


# Using FPGAs to Perform Cryptanalytic Attacks on Random Number Generators

Andreas Stocker

*University of Nicosia*


## Abstract



This **paper** will explore **FPGAs** and cryptanalysis and finally culminate in a simple project that shows how these can be used together. The hunger for parallel computation that cryptanalysis has is a natural match for the FPGA. One interesting aspect of this project is that it combines two fields where experts in one field often will not be experts in another. When it comes to attacking real world systems the technique used here is not really useful for that and is rather academic. Though if principles described here were to be used to handle more complex systems those could be significant in the world of computer security.

## 1 Introduction to FPGAs

The sophisticated FPGAs of today are the product of numerous incremental improvements over the years [2]. One such step in the evolution are Programmable Read Only Memories also known as PROMs. These **proms** were



used to implement logic gates. There are also different varieties of PROMs where some of them can only be programmed once and others which could be reprogrammed multiple times. PROMs had a drawback in that sequential logic could not be completely encapsulated within a PROM and would need to be added to a circuit as separate components. Another glaring drawback of PROMs was their lack of speed.

Programmable Logic Arrays also known as PLAs made significant improvements over PROMs [2]. Namely, PLAs were generally **must** faster than PROMs. They could also support a far larger number of inputs. Though one drawback was that **number of combinations** of logic elements was slightly more constrained than that of PROMs.

Programmable Array Logic (not to be confused with Programmable Logic Arrays) were the next step in the evolution that would culminate in the FPGAs of today [2]. Programmable Array Logic also known as PALS added support for clock elements as well as flip flops. They were much more sophisticated in their support for expressing sequential logic. Furthermore, they had the benefit of great performance.

The FPGA was designed with the goal of accomplishing the same computations as **ASICs** but with the added benefit of reprogrammability. Because of this, FPGAs are frequently used to emulate ASICs, as well as to act as temporary stand-ins while ASICs are still being produced. FPGAs as they are today have benefitted greatly from advances in CMOS design that were initially made with improving CPUs in mind. However, it was precisely because CPUs were becoming so efficient that custom hardware lost a good amount of popularity [3]. It became easier for companies to simply use general purpose CPUs instead of investing in tailor-made hardware level designs. One example of such a use-case where general purpose CPUs won is databases [3]. This was in the late 70's and researchers were trying to create a "database-machine". It was specifically tailored to run database queries on low-level hardware.

When it comes to the benefits of using FPGAs **verses** CPUs there are several factors to keep in mind. Image and signal processing are two applications where FPGAs shine [3]. Partially this is because FPGAs can generally offer

a greater level of determinism than CPUs. In an FPGA latency for some tasks can often not only be lower, but the latency and also be predictable. This is crucial for applications where real-time responsiveness is key. Another place where FPGAs shine is parallelism [3]. CPUs main unit of parallelism is their cores. FPGAs however, have a far more granular unit of parallelism which is their logic blocks. This allows for orders of magnitude more potential parallelism over CPUs. These factors make FPGAs ideal candidates for high-throughput low-latency applications.

When it comes to solving real-world problems, FPGAs are quickly moving out of niche, highly-specialized projects and are becoming more common in commodity setups [3]. The main areas where FPGAs are gaining popularity are both in the networking sector as well as the graphics processing sector. One reason for this is that the amount of data generated in the world is growing rapidly. For this reason, the raw processing power of FPGAs is expected to become indispensable for many more companies in the future. One concrete use case for FPGAs is database co-processing [3]. This is because streaming databases are required to process data with a low latency even under heavy load. Ironically, this is one area that is similar to the "database-machine" style projects of the 70's. So perhaps FPGA designers will run into the same sort of problems that "database-machine" researchers encountered [3] all those years ago. Undeniably though, there is a large room for improvement in this sector. One reason FPGAs are still somewhat niche is the fact that FPGA design is not very accessible to "mainstream" programmers. This is because "mainstream" programmers are familiar with the so-called Von Neumann architecture. The Von Neumann architecture assumes instructions are executed from memory and run in a given order. Neither of these factors apply when programming FPGAs.

There are several projects which aim to overcome the reduced ease of use of FPGAs for mainstream programmers. The "Kiwi" and "Liquid Metal" projects aim to do exactly this [3]. The goal is that FPGAs be used with general purpose languages. However the general sentiment in the FPGA developer community seems to be that these projects that allow mainstream languages be used with FPGAs are not developed enough to be used for

mission-critical applications. It could be that at the most fundamental level FPGAs are incompatible with languages designed for Von Neumann run-times. Another way in which mainstream appeal for FPGAs can be improved is by providing developers with out of the box IP that developers can setup on FPGAs to interface with projects running on CPUs programmed in mainstream languages like C.

Another point of comparison between FPGAs and CPUs is clock speed [3]. FPGAs need to operate at significantly lower clock speeds than CPUs. Though the amount of work done by FPGAs in a single clock cycle can sometimes be orders of magnitude greater in FPGAs. This benefits FPGAs by reducing the amount of power consumed by the device relative to a CPU. One downside is that the FPGA may be able to do less computations overall due to this slower speed.

Yet another noteworthy point of comparison between FPGAs and CPUs is their memory model. CPUs with their Von Neumann architecture, generally have memory on a separate chip than the CPU. Though even if memory is technically on the same die as the CPU, like in Apple's M1 architecture, CPUs will always suffer from what is called the "Von Neumann bottleneck". Also termed the "memory wall" occurs because the entire CPU can only access one area of memory at once. One thing that mitigates this disadvantage is the ability to access a continuous block of memory at once. This advantage does not exist if the memory is non in a continuous block however. Memory is significantly different in FPGAs though. FPGAs have flip-flop registers as well as block ram [3]. Sparse lookup tables can also be reprogrammed at run-time and used as additional memory. So in general, FPGA memory boasts far superior locality compared to CPU memory. It therefore has much better throughput, as well as lower latency compared to the CPU. One nifty feature that exists because of the FPGA's memory model is known as "content addressable memory" (CAM). Content addressable memory can be used to implement a key value store with a constant lookup time.

One thing modern CPUs have going for them when it comes to competing with FPGAs are SIMD operations [3]. This allows for parallel operations that mimic those of FPGAs. This means an operation (like addition for ex-

ample) can be performed on two fixed sized arrays of values.

## 1.1 Open Source and FPGAs

When it comes to the development workflow, there is one major place where mainstream CPU programming is ahead of the tools used to develop for FPGAs. That is the arena of open source tools. Open source tools have grown to a point today where virtually all major tech companies either use open source or even contribute to it. These include Google, Apple and Amazon just to name a few. There are a number of benefits to having an open source ecosystem. The first major benefit is accessibility. If anyone can install a tool for free this is great for students and anyone who wants to learn about the project. Another benefit of open source is reliability. If a vendor simply stops updating a closed source project then everyone **dependant an it** is in serious trouble if bugs crop up and there is no one to fix them. Another factor is security. The more experienced people read and understand a piece of code the more likely it is that issues can be discovered. Open source is also great if the consumers of a project want to add their own features. There are some upsides to closed source software though. Closed source software with licensing fees means that a company can pay experienced developers to work on the project full time. This means the project is not dependant on people working on it on their free time. Of course, this does not apply to all open source projects as some projects are popular enough to have paid full time developers working on them. Also, when it comes to security through obscurity, the argument can be made that if attacker do not have access to the source code of a project, they can't find exploits for it as easily.

The current state of open source tools for FPGA designs is as follows: project **Icestorm** provides decently feature-rich tools for a small number of FPGA models. Not only does Icestorm provide most features needed throughout the FPGA design workflow, it is even superious to proprietary tools in some ways. Icestorm does not suffer from the massive bloat and slowness of conventional FPGA tools. Not only are the tools included in Icestorm free from bloat, they can also be a lot faster some times.

The downside of relying on the open source Icestorm project is that it largely

depends on people investing their free time, and if these people lose interest in a particular tool that is part of the project, the entire workflow that uses this project may become untenable.

## 2 FPGA Architecture

An FPGA's internal architecture affects its speed, area efficiency, and power consumption [1]. FPGAs compete with ASICs in that they both allow a digital circuit to be created. They also have several advantages over ASICs because they can be reprogrammed in seconds and cost orders of magnitude less. ASICs can cost millions to produce but they do have their own benefits. FPGAs pay the price for their easy reconfigurability in area, delay, and power usage. ASICs are 20-35 more compact than FPGAs and require 10 times less power. This is because the FPGAs programmable routing circuitry causes overhead. Nevertheless, despite these downsides, FPGAs are far more viable for small to medium sized companies that can't spend millions on ASICs. ASICs cost millions because of three main factors. First, ASICs need expensive software in order for their circuits to be designed. Second, ASICs need a mask so their circuitry can be etched into silicon. The cost of this mask can be reduced by multiple different ASICs sharing a mask. Finally, hiring engineers to design a complex ASIC over multiple years is also quite expensive. These engineers cannot make even a small mistake in their design because that would ruin a mask which costs millions to produce. FPGAs suffer from no such complications because they can instantly be reprogrammed. In fact, FPGAs are often used by ASIC engineers to prototype and test their designs.

Then it comes to their internal architecture, FPGAs consist of a variety of different components [1]. These include logic, memory, and multiplier blocks. All around these blocks is a programmable routing fabric that allows blocks to be configured to both communicate with each other as well as with the outside world through inputs and outputs.

When it comes to memory, modern FPGAs use either flash, static memory, or anti-fuses [1]. SRAM, a type of static memory, is very common in modern FPGAs like from the manufacturers Xilinx, Lattice and Altera. These SRAM cells perform two main roles in their FPGA's architectures. First, a majority of SRAM is used to configure interconnect signals. Most of the left over SRAM is used to persist information in lookup tables also known as LUTs.

## 2.1 Memory Variants

SRAM is used frequently in modern FPGAs because it has a number of advantages [1]. SRAM does not have a limit to how many times it can be reprogrammed. This is unlike the EPROM of early FPGA which could only be reprogrammed once or a couple hundred times. Second, SRAM does not require any special electrical components and can be etched in silicon with CMOS techniques. These CMOS techniques allow FPGAs to benefit from all the production advancements made for modern CPUs.

SRAM is not without its drawbacks however. Every SRAM cell needs 5-6 transistors which is quite a lot. Second, SRAM cannot persist information if a system is powered off. This necessitates the need for another system of persistent storage which increases the complexity of the system. Persistent storage is often provided by flash or EEPROM. Finally, storing a design in a persistent storage system like flash or EEPROM also makes it far easier to competitors of a company to reverse engineer a design. To mitigate this risk encryption is used.

An alternative to SRAM is the so called floating gate technology [1]. This is used in flash or EEPROM which do not lose the data stored in them when power is lost. This flash based setup offers a number of benefits besides persistence. Persistence removes the need for a separate storage mechanism. Persistence also makes it possible to run the device immediately after it starts up because there is no need for the programming step. This is important for certain use cases. Additionally, flash boasts greater area-efficiency compared to SRAM.

One disadvantage brought by flash is that the floating gate requires that charge injection needs to be prevented. Another more serious downside of flash is its limited lifespan because can only be reprogrammed a fixed number of times. This is in contrast to SRAM which can be reprogrammed a virtually infinite amount of times. Depending on the application, this can be a serious disadvantage or a non-issue. The "Actel ProASIC3" for example can only be reprogrammed 500 times. This definitely could be an issue at least for prototyping where dozens of reprogramming cycles can occur in a day. Another very serious issue with flash where SRAM is superior is in the



production run. One previously mentioned benefit of SRAM is that it can be produced using standard CMOS techniques. This is not the case with flash. Flash requires specialized components cannot be created by etching silicon like with CMOS.

In modern FPGAs there is a trend towards using a combination of flash storage and SRAM. This has the benefit of allowing infinite reconfigurability but this comes at a price of greater area overhead.

The final type of FPGA configuration systems that are common today are anti-fuses [1]. The most glaring difference between anti fuses and flash and SRAM is that the former can only be programmed once. This of course makes it unsuitable for many FPGAs applications like ASIC prototyping where multiple configuration runs are a necessity. The name anti-fuse comes from the idea that the FPGA fabric consists of a number of fuses that can be selectively "blown". Though a more accurate description of this process would be the idea of connecting these "fuses". This connection occurs when high voltage is sent through the "fuse". Most modern anti-fuses are metal-to-metal based.

While the fact that anti-fuses cannot be reprogrammed after the initial configuration, they do come with the benefit of greater area efficiency [1]. In metal-to-metal anti-fuses this is because no silicon area is expended to allow configurable connectivity. One thing that does consume a significant amount of area however is the need for programming transistors.

Another benefit of anti-fuses is the ability to include a greater number of switches because of lower on resistances and parasitic capacitance. Also, another obvious benefit that is shared with flash base FPGAs is the ability to work instantly when powered on since the fabric cannot change once programmed.

Yet another benefit shared with flash based systems is that the design of the FPGA is harder to reverse engineer because it is not stored anywhere except the logic configuration itself. The security of the FPGA also benefits from the fact that because the FPGA can only be programmed once a malicious actor cannot reprogram the FPGA if given access to the device.

When it comes to the manufacturing process, however, anti fuse FPGAs can-

not benefit from using a CMOS silicon etching process [1]. Even worse, the fabrication process of anti fuse FPGAs is beginning to run into scaling problems and is generally lagging behind the sophistication of modern CMOS. To summarize, all three of these memory systems have their own pros and cons and which memory system will be used needs to be considered in light of the application and its constraints and goals. SRAM (as well as the flash SRAM hybrid approach) can be considered to be dominant however. This can be attributed to its compatibility with the sophisticated CMOS manufacturing process.

## 2.2 Logic Blocks

The basic unit of computation and storage within an FPGA is known as a logic block [1]. The smallest a single logic block can be is to take the form of a single transistor. Though logic blocks can be far larger than that and even take the form of an entire CPU in theory. The size of a logic block, however, is something that needs to be carefully balanced for maximum efficiency to be achieved.

If a logic block is too small for example, it will suffer from area inefficiency because it will increase the need for programmable routing. Furthermore, performance, and power consumption will also be negatively impacted.

On the other extreme, logic blocks that encapsulate a large amount of logic may be performant, but they nullify the benefits an FPGA provides in the form of programmable logic. In effect, the circuit will become less and less of an FPGA, and more of a standard ASIC.

Selecting logic blocks according to their size and functionality is a key consideration that FPGA creators (as in people creating the FPGA itself, not a design for it) need to consider. There are three main factors that need to be considered when selecting logic block type and instance count: these are area, speed and power.

Logic blocks in modern FPGAs fall into two main categories: normal logic blocks, and blocks that perform specialized computations, like addition for example. What specialized computations will be included need to be carefully considered because unused specialized logic will inevitably lead to wasted

area.

Despite FPGAs being considered a completely general purpose device, creators of the FPGAs will in practice need to consider what kind of designs will run on a target FPGA [1].

There is one main tradeoff that needs to be considered here: how much functionality is encapsulated into every logic block and how many logic blocks total are needed. Increasing the functionality in a single logic block reduces to total number of logic blocks needed (up to a diminishing point that is). Though as the logic block size increases, the number of wires connecting it in the routing increases.

Most modern industrial FPGAs use a hierarchical approach that uses clusters of LUTs and flip flops [1]. This helps control the granularity of logic blocks. This involves grouping basic logic components together and connecting them with a local interconnect. This approach is used instead of increasing LUT size. The benefit is that the needed routing only grows quadratically instead of exponentially.

To consider how logic block size affects speed several factors need to be considered. As stated before, increasing the size of logic blocks decreases the number of them that need to be used. Fewer logic blocks being used means that less routing logic needs to be employed. This in turn means greater performance because the electrical signal needs to travel a shorter distance.

However, this reduction in delay which is outside the logic blocks is accompanied by a larger delay within the logic blocks.

When it comes to power consumption, the tradeoffs are similar to those with area and speed [1]. Reducing the area also reduces the power needed.

Calculating power, area, and speed is simpler when considering only one size of LUT. However, a heterogeneous combination of LUTs can sometimes be more efficient when considering the different metrics.

In contrast to LUTs, specific purpose logic is more efficient (if its being used) [1]. Specific purpose logic is more efficient because internally, it has all the benefits of an ASIC and non of the inefficiencies of programmable logic.

A downside of specific purpose logic is that if its not being used, its a definite net negative on all fronts.

## 2.3 Routing

The purpose of routing in an FPGA is to provide a programmable fabric that connects logic blocks and IO ports. Under the hood, routing uses wires which are connected by programmable switches [1]. For routing it is important that a large variety of circuit configurations are possible. Just like with logic blocks, performance and power consumption are in need of consideration when it comes both to designing the routing and configuring it to take the shape of a design.

”Locality” in routing refers to the level of proximity that interconnected components exhibit. In general, it can be said that circuits exhibit a high level of locality because most components are close. Of course, there is also the need for connecting components that are far from each other.

There are also special types of signals which need to be available globally in the FPGA. [1] These include clocks and resets. These kinds of signals get special treatment in the form of a dedicated interconnect system. These interconnect systems are designed to minimize skew. This means that the variation or noise added to the signal as it travels is minimized.

The main type of routing that designers need to keep in mind is the so called general purpose routing. This is different from the global interconnect routing mentioned previously.

One type of general purpose routing is called global routing. Global routing considers the properties of a routing design on a higher level while not paying attention to the details. general purpose routing mainly considers the locations of routing channels, the number of wires in a given channel, and how various channels communicate.

In contrast to general purpose routing, detailed routing takes into consideration wire length, switch counts and how wires and logic block pins connect. When it comes to the global architecture of an FPGAs routing there are two main styles. The first global routing style is the hierarchical style. In this style, logic blocks are separated into groups. Logic blocks contained in the same group are connected by wire segments. Logic blocks in different groups are connected by wires that traverse multiple levels of routing segments. The further routing channels get from logic blocks, the more wires they tend to

contain.

There are several factors that need to be considered when using a hierarchical routing layout. One benefit of the hierarchical routing layout is that the delay between logic blocks is often more predictable. Performance can also be better for certain types of designs. However, if there is a mismatch between the length of a design's wires and the hierarchical distribution, problems may arise. Furthermore, moving between levels of the hierarchy can cause a significant delay. These are some of the reasons why modern FPGAs largely do not make use of the hierarchical routing system.

What modern FPGAs use now for routing is the so called island style of routing. Island style routing arranges logic blocks in a two dimensional array and surrounds them with routing resources. One key decision that the designers of the FPGA itself need to consider here is the width of a channel. There are several benefits to using the island style of routing and it is likely that these contributed to making it the most popular style of routing in use today.

Since a logic block has access to a variety of wire lengths, the most efficient length wire can be selected. Furthermore, minimum routing delay between logic blocks is trivial to estimate.

Within the island style design, there are a variety of switch block designs that can be used. These include the Wilton, disjoint and universal switch block designs.

## 2.4 Challenges of using FPGAs

Despite all of the improvements made in the field of FPGAs, there are a number of challenges that need to be overcome so further development can be made [1]. As CMOS processes have continued to become more sophisticated, several issues have become more prominent. As silicon chips continue to decrease in size, so called soft errors become more and more of a problem. A soft error occurs when ionizing radiation corrupts some data in a circuit. This sort of corruption does not only occur in FPGAs. It is also known problem in regular computer RAM. This is actually the reason why enterprise grade RAM uses ECC technology to detect this sort of corruption.

The source of the radiation itself can come from both radioactive packaging and from cosmic radiation.

There are a number of ways in which soft errors can be reduced in an FPGA. The first mitigation is at the circuit and technology level. One useful circuit level change is selecting an optimal memory supply voltage. Another technique is to add metal capacitors to memory nodes which decreases their sensitivity to radiation.

A higher level mitigation is at the system level. Here, the designers of the FPGA have added builtin checks to find and correct corruption [1]. One system level mitigation periodically checks the state of configuration memory with its correct value. If an error is found, the FPGA will need to be reprogrammed. A "don't care" flag is set for resources that are not currently in use since corruption in used resources will not affect the FPGAs functionality. Another, far more expensive form of mitigation, is the so called triple modular redundancy. Effectively, the FPGA design is replicated three full times. The circuitry will then vote on what output values are correct. This extreme of an approach would likely not work for most everyday use cases. Though for things like medical equipment, or for any place that handles money this could be useful. It is also noteworthy that routing is the cause for a majority of soft errors.

When it comes to user visible memory this can also be a source of soft errors, the flip flops used here are actually not as vulnerable to corruption as SRAM. This is because SRAM is far smaller in size than flip flops. Nevertheless, like in enterprise grade RAM, error correction can also be used here. It should be noted however, that the error correction itself is not perfect. Some errors will still evade detection.

Adding such error correction comes with a cost. This is because additional data will need to be stored and because of the need for special encoding and decoding circuits.

This cost can be reduced in modern FPGAs by the use of hard circuits that do memory encoding. Again however, like with ECC, these error correction mechanisms only reduce errors and do not eliminate them entirely.

Also, a distinction needs to be made between mere error detection and full

error correction. Since the user's design has full control over memory, the onus falls upon the user to handle errors. This is less desirable than handling errors at the FPGAs system level. This is because handling errors at the system level would only need to be done once, at the time of the FPGAs creation. These user level checks will add additional complexity to each and every design that uses an FPGA. Again, it is also up to the user to decide on the level of error correction depending on their application. Some applications might not cause serious consequences if soft errors do occur.

There is another source of irregularities that can occur in FPGAs. The silicon etching process produces products that exhibit a certain degree of variations. Not all of these variations cause outright errors. Some of these variations simply cause decreased performance or increased power consumption. During manufacturing, as is done with CPUs, every FPGA needs to be individually tested.

This is another area where shrinking CMOS sizes have caused an increase in complications. The smaller the circuits, the greater the variations in performance and power consumption become. Not only do FPGA have variation between each other as a result of this manufacturing process, there also exists variation within the FPGA itself. This form of variation is far more problematic. This is because entire FPGAs can easily be discarded, whereas malformed areas within an FPGA cannot be removed. These malformed areas mean that the clock speed of the entire FPGA needs to be reduced. For a 22nm process, this variation in performance can be as much as 22.4% [1]. Another type of problem that affects FPGA functionality are manufacturing defects. These are different from the process variations mentioned above because an FPGA with process variations may still be able to work, whereas an FPGA with defects may be totally unusable. In the chip manufacturing industry there is a concept of "yield". Yield refers to the percentage of chips in a manufacturing run that are usable. Chips that are not part of the yield need to be discarded. These discarded, unusable chips need to be factored into manufacturing costs.

Yet again, as CMOS processes improve and chips become smaller, complications arise. Smaller, more complex CMOS processes generally result in man-

ufacturing defects and therefor significatly lower yields.

A possible workaround for manufacturing defects, is to still use FPGAs that have manufacturing defects and simply not to make use of defective areas.

The downside of this workaround is that testing a large number of FPGA chips for compatability with a single design may not be economical.

Instead of simply ignoring defective areas, another approach is to build in redundancy from the start. This sort of approach is common in memory devices [1]. The level of redundancy can either be fine grained or coarse grained. Coarse grained redundancy adds entire rows or columns of tiles.

This course grained approach has the downside that a significant amount of additional routing logic becomes necessary. This additional routing in turn has a negative impact on performance and power consumption. Challenges also arise when coarse grained redundancy interacts with heterogeneous block layouts. A more fine grained approach merely adds additional switches. These additional switches are used to bypass defective routing.

One benefit of fine grained redundancy is that these additional switches can even be used if no defects are present. One challenge of the fine grained approach is that a very detailed map of defects is needed so place and route tools can work around them. This detailed map adds significant complexity to the design workflow and tooling.

Coarse grained redundancy also has the benefit of being able to handle defects in routing as well logic. This is in constrast to fine grained redundancy which cannot deal with logic block errors. Though the fine grained approach is better and handling interconnect errors.

## **2.5 FPGA Architecture Conclusion**

The last couple of sections illustrate that FPGAs are a complex and multifacted technology. The people creating the FPGAs themselves as well as the developers of FPGA designs needs to consider a wide range of different factors. This massive complexity does have an upside however. It allows solutions that are very well tailored to specific use cases.



### 3 FPGA Design Workflow

The design flow deals with the steps needed to eventually program an FPGA to solve a certain problem [2]. The design flow of FPGAs is actually quite similar to the design flow used for **CLPDs** and ASICs. In the previous section the creation of the actual FPGA itself was discussed, as well as programming the FPGA. This section will only cover the programming part as well as the steps leading up to it. The main distinction between creating an FPGA and programming for it is that the FPGA creators can choose what components end up on the FPGA while programmers merely make use of the components provided on their target FPGA.

#### 3.1 The Specification

While some may view a formal specification as optional, a serious project will most likely suffer greatly if its creators forgo a real specification [2]. Furthermore, in a team setting, a specification helps communicate to each team member not only what their own role is but also how their section fits into the larger project. This prevents multiple engineers from designing pieces that do not work together.

A specification covers a number of different implementation details. One very useful tool to have in a specification is block diagrams [2]. These block diagrams can both show how a design fits into a larger system, as well as the internal design of the FPGA. An internal block diagram shows how major components within the FPGA exist and are connected to one another.


Another important piece of content is a description of what input and outputs will connect the FPGA to the outside world. Another important concept that needs to be documented when it comes to I/O are timing estimates. The timing estimate for input pins covers setup and hold times. Output pins on the other hand need to have propagation times detailed.

Finally there is a global clock cycle time that needs to be clearly visible. After the initial draft of a specification is complete, it becomes important to prevent the specification from growing stale by always keeping it updated. It's simply not possible to know every aspect of the design beforehand so its

important that when the team decides to make changes that these changes are also committed to the specification.

Part of the work of drafting a specification deals with selecting what major components the design will have. Weighing the positives and negatives of various FPGA products is one of the major parts of this process. There are numerous factors that need to be considered. These include the cost of the parts, their performance, their compatability with each other, among other factor. Another dominant factor may be the level of experience members of the team have with different vendors and product lines.

Another incredibly important step in the designing phase is selecting a design entry method [2]. Small and less complex designs may simply use a schematic entry. In a schematic entry all of the routing is done manually. This approach has the upside of designers getting a very fine degree of control over how the FPGA is configured. There is however, a very serious downside to such an approach and that is that it is not viable for any even somewhat sophisticated design.



The two technologies that are by far the most common for design entry are the languages **Verilog and VHDL**. These languages are highly portable in that they abstract over the low level differences that exist between FGPA models. They are also far more readable, flexible and expressive. These languages allow for what is called "synthesis" in which a software program is read and a low level logic gate design is generated from it.

While these languages may share some similarities with mainstrea programming languages like Java, such as a shared concept of a "for" loop, they are in fact quite different. For starters, these languages make a distiction between logic connections that always exist, as well as synchronous changes that occur every clock cycle.

Another large difference between HDLs and high level programming languages is that they are mostly built on the concept of wires and what connections occur between these wires. There have been some experimental research projects like "Kiwi" [1] that attempted to allow compilation from standard high level languages like C to HDLs. These however, have not gained mainstream traction in industry as well as for any serious projects in general.

It seems there is just too much of a mismatch between these systems that prevents elegant abstractions from being created.

After an HDL has been selected, the next step in the design process is to choose a synthesis tool. The purpose of the synthesis tool is to generate a logic gate from an HDL. Various different synthesis tools exist, but one tool that is particularly notable is the tool that is included in the Icestorm project. The Icestorm project is notable because it is completely open source, something which is quite rare in the hardware design industry in general. Following the selection of a synthesis tool, the next step in the process is designing a chip [2].

## 3.2 Simulation and Testing

While chip design is ongoing, it is important to be also meanwhile running simulations. Even smaller components should be tested using simulation. A lack of simulations can become a serious issue due to the rise of bugs that can be hard to trace if the design becomes a huge black box. Another great benefit of simulation is that in addition to getting a design running correctly, simulations are very useful as a form of documentation. If a developer begins working a section they know nothing about, simulations can serve as a reference of what the design is supposed to do. Simulations can even be useful to the initial developer of a component if they design something and come back to it weeks or months later. Another key role of simulation is to speed up the workflow. Since simulations are running regular CPUs and not on FPGAs themselves, they allow for HDLs to be recompiled far faster. This is critical to the workflow, since depending on the complexity of the design, spending minutes or even hours resynthesizing for every little change can seriously hamper developer productivity. Not only is simulation faster, it also allows developers to have complete knowledge of the state of a system. This means that developers can inspect the states of wires that are internal to the design instead of just being able to inspect the outputs.

However, after developers are satisfied with the design and it has been well tested in simulation, a synthesis step indeed becomes necessary. During synthesis, the high level hardware description language needs to be converted to

an FPGA configuration. Part of this process involves performing place and route calculations.

Another key step in converting HDL code into a real FPGA design is timing analysis. Timing analysis ensures that the electrical signals that travel through the FPGA have enough time to propagate. Without proper timing analysis, serious errors can occur due to the FPGA running on too high a clock speed. Once timing analysis is complete, designers will be given a maximum clock speed. Some FPGA tools, like the Icestorm project, will prevent the FPGA from being programmed with too high a clock speed.

After a design is generated and the maximum clock speed has been found, it is finally time to program the FPGA.

Once the design is programmed onto the the FPGA, manual tests can be performed to ensure the FPGA outputs the correct values for a given group of input values.

Or if the FPGA is part of a larger circuit, the testing step involves ensuring that the complete system works as intended. If any problems are discovered, the workflow repeats itself and additional changes to the HDL as well as the simulation become necessary. Also, known bugs and other issues that were found should be added to the specification.

When everything works as intended, the system will be ready to be put into production. A burn in test is used to ensure that a system keeps working over a long amount of time. However, even if a system was perfectly designed, electrical and mechanical issues can still arise and that is why burn in tests are useful.

### 3.3 Potential Design Problems

One serious design problem that may occur is race conditions [2]. A race condition can occur if two signals which affect a given output can be triggered in a non deterministic order. Race conditions are dangerous because they depend on miniscule delays caused by variations in internal timing. Even a tiny change in voltage or temperature can cause a change in the output.

This can become a very serious problem because chips that were working perfectly may suddenly exhibit erroneous output after weeks, months, or even

years of reliable operation. One way in which race conditions can be dealt with is to introduce a delay by converting an asynchronous operation to a synchronous one. However there really isn't a magic fix to eliminate all race conditions since they depend on a variety of complex, interwoven states. Another serious problems that can arise in an FPGA design is hold time violations [2]. They are similar to race conditions in that they exist because of a timing problem. A hold time violation occurs if data changes at the exact same time as a clock edge. This means the resulting value is non-deterministic and can go either way.

Yet another design problem caused by inconsistent timing is glitches. These occur when an output goes high for a very short amount of time. Unlike race conditions and hold time violations, eliminating glitches is more straightforward. To eliminate a glitch an output can be synchronized by sending it through a flip flop.

"Metastability" is another design problem. Metastability occurs when a asynchronous signal is fed into a synchronous flip flop. In essence, the problem occurs because an asynchronous part of the design is interacting with a synchronous part. The fix occurs by correctly synchronizing an asynchronous signal to a given clock. However, there is no easy fix for metastability. Use of a synchronizer flip flop is a partial solution but there is still the danger of a very small chance that the flip flop will not resume a valid logic level. The danger of this occurring increases with high clock frequencies. Some FPGA manufacturers include special synchronizer flip flops for exactly the purpose of mitigating this problem. Another way to decrease the chance a synchronizer flip flop will not work is to simply include more of them.

In general, asynchronous design is prone to a greater number of problems than synchronous design. This is because small, hard to debug timing errors can arise in asynchronous design. In synchronous design the delay is controlled by flip flop that are attached to a single clock. All of the problems discussed previously are because of asynchronous design and can be fixed by using a synchronous style.

## 4 History of Cryptology

Cryptology is the study of secret writing [4]. This study of secret writing can be broken down into two main disciplines. The first of these, "cryptography", concerns itself with techniques for creating secret writing. In a way, cryptography represents what could be considered the defensive side. It seeks to aid people in protecting secrets. Cryptanalysis on the other hand, also deals with finding ways of breaking cryptography. This is the offensive side of this kind of research.

Throughout the history of cryptology, which spans over 2 millenia, a wide range of methods were created with goal of hiding sensitive messages.

### 4.1 Ancient Times

One of the earliest recorded uses of cryptography starts with the ancient Greeks [4]. Polybius, a Greek historian, developed a monoalphabetic substitution cipher. It was so influential that the techniques it used influenced cryptography two thousand years after its creation. When creating the cipher, Polybius' goal was initially not even to hide a message. He started with the intention of creating a system that could allow signal fires to communicate of large distances. In this system, every letter was represented by two numbers. It had the downside of having a ciphertext that was twice as long as necessary, though some inefficiencies could be expected given that this was one of a very early variant.

A few hundred years after Polybius the Roman empire had risen to prominence. The famous Julius Caesar himself actually was recorded as having used cryptography in his military campaigns in Gaul [4].

Though this is not Caesar's only use of cryptography during his lifetime. The cipher he used here was fairly straightforward. It takes the letters A-Z and randomly reorders them. This is the first time a monoalphabetic substitution cipher with a shifted alphabet was used. Caesar used this cipher to communicate confidentially with people loyal to him.

After the decline of Rome, cryptography became less and less used in the western world for some time. This decline also coincided with a downtrend of

literacy and general scholarly activity.

Though in the Arab world in the middle ages there is some recorded use of cryptography. For nearly a thousand years, the monoalphabetic substitution cipher was one of the strongest ciphers known. Though in the Arab world a cipher would be developed that finally improved upon the status quo.

During the ninth century AD, in a period also known as the Islamic Golden Age, there was a polymath scholar named Al-Kindi. Al-Kindi, being a polymath, was well versed in various scientific disciplines which included astronomy, philosophy and medicine among other fields. He also authored a book intended for secretaries in royal courts which showed how secret messages can be used.

## 4.2 The Renaissance

As stated before, after the Romans, there was a significant decline in virtually all fields of scholarship in the western world, specifically Europe.

This changed with the start of the Renaissance in the 13th century.

During this period, there lived a Franciscan monk named Roger Bacon. Though he lived in monasteries for most of his life, he did a lot of writing and experimenting. One of his most significant scholarly achievements was the translation of various Arabic texts on science and mathematics.

He did not write a letter on cryptography but instead included various cryptographic techniques in a letter for William of Paris.

Seven techniques were described by him in total.

His philosophy about cryptography was that it was good for keeping secrets from the ignorant or uneducated.

Around this time there was another user of cryptography who was arguably far more influential. This influential figure was none other than Geoffrey Chaucer. Chaucer was a poet who also dabbled in astronomy. He makes use of encryption in one of his astronomical books in sections where the usage of some astronomical tools is described. The cipher Chaucer used was the monoalphabetic substitution cipher which was hundreds of years old by this point.

So far, all the people described here were only mentioned working with

cyprography. That is, they only dealt with the protection of a secret. In this section the development of cryptanalysis, which deals with ways of breaking cryptography, will be explored.

One of the first cryptanalytic tools that was created was frequency analysis. Frequency analysis allowed ciphers to be broken by examining the frequency of encrypted letters. Some letters, such as the letter "a", are simply more common in language in general. The polymath Al-Kindi discovered this technique. He noted that, in monoalphabetic substitution cipher, simply substituting one letter for another does nothing to obscure the frequency of that letter.

### 4.3 Cryptanalysis

This discovery by Al-Kindi was a major step forward in the field of cryptanalysis.

Another variant of frequency analysis counts not only single letters, but pairs of letters. These pairs, also known as "digraphs", are even more powerful in breaking monoalphabetic substitution ciphers.

It wasn't until the 16th century however that cryptology began to gain mainstream appeal. Cryptology started to be used in both the military as well as in commercial applications. It was also in this time that a cipher that was far stronger was created. This far more advanced cipher would remain virtually unbreakable for hundreds of years. It was called the polyalphabetic substitution cipher.

Another important cipher was the biliteral cipher. This cipher was actually making use of stenographical techniques. This cipher was created by Sir Francis Bacon. Bacon was actually not only a scholar, but also quite involved in government. His exploits gained him quite a significant following. What was significant about Bacon's biliteral cipher, was that it was resistant to frequency analysis. It is likely that Bacon was aware of frequency analysis, which had become quite well known during his time, and designed his cipher specifically to be unbreakable.

Where stenography comes into play here is in the encoding of an important message inside a fake, meaningless message. Furthermore, the meaningless



message is a couple times longer than the actual message.

According to some followers of Francis Bacon, Bacon used cryptographic techniques to encode a message in plays. They also argue that Shakespeares plays were not in fact written by Shakespeare, but by Bacon. This is what is known as the "Baconian Theory".

Another significant development in the cryptology world during this time was the use of nomenclators. Frequency analysis was a well known weakness of monoalphabetic substitution at the time. The basic idea was the addition of homophones as well as a code book. This technique for mitigating frequency analysis did prove quite useful.

The first nomenclator was created by Gabriele di Lavinde in 1379 in Italy. He created this technique for the antipope Clement VII. It takes a regular monoalphabetic substitution cipher and combines it with a small code book. Improvements on this techniques that were made later did not use a code-book but homophonic substitution.

There are several downsides of using nomenclators though. The main downside was that despite frequency analysis becoming more difficult, it was still somewhat effective here.

Also, all parties that wanted to encrypt or decrypt a message needed to have a codebook. This codebook could be found by an adversary and used to break the encryption.

Despite suffering from all these downsides, nomenclators became popular for diplomatic, and to a lesser extent, military applications. As the popularity of encryption grew, the demand for cryptanalysis grew as well. This led to the rise of so called "Black Chambers". These had the purpose of breaking cryptography of rival or enemy factions.

One elite Black Chamber served the Vatican.

With dedicated code breakers in the form of Black Chambers growing in popularity, the arms race between code authors and code breakers continued. Cryptoanalysis had begun to gain an upper hand over cryptography, meaning that it became harder and harder to make unbreakable encryption. This created a need for stronger cryptography. Two main methods were created to strengthen cryptography.

These were the so called "modern code", as well as the polyalphabetic substitution cipher. The monoalphabetic substitution cipher, the cipher used by Julius Caesar all those years ago, could not stand against frequency analysis. This was because it simply did not obscure the original text enough which made it too easy to observe characteristics about the text despite it being encrypted. Something that helped a little bit in obscuring the original text was simply the removal of spaces, periods, commas, and so on. This still is not enough to hide letter frequencies however. A better way to obscuring letter frequencies is to use multiple cipher alphabets. This means that a letter can be mapped to various different letters.

#### 4.4 America at War

A few hundred years later, the American revolution also saw a large use of cryptology. This was only in the later parts of the war that the Americans made use of cryptography. At the start of the war cryptography actually was not used at all in communications. This changed when amateur cryptographers that pushed for the use of cryptography to prevent the British from intercepting their messages.

After the Americans began using cryptography, the level of cryptographic sophistication that both sides had was about equal.

A lot of the cryptography on both sides was somewhat experimental however.

Undoubtedly though, cryptography played a crucial role in the war. This was because it protected important strategic information. Despite the significant role cryptography played in the war, neither side had dedicated, full time cryptographers in their employ. As the war progressed, both side's military intelligence agencies became more and more sophisticated.

In the American Civil War, significant progress was made in both the fields of cryptography as well as cryptanalysis.

In 1844 the telegraph became more and more popular. Naturally, this technology also became adapted for use in the military. The first time the telegraph was used in war was actually by the British in Crimea in 1853. The telegraph was used by both sides in the American Civil War.

One improvement that came as a consequence of the war, was an step up from traditional codes. This was because codes were both challenging to use in the theater of war, and if one code book was lost, every codebook that contained a copy of those codes would have to be replaced.

Another downside of using codes was that they were not practical for the sheer amount of information that was flowing through telegraph stations. To fix these downsides, so called field ciphers were implemented.

One instance of cryptanalysis during the war occurred when a Confederate soldier named Alexander received a Union cryptogram from a captured courier [4]. The encryption used here was new to Alexander. He was able to quickly realize that the encryption hid its secret by simply reordering words. He termed this a "jumble". After expending significant effort to try to break the jumble, Alexander was not able to break it.

This jumble turned out to be the Union's main cipher. This cipher was created by the telegrapher named Anson Stager. Stager initially started with a simple route word transition cipher. Then Stager modified the cipher to make it even harder to break. One thing he did was to add "nulls" or "blind words". One additional benefit of these nulls was their ability to be used for checking the consistency of a message. This was extra useful for telegraph messages which could become corrupted during their encoding and decoding processes. Stager also encoded some special words through the use of code words. Additionally, he added a "commencement word". The purpose of the commencement word was to encode certain encryption properties like the size of the rectangle.

The complexity of ciphers grew a lot during the war. The cipher used at the start of the war could fit on a small card. In contrast, the fourth cipher that was released towards the end of the war was so complex it needed a book with dozens of pages.

When it comes to comparing cipher complexity used by both sides, the Union used just one somewhat simple cipher. This is in contrast to the confederates who used two completely different ciphers. One of these ciphers was the Vigenere cipher. The Vigenere cipher was supposed to be very secure. Though somehow the Union was able to break many messages that made use of this

cipher. Polyalphabetic substitution ciphers were actually quite secure that this time, but there was a certain factor that made the Vigenere cipher vulnerable.

The Vigenere cipher was vulnerable because of the way it was used. There were a number of factors that led to the security of this cipher to be compromised. Their first mistake was that word divisions were not kept hidden but were plainly visible on the cryptograms. This made it much easier for the Union's cryptanalysis experts to guess words. Another serious mistake made by Confederate cryptologists was to only encrypt parts of messages. This was likely because encryption and decryption were quite time intensive. While this may seem like a clear problem for security, it could actually be argued that this would strengthen the security of the encrypted message. This is because the less text cryptanalysis personnel has to work with, the harder it should be to guess the original message. In practice however, this made messages easier to break because the unencrypted message gave context to the encrypted sections. Another mistake the Confederates made with their encryption was to use only three keys for their highest security cipher.

Even without the presence of implementation flaws, the Vigenere cipher itself was broken in the middle of the nineteenth century. A critical flaw in the cipher would be found that created a reliable cryptanalytic technique for breaking the cipher. The Cambridge mathematics professor Charles Babbage had a great deal of experience in various types of mathematics. He even pioneered several ideas that would be used in the computers we have today. Though one could say that he had the character flaw of always starting new projects and never finishing anything. In 1852 Babbage found a way of breaking polyalphabetic ciphers.

Without any communication with Babbage, another solution for breaking the polyalphabetic cipher was actually discovered independently. This solution would come from a retired Prussian army commander. His name was Friedrich Kasiski and he published a book on his techniques for breaking the cipher in 1863.

Both of these men found the same flaw in the Vigenere cipher independently. The main idea needed to break the cipher made use of the fact that a fatal

flaw in its implementation was the repetition of the key.

This technique wasn't perfect however. In short ciphertext it was possible there would be no repeated sections. Also, the key could be too long or certain duplications could be mere coincidences. One improvement was to have a way of finding the key length. This technique was discovered some time later in 1920 by the American William Friedman. Friedman had developed numerous techniques while working at the Cipher Department in Illinois. Friedman's technique made use of rigorous statistical methods.

In conclusion, in the twentieth century cryptanalytic techniques were becoming more and more sophisticated. The invention of the telegram also coincided with increased use of cryptography. With techniques developed by Babbage, Kasiski and later Friedman, polyalphabetic ciphers became less and less reliable. This set the stage for more sophisticated cryptographic ciphers.

## 4.5 World Wars

The first World War saw the introduction of cryptographic systems that were quite similar to what is used today [4]. Like telegraphs in the nineteenth hundreds, the introduction of radio revolutioned communication. This also had a large influence on the development of cryptography and cryptanalytic techniques. Army commanders were using radio to send commands to their troops. This greatly increased the volume of communication which again affected how cryptography would be used because of the large increase of raw ciphertext. For the first time, ciphers became so sophisticated that specialized machines needed to be built. The first World War also saw the creation of America's first dedicated cryptanalytic agency.

Cryptographic techniques became so sophisticated that during the course of the war, specialized machines became necessary for cryptography. This wasn't only because they were more sophisticated. This was also because of the use of radios greatly increased the bandwidth that needed to be encoded and decoded. For the majority of history, messages were sent through horseback riders which greatly limited the bandwidth of messages. Telegraphs increased this bandwidth but there was still a limited amount of telegraph stations again limited the amount of information that needed to be decrypted

and encrypted. Radios were not limited to such telegraph stations nor did they require wires to be laid.

Processing so much encrypted data simply became untenable for manual encryption methods.

Another reason machines were used is **because** electronic equipment became quite a bit more sophisticated during this time.

Yet another trend that emerged at the start of the first world war was the creation of permanent intelligence organizations. Previously, intelligence organizations were built up from scratch at the start of every conflict and disbanded soon after. This changed with the creation of permanent intelligence agencies during this period. Not only were intelligence agencies permanent, their perceived importance by governments grew. With this increase in perceived importance the amount of resources made available to them grew as well.

One such agency was "Room 40". Room 40 was created by the British and at the beginning none of its recruited members had experience in cryptology. The British also created another cryptanalytic group known as "MI1b". Their purpose was breaking German cryptograms.

There is a large difference between "ciphers" and "codes". These are independent cryptographic systems. The cryptanalytic techniques for cracking them are also quite different. When it comes to breaking ciphers, frequency analysis, as well as various other techniques are employed. Codes require different techniques for breaking them. Codes rely mostly on guessing in order for a codebook to be incrementally built up. Another way of breaking codes simply involves recovering code books. Cryptoanalysis during this time involved processing a very large amount of ciphertext. Room 40 made significant progress when it was given access to a number of German naval codebooks. Another notable achievement of Room 40 was the breaking of the 13040 code. They did not break this code by finding codebooks. They broke this code by sifting through hundreds of coded messages. Room 40 ended up breaking superencipherments so often that the Germans ended up changing these keys every three months. In total, Room 40 decrypted more than 15 thousand German naval messages over the war. The Room 40 of-

fice also ended up employing hundreds of people towards the end of the war. One notable fact about Room 40 was that they were reticent in sharing their knowledge with their allies. The French for example, shared all of their information with the British despite Room 40 not sharing anything with them. Arguable, Room 40's greatest achievement during the first world war was when it decrypted a message about the Germans increasing submarine warfare. Room 40 also decrypted a German message that promised Mexico territory in the United States in exchange for their help. This message was significant because it would be important in America's decision to go to war. When it came to America's cryptological offices, it was quite far behind. Whereas the Europeans had had their Black Chambers for hundreds of years by now, by the start of the twentieth century the Americans had no official office for cryptography. The cryptographic organizations that America had been using in previous wars were all temporary and were disbanded after these conflicts ended.

This finally changed with the creation of the American "Army Signal School" in 1911. Notable figures that worked here included Joseph Mauborgne and Parker Hitt. In 1916, Hitt published the "Manual for the Solution of Military Ciphers". This manual detailed various kinds of ciphers as well as cryptanalytic techniques that could be used in breaking them. Despite being somewhat outdated to techniques used in Europe, this book remained the go to handbook for American army cryptologists for a number of decades.

When America joined the first world war, the American army had only a handful of cryptologists [4]. One significant American cryptologist at this time was Herbert Yardley. He worked as a code clerk for the Washington State Department. Within the span of only a few hours, Yardley was able to break the encryption used in a letter from President Woodrow Wilson to one of his aids. Another one of Yardley's key achievements was the creation of a 100 page book that covered the codes and ciphers that the state used.

Soon after the war started, Yardley was put in charge of the new cryptologic section named MI-8 or Section 8 [4]. In under a year, Section 8 grew from a handful of people, to being composed of various departments. By the time the war had ended it was composed of 165 people. Towards the end

of the war, Yardley traveled to England and France to improve the sharing of intelligence between America and its allies. Again however, the British only shared very little with their American allies and did not even let Yardley visit Room 40. On his visit to France however, Yardley met with many French cryptanalysts. Though even the French did not share the diplomatic codes and ciphers they had recovered. By the end of the war, Yardley was part of the American delegation at the Versailles Peace Conference. MI-8 however, was already being scaled back in preparation for peace.

Another major American intelligence unit during the war was the A.E.F. which operated in France. This unit was more geared towards tactical codes and ciphers. The American army in France had two main cryptologic branches. There was Military Intelligence and the Army Signal Corps. Military Intelligence also had a Radio Intelligence Section. Radio Intelligence was not only focused on code and cipher cryptanalysis, but also did traffic analysis and telephone interception. Furthermore, it also kept an eye on American communication security to check that security guidelines were properly followed. The Signal Corps on the other hand, had two code and cipher sections. These were the Code Compilation section and the radio interception section. The radio interception section listened in on German cryptograms sent by radio and then forwarded these on. The overall organizational structure of the American cryptological sections was overall quite similar to that of the British and French.

Through the course of the war, both the Germans and the allies switched from ciphers to codes for use on the front lines. This was because ciphers proved to be difficult to use in this scenario. These codes on the front lines were also called "trench codes". German "Satzbuch" codes were changed monthly which meant that the Americans had to work quickly breaking them. The trench codes for the American army were created by the Code Compilation Section of the Signal Corps. The Code Compilation Section created these trench codes by starting with an outdated British trench code which they then improved upon.

While developing their trench code, the Americans assigned Rives Childs to try to break the code. Given only 44 ciphertexts, Childs was able to find the



entire cipher alphabet within five hours. This trench code was then completely discarded and the development of a new code began. A 2 part code was selected which, unlike the previous code, would use no superencipherment. The benefit of this 2 part code was that it was easy to encode and decode.

The downside of this code was that enough ciphertext was recovered by the enemy, more and more parts of the codebook could be recovered. There was also the risk of a codebook being captured by the enemy. To mitigate these problems, a new codebook would be issued every two weeks.

This series of codes would be dubbed the "River" codes and would prove to do their job well throughout the war. Every new variant of the code that would be issued was named after an American river. The entire code could fit in a pocket as it was only 47 pages long. Another nice feature the code had was the use of a font that was easy to read.

The American 2nd Army received its own variation of the code dubbed the "Lake" series. This system was proved to be every effective throughout the war. Despite codebooks being captured by the Germans on three occasions, security was maintained due to new codes being issued within days.

Another feature that was introduced to the system was the emergency code list. This code list was highly portable and easy to use.

While trench codes were almost completely dominant among the armies of all sides, ciphers were still used to some degree. For the first two years of the war, the British made use of a field cipher for tactical communications.

The Germans on the other hand also made use of a complex field cipher throughout the entire war but only for high level communication.

The British used a cipher known as the "Playfair" cipher. The polymath Sir Charles Wheatstone invented this cipher back in 1854. It is called the "Playfair" cipher because it was heavily promoted by the Baron Lyon Playfair who was intent on having the British government use the cipher. A few decades after its invention, the Playfair was adopted as the government's official field cipher in the 1890s. Being first used in the Boer War, the cipher was used by the British in the first few years of the first world war. The Playfair is a variant of digraphic substitution cipher. It works by encrypting

two letters at a time and encrypting these plaintext digraphs into ciphertext digraphs.

Where cryptanalysis is concerned, the most straightforward technique for breaking a Playfair cipher is frequency analysis. For frequency analysis to work here, a large volume of ciphertext is needed. The frequency analysis will be performed on two letters at a time since the Playfair cipher uses digraphs.

Arguably the most famous cipher of the first world war is the ADFGVX cipher. This cipher was created by the Germans who were preparing for a last desperate push to break the standstill. ADFGX was quite unique from anything the Germans had used throughout the war. ADFGX is a variant of fractionating cipher. Its called a fractionating cipher because it produces digraphs which are later broken in two. Encryption entailed three different steps.

The only way the ADFGX cipher could be broken was by finding the sorted transposition key order. This was the task that Georges Painvin faced as he attempted to break this cipher in March 1918 just as the Germans were launching their offensive. One challenge that Painvin faced was a lack of ciphertext volume.

Though this changed with the start of the offensive as more and more German communications were intercepted.

Painvin's first step was find a bunch of messages that had similar beginnings and endings. Within three weeks he was already beginning to recover keys and break the cipher. He grew so proficient in breaking this cipher that on some days he was able to decrypt half of all messages. The Germans threw him a curve ball so to speak when they replaced the ADFGX cipher with a new variant known as ADFGVX. ADFGVX was different because a new row and column were added to the Polybius square. Painvin overcame the challenge of breaking this new cipher by working just one day and night.

This feat made him the most famous cryptographer of World War 1.

The first world war saw many advances in cryptography and cryptanalysis. One of the main lessons of the war was that the sheer volume of messages as well as the risk of human error made manual encryption and decryption un-

reliable. These flaws could be overcome with the introduction of specialized cryptographic machines. The most basic cryptographic machines used at this time, cipher disks, had already been hundreds of years old by this point. Another one of these machines was the cipher cylinder. Five years after the first world war had ended, various kinds of machines had been created to generate polyalphabetic ciphertext.

Arguably the most famous of all cipher machines that existed during this period is the Enigma. Rotors are used for both the encryption and decryption of polyalphabets. Arthur Scherbius first developed it in the 1920s. Various improvements were made to it over time. A few years after its creation it was officially being used by the German Army and Navy.

One key attribute of the Enigma machine is that it is "self inverse". Self inverse cryptographic machines operate with the principle that encryption and decryption both use the same steps, albeit in a reverse order. This characteristic of the Enigma machine would later turn out to be its main weakness.

Another serious weakness the Enigma machines had was that they had a reflector components which prevented any letter from being encrypted to itself. The Enigma machine would need two or three people to operate it. One inefficiency in its operation was due to the fact that letters were not printed out but instead had to be read through a set of lights.

The Enigma machine was considered unbreakable by the Germans. This confidence in the machine's security was because of its great complexity as well as its high number of possible alphabets.

The French, British, and Americans all bought Enigma machines and tried to break them. These early attempts at breaking the Enigma machine were met with failure. It was arguably the Poles however that had the greatest need to break the Enigma machine. They invested the most resources in breaking the Enigma machine because they knew that in the event of war, they would be among the first the Germans would attack. Poland had been part of the Prussian empire so the Germans were expected to want this territory back.

This led to the creation of Poland's own cipher bureau. With the creation of this bureau, mathematicians were also recruited and trained in cryptanal-


ysis. One of these mathematicians was Marian Rejewski. In under a year, Rejewski had made enough progress to be able to decrypt certain Enigma messages. Rejewski's approach to break the cipher made use of the mathematical theory of permutations. He also had the advantage of having access to the day keys the Germans used with their Enigma machines. These advantages allowed the Polish to read decent amount of the German's Enigma messages.

Then in 1938 the Germans made a large change in how the Enigma machine operated that made the techniques developed by the Poles so far useless. They accomplished this first by changing the indicator settings and later by adding two more rotors. These, among other improvement the Germans made, made the Enigma ciphers orders of magnitude harder to break. The Polish simply did not have the resources to overcome these new difficulties so they passed the knowledge they had gained so far to their allies. The Enigma machines were so sophisticated that attempting to break their encryption by hand would be impossible and the Allies would need to build specialized machines to break the encryption.

Arguably the most notable figure in the story of the Enigma machines is Alan Turing. He graduated from Cambridge with first class honors in mathematics. He gained significant popularity in 1936 when he published the paper "On Computable Numbers". This paper was a response to a challenge set forth by the German mathematician David Hilbert. Turing answered Hilbert's decision problem through a theoretical abstract machine. This theoretical machine would later be known as the Turing Machine.

In 1939, after German modifications to their Enigma machines thwarted Polish cryptanalytic techniques, the British began looking for other cryptanalytic techniques. In general, Turings approach for breaking the Enigma ciphers was to rule out as many impossible answers as possible. To this end he built a machine which he called "bombe". After some improvements and by using a large number of these "bombe" machines, the British were able to break the Enigma's daily keys in mere hours.

## 5 Random Number Generators



Random number generators, also known as RNGs, can be used for various different purposes [6]. One of the main uses of RNGs is in simulations. Another use of RNGs is in sampling. The purpose of sampling is to choose "samples" at random which are a good representation of the true average. Random numbers are also generally used in programming. So called randomized algorithms make use of the indeterminism of random numbers. A very straightforward use of random numbers in programming is the generation of secret tokens that are used in authentication.

### 5.1 History of Random Number Generators

Another use of random numbers is when an unbiased decision needs to be made, like with a coin flip for example.

When it comes to the actual meaning of randomness, a number of its own cannot be random. Instead we have a sequence of random numbers that follow a given distribution. This means that there should be no observable pattern between the numbers. The concept of uniformity that all numbers are equally likely to show up, without any favoritism making some numbers more common.

In a truly random number generator, what numbers were generated in the past should not have any impact on the next number that is generated.

Though sometimes pseudo random number generators are better for an application because they conform to the gambler's fallacy. The gambler's fallacy assumes that if a "random" result or set of random results has occurred, the next values that appear should skew towards being different to the random results observed so far.

So for example, if a coin was flipped five times and the result was always heads, the gambler's fallacy would assume that there is a very high probability of the next result being tails. This is of course completely false in a truly random system. However in some cases it becomes desirable to have a pseudo random number system that exhibits this property.

Why? Because sequences of a few items that skew greatly towards one aver-

age are less likely to appear. The final result of this is that the random number generator could be considered to be more "fair". This is no **doubt** the reasoning behind the creators of the computer game Dota 2 implementing a system that has such "random" properties.

Before machines were created to generate random numbers, several simple tools were used for this purpose. Such simple tools include dice, which have been used for thousands of years, as well as numbered balls that would be chosen at random. Cards were another way of generating random values. In 1927 a table of 40 thousand numbers was published by L. Tippett. Soon after this time however, devices were created that generated random values. In 1939 such a machine was created by Kendal and Babington-Smith to generate a table of 100 thousand random values. One very early computer, the Ferranti Mark I, had a dedicated instruction for generating random bits and storing them into an accumulator. The source of entropy used by this computer was resistance noise. Alan Turing had previously had an idea for such a system and this was its implementation. Another such device, named ERNIE, was used by the British lottery to generate the winning numbers. Once computers began to become popular, people started looking for ways to generate random values within software. Connecting a computer to a random generator like the ERNIE had the serious downside of not being able to reproduce a series of number deterministically when using the same seed. Another downside of using such random number generating machines was that they could seriously malfunction in ways that were very difficult to detect. The practice of distributing lists of random numbers became popular again in the 90's where a billion random numbers could be stored on a CD. In 1946, von Neumann came up with an algorithm for generating random numbers completely using arithmetic operations on a computer. Of course, such a sequence that was generated deterministically through arithmetic operations could be argued to not actually be random.

These numbers would have the appearance of being random however. This early software RNG used a middle square to generate subsequent outputs from a seed. Despite the philosophical concept of these numbers lacking true randomness, it can still be argued that random number generation with

arithmetic is still practical for most applications.

Von Neumann's initial approach using the middle square proved to have some serious flaws though. The problem was that this RNG could reach states where the RNG would repeat the same short sequence of numbers again and again. For instance if zero was reached by the RNG it would stop generating different values entirely and stay at zero.

Some improvements to the middle square method were made and this allowed the generator to output nearly a million values before it became trapped in an undesirable cyclic state.

In 1959, another type of RNG known as "Algorithm K" was created [6]. Despite looking promising from the outset, Algorithm K, upon closing inspection, would be shown to have serious flaws.

It has a very small period of only three thousand numbers. Algorithm K proved that RNGs need to be crafted very deliberately and cannot simply be created by adding various arbitrary instructions and hoping for the best. Greater theoretical understanding was needed.

## 5.2 Reliable RNG Algorithms

One of the most reliable and widely adopted RNG is the Linear Congruential Method [6]. It was first introduced by Lehmer in 1949 and has four internal values that can be adjusted as needed. These include the modulus, multiplier, increment and starting value (also known as the seed). These numbers need to be methodically selected in order for a desirable output or random numbers to appear. A wrong selection of these numbers can cause the output to have a very small period, repeating only a handful of numbers again and again. There are actually two different variants of this generator, the multiplicative congruential method as well as the mixed congruential method. The difference between these variations is that the former uses an increment value equal to zero while the latter uses a nonzero increment.

### 5.3 Testing an RNG

One main attribute of random sequences that are desirable is that they do not repeat the same numbers again and again when they are used. What this means in practice, is that the period should be high enough so that no repetitions become visible during the intended use of an RNG. The period indicates how many values an RNG will output before reaching its initial state again.

If a person is asked to generate random numbers manually and write them down, the result won't actually be random numbers of very good quality. It's just part of human nature to want certain patterns.

On the flip side, if a person is asked to give their opinion on a sequence of random numbers the result is likely that they will find some patterns even though the sequence is a statistically sound random sequence.

This is because people are good at finding patterns and patterns help people remember things.

Knowledge of statistics is paramount when it comes to analysing the quality of random numbers. There are a virtually endless amount of statistical tests that could be performed on a sequence of random numbers. There are certain tests however that can be considered the best for testing RNGs that also have the useful property of being easy to implement as a computer program.

There are two main groups of tests that are used to check RNGs. The first of this group are empirical tests. These tests observe actual numerical outputs and make observations about them. The second class of tests are the theoretical tests. These do not work with concrete outputs but with mathematical formulas.

One statistical test that can be done is known as the Chi square test.

There is another very important test that is actually also quite good for testing linear congruential generators. It's one of the most powerful tests for RNGs because it has been used to test a large number of RNGs. Most RNGs with good randomness properties will pass the test while RNGs that do not output good random values fail the test.


This actually makes it one of the best of all RNG tests if not the best. The



spectral test is a fusion between empirical and theoretical testing. Its theoretical portion deals with the properties of sequence. While its empirical aspect comes from computationally examining results.

## 5.4 Inadequate RNGs

There were many implementations of LCGs, both multiplicative and mixed linear congruential generators that suffered from serious problems [5].

 Of the most well known broken generator was "RANDU". RANDU in turn inspired many other RNGs that were also seriously flawed. Being introduced in the 60's one of RANDU's main flaws was that it used a non-prime modulus. It also suffers from a lack of a full period. The famous computer scientist Donal Knuth called it "really horrible" [6].

Many flawed implementations of RANDU use a technique called controlled overflow. Despite the major flaws in RANDU, it continued to be featured in textbooks well into the 80's.

There were some attempts to improve RANDU by selecting another multiplier variable. This did not fix the problem because it still did not have the maximum possible period.

The random output here was so bad that some companies who used the RNG even took multiple outputs from it and manually randomized those again.

In 80's there were a number implementations like one proposed by Maryanski as well as that used by the Modula-2 system that were seriously broken. The main problem was that their period was incredibly short, being only thousands of values long. In one LISP text there was an RNG with a period of only 125.

In the 60's mixed linear congruential generators were expected to be possibly superior to multiplicative linear congruential generators like RANDU. This did not turn out to be true however and these days experts are against the use of most mixed generators. They still remained quite popular in academia for some time however and were featured in a number of textbooks.

## 6 Breaking RNGs

It can be argued that random number generators need to be as secure as any other cryptologic primitive [7]. The security of RNGs needs to be thoroughly verified due to their various sensitive use cases. These use cases range from generating session keys, initialization vectors and also generating random salts to hash passwords. Other applications include generating parameters for digital signatures as well as nonces for secure communication protocols. Often however, random numbers are not generated from reliable sources of entropy like thermal noise or Geiger counter clicks. Instead random values are instead created by pseudo random number generators.

What a PRNG does is collect a small amount of randomness from a input stream and then uses that randomness to generate a sequence of random values that have no surface level relationship to each other.

When an attacker wants to target a RNG, possibly one of their main goals is to use previous outputs to predict future ones. One aspect of the security of RNGs is a lack of general knowledge of attacking them. This knowledge of how RNGs can be compromised is critical however it comes to being able to secure them properly. RNGs are a desirable attack target for malicious actors because even if all other aspects of a system are hardened against attack, an insecure random number generator can lead to the entire system being compromised.

Insecure random number generators can be the result of selecting an insecure algorithm. They can also be rendered insecure as a result of a normally secure random number generator being used in a wrong way.

If a poorly designed RNG algorithm is used, there usually isn't anything that can be done but switching to another, more secure, algorithm [7].

The basic idea behind attacking an RNG is to gain the ability to predict or even control future RNG outputs [7].

One type of attack is the "Direct Cryptanalytic Attack". This attack relies on the attack being able to know how an RNG was used to generate random outputs. If the attacker cannot directly see RNG outputs this attack will not work.

Input based attacks happens when an attacker has knowledge or control of

RNG outputs. There are three different categories of input based attacks. Chosen input attacks often involve the attacker being able to control sources of entropy that are fed into RNGs. If an attacker knows how these sources of entropy are used, they may be able to cause random numbers to be generated that somehow give them an advantage.

Another type of input based attack other than the chosen input attack is the replayed input attack. With replayed input attacks the attack needs less control over the target system than with chosen input attacks.

The third subclass of input attacks is the known input attack. These attacks rely on the attack being able to guess or recover some internal state of the RNG.

The known input attack can often be the result of an attacker being able to predict some source of entropy that the system owners believe is random.

The final class of attacks is known as state compromise extension attacks. This type of attack occurs when an attacker is able to recover RNG state by means of gaining access to internal system state that regular users of the system are not given access to. This knowledge of internal state then allows attackers to predict future outputs and use this to their advantage. A very simple source of state compromise attacks can be the lack of an RNG seed that has enough entropy. If the attack knows with what state an RNG will start, this knowledge of the internal state will easily allow them to predict future outputs.

There are various types of state compromise attacks [7]. The first of these variants is the backtracking attack. The purpose of the backtracking attack is to somehow recover an RNG's internal state and use that to guess previously generated values.

A permanent compromise attack is somewhat similar to the backtracking variant. The main difference here is that the permanent compromise attack allows future as well as previous values to be known.

Another type of attack is the iterative guessing attack. This attack uses the knowledge of the state at a certain time along with outputs collected from the RNG in order to recover the RNG's state at a certain time.

The final attack type, a meet in the middle attack, is a combination of an



iterative guessing attack and a backtracking attack.

One RNG that is vulnerable to attack is X9.17 [7]. This RNG algorithm is used to generate DES keys and initialization vectors. It's also been used as a general purpose RNG.

X9.17 has been used in applications where very high security is paramount like in banking.

When it comes to direct cryptanalytic attacks, a cryptanalysis of triple-DES would be needed which is not yet known to be possible. This rules out the use of direct cryptanalytic attacks against X9.17.

Input based attacks against X9.17 could be possible in theory, but in practice require  $2^{63}$  outputs to be observed. This makes this kind of an attack an "academic" attack since it's only possible in theory.

One variant of attacks that does work is the state compromise extension attack. This type of attack works if an attacker can somehow get ahold of the X9.17 triple-DES key. If this key is found by the attack they will be able to recover the internal state of the X9.17 RNG.

This kind of attack is possible due to several flaws present in the RNG algorithm.

The first of these flaws is that the usable seed length is only 64 bits. Another flaw is the fact that the next seed that is generated is a function of a previous output.

Essentially, the attack can in theory brute force 64 bits to learn the internal state. Though there is a far more efficient way an attack can use.

The attacker can reduce the entropy of the RNG by making an educated guess about what range of values the timestamp that is used may have. The attack can know the nearest second of the timestamp.

If an attacker can obtain two successive values they can do a meet in the middle attack. This only requires them to search an eleven bit space of numbers.

Another possible attack that can be mounted against X9.17 is the iterative guessing attack [7]. Again, the vulnerability being exploited here is a lack of entropy due to a reliance on the timestamp. Here, unlike with the key compromise attack, the adversary merely needs to see a function of the output

rather than a raw output value.

The backtracking attack can also work here. Here, functions of RNG outputs are needed or two successive direct outputs.

In conclusion, the X9.17 random number generator is mostly secure against attacks except for ones that stop the timer or ones that use a recovered triple-DES key.

There are several ways in the insecurity of weak RNGs can be overcome [7].

One fairly straightforward method of increasing the security of an RNG is to simply send the outputs through a hash function. By hashing the outputs, the attack will not be able to recover the initial outputs without the serious effort of using the inefficient brute force technique.

Hashing an RNG's outputs improves insecure random number generators by quite a large amount but there is another improvement that builds on this technique. That is, the output that is hash can be combined with either the output of a counter or with a timestamp.

Another effective way of mitigating RNG attacks is by restarting the RNG state periodically. This is especially useful at preventing attack techniques that involve guessing and other attacks that cannot be performed with a trivial amount of computation. Though even if RNG are being frequently restarted, care still needs to be taken to seed these RNGs with sources that provide sufficient entropy. If an attacker can guess the initial seed, then restarting the RNG will not help. It could also be argued that in some cases frequently restarting an RNG could actually be detrimental. This is because a frequently restarted RNG means that the number of values an attacker needs to check to see if a certain value is the seed is far lower.

Finally, to improve the security of potentially weak RNGs it is paramount to ensure they are seeded with reliable sources of entropy.

When it comes to designing new RNGs, there are several factors that warrant consideration [7].

First, this new RNG needs to be built on a foundation of algorithms that are secure against cryptanalysis. It would be best if these primitives are theoretically proven to be secure.

Second, one needs to ensure that the entire secret internal state of the RNG

changes over time. This keeps the system from suffering from single state compromise.

Another technique, which was mentioned before, deals with reseeding. The internal state should be kept separate from whatever source of entropy is used. The system should only be reseeded if the source of entropy used is large enough. This prevents iterative guessing attacks.

Newly designed secure RNGs also need to be engineered in such a way that makes backtracking attacks very difficult. One straightforward mitigation that can be used here is the use of a one way function. Every few outputs the entire RNG state can be sent through this function to prevent backtracking.

Resistance to chosen input attacks should also be considered. To prevent these, one needs to be sure an attacker can never have both the RNG state and the input sequence.

One aspect of RNG design is performance. When it comes to building faster RNGs there may arise the need for designing completely novel algorithms. This is not small feat because most RNGs today are built on proven cryptographic primitives.

New RNGs could also benefit from the use of the reliability of theoretical proofs about their properties.

In conclusion, there are a number of different factors that need to be considered when it comes to the creation of secure RNGs. Though even for fundamentally insecure RNGs there are ways to quickly make them a lot harder for attackers to break. Hashing, especially, is one of such techniques. One can also conclude that RNG security is a topic that has not received a large amount of attention in research. Perhaps this is because software developers largely use RNGs as a sort of black box without much thought to their internal mechanisms. Attackers or security researchers in general though, can benefit greatly from being aware of what weaknesses random number generators can have and how these can be exploited.

## 7 Using an FPGA to Recover RNG State

The purpose of this section is to demonstrate how a real FPGA device can be used in the cryptanalysis of a toy, vulnerable on purpose, LCG random number generation algorithm variant.

The FPGA used for this experiment belongs to the **Lattice Ice40** family of FPGA's. The board that houses this FPGA is the **TinyFPGA BX board**.

This toy random number generator has the following purposeful vulnerability: the seed length is only 32 bits, small enough to make it easy to brute force.

The 32 bit seed can only have about 4 billion different values which is not very high when considering the number of computations modern hardware can perform per second.

In order to verify the FPGA's implementation is correct, an implementation in the C language will be used for comparison.

As mentioned in a previous section about best practices in FPGA design, the design workflow will first make use of a simulation. This simulation will be used to get the first working version of the RNG on the FPGA.



## 7.1 The C Implementation

```
#include <stdio.h>

int MODULUS = 993441; // m
int MULTIPLIER = 4001; // a
int INCREMENT = 60211; // c

int gen_rand(int prev) {
    // TODO - add the modulus back
    // return ((prev * MULTIPLIER) + INCREMENT) % MODULUS;
    return ((prev * MULTIPLIER) + INCREMENT);
}

int main() {

    int seed = 96; // aka the starting value or X0

    int out0 = gen_rand(seed);
    int out1 = gen_rand(out0);
    int out2 = gen_rand(out1);

    printf("first outputs for seed: %i \n%i \n%i \n%i \n", seed, out0, out1, out2);
}
```

This C design takes a given seed (hardcoded to 96 in the above example) and generates three random numbers.

This implementation has a number of significant properties. First of all, it only support a seed with 32 bits of length. The seed is purposefully limited in size for demonstration purposes.

This algorithm is actually not even a real LCG though it can be trivially converted to be one.

The reason a full fledged LCG was not used was due to the issues the modulus operation caused for the non-simulated, real world FPGA program.



Another noteworthy attribute of this C operation is that no special effort has been made to select secure constants.

Donald Knuth offers a great amount of detail in his book [6] on how secure LCG constants can be found.

## 7.2 The FPGA Simulation

The purpose of the FPGA design would be through all possible 32-bit seeds starting from zero. The design would take the first three known random values (generated by the C program) for every possible seed and return the seed once a value sequence of number was discovered.

As recommended in a previous section, most of the development of the Verilog code was done in a simulated environment. This was especially important for this project because the only way in which the developer could extract any data from the FPGA was by checking a blinking light.

For a more involved computation, the FPGA could be configured to send data over ethernet. This data could in turn then be read using a **program** like Wireshark.



Though because most work was done in a simulation, this lack of detailed outputs did not cause significant problems or slowdowns in the workflow. Another aspect of the workflow that was somewhat primitive was a lack of verification. The simulation merely consisted of assertions combined with 120 iterations.

The simulation itself was quite convenient for the development workflow because it took less than two seconds.

Also, the Verilog code in general suffers from a lack of developer experience in Verilog specifically, but also a lack of hardware design experience.

## 7.3 The Real FPGA Program

When it came to programming an actual FPGA with this project, there were a few issues that posed significant roadblocks to development.

The most serious roadblock came in the form of the synthesis tools limiting the clock speed of the overall design to a mere 2 - 5 Mhz.

Attempts to slow the system clock to this lower frequency were met with significant roadblocks. The Ice40 model of FPGAs comes with a logic element known as a phase locked loop. These PLL allow an input clock to be converted to a clock with a different frequency.

However the specific PLL used by the Ice40 FPGA does not allow clock speeds under 16 Mhz to be generated. This led to naive attempts to generate a lower clock speed using generic Verilog operations. These attempts simply did not work.

Luckily, beside reducing the clock speed on an FPGA, a slow design can be sped up by removing functionality. This approach ultimately made it possible to forgo all PLLs and frequency modification methods.

One possible compromised that was considered was the radical step of simply halving the size of all numeric values from 32 bits to 16 bits.

This alone was not enough however and the max allowed frequency was still too low even after this change was made. Another change that was made was to reduce the depth of random values that would be scanned for each seed. This, combined with the bit reduction method still proved not to be enough.

However, there was still a ray of hope, a compromise that could be made to radically simplify the design. This final method took some effort to find however. The reason it wasn't obvious from the start, lies in the fact that the Verilog language greatly obscures what the low level logic block layout is. Some operations are simply far more expensive but it takes foreknowledge of this or simply trial and error. One good rule of thumb for how complex a design is is to simply note how long the synthesis and routing programs take to execute. Another more concrete approach to knowing a design's complexity is to examine the output logs for how many logic elements were used for the design.

The operation that was causing the huge slowdown for the FPGA turned out to be the modulus operation. As mentioned in previous sections of the paper, some operations, often common ones like addition and multiplications, have dedicated logic blocks available on the FPGA. Though this did not appear to be the case for the modulus operation.

Without the modulus operation, the FPGA could run this design at it's standart 16Mhz. That is, it would do 16 million iterations per second. With the design now working on a real FPGA, more attempts were made to change the clock frequency. Except this time, the clock frequency was increased. For unknown reasons, and without protest from the synthesis tools, these higher clock frequencies simply caused the design to fail silently and were therefore not used.

```

module lcg (
    input CLK,      // 16MHz clock
    output LED,     // User/boot LED next to power LED
);

    // scans expected rng outputs
    // the led will light up if a valid seed is found

    reg done;

    lcg_guess lcg_guess0 (
        .CLK(CLK),

        .done(done),

        .MODULUS(993441),
        .MULTIPLIER(4001),
        .INCREMENT(60211),

        .done(done),
        // .valid_seed(valid_seed),

        // check if the fpga is doing real calculations by
        // setting a wrong sequence value and ensuring the light stays off

```

```

        // .expected_v0(0),
        .expected_v0(444307),
        .expected_v1(1777732518),
        .expected_v2(242022553),

    );

    assign LED = done;

endmodule

module lcg_guess(
    input CLK,

    input [31:0] MODULUS, // m
    input [31:0] MULTIPLIER, // a
    input [31:0] INCREMENT, // c

    input [31:0] expected_v0,
    input [31:0] expected_v1,
    input [31:0] expected_v2,

    output done,
    output [31:0] valid_seed,
);

    reg [31:0] scan_seed = 0;
    reg [31:0] scan_v0;
    reg [31:0] scan_v1;
    reg [31:0] scan_v2;

    initial
        done = 0;

```

```

always @(posedge CLK) begin
    // TODO - add the modulus back
    // scan_v0 = ((scan_seed * MULTIPLIER) + INCREMENT) % MODULUS;
    scan_v0 = ((scan_seed * MULTIPLIER) + INCREMENT);
    scan_v1 = ((scan_v0 * MULTIPLIER) + INCREMENT);
    scan_v2 = ((scan_v1 * MULTIPLIER) + INCREMENT);

    if (expected_v0 == scan_v0 &&
        expected_v1 == scan_v1 &&
        expected_v2 == scan_v2) begin

        done = 1;
        valid_seed = scan_seed;
    end

    scan_seed = scan_seed + 1;
end
endmodule

`ifdef FORMAL
module testbench(input CLK);

    reg done;

    reg [31:0] valid_seed;

    reg [31:0] counter = 0;

    reg [31:0] expected_valid_seed = 96;

    lcg_guess lcg_guess0(
        .CLK(CLK),

```

```

        .MODULUS(993441),
        .MULTIPLIER(4001),
        .INCREMENT(60211),

        .done(done),
        .valid_seed(valid_seed),

        .expected_v0(444307),
        .expected_v1(1777732518),
        .expected_v2(242022553),
    );

    always @(posedge CLK) begin

        // 100 > 96 so we just scan deep enough
        if(counter == 100) begin
            assert (done == 1);
            assert (valid_seed == expected_valid_seed);
        end

        counter = counter + 1;
    end
endmodule
`endif

```

## 8 Conclusion

This experiment showed that FPGAs can be quite useful for cryptanalysis, specifically for breaking random number generators.

Running at 16Mhz, it would only take this specific model of FPGA a mere 4 minutes to search the 4 billion permutations possible in 32-bit space.

And this FPGA, which costs only around 15 dollars, pales in comparison

when compared to the resources FPGAs costing thousands of dollars posses. Of course, the actual calculations this FPGA design performed, are not strictly speaking a real linear congruential generator do to the lack of a modulus. Though adding this modulus back in is as simple as changing a few lines of commented out code. It is also very likely that higher end FPGA models will have dedicated modulus logic blocks.

Another than using a modulus operation and getting a more powerful FPGA there are additional ways of improving this project.

The work of searching the possible seeds of a random number generator could easily be split among multiple different FPGAs. The average software developer does not own an expensive FPGA but they could rent FPGAs on the cloud from cloud providers like Amazon. Another improvement to the algorithm could be looking for mathematical methods that could be a lot more efficient the a simple brute force method.

## References

1. Kuon, Ian, Russell Tessier, and Jonathan Rose. FPGA architecture: Survey and challenges. Now Publishers Inc, 2008.
2. Zeidman, Bob. "Introduction to CPLD and FPGA Design." Embedded System Conference, San Fransisco. 2004.
3. Mueller, Rene, and Jens Teubner. "Fpga: What's in it for a database?." Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. 2009.
4. Dooley, John F. History of cryptography and cryptanalysis: Codes, Ciphers, and their algorithms. Springer, 2018.
5. Park, Stephen K., and Keith W. Miller. "Random number generators: good ones are hard to find." Communications of the ACM 31.10 (1988): 1192-1201.
6. Knuth, D.E. The Art of Computer Programming. 3rd Ed. Addison-Wesley, Reading, Mass. 1981.
7. Kelsey, John, et al. "Cryptanalytic attacks on pseudorandom number generators." International workshop on fast software encryption. Springer, Berlin, Heidelberg, 1998.