Πολυτεχνείο Κρήτης

Less is more: Increasing the scope of the Xilinx Integrated Logic Analyzer with compression

Κωσταρέλος Φώτης

Abstract

Complexity and size of large-scale hardware designs is gradually increasing due to tool's-automation and the use of multiple IPs in the same design. Therefore efficiently debugging the entire system seems to be very hard, while hardware designers have to deal with finding and addressing the various vulnerable regions within these complex systems. For that purpose different modules and wires need to be monitored. Vivado's ILA (Integrated Logic Analyzer) IP is an advanced tool provided by vendor that is used to accomplish this mission. Every design under construction which needs testing can easily embed this IP pre-synthesis or post-synthesis. Despite the advantages that a user gets, the ILA has some defects e.g memory constraints. Hence an improvement to the memory management would be beneficial. How we are going to use the available resources is crucial. So, a lossless data compression module, placed between the ILA and a signal inside a design, is proposed. This technique implicitly will offer a gain in space, thing that it's going to be more clear later on. In order to select an algorithm for the compression module, we had 2 criteria to consider, first the Compression Ratio(CR) and second the needs of real-time debugging. From a search with these directions came that LZW is the desired algorithm. LZW stands for Lempel–Ziv–Welch, an improved implementation of the LZ78 algorithm. Thus, the main contribution of this thesis is an IFT(Integrated-Logic-Analyzer Facilitation Technique), a novel framework for hardware debugging that utilizes the LZW compression algorithm and exploits the debugging capabilities of the ILA cores. The tools used for the deployment of the proposed framework were: High Level Synthesis(HLS) Vivado and the Software Developement Kit(SDK).

keywords: *IP,ILA,LZW,IFT,HLS,Vivado,SDK*

Contents

Abstract **i**

List of Figures **iii**

List of Tables **iv**

1 Introduction **1**

1.1 Debugging without ILA 1
1.2 Debugging with ILA 1
1.3 Disadvantages of ILA(Motivation) 2
1.4 Compression with ILA(Contribution) 3
1.5 LZW Algorithm 4
1.6 IFT Architecture 5
1.7 Thesis Organization 6

2 Related Work **1**

2.1 Cusotm ILA 1
2.2 Compression in hardware 1
2.3 Deployment Tools 4
2.3.1 Vivado 4
2.3.2 High-Level Synthesis 4

3 Integrated-Logic-Analyzer Facilitation Technique **1**

3.1 IP (Intellectual Property) 2
3.2 Design IPs 2
3.2.1 Integrated Logic Analyzer 2
3.2.1.1 Probes comparators and triggering condition 2
3.2.2 Zynq programmable system 3
3.2.3 LZW IP 6
3.2.3.1 The algorithm 6
3.2.3.2 Compress 9
3.2.3.3 Decompress 14

4 System Implementation **19**
4.1 Customizing ILA core 19
4.1.1 Setting up the core 19
4.1.2 Core instantiation 21
4.2 Hardware Manager 21
4.2.1 Ila Dashboard 22
4.3 LZW Algorithm modification 24
4.3.1 HLS Implementation 25
4.4 IFT Software part 28
4.5 Evaluation 30

5 Conclusion-Future Work **32**

A Images **34**

B Code **35**

Bibliography **36**

List of Figures

1.1 Compression & ILA 3
1.2 CR = SizeOriginal/SizeCompressed 4
1.3 Compress-FIFO-FSM 6

3.1 System BD 1
3.2 Zynq Processing System 4
3.3 compress flow chart 10
3.4 compress example 12
3.5 decompress flow chart 15
3.6 decompress example 17

4.1 pre-synthesis dialog box 20
4.2 post-synthesis dialog box 20
4.3 ila dashboard 22
4.4 ap-ctrl signals 27
4.5 axi-stream signals 28

A.1 giphy image 34
A.2 rotating earth image 34

List of Tables

3.1 giphy_data less repetition 7
3.2 rotating_earth_data more repetition 8
3.3 infile even more repetition 8

4.1 ILA-LZW Comparison 31

Chapter 1

Introduction

1.1 Debugging without ILA

The size, in terms of internal connections or nets, of a modern embedded system can be immense, resulting from numerous IPs and interfaces from different vendors included in a single design. A system with such a size is difficult to be tested pre-implementation even by very experienced users. Conventional methods for debugging like simulation can be time-consuming and compute-intensive as well. In order to simulate a design with billions of signals in a virtual environment, requires considerably computational resources and time. In addition, there is no reason to simulate an entire system if just a component within it has been spotted erroneous. So, the usual methods for debugging lack of reliability and flexibility with possible post-implementation errors to arise, consequently are error-prone methods and inadequate.

1.2 Debugging with ILA

Integrated Logic Analyzer overcomes these issues by actual modeling the debugging procedure. The logic analyzer IP core includes many advanced features of modern logic analyzers, including Boolean trigger equations and edge transition triggers, using Block Rams to store captured signals synchronously and in real time with the design under test. The procedure of on-chip debug logic routes signals for observation and contains the following

steps.
• The user first defines the signals for probing
• Second the user has to choose the number of samples to be captured by the ILA and the triggering method.
While a design is downloaded on the board, the ILA core is armed and waiting for the trigger once you open your hardware manager, and connect your board with your PC through JTAG cable. Then samples acquisition takes place per discrete time intervals depended on the triggering method. If no trigger selected, samples are captured in each clock cycle. Finally, the captured values that are temporary stored in trace buffers, i.e Block Rams, transferred to the host PC for the user to view and further processing if needed. The ability to handle these values is also a very useful profit and utilized in the context of this project. This is done by exporting the captured values in a .csv file that will be explained more in later chapter.

1.3 Disadvantages of ILA(Motivation)

As a part of the design, ILA is limited by the available on-chip resources that must be distributed to the components. A memory heavy design has to share the BRAMs with the analyzer. Obviously, the user utilizes those resources at will, and if the design needs a lot of BRAMs then it has priority compared to the analyzer. So, the user then wont be able to debug the design efficiently because there is no more room left for ILA cores. Along with this restriction, the amount of debug data that can be stored into an on-chip trace buffer during embedded logic analysis is limited by the trace-buffer width, which constraints the number of signals to be probed, and its depth, which limits the number of samples to be captured. The second part of this restriction combined with the memory bounded by the ILA can be improved using compression. our goal was to increase the number of recorded samples without increasing the memory overhead. Memory overhead and management is always a challenge for designers in every aspect of computers as well as in debugging. Because of the limitation to the size of the trace buffers for each board, and the fact that is not a good practise to make use of all the resources that are available just for debugging, we exploit the compressed data.

1.4 Compression with ILA(Contribution)

The compression module which is placed before the ILA core has an input that is fed from the targeted signal for tracing, and an output that feeds the probe of the ILA core as shown in figure 1.1. The input of this module will be the "unknown" a priori values of the signal that is marked for debugging, while the output is the codes that are produced from the module during a design test. Then, the ILA in essence debugs these codes, but the total sum of codes captured by the tool, contains all the information required for decompression in order to restore the initial stream. Also, our intervention with this compression "filter" has to give an output for every single value enters the module, but more details in explanation of the architecture implementation.

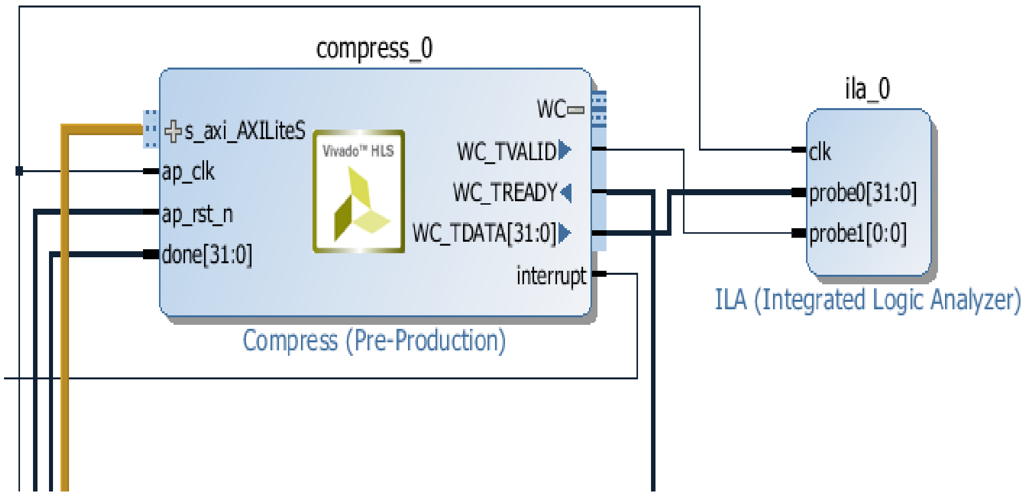


Figure 1.1: Compression & ILA

1.5 LZW Algorithm

Since there is a necessity for real-time debugging, an equivalent procedure was chosen to meet that need. Values from a signal come in a streaming manner and must processed individually. Considering we have no awareness of the input stream size and the values to be compressed, we must use an algorithm that processes values on the fly. This process just needs to have a signal that indicates the stop of the compression, or alternatively the end of the stream. A lossless compression algorithm that can easily be adapted for that purpose is LZW. It's a dictionary-based algorithm, and uses data statistical redundancy to encode information with the use of less resources. In addition, the algorithm tested so as to have a proper overview of it's Compression Ratio(CR example in figure 1.2). After many experiments with different inputs and dictionary sizes, it turned out that the algorithm could achieve a decrease to the input even 9 times to it's original size, given that the repetition of samples was high. Fact that lead to it's selection.

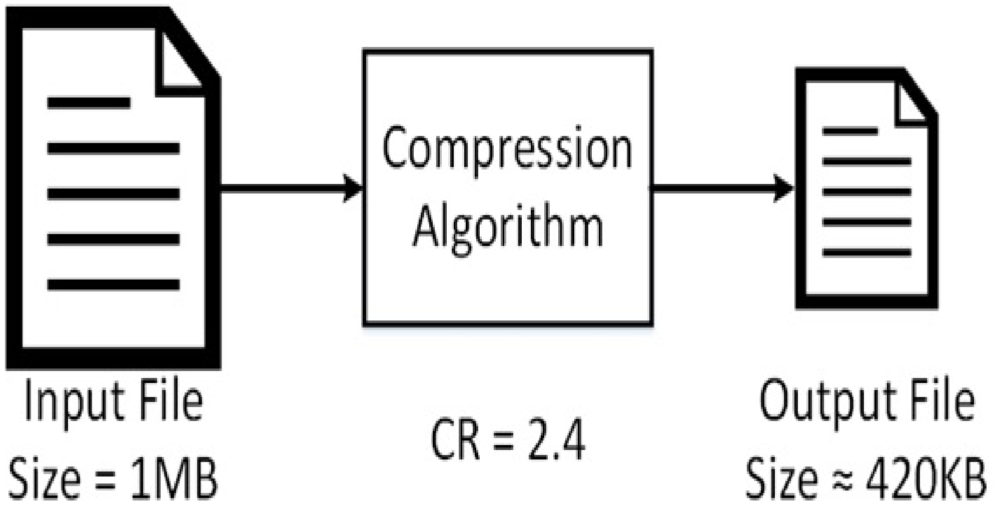


Figure 1.2: CR = SizeOriginal/SizeCompressed

.

1.6 IFT Architecture

LZW IP developed in HLS and integrated in a design along with the ARM processor in order to control it and represent the Design Under Test(DUT). Specifically it's a hardware accelerator that functions under ARM processor control. -In this point let's mention something. Hardware accelerators used widely to perform some procedures more efficiently than is possible in software running on a more general-purpose CPU. The reason that a hardware accelerator developed for this thesis is just for the sake of simplicity.- This is really an abstraction. ARM processor "plays the role" of two "hide" components a fifo and an fsm. As long as the compression module processes a value, a fifo needs to store data temporary, until the module is ready again for the incoming values. The compression needs some clock cycles to produce an output, and in this time interval values may come, so the fifo stores them in order not to lose them and process later. Apparently, the fsm controls the fifo and the compression module as shown in figure 3.1.

The steps for construction of this system are 1)After the lzw IP is exported from HLS, joined with ARM in a design and some peripherals that the Vivado tool adds automatically 2)The ILA is imported pre-synthesis in order to make some necessary connections. As shown in figure 1.1, ILA is an endpoint to this design and the compression module can't be synthesized with unconnected output ports 3)A simple application is written in Software Developement Kit which is running on ARM processor. These steps related with the hardware part of the project. The software one includes, the decompression in Python, and the presentation of the original data in GTKWave viewer. In order to display the data in the viewer, a C program converts the decompressed values from a txt file in a familiar file format for GTKWave which is the vcd(value change dump). All of these in detailed later on document.

Finally the system evaluation as it emerges from the tools reports, has a disadvantage in time. So, the trade-off of the Integrated-Logic-Analyzer Facilitation Technique is the overhead in execution time, but potentially with a gain in memory.

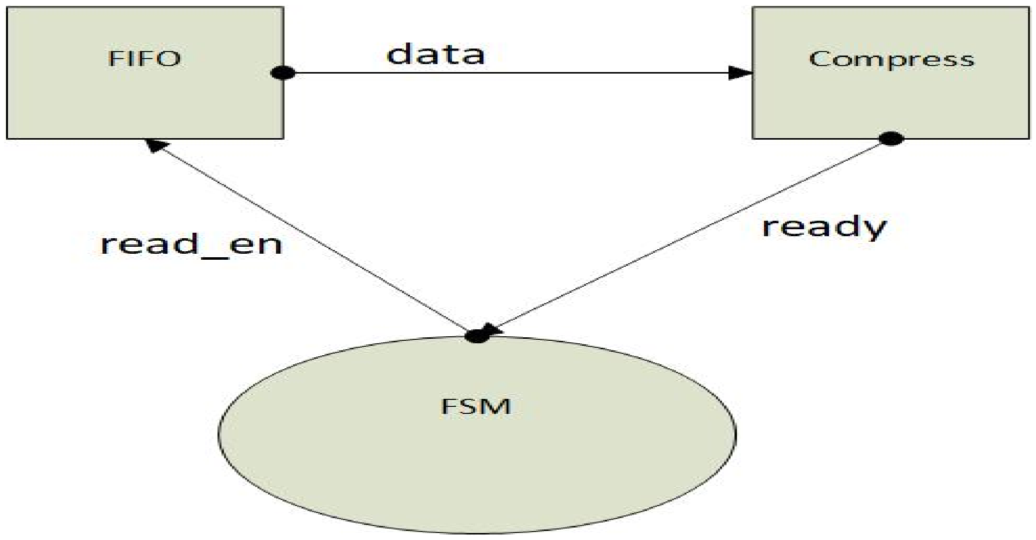


Figure 1.3: Compress-FIFO-FSM

.

1.7 Thesis Organization

The rest of this thesis organized as follows. In chapter 2 we present similar works that have been done towards compression algorithms and alternatives of them. Also, a reference to individual and custom systems for debugging, are covered in this chapter. Next in chapter 3, the system architecture is analyzed. An overview of the components in detail and explanation under the hood, covered there. We offer a detailed description of the LZW algorithm in which this thesis is based on. Also, we present the modification that applied to the algorithm in order to "clasp" with the ILA. Moreover, we are talking about vivado integrated logic analyzer and the zynq processing system briefly within this chapter.

Chapter 2

Related Work

2.1 Cusotm ILA

The first complete attempt for an Internal Logic Analyzer presented in [1]. An electronic design automation (EDA) software tool used to specify, signal for monitoring, number of samples, the system clock and the triggering condition. The EDA inserts the logic analyzer circuit into the design of the PLD(Programmable Logic Device). Through the communication interface between the host computer and the PLD, EDA arms the circuit and once the acquisition ends, directs the analyzer to unload the data from its capture buffer and then displays the data on the computer. They also provide the possibility to change the triggering condition without recompiling.

2.2 Compression in hardware

Lossless vs Lossy Compression. Compression can be either lossy or lossless. With lossless compression, every single bit of data that was originally in the input stream remains after the decompress. All of the information is completely restored. On the other hand, lossy compression reduces the input stream by permanently eliminating certain information, especially redundant information. When the compressed stream is decompressed, only a part of the original information is still there. Lossy compression is generally used for video and sound processes that a certain amount of information loss will not be detected by the user, in contrast with debugging where a data loss, obviously, is not acceptable.

In [2] a hardware compressor is used for debugging a multicore system. Their trace data come from Ericsson's ASIC platform. Trace data adopts the Nexus IEEE-ISTO5001-2012 standard. Nexus 5001 is a standard for global embedded processor debug interface, which is published by IEEE´s Industry Standard and Technology Organization. The Nexus standard defines trace and debug interface, including associated protocols and infrastructure that can serve tracing and controlling of multiple cores on a chip from the software debugger. The Nexus data employs a packet-based messaging tracing scheme with packet headers providing information about data source, destination, and type of payload. An optimized LZ77 compression algorithm is introduced based on the characteristics of these trace data. There are two modified aspects. The first aspect, the compressed data are output in a pair (1, distance) instead of a triple (1, distance, length). The match length information is omitted, because the maximum match length is equal to the minimum match length based on the characteristics of Nexus trace data. The second aspect, the entries of their hash table increased in order to reduce the hash collision probability and the amount of matching iterations.

Many works have implemented lossless or dictionary-based compression algorithms and their variants. For example in [3] a reference is made to Parallel Dictionary LZW(PDLZW) and Adaptive Huffman. PDLZW uses hierarchical parallel dictionary set with successively increasing word widths. The algorithm instead of initializing the dictionary with single character or different combinations of characters, a virtual dictionary with the initial $|\Sigma|$ address space is reserved, where Σ represents the set of input symbol and $|\Sigma|$ is the number of elements of the set Σ. The simplest dictionary update policy called first-in first-out (FIFO) is used to simplify the hardware implementation. The second algorithm, Adaptive Huffman, does not need to know the probability of the input symbols in advance. A frequency table is built dynamically according to the data statistics up to the point of encoding and decoding. Rather than using tree-based codewords, an ordered list of the tree structure is used to maintain the frequency table required in the AH algorithm.

Another work that utilized such algorithms is [4]. An LZW variation called Word-based dynamic Lempel-Ziv-Welch(WDLZW), operates similar with PDLZW. The second algorithm that appears in this paper is the modified BSTW(MBSTW). Their proposal supports multiple symbols encoding with a reduction in the encoder area. In these two methods the dictionary is implemented in hardware using CAM(Content Addressable Memory) and the CAM's depth represents the total number of entries in the dictionary.

However, towards efficient hardware debugging both types of data compression have been used. In [5] Anis and Nicolici use a lossy data compression algorithm for efficient silicon debug. They perform MISR on the selected debug data signals, thus identifying the interval with the faulty behaviour and then record the actual values of the signal. Accordingly, an approach to silicon debug for increasing the capacity of on-chip trace buffer via compression shown in [6]. They employ source transformation functions in order to reduce the entropy of the trace data. Source transformation refers to the idea of transforming un-encoded data set T into a new data set, T', which is more amenable for compression.

Finally, a novel area of introducing data compression to hardware debugging is in HLS designs. More specifically Goeders and Wilton in [7] present a novel debugger, which can be used with HLS tools. This debugger provides the users with the ability to debug their design using the original source code, without detailed knowledge of the underlying hardware, while the circuit executes on the FPGA.

A note, resulting from selecting a suitable compression algorithm, is that is a trade-off process. While one algorithm can achieve good compression ratio, the other can be with high throughput. In our case, the original LZW meets all our needs that will be explained in situ.

2.3 Deployment Tools

2.3.1 Vivado

Vivado is a design environment for FPGA products from Xilinx. Vivado design suite is a software tool for synthesis and analysis of HDL designs which replaced ISE. Notable additional features of the tool compared to it's precursor are high-level synthesis and the ILA, the updated version of Chipscope. Also, just to mention, if we look under the hood every action in Vivado tool is a tcl command in essence, and is shown in Tcl Console. But we will not go any further with these commands because is not in the context of the project.

2.3.2 High-Level Synthesis

Vivado HLS converts a C, C++ or System C source code into a Register Transfer Level implementation with both VHDL and Verilog codes apart. Working on a higher level of abstraction designers get benefits like the directives. Directives is a guide for the behaviour of the exported accelerator. Through directives we define the interfaces of the core and we can optimize it. They can be applied to the whole source code or to a specific part of it, in order to improve it. We need three types of directives, one for the control of the LZW IP by the Zynq processor, second for the interface between the core and the other modules, and third optimization directives within the code.

Chapter 3

Integrated-Logic-Analyzer Facilitation Technique

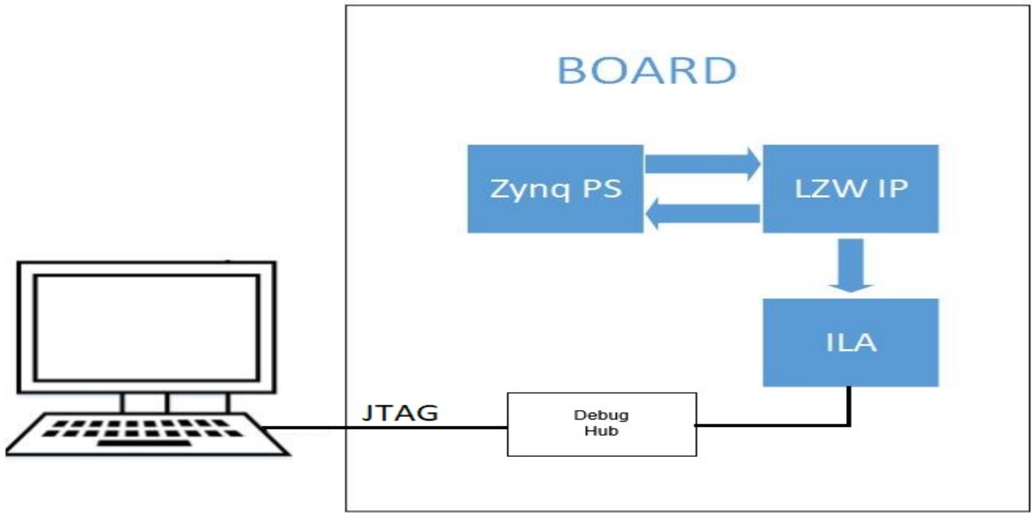


Figure 3.1: System BD

. IFT system consists of 3 components, LZW IP, ILA IP, and the Zynq programmable system. We will first examine the vendor's analyzer, next the Zynq programmable system, and then the LZW IP. We will look at the structure of the design which is these major components and then the implementation of this design which is their "cooperation". As two components provided by vendor we will look at them more briefly compared to the LZW IP which is the main object of the project and we will check it thoroughly.

3.1 IP (Intellectual Property)

An IP (intellectual property) core is a block of logic or data that is used in making a field programmable gate array (FPGA) or application-specific integrated circuit (ASIC) for a product. As essential elements of design reuse , IP cores are part of the growing electronic design automation (EDA) industry trend towards repeated use of previously designed components. Ideally, an IP core should be entirely portable - that is, able to easily be inserted into any vendor technology or design methodology. Universal Asynchronous Receiver/Transmitter (UARTs), central processing units (CPUs), Ethernet controllers, and PCI interfaces are all examples of IP cores.

3.2 Design IPs

3.2.1 Integrated Logic Analyzer

ILA is another IP fabricated by vendor and can be found in the IP Catalog of the Vivado. Because of its origin, ILA is compatible with every fpga chip of provider. Also, designers don't have to worry about ILA's timing. Because the ILA core is synchronous to the design being monitored, all design clock constraints that are applied to any design are also applied to the components inside the ILA core. If more than one clock signals exists in the same design, it is imperative to use corresponding cores. The one and only clock input of the core is attached to the probe ports. So, signals with different clock origins wont be able to be monitored by one ILA core.

3.2.1.1 Probes comparators and triggering condition

The probe ports of the ILA are connected to comparators that are capable of performing various operations. Trigger comparators can perform simple operations like equality or not to a given value, or more complex comparisons that are, greater & greater or equal and less & less or equal. The triggering condition is the actual result of a Boolean "AND" or "OR" calculation of each of the ILA probe trigger comparator result. The "AND" setting causes a trigger event when all of the ILA probe comparisons are satisfied. The "OR" setting causes a trigger event when any of the ILA probe comparisons are satisfied. The trigger condition is the trigger event used for the ILA trace measurement. Triggering condition is a very important point for the thesis. Later we will see how we configured the triggering condition of the core properly, in order to capture our "codes".

3.2.2 Zynq programmable system

Nowadays FPGAs and a cpu system are available for designers integrated in a single chip. Depending on the task for processing there is a trade off between them. FPGA's are more efficient when a large amount of data need to be processed under the same procedure. On the other hand cpus perform better when complicated flow control operations are demanded, i.e many if-else statements & for loops. So, these functional units combined together give us modern SoCs(System on Chip). The Zynq Processing System(PS) integrates the software programmability of an ARM-based processor with the hardware programmability of an FPGA. Zynq PS consists of the programmable system and the programmable logic. Figure 3.4 is the block design of the Zynq PS as seen in Vivado.

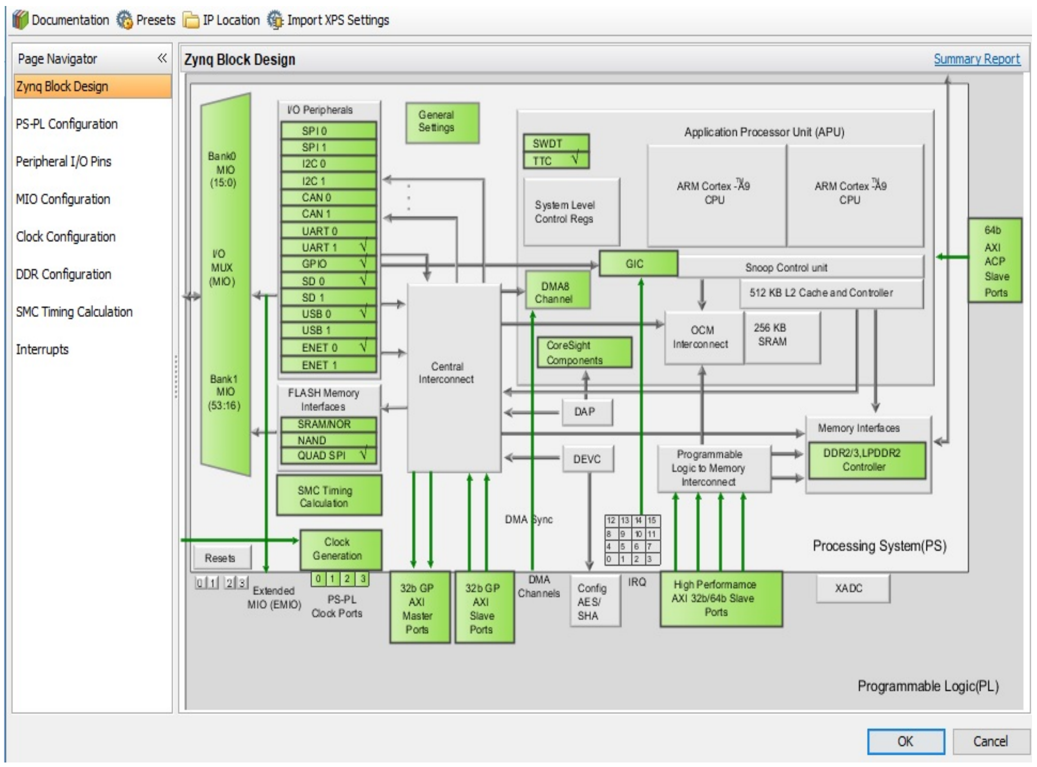


Figure 3.2: Zynq Processing System

.

The Zynq-7000 AP SoC is composed of the following major functional blocks:

- Processing System (PS)
  - Application processor unit (APU)
  - Memory interfaces
  - I/O peripherals (IOP)
  - Interconnect
- Programmable Logic (PL)

The Programmable Logic(PL) part is used to extend the functionality to meet specific application requirements. The PL includes many different types of resources including configurable logic blocks (CLBs), port and width configurable block RAM (BRAM) and digital signal processing (DSP) slices. This is the area that "hosts" the compression module.

We are interesting about the transactions between the PL and the PS. Through the interconnect there is a set of AXI interfaces for communication and transferring data from the PS to PL and vice versa. AXI stands for Advanced eXtensible Interface and is a protocol. This protocol simply sets up the rules for how different modules on a chip communicate with each other, requiring a handshake-like procedure before all transmissions. It's a standard through which modules can talk to the outside world. When a whole product family conforms to these kind of rules then maintenance and portability of designs becomes very easy. The latest protocol that is also utilized by ZedBoard is AXI4.

There are three types of AXI4 interfaces:

- AXI4 for high-performance memory-mapped requirements.
- AXI4-Lite for simple, low-throughput memory-mapped communication (for example, to and from control and status registers).
- AXI4-Stream for high-speed streaming data.

The concept of communication between modules is formed by AXI Master & AXI Slave. The AXI Master is the module that initiates a transaction and the AXI Slave is the module which responds to the initiated transaction. AXI4-Lite protocol applied to the input of our compression module in order to communicate with the Zynq IP which is the Master and the compression module is the Slave. We implemented this protocol because we need single data transaction each time. As mentioned above, there are several AXI-based ports and each one serves a different purpose. General Purpose(GP) I/O port is one of these ports, and the interface between the Zynq and the compression module as well. Through this port the Zynq processor can control the module and transfer data to it. Furthermore, the compression module needs somehow to inform the Zynq system whenever it finishes it's process. Obviously this is an interrupt signal that halts the processor. For that purpose one more port of the Zynq IP must be enabled, called IRQ_F2P. Finally the Zynq PS is the clock generator of the design and we are able to increase or decrease it's frequency. The reset is driven from an auto-instantiated special subsystem.

3.2.3 LZW IP

A compression module like this needs to process values individually as already mentioned. Since it's functionality depends on ILA we need to adapt the algorithm to operate accordingly. That is, ILA will record values each clock cycle according to a trigger condition, so we have to adjust that condition. The original compression algorithm that we will see, as long as it has successful search to it's dictionary it doesn't change the output stream. So, the output stream may not change for several clock cycles. Therefore, we must use an extra information which indicates that a match occurred. Apparently, when a match occurs the ILA will not record this value because it doesn't contain information about the initial stream.

3.2.3.1 The algorithm

Terry Welch first introduced the algorithm in 1984 as an ameliorated verof LZ78 which created by Abraham Lempel and Jacob Ziv at 1978. Although LZW is often explained in the context of compressing text files, it can be used on any type of file. However, it generally performs best on files with repeated substrings, such as text files. It is the algorithm of the widely used Unix file compression utility compress and is used in the GIF image format. Also, it was used to pdf files but in recent applications LZW has been replaced by the more efficient Flate algorithm. LZW exploits the repetition of data by replacing repeated elements with indexes to a dictionary. Software implementation as it is oriented to compress files, it has to recognize characters. ASCII characters are 256 in total. So, the original concept encodes sequences of 8-bit data as fixed-length 12-bit codes. The codes from 0 to 255 represent single-character sequences consisting of the corresponding 8-bit character, and the codes 256 through 4095, are created in a dictionary for sequences encountered in the data as it is encoded. In fact, the size of the dictionary is variable when in software. Dictionary is built during compression is in progress, memory is allocated dynamically so it's size is unpredictable. Also, the algorithm is designed so that the string table can be reconstructed by the decoder based on information in the encoded stream without transmit the table. Finally the algorithm has an upper limit to compress ratio. No matter how much space we gonna give to the dictionary every time, it's gonna reach that limit once and then stabilizes. For a given input stream there is a refined encoded stream that can no longer be improved regardless the size of the dictionary. Of course that stream results with experiments to the size of the dictionary.

Table 3.1: giphy_data less repetition

uncompressed file	dictionary size	compressed file
7 kb	1024	5 kb
7 kb	2048	5 kb
7 kb	4096	5 kb

Table 3.2: rotating_earth_data more repetition

uncompressed file	dictionary size	compressed file
56 kb	1024	49 kb
56 kb	2048	37 kb
56 kb	4096	33 kb
56 kb	8192	30 kb
56 kb	16384	29 kb

Table 3.3: infile even more repetition

uncompressed file	dictionary size	compressed file
102 kb	1024	18 kb
102 kb	2048	18 kb
102 kb	4096	18 kb
102 kb	8192	18 kb
102 kb	16384	18 kb

Tables above list the results from three experiments. For the giphy and the rotating_earth images we used the DiskEditor program. This program views the raw data of any file. Data of those images include multiple symbols in contrast with the infile.txt in which we used only two symbols in order to have high repetition. As we can see the first and the third experiment achieve the optimal Compression Ratio at once CR1=1.4 & CR3=5.6. For the second, we saw a normalization after the 4th attempt, around at CR2=1.6. These are "average" cases, marginal cases were omitted.

3.2.3.2 Compress

At the beginning, the dictionary contains all possible values of a byte, i.e 256 ASCII single-characters. Algorithm reads a character from the input, once per repetition, and creates series of characters. A series of characters is added to dictionary when there is no entry equivalent to this. Every sequence in the dictionary has a corresponding index. Once a character produces a sequence that matches in dictionary, the algorithm reads the next character. On the opposite case, the algorithm outputs the index for that sequence without the new character, so this index supersedes this series of characters to the encoded stream. Finally a new entry added to the dictionary with this character. Following are the steps and a flow chart.

Step 1. Initialize dictionary to contain one entry for each byte. Initialize the encoded string with the first character of the input stream.

Step 2. Read the next character from the input stream.

Step 3. If the byte is an EOF goto step 6.

Step 4. If concatenating the character to the encoded string produces a string that is in the dictionary: concatenate the the byte to the encoded string. go to step 2.

Step 5. If concatenating the character to the encoded string produces a string that is not in the dictionary: add the new sting to the dictionary. write the code for the encoded string to the output stream. set the encoded string equal to the new character. go to step 2.

Step 6. Write out code for encoded string and exit.

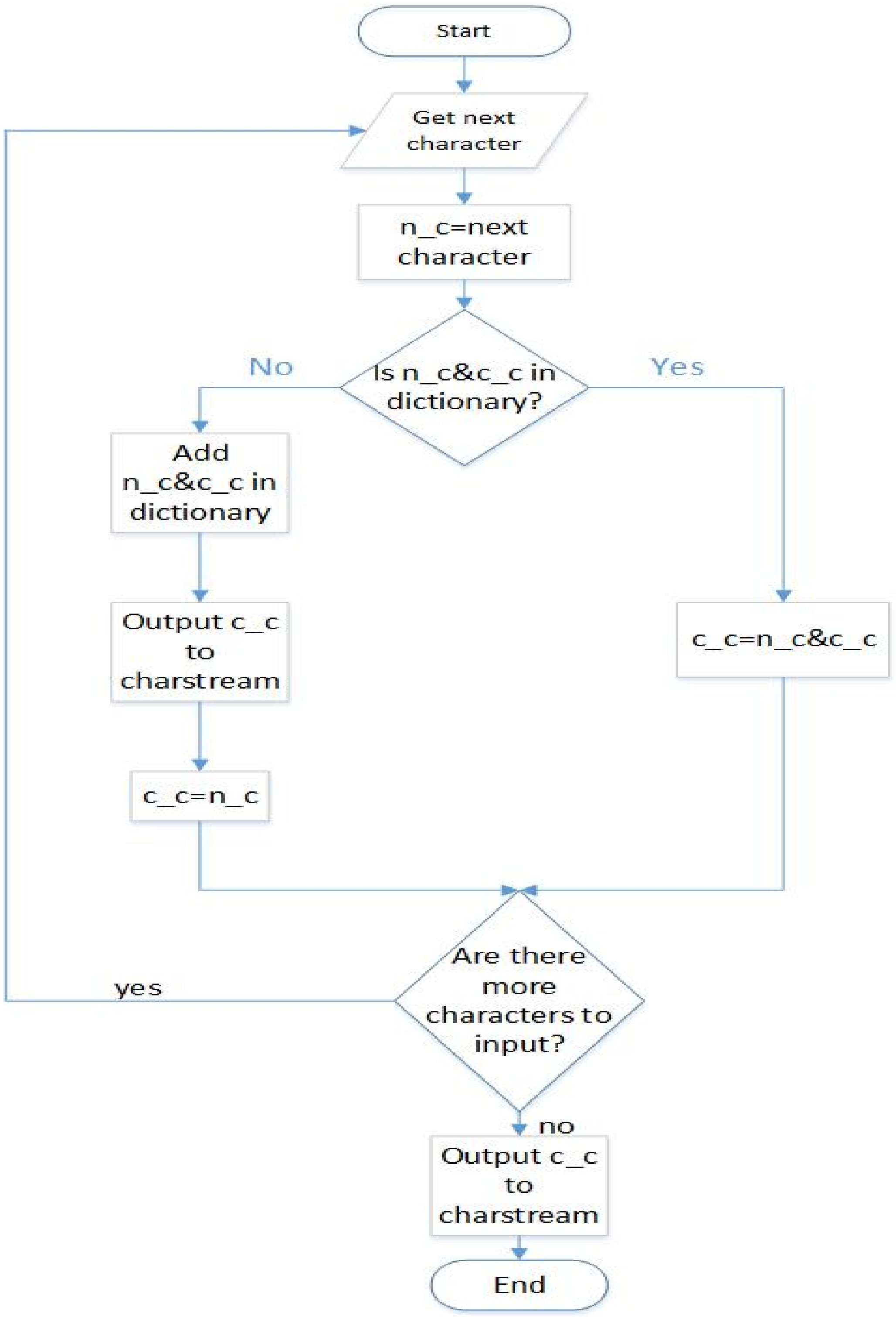


Figure 3.3: compress flow chart

.

c_c = current_character

n_c = next_character

cs = charstream /encoded stream

& = concatenation

To the next page we are about to see a demonstration of the compression process. The input stream is a special case that chosen because of the exception which arises in decompress. It's mandatory to present this exceptional case, because it's extremely frequent to steams with high repetition. So, the experimented stream is abcabcabcabcabcabc, i.e 6 times abc. The dictionary is initialized with the three possible characters and their indexes, 1=a 2=b 3=c. The left column of the dictionary is the index and the right is the data. When i=0 the algorithm reads the first character of the stream and fills the n_c variable. Then concatenates the empty variable c_c with n_c and the result which is "a" obviously exists is dictionary(✓=match). To the next iteration c_c is equal to the previous value of n_c and now n_c=b. This is an entry that does not exist in dictionary yet. So, the first character of "ab" is an output to the charstream and a new record to dictionary is created with index 4. After the "|" is the output to charstream at each stage. And so on until the EOF, where the current character is written to the output.

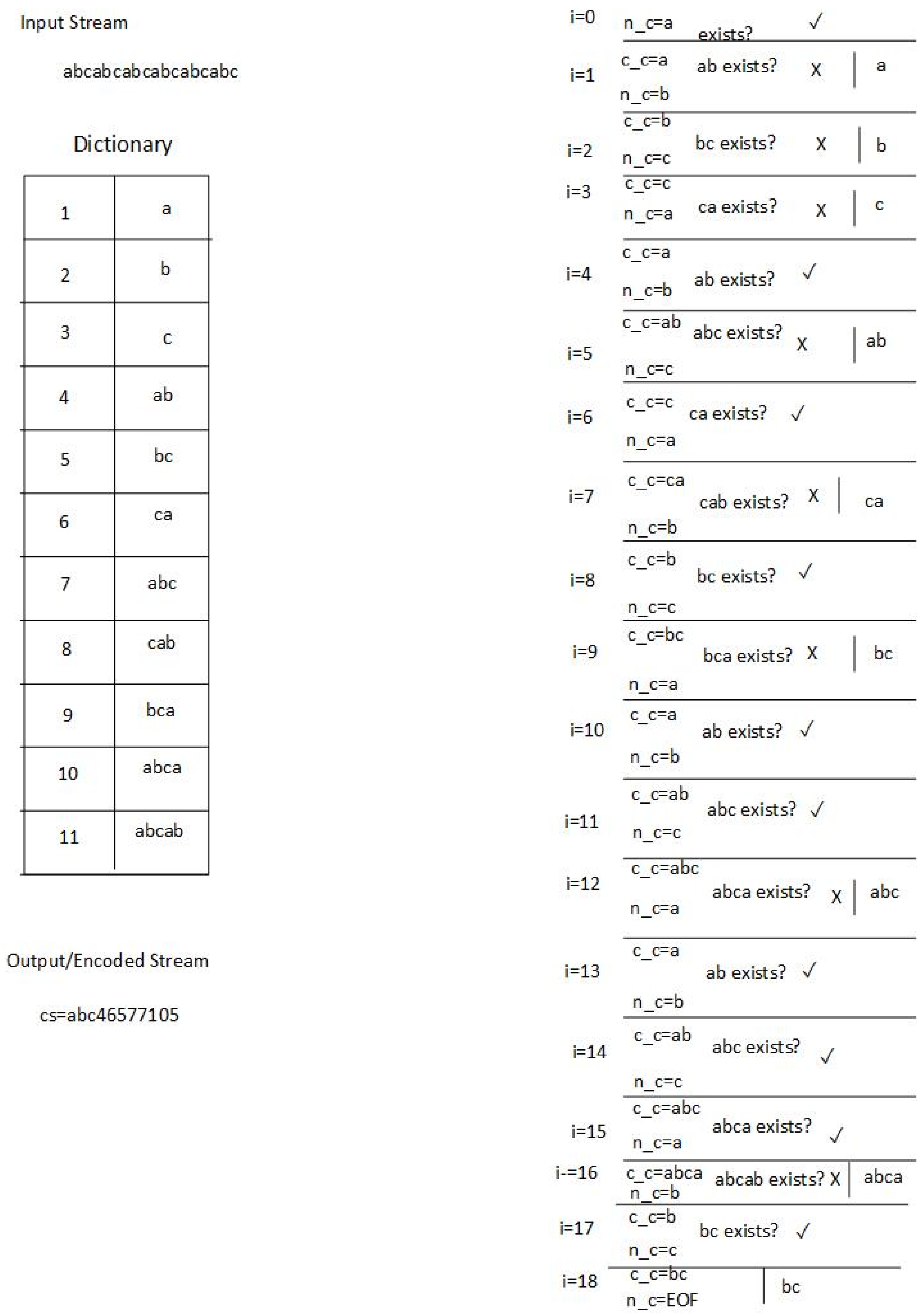

Input Stream
abcabcabcabcabcabc
Dictionary
1 a
2 b
3 c
4 ab
5 bc
6 ca
7 abc
8 cab
9 bca
10 abca
11 abcab
Output/Encoded Stream
cs=abc46577105
i=0 n_c=a exists? ✓
i=1 c_c=a n_c=b ab exists? X a
i=2 c_c=b n_c=c bc exists? X b
i=3 c_c=c n_c=a ca exists? X c
i=4 c_c=a n_c=b ab exists? ✓
i=5 c_c=ab n_c=c abc exists? X ab
i=6 c_c=c n_c=a ca exists? ✓
i=7 c_c=ca n_c=b cab exists? X ca
i=8 c_c=b n_c=c bc exists? ✓
i=9 c_c=bc n_c=a bca exists? X bc
i=10 c_c=a n_c=b ab exists? ✓
i=11 c_c=ab n_c=c abc exists? ✓
i=12 c_c=abc n_c=a abca exists? X abc
i=13 c_c=a n_c=b ab exists? ✓
i=14 c_c=ab n_c=c abc exists? ✓
i=15 c_c=abc n_c=a abca exists? ✓
i-=16 c_c=abca n_c=b abcab exists? X abca
i=17 c_c=b n_c=c bc exists? ✓
i=18 c_c=bc n_c=EOF bc

Figure 3.4: compress example

.

Two notable mentions are, one about the encoded stream, and the second about character sequence representation on computers.

Every encoded stream is a reproduction of the input stream until a repetition of characters occurs. For example, look at the output stream of the above demo. It starts with abc and then codes appear. That is, the length of the output stream is less or equal to the initial stream. So, compression achieved along with the bits for codes encoding. Usually codes need less bits than the data of the input stream.

A sequence of characters apparently is an abstract in terms of bits. Let's see under the hood to realize how they represented. The dictionary implementation in C, consists of three variables, index, prefix, character. Index it's obvious the code for each "sequence". Character, is the current read character from the input stream. Prefix it's little more complicated, so let's consider an example. Suppose the input stream is "aaabaaab". Dictionary at the beginning has only 1=a and 2=b. Then 3=aa 4=aab 5=ba 6=aaa 7=ab. So, "aaa" is stored with a prefix=3 and character=a in dictionary. This is how a search in dictionary is done. In the source code when a match occurs the prefix variable holds the current value of index, and in the opposite case takes the current character as usually.

3.2.3.3 Decompress

As already mentioned, the only information that decompress needs is the encoded stream. The dictionary is initialized similar to compression one with all possible values of a byte, i.e 256 ASCII single-characters. No dictionary from the previous step is necessary for decompress. A dictionary identical to the one created at compression is reconstructed during the process which follows these steps.

The first character of the encoded stream is outputted immediately. Then the algorithm reads the character or the codeword from the stream and outputs the corresponding string from the dictionary. In order to rebuild the dictionary in the same way as it was built during encoding, it also obtains the next value from the input and adds to the dictionary the concatenation of the current string and the first character of the string obtained by decoding the next input value. Moreover there is an exception to the process. Codewords that have not been added yet in decompressed dictionary cause that exception. Decompression process is always a step behind the compression so it maybe face an unknown codeword. The experiment we have seen in compress will cause this exception in decompress that we will see after the steps and the flow chart and how it is handled.

Step 1. Initialize dictionary to contain one entry for each byte.

Step 2. Read the first code word from the input stream and write out the string it encodes.

Step 3. Read the next code word from the input stream.

Step 4. If code word is present in dictionary

a)write the corresponding string to the output

b)Concatenate the first character in the new code word to the string produced by the previous code word and add the resulting string to the dictionary. Go to step 3.

Step 5(Exception). If code word is not present in dictionary.

a) Concatenate the first character in the previous word to the string produced by the previous code word and add the resulting string to the dictionary and write it to the output. Go to step 3.

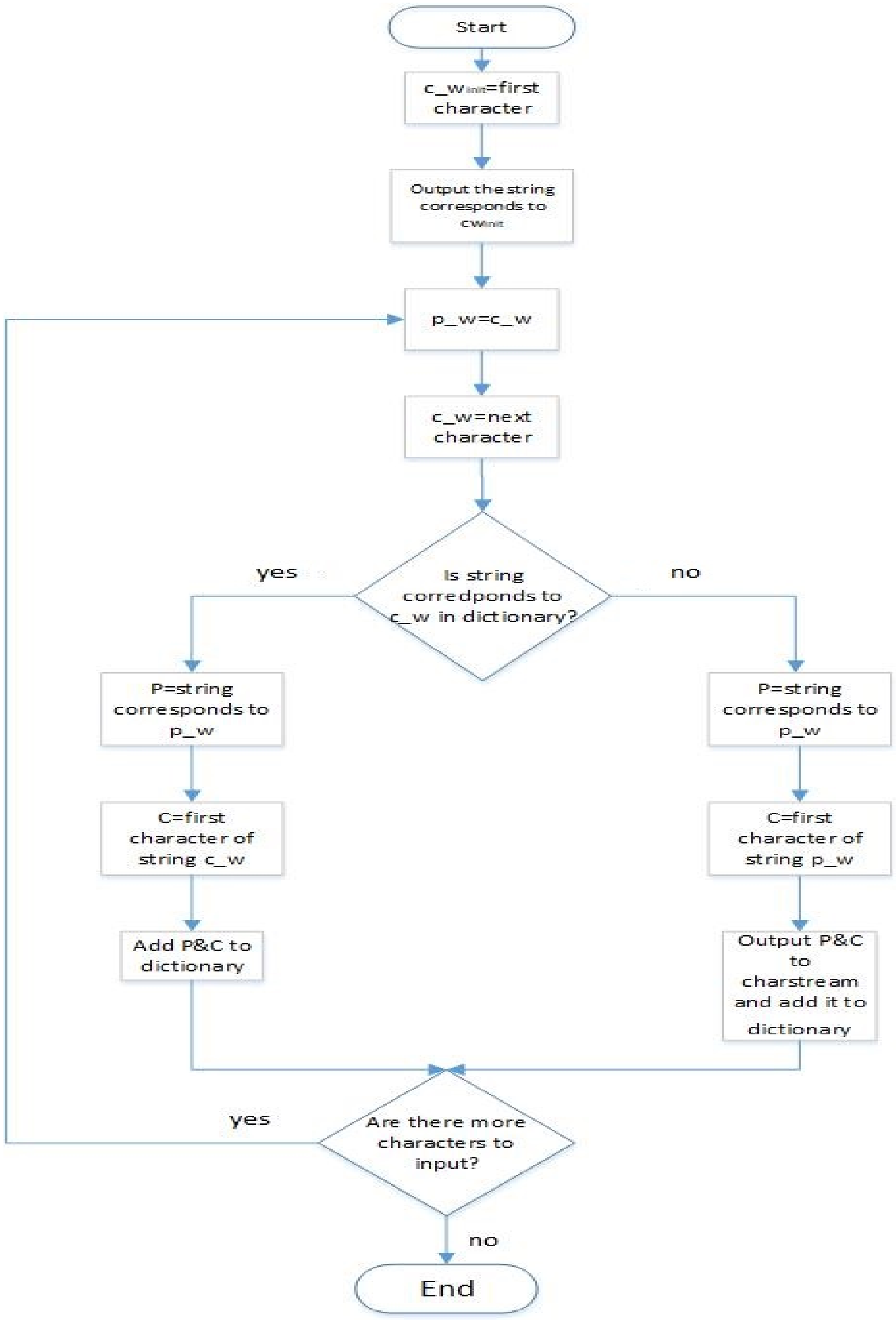

Start
c_winit=first character
Output the string corresponds to cwinit
p_w=c_w
c_w=next character
Is string corredponds to c_w in dictionary?
yes
no
P=string corresponds to p_w
C=first character of string c_w
Add P&C to dictionary
P=string corresponds to p_w
C=first character of string p_w
Output P&C to charstream and add it to dictionary
Are there more characters to input?
yes
no
End

Figure 3.5: decompress flow chart

.

cw_init = the first character of the encoded stream

p_w = previous_word

c_w = current_word

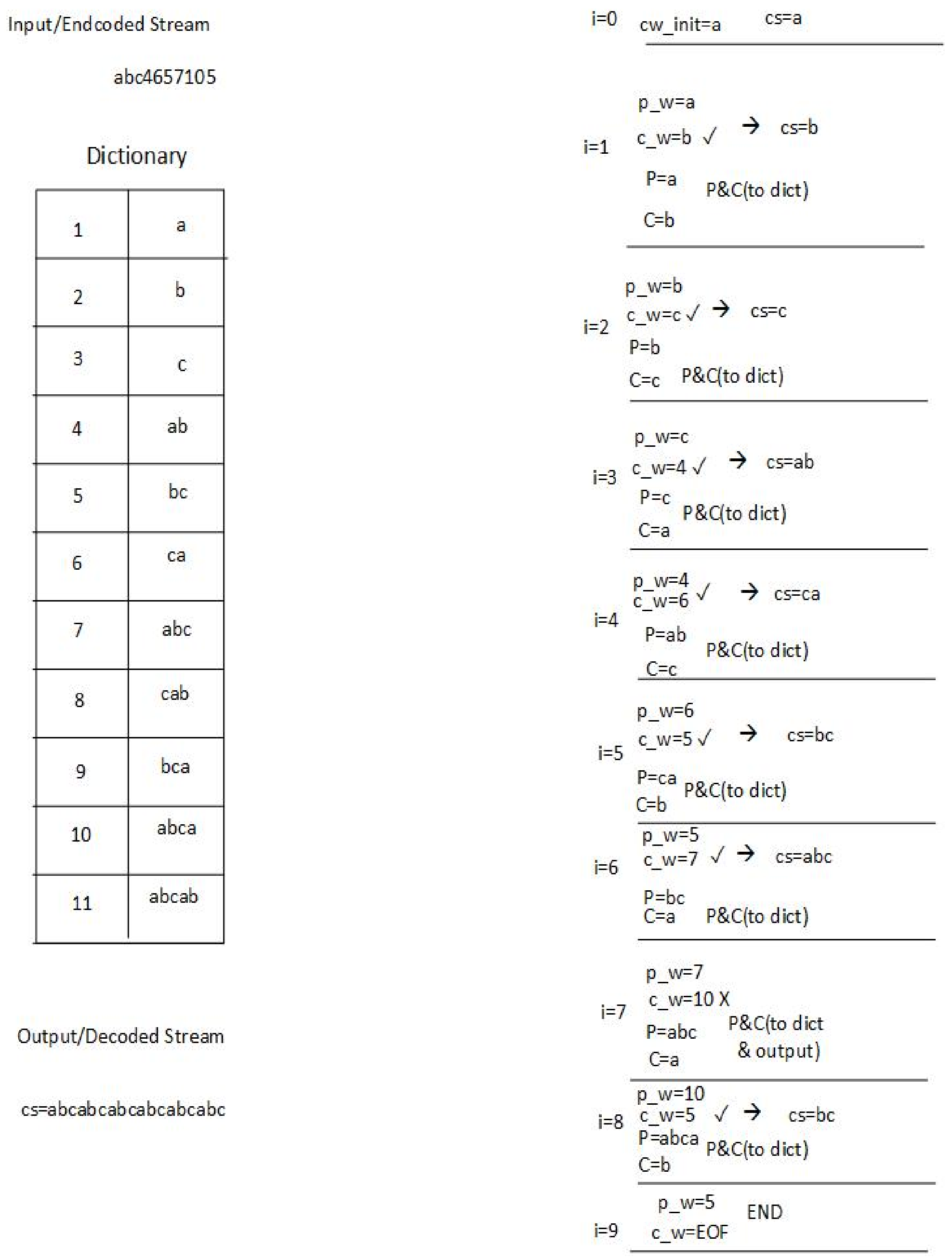
cs = charstream /decoded stream

& = concatenation

P = string corresponds p_w

C = first character of string c_w or p_w respectively

Variable cw_init reads the first character and is written to the output right away. It's 100% that the first element of the encoded stream is a character and not a codeword. Again the dictionary is initialized with the three possible characters and their indexes, 1=a 2=b 3=c. At the second iteration variable P is filled with the most recent value of p_w and C with the first character of string corresponds to c_w. Concatenation of P and C (P&C) is the new entry to the dictionary. p_w always holds the current value of c_w that reads the next value of the encoded stream. The same procedure followed until the 8th iteration i=7. There is the occurrence of the exception. As we can see, the decoder comes across an index of 10 while the entry that belongs there is currently being processed. The only case in which this special case can occur is if the substring begins and ends with the same character ("abca" is of the form <char><string><char>). So, to deal with this exception, you simply take the substring you have so far, "abc", and concatenate its first character to itself, "abc"+"a" = "abca", instead of following the procedure as normal. In order to get more familiar with this exception, the only way is by trial and error.



1	a
2	b
3	c
4	ab
5	bc
6	ca
7	abc
8	cab
9	bca
10	abca
11	abcab

i=0 cw_init=a cs=a

i=1 p_w=a c_w=b ✓ → cs=b P=a C=b P&C(to dict)

i=2 p_w=b c_w=c ✓ → cs=c P=b C=c P&C(to dict)

i=3 p_w=c c_w=4 ✓ → cs=ab P=c C=a P&C(to dict)

i=4 p_w=4 c_w=6 ✓ → cs=ca P=ab C=c P&C(to dict)

i=5 p_w=6 c_w=5 ✓ → cs=bc P=ca C=b P&C(to dict)

i=6 p_w=5 c_w=7 ✓ → cs=abc P=bc C=a P&C(to dict)

i=7 p_w=7 c_w=10 X P=abc C=a P&C(to dict & output)

i=8 p_w=10 c_w=5 ✓ → cs=bc P=abca C=b P&C(to dict)

i=9 p_w=5 c_w=EOF END

Figure 3.6: decompress example

.

Chapter 4

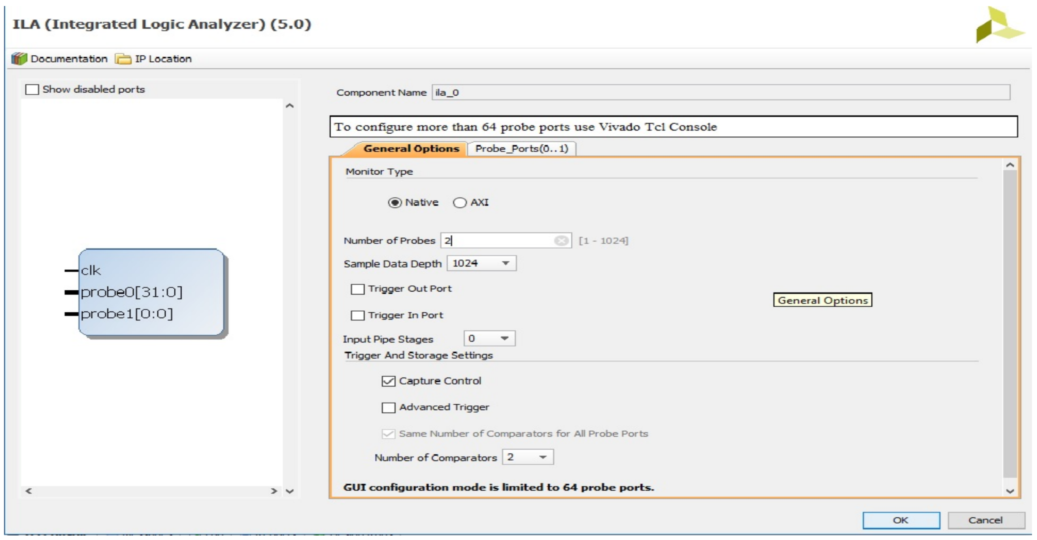
System Implementation

4.1 Customizing ILA core

Inserting ILA in a design can be possibly pre-synthesis or post-synthesis. The insertion flow for the signals differs in these states only a little. Pre-synthesis procedure maybe characterized as more "manually" than post-synthesis where the netlist is used to locate the signals. The netlist provides a better and structured typification of the design than a block diagram or a source code, so it's easier to search for the desired signals. Now lets see how the core is set up.

4.1.1 Setting up the core

Pre-synthesis set up is done by double clicking the ila ip in the block design, and post-synthesis is done by selecting the Set Up Debug wizard. The dialog box with the core options is almost the same in both cases, an example viewed in figure 3.1 and figure 3.2.


ILA (Integrated Logic Analyzer) (5.0)

Component Name: ila_0

To configure more than 64 probe ports use Vivado Tcl Console

Monitor Type: [✓ Native] AXI

Number of Probes: 2 [1 - 1024]

Sample Data Depth: 1024

Trigger Out Port: ☐

Trigger In Port: ☐

Input Pipe Stages: 0

Trigger And Storage Settings

Capture Control: ☑

Advanced Trigger: ☐

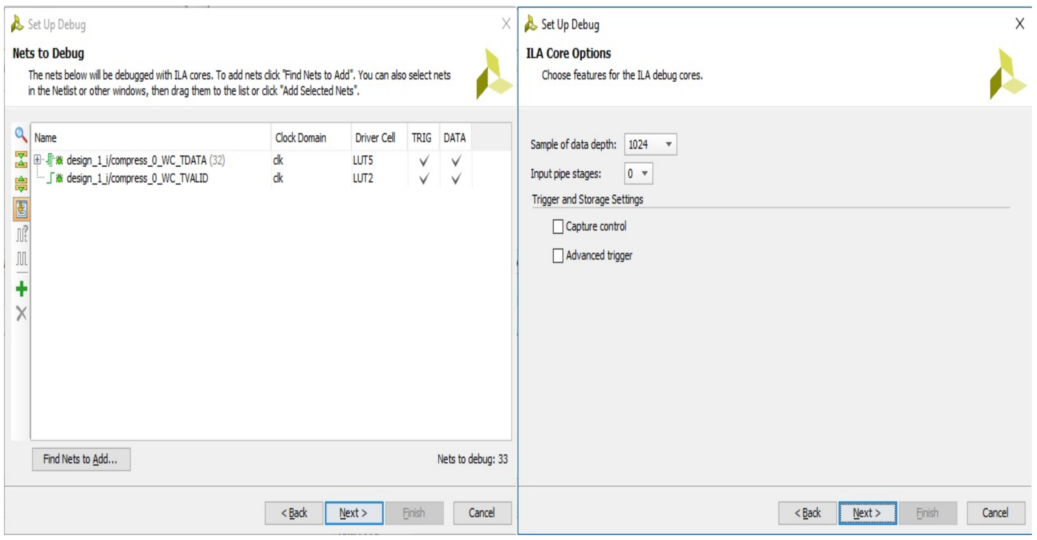
Same Number of Comparators for All Probe Ports: ☑

Number of Comparators: 2

GUI configuration mode is limited to 64 probe ports.

Figure 4.1: pre-synthesis dialog box

.


Set Up Debug

Nets to Debug

The nets below will be debugged with ILA cores. To add nets click "Find Nets to Add". You can also select nets in the Netlist or other windows, then drag them to the list or click "Add Selected Nets".

Name	Clock Domain	Driver Cell	TRIG	DATA
design_1_i/compress_0_WC_TDATA (32)	clk	LUT5	✓	✓
design_1_i/compress_0_WC_TVALID	clk	LUT2	✓	✓

Nets to debug: 33

ILA Core Options

Choose features for the ILA debug cores.

Sample of data depth: 1024

Input pipe stages: 0

Trigger and Storage Settings

Capture control: ☐

Advanced trigger: ☐

Figure 4.2: post-synthesis dialog box

.

4.1.2 Core instantiation

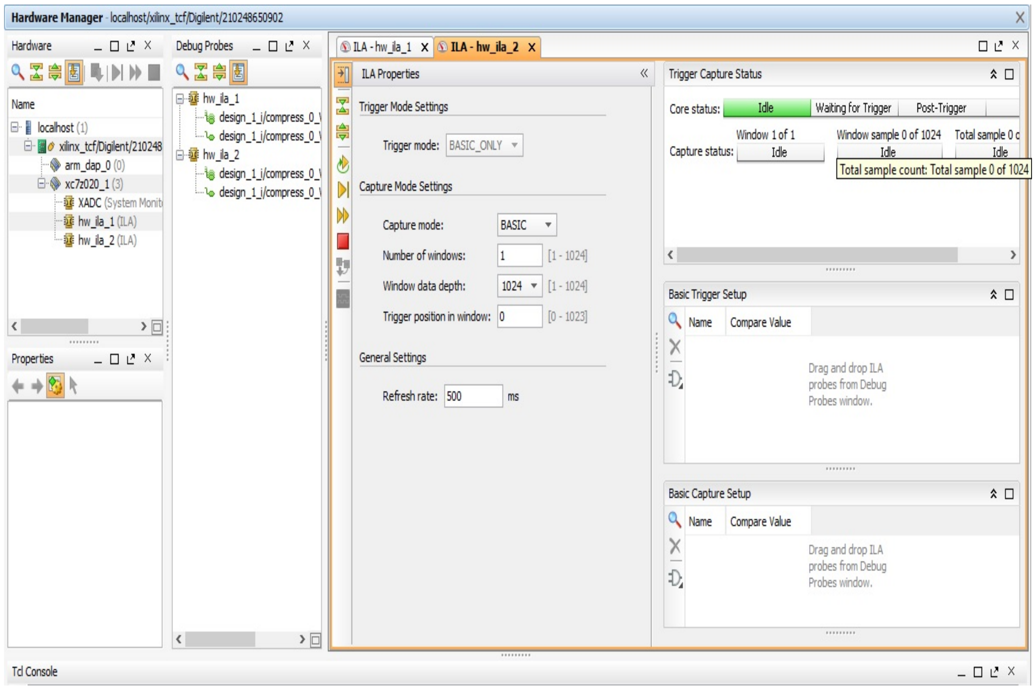
As shown in figure 4.1 the dialog box consists of two tabs. The first one is the General options for the core configuration. There are two monitor types of ILA IP, Native and AXI. AXI interfaces groups signals in a single port, so there is the ability to monitor all of them together. Native type obviously monitors signals separately. Continuing with the native type, cause that type used, we define the number of probes. In our case the number of probes was two, but the range of them is 64 within the GUI configuration, and for more than 64 probes, we should use a tcl command. Then, the Sample Data Depth drop down menu prompts to select the suitable number for us. We set this number to 1024 samples. There are two check boxes in trigger and storage settings. Capture control option allows the user to have a supervision of the debugging progress(more on hardware manager section). Advanced trigger option is an extra triggering condition. It's a state machine-based trigger sequencing. This is used when there is need to begin samples-capturing when multiple events occur consecutively. In the tab Probe_Ports we select each probe's width according to the number we selected on the previous step. Width's range extends from 1 to 4096. Post-synthesis dialog box first shows the nets that already selected for debug. This is done by marking a signal from the netlist for debug. Nonetheless, we may add more nets if we want with the Find Nets to Add option. When a net has marked, we don't need to define the related probe, this dialog box does this automatically. The rest of the steps as shown in figure 3.2 are identical.

One thing that we must accentuate, is that the samples data depth multiplied with the width of the probes, is really the memory that ila will use from the available on-chip buffers i.e Block Rams.

4.2 Hardware Manager

The hardware manager connects the target board(ZedBoard) with the computer. A hardware server that is locally running in our machine identifies the board and establishes a communication with it. Communication with the ILA core and the Vivado environment is conducted using an auto-instantiated debug core hub that connects to the JTAG interface of the FPGA. This hub is responsible for transferring the captured data through the JTAG cable to the tool in order to view them to the user. Once the connection completes successfully, we should see our device under the localhost in the Name window. The next step is to program the device with bitstream and the Debug Probes file. Alternatively we must refresh the device, because the device can also be programmed by the software development kit. The ila cores shows up under the device when the bitstream is loaded to the device. When ila cores are detected upon refreshing a hardware device, the default dashboard for each debug core is automatically opened.

4.2.1 Ila Dashboard



Trigger mode: BASIC_ONLY

Capture mode: BASIC

Number of windows: 1 [1 - 1024]

Window data depth: 1024 [1 - 1024]

Trigger position in window: 0 [0 - 1023]

Refresh rate: 500 ms

Core status: Idle | Waiting for Trigger | Post-Trigger

Capture status: Window 1 of 1 Idle | Window sample 0 of 1024 Idle | Total sample 0 o Idle

Total sample count: Total sample 0 of 1024

Figure 4.3: ila dashboard

.

Every default dashboard contains windows relevant to the debug core the dashboard is created for. The default dashboard created for the ILA debug core contains five windows as shown in figure 3.3:

- Settings window
- Status window
- Trigger Setup window
- Capture Setup window
- Waveform window

The interaction with core represented from those windows.

Settings window contains the trigger mode option, which is BASIC_ONLY in our case. The other option is ADVANCED and is the triggering using state machine. The ILA core can capture data samples when the core status is Pre-Trigger, Waiting for Trigger, or Post-Trigger. The Capture mode control is used to select what condition is evaluated before each sample is captured. ALWAYS option, stores a data sample during a given clock cycle regardless of any capture conditions. BASIC option, stores a data sample during a given clock cycle only if the capture condition evaluates true. We selected the second one. The ILA capture data buffer can be subdivided into one or more capture windows, the depth each of which is a power of 2 number of samples from 1 to (((buffer size) / (number of windows)) - 1). The Trigger position control in the Capture Mode Settings window sets the position of the trigger marker in the captured data window.

The Trigger Capture Status window displays the current state of Capture and the Core status. When the ILA core is idle and waiting for its trigger to be run, if the trigger position is 0, then the ILA core will transition to the Waiting for Trigger status once the trigger is run, otherwise the ILA core will transition to the Pre-Trigger status. Waiting for Trigger box indicates that the core is armed and is waiting for the trigger event to occur. Finally, when the ILA core capture buffer is full and is being uploaded to the host for display, the core will transition to the Idle status once the data has been uploaded and displayed.

Basic Trigger Setup window defines the trigger condition. We used two probes with a Boolean "AND" calculation.

4.3 LZW Algorithm modification

First of all the dictionary. It is implemented as an array of structures consist of three variables, index, character, prefix. The size of this array set to 1024 so as to have a metric of how much gain we have in memory compared with the ILA. That is, the length of the decoded stream - 1024 is the profit we succeeded in memory. If for example the total characters after decompression are 3000 then we have gain 1976 memory locations that alternatively would have been used by the ILA. Also, in dictionary we act on three different ways. Initialization(dictionary_init) search(dictionary_lookup) and insert(dictionary_add). Index variable does not need specific initialization because the values which potentially will obtain are known. Then we initialize prefix and character variables properly for efficient search. Apparently searching in hardware is a static process, namely we would have to make 1024 accesses to memory every time. Prefix variable initialized at minus one and character variable at every possible known value, we will see that after. When we find a prefix variable at minus one means that this particular cell is empty. So the search stops at this point, and we don't have to search 1024 cells every time. At the moment of insert, the initial or the current value of the variables changes and they are not empty any more.

Next is our intervention to "known" values. As we have already mentioned, software implementation recognizes all ASCII characters i.e 256 values, in contrast with a hardware implementation that these characters are simply values of bits. So, the values from 0 to 255 are recognizable from the algorithm as the alphabet of the incoming stream, and values from 255 to [upper limit] are the codewords or the dictionary indexes. We can adjust the number of "known" values at will. This number allows us to have a width range for the signals that will be probed by the LZW IP. We set this number to 999 which are all 3-digit numbers. 256 "known" values are $2\hat{8}=256$ i.e 8-bits signal, 512 "known" values are $2\hat{9}=512$ i.e 9-bits signal, 1024 "known" values are $2\hat{1}0=1024$ i.e 10-bits signal. This is a restriction of IFT. It can trace n-bit signals but values just from 0 up to 999, obviously the rest of the values are codewords. Here is clear that the initialization must contain all possible "known" values i.e 0-999.

Another modification related with the output. We are aware of how the algorithm works so far. Namely, when a match occurs in dictionary search the algorithm does not change the output stream and this is something we don't want. Each clock cycle the ILA checks the trigger condition and accordingly then captures the value, or not. So, we set the output to minus one when we have successful search to the dictionary and that indicates that the ILA will not record this value. We need to adapt and the output when the input stream ends. The end of the stream may happen when we have a match in dictionary or at the opposite case. The output at the first case is just the index of the current string in dictionary. When we dont have a match and we are at the end of the stream we have to output the current and the previous word, so we pack them in a single variable. The end of the stream is indicated by an input variable.

4.3.1 HLS Implementation

Lzw hardware approach through hls is carried out with directives. Directives will specify to the compiler how to manage the C-source code and interpret it to hardware primitives. Our dictionary is a region in memory in plain. But with the appropriate directive, this memory maybe implemented as a ROM, a BRAM or a group of individual registers. We implemented the dictionary as a BRAM. For dictionary initialization we used a performance optimization directive called UNROLL. This directive unrolls a loop and improves parallelism. It's a fixed-bound loop so this directive can be applied to this operation. On the other hand when we search to dictionary we are not able to apply any directive. This happens because is undefined when the search will stop. Searching in dictionary has multiple exit conditions, namely three, so no directive can be applied there. Inserting data to BRAM don't need directive.

The rest of the directives we used are about the interface of the core and the control. In fact the control of the IP is also an interface. So, these directives are applied to the ports of the core, and are AXI-Lite for the control and the input, and AXI-Stream for the output. There are four signals to control an HLS IP. These are included to a bus interface called ap_ctrl and are ap_start, ap_idle, ap_ready, and ap_done as shown in 4.4. The directive specifically is applied to this bus. Every hardware accelerator has a purpose. A task is assigned to it, so it needs a protocol for communication with it's controller. For every IP block generated by the Vivado HLS tool, a complimentary API is automatically created to enable software development for the processor. It's more less clear what these signals indicate, but they are used through function calls by the processor when we export our hardware for software development. First of all, the processor identifies the instance of the core and stores information which is required by all other driver functions to ensure communication with the correct hardware module. When this is done, the start function, XCompress_Start(), pulses the ap_start signal on the target hardware accelerator. This is a 1 clock cycle pulse that initiates the operation of the IP block. This operation forces the signal ap_idle to go low, the corresponding function XCompress_isIdle() just informs the processor if the core is running or not when is called. It is a checking function. This function returns false from the time the start function is executed by the processor until the ap_done signal is asserted by the IP block. Also, ap_done and ap_ready with the related functions are just a report to the processor, with a small difference. A task completion forces the ap_done signal to go high but the core may not be ready presently for the next value, so the signal ap_ready will stay at low until the core is ready.

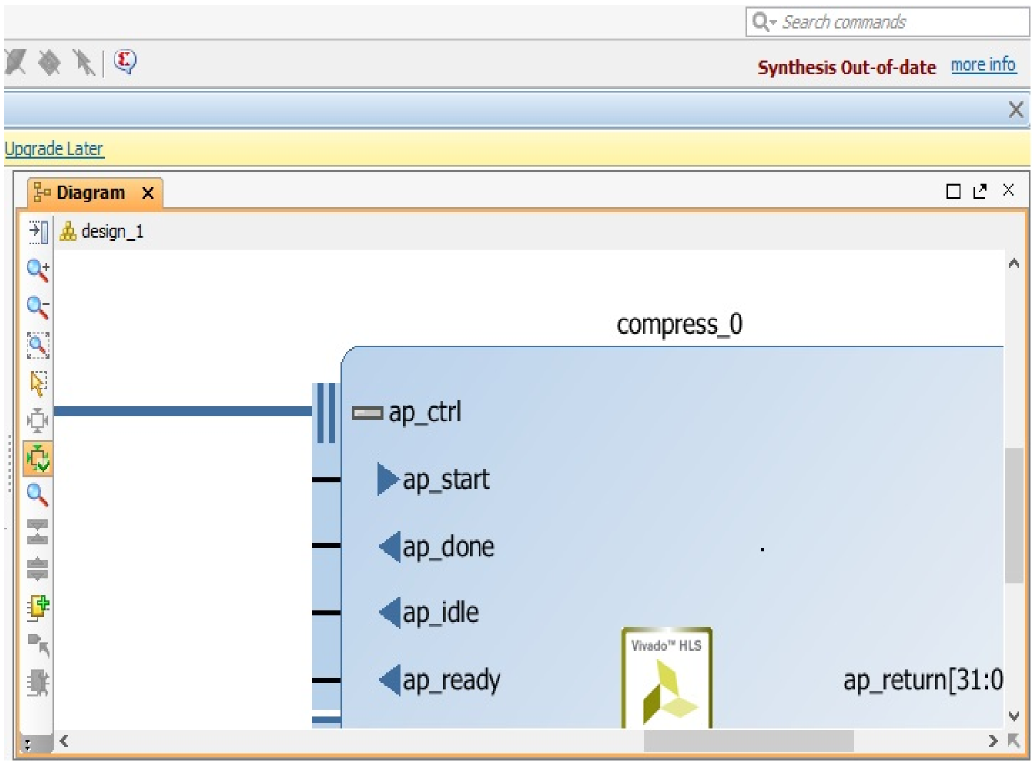


Figure 4.4: ap-ctrl signals

.

Axi-stream interface(4.5) consists of three signals by default, data, ready and valid, there are also more optional signals for the actual purpose of this interface but it's out of this context. These are grouped to a bus as well. Ready is the input of this bus and data & valid the output. This protocol suits us to "clasp" properly with the ILA. In order to prevent the ILA to record trash values, we need to inform the analyzer whenever we want to record, and this is done through valid signal. In essence the condition which is checked every clock cycle by the ILA is (data$\neq$-1 & valid=1).

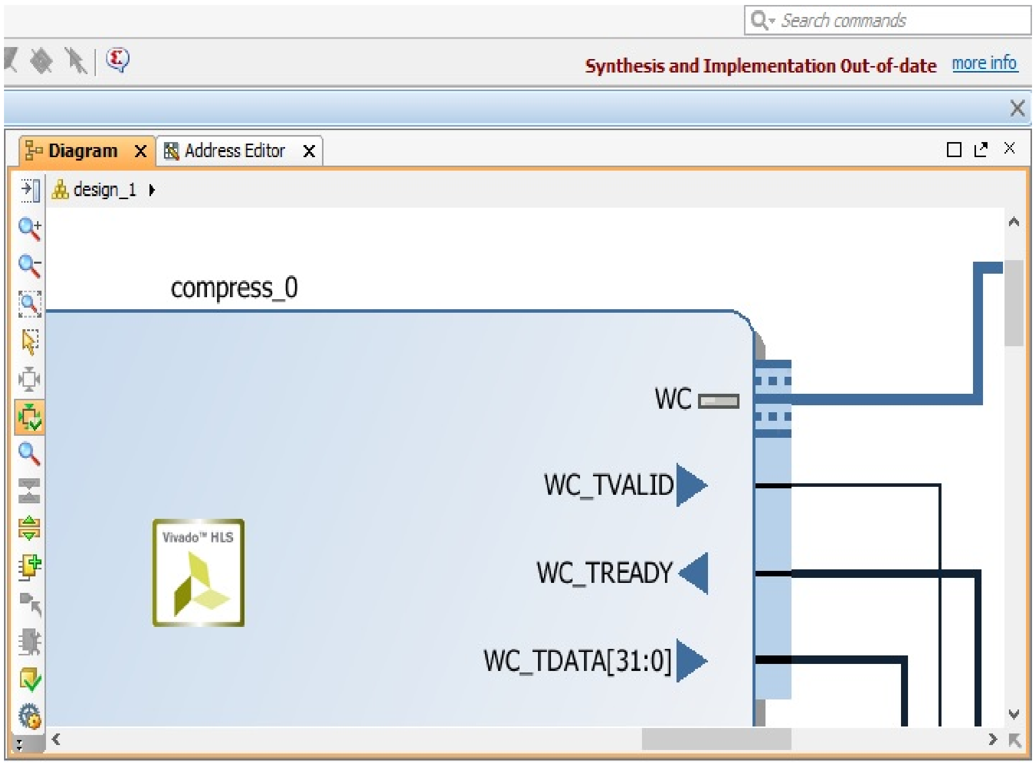


Figure 4.5: axi-stream signals

.

AXI-Lite is a light-weight, single transaction memory-mapped interface, and that's why we used it. We check the current state of the core and we pass the next value to it accordingly, one value per time.

4.4 IFT Software part

Vivado provides a set of tcl commands(Tool Command Language) which perform all operations of the tool. One of them is used to export the captured data from the ILA in .csv file(comma separated values). The command is written in tcl console as follows after the ILA ends data acquisition.

write_hw_ila_data -force my_waveform.zip

This file lists the Sample in Buffer, Sample in Window, TRIGGER and the captured values of the signals. Every element and the values of it, are formed in a column which is separated with a comma from the "neighbor". Through a program written in python we target the desirable column and we access the file vertically. This program also implements the decompress. Decompression algorithm needs no modification. The program simply converts the binary values from the .csv file to integers, and outputs the initial stream to a .txt file. The values in this file are written with a blank space between of them in order to help us read them properly to the last step.

Here we present the test-data of our project. As we have emphasized earlier, ARM processor is the controller and the data generator of the design. An application which is running on the processor takes care of both of them, and the base source code obtained from [23]. It's a complete tutorial for hls designs and covers all aspects of developing and programming hardware accelerators right to the end. We wrote a small script and we injected it inside this code. This part of the code is responsible for the value that is gonna be an input to the compression module. Of course we await a repetition of data in order to achieve compression. So this code simply counts up and down to limits defined by us. We set the code to count up and down 20 values, particularly from 48 to 68 and from 68 to 48 and over again. The algorithm for this counting placed at appendix.

Final thing is the display of the decompressed data to GTKWave viewer. The viewer takes as input a .vcd file. VCD is an ASCII-based format for dumpfiles generated by EDA logic simulation tools. The vcd file consists of three sections, Header section, variable definition section and value change section. Header section includes, the date-timestamp, the version of the simulator and the timescale which is the time interval between a value change in simulation matter. Variable definition section contains scope and variables declaration. Each variable is assigned an ASCII character from ! to ˜ (decimal 33 to 126) for use in the value change section. After variables definition follows the variables initialization. Then the value change section, lists all signals multi-bit or single-bit and their new value every time indicated by a timescale pointer.

4.5 Evaluation

Above we presented the IFT, a system that includes a hardware implementation of LZW compression algorithm which "cooperates" with the xilinx ILA in order to demonstrate the contribution of this compressor in the context of hardware debugging process. However we compare these 2 IP's so we can have a final overview of the LZW compressor. The table below lists the resources used from the ILA and the LZW IP under the same configurations. By the same configurations we mean that we want to answer this: In order to capture the same amount of samples, how much of the given resources are utilized??? The numbers of the samples that we demonstrate is an example that have a small deviation from one another. Obviously ILA IP has a static number of samples that can record, to the other hand we should try out some tests so we can achieve a number of samples recorded by the LZW IP that approaches the number of the ILA. Apparently the samples recorded by the LZW IP is the data that we get after the decompression. So, as we can see below the compression module performs better with a decent gain in memory(about two times less) which is our purpose. Of course this happens if a repetition of data occurs in a stream-test data.

Continuing, we isolated these modules and we observed the resources used from each one for the process of debugging from it's scope. These are the results as we obtained them from the report of the resource utilization by the corresponding tool, vivado for the ILA and HLS for the LZW IP. Every design that is downloaded on fpga is translated to hardware primitives. These are, LookUp Tables(LUTs) FLip-Flops and Brams.

IFT* is the system in a standalone mode. That is, the only memory we count is the memory of the compression module, and the ila is used as an intermediate buffer with the outside world and consumes no memory at all

Table 4.1: ILA-LZW Comparison

	LUTs	**Flip-Flops**	**BRAMs**	**Samples Recorded**
Xilinx_ILA	957	1405	15	2048
IFT*_1024	532+957	389+1405	6+1	2.309
Gain	-	-	2.14x	1.12x

Chapter 5

Conclusion-Future Work

Summing-up, a certain thing is that using the compression module for debugging purpose is a trade-off process as we mentioned earlier. IFT system consumes significant memory for compression module but potentially it may be offer a gain in space. Although it's serious drawback is in execution time. LZW IP has an internal process for any incoming input that may take several clock cycles for the output to be produced, from a few dozen to thousands. Using IFT system, possibly, a gain in memory is the profit but with a considerable time delay.

So, the first thing we could try in the future is to optimize the IP. The presented work on LZW IP is implemented with HLS in order to use to the zynq processing system. Therefore we must evaluate if it's necessary to have an IP like this in real-life with DUT's. Avoiding HLS will offer code optimization and consequently a better performance, because HLS overloads the designs.

Another thing that could be an object for further research and work, is the most important one, the execution time. Searching serially in the memory of the dictionary every time is the handicap of the IP. A smarter way of searching or a better and more efficient memory design oriented appropriately, is what we need. Using CAMs(Content-addressable memory) for that reason is the first thing we could try. From related works came out that this type of memory increased the performance of the implementation.

Commonly is used for very-high-speed searching applications.

Final and most ambitious thing we would be able to try is the decompression in hardware. This is a subject of extensive discussion because from performing LZW compression and decompression altogether may arise new things that we haven't even imagine yet.

Appendix A

Images

Figure A.1: giphy image

.

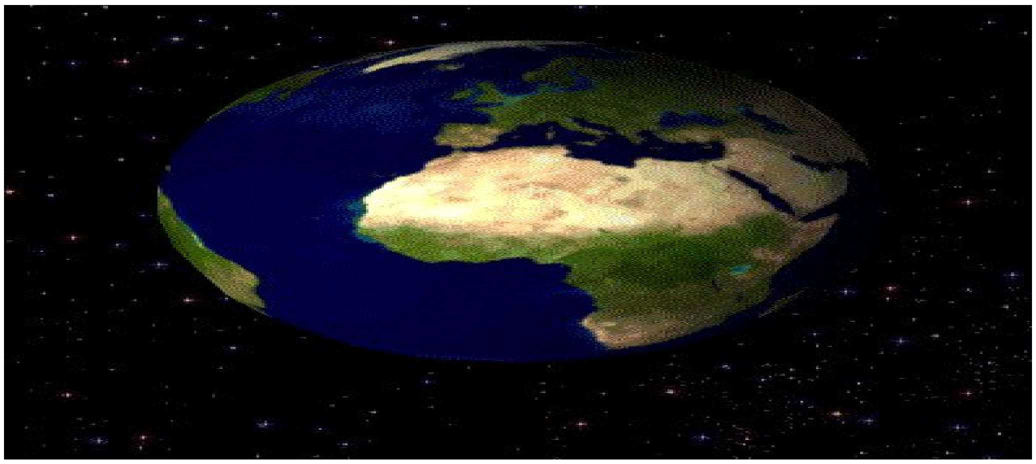


Figure A.2: rotating earth image

.

Appendix B

Code

This is code for counting up and down

a)variable ud(1=up 0=down) is the initial direction.

b)temporary variable tmpud holds the value of ud as long as it hasnt reach upper or lower limit.

c)the first if, checks the value of number is between the limits in order to change the value of ud or leave it as it is.If the number reaches one those values then the counting(ud) changes direction. The second if, checks the value of ud and if it is one(1) increments number or if it is zero(0) decrements it.

```
tmpud=ud;

if(number>=upper_limit)ud=0;
else if(number<=lower_limit)ud=1;
else ud=tmpud;

if(ud==1)number=number+1;
else number=number-1;
```

Bibliography

[1] A. L. Herrmann and G. P. Nugent, “Embedded logic analyzer for a programmable logic device,” May 14 2002, uS Patent 6,389,558.

[2] http://www.eit.lth.se/sprapport.php?uid=900

[3] M. b. Lin, J. f. Lee, and G. E. Jan, “A lossless data compression and decompression algorithm and its hardware architecture,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 14, no. 9, pp. 925–936, Sept 2006.

[4] E. Anis and N. Nicolici, “On using lossless compression of debug data in embedded logic analysis,” in 2007 IEEE International Test Conference, Oct 2007, pp. 1–10.

[5] E. Anis and N. Nicolici, “Low cost debug architecture using lossy compression for silicon debug,” in 2007 Design, Automation Test in Europe Conference Exhibition, April 2007, pp. 1–6.

[6] http://ieeexplore.ieee.org/document/5718828/

[7] J. Goeders and S. J. E. Wilton, “Using dynamic signal-tracing to debug compiler-optimized hls circuits on fpgas,” in 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, May 2015, pp. 127–134.

[8] http://whatis.techtarget.com/definition/IP-core-intellectual-property-core

[9] https://en.wikipedia.org/wiki/Hardware_acceleration

[10] https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_1/pg172-ila.pdf

[11] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_4/ug908-vivado-programming-debugging.pdf

[12] https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[13] https://www.xilinx.com/support/documentation/ip_documentation /ug761_axi_reference_guide.pdf

[14] http://www.googoolia.com/wp/category/zynq-training/

[15] https://en.wikipedia.org/wiki/Lempel

[16] https://www2.cs.duke.edu/csed/curious/compression/lzw.html

[17] http://michael.dipperstein.com/lzw/

[18] https://en.wikipedia.org/wiki/Lempel

[19] https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node214.html

[20] http://whatis.techtarget.com/definition/LZW-compression

[21] http://www.dspguide.com/ch27/5.htm

[22] https://www.xilinx.com/support/documentation/application_notes/xapp745-processor-control-vhls.pdf

[23] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_1/ug871-vivado-high-level-synthesis-tutorial.pdf

[24] https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/u
vivado-axi-reference-guide.pdf

[25] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

[26] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug936-vivado-tutorial-programming-debugging.pdf