ECE1373S - Accelerating Gene Sequencing with FPGA Technology

Meysam Roodi, David McTavish, Arash Moghimi, Leon Xiang Li

May 3, 2016

1 Introduction

Gene sequencing is the procedure of searching small portions of human gene in the whole or part of human reference genome. Gene small portions are usually referred to as short reads. The reference genome as well as short reads are composed of four possible protein types; A, C, G and T. The protein types are structured in pairs in a gene, called base pairs. Therefore, gene sequencing could be looked at as a string matching problem where the strings are created from a four letter alphabet set. The human reference genome is 3.2 billion base pairs long. Short reads usually contain between 200 to 500 base pairs.

There are several algorithms of gene sequencing. FM-index is an efficient and popular gene sequencing algorithm which finds short reads in a reference sequence in a time proportional to the length of short reads.

FM-index algorithm first creates two tables from the reference sequence. In order to construct the tables, FM-index first finishes the reference sequence with a $ character and then lists all possible rotations of the reference sequence and also sorts them. This is shown in the simple example of Figure 1. The $ special character is assumed to be smaller than the rest of characters. The last characters of the sorted list are attached together to create a new sequence whose length is equal to the length of the original reference sequence. The tables are created using this new sequence which will be referred to as Borrows-Wheeler Transform sequence (BWT sequence.)

The two tables of FM-index algorithm, I-table and C-table, are constructed using the BWT sequence. Each entry of the I-table , I(x), shows the number characters in the BWT sequence that are smaller than x (figure 2). For example, the value of 5 for I(C) shows that there exists 5 characters that are smaller than C in the BWT sequence. These 5 characters include the $ finishing character and four A characters. C(x,n) represents the number of x characters that could be found in the BWT sequence from 0th to nth position.

In order to search for a short reads having the two tables, FM-index initializes two top and bottom pointers. The top initial value is zero and bottom pointer initial value is equal to the length of the reference sequence. FM-index updates the value of top and bottom pointers through the following formula using the short read characters from right to left. If at any point of time the bottom pointer becomes less than or equal to the top pointer, the short read is not found in the reference sequence.

$$Top_{new} = C(x, Top_{current}) + I(x)$$

$$Bottom_{new} = C(x, Bottom_{current}) + I(x)$$

Original String: GCTAATTAGGTACC$	
Rotations:	**Sorted Rotations:**
gctaattaggtacc$	$gctaattaggtac – C
ctaattaggtacc$g	aattaggtacc$gc – T
taattaggtacc$gc	acc$gctaattagg – T
aattaggtacc$gct	aggtacc$gctaat – T
attaggtacc$gcta	attaggtacc$gct – A
ttaggtacc$gctaa	c$gctaattaggta – C
taggtacc$gctaat	cc$gctaattaggt – A
aggtacc$gctaatt	ctaattaggtacc$ – G
ggtacc$gctaatta	gctaattaggtacc – $
gtacc$gctaattag	ggtacc$gctaatt – A
tacc$gctaattagg	gtacc$gctaatta – G
acc$gctaattaggt	taattaggtacc$g – C
cc$gctaattaggta	tacc$gctaattag – G
c$gctaattaggtac	taggtacc$gctaa – T
$gctaattaggtacc	ttaggtacc$gcta – A
Burrows-Wheeler Transform: CTTTACAG$AGCGTA	

Figure 1: sorted list of all possible rotations of a reference sequence, borrowed from [1]

The figure 3 illustrates an example with the details of how the top and bottom pointers change while traversing the characters of a short read of length 4.

I-table

A	C	G	T
1	5	8	11

C-table

Index	BWT(Q)	A	C	G	T
0	C	0	0	0	0
1	T	0	1	0	0
2	T	0	1	0	1
3	T	0	1	0	2
4	A	0	1	0	3
5	C	1	1	0	3
6	A	1	2	0	3
7	G	2	2	0	3
8	$	2	2	1	3
9	A	2	2	1	3
10	G	3	2	1	3
11	C	3	2	2	3
12	G	3	3	2	3
13	T	3	3	3	3
14	A	3	3	3	4
15	Total	4	3	3	4

Figure 2: FM-index I-table and C-table, borrowed from [1]

This project conducts a Burrows-Wheeler transform on a Sequence of DNA as well as a short read sequence that is being searched for inside of the DNA. The tables that are generated from the transform are then run through an FM index search in order to determine of the short read exists inside of the original DNA.

We have taken advantage of both the processor as well as the programmable logic of the Mini-ITX board and partitioned parts of the project to each. The Burrows-Wheeler transform is conducted in the ARM processor while the FM indexing is done in the programmable logic. The system can be interfaced with using a PC and a serial communication application such as Terra-Term.

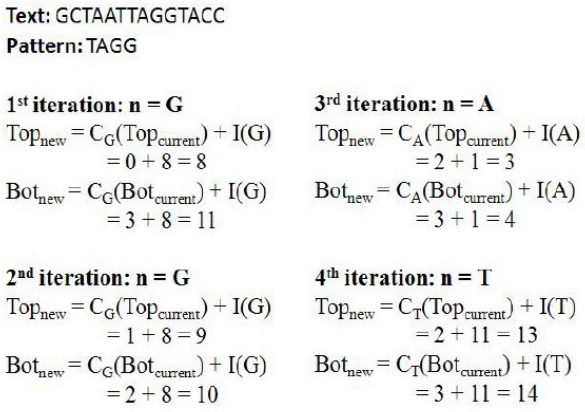

Figure 3: FM-index search process example, borrowed from [1]

2 Current Status

We have completed the design that we had initially planned to complete with a few small modifications.

2.1 Completed Functions

- ARM Processor and Programmable Logic Interface
  - The Petalinux operating system was run on the hardened ARM processor.
  - The software that would generate the Burrows-Wheeler transform tables was written and run on the ARM processor.
  - An application was written which could transfer the tables that were created from the processor's memory to the programmable logic's memory.
- Burrows-Wheeler Transformation Algorithm
  - The software implementation of the BWT algorithm was completed and tested out in a Linux environment. The program was also successfully loaded into the Linux system running on the ARM processor on the Mini-ITX board.
  - The stand-alone version of the software is able to create and write 6 different tables - one I-table, the four C-tables for A, T, G and C base pairs in a DNA sequence, and a suffix table (s-table) for finding the exact location where a match occurs.
  - The software is successfully integrated and is able to write the tables to desired memory locations where the tables will be stored.
- Sequencer Module running FM Index Algorithm
  - The sequencer is fully implemented and tested using Vivado HLS tool. The sequencer receives a short read and its length and searches for the short read in a reference sequence. As for the reference sequence, the sequencer only has access to the Borrows-Wheeler

transform of the reference sequence. The figure 4 shows the block diagram of the HLS sequencer. It has four interface to the four C-tables and an interface to the I-table.

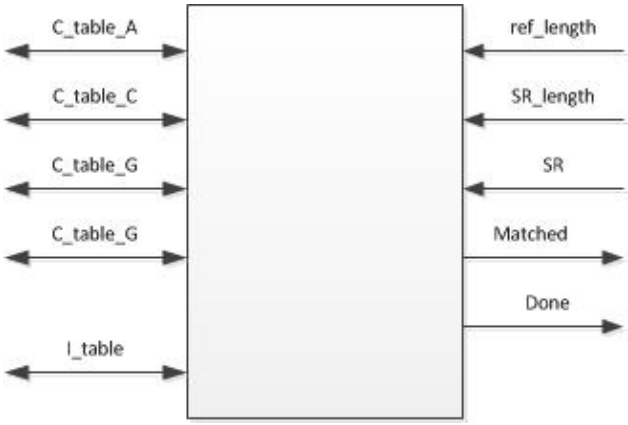

Figure 4: Vivado HLS block

- Control of Sequencer Module from Arm Processor
  - Three copies of the sequencer can run in parallel matching different short reads with the same reference sequence
  - Controlled through user-space drivers
- Communication of Results to User
  - The user can create reference sequences and short reads as text files. They can be stored on a USB drive and mounted on the Linux file-system to be accessed by the programs
  - The results of the matching for each short read are displayed on a terminal console

2.2 Future Improvements

- Arm Processor and Programmable Logic Interface
  - Could set up a DMA module to do streaming data transfers from Arm's DDR to programmable logic's DDR memory
- Further Improvement for Burrows-Wheeler Transformation Algorithm
  - Possible improvement in reducing the memory consumption of the BWT software during runtime by using four bits, instead of eight bits, to represent a character in the sequence.
- Improvements to Drivers
  - Current user-space drivers are implemented using "/dev/mem", which could be a security risk if the system was integrated with another system, as it gives direct access to physical memory. A more proper alternative would be to use Userspace-IO (UIO) driver functions to have more security.

  - The current drivers do not do any checking of bounds. If a very long reference sequence is provided, the drivers will not check to see if it exceeds the size of the DDR. Checks should be done either in the program or the drivers themselves to prevent writes to locations not mapped to the device.

- Running more sequencers in parallel
  - The system is restricted to only three sequencers, despite there being more resources available on the FPGA. The limitation is in master AXI ports to the interconnect (the interconnect can only support 16, but each sequencer uses 4 to read the C-table entries). The sequencer could be redesigned to use fewer ports.
  - Alternatively, each sequencer could run multiple short reads (in addition to there being multiple sequencers), thus reusing the ports.

- Reporting what position a short read matched in the reference sequence
  - The final values of the top and bottom pointer can be used to determine how many matches there were, and what their original position was in the reference sequence. The information is available in the system, we would just need to link it and present it to the user.

3 Initial Architectural Design

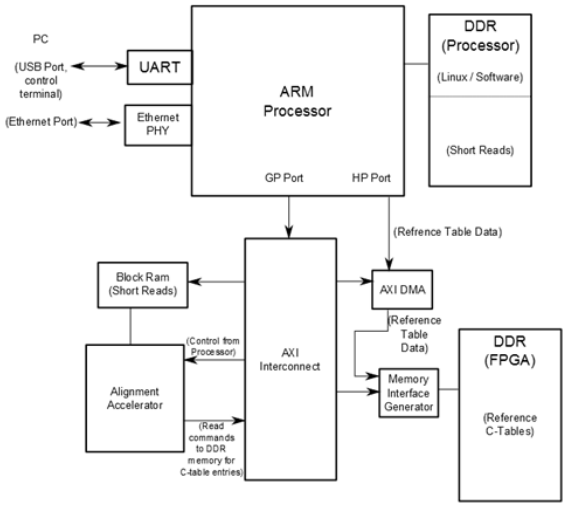

Figure 5: Initial Block Diagram

3.1 Process for Creating Original Architecture

When creating our initial design, we first decided on the minimal set if functions it need. The set of tasks our system needed to perform were:

- Send the data for the reference sequence and short reads from the user to the system.
- Compute the BWT of the reference sequence
- Allow the short reads and the results of the BWT (C-table and I-table) to be visible to the programmable logic
- Perform the matching (in the programmable logic, with as much parallel operation as possible) of the short reads withing the reference sequence
- Communicate to the user whether there was a match for each short read

With this list established, we selected components that could implement that could perform these functions.

3.2 Initial Architecture for Arm Processor and Programmable Logic Interface

Firstly, we had to make the decision of whether to run a Linux operating system on the ARM processor, or simply have a bare metal application. It was initially believed that setting up Linux on the hardened ARM processor would be too difficult to actually implement in the 7 week period we had available. However, with the support of Dr. Chow we were able to get in contact with a graduate student who gave us a guide on how to do this. Thus, our initial architecture for the ARM processor was decided to be implemented using Linux instead of bare metal. Using Linux would provide some additional challenges, as virtual memory could make it difficult to access the physical addresses of memory/accelerators in the programmable fabric. However, the Linux system would simplify communicating with the host PC, either by transferring files or with serial communication. The difficulty of implementing a usable communication protocol in a bare metal application would likely be higher (as accessing the programmable fabric should be a standard operation for these types of systems that Xilinx's tools should be able to accommodate), we settled on using Linux.

To load the data onto the system, one option was to use to UART attached to the ARM processor to transmit the data serially. However, with larger sequences, the bandwidth would likely be too low. Therefore, our initial proposal had the board connected to a host PC via the Ethernet port. This would allow text files to be be freely passed between the two computers. Various demo projects with the Mini-ITX board used the Ethernet port, so it was assumed that it would be straightforward to set up.

For computing the BWT, there were three main options. First, the transform could be performed on the host PC, with the results being transmitted to the board. Second, the ARM processor could perform the transform on the board itself. Third, the transformation could be done on hardware in the programmable logic. Option three had little benefit. Producing the hardware would take more designer and re-compiling time, even if using HLS, and the algorithm is serial, with little opportunities for parallelism. The transform is also only performed once, and so having dedicated hardware might be wasteful of the FPGA resources. The first option was also not chosen, since the results of the transform requires data than the original reference sequence, which means more time would be required for the initial data transfer. Therefore, we chose to perform the transform on the ARM processor.

The initial software implementation of BWT algorithm was designed to read specifically from a input sequence file named "input_file.txt", where the file would first start with an integer number specifying the length of the original reference sequence, and then followed by the sequence. It is

necessary to keep the length of the sequence exactly the same as the number, or else the program would crash or report an error. The initial software implementation was able to generate the I-table and C-table correctly and write each table in a .txt file. This was designed to validate the correctness of the implementation of the algorithm, but certainly not optimized for integration with the rest of project. The initial implementation of the software.

Making the data visible to the programmable logic required moving the data from the processor's memory into memory blocks (either the DDR chip or block RAMs) on the FPGA. Initially the transfer was planned to be done using a DMA. Since the tables could be quite large (proportional to the length of the reference sequence), a block transfer was believed to be the most efficient way to move all the data. The short reads were not planned to have a DMA, as they would likely be much shorter than the reference sequence.

The matching itself was to be performed on the FPGA, using a dedicated accelerator created in HLS. The matching had the most parallel computation possible, as multiple short reads can be run independently of one another.

The communication of results back to the user was decided to use the UART attached to a console on the host PC. The initial tutorials for developing on the Mini-ITX board documented how to set up this method, and allowed the programmers to use familiar C functions to display results.

A block diagram of the initial system containing all of these described components is shown in Figure 5.

4 Specification Evolution and Final Architectural Design

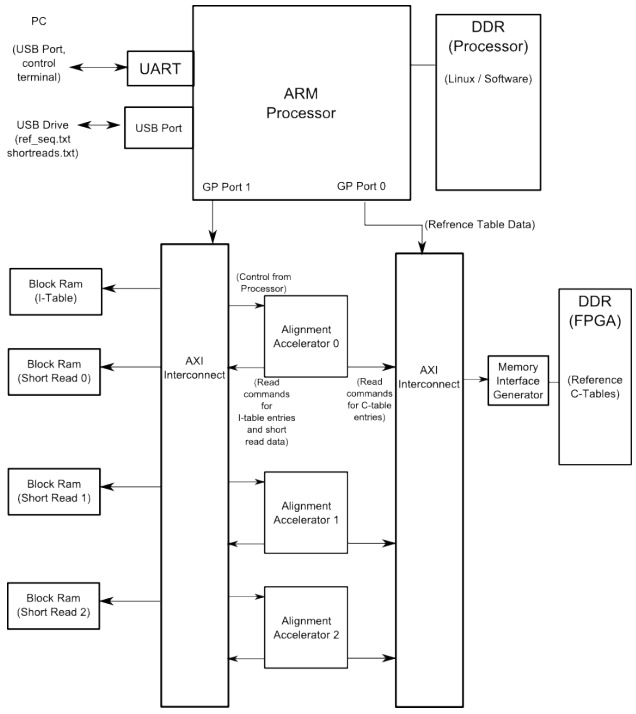

Figure 6: Final Block Diagram

4.1 Problems Encountered with Initial Design

Using Linux complicated the use of a DMA to transfer data to the programmable logic. Due to Linux's use of paged memory, a DMA transfer could be at most the size of a page, and the program setting up the transfer would have to be aware of page boundaries (as pages may not be physically contiguous in the processor's memory). Thus, a DMA would not gain the full benefit of using a single burst transfer to copy the data. One way around this may have been to prevent Linux from using a portion of the processor's DDR chip, using that to store the transformed tables in a contiguous block. However, this would limit the length of the reference sequences the system could work on. Looking forward, we wanted to use all of the programmable logic's DDR to store the reference tables to have the longer possible reference sequences. However, using this technique with the DMA has the system limited to the memory on the processor's DDR that isn't already used up by Linux. Therefore, we decided to not use the DMA and just perform regular memory mapped accesses to write to the FPGA's DDR. The performance loss should not be significant. When using the DMA, we would have had to create all the table entries before initiating the transfer. With simple accesses however, we can write a row of the table as soon as it is created, allowing the writes to happen in parallel with the next row being computed.

The original proposed design had the user transfer file to the board over Ethernet. This would likely have been doable, however there would be some difficulties. In addition to configuring the system on the board, the host PC would also need some configuration, added extra work for a user. In addition, the example design in the tutorials for using Ethernet on the Mini-ITX board had a web server in mind, rather than connecting two computers over a LAN-like connection. Therefore, we switched the design to use a USB drive to transfer files. This took less configuration, and achieved the same function.

4.2 Other Optimizations Added

The Final Implementation of BWT algorithm gave more flexibility to the user, where the software could take in one more argument specifying the file with the reference sequence when the user initiated the program. The program was capable of saving the C-table and I-tables in byte files instead of .txt file, which was more effective for data accessing and obtaining the integer values in the tables since the tables only contained 4-byte integer values easily interpreted by hardware. To further improve memory utilization and data access, the C-table was separated into four different tables for A, T, G and C base pairs. The software program was also programmed to generate the suffix table for locating the precise location where a matching occurs. Eventually, the software was integrated with the entire project, saving the I-table in block ram and the C-tables in DDR memory.

The original design only had one sequencer module, meaning only one short read could be run at a time. Since there was bandwidth remaining on the AXI bus to the DDR, we could run multiple sequencers in parallel. The AXI protocol could manage and schedule the multiple accesses.

In order to get more bandwidth to the DDR, a second AXI interconnect was added. The sequencer needs 3 reads for each short read character (one for the character, and two for the C-table entries). The two C-table entries have to access the same memory, but the bandwidth from reading the short read could be moved to another bus. If we scale our system further, this would allow us to use more of the DDR's bandwidth and run more sequencers in parallel.

4.3 Final Architecture for ARM Processor and Programmable Logic Interface

A block diagram showing the final system is given in Figure 6.

All the components are summarized below:

- ARM Processor:
  - Managed communication between the system and user
  - Software performed the BWT
  - Transferred BWT tables to DDR memory
  - Transferred short read data to block RAMs
  - Controlled the HLS accelerators by informing them were data was stored, triggering them to run, and checking the results.
  - (Hard block on the Mini-ITX board)
- UART Port:
  - Was connected to a serial terminal on the Host PC
  - Allowed Linux terminal commands entered by the user to be sent to the processor
  - Transferred standard console output back to the user
  - (Hard block on the Mini-ITX board, directly attached to the ARM Processor)
- USB Port:
  - Allowed a user to store their own reference sequences and short read files
  - Accessed by mounting the remote drive onto the Linux file-system, using Linux commands
  - (Hard block on the Mini-ITX board, directly attached to the ARM Processor)
- Processor DDR:
  - Stored the memory needed by the Linux OS and file-system
  - Provided memory to be used by the BWT software
  - (Hard block on the Mini-ITX board, directly attached to the ARM Processor)
- AXI Interconnect, Left:
  - Facilitated communication via memory mapped transfers between the ARM processor, the block RAMS, and the HLS accelerators.
  - The processor was a master that could write to the block RAMS and read/write control signals to the accelerators' slave registers.
  - The accelerators were both a slave device for control signals, and a master that could read from the block RAMs.
  - The block RAMs were slave devices
  - (Implemented in programmable logic, Xilinx IP)

- I-Table Block RAM
  - Used to store the 4 integers of the I table, needed by the FM index algorithm running on the accelerators
  - (Implemented in programmable logic, Xilinx IP)
- Short Read Block RAMs (3)
  - Used to store the character arrays of the short reads to be matched by the accelerator
  - (Implemented in programmable logic, Xilinx IP)
- Sequencer Accelerators (3)
  - Performs the matching of a single short read against a reference sequence.
  - Reads data short read data and I-table from block RAMs, and C-table entries from DDR
  - (Implemented in programmable logic, Custom HLS Module)
- AXI Interconnect, Right
  - Facilitated communication via memory mapped transfers between the ARM processor, the HLS accelerators, and the FPGA's DDR chip.
  - The processor was a master that could write to the DDR chip.
  - The accelerators had master ports that could read from the DDR.
  - The DDR was accessed through the memory interface generator, acting as a slave device
  - (Implemented in programmable logic, Xilinx IP)
- FPGA DDR:
  - Stored the C-tables produced by the BWT so it could be accessed by the accelerators
  - (Hard block on the Mini-ITX board, attached to the programmable fabric via the memory interface generator, a Xilinx IP in the FPGA fabric.)

The operation of the system would follow these steps:

- The Linux system would boot up on the processor from a micro-SD card.
- As part of the boot processes, the bitstream programs the FPGA.
- When prompted on their serial console on the connected host PC, the user would log into Linux
- The user would insert their USB drive containing their reference sequence and a set of short read. The user would mount the USB drive onto the Linux file-system.
- The user would run the program "RunSequencer", providing the reference sequence and short reads as arguments.
- The program would read the reference sequence and compute the Barrows-Wheeler transform. The resulting C-table is stored in the FPGA's DDR. The I-table is stored in it's designated block RAM.

- The program reads up to 3 short reads from the short reads file, copying the characters into the appropriate block RAMs.
- The program sets where the accelerators need to read C-table, I-table, and short read data from. It then triggers the module to begin running.
- The sequencer accelerator performs the matching using the FM index algorithm, reading data from the short read block RAMs and the C-table as needed.
- The program polls the status registers of the accelerators until all three are done.
- The program reads the result from the accelerator's slave registers and displays it to the terminal.
- The program continues reading more short reads until all from the file have been run.

5 Methodology

5.1 Design Environment

- Platform Used: The Mini-ITX board was used to implement the design. This board has a hardened arm processor, which is attached to 1Gb of DDR RAM, as well as a Zynq FPGA, which is attached to another 1Gb of DDR RAM.
- Tools Used: Vivado design studio, Vivado HLS, Petalinux Tools, Visual Studio 2013
- Source Code Control: There was no need to implement a source code control software as the design was partitioned into mutually exclusive segments that various members of the group were assigned to. In this way only a single member of the team was responsible for a particular file which allowed us to bypass code control.

5.2 Partitioning

In a nutshell, our DNA sequence project was split into two parts - the first part focused on implementing and running the BWT algorithm in the Petalinux environment on the ARM processor and generating the necessary tables, the second part focused on implementing and running the FM-index algorithm on the FPGA by accessing the tables and short read sequences and determine whether there is an exact match between the short read sequences and the reference sequence. To address the two challenges, our design was partitioned into four parts, one part for each individual:

- How to implement the BWT algorithm as a software program capable of reading from a file containing the reference sequence and generate the C-table and I-table.
- How should the ARM processor interact with different memory blocks (Block Ram and the DDR memory) in order to store the generated tables properly, allowing the FPGA to access the data efficiently.
- How to implement the FM-index algorithm in FPGA using HLS, and what are some of the strategies we can apply to the implementation to make it more efficient.

- How to implement our HSL design in the FPGA and enable it to communicate with different memory blocks containing the short read sequences, the I-table, the C-tables and the suffix table.

The division between the BWT component and the FM-Index components are because both of there steps could be implemented and verified in parallel. The BWT transform only required the reference sequence as an input. While the FM-index used the results of the BWT, from our initial research we already had a small number of example sequences (and their transformed tables) that we could use.

There was some overlap in the work, as interacting with the memory blocks and the HLS module use similar methods. However, this overlap was intentional. At the beginning of the project, the communication between the processor running Linux and the programmable fabric was highlighted as a potential bottle neck, as none of our group had earlier experience in it. Having two people learn it independently allowed the group to perform more research and experimentation. The two group members could explore different approaches and be more likely find a working solution.

5.3 Simulation, Verification and Testing

Our overall testing strategy was to divide up the system into components, testing them all individually. If our testing at each stage was thorough, there would be fewer problems in integration, as we would understand how each component behaves.

5.3.1 Verification of BWT

To verify that the Burrows-Wheelers Transform algorithm can generate the C-table and I-table correctly, it is essential to verify that the sequence is properly transformed, since both C-table and I-table are dependent on the transformed sequence.

The first test case was a sequence obtained from the work by E. Fernandez [1]. The sequence was made up of 14 characters, and the transformed sequence was compared against the result from the paper, and it is proved that the transformation was correct.

For longer sequences, an online BWT software from http://kodu.ut.ee/~lipmaa/research/bwt/, where the same reference sequence, with an extra "$" sign added to the end of the sequence as required by the BWT algorithm, was used as the input to both the online software and the BWT algorithm we developed. One sequence of 50 characters and another sequence of 200 characters are tested, and the transformed sequence for both reference sequence are verified to be correct. The results are shown in Table. 1.

5.3.2 Verification of HLS Modules

The HLS module is verified in Vivado HLS environment using a C testbench. The C testbench reads in the I-table and the C-tables in binary format and stores them in arrays. A short read is also provided to the HLS module in the form of a C string. The lengths of the short read and the reference sequence are also fed to the HLS module. The testbench then calls the HLS module and runs the sequencing and waits for the module to finish. It detect the end of sequencing when the done signal is driven high by the HLS module. The testbench verifies the validity of the HLS module function by checking the matched signal.

Table 1: Three test cases with transformed sequences used to verify the BWT algorithm.

Original reference sequence	Transformed sequence (BWT software)	Transformed sequence (online software)
GCTAATTAGGTACC$ (14)	CTTTACAG$AGCGTA	CTTTACAG$AGCGTA
ACAATTCTTATGCGATTA TGGCATATCGATCGATG CATTCGATCGATGCA$ (50)	ACC$CTGGGGTTGCAGA GGTTTTGTCCCCCTGTT TATTTAAATAAAACAAA	ACC$CTGGGGTTGCAGA GGTTTTGTCCCCCTGTT TATTTAAATAAAACAAA
ATCGATGCATGCATGCAT CTAGCACGATCAGTCGA TCGATCGATGCATCAGT CATGATAGCATCAGTTA GAATTATAAATGCGGCG CCTCCCTATATATATTATA TAGCAGCTAGCGCGCTC CCTCTCTTCCCCTTAGAT TAGCATGCATTATGGATC TGTAGTAGCATATGCTAC TTCAGTCAGTAGCTAGC TAAATGC$ (200)	CTTAAGCTTTTTTTTTT CTTCCCCCTTGTTTCTC GCGG$CGCAGCCGCATT GATCGGGTTTTTGGGGT GGGGTTTCCGCCATTTT GGGGGGTGGCCGCTTCA TATCCCGCCATAAATATT TATCGACTATAACTCAA TAAAACACTTCAGATCC GGTATCAAATAGTAAAG TCCGAAAACACAAAAAA AAAACGCAAAACC	CTTAAGCTTTTTTTTTT CTTCCCCCTTGTTTCTC GCGG$CGCAGCCGCATT GATCGGGTTTTTGGGGT GGGGTTTCCGCCATTTT GGGGGGTGGCCGCTTCA TATCCCGCCATAAATATT TATCGACTATAACTCAA TAAAACACTTCAGATCC GGTATCAAATAGTAAAG TCCGAAAACACAAAAAA AAAACGCAAAACC

To confirm how the HLS module expected the input arrays to be stored, a waveform of a simulation was dumped and examined. In particular, this waveform showed how the 8-bit characters in a short read were packed into 32-bit words transferred on the AXI bus and the ordering of multiple characters within a word.

5.3.3 Verification of Memory Drivers

The memory writes were verified by reading the address that had been written to immediately after each write. If the value that was read back was the same as the value that was written then the write has worked correctly. The reads were conducted by a function referred to as Peek while the writes were conducted by a function referred to as Pokes. The verification of the memory writes was as simple as calling the Peek function on the same address that the Poke function had written to.

5.3.4 Verification of Integrated System

With the previous steps verified, any errors occurring in the integrated system must me due to the communication between the components.

The physical registers in the HLS module could be probed from Linux using the built in Peek program. This was used to ensure that the program correctly set up the memory locations to read from.

To gain more insight about how the HLS module was running within the system, some additional

slave registers were added or re-purposed to show additional debugging information that could be read with Peek. Originally, the matched slave register returned either a 1 for matched or a 0 for not matched. The module was modified so that if an invalid short read character was read, its ASCII code would be written to the register instead. This allowed a debugger to discover why a run failed if it was not reading the short read characters correctly.

One possible problem that is not checked by our current testing is if a user enters an incorrect reference sequence or short read file. Neither our program or hardware checks this, so it will cause unknown results.

We also did not exhaustively check the effects of different sized reference sequences or short reads. Most of our test sequences were relatively short, as we would be able to check them manually. If a sequence is to long it will be written to an address outside the available device memory, with unknown effects.

6 Contributions

The members of our group spent an equal amount of time contributing to this project.

6.1 Arash Moghimi

- Set up the Petalinux operating system to run on the hardened ARM processor.
- Created Petalinux application for program that could generate the Burrows-Wheeler transform tables that could run on the ARM processor.
- Created an application which could transfer the tables that were created from the processor's memory to the programmable logic's memory.

6.2 David

- Created user space drivers to control the HLS accelerator and write to DDR/block RAM memories
- Created the block diagram of the system in Vivado
- Configured the Petalinux system to allow the mounting of USB drives.
- Re-wrote the application and drivers to accommodate multiple sequencers running in parallel

6.3 Meysam

- Developed the HDL module sequencer in Vivado HLS, in C
- Tested the basic functionality of the sequencer block, using some simple sanity tests
- Verified the sequencer using the I-table and C-table generated by the software

6.4 Leon Xiang Li

- Developed and improved the software implementation of Burrows-Wheeler Transformation algorithm, capable of generating the I-table, C-table and suffix table used for FM-index algorithm.
- Tested and debugged the BWT software on a Linux machine before migrating it to the ARM processor on Mini-ITX.

7 Design Characteristics

7.1 Resource Utilization

Table 2 shows the utilization of a single instantiation of the HLS sequencer accelerator. One accelerator takes 13 clock cycles to process each character in a short read.

Table 2: Utilization of a Single Instance of the HLS Sequencer Accelerator

Resource	Utilization
LUT	2,772
FF	3,920
DSP	0
BRAM	0
SRL	298

Table 3 shows the utilization of the entire system, including 3 insatantiations of the accelerator. At this time, the FPGA is not yet limited for resources, and more accelerators could be instantiated for more speed up.

Table 3: Utilization of a Entire System

Resource	Utilization	Pecentage (of the xc7z100ffg900-2 on the Mini-ITX board)
LUT	41,573	14.99%
FF	56,561	10.19%
BRAM	9	1.19%

7.2 Where the Time Went

Below are the task done in this project, and how many dedicated days one person had to spend on them:

- Developing the first functional version of the BWT transformation software (1 week)

- Optimizing the BWTsoftware for integration with the rest of the system, testing and verifying the software (1 week)
- Learning how to interface with HLS modules, using bare metal programming (2 days)
- Creating a test vivado project, petalinux project, and running it with a serial consol on a host PC (1 week)
- Creating test Userspace dirvers to access memory (4 days of research, 1 day to program and test)
- Controlling a test HLS module with user space drivers (2 days)
- Creating the HLS sequencer module (1 day)
- Implementing HLS module test bench in C (1 day)
- Testing HLS sequencer module in Vivado HLS environment with the C test bench and using the I and C tables generated by the test program (3 days)
- Creating the Vivado block diagram of the final system (1 day, mostly on running sunthesis/implementation)
- Creating the user space drivers to control the HLS module (2 days)
- Debugging the system when merging the BWT software with the driver code (3 days)
- Modifying the drivers to accomodate multiple sequencers running in parallel (2 days)
- Adding the ability to mount a USB drive in Petalinux (1 day)

7.2.1 Major Bottlenecks

- In designing the BWT algorithm is that at the software has to temporarily store a NxN byte size of rotated sequences in order to obtain the transformed sequences. This would put a limit the size of the sequence the program can process based on the hardware resource of the system it is running on.
- Learning how to integrate the HLS module with the the Linux OS took a lot of learning time. There are multiple ways to write a driver, and so some time was spent reading on several of them before deciding which was the simpliest way. There also did not seem to be much information on controlling the HLS modules in particular. Viviado HLS seems to produce some drivers ready for use in both bare-metal and Linux, but a day was spent trying to research how to access them in the new version of Petalinux without success.

8 Problems

- The Petalinux tools underwent a major change between 2014 and 2015. We were using the 2015 version, which overall seems to be much easier to use and get a Linux systems running. However, this did mean that it was difficult to search for solutions to any problems

we encountered. The answers in the Xilinx forums mostly refered to the older version of the tools, which did not always apply to us and could waste our time trying the given solutions.

- There were some difficulties interfacing the BWT software with the driver, partly due a minor miscommunication. One person had designed the hardware assuming that the I-table would be written into block RAM, so the I-table's master AXI port was connected to the block RAM interconnect (the "left" interconnect in Figure 6). However, the person adding the driver calls to the BWT software had assumed the I-table would be in the DDR, and so wrote the data there. The HLS accelerator could not access the DDR thorugh it's master port, and so returned incorrect results. Due to our earlier testing, we knew the values were being written to memory correctly, and we could narrow the cause down to the I-table. Our group was working together on the integration, so we were able to discover the mismatch.

9 Retrospective, Conclusions, Suggestions, Comments

9.1 Experience with HLS

In general Vivado HLS enabled us to develop and verify the sequencer block very fast within a couple of days. Without HLS we had to implement all the details of hardware in HDL and experience shows that the most time has to be spent not on developing the main required functionality but on debugging little details and corner cases. Using the directives, we were able to simply define the interfaces as AXI. There is no further action required whatsoever to verify the true functionality of AXI interfaces. The HLS sequencer block was effectively integrated in the top project and the AXI interfaces let us instantiate three samples of the sequencer and through using AXI interconnect modules share the DDR interface and block RAMs among them.

On the other hand, Vivado HLS sometimes provides little control on the code itself and how the code should be synthesized and implemented. Although, directives are designed to enable a user to control the hardware implementation details, sometimes they do not behave as expected. For example, Vivado HLS is not fully capable of unrolling the loops when the loop is needed to terminate because of a special case (for loop break case).

Another feature which current Vivado HLS lacks is the capability of embedding Verilog code in the C code. This might be essential because a user may need full control over a portion of a design such that he intends to provides either the RTL or even a gate level implementation of that portion and does not want to leave up to the compiler and synthesizer to implement it. This is similar to case where a programmer wants to embed a piece of assembly code in a C code.

Without HLS, developing the sequence block itself could take 5 to 10 times longer than what it took us with HLS. As mentioned above, all the small details of hardware should have been implemented in hardware, interface protocols should have been supported and implemented in hardware and more importantly a complete test bench should have been developed to thoroughly verify the functionality of hardware. The test bench should have included memory models, interfaces from memory models to the sequencer block and also a data collector and finally a result checker to make sure that the hardware produces correct results. It could have taken much longer if the hardware block was larger. In other words, the time saving factor of Vivado HLS is directly proportional to the complexity of the design; the more complex the design, the more time is saved by implementing it through Vivado HLS.

Overall, the Vivado HLS tool is very useful and impressive. It is a promising tool for the near future, knowing the fact that it is going to be improved and capable of handling more complicated cases.

9.2 Other

9.2.1 What We Would Do For Next Time

- Try to integrate into an initial complete system sooner. This would allow us to have more time to optimize the system, and learn earlier if we are missing any important components.
- Find a better way to implement the BWT algorithm without using too much memory space, either by using less bits to represent a character, or find a new way to perform the transformation.
- Improve the sequencer by adding in another feature - finding the exact location where a match occurs between the reference sequence and the short read. This feature can be realized by referring to the suffix table generated by our BWT software.

9.2.2 What We Learned

- A very good exposure to the concept of High Level Synthesis and a specific HLS tool (Vivado HLS) gained through finishing the two assignments and the final project.
- An important step in system design is partitioning. This project gave us a very good understanding about partitioning and how it is important in efficiency of the porject and collaboration among team members.
- As a result of using the embedded Linux, we learned how and operating system needs to interact with remote devices. (Eg. via the device tree, user-space or kernel level drivers.)

9.2.3 Thoughts on The Course

- Finding information on how to interface an HLS module with a proccessor system (both the hardware interface and drivers, even the bare metal ones) was surprisingly difficult for a relatively simple and common task. Perhaps that could be added to one of the assignments or made into an additional small assignment so that the information is easier to get?

Appendix: Documentation

The GitHub repository for this project can be found at: "https://github.com/davidmctavish/ECE1373_2016_GeneS

Instructions for running the existing system are found in the README file.

Also included in the repository are copies of 4 tutorials we read when creating the Petalinux system. For someone who needs to modify and recompile our system, or create a similar system from scratch, we recomend following them in this order:

- Zynq_Mini_ITX_Board_Files_Installation_Instructions_and_Tutorial.pdf
- Zynq_Mini_ITX_Embedded_Design.pdf

- Zynq_Mini-ITX_PetaLinux_2015_2.pdf
- ug1144-petalinux-tools-reference-guide.pdf

Appendix: Schematics and other Details

The block diagram included as Figure 6 is the best overview of our system. As an additional reference, the block diagram created by Vivado is also given here as Figure 7.

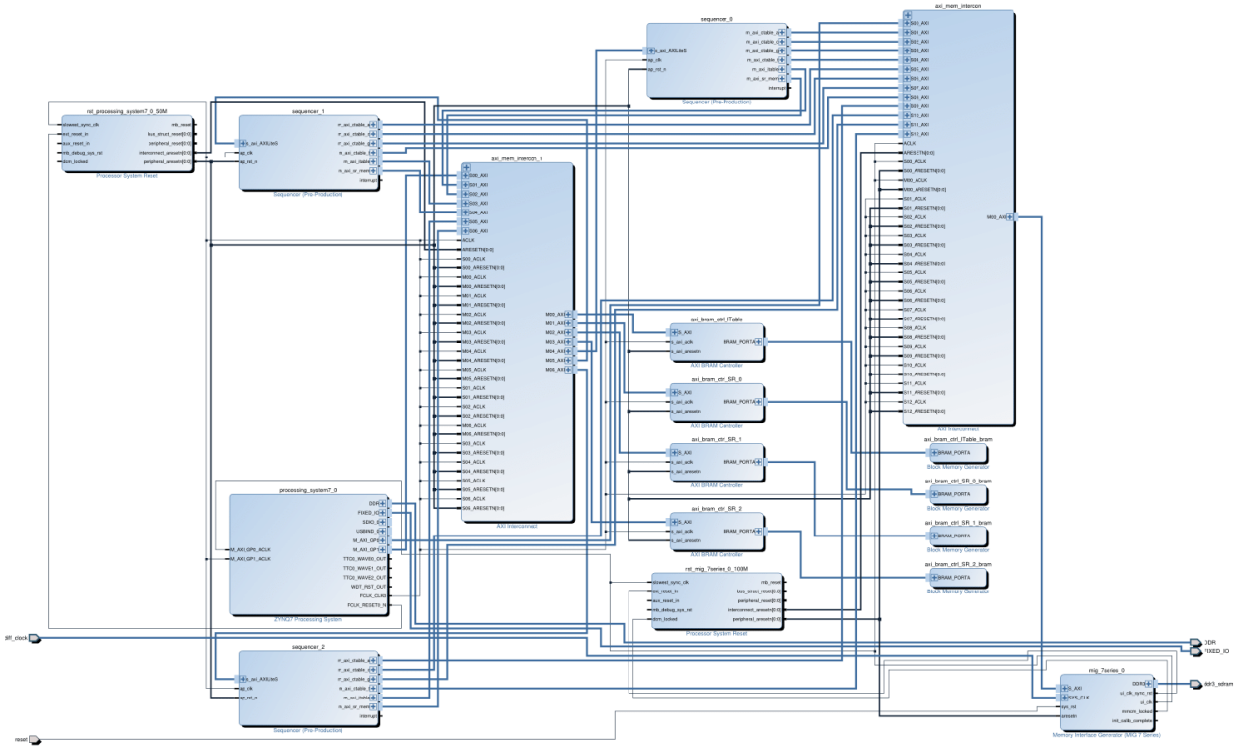

Figure 7: Vivado Block Diagram

Appendix: Hints for the Next Time

- Keep an updated copy of a block diagram for the overall system where all members can easily see it. This will help communicate where all the data in the system is kept.
- Remember that recompiling the hardware takes a long time, and that if you change the HLS module you also need to recompile the whole system in Vivado. If you can think ahead, add extra slave registers to store debugging information from your HLS module at the begining.

References

[1] Fernandez, Edward, Walid Najjar, and Stefano Lonardi. "String matching in hardware using the FM-index." Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on. IEEE, 2011.