TRIPLICATED INSTRUCTION SET RANDOMIZATION IN PARALLEL HETEROGENOUS SOFT-CORE PROCESSORS

by

Trevor James Gahl

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

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2019

ACKNOWLEDGEMENTS

I would like to acknowledge my friends and family for standing by me and helping me throughout my time in grad school. My friend and lab-mate Skylar Tamke for always letting me bounce ideas off of him and helping me blow off steam. My advisor Dr. Brock LaMeres for helping me refocus in a field I love. More than anybody though, I'd like to acknowledge my wife Kaitlyn for helping me with anything I needed, whether that was bringing me food late at night or listening to rants about compiler errors and code bugs, I couldn't have asked for a better support system.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES – CONTINUED

Figure Page

LIST OF FIGURES – CONTINUED

Figure Page

ABSTRACT

Today's cyber landscape is as dangerous as ever, stemming from an ever increasing number of cybersecurity threats. A component of this danger comes from the execution of code-injection attacks that are hard to combat due to the monoculture environment fostered in today's society. One solution presented in the past, instruction set randomization, shows promise but requires large overhead both in timing and physical device space. To address this issue, a new processor architecture was developed to move instruction set randomization from software implementations to hardware. This new architecture consists of three functionally identical soft-core processors operating in parallel while utilizing individually generated random instruction sets. Successful hardware implementation and testing, using field programmable gate arrays, demonstrates the viability of the new architecture in small scale systems while also showing potential for expansion to larger systems.

1

INTRODUCTION

Introduction

Today's society is driven by computers. Everything from shopping to filing taxes have become online processes. It's commonly believed that the majority of computer systems produced are used for general computing devices such as laptops and desktops. However, of the over 6 billion microprocessors manufactured in 2008 only 2% were used for general computing devices [1]. The rest were used in embedded systems in devices ranging from kitchen equipment to automotives. Embedded systems are such a fundamental part of today's society that as of 2017 over 100 billion ARM processors were shipped [22]. All of these devices are susceptible to interference from malicious sources. Cyber attacks such as these generally fall into one of three categories; configuration attacks, technology attacks, and trust attacks.

Configuration attacks are any attack that targets configuration exploits. A lot of these are geared towards default manufacturer configurations such as default passwords. Trust attacks are any attacks that target the trust between different systems. Commonly these are network level attacks that are geared towards spreading viruses to increase an attackers available computing power. Configuration attacks and trust attacks are largely outside the scope of this project but will be briefly covered.

Technology attacks are any attacks that target the technology of the victim machine. A system's "technology" can be considered the underlying architecture and operating system. Of these attacks, there are again two main categories, software based attacks and hardware based attacks. A large number of attacks utilize software to attack other software, these are commonly attacks that target passwords and

remote access to a variety of internet connected systems.

Beyond software attacks are hardware attacks. However, there is some overlap between the two as many such attacks target computer hardware through software manipulation. Of these software based hardware attacks, code-injection attacks have been particularly devastating. This method of attack was even utilized as part of the Morris Worm's devastating propagation in 1988 [9]. While code-injection attacks have been around for decades, they still make up a large portion of today's cyber exploits.

As the number of cyber-based attacks increased, a new field of research was developed to combat the criminal application of technology. The field of cybersecurity is focused on finding methods to mitigate any malicious action targeting technology based systems, both hardware and software. In the event a software flaw is discovered, a fix can be readily supplied through a patch and updated software release. These can be quickly implemented and rapidly released in the case of day one bugs. Examples of this can be seen in the updates pushed out through the Windows Update service. Windows Update is utilized to constantly release patches and code-fixes to the large number of machines running the Windows operating system.

If a hardware flaw is discovered, and exploited as an attack vector, the solution is not as quickly distributed. Currently fixing a hardware flaw can cost millions of dollars and take years to implement. This massive time delay between exploit detection and patch release is caused by the requirement to manufacture an entirely new product. In some rare cases, microcode, machine code that is used for processor configuration, can be used to patch hardware flaws, but only in compatible devices. Examples of this can be seen in the recent Spectre and Meltdown attacks. These two attacks focused on hardware exploits targeted towards reading privileged memory locations.

Ideally it would be possible to deploy hardware patches as quickly and efficiently as software patches. The ability to make these rolling releases for hardware patches exists through implementing soft core processors in the fabric of a field programmable gate array (FPGA). An FPGA is a semiconductor device made of custom programmable logic components with programmable interconnects, allowing for custom digital logic to be developed and implemented. Devices such as these are commonly used in a variety of industries ranging from aerospace to medical to automotive and audio. Custom configurations are developed through a hardware description language such as VHDL or Verilog. Configurations are then synthesized and implemented in hardware through generated bitstreams, informing the device which logic components are required and how they are connected.

Montana State University (MSU) has extensive history in the application of FPGA's to a multitude of engineering problems. As part of a new focus on cybersecurity MSU has begun to leverage this expertise as a method to solve the prevalence of technology attacks in today's cyber landscape. To this effect, this thesis provides a solution to a common form of cyberattacks, code-injection, through the implementation of triplicated instruction set randomization using a triple core softprocessor on an FPGA. Each core provides identical functionality through the implementation of individual instruction sets. Testing is performed using a custom built softcore processor implemented on a Xilinx Artix-7 FPGA. Multiple test programs were written using a standard assembly language that is then run through a basic assembler to generate the processor memory. Additionally, this assembler also generates three pseudo-random sets of instruction sets, one for each core. An attack voter system is used to detect if a core has been compromised and halts the system if an attack is detected.

Successful testing was performed through serial code injection. This demonstrates the functionality of such a system. An added benefit to a method such as this, hardware implementation using a FPGA, is the ability for future hardware patches to be released through an updated bitstream that can be rapidly deployed without the need for new hardware to be manufactured. While future expansion is necessary for industrial usage, this thesis serves as a proof of concept in hardware based instruction set randomization without the need for secondary encryption.

5

BACKGROUND AND MOTIVATION

At the advent of the computer age there were a large number of computer manufacturers. Companies such as Apple, Commodore, Tandy, Radio Shack, IBM, Atari and Sinclair, were all manufacturing their own custom computer systems. As the market was still new, all of these companies were competing for dominance in the marketplace. This heterogeneous environment for computing was defined by the broad selection of devices produced by manufacturers. Today, this is no longer the case. While there may be a plethora of computer brands such as Dell, HP, and Lenovo, the components that power them and the software they run are dominated by just a few companies. This can be seen in today's homogeneous, or monoculture, environment.

Monoculture Environment

There are predominantly three major operating systems in use today; Windows, Mac, and Linux. Windows clearly demonstrates market dominance at over 70% of the market share [25]. Distantly following Windows is Mac at approximately 12% with the remaining market shares consisting of Linux, ChromeOS and miscellaneous or unknown. This can be seen graphically in Figure Figure 2.1.

While there are multiple operating systems available, the architecture that runs them is a different story. Two major processor manufacturers produce the majority of the hardware utilized in the ever growing number of technology systems; Intel and AMD. As of April 2019, Intel owns a lions share of the market at approximately 77% of the market share while AMD controls the other 23% [4]. This can be seen in Figure 2.2. It's worth noting that this market share reflects only the x86 processor architecture. X86 is the architecture required to run most desktop level operating

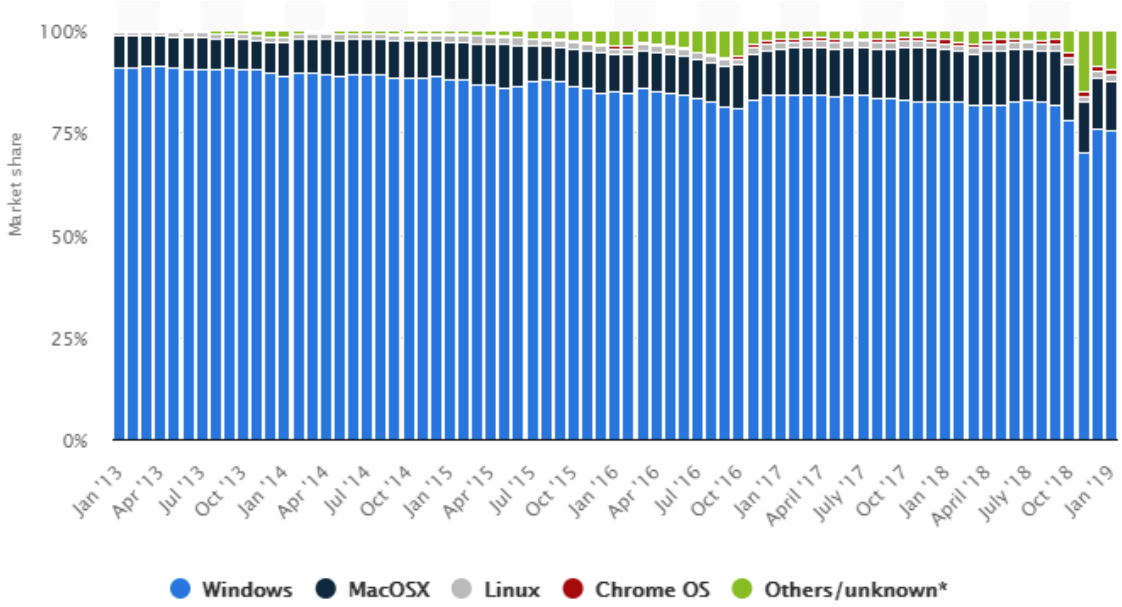

Figure 2.1: Operating system market share from 2013 to 2019 [25]

systems, such as Windows and Mac. Other manufacturers such as ARM have separate architectures that are utilized in a variety of different microprocessor systems, such as the Raspberry Pi family [19]. These alternative architectures are more likely to be compatible with Linux and the assorted other operating systems.

While Figure Figure 2.2 displays an alternative to Intel, the underlying instruction set architecture is the same between AMD and Intel. While certain technological aspects are different between the two processor manufactures, their instruction sets are the same, resulting in an overwhelming number of general computing devices using the same instruction set.

The homogeneous nature of the computer market comes with both pros and cons. Today's data centers, driving the ever expanding computational cloud, are made up of these monoculture environments. Due to the monoculture nature of these server farms, configuration can be easily automated. With a networked configuration of identical systems comes the ability to deploy configurations simultaneously across

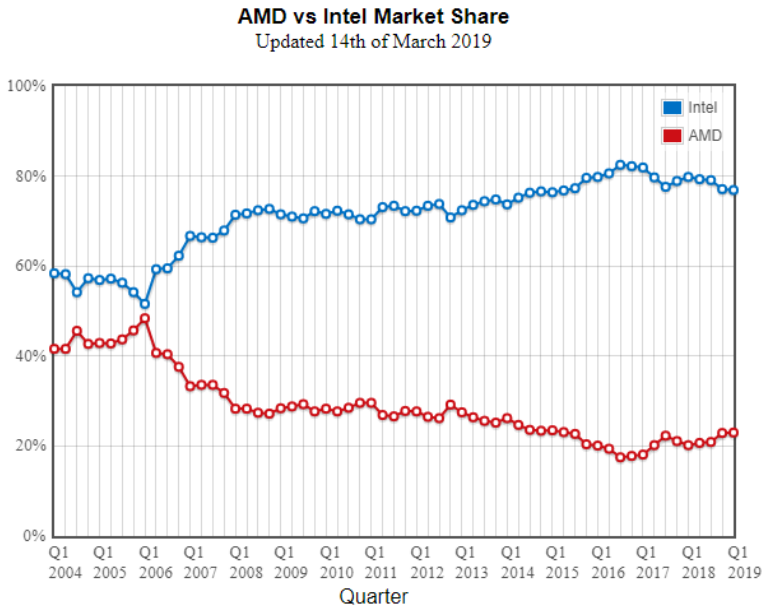

Figure 2.2: AMD vs Intel market share (Q1 2019) [4]

the entire network. Creating a datacenter in this fashion ensures that each individual system operates in the same fashion. Having every system utilize the same architecture and operating system also cuts down on the training cost of employees by reducing time spent teaching multiple systems.

What drives the ability to construct these monolithic data centers is the versatility of the processor driving every individual component of the system. This versatility comes from the opcodes and instruction set architectures configured in each processor.

Opcodes and Instruction Sets

Understanding a computers ability to execute software is crucial in understanding the way in which instruction set randomization can mitigate monoculture risks. At it's core, a computer is simply a piece of hardware capable of executing strings

of instructions. These strings of instructions, crafted into very specific sequences, become the software used to execute desired tasks. At a lower level, these instructions are called a systems "instruction set", which is comprised of a set of "opcodes". An opcode is a type of machine code that specifies what operation is desired. They commonly include the type of data, called the operand, they will be processing as well as the operation. Opcodes, as well as the format of the expected operands, are created during the processor's design cycle. When finished, these instruction sets can be very large, allowing for a wide variety of potential uses, or very small in the case of a processor geared towards a specific purpose. An example of a device geared towards a wide range of uses can be seen in the instruction set of the HCS08 microcontroller, seen in Figure 2.3.

Bit-Manipulation		Branch	Read-Modify-Write					Control		Register/Memory					
00 5 BRSET0 3 DIR	10 5 BSET0 2 DIR	20 3 BRA 2 REL	30 5 NEG 2 DIR	40 1 NEGA 1 INH	50 1 NEGX 1 INH	60 5 NEG 2 IX1	70 4 NEG 1 IX	80 9 RTI 1 INH	90 3 BGE 2 REL	A0 2 SUB 2 IMM	B0 3 SUB 2 DIR	C0 4 SUB 3 EXT	D0 4 SUB 3 IX2	E0 3 SUB 2 IX1	F0 3 SUB 1 IX
01 5 BRCLR0 3 DIR	11 5 BCLR0 2 DIR	21 3 BRN 2 REL	31 5 CBEQ 3 DIR	41 4 CBEQA 3 IMM	51 4 CBEQX 3 IMM	61 5 CBEQ 3 IX1+	71 5 CBEQ 2 IX+	81 6 RTS 1 INH	91 3 BLT 2 REL	A1 2 CMP 2 IMM	B1 3 CMP 2 DIR	C1 4 CMP 3 EXT	D1 4 CMP 3 IX2	E1 3 CMP 2 IX1	F1 3 CMP 1 IX
02 5 BRSET1 3 DIR	12 5 BSET1 2 DIR	22 3 BHI 2 REL	32 5 LDHX 3 EXT	42 5 MUL 1 INH	52 6 DIV 1 INH	62 1 NSA 1 INH	72 1 DAA 1 INH	82 5+ BGND 1 INH	92 3 BGT 2 REL	A2 2 SBC 2 IMM	B2 3 SBC 2 DIR	C2 4 SBC 3 EXT	D2 4 SBC 3 IX2	E2 3 SBC 2 IX1	F2 3 SBC 1 IX
03 5 BRCLR1 3 DIR	13 5 BCLR1 2 DIR	23 3 BLS 2 REL	33 5 COM 2 DIR	43 1 COMA 1 INH	53 1 COMX 1 INH	63 5 COM 2 IX1	73 4 COM 1 IX	83 11 SWI 1 INH	93 3 BLE 2 REL	A3 2 CPX 2 IMM	B3 3 CPX 2 DIR	C3 4 CPX 3 EXT	D3 4 CPX 3 IX2	E3 3 CPX 2 IX1	F3 3 CPX 1 IX
04 5 BRSET2 3 DIR	14 5 BSET2 2 DIR	24 3 BCC 2 REL	34 5 LSR 2 DIR	44 1 LSRA 1 INH	54 1 LSRX 1 INH	64 5 LSR 2 IX1	74 4 LSR 1 IX	84 1 TAP 1 INH	94 2 TXS 1 INH	A4 2 AND 2 IMM	B4 3 AND 2 DIR	C4 4 AND 3 EXT	D4 4 AND 3 IX2	E4 3 AND 2 IX1	F4 3 AND 1 IX
05 5 BRCLR2 3 DIR	15 5 BCLR2 2 DIR	25 3 BCS 2 REL	35 4 STHX 2 DIR	45 3 LDHX 3 IMM	55 4 LDHX 2 DIR	65 3 CPHX 3 IMM	75 5 CPHX 2 DIR	85 1 TPA 1 INH	95 2 TSX 1 INH	A5 2 BIT 2 IMM	B5 3 BIT 2 DIR	C5 4 BIT 3 EXT	D5 4 BIT 3 IX2	E5 3 BIT 2 IX1	F5 3 BIT 1 IX
06 5 BRSET3 3 DIR	16 5 BSET3 2 DIR	26 3 BNE 2 REL	36 5 ROR 2 DIR	46 1 RORA 1 INH	56 1 RORX 1 INH	66 5 ROR 2 IX1	76 4 ROR 1 IX	86 3 PULA 1 INH	96 5 STHX 3 EXT	A6 2 LDA 2 IMM	B6 3 LDA 2 DIR	C6 4 LDA 3 EXT	D6 4 LDA 3 IX2	E6 3 LDA 2 IX1	F6 3 LDA 1 IX
07 5 BRCLR3 3 DIR	17 5 BCLR3 2 DIR	27 3 BEQ 2 REL	37 5 ASR 2 DIR	47 1 ASRA 1 INH	57 1 ASRX 1 INH	67 5 ASR 2 IX1	77 4 ASR 1 IX	87 2 PSHA 1 INH	97 1 TAX 1 INH	A7 2 AIS 2 IMM	B7 3 STA 2 DIR	C7 4 STA 3 EXT	D7 4 STA 3 IX2	E7 3 STA 2 IX1	F7 2 STA 1 IX
08 5 BRSET4 3 DIR	18 5 BSET4 2 DIR	28 3 BHCC 2 REL	38 5 LSL 2 DIR	48 1 LSLA 1 INH	58 1 LSLX 1 INH	68 5 LSL 2 IX1	78 4 LSL 1 IX	88 3 PULX 1 INH	98 1 CLC 1 INH	A8 2 EOR 2 IMM	B8 3 EOR 2 DIR	C8 4 EOR 3 EXT	D8 4 EOR 3 IX2	E8 3 EOR 2 IX1	F8 3 EOR 1 IX
09 5 BRCLR4 3 DIR	19 5 BCLR4 2 DIR	29 3 BHCS 2 REL	39 5 ROL 2 DIR	49 1 ROLA 1 INH	59 1 ROLX 1 INH	69 5 ROL 2 IX1	79 4 ROL 1 IX	89 2 PSHX 1 INH	99 1 SEC 1 INH	A9 2 ADC 2 IMM	B9 3 ADC 2 DIR	C9 4 ADC 3 EXT	D9 4 ADC 3 IX2	E9 3 ADC 2 IX1	F9 3 ADC 1 IX
0A 5 BRSET5 3 DIR	1A 5 BSET5 2 DIR	2A 3 BPL 2 REL	3A 5 DEC 2 DIR	4A 1 DECA 1 INH	5A 1 DECX 1 INH	6A 5 DEC 2 IX1	7A 4 DEC 1 IX	8A 3 PULH 1 INH	9A 1 CLI 1 INH	AA 2 ORA 2 IMM	BA 3 ORA 2 DIR	CA 4 ORA 3 EXT	DA 4 ORA 3 IX2	EA 3 ORA 2 IX1	FA 3 ORA 1 IX
0B 5 BRCLR5 3 DIR	1B 5 BCLR5 2 DIR	2B 3 BMI 2 REL	3B 7 DBNZ 3 DIR	4B 4 DBNZA 2 INH	5B 4 DBNZX 2 INH	6B 7 DBNZ 3 IX1	7B 6 DBNZ 2 IX	8B 2 PSHH 1 INH	9B 1 SEI 1 INH	AB 2 ADD 2 IMM	BB 3 ADD 2 DIR	CB 4 ADD 3 EXT	DB 4 ADD 3 IX2	EB 3 ADD 2 IX1	FB 3 ADD 1 IX
0C 5 BRSET6 3 DIR	1C 5 BSET6 2 DIR	2C 3 BMC 2 REL	3C 5 INC 2 DIR	4C 1 INCA 1 INH	5C 1 INCX 1 INH	6C 5 INC 2 IX1	7C 4 INC 1 IX	8C 1 CLRH 1 INH	9C 1 RSP 1 INH		BC 3 JMP 2 DIR	CC 4 JMP 3 EXT	DC 4 JMP 3 IX2	EC 3 JMP 2 IX1	FC 3 JMP 1 IX
0D 5 BRCLR6 3 DIR	1D 5 BCLR6 2 DIR	2D 3 BMS 2 REL	3D 4 TST 2 DIR	4D 1 TSTA 1 INH	5D 1 TSTX 1 INH	6D 4 TST 2 IX1	7D 3 TST 1 IX		9D 1 NOP 1 INH	AD 5 BSR 2 REL	BD 5 JSR 2 DIR	CD 6 JSR 3 EXT	DD 6 JSR 3 IX2	ED 5 JSR 2 IX1	FD 5 JSR 1 IX
0E 5 BRSET7 3 DIR	1E 5 BSET7 2 DIR	2E 3 BIL 2 REL	3E 6 CPHX 3 EXT	4E 5 MOV 3 DD	5E 5 MOV 2 DIX+	6E 4 MOV 3 IMD	7E 5 MOV 2 IX+D	8E 2+ STOP 1 INH	9E Page 2	AE 2 LDX 2 IMM	BE 3 LDX 2 DIR	CE 4 LDX 3 EXT	DE 4 LDX 3 IX2	EE 3 LDX 2 IX1	FE 3 LDX 1 IX
0F 5 BRCLR7 3 DIR	1F 5 BCLR7 2 DIR	2F 3 BIH 2 REL	3F 5 CLR 2 DIR	4F 1 CLRA 1 INH	5F 1 CLRX 1 INH	6F 5 CLR 2 IX1	7F 4 CLR 1 IX	8F 2+ WAIT 1 INH	9F 1 TXA 1 INH	AF 2 AIX 2 IMM	BF 3 STX 2 DIR	CF 4 STX 3 EXT	DF 4 STX 3 IX2	EF 3 STX 2 IX1	FF 2 STX 1 IX

Figure 2.3: HCS08 instruction set opcode table [23]

An opcode table representation of an instruction set, as seen in Figure 2.3, contains a lot of information important to the proper use of a system. Each square contains the mnemonic representation of the opcode, the opcode value in hex, the byte size of the instruction, the number of clock cycles required to execute the instruction, and the addressing mode for the specified instruction. A breakout of the index mode subtraction opcode can be seen in Figure 2.4.

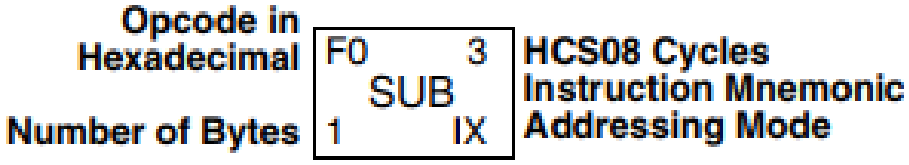

Figure 2.4: Elaboration on information contained in an opcode [23]

In a broader sense, these opcodes can be generally separated into three categories. Arithmetic/logic, data movement, and flow control. Arithmetic opcodes are operations that perform some sort of arithmetical function such as adding, subtracting, multiplying or dividing. Logical operations are anything that performs a logical manipulation on a set of data, these include operations such as "and", "or", and "exclusive or". Data movement refers to any operation that pertaining to the movement of data from one register to another, these include instructions such as load, move and store. The final primary category, flow control, consists of operations that allow for movement throughout the program. These generally include branch and jump instructions that allow for code execution dependent on a variety of different flags being manipulated in the processor. Depending on the processor, there is a fifth category, the special instructions. For example, the CPUID instruction for Intel's x86 architecture returns information about the processor to the software that called it.

When a monoculture environment is fostered, leading to a single prevalent architecture, the instruction set to this architecture becomes widely known and

implemented. While this is necessary and beneficial for software development, it creates a vector for cyber attacks. Knowing the fundamental building blocks of a processor allows an attacker to manipulate the processor in unexpected and potentially malicious ways. These malicious manipulations of any device can be considered a cybersecurity threat.

Cybersecurity Threats

Hand in hand with the rise in computer systems is the rise in cyber-crimes. Starting in 1988 with the release of the Morris Worm [9], cyber-crimes have become more sophisticated, malicious and far more numerous. In 2007 a study was conducted at the University of Maryland that found four exposed Linux systems were attacked every 39 seconds on average, with an average daily total of 2,244 attacks [18]. More recently, the Common Vulnerabilities and Exposures (CVE) system, a database of publicly known information-security vulnerabilities, reported 16,555 reported vulnerabilities in 2018 alone [16]. Since 1999 the CVE system has a total 111,684 reported vulnerabilities [16].

These reported vulnerabilities can be broadly categorized into three main categories. Configuration attacks, technology attacks and trust attacks [21]. While some of these categories see net benefits in terms of a monoculture environment, others do not.

Configuration attacks are any exploit that targets the configuration of a machine. This can be either from configurations performed by users or, more likely, configurations provided by the vendor. Exploits targeted towards machine configurations are numerous. Even to the extent that a number of websites, such as routerpasswords.com, provide default login information to a large number of varying computer systems.

Technology attacks are exploits that target the technology of the target system, such as the programming or the hardware vulnerabilities. These attack types are also often called hardware targeted software vulnerabilities, meaning that software can be leveraged to exploit hardware flaws. This susceptibility is a known risk of deploying a monoculture environment. In the past there have been a plethora of attempts at artificially diversifying homogeneous systems to prevent against this inherent risk. These include methods such as padding the runtime stack by random amounts, rearranging basic blocks and code within basic blocks, randomly changing the name of system calls, instruction set randomization and random heap memory allocation. Some of these, such as instruction set randomization, have been more successful than others.

Finally, trust attacks are exploits that occur when a command comes from a trusted source. This commonly occurs on enterprise networks. Once a single computer is compromised in a network, every computer in the network is potentially compromised. As such, this is a common method for worms to spread.

While configuration attacks and trust attacks are a serious concern, a further exploration of them is outside the scope of this thesis project. However, an overview of many of these attacks can be found in a literature review performed by Goyal et al. [6].

Technology Attacks

As mentioned previously, technology attacks are any exploit that target the technology of the system, rather than the systems configuration or trust amongst other machines.

Spectre

A recent form of a technology attack can be seen in the Spectre exploit demonstrated by Kocher et al [10]. Spectre utilizes predictive branching to access and leak sensitive information on a target computer through a side channel. Predictive branching is a technique utilized in high speed processors that allow the processor to prematurely calculate likely future execution paths. If the calculated path is correct then the processor commits that pre-calculated path and continues to run, otherwise it discards the execution.

To exploit conditional branches, the branch predictor needs to be trained to direct to a desired branch. An example of this exploit can be seen here [10]:

```
if (x < array1_size)
  y = array2[array1[x]*256];
```

The if statement will compile to a branch instruction that checks if the value of x is within a desired range. While this value is calculated the processor will calculate speculated execution paths. By running this several times with a value that is correctly defined and within the defined bounds the branch predictor will begin to predict that the execution will be returned true.

After having trained the branch predictor, if the code is run with an x that is larger than the defined value, and array1_size is uncached, the predictor will speculatively perform the read. This value will then be stored to a location that can be read by the attacker. By changing the value of x, the location being read may also be adjusted to read the target's entire memory.

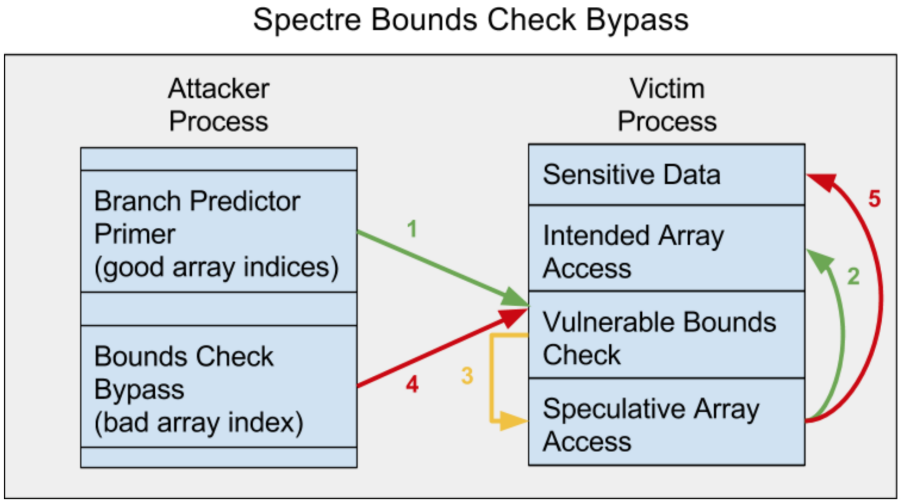

Figure 2.5: Breakdown of the Spectre attack. Arrows 1 and 2 indicate the repeated runs of the correct values. Arrow 3 demonstrates the effect that has on the branch predictor. Arrows 4 and 5 display the results of an attack after training [26].

Meltdown

Similar to Spectre is the Meltdown exploit. Operating system security has long worked under the assumption that that an application being run by a user cannot access memory in the kernel space. However, the operating system and kernel rely on the processor to enforce this separation. In 2018 it was discovered that not all processors do this.

Meltdown utilizes out of order execution to leak kernel information to a user defined space long enough for a side cache to capture the desired information [12]. For example, if the following three steps were performed:

1. Invalidate the cache for a defined user space AttackBuffer
2. Read a byte of information from kernel space KernelByte
3. Read from AttackBuffer at the offset of KernelByte

If the steps were executed in order than stage 2 would result in a segmentation fault. However, to speed up processing times, the processor assumes that at some point stage 3 will need to be executed and so begins to execute that in parallel to stage 2. This triggers a race between when stage 2 will finish executing and return a fault, and when stage 3 will finish.

Even though the fault will remove the processor's results from any code that was executed out of order, it doesn't change any cache effects. This leaves the information open to a side-channel attack.

Buffer Overflow

There is a particular subset of attacks that make up a large portion of all vulnerabilities. Hardware+Software Vulnerabilities made up 43% of the CVE and DoD vulnerability entries reported for 2015 [20]. These vulnerabilities consist of software attacks directed at hardware, things such as buffer errors, numeric errors, crypto errors and code injection, information leakage, resource management and permission, privileges and access vulnerabilities. The breakdown of these attacks can be seen in Figure 2.6.

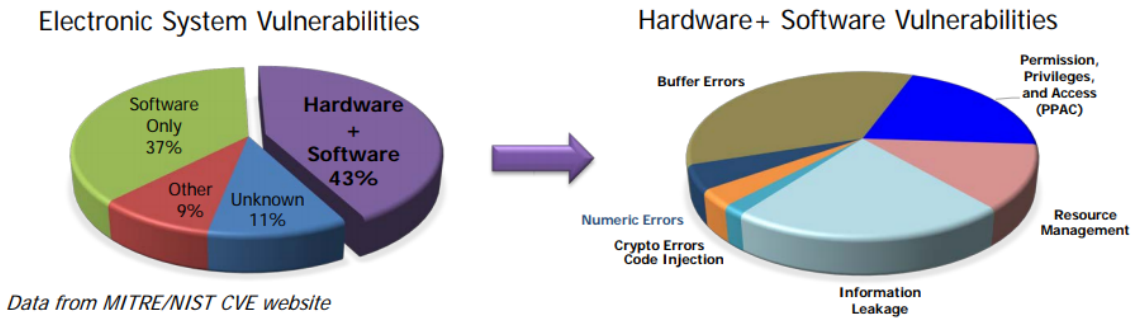

Figure 2.6: Breakdown of reported vulnerabilities in 2015 [20]

As shown in Figure 2.6 a large portion of the hardware directed software

vulnerabilities are buffer overflow vulnerabilities. Though a buffer overflow attack can be be very complex, at its core it's a simple vulnerability. By storing a value that exceeds the size of a destination buffer it's possible to inject information in adjacent memory locations. A simple example can be seen below:

```
#define BUFSIZE 256
int main(int argc, char **argv) {
char *buf;
buf = (char *)malloc(sizeof(char)*BUFSIZE);
strcpy(buf, argv[1]);
}
```

In the above example the buffer is allocated a fixed size of heap memory, however there is no limitation placed on the length of the string in argv[1] [15]. An inserted string that exceeds the length of the allocated heap size will have the extra values placed in adjacent memory. Done correctly, malicious code can be encoded in the string and stored in the adjacent memory. If the opcodes of a targeted architecture are known, its possible to insert opcodes into the adjacent memory and redirect to the processor to execute those malicious instructions.

This method of attack has been increasingly prevalent as time progresses. The CVE database system has 2,492 reported overflow attacks reported in 2018, making up 15% off all vulnerabilities reported that year. One proven method of defeating buffer overflow, or more broadly, code-injection attacks, is instruction set randomization.

Instruction Set Randomization

The idea of instruction set randomization has been around for nearly two decades, first being introduced in 2003 concurrently by both Barrantes et al., and

Kc et al. [2, 3, 8]. Conceptually, instruction set randomization is the randomization of the underlying instruction set architecture of a processor. This gives the system the impression that each program runs its own set of instructions codes that are incompatible with any other program. For example, one program may utilize 0xAC as the ADD instruction while another may utilize 0xCD or any other viable opcode value.

K.C. et al. used the idea of creating an independent execution environment for every created process. This was used to create individual opcode encryption environments. If learned instructions from one execution environment were used in a separate environment the decryption would fail, resulting in an illegal opcode. The fundamental idea can be seen in Figure 2.7.

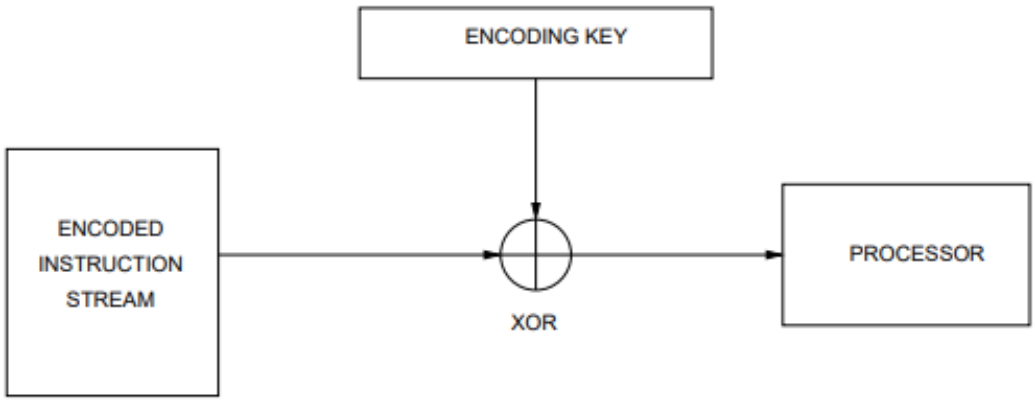

Figure 2.7: Early implementation of instruction set randomization [8]

This method was implemented using the bochs-x86 Pentium emulator, an open source emulator of the x86 architecture, and a modified Linux kernel. While the implementation tested successfully against several types of buffer overflow attacks, it was not successful against any attack that only modified the contents of the stack or heap variables that cause changes to program flow or logical operation. Additionally

this was only validated in an emulator.

Concurrent development by Barrantes et al. demonstrated a very similar concept with the exception of implementation being achieved using the randomized instruction set emulator (RISE), based on the open-source Valgrind x86-to-x86 binary translator. The RISE system is also emulator-based and relies on insertion between a supported processor architecture and the execution environment.

Both of these original methods of instruction set randomization utilized an emulation method where the code was encrypted at the binary level and then decrypted in memory prior to execution. While this works as a proof of concept, it doesn't actually randomize or modify the instruction set for any architecture. They also both relied on simplistic XOR encryption that is easily broken and not viable for defending against modern attacks.

Hardware Implementations

After the initial concept was demonstrated it was another decade before a successful implementation in hardware. In 2013, Antonis et al. first proposed a hardware supported ISR architecture they called ASIST [17]. Implementation was performed with a modified Leon3 SPARC V8 processor on a Xilinx XUPV5 ML509 FPGA.

The ASIST architecture worked by adding new registers that would supply encryption keys usable by the kernel to encrypt and decrypt instructions.

Depending on the level of the running process, one of two keys are selected using a supervisor bit to decode the instructions. Either the user key (usrkey) for user-level running processes, or the operating system key (oskey). These two keys are used to decrypt all instructions before reaching the instruction cache using an XOR.

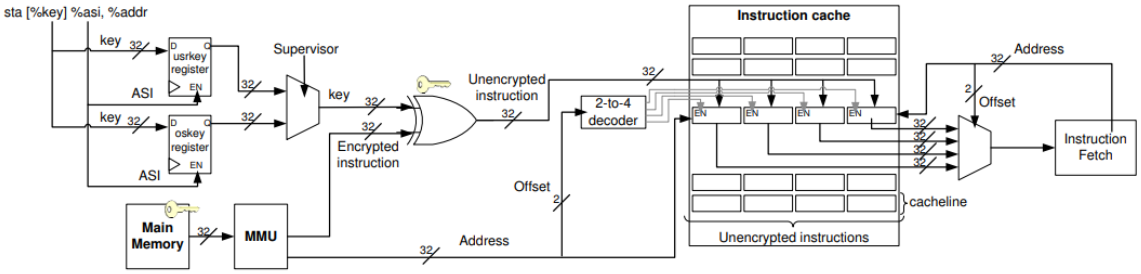

Figure 2.8: ASIST hardware support for run-time instruction decryption. Demonstrating the 32-bit key selected for either user space (usrkey register) or OS space (oskey register). Every instruction is decoded before reaching the instruction cache [17].

ASIST sidestepped many of the problems prevalent in the original implementations of instruction set randomization, such as the lack of support for shared libraries by providing this level of hardware support. This shifted the decryption to the hardware instead of just keeping the keys in ELF files. However, the hardware support was still only capable of supporting simple encryption such as XOR and transposition, both of which are easily broken.

A few years later, the use of instruction set randomization was again revisited as a method to defend against code-injection attacks as well as code-reuse attacks. Sinha et al., developed a new randomization system named Polyglot [24]. Polyglot utilized much stronger encryption in the form of AES. Additionally Polyglot encrypts at the page level, allowing for use throughout the entire software stack, from boot-loader to user applications.

While Polyglot is able to prove the effectiveness of instruction set randomization in both code-injection and code-reuse attacks, it takes a large amount of FPGA fabric. Similar to the implementation performed with ASIST, Polyglot used the Leon3 SPARC processor implemented on an FPGA. Polyglot used a Xilinx Virtex5-based XUPV5-LX110T FPGA. After the processor modifications were made the LUT

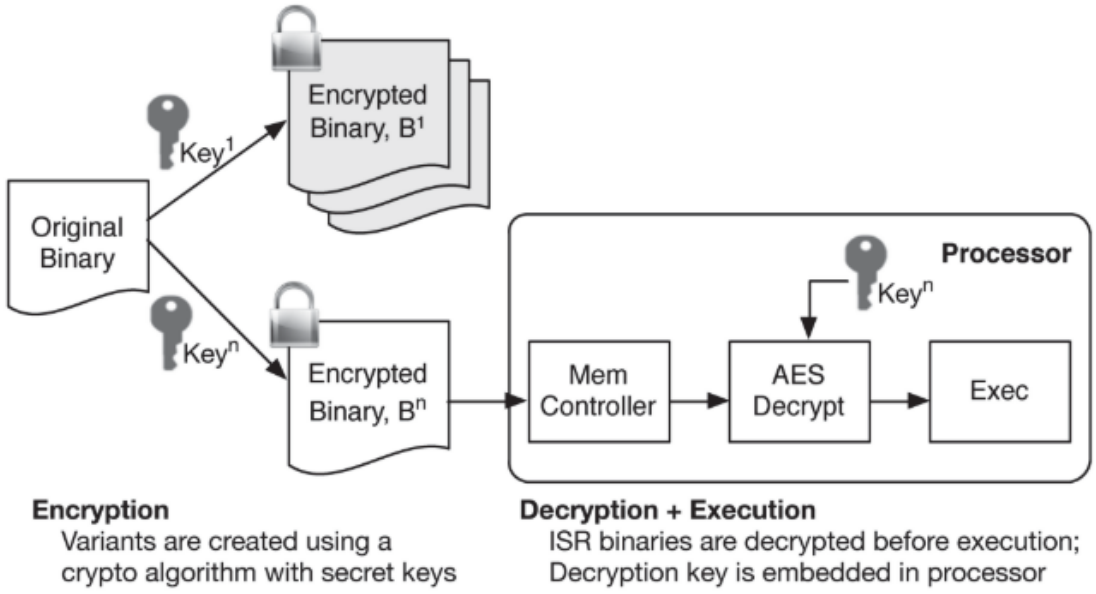

Figure 2.9: High level view of Polyglot [24]

usage increased from 13,986 to 49,724, a 356% increase.

What's Next

While multiple attempts have been made at utilizing instruction set randomization in hardware, they all still attempt to utilize the same methods as the past. Rather than randomizing the instruction sets in the processor hardware, the instruction stream is encrypted. At best this can be considered instruction set masking. Additionally, the process of encrypting adds a large amount of overhead, both in terms of fabric space and timing.

In the past, its been difficult to find a solution to these problems using existing processors. However, today it's possible to address these concerns, while still defending against code-injection attacks, by implementing instruction set randomization in hardware. The ability to do this is due to the progress made in FPGA fabric

design. Current Stratix-10 FPGAs from Intel are capable of operating at 10 TFLOPS compared to Intels Skylake processors 1-2 TFLOPS [13, 14]. With the increase in speed provided in modern FPGA's, there exists the option of creating a softprocessor inherently protected against code-injection attacks at the hardware level.

This thesis outlines the solution as a new system that utilizes three independent randomized instruction sets in three functionally identical parallel cores. Each core's output is monitored in relation to each other core through an attack voter system. If a core is detected to have a compromised instruction set then the attack voter will flag the rest of the system and trigger a system halt to prevent execution of maliciously injected code.

MONTANA STATE UNIVERSITY CONTRIBUTIONS

Existing Lab Research

The team at Montana State University - Bozeman has done extensive research into triple modular redundant computing on a range of different FPGA's using both the LEON3 soft-core processor and Xilinx's MicroBlaze soft-core processor as part of an attempt to create reliable aerospace avionics. Prior systems utilize nine MicroBlaze processors in the fabric of the FPGA, keeping three of them active at any time, with six held in reserve. A voter system is used to detect if a processor is operating erroneously. In the case of processor failure, a reserve processor is brought online while the faulted processor is taken offline for reconfiguration. After reconfiguration the faulted processor is marked available in the event another processor faults. Allowing for the constant cycling of processors in the event of continual faulting.

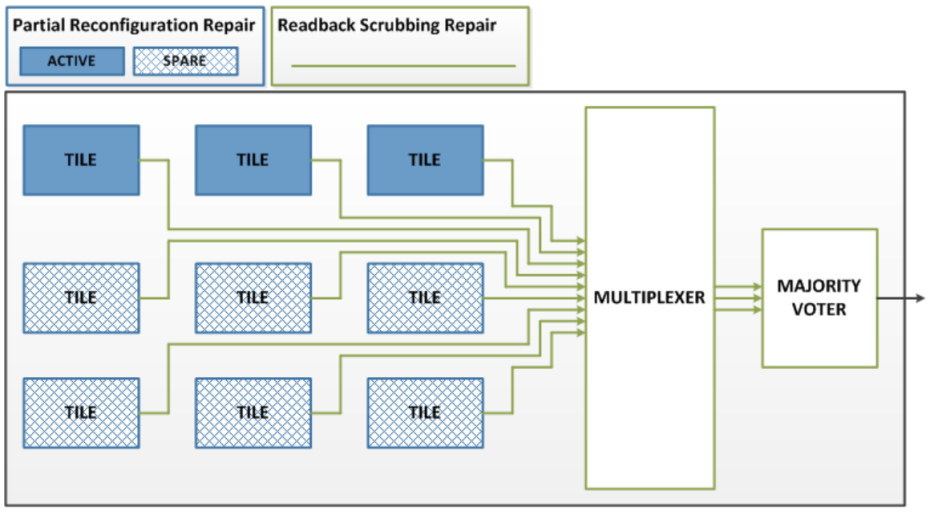

Figure 3.1: Prior implementation of triple modular redundancy [7]

While the current triple modular redundant computing demonstrates the ability to determine which system has faulted it falls short regarding cyber-security research.

For one, the only triplicated system is the MicroBlaze processor itself, the memory component is not triplicated and would be susceptible to an attack. Additionally, the MicroBlaze is a proprietary system, therefore access to the internal workings of the processor is restricted, preventing modifications to opcodes.

Other previous MSU research examined the use of the LEON3 soft-core processor as a potential open-source replacement for the MicroBlaze. A previous effort was able to implement a four core LEON3 system with similar capabilities to the nine-tile MicroBlaze system [27]. However, it required a much larger amount of FPGA fabric space to implement only four cores compared to nine.

Unlike the MicroBlaze, the LEON3 is an open source system and is therefore able to be modified. However, the ability to program the processor relies on the ability to program a single set of opcodes. If multiple versions of opcodes are desired then multiple systems with multiple I/O's are required rather than a single system with heterogeneous opcodes.

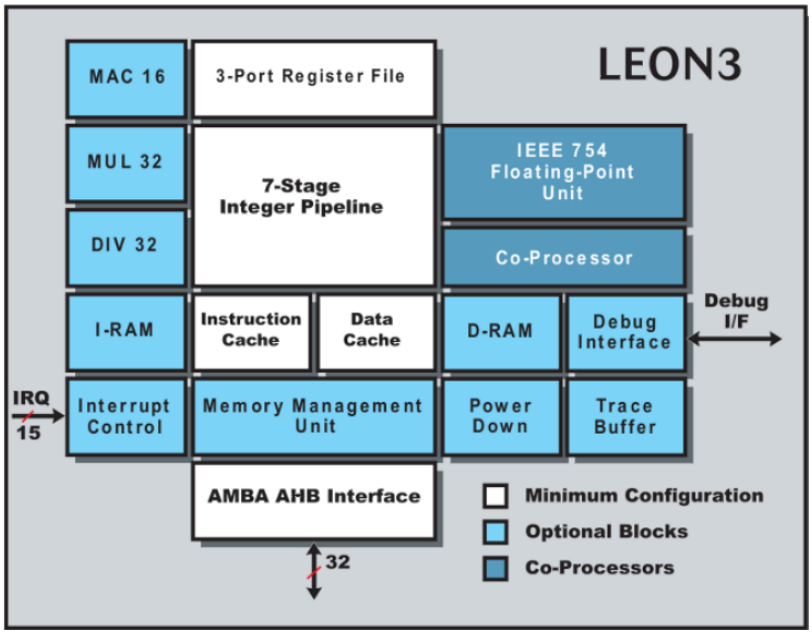

Figure 3.2: LEON3 Processor [27]

Through previous efforts it can be seen that real-time reconfiguration of soft-core processors is practical. It's also demonstrated that the expansion of such real-time modifications set the stage nicely for a heterogeneous architecture. My contribution is additive to this body of work. Utilizing the existing ideas it's possible to demonstrate a proof of concept using instruction set randomization in a triplicated processor architecture. The idea is to take a single core system, and add two more cores with independent memory and instruction sets. The underlying instruction set architecture will be maintained, allowing for the same program to be executed simultaneously on all three cores, all while running independently generated opcodes. This ideal can be seen throughout my system design and testing process.

SYSTEM DESIGN

The overall goal of this project is to actualize a computer architecture that is inherently resilient to code-injection attacks. A proof of concept of this can be realized using the soft-core processor architecture developed as part of the EELE 367 course at Montana State University. Before anything else, testing of this processor was done to verify the bare bones functionality of this framework.

Processor Foundation

The foundation of the soft-core processor developed throughout this thesis project was created as part of Montana State University's Logic Design course (EELE 367). This foundation included the following:

- Control Unit with 10 Instructions
  - LDA_IMM - LDB_IMM
  - LDA_DIR - LDB_DIR
  - STA_DIR - STB_DIR
  - BRA - BEQ - ADD - SUB
- Data Path
  - 'A' Register
  - 'B' Register
  - Buses 1 and 2
  - ALU
- 128 Bytes of Program Memory
- 96 Bytes of RAM
- 16 Input Ports
- 16 Output Ports

It's worth noting that during initial testing and implementation most of the existing instructions were removed for the sake of simplicity. After modifications, the processor only had LDA_IMM, LDA_DIR, STA_DIR, and BRA.

A top level view of the processor foundation can be seen in Figure 4.1. Demonstrating the number of available GPIO available and the bus widths for the communication between the CPU and memory components.

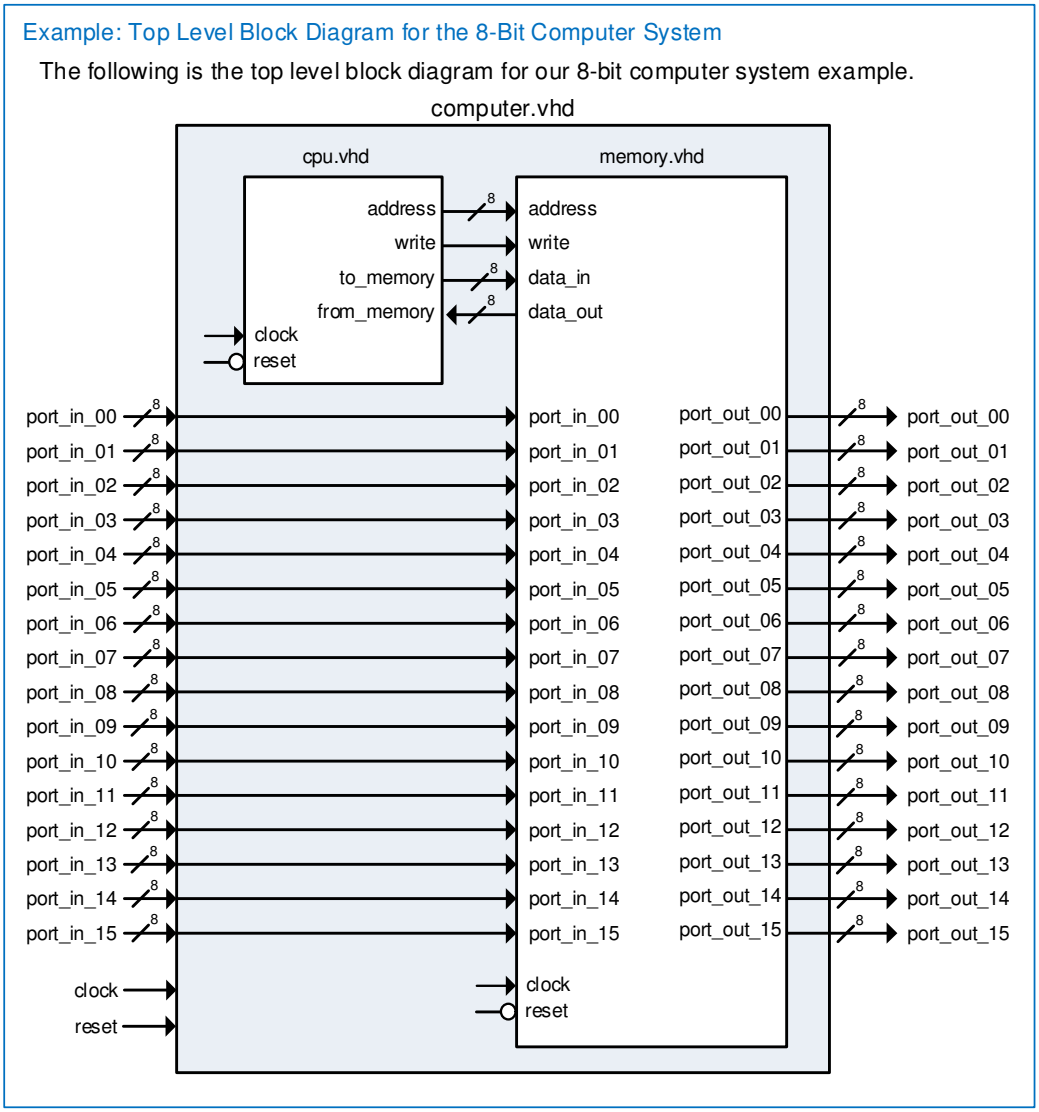

Figure 4.1: Top level view of the processor developed as part of Montana State University's EELE 367 course [11]

The memory component consisted of the program memory space, R/W memory, and I/O ports. Each section of memory is defined as its own component. Implementation of each memory component is performed through port maps in the memory component. When accessing a desired memory location, the address is used to enable the desired location. Program memory consists of the address range 0-127, R/W memory is 128-223, and the I/0 address space is 224-255. A block diagram of the memory component can be seen in Figure 4.2.

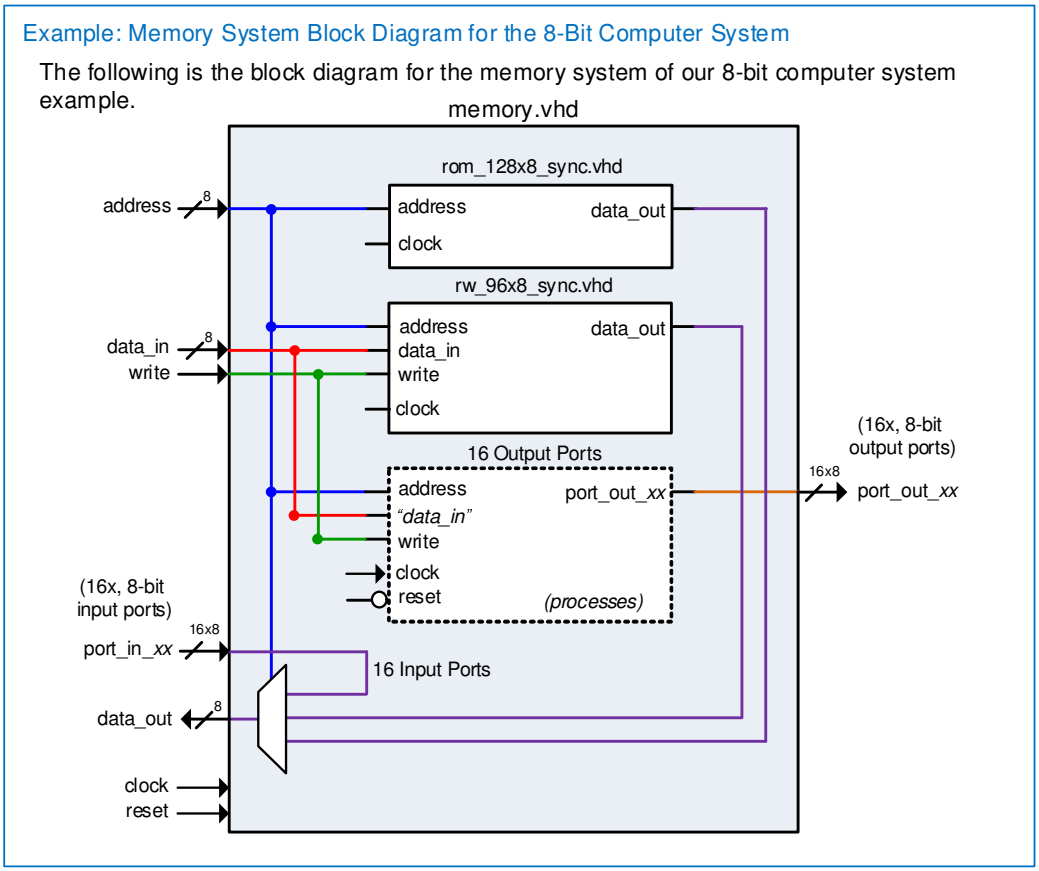

Figure 4.2: Memory block diagram [11]

The CPU component displayed in the top level view consists of three sub-components and a variety of signals. The control unit is the component that holds the state machine driving the instruction set. Any additions to the instructions set or

modifications regarding any aspect of system control are reflected in this component. Interfaced with the control unit is the data path. As the name implies, the data path is utilized to handle data. This is where core CPU registers are located such as the program counter, instruction register, memory address register and the two data registers A and B. The signals connecting the control unit to the data path are used to manipulate these registers. Housed inside of the data path is the ALU component. The ALU is responsible for the arithmetic and logic operations requested by the control unit. Depending on the results of the ALU, the CCR register will be updated which is used in branching statements. The full block diagram can be seen in Figure 4.3.

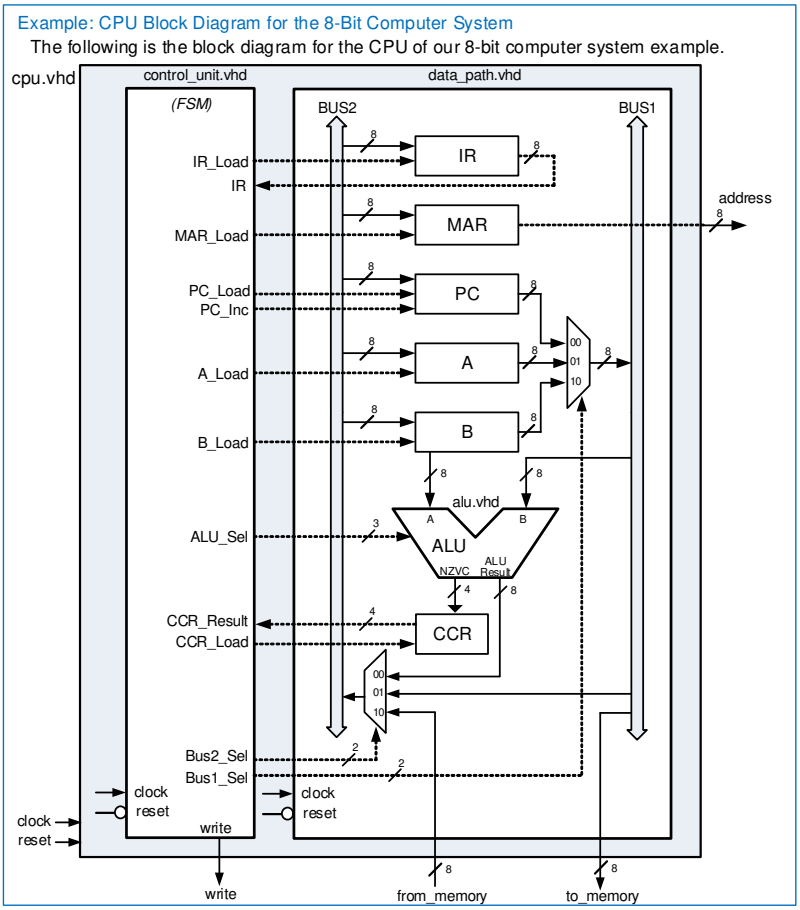

Figure 4.3: CPU block diagram [11]

This foundation was tested using a simple load and store program manually coded into the program memory. The processor would load A with xAA, store that value to xE0 (the first output register), then load the A register with xBB and again store to xE0. After the final store of xBB to xE0 the program branches back to the start of program memory and repeats.

Triplication

After verifying functionality of the processor foundation, the first step was to triplicate the existing architecture. Each core needed to be instantiated entirely independent from the others. Therefore everything seen in Figure 4.1 needed to be triplicated as well as the components internal to both the cpu and memory components.

- CPU Components
  - Control Unit
  - Data Path
  - ALU
- Memory Components
  - ROM (Program Memory)
  - RW Memory

Once each core had been created, a wrapper needed to be generated. One capable of handling the I/O for each core. At this stage every core shares the same input and output lines as seen in Figure 4.4.

Since every core is instantiated separate from the others, they have no shared memory. This requires each core to be programmed independently. During this stage of the design each processor is hard coded with the same set of opcodes, creating a homogeneous environment.

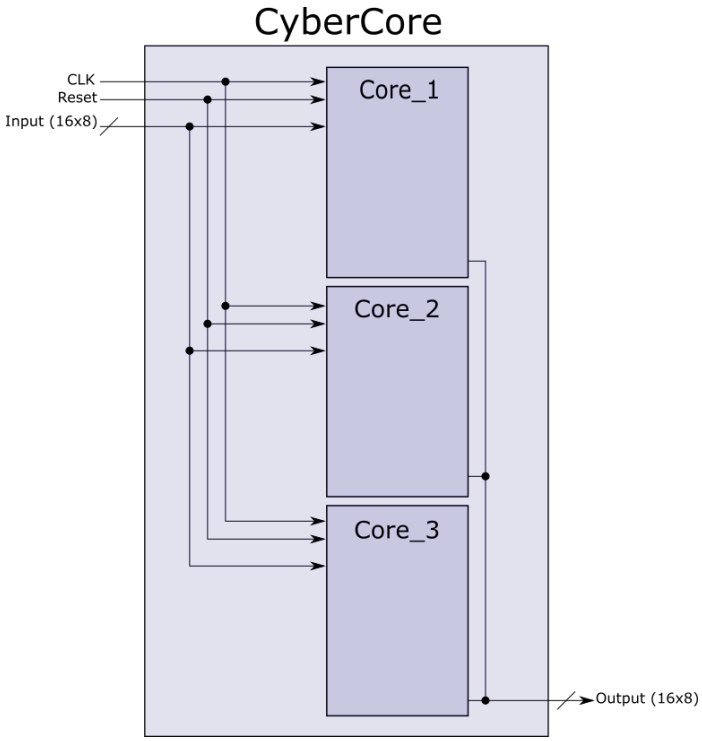

Figure 4.4: Processor triplication with shared inputs and outputs

Illegal Opcode Fault

With the basic triplication completed and tested, the next stage of the design was to implement an illegal opcode fault and voting system that will determine which processor faulted. The exception flag was created by adding a signal to the control unit of each core. This signal is either asserted or de-asserted in each state to signify if the core has received an illegal opcode.

To trigger the assertion of the exception flag a new state was created and added to the control unit after the decoding state. Once the control unit loads the instruction register with the waiting instruction the processor attempts to decode the instruction. If it matches any known opcode than the next state corresponding to the known opcode is selected and the processor continues. However, in the event

that the decoding state cannot interpret the value loaded into the instruction register, the processor proceeds into the illegal opcode fault and asserts the exception signal. Once the signal is asserted it's passed through the CPU and processor core, out to the CyberCore wrapper.

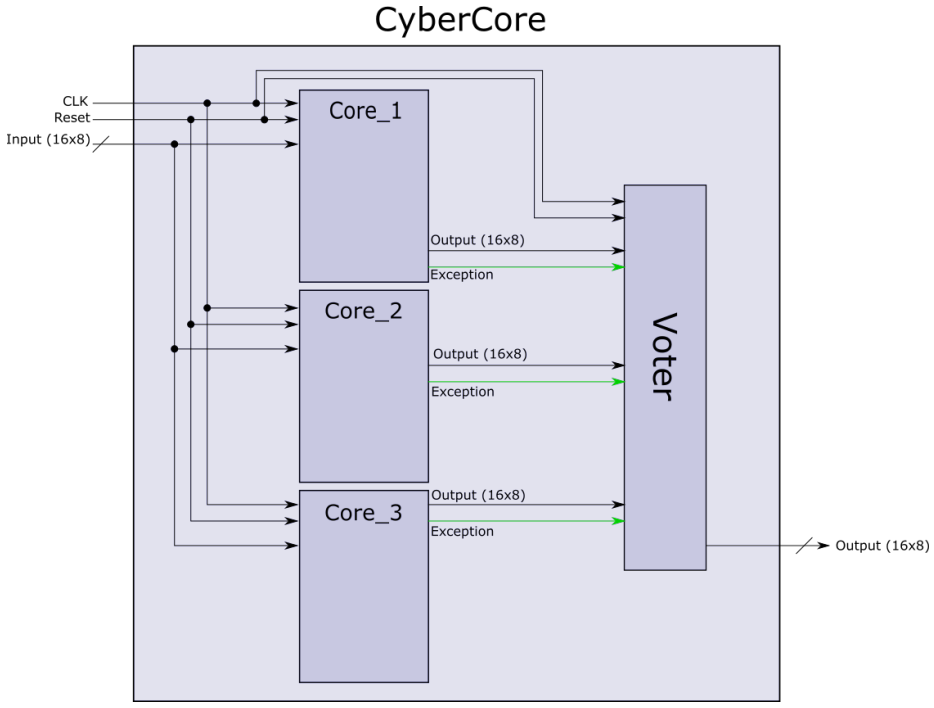

Figure 4.5: Processor triplication with exception and output voter system

During every clock cycle the CyberCore wrapper passes the newly received exception flag to the voting system. If any core has an asserted exception flag then an error code is passed back to the CyberCore wrapper, if no assertion is detected then values are passed through the voter with no interference.

Opcode Packages and Assembler

With the addition of the voting component, the system was ready for each core to instantiate different opcodes. The processor foundation utilized hard coded

opcodes in the ROM and control unit components. In order to lay the foundation for randomization this was converted to individual packages. Each core included a separate instruction code package that would replace the hard coded values. These packages are referenced in both the program memory, a component generated through the assembler, as well as the control unit of each processor. The use of instruction code packages allowed for an easier implementation of randomized opcodes in the control unit and prevented the need to generate a new control unit for every implementation. An example of the final instruction set package can be seen in Appendix Figure A. The example is complete with 27 instructions. Figure Figure 4.6 demonstrates a high level view of the developed system.

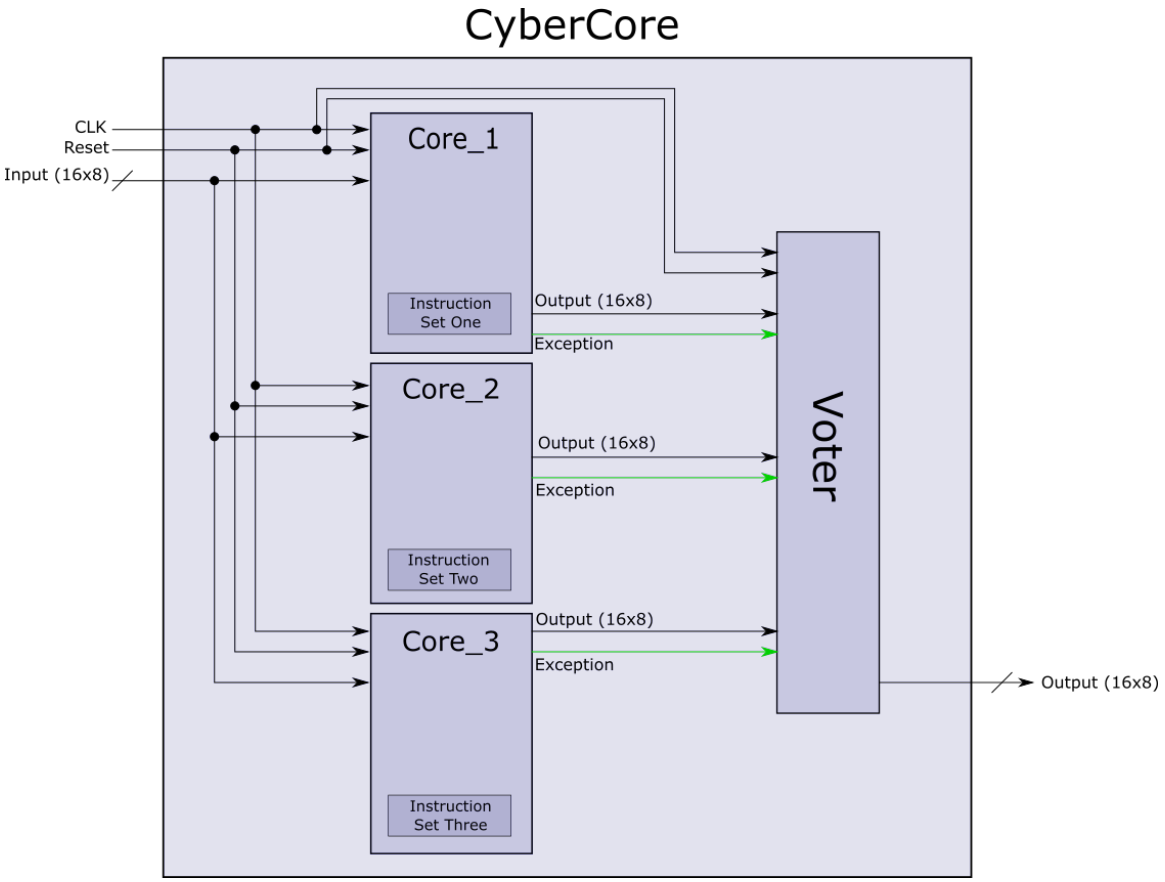

Figure 4.6: Processor triplication with independent opcodes

To generate the individual packages used for the different cores, as well as to program the controller, an assembler was built using Python. The assembler works through several steps, first it reads an asm file that is created by the user. A simple program may look like the following:

```
1  LDA_IMM -- Load A (Immediate Addressing)
2  x"AA"   -- Operand (Value)
3  STA_DIR -- Store A (Direct Addressing)
4  x"E0"   -- Operand (Memory Location)
5  LDA_IMM -- Load A (Immediate Addressing)
6  x"BB"   -- Operand (Value)
7  STA_DIR -- Store A (Direct Addressing)
8  x"E0"   -- Operand (Memory Location)
9  BRA     -- Branch Always
10 x"00    -- Operand (Memory Location)
```

It should be noted that the comments are added for readability for this paper and are not currently supported in the developed assembler. What the assembler looks for are the mnemonic representations of the opcodes and the operands. These values are read from the created asm file and parsed into a Python list structure. Once the program list is created the createProgramMemory() function is run three times, once for each core. This ensures that the same program is written for each processor core using the instruction mnemonic instead of hard coded opcode values.

In addition to generating the program memory, the assembler also generates the opcode packages mentioned previously. This stage is done using the createInstructionFile() function. By utilizing the python package "random" it's possible to create pseudo-random values for the opcode definitions.

To ensure that there is no overlap between the opcodes, ie. no two opcodes have the same value, the values are created using the random.sample(range(16,225)27) function call. This creates a list of 27 random values between 16 and 225. After creating the values it's necessary to translate them into a hex value that can be used to easily generate a vhdl package. The built-in python package "hex" allows for easy conversion between the randomly generated decimal value, and a hex value. From there, list manipulation is leveraged to put the opcode values into the desired format.

The full assembler program can be seen in Appendix Figure B.

Preliminary Testing

At this stage it was prudent to test the developed architecture to verify initial functionality with the limited instruction set currently implemented. Before testing the system under attack it was necessary to ensure normal system functionality. To do this, a program was written that would load alternating values into A and store them into a desired memory location. The initial testing can be seen in Figure 4.7. Demonstrated is the ability for the three functionally identical processor cores to function using independent sets of instruction codes.

After verifying the ability to run while not under attack, an attack was manually inserted into the system to verify attack response. Testing can be seen in Figure 4.8. As soon as the manually injected code attempts to execute, the system halts two of the cores and prevents the output from being modified.

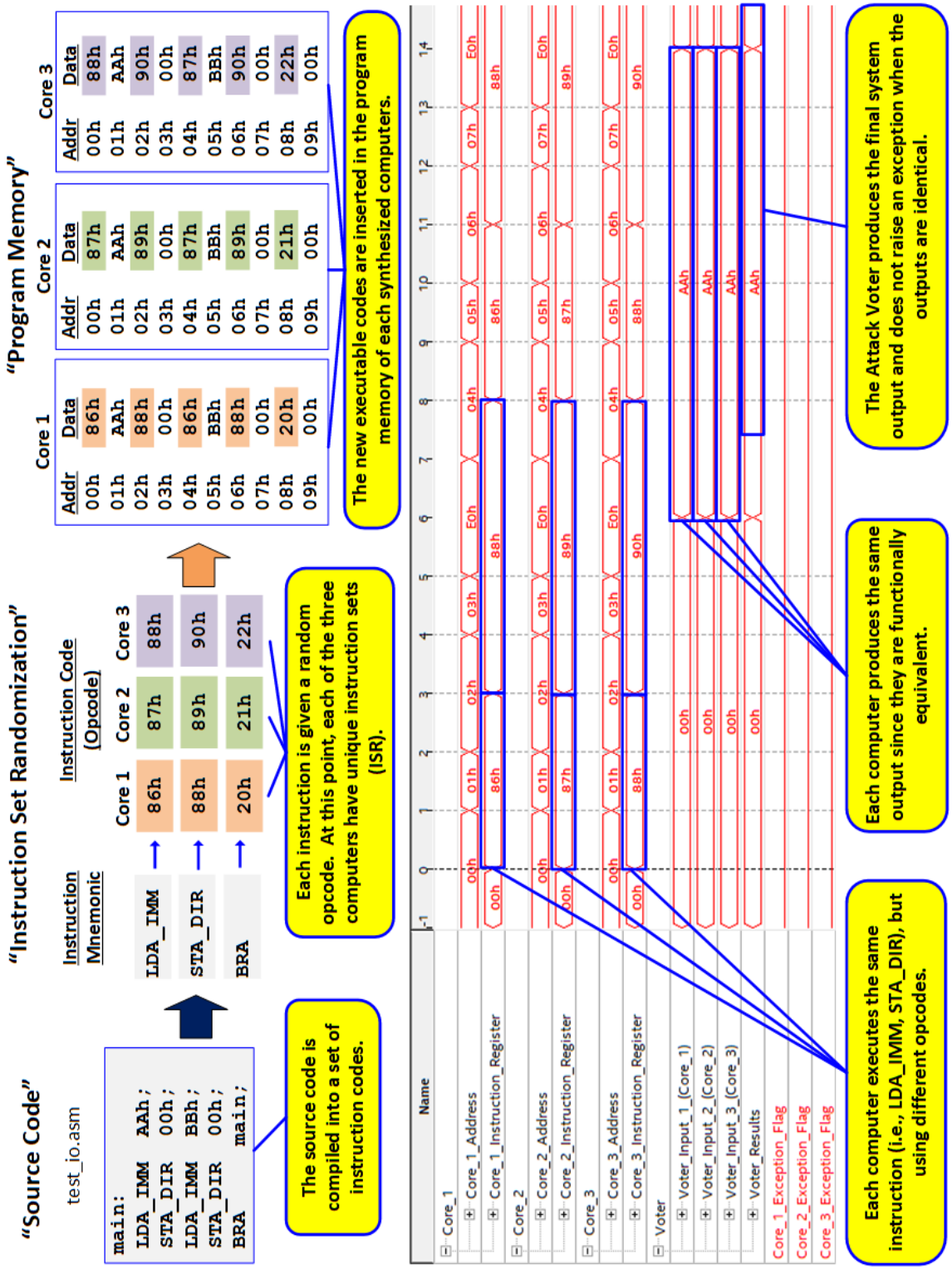

Figure 4.7: Initial testing of system. This figure demonstrates operation in the absence of attack.

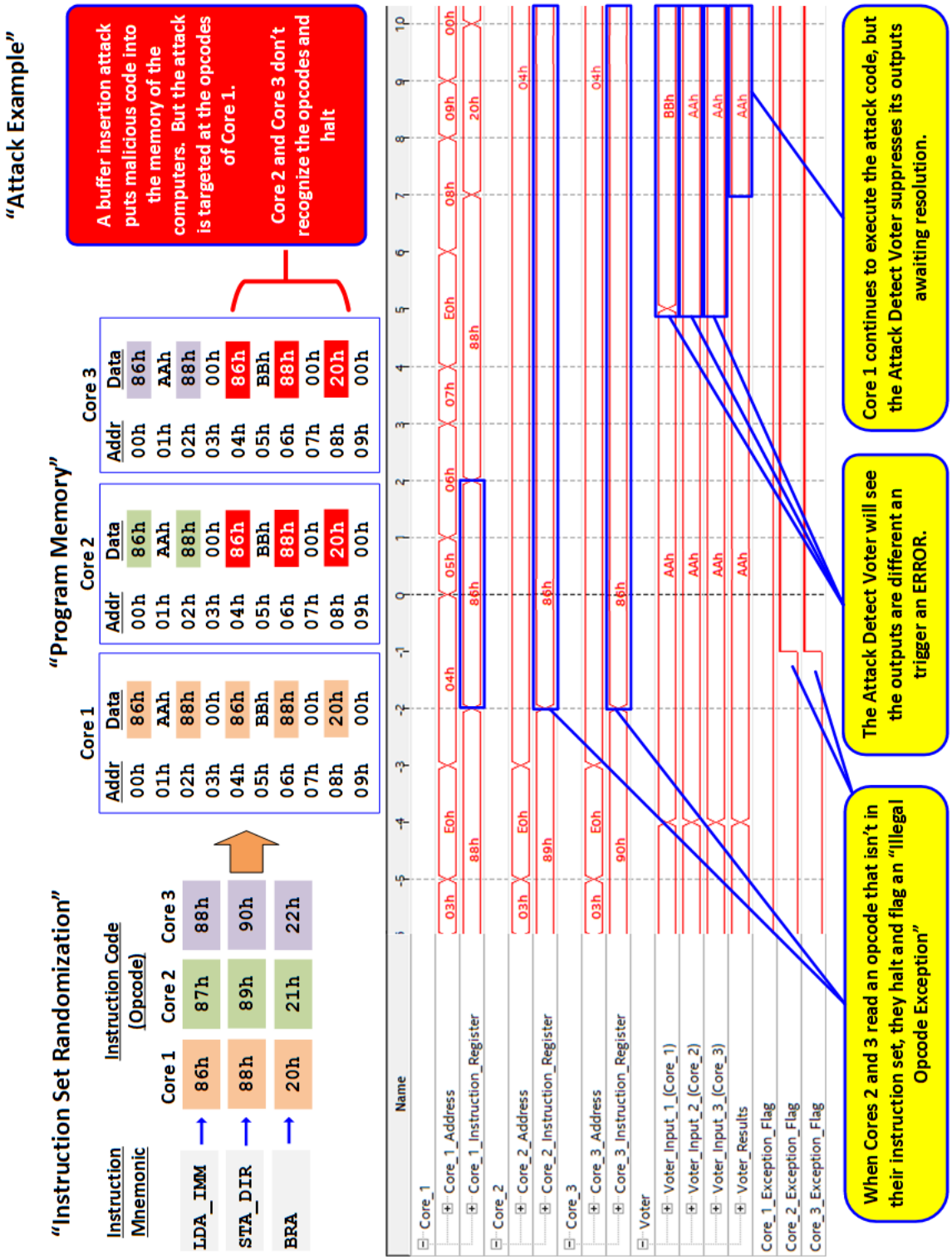

Figure 4.8: Initial testing of system. This figure demonstrates operation while under attack.

Processor Expansion

After testing the preliminary proof of concept it was necessary to prove that the system would work in a more robust system with an injected attack. Consideration was given to multiple different processors including the Xilinx Picoblaze, the Leon3 and the OpenRisc 1200.

First examined was the Leon3 processor from Cobham Gaisler. The Leon3 processor was designed by the European Space agency and has full Linux functionality. However, while the processor's opcodes can be seen in its packaged sparc.vhd file it became an issue of fabric space. For most purposes the Leon3 allows for multi-core functionality to be generated using its built in tools. Using this method, it's only possible to create a triple core processor with a single set of instructions, rather than a triple core processor with triplicated instruction sets. To triplicate the processor, with entirely independent memory and periphals, would exceed the fabric space limit on the desired development board.

Having ruled out the use of the Leon3 processor due to space limitations, it was decided to examine the Picoblaze softprocessor developed by Xilinx. The Picoblaze is a simple 8 bit microcontroller designed to take up very little space in fabric, as little as 26 slices depending on device family.

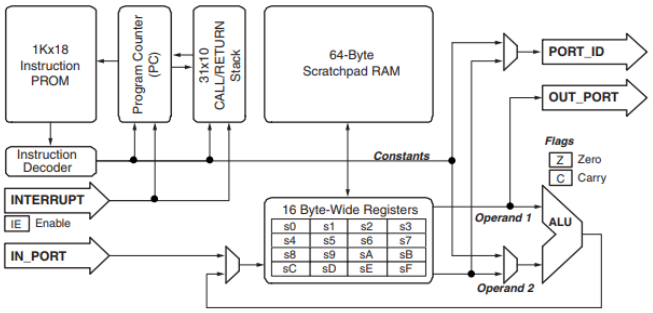

Figure 4.9: Picoblaze block diagram [28]

While the Picoblaze would easily fit into the FPGA fabric, it became apparent that there was no suitable way to modify the instruction decoder. Without being able to modify the instruction decoder it wasn't possible to implement three different sets of instructions that are required for the project. Every implementation of the Picoblaze would be forced to interpret every instruction set the same.

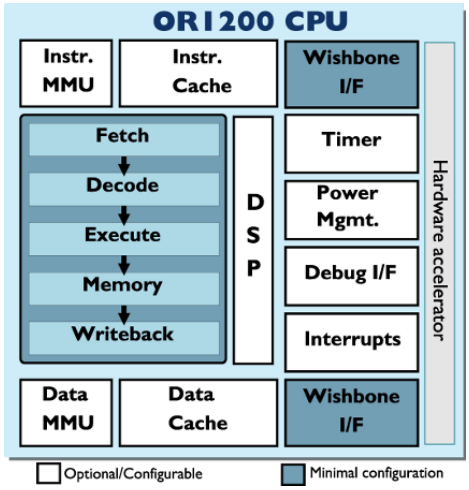

Figure 4.10: Block diagram of OpenRISC 1200 processor [5]

The last open source processor examined was ruled out early on in evaluation. OpenRisc 1200 is written in Verilog. My ability to use Verilog is much less than my ability to use VHDL. While the processor itself may fit well for the rest of the project, the difficulty associated with integrating a new language into the project precluded selection of this processor. However, it can be seen in Figure 4.10 that the layout of the processor is very similar to that of the Leon3 processor. Noticeably different is the use of the Wishbone I/F rather than the AMBA AHB interfaces.

After ruling out the existing available architectures, it was decided to flesh out the existing implementation to demonstrate product viability. Examining the block diagram of the Picoblaze, seen in Figure 4.9, shows that the foundation 367 processor is missing only a couple components to be considered as a viable processor.

Predominantly the following.

- Stack
- PUSH/PULL
- Expanded instruction set
- Interrupts

By adding these components to the foundation processor it would be possible to demonstrate viability.

Stack

Implementing a stack into the system is relatively straightforward. The concept of a stack is a memory structure that is considered to be first in last out (FILO). Meaning that whatever is added the the stack first is going to be the last thing that can be removed.

First, a new VHDL component was created mimicking the current R/W memory structure. The two memory structures are very similar, the stack will also need to have the capability to both read from and write to. To account for this additional memory structure within the available memory space, 256 bytes total, the two other memory components were shrunk. Program memory was decreased from 128 bytes to 96 bytes, R/W memory was reduced from 96 bytes to 72, leaving 56 bytes of address space available for the stack.

Now that the memory structure of the stack was built, the next step was to build the stack pointer register in the data path component. To enable communication and control between the control unit and the data path, three signals were created. SP_Enable, SP_Inc, and SP_Dec. SP_Enable allows for interaction with the stack and the stack pointer. SP_Inc and SP_Dec are used to increment and decrement the stack pointer. The full stack pointer control system implemented in the data path can be

seen in the following code.

```
STACK_POINTER : process (clock, reset)
  begin
    if (reset = '1') then
      SP_uns <= x"C8";
    elsif(clock'event and clock = '1') then
      if (SP_Enable = '1') then
        if(SP_Inc = '1') then
          SP_uns <= SP_uns +1;
        elsif(SP_Dec = '1') then
          SP_uns <= SP_uns -1;
        end if;
      end if;
    end if;
  end process;
  SP <= std_logic_vector(SP_uns);
```

On the rising edge of every clock cycle the control system checks to verify if the stack pointer is enabled. If it is then it checks to see if the value needs to be incremented or decremented and performs the corresponding adjustment. For ease of implementation, the stack pointer value is created as an unsigned type and then assigned to SP.

Push/Pull

To test the stack implementation six new instructions were added to processor. The first three are designed to push values from core registers to the stack memory while the last three will recover data from the stack and place into the core registers.

- PUSH_A
- PUSH_B
- PUSH_PC
- PULL_A
- PULL_B
- PULL_PC

These instructions were selected both for the ability to test the stack, and the fact the fact the functionality was required for interrupts.

Push

The first instruction implemented was PUSH_A which simply pushes the value of the A register to the stack. Implementation of a push instruction required two additional states. In this case S_PSH_A_4 and S_PSH_A_5. The two states can be seen in Figure 4.11.

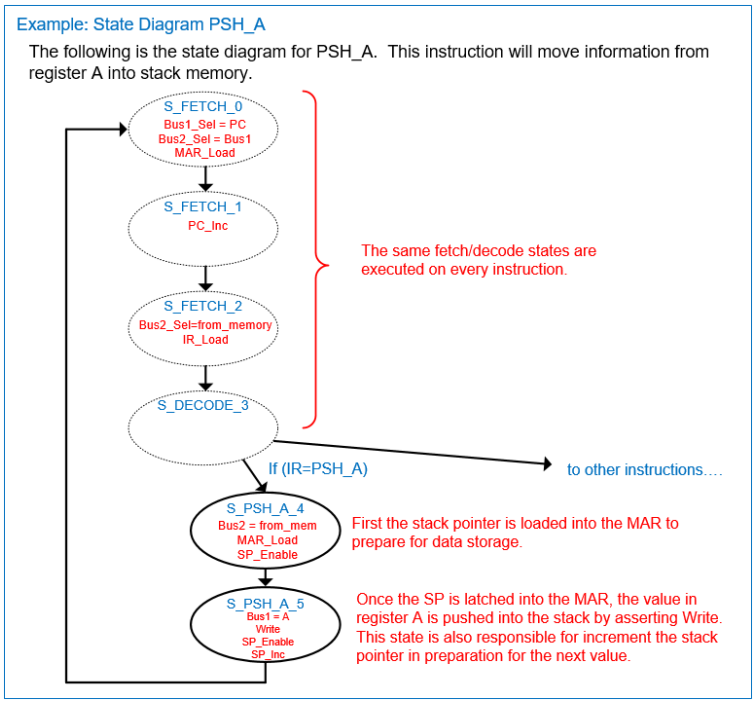

Figure 4.11: PSH_A State Diagram

The first added state is used to load the value of the stack pointer into the memory address register (MAR). To do so the MAR_Load signal is asserted, and SP_Enable is asserted. With the stack pointer address loaded into the memory address, a store is ready to occur. Bus1_Sel is loaded with 01 to indicate that we are storing the value in register A, and the write bit is asserted. Now that the value is stored into the stack the final thing to do is to increment the stack pointer to prepare for the next write. Asserting both SP_Inc and SP_Enable at the same time fulfills this requirement.

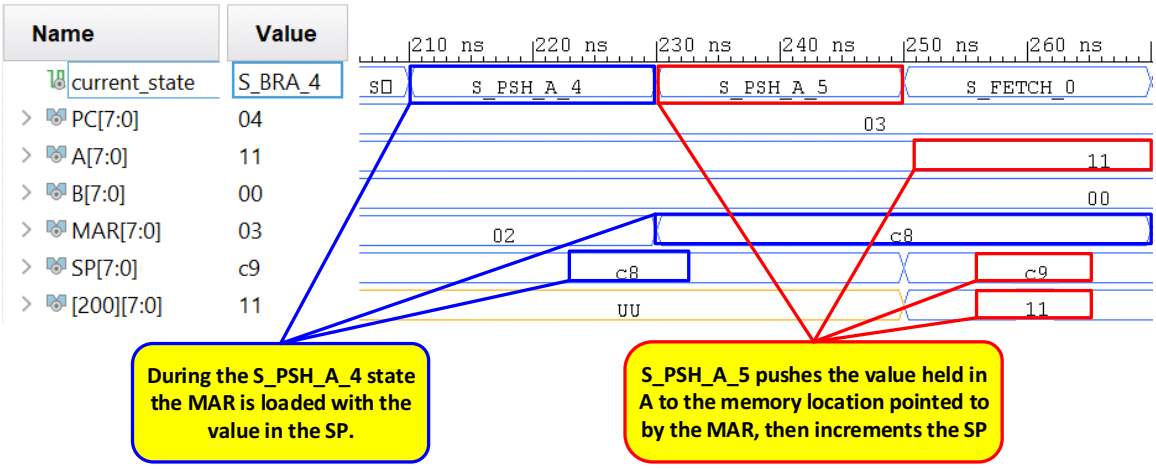

Figure 4.12: Push_A simulation

Verification of the push instruction was conducted using Vivado's built in simulation tool. Results can be seen in Figure 4.12. It can be seen that the value in A, x11, is stored into memory location 200 (xC8 as seen in the SP) at the end of the S_PSH_A_5 state. At this same time the stack pointer is incremented from xC8 to xC9 in preparation for the next interaction. This demonstrates that the stack memory is functional as well as the ability to push the contents of A to the stack.

After verifying that the functionality is there for A, functionality needed to be added to push both register B, and the contents of the program counter (PC) to the

stack. Doing so is very similar to pushing A to the stack with one small change. To push B, Bus1_Sel needs to be loaded with 10 and to push the program counter to the stack, Bus1_Sel needs to be loaded with 00.

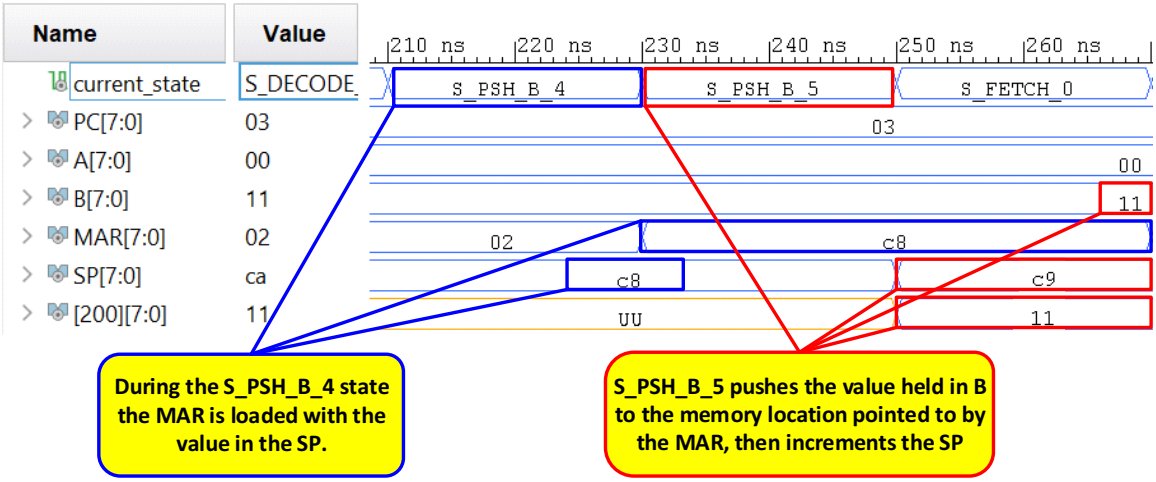

Figure 4.13: Push_B Simulation

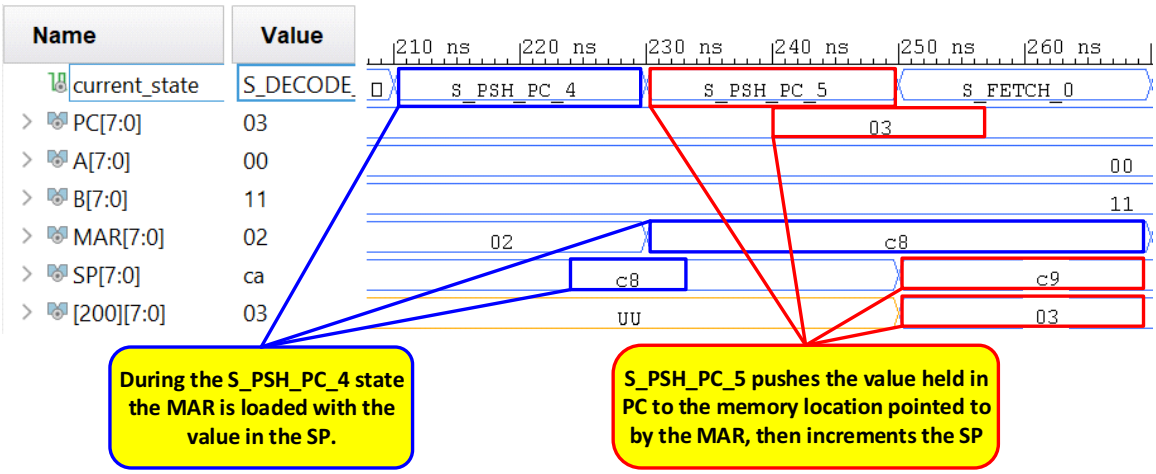

Figure 4.14: Push_PC Simulation

Verification of these instructions was done in the same fashion as PSH_A, and can be seen in Figure 4.13 and Figure 4.14. In the B simulation it can be clearly seen that the value held in the B register, x11, is loaded into memory location 200 pointed

to by the stack pointer. The same can be seen in the PC simulation where x03 is pushed into the stack.

Pull

With the ability to push to the stack implemented, I moved onto adding the functionality to pull from the stack. Similar to the pushing functionality, multiple states needed to be added to the control unit to implement this functionality. However, unlike pushing where only two additional states per register were required, pulling required four additional states per register. Seen in Figure 4.15 are the four added states to add PLL_A; S_PLL_A_4, S_PLL_A_5, S_PLL_A_6, and S_PLL_A_7. First, the stack pointer is decremented to point to the first location in the stack that holds a value. This is done by asserting both the SP_Enable and SP_Dec signals. The next state loads the memory address register with the new stack pointer value. This is done using the same combination of signals as when pushing a value to the stack. Our third state "chews up a clock cycle" to ensure that all values are appropriately latched and the final state loads the value into register A. The final state is accomplished by asserting the load signal for the desired destination register and loading Bus2_Sel with 10 to indicate that the value is coming from the memory. In this case the desired destination register is A so A_Load is asserted. The full state configuration for these four states is as follows:

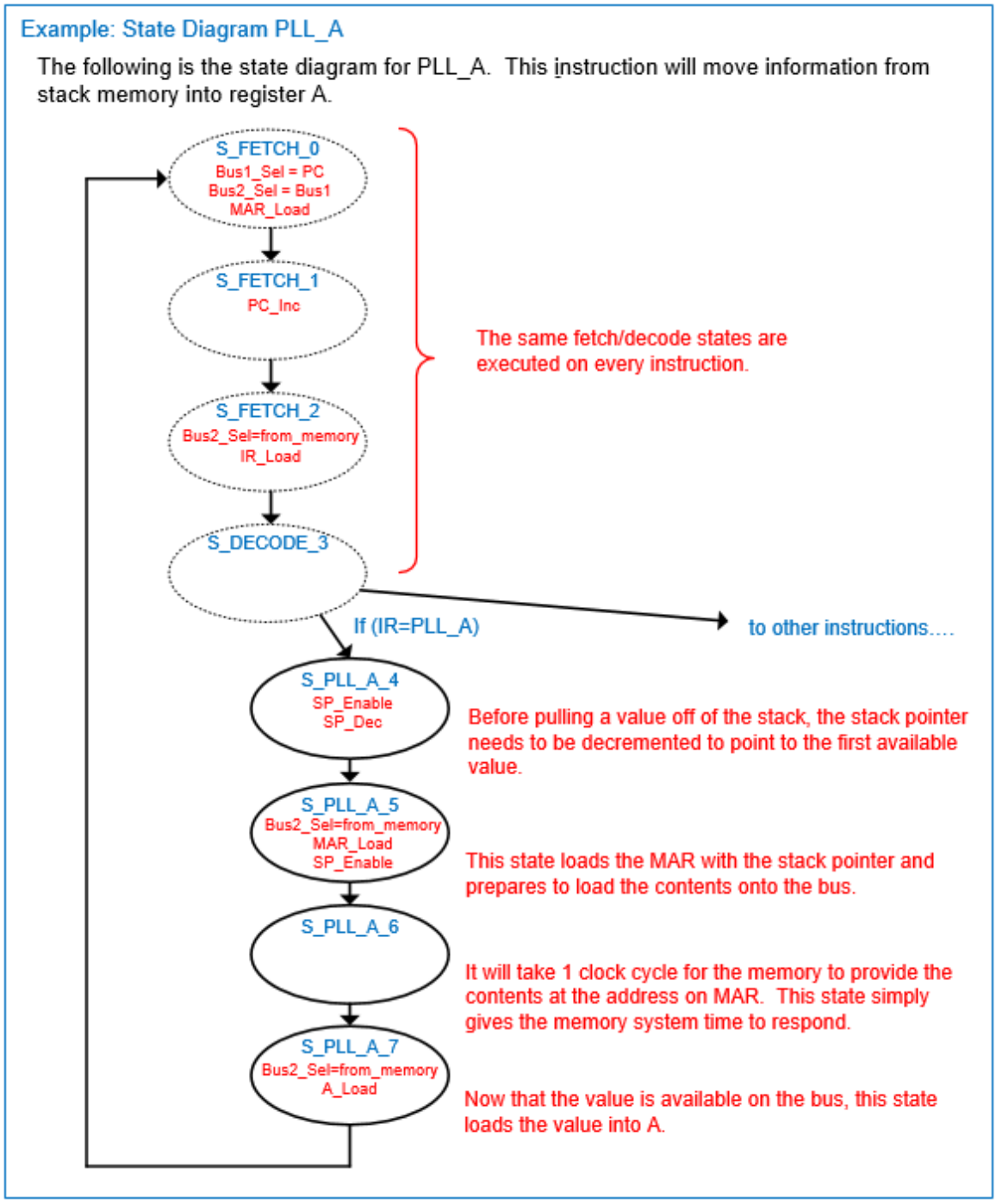

Figure 4.15: PLL_A State Diagram

Verification of the instruction functionality was performed via simulation. PLL_A simulation can be seen in Figure 4.16. The functionality described above can be seen in the provided figure. At the end of the first pull state the stack pointer is decremented from xC9 to xC8.

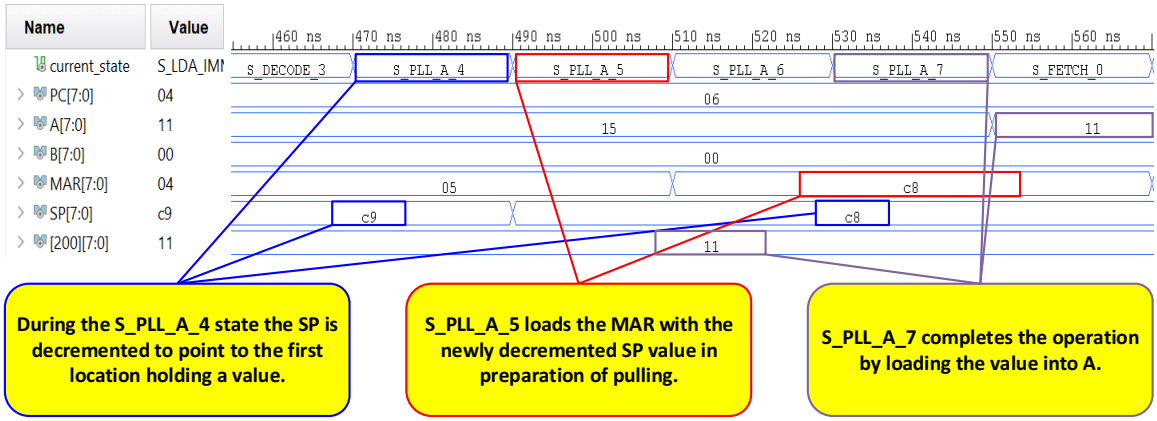

Figure 4.16: Pull_A Simulation

After the second state the memory address register is loaded with the newly decremented stack pointer. Nothing is modified during the third pull state and finally during the fourth pull state it can be seen that the value stored in xC8 is loaded into the A register as desired.

Once the instruction was implemented for pulling from the stack to the A register, it was necessary to implement the same instruction format for both B and the program counter. Four additional states were required each to allow for instruction implementation. The only differences between these states and the states described above for pulling to the A register is what load signal is asserted. To pull to the B register, B_Load is asserted and to load to the program counter PC_Load is asserted. Simulation of these two instructions can be seen verified in Figure 4.17 and Figure 4.18 respectively.

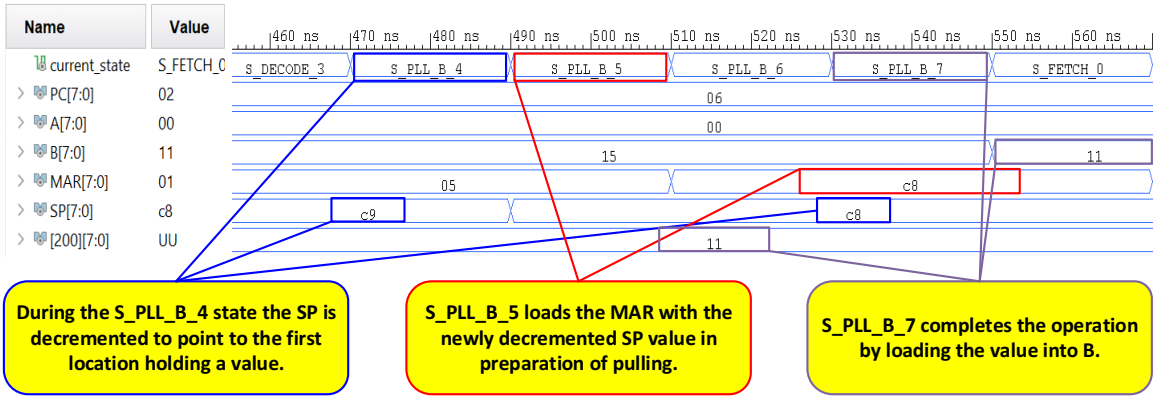

Figure 4.17: Pull_B Simulation

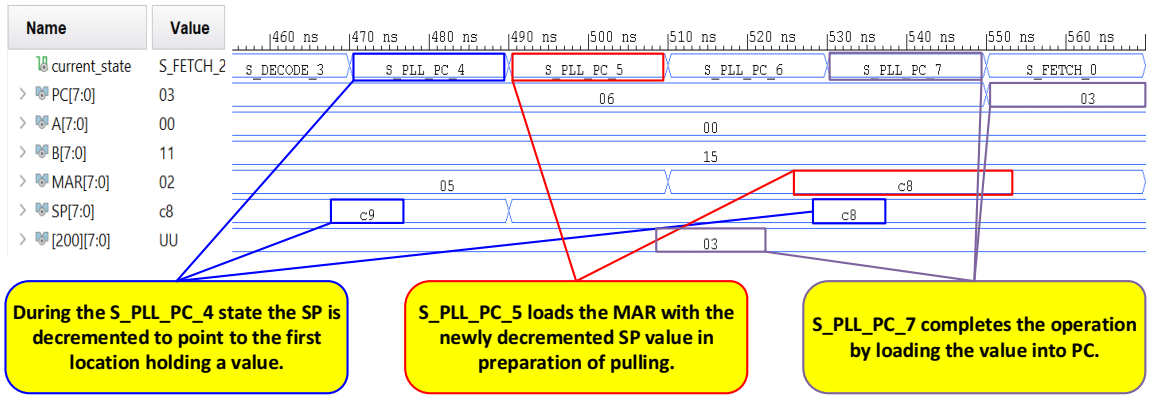

Figure 4.18: Pull_PC Simulation

Interrupts and Faults

Having added a stack and the ability to push and pull to and from said stack, the next step was to create an interrupt system. Before anything else, an interrupt system requires a method to tell the control unit that an interrupt is being requested. In this case an interrupt register was created. The created interrupt register is 4 bits wide to allow for the addition of future interrupt vectors. With this in mind, only a single interrupt vector is created with full functionality while a second is included in the data path to test proof of concept. Additionally a fault detection system was implemented to account for any internal faults and fault handling that may be

required for future implementations.

With the interrupt signal being created a system to handle it needed to be instantiated. To do so, a total of 18 states were created to handle the start of interrupt and return from interrupt procedures. During every S_Fetch_0 state, a check is made to see if either the interrupt flag or the fault flags are asserted. In the event the interrupt signal is asserted, the processor is prevented from going to S_Fetch_1 instead routing to S_STI_4 which is the first state in the "Start of Interrupt" procedure chain. States S_STI_4 through S_STI_9 are used to save the processor state to the stack. Processor state is preserved by pushing the PC, register B and register A to the stack, in that order. After preserving the processor state, the state machine in the control unit transitions to S_LD_INT_VEC_4. This state is used to decode the desired interrupt.

Example: State Diagram STI

The following is the state diagram for STI. STI is a state chain entered when an interrupt is detected and is used to preserve the processor state before executing an interrupt.

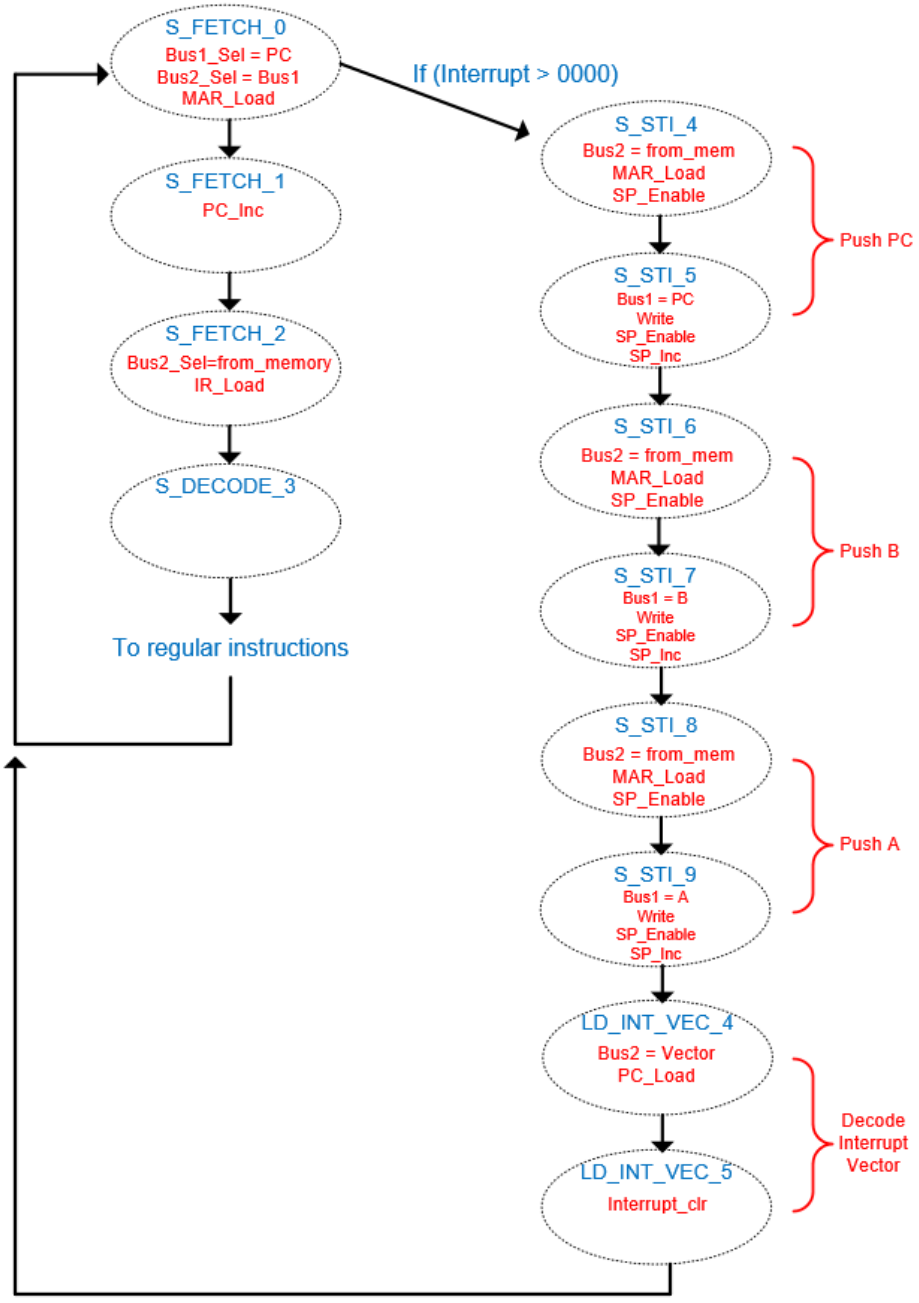

Figure 4.19: STI State Diagram

Interrupt decoding was achieved by adding a fourth option to the Bus2_Sel multiplexer system. If Bus2_Sel is loaded with 11 it informs the data path that an interrupt is being triggered and to decode the desired vector memory location. The vector decoding process can be seen in the following code:

```
INTERRUPT_VECTOR0 : process (interrupt)
  begin
    case (interrupt) is
      when "0001" => Interrupt_Vector <= x"78";
      when "0010" => Interrupt_Vector <= x"52";
      when others => Interrupt_Vector <= x"00";
    end case;
  end process;
```

The interrupt value is an input that traces all the way to the top level module while the Interrupt_Vector is a signal internal to the data path. As soon as an external interrupt is triggered, the data path will decode the interrupt and load the Interrupt_Vector signal with the memory location of where the interrupt subroutine resides. However, it isn't until the control unit signals that it's ready to load this value into the program counter that the value is passed along. This can be seen in the Bus2 multiplexer located in the data path component seen in the following code segment.

```
MUX_BUS2 : process (Bus2_Sel, ALU_Result, Bus1, from_memory)
  begin
    case (Bus2_Sel) is
      when "00" => Bus2 <= ALU_Result;
      when "01" => Bus2 <= Bus1;
      when "10" => Bus2 <= from_memory;
      when "11" =>
        if(illegal_op = '1') then
          Bus2 <= Fault_Vector;
        else
          Bus2 <= Interrupt_Vector;
        end if;
      when others => Bus2 <= x"00";
    end case;
  end process;
```

Once the memory location is loaded into the program counter, the interrupt subroutine is ready to be executed. It should be noted that currently the only way to handle a fault is through the use of the internal illegal_op flag. The rest of the fault handling system was temporarily disabled to prove that the processors will halt when an illegal opcode is detected. By uncommenting a few sections of the control unit state machine it's possible to enable fault handling that will recover from an illegal op code. However, this will only work with a single core as a synchronization method needs to be implemented to stall the other two cores until the fault handling is complete.

The next state, S_LD_INT_VEC_5, is the last state in the chain that initiates an interrupt. This state is utilized to clear the interrupt flag. An intterupt_clr signal

is asserted that is sent to the component that triggered the interrupt to begin with. Upon receiving this signal the interrupt vector is reset to 0000 to indicate that the interrupt has been acknowledged. Acknowledging the interrupt is the last step needed to prevent the control unit from infinitely triggering interrupts.

In another version of this processor a secondary flag called "internal_interrupt" is asserted that allows for the control unit to internally acknowledge the interrupt without externally clearing the flag. To clear the interrupt flag in this version of the processor, a CLI instruction was required. This method prevented the ability to stack interrupts and was thus removed in the final iteration of the processor.

Returning from the interrupt is achieved through an RTI instruction. When RTI is loaded into the instruction register it begins a 12 state chain. This chain restores the previous processor state from the stack. Since the processor state is stored in the order of PC, B, A, the processor state is restored in the order A, B, PC.

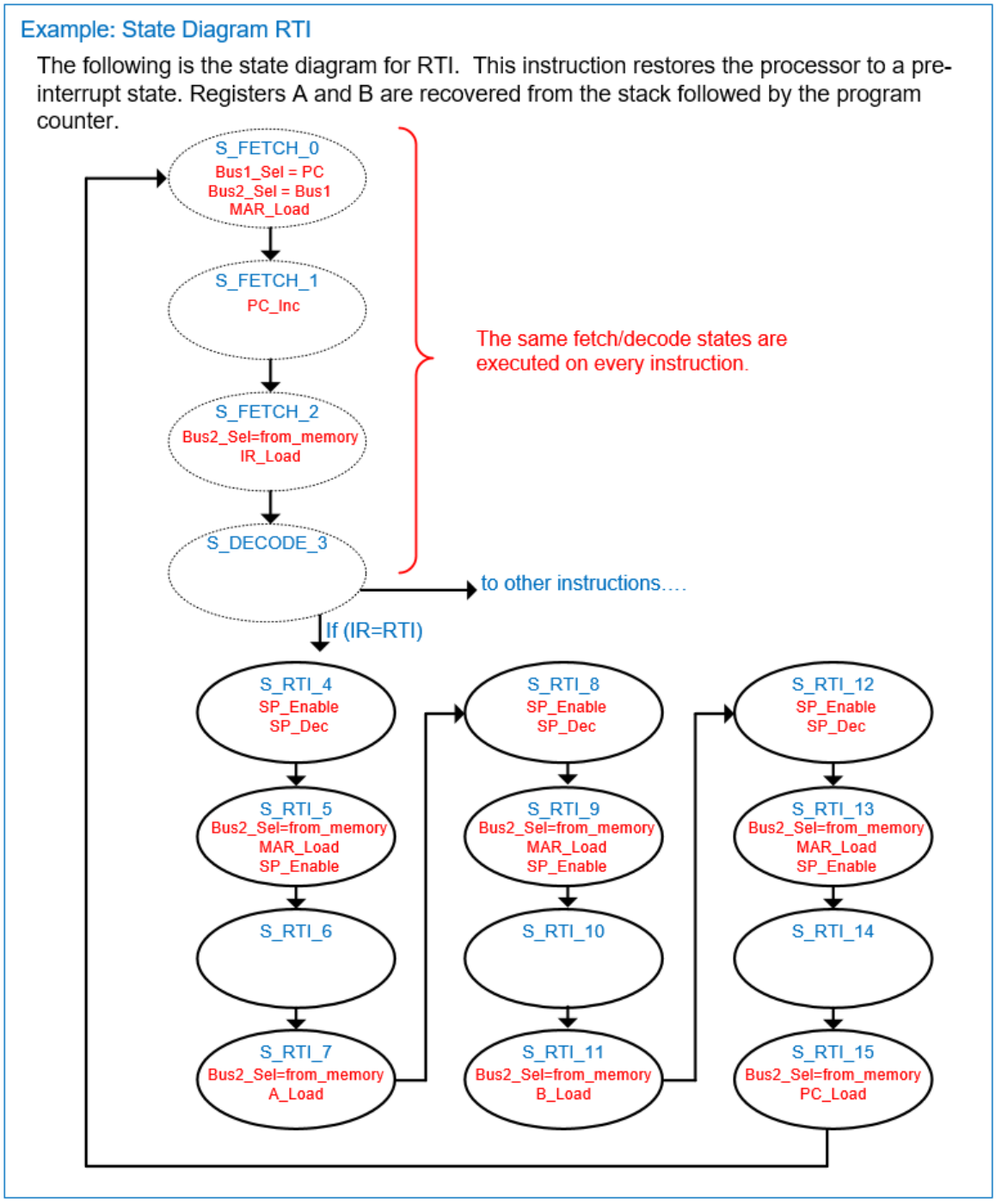

Figure 4.20: RTI State Diagram

Testing the use of the STI and RTI state chains was performed through simulation. An internal counter was built 6 bits wide, allowing for a maximum value of 63. When the counter register is full, the interrupt flag is triggered. After assertion of the interrupt flag it's necessary to wait for the current instruction to finish executing before checking for the interrupt in S_Fetch_0. This prevents any operation from being halted before completion. After arriving at S_Fetch_0 the interrupt is detected and the state machine is routed to the STI state chain.

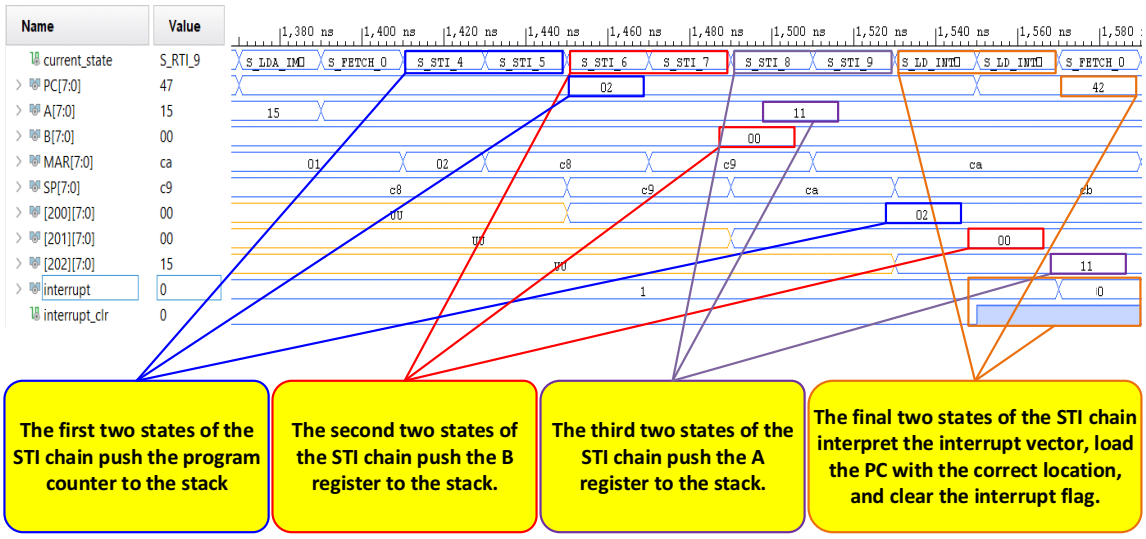

Figure 4.21: STI state chain simulation.

As shown, when the STI chain is initiated, the processor state is preserved by pushing the program counter, register B, and register A to the stack, in that order. If nothing else is held on the stack, the final stack pointer value should hold xCB as the next available location. Once the processor state is preserved, the interrupt vector is decoded and the program counter is loaded with the vector location. In this case the vector is located in memory location x42 as seen after executing the last state in orange.

After execution of the interrupt subroutine, the processor needs to return to the previous program counter location to continue executing. Testing of the RTI system can be seen in Figure 4.22.

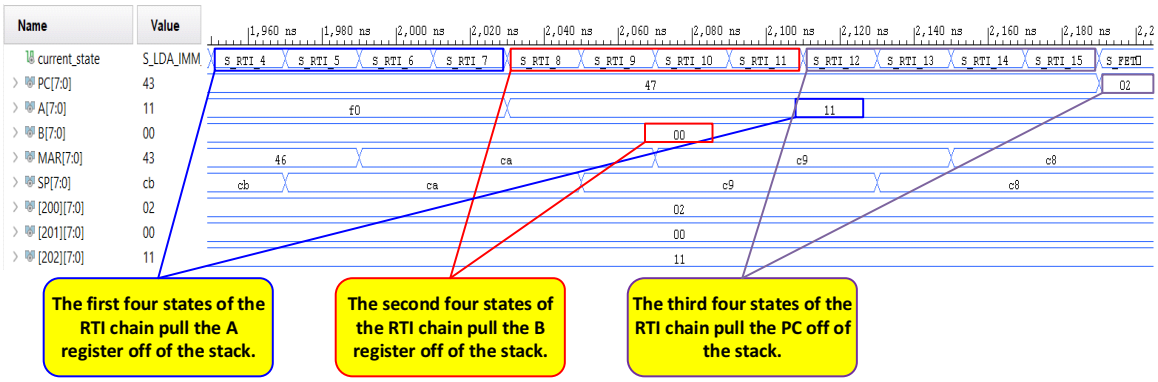

Figure 4.22: RTI state chain simulation.

Displayed is the recovery of the saved processor state from the stack. First, the value in register A is recovered, then the value from register B and finally the program counter is restored. As each value is recovered from the stack, the stack pointer is decremented to indicate the newly available top location of the stack. When the entire process is finished the stack pointer is returned to the initial starting value of xC8. After the state chain is fully completed, the process is restarted at S_Fetch_0 to begin loading in the original program starting from where the program counter left off at interrupt.

UART

With the completion of an interrupt capable system, a method of triggering the interrupt needed to be added. While something such as a timing interrupt may have been easier to implement, it was decided to implement a UART interrupt to allow for code to be injected over serial.

UART communication was added by creating two separate components. One for receiving and one for transmitting. Both components have been instantiated and tested through simulation, however, only one side of the communication system is actually used in the final product. Since the project scope is to demonstrate the response to a code injection attack, the receive system is the side of the communication that is fully actualized and utilized in the final product. It should be noted that the transmit component is still instantiated in the final product and would only need a correct port map to be fully usable.

Serial communication settings can be seen in Figure 4.23. The standard baudrate is set to 115200 but can be modified through the use of generics in the UART component. A combination of the value assigned to the generics and the speed of the clock will determine the baudrate that is required.

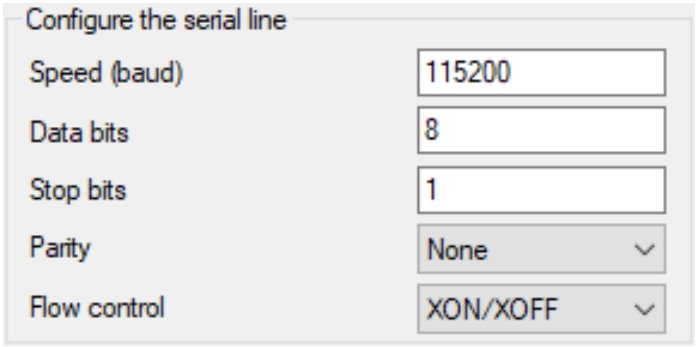

Figure 4.23: Serial communications settings for processor

Baudrate can be calculated using the following equation.

$$Clks\_per\_bit = ClockFrequency/Baudrate$$

for example, using a 10 MHz clock and a desired baudrate of 115200 we see:

$$Clks\_per\_bit = 10MHz/115200 = 87$$

Therefore, in the test cases done, the UART component was instantiated at 10 MHz, the baudrate as seen in Figure 4.23 is set to 115200 and the instantiation of the component sets the generic value to 87 as shown in the following code snippet.

```
uart_receive : uart_rx
  generic map (
    clks_per_bit => 87
    )
  port map (
    clk       => uart_clk,
    rx_serial => Rx,
    rx_dv     => rx_dv_sig,
    rx_byte   => rx_byte_val
    );
```

Simulation of the UART system demonstrated a usable peripheral by implementing a loop-back communication system. The system receives a simulated transmission of a value which is then immediately transmitted back out, if the same value is transmitted that was received then the test was considered to be a success. However, final testing was done by adding a circular serial buffer.

Serial Buffer

The serial buffer allows for 8 bytes of data to be stored before the pointer is reassigned to the beginning of the buffer. This is achieved by using a case statement that handles a buffer memory pointer as well as signaling when the buffer is full. A buff_ready signal is used to inform the top level component that the buffer is ready, which in turn triggers an interrupt.

The buffer process is sensitive to the falling edge of the UART receive dv signal. Rx_dv_sig is a signal component of the UART receive component that is asserted when data is being received and de-asserted after the stop bit. As soon as the flag is cleared the received byte is stored into the buffer and the buffer address is incremented. An empty buffer can be seen in Figure 4.24.

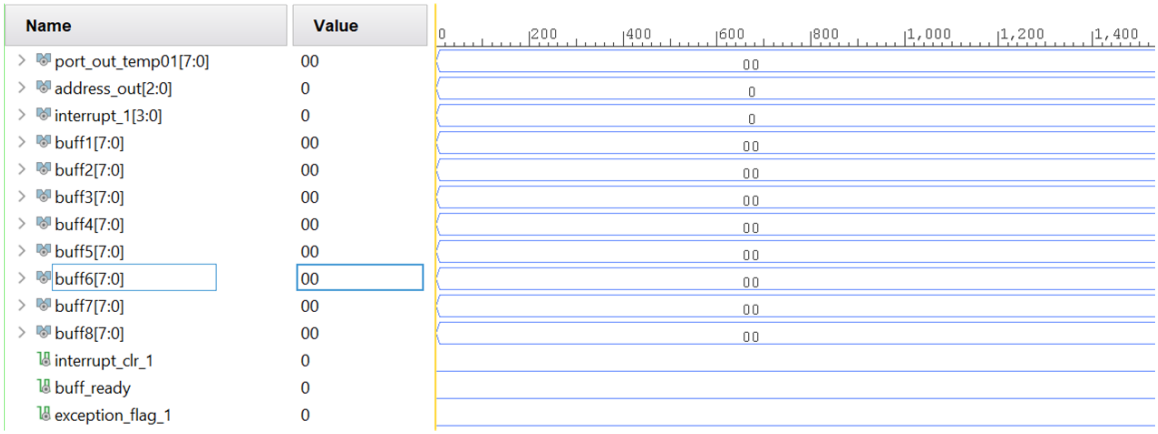

Figure 4.24: Empty serial buffer

As the values are sent across to the processor, the buffer begins to fill up as seen in Figure 4.25. In this figure it can be seen that the value held in buff1 has changed to x30. Additionally the address_out value has been incremented to 1, indicating that the first buffer address has been filled.

This process will repeat until the buffer is full. As soon as the buffer is full a signal is asserted to indicate that the buffer is ready to be read from. Implementation of the full buffer can be seen in Figure 4.26. Once the value in buff7 has been stored the buff_ready flag is asserted. In turn, on the rising edge of buff_ready, the interrupt is triggered.

Final single-core testing utilized the entire buffer as a method of code injection. To inject the code, the interrupt vector was assigned the value of x78 which is configured as the bottom 8 bytes of GPIO input. Addresses x78-x7F are now

connected to the serial buffer in parallel, allowing for 8 bytes of code to be injected into the processor. Injected code can be seen in Figure 4.26

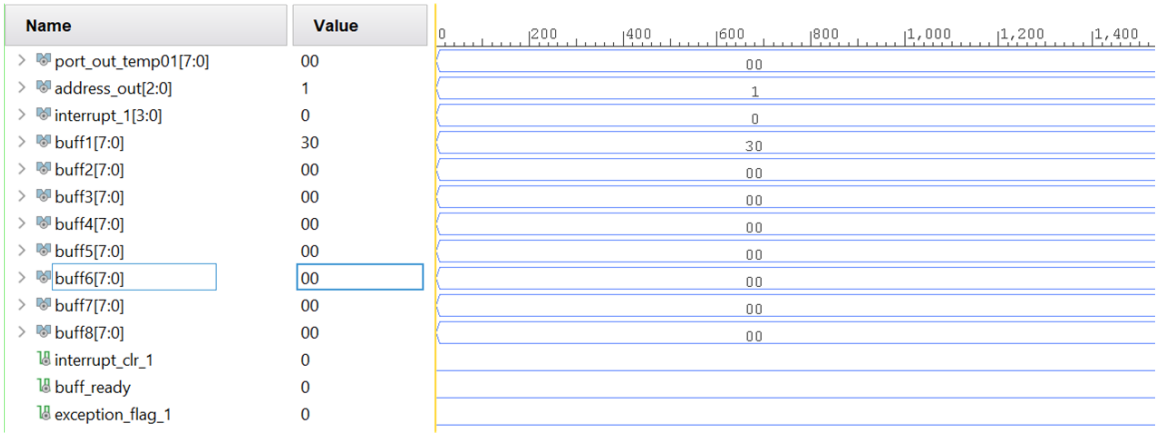

Figure 4.25: Serial buffer holding first value

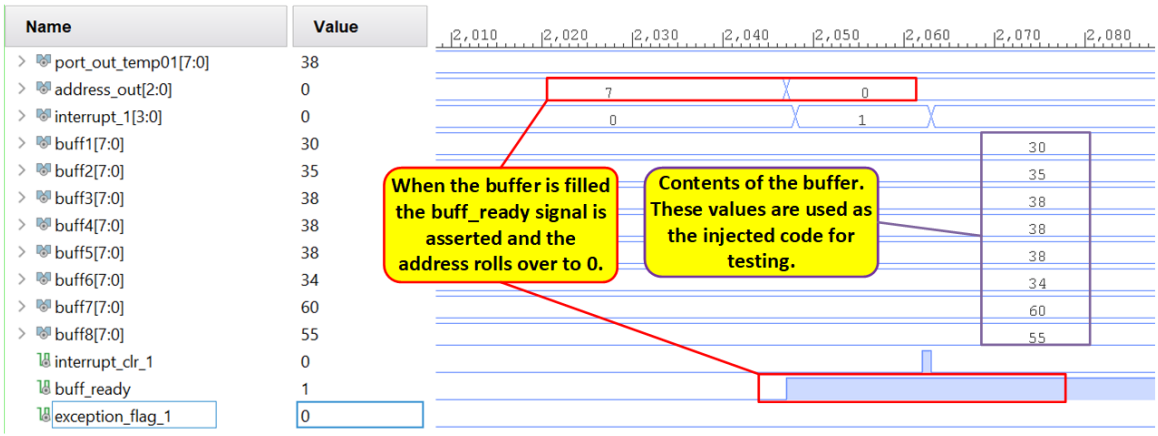

Figure 4.26: Full serial buffer triggering injection

Triplication of Extended Processor

Implementation of a UART serial buffer was the last required component for the expansion of the foundational processor. Single core testing via serial buffer code injection demonstrated the functionality of the system. The final stage of the project required triplicating the processor. Triplication involved the same process that was done for the first triplication step. Each processor core required its own entirely

independent system. The only components that were not triplicated were the UART, display driver, char decoder, attack voter and clock dividers. Everything else was triplicated for use in three independent yet functionally identical processor cores.

RESULTS

After triplicating the processor cores, final system testing was performed. A new set of opcodes were generated for three processors that would run a simple load and increment program. The A register would be loaded with the value x65, then incremented twice resulting in a final value of x67. Finally the program would branch back to memory location x00 and start over. The relevant generated opcodes can be seen in Figure 5.1.

For testing purposes, the instruction set for Core 1 was changed to reflect ASCII characters. This allowed for instructions to be injected via serial communication, ie. LDA_IMM is ASCII 0, INC_A is 8, and BRA is D.

Demonstrated is the functionality of the entire system as a whole. When each processor is supplied the correct instruction set they are able to operate synchronously. This is shown in the continued operation of the state machine as well as the contents being held in the A register.

Testing the attack system was done through the use of an internal logic analyzer (ILA). An ILA allows for the monitoring of desired signals during actual operation after the bitstream is generated and loaded into the FPGA. Using this method allows for the attack vector to be seen occurring in real time and is demonstrated in Figure 5.1 and Figure 5.2.

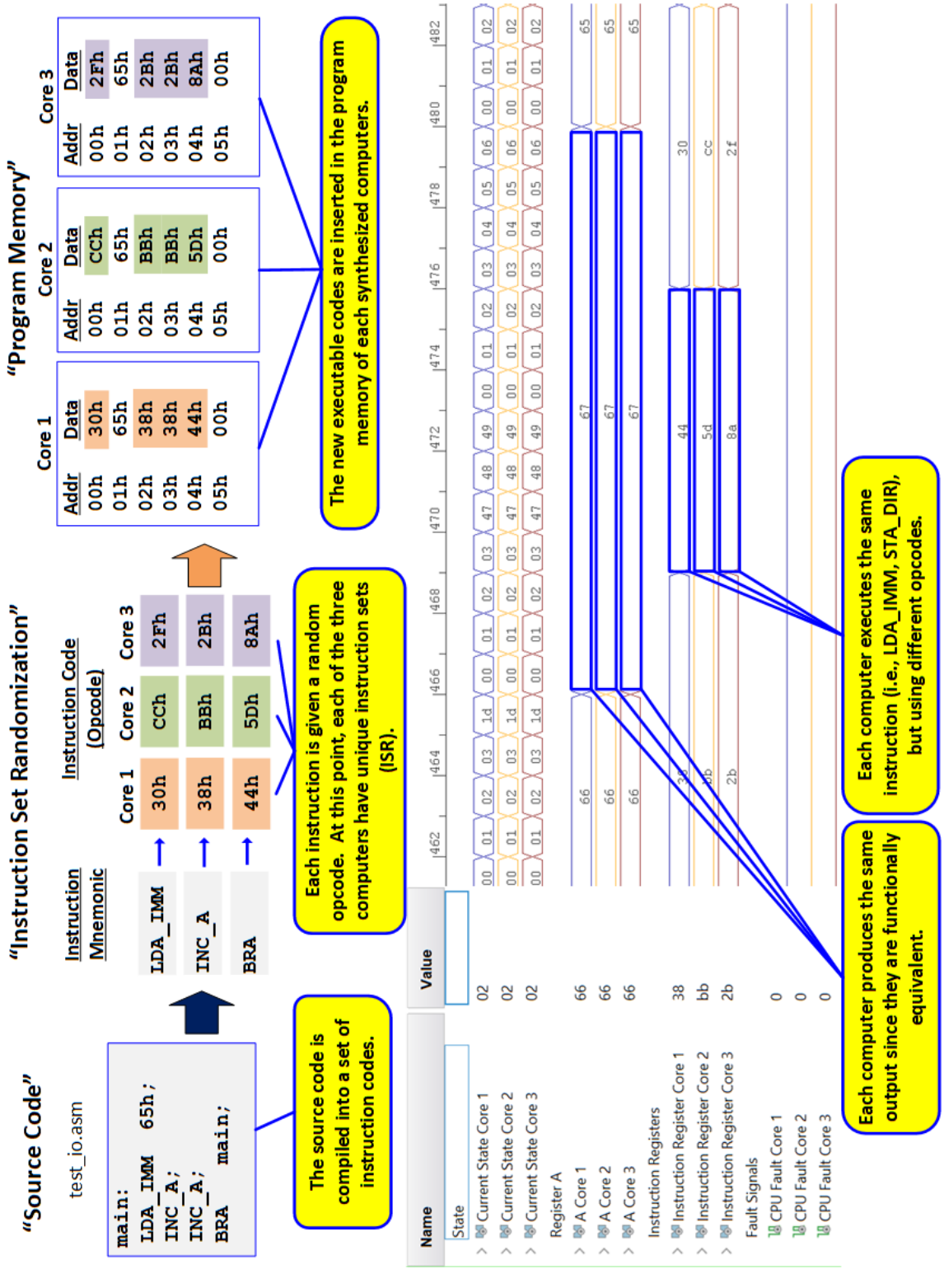

Figure 5.1: Triplicated processor, normal operation

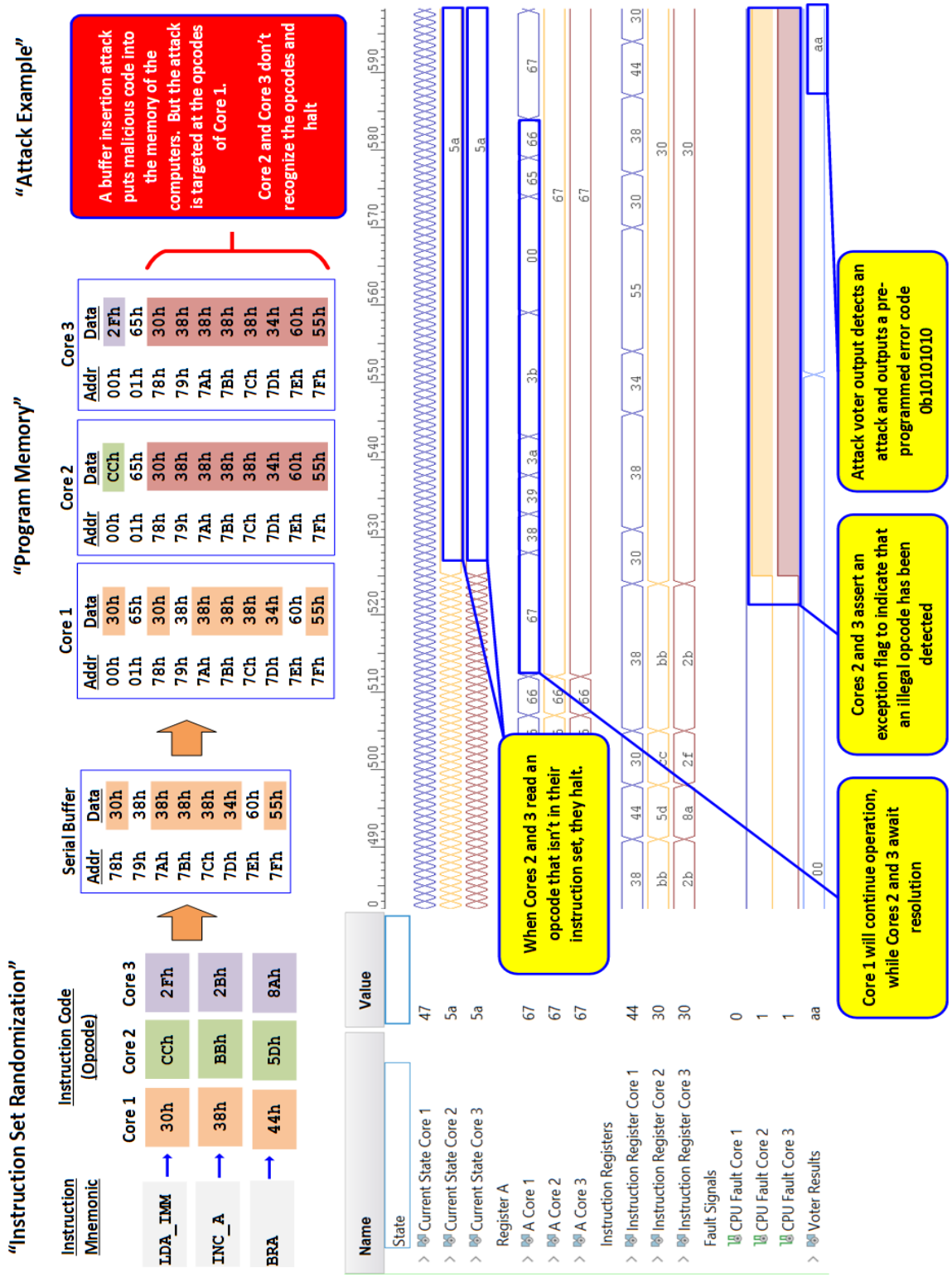

Figure 5.2: Triplicated processor, serial code-injection

The injected attack depicted in Figure 5.2 demonstrates the response when the opcodes are known for a single core and used maliciously. A small program is injected via serial communication, knowing that when the buffer is full the values in the serial buffer are executed. In this demonstration, the injected program attempts to load a value into A, increment it multiple times, and store the value to an output. An example of the full code can be seen in Figure 5.2 and a break down of the known opcodes being utilized can be seen in Table 5.1.

Known Opcode	Function
x30	Load A Immediate
x34	Store A Direct
x38	Increment A
x55	Return from Interrupt

Table 5.1: Known attack opcodes

Since these known opcodes are only viable for Core 1, as soon as the interrupt is triggered and the first opcode is executed the other two cores halt. It can be seen that even though Core 1 continues to operate normally, Cores 2 and 3 halt entirely while waiting for a resolution to the illegal opcode. It's also demonstrated that as soon as the attack is detected the fault exception flags are asserted. Shortly after the flags are asserted the attack voter outputs a value informing the user of an attack. In this case the pre-programmed value is 0b10101010 which can be represented in hex as xAA.

To verify complete functionality of the system, another test was done using a new Vivado project generated through the assembler. Again, for testing purposes, one of the instruction sets was replaced with ASCII friendly opcodes. This secondary

test program consisted of a simple load and store program. The hex value of x65 is loaded into register A and then stored directly to register x8F. Normal operation can be seen in Figure 5.3.

The loaded value is demonstrated to be a constant value that is never changed as the LDA_IMM instruction is perpetually repeated. Similar to the program tested in the first ILA test, it can be seen that the opcodes are different for each core, except for Core 1, which is using the same instruction set as the first ILA test.

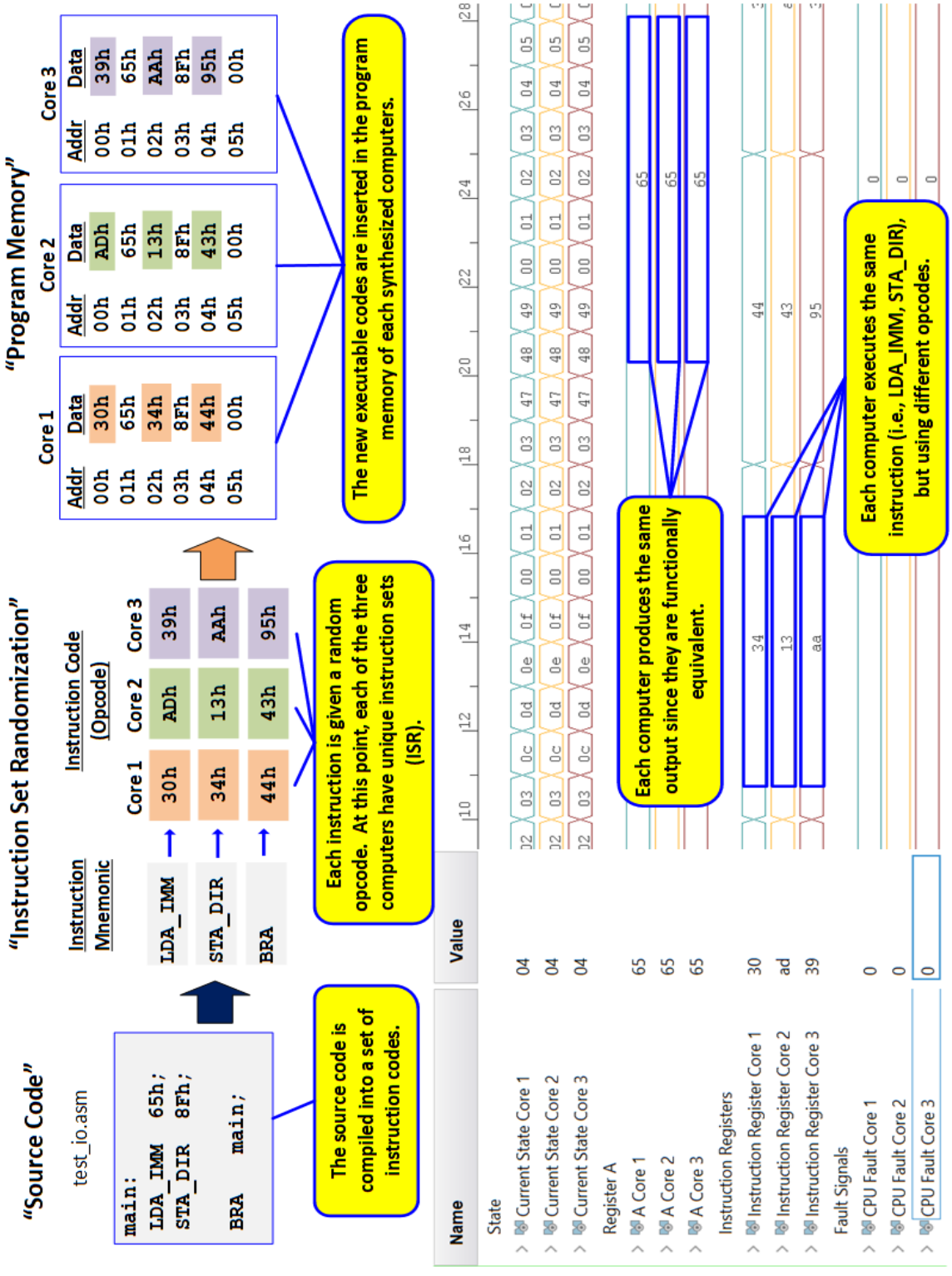

Figure 5.3: Triplicated processor, normal operation. Load and store program

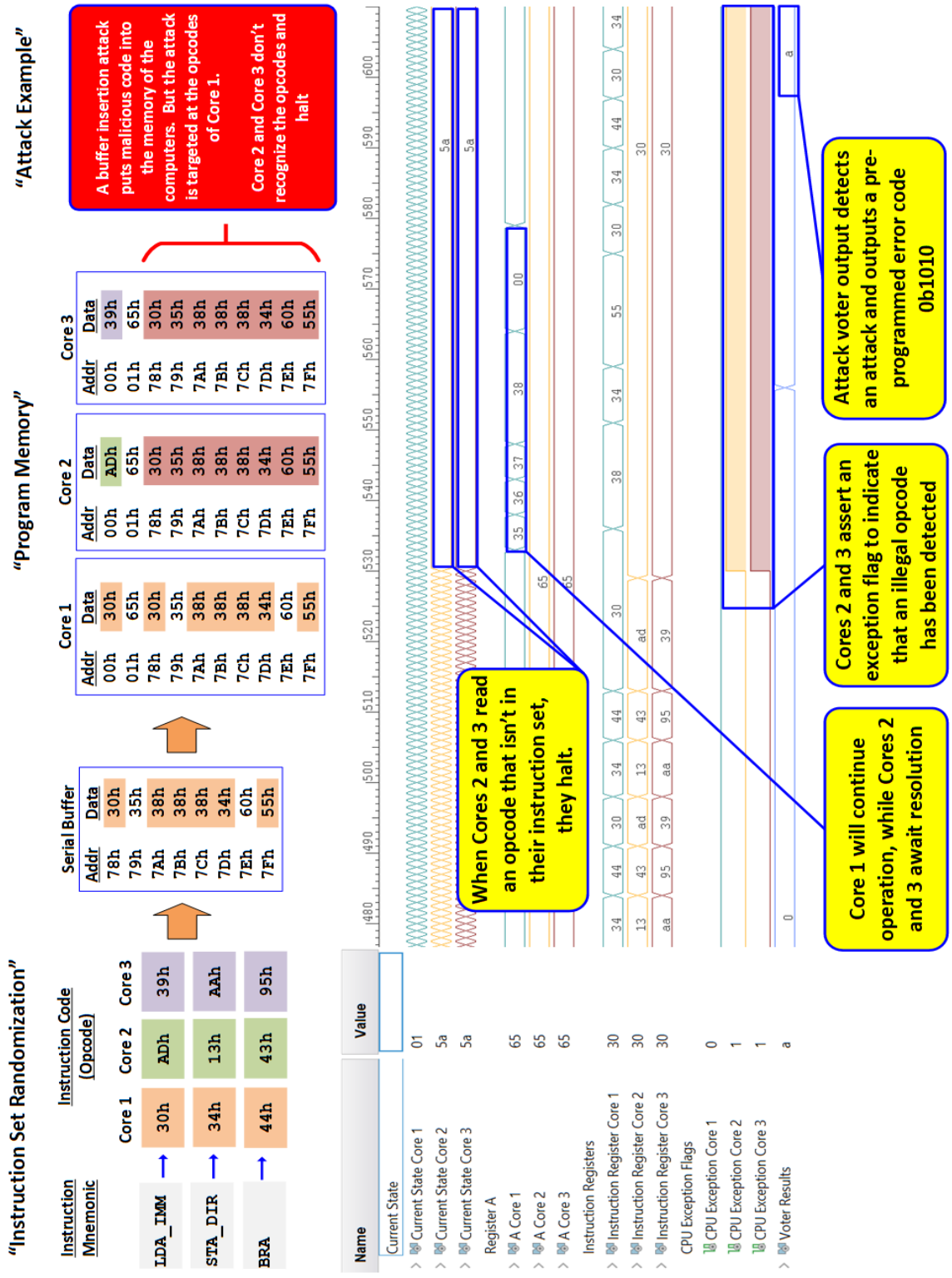

Figure 5.4: Triplicated processor, under attack. Load and store program

After multiple tests demonstrating the functionality of the system, I have concluded that this expanded processor is capable of defending against code-injection attacks. Whether injected directly in program memory as seen in Figure 4.8 or injected through a serial buffer redirect as shown in Figure 5.2 and Figure 5.4.

It's also demonstrated that each implementation of the processor is capable of running with pseudo-randomly generated opcodes. The first ILA test performed using the load and increment program used entirely different instruction sets from the ILA test performed using the load and store program. Even using the same source assembly file, when the memory and instruction sets are generated using the assembler, no two implementations will result in the same sets of instructions. This provides a source of constantly changing instruction sets that preclude attackers from attacking even two implementations of the same program. As a result, a program can be written using standard homogeneous software engineering methods while being deployed in heterogeneous clusters.

Overhead

In terms of overhead, the developed system results in a minimal increase in time and space overhead. There is no overhead increase due to the instruction set randomization as these components are included and required in the design regardless. The only component that provided any additional overhead to the design was the attack voter. While the attack voter system had a minor effect on the setup time, roughly 130 ps, there was not a significant enough change in the utilization percentage as the Vivado tool chain reported the same percentage both with and without the attack voter. However, while the setup timing was affected, both cases are still able to operate at a clock rate of 100 MHz.

It should also be noted that the triplication of the processor does increase the

FPGA fabric space required, as expected. A single core implementation of this processor architecture takes up 371 LUTs total as seen in Figure 5.5

Resource	Utilization	Available	Utilization %
LUT	371	20800	1.78
LUTRAM	8	9600	0.08
FF	213	41600	0.51
BRAM	0.50	50	1.00
IO	23	106	21.70
BUFG	4	32	12.50
MMCM	1	5	20.00

Figure 5.5: Single core utilization report

Also displayed is the low utilization seen in the other FPGA resources. When the system is triplicated the resulting system size is roughly 268% the size of the single core architecture, as seen in Figure 5.6

Resource	Utilization	Available	Utilization %
LUT	993	20800	4.77
LUTRAM	24	9600	0.25
FF	388	41600	0.93
BRAM	1.50	50	3.00
IO	31	106	29.25
BUFG	4	32	12.50
MMCM	1	5	20.00

Figure 5.6: Triple core utilization report

Noteworthy is the fact that while the components increase relative to the number of cores implemented, the power increase between a single core system and a triple core system with an attack voter is only 8 mW, for a total of 200 mW. This translates to a 4% increase in power between a single core system and a triplicated injection hardened implementation.

This means that the entire system architecture space overhead is still less then the overhead seen in the PolyGlot architecture. However, further system expansion and analysis needs to be done to compare the total security of the respective systems.

FUTURE RESEARCH

While this thesis project demonstrates the viability and serves as a proof of concept for a hardware based instruction set randomization defense, further expansion is necessary. For this system to be implemented in industry, several further steps should be be researched, including:

- Further instruction set expansion
- 32 or 64 bit expansion
- Partial reconfiguration
- Compiler
- Linux

Instruction Set Expansion

At this stage in development, the processor can interpret and execute 27 separate instructions. This demonstrates a proof of concept, but falls short of most industry instruction sets. As seen in Figure 2.3, there exists a large number of opcodes in even a small 8-bit microcontroller.

One opcode is particularly desirable, a JSR. JSR is a jump to subroutine instruction and provides the backbone for allowing multiple subroutines to exist inside a single program. Since the ability to handle interrupts already exists in the current architecture, adding this instruction shouldn't pose a problem in future development as it is functionally the same as an interrupt on demand.

Processor Size Expansion

Currently, the developed processor is 8-bit, designed to function similarly to the Picoblaze. However, for wide-spread implementation a larger processor is necessary. Through expanding the processor size to 32 or 64 bits, it would be possible to utilize

existing standards. Additionally, the larger processor size would allow for a drastic increase in available memory. At this point the processor is limited to 256 bytes of memory, which is small compared to any modern processor system. By expanding the MAR register in conjunction with the processor it would be possible to expand the amount of memory available to the system which could easily be allocated and spread across the three current memory systems; program memory, read/write memory, and stack memory.

Component Randomization

In addition to randomizing the instruction set, other components of the device could be randomized. For example, the ALU instruction vector could be randomized for different operations. At this junction the ALU instructions are predefined and hardcoded during the assembler process as seen in A. Similar to the opcode randomization, these values could also be randomized.

Even beyond the randomization of ALU instructions, address space layout could be randomized as well. Right now the address space of the system is linearly defined. Low level registers are defined as program memory, followed by R/W memory and finally followed by the stack and I/0. However, it would be possible to interweave these memory locations per core during the assembler process, preventing any attackers from knowing the locations of protected memory space.

Partial Reconfiguration

The system currently halts when an attack is detected. This works as a proof of concept but is not viable in an actual system. In the event that an attack is detected, it should be mitigated and then the processor should move on to the next instruction. To mitigate attacks, one solution is to enable partial reconfiguration. In the event that an attack is detected, the compromised core can be taken offline while a new

core is brought online. A system such as this is similar to the 9-tile microblaze system seen previously.

Potentially this style of system could be used in conjunction with a hardware based pseudo-random number component that would constantly cycle which processor cores are online. Doing so would not only maintain the current functionality of defending against injection attacks, in the case that an opcode set is known, but also prevent knowledge of which core is active at any given time. This would assist in the ability to prevent attacks from deciphering any given instruction set.

Compiler

Beyond the expansion of the processor hardware, a more robust compiler system will need to be implemented. By using either LLVM, gcc or ANTLR the possibility to program the device using a higher language, such as C, will become a possibility.

It's important to note that the compiler process would need to be slightly modified. For example, if generating a compiler for C, the compiler would need to stop shy of actually generating the machine code (opcodes) and instead just generate the mnemonics of the program. Since the opcodes will be randomized for each core it's not feasible to restructure the compiler every time a new bitstream is created. This will also prevent potential attackers from learning the opcodes of each processor as the final stage will only present users with commonly implemented opcode mnemonics.

Linux

Perhaps the final expansion for the future would be to create a Linux kernel capable of running on the newly developed processor. This could be considered the final stage since all the previous future research topics would need to be completed for this to be feasible. The processor would need to have a larger instruction set capable required by most Linux kernels, for the system to be practical the processor size itself

would need to be expanded to account for a larger address space and a compatible compiler would be required to compile the necessary programs.

Future Use Implementations

A potential use for this would be to replace homogeneous data center configurations. A large data center could be built with the replacement of general purpose processors with FPGAs. Each FPGA would have an independent bitstream generated for it, which would then be pushed out sequentially to each machine in the data center. The initial configuration of each machine could be compared to the imaging process currently used in deploying configurations to large system networks. Once the processor has been scaled to the point of being Linux compatible, any software updates could be deployed in the same fashion used by current systems with new bitstreams, containing newly randomized instruction sets, to each machine on a desired basis. The instruction set deployment could be performed either through generating entirely new bitstreams for each component or by simply regenerating the portion of the existing bitstream that contains instruction set information.

Ultimately, these steps will allow for the broad expansion of this device leading to implementation for a variety of applications. A further exploration of this system could potentially provide a multitude of solutions to a variety of modern cybersecurity threats.

REFERENCES CITED

[1] M. Barr and A. Massa. *Programming embedded systems: with C and GNU development tools*. " O'Reilly Media, Inc.", 2009.

[2] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security*, 8(1):3–40, feb 2005.

[3] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, D. D. Zovi, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communication security - CCS '03*, page 281, New York, New York, USA, 2003. ACM Press.

[4] Cpubenchmark.com. PassMark CPU Benchmarks - AMD vs Intel Market Share.

[5] Eric Brown. Linux 3.1 released with NFC support, 2011.

[6] P. Goyal, S. Batra, and A. Singh. A Literature Review of Security Attack in Mobile Ad-hoc Networks. Technical Report 12, 2010.

[7] J. A. Hogan. Reliability analysis of radiation induced fault mitigation strategies in field programmable gate arrays. pages 1–105, 2014.

[8] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.

[9] C. Kelty. The morris worm. *Limn*, 1(1), 2011.

[10] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. Technical report, 2018.

[11] B. J. LaMeres. *Introduction to Logic Circuits & Logic Design with VHDL*. Springer, 2016.

[12] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. Technical report, 2018.

[13] D. McNamara. Intel's Stratix 10 FPGA: Supporting the Smart and Connected Revolution — Intel Newsroom, 2016.

[14] Microway. Detailed Specifications of the "Skylake-SP" Intel Xeon Processor Scalable Family CPUs — Microway, 2017.

[15] Mitre Corporation. CWE - CWE-122: Heap-based Buffer Overflow (3.2).

[16] S. Ozkan. CVE security vulnerability database. Security vulnerabilities, exploits, references and more.

[17] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, S. Ioannidis, A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis. ASIST. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, pages 981–992, New York, New York, USA, 2013. ACM Press.

[18] D. Ramsbrock, R. Berthier, and M. Cukier. Profiling attacker behavior following ssh compromises. In *37th Annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*, pages 119–124. IEEE, 2007.

[19] Raspberry Pi Trading. Raspberry Pi 3 Model B+. Technical report.

[20] L. Salmon. System Security Integrated Through Hardware and Firmware (SSITH) Proposers Day Overview. Technical report, 2017.

[21] F. B. Schneider and K. P. Birman. IT Monoculture 14. Technical report, 2009.

[22] S. Segars. Enabling Mass IoT connectivity as Arm partners ship 100 billion chips - IoT blog - Internet of Things - Arm Community.

[23] F. Semiconductor. Freescale Semiconductor Addendum Addendum for New QFN Package Migration. Technical report.

[24] K. Sinha, V. P. Kemerlis, and S. Sethumadhavan. *Reviving Instruction Set Randomization*.

[25] Statista. Desktop OS market share 2013-2018 — Statista.

[26] C. Stevens, N. Poggi, T. Desrosiers, and R. Xin. Meltdown and Spectre: Exploits and Mitigation Strategies - The Databricks Blog, 2018.

[27] D. L. D. Turner. Implementation of a radiation-tolerant computer based on a LEON3 architecture. pages 1–90, 2015.

[28] Xilinx and Inc. PicoBlaze 8-bit Embedded Microcontroller User Guide for Extended Spartan ®-3 and Virtex ®-5 FPGAs Introducing PicoBlaze for Spartan-6, Virtex-6, and 7 Series FPGAs. Technical report.

APPENDICES

APPENDIX A

INSTRUCTION PACKAGE EXAMPLE

```
package instructions_core_3 is
 constant LDA_IMM : std_logic_vector (7 downto 0) :=x"99";
 constant LDB_IMM : std_logic_vector (7 downto 0) :=x"8a";
 constant LDA_DIR : std_logic_vector (7 downto 0) :=x"4e";
 constant LDB_DIR : std_logic_vector (7 downto 0) :=x"b6";
 constant STA_DIR : std_logic_vector (7 downto 0) :=x"78";
 constant STB_DIR : std_logic_vector (7 downto 0) :=x"c3";
 constant ADD_AB  : std_logic_vector (7 downto 0) :=x"c1";
 constant SUB_AB  : std_logic_vector (7 downto 0) :=x"66";
 constant INC_A   : std_logic_vector (7 downto 0) :=x"6a";
 constant DEC_A   : std_logic_vector (7 downto 0) :=x"b9";
 constant INC_B   : std_logic_vector (7 downto 0) :=x"13";
 constant DEC_B   : std_logic_vector (7 downto 0) :=x"b2";
 constant AND_AB  : std_logic_vector (7 downto 0) :=x"85";
 constant ORR_AB  : std_logic_vector (7 downto 0) :=x"77";
 constant BRA     : std_logic_vector (7 downto 0) :=x"ac";
 constant BMI     : std_logic_vector (7 downto 0) :=x"d3";
 constant BEQ     : std_logic_vector (7 downto 0) :=x"a2";
 constant BCS     : std_logic_vector (7 downto 0) :=x"69";
 constant BVS     : std_logic_vector (7 downto 0) :=x"2a";
 constant PSH_A   : std_logic_vector (7 downto 0) :=x"c8";
 constant PSH_B   : std_logic_vector (7 downto 0) :=x"15";
 constant PSH_PC  : std_logic_vector (7 downto 0) :=x"2c";
 constant PLL_A   : std_logic_vector (7 downto 0) :=x"a5";
 constant PLL_B   : std_logic_vector (7 downto 0) :=x"59";
 constant PLL_PC  : std_logic_vector (7 downto 0) :=x"72";
 constant RTI     : std_logic_vector (7 downto 0) :=x"24";
 constant STI     : std_logic_vector (7 downto 0) :=x"d2";
 constant add     : std_logic_vector (7 downto 0) :="000";
 constant sub     : std_logic_vector (7 downto 0) :="001";
 constant andab   : std_logic_vector (7 downto 0) :="010";
 constant orrab   : std_logic_vector (7 downto 0) :="011";
 constant inca    : std_logic_vector (7 downto 0) :="100";
 constant deca    : std_logic_vector (7 downto 0) :="101";
 constant incb    : std_logic_vector (7 downto 0) :="110";
 constant decb    : std_logic_vector (7 downto 0) :="111";
 end package instructions_core_3;
```

APPENDIX B

ASSEMBLER

```
import random
import os
import shutil
import errno

programCode = []

def createProgramMemory(filename, programList, core):
    with open(filename,'w') as file:

        fileStart = """
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library xil_defaultlib;
"""
        instructionFile = "use
        ↪  xil_defaultlib.instructions_core_%s.all;" %core

        entityName = "entity rom_128x8_sync_core_%s is" %core
        entityBlock = """
    port (address   : in  std_logic_vector (7 downto 0);
            clock     : in  std_logic;
            data_out  : out  std_logic_vector (7 downto 0));
end entity;"""
        architectureName = "architecture rom_128x8_sync_arch of
        ↪  rom_128x8_sync_core_%s is" %core
        architectureBlock = """
signal EN:        std_logic;

type rom_type is array (0 to 95) of std_logic_vector(7 downto 0);

constant ROM : rom_type :=         ("""

        fileEnd = """
        others => x"00");
begin
    --enables ROM and port_outs
    enable: process(address)
     begin
      if ((to_integer(unsigned(address)) >=0) and
          (to_integer(unsigned(address)) <=95)) then
        EN <='1';
```

```
         else
        EN <='0';
       end if;
      end process;

    memory: process(clock)
     begin
      if (clock'event and clock='1') then
       if (EN='1') then
        data_out <= ROM(to_integer(unsigned(address)));
       end if;
      end if;
     end process;
end architecture;
        """
        file.write("%s" %(fileStart))
        file.write("%s \n" %(instructionFile))
        file.write("%s \n" %(entityName))
        file.write("%s \n" %(entityBlock))
        file.write("%s \n" %(architectureName))
        file.write("%s \n" %(architectureBlock))
        for item in programCode:
            file.write("        %s\n" % item)
        file.write("%s" %(fileEnd))

def createProgramList(filename):
    lines = [line.rstrip('\n') for line in open(filename)]
    for x in range(len(lines)):
        instruction = '%i => %s,' %(x,lines[x])
        programCode.append(instruction)

def createInstructionFile(filename,core):
    with open(filename, 'w') as file:

        opcodes = random.sample(range(16,225),27)
        for x in range(len(opcodes)):
            opcodes[x] = hex(opcodes[x])
            opcodes[x] = opcodes[x][-2:]
        print(opcodes)

        LDA_IMM = opcodes[0]
        LDB_IMM = opcodes[1]
        LDA_DIR = opcodes[2]
```

```
        LDB_DIR = opcodes[3]
        STA_DIR = opcodes[4]
        STB_DIR = opcodes[5]
        ADD_AB  = opcodes[6]
        SUB_AB  = opcodes[7]
        INC_A   = opcodes[8]
        DEC_A   = opcodes[9]
        INC_B   = opcodes[10]
        DEC_B   = opcodes[11]
        AND_AB  = opcodes[12]
        ORR_AB  = opcodes[13]
        BRA     = opcodes[14]
        BMI     = opcodes[15]
        BEQ     = opcodes[16]
        BCS     = opcodes[17]
        BVS     = opcodes[18]
        PSH_A   = opcodes[19]
        PSH_B   = opcodes[20]
        PSH_PC  = opcodes[21]
        PLL_A   = opcodes[22]
        PLL_B   = opcodes[23]
        PLL_PC  = opcodes[24]
        RTI     = opcodes[25]
        STI     = opcodes[26]


        header = """library IEEE;
use IEEE.STD_LOGIC_1164.ALL;"""
        line1 = "package instructions_core_%s is" %core
        line2 = "constant LDA_IMM : std_logic_vector (7 downto 0)
         ↪  :=x\"%s\";" %LDA_IMM
        line3 = "constant LDB_IMM : std_logic_vector (7 downto 0)
         ↪  :=x\"%s\";" %LDB_IMM
        line4 = "constant LDA_DIR : std_logic_vector (7 downto 0)
         ↪  :=x\"%s\";" %LDA_DIR
        line5 = "constant LDB_DIR : std_logic_vector (7 downto 0)
         ↪  :=x\"%s\";" %LDB_DIR
        line6 = "constant STA_DIR : std_logic_vector (7 downto 0)
         ↪  :=x\"%s\";" %STA_DIR
        line7 = "constant STB_DIR : std_logic_vector (7 downto 0)
         ↪  :=x\"%s\";" %STB_DIR
        line8 = "constant ADD_AB  : std_logic_vector (7 downto 0)
         ↪  :=x\"%s\";" %ADD_AB
```

```
line9 = "constant SUB_AB  : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %SUB_AB
line10 = "constant INC_A   : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %INC_A
line11 = "constant DEC_A   : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %DEC_A
line12 = "constant INC_B   : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %INC_B
line13 = "constant DEC_B   : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %DEC_B
line14 = "constant AND_AB  : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %AND_AB
line15 = "constant ORR_AB  : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %ORR_AB
line16 = "constant BRA     : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %BRA
line17 = "constant BMI     : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %BMI
line18 = "constant BEQ     : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %BEQ
line19 = "constant BCS     : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %BCS
line20 = "constant BVS     : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %BVS
line21 = "constant PSH_A   : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %PSH_A
line22 = "constant PSH_B   : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %PSH_B
line23 = "constant PSH_PC  : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %PSH_PC
line24 = "constant PLL_A   : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %PLL_A
line25 = "constant PLL_B   : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %PLL_B
line26 = "constant PLL_PC  : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %PLL_PC
line27 = "constant RTI     : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %RTI
line28 = "constant STI     : std_logic_vector (7 downto 0)
 ↪  :=x\"%s\";" %STI
line29 = "constant add     : std_logic_vector (2 downto 0)
 ↪  :=\"000\";"
```

```
        line30 = "constant sub     : std_logic_vector (2 downto 0)
        ↪  :=\"001\";"
        line31 = "constant andab   : std_logic_vector (2 downto 0)
        ↪  :=\"010\";"
        line32 = "constant orrab   : std_logic_vector (2 downto 0)
        ↪  :=\"011\";"
        line33 = "constant inca    : std_logic_vector (2 downto 0)
        ↪  :=\"100\";"
        line34 = "constant deca    : std_logic_vector (2 downto 0)
        ↪  :=\"101\";"
        line35 = "constant incb    : std_logic_vector (2 downto 0)
        ↪  :=\"110\";"
        line36 = "constant decb    : std_logic_vector (2 downto 0)
        ↪  :=\"111\";"
        line37 = "end package instructions_core_%s;" %core
        file.write('%s \n' %(header))
        file.write("%s \n %s \n %s \n %s \n %s \n %s \n %s \n %s \n %s
        ↪  \n %s \n %s \n %s \n %s \n %s \n %s \n %s \n %s \n %s \n
        ↪  %s \n %s \n %s \n %s \n %s \n %s \n %s \n %s \n %s \n %s
        ↪  \n %s \n %s \n %s \n %s \n %s \n %s \n %s \n %s \n %s"
        ↪  %(line1, line2, line3, line4, line5, line6, line7, line8,
        ↪  line9, line10, line11, line12, line13, line14, line15,
        ↪  line16, line17, line18, line19, line20, line21, line22,
        ↪  line23, line24, line25, line26, line27, line28, line29,
        ↪  line30, line31, line32, line33, line34, line35, line36,
        ↪  line37))


def copy(src, dest):
    try:
        shutil.copytree(src, dest)
    except OSError as e:
        # If the error was caused because the source wasn't a
        ↪  directory
        if e.errno == errno.ENOTDIR:
            shutil.copy(src, dest)
        else:
            print('Directory not copied. Error: %s' % e)


def projectGen(source,destination):
    if not os.path.exists(destination):
        os.makedirs(destination)
```

```
    src_files = os.listdir(source)
    for file_name in src_files:
        full_file_name = os.path.join(source, file_name)
        if (os.path.isfile(full_file_name)):
            shutil.copy(full_file_name, destination)


source = 'CyberCore Sources'
destination = input('Enter Project Name: ')
copy(source,destination)
file1 = '%s/instructions_core_1.vhd' %destination
file2 = '%s/instructions_core_2.vhd' %destination
file3 = '%s/instructions_core_3.vhd' %destination
program = 'program.asm'
mem1 = '%s/rom_128x8_sync_core_1.vhd' %destination
mem2 = '%s/rom_128x8_sync_core_2.vhd' %destination
mem3 = '%s/rom_128x8_sync_core_3.vhd' %destination

#projectGen(source,destination)
createInstructionFile(file1,'1')
createInstructionFile(file2,'2')
createInstructionFile(file3,'3')
createProgramList(program)
createProgramMemory(mem1,programCode,'1')
createProgramMemory(mem2,programCode,'2')
createProgramMemory(mem3,programCode,'3')
```