CRISPR/Cas-9 Guide-RNA Generation and FPGA Technology

Peter Ferrarotto

Contents

Background - Crispr 2
What is CRISPR? 2
Guide RNA 2
Guide RNA Generation 3
The Issue 4
An Alternative 5
The Design 6
Specification 1 – The Searching Algorithms 6
Specification 2 – SD I/O 9
Specification 3 – Control 10
Final Criteria – Grading 12
Moving Forward 12
Acknowledgements 14

Background - Crispr

What is CRISPR?

In multiple facets of biology, CRISPR – Clustered Regularly Interspaced Short Palindromic Repeats – has become a mainstay of various methods to edit and analyze the genome of various species.

Initially discovered as a defense and propagation system in bacteria, it has been adapted to the needs of researchers across multiple fields. By using the enzyme Cas9 in conjunction with the CRISPR system – a combination referred to as CRISPR/Cas9 – the genome can be cut, and various other methods can then be used to repair the genome, and introduce the desired DNA sequence into the organism.

This is primarily used with stem cells – the DNA is extracted, cut, and the gene for a disease or disorder (such as Alzheimer's, Huntington's Chorea, or the like) is introduced. Then, the cell is specialized into either a neuron, or a fibroblast (skin cell), or a multitude of other types of cells, and can then be examined in closer detail to see how the disorder impacts the body.

Furthermore, these modified cells can then be introduced to various drugs used to treat the disease or disorder, allowing for researchers to determine how the patient's body will react before administration of pharmaceuticals, rather than waiting for side effects in the patient.

To make these cuts and facilitate this research, the Cas9 enzyme needs to be instructed as to where to make the cut in the genome. This is done with Guide RNA.

Guide RNA

Guide RNA, or gRNA, is a sequence of nucleotides constructed in the laboratory. Since it consists of RNA and not DNA, the nucleotides used are the compliments – Guanine is replaced with Cytosine, Thymine is replaced with Adenine, and Adenine is replaced with Uracil. (In RNA, there is no Thymine – only Uracil.)

Unfortunately, gRNA cannot just be arbitrarily selected – there is a process for generating the sequence that must be used to result in the most accurate cut. This is because Cas9 does not care about being exact – instead, it will cut at approximate locations. These locations are referred to as "off-targets".

Off-target sequences are defined by their mis-matches – the locations in the sequences where there is a discrepancy. For example, between the sequences AGTA and AGTC, there is a mis-match in the fourth position between A and C.

Each of these off-target sites, referred to as "hits", are scored, and the score of all off-target sites (and number of off-target sites) are used to determine the sequence's total score. This is done when the researcher generates the gRNA needed to make the cut they desire.

Guide RNA Generation

To generate the Guide RNA, researchers will utilize an online tool – such as MIT's gRNA generation tool – that takes a particular sequence, usually around 500 nucleotides long, and extract the guide sequences from it. This is done by finding PAM sites – sites where a particular string of nucleotides are found.

The PAM sequence differs per the enzyme being used for the CRISPR cutting – for Cas9, the PAM sequence is 'NGG' – N being any nucleotide. This means that in a given sequence, a PAM site consists of three nucleotides, the first of which does not matter, but the last two of which *must* be two consecutive Guanines. The bulk of the guide sequence is twenty base pairs long, and will always be appended with the three base pairs of the PAM site – meaning that each guide is 23 base pairs long.

After this, each guide sequence is graded by finding and grading each of the off-target hits in the target genome. An off-target will actually consist only of twenty nucleotides – the last three actually don't matter for the off-target cutting, as they're only used for the Cas9 enzyme to properly grab onto the guide sequence. However, the proximity to the PAM sequence does matter, and is reflected in the formula used to score each off-target hit:

$$S_{hit} = \prod_{e \in M} (1 - W[e]) \times \frac{1}{\left(\frac{19 - \bar{d}}{19} \times 4 + 1\right)} \times \frac{1}{n_{mm}^2}$$

M = [0, 0, 0.014, 0, 0, 0.395, 0.317, 0, 0.389, 0.079, 0.445, 0.508, 0.613, 0.851, 0.732, 0.828, 0.615, 0.804, 0.685, 0.583]

Here, n_{mm} is the number of mis-matches, $\bar{d}$ is the mean pair-wise distance of mismatches, and M is a set of experimentally-determined values that pertain to the impact of each mismatch position on the score of the off-target.

The score of the off-target is how likely the Cas9 protein is to make a cut in that position – the higher the score, the more likely the enzyme is to make a cut there. In this case, a lower score is preferable, as it means that the enzyme is more likely to make a cut in a more accurate location. (When there is only an off-target and no on-target, the researchers generally will discard this guide due to its unreliability.)

When scoring off-targets and extracting information, researchers are also informed as to where on the genome this off-target cut is most likely to happen, which allows for researchers to reliably select guide sequences with a predictable cut location.

After the off-targets are scored, the guides themselves must be scored, and this is done by the following formula:

$$S_{guide} = \frac{100}{100 + \sum_{i=1}^{n_{mm}} S_{hit}(h_i)}$$

The Issue

While this is an effective way to reliably generate gRNA sequences, there comes with it an inherent issue – several, in fact. There is a bottleneck issue, a location issue, and an overhead issue.

The first issue is the most prominent one – the bottleneck. All of these requests for gRNA are offloaded onto an application's server – such as MIT's Crispr gRNA generation server. When too many requests come in at once, they must be slowly and gradually let through the bottleneck that this influx of demand creates. As stem cell and genetics research continues to expand and Crispr is used more and more frequently, this issue will only continue to grow, and organizations such as MIT will have to make costly expansions to their infrastructure to accommodate for this bottleneck.

Next, the location presents a problem – since all these gRNA generation applications perform their applications on servers, this means that the researcher never has direct access to it. While this is similar to the bottleneck issue, it also results in being completely unable to generate gRNA when internet connection is not readily available, or when the server is undergoing maintenance.

Lastly, there is a significant overhead created by this process. In order to find these sites, the BLAST software library is used. The BLAST library is an extensive set of algorithms that quickly and efficiently search for DNA sequences in a handful of organism genomes, using an online database to quickly find known locations for particular sequences. While efficient, this creates a significant overhead, as the BLAST software requires a significant amount of processing power to run efficiently enough to be a viable solution. This is part of why servers are relied on so heavily for this process, as it is exceedingly expensive to build a computer setup powerful enough to reliably run this software.

An Alternative

Overall, the generation of gRNA is riddled with issues that require a significant investment to solve. However, there is a much less expensive alternative that can be explored – the design and manufacturing of hardware specifically made to generate guide RNA.

There are several boons that this alternative would grant – first off, hardware is significantly faster than software, as software has to essentially wrestle with the hardware it's installed on to run reliably, as this hardware also has to run an operating system, take input and generate output, and various other operations. On the other hand, specialized hardware does not need to wait for resources or compile its processes into commands – instead, it is a near-instantaneous movement of current through the existing pathways that generate the output the user desires.

Furthermore, if it can reliably manufactured, this hardware solution could be installed in research labs to relieve the reliance upon servers, and allow for a faster and more self-sustaining environment.

Despite all these possible benefits, there is one major caveat – designing and testing hardware is expensive and time-consuming, and when the inevitability of the hardware needing to be adjusted occurs, then the entire process must restart. However, there is a solution for this issue, as well – by prototyping the hardware design on an FPGA board.

An FPGA board utilizes an FPGA chip – a Field Programmable Gate Array chip. An FPGA chip has hardware that can be reconfigured an unlimited number of times, meaning that an infinitude of hardware designs can be tested on it (within the specifications and limitations of the chip and the board it is incorporated in).

In order to prototype this alternative, I've decided to utilize the Nexys 4 DDR FPGA board, produced and sold by Xilinx. I used this because it has both DDR memory – allowing for an expanded RAM – and a micro SD card slot. This micro SD card slot allows for the board to take input from, and store output on, the card without human intervention.

In order to design the hardware needed, I created a list of specifications I would have to adhere to. They are:

1. The design *must* be able to find locations in a sequence based upon input

2. The design *must* be able to access the micro-SD card for I/O
    a. Also, *must* be able to read the files in a way that it can easily search for data

3. The searching algorithm/hardware *must* be controllable, with the ability to accept input and continuously run over a large set of data

4. The design *must* be able to reliably score guides and off-target hits

The Design

Specification 1 – The Searching Algorithms

Before working on any kind of input and output, we should first determine what to do with the input data.

While it would be best to implement the BLAST search library on the FPGA board, this is actually impossible due to the sheer complexity, and the limitations of the board (and current FPGA technology in general). Since the BLAST algorithm is open-source, it could be made into hardware given enough time and space on the board, but it is about as complex as – if not more so than – the standard template library for C++, which still has yet to be implemented on Arduino platforms.

Since the BLAST algorithm is out, the next option is to try an optimized searching algorithm converted into hardware – however, all optimized searching algorithms require a pre-sorted set, such as binary search or merge search. So, a linear search is required – but rather than just go one nucleotide at a time, I've decided to use two alternate forms of a chunk-based linear search.

Unfortunately, I can't just call "find" and return all the locations I want when I'm using hardware. Furthermore, I can't easily loop through the set of data I'm trying to search through, as looping syntax in hardware languages is only used to generate repetitive hardware, such as consecutive gates or registers.

Finally, I have to be able to hold the sequences I'm searching through in a convenient and efficient manner.

The first issue I decided to address is the issue of holding the sequence data efficiently. DNA data is stored as ASCII text, and each character is represented by eight bits. To lessen the overhead this creates, I utilized four bits and a positional logic system to represent each base pair – 0001 for A, 0010 for G, and so on and so forth. In the field of genetics research, when there is a nucleotide that is not definite, it is represented by an extraneous character – usually 'X', for being anything. As such, these four bits can be used in combinations to reflect this uncertainty.

Through this, the overhead is halved, while retaining the amount of data being represented.

As mentioned earlier, two chunk-based linear search algorithms were employed. The first uses a chunk size of eight bits, or two of the positional-logic based nucleotide values. This is used to find the PAM sites – the two subsequent Guanine nucleotides.

The second uses a chunk size of sixteen bits (four nucleotides) to find off-target sites. It uses five of these chunks to find the twenty nucleotide off-target sites, ignoring the three nucleotides that make up the PAM site (as mentioned earlier).

To achieve this, I had to design two distinct state machines for each algorithm, both of which require various input and output registers, as well as registers for storing input and performing calculations. In order to achieve this, I had to select between the two primary hardware languages – VHDL, and Verilog.

Initially, I wanted to use VHDL, as it's the language I'm most familiar with. However, it cannot quickly perform calculations such as addition, subtraction, or comparisons on registers, whereas Verilog can do this on any sized wire or register. As such, I selected Verilog as the most appropriate language for this application.

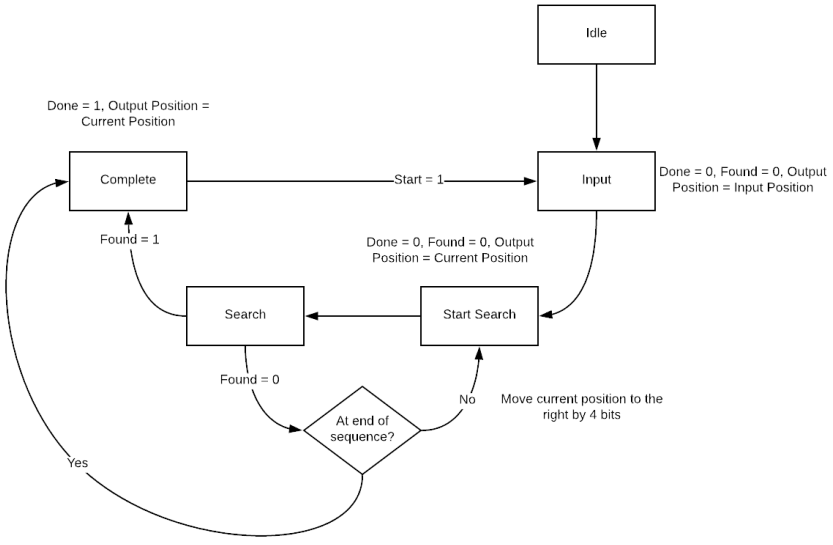
Finally, the search modules need input and output registers. First off, the PAM search module needs the following registers:

1. Input of 500 base-pairs (the typical size for guide sequence generation) – each represented in 4 bits
    a. One input register of 2000 bits

2. Can only return one value at a time, and also needs a position to start at
    a. Must represent up to the number 1999 – the final position in the 500 base pair sequence (registers are zero-indexed)
    b. 11 bits for location of found sequence, and 11 bits for starting location of search

3. Also needs multiple control bits
    a. Generic – clock bit and reset bit
    b. Input – needs a 'Start' bit
    c. Output – needs a 'Done' and 'Found' bit

Finally, I designed a module with the following state machine:



Next, I had to design the search module for off-targets. For this module, I needed the following input and output registers:

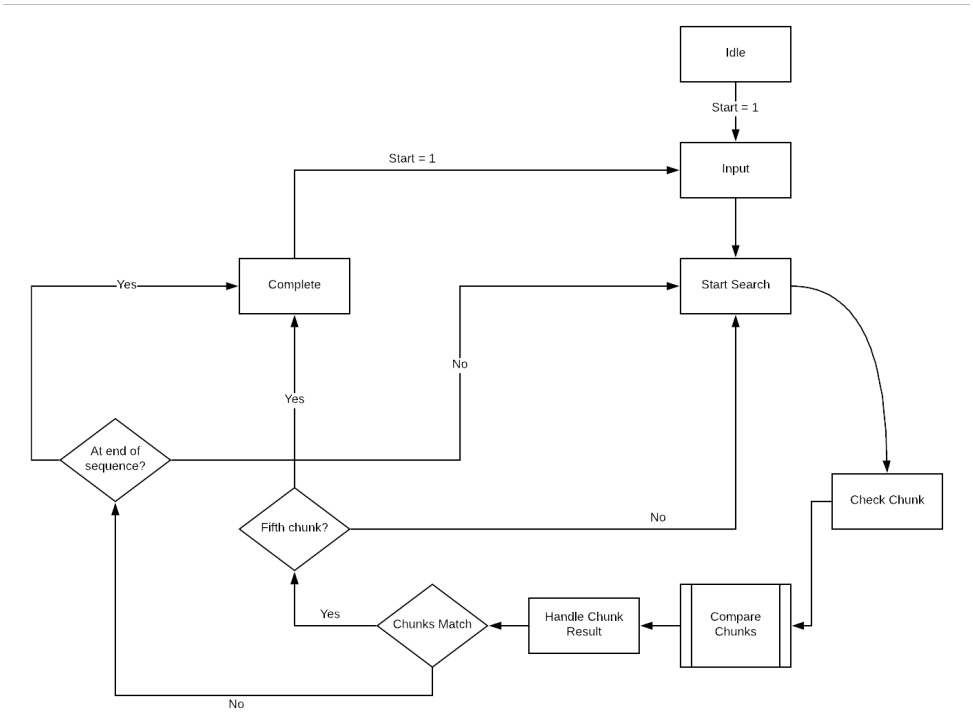
1. I decided to make a maximum input of 1000 base pairs due to a fear of register overflow
    a. This needs 4000 bits – 4 for each nucleotide

2. Need 20 base pair guide sequence
    a. 80 bits

3. Two position values – one for start position, one for the location of the off-target
    a. 12 bits for each – need to represent up to 3999

4. Control signals
    a. Generic – clock and reset bits
    b. Start bit
    c. Done and Found bits

Finally, the off-target module has the following state machine:



Specification 2 – SD I/O

The primary challenge of this entire project is handling input and output on an FPGA board, as how I decide to do it will reshape the design of the system immensely.

While I could directly access the SD card through hardware, I would need to interface directly with the memory on the SD card, and wouldn't be able to create or read specific files. So, it would be better to use a file system, such as the generic FatFS file system. Recreating this, however, would be a project all of its own.

As such, I decided to use the Microblaze softcore processor, which is the only softcore processor that will run on the Nexys 4 DDR. It can implement generic modules provided by Xilinx – such as an SD card module – and even provides a basic kernel for executing C/C++ code.

In order to execute this C/C++ code, the Xilinx SDK application loads the softcore processor's hardware onto the FPGA board, runs the "bootloader" (which starts up memory allocation and other crucial processes in the processor), and waits for the code to be sent over the USB connection. The user starts the process of running the code on the processor, and Xilinx takes over compiling it into assembly that Microblaze can understand.

While capable of utilizing custom hardware that will make certain processes much faster, the main drawback of the Microblaze processor is that the compiled code runs very slowly – and accessing memory is slow as well.

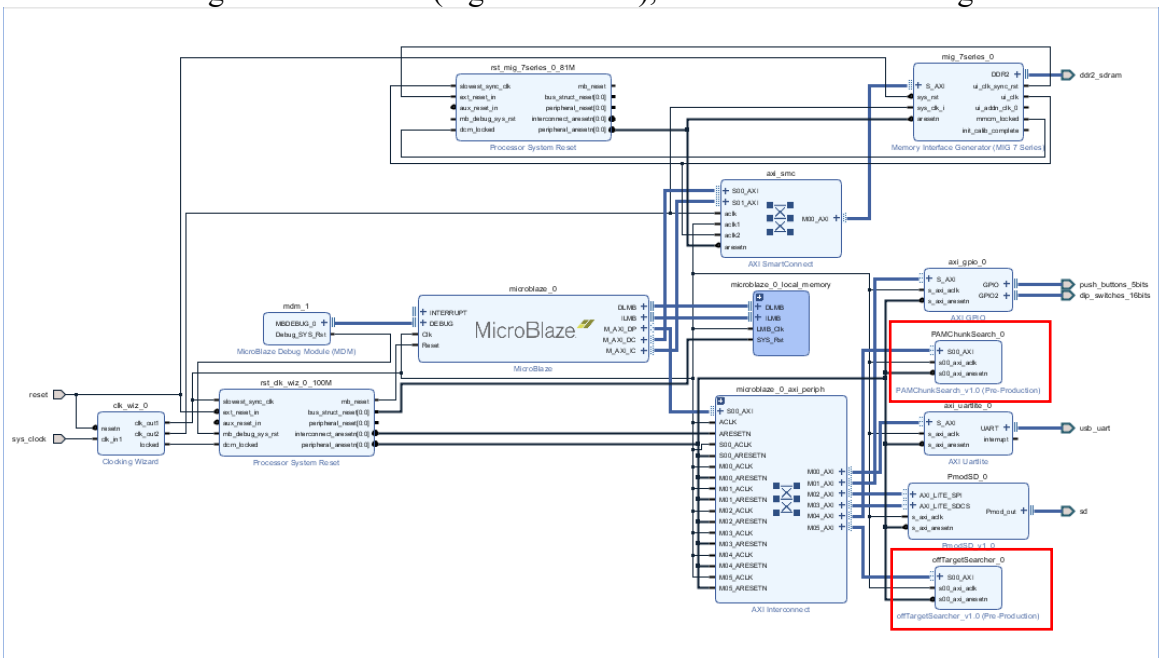
Specification 3 – Control

Now that Microblaze is being used, the system has to be controlled using C++ - C++ is required as it is the only language that will be able to properly utilize the SD module to read and write files on the SD card.

In order to ensure continuous execution based upon input from the SD card, the search modules have to be controlled by Microblaze, and the C++ code. In order to do this, I had to make the searching hardware into IP Modules, which is what is used in the block design in Vivado. This provides a framework for interaction between C/C++ and VHDL/Verilog code.

An issue with this is that IP modules use 32-bit registers as input and output, up to a maximum of 512 registers per module. As such, I had to write the code so that the software would split the input up into 32-bit chunks and write those to the registers; while the hardware will take that segmented input and put it all into one continuous internal register.

After making the IP modules (highlited in red), the final hardware design is as such:



After all of this, I still have one major issue to address with regards to the control and continuous execution of the search modules – the size of the data I'm trying to work with.

As I earlier established, the data sizes I've chosen to work with are 500 nucleotides (2,000 bits) from which the guide sequences are extracted, and 1,000 nucleotides (4,000 bits) from which off-targets are extracted. I chose these values because they are well within the safe operation limits of the Nexys 4 DDR board, and will not compromise the board's ability to actually run the hardware.

However, the issue comes when I attempt to search through the genome of any organism. Most organisms have absolutely massive genomes – humans have, on average, 3.1 million base pairs, mice have 2.7 billion, and some amoeba have up to 670 billion. With 4 bits for each

nucleotide, this comes to 12.4 billion bits for the human genome – which the board is completely incapable of containing.

As such, for this proof of concept, I've elected to utilize the shortest genome on record – that of Carsonella Candidatus Radii, which has only about 170,000 base pairs. However, this will still require over 600,000 bits, and per my own limitations, the current hardware only accepts 1,000 bit segments of the organism's full genome.

So, my solution is simple – I utilize python to split the complete genome into distinct files, each containing 1,000 base pairs. To ensure I don't miss any off-targets that span overlaps, I start the next sequence at the 981st position, ensuring that it does not miss any viable off-targets. I chose 981st as, if I had done 980th, it could possibly find the same off-target twice.

To ensure the effective use of this data, I programmed the code to pre-load all of these files onto the board's memory. Since the board has a memory of 4 Megabytes, this amount of data being stored in memory is feasible – and when I'm looking for off-targets, all I have to do is pass the next segment into the registers until I reach the end, rather than opening the files every time I need to switch the segment I'm looking at.

Final Criteria – Grading

For the final piece of my design, while I could potentially grade the off-targets in hardware, I decided to stick with keeping it in the software – this is because there are very complex operations that occur in the grading function, and rather than struggle to implement those complexities in hardware, I decided to let the C++ compiler take care of that for me. For now, this is sufficient for the process, but is subject to change in the future.

Moving Forward

While I was able to build a reliable platform that will continuously run over large sets of data and score the guides predictably and accurately, there are still many issues that would need to be addressed in the design before it could be made into a viable product for use in the laboratory, and they are:

1. Microblaze is only an emulation of a computer/kernel – as such, it is much slower at executing instructions than an actual computer
    a. Furthermore, it disallows multithreading or forking – only one thread and one process, slowing the execution of code further

2. There isn't much to benchmark this against – this is because MIT's proprietary software does not offer Carsonella as an organism from which to generate guide sequences, and to try to extract performance metrics from it would require direct access to their underlying software
    a. This software is also highly dependent upon hardware, and would need to be run on their computers directly
    b. Just running the process on their site is not suitable as the run time is highly dependent on connectivity

3. Lastly, when passing subsequent segments into the off-target module, a slight bottleneck forms due to the necessity for register writing and some minor I/O operations, which slows performance even further.

    To possibly address these issues, I could make the following changes (respectively):

    1. I could switch from an FPGA board to an FPGA shield, and put that on an Arduino Yun
        a. The Yun has an on-board Linux processor, removing the need for a slow soft-core processor
        b. This could significantly simplify the necessary hardware designs, and allow for larger searching modules

    2. Using an Arduino would allow for larger storage space, which means I could run a benchmark by using files containing the segmented genome of one of the example organisms from MIT's site

3. I could redesign the hardware for off-target searching with a higher volume of data traffic in mind
    a. On an Arduino, could have multiple instances of off-target modules on the FPGA shield to allow for threading and multiple processes running at once
    b. Possibly add the ability to hold the entire organism's genome in the hardware module
        i. This would require error bits for overflow or data loss

Overall, due to the highly experimental nature of both FPGA and Crispr technologies, a lot of these designs and concepts may soon no longer be applicable, and would require a complete reworking from the ground up. Be that as it may, I believe it is still worthwhile to research how to make the current process more efficient, given that it is so widely used and embraced throughout the research community.

Acknowledgements

- https://www.crispr.mit.edu/about
- https://www.cam.ac.uk/subjects/crispr
- https://www.addgene.org/crispr/guide/
- Dan Paull – New York Stem Cell Foundation