A Lemple -Ziv based Configuration Management Architecture for Reconfigurable Computing

by

Melany Ann Richmond

A project report submitted in partial fulfillment
of the requirements for the degree of

Master of Science

University of Washington

2001

Program Authorized to Offer Degree:

Electrical Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a master's project by

Melany Ann Richmond

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

____
Scott Hauck

____
Carl Sechen

Date: _______________________

A Lemple -Ziv based
Configuration Management Architecture
for Reconfigurable Computing

By Melany Ann Richmond

Chairperson of the Supervisory Committee

Professor Scott Hauck
Department of Electrical Engineering
University of Washington

Abstract

FPGAs are a powerful technology for developing high-performance embedded systems. The density and performance of FPGAs has steadily improved over time, making their use ever more attractive. As FPGAs have increased in density, the size of the configuration bit stream needed to program the device has also grown. As a result, the time necessary to configure an FPGA, as well as the amount of storage needed on the device, has grown. Because of this, techniques have been investigated to decrease the amount of configuration data being sent into an FPGA. One technique in particular, investigated in depth by Hauck [Hauck99b] and Li [Li01], looks at ways where generic data compression algorithms can be applied to reduce the size of the configuration bit stream. Data compression is used in many applications as a means of reducing inherent redundancies found in data representations. Compression techniques typically decrease storage and communication costs associated with writing to a device. Previous research investigates multiple algorithms for configuration bit stream compression [Hauck99b], [Li01]. Based upon that work, this project looks to implement one in particular. A Lemple-Ziv based decompression engine was designed to work with current FPGA technology. The goal of this project was to investigate the feasibility of building such a design into hardware as well as the impact the added hardware would have upon an FPGA.

TABLE OF CONTENTS

List of Figures	ii
List of Tables	iii
Introduction	1
Background – Field-Programmable Gate Arrays	3
Target Device – Xilinx Virtex Series FPGA	8
Target Compression Algorithm – Lemple Ziv	10
Symbol Size Factors	14
Compressed Bitstream Characteristics	15
Design Overview	16
Hardware Details	19
Shifter Component	21
Control Component	26
Hardware Implications	29
Conclusion	36
References	37
Appendix 1: Verilog Model	39

LIST OF FIGURES

Figure 1: Programming bit for SRAM-based FPGAs. 4

Figure 2: Pass Gate connecting two routing resources. 5

Figure 3: A 2-input LUT 5

Figure 4: Typical Island Style routing structure used in FPGAs. 6

Figure 5: Overview of Xilinx CLB [Xilinx99] 7

Figure 6: Virtex architecture overview [Virtex00]. 8

Figure 7: CLB Column Frame Organization 9

Figure 8: FDR buffer overview. 10

Figure 9: Example of LZ77 buffer. 11

Figure 10: Extended FDR buffer to aid LZ compression. 13

Figure 11: Hardware overview 17

Figure 12: LZ Decompression Input and Output diagram 18

Figure 13: Design Architecture Overview. 19

Figure 14: Completed Cadence layout of design. 21

Figure 15: Shifter component overview 22

Figure 16: Shifter Mux with D-Flip Flop. 23

Figure 17: Single shift register schematic 23

Figure 18: Layout of a single shift register. 24

Figure 19: 512 to 1 Multiplexer 25

Figure 20: 512:1 MUX and symbol shifter layout. 25

Figure 21: Overview of Control Hardware 26

Figure 22: Logic for down counter decrementor. 28

Figure 23: Control component layout. 29

Figure 24: Optimized SRAM cell with an area of 52.65µm2 30

LIST OF TABLES

Table 1: Truth table for internal feedback signal. 27

Table 2: Truth table for down counter adder. 28

Table 3: Approximated area for each Virtex-E device **in the 0.25µ process used for this design.**. 30

Table 4: Virtex-E Series FPGA Family Members and corresponding characteristics used for area calculations. [Xilinx00b]. 32

Table 5: Approximated component sizes for shifter 33

Table 6: Area Approximation Calculations for MUX. 33

Table 7: Control Area Approximations 34

Table 8: Overall area approximations for added hardware. 35

Table 9: Approximated Area Comparison. 35

Introduction

Traditionally, algorithms are accomplished in one of two ways. One common method is to utilize a general-purpose computer, using software to implement a given task. This type of process is based upon a central processor with predetermined resources. The software will use the available resources as orchestrated by the processor. The advantage of this practice is the large amount of available flexibility. If the algorithm changes, the software can be changed, usually in a short amount of time. However, with added flexibility there is loss of performance. For many functions, a general-purpose computer can be slow and cumbersome. The performance loss can be attributed to the lack of hardware-optimized components within the general-purpose computer for specific tasks. Without specialized structures optimized for specific functions, some computations will ultimately forfeit performance.

It is impossible to assume that hardware structures for every possible computation could be included on a general-purpose computer. As a result, an approach taken for some algorithms is to implement them using an Application Specific Integrated Circuit (ASIC). This process optimizes the hardware to accomplish a specific problem. For that reason, the ASIC will generally achieve better performance. However, achieving optimization in hardware will cause a loss in flexibility. Once the design has been fabricated, any changes to the algorithm would require that a new ASIC be designed and fabricated. The design and fabrication process can have a long turn-around time, possibly months for a new ASIC. With this in mind, traditional design options force the designer to choose at the start of a design whether flexibility, speed, or even time to market is of greater importance.

Reconfigurable computing serves to bridge the gap between hardware and software, offering greater flexibility than a hardware solution and delivering increased speed over a software solution. In particular, Field Programmable Gate Arrays (FPGAs) are one method making this possible. The FPGA can be utilized to speed-up compute intensive algorithms by configuring the algorithm onto the resources available on the chip. FPGAs are useful for numerous applications. For example, an FPGA can be used in conjunction with a general-purpose computer. In this case, the functions not possible to speed up utilizing an FPGA could be run simultaneously on a general-purpose machine with only the compute-intense algorithms configured on the FPGA. Similarly, for problems requiring more resources than physically available on a single FPGA, multiple devices can be tied together and configured to solve even larger problems.

Configuration compression looks to reduce the time needed to program an FPGA. In the past, time taken to reconfigure a device was of little importance. However, many FPGAs can now be reconfigured during run-time. When an FPGA is configured, the reconfiguration delay directly affects system performance. With larger numbers of applications finding the need for run-time reconfiguration, faster reconfiguration times are becoming much more desirable and needed.

In this project, a hardware architecture was implemented that takes a compressed configuration bit stream and decompresses it in hardware. The design was based upon previous work [Hauck99b], [Li01], [Dandalis01]. The goal of this project was to investigate the feasibility of such a hardware addition to an FPGA by building a Lemple-Ziv (LZ) decompression engine designed to work with current SRAM-based FPGA structures. At the same time, the project looked to investigate the impact that the added hardware would have upon the size of an FPGA.

In this paper, you will first find a short background on FPGAs. Second, an overview is given for the target device that could benefit from this type of hardware, Xilinx Virtex FPGAs. Next, an overview of the Lemple-Ziv compression is presented. Following that, a discussion on the symbol size factors can be found. Subsequently, a discussion on the configuration bit stream characteristics needed for LZ decompression hardware is presented. Next, an overview of the design is presented with a more detailed discussion closely following. the hardware design is discussed and each of the major components is explained. Finally, the hardware impact upon current Virtex FPGAs are discussed..

Background – Field-Programmable Gate Arrays

FPGAs contain configurable logic blocks (CLBs), input-output blocks (IOBs), memory, clock resources, programmable routing, and configuration circuitry. These logic resources are configured through the configuration bit stream allowing a very complex circuit to be programmed onto a single chip. The configuration bitstream can be read or written through one of the configuration interfaces on the device.

FPGAs have been developed using various programming technologies. Two of the most well known types of programmable devices are antifuse-based or SRAM-based. Antifuse FPGAs are one-time programmable devices. An antifuse is initially an open connection. In order to program this type of device, high voltage is applied high enough to destroy the antifuse. When an antifuse is “blown” a connection is permanently made. At the same time, “unblown” antifuses are open connections. Thus, once the antifuses are blown the FPGA is now permanently programmed. On the other hand, an SRAM based FPGA would be Reprogrammable. Therefore, the permanent nature of an antifuse makes this type of FPGA not useful for reconfigurable applications, such as the target application for this project.

At this time, SRAM programmable FPGAs are very popular for reconfigurable applications. For such a device, SRAM cells, as shown in Figure 1 below, are connected to the configuration points within the FPGA. The configuration data from the input bit stream is written to the SRAM cell. The outputs, Q and $\overline{Q}$, connect to the FPGA resources. The actual control of the FPGA is therefore handled by the outputs of the SRAM cells scattered throughout the device. Thus, an FPGA can be programmed and reprogrammed as simply as writing to and reading from a standard SRAM.

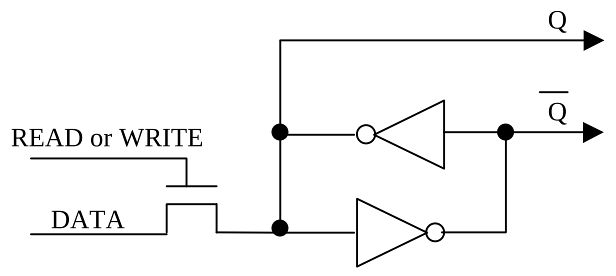


Figure 1: Programming bit for SRAM-based FPGAs.

In Figure 1 above, a SRAM cell is made up of two cross-coupled inverters and a pass-transistor. The pass-transistor is included to read from or write to the cell and is controlled by an enable signal. When the enable signal is set true, the data line is connected and now can write over or read from the current stored value. When the enable signal is set false, the cross-coupled inverters work together to maintain the value stored in the cell. However, the upper inverter could adversely affect the cell's ability to reprogram simply by reinforcing the old value to the input at the same time as data is being written. Therefore, the upper inverter and pass-transistor must be balanced by the ratio of gate sizes such that a signal written in from the data will be able to overpower the upper inverter.

In an FPGA, the routing is configured using a pass-gate structure. When a program bit is set true, the two routing resources are connected as shown in Figure 2 below. The pass gate completes the circuit, allowing a signal to flow from one routing resource to another. On the other hand, if the program bit is configured to be false, the connection is left open and the routing resources are not connected.

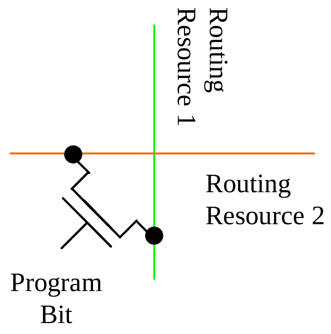
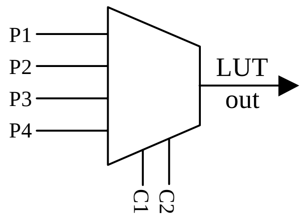


Figure 2: Pass Gate connecting two routing resources.

Lookup Tables (LUTs) are another useful structure normally included on FPGAs. These are small memories provided for computing arbitrary logic functions. By design, LUTs can compute any Boolean logic function with *n* inputs. This is done by connecting 2^n program bits to the multiplexer on the LUT. The LUT holds truth table outputs within the memory instead of computing the output directly through combinational logic. In a LUT, multiplexers implement logic function by choosing from the program bits in the table. For example, in Figure 3 below, n is equal to two, meaning there are four possible outputs. If all the program bits of the 2-input LUT were set to false except for the one corresponding to a control signal equal to 00 (P1), the LUT in the figure would act like a 2-input NOR gate. Alternatively, if all the program bits were set to true except 11 (P4), the LUT would operate like a 2-input NAND gate. Therefore, depending upon how the program bits are configured, any 2-input function can be implemented using the LUT. This means for any *n*-input LUT, any *n*-input function can be implemented.



C1 C2	NAND	NOR	
00	1	1	P1
01	1	0	P2
10	1	0	P3
11	0	0	P4

Figure 3: A 2-input LUT

Routing structures on FPGAs conventionally are an island-style layout as shown in the Figure 4. This type of design surrounds the logic structures on all sides with horizontal and vertical routing channels, permitting arbitrary communication between resources. For each Logic Block, the inputs and outputs connect to the routing channels through a programmable Connect Block. Switch Blocks are used to change routing direction by allowing junctions of horizontal and vertical connections. This type of structure ensures that any arbitrary circuit can be mapped and routed onto the FPGA.

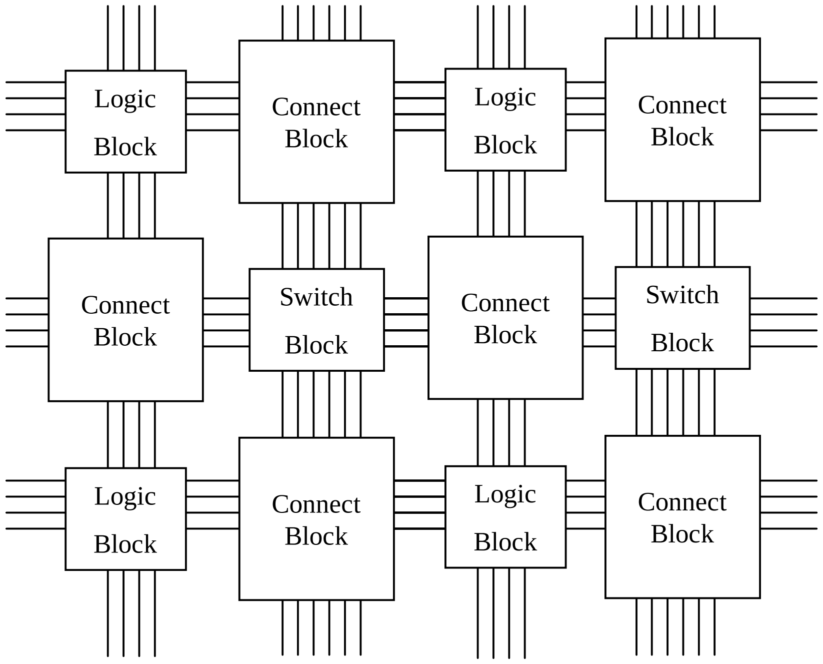


Figure 4: Typical Island Style routing structure used in FPGAs.

Within an FPGA, Logic Blocks implement functions using a combination of multiplexers, LUTs and flip-flops contained within each block. Program bits are connected to the control and/or the data lines of each multiplexer. The program bits select between outputs of different logic resources within the interconnected logic blocks. An overview of a Xilinx configurable logic block (CLB) can be seen in Figure 5. Many SRAM-based FPGAs will use an architecture very similar to this.

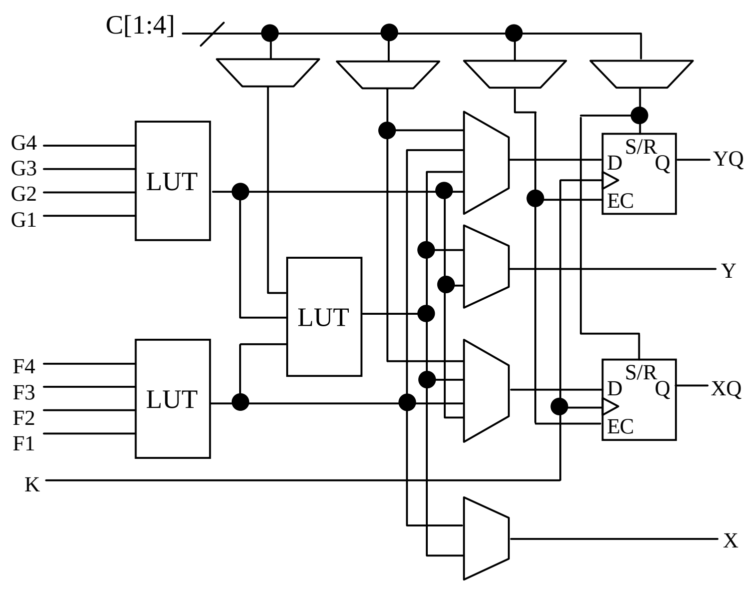


Figure 5: Overview of Xilinx CLB [Xilinx99]

The CLB structure is very flexible, able to compute complex functions. In the simplified CLB overview shown above, the overall functionality of the LUTs, multiplexers, and flip-flops is determined by the configuration bits. Utilizing the three LUTs, it is possible to implement a nine-input logic function. For smaller functions, the CLB could compute any two four-input functions, utilizing the two four-input LUTs separately. Control signals provide an enable as well as a set or reset for the flip-flops. They also provide a means to connect directly to the flip-flops. At the same time, the control signals control another input connection for the third LUT. The inputs F and G are connected to the CLB from routing adjacent to the block. The outputs will connect to other CLBs through the routing wires also adjacent to the block. Utilizing the numerous CLBs in conjunction with all the other components contained on an FPGA, complex systems can be implemented.

Target Device – Xilinx Virtex Series FPGA

Like the generic FPGAs discussed above, Virtex devices contain configurable logic blocks (CLBs), input-output blocks (IOBs), memory (SRAM), clock resources, programmable routing, and configuration circuitry. The CLBs are the functional units for constructing logic. Furthermore, the IOB blocks provide and interface between the CLBs and the package pins on the FPGA. The Virtex architecture overview can be seen in Figure 6. These resources are configured through the configuration bitstream. The configuration bitstream is made up of a mixture of commands and data and is typically read by the Virtex device through one of the configuration interfaces.

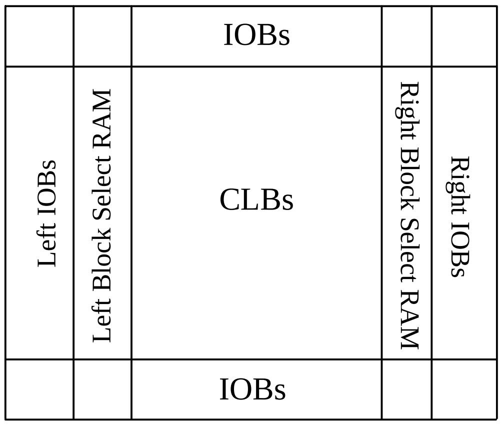


Figure 6: Virtex architecture overview [Virtex00].

The configuration memory is traditionally thought of as a rectangular array of bits. The bits are grouped into vertical, one-bit wide frames extending from top to bottom in the array. A frame is the smallest portion of the configuration memory that can be written to or read from at any given time. Frames are grouped together into columns. In the different types of Virtex devices there are different designations for the columns. Typically these will include a center column, which includes the configuration for the global clocks. Two IOB columns will represent the configuration for all the IOBs on the edges of the device. Most of the remaining columns are devoted to CLBs. Lastly, the remaining columns are RAM.

The frames are situated such that they are vertical in the device. In this manner, the "front" of the frame appears at the top. In Figure 7 below, (shown horizontally in the figure) the top 18 bits of the column would specify the IOBs located at the top of the columns. Next, 18 bits are designated for each of the *n* CLBs, where *n* is determined by the type of device. Lastly, 18 more bits will specify the bottom IOBs. The actual Frame size varies, depending up the number of rows in the device. The number of configuration bits in a frame is $18 \times (\#\,\text{CLB rows} + 2)$ plus extra padding bits. The padding bits are used because the configuration bitstream is written into 32 bits words beginning with the top of the frame. In the case where the last 18-bit word does complete an entire 32-bit word, "padding" bits are added onto the right in the form of extra zeros [Virtex00].

Top IOBs	CLB R1	CLB R2	...	CLB Rn	Bottom 2 IOBs
$\longleftrightarrow$ 18 bits	$\longleftrightarrow$ 18 bits	$\longleftrightarrow$ 18 bits		$\longleftrightarrow$ 18 bits	$\longleftrightarrow$ 18 bits

Figure 7: CLB Column Frame Organization

The circuit configuration for a Virtex device is done using the Frame Data Input Register (FDR). The FDR is a shift register into which the data is loaded prior to transfer to the many configuration points within the device. An overview can bee seen in Figure 8, below. In particular, the starting address of the consecutive frames about to be configured is loaded into the FDR and then transferred to the frames in the order specified [Li01].

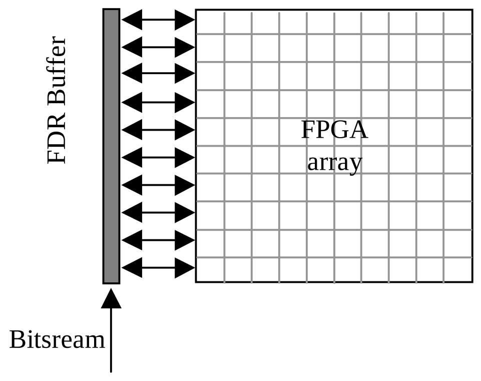


Figure 8: FDR buffer overview. The configuration data for each frame is written to the FDR and then transferred to the frames within the FPGA array.

Target Compression Algorithm – Lemple Ziv

Lemple-Ziv (LZ) is a generic compression algorithm utilizing regularities in a bitstream. LZ works by assigning frequently occurring groups of symbols to a dictionary. When the symbols appear in the data, they are replaced with a reference to the location in the dictionary. The reference will tell where the match previously occurred, how long the match is, and give a new symbol. By this means data can be represented in a much smaller form.

There are multiple versions of LZ compression; LZ77, LZ78 and LZW being the most common. LZ78 and LZW both generate better compression over a finite bitstream compared to LZ77 [Li01]. However, LZ78 and LZW both utilize static dictionaries. For this type of design a look-up table holding the recurring symbols would need to be built in hardware. This method has been implemented in previous research [Dandalis01]. Adding a look-up-table means that in order to decompress, LZ78 and LZW have a larger impact on the necessary hardware. Because of the hardware impact, these versions of LZ were not considered for configuration compression [Li01]. On the other hand, LZ77 utilizes a dynamic dictionary. For this type of design the added hardware would be a Shifter and Control; a look-up table would not be needed. Instead of utilizing a look-up table, the shifter would serve as the fixed size dictionary, with the location of a match used as a reference address. As a result, LZ77 has a smaller impact upon the extra hardware needed for decompression. Therefore, this is the version chosen to integrate configuration compression onto an FPGA.

The LZ77 compression algorithm holds the last *n* symbols of the input bitstream in a buffer, where *n* is the size of the buffer. When a portion of the bitstream about to be written into the buffer is found to match a segment already contained in the buffer, a reference using the segment match location is sent instead. This reference is made up of three values: a pointer to the location of the first matched data symbol in the buffer, a length value showing how many symbols match, and the next symbol following the matched data in the bitstream. An example of LZ77 encoding is shown in Figure 9 below. A matching segment, “A0014”, begins at position 8 in the buffer. This match has a length of 5, and the symbol following is an “F”. The corresponding codeword sent to the configuration file is “8, 5, F”.

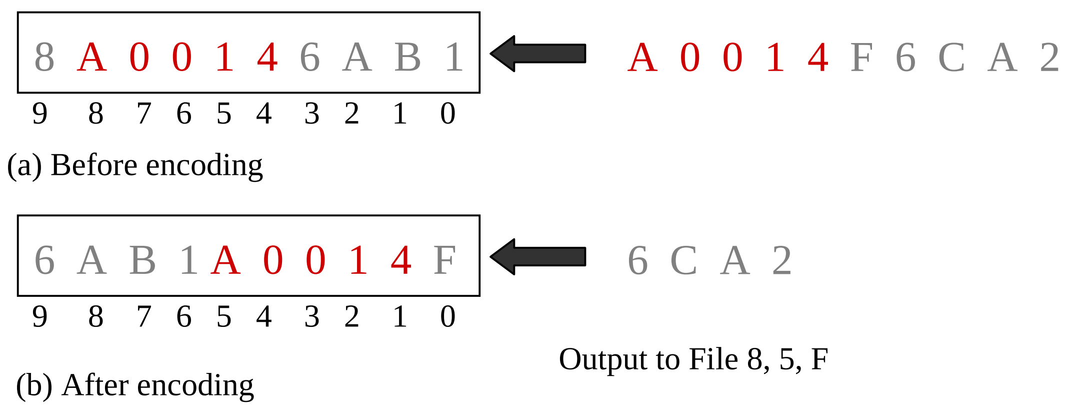


Figure 9: Example of LZ77 buffer. The matching pattern “A0014” is found and the code word “85F” is written to the compressed file.

For a finite bitstream LZ77 is not guaranteed to be efficient. For the case where no matching is found, rather than outputting a symbol, the algorithm will produce a 3-field codeword. In this situation, it is possible for the compressed file to actually become larger than the original file and the compression ratio will worsen. A variation of LZ77 that will improve upon this and help the compression ratio is LZSS. LZSS institutes limitations on what can or cannot be encoded. The length variable must be larger than a predetermined minimum. If the match is found and the length is smaller than the minimum, the current symbol is written directly to the file and the match is not encoded. However, when the length of the match is larger than or equal to the minimum, the codeword containing the index pointer of the first symbol in the match and length of the match is written to the file. The codeword would be smaller than writing the symbols directly and the codeword will be written to the file. In addition, LZSS will need to add a flag bit to indicate whether the bits are a compressed LZ-packet or a symbol. This could cause a file to become slightly larger and worsen the compression ratio.

As previously mentioned, LZSS is able to fully utilize the intra-frame regularities. However in current devices, the FDR buffer can only contain a single frame of configuration data at a time. Therefore, using it as the LZ-buffer will not fully take advantage of naturally occurring inter-frame regularities. A solution proposed in previous research is to extend the FDR buffer. An overview of this is shown in Figure 10 below. The lower portion of the buffer works in much the same was as the current FDR buffer built onto Virtex FPGAs. The upper portion is solely for decompression. As data is written to the device, it is simply shifted upwards into the upper section of the buffer. The next portion of the configuration data is shifted into the buffer and is transferred to the specified frame once the lower section is filled with the new frame data.

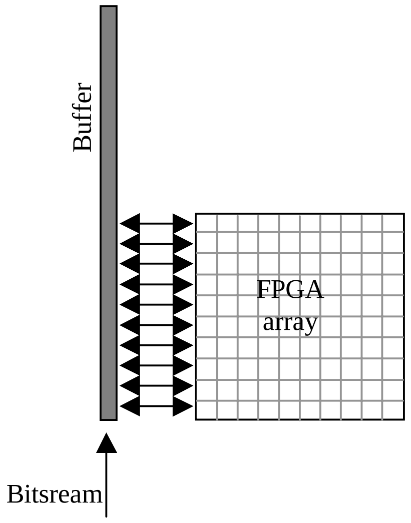


Figure 10: Extended FDR buffer to aid LZ compression.

LZ compression works extremely well as a compression algorithm for cases containing regularities in a bitstream. Finding these regularities in a given configuration bitstream will optimize compression. As previously mentioned, LZ works very well at utilizing naturally occurring regularities in the FDR buffer, however as discussed in earlier sections, LZ requires a large buffer to do this. The hardware costs need to be minimized and the FDR cannot grow without bounds. Therefore, the size the FDR is expanded to must be limited. At the same time, methods in prior research were investigated to exploit the configuration bitstream to aid in LZ compression [Li01]. To take advantage of some regularities, the bitstream can be externally reordered such that large redundancies between frames will occur sequentially in the bitstream. This should help the compression ratio. As this is external to the architecture built for this project.

Symbol Size Factors

Current Virtex devices load entire frames of data at a time. Due to similarity of the resources available on the device, there is typically some regularity between the many frames, i.e. inter-frame regularity [Li01]. Inter-frame regularities appear in circuits containing similar configurations among rows. LZ utilizes regularities in a bitstream, using recently loaded data as a fixed sized dictionary for later writes. Therefore, if the frames in the configuration bitstream can be reordered such that those with similarities would be loaded consecutively, better compression should be achieved. There is some coding overhead incurred by moving the frames; however, this is taken care of external to the device and will not directly impact the hardware costs.

Reordering the frames in the bitstream to utilize the inter-frame regularities does not guarantee the compressed file will be smaller. On current Virtex devices, one frame of data is loaded at a time. To exploit this natural regularity, the FDR buffer will be modified to extend past the length of one frame. Previous research found that extending the buffer length improved the compression ratio [Li01].

Similar to regularity among frames, there usually is some regularity within a single frame, i.e. intra-frame regularity [Li01]. For Lemple-Ziv compression, the shift-based design of the FDR buffer naturally takes care of this. At the same time, the size of the FDR buffer as well as the size of the symbol will utilize this. The larger the buffer, the smaller the compressed file. This is another reason the FDR buffer is extended past the length of a single frame.

As mentioned, on Virtex devices the configuration bitstream is made up of 32-bit words. However, the 32-bit words are made up of 18-bit words with extra padding bits used to make up the difference. As shown in Figure 7 above, each CLB is segmented into 18-bit words, not 32-bits. In order to preserve the regularities, the 32-bit configuration bitstream will be broken into the original 18-bit frame size when it is compressed.

Therefore, if the symbol size was instead chosen to utilize a 32-bit word, i.e. powers of 2, much of the regularity would be missed [Li01].

The physical size of a symbol has a large impact upon the compression ratio. First, for very short symbols, the coding overhead may greatly increase [Li01]. For example, using a LZSS format, each symbol-packet written to the compressed file will increase by 1-bit by the addition of a flag bit. As an illustration, for a 2-bit symbol the packet written to the compressed file is 3-bits long, 33% larger. This can have an obvious impact on the size of the compressed file. On the other hand, very large symbols can also affect the coding overhead. For very large symbols, the possible matches in LZ-compression will shrink. The larger groupings will encompass more of the buffer. This means there are physically less symbols contained in the buffer to match. The buffer would need to be extended much further in order to maintain a useful compression ratio. However, the added hardware costs of a very large buffer far outweigh any possible gain.

As it turns out, the decompression is done in hardware and the potential hardware cost was considered [Li01], [Hauck99]. In related research, LZ approaches were found to work best with a 6-bit symbol size and a FDR buffer at least the length of two frames [Li01].

Compressed Bitstream Characteristics

In the LZ decompression engine, the FDR buffer for this design is 3072-bits. Each symbol is 6-bits long by itself. This is based upon previous research [Li01]. For a six-bit symbol size, the FDR buffer is actually a 512-symbol shifter.

The bitstream is made up of a mixture of compressed Lemple-Ziv packets and uncompressed symbols. As previously discussed, the compressed Lemple-Ziv data packets are made up of a flag, a pointer and a length value. Analogously, each noncompressed symbol will be made up of a flag and a symbol. In order to address all 512 locations, the pointer is 9-bits long. The length variable is also 9-bits long based on prior work. If the length and pointer are at least the same size, it aids in compression [Hauck99a]. The flag indicating what type of packet is being sent into the decompression engine is a single bit long. This translates the total length 19-bits for a LZ packet and 7-bits for a symbol-packet.

The flag is used to indicate what type of packet is being sent into the decompression engine. For an uncompressed symbol, the flag is set to true. When this is read by the hardware, the 6-bit symbol following the flag will be read directly into the LZ-shifter. On the other hand, for a compressed LZ-packet, the flag is set to false. When this is read, the 18-bits following the flag, encompassing first the pointer and second the length, are read into the hardware. At this point, data is not read from the compressed input bitstream until the remaining symbols are decompressed and read into the hardware.

Design Overview

The hardware is made up of two major components. The first component is the LZ-shifter, which is the extended FDR buffer mentioned earlier in this report. The second component is the control logic component. Furthermore, the hardware is designed to work with current technology such that it can be integrated into a current Virtex FPGA device. An overview of the design when built with such an FPGA can be seen in Figure 11 below.

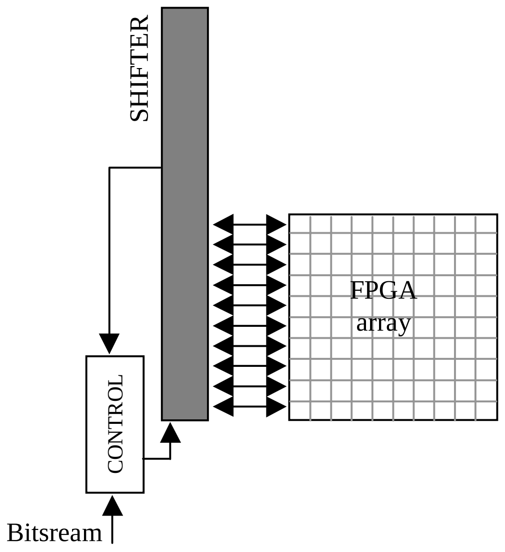


Figure 11: Hardware overview

The control component reads in the compressed configuration bitstream a packet at a time. The bitstream will contain a mixture of compressed LZ-packets and uncompressed symbols. Each packet, regardless to type, will begin with a flag that specifies the format to expect. If the flag is true, the 6-bit symbol is shifted directly onto the buffer. However, if the flag is false, the 18-bit codeword encompassing the pointer and length is sent in and a symbol is copied from the shifter. As mentioned, the shifter will be an extended version of a Virtex FDR buffer. For a LZ-packet, the shifter will select a symbol to copy using a large multiplexer.

A block diagram of the inputs and outputs can be seen in Figure 12 below. The list of inputs to the system include the configuration bitstream where 19-bits could be accessed during any given clock cycle. Other inputs include a preset for the control logic and a hold used to temporarily disable any running processes. The LZ-hardware also has an input coming from the FPGA substrate. The buffer has a readback function that can load into the shifter a frame already programmed on the device (50 18-bit groupings for this design). The last input is a select, controlling readback function just mentioned.

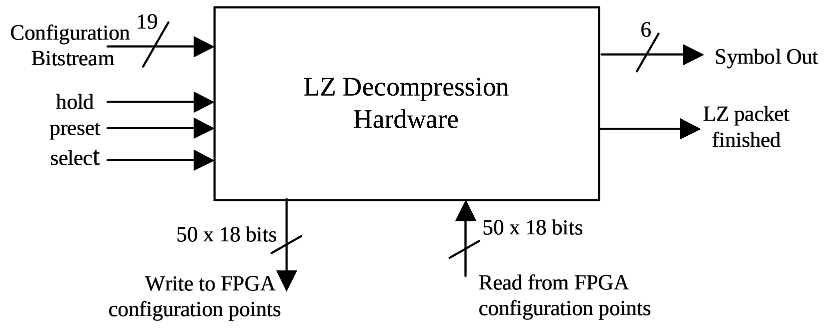

Configuration Bitstream
19
hold
preset
select
LZ Decompression Hardware
6
Symbol Out
LZ packet finished
50 x 18 bits
Write to FPGA configuration points
50 x 18 bits
Read from FPGA configuration points

Figure 12: LZ Decompression Input and Output diagram

In a similar fashion, the outputs of the system include the last symbol shifted out of the shifter. In addition, a control output signal indicates when a LZ-packet has been decompressed. This signal is normally true and is set false while a decompression is being processed. Lastly, after some time, an entire frame will eventually be loaded on the shifter. An output from the buffer is used to load this frame into the main substrate of the FPGA, which configures the device.

The shifter reads in a symbol at a time from the control logic while the decompression hardware is running. The shifter is connected to a large multiplexer which is used during LZ decompression to select a symbol to copy. At the same time, the input of each shift register is multiplexed. The first input is connected to the previous shifter register in order to shift a symbol. The second input is comes from configuration points within the FPGA substrate. This is in order for the FPGA to readback a frame of configuration points onto the shifter. However, only some of the registers are used for the readback, the number equal to the size of a frame. The shifter is extended past the length of a single frame, but only a frame of data will be read back at a time. This can be seen in Figure 11 above. Meanwhile, the output of each shift register is connected to the large multiplexer used for symbol selection during LZ-decompression. Likewise, each shift register is connected to the input of the next shift register in the buffer. At the same time, some of the registers at the start of the shifter are connected to the FPGA substrate. At the front of the shifter, a section the length of a frame is connected to the FPGA substrate.

Hardware Details

As discussed above, the major hardware components necessary for Lemple Ziv decompression encompass a shifter and control. The architecture detail can be seen in Figure 13 below. The symbol shift register is the dynamic dictionary used during Lemple-Ziv decompression. It also is the connection point where configuration points on the FPGA will be written. The shifter is 3072-bits long, which works out to 512 symbols. As a result, Connected to the outputs of each symbol contained in the shifter is a 512 to 1 Multiplexer. This multiplexer uses the pointer value to select a symbol to be copied onto the buffer.

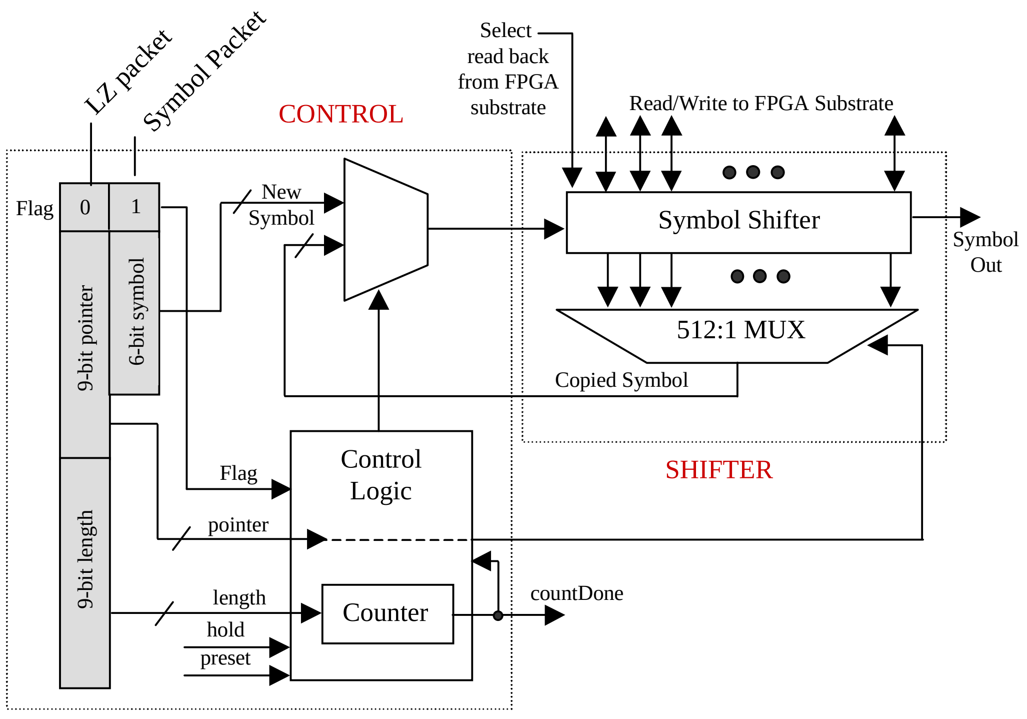


Figure 13: Design Architecture Overview.

The control component reads in the compressed configuration bitstream a packet at a time. Using the flag, the control determines which type of packet is being read in. The control will react to one of three cases. First, the flag would indicate a LZ-compressed codeword is being read in. Second, the flag would indicate that an uncompressed symbol would be shifted in. Lastly, regardless to the flag value, the hold signal would be enabled and nothing will be read in until this is released.

When a LZ-packet is read in by the decompression hardware, a number of things happen. First, the flag sets the “countDone” signal to false. While this is false, no more packets will be read in to the hardware. Meanwhile, the pointer value is fed to the large MUX in order to locate the symbol indexed by the compressed codeword in the shifter. The MUX selects the symbol to be copied back and shifts it onto the input of the symbol shifter. The pointer value will remain the same for the entire duration of the LZ packet decompression. This is stored in a register when the initial LZ-packet is read in by the hardware. During decompression, the buffer shifts a symbol at a time. Consequently, the next sequential matched symbol will conveniently shift to the location of the original indexed match. One by one, each of the matched symbols will be copied onto the buffer. At the same time, the number of symbols copied is determined by the length value from the LZ codeword. The length is fed into a down counter that was built into the control. For each symbol copied the counter decrements by one at the start of each clock cycle. Once the down counter reaches zero it sends a signal (countDone is set true) to the Control indicating that the decompression is complete. At this point, the control logic looks for a new packet in the compressed input data stream.

As mentioned, some of the packets will be uncompressed symbols. Similar to an LZ-packet, a symbol packet begins with a flag indicating that a symbol is to be read. The flag is striped off and the symbol is shifted directly onto the buffer. As this is completed in one clock cycle, the control logic will look for a new packet in the compressed input data stream at the start of the next clock cycle.

The hold symbol is an input used to stop all running processes. No matter what the decompression is doing at the time hold is enabled, the hardware will remain in its current state until the signal is disabled.

The overall Cadence layout for the entire LZ-decompression hardware can be seen in Figure 14 below.

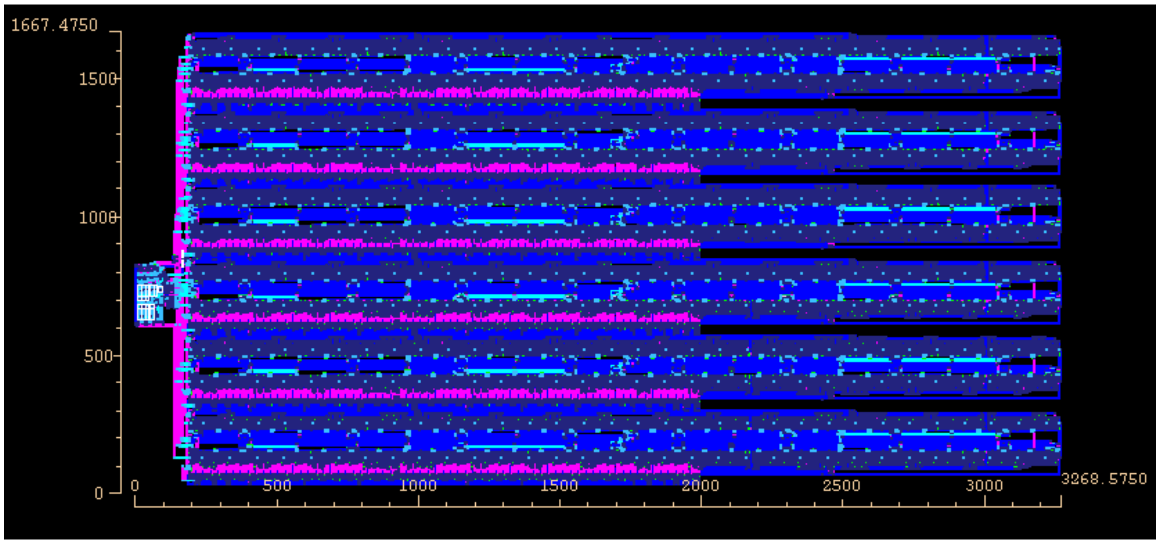


Figure 14: Completed Cadence layout of design.

Shifter Component

The shift register is designed to shift 512 6-bit symbols. As mentioned, this is the extended version of the FDR buffer found in Virtex FPGAs. An overview of the shifter component is shown in Figure 15 below.

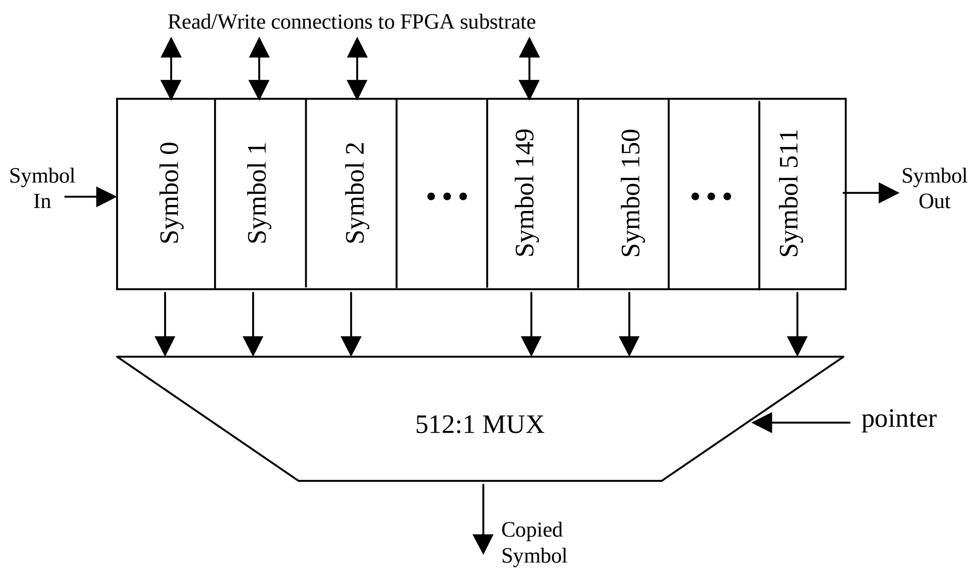


Figure 15: Shifter component overview

While the decompression engine is running, a symbol per clock cycle (6-bits) is shifted towards the output (right-shift in Figure 14). As mentioned, the shifter is extended past the length of a single frame to aid the compression, in other words, a single frame of the configuration bitstream is connected to the FPGA substrate for configuration or readback. Consequently, the input of each shift register is multiplexed to allow the readback capability. A select signal controls whether the buffer is reading a symbol or a frame from the substrate.

Each shift register is made up of small, fast D-flip-flops, the schematic of which is shown in Figure 16 below. As mentioned, a multiplexer is located on the input of each flip-flop. Utilizing this, the input can be chosen from the shifter or read directly from the substrate. The output will be connected to the next register in the shifter and the 512 to 1 Multiplexer. The outputs of the first 150 6-bit symbols are connected to the configuration memory located on the target device,

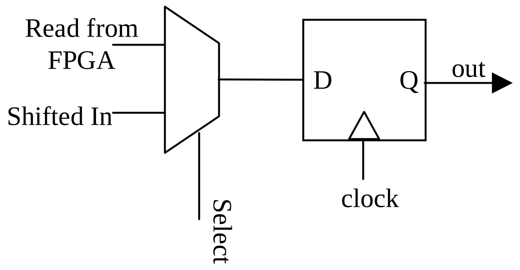


Figure 16: Shifter Mux with D-Flip Flop.

The symbol shifter will dominate the final size of the design due to the shear number of registers needed. Therefore, it was necessary to design the shift register to be as small as possible. As a result, a minimal number of transistors were used. The schematic for can be seen in Figure 17 and the Cadence layout of a shift register can be seen in Figure 18

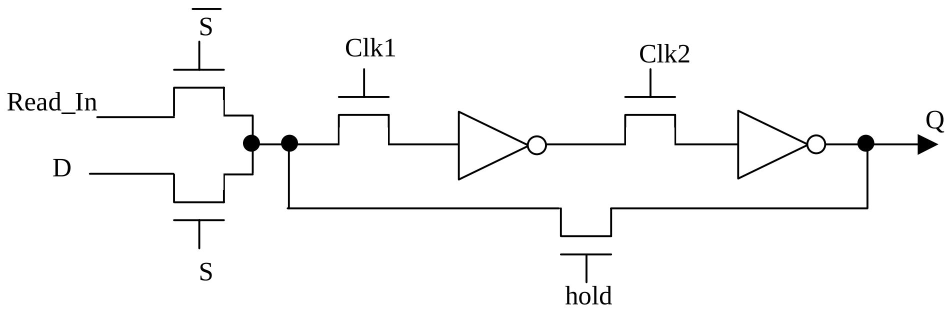


Figure 17: Single shift register schematic

As you can see, the shift register uses as few transistors as possible. Because of the large size of the LZ-buffer, this will help to minimize the final size of the entire buffer.

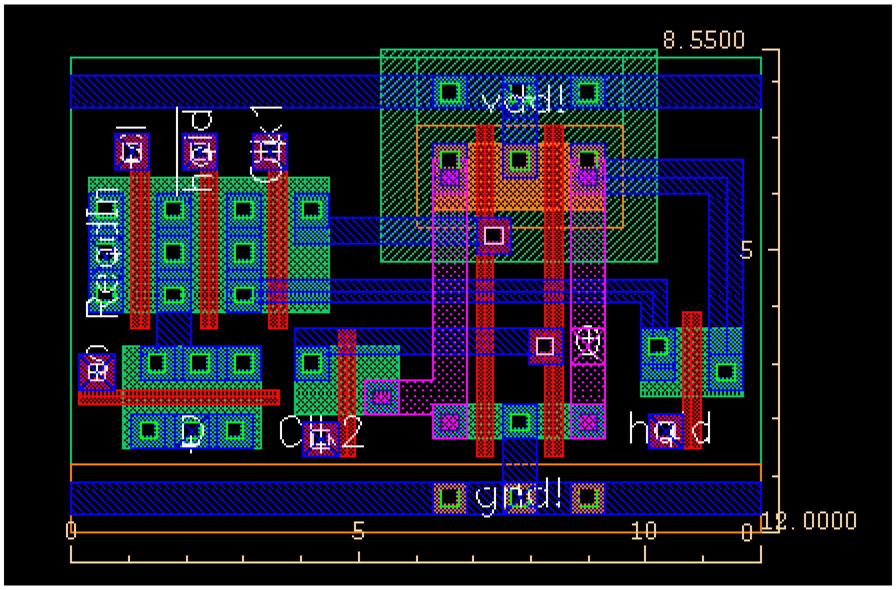


Figure 18: Layout of a single shift register.

The 512 to 1 Multiplexer was built directly into the shift component. The large multiplexer is actually a combination of a number of smaller multiplexers. This is to limit the use of long chains of series transistors. Using Hspice and Cadence extracted netlists, multiple designs using different combinations of 8 to 1, 4 to 1 and 2 to 1 were tested for speed. The fastest design used 2 to 1 MUXs and 4 to 1 MUXs. An overview of the MUX design is shown in Figure 19 below.

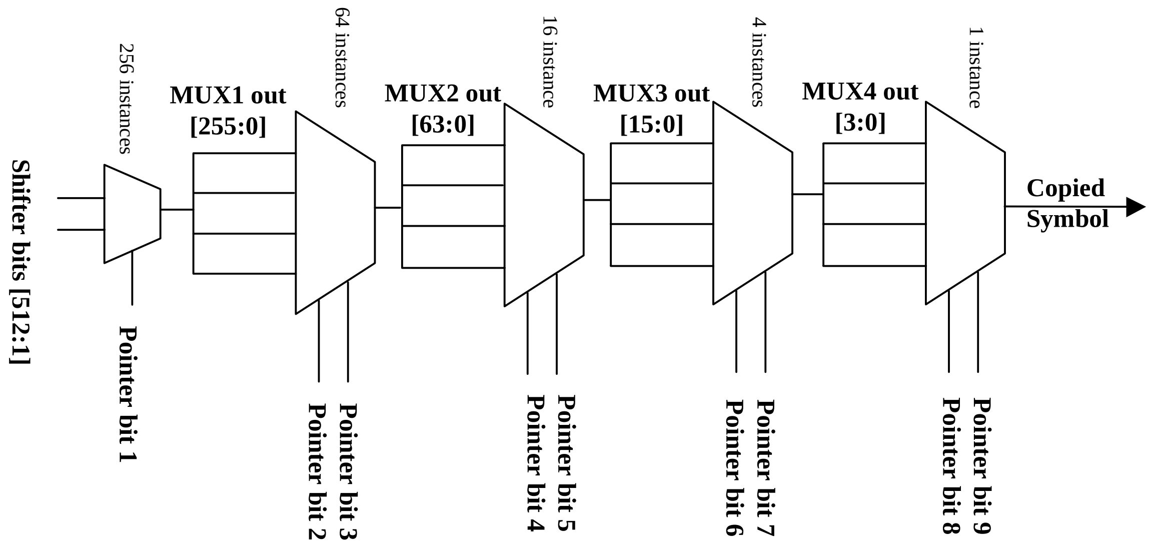


Figure 19: 512 to 1 Multiplexer is built using a combination of smaller 2 to 1 and 4 to 1 multiplexers.

A Cadence layout for the entire shifter component can be seen in Figure 20 below.

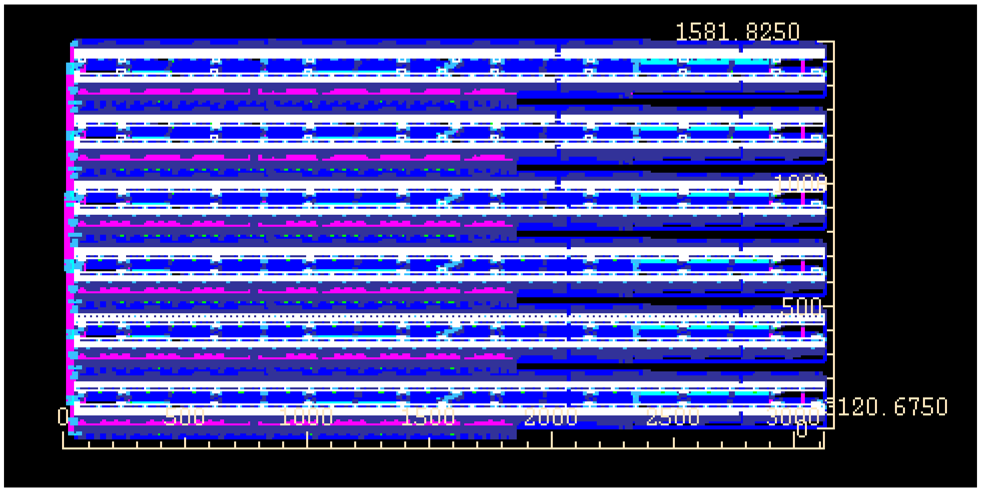


Figure 20: 512:1 MUX and symbol shifter layout.

Control Component

The control component is made up of a down counter, control logic for the pointer, and logic selecting the input to the LZ-buffer. An overview is shown in Figure 21 below. First, the control determines the type of packet being sent in. A 2-bit control signal is generated based upon the flag indicator at the start of a packet and the hold input signal. The feedback value is used by the control MUX to select a copied symbol or a to read a symbol from the input bitstream. Next, a block is included devoted to the pointer control. This logic block stores the pointer value for the duration of a LZ decompression. This block uses the countDone variable on the output of the down counter to determine whether the pointer value should be held. Next, a down counter is included. This utilizes the length value in an LZ codeword to determine how many symbols to copy from the LZ-shifter. The output of the down counter is normally set true, but is set false once a LZ-packet is read in. The down counter decrements by 1 for each symbol copied. This cycle continues until the down counter reaches “0”. Once the counter is finished, a signal (countDone) indicates to the LZ-packet has been decompressed and a new packet is read in at the input.

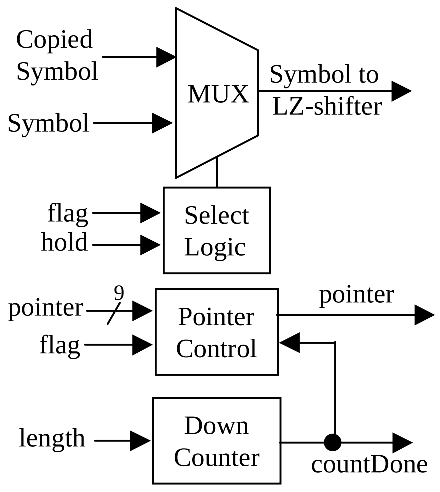


Figure 21: Overview of Control Hardware

The output of the select logic is named “feedback”. This is a 2-bit symbol that is used internally to the control once a packet has been identified as either compressed or uncompressed. The truth table for this signal can be seen in Table 1 below.

Table 1: Truth table for internal feedback signal.

Flag	Hold	Feedback	
X	1	00	HOLD
0	0	10	LZ
1	0	01	SYMBOL

The down counter is predetermined to subtract 1-bit on each clock, or simply a subtractor. One very simple way to do this is to use two’s complement arithmetic. Using two’s complement form, subtraction can be handled as addition by complementing the subtrahend such that A – B = A + (-B). Therefore, using this representation means the subtraction can be accomplished by taking the 2’s complement of the subtrahend and adding it to the minuend. In other words, the number is complemented and added using a conventional adder.

At the same time, the down counter will always subtract 1-bit from the length value. Therefore, a general-purpose adder is not the best solution. Since the subtrahend is a constant value, it is possible to push this into the logic and resultantly using fewer levels of logic in the adder. First, the 2’s complement value is found to be 111111111. Second a truth table was developed for a 1-bit Full-adder and is shown below in Table 2. The constant 1 has been omitted. Furthermore, the Boolean logic and equivalent gates are shown in Figure 22.

Table 2: Truth table for down counter adder.

In	Carry_in	Sum	Carry_out
0	0	1	0
0	1	0	1
1	0	0	1
1	1	1	1

$$Sum = 1 \oplus In \oplus Carry\_in = \overline{In \oplus Carry\_in}$$
$$Carry\_out = 1 \bullet In + 1 \bullet Carry\_in + In \bullet Carry\_in = In + Carry\_in$$

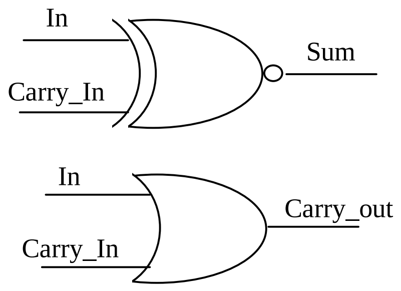


Figure 22: Logic for down counter decrementor.

From the truth table, the logic used for the adder is an OR for the Carry_out value and an XNOR for the Sum. Similarly, the half adder is found to have the Carry_out equal to the In-bit. In addition, inverting the In-bit will result in the sum.

The down counter outputs a signal when the counter reaches “0”. A psuedo NMOS NOR gate is used to determine this. When all 9-bits of the count are “0”, the output of the NOR gate is true and the counter is complete.

The Cadence layout of the control component is shown in Figure 23 below.

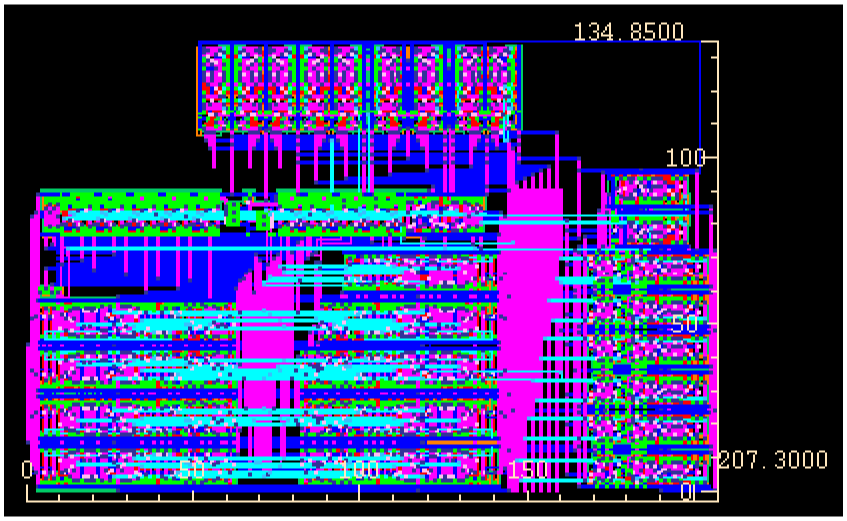


Figure 23: Control component layout.

Hardware Implications

This purpose of this project was to implement in hardware a LZ decompression engine investigated in previous research [Hauck99b], [Li01]. One of the goals was to determine the impact such a design would have upon the size of current devices. In particular, how much larger is the device going to become.

At this time, SRAM based FPGAs can attribute roughly 25% of the total area directly to the memory cells on the chip. Fortunately, the number of distributed RAM cells is known for each device [Virtex00a]. Consequently, using a memory cell as the unit of measure, an approximate area can be found for the FPGA. These calculations were done by first building a SRAM cell and optimizing it in the same manner as the rest of the design. The area measured for this cell is used as the basic unit of measure for subsequent calculations. The area for the different devices as they currently stand was found using this formula: $4 \times (\#\_of\_distributed\_RAM\_cells) \times \lambda$, where λ is the cell area for a SRAM cell. For the process used, the area of an SRAM cell is $52.65\mu m^{2.}$ Using this measurement and the above calculation, an area approximation can be found for each device. This calculation can be seen in Table 3.

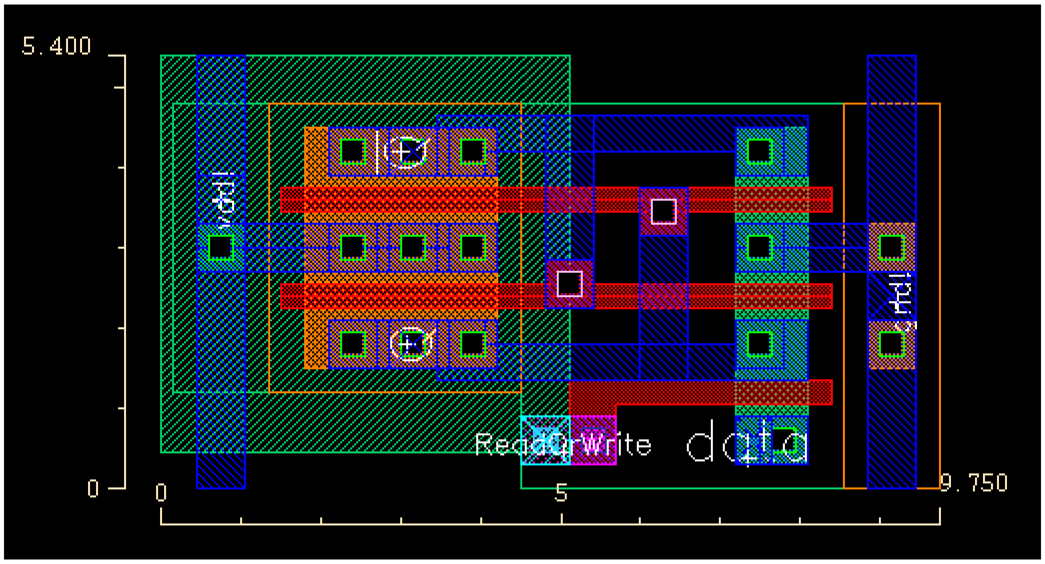


Figure 24: Optimized SRAM cell with an area of 52.65µm^2.

Table 3: Approximated area for each Virtex-E device in the 0.25µ process used for this design.

Device	Distributed RAM bits	Approximate original Chip Area (mm^2)
XCV50E	24576	5.18
XCV100E	38400	8.09
XCV200E	75264	15.9
XCV300E	98304	20.7
XCV400E	153600	32.3
XCV600E	221184	46.6
XCV1000E	393216	82.8
XCV1600E	497664	105
XCV2000E	614400	129
XCV2600E	812544	171
XCV3000E	1038336	219

The area for a single shift register (shown in Figure 17) is 102µm^2. As shown in Figure 24 above, an SRAM cell built using the same process has an area of 52.65µm^2; roughly 50% smaller than a shift register. In a typical device, SRAM cells account for approximately 25% of the overall area of the chip. Thus, the SRAM cell size will be used as a reference from which the overall percentage of the area increase caused by the added hardware in each device can be approximated

The hardware costs are a function of the Virtex device into which it is built. The large contributing factor is the variation in frame size from device to device. Specifically, the physical size of every component in the decompression hardware can be directly linked to the frame size used for a given deice. In particular, the LZ- buffer should be two or more times larger than a frame length to aid in compression [Li01]. For example, this design uses a LZ-buffer over three times larger than a frame. Thus, for the device with the smallest frame size, multiplying this by three does not result in the length of single frame of the largest device. Therefore, the different frame length for each device will cause each to need different buffer lengths. At the same time, the size of the pointer and length variables used is directly linked to the buffer length. In the same way, the size of these variables influences the size of the down counter and control logic.

In Table 4 below, the hardware characteristics of each device is outlined. The number of configuration bits contained in each frame is a function of the number of CLB rows. Each shifter is than extended by two frames, resulting in a final length three times larger. The symbol size is the same for each device, 6-bits. Therefore, the resulting symbol shifter size can be found by dividing the LZ shifter by 6. The symbol shifter must be a power of 2^N and actual length will become larger, increasing the LZ-shifter.

Table 4: Virtex-E Series FPGA Family Members and corresponding characteristics used for area calculations. [Xilinx00b].

Device	CLB Array	CLB Rows	Bits per Frame	Bits per extended LZ Shifter	Minimum Symbols in LZ Shifter	Symbols per LZ Shifter	Address Bits
XCV50E	16×24	16	324	972	162	256	8
XCV100E	20×30	20	396	1188	198	256	8
XCV200E	28×42	28	540	1620	540	512	9
XCV300E	32×48	32	612	1836	306	512	9
XCV400E	40×60	40	756	2268	378	512	9
XCV600E	48×72	48	900	2700	450	512	9
XCV1000E	64×96	64	1188	3564	594	1024	10
XCV1600E	72×108	72	1332	3996	666	1024	10
XCV2000E	80×120	80	1476	4428	738	1024	10
XCV2600E	92×138	92	1692	5076	846	1024	10
XCV3000E	104×156	104	1908	5724	954	1024	10

The hardware built assumed 48 CLB rows (similar toXCV600E device). As mentioned the number of bits per frame in a Virtex device is dependent upon the number of CLB rows per column. As mentioned before, the actual formula for calculating the number of bits is: $18 \times (\#\,\mathrm{CLB\,rows} + 2)$. This means the frame size is 900 bits for this project. Next, the FDR buffer is extended as discussed in earlier sections. The buffer used in this design is 3072-bits long (512 6-bit symbols); well over three times larger than a frame size.

The impact the LZ-shifter would have on the hardware is first the added length. The LZ-buffer is there times the original size with the exact value is device dependent. Typical values are shown in Table 4. The shifter size for a generalized device could be approximated by the following formula: $2 \times (\#\,Bits / Frame) \times \lambda_1$ where λ_1 is the area of the shift register. This was measured to be $102\mu m^2$. The resulting LZ-shifter calculations using the SRAM measurements can be seen in Table 5.

Table 5: Approximated component sizes for shifter

Device	CLB Rows	Approximate shifter Area (mm^2)
XCV50E	16	0.198
XCV100E	20	0.242
XCV200E	28	0. 330
XCV300E	32	0.375
XCV400E	40	0.463
XCV600E	48	0.551
XCV1000E	64	0.727
XCV1600E	72	0.815
XCV2000E	80	0.903
XCV2600E	92	1.04
XCV3000E	104	1.17

In the same fashion, the impact the large MUX would have on the hardware also depends upon the buffer size, more specifically, the number of bits needed to be addressed. This is once again a function of the frame size. As mentioned in the hardware overview, the mux is made up of smaller muxes. Therefore, the large MUX area is based upon the areas for a 2 to 1 and 4 to 1 mux, respectively. The area of the MUX has been approximated using this calculation Table 4, below.

Table 6: Area Approximation Calculations for MUX.

MUX	Formula for MUX Area Approximation	Mux Area Calculations
2 to 1	$\lambda_{2\text{ to }1}$	136.8 μm^2
4 to 1	$\lambda_{4\text{ to }1}$	388 μm^2
8 to 1	$4\lambda_{2\text{ to }1} + \lambda_{4\text{ to }1}$	935 μm^2
16 to 1	$5\lambda_{4\text{ to }1}$	1940 μm^2
32 to 1	$16\lambda_{2\text{ to }1} + 5\lambda_{4\text{ to }1}$	4130 μm^2
64 to 1	$21\lambda_{4\text{ to }1}$	8150 μm^2
128 to 1	$64\lambda_{2\text{ to }1} + 21\lambda_{4\text{ to }1}$	0.00169 mm^2
265 to 1	$85\lambda_{4\text{ to }1}$	0.0033 mm^2
512 to 1	$256\lambda_{2\text{ to }1} + 85\lambda_{4\text{ to }1}$	0.0068 mm^2
1024 to 1	$341\lambda_{4\text{ to }1}$	0.0132 mm^2

Lastly, the impact caused by the control is much less substantial than the LZ-shifter and large MUX. This can be seen in Figure 14 showing the final layout of the entire design. However, it is still important to discuss the factors this component would contribute to the overall area. The size of the control is influenced by the pointer and length values.

The overall area would be approximated in the following way: $(\#Bits/PTR)\times\lambda_4+(\#Bits/LTH)\times\lambda_5+C$. The constant, C, would not change unless the symbol size changed. For example, regardless to device, the feedback bits will be two bits and this are is constant. The constant cost was found to be 0.00598 mm^2. Similarly, λ_4 and λ_5 were also based upon the design built. The area was found by measuring the control sections that utilize first the pointer and second the length and calculating the cost per bit, The cost per pointer bit (λ_4) was found to be 0.00438mm^2. Next, the cost per length bit (λ_5) will be 0.0129mm^2. The calculated impact upon each device can be seen in Table 7 below.

Table 7: Control Area Approximations

Device	**Control Area Approximaton (mm2)**
XCV50E	0.057333
XCV100E	0.057333
XCV200E	0.057333
XCV300E	0.063751
XCV400E	0.063751
XCV600E	0.063751
XCV1000E	0.063751
XCV1600E	0.070169
XCV2000E	0.070169
XCV2600E	0.070169
XCV3000E	0.070169

The overall area approximation for the added hardware was found by adding the area impacts for the different components. The overall area approximation can be found in Table 8 below. The approximated area comparison can be found in Table 9 below.

Table 8: Overall area approximations for added hardware.

Device	**Approximate shifter Area (mm^2)**	**Mux Area Calculations (mm^2)**	**Control Area Approximation (mm^2)**	**Combined Area of new hardware (mm^2)**
XCV50E	0.198	0.0033	0.057333	0.258633
XCV100E	0.242	0.0033	0.057333	0.302633
XCV200E	0. 330	0.0068	0.057333	0.394133
XCV300E	0.375	0.0068	0.063751	0.445551
XCV400E	0.463	0.0068	0.063751	0.533551
XCV600E	0.551	0.0068	0.063751	0.621551
XCV1000E	0.727	0.0132	0.063751	0.803951
XCV1600E	0.815	0.0132	0.070169	0.898369
XCV2000E	0.903	0.0132	0.070169	0.986369
XCV2600E	1.04	0.0132	0.070169	1.123369
XCV3000E	1.17	0.0132	0.070169	1.253369

Table 9: Approximated Area Comparison.

Device	**Approximate Area of Original Device (mm^2)**	**Approximate Area of combined Device (mm^2)**	**Percent of Area Added**
XCV50E	5.18	5.438633	4.75 %
XCV100E	8.09	8.392633	3.60 %
XCV200E	15.9	16.29413	2.41 %
XCV300E	20.7	21.14555	2.11 %
XCV400E	32.3	32.83355	1.63 %
XCV600E	46.6	47.22155	1.32 %
XCV1000E	82.8	83.60395	0.96 %
XCV1600E	105	105.8984	0.85 %
XCV2000E	129	129.9864	0.76 %
XCV2600E	171	172.1234	0.65 %
XCV3000E	219	220.2534	0.57 %

From Table 9, it can be seen that the impact this hardware has on the device is very minimal, at most 4.75%. This means the hardware can be added to current Virtex FPGAs with very little cost to the overall area of the chip. At the same time, the size of the LZ-decompression hardware will increase linearly, versus the FPGA increase quadratically. Therefore for larger devices, the impact upon the overall area is less than 1%.

Conclusion

This project took a technique proposed in previous research [Hauck99b], [Li01] that integrated a Lemple-Ziv compression algorithm into hardware for configuration compression on FPGAs. This approach was found to decrease the size of the configuration bitstream needed to program a device and ultimately should speed up configuration time. The goal of this project was to investigate the feasibility of building such a design in hardware as well as the impact the added hardware would have upon an FPGA.

As it turns out, the impact this approach had on the FPGA hardware was minimal. The largest overall increase in area was approximately 4.75%. Furthermore, the larger devices resulted in an even smaller impact on the overall increase in area at approximately 0.5%. This can be attributed to how the Virtex series of FPGA devices increase in size. The number of CLBs in the Virtex-E Family of FPGAs will increase quadratically. However, the LZ-shifter is based upon the frame size, which will increase linearly. Consequently, the extra area needed for the decompression hardware on larger devices will increase slower than the growth of the device. All Virtex-E devices are good candidates for this technique. In particular, the larger devices could benefit simply because the impact to the area is so small. Overall, this configuration decompression hardware is a powerful solution for decreasing the storage and communication costs associated with configuring an FPGA.

Acknowledgements

The author would like to thank fellow graduate student Zhiyuan Li his on research configuration compression. The author would also like to thank her advisor, Scott Hauck, for his assistance in preparing this report. This research is funded in part by DARPA and Xilinx.

References

[Compton99] K. Compton, *Programming Architectures for Run-Time Reconfigurable Systems*, M.S.E.E. Thesis, Northwestern University, Dept. of ECE, December 1999

[Dandalis01] N. Dandalis, V. Prasanna, "Configuration Compression for FPGA-based Embedded Systems", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2001

[Hauck98a] S. Hauck, "Configuration Prefetch for a Single Context Reconfigurable Coprocessors", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays,* pp 65-74, 1998

[Hauck98b] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems", *Proceedings of the IEEE,* Vol. 86, No. 4, pp. 615-639, April 1998

[Hauck99a] S. Hauck, W. Wilson, "Runlength Compression Techniques for FPGA Configurations", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1999.

[Hauck99b] S. Hauck, Z. Li, E Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 18, No. 8, pp 1107-1113, August 1999

[Hwang01] S Hwang, "Unified VLSI Systolic Array Design for LZ Data Compression", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 9, No. 4, August 2001.

[Li01] Z. Li, S. Hauck, "Configuration Compression for Virtex FPGAs", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001

[Ranganathan93] N. Ranganathan, " High Speed VLSI Designs for Lemple-Ziv-Based Data Compression", *IEEE Transactions on Circuits and Systems – II: Analog and Digital Signal Processing,* Vol. 40, No. 2, February 1993

[Xilinx99] Xilinx, *Xilinx 1999 Databook*, San Jose, CA, Xilinx, Inc, 1999

[Xilinx00a] *Virtex Series Configuration Architecture User Guide*, San Jose, CA: Xilinx, Inc, 2000

[Xilinx00b] Xilinx, *Xilinx 2000 Databook*, San Jose, CA, Xilinx, Inc, 2000

Appendix 1: Verilog Model

Header.v

```
// DEFINES
`define BITS        19   // bits in input stream
`define BUFFER      512  // 512 total Symbols
`define LTH         9    // Length bit size
`define PTR         9    // Pointer bit size
`define SYM         6    // Symbol value
`define STATUS_SYM  2'b11     // shift a sym -> flag set 1
`define STATUS_LZ   2'b10     // shift a copied symbol ->
                              // flag set to 0
`define STATUS_HOLD 2'b0 // do nothing!
```

LZ.v

```
// Include header.v with global define values
`include "header.v"

module LempZiv (clk, preset_n, hold, inbits,
                shifter5, // Symbol MSB
                shifter4,
                shifter3,
                shifter2,
                shifter1,
                shifter0, // Symbol LSB
                off_chip);

   input                  clk;           // clk
   input                  preset_n;      // Active low
                                         // preset, sets
                                         // all bits to 0
   input                  hold;          // stop system;
   input [`BITS-1:0]     inbits;         // value shifted in

   output [`SYM-1:0]       off_chip;
   output [`BUFFER:0]      shifter5; // MSB
   output [`BUFFER:0]      shifter4;
   output [`BUFFER:0]      shifter3;
   output [`BUFFER:0]      shifter2;
   output [`BUFFER:0]      shifter1;
   output [`BUFFER:0]      shifter0; // LSB

   // Converter Registers
```

```
reg  [`SYM-1:0]  symbol;        //change to symbol
reg  [1:0]       feedback; // LZ Shift by PTR & LTH,
                           // Shift by SYM, or set HOLD
                           // Set's up for next clock

// Counter Registers
reg         count_done;
reg  [`LTH-1:0]  count;

// Shifter Registers
reg [`SYM-1:0]   off_chip;
// each symbol is 6-bits long -- using 6 1-bit shifters.
reg [`BUFFER:0]     shifter5; // MSB
reg [`BUFFER:0]     shifter4;
reg [`BUFFER:0]     shifter3;
reg [`BUFFER:0]     shifter2;
reg [`BUFFER:0]     shifter1;
reg [`BUFFER:0]     shifter0; // LSB

// Big Mux registers
reg [`PTR-1:0]   ptr;
reg [`SYM-1:0]   copied;        //

always @(posedge clk or negedge preset_n) begin

   // Converter
   if (~preset_n) begin
      symbol <= `BITS'b0;
      feedback <= `STATUS_HOLD;
   end
   // Set feedback to LZ
   else if (inbits[`BITS-1] == 1'b0) begin
      symbol <=  copied;
      feedback <= `STATUS_LZ;
   end
   // Set feedback to SYM
   else if (inbits[`BITS-1] == 1'b1) begin
      symbol <=  inbits[`BITS-2:`BITS-1-`SYM];
      feedback <= `STATUS_SYM;
   end
   // Set feedback to HOLD
   else if (hold == 1'b1) begin
      feedback <= `STATUS_HOLD;
   end // Converter
```

```
      // Down Counter
      if (~preset_n) begin
         count <= `LTH'b0;
         count_done <= 1'b1;
      end
      // Counter should start same clk as fb set to lz
      else if ((feedback == `STATUS_LZ) | ~count_done)
begin
         // need to look at incoming Flag for next clk
         if (count_done) begin
            // look at next possible length value - used to
catch errors
            if (inbits[`BITS-(1+`PTR+1):`BITS-
(1+`PTR+`LTH)] < 1'h2) begin
               count_done <=  1'b1;
               count <= `LTH'b0;
            end
            // next possible length is greater than 0, need
to decompress
            else if (inbits[`BITS-(1+`PTR+1):`BITS-
(1+`PTR+`LTH)] > 1'h0) begin
               count <= inbits[`BITS-(1+`PTR+1):`BITS-
(1+`PTR+`LTH)] - `LTH'b1;
               count_done <= ~(|(inbits[`BITS-
(1+`PTR+1):`BITS-(1+`PTR+`LTH)]-`LTH'b1));
            end
         end
         // count_done will be set true same clk as counter
reaches zero
         else if (~count_done) begin
            count <= count - `LTH'b1;
            count_done <= ~(|(count-`LTH'b1));
         end
      end // Down Counter


      // Symbol Shifter and output buffer //
      if (~preset_n) begin
         shifter5 <= `SYM'b0;
         shifter4 <= `SYM'b0;
         shifter3 <= `SYM'b0;
         shifter2 <= `SYM'b0;
         shifter1 <= `SYM'b0;
         shifter0 <= `SYM'b0;
         // buffer <= `FRAME'b0;
         off_chip <= `SYM'b0;
      end
      else if (feedback != `STATUS_HOLD) begin
         shifter5 <= {shifter5[`BUFFER-2:0], symbol[`SYM-
1]};
         shifter4 <= {shifter4[`BUFFER-2:0], symbol[`SYM-
2]};
         shifter3 <= {shifter3[`BUFFER-2:0], symbol[`SYM-
3]};
         shifter2 <= {shifter2[`BUFFER-2:0], symbol[`SYM-
4]};
         shifter1 <= {shifter1[`BUFFER-2:0], symbol[`SYM-
5]};
         shifter0 <= {shifter0[`BUFFER-2:0], symbol[`SYM-
6]};
      end // Symbol Shifter and output buffer

      // big mux -> takes in address of MSB of symbol to
copy.
      if (~preset_n) begin
         ptr <= `PTR'b0;
         copied <=`SYM'b0;
      end
      else if (feedback==`STATUS_LZ) begin
         copied <= {shifter5[ptr], shifter4[ptr],
shifter3[ptr], shifter2[ptr],  shifter1[ptr],
shifter0[ptr]};
      end
      else if (inbits[`BITS-1] == 1'b0) begin
         if (feedback != `STATUS_LZ) begin
            copied <= {shifter5[inbits[`BITS-2]],
                       shifter4[inbits[`BITS-3]],
                       shifter4[inbits[`BITS-4]],
                       shifter4[inbits[`BITS-5]],
                       shifter4[inbits[`BITS-6]],
                       shifter4[inbits[`BITS-7]]};
            ptr <= inbits[`BITS-2:`BITS-1-`PTR];
         end
      end // BIG MUX

   end
endmodule //
```