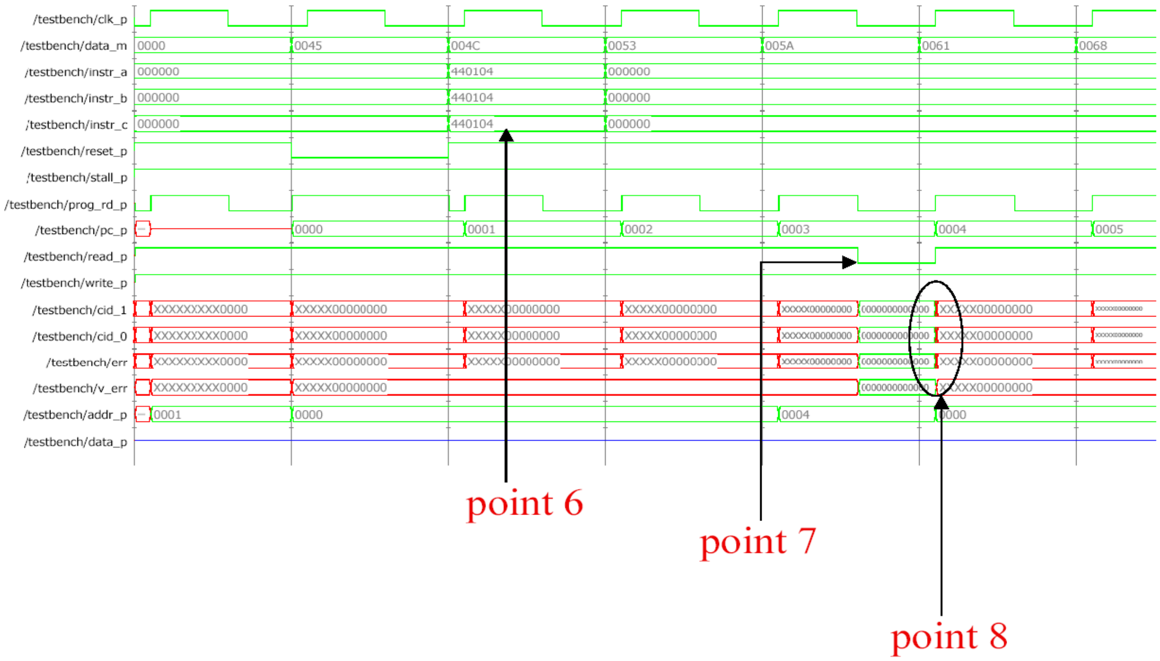


Figure 30. Modified TMR Assembly Simulation 2-1

Points 6 and 7 in Figure 30 are identical to points 1 and 2 of Figure 27. The improvement of the modified TMR Assembly appears at point 8. The latched data is still available at the point where *read_p* goes high and all three processors now read the value $005A_{16}$. The clipping at point 3 in Figure 27 disappears.

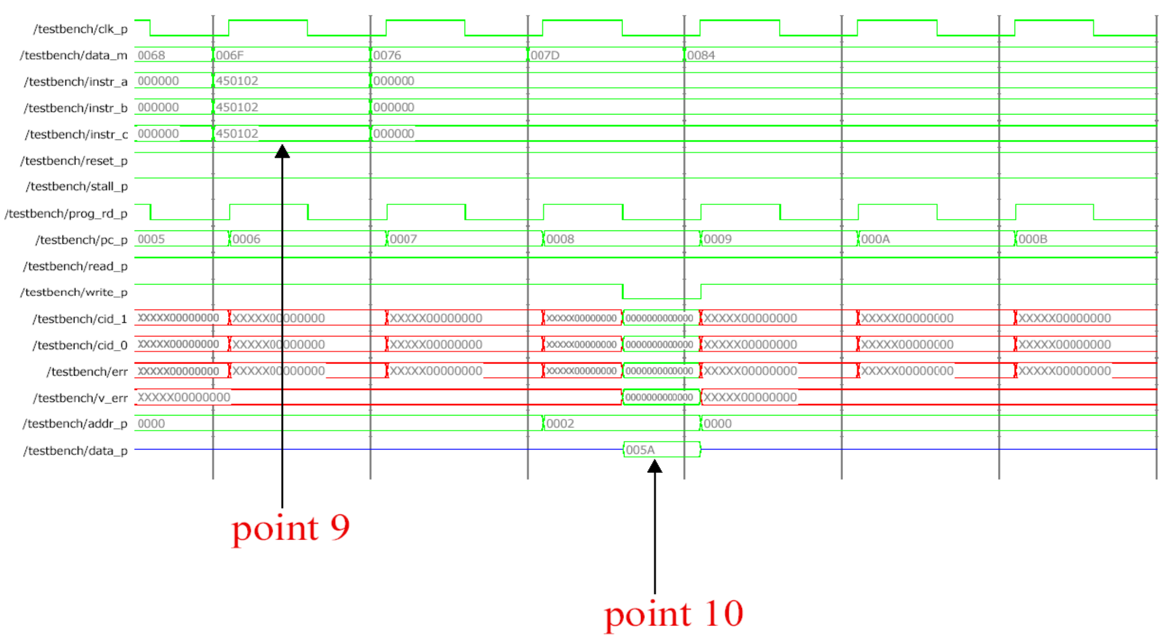


Figure 31. Modified TMR Assembly Simulation 2-2

Figure 31 continues the simulation to store the content of R1 to memory location 02_{16} at point 9. Following the signal *write_p* to point 10, one can find that the data on *data_p* is $005A_{16}$. Signals *cid_1*, *cid_0*, *err* and *v_err* show that no error is reported.

D. TMR ASSEMBLY WITH MEMORIES

Since a working TMR Assembly has been generated, the final step is to hook it up with memories. The latch added in Figure 29 guarantee that the processors will read what they need to read. The schematic symbol of the TMR Assembly is shown in Figure 32. The whole circuit is shown in Figure 33.

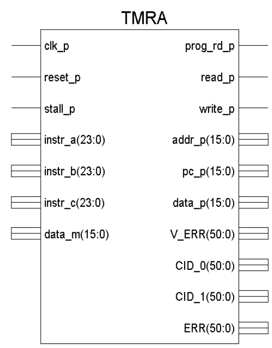


Figure 32. Schematic Symbol of the Modified TMR Assembly

Many of the signals in Figure 33 are for the purpose of monitoring the simulation. As a convention, the memory at the left is the instruction memory and the one at the right is the data memory. Two buffers are used to control the data flow. Data flows into the data memory only when the write signal is low and flows to the *TMRA* only when the read signal is low.

The instruction memory is pre-configured with the following Opcodes: 440301_{16}, 413406_{16}, and 450407_{16}. The first one will load data from memory location 01_{16} to R3. The second one will add an immediate value 06_{16} to R3 and save the result to R4. The final instruction will store the content of R4 to memory location 07_{16}. Figure 34 shows the simulation result.

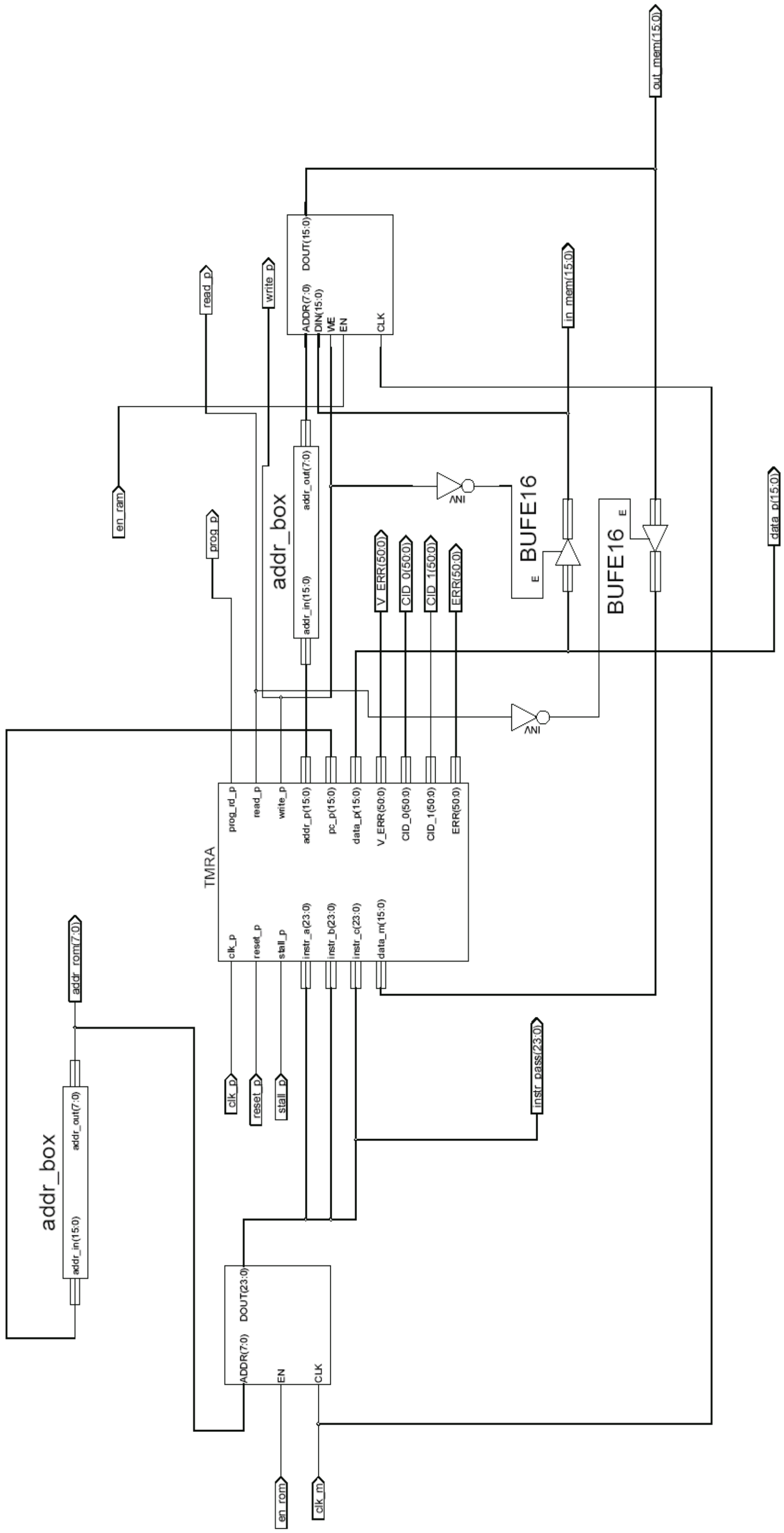


Figure 33. Modified TMR Assembly with Memories

Figure 34. Simulation of Modified TMR Assembly with Memories

Unfortunately, no error was reported but no data was sent out from the data memory. If this design worked correctly, an output value 0009_{16} should be seen when the *TMRA* writes to memory. Obviously, this did not happen when *addr_rom* was $0E_{16}$. Since no timing mismatches occured anywhere, this design was hard to debug. The modified TMR Assembly works fine without memories, so the problem could have been the settings of this test bench. The time constraints of this test bench are listed in Table 18.

Processors		Memories	
Clock High Time	50 ns	Clock High Time	50 ns
Clock Low Time	50 ns	Clock Low Time	50 ns
Input Setup Time	10 ns	Input Setup Time	5 ns
Output Valid Delay	10 ns	Output Valid Delay	5 ns
Time Offset	0 ns	Time Offset	0 ns

Table 18. Time Constraints of Test Bench for Modified TMR Assembly

Memories have less setup time and hold time, so they should be ready before the processors need their data. From this point of view, the test bench seemed not to be the problem. While the problem might have been incompatibility with the choice of memory, the next alternative approach was to try the original TMR Assembly without the data latch as shown in Figure 26. Since all input and output signals are the same with this modified TMR Assembly, the schematic and complete design of the original TMR Assembly are still identical to Figures 32 and 33, respectively. Using the same test bench and simulation as the first design produced the result shown in Figure 35.

This version works. There is almost no timing mismatches and the data clippings are small enough to be ignored. This circuit sends out exactly the right value after the last instruction is executed. When *addr_rom* is at $0E_{16}$, 0009_{16} is sent out from the *TMRA* to the data memory at the lower half clock cycle. The data as seen on *out_mem* has another half clock delay caused by memory. Signals *cid_1*, *cid_0*, *err* and *v_err* verify that no error is reported.

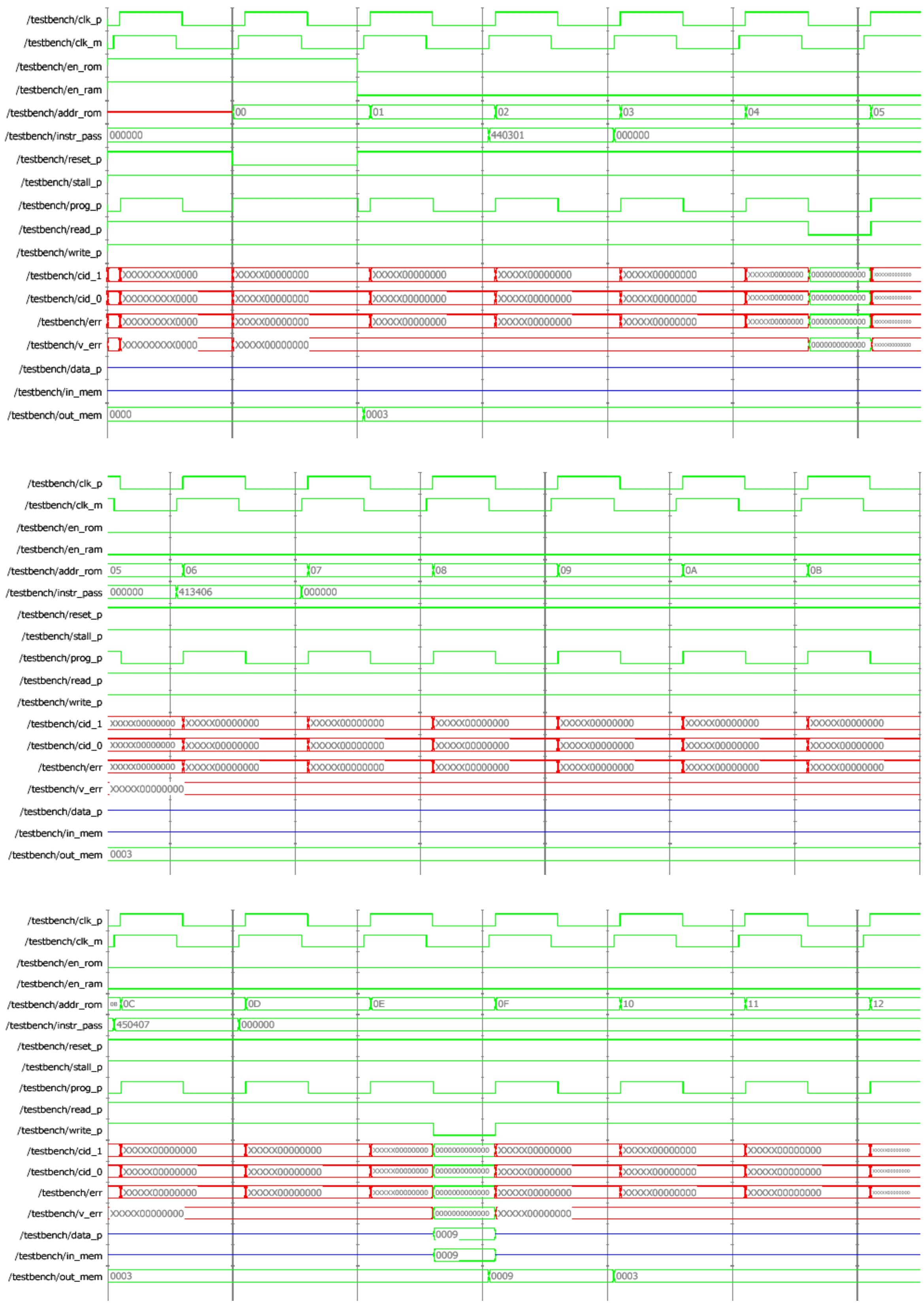


Figure 35. Simulation Result of First TMR Assembly with Memories

The final conclusion is that the latch added in Figure 29 does not help when the *TMRA* is connected with memories. The simulation results in Figures 30 and 31 worked because the input data was set manually. These manual changes set the error regardless of the changing of the read or write signals from the processors. Therefore, a latch was needed in this manual test bench.

When the *TMRA* is connected with memories, the memories will interact with the write signal of the KDLX even though the detailed interaction among them are not visible in the test bench. A latch in the *TMRA* in this design will ruin the timing between the *TMRA* and the data memory. Thus, the simulation result in Figure 34 shows that the *TMRA* is totally unable to communicate with the data memory, while in Figure 36, without the latch the design works.

E. TEST ON FAULT TOLERANCY OF TMR ASSEMBLY

The concept of the TMR Assembly has been described and explained earlier in this chapter. The usage of the voters has been emphasized as well. Since the TMR Assembly has been designed and simulated, the next requirement is to test the fault-tolerant ability. In order to provide errors, three instruction memories are necessary and more signals need to be monitored.

1. Schematic and Simulation

Figure 36 is a complete schematic with all of the components for the fault-tolerant testing. The concept is to change one of the instructions loaded into the *TMRA* and see if the voters can catch the error, correct it, and report it. Since the inconsistent instruction will lead one of the KDLX processors to do something different that the other two, voters should flag the inconsistency and point out the faulty processor, i.e., either *cid_1* or *cid_0* or both should not be zero. Some bits in the error detection bus, *err*, ought to be 1 whenever any error exists. If all these signals work properly, the *TMRA* will be able to catch an error and trigger an interrupt routine.

Three instruction memories, *ROM A*, *ROM B* and *ROM C*, are pre-configured with three different instruction maps. The data memory at the right side, *RAM*, has non-repeated value in its memory locations. This makes the data in the simulation more easily identified since each memory address holds a unique value. Memory maps for the *ROM*s and *RAM* are displayed in Table 19.

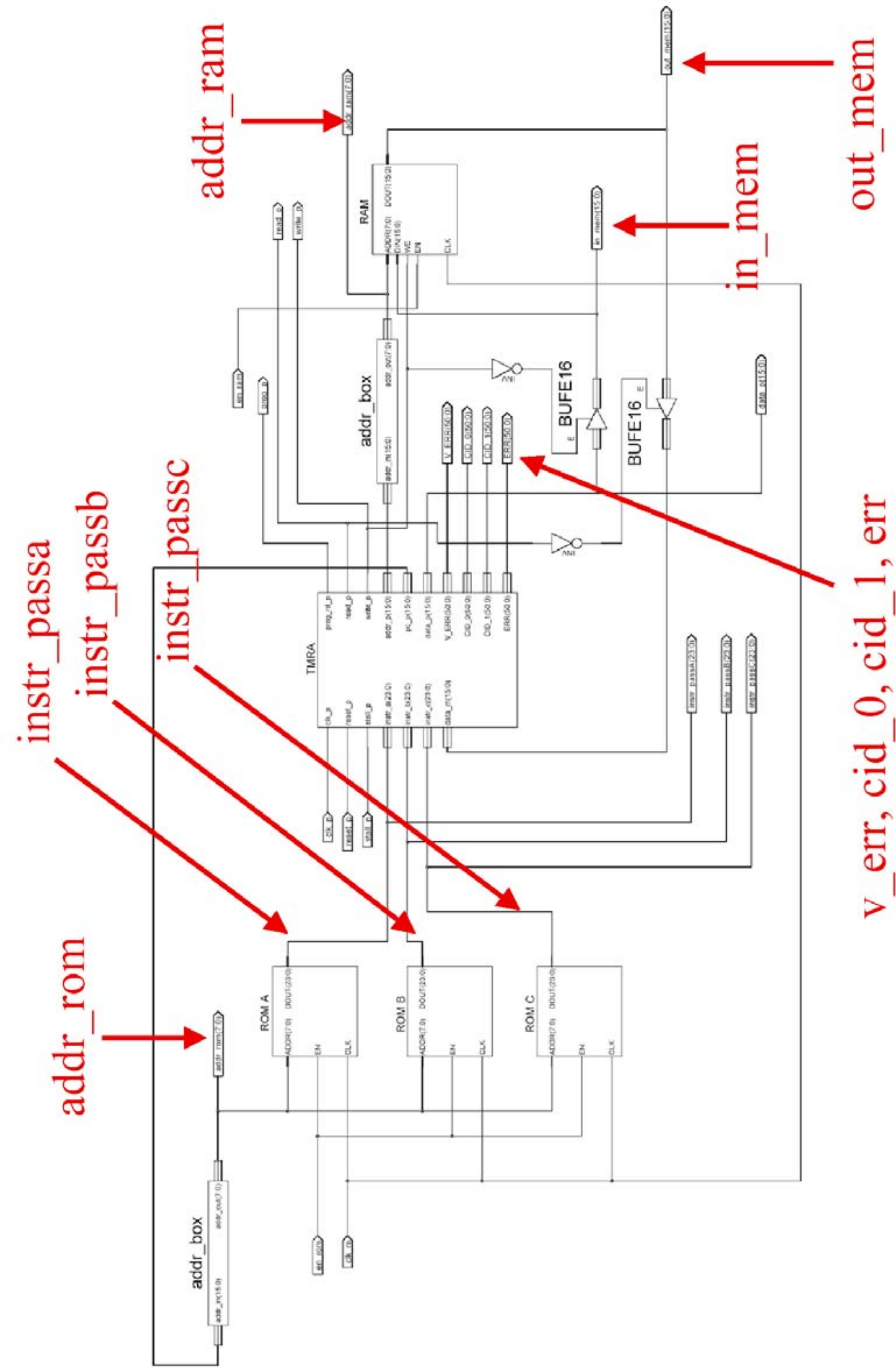


Figure 36.   Schematic for Fault-Tolerant Testing

ROM A		ROM B		ROM C		RAM	
00	000000	00	000000	00	000000	00	20
01	000000	01	000000	01	000000	01	21
02	000000	02	000000	02	000000	02	22
03	44010A	03	44010A	03	44010A	03	23
04	440203	04	44020B	04	44020B	04	24
05	44030C	05	440A0C	05	44030C	05	25
06	44040D	06	44040D	06	350911	06	26
07	000000	07	000000	07	000000	07	27
08	000000	08	000000	08	000000	08	28
09	000000	09	000000	09	000000	09	29
0A	000000	0A	000000	0A	000000	0A	2A
0B	450106	0B	450103	0B	450103	0B	2B
0C	450208	0C	450207	0C	450208	0C	2C
0D	450309	0D	450309	0D	450302	0D	2D
0E	450410	0E	450410	0E	450410	0E	2E

Table 19. Instruction And Data Memory Maps

The inconsistent instructions are grayed out in Table 19. The TMR Assembly simulation is shown in Figures 37, 38, and 39.

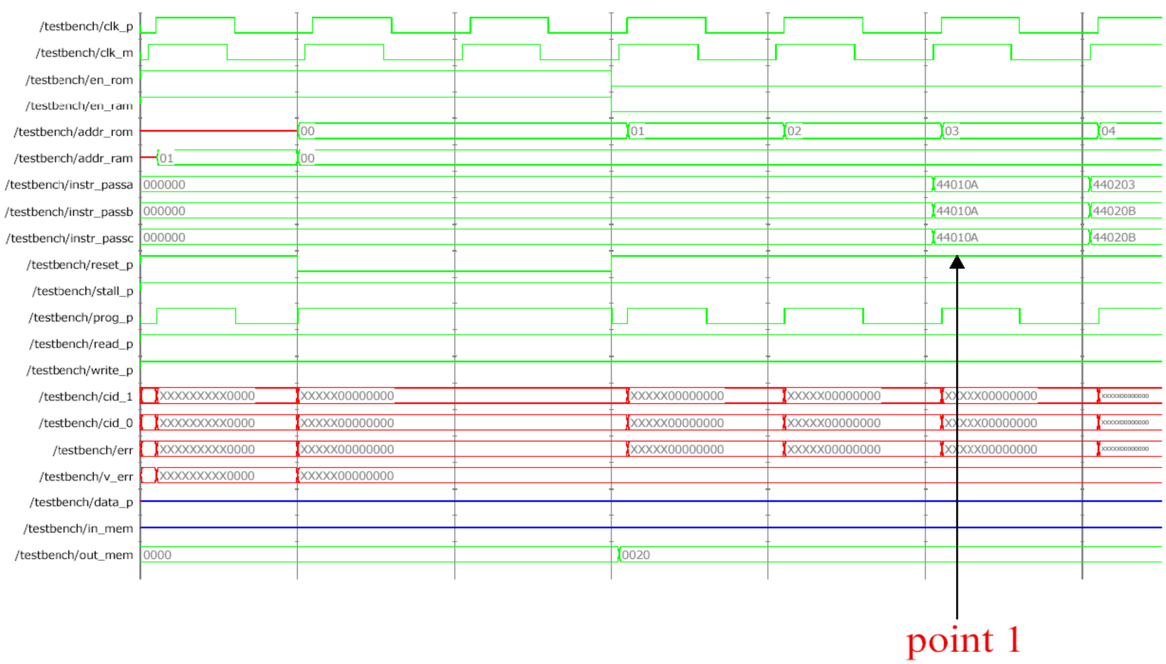


Figure 37. Simulation of Fault-Tolerant Testing

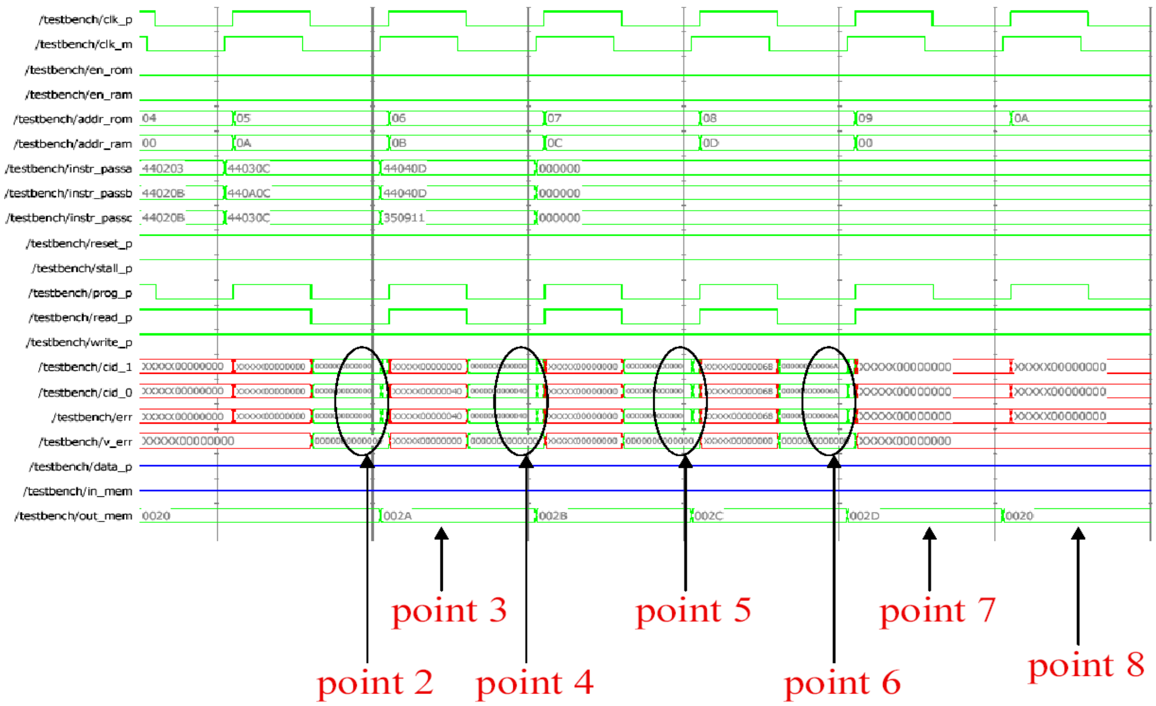


Figure 38. Simulation of Fault-Tolerant Testing (continued)

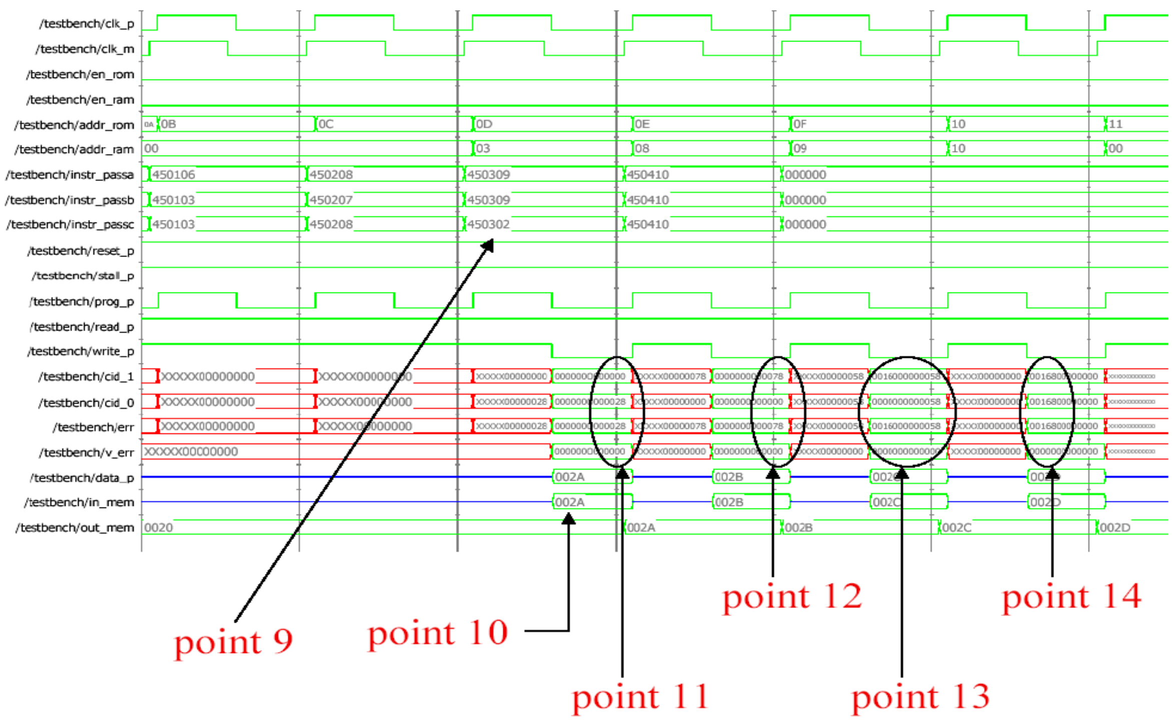


Figure 39. Simulation of Fault-Tolerant Testing (continued)

In Figure 37, when the signal *reset_p* goes from low to high, the *TMRA* starts fetching instructions. Notice the signal *out_mem* shows 20_{16} which is the first value at address 00_{16}. The instructions at address 03_{16} of the *ROM*s are fetched at point 1. Following that, three more instructions are fetched in sequence. The first instruction, $44010A_{16}$, is executed at point 2 in Figure 38 while *addr_rom* is 05_{16} and *addr_ram* is $0A_{16}$. The *addr_rom* contains the address of the instruction being fetched, i.e., 05_{16}. The *addr_ram* contains the address that the first instruction, i.e., $44010A_{16}$, is using to access *RAM*. In this case, $0A_{16}$ is the correct address for this first instruction.

From this point in the simulation, inconsistencies have been introduced in the instruction memory. The bit distribution of the bus needs to be introduced in the next section before the simulation analysis is presented.

2. Bit Distribution

Recall the schematic in Figure 26. Four signals (i.e., *V_ERR*, *CID_0*, *CID_1*, and *ERR*) are collected into four different buses and each bus is 51-bit wide. Since one 51-bit bus consists of outputs from 6 different voters, each voter has a range in the bus distribution. By looking at the bits in the distribution, one can tell which signal on which processor is wrong. The bit distribution for *CID_1*, *CID_0*, and *ERR* is shown in Figure 40.

CID_1(50:0) & *CID_0(50:0)* & *ERR(50:0)* Bit Distribution					
data(15:0)	pc(15:0)	addr_int(15:0)	wr	rd	prog_rd
50 35	34 19	18 3	2	1	0

Figure 40. Bit Distribution of *CID_1*, *CID_0* and *ERR* Buses

In Figure 40, the bit distributions of all three buses are identical. For example, a 1 at bit 20 of the *ERR* bus means that one of the KDLX processors has an error in its program counter. At the same time, bit 20 of the *CID_1* and *CID_0* buses will point out the faulty processor.

3. Simulation Analysis

The three instructions fetched by the *TMRA* at point 1 in Figure 37 are identical so no error is reported at point 2. Since there is no error in any one of the processors, the *cid_1* and *cid_0* buses will not identify any processor. It was mentioned that the memory needs a half clock cycle to send out data once it receives signals. That is why the first data is not on the *out_mem* bus until point 3. It can be verified that the *TMRA* is loading a correct value.

When the instructions become inconsistent, the error detection signal is no longer zero. Meanwhile, the *cid_1* and *cid_0* locate the faulty processor. This can be checked from point 4 to 6. Figure 41 is the bit distribution of the error detection signals for the first Opcode, $44010A_{16}$. The hexadecimal number in the simulation is translated to a binary number when doing this data analysis.

Bit	50	49……………………………11	10	9	8	7	6	5	4	3	2	1	0
err	0	0……………………………0	0	0	0	0	1	0	0	0	0	0	0

error

Figure 41. *ERR* Analysis for the First Opcode

It is obvious that the sixth bit is inconsistent in three processors. In order to verify the error, the signals *cid_1* and *cid_0* should be analyzed. Converting the hex numbers in the simulation to binary numbers and comparing the bit distribution with Figure 40 indicates that (Figure 42) the inconsistent bit is on the address bus and Processor A is the faulty processor. Recall from Table 17 that *cid_1* is the most significant bit, so 01_2 stands for the first processor (i.e., Processor A). It is true that the instruction at address 01_{16} in *ROM A* is the actual location of the error, but since this instruction is only sent to the first processor in the *TMRA*, Processor A is identified as faulty.

prog_rd

wr

rd

addr_int(15:0)

Bit	50	49……………………………11	10	9	8	7	6	5	4	3	2	1	0
cid_1	0	0……………………………0	0	0	0	0	0	0	0	0	0	0	0
cid_0	0	0……………………………0	0	0	0	0	1	0	0	0	0	0	0

Processor A

Figure 42. *CID_1* and *CID_0* Analysis for the First Opcode

The reason that the error is at bit 6 is because that is the only location where the output bits are not consistent in the three processors. Figure 43 shows the situation.

	Hex	Binary
Correct Address	0B	0000 1011
Wrong Address	03	0000 0011

bit 3

Figure 43. Address Comparison for the First Opcode

The second Opcode in *ROM B* has an incorrect destination register. Since there are no output signals on KDLX for the destination register, point 4 in Figure 38 reports no error, even though this wrong Opcode loads a correct data into the wrong register. The contents of R3 are now inconsistent between the three processors as are the contents of R10. This kind of error will only be found when the content of the faulty register is used. Point 9 in Figure 39 stores the contents of R3 to memory location 09_{16}. It is known that the data in R3 is wrong in Processor B, but the Opcode difference at point 9 also means that the memory address of Processor C is wrong. Figure 44 shows the simulation result for point 13 in Figure 39. Six inconsistent bits were caught.

Bit	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35
err	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0
Hex		0			0				1				6			

Bit	34……………………………………8	7	6	5	4	3	2	1	0
err	0……………………………………...0	0	1	0	1	1	0	0	0
Hex	0…………………0		5				8		

Figure 44. *ERR* Analysis at Point 13

The contents of R3 in Processor B are zero, but in Processors A and C they are $2C_{16}$. For *cid_1* and *cid_0*, it is expected that the data portion in the bit distribution indicates that Processor B is wrong. Figure 45 shows the inconsistent bits between the correct and wrong data.

	Hex	Binary
R3 of A and C (correct)	2C	0010 1100
R3 of B (wrong)	00	0000 0000

bit 5
bit 3
bit 2

Figure 45. Data comparison for R3

The bit distribution of *cid_1* and *cid_0* should put $002C_{16}$ in the data portion and indicate all inconsistencies caused by Processor B. Figure 46 illustrates that it does.

Bit	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35
cid_1	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0
cid_0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0 0 2 C

Processor B

Figure 46. *CID_1* and *CID_0* Data Portion Analysis at Point 13

In addition, the address differences from Processor C at point 9 should also be indicated by *cid_1* and *cid_0*. This is shown in Figure 47.

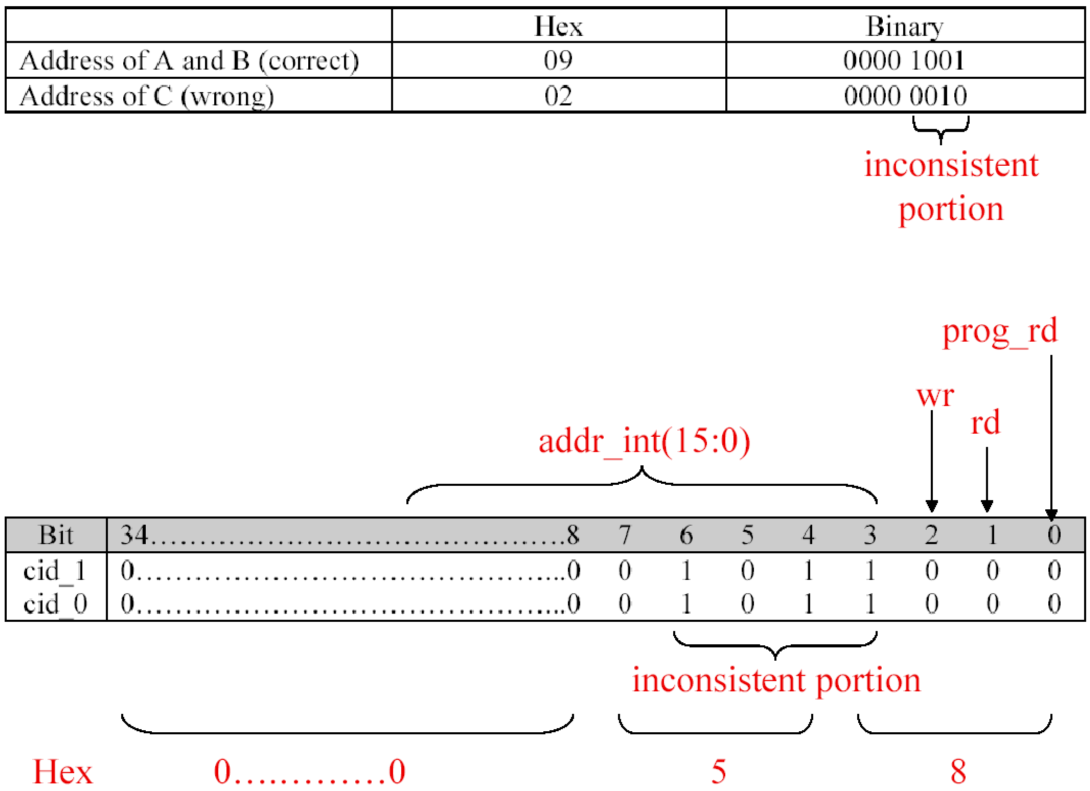


Figure 47. *CID_1* and *CID_0* Address Portion Analysis at Point 13

Notice that both *cid_1* and *cid_0* at point 13 have hex number 58. The inconsistent bits of the addresses are reflected correctly in the bit distribution. The Processor C is identified as the faulty one that gives a different address to the voter than the others. This proves that *cid_1*, *cid_0* and *err* signals can deal with these kinds of multiple errors and still report flawlessly.

Following the same procedure to analyze data on buses, one should be able to realize how the voter works and the way to utilize these signals for an interrupt routine. The rest of the simulation also performs correctly. The Opcode at address 06_{16} of *ROM C* is a disaster since there is no such instruction. Based on the experience just learned, this kind of error will still be corrected. The inconsistency of register contents will be corrected the next time they are used and the wrong addresses will not affect anything as long as the other two addresses are correct. Correct data will still be fetched at point 7 in

Figure 38. The memory output data bus switches back to 0020_{16} at point 8. Next, three store instructions are fetched in series. The first data written to memory shows up at point 10. Simple address inconsistencies at point 11 and 12 are easily analyzed. Errors at point 14 are detected, even though all three Opcodes, 450410_{16}, are the same. That is because the data loaded into R4 earlier was different and the error occurs only when R4 is routed to the output.

F. IMPORTANT SIMULATION CONCEPTS REVIEW

Simulation results are used a lot in this chapter to explain the operation of the TMR. Fundamental ideas on how to construct a test bench and how to analyze results have been established. Due to the different properties of the different components, a design may not work when additional components are connected. Generating a good test bench is not easy since most timing problems are unpredictable. Some important knowledge for simulation needs to be introduced in order to help shrink the time for invention.

1. KDLX Was Designed to Work with Asynchronous Memory

In a personal conversation with Dr. Kenny Clark, I learned that the KDLX was designed for an asynchronous memory. Although it will work with a synchronous instruction memory, an asynchronous memory is recommended since one should assume that the instruction memory and the data memory are in the same physical memory. Always provide some different time constraints between KDLX and memories when generating a test bench.

2. Start with A Simple Test Bench First

Trying to test everything on a new design is a bad idea. Too many signals need to be tracked and multiple errors are hard to debug. It is a good idea to start with a simple test bench which only tests a small part of the design. Revise the test bench to become more complicated step by step. It is also good to individually test every component generated before constructing a top-level design.

3. Test Bench Is Optimized for the Current Design

As introduced earlier, the simulations have different time constraints. A test bench is used to check to see if a design works under reasonable assumptions. Circuits will be modified many times until the full design is complete. It is hard to specify the requirement for a test bench before a circuit is actually built, so it is almost impossible to have an ideal test bench for a full design and every single component. In addition, a test bench that works on the top-level design may not fit to a single component. Timing mismatches always change with different wiring.

4. Keep Old Designs

It was shown in the TMR Assembly schematic that sometimes an old design is the real useful one. Incorrect settings for a test bench can mislead a designer to make a wrong decision and a modified design can become useless when other components are connected. Features on different components sometimes will balance out timing mismatches between them. Going over previous designs helps a designer to retrieve original thoughts and keeping those files available is important.

5. Working on the Copy of Source

Based on personal experience, it is good to add a copy of a tested circuit into a large design rather than adding the original. This not only keeps the integrity of the original file but also makes it easy to review. Without making a copy, the new design will associate with the original design. Any modification in the new design directly affects the original file. Therefore, it will be impossible to keep the original source file.

Keeping the integrity of each circuit is also important. People always want to see and test the fundamental design before they jump into the full design. For example, a new designer may want to understand voters before realizing the TMR Assembly. Making all correct and incorrect circuits into one project is convenient for a designer, but this does not help other people to understand. By the way, having all sources in one project lacks independency while doing individual tests.

There is no question that making a copy of a source file definitely increases the size of folder and requires more time to manage individual projects. The big benefit of this is that a designer can always have original designs in hand as well as all projects left are tested and ready to go. A new designer thus has a chance to see the function of a voter before sinking into the confusion of the complete TMR Assembly. Since another new project will be generated once a project has failed, a design like the TMR Assembly may have different versions. The useful version contains only useful schematics and test benches. From this point of view, all projects left are not only useful but also have few or no junk sources inside.

Since hard drive space nowadays is huge and cheap, working on a copy file not only gives people a chance to review but also make all projects look clean and easy to understand.

G. CHAPTER SUMMARY

This chapter introduced the kernel of the full TMR design, i.e., the TMR Assembly. Understanding how voters catch errors and how to analyze simulation results is the main point in this chapter. Many explanations of simulation results are provided in order to help one realize the spirit of the TMR design. After reading so many simulations, one should have a feeling on how to use and generate a test bench. A quick review on simulation concepts is put at the end of this chapter after one has studied some simulations and before he/she jumps into a more complex design.

Other components associated with the TMR Assembly like the *Reconciler*, *Interrupt* and *Error Syndrome Storage Device (ESSD)* will be explained in following chapters. The *Reconciler* is an interface between KDLX and memory; the *Interrupt* is the one generating ISR; the *ESSD* is responsible for storing error syndromes whenever an error occurs.

VI. RECONCILER

Due to the different memory architectures between KDLX and CFTP as described in Chapter IV, the *Reconciler* is used to satisfy the timing requirements on both sides and properly route the data. Since KDLX can only access memory via load and store instructions, the *Reconciler* only needs to monitor the read and write signals from KDLX and direct the data to the correct destinations.

In this chapter, no error detection or correction will be discussed since the *Reconciler* is not responsible for this. The TMR Assembly is responsible for error detection. Error correction is done by the *Interrupt* and the voters in the TMR Assembly. Storing the error syndromes is the job of the *Error Syndrome Storage Device (ESSD)*.

A. CONSTRUCTION AND FUNCTION

Only one physical memory is available in the CFTP. In order to make this one memory act as the both instruction memory and data memory in each KDLX clock cycle, the physical memory has to run at twice the speed of KDLX. For the same reason the *Reconciler* has also to run twice as fast as KDLX. For each KDLX clock cycle, one address bus access and one data bus access for instructions needs to be available. Meanwhile, one address bus and one data bus access for data also needs to be available. To fetch an instruction and do a data read or write, the *Reconciler* has to act as an instruction memory in the first half of the KDLX clock cycle and act as a data memory in the second half of the KDLX clock cycle. This function is illustrated in Figure 48.

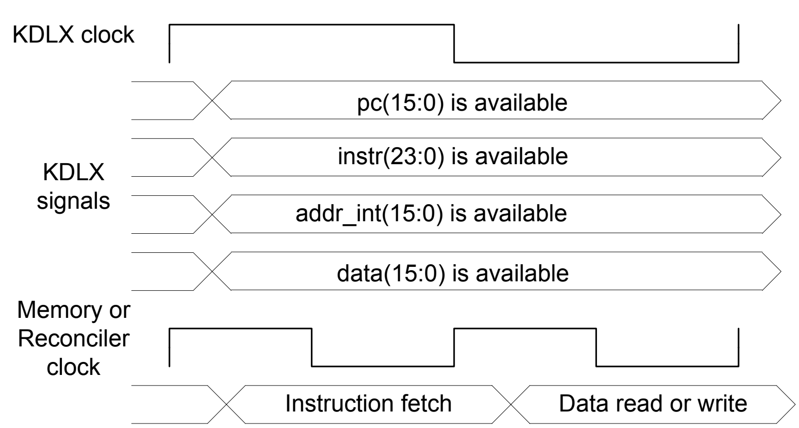


Figure 48. Illustration of *Reconciler* Function

The *Reconciler* is composed of a state machine coded in VHDL and is presented completely in Appendix C, section A. The state machine contains five states: one starting point, two for normal operations, one for read, and one for write. This function can be seen clearly in Figure 49.

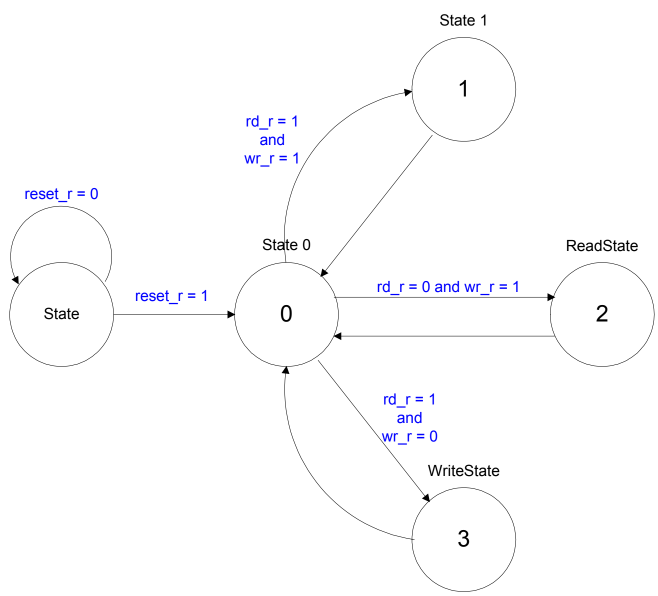


Figure 49. State Machine of the *Reconciler*

The name of the state is on the top of each circle except for the initial state named *State*. The number in each state is the state number designed for tracking purposes in the simulation. The two normal operations, *State0* and *State1*, are identical and are for fetching instruction. Without reading or writing, these two states just pass the program counter to memory, fetch the instruction and send it back to the KDLX. At this time, the memory acts as a ROM and its data-input bus is in a high impedance state. Since only the instruction bus is used, the data bus of the KDXL is also in a high impedance state. State *State1* is a duplication of *State0* so the state machine can be revised to stay at *State0* when neither *rd_r* nor *wr_r* is 0. The reason for using two states is to provide tracking in simulation. Since the *Reconciler* runs twice as fast as the KDLX, reading and writing actions only occur at *State0*. Without the separation into two states, it is hard to tell if a read or write occurs at the proper state.

When *rd_r* is 0 and *wr_r* is 1, the state machine goes to the *ReadState*. KDLX wants to read data from memory so the *Reconciler* will pass a high write signal to the memory and direct data from the memory to KDLX. When *rd_r* is 1 and *wr_r* is 0, the *Reconciler* knows that KDLX wants to write data to the memory, so it passes a low write signal to memory and directs data from KDLX to memory.

The initial state, *State*, is not used until the next reset. It is null and there are no actions in this state. Without this state, the state machine would use *State0* as the initial state and start at *State1* after reset.

B. SCHEMATIC AND SIMULATION OF RECONCILER ONLY

Converting a VHDL code to a schematic symbol is a useful function in the ISE software. The schematic symbol of *Reconciler* is shown in Figure 50.

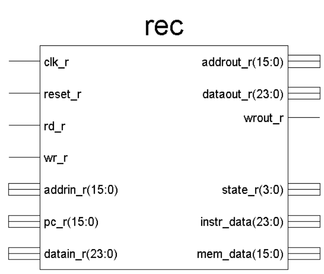


Figure 50. Schematic Symbol of *Reconciler*

Simulation of the *Reconciler* itself is quite simple. Since it is basically a state machine, a state will either stay at current state or jump to a new state every clock cycle. Figure 51 is the simulation result.

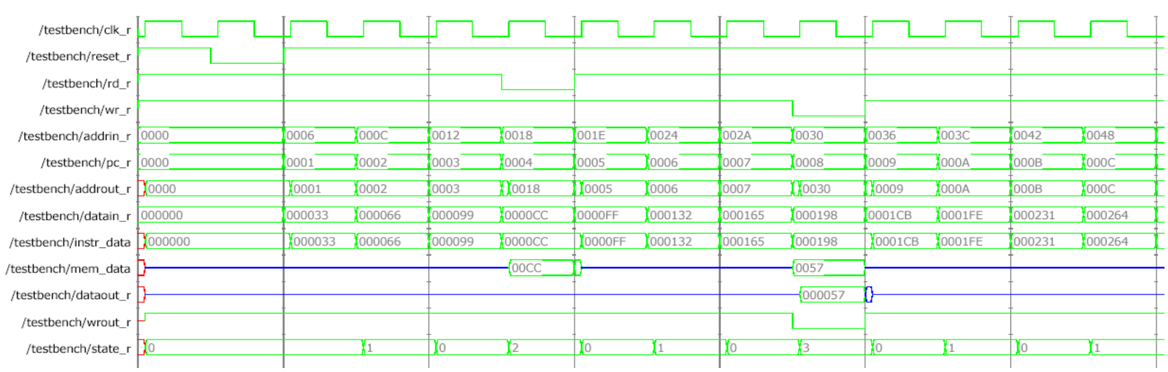


Figure 51. Simulation Result of the *Reconciler*

The signal at the bottom in Figure 51 is the state number used to track which state is active. The state machine starts at *State0* after reset. The signal *addrout_r* is the bus connected with the memory address bus. It sends out either *pc_r* or *addrin_r* depending on whether the system is doing an instruction fetch or a data read/write. In *State0* and *State1*, the *addrout_r* is always the same value as *pc_r*. The memory data output bus connects with the signal *datain_r* on *Reconciler* and sends out either an instruction or a data value. When *rd_r* is low, data on *datin_r* will be forwarded to *mem_data* which connects to the data bus of KDLX. When *wr_r* is low, the state machine goes to the *WriteState*. At this state, data from KDLX is available on *mem_data* and *Reconciler* will direct this data to *dataout_r* which connects to the data input bus of memory.

The *instr_data* is never in a high impedance state regardless of whether the data on *datain_r* is an instruction or not. The reason is to make an instruction stay available until the next KDLX clock cycle. Even during *ReadState* and *WriteState*, the next instruction for the KDLX is alive on the instruction bus. Remember that the *Reconciler* is twice as fast as the KDLX. If the next instruction is only available for the first half of the KDLX clock cycle, it will not be fetched at the rising edge of the next KDLX clock. This concept will be described again when the *Reconciler* is hooked-up with a KDLX processor.

C. SCHEMATIC AND SIMULATION OF RECONCILER WITH KDLX

The last step for testing the *Reconciler* is to simulate it with a KDLX. The schematic of this part of the design is shown in Figure 52.

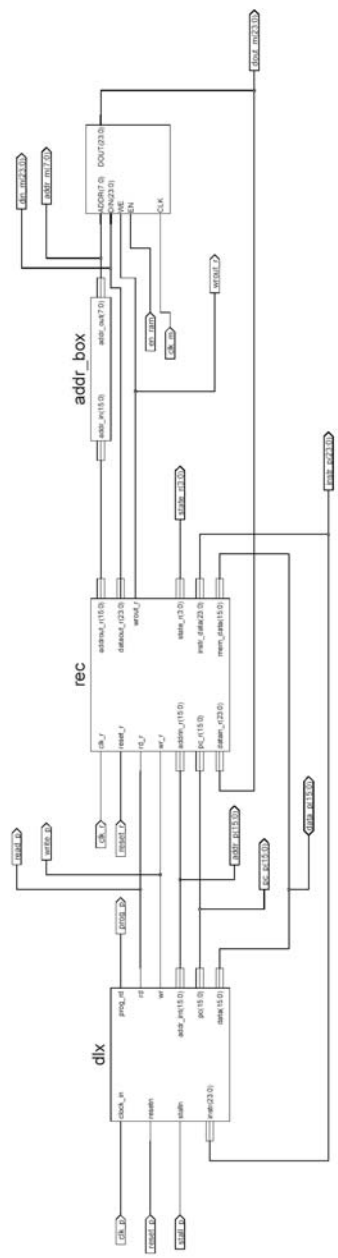


Figure 52. Schematic of *Reconciler* with KDLX and Memory

The memory offered in the ISE software is not a real Von Neumann architecture. Instead of having one bi-directional data bus, the *Reconciler* is designed to have two separated buses for data, *datain_r(23:0)* and *dataout_r(23:0)*. The *mem_data(15:0)* on *Reconciler* is bi-directional in order to transfer data back and forth with the KDLX.

The simulation for this circuit is done with a series of load and store instructions in order to see if the *Reconciler* can handle both instructions and data correctly. Figure 53 is the first part of the simulation result.

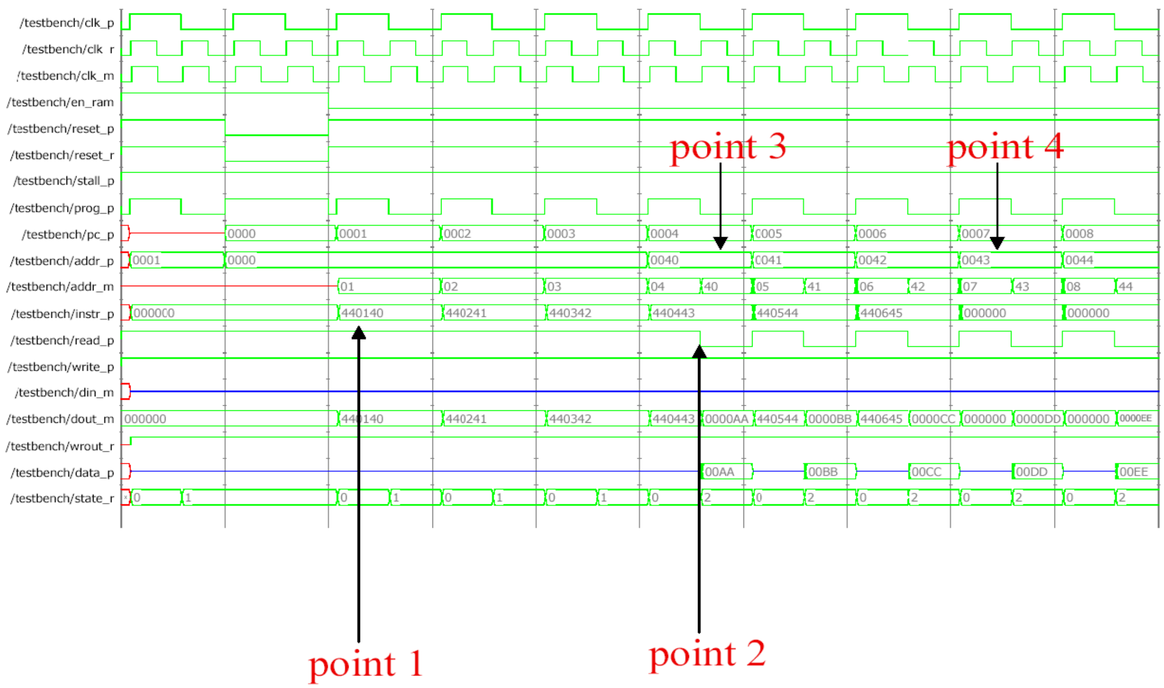


Figure 53. The First Part of the Simulation Result for *Reconciler*

In Figure 53, the first instruction in memory is fetched at point 1 when *pc_p* was sent. It can be seen clearly from the status of *state_r* that the *Reconciler* is in double speed. At point 2, the Opcode 440140_{16} is executed and wants to load data into R1. At the same time, the KDLX is going to fetch the Opcode 440443_{16}. The address of data for the first instruction is available at point 3 in this time interval. Therefore, the signal *addr_m* fetches *pc_p* at the first half of the KDLX clock cycle and fetches *addr_p* at the second half of the KDLX clock cycle. The data at memory location 0040_{16} thus is sent from memory to KDLX when *state_r* is 2. Notice that at this time interval Opcode 440443_{16} is available on the bus until the next KDLX clock. This is important since KDLX is triggered at the rising edge of the clock. Failure to keep an instruction until the next rising edge will mean that the KDLX will not be able to fetch this instruction and the memory location for data will not appear at point 4. This is why the instruction bus is not set to a high impedance state at the *ReadState* and *WriteState* in the *Reconciler*. The rest of this simulation in Appendix A, section H does a series of writes followed by a series of reads in order to check if the *Reconciler* functions properly.

D. TIMING CONCERNS

An added complexity for this simulation is the fact that it has three different clocks. To make this simulation work, the time constraints of the test bench have to be set properly. The sequence of execution in this circuit is that the KDLX sends its program counter to the *Reconciler* first. Then *Reconciler* forwards this address to the memory. Next, the memory selects the instruction and sends it to the *Reconciler*. Finally, the *Reconciler* forwards this instruction to the KDLX. This is a simple example of how KDLX fetches an instruction.

In order to successfully fetch an instruction, the KDLX has to have its program counter ready before the *Reconciler* needs it. The *Reconciler* has to have the address set before the memory is ready to receive it. Considering setup time and hold time for each clock, the relationship among these three clocks is shown in Figure 54.

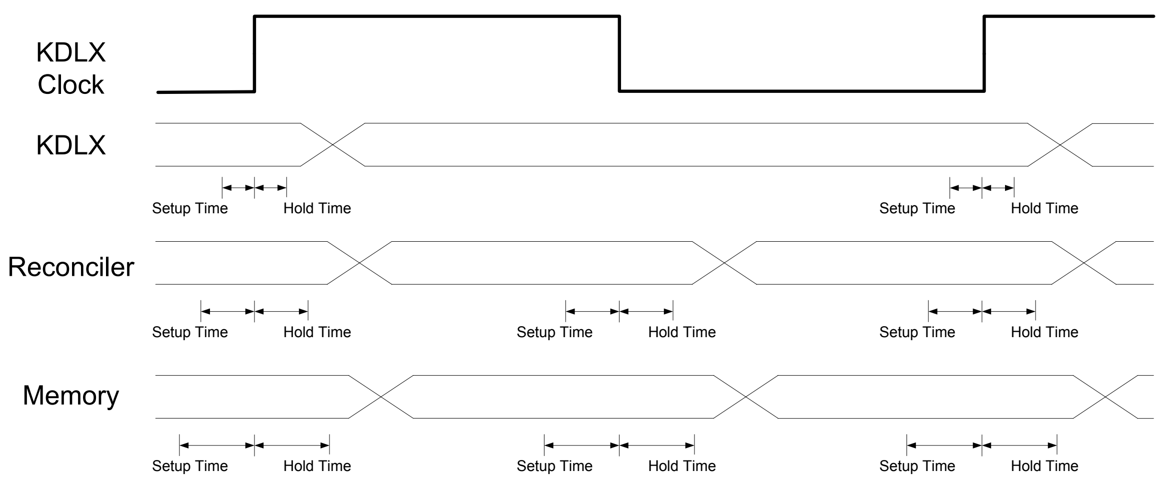


Figure 54. Timing Relationship Among Clocks

It does not matter that the *Reconciler* and memory clocks are faster than KDLX since the KDLX has to be ready whenever the *Reconciler* needs data. In Figure 54, all three clocks are shown together as they were in the simulation for comparing timing requirements. Since the *Reconciler* has a hold time longer than KDLX, the KDLX will be ready before the *Reconciler* is ready. The *Reconciler* will be set before the memory needs input signals.

When KDLX is executing a read-data instruction, the memory will have the data available later than the KDLX starts to read. Therefore, a little clipping occurs every time that KDLX reads data. To minimize this clipping, the setup and hold time between the three clocks have to be as close as possible.

In this simulation, if any two clocks have identical setup and hold time, the testing will fail. Since the *Reconciler* is a state machine, the current state will jump to a different state if the conditional requirements are not met in time. This causes the KDLX to fail to interact with the memory; therefore the following instructions will not be fetched.

E. CHAPTER SUMMARY

This chapter introduced the function of the *Reconciler* in the TMR design. This component is designed to consolidate two different architectures in a circuit and is not directly associated with error detection or correction in the TMR. This is the first time in this thesis that time constraints were discussed in detail since there are specific timing requirements for the *Reconciler*. The concept of establishing the setup time and hold time for a test bench is more important after this chapter because more components are involved in the TMR design.

Another component (called *Interrupt*) is discussed in the next chapter. This component leads the TMR design to the Interrupt Service Routine (ISR) when an error occurs. How to intercept the current execution of the KDLX to start an ISR and how it works with other components in the TMR design will be described as well.

VII. INTERRUPT

The TMR Assembly, consisting of processors and voters, is able to detect an error and correct it. Even though voters are able to correct errors as they come out the system, whichever of the KDLX processors that caused the error will still have the wrong data inside. If an error in one processor is not corrected in time, another error occurring in another processor may not be detected by voters. As was described earlier in Chapter V, a majority voter is not able to handle multiple identical errors.

In order to correct an error in the KDLX, the normal operation has to be stopped and all contents of registers in the three processors have to be voted. The voters will correct any inconsistency between the three processors in this process while storing all correct data into memory and then reloading them back into the original registers. Once this procedure is done, all contents of registers are identical between the three processors. The *Interrupt* is the circuit used to stop normal operation and switch the circuit to do this error correction.

A. CONSTRUCTION AND FUNCTION

The *Interrupt* is also a state machine coded in VHDL. The state machine is shown in Figure 55. The concept is to have it look for the error detection signal from the TMR Assembly. If an error occurs, it will latch the current program counter and send out a TRAP instruction to processors. Two NOPs follow the TRAP instruction in order to clean the pipeline of the processors. Only two NOPs are needed because the TRAP instruction will start to be executed right after the second NOP. Any instruction after the second NOP will either be useless or mask out instructions that the TRAP wants to fetch. After the second NOP, the TMR Assembly is in the ISR and the *Interrupt* waits for an RFE instruction from memory, placed to mark the end of the ISR.

When the processors receive the TRAP instruction sent from *Interrupt*, they jump to a specific memory location and start the ISR for storing and reloading the contents of all of the registers. The last instruction in the ISR is the RFE instruction. When memory sends out this instruction, it will be seen by the *Interrupt* and the *Interrupt* will replace the RFE instruction with a new Jump instruction. This new Jump instruction is constructed by the *Interrupt* from the Opcode $C8_{16}$ plus the latched program counter to force the processors to jump back to where the trap occurred.

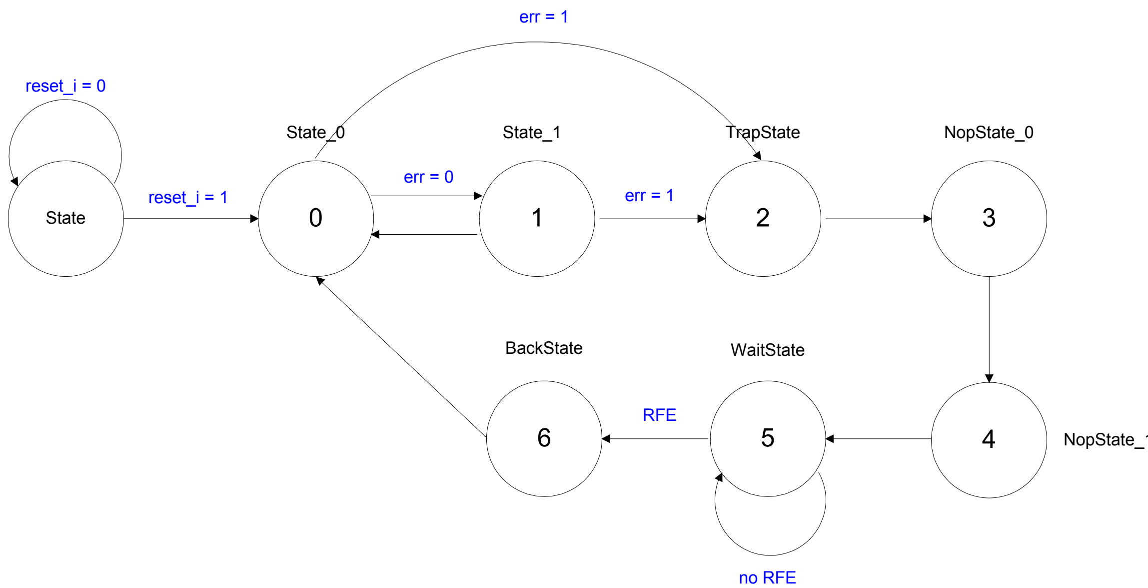


Figure 55. State Machine of *Interrupt*

Recall the function of TRAP and RFE instructions in Table 13. The reason to replace the RFE instruction with a Jump instruction is because the RFE instruction does not jump back to where the TRAP instruction occurs. It is known that the RFE will jump to the address stored in the IAR which is two clock cycles later than when the TRAP occurred. The choice was between revising a tested version of KDLX and building a separate circuit to be able to generate a new Jump instruction. The separate circuit is easier to achieve for this *Interrupt* since it is a state machine and is coded in VHDL. First, a state machine can do several different things in one clock cycle. Because the new Jump instruction is not needed until the *BackState*, two NOP clock cycles are sufficient for generating an instruction. Second, data on different buses can be more easily combined in VHDL than other methods, e.g., schematics.

The *Reconciler* discussed in the previous chapter only allows an instruction to be fetched in the first half of the KDLX clock cycle, but the state machine shown in Figure 55 works with a KDLX at the same speed. In order to interrupt and insert instructions at the correct timing, the *Interrupt* has to match the speed of the *Reconciler*. Doubling the speed of the *Interrupt* is not the same as that of the *Reconciler* since the *Interrupt* has several different states in series. The methodology here is to duplicate each state, which makes the state machine twice as long. The new state machine is shown in Figure 56 and its VHDL code is in Appendix C, section B.

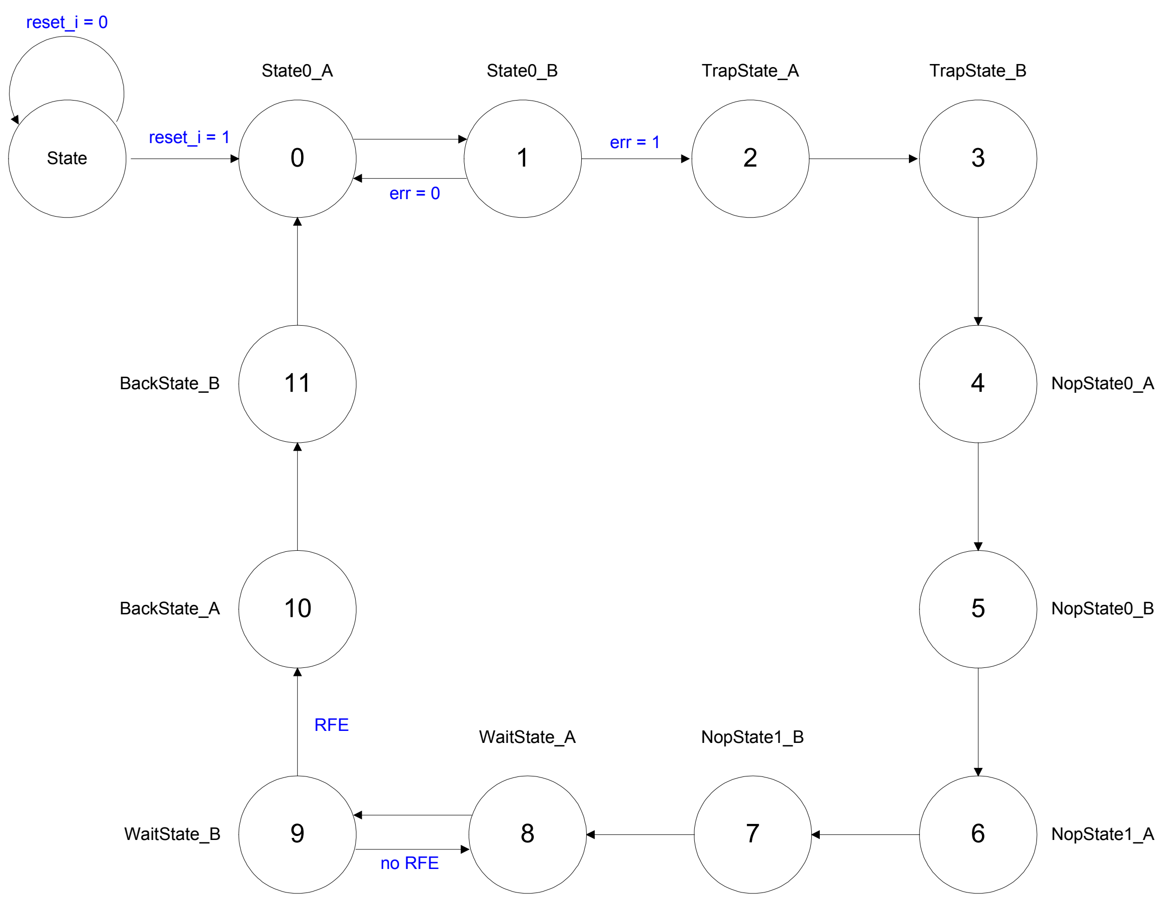


Figure 56. New State Machine of *Interrupt*

The first two states, *State0_A* and *State0_B*, do not need to be duplicated in spite of the even number of states. The state machine is also revised so that only *State0_B* can go to *TrapState_A*. In spite of double speed, *State0_A* still needs to go to *State0_B* even if an error occurs at *State0_A*. On the other hand, the KDLX reads and writes data at the falling edge of clock, which means that a data error always occurs at *State0_B*. After *NopState1_B*, the TMR design starts the ISR and the *WaitState_B* waits for the RFE instruction. Once the RFE instruction is sent out from memory, the *Interrupt* takes over the instruction bus again and injects the new Jump instruction at the *BackState_A*. The TMR design goes back to normal operation when the new Jump instruction is executed by the processors.

B. SCHEMATIC

The functions of *Interrupt* can be easily understood from the simulation result shown in Appendix A, section I. The simulation for the *Interrupt* only is not explained here since the *state_i* indicates active states in Figure 56 explicitly. Figure 57 is the schematic symbol of *Interrupt*.

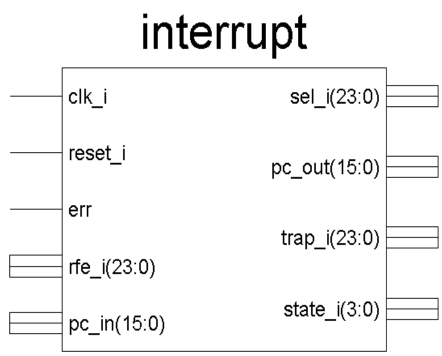


Figure 57. Schematic Symbol of *Interrupt*

The input signal *err* is used to monitor the occurrence of an error. When this signal goes high, the ISR starts. Once the ISR is triggered, the program counter where the error occurs is sent to *pc_in(15:0)* where it will be latched and this latched program counter will be output instantly at *pc_out(15:0)*. The *Interrupt* uses signal *sel_i(23:0)* to switch a mux and sends out the TRAP instruction via *trap_i(23:0)*. After that, *sel_i(23:0)* switches the mux back to normal and the input signal *rfe_i(23:0)* starts monitoring the Opcodes passing through on the instruction bus. When the RFE instruction is sent out from memory, *sel_i(23:0)* actives again and *trap_i(23:0)* sends out the new Jump instruction. Consequently, the TMR design is back to its normal operation. Figure 58 is the design of the *Interrupt* with a processor and two memories.

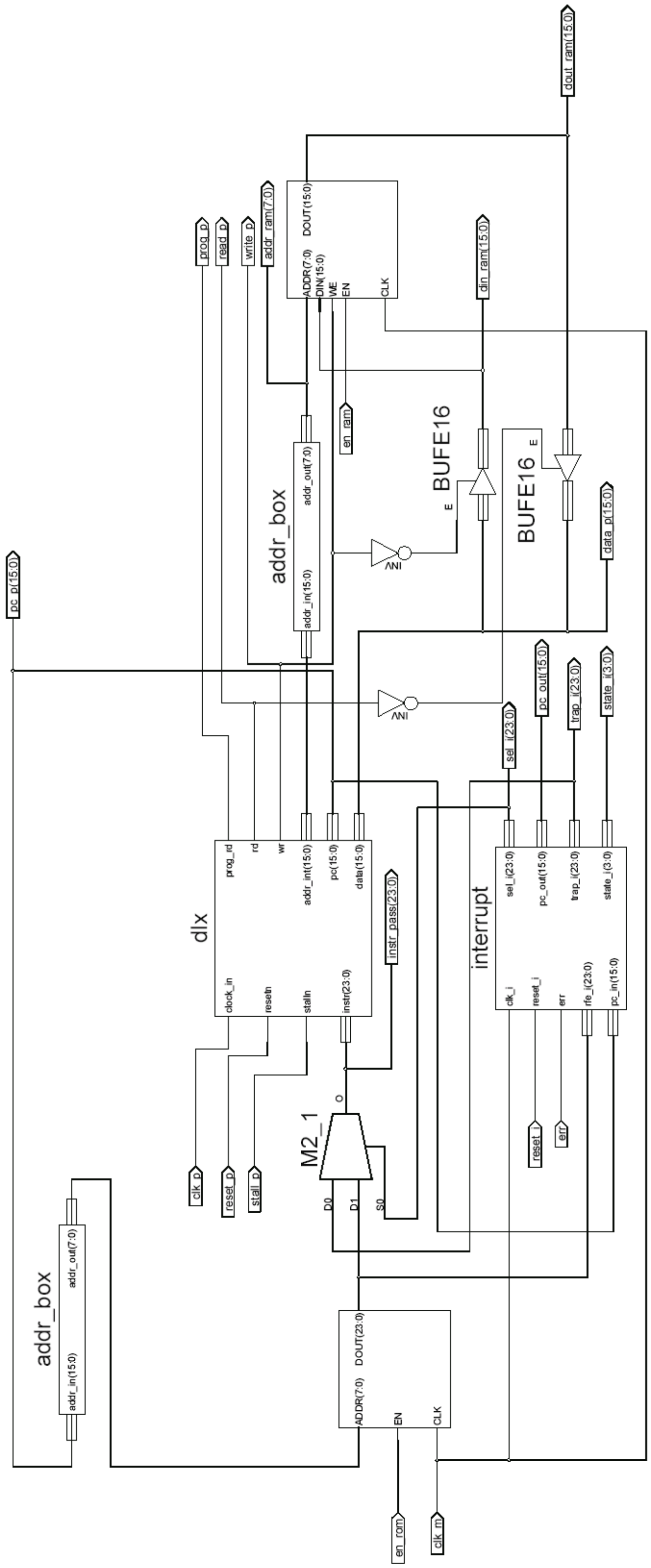


Figure 58. Schematic of the *Interrupt* with KDLX and Memories

The mux located between instruction memory and KDLX is used for *Interrupt* to inject the TRAP instruction. Normally, the KDLX fetches instructions from the instruction memory and the mux allows this traffic to pass. When an error occurs, the mux controlled by *Interrupt* immediately switches to the other bus and a TRAP instruction generated by the *Interrupt* will be sent to the KDLX. The original instruction at this time is blocked on the bus and the KDLX receives the TRAP instruction instead. The Opcode for the TRAP instruction in this thesis is 280030_{16} which uses memory location 0030_{16} as the starting point of the ISR. This value can be easily changed in *Interrupt*'s VHDL code. The basic idea is not to have the ISR address too close to the address of normal operations in memory to keep it from being overwriten. Simulations in this thesis are carefully designed and small address spaces let people see the complete implementation in memories.

C. SIMULATION

Table 20 shows the contents of the memories and the registers before and after the simulation.

Instruction Mem			
00		2D	
01		2E	
02	440101	2F	
03	440202	30	000000
04	440303	31	000000
05	440404	32	000000
06	440505	33	450420
07	440606	34	450520
08	440707	35	450620
09	440808	36	450720
0A	440909	37	411A11
0B	450110	38	411B22
0C	450211	39	411C33
0D	450312	3A	000000
0E	450413	3B	000000
0F	450514	3C	000000
10	450615	3D	F80000
11	450716	3E	000000
12	450817	3F	000000
13	450918	40	000000
14	450A19	41	
15	450B1A	42	
16	450C1B	43	
⋮	⋮	44	
		45	
2C		46	

Register	
00	
01	0044
02	0045
03	0046
04	0047
05	0048
06	0049
07	004A
08	004B
09	004C
10	0055
11	0066
12	0077
13	
14	
15	

Data Mem	
00	
01	0044
02	0045
03	0046
04	0047
05	0048
06	0049
07	004A
08	004B
09	004C
0A	
0B	
0C	
0D	
0E	
0F	
10	0044
11	0045
12	0046
13	0047
14	0048
15	0049
16	004A
17	004B
18	004C
19	

Table 20. Tables of Registers and Memories in Simulation

Part of the complete simulation is shown in Figures 59 and 60. An error is seen at point 1 and the instruction at point 2 is intercepted by the *Interrupt*. It can be seen clearly that the value of signal *sel_i* changes and a TRAP instruction followed by two NOPs are injected at point 3.

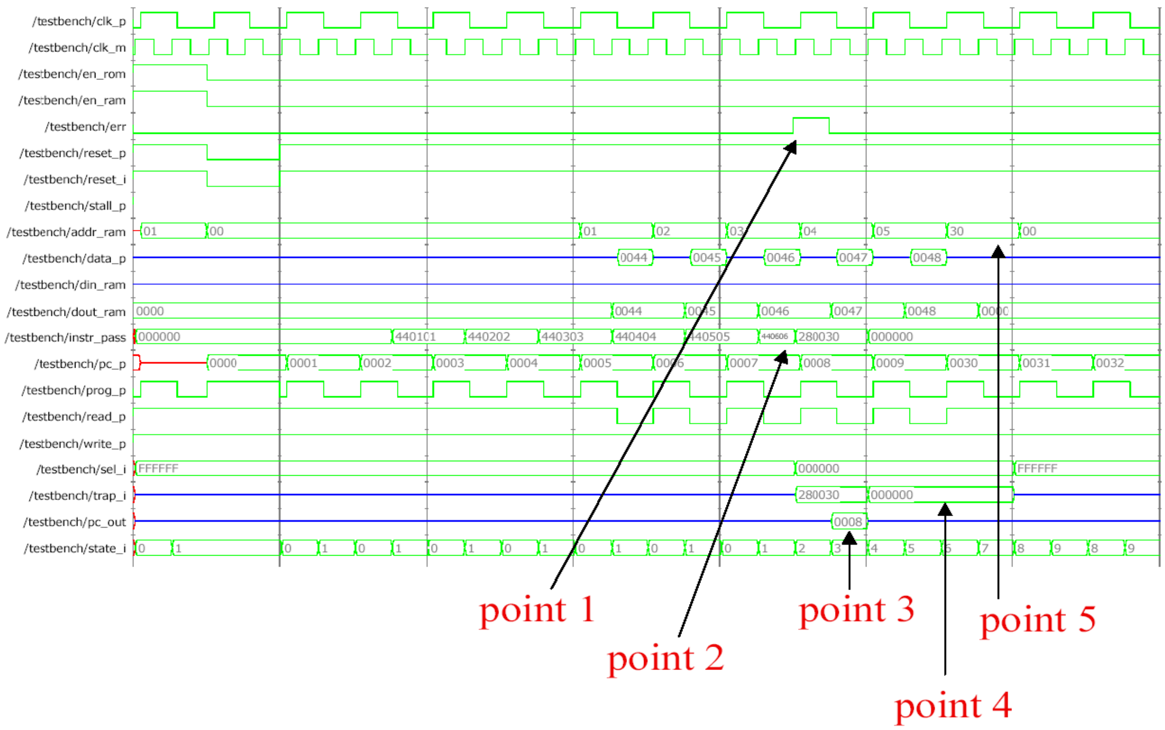


Figure 59. Partial Simulation Result of *Interrupt* with KDLX

One important thing here is that the time an error is seen is not the time an error occurs. The reason is because the KDLX is pipelined and the memory stage is the fourth pipeline stage. Including the time for the *Interrupt* to respond, the total delay from the instruction causing the error is **four** KDLX clock cycles. This feature cannot be seen in this simulation because the error was set manually.

The program counter latched by the *Interrupt* at point 3 is 0008_{16} in this simulation. The instruction intercepted is 440606_{16} which is at address 07_{16} in Table 20. The concept is to jump back to where the TRAP was inserted. Theoretically, the program counter latched should be 0007_{16} not 0008_{16}. Because of the change of the *pc_p* at point 3 and the instruction delay from memory, the latched program counter is a wrong value. Another possible reason is since this error is generated from the test bench not from the circuit itself, the timing for the occurrence of an error could be in the wrong place. This issue will be discussed again and resolved in Chapter VIII when the full design without *ESSD* is presented.

The TRAP instruction inserted at point 3 affects the circuit at point 5. Opcodes from instruction memory address 30_{16} to 40_{16} are the ISR. Instructions in the ISR can be related or unrelated to the original commands, but the purpose is to correct the error. Since there is no actual error in this simulation, the ISR is designed just to do something else. The full function of the real ISR is to store all contents of the registers to memory and reload these contents back to registers. The ISR in this simulation is incomplete.

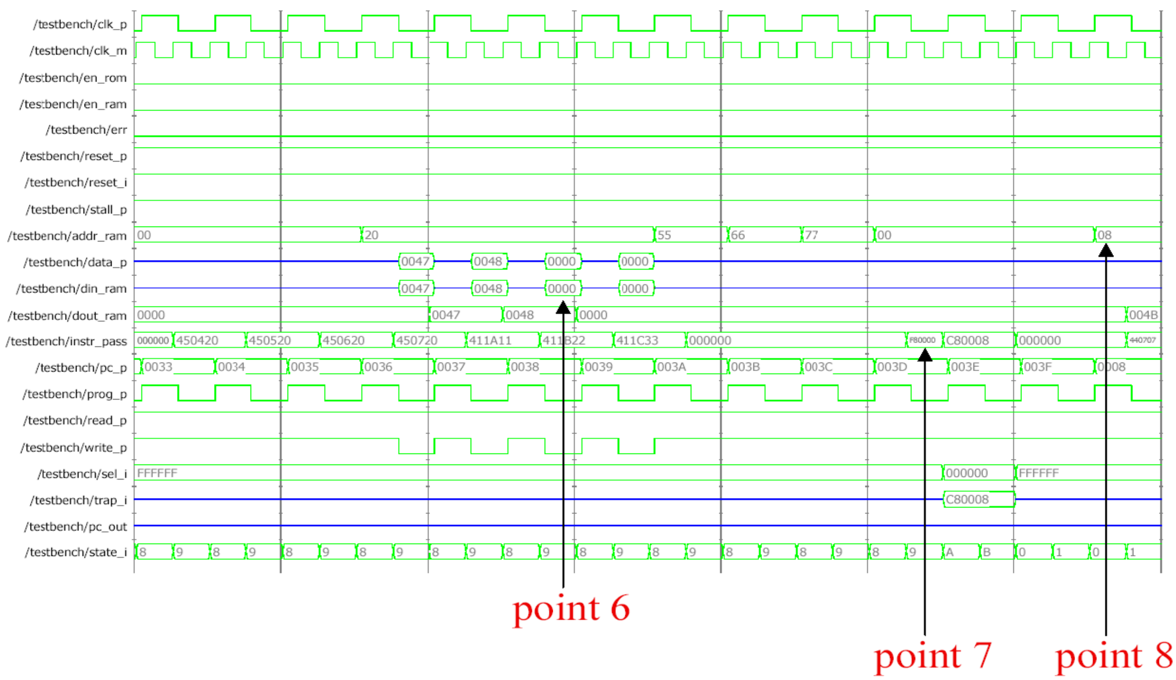


Figure 60. Partial Simulation Result of *Interrupt* with KDLX (continued)

Storing the contents of R4 to R7, the simulation shows R6 and R7 at point 6 are not loaded with any value. This proves that the *Interrupt* can successfully insert the TRAP instruction. At point 7, the RFE instruction (i.e., $F80000_{16}$) is detected by the *Interrupt*. Instantly, *sel_i* switches to zero and *trap_i* sends out the new Jump instruction, $C80005_{16}$. As described earlier, the new Jump instruction is formed from ($C8_{16}$+latched program counter). Therefore, the Opcode $C80005_{16}$ is generated and executed at point 8.

The rest of simulation in Appendix A, section J checks the contents of registers to verify the operation.

D. CHAPTER SUMMARY

The functions of the *Interrupt* were described and simulated in this chapter. When an error occurs, the *Interrupt* should lead the TMR design to do error correction and also be able to bring the circuit back to its normal operation. The purpose is to correct an error as soon as possible after it occurs. Thus the error will not be propagated making the circuit lose control.

The first design of the *Interrupt* was to replace instructions in memory in order to implement the ISR. This could not be done in this design because a ROM is used as the instruction memory. Since the real CFTP design uses only one RAM, the instruction set could be changed in memory. However, changing original instructions is the last thing people want to do because it may cause an unrecoverable error.

In the next chapter, the full design without *ESSD* will be introduced. The usage of the ISR will be described clearly and the interactions between *Interrupt* and *Reconciler* will be expressed as well. The simulation of the full design should clarify any confusions among the different components.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. THE FULL DESIGN WITHOUT ESSD

The full design in this chapter consolidates the *TMRA* from Chapter V, the *Reconciler* from Chapter VI and the *Interrupt* from Chapter VII. The *TMRA* contains three KDLX processors and six voters. All outputs of the processors are voted and any error will be corrected. The *Reconciler* is responsible for integrating the Harvard and Von Neumann architectures. It runs in double speed in order to act as an instruction memory in the first half of the KDLX clock and as a data memory in the second half of the KDLX clock. The component used to correct errors besides the voters is *Interrupt*. It intercepts normal operation of the *TMRA* when an error occurs, forces it to do an ISR and makes it jump back to normal operation after the error is corrected. The error signal for the *Interrupt* is given by the *TMRA*. For this design the voter is assumed to be error-free and the voter error detection signal is not used.

Each component discussed earlier has been simulated to prove its function with or without the KDLX and memories. Simulating all these components together in a circuit should be able to catch and correct an error. This is the goal for the full design and its function will be proved in this chapter.

A. SCHEMATIC

The *TMRA* itself basically connects with the memories as just one KDLX would. Most input and output buses are the same except the number of signals increases or decreases. The *Reconciler* sitting between the *TMRA* and the memory has to receive all output signals that the original KDLX has, except the program read signal, i.e., the read and write signals, the program counter, the address for data, and the data bus. The *Interrupt* needs the error signal to trigger the ISR, the program counter to generate a new Jump instruction, and instructions for doing TRAP, RFE and Jump.

In order to test the circuit, several buses and memory have to be triplicated. The way to test the error handling of the system is to program an inconsistency into one of the three memories and expect that the circuit can catch the error and correct it. Without this artifice, the *Interrupt* will never work and the ISR will never be triggered. The alternate would be to assign an error signal to change data on the bus manually in the test bench and that is not realistic. The full design constructed for testing is shown in Figure 61.

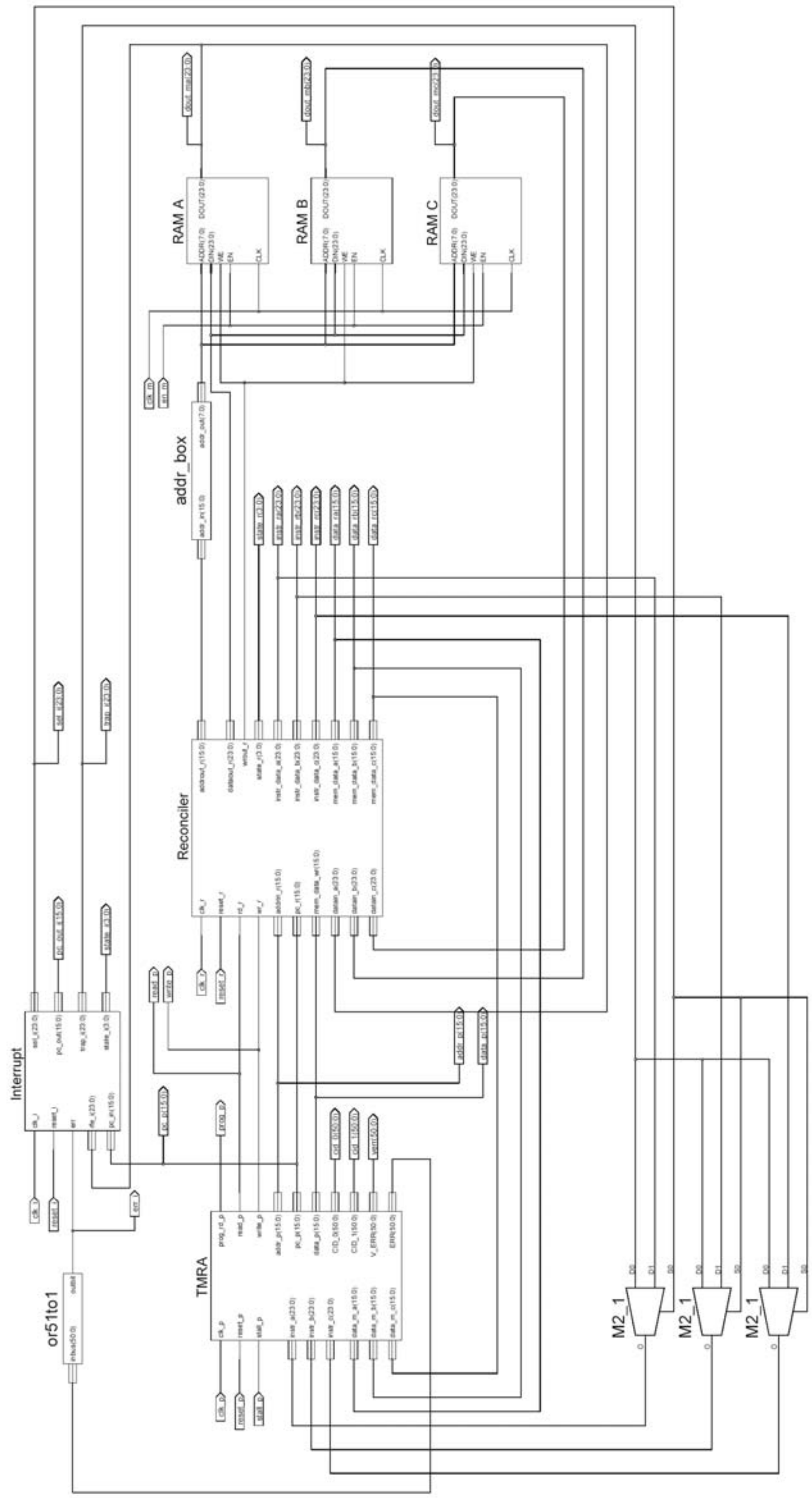


Figure 61.   The Full Design

In Figure 61, only the *Interrupt* is unchanged since it does not have any data bus connections. Three *RAM*s are used, and a bus connects each to one of the processors. Therefore, both *Reconciler* and *TMRA* have more buses than before. The three muxes at the bottom left are used to intercept the TRAP and Jump instructions. The box at the top left (called *or51to1*) is coded by VHDL and ORs 51 bits from *ERR(50:0)* into 1 bit. Any error that occurs at any output signal of the KDLX will trigger the ISR. The revised VHDL code for *Reconciler* is in Appendix C, section C.

Because the *Interrupt* must monitor a memory bus in order to detect the RFE for testing, one of the memories must always be correct. This design chooses *RAM A* as the monitored RAM; therefore its contents are always correct.

B. SIMULATION

The three *RAM*s are pre-configured as shown in Figure 62. In order to express the concept of the TMR and keep the simulation simple, only the data at memory location $4C_{16}$ is different for *RAM B*. The ISR is designed to start at address 30_{16} and end at $3C_{16}$. What the ISR does is to store contents of registers to memory, relying on the voters to ensure that the correct contents are written into memory. (In the real circuit, the ISR then restores all registers from these correct values in memory.) The Opcode $F80000_{16}$ is the RFE instruction used to tell *Interrupt* where the end of the ISR is. Instructions from address $0A_{16}$ to 10_{16} are used to check data in registers.

RAM A, B and C				
00	000000	2D		
01	000000	2E		
02	44014A	2F		
03	44024B	30	45014A	
04	44034C	31	45024B	
05	44044D	32	45034C	
06	44054E	33	45044D	
07	44064E	34	45054E	
08	000000	35	45064F	
09	000000	36	000000	ISR
0A	000000	37	000000	
0B	44014A	38	000000	
0C	44024B	39	F80000	
0D	44034C	3A	000000	
0E	44044D	3B	000000	
0F	44054E	3C	000000	
10	44064E	3D		
11	000000	3E		
12	000000	.	.	
13	000000	.	.	
14	000000	.	.	
.	.	4A	0000AA	
.	.	4B	0000BB	
.	.	4C	0000CC	RAM B has 00011
.	.	4D	0000DD	
.	.	4E	0000EE	
.	.	4F	0000FF	
2C		50		

Figure 62.   Memory Pre-configurations

Figures 63, 65, and 66 display the full simulation result and some trivial signals are not shown. There are four clocks in this design. Clock signals *clk_p*, *clk_i*, *clk_r*, and *clk_m* are for the KDLXs, *Interrupt*, *Reconciler*, and *RAM*s, respectively. The KDLX clock runs at one-half the speed of the others. Since the *Interrupt* does not need signals from the *Reconciler* and vice versa, these two components are running at the same clock speed. The *RAM*s are looking for the outputs of the *Reconciler* so the memory clock has the longest setup and hold time.

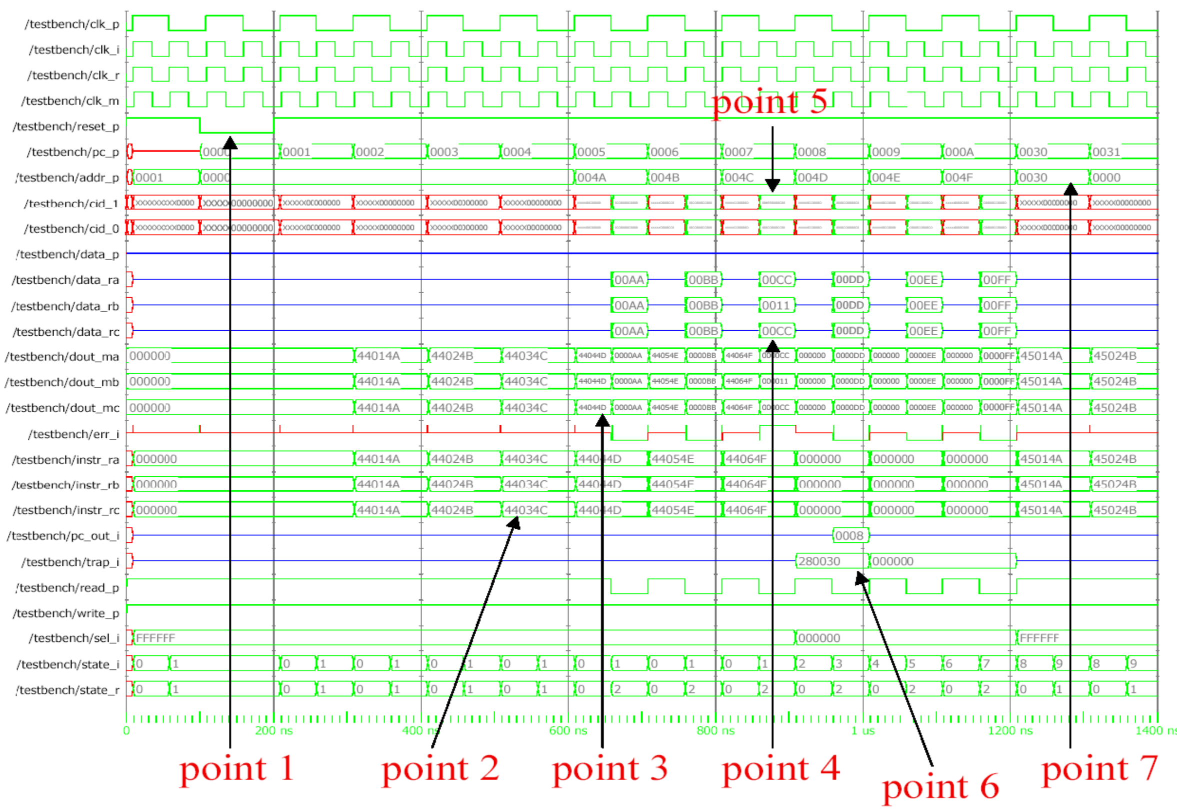


Figure 63. Simulation of the Full Design without *ESSD*

The KDLXs, *Interrupt* and *Reconciler* are reset at point 1 and only *rest_p* for processors is shown. When the program counter, *pc_p*, is 0002_{16}, the first instruction is fetched. It is known that the instruction at point 2 should cause an error because the data at address $4C_{16}$ is not consistent between *RAM*s. Tracing the simulation to point 3, the function of the *Reconciler* is shown clearly here. Half of the KDLX clock cycle is fetching the instruction at the corresponding program counter and the other half cycle is reading data from the memory for the first instruction. So the *Reconciler* actually reads the instruction at memory address 0005_{16} first and then reads the data at address $004A_{16}$. This feature makes it possible to consolidate the two different architectures. As discussed earlier, the instructions should be held until the next rising edge of the KDLX clock. Thus the *Reconciler* should not block any data or make a bus high impedance on *instr_ra*, *instr_rb*, and *instr_rc*.

Instructions at point 2 are executed one KDLX clock cycle after point 3. The data needed for these instructions is offered at point 4. The wrong data in *RAM B* is sent to R3 of the second KDLX in the *TMRA* at this time. It is hard to see but *cid_0* and *cid_1* at point 5 do report errors. The main purpose for this simulation is to show how different components work together and realize the concept of the TMR. Therefore, the error reports will be analyzed later.

Since the voters are hooked-up to the output buses of the KDLXs, it may be confusing that the *TMRA* reports an error while it is loading data not storing. If this error is not seen while loading, then the TMR will not be able to find it until the next time this error is stored into memory. Figure 64 is only a part of the TMR Assembly in Figure 26 and shows how input data flows.

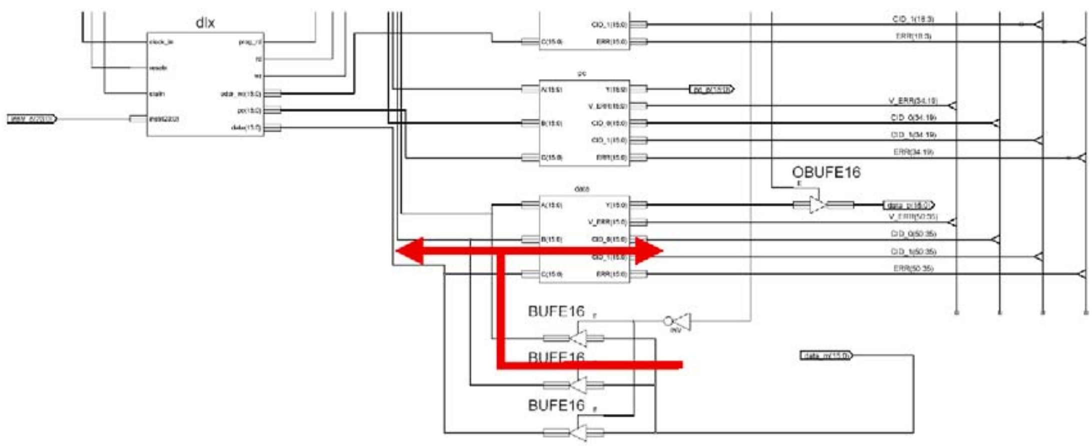


Figure 64. Flowing Direction of the Input Data in *TMRA*

The flowing direction of the input data to the KDLXs is expressed clearly in Figure 64. Even though the buses on the voters are not bi-directional, the input data can still be voted by this scheme. Therefore, the TMR can check data either on loading or storing without waiting until the wrong data is used.

Going back to point 4 in the simulation result. An error is caught by the voter so the *err_i* becomes high and triggers the ISR. At point 6, the signal *sel_i* switches to 000000_{16} which allows the *Interrupt* to insert one TRAP instruction and two NOPs to *TMRA*. Notice that the *state_i* changes to 2_{16} which is the *TrapState* of *Interrupt*. The program counter latched is 0008_{16} so the TMR should jump back to this address when the ISR is done. At point 7, the TRAP instruction is executed by the KDLX and starts the ISR portion in Figure 62.

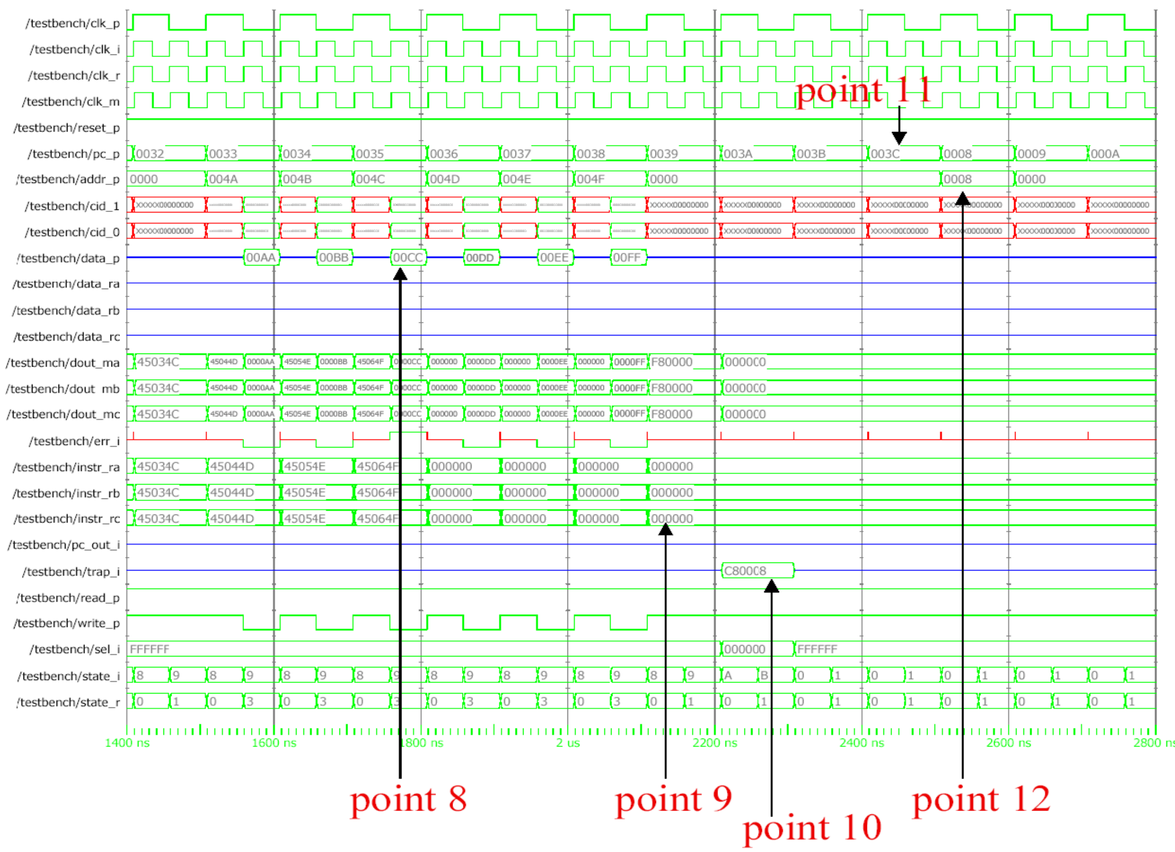


Figure 65.   Simulation of the Full Design without *ESSD* (continued)

The implementation in this ISR is to store all contents of registers to memory. All data in registers will be voted this time and any inconsistency should vanish. The wrong data in *RAM B* ought to be corrected after this implementation. Normally the ISR will not write to original data. The reason for doing this here is because this test is to prove the ability to correct an error. Thus the same error should not appear next time when the same instruction is executed.

The contents of R3 shows up again at point 8 in the ISR. Any error detected while in the ISR will be ignored since this procedure is correcting an error and voters will take care of other errors. The *err_i* flags at point 8 will be ignored again because it is known that the data in R3 of the second processor is wrong. Signals *cid_0* and *cid_1* at this point report the same error syndrome as the one at point 5. It could be explained easily since data is the only thing having a problem. If the third Opcode for ISR is different in one of the processors, signals *cid_0* and *cid_1* at point 8 will have a different error syndrome. It could be seen that *Interrupt* stays at the *WaitState* until it sees the RFE instruction.

Once the *Interrupt* detects the RFE instruction sent out from the *RAM A*, it starts its *BackState* at point 10. The instruction buses of the *Reconciler* (i.e., *instr_ra*, *instr_rb* and *instr_rc*) are forced to zero at point 9 when the RFE instruction is detected. The RFE instruction can never be passed to the *TMRA* or it will be fetched and executed at point 12. If so, the new Jump instruction at point 10 becomes useless.

The *Interrupt* inserts the new Jump instruction, $C80008_{16}$, one clock after point 9. Therefore, it takes three clock cycles to have the new program counter used after $F80000_{16}$ is seen by *Interrupt*. The operation code from address $3A_{16}$ to $3C_{16}$ in Figure 62 will not be implemented since the *Reconciler* wants to clean the pipeline before the TMR goes back to normal operation. So point 11 in the simulation is where the ISR stops. At this time, both *Reconciler* and *Interrupt* are already back to normal states. The TMR goes back to normal operation at point 12.

Doing exactly the same instruction set again from address 08_{16} to 10_{16} in Figure 62 proves the error in *RAM B* has been corrected. No error is reported and the ISR is not triggered again at point 13 in Figure 66.

A complete ISR should store all contents of registers to memory and reload them back to the original registers. Inconsistent data between the three processors should vanish. The ISR shown in Figure 62 is not complete in order to keep the simulation simple. Generally speaking, the ISR should not overwrite the original data. A temporary memory location needs to be specified for storing and reloading purposes in the ISR. The simulation in this design of overwriting the original data just proves the function of the error correction.

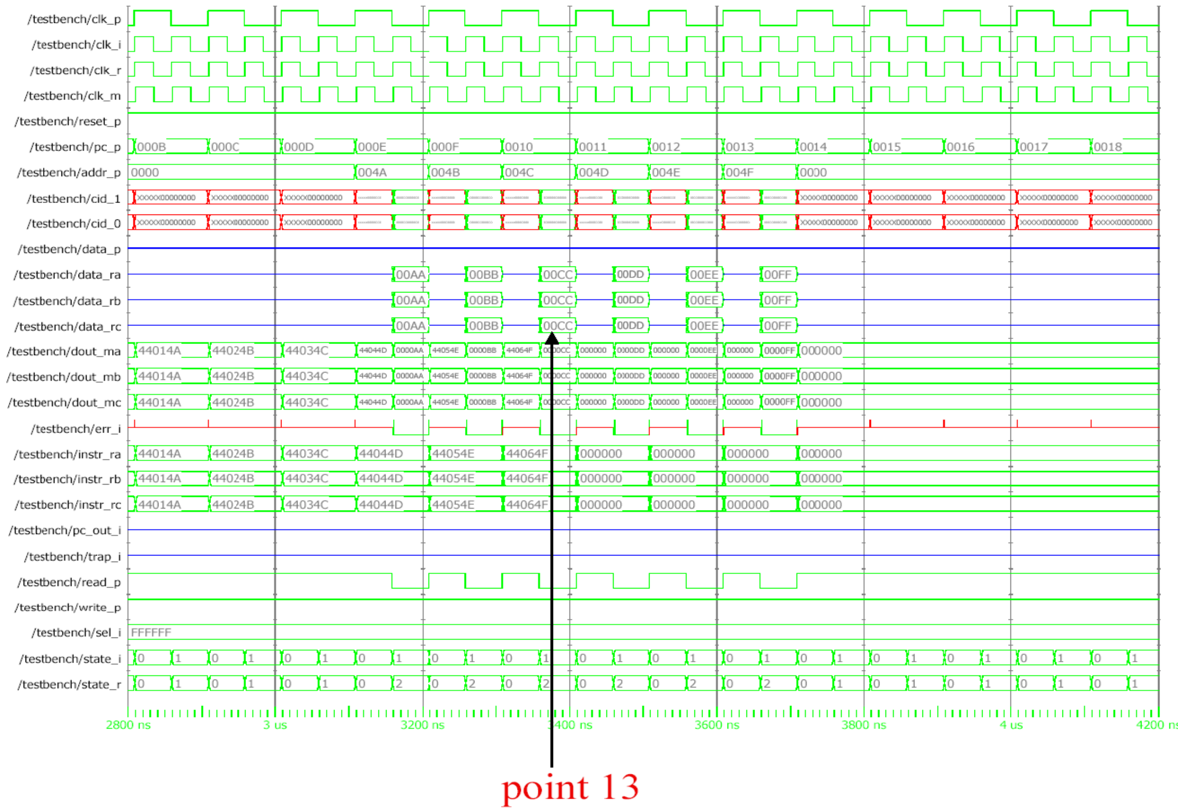


Figure 66. Simulation of the Full Design without *ESSD* (continued)

C. ERROR ANALYSIS

The analysis of the error in this simulation is quite easy since the data portion is the only part that needs to be checked. Figure 67 shows the way to check the error.

At point 5 in the simulation, the cid_1 is $006E800000000_{16}$ and the cid_0 is all zero. A zoom-in on point 5 is shown in Appendix A, section K. It can be quickly identified as an error from the second processor. Comparing the inconsistent portion of the data with *cid* data shows that they have the same pattern which demenstrates that the error report in this design is correct.

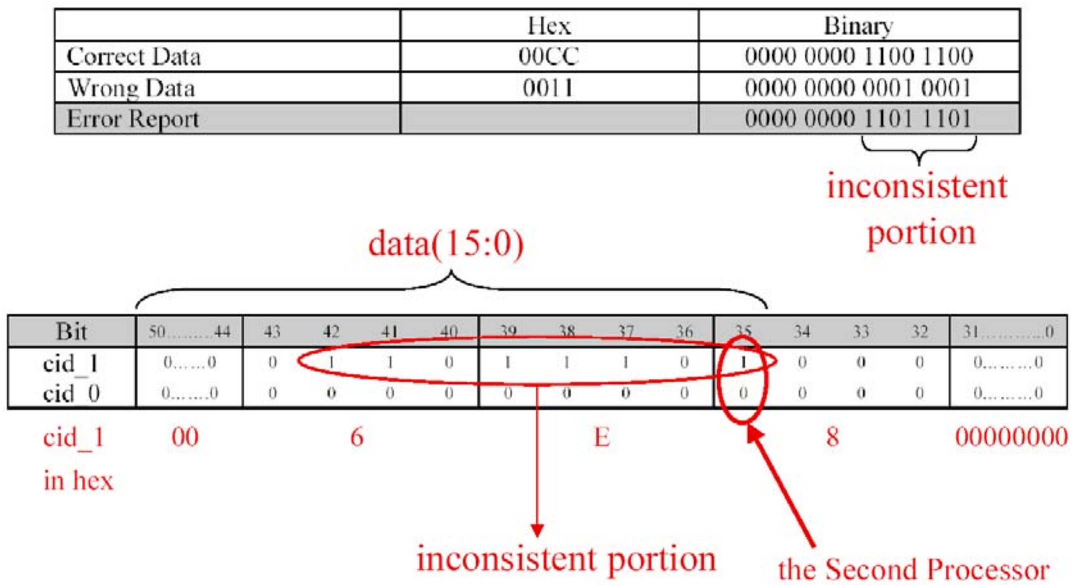


Figure 67. Error Analysis for the Full Design

D. CHAPTER SUMMARY

It is exciting to see that this full design works in simulation. The three KDLX processors work in parallel and the design functions as desired. Confusion on how *Interrupt* or *Reconciler* works should have been cleared up by the material in this chapter. The program counter is not latched properly in Figure 59, but works perfectly in the full design. The timing issues of the simulation arise again. Changing the way to latch the program counter in the *Interrupt* to make it work in Figure 59 may cause the simulation of the full design to fail.

The last component for a complete TMR design is the *Error Syndrome Storage Device (ESSD)*. This is a device used to store error syndromes for future analysis. The full design with *ESSD* will be introduced in the next chapter.

IX. THE FULL DESIGN WITH ESSD

After designing and simulating different components, the TMR design is almost completed. In the previous chapter, it has been shown that the voters are able to report and locate an error when it occurs. Errors on different buses will be reported by *cid_1(50:0)*, *cid_0(50:0)*, *err(50:0)*, and *v_err(50:0)*. The pattern generated for an error on these buses is called the error syndrome.

A space system like CFTP will leave the earth for a long time. It is desired to have some kind of device to collect the error syndrome whenever an error occurs. The error syndrome can be used to analyze the health of the system or help understand the space environment for a system on orbit. If the same error is generated several times, it can be assumed that a certain device is defective or deviant. The solution may be to re-program the FPGA or reset the system. The *ESSD* is the device designed to collect error syndromes. In order to be able to download this data after a period of time, the *ESSD* has to store the error syndromes to memory.

A. THE FUNCTION OF ESSD

Simulation for the full design without *ESSD* was introduced in the previous chapter. Therefore, the functions of *ESSD* are to store the error syndromes and where they are located in the system. The *ESSD* is designed pretty much following the concept of building the *Interrupt*. It is a state machine coded in VHDL and runs in double speed, that is in synchronization with the memory clock. It has to run in double speed in order to work with errors generated in either half of the KDLX clock cycle. Because the ISR will be triggered when an error occurs, choices for where *ESSD* is to be implemented are before, after or sometime within the ISR.

Halting normal operation is the last choice since the ISR is already designed to do that. It is reasonable not to interrupt the normal operation unless absolutely necessary. Too many interruptions may decrease the performance of a system or cause the program to lose track of the instruction sequence. Due to these reasons, the *ESSD* is implemented in the ISR instead of triggering another interrupt routine somewhere in normal operation.

To minimize the impact on ISR, the *ESSD* is designed to start right before the first instruction in ISR begins. The two NOPs following the TRAP instruction are a good starting point for *ESSD* since the pipeline is cleaned and no useful instruction is executing. Consolidating all of the concepts above, the state machine for *ESSD* is constructed as Figure 68 and its VHDL code is in Appendix C, section D.

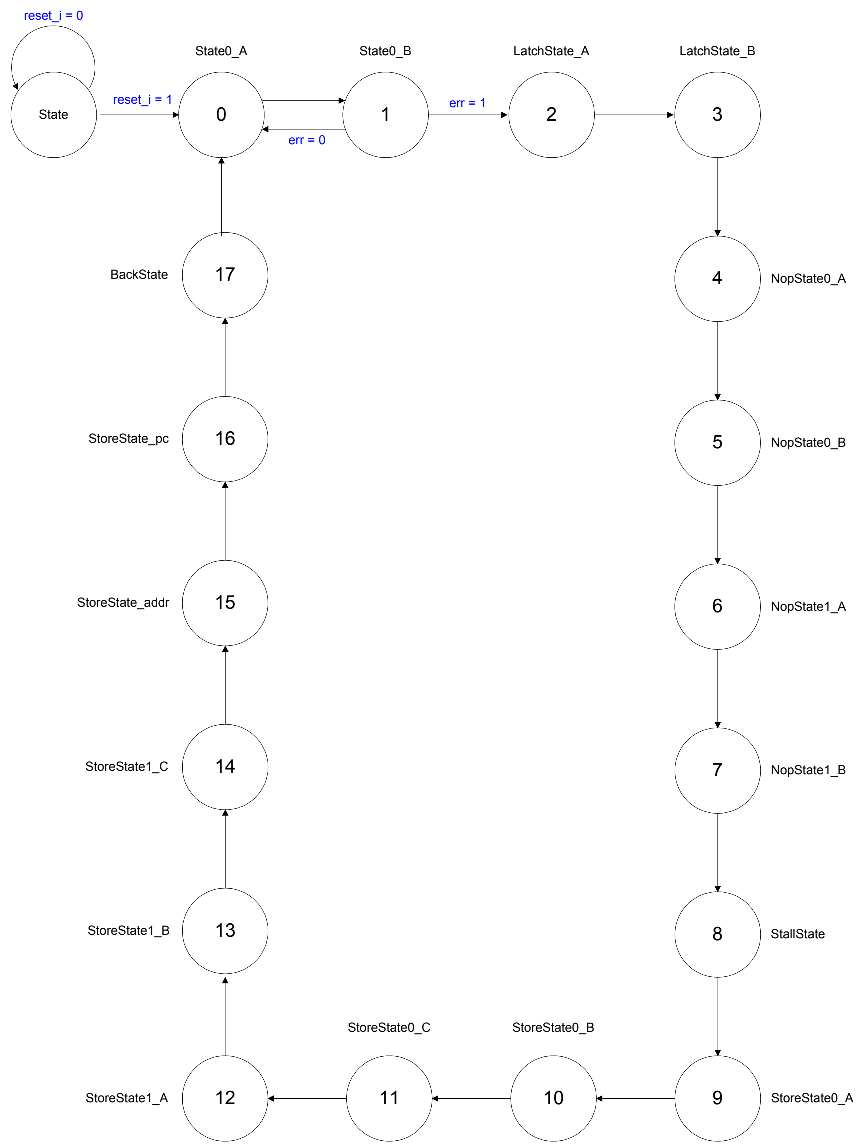


Figure 68. State Machine of *ESSD*

The first eight states are very similar to the states in *Interrupt*. This is because the *ESSD* has to wait until two NOPs are inserted. The *LatchState_A* latches the program counter, the data address, and the 51-bit data on the *cid_0* and *cid_1* buses. The *Stall-*

State stalls KDLX in order to start storing the latched error syndromes. The *ESSD* stores data to memory as a stack which starts at the bottom and runs to the top. For simplicity and explanation purpose, we use address 0059_{16} as the starting point and store data from the least significant bit to the most significant. This function is illustrated in Figure 69.

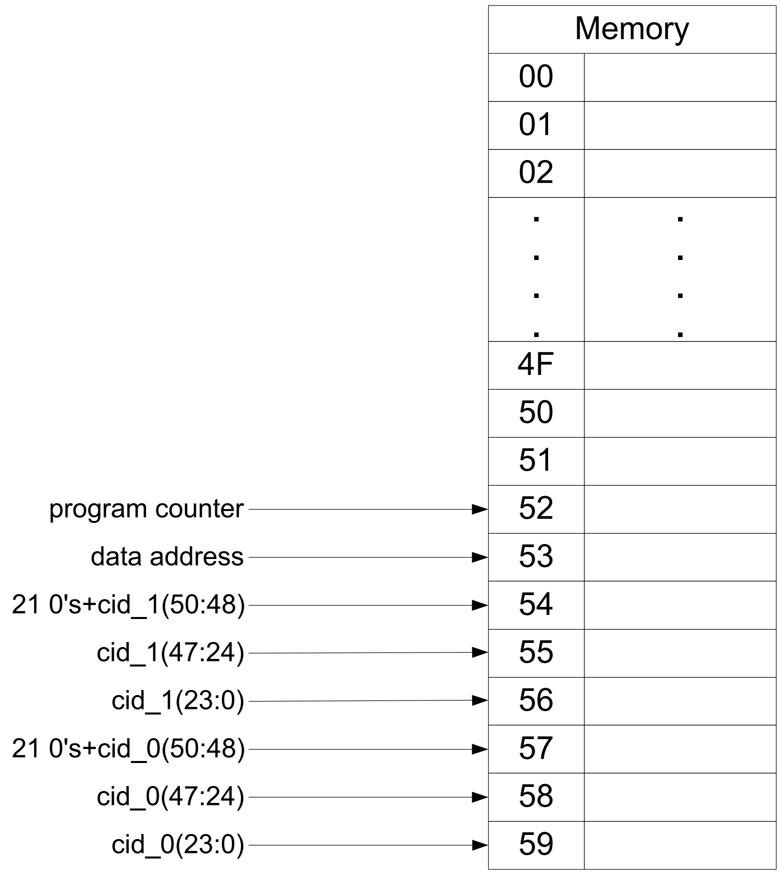


Figure 69. Function of *ESSD* Storing

Each data word in memory is 24-bits wide so a 51-bit data syndrome takes three clock cycles to store. The most significant three bits of *cid_0* and *cid_1* are stored with 21 zeros ahead. A counter is used internal to *ESSD* to track the memory locations. The next error syndrome will start at address 51_{16}. States from *StoreState0_A* to *StoreState_pc* implement the actions described here. During this period, all of the processors are stalled and the memory is controlled by *ESSD*. The last state is the *BackState* which releases the processors to start the ISR.

The *ESSD* runs at twice the speed of the *TMRA* but states after the *NopState1_B* are not doubled as the other state machines do. Because the *ESSD* and the memory are both in double speed, one memory access can occur in every *ESSD* state. Therefore, states between *StoreState0_A* and *BackState* do not need to be duplicated. The *Interrupt* and *Reconciler* stop functioning when KDLX is stalled. The schematic symbol of *ESSD* is shown in Figure 70.

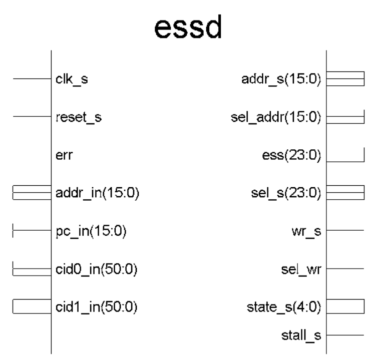


Figure 70. Schematic Symbol of *ESSD*

Input signals at the left side are used for latching data from the buses. Output signals, *sel_addr(15:0)*, *sel_s(23:0)*, and *sel_wr* are used to switch muxes in order to insert data on *addr_s(15:0)*, *ess(23:0)*, and *wr_s*, respectively. The *stall_s* goes low to stall KDLX when error syndromes are ready to be stored.

B. THE FULL DESIGN WITH ESSD

1. Schematic

The schematic for the full design with *ESSD* is shown in Figure 71. Comparing with Figure 61, the *ESSD* is added at the bottom right and all incoming or outgoing buses are intercepted with muxes. The *ESSD* obviously takes over *RAM*s once it starts to store error syndromes. Three muxes at the input side of *RAM*s are used to insert the data address, data and write signal. The other three muxes on the output buses of *RAM*s are used to intercept any unrelated data to *Reconciler* while storing the error syndromes.

Two big latches called *latch51* are sitting on the *cid_0* and *cid_1* buses ahead of the *ESSD*. This part is coded in VHDL and is necessary for this design. It latches data when *err* is high and keeps the latched data until the next error is detected. Therefore, the

ESSD can capture *cid_0* and *cid_1* whenever it wants because this data is available and stable on the bus. More explanation of how it functions and why it is vital in this design will be described in the simulation discussion.

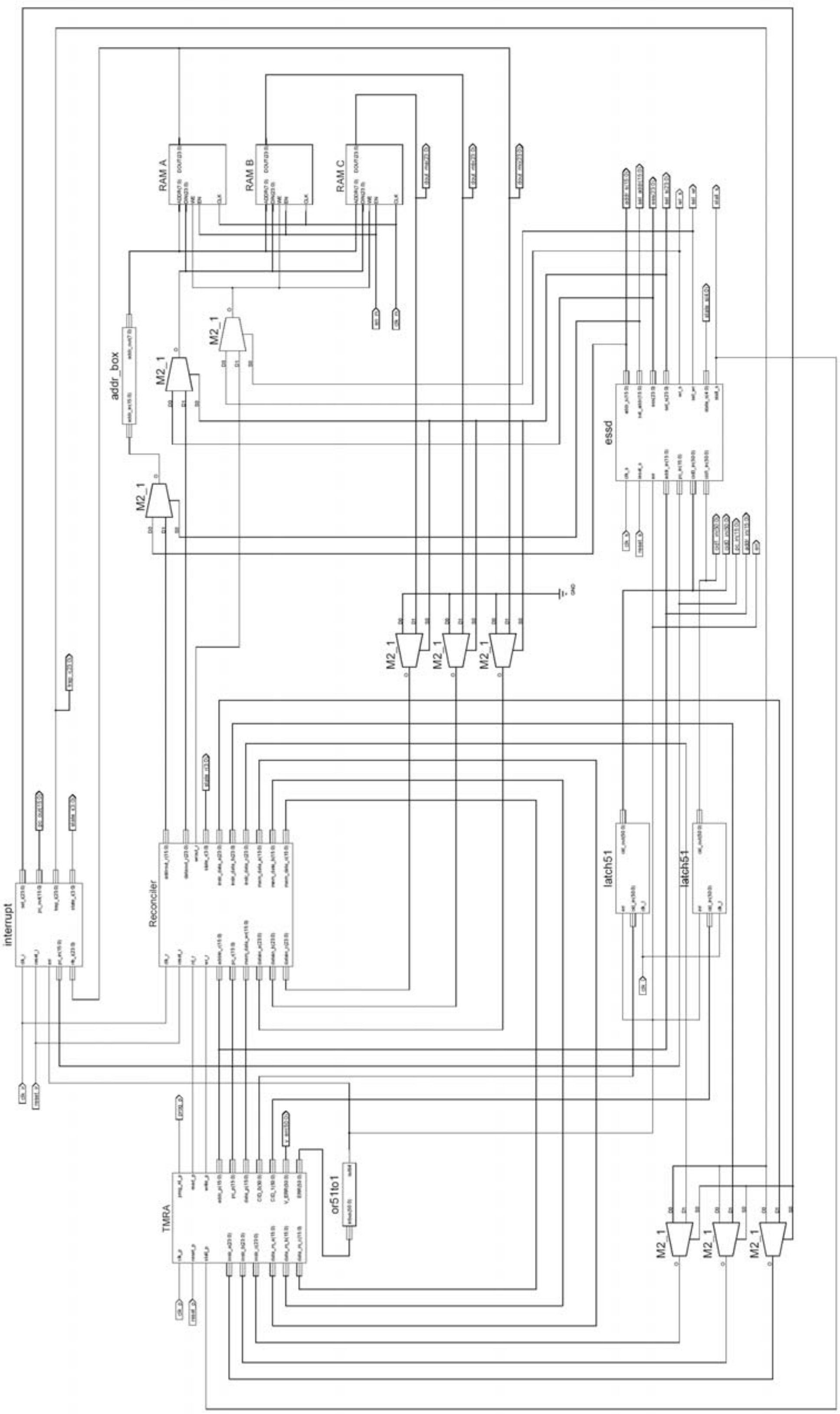


Figure 71. Schematic of the Full Design with *ESSD*

2. Simulation

Fewer signals are monitored here than with the full design in the previous chapter, since the test bench is almost identical except for a few extra instructions for checking stored error syndromes in memory. Functions of the *TMRA*, *Interrupt* and *Reconciler* in the full design without *ESSD* have been described so this simulation just shows how the *ESSD* works. Important signals and all buses on the *ESSD* are monitored in the simulation shown in Figures 72 and 74. This simulation ignores most identical parts introduced in the previous chapter. Only the important functions of the *ESSD* are shown for explanation.

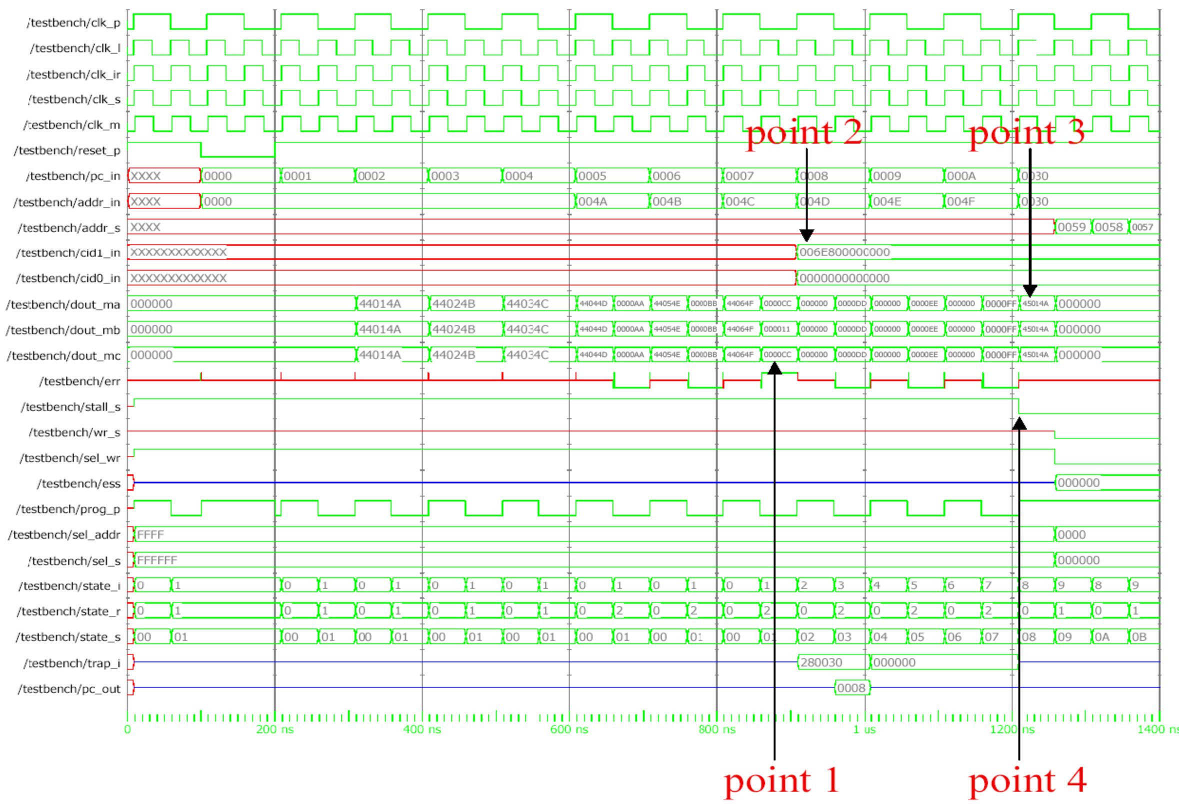


Figure 72. Simulation of the Full Design with *ESSD*

In Figure 72, five clocks are listed. The *Reconciler*, *Interrupt* and *ESSD* all work in parallel so the time constraints for *clk_ir* and *clk_s* are identical. The new clock, *clk_l*, for *latch51* needs to run at double speed, and it has to be stable before the *ESSD* is ready. Because of this, the *latch51* has less setup and hold time comparing with the *ESSD*.

As before, the error is caught at point 1 and *cid_1*, *cid_0* indicate where the error is. One needs to know that *cid_1* and *cid_0* are output data of *latch51*. Unlike the simulation in previous chapter, data on *cid_1* and *cid_0* show up at point 2 and are latched until the next error is reported in normal operation. The *ESSD*, therefore, is able to store these two data when *state_s* is 02_{16}.

The most important reason for using *latch51* is to make the data stable on the bus. The zoom in at point 5 in Figure 63 is shown in Figure 73. The data of *cid_1* and *cid_0* is available after the memory clock cycle and becomes unstable before the next rising edge of the *Interrupt* or *Reconciler* clock cycle. Because the *ESSD* is running exactly the same clock speed as the *Interrupt* and *Reconciler*, both *cid_1* and *cid_0* have to be available until the next rising edge of the *Interrupt* (or *Reconciler*) clock in order to be latched correctly for the *ESSD*. Due to this reason, the *latch51* is designed to keep the data stable and the *ESSD* thus can latch it at any state before storing the error syndromes.

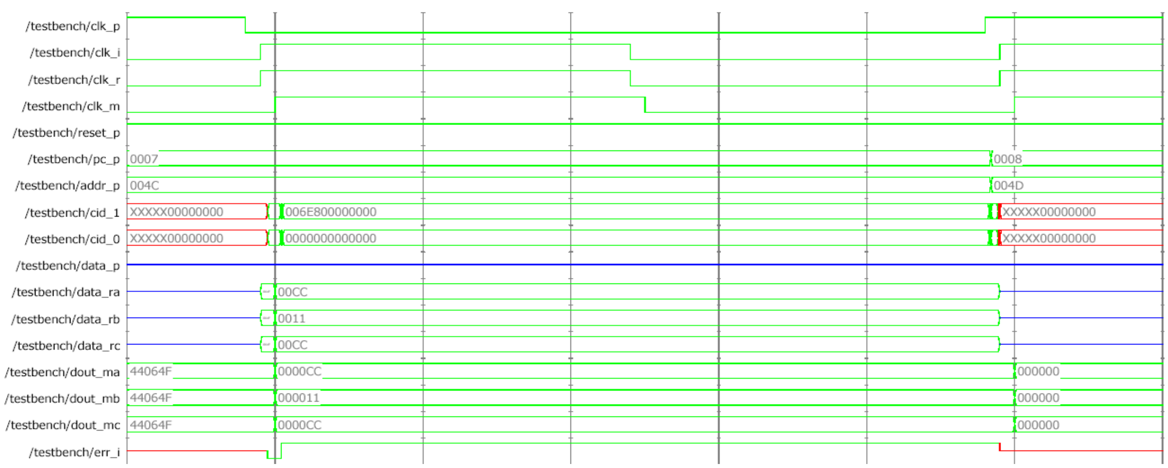


Figure 73. Detail Timing at point 5 in previous simulation

Back to Figure 72, point 3 is the first instruction fetched in the ISR. At the same time the KDLX is fetching this instruction, the *ESSD* triggers *stall_s* at point 4 to stall the processors. In the next clock cycle, the muxes are switched to zeros and 0059_{16} appears on the address bus to the *RAM*s.

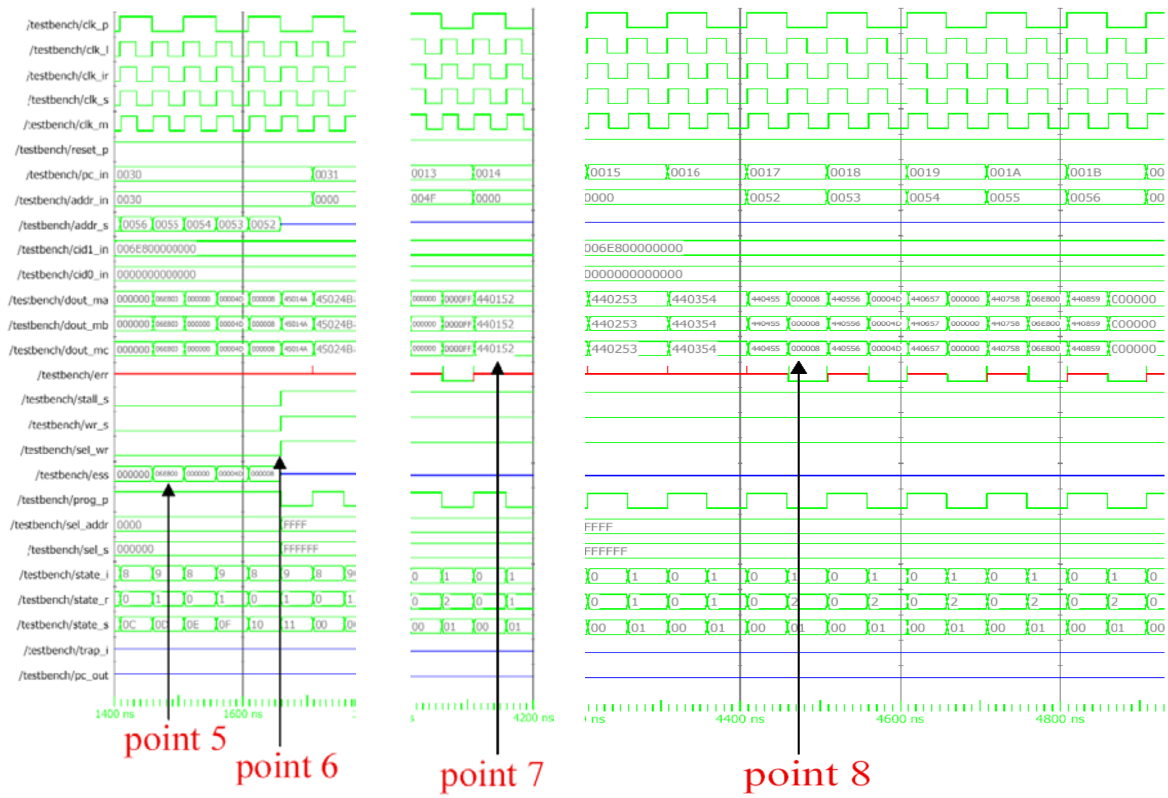


Figure 74. Simulation of the Full Design with *ESSD* (continued)

Following the algorithm explained in Figure 69, the bus *ess* at point 5 proves this function works. Once the *ESSD* finishes at point 6, it gives all of the buses back and releases the processors. The first instruction of the ISR starts in the next clock cycle.

Extra instructions in the *RAM*s are for loading error syndromes stored in memory back to the registers for checking purposes. These instructions start at point 7 and the output data at point 8 proves that all values are stored correctly.

C. CHAPTER SUMMARY

All components for a complete design have been introduced. The reason for not discussing the *ESSD* until this chapter is to simplify the simulation. There were too many things that needed to be explained in the simulation result if the *ESSD* is not described separately. This would make the whole simulation look complicated and may not emphasize the importance of the ISR. Introducing the *ESSD* separately means that the functions of the *Reconciler*, *Interrupt*, and *ESSD* are shown clearly in all simulations.

Not a conceptual design, this full design was simulated and checked. Design of these components can be improved and more information is needed for a better performance of the TMR system. These topics for follow-on research will be discussed in the next chapter.

X. CONCLUSIONS AND FOLLOW-ON RESEARCH

This thesis has described the design of a premiere TMR design on an FPGA for the CFTP. Major components have been defined in previous theses but most of them had to be redesigned due to more understanding of the KDLX processor. Each component was simulated to prove its function. Some timing issues were discussed when different components were connected with each other. The full design has proved the ability to detect and correct an SEU in simulation as well.

A. OVERVIEW

The TMR Assembly consists of three KDLX processors and voters in order to detect and correct errors. A majority voter can only handle one error per time. Since the TMR Assembly has several voters in it, it is able to report errors on different signals simultaneously. For example, *cid_1* and *cid_0* buses of the TMRA can identify errors on the program counter and data at the same time. The processor causing errors on the program counter may not be the same one that generates errors on data.

In order to coordinate memory access, the *Reconciler* is built to consolidate the Harvard and Von Neumann architectures. It runs twice as fast as the KDLX clock cycle and has instruction memory access first followed by the data memory access second. This component purely implements read and write access with memory and does not relate directly to error detection or correction. The *Interrupt* provides an ISR to correct any inconsistency in registers between the three processors. This unit is triggered when an error is found by the *TMRA*. If an error is caused somewhere on the bus but not inside registers, the ISR will still be triggered but no error will be found. An error syndrome records the program counter, the memory address, and any inconsistent bits on data, address, program counter, read, write and program read in *cid* buses. This information is latched in *ESSD* and will be stored to memory during the ISR. Analyzing error syndromes can help a designer to correct or fix the current design.

B. CONCLUSIONS

A simple flow chart in Figure 75 illustrates the overall procedure to correct an error in TMR. The role of each component in the full design can be understood clearly. The *Interrupt* is generated for error correction purpose only and the *ESSD* is for storing error syndromes only.

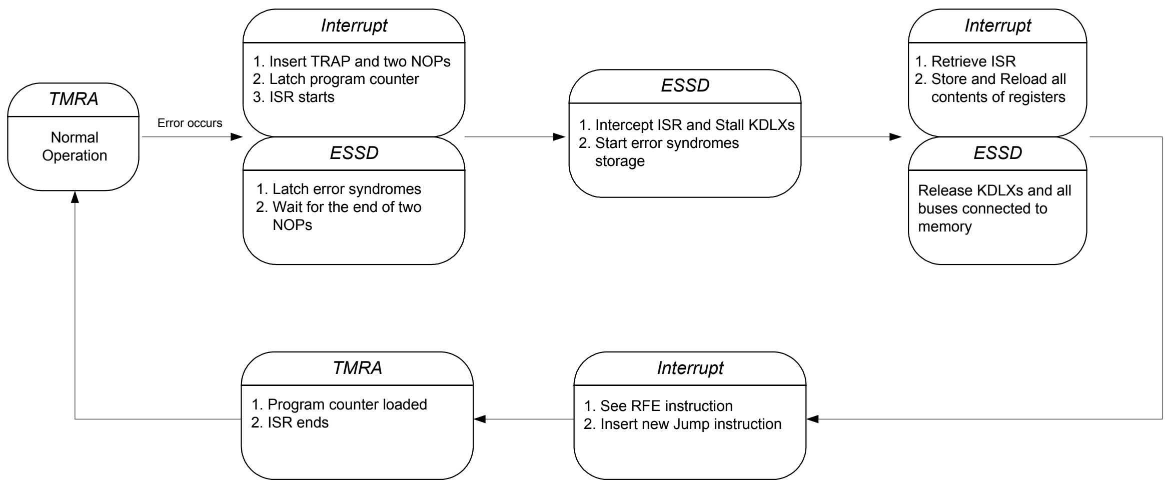


Figure 75. Flowchart of Error Correction for TMR design

A reprogrammable space device such as CFTP has a great potential for the future. The TMR on an FPGA functions as a SOC which saves space on board and offers the flexibility of modification. Utilizing the TMR design with some other features makes the CFTP act as an error-free device. Its powerful feature of reconfigurability widens its usage in missions and lets the state-of-the-art technology be applied to many applications.

C. FOLLOW-ON RESEARCH

A premiere functioning TMR design is complete. This circuit was simulated and proved on software. It is possible to instantiate this design onto a development board to verify its function. Before doing that, some modifications need to be done. Performance of each component can be improved as well. Furthermore, using a faster soft-core processor to speed up the overall performance of the TMR is inevitable.

1. Modification on Current Design

Most components like *Reconciler*, *Interrupt* and *ESSD* are essentially state machines coded in VHDL. It is possible to have these three in one big state machine since they all run in double speed. One needs to have a clear mind on the different functions of the different components in order to do this. Debugging this kind of big state machine needs to be carefully done since any modification on one state may affect functions on other states. On the other hand, there are several different ways to code a component. Other methodologies sometimes are better than using a state machine depending on characteristics of these different components.

A voter error is not considered in this thesis due to time constraints. This kind of error does not need to trigger the ISR. When a voter votes incorrectly, the output is not trustful. The data can be either discarded or re-voted based on the situation. The *ESSD* may need to be revised so as not to save all error syndromes in order to save memory space.

The memory selected for the simulation is based on the availability of the ISE software. If possible, a real Von Neumann architecture memory should be built. Modifications on the *TMRA* and *Reconciler* will be necessary at that time. The real environment on the development board must be considered before these modifications. This avoids duplicate work and makes it possible to compare the simulation result on software with the one on hardware.

An SEU can occur anywhere in the TMR design. More issues need to be solved if this error occurs on the *Reconciler*, *Interrupt* or *ESSD*. Increasing the reliability also increases the probability of having an SEU. The trade-off between these conditions needs more discussion.

2. Faster Processors

Several requirements are considered when searching for a faster processor. First, The new processor has to be faster than the current 16-bit RISC KDLX. Second, it has to be a soft-core processor. Third, it needs to be compatible with Xilinx Virtex XCV800 HQ240 FPGA selected for the CFTP. Other features such as using cache or Harvard architecture can be reconsidered.

Many soft-core processors nowadays use cache to improve their performance even though it is possible to have an SEU in it. Detecting and correcting an SEU in a cache cannot use the same method as with the registers. The contents of the caches need to be reloaded by some method. Study of the SEE on a Pentium®[5] III processor proves that utilizing cache in different ways can change the testing result dramatically [12]. Therefore, it is possible to take advantage of cache without increasing the probability of having an error, and consideration of future processors should include ones with cache.

Using a Von Neumann architecture processor would simplify the TMR design. The *Reconciler* can be removed and less control in *TMRA* are needed for the data bus. Table 21 lists some candidate commercial processors that are currently available.

Commercial Processors			
Company	Processor	Architecture	Features
Xilinx	MicroBlaze	32-bit RISC	1. No cache 2. Harvard bus
ARM	ARM7TDML	32-bit RISC	1. Most have cache 2. Von Neumann bus 3. Hard core
MIPS	MIPS64 5Kc(5Kf)	64-bit RISC	1. Programmable cache 0-64KB 2. Co-processor interface 3. Floating-point pipline 4. Hard core
MIPS	MIPS64 20Kc	64-bit RISC	1. 32KBcaches 2. Superscalar 3. Hard core
Sandcraft	SR71010B	64-bit RISC	1. MIPS64 based 2. L1 32KB cache
Tensilica	Xtensa	32-bit RISC	1. Local data and instruction caches
Altera	Nios	32-bit RISC	1. Instruction master is a 16-bit wide, latency-aware Avalon bus master 2. Configurable cache size
ARC	ARCtangent-A4	32-bit RISC	1. Processor can be configured with Harvard bus architecture (separate instruction/data buses) or a von Neumann bus architecture (unified instruction/data buses) 2. User-configurable instruction and data cache

Table 21. Commercial Soft-Core Processors

[5] Pentium is a registered trademark of Intel Corporation.

Some processors have configurable cache which gives the user some flexibility. The advantage and disadvantage between a soft-core and a hard-core processor has been described in Chapter I so no hard-core processors are considered. Candidates for the TMR are MicroBlaze, SR71010B, Xtensa, Nios, and ARCtangent-A4.

Commercial processors are always expensive because of the proprietary issues. Sometimes these processors come with their own development kit which makes implementation on other software impossible. Part of the design of a commercial processor is sometimes protected by the company and not accessible for the user. Even though revising a processor is not always required, studying source code is a good and fast way to understand the processor itself. On the other hand, information of these commercial processors is limited since only the data sheet on the Internet can be found most of the time.

Sometimes people share their invention or modification of cores with the public. These cores may or may not be fully tested and usually the designer is looking for other people to test it. These cores are called OpenCores. OpenCores are free and can be easily downloaded from the Internet. The disadvantage of using OpenCores is that they are hard to use. Some designers do not describe their design in detail and development tools vary from different designers. People post their questions on the website and hope someone will answer it. Therefore, there is no customer support like the commercial processors. Some Opencores are collected in Table 22.

Some information is not complete due to the lack of description by designers or other users. These cores do not have many restrictions and can be modified if desired. Based on the information found, the SPARC and RISC R1000 are very common processors. The RISC R1000 has been tested and successfully ran a video image program. Many devices are also compatible with this processor. The RISC R1200 is almost an identical processor with R1000 except for the cache inside. The Yellow Star which is actually the MIPS32 R3000 processor is known as a very powerful processor. It has been tested by many users as well.

OpenCores		
Architecture	Name	Features
SPARC V8	LEON VHDL 32 bit	1. AMBA AHB and APB on-chip buses 2. Data cache is a direct-mapped cache configurable to 1-64 kbyte
SPARC V7	ERC32 32 bit	1. A radiation-tolerant processor developed for space applications 2. Two platforms are supported: SPARC Solaris-2.5.1 (or higher),and x86 linux (libc5) 3. VHDL model runs on Unix systems
RISC	OpenRisc R1000 32 bit	1. Tested on Xess XSV800 and Flextronics Semiconductor development boards
RISC	OpenRisc R1200 32 bit	1. Tested on Xess XSV800 and Flextronics Semiconductor development boards 2. cache
RISC	Yellow Star (MIPS32 R3000) 32 bit	1. Capable of executing 32bit instructions based on the MIPS R3000 microprocessor instruction set and has been tested running large blocks of compiled C code. 2. Fully functional and compatible interrupt system. Can handle all exceptions cleanly and correctly. 3. On-chip cache control and Memory Management Unit
RISC	Risc 16f84	1. The "risc16f84_clk2x.v" core has been coded completely, synthesized and tested for correct operation (and debugged!) inside a Xilinx XC2S200 FPGA
RISC	Plasma	1. Support interrupts and all MIPS I(TM) user mode instructions except unaligned load and store operations (which are patented) and exceptions which can be easily avoided. 2. Tested on an Altera FPGA running at 16.5 MHz (synthesized for 29.8 MHz) 3. Currently running on an Altera EP20K200EFC484-2X FPGA and a Xilinx FPGA

Table 22. OpenCores

These OpenCores are tested and proved with certain FPGAs. In order to use these processors in the TMR design, more study and research on source codes are required. Finally, they will need to be tested and simulated on the ISE software before any design work related to the TMR.

APPENDIX A: SCHEMATICS

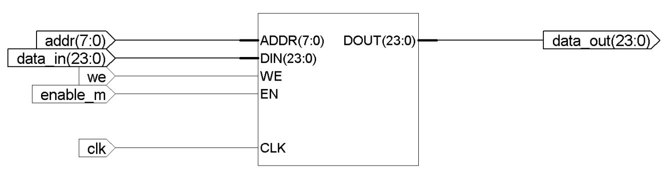
Appendix A contains all schematics, test benches and simulation results of the components in this thesis. Simple schematic symbols are introduced as figures and are not included here. Features and settings of each component and test bench are briefed as well. The long test bench is chopped into pieces and only the important parts are shown. Sometimes a different expression is used in order to explain how a component will be tested.

The simulation result is always shown completely. Important parts that need to be explained are duplicated or modified in contents. All values used in the test bench and the simulation result are hexadecimal and R0 is always zero.

A. 24-BIT MEMORY

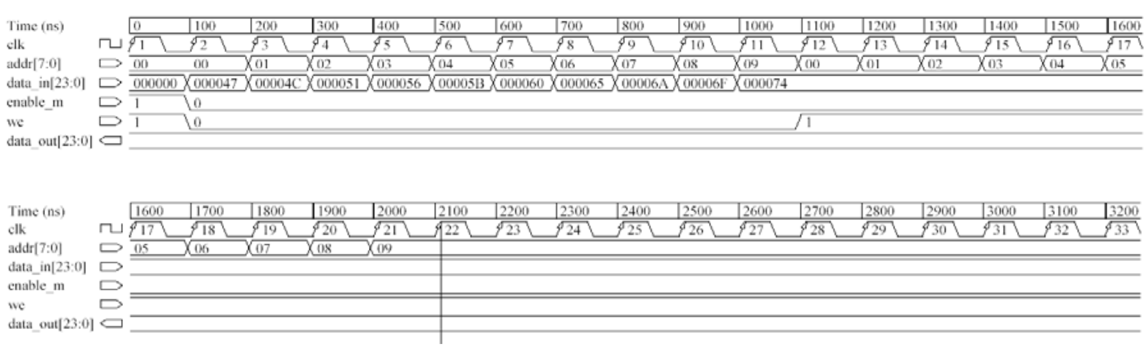
1. Schematic

This memory is a RAM. It is triggered at the rising clock edge. Both write enable (i.e., WE) and memory enable (i.e., EN) pins are active low. Default value of this memory is zero.

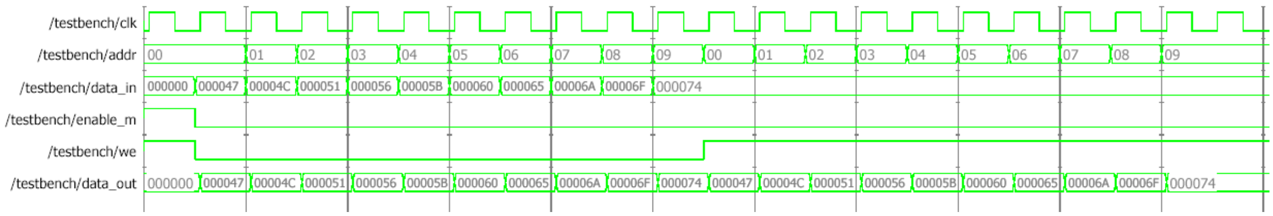


2. Test Bench

This test bench was originally in a single row. It is cut into two rows in order to fit the paper size. The vertical line at time 2100 ns is the stop point of the simulation. Clock high time and low time is 50 ns. Input setup time and output valid delay is 10 ns.

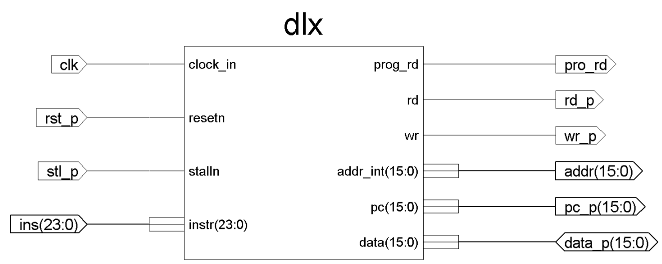


3. Simulation Result



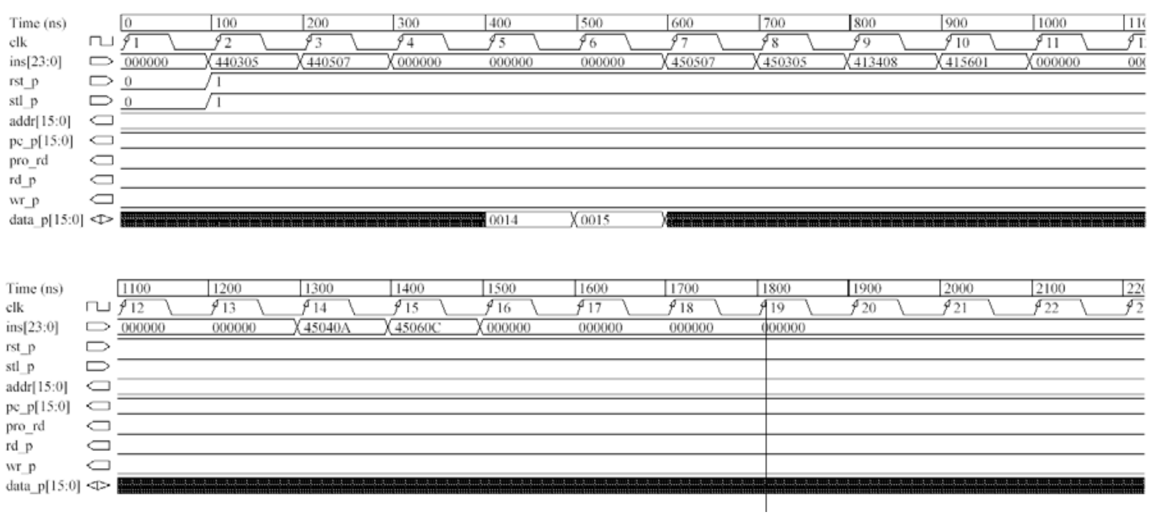
B. KDLX WITHOUT MEMORY

1. Schematic

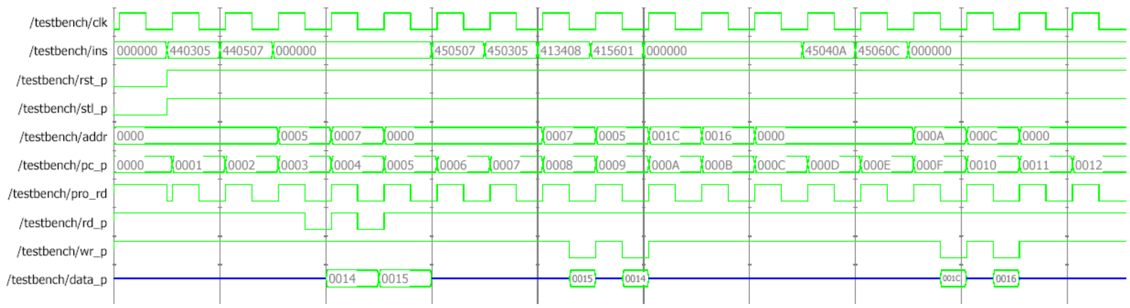


2. Test Bench

The data bus is high impedance. Two values are offered at clock 5 and 6 for KDLX to load into registers. Clock high time and low time is 50 ns. Input setup time and output valid delay is 10 ns.



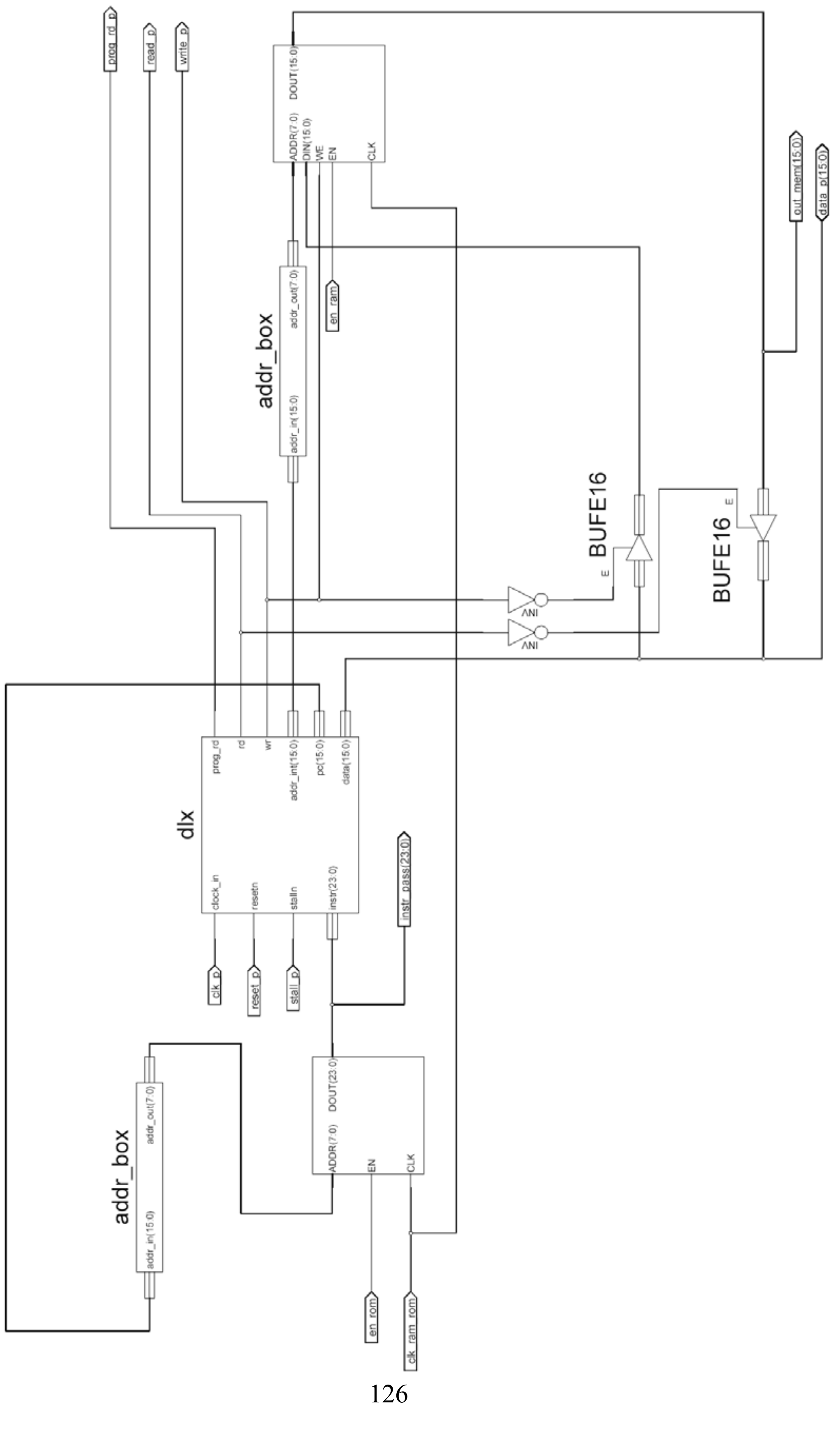
3. Simulation Result



C. KDLX WITH MEMORIES

1. Schematic

The instruction memory at the left side is a ROM. The data memory at the right side is a RAM. Data memory is pre-configured with 0003_{16}. Both memories are triggered at the rising clock edge.

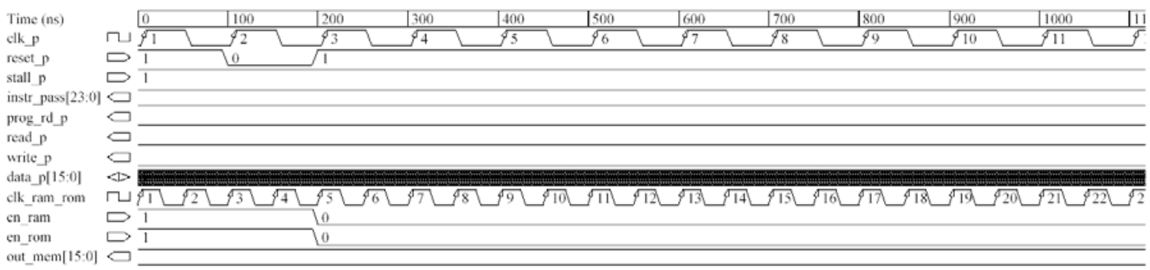


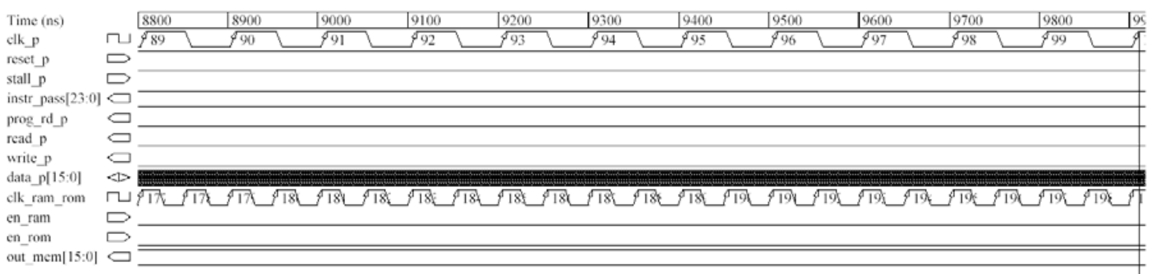
prog_rd_p
read_p
write_p
ADDR(7:0)
DIN(15:0)
WE
EN
CLK
DOUT(15:0)
addr_box
addr_in(15:0)
addr_out(7:0)
en_ram
out_mem(15:0)
data_p(15:0)
BUFE16
E
BUFE16
E
INV
INV
dlx
prog_rd
rd
wr
addr_int(15:0)
pc(15:0)
data(15:0)
clock_in
resetn
stalln
instr(23:0)
instr_pass(23:0)
clk_p
reset_p
stall_p
ADDR(7:0)
DOUT(23:0)
EN
CLK
addr_box
addr_in(15:0)
addr_out(7:0)
en_rom
clk_ram_rom

2. Test Bench of Instruction Set

For the processor, clock high time and low time is 50 ns; input setup time and output valid delay is 10 ns. For memories, all timing settings are half of the processor clock. The bi-directional bus is high impedance.

Nothing special is needed in the test bench thus only the first and last parts are shown here. The KDLX is reset and memories are enabled at time 200 ns. Since the instruction is configurable, the test benches for all instructions sets are the same.





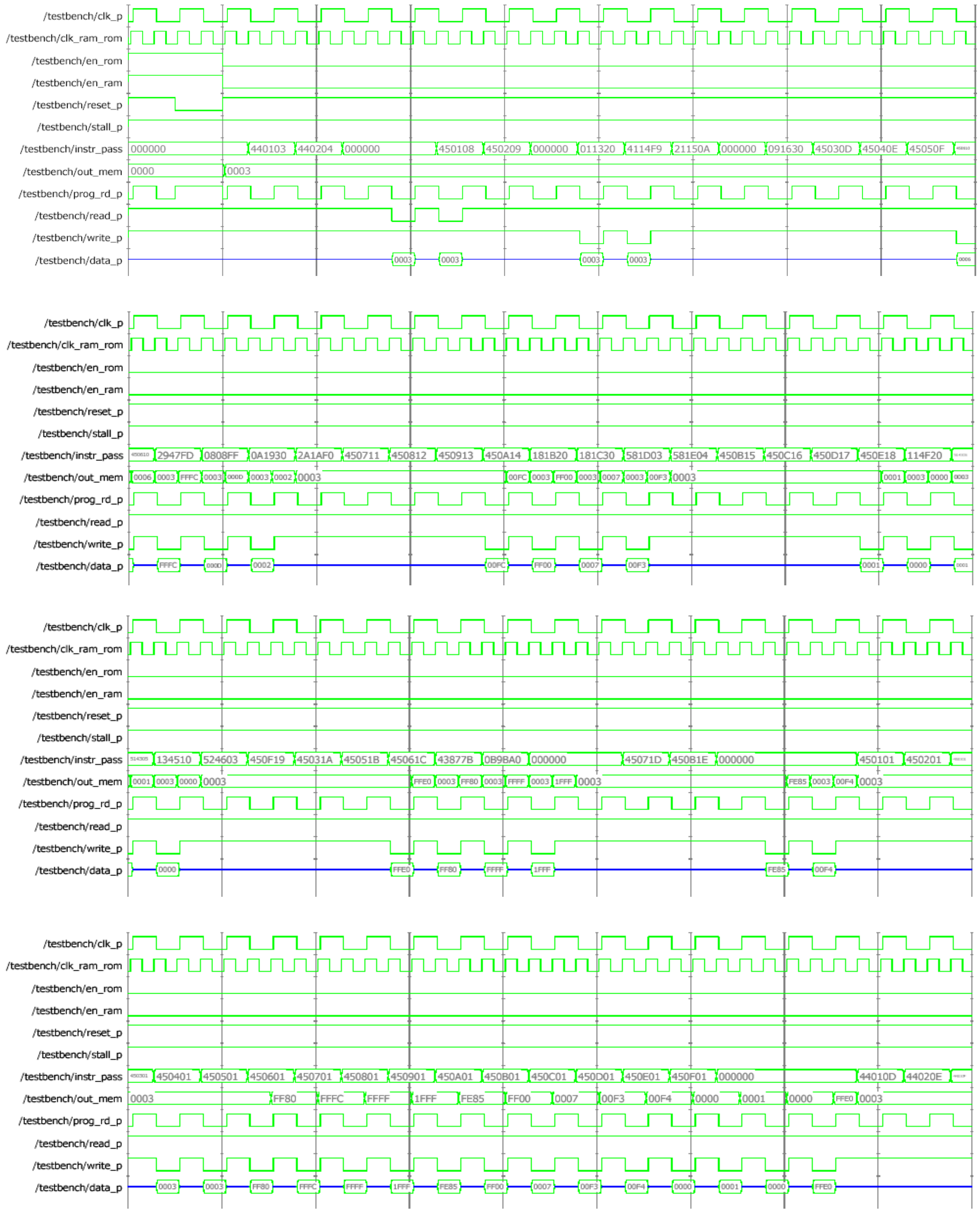
3. Tables and Simulation Results of Instruction Sets

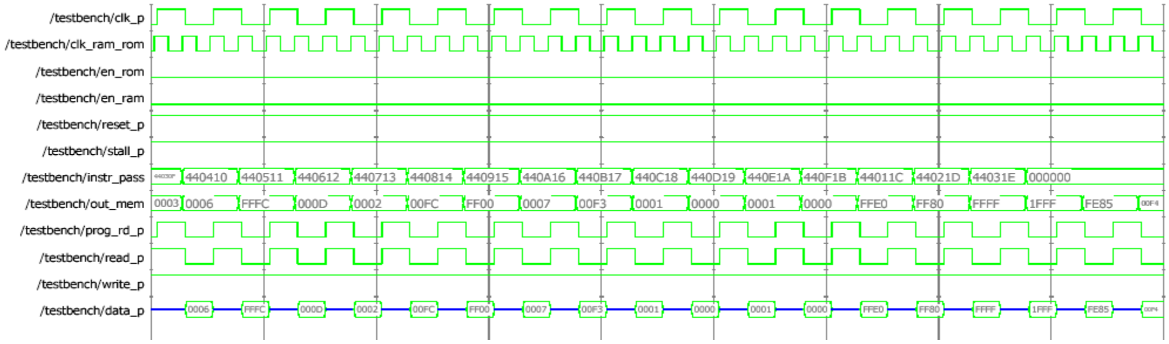
a. Implementation Table of Instruction Set 1

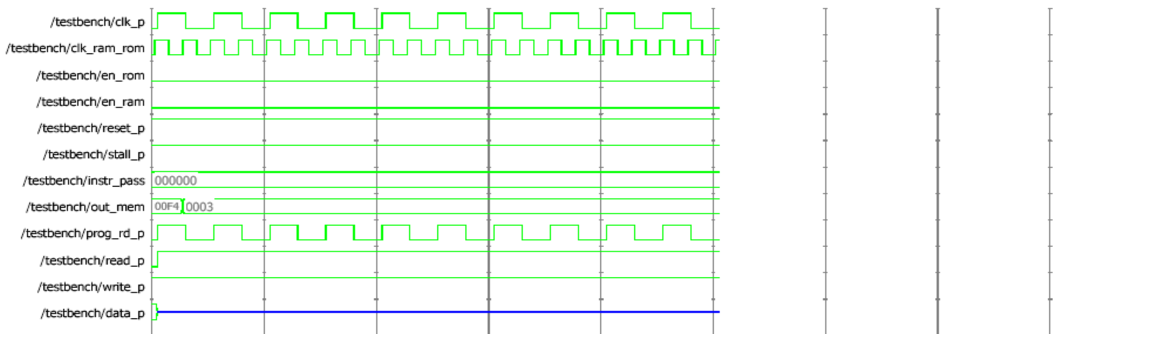
Instruction (operation symbol)		Opcode	Expected Value
LW	R1←Mem(R0+03)	440103	
SW	R1→Mem(R0+08)	450108	0003
LW	R2←Mem(R0+04)	440204	
SW	R2→Mem(R0+09)	450209	0003
ADD	R1+R2→R3	011320	
SW	R3→Mem(R0+0D)	45030D	0006
ADDI	R1+ext(F9)→R4	4114F9	
SW	R4→Mem(R0+0E)	45040E	FFFC
ADDUI	R1+(0A) →R5	21150A	
SW	R5→Mem(R0+0F)	45050F	000D
AND	R1•R3→R6	091630	
SW	R6→Mem(R0+10)	450610	0002
ANDI	R4•(FD)→R7	2947FD	
SW	R7→Mem(R0+11)	450711	00FC

Instruction (operation symbol)		Opcode	Expected Value
LHI	$R8 \leftarrow FF \Vert (0)^8$	0808FF	
SW	$R8 \rightarrow Mem(R0+12)$	450812	FF00
OR	$R1+R3 \rightarrow R9$	0A1930	
SW	$R9 \rightarrow Mem(R0+13)$	450913	0007
ORI	$R1+(F0) \rightarrow R10$	2A1AF0	
SW	$R10 \rightarrow Mem(R0+14)$	450A14	00F3
SEQ	$R1=R2 \rightarrow R11=1$	181B20	
SW	$R11 \rightarrow Mem(R0+15)$	450B15	0001
SEQ	$R1 \neq R3 \rightarrow R12=0$	181C30	
SW	$R12 \rightarrow Mem(R0+16)$	450C16	0000
SEQI	$R1=(0003) \rightarrow R13=1$	581D03	
SW	$R13 \rightarrow Mem(R0+17)$	450D17	0001
SEQI	$R1 \neq (0004) \rightarrow R14=0$	581E04	
SW	$R14 \rightarrow Mem(R0+18)$	450E18	0000
SLL	$R4^{\leftarrow R2=(0003)} \rightarrow R15$	114F20	
SW	$R15 \rightarrow Mem(R0+19)$	450F19	FFE0
SLLI	$R4^{\leftarrow (0005)} \rightarrow R3$	514305	
SW	$R3 \rightarrow Mem(R0+1A)$	45031A	FF80
SRA	$R4^{\rightarrow R1=(0003)} \rightarrow R5$	134510	
SW	$R5 \rightarrow Mem(R0+1B)$	45051B	FFFF
SRLI	$R4^{\rightarrow (0003)} \rightarrow R6$	524603	
SW	$R6 \rightarrow Mem(R0+1C)$	45061C	1FFF
SUBI	$R8 - ext(7B) \rightarrow R7$	43877B	
SW	$R7 \rightarrow Mem(R0+1D)$	45071D	FE85
XOR	$R9 \oplus R10 \rightarrow R11$	0B9BA0	
SW	$R11 \rightarrow Mem(R0+1E)$	450B1E	00F4

b. Simulation Result of Instruction Set 1




/testbench/clk_p
/testbench/clk_ram_rom
/testbench/en_rom
/testbench/en_ram
/testbench/reset_p
/testbench/stall_p
/testbench/instr_pass
/testbench/out_mem
/testbench/prog_rd_p
/testbench/read_p
/testbench/write_p
/testbench/data_p
440410
440511
440612
440713
440814
440915
440A16
440B17
440C18
440D19
440E1A
440F1B
44011C
44021D
44031E
000000
0003
0006
FFFC
000D
0002
00FC
FF00
0007
00F3
0001
0000
0001
0000
FFE0
FF80
FFFF
1FFF
FE85


/testbench/clk_p
/testbench/clk_ram_rom
/testbench/en_rom
/testbench/en_ram
/testbench/reset_p
/testbench/stall_p
/testbench/instr_pass
/testbench/out_mem
/testbench/prog_rd_p
/testbench/read_p
/testbench/write_p
/testbench/data_p
000000
00F4
0003

c. Tables of Registers and Memories in Simulation 1

Instruction Mem			
00		2D	45071D
01	440103	2E	450B1E
02	440204	2F	000000
03	000000	30	000000
04	000000	31	000000
05	450108	32	450101
06	450209	33	450201
07	000000	34	450301
08	011320	35	450401
09	4114F9	36	450501
0A	21150A	37	450601
0B	000000	38	450701
0C	091630	39	450801
0D	45030D	3A	450901
0E	45040E	3B	450A01
0F	45050F	3C	450B01
10	450610	3D	450C01
11	2947FD	3E	450D01
12	0808FF	3F	450E01
13	0A1930	40	450F01
14	2A1AF0	41	000000
15	450711	42	000000
16	450812	43	000000
17	450913	44	44010D
18	450A14	45	44020E
19	181B20	46	44030F
1A	181C30	47	440410
1B	581D03	48	440511
1C	581E04	49	440612
1D	450B15	4A	440713
1E	450C16	4B	440814
1F	450D17	4C	440915
20	450E18	4D	440A16
21	114F20	4E	440B17
22	514305	4F	440C18
23	134510	50	440D19
24	524603	51	440E1A
25	450F19	52	440F1B
26	45031A	53	44011C
27	45051B	54	44021D
28	45061C	55	44031E
29	43877B	56	000000
2A	0B9BA0	57	000000
2B	000000	58	000000
2C	000000	59	000000

Register		
00		
01	0003	
02	0003	
03	~~0006~~	FF80
04	FFFC	
05	~~000D~~	FFFF
06	~~0002~~	1FFF
07	~~00FC~~	FE85
08	FF00	
09	0007	
10	00F3	
11	~~0001~~	00F4
12	0000	
13	0001	
14	0000	
15	FFE0	

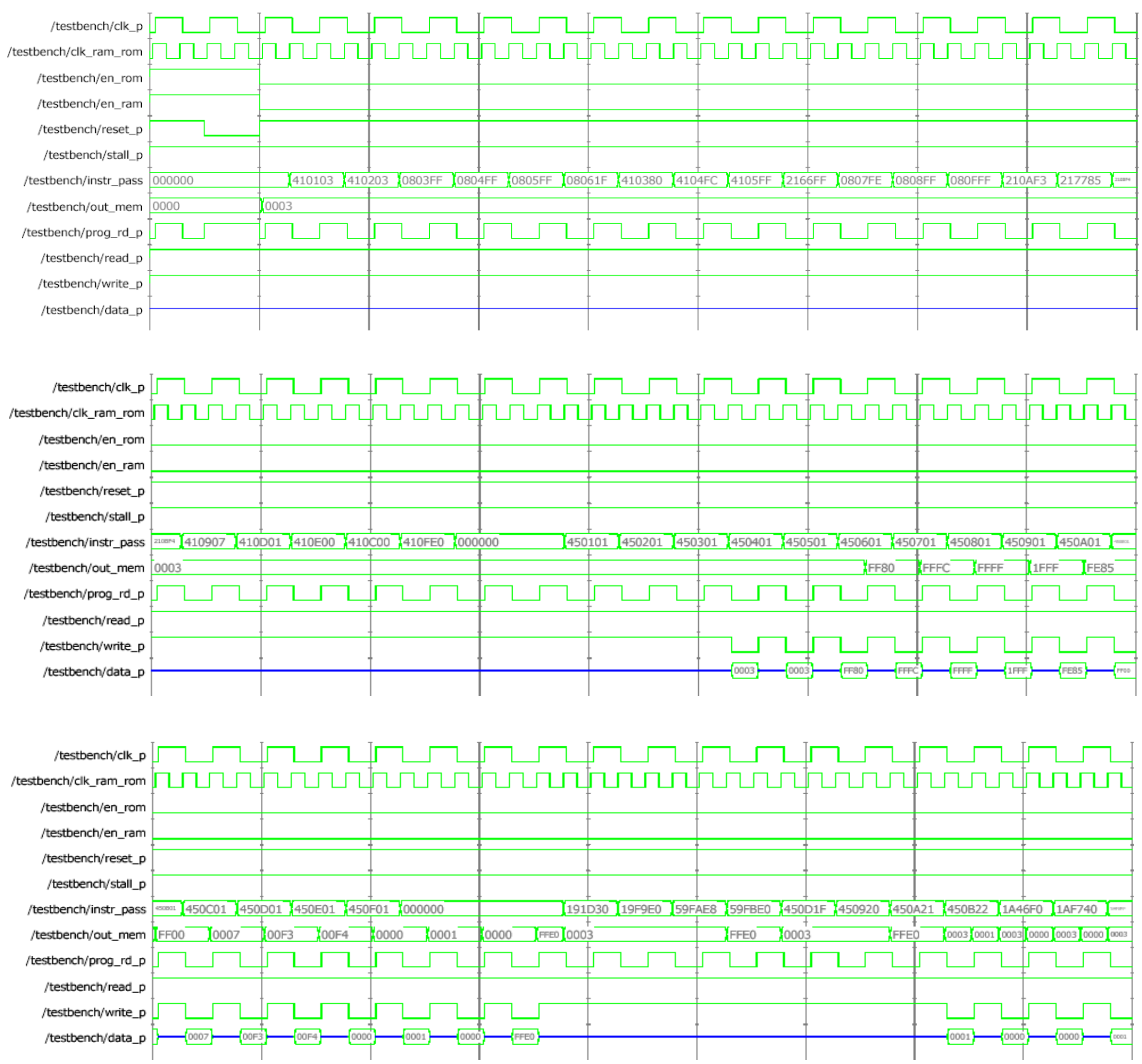
Data Mem	
00	
01	
02	
03	
04	
05	
06	
07	
08	0003
09	0003
0A	
0B	
0C	
0D	0006
0E	FFFC
0F	000D
10	0002
11	00FC
12	FF00
13	0007
14	00F3
15	0001
16	0000
17	0001
18	0000
19	FFE0
1A	FF80
1B	FFFF
1C	1FFFF
1D	FE85
1E	00F4
1F	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
2A	

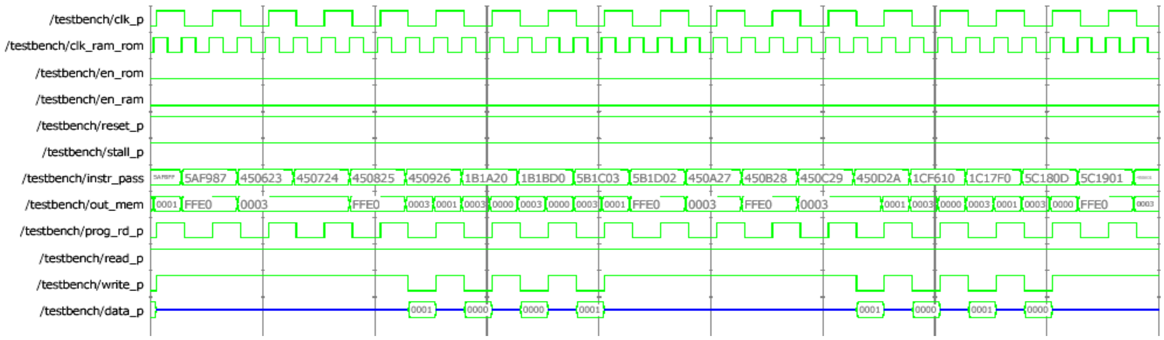
d. Implementation Table of Instruction Set 2

Instruction (pseudo code)		Opcode	Expected Value
SGE	R1>R3→R13=1	191D30	
SW	R13→Mem(R0+1F)	450D1F	0001
SGE	R15>R14→R9=0	19F9E0	
SW	R9→Mem(R0+20)	450920	0000
SGEI	R15≥ext(E8)→R10=0	59FAE8	
SW	R10→Mem(R0+21)	450A21	0000
SGEI	R15≥ext(E0) →R11=1	59FBE0	
SW	R11→Mem(R0+22)	450B22	0001
SGT	R4>R15→R6=1	1A46F0	
SW	R6→Mem(R0+23)	450623	0001
SGT	R15>R4→R7=0	1AF740	
SW	R7→Mem(R0+24)	450724	0000
SGTI	R15>ext(FF)→R8=0	5AF8FF	
SW	R8→Mem(R0+25)	450825	0000
SGTI	R15>ext(87)→R9=1	5AF987	
SW	R9→Mem(R0+26)	450926	0001
SLE	R1=R2→R10=1	1B1A20	
SW	R10→Mem(R0+27)	450A27	0001
SLE	R1<R13→R11=0	1B1BD0	
SW	R11→Mem(R0+28)	450B28	0000
SLEI	R1≤ext(03)→R12=1	5B1C03	
SW	R12→Mem(R0+29)	450C29	0001
SLEI	R1≤ext(02)→R13=0	5B1D02	
SW	R13→Mem(R0+2A)	450D2A	0000
SLT	R15<R1→R6=1	1CF610	
SW	R6→Mem(R0+01)	450601	0001
SLT	R1<R15→R7=0	1C16F0	
SW	R7→Mem(R0+02)	450702	0000
SLTI	R1<ext(0D)→R8=1	5C180D	
SW	R8→Mem(R0+03)	450803	0001
SLTI	R1<ext(01)→R9=0	5C1901	
SW	R9→Mem(R0+04)	450904	0000
SNE	R1≠R2→R10=0	1D1A20	
SW	R10→Mem(R0+05)	450A05	0000
SNE	R1≠R15→R11=1	1D1BF0	
SW	R11→Mem(R0+06)	450B06	0001
SNEI	R1≠ext(03)→R12=1	581C03	
SW	R12→Mem(R0+07)	450C07	0001
SNEI	R15≠ext(E1)→R13=0	58FDE1	
SW	R13→Mem(R0+08)	450D08	0000
SRAI	$R3^{\rightarrow(0006)}$→R6	533606	

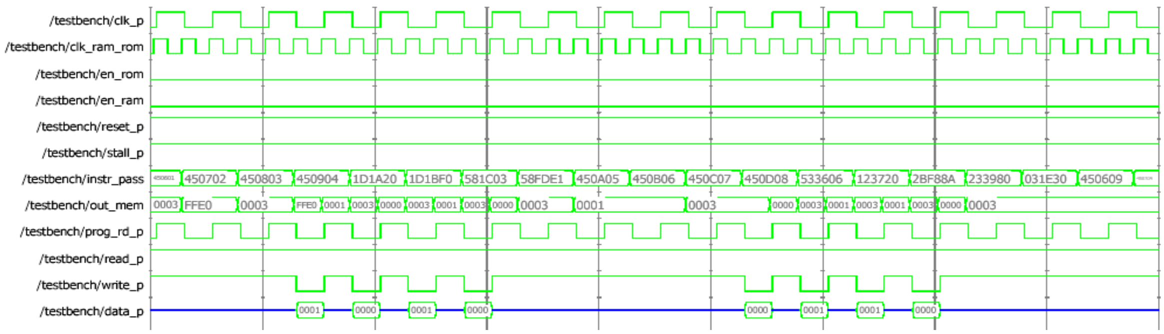
Instruction (pseudo code)		Opcode	Expected Value
SW	R6→Mem(R0+09)	450609	FFFE
SRL	$R3^{\rightarrow R2=(0003)}$→R7	123720	
SW	R7→Mem(R0+0A)	45070A	1FF0
XORI	R15⊕(8A)→R8	2BF88A	
SW	R8→Mem(R0+0B)	45080B	FF6A
SUBUI	R3–(80)→R9	233980	
SW	R9→Mem(R0+0C)	45090C	FF00
SUB	R1–R3→R14	031E30	
SW	R14→Mem(R0+0D)	450E0D	0083

e. Simulation Result of Instruction Set 2

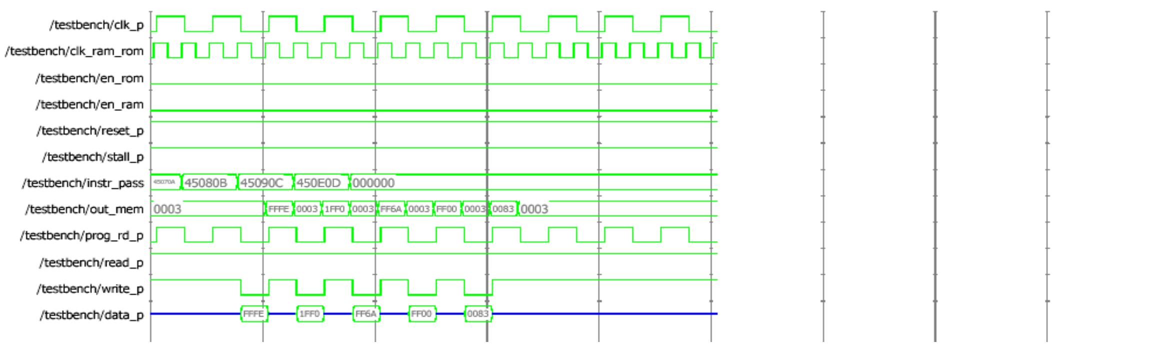




/testbench/clk_p
/testbench/clk_ram_rom
/testbench/en_rom
/testbench/en_ram
/testbench/reset_p
/testbench/stall_p
/testbench/instr_pass
/testbench/out_mem
/testbench/prog_rd_p
/testbench/read_p
/testbench/write_p
/testbench/data_p
5AF987
450623
450724
450825
450926
1B1A20
1B1BD0
5B1C03
5B1D02
450A27
450B28
450C29
450D2A
1CF610
1C17F0
5C180D
5C1901
FFE0
0003



/testbench/clk_p
/testbench/clk_ram_rom
/testbench/en_rom
/testbench/en_ram
/testbench/reset_p
/testbench/stall_p
/testbench/instr_pass
/testbench/out_mem
/testbench/prog_rd_p
/testbench/read_p
/testbench/write_p
/testbench/data_p
450702
450803
450904
1D1A20
1D1BF0
581C03
58FDE1
450A05
450B06
450C07
450D08
533606
123720
2BF88A
233980
031E30
450609
0003
FFE0
0001



/testbench/clk_p
/testbench/clk_ram_rom
/testbench/en_rom
/testbench/en_ram
/testbench/reset_p
/testbench/stall_p
/testbench/instr_pass
/testbench/out_mem
/testbench/prog_rd_p
/testbench/read_p
/testbench/write_p
/testbench/data_p
45080B
45090C
450E0D
000000
0003

f. *Tables of Registers and Memories in Simulation 2*

Instruction Mem			
00		30	450A21
01	410103	31	450B22
02	410203	32	1A46F0
03	0803FF	33	1AF740
04	0804FF	34	5AF8FF
05	0805FF	35	5AF987
06	08061F	36	450623
07	410380	37	450724
08	4104FC	38	450825
09	4105FF	39	450926
0A	2166FF	3A	1B1A20
0B	0807FE	3B	1B1BD0
0C	0808FF	3C	5B1C03
0D	080FFF	3D	5B1D02
0E	210AF3	3E	450A27
0F	217785	3F	450B28
10	210BF4	40	450C29
11	410907	41	450D2A
12	410D01	42	1CF610
13	410E00	43	1C17F0
14	410C00	44	5C180D
15	410FE0	45	5C1901
16	000000	46	450601
17	000000	47	450702
18	450100	48	450803
19	450200	49	450904
1A	450300	4A	1D1A20
1B	450400	4B	1D1BF0
1C	450500	4C	581C03
1D	450600	4D	58FDE1
1E	450700	4E	450A05
1F	450800	4F	450B06
20	450900	50	450C07
21	450A00	51	450D08
22	450B00	52	533603
23	450C00	53	123720
24	450D00	54	2BF88A
25	450E00	55	233980
26	450F00	56	031E30
27	000000	57	450609
28	000000	58	45070A
29	000000	59	45080B
2A	191D30	5A	45090C
2B	19F9E0	5B	450E0D
2C	59FAE8	5C	000000
2D	59FBE0	5D	000000
2E	450D1F	5E	000000
2F	450920	5F	000000

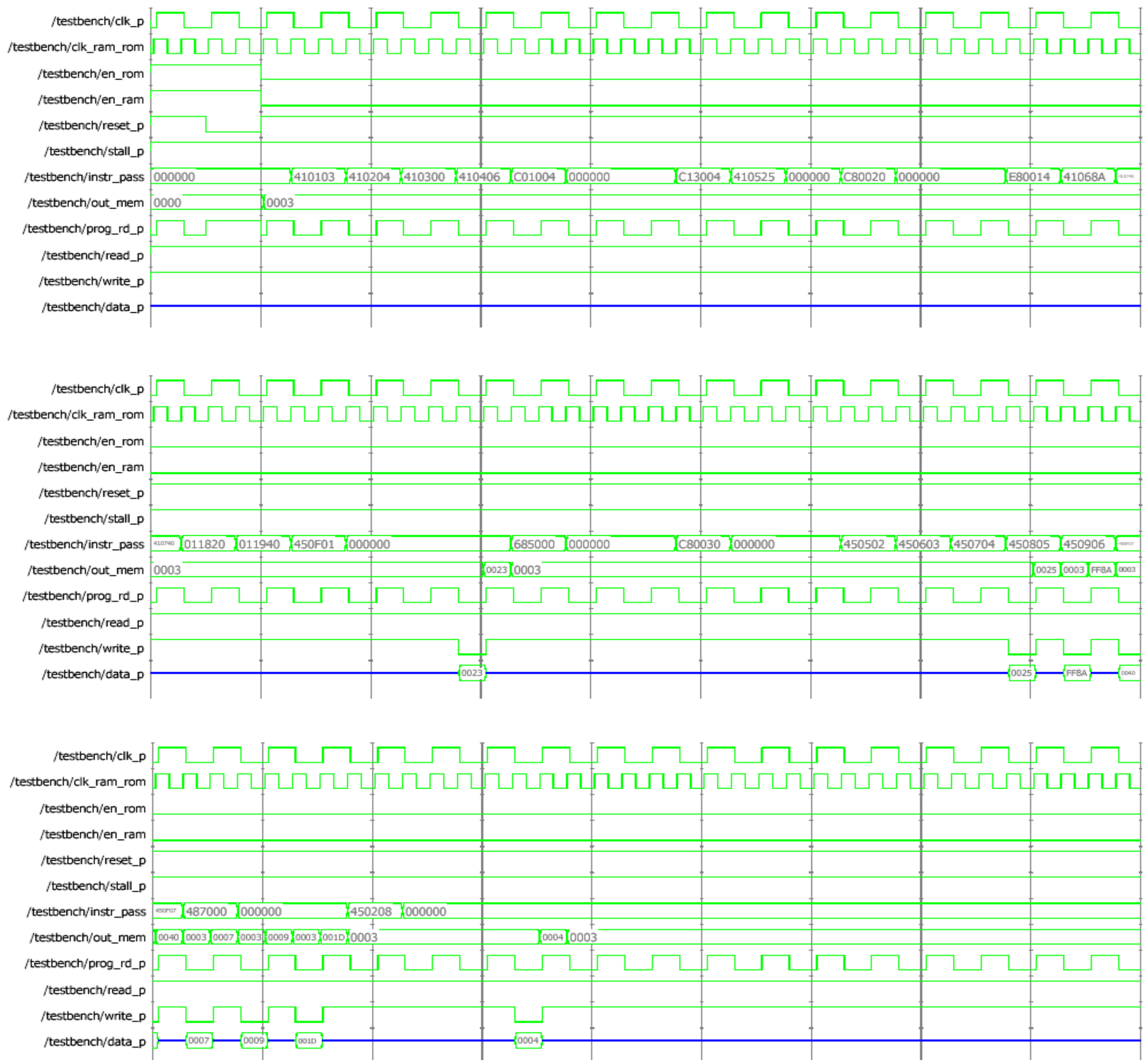
Register		
00		
01	0003	0003
02	0003	0003
03	FF80	FF80
04	FFFC	FFFC
05	FFFF	FFFF
06	1FFF	FFFE
07	FE85	1FF0
08	FF00	FF6A
09	0007	FF00
10	00F3	0000
11	00F4	0001
12	0000	0001
13	0001	0000
14	0000	0083
15	FFE0	FFE0

Data Mem	
00	
01	0001
02	0000
03	0001
04	0000
05	0000
06	0001
07	0001
08	0000
09	FFFE
0A	1FF0
0B	FF6A
0C	FF00
0D	0083
0E	
0F	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
1A	
1B	
1C	
1D	
1E	
1F	0001
20	0000
21	0000
22	0001
23	0001
24	0000
25	0000
26	0001
27	0001
28	0000
29	0001
2A	0000

g. Implementation Table of Instruction Set 3

	Instruction (pseudo code)	Opcode	Expected Value
LW	R1←Mem(R0+03)	410103	
LW	R2←Mem(R0+04)	410204	
LW	R3←Mem(R0+00)	410300	
LW	R4←Mem(R0+06)	410406	
BNEZ	R1≠0→Prog_Addr←(05)+1+ext(04) Note: PC=05 and (05)+1+ext(04)=0A	C01004	
BEQZ	R3=0→Prog_Addr←(0A)+1+ext(04) Note: PC=0A and (0A)+1+ext(04)=0F	C13004	
ADDI	R0+ext(25)→R5	410525	
J	(0020)→Prog_Addr	C80020	
JAL	(0014)→Prog_Addr ; (23)→R15 Note:(23) is return address	E80014	
ADDI	R0+ext(8A)→R6	41068A	
ADDI	R0+ext(40)→R7	410740	
ADD	R1+R2→R8	011820	
ADD	R1+R4→R9	011940	
SW	R15→Mem(R0+01)	450F01	0023
JALR	R5→Prog_Addr ; (1D)→R15 Noter:(1D) is return address	685000	
J	(0030)→Prog_Addr	C80030	
SW	R5→Mem(R0+02)	450502	0025
SW	R6→Mem(R0+03)	450603	FF8A
SW	R7→Mem(R0+04)	450704	0040
SW	R8→Mem(R0+05)	450805	0007
SW	R9→Mem(R0+06)	450906	0009
SW	R15→Mem(R0+07)	450F07	001D
JR	R7→Prog_Addr	487000	
SW	R2→Mem(R0+08)	450208	0004

h.   Simulation Result of Instruction Set 3



i. *Tables of Registers and Memories in Simulation 3*

Instruction Mem	
00	
01	410103
02	410204
03	410300
04	410406
05	C01004
06	000000
07	000000
08	
09	
0A	C13004
0B	410525
0C	000000
0D	
0E	
0F	C80020
10	000000
11	000000
12	
13	
14	011820
15	011940
16	450F01
17	000000
18	000000
19	000000
1A	685000
1B	000000
1C	000000
1D	
1E	
1F	
20	E80014
21	41068A
22	410740
23	
24	
25	C80030
26	000000
27	000000
28	
29	
2A	
30	450502
31	450603
32	450704
33	450805
34	450906
35	450F07
36	487000
37	000000
38	000000
39	
⋮	
40	450208
41	000000
42	000000
43	000000

1D→R15

23→R15

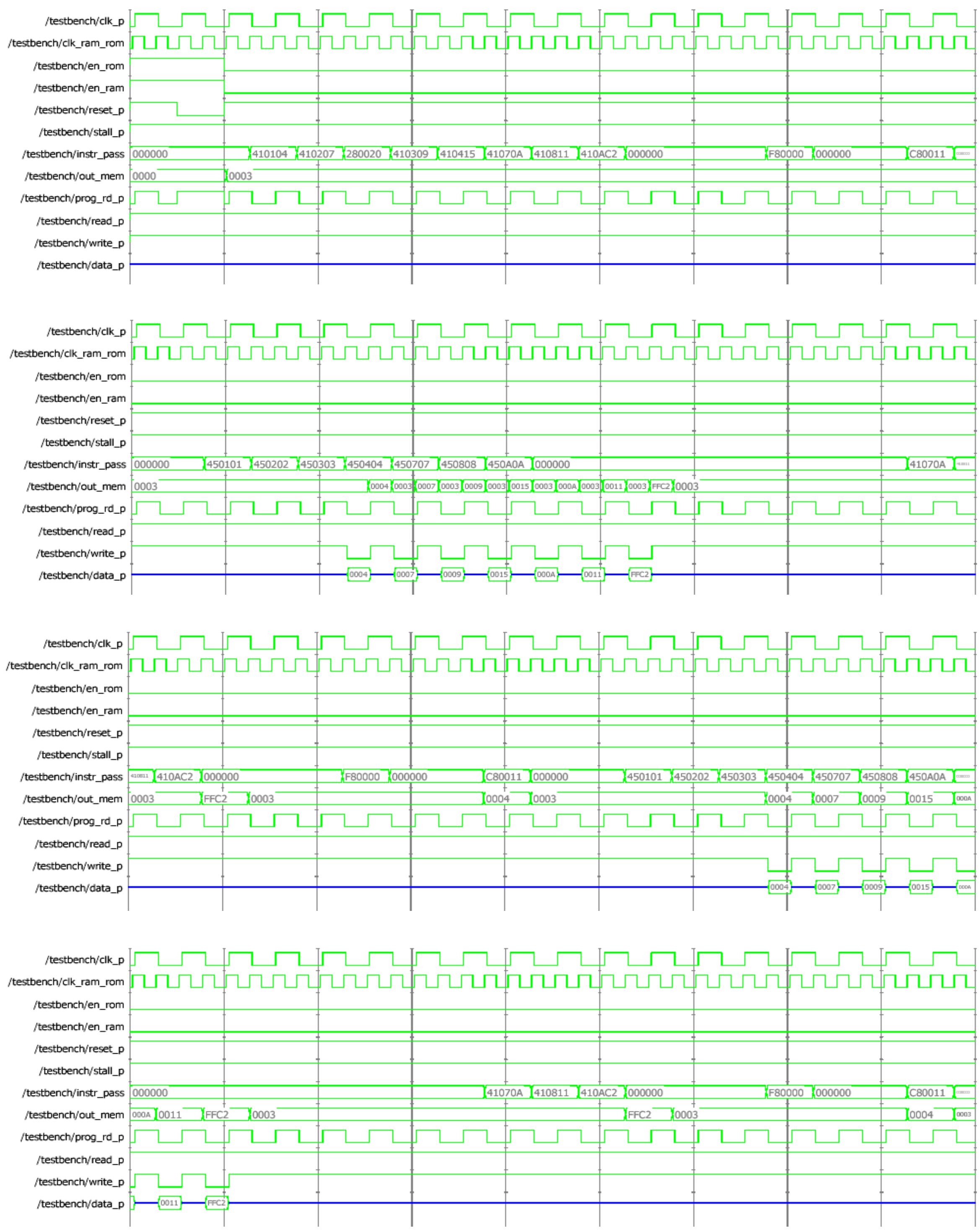
Register	
00	
01	0003
02	0004
03	0000
04	0006
05	0025
06	FF8A
07	0040
08	0007
09	0009
10	
11	
12	
13	
14	
15	

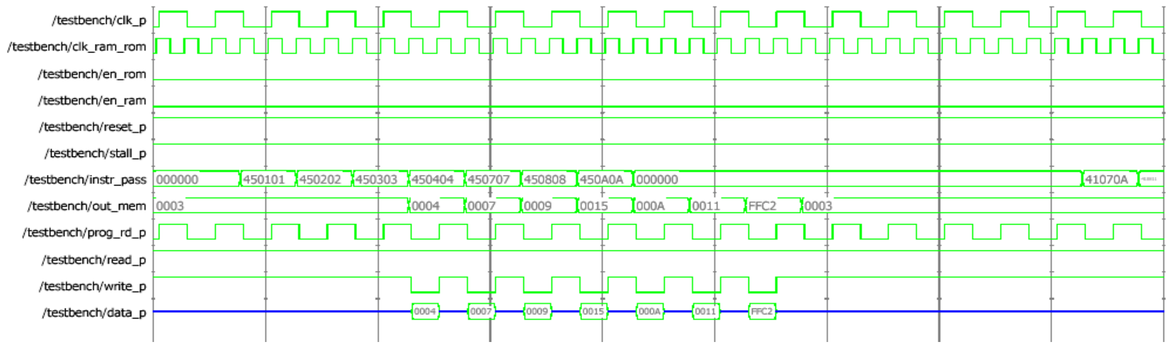
Data Mem	
00	
01	0023
02	0025
03	FF8A
04	0040
05	0007
06	0009
07	001D
08	0004
09	
0A	
0B	
0C	
0D	
0E	
0F	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
1A	
1B	
1C	
1D	
1E	
1F	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
2A	

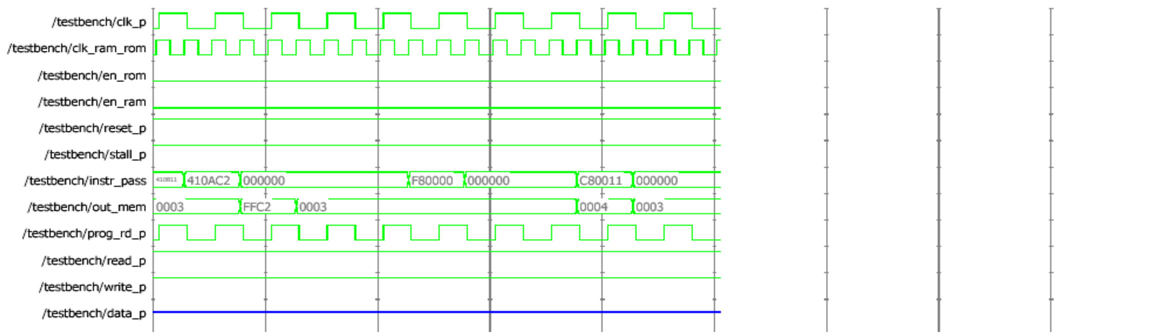
j. Implementation Table of Instruction Set 4

Instruction (operation symbol)		Opcode	Expected Value
ADDI	R0+ext(04)→R1	410104	
ADDI	R0+ext(07)→R2	410207	
TRAP	(0020)→Prog_Addr ; (06)→IAR Note: (06) is return address	280020	
ADDI	R0+ext(09)→R3	410309	
ADDI	R0+ext(15)→R4	410415	
ADDI	R0+ext(0A)→R7	41070A	
ADDI	R0+ext(11)→R8	410811	
ADDI	R0+ext(C2)→R10	410AC2	
RFE	(06)→Prog_Addr Note: (06) is IAR	F80000	
J	(0011)→Prog_Addr	C80011	
SW	R1→Mem(R0+01)	450101	0004
SW	R2→Mem(R0+02)	450202	0007
SW	R3→Mem(R0+03)	450303	0009
SW	R4→Mem(R0+04)	450404	0015
SW	R7→Mem(R0+07)	450707	000A
SW	R8→Mem(R0+08)	450808	0011
SW	R10→Mem(R0+0A)	450A0A	FFC2

k. *Simulation Result of Instruction Set 4*




/testbench/clk_p
/testbench/clk_ram_rom
/testbench/en_rom
/testbench/en_ram
/testbench/reset_p
/testbench/stall_p
/testbench/instr_pass
/testbench/out_mem
/testbench/prog_rd_p
/testbench/read_p
/testbench/write_p
/testbench/data_p
000000 450101 450202 450303 450404 450707 450808 450A0A 000000 41070A
0003 0004 0007 0009 0015 000A 0011 FFC2 0003
0004 0007 0009 0015 000A 0011 FFC2


/testbench/clk_p
/testbench/clk_ram_rom
/testbench/en_rom
/testbench/en_ram
/testbench/reset_p
/testbench/stall_p
/testbench/instr_pass
/testbench/out_mem
/testbench/prog_rd_p
/testbench/read_p
/testbench/write_p
/testbench/data_p
410AC2 000000 F80000 000000 C80011 000000
0003 FFC2 0003 0004 0003

l. Tables of Registers and Memories in Simulation 4

06 → IAR

Instruction Mem	
00	
01	410104
02	410207
03	280020
04	410309
05	410415
06	C80011
07	000000
08	000000
09	
0A	
0B	
0C	
0D	
0E	
0F	
10	
11	450101
12	450202
13	450303
14	450404
15	450707
16	450808
17	450A0A
18	000000
19	000000
1A	000000
1B	
1C	
1D	
1E	
1F	
20	41070A
21	410811
22	410AC2
23	000000
24	000000
25	000000
26	F80000
27	000000
28	000000
29	
2A	

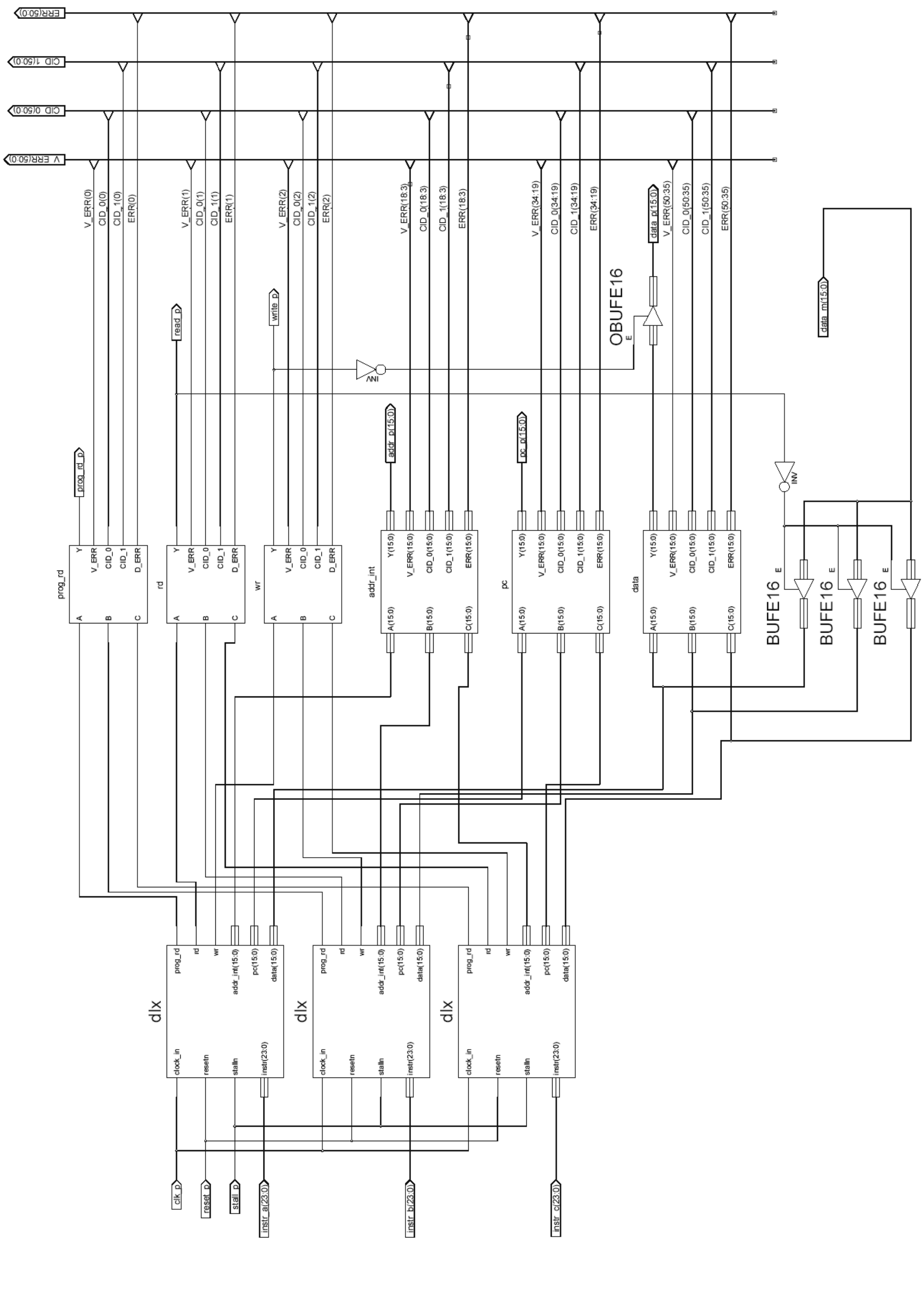
Register	
00	
01	0004
02	0007
03	0009
04	0015
05	
06	
07	000A
08	0011
09	
10	FFC2
11	
12	
13	
14	
15	

Data Mem	
00	
01	0004
02	0007
03	0009
04	0015
05	
06	
07	000A
08	0011
09	
0A	FFC2
0B	
0C	
0D	
0E	
0F	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
1A	
1B	
1C	
1D	
1E	
1F	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
2A	

D. TMR ASSEMBLY WITHOUT MEMORIES

1. Schematic

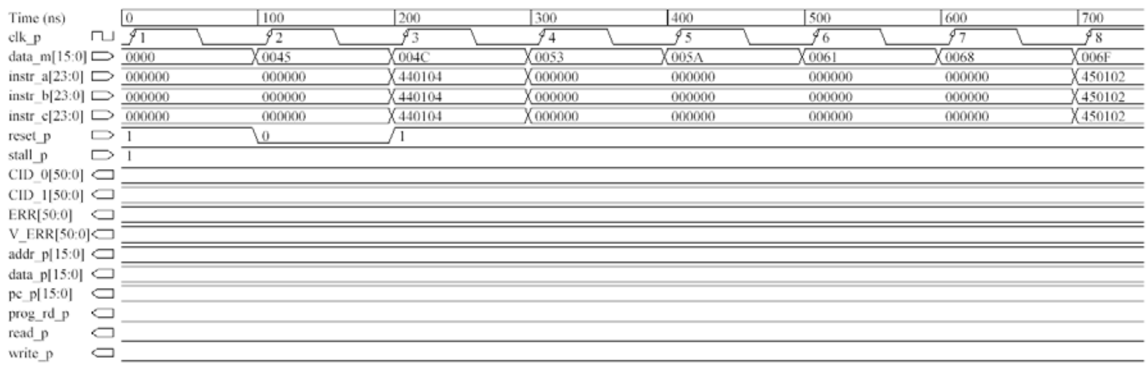
This is the design without the latch at the bottom. Three KDLX processors are at the left and the six voters at the center. Signals such as *V_ERR*, *CID_1*, *CID_0*, and *ERR* are collected individually to four buses at the right. The read signal is used to enable buffers for data from memory. The write signal is used to enable buffers for data to memory.

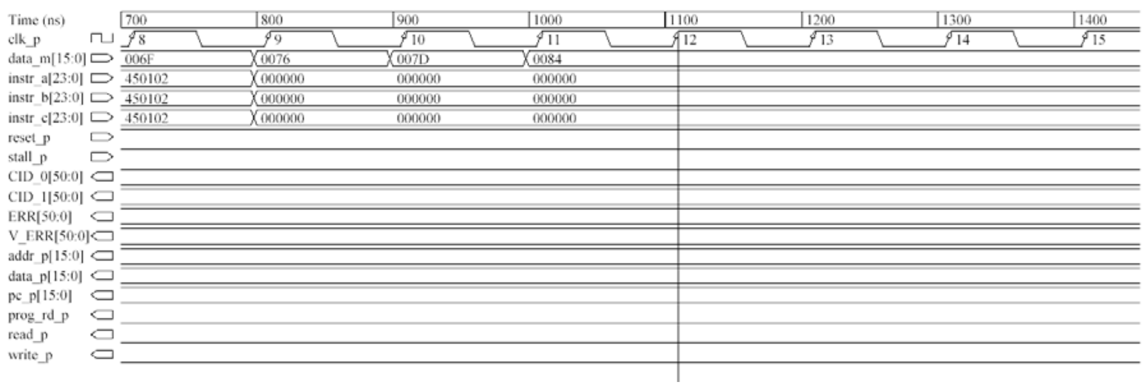


ERR(50:0)
CID_1(50:0)
CID_0(50:0)
V_ERR(50:0)
OBUFE16
BUFE16
BUFE16
BUFE16
INV
INV
prog_rd_p
read_p
write_p
addr_p(15:0)
pc_p(15:0)
data_p(15:0)
data_m(15:0)
prog_rd
rd
wr
addr_int
pc
data
dlx
dlx
dlx
clk_p
reset_p
stall_p
instr_a(23:0)
instr_b(23:0)
instr_c(23:0)

2. Test Bench

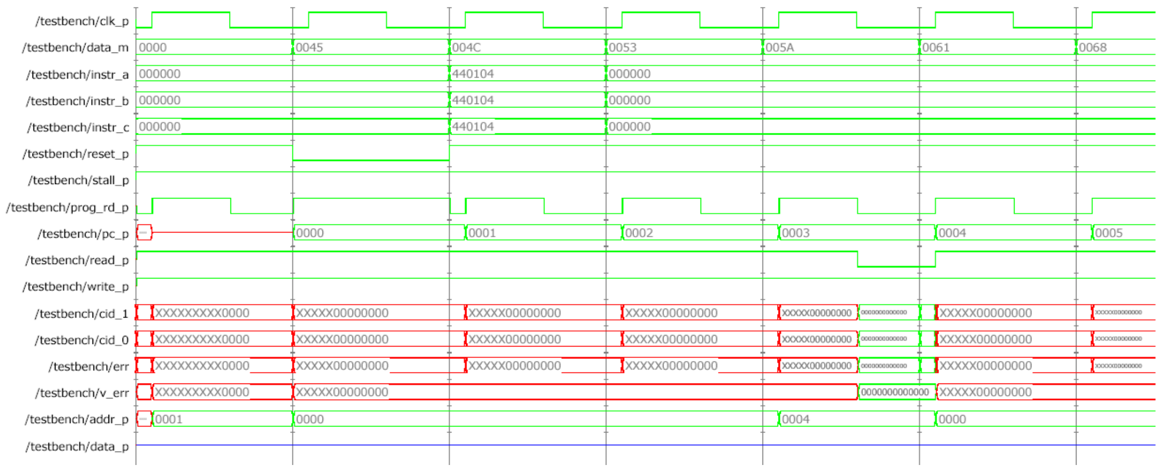
The clock high and low times are each 50 ns. The input setup time and output valid delay times are each 10 ns. Since there are only two instructions, the test bench looks simple. It loads data in registers and stores back to memory to check whether this schematic works properly.

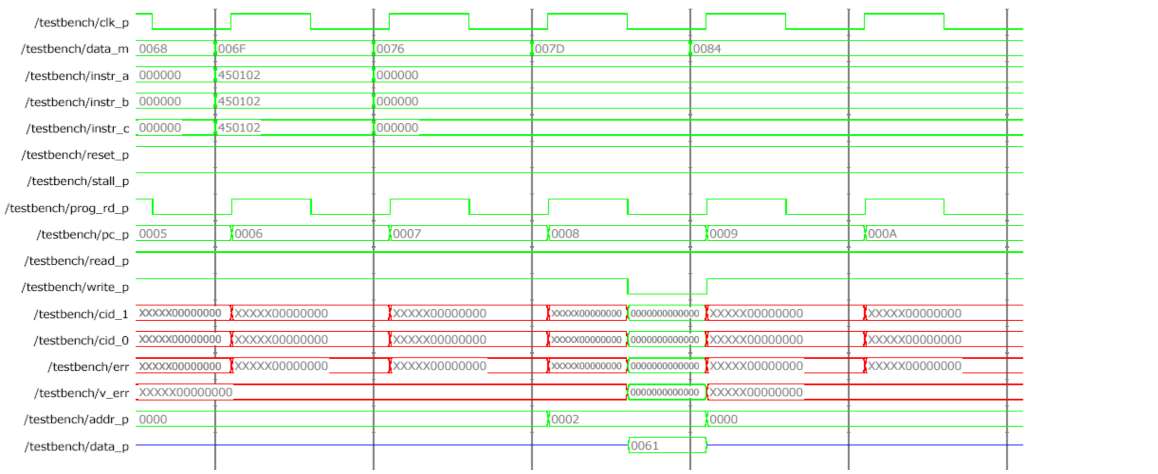




3. Simulation Result

As described in Chapter V this schematic without a latch does not write correct data into the registers due to a timing problem. This kind of error disappears when memories are connected. Because this appendix only displays the final design of each component, the imperfect simulation result is still contained here. The TMR with a latch is discussed in Chapter V so it is not contained here even though it works perfectly without memories.

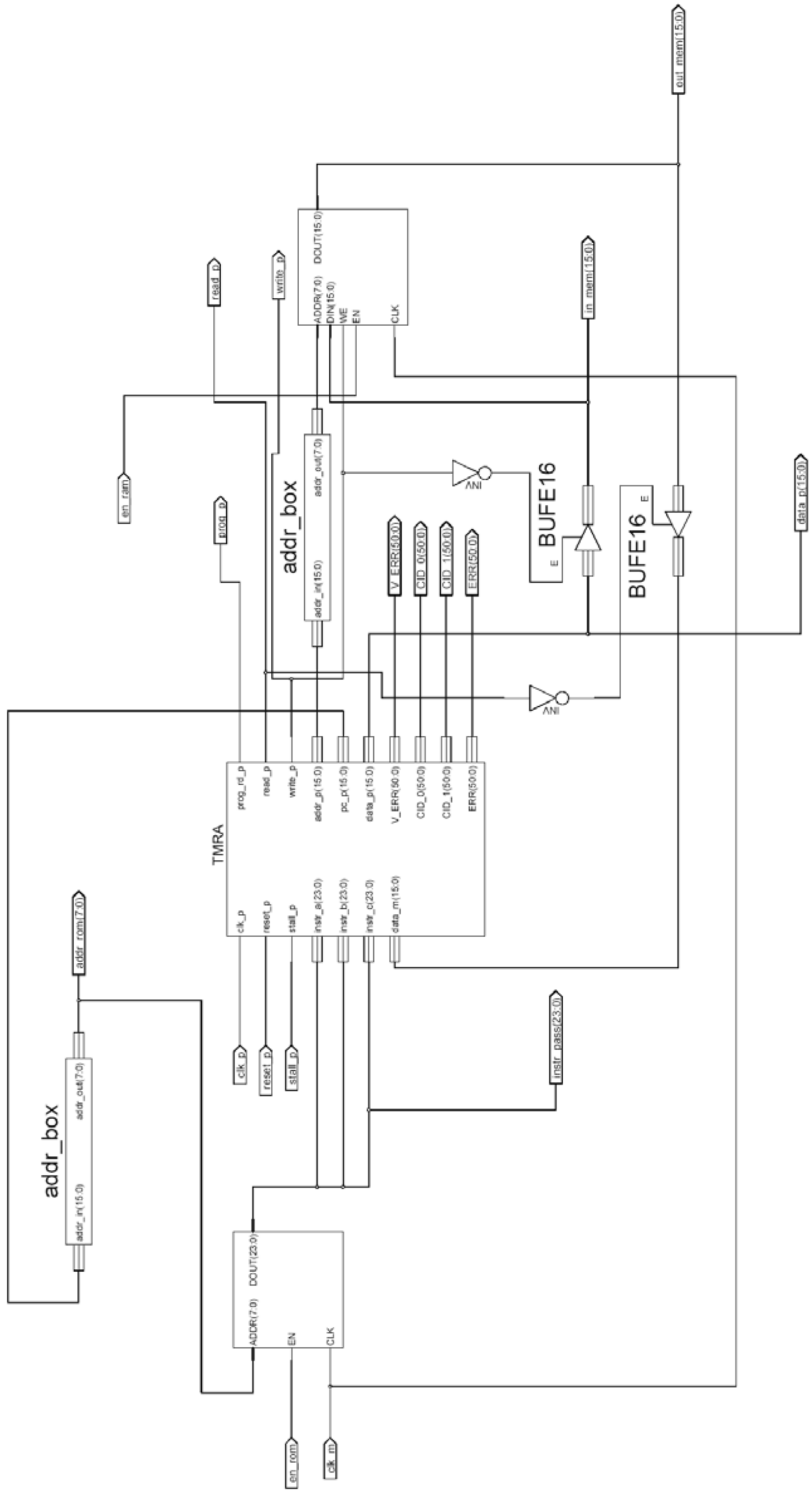




E. TMR ASSEMBLY WITH MEMORIES

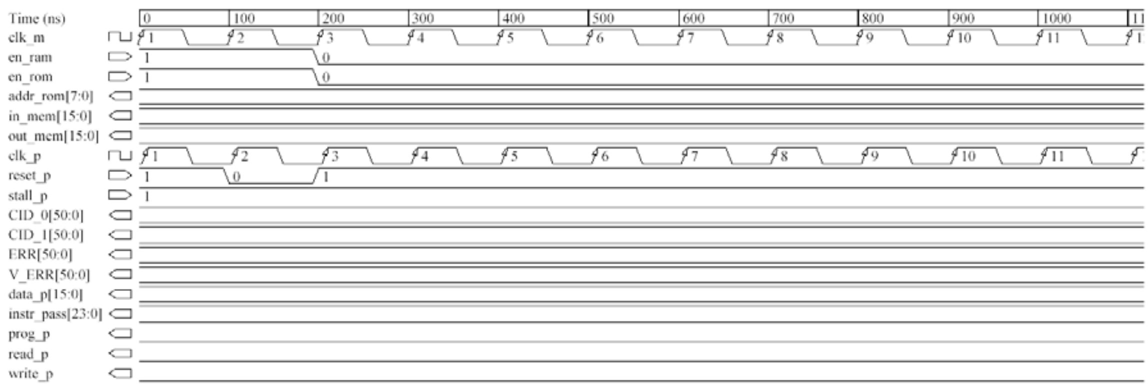
1. Schematic

This schematic uses the TMR Assembly without a latch. The instruction memory on the left side sends one instruction to the three processors at the same time. Therefore, this schematic is used only for checking basic functions. Nothing related with fault tolerant can be tested here.

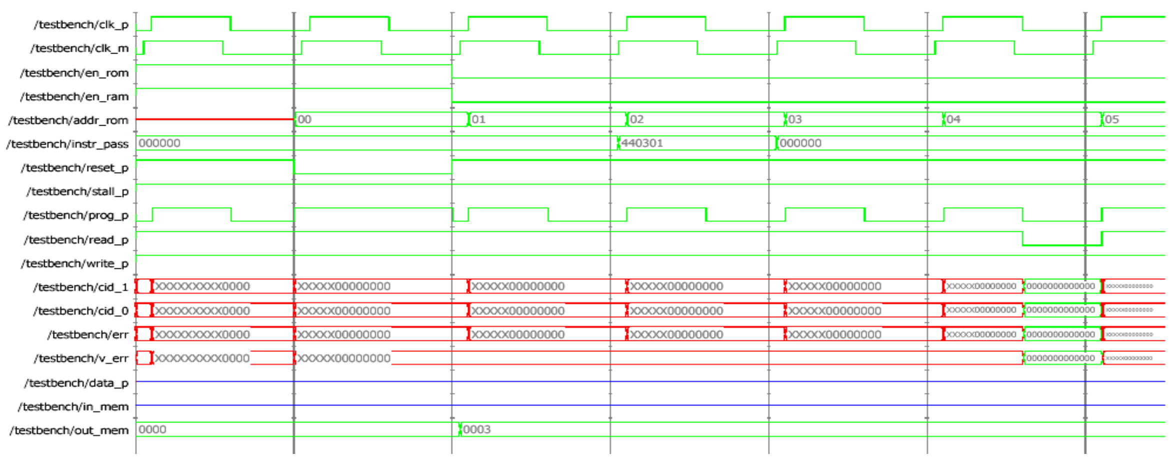

TMRA
addr_box
addr_box
BUFE16
BUFE16
out_mem(15:0)
in_mem(15:0)
read_p
write_p
en_ram
prog_p
data_p(15:0)
V_ERR(50:0)
CID_0(50:0)
CID_1(50:0)
ERR(50:0)
addr_rom(7:0)
instr_pass(23:0)
clk_p
reset_p
stall_p
en_rom
clk_m

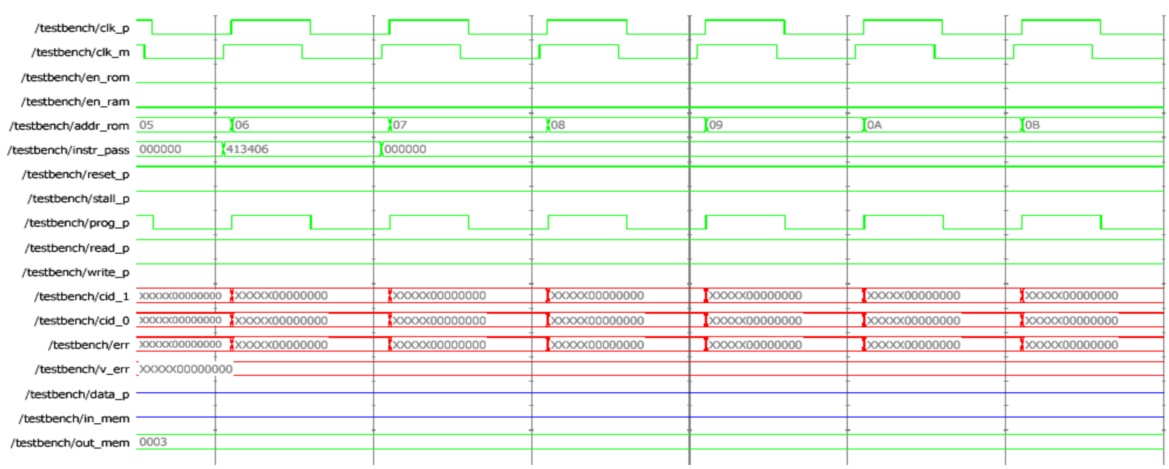
2. Test Bench

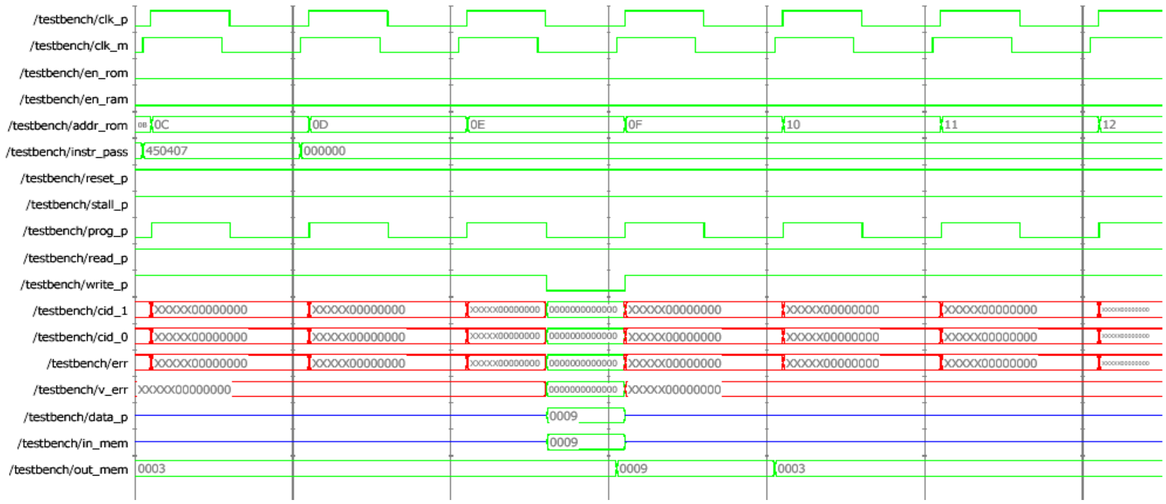
Since the instruction is pre-configured in *ROM* and *RAM* has default value 0003_{16}, no data needs to be assigned. The test bench ends at 2900 ns. The clock high and low times for both memories and processors are each 50 ns. The input setup time and output valid delay are 10 ns for processors and 5 ns for memories.



3. Simulation Result



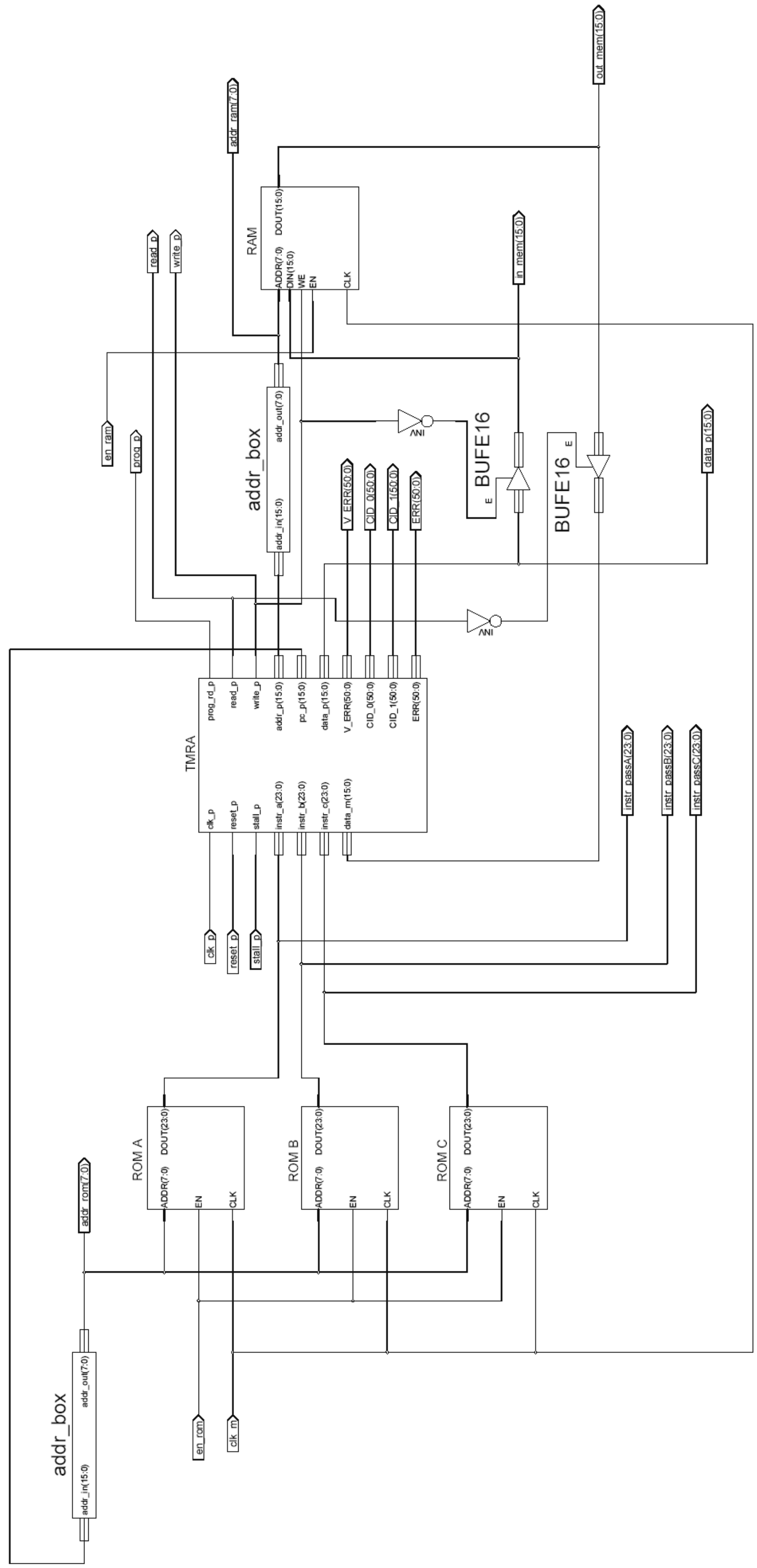




F. FAULT-TOLERANT TESTING

1. Schematic

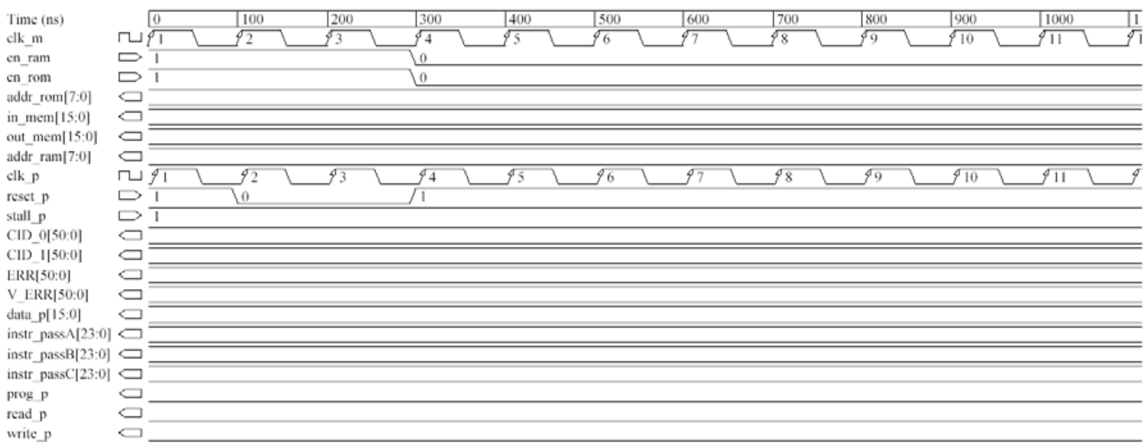
This simulation uses three *ROM*s to achieve the goal of inserting different instructions. This simulates the condition whenever three processors have inconsistent instructions. The *TMRA* can also be modified to connect with three different *RAM*s. Then the simulation will be more complex and much more time needed for analysis. As discussed in Chapter V, such errors should be caught and corrected by the voters as long as no more than one SEU occurs in a voter.



RAM
ADDR(7:0)
DIN(15:0)
WE
EN
CLK
DOUT(15:0)
addr_ram(7:0)
out_mem(15:0)
read_p
write_p
in_mem(15:0)
en_ram
prog_p
addr_box
addr_in(15:0)
addr_out(7:0)
INV
BUFE16
E
data_p(15:0)
V_ERR(50:0)
CID_0(50:0)
CID_1(50:0)
ERR(50:0)
TMRA
prog_rd_p
read_p
write_p
addr_p(15:0)
pc_p(15:0)
data_p(15:0)
V_ERR(50:0)
CID_0(50:0)
CID_1(50:0)
ERR(50:0)
clk_p
reset_p
stall_p
instr_a(23:0)
instr_b(23:0)
instr_c(23:0)
data_m(15:0)
instr_passA(23:0)
instr_passB(23:0)
instr_passC(23:0)
ROM A
ROM B
ROM C
ADDR(7:0)
EN
CLK
DOUT(23:0)
addr_rom(7:0)
en_rom
clk_m
addr_box
addr_in(15:0)
addr_out(7:0)

2. Test Bench

The memories are pre-configured so no special settings are needed in this test bench. The simulation ends at 3400 ns. The clock high and low times for both memories and processors are each 50 ns. The input setup time and output valid delay are 10 ns for processors and 5 ns for memories.

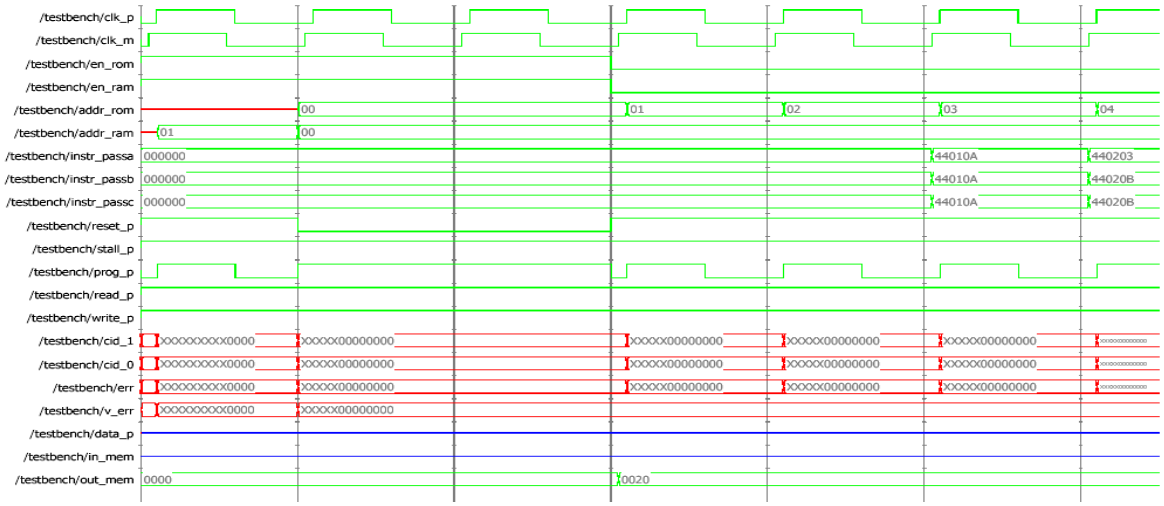


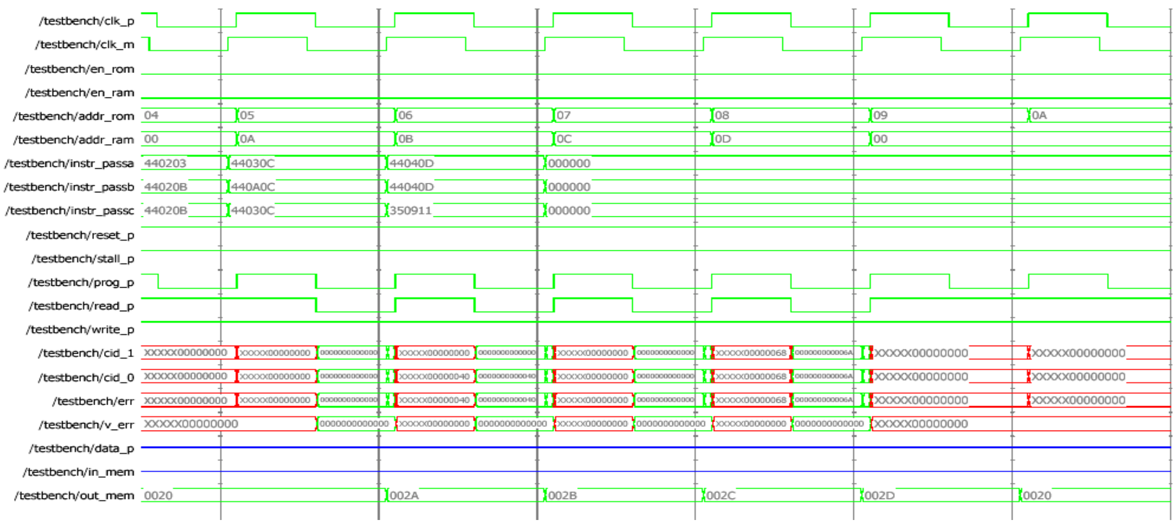
3. Memories Pre-configuration

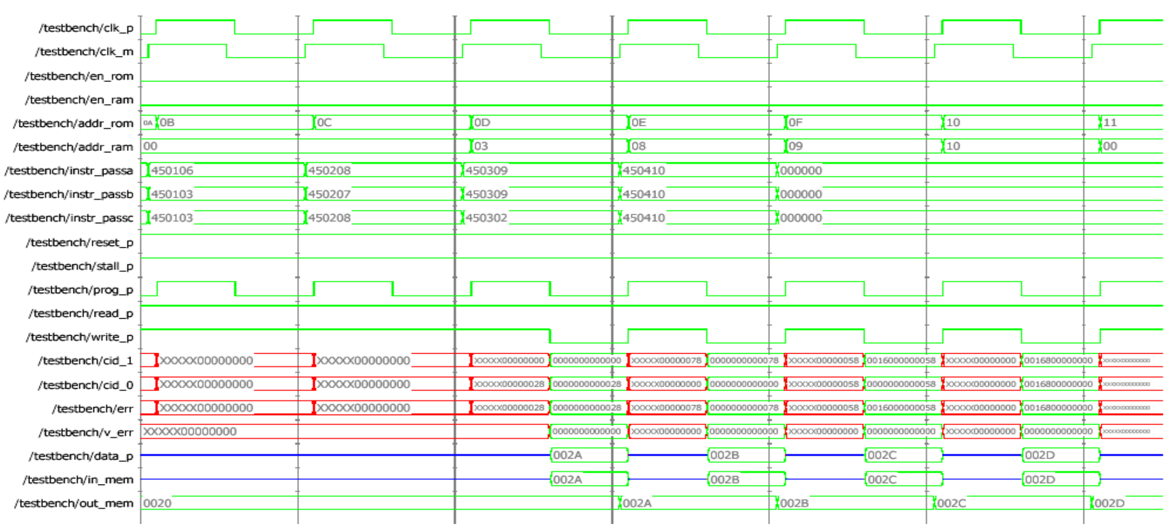
Only one instruction is different in each address of *ROM*s. This avoids multiple errors being sent to the voters at the same time. The *RAM* contains non-repeated data in each address. Details on how to read the error detection signal and analyze the error are discussed in Chapter V.

ROM A		ROM B		ROM C		RAM	
00	000000	00	000000	00	000000	00	20
01	000000	01	000000	01	000000	01	21
02	000000	02	000000	02	000000	02	22
03	44010A	03	44010A	03	44010A	03	23
04	440203	04	44020B	04	44020B	04	24
05	44030C	05	440A0C	05	44030C	05	25
06	44040D	06	44040D	06	350911	06	26
07	000000	07	000000	07	000000	07	27
08	000000	08	000000	08	000000	08	28
09	000000	09	000000	09	000000	09	29
0A	000000	0A	000000	0A	000000	0A	2A
0B	450106	0B	450103	0B	450103	0B	2B
0C	450208	0C	450207	0C	450208	0C	2C
0D	450309	0D	450309	0D	450302	0D	2D
0E	450410	0E	450410	0E	450410	0E	2E

4. Simulation Result

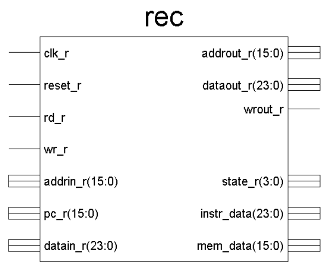






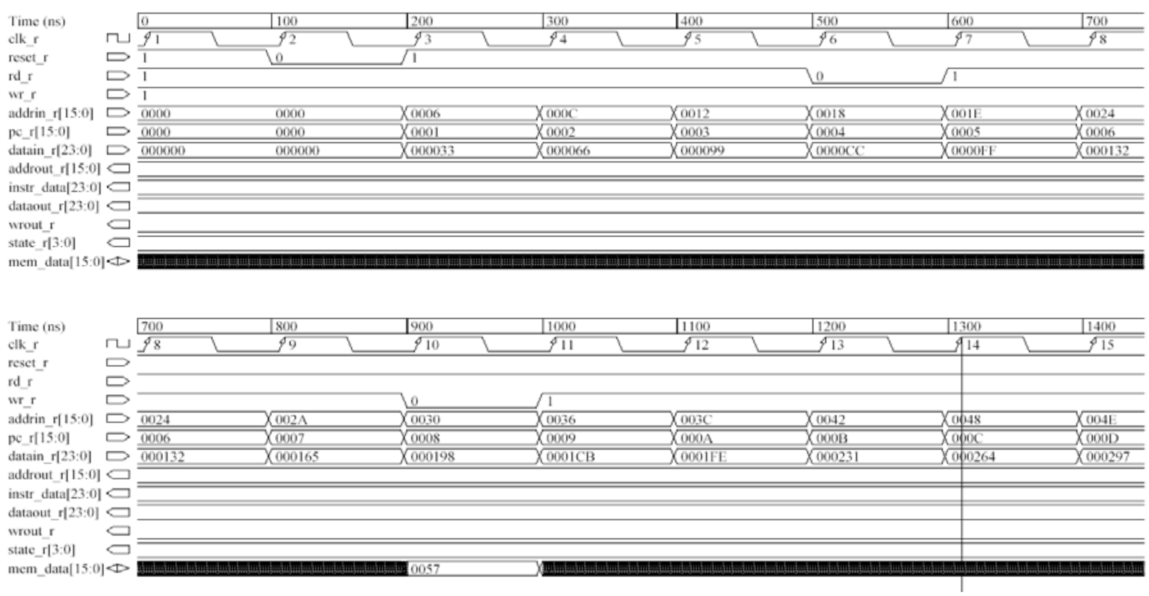
G. RECONCILER

1. Schematic

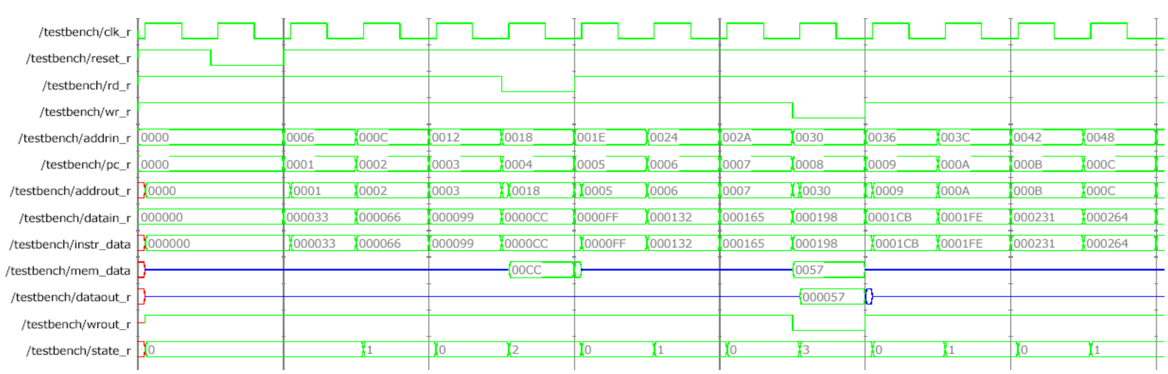


2. Test Bench

The clock high and low times are each 50 ns. The input setup time and output valid delay are each 10 ns. Manually set values in the data address, the program counter and the data were used to distinguish which one was fetched.

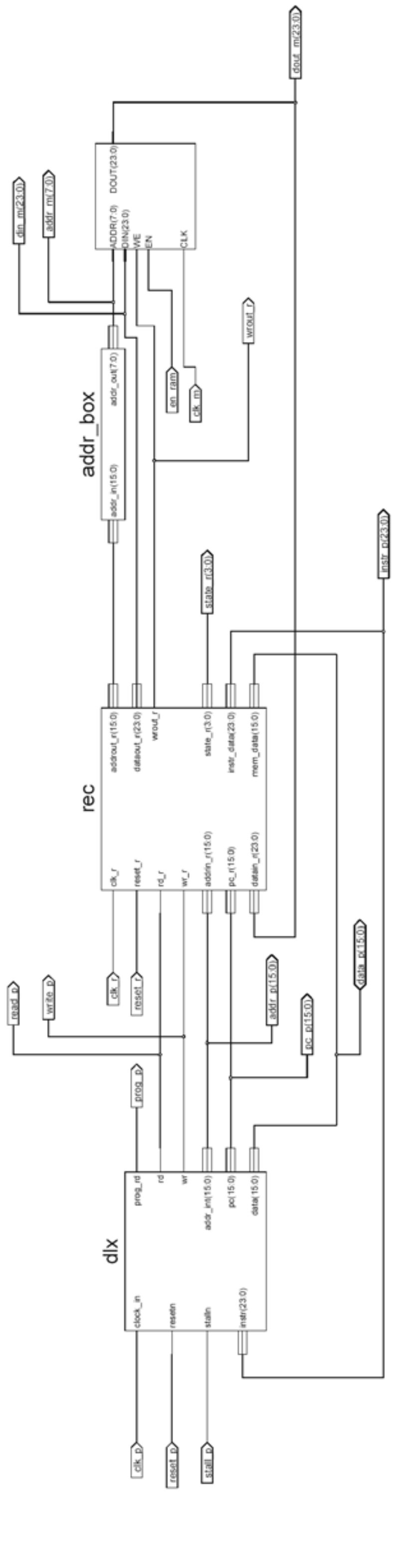


3. Simulation Result



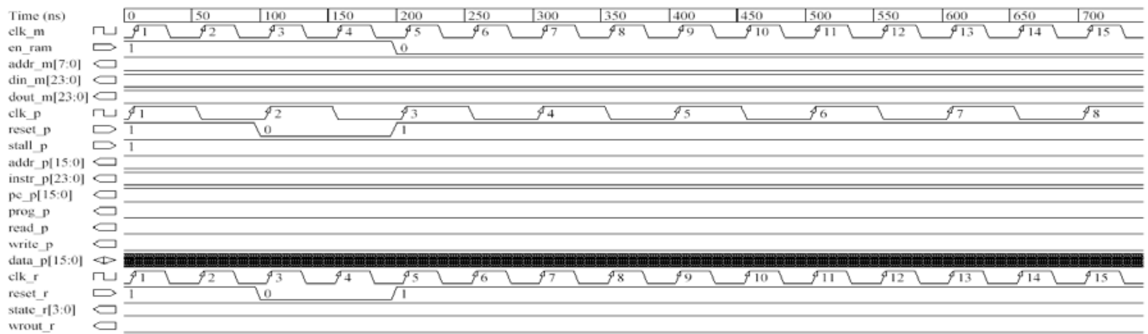
H. RECONCILER WITH KDLX AND MEMORY

1. Schematic

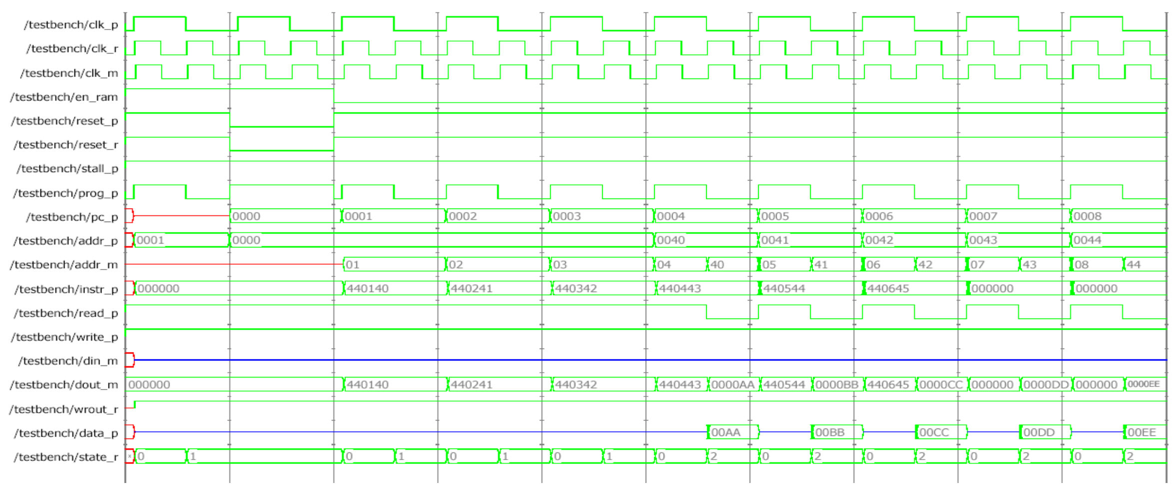


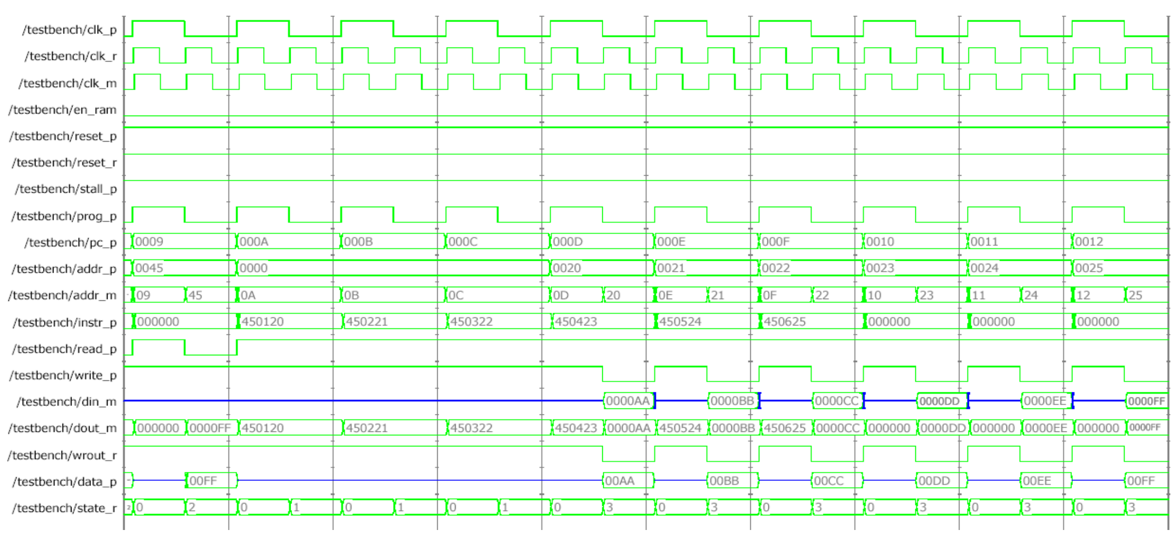
2. Test bench

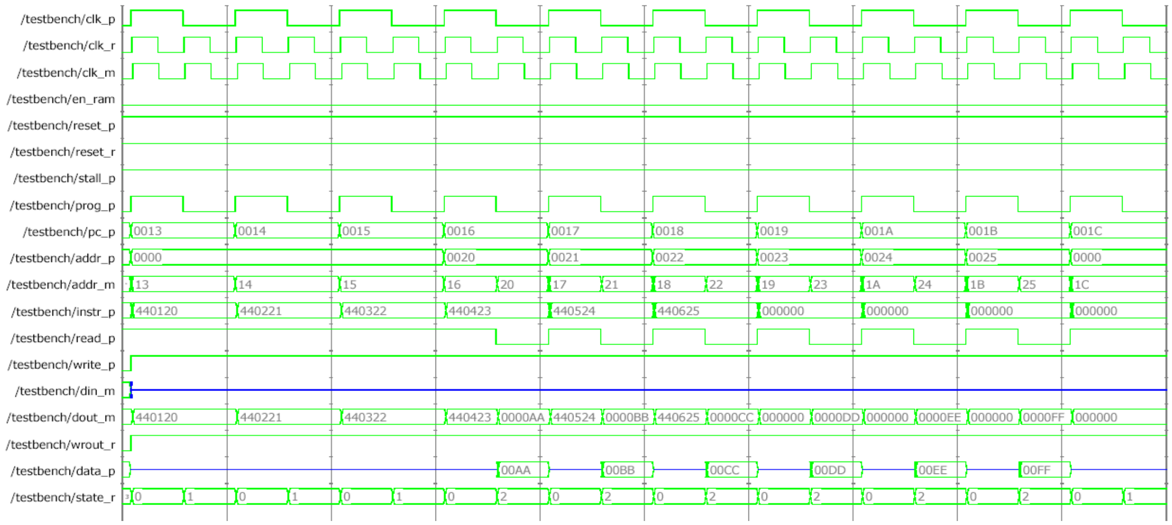
The clock high and low times for KDLX, *Reconciler*, and memory are 50 ns, 25 ns, and 25 ns, respectively. The input setup times and output valid delays for KDLX, *Reconciler*, and memory are 8 ns, 9 ns, and 10 ns, respectively.



3. Simulation Result



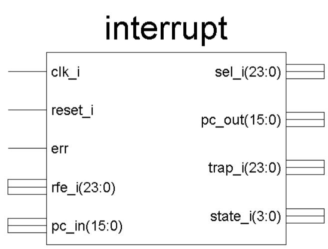




I. INTERRUPT

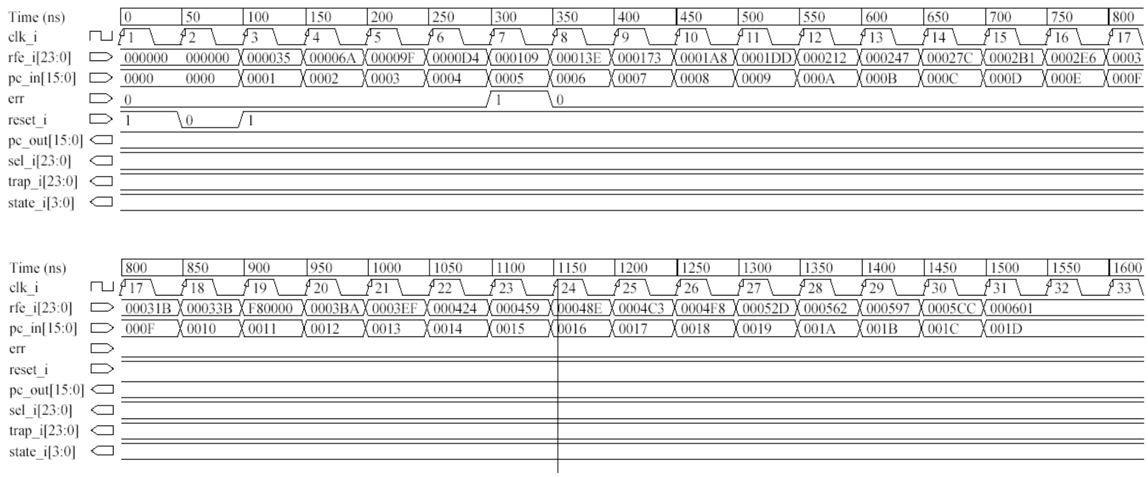
1. Schematic

The *rfe_i(23:0)* is used to monitor the RFE instruction. The *pc_in(15:0)* is connected to the program counter of KDLX. The signal *sel_i(23:0)* controls the muxes in order to insert the TRAP and Jump instruction sent out from *trap_i(23:0)*.

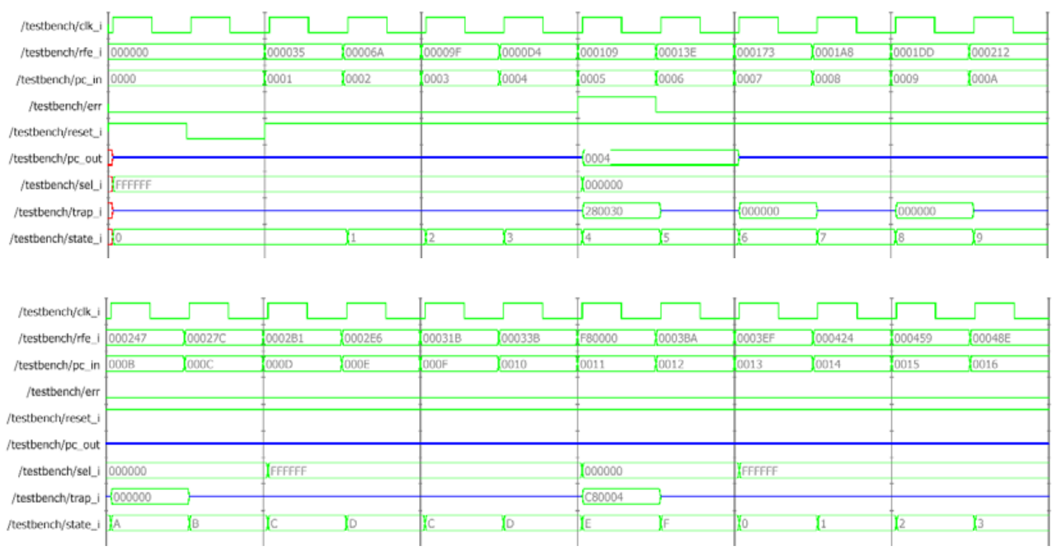


2. Test Bench

Random numbers are assigned to *rfe_i(23:0)* and *pc_in(15:0)*. An RFE instruction at time 900 ns emulates the end of the ISR.



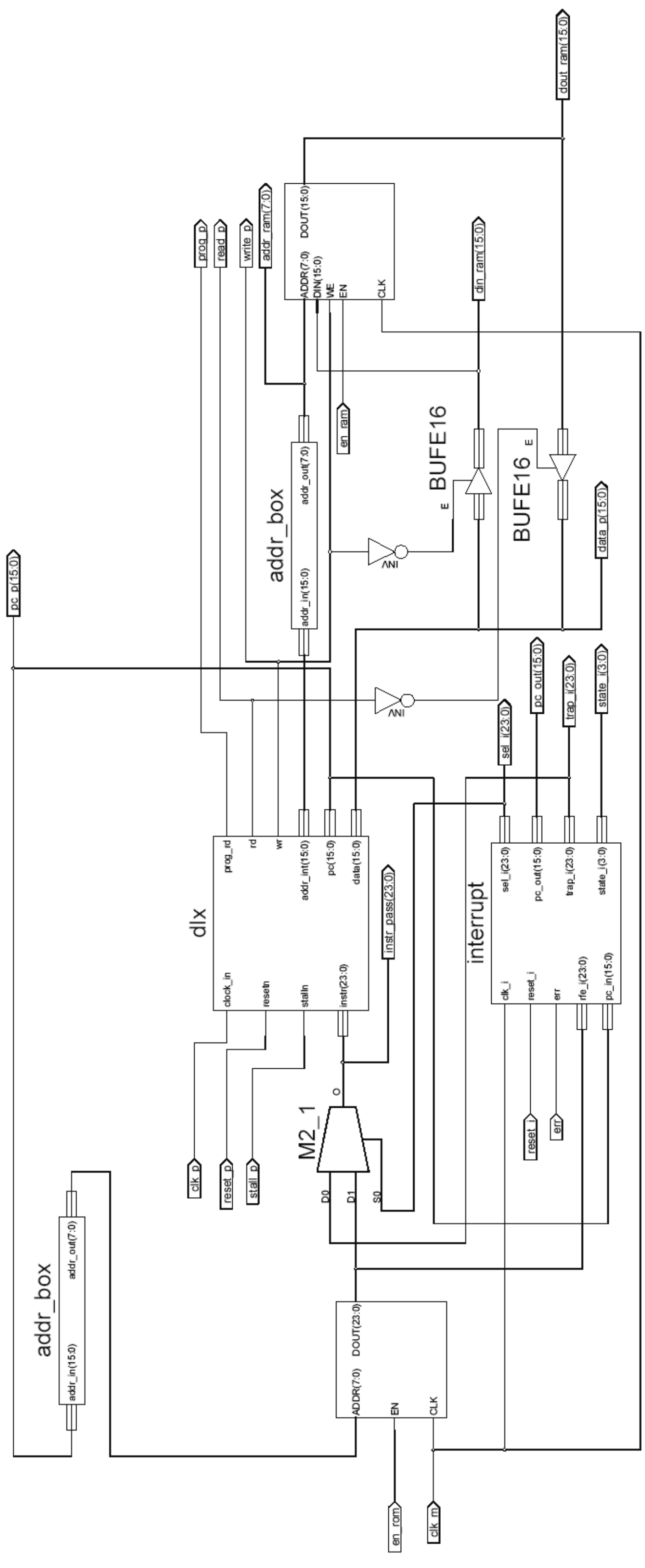
3. Simulation Result



J. INTERRUPT WITH KDLX AND MEMORY

1. Schematic

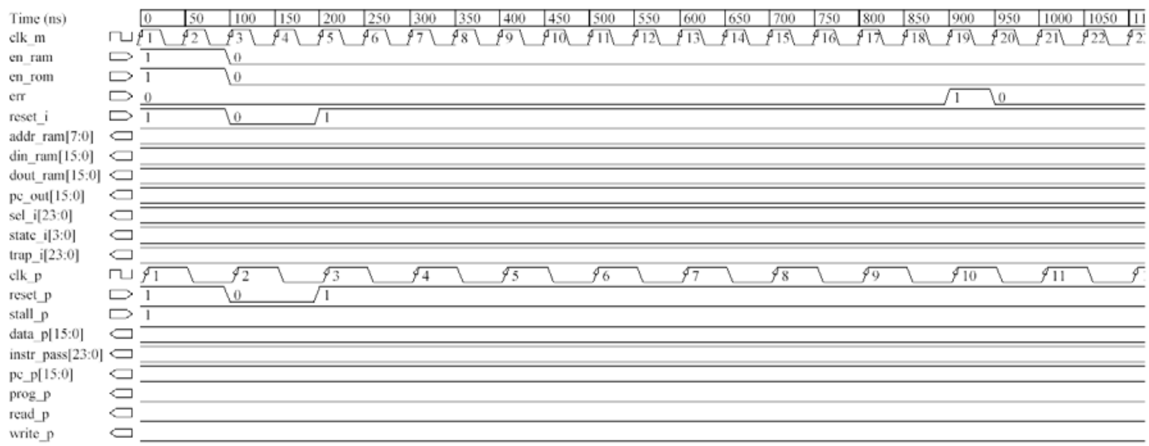
The *Reconciler* is not included in this schematic so two memories are used for a Harvard architecture. In this design, the *Interrupt* only needs to monitor the instructions from the ROM. The error signal is triggered manually in the test bench. Once the ISR starts, the instruction on the bus will be replaced with the TRAP instruction and lead the KDLX to implement the specific ISR. The last instruction in the ISR is the RFE instruction which activates the *Interrupt* to insert a new Jump instruction into KDLX. Then the circuit goes back to its normal operation.



dlx
interrupt
addr_box
addr_box
BUFE16
BUFE16
M2_1
INV
INV
prog_p
read_p
write_p
addr_ram(7:0)
din_ram(15:0)
dout_ram(15:0)
data_p(15:0)
pc_p(15:0)
en_ram
sel_i(23:0)
pc_out(15:0)
trap_i(23:0)
state_i(3:0)
instr_pass(23:0)
clk_p
reset_p
stall_p
reset_i
err
en_rom
clk_m
ADDR(7:0)
DIN(15:0)
WE
EN
CLK
DOUT(15:0)
DOUT(23:0)
addr_in(15:0)
addr_out(7:0)
prog_rd
rd
wr
addr_int(15:0)
pc(15:0)
data(15:0)
clock_in
resetn
stalln
instr(23:0)
clk_i
rfe_i(23:0)
pc_in(15:0)
D0
D1
S0
O
E

2. Test Bench

The KDLX clock high and low times are each 50 ns. The input setup time and output valid delay are each 10 ns. The *Interrupt*, ROM and RAM all run in double speed with a clock high and low time of 25 ns. The setup time and hold times are each 3 ns. Generate an error in the test bench at time 900 ns to check the function of the state machine. This test bench stops at time 4900 ns.



3. Memory Pre-configuration and Results

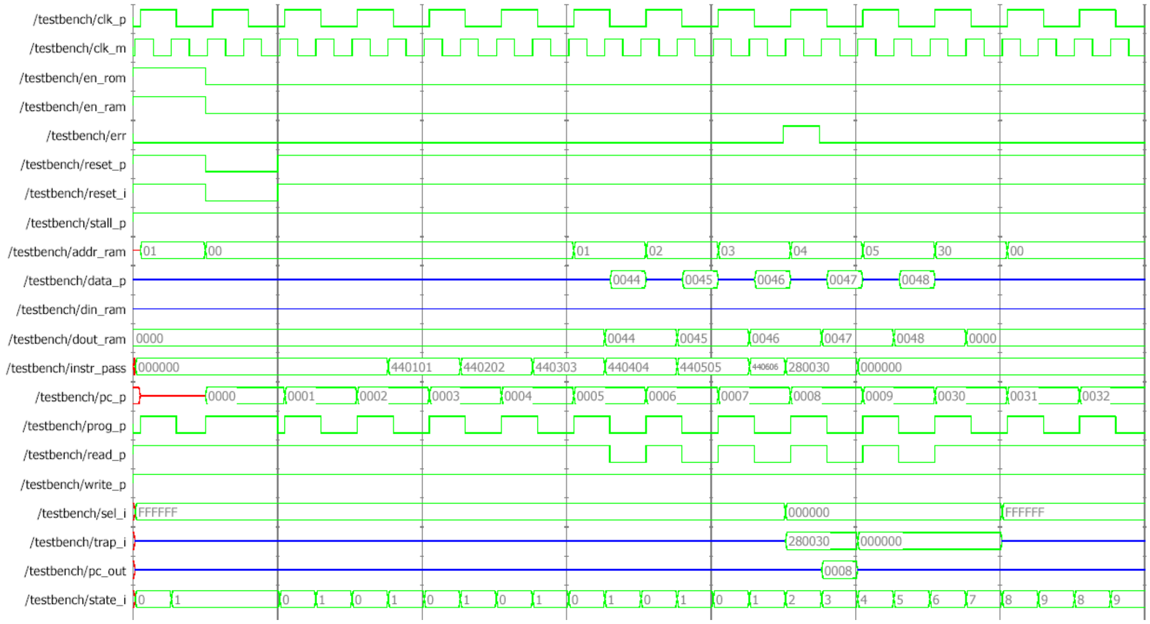
The highlighted Opcode is where an error occurs in the test bench. Contents in the *Instruction Mem* and the upper half data of the *Data Mem* are pre-configured. Registers and the lower half data of the *Data Mem* are the final values after the simulation is done.

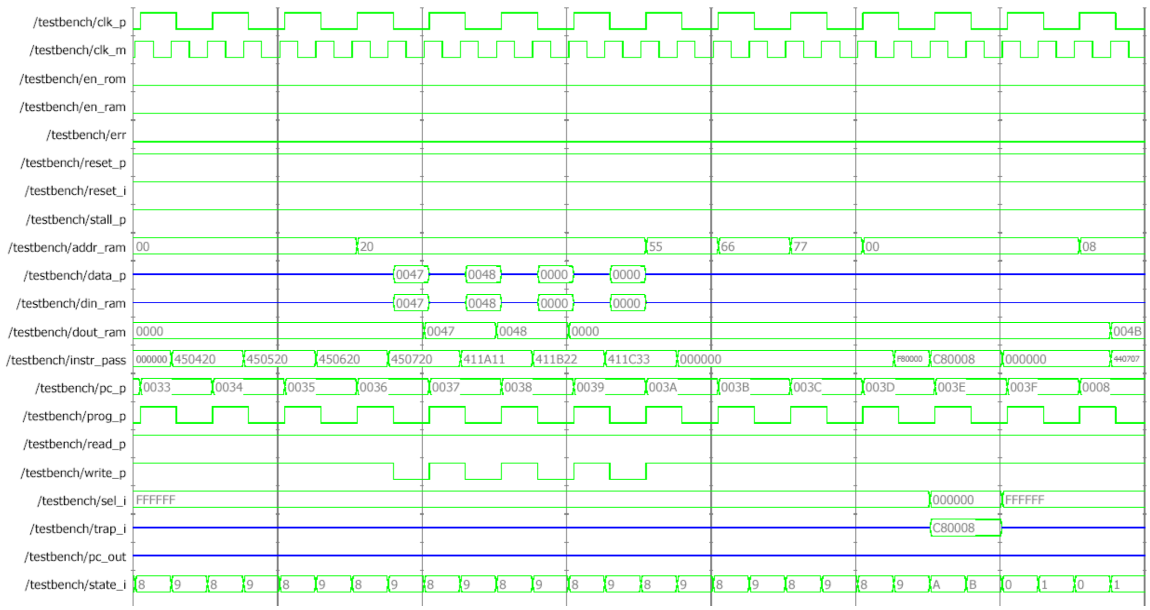
Instruction Mem			
00		2D	
01		2E	
02	440101	2F	
03	440202	30	000000
04	440303	31	000000
05	440404	32	000000
06	440505	33	450420
07	440606	34	450520
08	440707	35	450620
09	440808	36	450720
0A	440909	37	411A11
0B	450110	38	411B22
0C	450211	39	411C33
0D	450312	3A	000000
0E	450413	3B	000000
0F	450514	3C	000000
10	450615	3D	F80000
11	450716	3E	000000
12	450817	3F	000000
13	450918	40	000000
14	450A19	41	
15	450B1A	42	
16	450C1B	43	
⋮	⋮	44	
		45	
2C		46	

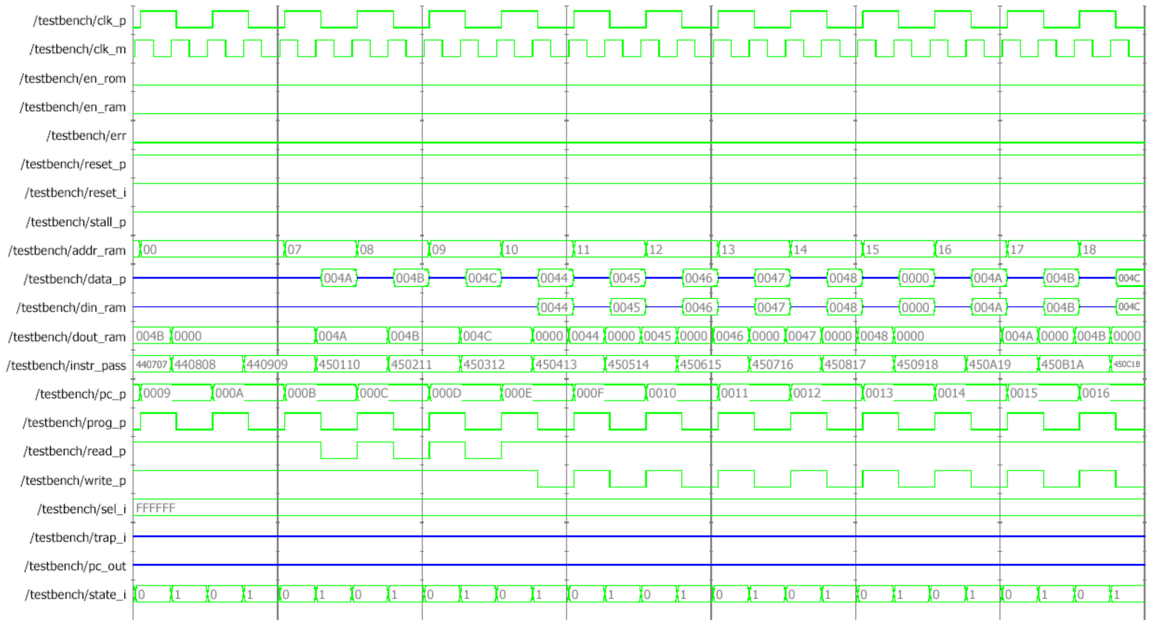
Register	
00	
01	0044
02	0045
03	0046
04	0047
05	0048
06	0049
07	004A
08	004B
09	004C
10	0055
11	0066
12	0077
13	
14	
15	

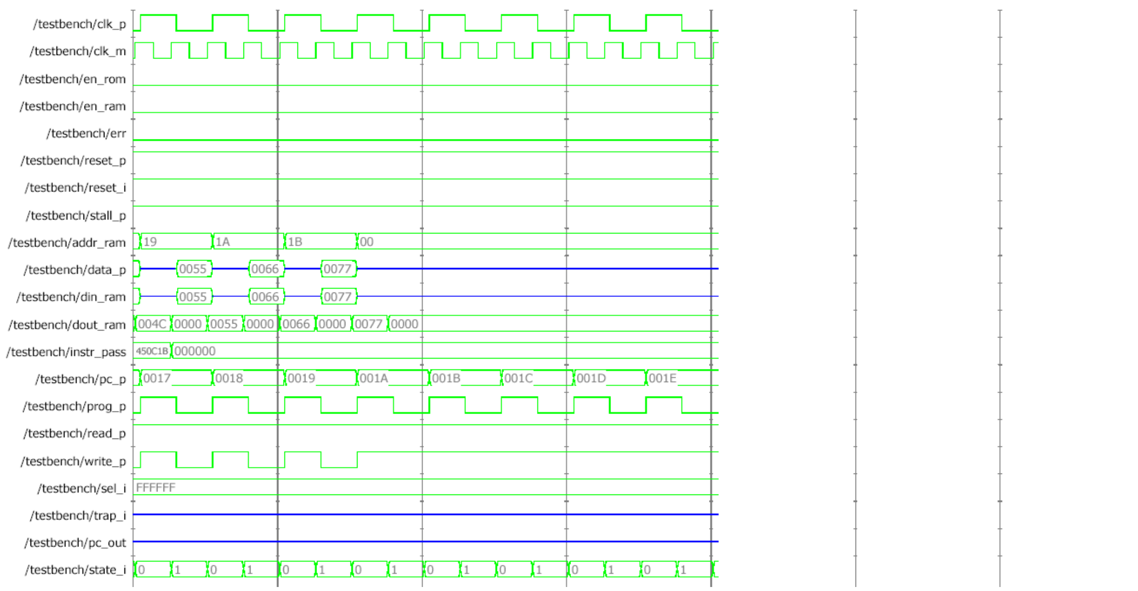
Data Mem	
00	
01	0044
02	0045
03	0046
04	0047
05	0048
06	0049
07	004A
08	004B
09	004C
0A	
0B	
0C	
0D	
0E	
0F	
10	0044
11	0045
12	0046
13	0047
14	0048
15	0049
16	004A
17	004B
18	004C
19	

4. Simulation Result





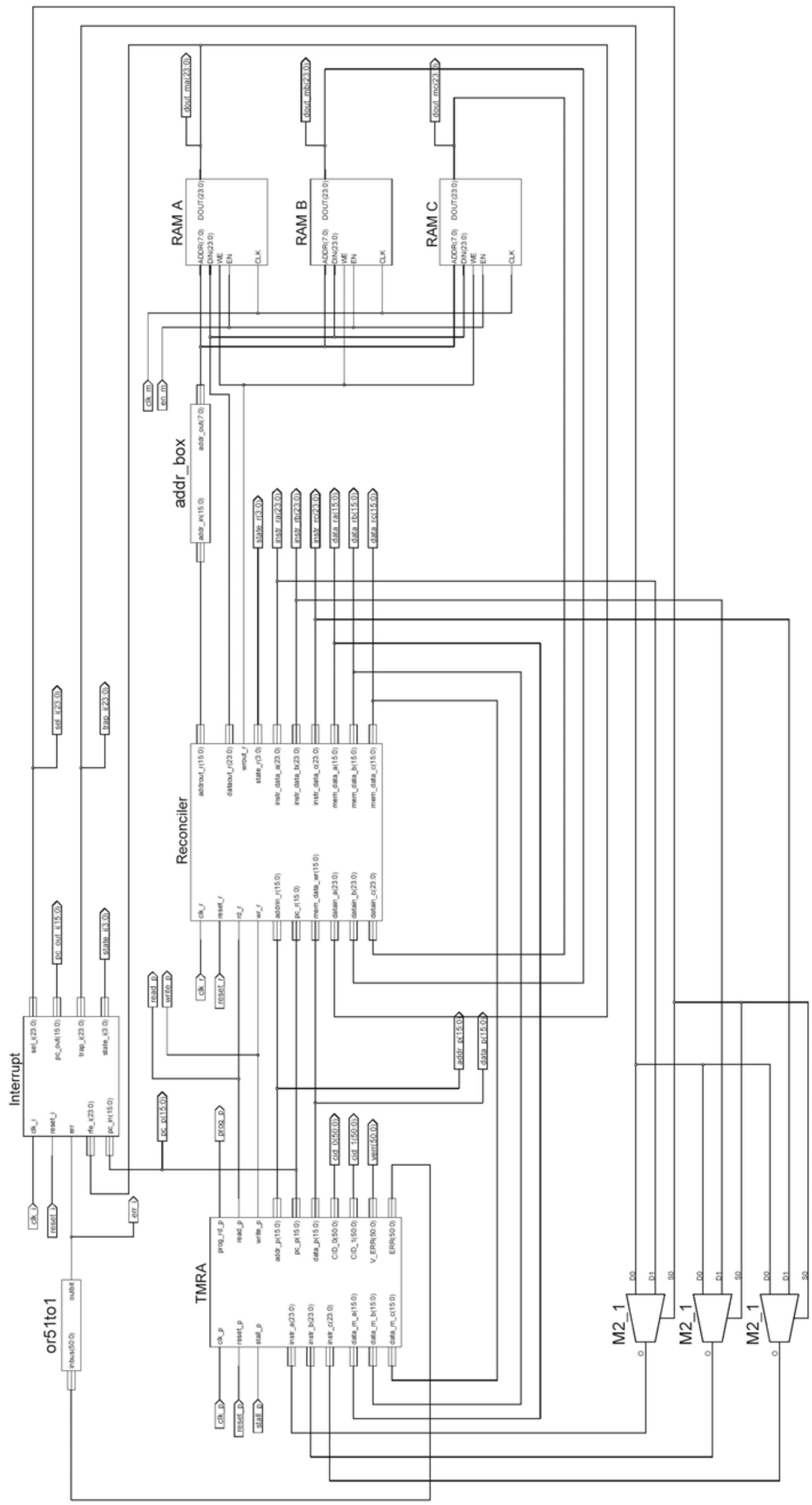




K. THE FULL DESIGN WITHOUT ESSD

1. Schematic

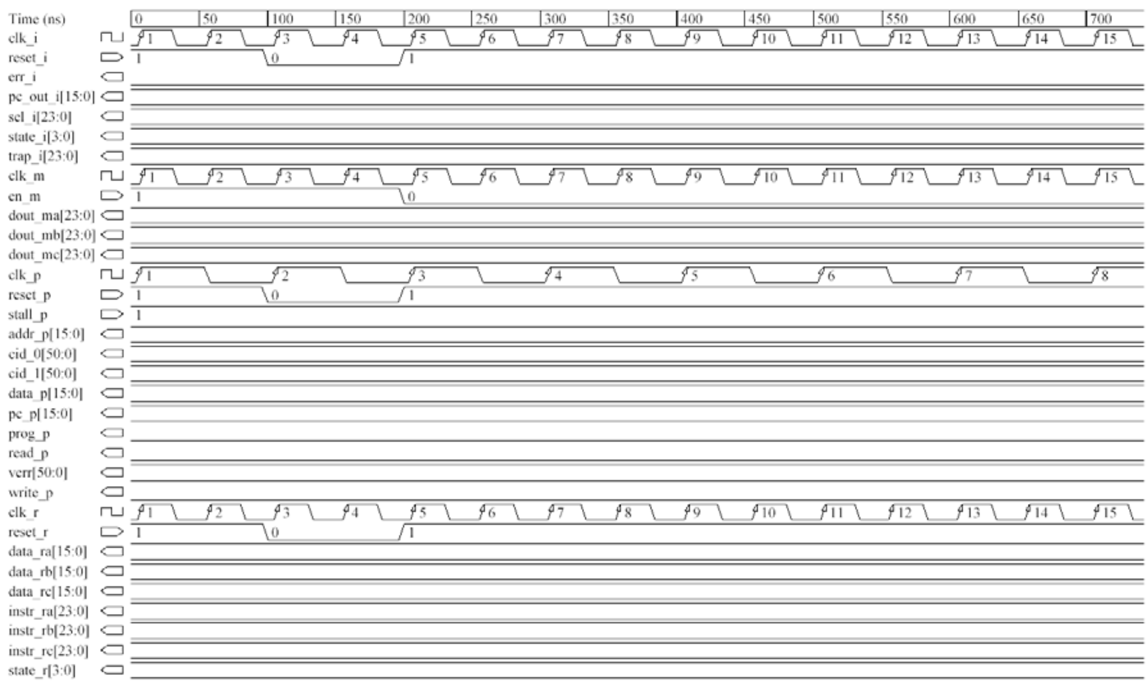
Three *RAM*s are used to provide inconsistent data to *TMRA*. This schematic is designed for simulating the circumstance at the occurrence of an error. The real design needs only one RAM and does not have to triplicate the instruction and data buses.


RAM A
RAM B
RAM C
addr_box
Reconciler
Interrupt
TMRA
or51to1
M2_1
M2_1
M2_1

2. Test Bench

The clock high and low times for KDLX, *Reconciler*, *Interrupt*, and memory are 50 ns, 25 ns, 25 ns, and 25 ns, respectively. The input setup times and output valid delays for KDLX, *Reconciler*, *Interrupt*, and memory are 8 ns, 9 ns, 9 ns, and 10 ns, respectively. The ending point of this test bench is at 4900 ns.

The signals between *clk_i* and *clk_m* are associated with the *Interrupt* clock cycle. The signals between *clk_m* and *clk_p* are associated with the memory clock cycle. Each signal in simulation has to be associated with one clock.



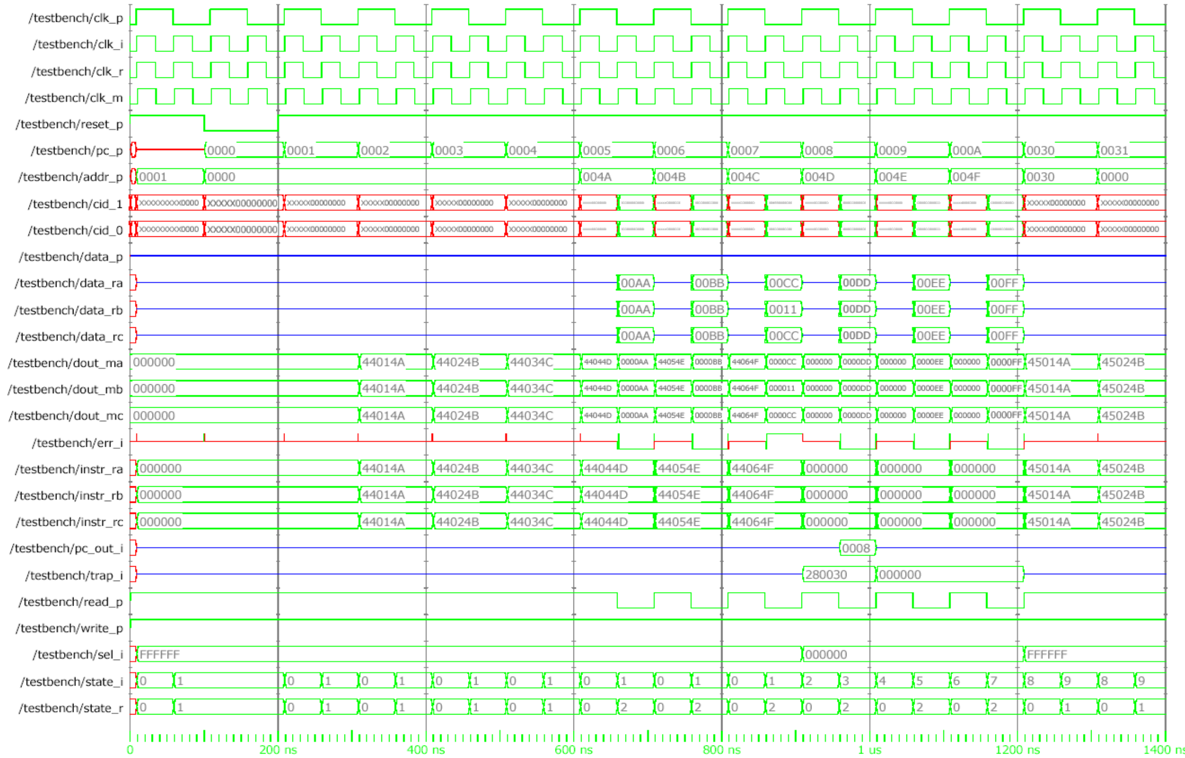
3. Memory Pre-configurations

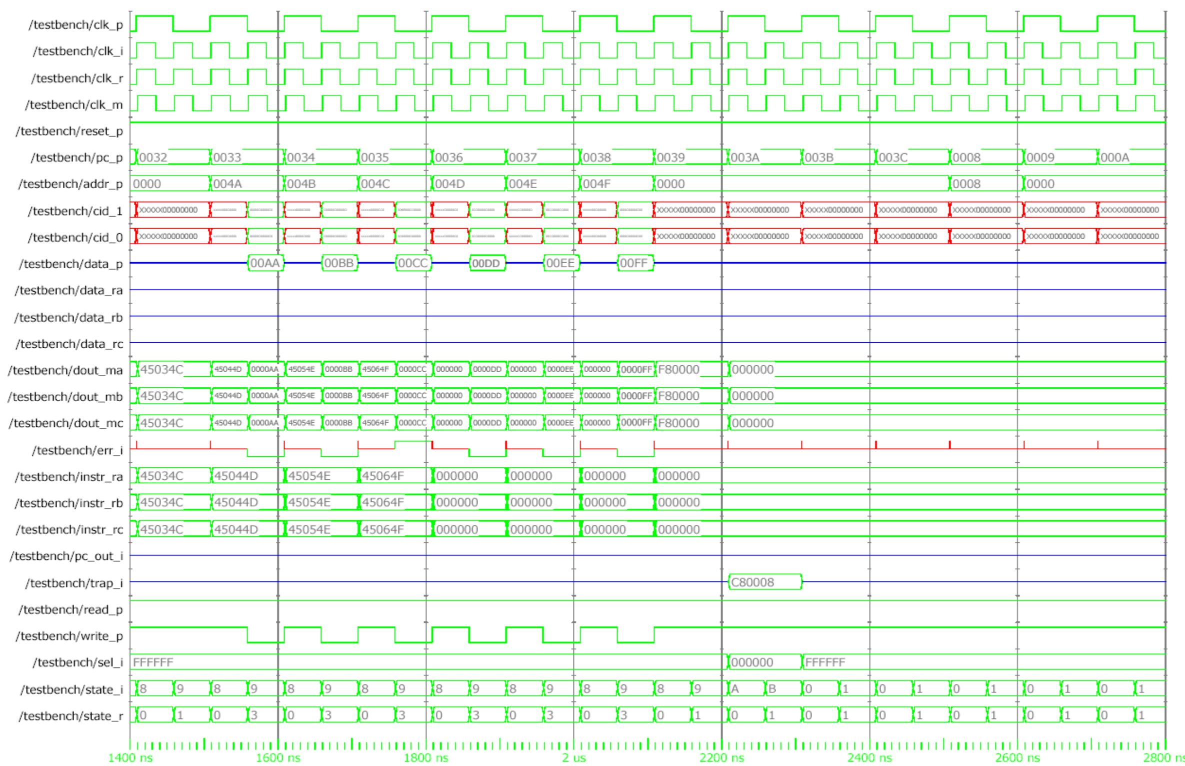
RAM A, B and C			
00	000000	2D	
01	000000	2E	
02	44014A	2F	
03	44024B	30	45014A
04	44034C	31	45024B
05	44044D	32	45034C
06	44054E	33	45044D
07	44064E	34	45054E
08	000000	35	45064F
09	000000	36	000000
0A	000000	37	000000
0B	44014A	38	000000
0C	44024B	39	F80000
0D	44034C	3A	000000
0E	44044D	3B	000000
0F	44054E	3C	000000
10	44064E	3D	
11	000000	3E	
12	000000	.	.
13	000000	.	.
14	000000	.	.
.	.	4A	0000AA
.	.	4B	0000BB
.	.	4C	0000CC
.	.	4D	0000DD
.	.	4E	0000EE
.	.	4F	0000FF
.			
2C		50	

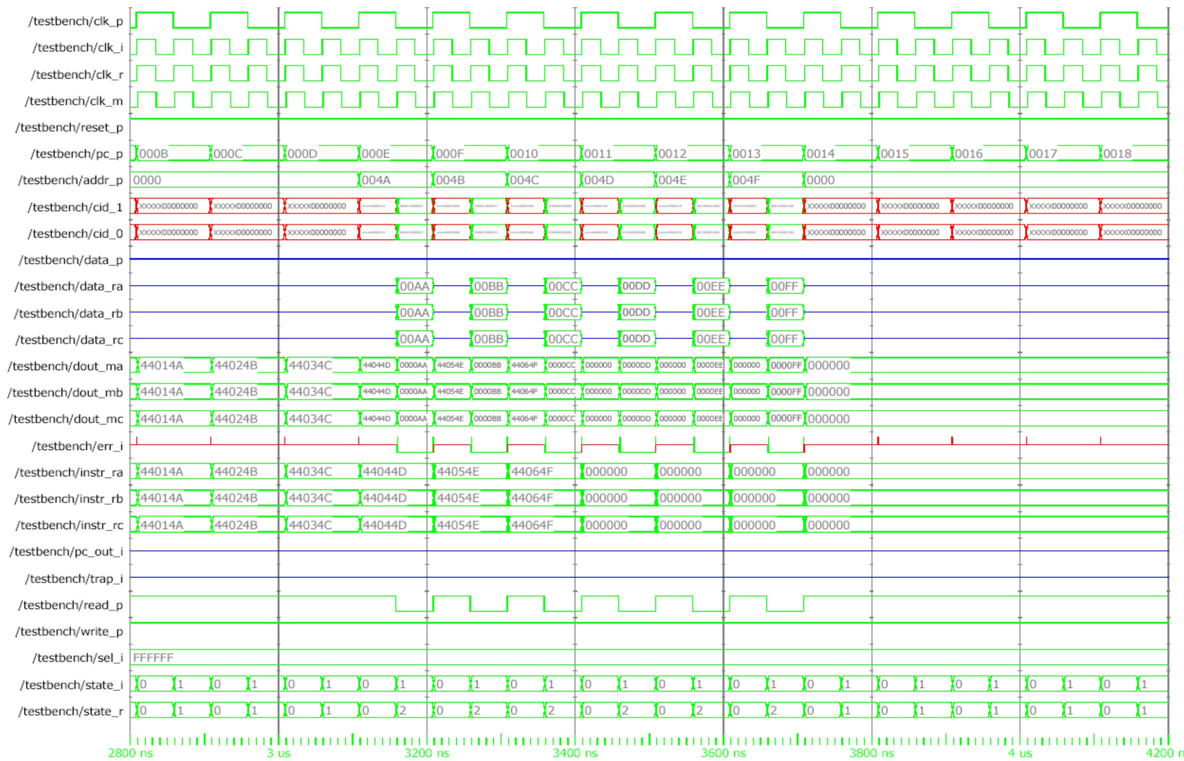
ISR

RAM B has 00011

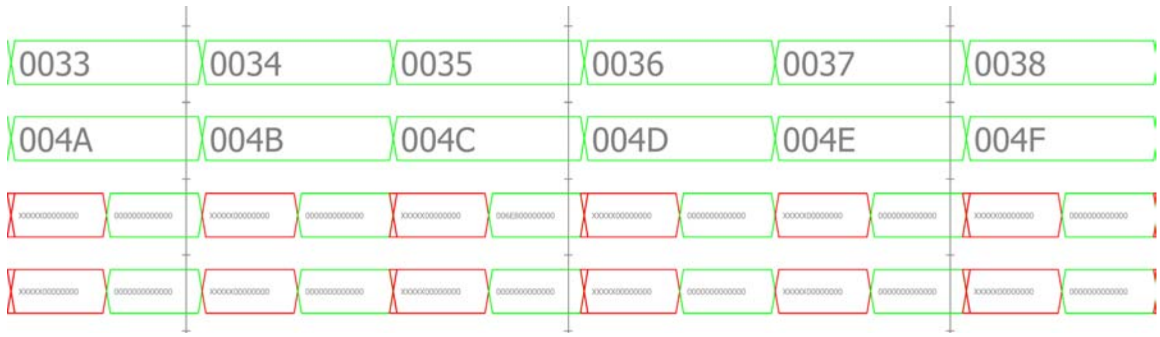
4. Simulation Result

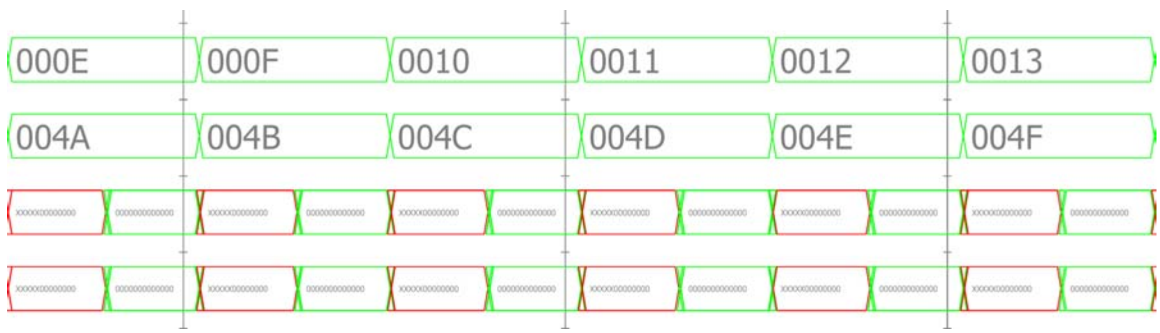






5. Zoom-in Figures of *cid_1* and *cid_0*

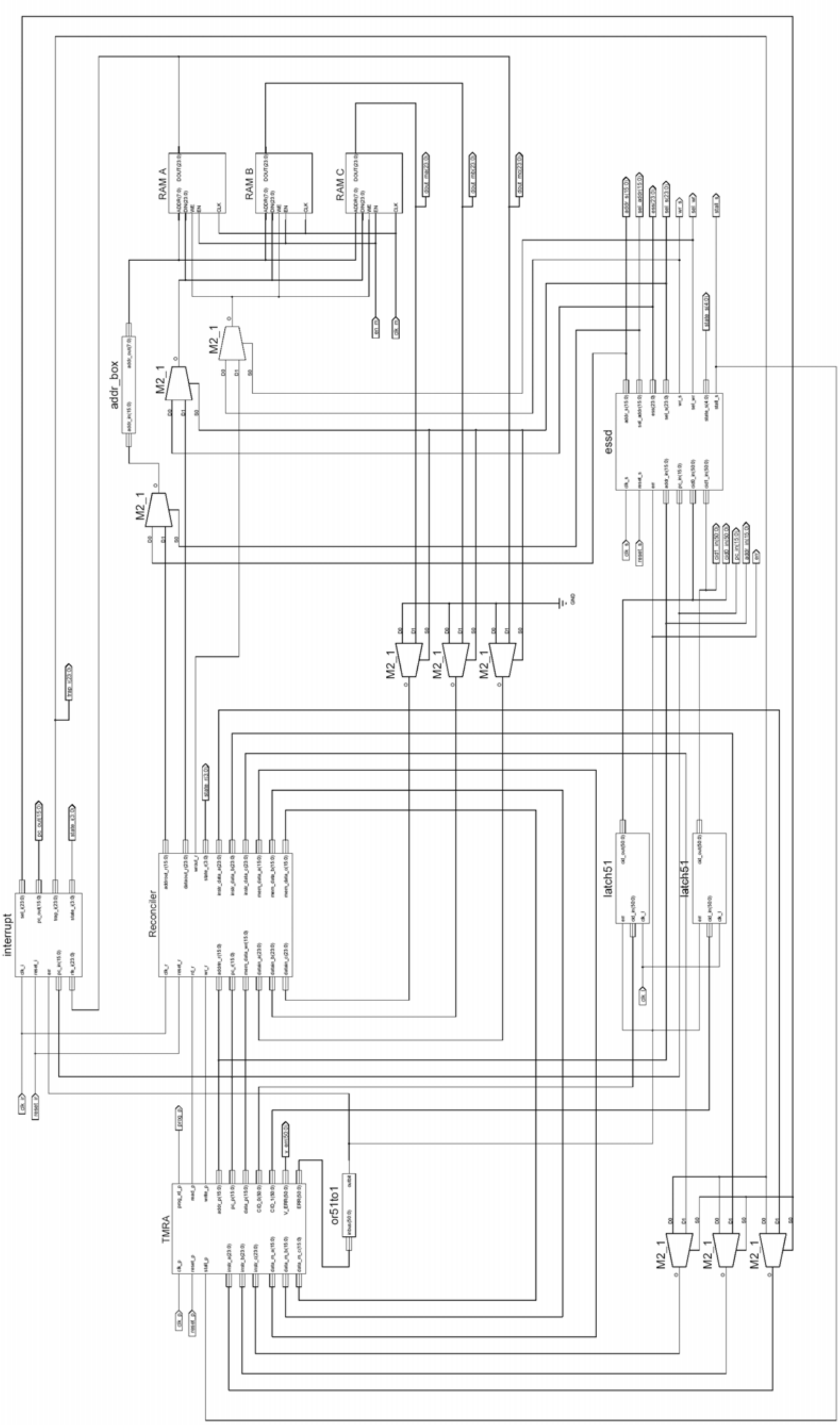




L. THE FULL DESIGN WITH ESSD

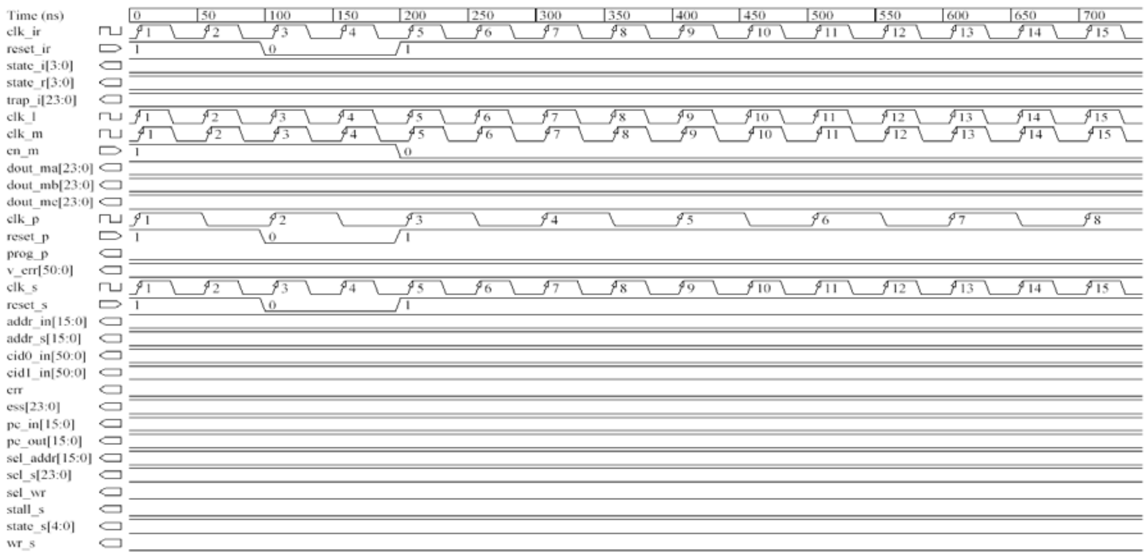
1. Schematic

The *ESSD* intercepts all connections on *RAM*s when the error syndromes are being stored. The clock for *Interrupt* and *Reconciler* are wired together since they work in parallel.

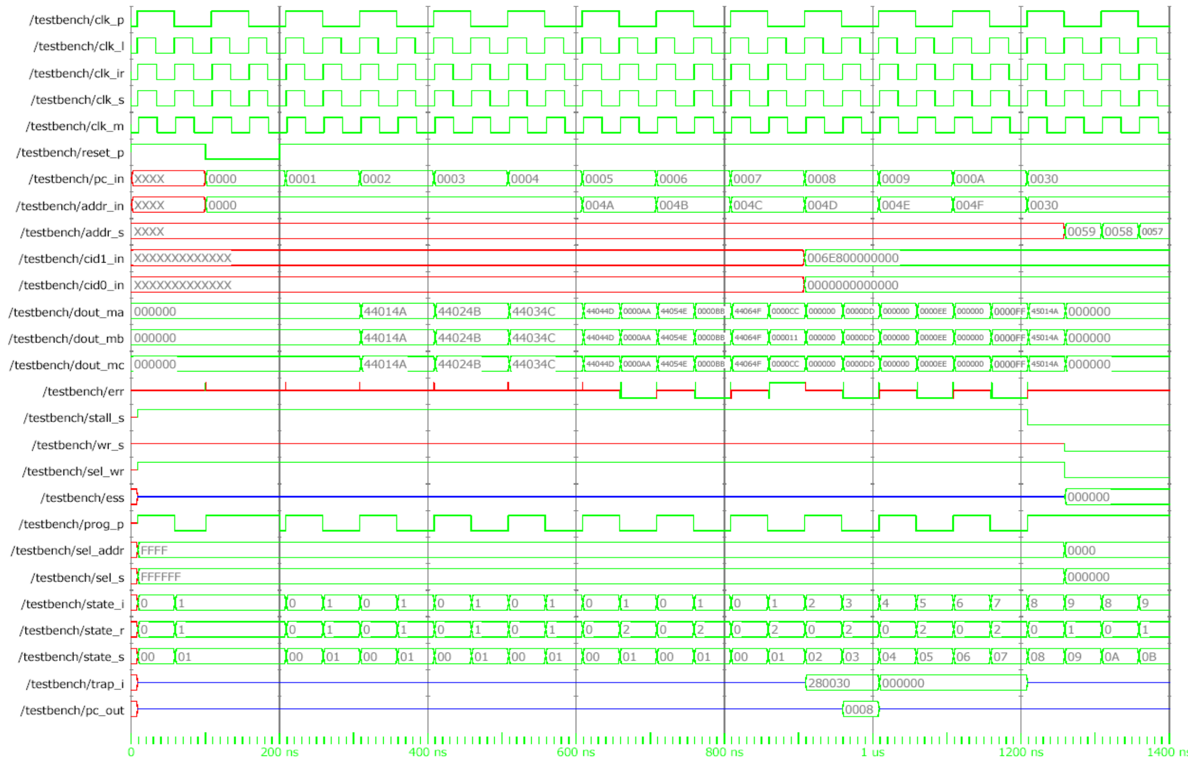

RAM A
RAM B
RAM C
addr_box
M2_1
essd
interrupt
Reconciler
latch51
TMRA
or51to1

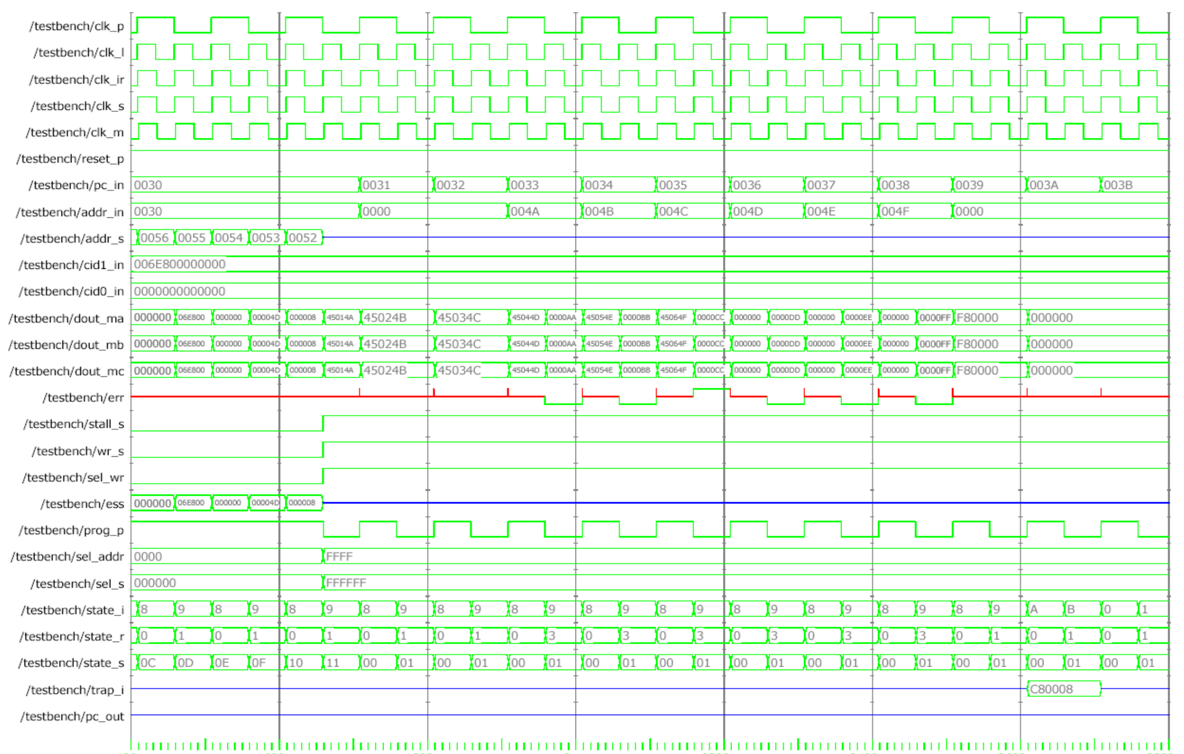
2. Test Bench

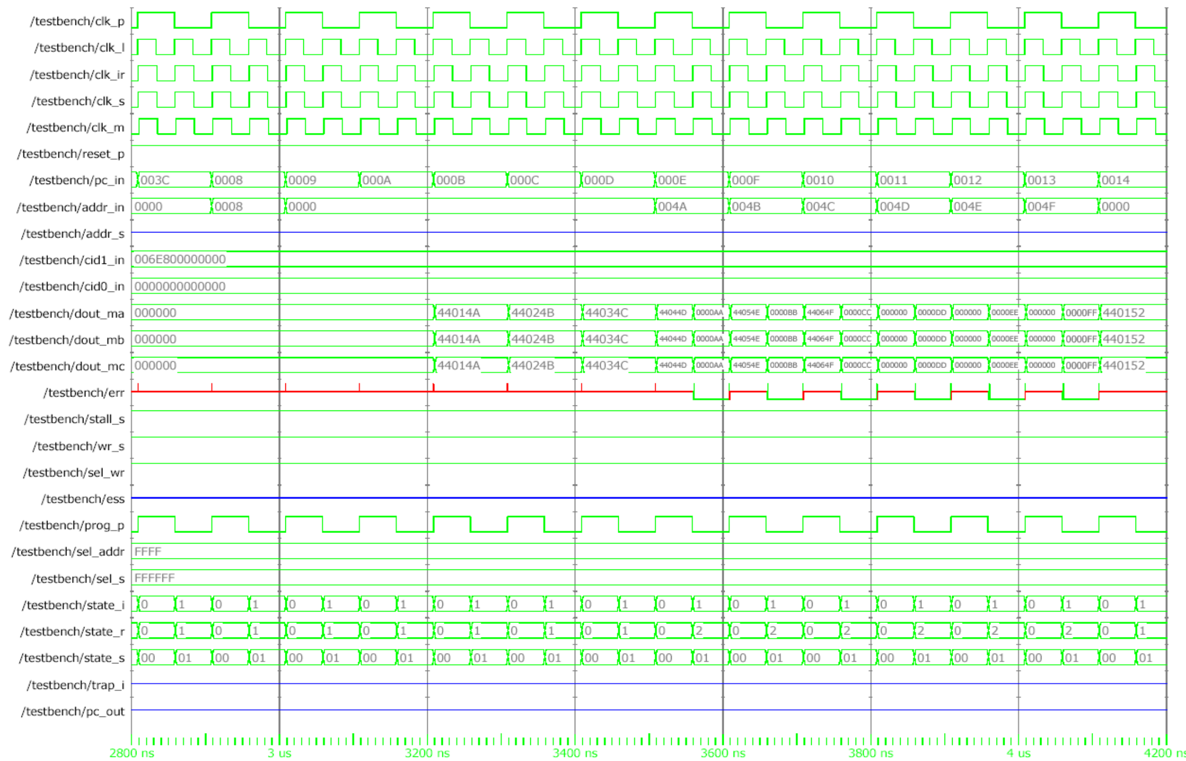
The clock high and low times for KDLX, *latch51*, *Reconciler* (or *Interrupt*), *ESSD*, and memory are 50 ns, 25 ns, 25 ns, 25 ns, and 25 ns, respectively. The input setup times and output valid delays for KDLX, *latch51*, *Reconciler* (or *Interrupt*), *ESSD*, and memory are 8 ns, 8 ns, 9 ns, 9 ns, and 10 ns, respectively. The test bench ends at time 4900 ns.

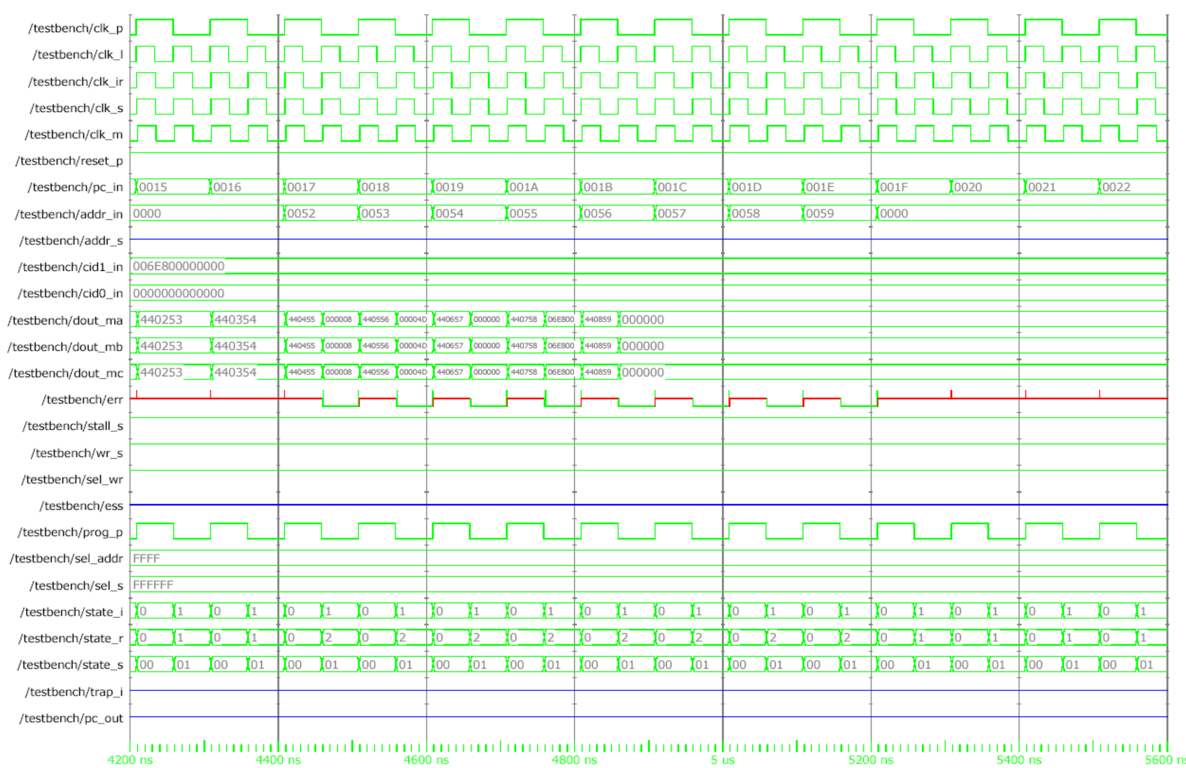


3. Simulation Result









APPENDIX B: KDLX INSTRUCTION SET DESCRIPTION

This appendix lists all of the operation codes and functions of the instructions used in the KDLX. This reference was originally contained in Dr. Kenneth Clark's dissertation [8]. Some errors were found and have been checked with the author. The function of the correct operation codes has been proved in the simulations of this thesis. The operation description is revised in order to give a clear discription of how data transfers.

Some symbols used in this appendix need to be introduced first. Rs1 represents one of the 15 registers in KDLX. Rs2 represents one of the 15 registers in KDLX as well. Rs1 and Rs2 could be the same register. Rd represents one of the 15 registers in KDLX used as a destination register. Immed_7 represents the most significant bit of a 7-bit immediate value. $[(\text{Immed}_7)^8 \,||\, \text{Immed}]$ represents an 7-bit immediate value being sign extended to 16-bit long.

Instruction: ADD (Register Add)

23 20	19 16	15 12	11 8	7 4	3 0
Opcode: 0x01		Rs1	Rd	Rs2	Unused

Usage: ADD Rd, Rs1, Rs2

Operation: $\text{Rd} \leftarrow (\text{Rs1}+\text{Rs2})$

Instruction: ADDI (Add Immediate)

23 20	19 16	15 12	11 8	7 4	3 0
Opcode: 0x01		Rs1	Rd	Rs2	Unused

Usage: ADDI Rd, Rs1, Immed

Operation: $\text{Rd} \leftarrow (\text{Rs1}+[(\text{Immed}_7)^8 \,||\, \text{Immed}])$

Instruction: ADDUI (Add Unsigned Immediate)

23 20	19 16	15 12	11 8	7 4	3 0
Opcode: 0x21		Rs1	Rd	Immed	

Usage: ADDUI Rd, Rs1, Immed

Operation: $\text{Rd} \leftarrow (\text{Rs1}+[(0)^8 \,||\, \text{Immed}])$

Instruction: AND (Register AND)

23 20	19 16	15 12	11 8	7 4	3 0
Opcode: 0x09		Rs1	Rd	Rs2	Unused

Usage: AND Rd, Rs1, Rs2

Operation: Rd $\leftarrow$ (Rs1 (logical-and) Rs2)

Instruction: ANDI (AND Immediate)

23 20	19 16	15 12	11 8	7 4	3 0
Opcode: 0x29		Rs1	Rd	Immed	

Usage: AND Rd, Rs1, Immed

Operation: Rd $\leftarrow$ (Rs1 (logical-and) $[(\text{Immed}_7)^8 \| \text{Immed}]$)

Instruction: BEQZ (Branch if Equal to Zero)

23 20	19 16	15 12	11 8	7 4	3 0
Opcode: 0xC1		Rs1	Unused	Immed	

Usage: BEQZ Rs1, Immed

Operation: If Rs1=0, then Program_Address $\leftarrow$ (PC+1+$[(\text{Immed}_7)^8 \| \text{Immed}]$)

Instruction: BNEZ (Branch if Not Equal to Zero)

23 20	19 16	15 12	11 8	7 4	3 0
Opcode: 0xC0		Rs1	Unused	Immed	

Usage: BNEZ Rs1, Immed

Operation: If Rs1$\neq$0, then Program_Address $\leftarrow$ (PC+1+$[(\text{Immed}_7)^8 \| \text{Immed}]$)

Instruction: J (Jump)

23 20	19 16	15 12	11 8	7 4	3 0
Opcode: 0xC8		Immed			

Usage: J Immed

Operation: Program_Address $\leftarrow$ Immed

Instruction: JAL (Jump and Link)

23 20 19 16	15 12 11 8 7 4 3 0
Opcode: 0xE8	Immed

Usage: JAL Immed

Operation: Program_Addr $\leftarrow$ Immed;

R15 $\leftarrow$ Link_Program_Address

Instruction: JALR (Jump Register and Link)

23 20 19 16	15 12	11 8 7 4 3 0
Opcode: 0x68	Rs1	Unused

Usage: JALR Rs1

Operation: Program_Addr $\leftarrow$ (Rs1);

R15 $\leftarrow$ Link_Program_Address

Instruction: JR (Jump Register)

23 20 19 16	15 12	11 8 7 4 3 0
Opcode: 0x48	Rs1	Unused

Usage: JALR Rs1

Operation: Program_Address $\leftarrow$ (Rs1)

Instruction: LHI (Load High Immediate)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x08	Unused	Rd	Immed

Usage: LHI Rd, Immed

Operation: Rd $\leftarrow$ Immed $\|$ $(0)^8$

Instruction: LW (Load Word)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x44	Rs1	Rd	Immed

Usage: LW Rd, Rs1(Immed)

Operation: Rd $\leftarrow$ Mem{Rs1+[$(\text{Immed}_7)^8$ $\|$ Immed]}

Instruction: NOP (No Operation)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x00	Unused			

Usage: NOP

Operation: None

Instruction: OR (Register OR)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x2A	Rs1	Rd	Rs2	Unused

Usage: OR Rd, Rs1, Rs2

Operation: Rd ← (Rs1 (logical-or) Rs2)

Instruction: ORI (OR Immediate)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x2A	Rs1	Rd	Immed	

Usage: ORI Rd, Rs1, Immed

Operation: Rd ← (Rs1 (logical-or) Immed)

Instruction: RFE (Return from Exception)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0xF8	Unused			

Usage: RFE

Operation: Program_Address ← Interrupt_Address_Register

Instruction: SEQ (Set if Equal)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x18	Rs1	Rd	Rs2	Unused

Usage: SEQ Rd, Rs1, Rs2

Operation: If Rs1=Rs2, then Rd=0x0001 else Rd=0x0000

Instruction: SEQI (Set Equal Immediate)

23	20 19	16	15	12	11	8	7	4 3	0
Opcode: 0x58			Rs1		Rd		Immed		

Usage: SEQI Rd, Rs1, Immed

Operation: If Rs1=$[(Immed_7)^8 \| Immed]$, then Rd=0x0001 else Rd=0x0000

Instruction: SGE (Set if Greater Than or Equal)

23	20 19	16	15	12	11	8	7	4	3	0
Opcode: 0x19			Rs1		Rd		Rs2		Unused	

Usage: SGE Rd, Rs1, Rs2

Operation: If Rs1 $\geq$ Rs2, then Rd=0x0001 else Rd=0x0000

Instruction: SGEI (Set if Greater Than or Equal Immediate)

23	20 19	16	15	12	11	8	7	4 3	0
Opcode: 0x59			Rs1		Rd		Immed		

Usage: SGEI Rd, Rs1, Immed

Operation: If Rs1 $\geq [(Immed_7)^8 \| Immed]$, then Rd=0x0001 else Rd=0x0000

Instruction: SGT (Set if Greater Than)

23	20 19	16	15	12	11	8	7	4	3	0
Opcode: 0x1A			Rs1		Rd		Rs2		Unused	

Usage: SGT Rd, Rs1, Rs2

Operation: If Rs1>Rs2, then Rd=0x0001 else Rd=0x0000

Instruction: SGTI (Set if Greater Than Immediate)

23	20 19	16	15	12	11	8	7	4 3	0
Opcode: 0x5A			Rs1		Rd		Immed		

Usage: SGTI Rd, Rs1, Immed

Operation: If Rs1>$[(Immed_7)^8 \| Immed]$, then Rd=0x0001 else Rd=0x0000

Instruction: SLE (Set if Less Than or Equal)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x1B	Rs1	Rd	Rs2	Unused

Usage: SLE Rd, Rs1, Rs2

Operation: If Rs1 $\leq$ Rs2, then Rd=0x0001 else Rd=0x0000

Instruction: SLEI (Set if Less Than or Equal Immediate)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x5B	Rs1	Rd	Immed

Usage: SLEI Rd, Rs1, Immed

Operation: If Rs1 $\leq [(\text{Immed}_7)^8 \parallel \text{Immed}]$, then Rd=0x0001 else Rd=0x0000

Instruction: SLL (Shift Logic Left)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x11	Rs1	Rd	Rs2	Unused

Usage: SLL Rd, Rs1, Rs2

Operation: Rd $\leftarrow$ (Rs1) shifted left by Rs2(3:0) bits

Instruction: SLLI (Shift Logic Left Immediate)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x51	Rs1	Rd	Immed

Usage: SLLI Rd, Rs1, Immed

Operation: Rd $\leftarrow$ (Rs1) shifted left by Immed(3:0) bits

Instruction: SLT (Set if Less Than)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x1C	Rs1	Rd	Rs2	Unused

Usage: SLT Rd, Rs1, Rs2

Operation: If Rs1<Rs2, then Rd=0x0001 else Rd=0x0000

Instruction: SLTI (Set if Less Than Immediate)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x5C	Rs1	Rd	Immed

Usage: SLTI Rd, Rs1, Immed

Operation: If Rs1<$[(\text{Immed}_7)^8 \parallel \text{Immed}]$, then Rd=0x0001 else Rd=0x0000

Instruction: SNE (Set if Not Equal)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x1D	Rs1	Rd	Rs2	Unused

Usage: SNE Rd, Rs1, Rs2

Operation: If Rs1≠Rs2, then Rd=0x0001 else Rd=0x0000

Instruction: SNEI (Set if Not Equal Immediate)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x58	Rs1	Rd	Immed

Usage: SNEI Rd, Rs1, Immed

Operation: If Rs1≠$[(\text{Immed}_7)^8 \parallel \text{Immed}]$, then Rd=0x0001 else Rd=0x0000

Instruction: SRA (Shift Right Arithmetic)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x13	Rs1	Rd	Rs2	Unused

Usage: SRA Rd, Rs1, Rs2

Operation: Rd ← (Rs1) shifted by Rs2(3:0) bits, with Rs1(15) shifted in from right (for sign extension)

Instruction: SRAI (Shift Right Arithmetic Immediate)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x53	Rs1	Rd	Immed

Usage: SRAI Rd, Rs1, Immed

Operation: Rd ← (Rs1) shifted by Immed(3:0) bits, with Rs1(15) shifted in from right (for sign extension)

Instruction: SRL (Shift Right Logical)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x12	Rs1	Rd	Rs2	Unused

Usage: SRL Rd, Rs1, Rs2

Operation: Rd $\leftarrow$ (Rs1) shifted by Rs2(3:0) bits, with 0’s shifted in from right

Instruction: SRLI (Shift Right Logical Immediate)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x52	Rs1	Rd	Immed

Usage: SRLI Rd, Rs1, Immed

Operation: Rd $\leftarrow$ (Rs1) shifted by Immed(3:0) bits, with 0’s shifted in from right

Instruction: SUB (Register Subtract)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x03	Rs1	Rd	Rs2	Unused

Usage: SUB Rd, Rs1, Rs2

Operation: Rd $\leftarrow$ (Rs1−Rs2)

Instruction: SUBI (Subtract Immediate)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x43	Rs1	Rd	Immed

Usage: SUB Rd, Rs1, Immed

Operation: Rd $\leftarrow$ (Rs1$-$[(Immed$_7$)^8 || Immed])

Instruction: SUBUI (Subtract Unsigned Immediate)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x23	Rs1	Rd	Immed

Usage: SUBUI Rd, Rs1, Immed

Operation: Rd $\leftarrow$ (Rs1$-$[(0)^8 || Immed])

Instruction: SW (Store Word)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x45	Rs1	Rd	Immed

Usage: SW Rs2, Rs1(Immed)

Operation: $Mem\{Rs1+[(Immed_7)^8 \mid\mid Immed]\} \leftarrow Rs2$

Instruction: TRAP (Software Trap)

23 20 19 16	15 12 11 8 7 4 3 0
Opcode: 0x28	Unused

Usage: Trap Immed

Operation: Program_Address ← Immed;

Interrupt_Address_Register ← Link_Program_Address

Instruction: XOR (Register Exclusive-OR)

23 20 19 16	15 12	11 8	7 4	3 0
Opcode: 0x0B	Rs1	Rd	Rs2	Unused

Usage: XOR Rd, Rs1, Rs2

Operation: Rd ← (Rs1 (exclusive-or) Rs2)

Instruction: XORI (Exclusive-OR Immediate)

23 20 19 16	15 12	11 8	7 4 3 0
Opcode: 0x2B	Rs1	Rd	Immed

Usage: XORI Rd, Rs1, Immed

Operation: Rd ← (Rs1 (exclusive-or) Immed)

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: VHDL CODE

A.   RECONCILER

```
--*****************************************************************
-- Module: Reconciler
--
-- Function: The Reconciler is used as an interface between the KDLX
-- and memory.  It runs two times faster than the KDLX.
--
-- Author: Rong Yuan, TWAF
--
-- Date: Nov 14, 2003
--*****************************************************************


      library IEEE;
      use IEEE.STD_LOGIC_1164.ALL;
      use IEEE.STD_LOGIC_ARITH.ALL;
      use IEEE.STD_LOGIC_UNSIGNED.ALL;


      entity rec is Port (
           clk_r: in std_logic;
           reset_r: in std_logic;
           rd_r: in std_logic;
           wr_r: in std_logic;
           addrin_r: in std_logic_vector(15 downto 0);
           pc_r: in std_logic_vector(15 downto 0);
           datain_r: in std_logic_vector(23 downto 0);
           addrout_r: out std_logic_vector(15 downto 0);
           instr_data: out std_logic_vector(23 downto 0);
           dataout_r: out std_logic_vector(23 downto 0);
           mem_data: inout std_logic_vector(15 downto 0);
           wrout_r: out std_logic;
           state_r: out std_logic_vector(3 downto 0)
           );
      end rec;

      architecture fsm of rec is     -- fsm is Finite State Machine

      type targetFSM is (State, State0, State1, ReadState, WriteState);

      signal currState, nextState: targetFSM;


      begin

      nxtStProc: process ( currState, rd_r, wr_r)

      begin
```

```
      case currState is
           when State =>
                nextState <= State0;
           when State0 =>
                if (rd_r='0' and wr_r='1') then    -- read from memory
                    nextState <= ReadState;
                elsif (rd_r='1' and wr_r='0') then -- write to memory
                    nextState <= WriteState;
                else
                    nextState <= State1;
                end if;

           when State1 =>
                nextState <= State0;
           when ReadState =>
                nextState <= State0;
           when WriteState =>
                nextState <= State0;
       end case;

      end process nxtStProc;


      -- Process to register the current state

      curStProc: process (clk_r, reset_r)

      begin
           if (reset_r ='0') then
               currState <= State;
           elsif (clk_r'event and clk_r='1') then
               currState <= nextState;
           end if;
      end process curStProc;


      -- Process to generate outputs

      outConProc: process (currState, wr_r, pc_r, datain_r, addrin_r,
mem_data)

      begin

      case currState is
         when State =>         -- generated for reset only
              null;            -- without this state, state machine
starts at State1 after reset

         when State0 =>     -- doing instruction fetch
              state_r <= "0000";
              wrout_r <= wr_r;
              addrout_r <= pc_r;            -- sending pc to memory
              instr_data <= datain_r;       -- memory sends instruction
to KDLX
              dataout_r <= (others => 'Z');
              mem_data <= (others => 'Z');
```

```
   when State1 =>                  -- exactly the same as State0
                                   -- for keeping current state
        state_r <= "0001";
        wrout_r <= wr_r;
        addrout_r <= pc_r;
        instr_data <= datain_r;
        dataout_r <= (others => 'Z');
        mem_data <= (others => 'Z');
   when ReadState =>         -- When KDLX reads data from memory
        state_r <= "0010";
        wrout_r <= wr_r;            -- write signal is one
        addrout_r <= addrin_r;      -- sending address to memory
        mem_data <= datain_r(15 downto 0);
                                    -- memory sends data to KDLX
        dataout_r <= (others => 'Z');   -- block input to memory

   when WriteState =>         -- When KDLX writes data to memory
        state_r <= "0011";
        wrout_r <= wr_r;            -- write signal is zero
        addrout_r <= addrin_r;      -- sending address to memory
        dataout_r(15 downto 0) <= mem_data;
                                    -- KDLX sends data to memory
        dataout_r(23 downto 16) <= "00000000";
                                          -- sign extension data

end case;

end process outConProc;

end fsm;
```

B. INTERRUPT

```
--*********************************************************************
-- Module: Interrupt
--
-- Function: The Interrupt is used to switch to ISR when err occurs.
-- It runs in double speed and has the same time constraints with
-- Reconciler.  TRAP to other instruction set and jump back when done.
--
-- Notation: This Interrupt is revised to work with TMRA in this design
-- only.  This is the final version before ESSD is generated.  Only two
-- NOPs after TRAP.
--
-- Author: Rong Yuan, TWAF
--
-- Date: Nov 17, 2003
--*********************************************************************



library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity Interrupt is Port (
       rfe_i: in std_logic_vector(23 downto 0);
       pc_in: in std_logic_vector(15 downto 0);
       err: in std_logic;
       reset_i: in std_logic;
       clk_i: in std_logic;

       pc_out: out std_logic_vector(15 downto 0);
       sel_i: out std_logic_vector(23 downto 0);
       trap_i: out std_logic_vector(23 downto 0);
       state_i: out std_logic_vector(3 downto 0)
       );
end Interrupt;

architecture fsm of Interrupt is

type targetFSM is (State, State0_A, State0_B, TrapState_A, TrapState_B,
                   NopState0_A, NopState0_B, NopState1_A, NopState1_B,
                   WaitState_A, WaitState_B, BackState_A, BackState_B);

signal pc_latch: std_logic_vector(15 downto 0);
signal new_instr: std_logic_vector(23 downto 0);
signal currState, nextState: targetFSM;

begin

nxtStProc: process ( currState, err, rfe_i)

begin
```

```
case currState is

   when State =>
        nextState <= State0_A;

   when State0_A =>
        nextState <= State0_B;

   when State0_B =>
     if (err='1') then
         nextState <= TrapState_A;
     else
         nextState <= State0_A;
     end if;

   when TrapState_A =>
        nextState <= TrapState_B;

   when TrapState_B =>
        nextState <= NopState0_A;

   when NopState0_A =>
        nextState <= NopState0_B;

   when NopState0_B =>
        nextState <= NopState1_A;

   when NopState1_A =>
        nextState <= NopState1_B;

   when NopState1_B =>
        nextState <= WaitState_A;

   when WaitState_A =>
        nextState <= WaitState_B;

   when WaitState_B =>
     if (rfe_i(23 downto 16)="11111000") then         -- check F80000
         nextState <= BackState_A;
     else
         nextState <= WaitState_A;         -- stay if not seeing F80000
     end if;

   when BackState_A =>
        nextState <= BackState_B;

   when BackState_B =>
        nextState <= State0_A;

end case;

end process nxtStProc;

-- Process to register the current state

curStProc: process (clk_i, reset_i)
```

```
begin

     if (reset_i ='0') then
         currState <= State;
     elsif (clk_i'event and clk_i='1') then
         currState <= nextState;
     end if;

end process curStProc;

-- Process to generate outputs

outConProc: process (currState, pc_in)

begin

case currState is
   when State =>
        null;

   when State0_A =>
        state_i <= "0000";
        trap_i <= (others =>'Z');
        sel_i <= "111111111111111111111111";
        pc_out <= (others => 'Z');

   when State0_B =>
        state_i <= "0001";
        trap_i <= (others =>'Z');
        sel_i <= "111111111111111111111111";
        pc_out <= (others => 'Z');

   when TrapState_A =>
        state_i <= "0010";
        sel_i <= "000000000000000000000000"; --allow TRAP pass to KDLX
        trap_i <= "001010000000000000110000";    --TRAP instr 2800030
        pc_latch <= pc_in;               --latch pc for new instruction

   when TrapState_B =>
        state_i <= "0011";
        sel_i <= "000000000000000000000000";
        pc_out <= pc_latch;                    --show latched pc on bus

   when NopState0_A =>
        state_i <= "0100";
        trap_i <= "000000000000000000000000";       --allow NOP to KDLX
        sel_i <= "000000000000000000000000";
        pc_out <= (others => 'Z');

   when NopState0_B =>
        state_i <= "0101";
        sel_i <= "000000000000000000000000";        --allow NOP to KDLX
        pc_out <= (others => 'Z');

   when NopState1_A =>
        state_i <= "0110";
```

```
        trap_i <= "000000000000000000000000";
        sel_i <= "000000000000000000000000";
        pc_out <= (others => 'Z');

        --construct new JUMP instr
        new_instr(23 downto 16) <= "11001000";
        new_instr(15 downto 0) <= pc_latch;            --JUMP is C8+pc

   when NopState1_B =>
        state_i <= "0111";
        sel_i <= "000000000000000000000000";
        pc_out <= (others => 'Z');

   when WaitState_A =>
        state_i <= "1000";
        trap_i <= (others => 'Z');
        sel_i <= "111111111111111111111111";
        pc_out <= (others => 'Z');

   when WaitState_B =>
        state_i <= "1001";
        trap_i <= (others => 'Z');
        sel_i <= "111111111111111111111111";
        pc_out <= (others => 'Z');

   when BackState_A =>
        state_i <= "1010";
        trap_i <= new_instr;                   --allow new JUMP to KDLX
        sel_i <= "000000000000000000000000";
        pc_out <= (others => 'Z');

   when BackState_B =>
        state_i <= "1011";
        sel_i <= "000000000000000000000000";
        pc_out <= (others => 'Z');

end case;
end process outConProc;
end fsm;
```

C. RECONCILER FOR THE FULL DESIGN

```
--*******************************************************************
-- Module: Reconciler
--
-- Function: The Reconciler is used as an interface between TMRA and
-- memory.  It runs in double speed.  Act as instruction memory in the
-- first half KDLX clock and as data memory in the second half KDLX
-- clock.
--
-- Notation: This Reconciler is revised to work with the TMRA in this
-- design only.  Data buses are triplicated.
--
-- Author: Rong Yuan, TWAF
--
-- Date: Nov 14, 2003
--*******************************************************************


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity rec2 is Port (
       clk_r: in std_logic;
       reset_r: in std_logic;
       rd_r: in std_logic;
       wr_r: in std_logic;
       addrin_r: in std_logic_vector(15 downto 0);
       pc_r: in std_logic_vector(15 downto 0);
       datain_a: in std_logic_vector(23 downto 0);
       datain_b: in std_logic_vector(23 downto 0);
       datain_c: in std_logic_vector(23 downto 0);

       addrout_r: out std_logic_vector(15 downto 0);
       instr_data_a: out std_logic_vector(23 downto 0);
       instr_data_b: out std_logic_vector(23 downto 0);
       instr_data_c: out std_logic_vector(23 downto 0);
       dataout_r: out std_logic_vector(23 downto 0);
       mem_data_a: out std_logic_vector(15 downto 0);
                                             -- data from mem to KDLX
       mem_data_b: out std_logic_vector(15 downto 0);
       mem_data_c: out std_logic_vector(15 downto 0);
       mem_data_wr: in std_logic_vector(15 downto 0);
                                             -- data from KDLX to mem
       wrout_r: out std_logic;
       state_r: out std_logic_vector(3 downto 0)
       );
end rec2;

architecture fsm of rec2 is    -- fsm is Finite State Machine

type targetFSM is (State, State0, State1, ReadState, WriteState);
```

```
signal currState, nextState: targetFSM;

begin

nxtStProc: process ( currState, rd_r, wr_r)

begin

case currState is

    when State =>
         nextState <= State0;

    when State0 =>
      if (rd_r='0' and wr_r='1') then              -- read from memory
          nextState <= ReadState;
      elsif (rd_r='1' and wr_r='0') then           -- write to memory
          nextState <= WriteState;
      else
          nextState <= State1;
      end if;

    when State1 =>
         nextState <= State0;

    when ReadState =>
         nextState <= State0;

    when WriteState =>
         nextState <= State0;

end case;

end process nxtStProc;

-- Process to register the current state

curStProc: process (clk_r, reset_r)

begin

      if (reset_r ='0') then
          currState <= State;
      elsif (clk_r'event and clk_r='1') then
          currState <= nextState;
      end if;

end process curStProc;

-- Process to generate outputs
outConProc: process (currState, wr_r, pc_r, datain_a, datain_b,
                     datain_c, addrin_r, mem_data_wr)

begin

case currState is
```

```
-- without this state, state machine starts at State1 after reset

   when State =>                        -- generated for reset only
        null;

   when State0 =>                       -- doing instruction fetch
        state_r <= "0000";
        wrout_r <= wr_r;
        addrout_r <= pc_r;              -- sending pc to memory

     if (datain_a(23 downto 16)="11111000") then
         instr_data_a <= "000000000000000000000000";
         instr_data_b <= "000000000000000000000000";
         instr_data_c <= "000000000000000000000000";
     else
         instr_data_a <= datain_a;-- memory sends instruction to KDLX
         instr_data_b <= datain_b;
         instr_data_c <= datain_c;
     end if;

        dataout_r <= (others => 'Z');
        mem_data_a <= (others => 'Z');
        mem_data_b <= (others => 'Z');
        mem_data_c <= (others => 'Z');

   when State1 =>                      -- exactly the same as State0
                                       -- for keeping current state
        state_r <= "0001";
        wrout_r <= wr_r;
        addrout_r <= pc_r;

     if (datain_a(23 downto 16)="11111000") then
         instr_data_a <= "000000000000000000000000";
         instr_data_b <= "000000000000000000000000";
         instr_data_c <= "000000000000000000000000";
     else                          -- memory sends instruction to KDLX
         instr_data_a <= datain_a;
         instr_data_b <= datain_b;
         instr_data_c <= datain_c;
     end if;

        dataout_r <= (others => 'Z');
        mem_data_a <= (others => 'Z');
        mem_data_b <= (others => 'Z');
        mem_data_c <= (others => 'Z');


   when ReadState =>                 -- When KDLX reads data from memory
        state_r <= "0010";
        wrout_r <= wr_r;             -- write signal is one
        addrout_r <= addrin_r;       -- sending address to memory

        -- memory sends data to KDLX
        mem_data_a <= datain_a(15 downto 0);
        mem_data_b <= datain_b(15 downto 0);
        mem_data_c <= datain_c(15 downto 0);
        dataout_r <= (others => 'Z');          -- block input to memory
```

```
   when WriteState =>                -- When KDLX writes data to memory
        state_r <= "0011";
        wrout_r <= wr_r;             -- write signal is zero
        addrout_r <= addrin_r;       -- sending address to memory

        -- KDLX sends data to memory
        dataout_r(15 downto 0) <= mem_data_wr;
        dataout_r(23 downto 16) <= "00000000";   -- sign extension data

end case;

end process outConProc;

end fsm;
```

D. ESSD

```
--*********************************************************************
-- Module: Error Syndrome Storage Device (ESSD)
--
-- Function: The ESSD is used to store error syndrome when err occurs.
-- It runs in double speed and has the same time constraints with
-- Reconciler.  Stall KDLX at the beginning of ISR.
--
-- Notation: This ESSD works with the TMRA in this design only.  This
-- is the final version.
--
-- Author: Rong Yuan, TWAF
--
-- Date: Nov 21, 2003
--*********************************************************************



library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity essd is Port (
       addr_in: in std_logic_vector(15 downto 0);
       pc_in: in std_logic_vector(15 downto 0);
       cid1_in: in std_logic_vector(50 downto 0);
       cid0_in: in std_logic_vector(50 downto 0);
       err: in std_logic;
       reset_s: in std_logic;
       clk_s: in std_logic;

       stall_s: out std_logic;
       wr_s: out std_logic;
       sel_wr: out std_logic;
       addr_s: out std_logic_vector(15 downto 0);
       sel_addr: out std_logic_vector(15 downto 0);
       sel_s: out std_logic_vector(23 downto 0);
       ess: out std_logic_vector(23 downto 0);
       state_s: out std_logic_vector(4 downto 0)
       );
end essd;

architecture fsm of essd is

type targetFSM is (State, State0_A, State0_B, LatchState_A,
                   LatchState_B, NopState0_A, NopState0_B, NopState1_A,
                   NopState1_B, StallState, StoreState0_A,
                   StoreState0_B, StoreState0_C, StoreState1_A,
                   StoreState1_B, StoreState1_C, StoreState_addr,
                   StoreState_pc, BackState);

signal pc_latch, addr_latch: std_logic_vector(15 downto 0);
```

```
signal cid0_latchA, cid0_latchB, cid0_latchC, cid1_latchA, cid1_latchB,
       cid1_latchC: std_logic_vector(23 downto 0);
signal counter: std_logic_vector(15 downto 0);
signal currState, nextState: targetFSM;

begin

nxtStProc: process ( currState, err)

begin

case currState is

   when State =>
        nextState <= State0_A;

   when State0_A =>
        nextState <= State0_B;

   when State0_B =>
     if (err='1') then
         nextState <= LatchState_A;
     else
         nextState <= State0_A;
     end if;

   when LatchState_A =>
        nextState <= LatchState_B;

   when LatchState_B =>
        nextState <= NopState0_A;

   when NopState0_A =>
        nextState <= NopState0_B;

   when NopState0_B =>
        nextState <= NopState1_A;

   when NopState1_A =>
        nextState <= NopState1_B;

   when NopState1_B =>
        nextState <= StallState;

   when StallState =>
        nextState <= StoreState0_A;

   when StoreState0_A =>
        nextState <= StoreState0_B;

   when StoreState0_B =>
        nextState <= StoreState0_C;

   when StoreState0_C =>
        nextState <= StoreState1_A;
```

```
   when StoreState1_A =>
        nextState <= StoreState1_B;

   when StoreState1_B =>
        nextState <= StoreState1_C;

   when StoreState1_C =>
        nextState <= StoreState_addr;

   when StoreState_addr =>
        nextState <= StoreState_pc;

   when StoreState_pc =>
        nextState <= BackState;

   when BackState =>
        nextState <= State0_A;

end case;

end process nxtStProc;

-- Process to register the current state

curStProc: process (clk_s, reset_s)

begin

     if (reset_s ='0') then
         currState <= State;
     elsif (clk_s'event and clk_s='1') then
         currState <= nextState;
     end if;

end process curStProc;

-- Process to generate outputs
outConProc: process (currState, pc_in, addr_in, cid1_in, cid0_in)

begin
counter <= "0000000001011001";               --starting at address 0059

case currState is
   when State =>
        null;

   when State0_A =>
        state_s <= "00000";
        ess <= (others =>'Z');
        sel_s <= "111111111111111111111111";
        sel_wr <= '1';
        sel_addr <= "1111111111111111";
        stall_s <= '1';

   when State0_B =>
        state_s <= "00001";
        ess <= (others =>'Z');
```

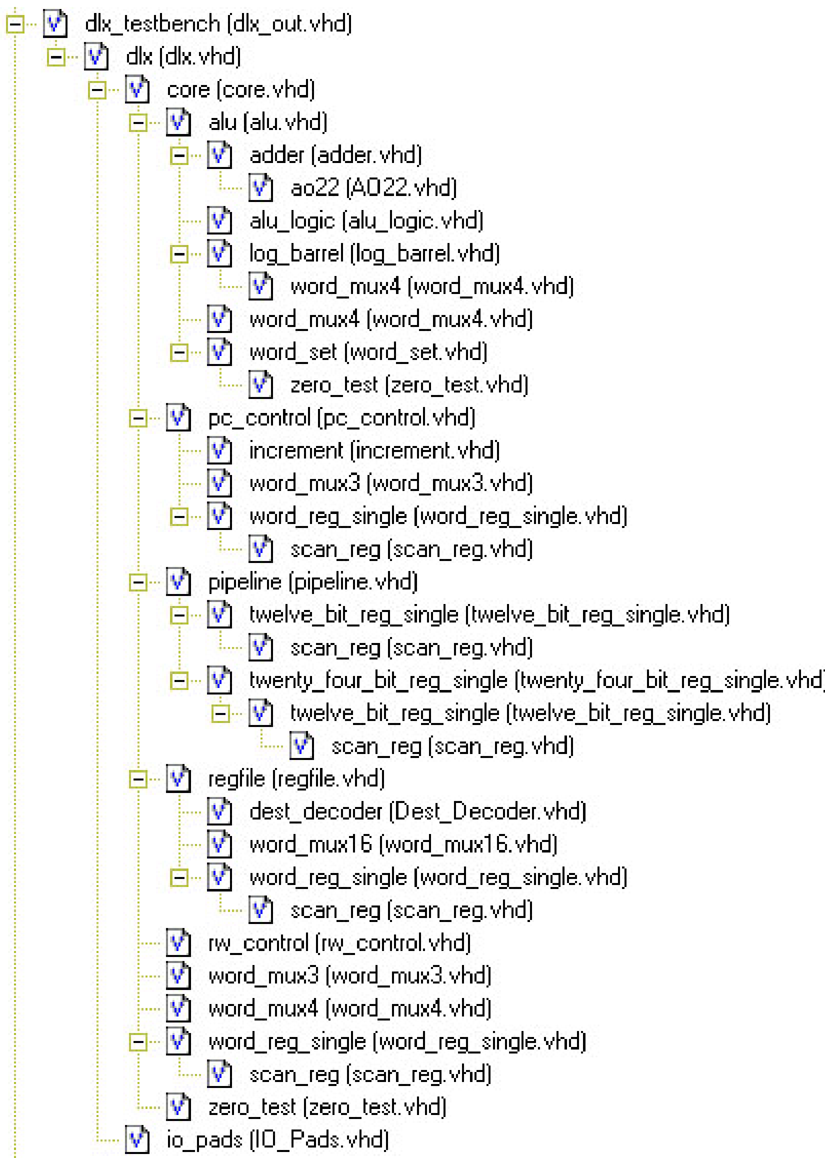
```
          sel_s <= "111111111111111111111111";
          sel_wr <= '1';
          sel_addr <= "1111111111111111";
          stall_s <= '1';

     when LatchState_A =>                              --latch all data here
          state_s <= "00010";
          sel_s <= "111111111111111111111111";
          sel_wr <= '1';
          sel_addr <= "1111111111111111";
          stall_s <= '1';
          pc_latch <= pc_in;
          addr_latch <= addr_in;

          --seperate input data
          cid1_latchC <= cid1_in(23 downto 0);
          cid1_latchB <= cid1_in(47 downto 24);
          cid1_latchA(2 downto 0) <= cid1_in(50 downto 48);
          cid1_latchA(23 downto 3) <= "000000000000000000000";
          cid0_latchC <= cid0_in(23 downto 0);
          cid0_latchB <= cid0_in(47 downto 24);
          cid0_latchA(2 downto 0) <= cid0_in(50 downto 48);
          cid0_latchA(23 downto 3) <= "000000000000000000000";

     when LatchState_B =>
          state_s <= "00011";
          sel_s <= "111111111111111111111111";
          sel_wr <= '1';
          sel_addr <= "1111111111111111";
       stall_s <= '1';

     when NopState0_A =>
          state_s <= "00100";
          sel_s <= "111111111111111111111111";
          sel_wr <= '1';
          sel_addr <= "1111111111111111";
          stall_s <= '1';

     when NopState0_B =>
          state_s <= "00101";
          sel_s <= "111111111111111111111111";
          sel_wr <= '1';
          sel_addr <= "1111111111111111";
          stall_s <= '1';

     when NopState1_A =>
          state_s <= "00110";
          sel_s <= "111111111111111111111111";
          sel_wr <= '1';
          sel_addr <= "1111111111111111";
          stall_s <= '1';

     when NopState1_B =>
          state_s <= "00111";
          sel_s <= "111111111111111111111111";
          sel_wr <= '1';
          sel_addr <= "1111111111111111";
```

```
          stall_s <= '1';

     when StallState =>           --stall KDLX
          state_s <= "01000";
          sel_s <= "111111111111111111111111";
          sel_wr <= '1';
          sel_addr <= "1111111111111111";
          stall_s <= '0';

     when StoreState0_A =>        --store cid0
          state_s <= "01001";
          sel_s <= "000000000000000000000000";
          sel_wr <= '0';
          sel_addr <= "0000000000000000";
          stall_s <= '0';
          addr_s <= counter;
          wr_s <= '0';
          ess <= cid0_latchC;
          counter <= counter-1;

     when StoreState0_B =>
          state_s <= "01010";
          sel_s <= "000000000000000000000000";
          sel_wr <= '0';
          sel_addr <= "0000000000000000";
          stall_s <= '0';
          addr_s <= counter;
          wr_s <= '0';
          ess <= cid0_latchB;
          counter <= counter-1;

     when StoreState0_C =>
          state_s <= "01011";
          sel_s <= "000000000000000000000000";
          sel_wr <= '0';
          sel_addr <= "0000000000000000";
          stall_s <= '0';
          addr_s <= counter;
          wr_s <= '0';
          ess <= cid0_latchA;
          counter <= counter-1;

     when StoreState1_A =>        --store cid1
          state_s <= "01100";
          sel_s <= "000000000000000000000000";
          sel_wr <= '0';
          sel_addr <= "0000000000000000";
          stall_s <= '0';
          addr_s <= counter;
          wr_s <= '0';
          ess <= cid1_latchC;
          counter <= counter-1;

     when StoreState1_B =>
          state_s <= "01101";
          sel_s <= "000000000000000000000000";
          sel_wr <= '0';
```

```
        sel_addr <= "0000000000000000";
        stall_s <= '0';
        addr_s <= counter;
        wr_s <= '0';
        ess <= cid1_latchB;
        counter <= counter-1;

   when StoreState1_C =>
        state_s <= "01110";
        sel_s <= "000000000000000000000000";
        sel_wr <= '0';
        sel_addr <= "0000000000000000";
        stall_s <= '0';
        addr_s <= counter;
        wr_s <= '0';
        ess <= cid1_latchA;
        counter <= counter-1;

   when StoreState_addr =>      --store mem addr
              state_s <= "01111";
        sel_s <= "000000000000000000000000";
        sel_wr <= '0';
        sel_addr <= "0000000000000000";
        stall_s <= '0';
        addr_s <= counter;
        wr_s <= '0';
        ess(15 downto 0) <= addr_latch;
        ess(23 downto 16) <= "00000000";
        counter <= counter-1;

   when StoreState_pc =>        --store pc
        state_s <= "10000";
        sel_s <= "000000000000000000000000";
        sel_wr <= '0';
        sel_addr <= "0000000000000000";
        stall_s <= '0';
        addr_s <= counter;
        wr_s <= '0';
        ess(15 downto 0) <= pc_latch;
        ess(23 downto 16) <= "00000000";
        counter <= counter-1;

   when BackState =>            --release KDLX
        state_s <= "10001";
        sel_s <= "111111111111111111111111";
        sel_wr <= '1';
        sel_addr <= "1111111111111111";
        stall_s <= '1';
        addr_s <= (others =>'Z');
        wr_s <= '1';
        ess <= (others =>'Z');

end case;
end process outConProc;
end fsm;
```

E. KDLX

The KDLX is a 16-bit RISC soft-core processor. It is 5-stage pipelined including fetch, decode, execute, memory, and write back. The KDLX is coded by Dr. Kenneth Clark and following is the construction of the source core in ISE software.



1. alu.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;

-- ***** adder model *****
-- external ports
ENTITY adder IS PORT (
      A : IN std_logic_vector(15 downto 0);
      B:   IN std_logic_vector(15 downto 0);
      alu_op1 : IN std_logic;
      alu_op3 : IN std_logic;
      alu_op4 : IN std_logic;
      Out_word : OUT std_logic_vector(15 downto 0)
);
END adder;

-- internal structure
ARCHITECTURE rtl OF adder IS

-- COMPONENTS

COMPONENT AO22
PORT (
      A : IN std_logic;
      B : IN std_logic;
      C : IN std_logic;
      D : IN std_logic;
      \Out\ : OUT std_logic
);
END COMPONENT;

SIGNAL Vdd : std_logic;
SIGNAL subtract : std_logic;
-- INSTANCES
BEGIN
Vdd <= '1';
AO22_1 : AO22   PORT MAP(
      A => Vdd,
      B => alu_op1,
      C => alu_op4,
      D => alu_op3,
      \Out\ => subtract
);

process (A, B, subtract)
begin
  if (subtract = '1') then
   out_word <= A-B;
  else out_word <= A+B;
  end if;
end process;
END rtl;
```

2. alu.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** alu model *****
-- external ports
ENTITY alu IS PORT (
      A : IN std_logic_vector (15 downto 0);
      alu_op : IN std_logic_vector (4 downto 0);
      alu_out : OUT std_logic_vector (15 downto 0);
      B : IN std_logic_vector (15 downto 0)
);
END alu;

-- internal structure
ARCHITECTURE structural OF alu IS

-- COMPONENTS

COMPONENT adder
PORT (
      A : IN std_logic_vector(15 downto 0);
      B : IN std_logic_vector (15 downto 0);
      alu_op1 : IN std_logic;
      alu_op3 : IN std_logic;
      alu_op4 : IN std_logic;
      Out_word : OUT std_logic_vector (15 downto 0)
);
END COMPONENT;

COMPONENT alu_logic
PORT (
      A : IN std_logic_vector (15 downto 0);
      B : IN std_logic_vector (15 downto 0);
      Func : IN std_logic_vector (1 downto 0);
      logic_out : OUT std_logic_vector (15 downto 0)
);
END COMPONENT;

COMPONENT log_barrel
PORT (
      ar_or_log : IN std_logic;
      In_Word : IN std_logic_vector (15 downto 0);
      l_or_r : IN std_logic;
      Out_word : OUT std_logic_vector (15 downto 0);
      Shift : IN std_logic_vector (3 downto 0)
);
END COMPONENT;

COMPONENT word_mux4
PORT (
      A : IN std_logic_vector (15 downto 0);
      B : IN std_logic_vector (15 downto 0);
      C : IN std_logic_vector (15 downto 0);
      D : IN std_logic_vector (15 downto 0);
      Sel : IN std_logic_vector (1 downto 0);
```

```
      Out_word : OUT std_logic_vector (15 downto 0)
);
END COMPONENT;

COMPONENT word_set
PORT (
      In_word : IN std_logic_vector (15 downto 0);
      set_op : IN std_logic_vector (2 downto 0);
      set_out : OUT std_logic
);
END COMPONENT;

-- SIGNALS
SIGNAL set_out : std_logic_vector (15 downto 0);
SIGNAL log_barrel_out : std_logic_vector (15 downto 0);
SIGNAL logic_out : std_logic_vector (15 downto 0);
SIGNAL Adder_Out : std_logic_vector (15 downto 0);

-- INSTANCES
BEGIN
set_out(15 downto 1) <= "000000000000000";
halfword_adder_1 : adder   PORT MAP(
      A => A,
      alu_op1 => alu_op(1),
      alu_op3 => alu_op(3),
      alu_op4 => alu_op(4),
      B => B,
      Out_word  => Adder_Out
);
halfword_alu_logic_1 : alu_logic   PORT MAP(
      A => A,
      B => B,
      Func => alu_op(1 downto 0),
      logic_out => logic_out
);
halfword_log_barrel_1 : log_barrel   PORT MAP(
      ar_or_log => alu_op(0),
      In_word => A,
      l_or_r => alu_op(1),
        Out_word => log_barrel_out,
      Shift => B(3 downto 0)
);
halfword_mux4_1 : word_mux4   PORT MAP(
      A => Adder_Out,
      B => logic_out,
      C => log_barrel_out,
      D => set_out,
      Out_word => alu_out,
      Sel => alu_op(4 downto 3)
);
halfword_set_1 : word_set   PORT MAP(
      In_word => Adder_Out,
      set_op => alu_op(2 downto 0),
      set_out => set_out(0)
);
END structural;
```

3. alu_logic.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** alu_logic model *****
-- external ports
ENTITY alu_logic IS PORT (
      A: IN std_logic_vector(15 downto 0);
      B : IN std_logic_vector(15 downto 0);
      Func: IN std_logic_vector(1 downto 0);
      logic_out : OUT std_logic_vector(15 downto 0)
);
END alu_logic;

-- internal structure
ARCHITECTURE rtl OF alu_logic IS

BEGIN

process (A,B, func)
begin
  case func is
   when "00" => logic_out <= A;
   when "01" => logic_out <= (A and B);
   when "10" => logic_out <= (A or B);
   when others => logic_out <= (A xor B);
   end case;
end process;

END rtl;
```

4. AO22.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;


entity AO22 is port (
  A, B, C, D: IN std_logic;
  \Out\ : OUT std_logic);
end AO22;

architecture behavioral of AO22 is
begin
  \Out\ <= (A and B) or (C and D);
end behavioral;
```

5. core.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;
-- ***** core model *****
-- external ports
ENTITY core IS PORT (
      Addr_Int : OUT std_logic_vector(15 downto 0);
      Clock_in : IN std_logic;
      Input_Data : IN std_logic_vector(15 downto 0);
  Output_Data : Out std_logic_vector(15 downto 0);
      Instr : IN std_logic_vector(23 downto 0);
      PC : OUT std_logic_vector(15 downto 0);
      Prog_Rd : OUT std_logic;
      Rd : OUT std_logic;
      Resetn : IN std_logic;
      Stalln : IN std_logic;
      Wr : OUT std_logic
);
END core;

-- internal structure
ARCHITECTURE structural OF core IS

-- COMPONENTS

COMPONENT alu
PORT (
      A : IN std_logic_vector(15 downto 0);
      alu_op : IN std_logic_vector(4 downto 0);
      alu_out : OUT std_logic_vector(15 downto 0);
      B : IN std_logic_vector(15 downto 0)
);
END COMPONENT;


COMPONENT word_mux3
PORT (
      A : IN std_logic_vector(15 downto 0);
      B : IN std_logic_vector(15 downto 0);
      C : IN std_logic_vector(15 downto 0);
      Out_word : OUT std_logic_vector(15 downto 0);
      Sel : IN std_logic_vector(1 downto 0)
);
END COMPONENT;

COMPONENT word_mux4
PORT (
      A : IN std_logic_vector(15 downto 0);
      B : IN std_logic_vector(15 downto 0);
      C : IN std_logic_vector(15 downto 0);
      D : IN std_logic_vector(15 downto 0);
      Out_word : OUT std_logic_vector(15 downto 0);
      Sel : IN std_logic_vector(1 downto 0)
);
END COMPONENT;
```

```
COMPONENT regfile
PORT (
      A : OUT std_logic_vector(15 downto 0);
      B : OUT std_logic_vector(15 downto 0);
      clock : IN std_logic;
      Data_In : IN std_logic_vector(15 downto 0);
      Dest : IN std_logic_vector(3 downto 0);
        stalln: IN std_logic;
      resetn : IN std_logic;
      RSone : IN std_logic_vector(3 downto 0);
      RStwo : IN std_logic_vector(3 downto 0);
      scan_data_in : IN std_logic;
      scan_enable : IN std_logic;
      wb_enable : IN std_logic
);
END COMPONENT;

COMPONENT word_reg_single
PORT (
      Clock : IN std_logic;
      Data_In : IN std_logic_vector(15 downto 0);
      Data_out : OUT std_logic_vector(15 downto 0);
      Enable : IN std_logic;
      Resetn : IN std_logic;
      Scan_Data_In : IN std_logic;
      Scan_Enable : IN std_logic
);
END COMPONENT;

COMPONENT pc_control
PORT (
      ALU_Out : IN std_logic_vector(15 downto 0);
      Clock : IN std_logic;
      D2_Inc_PC : OUT std_logic_vector(15 downto 0);
        D_Link_PC : OUT std_logic_vector(15 downto 0);
      IAR_Enable : IN std_logic;
      PC : OUT std_logic_vector(15 downto 0);
      PC_Sel : IN std_logic_vector(1 downto 0);
      Resetn : IN std_logic;
      Scan_Data_In : IN std_logic;
      Scan_Data_Out : OUT std_logic;
      Scan_Enable : IN std_logic;
      Stalln : IN std_logic
);
END COMPONENT;

COMPONENT pipeline
PORT (
alu_op : OUT std_logic_vector(4 downto 0);
        A_Mux : OUT std_logic_vector(1 downto 0);
        B_Mux : OUT std_logic_vector(1 downto 0);
        Clock : IN std_logic;
        Data_In : IN std_logic_vector(23 downto 0);
        Dest : OUT std_logic_vector(3 downto 0);
        Immed : OUT std_logic_vector(15 downto 0);
        PC_Sel : OUT std_logic_vector(1 downto 0);
```

```
        rd_enable : OUT std_logic;
        Reg_In_Sel : OUT std_logic_vector(1 downto 0);
        Resetn : IN std_logic;
        RSone : OUT std_logic_vector(3 downto 0);
        RStwo : OUT std_logic_vector(3 downto 0);
        Scan_Data_In : IN std_logic;
        Scan_Enable : IN std_logic;
        Stalln : IN std_logic;
        wb_enable : OUT std_logic;
        scan_out : OUT std_logic;
        IAR_Enable : OUT std_logic;
        wr_enable : OUT std_logic;
        zero_flag : IN std_logic
);
END COMPONENT;

COMPONENT rw_control
PORT (
Clock : IN std_logic;
        Prog_Rd : OUT std_logic;
        Rd : OUT std_logic;
        rd_enable : IN std_logic;
        resetn : IN std_logic;
        stalln : IN std_logic;
        Wr : OUT std_logic;
        wr_enable : IN std_logic
);
END COMPONENT;


COMPONENT zero_test
PORT (
      In_word : IN std_logic_vector(15 downto 0);
      zero_flag : OUT std_logic
);
END COMPONENT;

-- SIGNALS

SIGNAL wr_enable : std_logic;
SIGNAL zero_flag : std_logic;
SIGNAL IAR_Enable : std_logic;
SIGNAL wb_enable : std_logic;
SIGNAL pipeline_scan_out : std_logic;
SIGNAL Dest : std_logic_vector(3 downto 0);
SIGNAL A : std_logic_vector(15 downto 0);
SIGNAL D2_Inc_PC : std_logic_vector(15 downto 0);
SIGNAL Immed : std_logic_vector(15 downto 0);
SIGNAL D_ALU_Out : std_logic_vector(15 downto 0);
SIGNAL D_Link_PC : std_logic_vector(15 downto 0);
SIGNAL Reg_In_Sel : std_logic_vector(1 downto 0);
SIGNAL ALU_A : std_logic_vector(15 downto 0);
SIGNAL ALU_Out : std_logic_vector(15 downto 0);
SIGNAL ALU_B : std_logic_vector(15 downto 0);
SIGNAL Gnd : std_logic;
SIGNAL B : std_logic_vector(15 downto 0);
SIGNAL LD_Memory_In : std_logic_vector(15 downto 0);
```

```
SIGNAL output_en_n : std_logic;
SIGNAL rd_enable : std_logic;
SIGNAL pc_control_scan_out : std_logic;
SIGNAL Buf_Stalln : std_logic;
SIGNAL Buf_resetn : std_logic;
SIGNAL Clock : std_logic;
SIGNAL Buf_Addr_Int : std_logic_vector(15 downto 0);
SIGNAL Shift_En : std_logic;
SIGNAL alu_op : std_logic_vector(4 downto 0);
SIGNAL Buf_Scan_Data_Out : std_logic;
SIGNAL A_Mux : std_logic_vector(1 downto 0);
SIGNAL B_Mux : std_logic_vector(1 downto 0);
SIGNAL RSone : std_logic_vector(3 downto 0);
SIGNAL RStwo : std_logic_vector(3 downto 0);
SIGNAL PC_Sel : std_logic_vector(1 downto 0);
SIGNAL Data_Out : std_logic_vector(15 downto 0);
SIGNAL Regfile_In : std_logic_vector(15 downto 0);
SIGNAL zero_byte : std_logic_vector(7 downto 0);
SIGNAL Data_In : std_logic_vector(15 downto 0);
SIGNAL sign_ext_immed : std_logic_vector(15 downto 0);
SIGNAL scan_data_in : std_logic;
-- INSTANCES
BEGIN
clock <= clock_in;
shift_en <= '0';
scan_data_in <= '0';
Addr_Int <= Buf_Addr_Int;
zero_byte <= "00000000";
sign_ext_immed(15 downto 8) <= Immed(7) & Immed(7) & Immed(7) &
Immed(7) & Immed(7) & Immed(7) & Immed(7) & Immed(7);
sign_ext_immed (7 downto 0) <= Immed(7 downto 0);
Wr <= output_en_n;
Output_Data <= Data_Out;

Word_Reg_1 : word_reg_single PORT MAP(
      Clock => Clock,
      Data_In => B,
      Data_out => Data_Out,
      Enable => Stalln,
      Resetn => Resetn,
      Scan_Data_In => pc_control_scan_out,
      Scan_Enable => Shift_En
);

Word_Reg_2 : word_reg_single PORT MAP(
      Clock => Clock,
      Data_In => Input_Data,
      Data_out => LD_Memory_In,
      Enable => Stalln,
      Resetn => Resetn,
      Scan_Data_In => Data_Out(15),
      Scan_Enable => Shift_En
);
```

```
alu_1 : alu   PORT MAP(
      A => ALU_A,
      alu_op => alu_op,
      alu_out => ALU_Out,
      B => ALU_B
);
word_mux3_1 : word_mux3   PORT MAP(
      A => D_ALU_Out,
      B => LD_Memory_In,
      C => D_Link_PC,
      Out_word => Regfile_In,
      Sel => Reg_In_Sel
);
word_mux3_2 : word_mux3   PORT MAP(
      A => B,
      B(7 downto 0) => Immed(7 downto 0),
      B(15 downto 8) => zero_byte,
      C => sign_ext_immed,
      Out_word => ALU_B,
      Sel => B_Mux
);
word_mux4_1 : word_mux4   PORT MAP(
      A => A,
      B => D2_Inc_PC,
      C(7 downto 0) => zero_byte,
      C(15 downto 8) => Immed(7 downto 0),
      D => Immed(15 downto 0),
      Out_word => ALU_A,
      Sel => A_Mux
);
regfile_1 : regfile   PORT MAP(
      A => A,
      B => B,
      clock => Clock,
      Data_In => regfile_in,
      Dest => Dest,
        stalln => stalln,
      resetn => resetn,
      RSone => RSone,
      RStwo => RStwo,
      scan_data_in => pipeline_scan_out,
      scan_enable => Shift_En,
      wb_enable => wb_enable
);
word_reg_single_3 : word_reg_single   PORT MAP(
      Clock => Clock,
      Data_In => Buf_Addr_Int,
      Data_out => D_ALU_Out,
      Enable => Stalln,
      Resetn => resetn,
      Scan_Data_In => Buf_Addr_Int(15),
      Scan_Enable => Shift_En
);
word_reg_single_4 : word_reg_single   PORT MAP(
      Clock => Clock,
      Data_In => ALU_Out,
      Data_out => Buf_Addr_Int,
```

```
      Enable => Stalln,
      Resetn => resetn,
      Scan_Data_In => B(15),
      Scan_Enable => Shift_En
);
pc_control_1 : pc_control   PORT MAP(
      ALU_Out => ALU_Out,
      Clock => Clock,
      D2_Inc_PC => D2_Inc_PC,
      D_Link_PC => D_Link_PC,
      IAR_Enable => IAR_Enable,
      PC => PC,
      PC_Sel => PC_Sel,
      Resetn => resetn,
      Scan_Data_In => D_ALU_Out(15),
      Scan_Data_Out => pc_control_scan_out,
      Scan_Enable => Shift_En,
      Stalln => Stalln
);
pipeline_1 : pipeline   PORT MAP(
      alu_op => alu_op,
      A_Mux => A_Mux,
      B_Mux => B_Mux,
      Clock => Clock,
      Data_In => Instr,
      Dest => Dest,
      Immed => Immed,
      PC_Sel => PC_Sel,
      rd_enable => rd_enable,
      Reg_In_Sel => Reg_In_Sel,
      Resetn => resetn,
      RSone => RSone,
      RStwo => RStwo,
      Scan_Data_In => Scan_Data_In,
      Scan_Enable => Shift_En,
      Stalln => Stalln,
      wb_enable => wb_enable,
      scan_out => pipeline_scan_out,
      IAR_Enable => IAR_Enable,
      wr_enable => wr_enable,
      zero_flag => zero_flag
);
rw_control_1 : rw_control   PORT MAP(
      Clock => Clock,
      Prog_Rd => Prog_Rd,
      Rd => Rd,
      rd_enable => rd_enable,
      resetn => resetn,
      stalln => Stalln,
      Wr => output_en_n,
      wr_enable => wr_enable
);
zero_test_1 : zero_test   PORT MAP(
      In_word => A,
      zero_flag => zero_flag
);
END structural;
```

6. Dest_Decoder.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** Dest_Decoder model *****
-- external ports
ENTITY Dest_Decoder IS PORT (
      Dest : IN std_logic_vector(3 downto 0);
      Enable : OUT std_logic_vector(15 downto 1);
      wb_enable : IN std_logic
);
END Dest_Decoder;

-- internal structure
ARCHITECTURE rtl OF Dest_Decoder IS

-- SIGNALS
SIGNAL buf_enable : std_logic_vector(15 downto 1);

-- INSTANCES
BEGIN
with dest select
buf_enable <=  "000000000000001" when  "0001",
               "000000000000010" when  "0010",
               "000000000000100" when  "0011",
               "000000000001000" when  "0100",
               "000000000010000" when  "0101",
               "000000000100000" when  "0110",
               "000000001000000" when  "0111",
               "000000010000000" when  "1000",
               "000000100000000" when  "1001",
               "000001000000000" when  "1010",
               "000010000000000" when  "1011",
               "000100000000000" when  "1100",
               "001000000000000" when  "1101",
               "010000000000000" when  "1110",
               "100000000000000" when others;

      Enable <= buf_enable when (wb_enable = '1') else
"000000000000000";
      END rtl;
```

7. dlx.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;

-- ***** dlx model *****

-- external ports

ENTITY dlx IS PORT (
      Addr_Int : OUT std_logic_vector(15 downto 0);
      Clock_in : IN std_logic;
      Data : INOUT std_logic_vector(15 downto 0);
      Instr : IN std_logic_vector(23 downto 0);
      PC : OUT std_logic_vector(15 downto 0);
      Prog_Rd : OUT std_logic;
      Rd : OUT std_logic;
      Resetn : IN std_logic;
      Stalln : IN std_logic;
      Wr : OUT std_logic
);
END dlx;

-- internal structure

ARCHITECTURE structural OF dlx IS

-- COMPONENTS
COMPONENT core

PORT (
      Addr_Int : OUT std_logic_vector(15 downto 0);
      Clock_in : IN std_logic;
      Input_Data : IN std_logic_vector(15 downto 0);
      Output_Data : Out std_logic_vector(15 downto 0);
      Instr : IN std_logic_vector(23 downto 0);
      PC : OUT std_logic_vector(15 downto 0);
      Prog_Rd : OUT std_logic;
      Rd : OUT std_logic;
      Resetn : IN std_logic;
      Stalln : IN std_logic;
      Wr : OUT std_logic
);

END COMPONENT;

COMPONENT IO_Pads

PORT (
      Pads : INOUT std_logic_vector (15 downto 0);
      In_Data : OUT std_logic_vector (15 downto 0);
      Out_Data : IN std_logic_vector (15 downto 0);
      Output_En_n : IN std_logic
);
END COMPONENT;
```

```
-- SIGNALS
signal Input_data : std_logic_vector(15 downto 0);
signal Output_data : std_logic_vector(15 downto 0);
signal wr_int : std_logic;

-- INSTANCES
BEGIN

wr <= wr_int;

core1 : core PORT MAP(
      Addr_Int => Addr_Int,
      Clock_in => Clock_In,
      Input_Data => Input_data,
      Output_Data => Output_data,
      Instr => Instr,
      PC => PC,
      Prog_Rd => Prog_Rd,
      Rd => Rd,
      Resetn => Resetn,
      Stalln => stalln,
      Wr => Wr_int
);

IO_Pads_1 : IO_Pads PORT MAP(
    Pads => Data,
    In_Data => Input_Data,
    Out_Data => Output_Data,
    Output_En_n => wr_int
);

END structural;
```

8. dlx_out.vhd

```
-- Test bench shell

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dlx_testbench is end dlx_testbench;

architecture testbench of dlx_testbench is

-- Declaration of the component under test

component DLX

  port (
      Addr_Int : OUT std_logic_vector(15 downto 0);
      Clock_in : IN std_logic;
      Data : INOUT std_logic_vector(15 downto 0);
      Instr : IN std_logic_vector(23 downto 0);
      PC : OUT std_logic_vector(15 downto 0);
```

```
      Prog_Rd : OUT std_logic;
      Rd : OUT std_logic;
      Resetn : IN std_logic;
      Stalln : IN std_logic;
      Wr : OUT std_logic
      );
end component;

signal   addr_int : std_logic_vector(15 downto 0);
signal   instr : std_logic_vector(23 downto 0);
signal   pc : std_logic_vector(15 downto 0);
signal   data : std_logic_vector(15 downto 0);
signal   resetn : std_logic;
signal   prog_rd : std_logic;
signal   rd : std_logic;
signal   wr : std_logic;
signal   stalln : std_logic;
signal   clock_in : std_logic;

begin

process   --- 10 MHz clock
begin
  clock_in <= '0';

  wait for 25 ns;

  clock_in <= '0';

  wait for 25 ns;

  clock_in <= '1';

  wait for 25 ns;

  clock_in <= '0';

  wait for 25 ns;

end process;



process
begin     ---- power up reset process

wait for 1 ns;

resetn <= '0';
stalln <= '1';

wait for 10 ns;

 resetn <= '1';
  wait;

end process;
```

```
process
begin
wait for 1 ns;
instr <= X"000000";    --- NOP
data <=  "ZZZZZZZZZZZZZZZZ";
wait for 100 ns;
instr <= X"080101";   --- LHI R1, #1
wait for 100 ns;
instr <= X"080202";   --- LHI R2, #2
wait for 100 ns;
instr <= X"080303";   --- LHI R3, #3
wait for 100 ns;
instr <= X"080404";   --- LHI R4, #4
wait for 100 ns;
instr <= X"080505";   --- LHI R5, #5
wait for 100 ns;
instr <= X"080606";   --- LHI R6, #6
wait for 100 ns;
instr <= X"080707";   --- LHI R7, #7
wait for 100 ns;
instr <= X"080808";   --- LHI R8, #8
wait for 100 ns;
instr <= X"080909";   --- LHI R9, #9
wait for 100 ns;
instr <= X"080A0A";   --- LHI R10, #10
wait for 100 ns;
instr <= X"080B0B";   --- LHI R11, #11
wait for 100 ns;
instr <= X"080C0C";   --- LHI R12, #12
```

```
wait for 100 ns;

instr <= X"080D0D";   --- LHI R13, #13

wait for 100 ns;

instr <= X"080E0E";   --- LHI R14, #14

wait for 100 ns;

instr <= X"080F0F";   --- LHI R15, #15

wait for 100 ns;

instr <= X"4111FE";   --- ADDI R1, R1, FE

wait for 100 ns;

instr <= X"2122FD";   --- ADDUI R2, R2, FD

wait for 100 ns;

instr <= X"013340";   --- ADD R3, R3, R4

wait for 100 ns;

instr <= X"4344FF";   --- SUBI R4, R4, FF

wait for 100 ns;

instr <= X"235501";   --- SUBUI R5, R5, #1

wait for 100 ns;

instr <= X"036670";   --- SUB R6, R6, R7

wait for 100 ns;

instr <= X"2977FF";   --- ANDI R7, R7, FF

wait for 100 ns;

instr <= X"098880";   --- AND R8, R8, R9

wait for 100 ns;

instr <= X"2A99FF";   --- ORI R9, R9, FF

wait for 100 ns;

instr <= X"0AAAB0";   --- OR R10, R10, R11

wait for 100 ns;

instr <= X"2BBBF0";   --- XORI R11, R11, F0
```

```
wait for 100 ns;

instr <= X"0BCCD0";   --- XOR R12, R12, R13

wait for 100 ns;

instr <= X"450100";   --- SW R0, R1

wait for 100 ns;

instr <= X"451200";   --- SW R1, R2

wait for 100 ns;

instr <= X"452300";   --- SW R2, R3

wait for 100 ns;

instr <= X"453400";   --- SW R3, R4

wait for 100 ns;

instr <= X"454500";   --- SW R4, R5

wait for 100 ns;

instr <= X"455600";   --- SW R5, R6

wait for 100 ns;

instr <= X"456700";   --- SW R6, R7

wait for 100 ns;

instr <= X"457800";   --- SW R7, R8

wait for 100 ns;

instr <= X"458900";   --- SW R8, R9

wait for 100 ns;

instr <= X"459A00";   --- SW R9, R10

wait for 100 ns;

instr <= X"45AB00";   --- SW R10, R11

wait for 100 ns;

instr <= X"45BC00";   --- SW R11, R12

wait for 100 ns;

instr <= X"45CD00";   --- SW R12, R13

wait for 100 ns;
```

```
instr <= X"311104";   --- SLLI R1, R1, #4

wait for 100 ns;

instr <= X"112240";   --- SLL R2, R2, R4

wait for 100 ns;

instr <= X"326304";   --- SRLI R3, R6, #4

wait for 100 ns;

instr <= X"126440";   --- SRL R4,R6,R4

wait for 100 ns;

instr <= X"336504";   --- SRAI R5, R6, #4

wait for 100 ns;

instr <= X"136640";   --- SRA R6, R6, R4

wait for 100 ns;

instr <= X"387701";   --- SEQI R7, R7, #1

wait for 100 ns;

instr <= X"387800";   --- SEQI R8, R7, #0

wait for 100 ns;

instr <= X"3D7900";   --- SNEI R9, R7, #0

wait for 100 ns;

instr <= X"3D7A01";   --- SNEI R10, R7, #1

wait for 100 ns;

instr <= X"1D1B10";   --- SNE R11, R1, R1

wait for 100 ns;

instr <= X"1D1C20";   --- SNE R12, R1, R2

wait for 100 ns;

instr <= X"3C7D00";   --- SLTI R13, R7, #0

wait for 100 ns;

instr <= X"3C7E01";   --- SLTI R13, R7, #0

wait for 100 ns;
```

```
instr <= X"450100";   --- SW R0, R1

wait for 100 ns;

instr <= X"451200";   --- SW R1, R2

wait for 100 ns;

instr <= X"452300";   --- SW R2, R3

wait for 100 ns;

instr <= X"453400";   --- SW R3, R4

wait for 100 ns;

instr <= X"454500";   --- SW R4, R5

wait for 100 ns;

instr <= X"455600";   --- SW R5, R6

wait for 100 ns;

instr <= X"456700";   --- SW R6, R7

wait for 100 ns;

instr <= X"457800";   --- SW R7, R8

wait for 100 ns;

instr <= X"458900";   --- SW R8, R9

wait for 100 ns;

instr <= X"459A00";   --- SW R9, R10

wait for 100 ns;

instr <= X"45AB00";   --- SW R10, R11

wait for 100 ns;

instr <= X"45BC00";   --- SW R11, R12

wait for 100 ns;

instr <= X"45CD00";   --- SW R12, R13

wait for 100 ns;

instr <= X"45DE00";   --- SW R13, R14

wait for 100 ns;

instr <= X"187180";   --- SEQ R1, R7, R8
```

```
wait for 100 ns;
instr <= X"187290";   --- SEQ R2, R7, R9
wait for 100 ns;
instr <= X"1C7360";   --- SLT R3, R7, R6
wait for 100 ns;
instr <= X"1C6470";   --- SLT R4, R6, R7
wait for 100 ns;
instr <= X"1A6570";   --- SGT R5, R6, R7
wait for 100 ns;
instr <= X"1A7660";   --- SGT R6, R7, R6
wait for 100 ns;
instr <= X"5A8701";   --- SGTI R8, R7, #1
wait for 100 ns;
instr <= X"5A8800";   --- SGTI R8, R8, 0
wait for 100 ns;
instr <= X"5BB9FF";   --- SLEI R9, R11, FF
wait for 100 ns;
instr <= X"5BBA01";   --- SLEI R10, R11, #1
wait for 100 ns;
instr <= X"5BBB02";   --- SLEI R11, R11, #2
wait for 100 ns;
instr <= X"1B2C10";   --- SLE R12, R2, R1
wait for 100 ns;
instr <= X"1B2D40";   --- SLE R13, R2, R4
wait for 100 ns;
instr <= X"1B1E20";   --- SLE R14, R1, R2
wait for 100 ns;
instr <= X"450100";   --- SW R0, R1
```

```
wait for 100 ns;

instr <= X"451200";   --- SW R1, R2

wait for 100 ns;

instr <= X"452300";   --- SW R2, R3

wait for 100 ns;

instr <= X"453400";   --- SW R3, R4

wait for 100 ns;

instr <= X"454500";   --- SW R4, R5

wait for 100 ns;

instr <= X"455600";   --- SW R5, R6

wait for 100 ns;

instr <= X"456700";   --- SW R6, R7

wait for 100 ns;

instr <= X"457800";   --- SW R7, R8

wait for 100 ns;

instr <= X"458900";   --- SW R8, R9

wait for 100 ns;

instr <= X"459A00";   --- SW R9, R10

wait for 100 ns;

instr <= X"45AB00";   --- SW R10, R11

wait for 100 ns;

instr <= X"45BC00";   --- SW R11, R12

wait for 100 ns;

instr <= X"45CD00";   --- SW R12, R13

wait for 100 ns;

instr <= X"45DE00";   --- SW R13, R14

wait for 100 ns;

instr <= X"191120";   --- SGE R1, R1, R2

wait for 100 ns;
```

```
instr <= X"192210";   --- SGE R2, R2, R1

wait for 100 ns;

instr <= X"192320";   --- SGE R3, R2, R2

wait for 100 ns;

instr <= X"595402";   --- SGEI R4, R5, #02

wait for 100 ns;

instr <= X"5955FF";   --- SGEI R5, R5, FF

wait for 100 ns;

instr <= X"596500";   --- SGEI R6, R5, #0

wait for 100 ns;

instr <= X"450100";   --- SW R0, R1

wait for 100 ns;

instr <= X"451200";   --- SW R1, R2

wait for 100 ns;

instr <= X"452300";   --- SW R2, R3

wait for 100 ns;

instr <= X"453400";   --- SW R3, R4

wait for 100 ns;

instr <= X"454500";   --- SW R4, R5

wait for 100 ns;

instr <= X"455600";   --- SW R5, R6

wait for 100 ns;

instr <= X"C800FF";   --- J 0x00FF

wait for 100 ns;

instr <= X"000000";   --- NOP

wait for 100 ns;

instr <= X"000000";   --- NOP

wait for 100 ns;
```

```
instr <= X"000000";   --- NOP

wait for 100 ns;

instr <= X"000000";   --- NOP

wait for 100 ns;

instr <= X"E88000";   --- JAL 0x8000

wait for 100 ns;

instr <= X"000000";   --- NOP

wait for 100 ns;

instr <= X"000000";   --- NOP

wait for 100 ns;

instr <= X"000000";   --- NOP

wait for 100 ns;

instr <= X"000000";   --- NOP

wait for 100 ns;

instr <= X"450F00";   --- SW R0, R15

wait for 100 ns;

instr <= X"C1200F";   --- BEQZ R2, 0x0F

wait for 100 ns;

instr <= X"000000";   --- NOP

wait for 100 ns;

instr <= X"000000";   --- NOP

wait for 100 ns;

instr <= X"000000";   --- NOP

wait for 100 ns;

 instr <= X"000000";   --- NOP

wait for 100 ns;

instr <= X"C1000F";   --- BEQZ R0, 0x0F

wait for 100 ns;

instr <= X"000000";   --- NOP
```

```
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
 instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"C0000F";   --- BNEZ R0, 0x0F
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
 instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"C0200F";   --- BNEZ R2, 0x0F
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
 instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"48F000";   --- JR R15
```

```
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
 instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"68F000";   --- JALR R15
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
 instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"450F00";   --- SW R0, R15
wait for 100 ns;
instr <= X"28FF00";   --- TRAP FF00
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
```

```
 instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"F80000";   --- RFE
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
instr <= X"000000";   --- NOP
wait for 100 ns;
 instr <= X"000000";   --- NOP
wait for 100 ns;
DATA <= X"FFF1";
instr <= X"440100";   --- LW R0(0), R1
wait for 100 ns;
instr <= X"000000";  --- NOP
wait for 100 ns;
instr <= X"000000"; --- NOP
wait for 100 ns;
instr <= X"000000"; --- NOP
wait for 100 ns;
instr <= X"000000"; --- NOP
wait for 100 ns;
instr <= X"000000";  --- NOP
wait for 100 ns;
DATA <= "ZZZZZZZZZZZZZZZZ";
instr <= X"000000";  --- NOP
wait for 100 ns;
```

```
instr <= X"450100";  ---- SW R0(0), R1

wait for 100 ns;

instr <= X"000000";  --- NOP

wait for 100 ns;

instr <= X"000000"; --- NOP

wait for 100 ns;

instr <= X"000000";  --- NOP

wait for 100 ns;

instr <= X"000000"; --- NOP

wait for 100 ns;

instr <= X"000000"; --- NOP

wait for 100 ns;


end process;


-- Place stimulus and analysis statements here


dut :  DLX port map (
      Instr => Instr,
      Addr_int => addr_int,
      PC => PC,
      Data => data,
      Resetn => resetn,
      Prog_Rd => prog_rd,
      Rd => rd,
      Wr => wr,
      Stalln => stalln,
      Clock_in => clock_in
 );

end testbench;
```

9. increment.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_arith.all;

-- ***** dlx model *****
-- external ports

ENTITY dlx IS PORT (
      Addr_Int : OUT std_logic_vector(15 downto 0);
      Clock_in : IN std_logic;
      Data : INOUT std_logic_vector(15 downto 0);
      Instr : IN std_logic_vector(23 downto 0);
      PC : OUT std_logic_vector(15 downto 0);
      Prog_Rd : OUT std_logic;
      Rd : OUT std_logic;
      Resetn : IN std_logic;
      Stalln : IN std_logic;
      Wr : OUT std_logic
);

END dlx;
-- internal structure

ARCHITECTURE structural OF dlx IS

-- COMPONENTS

COMPONENT core
PORT (
      Addr_Int : OUT std_logic_vector(15 downto 0);
      Clock_in : IN std_logic;
      Input_Data : IN std_logic_vector(15 downto 0);
      Output_Data : Out std_logic_vector(15 downto 0);
      Instr : IN std_logic_vector(23 downto 0);
      PC : OUT std_logic_vector(15 downto 0);
      Prog_Rd : OUT std_logic;
      Rd : OUT std_logic;
      Resetn : IN std_logic;
      Stalln : IN std_logic;
      Wr : OUT std_logic
);
END COMPONENT;


COMPONENT IO_Pads
PORT (
      Pads : INOUT std_logic_vector (15 downto 0);
      In_Data : OUT std_logic_vector (15 downto 0);
      Out_Data : IN std_logic_vector (15 downto 0);
      Output_En_n : IN std_logic
);
END COMPONENT;


-- SIGNALS
```

```
signal Input_data : std_logic_vector(15 downto 0);
signal Output_data : std_logic_vector(15 downto 0);
signal wr_int : std_logic;

-- INSTANCES
BEGIN

wr <= wr_int;

core1 : core PORT MAP(
      Addr_Int => Addr_Int,
      Clock_in => Clock_In,
      Input_Data => Input_data,
      Output_Data => Output_data,
      Instr => Instr,
      PC => PC,
      Prog_Rd => Prog_Rd,
      Rd => Rd,
      Resetn => Resetn,
      Stalln => stalln,
      Wr => Wr_int
);

IO_Pads_1 : IO_Pads PORT MAP(
    Pads => Data,
    In_Data => Input_Data,
    Out_Data => Output_Data,
    Output_En_n => wr_int
);

END structural;
```

10. IO_Pads.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

---- *** IO_Pads Model ***
---- external ports

Entity IO_Pads is PORT (
      Pads : INOUT std_logic_vector (15 downto 0);
      In_Data : Out std_logic_vector (15 downto 0);
      Out_Data : In std_logic_vector (15 downto 0);
      Output_En_n : IN std_logic
);
END IO_Pads;

Architecture Behavior of IO_Pads is
Begin
  --In_Data <= Pads;
  Pads <= Out_Data when Output_En_n = '0' else (Pads'range =>
'Z');
  In_Data <= Pads;
end Behavior;
```

11. log_barrel.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** log_barrel model *****
-- external ports
ENTITY log_barrel IS PORT (
      ar_or_log : IN std_logic;
      In_word : IN std_logic_vector(15 downto 0);
      l_or_r : IN std_logic;
      Out_word : Out  std_logic_vector(15 downto 0);
      Shift: IN std_logic_vector(3 downto 0)
);
END log_barrel;

-- internal structure
ARCHITECTURE rtl OF log_barrel IS
signal sel1, sel2, sel3, sel4 : std_logic_vector ( 1 downto 0);
signal buf0b, buf0c, buf0d : std_logic_vector (15 downto 0);
signal buf1a, buf1b, buf1c, buf1d : std_logic_vector (15 downto
0);
signal buf2a, buf2b, buf2c, buf2d : std_logic_vector (15 downto
0);
signal buf3a, buf3b, buf3c, buf3d : std_logic_vector (15 downto
0);

component word_mux4
port (a : in std_logic_vector (15 downto 0);
         b : in std_logic_vector (15 downto 0);
         c : in std_logic_vector (15 downto 0);
         d : in std_logic_vector (15 downto 0);
        sel :  in std_logic_vector (1 downto 0);
        out_word : out std_logic_vector (15 downto 0)
);
end component;

begin
sel1(1) <= l_or_r and shift(0);
sel1(0) <= ar_or_log and shift(0);

sel2(1) <= l_or_r and shift(1);
sel2(0) <= ar_or_log and shift(1);

sel3(1) <= l_or_r and shift(2);
sel3(0) <= ar_or_log and shift(2);

sel4(1) <= l_or_r and shift(3);
sel4(0) <= ar_or_log and shift(3);


buf0b <= in_word(14 downto 0) & "0";
buf0c <= "0" & in_word(15 downto 1);
buf0d <= in_word(15) & in_word(15 downto 1);

buf1b <= buf1a(13 downto 0) & "00";
buf1c <= "00" & buf1a(15 downto 2);
```

```
    buf1d <= buf1a(15) & buf1a(15) & buf1a(15 downto 2);

    buf2b <= buf2a(11 downto 0) & "0000";
    buf2c <= "0000" & buf2a(15 downto 4);
    buf2d <= buf2a(15) & buf2a(15) & buf2a(15) & buf2a(15) & buf2a(15
downto 4);

    buf3b <= buf3a(7 downto 0) & "00000000";
    buf3c <= "00000000" & buf3a(15 downto 8);
    buf3d <= buf3a(15) & buf3a(15) & buf3a(15) & buf3a(15) &
buf3a(15) & buf3a(15) & buf3a(15) & buf3a(15) & buf3a(15 downto 8);


    mux1:  word_mux4
    port map (
        a => in_word,
        b => buf0b,
        c => buf0c,
        d => buf0d,
        sel => sel1,
        out_word => buf1a
        );
    mux2:  word_mux4
    port map (
        a => buf1a,
        b => buf1b,
        c => buf1c,
        d => buf1d,
        sel => sel2,
        out_word => buf2a
        );

    mux3:  word_mux4
    port map (
        a => buf2a,
        b => buf2b,
        c => buf2c,
        d => buf2d,
        sel => sel3,
        out_word => buf3a
        );

    mux4:  word_mux4
    port map (
        a => buf3a,
        b => buf3b,
        c => buf3c,
        d => buf3d,
        sel => sel4,
        out_word => out_word);

    end rtl;
```

12. pc_control.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** pc_control model *****
-- external ports
ENTITY pc_control IS PORT (
      ALU_Out : IN std_logic_vector(15 downto 0);
      Clock : IN std_logic;
      D2_Inc_PC : OUT std_logic_vector(15 downto 0);
      D_Link_PC : OUT std_logic_vector(15 downto 0);
      IAR_Enable : IN std_logic;
      In_PC : OUT std_logic_vector(15 downto 0);
      PC : OUT std_logic_vector(15 downto 0);
      PC_Sel : IN std_logic_vector(1 downto 0);
      Resetn : IN std_logic;
      Scan_Data_In : IN std_logic;
      Scan_Data_Out : OUT std_logic;
      Scan_Enable : IN std_logic;
      Stalln : IN std_logic
);
END pc_control;

-- internal structure
ARCHITECTURE structural OF pc_control IS

-- COMPONENTS
COMPONENT word_reg_single
PORT (
      Clock : IN std_logic;
      Data_In : IN std_logic_vector(15 downto 0);
      Data_out : OUT std_logic_vector(15 downto 0);
      Enable : IN std_logic;
      Resetn : IN std_logic;
      Scan_Data_In : IN std_logic;
      Scan_Enable : IN std_logic
);
END COMPONENT;

COMPONENT word_mux3
PORT (
      A : IN std_logic_vector(15 downto 0);
      B : IN std_logic_vector(15 downto 0);
      C : IN std_logic_vector(15 downto 0);
      Out_word : OUT std_logic_vector(15 downto 0);
      Sel : IN std_logic_vector(1 downto 0)
);
END COMPONENT;

COMPONENT increment
PORT (
      CI : IN std_logic;
      In_word : IN std_logic_vector(15 downto 0);
      Out_word : OUT std_logic_vector(15 downto 0)
);
END COMPONENT;
```

```vhdl
-- SIGNALS

SIGNAL IAR : std_logic_vector(15 downto 0);
SIGNAL PC_Incr : std_logic_vector(15 downto 0);
SIGNAL Buf_In_PC : std_logic_vector(15 downto 0);
SIGNAL Buf_PC : std_logic_vector(15 downto 0);
SIGNAL Buf_Scan_Data_Out : std_logic;
SIGNAL Buf_D1_Inc_PC  : std_logic_vector(15 downto 0);
SIGNAL Buf_D2_Inc_PC : std_logic_vector(15 downto 0);
SIGNAL Buf_D_Link_PC : std_logic_vector(15 downto 0);
SIGNAL Link_PC : std_logic_vector(15 downto 0);
SIGNAL Buf_Link_PC : std_logic_vector(15 downto 0);


-- INSTANCES
BEGIN
In_PC <= Buf_In_PC;
PC <= Buf_PC;
D2_Inc_PC <= Buf_D2_Inc_PC;
D_Link_PC <= Buf_D_Link_PC;
Scan_Data_Out <= IAR(15);

halfword_reg_single_1 : word_reg_single   PORT MAP(
      Clock => Clock,
      Data_In => Buf_In_PC,
      Data_out => Buf_PC,
      Enable => Stalln,
      Resetn => Resetn,
      Scan_Data_In => Scan_Data_In,
      Scan_Enable => Scan_Enable
);
halfword_mux3_1 : word_mux3   PORT MAP(
      A => PC_Incr,
      B => ALU_Out,
      C => IAR,
      Out_word => Buf_In_PC,
      Sel => PC_Sel
);
halfword_increment_1 : increment   PORT MAP(
      CI => '1',
      In_word => Buf_PC,
      Out_word => PC_Incr
);
halfword_reg_single_2 : word_reg_single   PORT MAP(
      Clock => Clock,
      Data_In => PC_Incr,
      Data_out => Buf_D1_Inc_PC,
      Enable => Stalln,
      Resetn => Resetn,
      Scan_Data_In => Buf_PC(15),
      Scan_Enable => Scan_Enable
);
halfword_reg_single_3 : word_reg_single   PORT MAP(
      Clock => Clock,
      Data_In => Buf_D1_Inc_PC,
      Data_out => Buf_D2_Inc_PC,
      Enable => Stalln,
```

```
      Resetn => Resetn,
      Scan_Data_In => Buf_D1_Inc_PC(15),
      Scan_Enable => Scan_Enable
);
halfword_increment_2 : increment   PORT MAP(
      CI => '1',
      In_word(0) => '1',
      In_word(15 downto 1) => Buf_D2_Inc_PC(15 downto 1),
      Out_word(15 downto 0) => Link_PC(15 downto 0)
);
halfword_reg_single_4 : word_reg_single   PORT MAP(
      Clock => Clock,
      Data_In(0) => Buf_D2_Inc_PC(0),
      Data_In(15 downto 1)  => Link_PC(15 downto 1),
      Data_out => Buf_Link_PC,
      Enable => Stalln,
      Resetn => Resetn,
      Scan_Data_In => Buf_D2_Inc_PC(15),
      Scan_Enable => Scan_Enable
);
halfword_reg_single_5 : word_reg_single   PORT MAP(
      Clock => Clock,
      Data_In => Buf_Link_PC,
      Data_Out => Buf_D_Link_PC,
      Enable => Stalln,
      Resetn => Resetn,
      Scan_Data_In => Buf_Link_PC(15),
      Scan_Enable => Scan_Enable
);
halfword_reg_single_6 : word_reg_single   PORT MAP(
      Clock => Clock,
      Data_In => Buf_D_Link_PC,
      Data_out => IAR,
      Enable => IAR_Enable,
      Resetn => Resetn,
      Scan_Data_In => Buf_D_Link_PC(15),
      Scan_Enable => Scan_Enable
);
END structural;
```

13. pipeline.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** pipeline model *****
-- external ports
ENTITY pipeline IS PORT (
      alu_op : OUT std_logic_vector(4 downto 0);
      A_Mux : OUT std_logic_vector(1 downto 0);
      B_Mux : OUT std_logic_vector(1 downto 0);
      Clock : IN std_logic;
      Data_In : IN std_logic_vector(23 downto 0);
      Dest : OUT std_logic_vector(3 downto 0);
      Immed : OUT std_logic_vector(15 downto 0);
      PC_Sel : OUT std_logic_vector(1 downto 0);
```

```
      rd_enable : OUT std_logic;
      Reg_In_Sel : OUT std_logic_vector(1 downto 0);
      Resetn : IN std_logic;
      RSone : OUT std_logic_vector(3 downto 0);
      RStwo : OUT std_logic_vector(3 downto 0);
      Scan_Data_In : IN std_logic;
      Scan_Enable : IN std_logic;
      Stalln : IN std_logic;
      wb_enable : OUT std_logic;
      scan_out : OUT std_logic;
      IAR_Enable : OUT std_logic;
      wr_enable : OUT std_logic;
      zero_flag : IN std_logic
);
END pipeline;

-- internal structure
ARCHITECTURE rtl OF pipeline IS

-- COMPONENTS

COMPONENT twelve_bit_reg_single
PORT (
      Clock : IN std_logic;
      Data_In : IN std_logic_vector(11 downto 0);
      Data_out : OUT std_logic_vector(11 downto 0);
      Enable : IN std_logic;
      Resetn : IN std_logic;
      Scan_Data_In : IN std_logic;
      Scan_Enable : IN std_logic
);
END COMPONENT;

COMPONENT twenty_four_bit_reg_single
PORT (
      Clock : IN std_logic;
      Data_In : IN std_logic_vector(23 downto 0);
      Data_out : OUT std_logic_vector(23 downto 0);
      Enable : IN std_logic;
      Resetn : IN std_logic;
      Scan_Data_In : IN std_logic;
      Scan_Enable : IN std_logic
);
END COMPONENT;


-- SIGNALS
SIGNAL Dec_Instr : std_logic_vector (23 downto 0);
SIGNAL Ex_Instr : std_logic_vector (23 downto 0);
SIGNAL Mem_Instr : std_logic_vector (11 downto 0);
SIGNAL WB_Instr :   std_logic_vector (11 downto 0);


-- INSTANCES
BEGIN

----- ****** decode pipeline stage *********
```

```
      twenty_bit_reg_single_1 : twenty_four_bit_reg_single   PORT MAP(
            Clock => Clock,
            Data_In => Data_In,
        Data_out => Dec_Instr,
            Enable => Stalln,
            Resetn => Resetn,
            Scan_Data_In => Scan_Data_In,
            Scan_Enable => Scan_Enable
      );

      process (Dec_Instr)
      begin
      RSone <= Dec_Instr(15 downto 12);

      ---- assign RS2 (check for SW instruction)
      if (Dec_Instr(23 downto 16) = X"45") then
        RStwo <= Dec_Instr(11 downto 8) ;
      else RStwo <=  Dec_Instr(7 downto 4);
      end if;
      end process;
      ------ ****** execute  pipeline stage **********

      twenty_four_bit_reg_single_2 : twenty_four_bit_reg_single   PORT
MAP(
            Clock => Clock,
            Data_In => Dec_Instr,
            Data_out => Ex_Instr,
            Enable => Stalln,
            Resetn => Resetn,
            Scan_Data_In => Dec_Instr(23),
            Scan_Enable => Scan_Enable
      );

      Immed <= Ex_Instr(15 downto 0);   ---- assign immediate value
      alu_op <= Ex_Instr(20 downto 16);  ---- assign alu opcodes
      b_mux <= Ex_Instr(22 downto 21);    --- assign b_mux


      PC_Sel <= "01" when Ex_Instr(23 downto 16) = X"C8" else  -----
when OP_J
                "01" when Ex_Instr(23 downto 16) = X"E8" else -----
when OP_JAL
                "0" & zero_flag when Ex_Instr(23 downto 16) = X"C1"
else ---when OP_BEQZ
                "0" & not(zero_flag) when Ex_Instr(23 downto 16) =
X"C0" else ---when OP_BEQZ
                "10" when Ex_Instr(23 downto 16) = X"F8" else ---OP_RFE
                "01" when Ex_Instr(23 downto 16) = X"28" else ----
OP_TRAP
                "01" when Ex_Instr(23 downto 16) = X"48" else ----
OP_JR
                "01" when Ex_Instr(23 downto 16) = X"68" else ----
OP_JALR
                "00";

      process (Ex_Instr)
```

```
begin

case Ex_Instr(23 downto 16) is
  when X"C8" =>           ----- when OP_J
     A_Mux  <= "11";
   when X"E8" =>          ----- when OP_JAL
     A_Mux  <= "11";
  when X"C1" =>           ----- when OP_BEQZ
     A_Mux  <= "01";
  when X"C0" =>           ----- when OP_BNEZ
     A_Mux  <= "01";
  when X"08" =>           ----- when OP_LHI
     A_Mux  <= "10";
  when X"F8" =>           ----- when OP_RFE
     A_Mux  <= "00";
  when X"28" =>           ----- when OP_TRAP
     A_Mux  <= "11";
  when X"48" =>           ----- when OP_JR
     A_Mux  <= "00";
  when X"68" =>           ----- when OP_JALR
     A_Mux  <= "00";
  when  others =>         ----- OTHERS
     A_Mux  <= "00";
end case;
end process;

------ ***** memory stage of pipeline ******* -----------

twelve_bit_reg_single_1 : twelve_bit_reg_single   PORT MAP(
      Clock => Clock,
      Data_In(11 downto 4) => Ex_Instr(23 downto 16),
      Data_In(3 downto 0) => Ex_Instr(11 downto 8),
      Data_out => Mem_Instr,
      Enable => Stalln,
      Resetn => Resetn,
      Scan_Data_In => Ex_Instr(23),
      Scan_Enable => Scan_Enable
);

process (Mem_Instr)
begin
case Mem_Instr(11 downto 4) is
  when X"45" =>
     rd_enable <= '0';     ----- OP_SW (write)
     wr_enable <= '1';
   when X"44" =>           ------ OP_LW (read)
     rd_enable <= '1';
     wr_enable <= '0';
   when others =>
     rd_enable <= '0';
     wr_enable <= '0';
end case;
end process;

------ ******** write back stage ********
twelve_bit_reg_single_2 : twelve_bit_reg_single   PORT MAP(
      Clock => Clock,
```

```
          Data_In => Mem_Instr,
          Data_out => WB_Instr,
          Enable => Stalln,
          Resetn => Resetn,
          Scan_Data_In => Mem_Instr(11),
          Scan_Enable => Scan_Enable
    );

    scan_out <= WB_Instr(11);
    process (WB_Instr)
    begin

    ---- check for Jump and Link Instructions to set Reg_In_Sel(0) =
0
      if (WB_Instr(11 downto 4) = X"E8" or WB_Instr(11 downto 4) =
X"68") then
         Reg_In_Sel(1) <= '1';
         Dest <= "1111";
      else Reg_In_Sel(1) <= '0';
         Dest <= WB_Instr(3 downto 0);
      end if;

    ---- check for TRAP to set IAR_Enable = 1
      if (WB_Instr(11 downto 4) = X"28") then
        IAR_Enable <= '1';
      else   IAR_Enable <= '0';
      end if;

    ---- check for LW to set Reg_In_Sel(1) = 1
      if (WB_Instr(11 downto 4) = X"44" ) then
         Reg_In_Sel(0) <= '1';
      else Reg_In_Sel(0) <= '0';
      end if;

    ------ set write_back enable
      case WB_Instr(11 downto 4) is
         when  X"C8" =>           ----- when OP_J
         WB_Enable <= '0';
      when X"C1" =>           ----- when OP_BEQZ
         WB_Enable <= '0';
      when X"C0" =>           ----- when OP_BNEZ
         WB_Enable <= '0';
      when X"45" =>           ----- when OP_SW
         WB_Enable <= '0';
      when X"F8" =>           ----- when OP_RFE
         WB_Enable <= '0';
      when X"28" =>           ----- when OP_TRAP
         WB_Enable <= '0';
      when X"48" =>           ----- when OP_JR
         WB_Enable <= '0';
      when X"00" =>           ----- when OP_NOP
         WB_Enable <= '0';
      when others =>
         WB_Enable <= '1';
      end case;
    end process;
    END rtl;
```

14. regfile.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-----******* regfile model ***********
---- external ports
ENTITY regfile IS PORT (
  A : OUT std_logic_vector(15 downto 0);
  B : OUT std_logic_vector(15 downto 0);
  clock : IN std_logic;
  Data_In : IN std_logic_vector(15 downto 0);
  Dest : IN std_logic_vector(3 downto 0);
  stalln : IN std_logic;
  RSone : IN std_logic_vector(3 downto 0);
  RStwo : IN std_logic_vector(3 downto 0);
  scan_data_in : IN std_logic;
  scan_enable : IN std_logic;
  Resetn : IN std_logic;
  wb_enable : IN std_logic
);
END regfile;

---- internal structure
ARCHITECTURE structural OF regfile is

---- COMPONENTS
COMPONENT Dest_Decoder
PORT (
      Dest : IN std_logic_vector(3 downto 0);
      Enable : OUT std_logic_vector(15 downto 1);
      wb_enable : IN std_logic
);
END COMPONENT;

COMPONENT word_reg_single
PORT (
      Clock : IN std_logic;
      Data_In : IN std_logic_vector (15 downto 0);
      Data_out : OUT std_logic_vector (15 downto 0);
      enable : IN std_logic;
      Resetn : IN std_logic;
      Scan_Data_In : IN std_logic;
      Scan_Enable : IN std_logic
);
END COMPONENT;

COMPONENT word_mux16
PORT (
      In_Word0 : IN std_logic_vector(15 downto 0);
      In_Word1 : IN std_logic_vector(15 downto 0);
      In_Word2 : IN std_logic_vector(15 downto 0);
      In_Word3 : IN std_logic_vector(15 downto 0);
      In_Word4 : IN std_logic_vector(15 downto 0);
      In_Word5 : IN std_logic_vector(15 downto 0);
      In_Word6 : IN std_logic_vector(15 downto 0);
      In_Word7 : IN std_logic_vector(15 downto 0);
```

```
      In_Word8 : IN std_logic_vector(15 downto 0);
      In_Word9 : IN std_logic_vector(15 downto 0);
      In_Word10 : IN std_logic_vector(15 downto 0);
      In_Word11 : IN std_logic_vector(15 downto 0);
      In_Word12 : IN std_logic_vector(15 downto 0);
      In_Word13 : IN std_logic_vector(15 downto 0);
      In_Word14 : IN std_logic_vector(15 downto 0);
      In_Word15 : IN std_logic_vector(15 downto 0);
      Out_word : Out std_logic_vector(15 downto 0);
      Sel : IN std_logic_vector(3 downto 0)
);
END component;

----- signals
signal Enable : std_logic_vector(15 downto 1);
signal Reg1_Data : std_logic_vector(15 downto 0);
signal Reg2_Data : std_logic_vector(15 downto 0);
signal Reg3_Data : std_logic_vector(15 downto 0);
signal Reg4_Data : std_logic_vector(15 downto 0);
signal Reg5_Data : std_logic_vector(15 downto 0);
signal Reg6_Data : std_logic_vector(15 downto 0);
signal Reg7_Data : std_logic_vector(15 downto 0);
signal Reg8_Data : std_logic_vector(15 downto 0);
signal Reg9_Data : std_logic_vector(15 downto 0);
signal Reg10_Data : std_logic_vector(15 downto 0);
signal Reg11_Data : std_logic_vector(15 downto 0);
signal Reg12_Data : std_logic_vector(15 downto 0);
signal Reg13_Data : std_logic_vector(15 downto 0);
signal Reg14_Data : std_logic_vector(15 downto 0);
signal Reg15_Data : std_logic_vector(15 downto 0);
signal RegA_Data : std_logic_vector(15 downto 0);
signal MuxA_Data : std_logic_vector(15 downto 0);
signal MuxB_Data : std_logic_vector(15 downto 0);
signal zero_word :  std_logic_vector(15 downto 0);


begin

zero_word <= "0000000000000000";

---- port maps

Dest_Decoder1 :  Dest_Decoder PORT MAP (
      Dest=> Dest,
      Enable => Enable,
      wb_enable => wb_enable
);
word_reg1 : word_reg_single PORT MAP (
      Clock => clock,
      Data_In => Data_In,
      Data_out => Reg1_Data,
      Enable => Enable(1),
      Resetn => Resetn,
      Scan_Data_In => Scan_Data_In,
      Scan_Enable => Scan_Enable
);
```

```
word_reg2 : word_reg_single PORT MAP (
      Clock => clock,
      Data_In => Data_In,
      Data_out => Reg2_Data,
      Enable => Enable(2),
      Resetn => Resetn,
      Scan_Data_In => Reg1_Data(15),
      Scan_Enable => Scan_Enable
);
word_reg3 : word_reg_single PORT MAP (
      Clock => clock,
      Data_In => Data_In,
      Data_out => Reg3_Data,
      Enable => Enable(3),
      Resetn => Resetn,
      Scan_Data_In => Reg2_Data(15),
      Scan_Enable => Scan_Enable
);
word_reg4 : word_reg_single PORT MAP (
      Clock => clock,
      Data_In => Data_In,
      Data_out => Reg4_Data,
      Enable => Enable(4),
      Resetn => Resetn,
      Scan_Data_In => Reg3_Data(15),
      Scan_Enable => Scan_Enable
);
word_reg5 : word_reg_single PORT MAP (
      Clock => clock,
      Data_In => Data_In,
      Data_out => Reg5_Data,
      Enable => Enable(5),
      Resetn => Resetn,
      Scan_Data_In => Reg4_Data(15),
      Scan_Enable => Scan_Enable
);
word_reg6 : word_reg_single PORT MAP (
      Clock => clock,
      Data_In => Data_In,
      Data_out => Reg6_Data,
      Enable => Enable(6),
      Resetn => Resetn,
      Scan_Data_In => Reg5_Data(15),
      Scan_Enable => Scan_Enable
);
word_reg7 : word_reg_single PORT MAP (
      Clock => clock,
      Data_In => Data_In,
      Data_out => Reg7_Data,
      Enable => Enable(7),
      Resetn => Resetn,
      Scan_Data_In => Reg6_Data(15),
      Scan_Enable => Scan_Enable
);
word_reg8 : word_reg_single PORT MAP (
      Clock => clock,
      Data_In => Data_In,
```

```
        Data_out => Reg8_Data,
        Enable => Enable(8),
        Resetn => Resetn,
        Scan_Data_In => Reg7_Data(15),
        Scan_Enable => Scan_Enable
  );
  word_reg9 : word_reg_single PORT MAP (
        Clock => clock,
        Data_In => Data_In,
        Data_out => Reg9_Data,
        Enable => Enable(9),
        Resetn => Resetn,
        Scan_Data_In => Reg8_Data(15),
        Scan_Enable => Scan_Enable
  );
  word_reg10 : word_reg_single PORT MAP (
        Clock => clock,
        Data_In => Data_In,
        Data_out => Reg10_Data,
        Enable => Enable(10),
        Resetn => Resetn,
        Scan_Data_In => Reg9_Data(15),
        Scan_Enable => Scan_Enable
  );
  word_reg11 : word_reg_single PORT MAP (
        Clock => clock,
        Data_In => Data_In,
        Data_out => Reg11_Data,
        Enable => Enable(11),
        Resetn => Resetn,
        Scan_Data_In => Reg10_Data(15),
        Scan_Enable => Scan_Enable
  );
  word_reg12 : word_reg_single PORT MAP (
        Clock => clock,
        Data_In => Data_In,
        Data_out => Reg12_Data,
        Enable => Enable(12),
        Resetn => Resetn,
        Scan_Data_In => Reg11_Data(15),
        Scan_Enable => Scan_Enable
  );
  word_reg13 : word_reg_single PORT MAP (
        Clock => clock,
        Data_In => Data_In,
        Data_out => Reg13_Data,
        Enable => Enable(13),
        Resetn => Resetn,
        Scan_Data_In => Reg12_Data(15),
        Scan_Enable => Scan_Enable
  );
  word_reg14 : word_reg_single PORT MAP (
        Clock => clock,
        Data_In => Data_In,
        Data_out => Reg14_Data,
        Enable => Enable(14),
        Resetn => Resetn,
```

```
      Scan_Data_In => Reg13_Data(15),
      Scan_Enable => Scan_Enable
);
word_reg15 : word_reg_single PORT MAP (
      Clock => clock,
      Data_In => Data_In,
      Data_out => Reg15_Data,
      Enable => Enable(15),
      Resetn => Resetn,
      Scan_Data_In => Reg14_Data(15),
      Scan_Enable => Scan_Enable
);
word_regA : word_reg_single PORT MAP (
      Clock => clock,
      Data_In => MuxA_Data,
      Data_out => RegA_Data,
      Enable => stalln,
      Resetn => Resetn,
      Scan_Data_In => Reg15_Data(15),
      Scan_Enable => Scan_Enable
);

A <= RegA_Data;

word_regB : word_reg_single PORT MAP (
      Clock => clock,
      Data_In => MuxB_Data,
      Data_out => B,
      Enable =>  stalln,
      Resetn => Resetn,
      Scan_Data_In => RegA_Data(15),
      Scan_Enable => Scan_Enable
);
MuxA : word_mux16 PORT MAP (
  In_Word0    => zero_word,
      In_Word1    =>  Reg1_Data,
      In_Word2    =>  Reg2_Data,
      In_Word3    =>  Reg3_Data,
      In_Word4    =>  Reg4_Data,
      In_Word5    =>  Reg5_Data,
      In_Word6    =>  Reg6_Data,
      In_Word7    =>  Reg7_Data,
      In_Word8    =>  Reg8_Data,
      In_Word9    =>  Reg9_Data,
      In_Word10  =>  Reg10_Data,
      In_Word11  =>  Reg11_Data,
      In_Word12  =>  Reg12_Data,
      In_Word13  =>  Reg13_Data,
      In_Word14  =>  Reg14_Data,
      In_Word15  =>  Reg15_Data,
      Out_word    =>  MuxA_Data,
      Sel => RSone
);
MuxB : word_mux16 PORT MAP (
  In_Word0    =>  zero_word,
      In_Word1    =>  Reg1_Data,
      In_Word2    =>  Reg2_Data,
```

```
      In_Word3    =>  Reg3_Data,
      In_Word4    =>  Reg4_Data,
      In_Word5    =>  Reg5_Data,
      In_Word6    =>  Reg6_Data,
      In_Word7    =>  Reg7_Data,
      In_Word8    =>  Reg8_Data,
      In_Word9    =>  Reg9_Data,
      In_Word10  =>  Reg10_Data,
      In_Word11  =>  Reg11_Data,
      In_Word12  =>  Reg12_Data,
      In_Word13  =>  Reg13_Data,
      In_Word14  =>  Reg14_Data,
      In_Word15  =>  Reg15_Data,
      Out_word    =>  MuxB_Data,
      Sel => RStwo
);

END structural;
```

15. rw_control.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** rw_control model *****
-- external ports
ENTITY rw_control IS PORT (
      Clock : IN std_logic;
      Prog_Rd : OUT std_logic;
      Rd : OUT std_logic;
      rd_enable : IN std_logic;
      resetn : IN std_logic;
      stalln : IN std_logic;
      Wr : OUT std_logic;
      wr_enable : IN std_logic
);
END rw_control;

-- internal structure
ARCHITECTURE rtl OF rw_control IS

-- SIGNALS
SIGNAL clockn : std_logic; --- inverted clock

BEGIN
clockn <= not(Clock);
Wr <=  not (clockn and wr_enable);
Rd <=  not (clockn and rd_enable);
Prog_Rd <= not (clockn and resetn and stalln);
end rtl;
```

16. scan_reg.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** scan_reg model *****
-- external ports

ENTITY scan_reg IS PORT (
      clk : IN std_logic;
      data_in : IN std_logic;
      data_out : OUT std_logic;
      enable : IN std_logic;
      resetn : IN std_logic;
      scan_data_in : IN std_logic;
      scan_enable : IN std_logic
);
END scan_reg;

-- internal structure

ARCHITECTURE rtl OF scan_reg IS

-- INSTANCES
BEGIN

process (clk, resetn)
begin
if (resetn = '0') then
  data_out <= '0';
elsif (clk = '1' and clk'event) then
  if (scan_enable = '1') then
    data_out <= scan_data_in;
  elsif (enable = '1') then
     data_out <= data_in;
  end if;
 end if;
end process;

END rtl;
```

17. twelve_bit_reg_single.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** twelve_bit_reg_single model *****
-- external ports
ENTITY twelve_bit_reg_single IS PORT (
      Clock : IN std_logic;
      Data_In : IN std_logic_vector(11 downto 0);
      Data_out : OUT std_logic_vector(11 downto 0);
      Enable : IN std_logic;
      Resetn : IN std_logic;
      Scan_Data_In : IN std_logic;
      Scan_Enable : IN std_logic
```

```
);
END twelve_bit_reg_single;

-- internal structure
ARCHITECTURE structural OF twelve_bit_reg_single IS

-- COMPONENTS
COMPONENT scan_reg
PORT (
      clk : IN std_logic;
      data_in : IN std_logic;
      data_out : OUT std_logic;
      enable : IN std_logic;
      resetn : IN std_logic;
      scan_data_in : IN std_logic;
      scan_enable : IN std_logic
);
END COMPONENT;

-- SIGNALS
signal buf_data_out : std_logic_vector (10 downto 0);


-- INSTANCES
BEGIN
Data_out(0) <= buf_data_out(0);
Data_out(1) <= buf_data_out(1);
Data_out(2) <= buf_data_out(2);
Data_out(3) <= buf_data_out(3);
Data_out(4) <= buf_data_out(4);
Data_out(5) <= buf_data_out(5);
Data_out(6) <= buf_data_out(6);
Data_out(7) <= buf_data_out(7);
Data_out(8) <= buf_data_out(8);
Data_out(9) <= buf_data_out(9);
Data_out(10) <= buf_data_out(10);

scan_reg_1 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(1),
      data_out => buf_data_out(1),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => buf_data_out(0),
      scan_enable => Scan_Enable
);
scan_reg_2 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(2),
      data_out => buf_data_out(2),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => buf_data_out(1),
      scan_enable => Scan_Enable
);
scan_reg_3 : scan_reg   PORT MAP(
      clk => Clock,
```

```
      data_in => Data_In(3),
      data_out => buf_data_out(3),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => buf_data_out(2),
      scan_enable => Scan_Enable
);
scan_reg_4 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(4),
      data_out => buf_data_out(4),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => buf_data_out(3),
      scan_enable => Scan_Enable
);
scan_reg_5 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(0),
      data_out => buf_data_out(0),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Scan_Data_In,
      scan_enable => Scan_Enable
);
scan_reg_6 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(5),
      data_out => buf_data_out(5),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => buf_data_out(4),
      scan_enable => Scan_Enable
);
scan_reg_7 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(6),
      data_out => buf_data_out(6),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => buf_data_out(5),
      scan_enable => Scan_Enable
);
scan_reg_8 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(7),
      data_out => buf_data_out(7),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => buf_data_out(6),
      scan_enable => Scan_Enable
);
scan_reg_9 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(8),
      data_out => buf_data_out(8),
      enable => Enable,
```

```
      resetn => Resetn,
      scan_data_in => buf_data_out(7),
      scan_enable => Scan_Enable
);
scan_reg_10 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(9),
      data_out => buf_data_out(9),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => buf_data_out(8),
      scan_enable => Scan_Enable
);
scan_reg_11 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(10),
      data_out => buf_data_out(10),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => buf_data_out(9),
      scan_enable => Scan_Enable
);

scan_reg_12 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(11),
      data_out => Data_out(11),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => buf_data_out(10),
      scan_enable => Scan_Enable
);
END structural;
```

18. twenty_four_bit_reg_single.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** twenty_four_bit_reg_single model *****
-- external ports
ENTITY twenty_four_bit_reg_single IS PORT (
      Clock : IN std_logic;
      Data_In : IN std_logic_vector (23 downto 0);
  Data_out : OUT std_logic_vector (23 downto 0);
      Enable : IN std_logic;
      Resetn : IN std_logic;
      Scan_Data_In : IN std_logic;
      Scan_Enable : IN std_logic
);
END twenty_four_bit_reg_single;

-- internal structure
ARCHITECTURE structural OF twenty_four_bit_reg_single IS
```

```
-- COMPONENTS

Component twelve_bit_reg_single
 PORT (
      Clock : IN std_logic;
      Data_In : IN std_logic_vector(11 downto 0);
      Data_out : OUT std_logic_vector(11 downto 0);
      Enable : IN std_logic;
      Resetn : IN std_logic;
      Scan_Data_In : IN std_logic;
      Scan_Enable : IN std_logic
);
END Component;



-- SIGNALS
SIGNAL Buf_Data_out11 : std_logic;


-- INSTANCES
BEGIN
Data_out(11) <= Buf_Data_out11;

twelve_bit_reg_single1 : twelve_bit_reg_single PORT MAP(
  Clock => Clock,
  Data_In => Data_In(11 downto 0),
  Data_Out(10 downto 0) => Data_Out(10 downto 0),
  Data_Out(11) => Buf_Data_out11,
  Enable => Enable,
  Resetn => Resetn,
  Scan_Data_In => Scan_Data_In,
  Scan_Enable => Scan_Enable
);
twelve_bit_reg_single2 : twelve_bit_reg_single PORT MAP(
  Clock => Clock,
  Data_In => Data_In(23 downto 12),
  Data_Out => Data_Out(23 downto 12),
  Enable => Enable,
  Resetn => Resetn,
  Scan_Data_In => Buf_Data_out11,
  Scan_Enable => Scan_Enable
);
END structural;
```

19. word_mux16.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;


-- ***** word_mux16 model *****
-- external ports

ENTITY word_mux16 IS PORT (
      In_Word0 : IN std_logic_vector(15 downto 0);
```

```
      In_Word1 : IN std_logic_vector(15 downto 0);
      In_Word2 : IN std_logic_vector(15 downto 0);
      In_Word3 : IN std_logic_vector(15 downto 0);
      In_Word4 : IN std_logic_vector(15 downto 0);
      In_Word5 : IN std_logic_vector(15 downto 0);
      In_Word6 : IN std_logic_vector(15 downto 0);
      In_Word7 : IN std_logic_vector(15 downto 0);
      In_Word8 : IN std_logic_vector(15 downto 0);
      In_Word9 : IN std_logic_vector(15 downto 0);
      In_Word10 : IN std_logic_vector(15 downto 0);
      In_Word11 : IN std_logic_vector(15 downto 0);
      In_Word12 : IN std_logic_vector(15 downto 0);
      In_Word13 : IN std_logic_vector(15 downto 0);
      In_Word14 : IN std_logic_vector(15 downto 0);
      In_Word15 : IN std_logic_vector(15 downto 0);
      Out_word : Out std_logic_vector(15 downto 0);
      Sel : IN std_logic_vector(3 downto 0)
);
END word_mux16;

-- internal structure
ARCHITECTURE rtl OF word_mux16 IS


BEGIN
with sel select
   Out_word <= In_Word0 when  "0000",
               In_Word1 when  "0001",
                       In_Word2 when  "0010",
                       In_Word3 when  "0011",
                       In_Word4 when  "0100",
                       In_Word5 when  "0101",
                       In_Word6 when  "0110",
                       In_Word7 when  "0111",
                       In_Word8 when  "1000",
                       In_Word9 when  "1001",
                       In_Word10 when "1010",
                       In_Word11 when "1011",
                       In_Word12 when "1100",
                       In_Word13 when "1101",
                       In_Word14 when "1110",
                       In_Word15 when others;

END rtl;
```

20. word_mux3.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** word_mux3 model *****
-- external ports
ENTITY word_mux3 IS PORT (
      A : IN std_logic_vector(15 downto 0);
```

```
      B : IN std_logic_vector(15 downto 0);
      C : IN std_logic_vector(15 downto 0);
      Out_word : Out std_logic_vector(15 downto 0);
      Sel : IN std_logic_vector(1 downto 0)
);
END word_mux3;

-- internal structure
ARCHITECTURE rtl OF word_mux3 IS
BEGIN
process (A, B, C, Sel)
begin
case sel is
  when "00" => Out_word <= A;
  when "01" => Out_word <= B;
  when others => Out_word <= C;
end case;
end process;
END rtl;
```

21. word_mux4.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** word_mux4 model *****
-- external ports
ENTITY word_mux4 IS PORT (
      A : IN std_logic_vector(15 downto 0);
      B : IN std_logic_vector(15 downto 0);
      C : IN std_logic_vector(15 downto 0);
      D : IN std_logic_vector(15 downto 0);
      Out_word : Out std_logic_vector(15 downto 0);
      Sel : IN std_logic_vector(1 downto 0)
);
END word_mux4;

-- internal structure
ARCHITECTURE rtl OF word_mux4 IS
BEGIN
process (A, B, C, D, Sel)
begin
case sel is
  when "00" => Out_word <= A;
  when "01" => Out_word <= B;
  when "10" => Out_word <= C;
  when others => Out_word <= D;
end case;
end process;
END rtl;
```

22. word_reg_single.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** word_reg_single model *****
-- external ports

ENTITY word_reg_single IS PORT (
      Clock : IN std_logic;
      Data_In : IN std_logic_vector (15 downto 0);
      Data_out : OUT std_logic_vector (15 downto 0);
      Enable : IN std_logic;
      Resetn : IN std_logic;
      Scan_Data_In : IN std_logic;
      Scan_Enable : IN std_logic
);
END word_reg_single;

-- internal structure
ARCHITECTURE structural OF word_reg_single IS



-- COMPONENTS

COMPONENT scan_reg
PORT (
      clk : IN std_logic;
      data_in : IN std_logic;
      data_out : OUT std_logic;
      enable : IN std_logic;
      resetn : IN std_logic;
      scan_data_in : IN std_logic;
      scan_enable : IN std_logic
);
END COMPONENT;

-- SIGNALS
SIGNAL Buf_Data_out : std_logic_vector(14 downto 0);


-- INSTANCES
BEGIN
Data_out(0) <= Buf_Data_out(0);
Data_out(1) <= Buf_Data_out(1);
Data_out(2) <= Buf_Data_out(2);
Data_out(3) <= Buf_Data_out(3);
Data_out(4) <= Buf_Data_out(4);
Data_out(5) <= Buf_Data_out(5);
Data_out(6) <= Buf_Data_out(6);
Data_out(7) <= Buf_Data_out(7);
Data_out(8) <= Buf_Data_out(8);
Data_out(9) <= Buf_Data_out(9);
Data_out(10) <= Buf_Data_out(10);
Data_out(11) <= Buf_Data_out(11);
```

```
Data_out(12) <= Buf_Data_out(12);
Data_out(13) <= Buf_Data_out(13);
Data_out(14) <= Buf_Data_out(14);

scan_reg_1 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(1),
      data_out => Buf_Data_out(1),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(0),
      scan_enable => Scan_Enable
);
scan_reg_2 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(2),
      data_out => Buf_Data_out(2),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(1),
      scan_enable => Scan_Enable
);
scan_reg_3 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(3),
      data_out => Buf_Data_out(3),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(2),
      scan_enable => Scan_Enable
);
scan_reg_4 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(4),
      data_out => Buf_Data_out(4),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(3),
      scan_enable => Scan_Enable
);
scan_reg_6 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(5),
      data_out => Buf_Data_out(5),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(4),
      scan_enable => Scan_Enable
);
scan_reg_7 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(6),
      data_out => Buf_Data_out(6),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(5),
      scan_enable => Scan_Enable
```

```
);
scan_reg_8 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(7),
      data_out => Buf_Data_out(7),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(6),
      scan_enable => Scan_Enable
);
scan_reg_9 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(8),
      data_out => Buf_Data_out(8),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(7),
      scan_enable => Scan_Enable
);
scan_reg_10 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(9),
      data_out => Buf_Data_out(9),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(8),
      scan_enable => Scan_Enable
);
scan_reg_11 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(10),
      data_out => Buf_Data_out(10),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(9),
      scan_enable => Scan_Enable
);
scan_reg_12 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(11),
      data_out => Buf_Data_out(11),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(10),
      scan_enable => Scan_Enable
);
scan_reg_13 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(12),
      data_out => Buf_Data_out(12),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(11),
      scan_enable => Scan_Enable
);
scan_reg_14 : scan_reg   PORT MAP(
      clk => Clock,
```

```
      data_in => Data_In(13),
      data_out => Buf_Data_out(13),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(12),
      scan_enable => Scan_Enable
);
scan_reg_15 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(14),
      data_out => Buf_Data_out(14),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(13),
      scan_enable => Scan_Enable
);
scan_reg_16 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(15),
      data_out => Data_out(15),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Buf_Data_out(14),
      scan_enable => Scan_Enable
);
scan_reg_5 : scan_reg   PORT MAP(
      clk => Clock,
      data_in => Data_In(0),
      data_out => Buf_Data_out(0),
      enable => Enable,
      resetn => Resetn,
      scan_data_in => Scan_Data_In,
      scan_enable => Scan_Enable
);
END structural;
```

23. word_set.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** word_set model *****
-- external ports
ENTITY word_set IS PORT (
      In_word : IN std_logic_vector (15 downto 0);
      set_op : IN std_logic_vector (2 downto 0);
      set_out : OUT std_logic
);
END word_set;

-- internal structure
ARCHITECTURE rtl OF word_set IS

component zero_test
PORT (
      In_word : in std_logic_vector(15 downto 0);
```

```
      zero_flag : OUT std_logic
);
END component;
signal zero_flag : std_logic;

begin
process (In_word, set_op, zero_flag)
begin
case set_op is
  when "000" => set_out <= zero_flag;
  when "001" => set_out <= (not(In_word(15)) or zero_flag);
  when "010" => set_out <= not(In_word(15)) and not(zero_flag);
  when "011" => set_out <= (In_word(15) or zero_flag);
  when "100" => set_out <= In_word(15);
  when others => set_out <= not(zero_flag);
end case;
end process;
zero_test1 : zero_test port map (
           In_word => In_word,
           zero_flag => zero_flag
);


END rtl;
```

24. zero_test.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- ***** zero_test model *****
-- external ports
ENTITY zero_test IS PORT (
      In_word : in std_logic_vector(15 downto 0);
      zero_flag : OUT std_logic
);
END zero_test;

-- internal structure
ARCHITECTURE rtl OF zero_test IS
begin

process (In_word)
begin
  if (In_word = "0000000000000000") then
    zero_flag <= '1';
  else zero_flag <= '0';
  end if;
end process;

END rtl;
```

APPENDIX E: GLOSSARY

BGA	Ball Grid Array
CFTP	Configurable Fault-Tolerant Processor
COTS	Commercial Off the Shelf
Coregen	CORE generator
CPLD	Complex Programmable Logic Device
ESSD	Error Syndrome Storage Device
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IAR	Interrupt Address Register
ISR	Interrupt Service Routine
LEO	Low-Earth Orbit
Mem	Memory
NPS	Naval Postgraduate School
Opcode	Operation code
RADHARD	Radiation Hardened
RAM	Ramdom-Access Memory
RFE	Return From Exception
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
SEB	Single Event Burnout
SEE	Single Event Effects
SEL	Single Event Latchup

SEP	Single Event Phenomenon
SERB	Space Experiment Review Board
SEU	Single Event Upset
SOC	System On a Chip
SPLD	Sequential (or Simple) Programmable Logic Device
STP	Space Test Program
TMR	Triple Modular Redundancy
VHSIC	Very High Speed Integrated Circuit
VDHL	VHSIC Hardware Description Language
WB	Write Back

LIST OF REFERENCES

1. Lashomb, Peter A., "Triple Modular Redundant (TMR) Microprocessor System for Field Programmable gate Array (FPGA) Implementation," Master's Thesis, Naval Postgraduate School, Monterey, California, March 2002.

2. "Single Event Latchup" http://www.aero.org/seet/primer/singleeventlatchup.html, October 2003.

3. Richard S. Wheatley, "Microcontrollers vs. Soft–Core Processors," http://www.agslab.cocse.unf.edu/projects/class/summer2001/eel4905/

4. Per Holmberg, "Searching the Ideal Core for FPGAs," http://www.xilinx.com/pci/searching.pdf, March 2003.

5. Johnson, Steven A., "Implementation of a Configurable Fault Tolerant Processor (CFTP)," Master's Thesis, Naval Postgraduate School, Monterey, California, March 2003.

6. "Virtex^{TM} 2.5V Field Programmable Gate Arrays," Xilinx Data Sheet DS003-1, San Jose, California, October 2003.

7. Hennessy, John L. and Patterson, David A., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, California, 1990.

8. Clark, Kenneth A., "The Effect of Signal Event Transients on Complex Digital Systems," Doctoral Dissertation, Naval Postgraduate School, Monterey, California, June 2002.

9. Ebert, Dean A., "Design and Development of a Configurable Fault Tolerant Processor (CFTP) for Space Applications," Master's Thesis, Naval Postgraduate School, Monterey, California, June 2003.

10. Xilinx answer database http://www.xilinx.com/support/searchtd.htm, October 2003.

11. ISE 5 In–Depth Tutorial ftp://ftp.xilinx.com/pub/documentation/ise5_tutorials/ise5tut.pdf, September 2003.

12. "Single Event Effects Testing of the Intel Pentium III (P3) Microprocessor" http://klabs.org/DEI/Processor/386_486/Radiation/pentium/howard_see_symp02.pdf, November 2003.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, ECE Department, John P. Powers
   Naval Postgraduate School
   Monterey, California

4. Professor Herschel H. Loomis
   Naval Postgraduate School
   Monterey, California

5. Professor Alan A. Ross
   Naval Postgraduate School
   Monterey, California

6. Doctor Kenneth A. Clark
   Naval Research Laboratory
   Washington, DC

7. LCDR Joe Reason, USN
   National Reconnaissance Office
   Chantilly, Virginia

8. CPT Brian Bailey, USAF
   National Reconnaissance Office
   Chantilly, Virginia

9. 1st Lt Rong Yuan, TWAF
   Air Force 499 Wing
   Hsinchu, Republic of China (Taiwan)