

USING INSTRUCTION CODE OBFUSCATION TO DEFEAT MALWARE ATTACKS

by

Tristan Tanner Running Crane

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

May 2023

©COPYRIGHT

by

Tristan Tanner Running Crane

2023

All Rights Reserved

ACKNOWLEDGEMENTS

I would like to acknowledge my advisor Dr. Brock LaMeres and Dr. Clemente Izurieta for allowing me work in their research lab. Friends and lab colleagues including Colter Barney, Hezekiah Austin, Chris Major and Justin Williams who've helped me during the process of my project and the class work over the course of grad school.

TABLE OF CONTENTS

1. INTRODUCTION	1
2. BACKGROUND AND MOTIVATION.....	3
Opcodes and Instruction Sets	3
RISC-V Instruction Set Architecture	4
Embedded System Security	6
Buffer Overflow Attacks	6
Instruction Code Obfuscation	7
3. RELATED WORK	9
MSU Contributions.....	9
4. SYSTEM DESIGN.....	11
Design Flow	11
Processor	13
Control Unit	13
R-Type Instructions:	14
I-Type Instructions:	15
S-Type Instructions:.....	16
B-Type Instructions:	17
U-Type Instructions:	19
J-Type Instructions:.....	20
Data Path.....	21
Program Counter:	22
Registers:	23
Immediate Generator:	23
Branch Comparator:.....	25
Arithmetic Logic Unit:	26
Memory	26
Instruction Memory:	27
DMEM:.....	27
Memory Map:.....	28
Peripherals	28
General Purpose Input Output:	28
Serial Peripheral Interface:.....	30
Universal Asynchronous Receiver-Transmitter:.....	30
Detection and Recovery System.....	31

TABLE OF CONTENTS – CONTINUED

5. SYSTEM VERIFICATION	35
Processor Verification.....	35
Peripheral Verification	35
Input Output Verification	36
Serial Peripheral Interface Verification	37
Universal Asynchronous Receiver-Transmitter Verification	37
Recovery Verification	40
Full System Verification	40
System application	40
Attack Vector and Method.....	41
Full System Results.....	43
6. FUTURE PROGRESS	48
Processor Expansion	48
Pipelining	48
Instruction Set Expansion	48
Full RV32I Implementation.....	49
RISC-V ISA Expansions	49
Privileged Mode Support	49
Control Status Registers	50
Operating System.....	50
Peripheral Additions	50
Interrupt Support.....	50
Timer Support	51
Inter-Integrated Circuit Support.....	51
Toolchain Improvements	51
JTAG Interface	51
Bootloader	52
Compilers	52
Malware Prevention Methods	52
Stack Protections	52
Non-Executable Stack	53
Address Space Layout Randomization	53
Stack guard	53
Diverse Obfuscation.....	54
Recovery Methods	54
Data Memory Refreshing	54
Partial Reconfiguration.....	54

TABLE OF CONTENTS – CONTINUED

REFERENCES CITED.....	57
-----------------------	----

LIST OF FIGURES

Figure	Page
2.1 RV32I Base Instruction Set [4]	5
2.2 Vulnerabilities Classified [7]	7
4.1 Design Flow for System	12
4.2 Processor Simulink Model.....	14
4.3 Control Unit Stateflow Model	15
4.4 R-Type Format [4]	15
4.5 ADD operation of R-Type State.....	16
4.6 I-Type Formats [4]	17
4.7 I-type States	18
4.8 S-Type Format [4]	18
4.9 STORE WORD operation of S-Type State	18
4.10 B-Type Format [4]	19
4.11 BEQ operation.....	19
4.12 U-Type Format [4]	20
4.13 U-type States [4]	20
4.14 J-Type Format [4]	21
4.15 J-type States	21
4.16 Simulink Model of RV32I Datapath with Control	22
4.17 Complete RV32I Datapath with Control [8]	23
4.18 Program Counter Implementation	24
4.19 Registers in Simulink.....	24
4.20 Immediate Generator I-type selection	25
4.21 Branch Comparator in Simulink.....	25
4.22 Arithmetic Logic Unit	26

LIST OF FIGURES – CONTINUED

Figure	Page
4.23 System Memory Map	29
4.24 Seven Paths of the Control Unit.....	32
4.25 Recovery System in Simulink	33
4.26 Recovery Finite State Machine.....	34
5.1 GPIO Input Simulation	36
5.2 Display.....	37
5.3 SPI Transmission in C.....	38
5.4 SPI on Analog Discovery 2	38
5.5 Simulation of Serial Communication Protocols	39
5.6 Portion of C Code Located in the Main Loop	41
5.7 C Setup Function.....	42
5.8 C Read Function.....	42
5.9 Utilization of FPGA Resources	44
5.10 Physical setup of the system	45
5.11 Display.....	45
5.12	46
5.13 Integrated Logic Analyzer Captures	47

ABSTRACT

This thesis investigates the use of obfuscated instruction codes within a redundant, RISC-V computer architecture to detect and defeat malware injected cyberattacks. The system abstracts the obfuscation from the user so that the system looks like a single processor edge computer. The system was tested using real-time sensor data coming from a camera while image processing algorithms were performed. The results of this thesis contribute to the body of knowledge on how to keep edge computers used in critical applications operational during a cyber attack.

INTRODUCTION

Computers have spread to every facet of our lives and are most known today as general computers such as desktops, laptops, and smartphones, however a majority of computers that can be found today are in the form of embedded systems. These Embedded systems can be found many industries from healthcare, aerospace, manufacturing, to everyday appliances such as microwaves, home climate control, and modern vehicles. An embedded system is a computer system, but with extra components either electrical or mechanical meant for specific tasks. With the introduction of Internet of Things (IoT) embedded systems have become more connected than ever, whether by local network or to the internet. This has opened up embedded systems to become more susceptible to malicious tampering. [1]

Cyber attacks are not new in the history of computing, people have been finding exploits in computer systems to collect, disrupt, or destroy data, or even the computer system itself. Because of this the field of cyber security was created for the development of methods, technologies, and policies to counter cyber attacks to maintain control of computer systems. General computers have been the primary target of cyber attacks thus cyber security to a large extent been developed for general computers in the form of anti-virus programs, software patches, and in extreme cases hardware updates. While appropriate for general computers, these techniques are not effective solutions for embedded systems as they do not have have the required resources for implementation. This brings a need for cyber security practices to adapt to devices that have reduced hardware budgets.

Because embedded systems are purpose-made they have limited resources in regards to processing power, memory, and storage, but excel in performance, reliability, and cost. Embedded systems are most often associated with microcontrollers, but embedded systems

can also be created from digital signal processors (DSPs) or Field-programmable Gate Arrays (FPGAs). [9] Embedded systems like general computers have continued to evolve over the past 50 years becoming smaller, cheaper, more power efficient, and incorporating new technologies; However before the rise of IoT embedded systems did not receive the same cyber security practices as it was largely unneeded. IoT is closely tied with embedded systems as it allows for embedded systems to connect and share data with other devices through a network allowing for it to be controlled remotely. [6]

Researchers at Montana State University (MSU) have spent the past 10 years exploring solutions to engineering problems with the use FPGAs. Currently researchers are exploring ways to improve cyber security for embedded systems. This paper builds upon previous works starting with a proof of concept paper for heterogeneous 8-bit soft-core processors on a FPGA for detecting a cyber attack.[2] More recently research which exceeds past proof of concept to a more complete system for not only detection, but recovery from attack on more advanced 16-bit soft core processors based on Texas Instruments MSP microcontrollers. [3] This paper further expands upon detection and recovery from an injection attack over serial with 4 individual, 32-bit RISC-V soft core processors implemented on a FPGA. The created toolchain uses a RISC-V C compiler and while in its early stages of development, allowed for a test program to be developed with a buffer overflow vulnerability, and implemented onto the system. The testing of the system consisted of first showing that it could perform a real world application of data processing, such as edge detection of an image; and secondly showing that it could defeat malware in the form of buffer overflow attacks. While the system created for this paper shows more than a proof of concept, further development is required before mission critical usage.

BACKGROUND AND MOTIVATION

Since the invention of the internet and the ever increasing interconnections of computer systems, the need for cyber security has had to evolve to meet the requirements for protecting these systems from malicious attacks. Embedded systems have also evolved over the past 50 years alongside the internet, cyber security, and computer systems. This is bringing a need for cyber security practices to adapt to devices that have reduced hardware budgets.

Opcodes and Instruction Sets

At the center of any embedded system is a computer, this computer is able to perform tasks many different tasks based on instructions it is provided with. The computer can be broken down into a number of hardware components, but the most important part is the processor. A processor is a circuit that is most commonly called the "brain" of the computer, a processor can be further broken down into a control unit, and a data path. The control unit can be represented as a Finite State Machine and is responsible for the process of fetching, decoding, and executing of instructions, while the data path is responsible for moving and manipulating the data based on the executed control signals from the control unit. Instruction themselves are a machine code representation of high level programming language code, usually produced by a compiler, and are encoded in a instruction set that is known to the processor. Instruction sets can either be CISC or RISC. CISC (Complex Instruction Set Computer) is an approach to designing a processors architecture in which the a given architecture is equipped with a large array of instructions. The general idea is that a single instruction can perform multiple operations to accomplish a task, thus requiring less processor cycles and less instructions in program memory. RISC (Reduced Instruction Set Computer) is the opposite, generally having fewer instructions in a set and only performing a single operation, while a single task is achieved with a series of operations.

RISC implementation is generally smaller and more power efficient than CISC due to the larger amount of hardware resources for CISC processors. Regardless of the processors instruction set in every instruction there is a format they follow, generally having a opcode and a operand. Opcodes are the processor decodes to perform a specific operation, operands are the data that the operations are carried out on. Opcodes can be broadly grouped into three main operations, arithmetic/logic, data movement, and flow control.

RISC-V Instruction Set Architecture

The RISC-V Instruction Set Architecture is an open source instruction set architecture developed by UC Berkeley. RISC-V is open source and as the name suggests is a RISC computer, and is has been gaining popularity in recent years for a number of reasons. RISC-V is customizable, there are extensions that add functionality to the base ISAs, such as M for multiplication and division, F for single-precision floating point, etc. Because of this you can pick and choose the ISA that is needed for a specific application, increasing it's power efficiency and reducing wasted space for unused instructions. RISC-V is scalable, there are four base ISAs, two primary base integer variants, the RV32I and RV64I, which provide a 32-bit and 64-bit address space respectively; the other two are variants, the RV32E and the RV128I, RV32Es has 16 registers compared to RV32Is 32 registers, and the RV128I is supports a 128-bit address space. These allow for RISC-V to be used in many different applications from embedded systems to desktop computing and large scale systems. Since RISC-V is gaining popularity the developer ecosystem growing along with it. Shown in the figure 2.1 below is the RV32I base instruction set.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI	
imm[31:12]				rd	0010111	AUIPC	
imm[20 10:1 11 19:12]				rd	1101111	JAL	
imm[11:0]		rs1	000	rd	1100111	JALR	
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ	
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE	
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT	
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE	
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU	
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU	
imm[11:0]		rs1	000	rd	0000011	LB	
imm[11:0]		rs1	001	rd	0000011	LH	
imm[11:0]		rs1	010	rd	0000011	LW	
imm[11:0]		rs1	100	rd	0000011	LBU	
imm[11:0]		rs1	101	rd	0000011	LHU	
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB	
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH	
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW	
imm[11:0]		rs1	000	rd	0010011	ADDI	
imm[11:0]		rs1	010	rd	0010011	SLTI	
imm[11:0]		rs1	011	rd	0010011	SLTIU	
imm[11:0]		rs1	100	rd	0010011	XORI	
imm[11:0]		rs1	110	rd	0010011	ORI	
imm[11:0]		rs1	111	rd	0010011	ANDI	
0000000	shamt	rs1	001	rd	0010011	SLLI	
0000000	shamt	rs1	101	rd	0010011	SRLI	
0100000	shamt	rs1	101	rd	0010011	SRAI	
0000000	rs2	rs1	000	rd	0110011	ADD	
0100000	rs2	rs1	000	rd	0110011	SUB	
0000000	rs2	rs1	001	rd	0110011	SLL	
0000000	rs2	rs1	010	rd	0110011	SLT	
0000000	rs2	rs1	011	rd	0110011	SLTU	
0000000	rs2	rs1	100	rd	0110011	XOR	
0000000	rs2	rs1	101	rd	0110011	SRL	
0100000	rs2	rs1	101	rd	0110011	SRA	
0000000	rs2	rs1	110	rd	0110011	OR	
0000000	rs2	rs1	111	rd	0110011	AND	
fm	pred	succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL	
000000000001		00000	000	00000	1110011	EBREAK	

Figure 2.1: RV32I Base Instruction Set [4]

Embedded System Security

As embedded systems become more involved in day-to-day life our dependency of embedded systems grows, this brings about a need for embedded systems to be secure to ensure reliability. However with the ever increasing threats to computers in general creating secure embedded systems is challenging with it's resource limitation. By examining the most common threats to embedded systems we can form an idea of making embedded systems secure efficiently. There are three categories of attacks that are most common in embedded systems, software based, network based, and side-channel attacks. The goal of software based attacks is to gain control over the embedded systems through vulnerabilities of the design either software or hardware, and includes malware, brute force, and buffer overflow attacks. Network based attacks take advantage of vulnerabilities in a networks infrastructure usually remotely, this includes distributed denial of service (DDoS) and man in the middle attacks. Side-channel attacks that exploit information that is exposed during normal operation of the system, this includes power, electromagnetic, and timing analysis.

Buffer Overflow Attacks

While it is useful to look at the most common threats to embedded systems, a review of the most common vulnerabilities of embedded systems is required to make a efficiently secure embedded systems. 43% of all vulnerabilities found by CVE-MITRE and the Department of Defense in 2015 were Hardware+Software. Of this Hardware+Software vulnerabilities a majority are Buffer Errors. This is shown in the figure 2.2 below. A buffer overflow attack is an attack which takes advantage of vulnerability found in a systems memory management. Through the exploitation of a buffer meant to hold temporary data, by writing much more data than the size of the buffer it is possible to inject data in adjacent memory locations of the buffer. With enough data written to the buffer it could cause overwriting of data

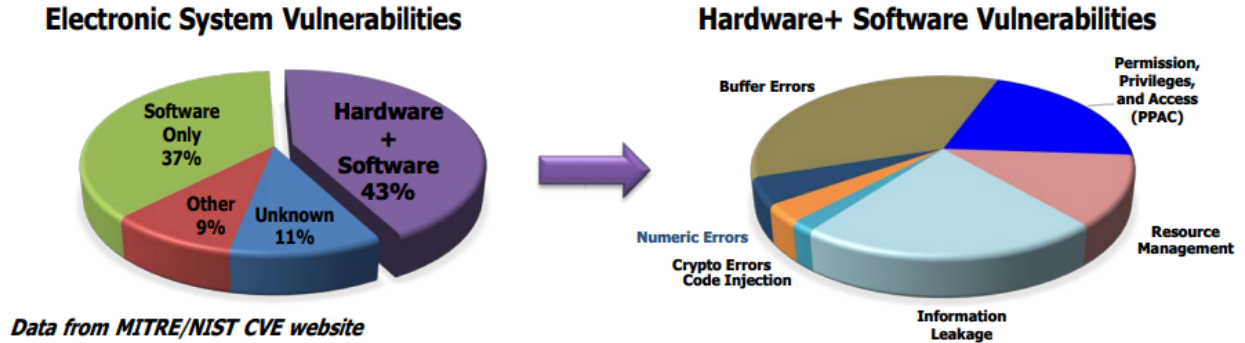


Figure 2.2: Vulnerabilities Classified [7]

structures that live in the same address space as the buffer, most notably the stack, or even exceed the data memory altogether. The complexity of the attack depends on the objective of the attacker as it is very simple exploit.

Instruction Code Obfuscation

This thesis purpose is to show a solution to buffer overflow attacks on a 32-bit embedded system. This is done with instruction code obfuscation and is just as the name suggests, by altering the operating codes of program to increase the difficulty of malware in the form of buffer overflow attacks. This was accomplished by first designing a schema for obfuscating opcodes of the instruction. While obfuscating opcodes in this design could be more complicated the RISC-V ISA simplifies this process by it's design, the opcodes for the RV32I specification are all located in the first 7 lower bits of the full 32 bit instruction; By simply adding or subtracting from this value with an integer, obfuscation is achieved. However to meet the goal of defeating malware, detecting modifications to the instruction memory (IMEM) is crucial, which cannot be done solely with obfuscation. Obfuscation in conjunction with redundancy in the form of multiplication of the complete system allows for detection, this is accomplished by implementing and instantiating copies of the design on

a FPGA. The obfuscation combined with multiple copies allows for the current instruction from memory to be monitored, and acted upon by a voter system. This voter system knows the schema of the obfuscation and can therefore determine if the IMEM has been altered, signifying a cyber attack has occurred, triggering the recovery system. Once the system recovers the program can again resume normal operation. A majority of the system was created in MATLABs Simulink tool which is a graphical programming environment. Utilizing the Stateflow and HDL coder tools in Simulink allows for creating the design of the processor and recovery system as graphical models and generate synthesizable VHDL code from said models.

RELATED WORK

MSU Contributions

This project can have its lineage traced years as researchers at MSU have done extensive research in the world of redundant computing implemented on a FPGA. The primary function of this redundant computing was to create radiation hardened soft cores for various space based applications. By taking advantage of redundancy in the number of soft cores implemented on a FPGA the outputs of multiple identical soft cores to sent to a voter mechanism. If this voter mechanism found a discrepancy in a soft cores output against a majority the system would then take advantage of a FPGAs ability to reconfigure hardware, which allowed for the failing soft core to be repaired. The main cause of error for a soft core in a space based application is through what's called a single event upset, a value of a bit is changed due to the hardware physically getting hit with a high energy particle, radiation.

A previous project took this idea of redundant computing and applied a randomization schema to the opcodes to create a proof of concept using a 8-bit soft core processor based on a design MSUs logic design course. This research showed the possibility of opcode obfuscation by being able to resist attacks of inserted instructions over serial. However with this soft core only being 8-bits it lacked the ability to show a modern real world application with its extremely limited memory address space and python scripts as the only toolchain for programming of the system. This led to a more recent project which took the idea of obfuscating opcodes and expanding upon soft core processor by applying it a commercial 16-bit processor such as the Texas Instrument MSP430 MCU. A soft core of a MSP430FR2355 was basis for this project as the goal was to show that application and usability of a obfuscated system could be achieved. This project not only replicated the processor, but with the use of TIs documentation replicated enough peripherals to show real world application in the form of a closed loop feedback system. By using a MSP430 as the

basis for this project TI's development environment, used for programming it's MCUs, was leveraged to support development of the application.

SYSTEM DESIGN

Design Flow

The design flow is an integral providing the system functionality and simultaneously the key to the obfuscation for this project. The toolchain in the design flow was created by a MSU researcher, but through collaboration the obfuscation technique and the memory map were created during the development of the toolchain. From a high level view the design flow takes a C program and converts it to a VHDL package file which is used to define the memory component instantiations of the system. This is done at the command line by using a makefile to compile the C program using a RISC-V compiler. Once the program is compiled the file binaries are created, with these binaries a python script is used to create a VHDL package file. This VHDL package file is similar to a library in a higher level language and contains the a set of constants which define the memory map as well as constant arrays which define both IMEM and data memory (DMEM). The design flow is shown in figure 4.1, in the Instruction Memory block the IMEM instead of being a single 32-bit block is broken into four 8-bit blocks for two reasons. The first being a limitation of implementation on a FPGA, where the largest block memory that can be instantiated is 16-bits, and this combined with the obfuscation technique of applying a offset to the 7-bit opcode for each core lead to breaking the IMEM into four blocks which combined create the full 32-bit IMEM. In each of the primary IMEM and DMEM VHDL files an array is declared as a signal, this signal is assigned a value from the package file. This is how the IMEM is initialized with the program, but since there are four VHDL files that make up IMEM for a single soft core, obfuscation only needs to be applied to the IMEM file that forms the lowest 8-bits of IMEM as this is where the opcode for each instruction lives. Obfuscation is done with using a function that is also from the package file, this function has two inputs, an array and an integer, that is the array constant declare in the same package file, and a

obfuscation offset which is a generic passed throughout each soft-core when instantiated at the top level. The output of this function is then assigned to the signal array declared of the IMEM file which called the function. This function, as designed for the recovery system, only applies the obfuscation to each cell of the array that contains data from the program, leaving the cells without data as zeros. Similar to the primary IMEM, the secondary IMEM does the same thing, except for it being a signal, it is a constant making it read only IMEM. The constants which make up the memory map mentioned previously are created from the custom C libraries that were also created by the same MSU researcher who developed the toolchain.

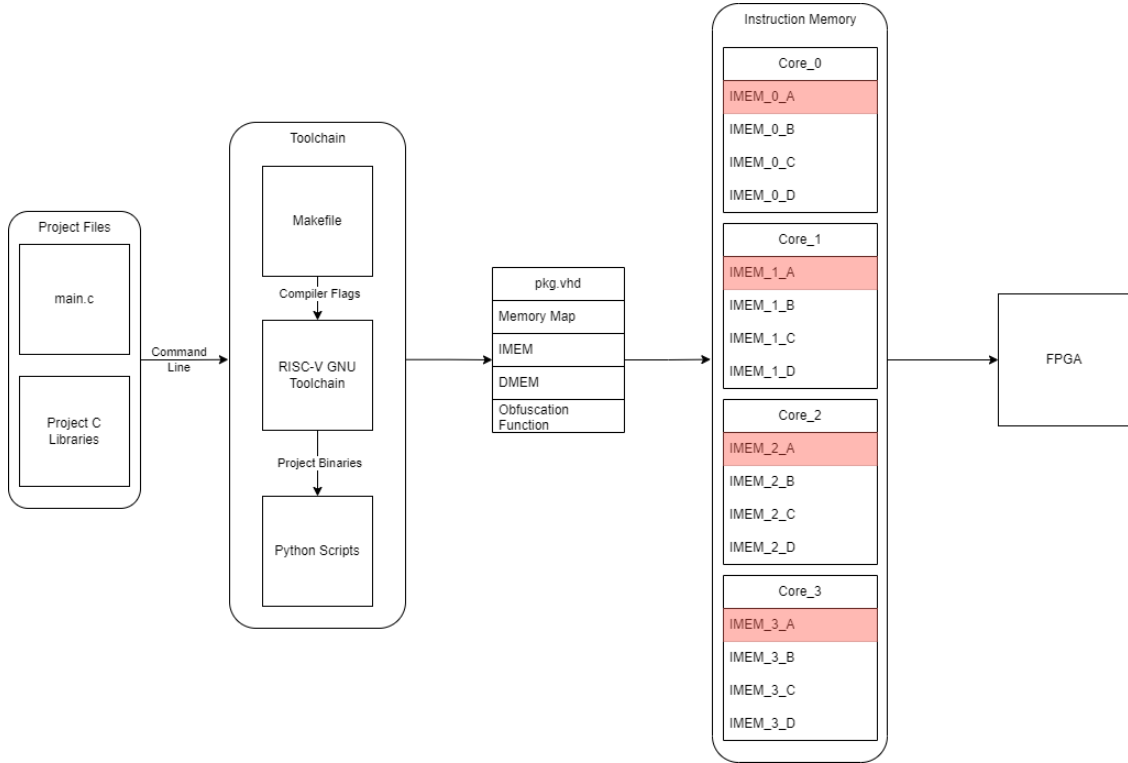


Figure 4.1: Design Flow for System

Processor

The processor is a 32-bit soft-core processor and was designed and created based off of the RISC-V ISA. The RISC-V architecture was chosen because it is an open source architecture, meaning it is well documented and can be easily modified in it's design to support obfuscation, and it's support for single clock cycle operations, yet this design is not fully single-cycle operation capable as the instructions dealing with memory require two clock cycles as this is a limitation of implementing the design on a FPGA. The RV32I base integer instruction set include 47 base instructions which are partitioned into 6 different formats based on their opcodes; R-type, I-type, S-type, B-type, U-type, and J-type instructions. This design however only implements 37 base instructions as the 10 additional instructions are needed for operations which were out of scope of this project. The processor consists of the control unit and the data path which is shown in figure 4.2 as the Simulink Model with control and feedback signals shared between the two as well as external I/O for interfacing with the voter and memory systems. The data path contains the program counter (PC), immediate generator, branch comparator, arithmetic logic unit (ALU), registers, and a feedback system for the processor registers as well the PC.

Control Unit

The control unit was created using Stateflow charts as it is a finite state machine (FSM) and is responsible for the operations of the system. The FSM performs fetching, decoding, and executing of instructions and repeats this cycle for the basic operation system. Since the RISC-V ISA groups instructions into 6 formats, this allows for 6 main branches that the control unit can traverse through. Within the 6 main branches are sub branches to traverse to specific instructions and for a majority of instructions this is a single state, but for 3 of the 6 formats this leads to a binary choice both of which are a single state of the state machine.

This enables the processor for most instructions to carry out single-cycle operations. Of course there is a reset state at the beginning of the FSM, but there is also a 7th branch which will be covered in the recovery section of the design. The control unit Stateflow chart is shown in 4.3

R-Type Instructions: these instructions cover the register-register operations. All R-type operations include a opcode; destination register, *rd*, two source registers, *rs1* and *rs2*, and a 2 function fields, *funct3* and *funct7*, to determine the operand, this is shown in figure 4.4. The figure 4.5 shows the implementation of a operation as a state and displays the control signals sent to the data path for a typical R-type operations, the only difference between other R-type operations is the *ALUSel* field, in this case the ADD operation.

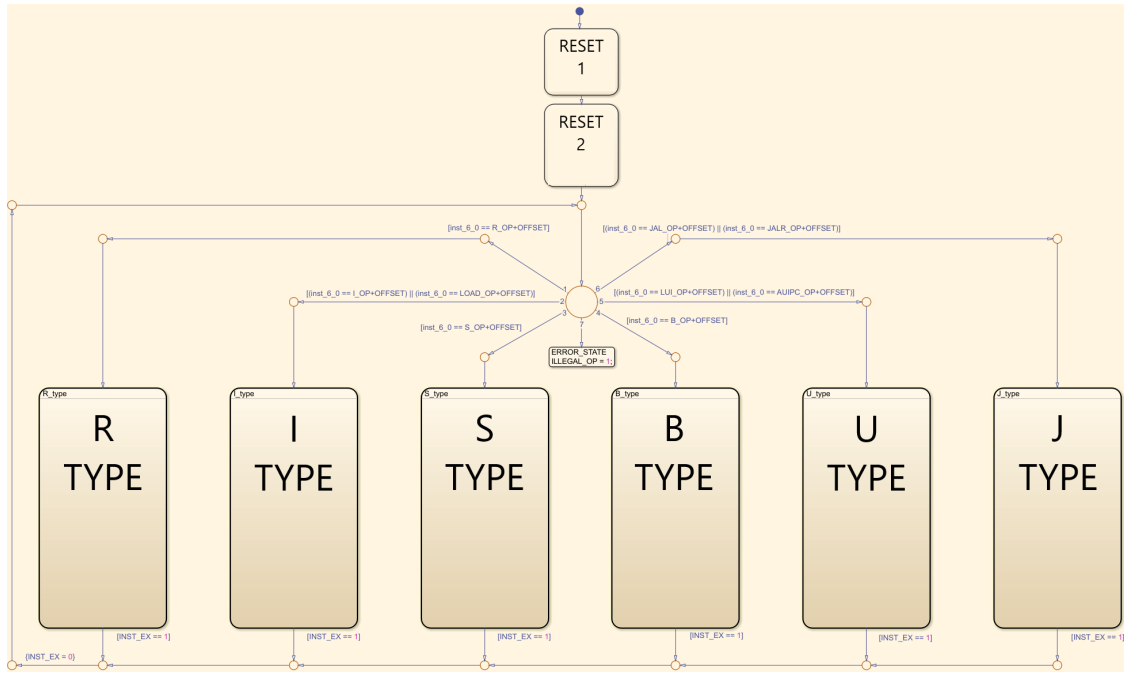


Figure 4.3: Control Unit Stateflow Model

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	src2	src1	ADD/SLT/SLTU	dest	OP	
0000000	src2	src1	AND/OR/XOR	dest	OP	
0000000	src2	src1	SLL/SRL	dest	OP	
0100000	src2	src1	SUB/SRA	dest	OP	

Figure 4.4: R-Type Format [4]

I-Type Instructions: similar to the R-type instructions, but instead of a second source register an immediate field which is 12 bits and allows for a value of range -2048 to 2047 to be generated for the arithmetic and logic operations, this is shown in figure 4.6a. Another difference is the shift operations are a specialized version of the I-type where the immediate is only 5 bits for the shift amount and the highest 7 bits of the instruction acts similar to the *funct7* field of the R-type operations. This is demonstrated in the figure 4.6b. Finally the I-type instructions include the load operations. Figure 4.6c shows that the load is the

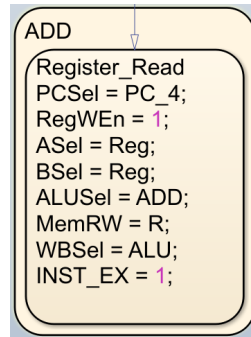
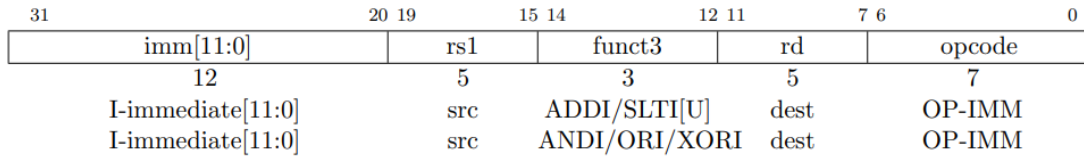


Figure 4.5: ADD operation of R-Type State

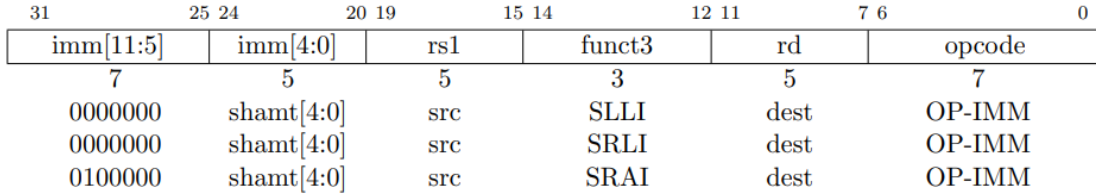
same format of the logic and arithmetic operations, but the fields are different. The opcode is a different opcode from the normal I-type operations, *rd* is the same, *funct3* is the width and the sign of the value copied from memory, *rs1* is the register which contains the base memory location, and the last 12 bits generate an offset, adding these two values gives the effective byte address.

The figure 4.7 shows the implementation of the typical I-type operation and a load operation as states. A noticeable addition to these states is the *ImmSel* field, this controls the signal tied to the immediate generator located in the data path. For all I-type operations except load, the states are the same sans the *ALUSel* field, in this case the ADD operation 4.7a. For load operations 4.7b implementation on a FPGA must be accounted for as block memory requires clock cycle to fetch data, this is done with a delay state 4.7c.

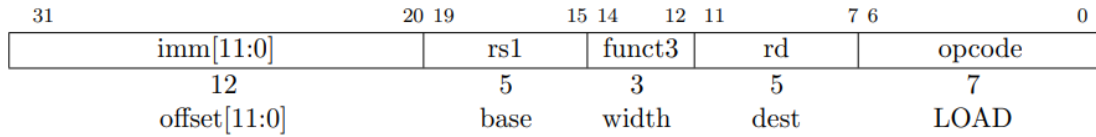
S-Type Instructions: is really only one operation which is storing values to memory. The difference between the S-type instructions is the width of the value stored. The figure 4.8 shows the format; the offset value is split between bits [7:11] and [25:31] and is still added to the base address *rs1* for the effective byte address; *funct3* is the width similar to the load instruction; *rs2* is the register location of the value to be stored. The figure 4.9 shows the implementation of a S-type state. For the S-type states the *ImmSel* field is different from I-type instructions, this is done to accommodate the change of the immediate value encoded



(a) Sub-Format 1



(b) Sub-Format 2



(c) Sub-Format 3

Figure 4.6: I-Type Formats [4]

in the instructions. Interestingly when implementing writing to block memory additional timing requirements in the form of a delay cycle is not needed.

B-Type Instructions: are the conditional branching operations. The format is shown in figure 4.10. The values of *rs1* and *rs2* are compared with the *funct3*, the operand, and the immediate field is the offset from the PC's current value, the offset is 12 bits, but within the instruction only [12:1] bits are encoded, the 0 bit is always included with the value of 0 to allow for 13 bit signed offset, giving a range of ± 4 kilobytes. The implementation of the B-type instructions is different from the rest of the implemented operations since it is dependant on feedback signals from the data path. Depending on the feedback signals a branch may or may not be taken, if taken then the PC is loaded with the current value of the PC and offset of the instruction summed, else the PC will continue incrementing through the program; this is shown in the figure 4.11 in the *PCSel* field. Since the immediate value

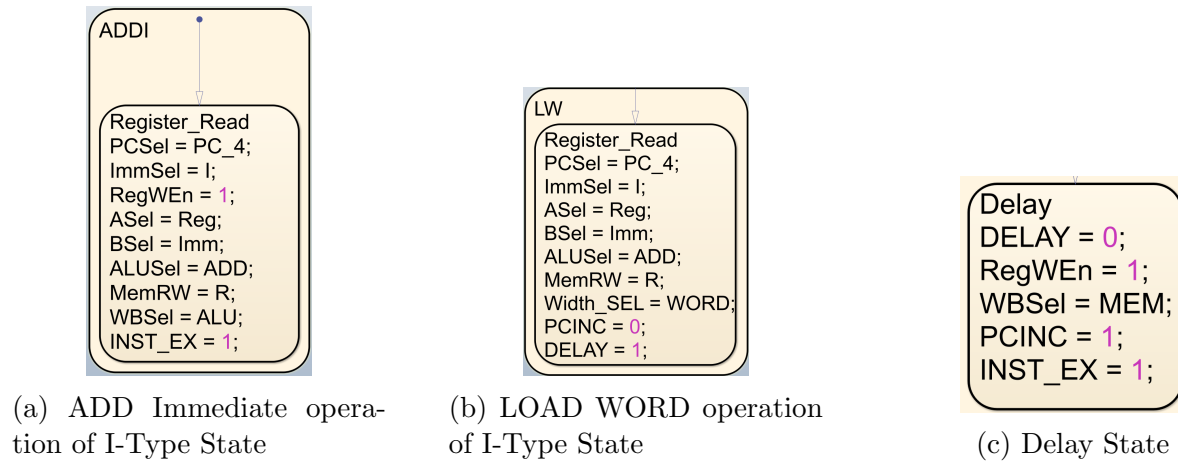


Figure 4.7: I-type States

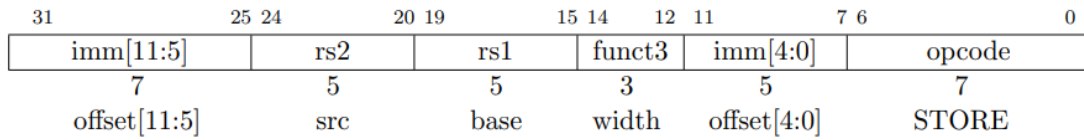


Figure 4.8: S-Type Format [4]

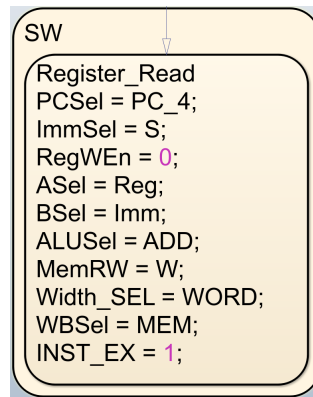


Figure 4.9: STORE WORD operation of S-Type State

is also encoded differently than I and S type instructions, a new signal must be sent to the immediate generator displayed in the *ImmSel* field.

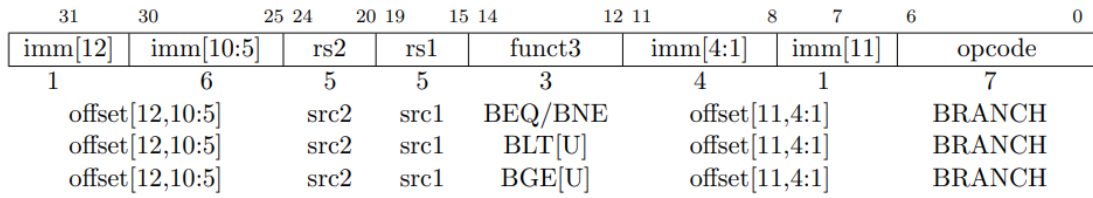


Figure 4.10: B-Type Format [4]

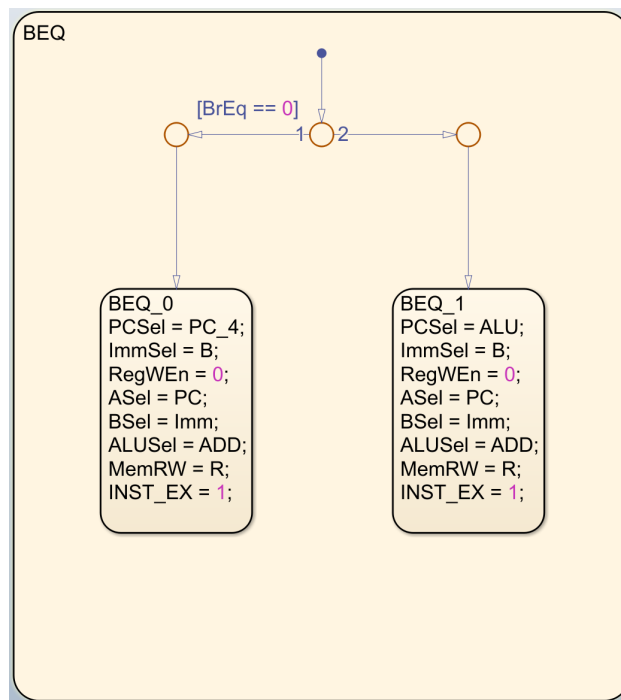


Figure 4.11: BEQ operation

U-Type Instructions: are operations to use the 20 bits encoded in the instruction to create a full 32-bit constant. The LUI instruction places the immediate value as the top 20 bits of the *rd* register filling the rest of the lowest bits as zeros. The AUIPC instruction is the same as LUI but adds the immediate value to the PC's current value, this is to create an address relative to the PC and storing this address in the *rd* register. Figure 4.12 shows this format. The implementation in figure 4.13 above reflects this format in the *ImmSel*, *ASel*, and *ALUSel* fields. For the LUI instruction the *ASel* is excluded as the ALU has a

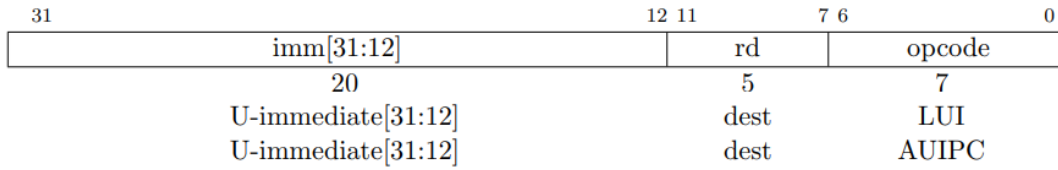


Figure 4.12: U-Type Format [4]

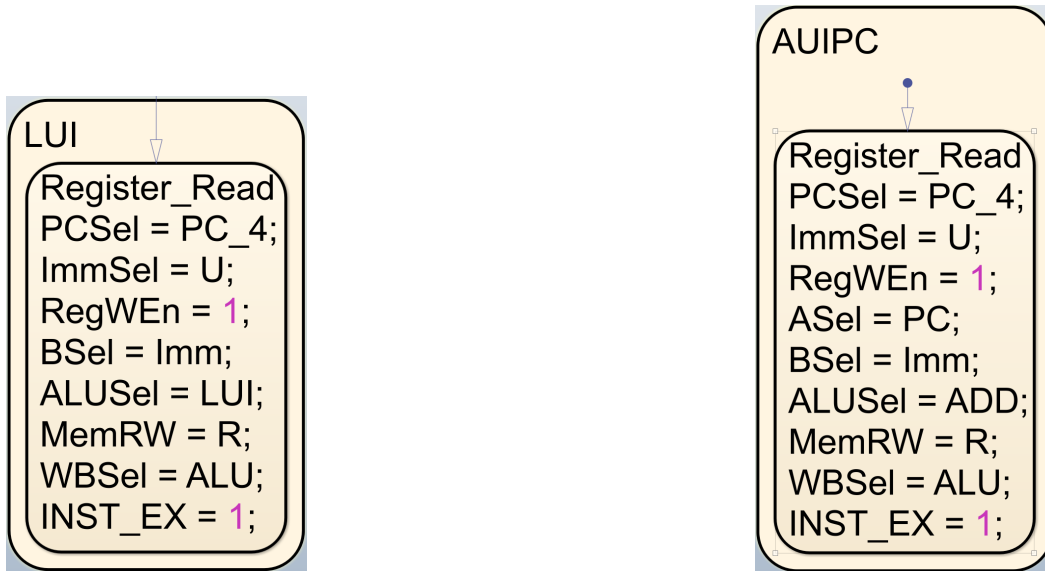


Figure 4.13: U-type States [4]

dedicated setting for loading LUI.

J-Type Instructions: are the second type of control transfer instructions, first being B-type, which is unconditional jumping. The 2 instructions of the J-type are JAL, and JALR. Similar to the U-type instructions, the JAL instruction uses 20-bit immediate value to form a sign extended value then shifting left by 1 bit to generate a jump range of ± 1 MiB; while the JALR uses the I-type encoding therefore creating the same value as if it were a I-type instruction, while giving the same range of ± 2 KiB as the I-type instructions.

Both the JAL and JALR instructions are implemented as states shown in figure 4.15,

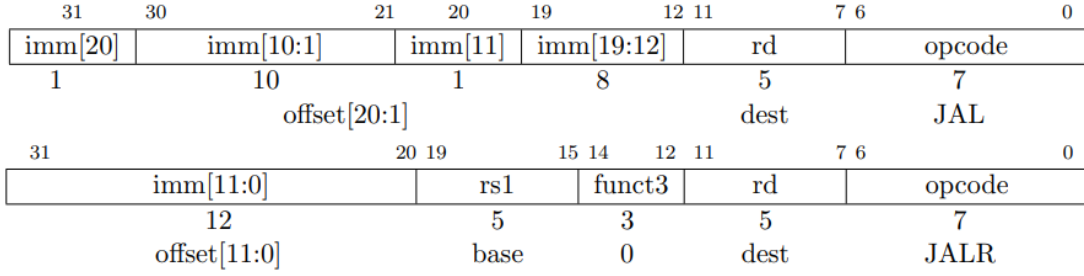
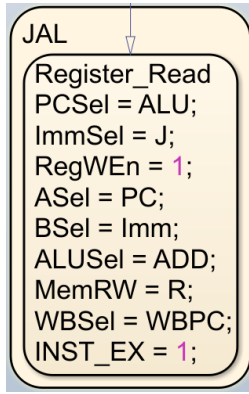
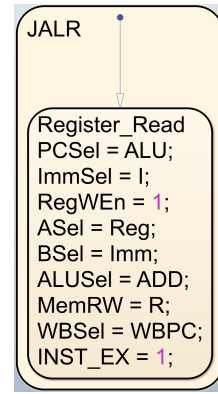


Figure 4.14: J-Type Format [4]

the most notable change compared to other instruction states is the *WBSel* field, this controls where data sent to registers comes from, in this case it will be from *PCSel* the current value of the PC for storage in the *rd* register. The *PCSel* and *ImmSel* are also different; *PCSel* sets the next PC's value to be the result of the ALU; the *ImmSel* for JAL generates a signal based on the J-type encoding while JALR reuses the I-type encoding for generating of values.



(a) JAL State



(b) AUIPC operation of U-Type State

Figure 4.15: J-type States

Data Path

The data path was created using Simulink shown in figure 4.16, in this design the IMEM is located outside of the processor and is represented by the yellow box of the Simulink model.

This Simulink model was heavily influenced by a graphical depiction of the data path in figure 4.17. While a majority of the model utilized HDL coders blocks for simple logic and data routing, a few blocks were created as a "blackbox", a way to use handwritten VHDL within a Simulink Model. These blackbox blocks include the program counter, register blocks, and the branch comparator. While Simulink is a great tool for creating a system graphically, which is more convenient than handwriting a majority of systems, the tool had it's limitations with generating synthesizable VHDL when implementing feedback with the branch comparator block. This issue only applies to the branch comparator as registers being a blackbox is a personal preference.

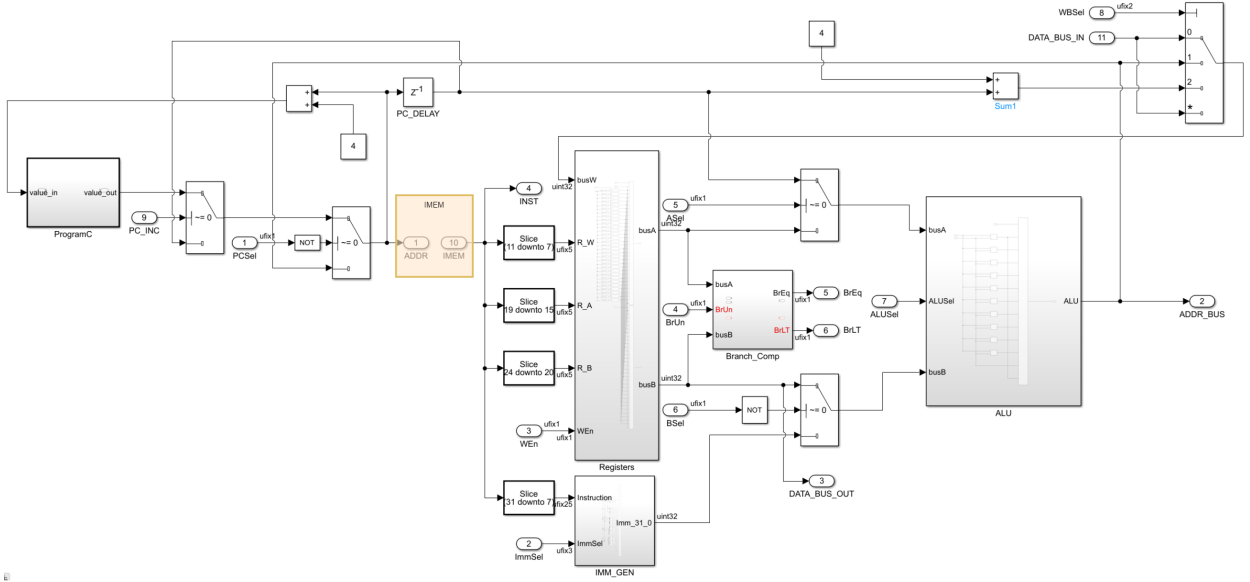


Figure 4.16: Simulink Model of RV32I Datapath with Control

Program Counter: is a 32-bit register which on it's own will continually increment by 4 until a instruction sends differing control signals. The current value of the PC is sent to the IMEM as the current address of the program, while reset value is the first address of program. The PC is a specialty register, it can not be written to the same way as the general purpose registers, only with B-type and J-type instructions, this is shown in the figure 4.18

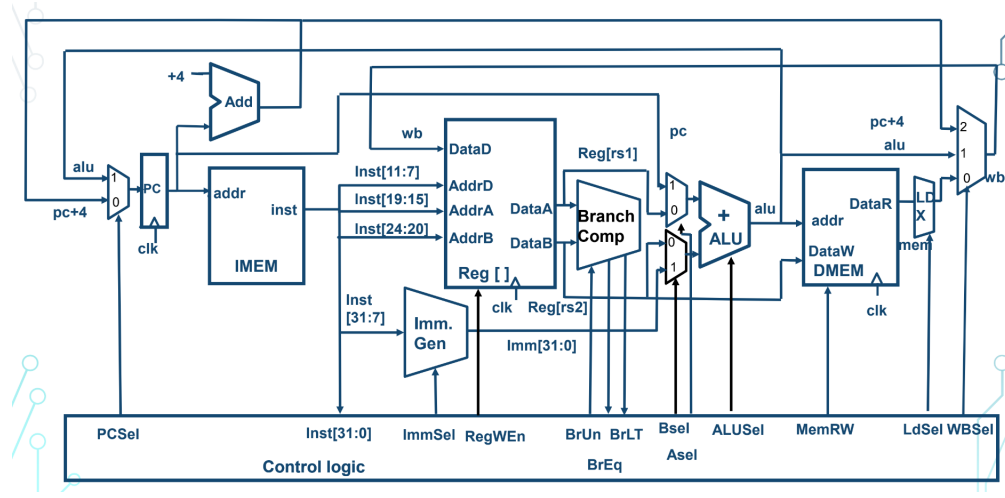


Figure 4.17: Complete RV32I Datapath with Control [8]

with the *PCSel* signal controlling a switch which chooses to either pass the PC+4 value or a value from the ALUs output. Another quirk of implementing the system on hardware with the goal of single cycle execution is that special care must be taken when reading from memory. Since reading from memory takes an additional clock cycle the PC must be held at the current value until the data is read, this is achieved with the *PC_INC* signal controlling a switch passing either PC+4 value or just PC.

Registers: are 32-bits wide and there are 32 of them, the 0 register which is hardwired to zero, while the rest are general purpose. In the Simulink model the *rs1*, *rs2*, *rd* registers are *R_A*, *R_B*, and *R_W* signals and are inputs to the Registers block. The signals are directly from the 5-bit slices of the instruction coming from the IMEM. In the figure 4.19 the registers block also has inputs *busW* from the write back multiplexer, and *WEn* a signal from the control unit that enables reads and writes to the registers.

Immediate Generator: has a wide 25-bit slice of the instruction since there is a variety of encoding formats. Regardless of the format the 25-bits are rearranged into a sign extended 32-bit value, which is then sent to the ALU. The figure 4.20 shows the I-type immediate

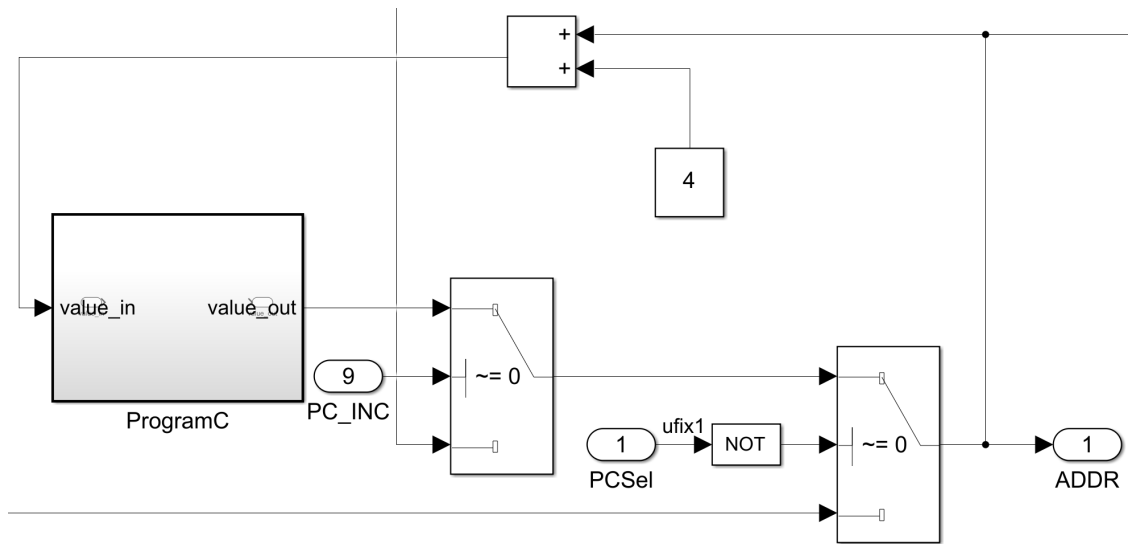


Figure 4.18: Program Counter Implementation

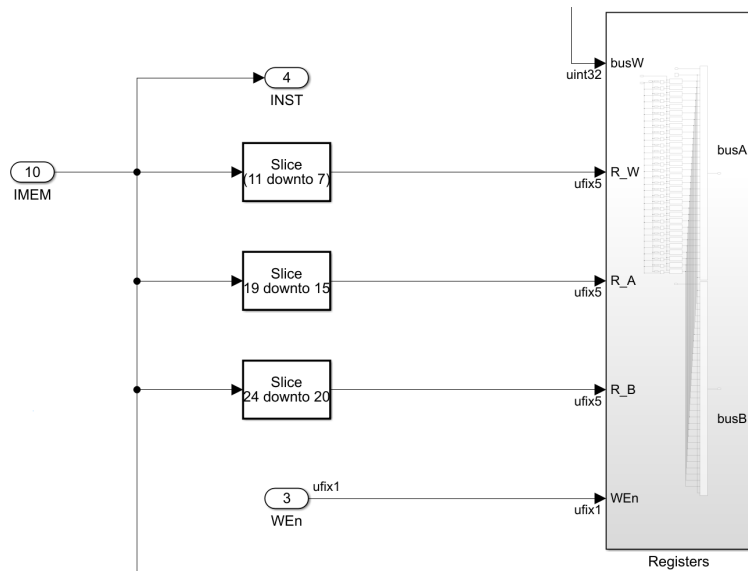


Figure 4.19: Registers in Simulink

generation and how the 25-bit slice is used to generate a value with concatenation, also shown is the 24th bit the sign bit and is used for sign extension.

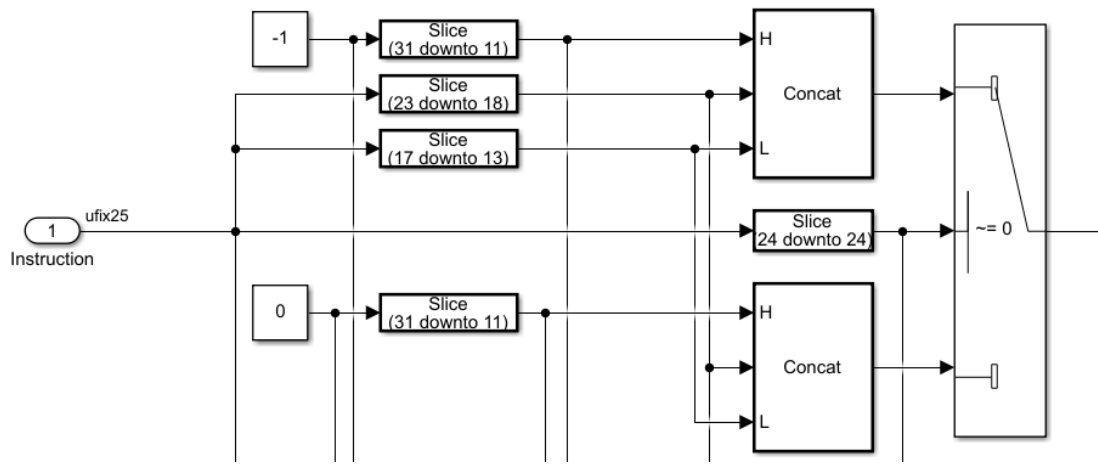


Figure 4.20: Immediate Generator I-type selection

Branch Comparator: is a feedback system that is used with B-type instructions to direct control of the program. In the model the *Branch_Comp* block is a blackbox block. Simulink has trouble generating VHDL for feedback loops that are completed on the same clock cycle, normally this wouldn't be a problem as the fix for this is adding a single clock cycle delay anywhere in the loop. This was not desirable as this would detract from the goal of a single cycle processor. The solution was removing the feedback from the model by only letting the model see the blackbox. In the figure 4.21 remnants of the error appear in red text, regardless this block has two outputs *BrEq*, and *BrLT* which informs the control unit whether to take a branch, based on a comparison of two registers either signed or unsigned.

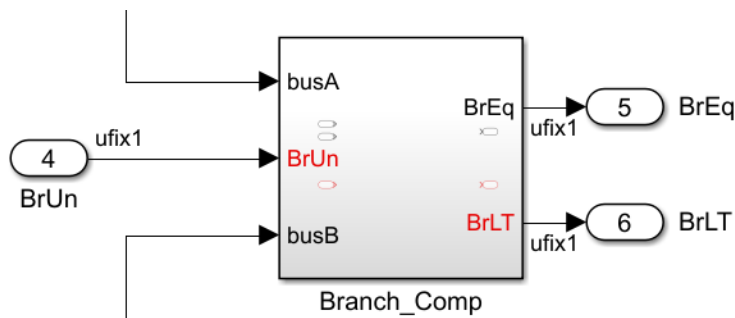
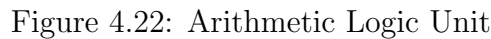


Figure 4.21: Branch Comparator in Simulink



The memory of the system was not constrained by design, rather by the hardware the system is implemented on. In this case the memory is constrained by the Artix-7 FPGA, for a single instance of this system 8 KiB of read/write instruction memory and 4 KiB of read/write DMEM, well within the limitations of the Artix-7, which has 1800 Kbits of block RAM. With the goal of defeating malware via instruction code obfuscation the memory needs grow rapidly. The design of the system has 4 identical systems implemented on a single FPGA, that is 4x8 KiB of read/write instruction memory, 4x4 KiB of read/write DMEM

and 4x4 KiB of read only instruction memory used for the recovery system. This brings the total memory usage of the system to 64 KiB, 29% estimate usage of the memory for the Artix-7 this is no problem, but to increase the size of the memory to bring the system more in line with a commercial product for example 32 KiB of read/write instruction memory would increase the total memory requirements to 288 KiB. For this reason it was key to find the right balance between needed IMEM and DMEM size required for functionality of the system and it being synthesizable to be implemented on a FPGA. The entire memory mapped system is fully word (32-bit), half word (16-bit), and byte (8-bit) addressable. Since the memory widths for the RISC-V architecture are 32-bits implementing this onto a FPGA as block RAM cannot be done with a single RAM instantiation, for this project all three memory systems, IMEM, DMEM, and the backup IMEM are divided into 4 different memory blocks each 8 bits wide to be synthesizable.

Instruction Memory: in this design is read/write capable, and handwritten to implement the obfuscation technique across the system copies. Since IMEM is both used for accessing the program via the PC and read/writeable from the memory management unit (MMU), the IMEM was implemented as a dual port RAM block which is synthesizable. The IMEM is accessed from the PC to return a instruction to the data path and also from the result of an instruction with the address generated from the ALU and the data from the *rs2* register. Due to the RISC-V architecture the instructions can only come from the IMEM, this in itself adds a layer of security as instructions inserted as a result from a buffer insertion attack on DMEM cannot influence the instructions executed. The additional read only IMEM doesn't directly interface with the memory system, rather it is only read during the recovery process.

DMEM: in this design is read/write capable, and is also handwritten, but is implemented as a single port RAM. The DMEM is accessed like the IMEM, from the ALU and

the *rs2* register.

Memory Map: in this project was created in conjunction with another researcher at MSU. The memory map is shown in figure 4.23. The address space of 0000h to 0FFFh is reserved for a multitude of control status registers by the RISC-V architecture, non of which are implemented in this project; the space of 1000h to 1FFFh is empty, but initially was the location of the IMEM, it was moved due to the test program needing more memory; 2000h to 2FFFh are reserved for peripheral registers; and 3000h to 3FFFh is the location of the DMEM. Since the address has a width of 32-bits the RISC-V architecture supports memory addresses up to FFFFFFFFh, however in the world of embedded systems and in this project the effective max address would be limited to FFFFh. The MMU for the system is implemented outside of the processor, and allows the processor to send and receive data from the memory system as well as the peripheral registers.

Peripherals

The peripherals of an embedded system allow it to perform more complex functions than could be done with solely the processor and memory system. For this system only 3 peripherals were chosen to demonstrate the objective of this project, this includes the general purpose input and output (GPIO), serial peripheral interface (SPI), and universal asynchronous receiver-transmitter (UART) modules. The peripheral selection is sparse when compared to commercial microcontrollers due to time constraints of the project. The peripherals unlike the processor were handwritten in VHDL.

General Purpose Input Output: is the primary way a embedded system interacts with the world around it. The implementation of the GPIO contains two ports, both of which are 32-bits wide, and are controlled with memory mapped registers.this project it's used

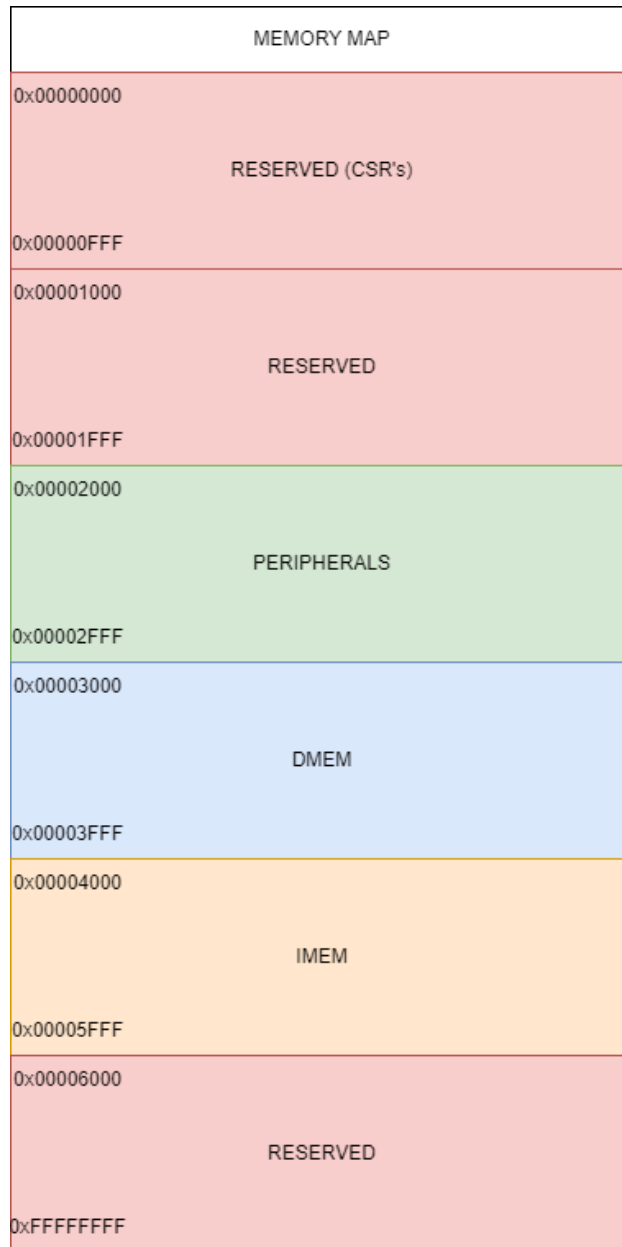


Figure 4.23: System Memory Map

to retrieve image data from a camera sensor. These registers include input, output, and direction; the input register will always represent the value of the physical pins; the direction register is controlled by the processor/program; the output is controlled by the direction register, if direction of a bit/pin is set to input then the output pin will reflect input register,

if the direction is set to output the bit/pin will be the value set by the processor or program. Since the registers are 32-bits wide they don't map directly to physical pins due to the test board used.

Serial Peripheral Interface: is a serial communication protocol that use commonly used peripheral in embedded systems. For this project it is used to send processed camera data to a display. SPI is implemented in this design with 2 ports, utilizing 1 for incoming image data and 1 for outgoing processed data. The SPI peripheral is composed of 4 memory mapped registers per port, this includes a input data, output data, transmission activity register, and a transmission status register. The peripheral also includes a hand written state machine which was created by a fellow researcher at MSU for a different project and reused in this design. This state machine ties directly to the 4 SPI signals, MOSI, MISO, SCLK, SS and updates the memory mapped registers with it's condition.

Universal Asynchronous Receiver-Transmitter: is another type of serial communication protocol and like SPI is a common peripheral with embedded systems. This project uses UART for receiving a character to change processing modes of the program that is running. Similar to SPI, the UART has hand written state machine created by the same MSU researcher and reused in this project. The state machine when receiving data updates the memory mapped registers with the incoming data in RX data register, whether it is actively receiving data in the RX ACTV register, and when it is done receiving data and is ready to be read in RX STAT register. When transmitting the state machine sends begins transmitting data stored in the TX register, during this time TX ACTV register is driven by the state machine, and the TX STAT register is updated signifying that new data is ready to be sent.

Detection and Recovery System

The process of defeating malware is composed of two steps, the detection step, and the recovery system. Detection is done by monitoring the current obfuscated opcode executed for each copy of the soft core. For this project four soft cores were implemented on the FPGA, meaning there are four unique set of instructions in each of the IMEMs. When operating under normal conditions each core will understand the obfuscation schema and will decode the obfuscated instructions as if they the base RV32I instructions. When the system has been attacked there are one of two assumptions, the IMEM has been altered with malicious instructions, or the return address is overwritten. Since the four cores are all running as a single system they all receive the same inputs while each has it's own set of outputs. This means that once PC arrives at the malicious section of code, the instructions read by each core will be the same. A combinational logic circuit, located at the top level of the design, and observes the current opcode execute by each soft core. When it identifies all cores are executing the same instruction the recovery system is triggered. Another redundant detection method is used when only the return address is altered. While under normal conditions the the control unit has six paths for each of the 6 instruction formats, there is a seventh path used when the control unit sees an unknown opcode and is shown in fig 4.24, this causes a flag to be raised and the same combinational logic circuit triggers the recovery system. If the return address is overwritten maliciously then it could send the PC into unused section of IMEM, but with the obfuscation technique used in this project only used address spaces of IMEM have the obfuscation applied to the instruction and all others are initialized to zero, meaning that if a unused instruction is executed the seventh path of the control unit will be taken and raise a error flag.

Once the detection system raises it's flag the signal is latched and the recovery process begins. The circuit for starting the recovery process is also located at the top level and the

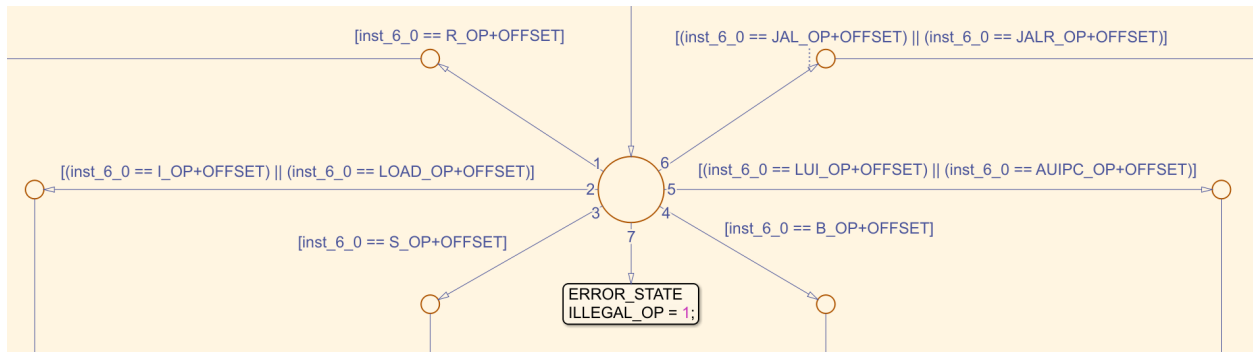


Figure 4.24: Seven Paths of the Control Unit

first action is to halt the clock for all systems, then it sends an enable signal to the recovery system located between the MMU and IMEM, this is shown in figure 4.25.

During normal operation the recovery system simply passes the read/write, address, and data signals sent to IMEM, but once enabled the inputs to IMEM are switched to outputs from a FSM located within the recovery system, shown in figure 4.26, the clock for IMEM is also switched to the clock of the recovery system. What this FSM does under normal operating conditions is sit idle, but when enabled it acts as the PC for both IMEM and the read only backup IMEM starting from the beginning of the program and begins reading values from the backup IMEM and writing the same value to the primary IMEM through it's second read/write port. This is repeated over the entirety of the IMEM. Once completed a signal is sent back to the recovery system located at the top level which sends a reset condition and re-enables the clock for the system. This allows the system to detect and recover from a malware attack, thus defeating it.

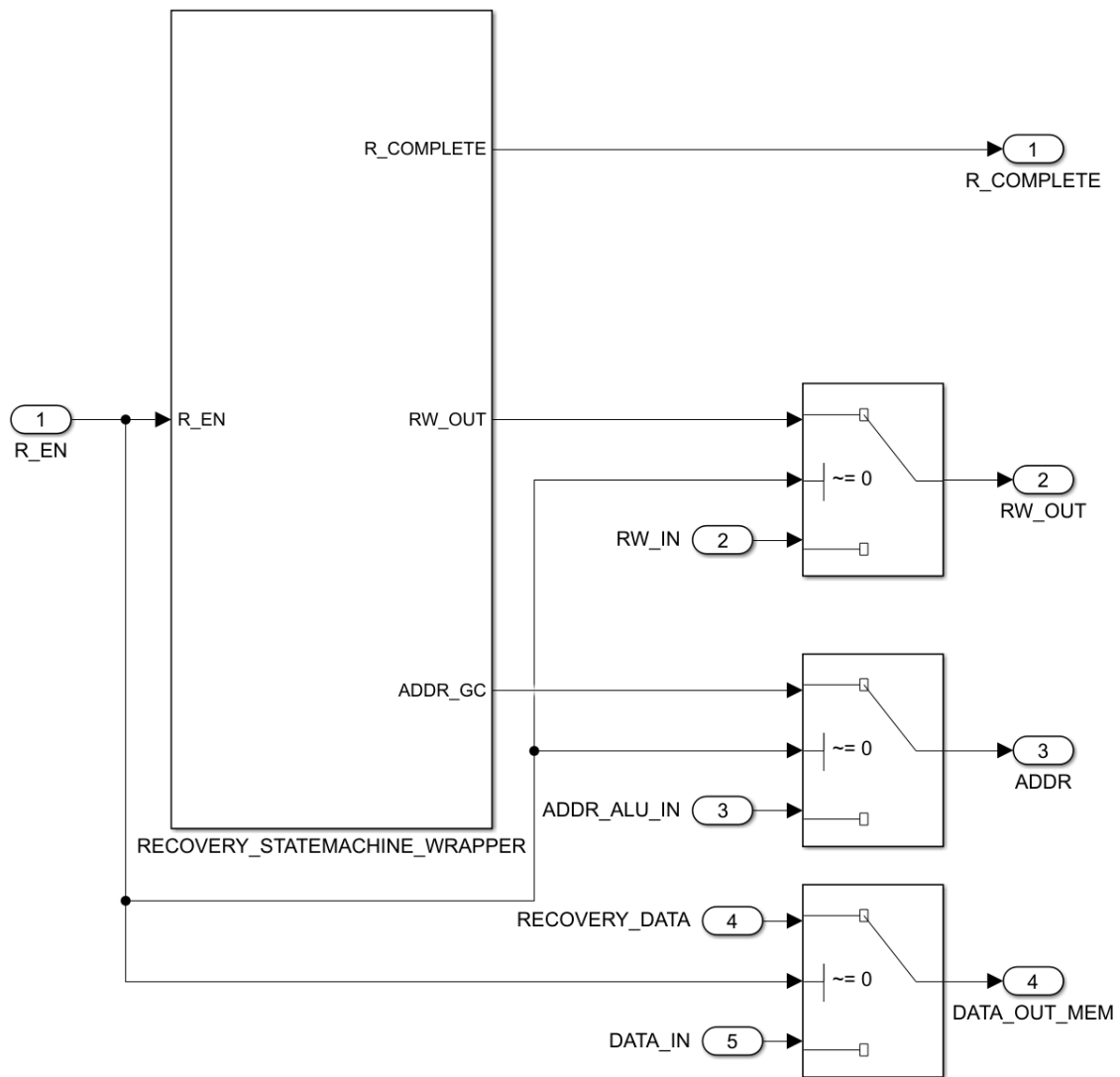


Figure 4.25: Recovery System in Simulink

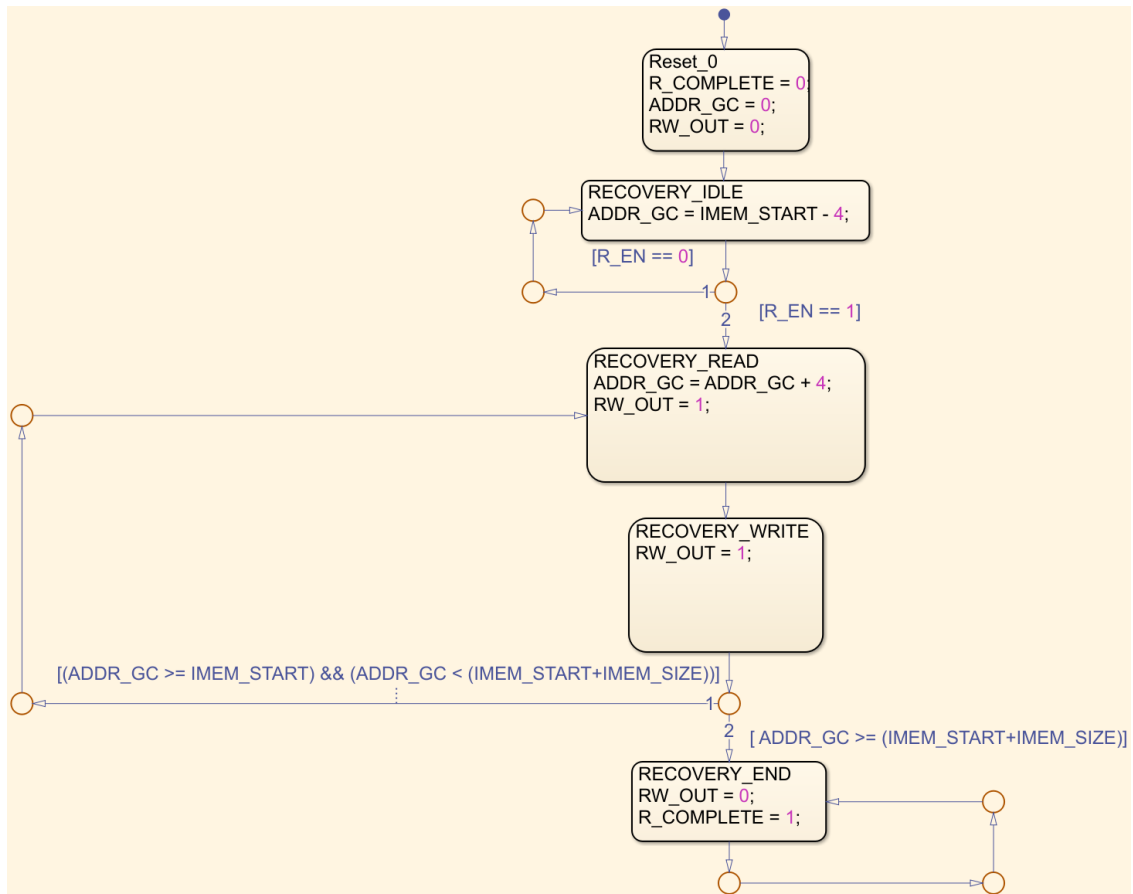


Figure 4.26: Recovery Finite State Machine

SYSTEM VERIFICATION

Over the course of this project incremental tests were made as each component was developed. After the completion of each major component tests were made as they were integrated into the full system. The bulk of tests conducted on the various system components were made in simulation during early development and eventually moved to hardware tests as the components became more complete. As was the overall system became more complete it was also subject to more hardware tests finally leading to tests for full system verification.

Processor Verification

The processor was the first component of the system developed and as such it was not tested in simulation until it mostly complete. The processor was also created before the toolchain was complete so the testing of instructions was made with a rudimentary IMEM, and for these reasons it is difficult to recreate the figures to show this process. All 37 base instructions were used in a assembly program made in a RISC-V simulator which also produced a disassembly file. With this disassembly file a python script was used to convert the disassembly file into the previously mentioned IMEM. The execution of this program was tested against the RISC-V simulator, closely monitoring and comparing the various signals of the data path and the final results of 31 general purpose registers at the end of the program. Of course the processor was not made perfect, so iteration upon the design was made until the results matched the RISC-V simulator.

Peripheral Verification

There are three peripherals that this system used for it's final design and by the time the peripherals were fully developed an early version of the tool chain was completed and

used for tests. The most complex portions of the peripherals was the FSM associated with both serial peripherals, therefore most of the testing was done on how it was integrated into the system.

Input Output Verification

The GPIO was by far the easiest to both develop and test of the three peripherals. The initial test was with a VHDL test bench, to stimulate inputs being recognized and registered by the input register, figure 5.1 shows the clock of the system in green, the test bench inputs in pink and the input register purple on the bottom. The next test was using a C program

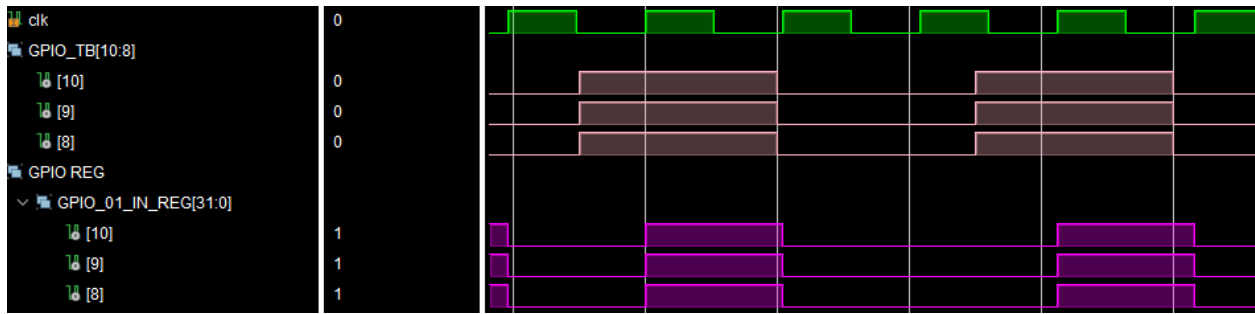


Figure 5.1: GPIO Input Simulation

to write to the GPIO specific registers shown in figure 5.2. Once the test appeared to work in simulation, a hardware test was conducted. The hardware test done by using the FPGA development board, which has switches, LEDs and push buttons. The input register was mapped to the push buttons, direction registers mapped to the switches, and the output register mapped to LEDs. This test proved the circuit was working as intended and was ready for testing in the full system with a C program. The C program was simply writing directly to the direction and output registers, and the input register again tied to the push buttons.

```
// Set GPIO 0 and GPIO 1 to inputs/outputs
GPIO_00_DIR = 0xFFFFFFFF;
GPIO_01_DIR = 0x00000700;

// Clear GPIO 1 value
GPIO_00_OUT = 0x00000000;
```

(a) Direction Register in C

```
void lcd_initialize() {
    // Reset Cycle
    GPIO_00_OUT |= 0x00000003;
    GPIO_00_OUT &= 0xFFFFFFFFC;
    GPIO_00_OUT |= 0x00000003;
```

(b) Output Register in C

Figure 5.2: Display

Serial Peripheral Interface Verification

The SPI was primarily tested in simulation using a C program. Since the SPI for this project is only used for sending data to a display the transmission mode was the primary focus of testing. This was done by using the C program that is used for the systems application shown in figure 5.3, a value is loaded to the transmission buffer register and a signal is sent to the SPI FSM. The MOSI signal was monitored in simulation to ensure proper data transfer shown in figure 5.5a. This simulation shows the process of sending a character with SPI, first the data is loaded into the transmission buffer (purple signal bus); then the TX registers signals the FSM (blue signal bus); the FSM starts the clock (green signal) for sending data while MOSI (yellow signal) changes each clock rising edge based on the value in the buffer; once this transmission is completed the TX done flag is raised (red signal) which updates the SPI status register (red signal bus). Again the output of the SPI tested and monitored but on hardware and monitored using a Analog Discovery 2 signal analyzer, shown in figure 5.4. These tests revealed a few inconsistencies when integrating the SPI FSM with it's registers and the MMU. These bugs were eventually fixed after thorough testing.

Universal Asynchronous Receiver-Transmitter Verification

The UART in this project is used to receive a character and change processing modes, because of this initial testing of UART was in simulation. The VHDL test bench stimulated

```
// Load SPI transmission buffer
SPI_00_I_DATA = (uint32_t) c;

// Send signal by activating TX line
SPI_00_TX_ACTV = 0x00000001;
SPI_00_TX_ACTV = 0x00000000;
```

Figure 5.3: SPI Transmission in C

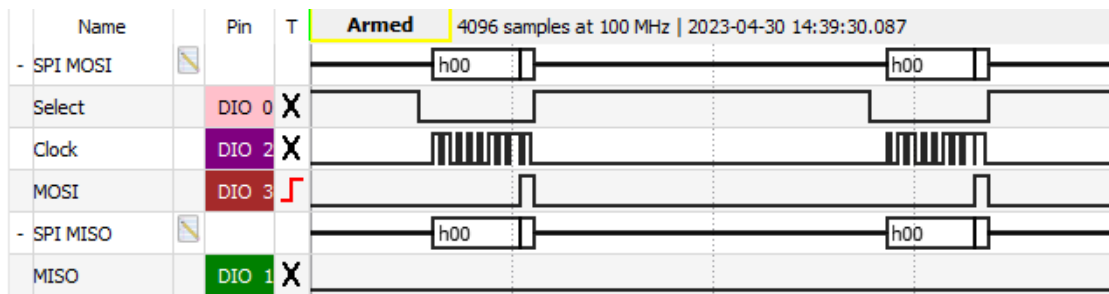
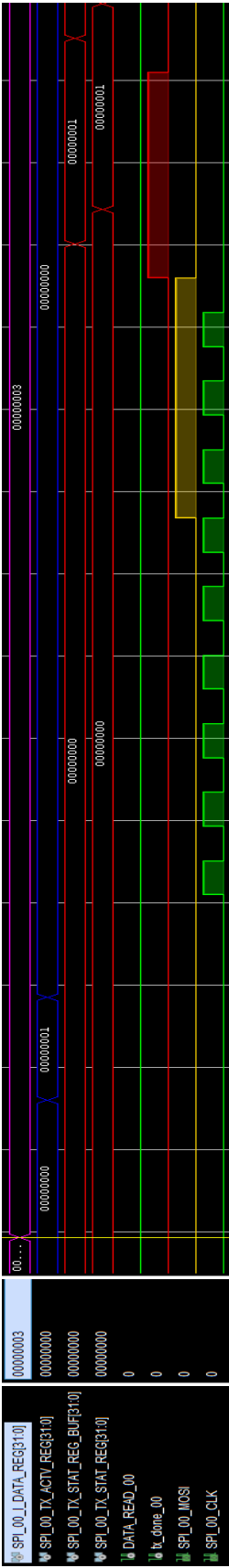
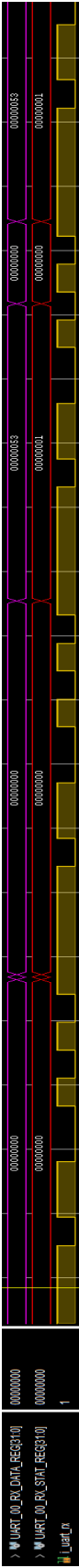


Figure 5.4: SPI on Analog Discovery 2

a TX transmission into the system, and the registers for receiving this data were monitored which is shown in figure 5.5b, the RX is sending "x56" to the system (yellow signal) which is received by the UART FSM, then data is transferred to the RX data register (purple signal bus), then a flag is raised (red signal bus) which indicates data has been received. A small C test program was used in this test, it would poll the UARTs peripheral RX status register, if new data had been received then it would place that data into memory, else the program would loop. This was a fairly painless process compared to the testing of the SPI peripheral.



(a) SPI in Simulation



(b) UART in Simulation

Figure 5.5: Simulation of Serial Communication Protocols

Recovery Verification

The testing of the recovery system was done in many small tests in simulation until the recovery system was fully complete the was then tested in simulation again then finally hardware. With the modified version of the C program used for the systems application a attack was created with a VHDL test bench, this test bench simulating a UART transmission into the system. Testing in hardware proved to be difficult with the clock speed of 48 MHz used, the external logic analyzer could not sample fast enough and no measurements could be taken, so a integrated logic analyzer (ILA) was used. With the use of the ILA the attack over a serial was able to be detected by monitoring key signals as well recovery by monitoring the current instruction executed.

Full System Verification

After completion of all major components it was time to fully test the system. This was done by utilizing a C program, which was written in part by a MSU graduate and MSU undergraduate researcher, this program used all the peripherals created to perform a non-trivial task.

System application

The application of the C program was for reading camera data over GPIO, performing edge detection processing on the image data, and sending the processed data to a display using SPI; while the processing mode could be changed by a character sent over UART. This was physically done by using a FPGA development board connected to a camera sensor over GPIO, small external display over a SPI port, and a serial connection via USB to a desktop computer to send commands for changing the processing mode.

Attack Vector and Method

Since the goal of this project is defeating malware with instruction code obfuscation, the application being performed by this system must be vulnerable to buffer overflow attacks. Inserting a vulnerability was done by creating a memory buffer for storing the character received by UART for changing processing modes. The program will progress through the main loop by first polling the UART peripheral checking if it has received data, if not it will then poll data from the camera, process the data, and send the processed data to the display then repeat, this is shown in the figure 5.6.

```
//UART has new character
if (UART_00_RX_STAT > 0) {
    uart_setup(); //Call setup function for receiving data
    mode = *buf; //Set the mode of processing the value of in memory buffer
}
```

Figure 5.6: Portion of C Code Located in the Main Loop

If there has been data received by the UART, the program will jump to a setup function where the buffers index will be set to zero then it will jump to another function where the received data is placed in the the buffer and the index for the buffer is incremented, the setup function is shown in fig 5.7 and the function for reading data is shown in figure 5.8; the program will then wait for a small delay to check if the UART is receiving more data, if so it well place the additional data at the index of the buffer. Once no more data is being received the program will then return to the setup function then again return to the main loop.

This is no problem if the data received is a single character for changing modes or even a small packet of configuration data, but a malicious actor decided to take advantage of this vulnerability by sending a lot of data the index will quickly rise and cause a number of problems. This method of creating a vulnerability actually creates two methods of attack,

```

void uart_setup(){
→   i_buf = 0; //Ensure data is placed at the begining of the buffer
→   uart_read_string(); //Call function to receive data
→   return;
→ }

```

Figure 5.7: C Setup Function

```

void uart_read_string(){
→   while(UART_00_RX_STAT > 0){ //Uart has new character
→       *(buf+i_buf) = (unsigned char) UART_00_RX_DATA; //Move data from UART to buffer
→       i_buf += 4; //Increment buffer index
→       for(i_del=delay_char; i_del>0; i_del--); //Short delay
→   }
→   return;
→ }

```

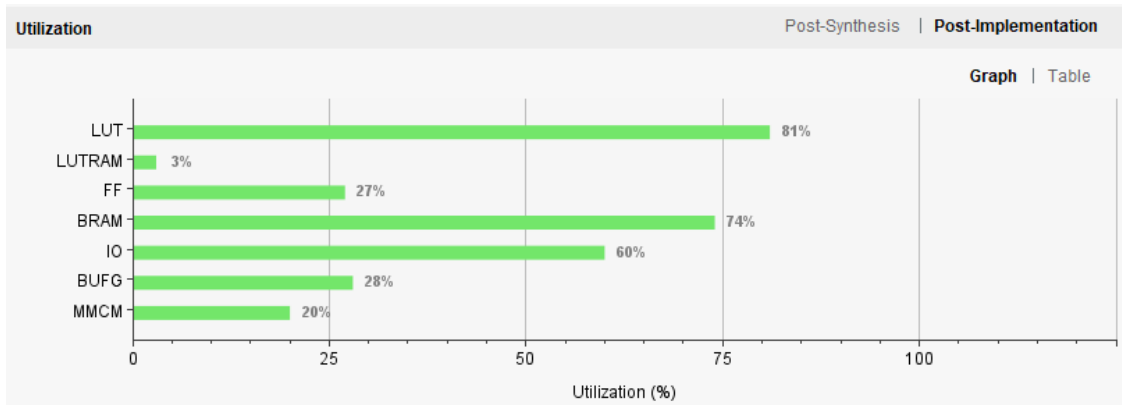
Figure 5.8: C Read Function

a brute force overflow, and a stack buffer overflow. The brute force overflow is done by simply writing so much data to the buffer that it will exceed the data memory, and with IMEM being in the memory map after data memory, the data will overflow into IMEM thus overwriting the current program. The second form of attack takes some sophistication as writing just enough data to not fill the data memory, but enough that the data could overwrite the stack, this can cause a number of problems including overwriting the return address of the program sending the program into unused space of IMEM. Both of these attacks were able to be performed on the system, with the detection system but without the recovery system, and both were able to disable the system. The sophisticated attack was tricky to perform as it required special circumstance of the system, but when testing if possible, favorable conditions were introduced to prove this was a valid attack. Due to this fact the more reliable brute force was chosen to show the results of the system defeating malware in hardware.

Full System Results

The system was fully demonstrated by first implementing the four soft cores on a Basys 3 Artix-7 development board. The overall utilization of hardware resources used on the XC7A35T FPGA used implementing this system can be seen in figure 5.9, it is worth noting the considerably high usage of Block Memory (BRAM) and LookUp Tables (LUTs) blocks used to implement this overall design. Connected with the GPIO pins of the development board was the camera sensor, and the display, internally the GPIO input and output registers for the camera, and the SPI pins for the display. The same USB port to upload the bitstream for the FPGA was used as the serial link for the UART peripheral. This specific setup can be seen in the figure 5.10.

During normal operation of the system is polling data from the camera sensor and since the default mode is the processing mode in the form of a edge detection algorithm, sends the processed data to the display, this is shown in the figure 5.11a. The system can also be sent a character to switch to no processing mode shown in figure 5.11b, and can sent another character to show the processed data mode. The attack performed is the brute force overflow, that is sending a lot of data to overflow the memory buffer, flooding the IMEM; since the attack needs to ensure that it overwrites IMEM this attack can span a couple of seconds, the figure 5.12 shows that over the duration of the display begins flashing white which is the same, which is the default color if no data is sent to it during it's refresh time. The attack is also monitored with the use of a ILA, the figure 5.13a shows the detection signal is triggered when the attack begins. After the detection of the attack the ILA is used to monitor for the reset condition that comes when the recovery system is finished repairing the IMEM, this is shown in figure 5.13b. Since ILA can only monitor a finite amount of samples it was not possible to monitor both the detection and recovery resetting the entire system at the same time because of the time it takes to repair the IMEM.



(a) Utilization Graph

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	16942	20800	81.45
LUTRAM	326	9600	3.40
FF	11087	41600	26.65
BRAM	37	50	74.00
IO	64	106	60.38
BUFG	9	32	28.13
MMCM	1	5	20.00

(b) Utilization Table

Figure 5.9: Utilization of FPGA Resources

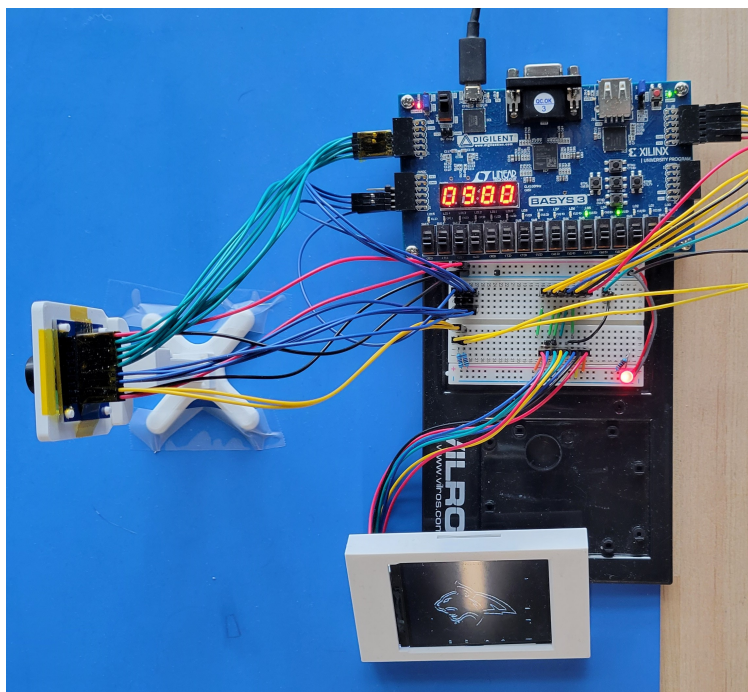
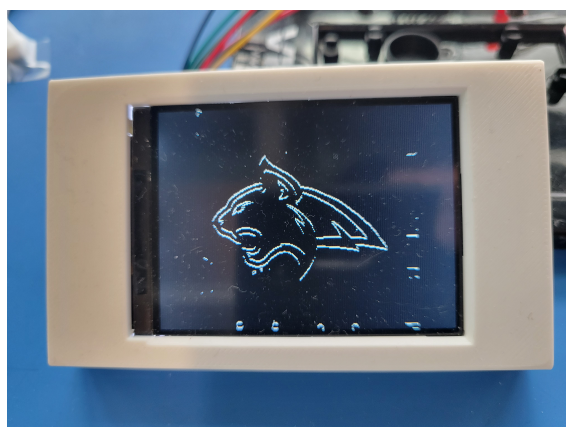
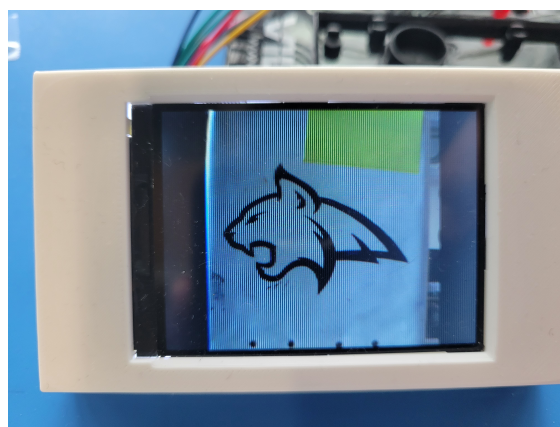


Figure 5.10: Physical setup of the system

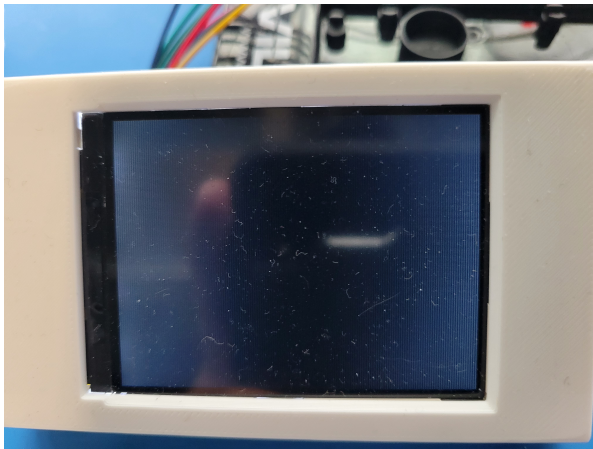


(a) Edge Detection

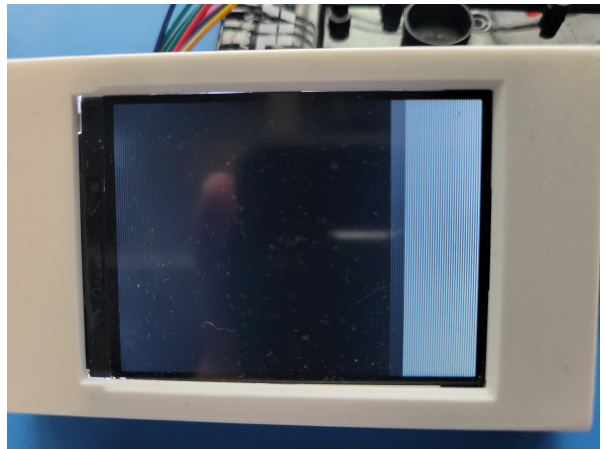


(b) Color Image

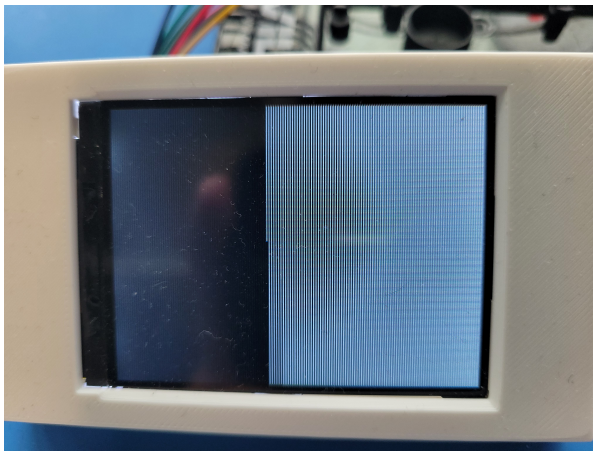
Figure 5.11: Display



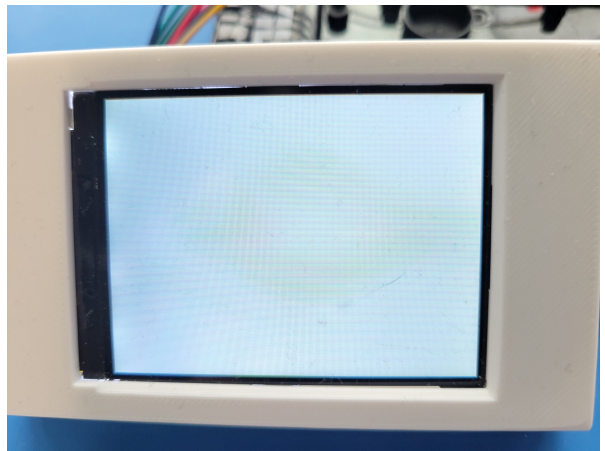
(a) Screen Flash 1



(b) Screen Flash 2

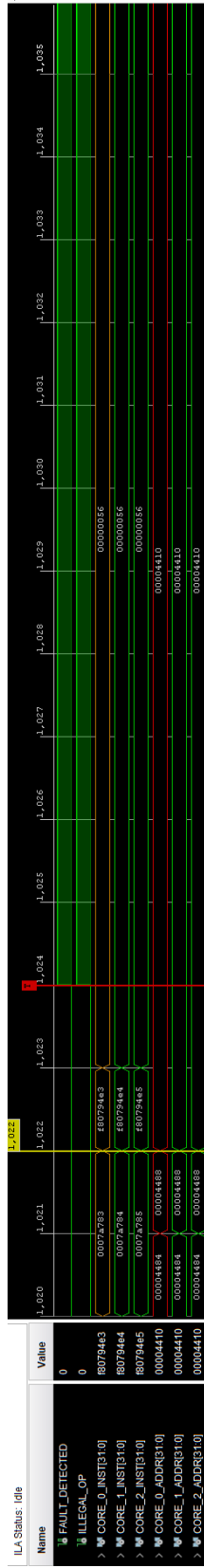


(c) Screen Flash 3

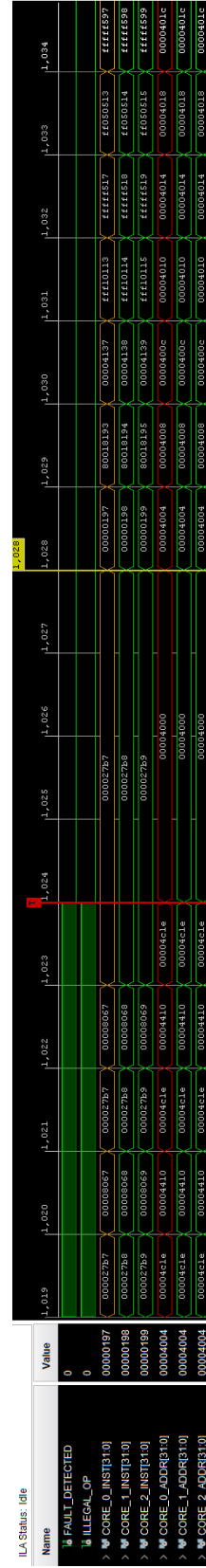


(d) Screen Flash 4

Figure 5.12: Display While Attack is Occurring



(a) Detection Signal of Attack



(b) End of Recovery of System

Figure 5.13: Integrated Logic Analyzer Captures

FUTURE PROGRESS

The RISC-V architecture has many applications in the embedded systems world, this malware resistant architecture even more so. However there are many paths forward to improve upon not only for functionality of the system, but increasing it's resilience against malware.

Processor Expansion

The RISC-V processor used in this project was based on the RV32I ISA, but does not include all the instructions from this ISA nor the full suite of ISA extensions. The processor currently only has one privilege mode implemented, and thus does not provide the protection nor the functionality that additional modes provide.

Pipelining

While the current system is a single cycle processor, save for the store instructions, when implementing onto a FPGA timing constraints are a larger concern with the overall design. The largest timing warnings in this design is the logic path between the processor and the memory, the easiest solution for this is by lowering the designs clock, but there is another way. Pipelining is dividing the processors fetch, decode, execute, memory access, and write back performed into multiple stages. Pipelining allows for execution of multiple instructions in parallel effectively increasing the throughput of the processor. Pipelining will still allow for it to be a single cycle processor but will reduce the logic time dependency in the current design.

Instruction Set Expansion

The RISC-V is an open-source ISA is a collection of ISAs of which there are four base ISAs, but there are many extensions to improve functionality of the processor.

Full RV32I Implementation The current processor was created with only 37 base instructions, but there a total of 47 for the the RV32I base instruction set. The 10 missing are the memory ordering, FENCE and FENCE.I, system instructions, ECALL and EBREAK, and control status register (CSR) Instructions. The memory ordering instructions are used for synchronizing memory access for multithreading. The system instructions are used for switching privilege modes, ECALL, and for giving control to a debugging environment, EBREAK. The CSR instructions of course used to control the CSRs. While the memory ordering instructions would only be required if multithreading was an objective, the system and CSR instructions would be required to add more functionality to the system development with a debugger, and additional modes for the processor.

RISC-V ISA Expansions Depending on the application of the system the control unit could easily be modified to add support for ISA extensions as it would only require another path. Of course care would be required for applying obfuscation to the extension so as not to cause collision of obfuscated opcode values.

Privileged Mode Support

There are currently five different modes the RISC-V Instruction Set Manual[5] describes, User, Supervisor, Hypervisor, Machine, and Debug mode. The mode that is implemented on the processor for this project is the machine mode as this the base mode of the processor of the project. The supervisor, hypervisor, and user are modes that have less access to the system hardware resources compared to the machine mode. The addition of a user mode would improve the security of an embedded system as this would restrict access certain instructions and resources during user level applications. The supervisor, and hypervisor mode have privileges levels between user mode and machine mode, designed to be used with operating systems and even multiple virtual operating systems, a feature that would be far in the future of the current system. The debug mode can be considered an additional mode,

but is really a privileged version of the machine mode, this mode as it suggests is used for debugging the software running on the processor and integral for RISC-V application development with the created system. Of course with the addition of modes to the system, the implementation of control status registers is needed.

Control Status Registers The CSRs are obviously needed to implement not only the previously mentioned topics, but are also required to for certain peripherals supported by the RISC-V ISA. The RISC-V ISA already reserves locations in memory for the CSRs and the memory map for the system reflects this.

Operating System

Finally if in the far future, if all other paths have been exhausted implementing an operating system onto this system could become a possibility. An operating system like Linux could allow the system to take on more complex tasks while improving it's usability, making the system more akin to a Raspberry Pi.

Peripheral Additions

While the three peripherals used for demonstrating the project, there are still peripherals that need to implemented to make this system not only more functional but to become more in line with commercial microcontrollers.

Interrupt Support

The interrupts are a huge boon to provide functionality to the system, without having the need to poll a system to check if an event has occurred provides a huge performance boost to the system. Implementing interrupts for the control unit is fairly easy as this would become a branch that has it's decision higher than that of any instruction. Implementing previously mentioned CSRs would be required as interrupts are apart of the RISC-V ISA.

Then implementation of interrupts to each peripheral would be required. All in all this not a small task, but would essential in making this system more practical.

Timer Support

Timers like interrupts require implementation of previous mentioned additions in the processor section, but a smaller time investment than interrupts, primarily implemented as CSRs, while increasing functionality.

Inter-Integrated Circuit Support

I2C is a common serial protocol found not only in commercial microcontrollers but in many sensors and devices found today. The implementation of I2C would be just like the SPI and UART peripherals, first creating a FSM then integrating into memory mapped registers.

Toolchain Improvements

The toolchain is necessary to the development of the software for the systems application. There are many improvements to the current toolchain that could ease the heartache caused from developing on the current system, such as debugging tool, bootloader, and even alternate compilers.

JTAG Interface

The biggest bottleneck in the development of the systems software currently is having to generate a bitstream for the FPGA, which can take on the order of minutes to complete, to make updates. A JTAG interface is commonly used in microcontrollers, and would not only provide a way to directly upload iterative software faster than a generating a bitstream, it would also allow for debugging to occur greatly boosting productivity of development.

Bootloader

As this system has the ability to become more developed a natural inclusion is a bootloader. A bootloader would allow for a more application specific device to be updated remotely.

Compilers

Compilers take the high level code and transform it into machine code, and are necessary in development of software in embedded system. There are a number of compilers available for the RISC-V architecture, and for the most part use C/C++ as the high level code. There is also RISC-V support from a Rust compiler. Rust is a comparative alternative to C, it is designed to have the performance of C while being more memory safe, and although it is a fairly new programming language it has good documentation and has been on the rise in terms of popularity in recent years.

Malware Prevention Methods

The primary goal of this research was to defeat malware with instruction code obfuscation. While this goal was achieved with this system by detection and recovery the world of cyber security is vast, with many types of malware attacks and techniques to thwart said attacks. There are a few different methods of increasing security for this that could be applied, some of which are found higher level systems. The process of adding modes with varying privileges to limit user programs was already covered previously.

Stack Protections

The stack is a data structure that contains various important values such as variables, function calls, and location in IMEM, or the PC value as called the return address. The stack is the most common target when performing a buffer overflow attack, this brought

about different stack protection techniques, such as Address Space Layout Randomization (ASLR), and Stack guard. All three of these are possible due to RISC-V supporting modes with various levels of privilege, but for an embedded system the level of protection needed normally doesn't outweigh the resource cost.

Non-Executable Stack It is also worth mentioning non-executable stack method of preventing overflow attacks, as it is common in operating systems for preventing buffer overflow. In this design instructions can only be executed if they come from the IMEM, if the program counter is out of bounds of the IMEM the instruction executed will be a default value set by the IMEM themselves. Because of this design there is no reason to spend hardware resources to marking address spaces are non-executable.

Address Space Layout Randomization ASLR is the process of placing data structures like the stack as well as other important values in randomly in memory. This makes it more difficult for an attack to target the stack because every time the program executes the placement is randomized again. ASLR is normally implemented with an operating system, which this system currently is no where close to.

Stack guard Stack guard or stack canary is a common way of preventing a buffer overflow, and occurs when a function call is made and a return address is placed on the stack, the stack guard will place a random value at the edge of the stack, located between the return address and the buffers/variables. This value is checked before returning from the function, if the value was overwritten, then an error can be created which could enable the recovery system. This could be implemented with the use of a compiler and for very little cost to hardware resources.

Diverse Obfuscation

Another method of defeating a buffer overflow attack is by further obfuscating key components in the processor. For this project the byte addressable space is $2^{32} - 1$ bytes for memory access. However for embedded system this much memory isn't always needed. For this project the memory map only ranges from 0000h to FFFFh, meaning only the lower 16 bits are realistically used. Obfuscation with same schema applied to the instructions, applied to either of the upper 16 bits, or even the top 8 bits if even more memory is required, of the PC could allow for monitoring by the same detection system located at the top of the design. If a value is written to the return address with a buffer overflow this value in the upper 16 bits would be the same across the soft cores, thus triggering the recovery system. This is a essentially hardware obfuscated version of a stack guard without the need for compiler configuration. Of course there would need to be careful considerations to ensure memory is still properly accessible.

Recovery Methods

Data Memory Refreshing

Since buffer errors are a very common vulnerability I don't think its too far fetched that an inexperienced programmer could use uninitialized memory in there programs. With this design all data memory locations are initialized to zero on hardware reset, but after recovery from an attack what is in data memory is the remnants of the attack. By including the data memory in the recovery system the data memory could be cleared, and could be used without worry.

Partial Reconfiguration

Since this project has it's roots in space based application implemented on a FPGA, which takes advantage of partial reconfiguration for recovering for radiation strikes on the

hardware. Partial reconfiguration could be an alternative, or an addition to the current recovery system if an attack is catastrophic enough to warrant this type of repair.

REFERENCES CITED

REFERENCES CITED

- [1] Michael Barr. *Programming embedded systems : with C and GNU development tools*. O'Reilly, Beijing, 2nd ed. edition, 2006.
- [2] Trevor Gahl. Triplicated instruction set randomization in parallel heterogeneous soft-core processors. Master's thesis, Montana State University, April 2019.
- [3] Brock LaMeres. Cyber shield: An approach to defeat malware in edge computers using hardware diversity., November 2022.
- [4] RISC-V International. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*, December 2019.
- [5] RISC-V International. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, December 2019.
- [6] Anthony Sabella, Rik Irons-Mclean, and Marcelo Yannuzzi. *Orchestrating and Automating Security for the Internet of Things: Delivering Advanced Security Capabilities from Edge to Cloud for IoT*. Cisco Press, 1st ed. edition, 2018.
- [7] Linton Salmon. System security integrated through hardware and firmware (ssith) proposers day overview, 2017.
- [8] Sophia Shao. Risc-v datapath and control ii.
- [9] Alan Trevennor. *A Brief History of Microcontrollers*. Apress L. P, 2012.