LZW SystemC Implementation

Table of Contents

1 Legend

Font	Type
Courier_New 12	Module name
Times New Roman 12 Bold	State Name
Arial 10 Bold	Signal Name

2 Introduction

This document describes the SystemC implementation of the LZW encoder.

LZW is a lossless compression algorithm which has found uses in many data archiving and imagery applications where lossy algorithms like JPEG cannot be used. One of the most famous applications of the LZW algorithm is the file compression program "compress/decompress" in Unix systems. In addition to software implementation the LZW algorithm has also been implemented in hardware for compression of files in various data archiving applications like solid state drives (SSD).

Our implementation of the LZW algorithm has the following features:

- 4KBytes of input/output storage
- 1KBytes of Hash Table Size
- 8Kbytes of Dictionary
- A serial port for receiving the input data from a PC (FPGA Demonstration purposes only)
- Overall system runs at 33MHZ.
- Can process a maximum of 4Kbytes of uncompressed data

A high level block diagram of the overall implementation is given below.

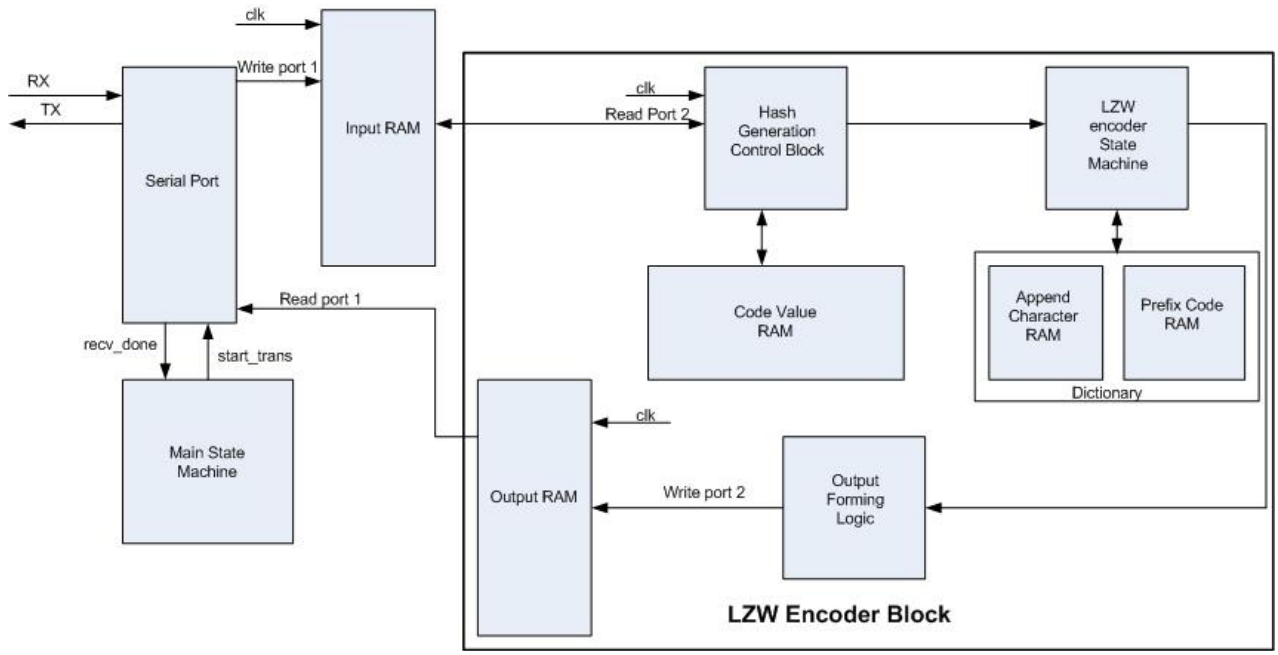


Figure 1 Block Level Diagram of LZW Encoder

The main state machine controls the data movement and control flow for all the total design including the serial port and the LZW encoder block. The input RAM is used to store the data received from the serial port of the PC. The LZW encoder block implements the LZW algorithm, both the control and the data path. The code value RAM is the Hash Table implementation, while the dictionary block implements the LZW dictionary. The output forming logic breaks and merges the 13-bit data output from the LZW data path on correct 8-bit boundaries before writing it to the output RAM. The output RAM is used to store the compressed data which is than transmitted through the serial port to the host PC to be displayed on the terminal.

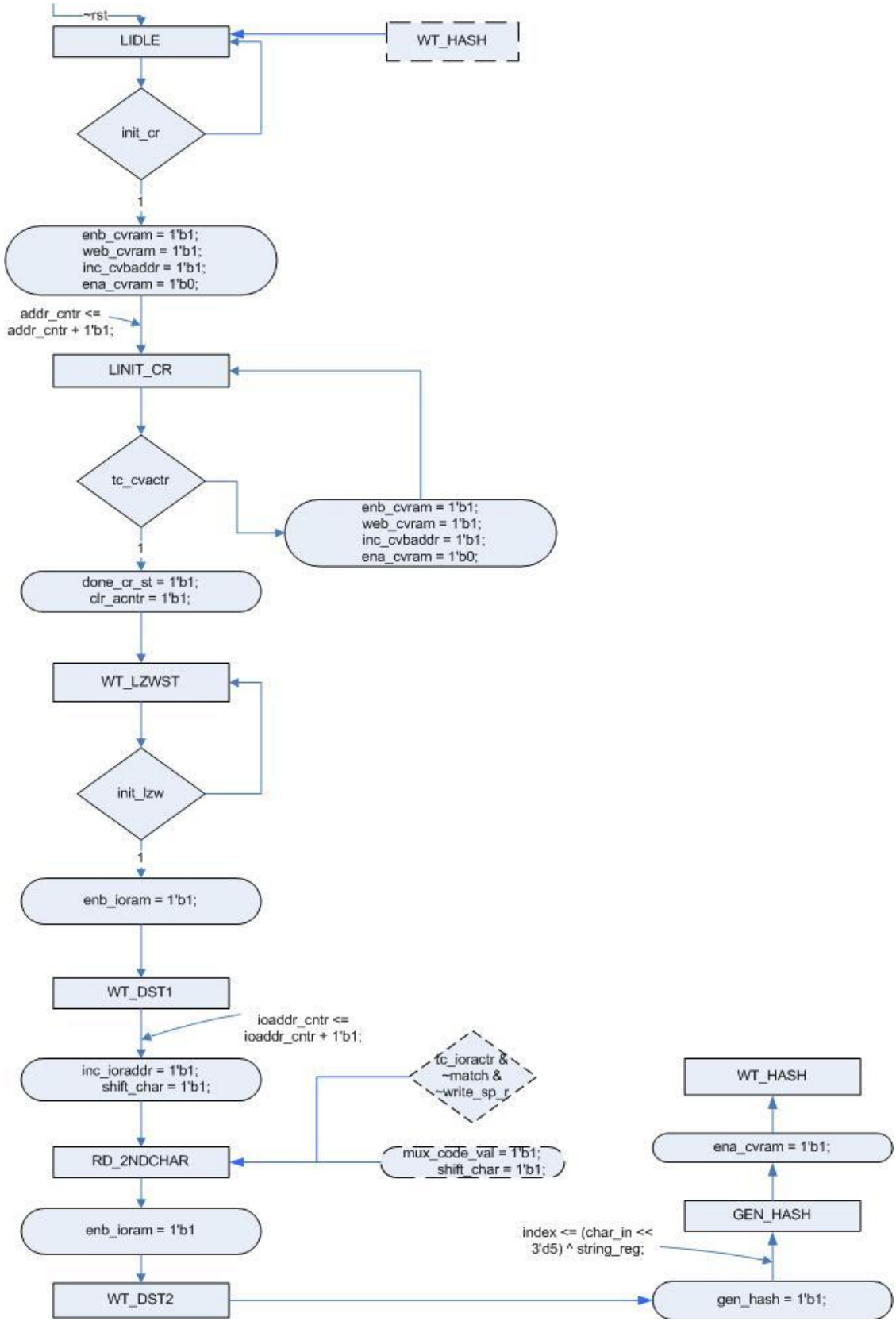
2.1 Input/Output Ports at the top level

Signal Name	Direction	Description
TX	Output	Serial transmit data
SER_RECV_DONE	Output	Serial port data receive done (Tied to LED on the board for debug purposes)
INIT_CR	Output	Initialize code value RAM (Tied to LED on the board for debug purposes)
INIT_LZW	Output	Initialize LZW main state machine (Tied to LED on the board for debug purposes)
DONE_CR	Output	Code value RAM initialization done (Tied to LED on the board for debug purposes)
LZW_DONE	Output	LZW done with processing all data (Tied to LED on the board for debug purposes)
FINAL_DONE	Output	All transactions have finished (Tied to LED

		on the board for debug purposes)
PWR_UP	Output	Power up for debug purpose on FPGA (Tied to LED on the board for debug purposes)
VALID_START_DEB	Output	Valid start on Rx detected for debug (Tied to LED on the board for debug purposes)
RX	Input	Serial receive data
CLK66	Input	System clock, 66MHZ
RST	Input	System reset, active high

3 ASMD Diagram

The following figure shows the ASMD diagram of the design. In the later sections of the report the details of each individual block in the design based on this ASMD is given.


~rst
LIDLE
WT_HASH
init_cr
1
enb_cvram = 1'b1;
web_cvram = 1'b1;
inc_cvbaddr = 1'b1;
ena_cvram = 1'b0;
addr_cntr <=
addr_cntr + 1'b1;
LINIT_CR
tc_cvactr
enb_cvram = 1'b1;
web_cvram = 1'b1;
inc_cvbaddr = 1'b1;
ena_cvram = 1'b0;
1
done_cr_st = 1'b1;
clr_acntr = 1'b1;
WT_LZWST
init_lzw
1
enb_ioram = 1'b1;
WT_DST1
ioaddr_cntr <=
ioaddr_cntr + 1'b1;
inc_ioraddr = 1'b1;
shift_char = 1'b1;
tc_ioractr &
~match &
~write_sp_r
RD_2NDCHAR
mux_code_val = 1'b1;
shift_char = 1'b1;
enb_ioram = 1'b1
WT_DST2
WT_HASH
ena_cvram = 1'b1;
GEN_HASH
index <= (char_in <<
3'd5) ^ string_reg;
gen_hash = 1'b1;

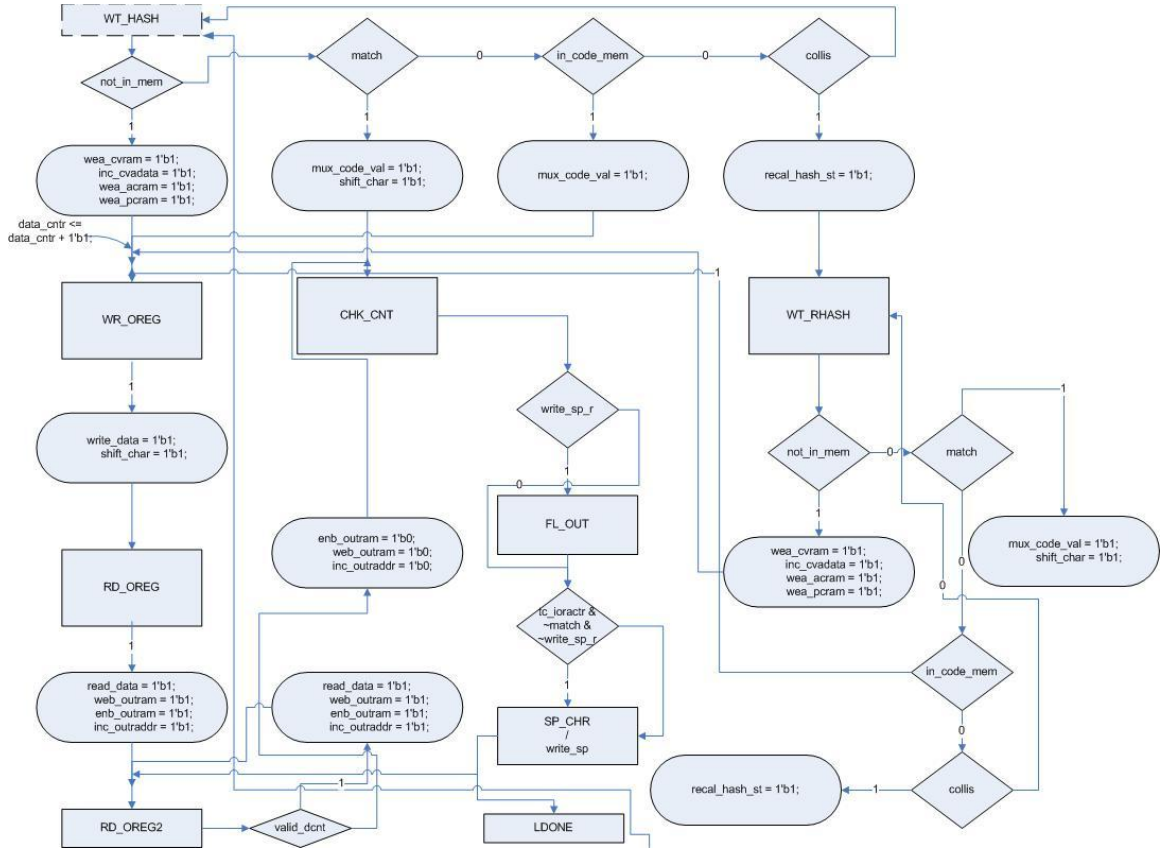


Figure 2 ASMD Diagram

4 Module Hierarchy

Module Name				Short Description
lzw.h				LZW top level
	io_ram.h			Input/Output RAM
	top_ctrl.h			Main State Machine
	ser.h			Serial Port
	lzw_enc.h			LZW Encoder
		hashl.h		Hash
		lzw_ctrl.h		LZW state machine
		outreg.h		Output Registers
		io_ram.h		Input/Output RAM
		code_value_ram.h		Code value RAM
		dictionary_ram.h		Dictionary RAM
			prefix_code_ram.h	Prefix Code RAM
			append_char_ram.h	Append Character RAM

5 Main State Machine

The main state machine is implemented in the `top_cntrl.h` file. The state machine interacts with the serial port, LZW encoder block, as well as the input/output RAM's. The state machine implementation is shown in the following diagram.

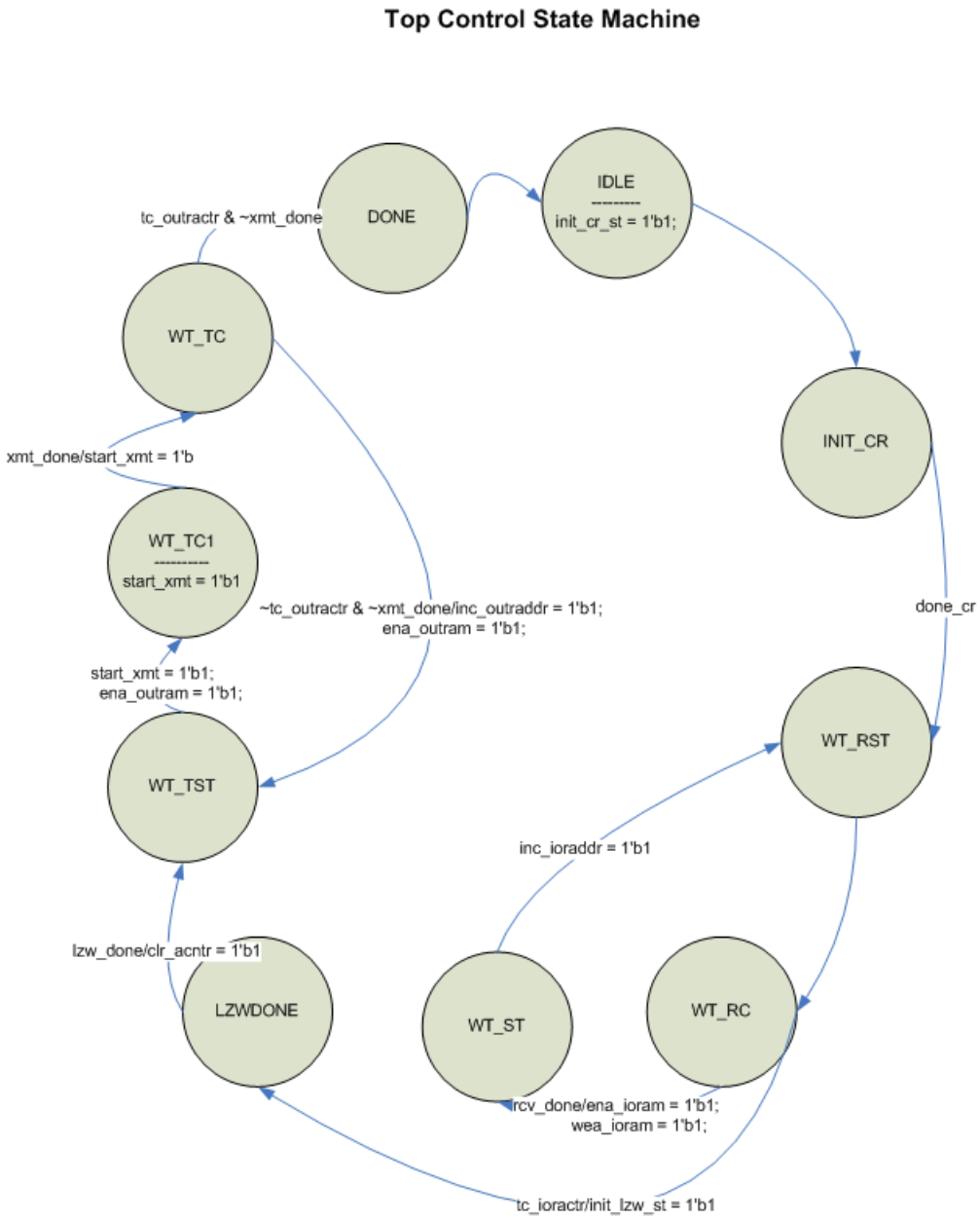


Figure 3 Main state Machine Diagram

After reset the state machine is in the **IDLE** state. The state machine transitions out of the **IDLE** state unconditionally into the **INIT_CR** (initialize code RAM) state. In the **INIT_CR** state the LZW state machine which is implemented in the `lzw_ctrl.h` block is triggered to initialize the entire code RAM block to the FFF value. Once the

LZW control state machine returns the **done_cr** signal indicating the code RAM has been initialized the state machine transitions into the **WT_RST** (wait for receive) state.

The state machine than transitions to the **WT_RC** (wait for receive character) state unconditionally. In the **WT_RC** state the state machine waits for the **rcv_done** signal from the serial port indicating that a byte of data has been received, and writes this byte of data into the `io_ram.h` block, it also checks for **tc_ioractr** (terminal count for the io RAM address counter) to be asserted in this state. The **tc_ioractr** signal indicates that either 4Kbyte of data has been received from the serial port or an end of file character has been received indicating that less than 4Kbyte of input data needs to be compressed. In the **WT_RC** state the control signals to the `io_ram.h` block are issued to write a byte of data. If the total input data has been received than the state machine transitions into the **LZWDONE** (LZW done) state and also triggers the LZW encoder state machine to start work on compressing the data, otherwise it transitions to the **WT_ST** (wait) state. The state machine transitions between the **WT_RST, WT_RC** and **WT_ST** states until the complete input data has been received.

Once the state machine enters the **LZWDONE** (LZW done) state it stays in this state until the **lzw_done** signal is asserted by the LZW encoder state machine (`lzw_ctrl.h`) block indicating that all the input data has been processed, and the compressed data has been written into the output ram (outram instance of the `ioram.h` block). With the **lzw_done** signal the LZW encoder block also informs the main control state machine as to how many data words have been written into, so that it can transmit the correct amount of data to the serial port.

The state machine transitions from the **LZWDONE** state into the **WT_TST** (wait for transmission) state. In this state it triggers the serial port (`serial.h`) to start transmission while reading out a byte of data from the output RAM, by generating the required control signals to the output ram (outram instance of the `ioram.h` block). The state machine transitions to the **WT_TC1** (wait for transmit character 1) state unconditionally and waits for the **xmt_done** signal from the serial port indicating that a byte of data has been transmitted to the serial port.

The state machine than transitions to the **WT_TC** (wait for transmit character) state where it checks if the **tc_outractr** (terminal count for the output RAM address counter) to be asserted in this state. The **tc_ outractr** signal indicates that all the compressed data has been transmitted out of the serial port, and the state machine transitions into the **DONE** (done) state. If the **tc_outractr** signal is not asserted than the state machine transitions to **WT_TST** (wait for transmission) state for transmitting the next byte of compressed data. The state machine transitions between the **WT_TST, WT_TC1** and **WT_TC** states until the complete compressed data has been transmitted.

Once the state machine enters the **DONE** (done) state it will remain in this state until the reset signal is received from the system. This is by design so that before encoding the next input data all the state machine and RAM's are brought back to their initial state.

In addition to the state machine this block also implements two address counters one for the input RAM (addr_cntr) and one for the output RAM (addr_outcntr). The count of these counters is used as the address for the input RAM and output RAM respectively. The incrementing of these counters is controlled by the main state machine, while comparators compare the count with the amount of data to be received or transmitted over the serial port, and when these counts are achieved terminal count signals are issued to the state machine to make transitions to the appropriate states as indicated in the description above.

6 LZW Block

The LZW encoder block consist of a state machine and RAM blocks for implementing the LZW compression algorithm.

6.1 Hash Block

The hash block (`hashl.h`) calculates the address for the dictionary RAM, and the code value RAM based on the current character and the previous string. The hash calculation is a simple logic based on performing a XOR of the current character and the previous character string. The output of this hash function is then used as the address to the code value RAM and to the dictionary RAM.

In addition to this XOR function the hash block also needs to check for validity of the data, a match in the dictionary RAM or a case of collision. All these functions are explained here.

Once the address is generated data is read from the code value RAM, and the dictionary RAM from the specified address. If the data read from the code value RAM is 1FFF it indicates that this address location is invalid, and this character is not in the dictionary RAM as well. In this case **not_in_mem** signal is generated to the LZW state machine, and the current character string is written to the dictionary RAM at the specified address. The data written to the code value RAM will be 256 the first time **not_in_mem** signal is generated, subsequently the data written to code value RAM will be incremented by 1 whenever **not_in_mem** signal is generated. This writing of data to code value RAM is part of the LZW algorithm.

The hash block generates a **match** signal to the LZW state machine whenever the data read out from the code value RAM is not 1FFF (indicating an initialized value) and the data read from the dictionary RAM matches the current character string. This indicates that the current character is already in the dictionary RAM and the character does not need to be added to the dictionary RAM.

The hash block generates a **collis** (collision) signal to the LZW state machine whenever the data read out from the code value RAM is not 1FFF (indicating an initialized value) and the data read from the dictionary RAM does not match the current character string. This indicates that the address calculated by the hash function based on the current

character has already been calculated for a different character already. In this case the state machine will ask for the hash block to perform a recalculation of the hash function. The hash block will than add 12(decimal) to the calculated address, and will present the new address to the code value RAM and the dictionary RAM. Based on this new address the **match, not_in_mem** or **collis** signal will be generated again, and the same procedure followed as described above.

6.2 Output Register

The function of the output register (`outreg.h`) block is to properly align the data on byte boundaries, and write out the data to the output RAM. This is necessary as the string data is 13-bits in width, while the encoded data that has to be stored is 8-bits wide. This is part of the LZW algorithm, as LZW expands the 8-bit ASCII characters to 12 or 13 bits.

The output register increments an internal counter by 13 (decimal) whenever a word is written to it under the control of the LZW state machine, and subtracts 8 (decimal) whenever a word is read from the output register. The output register also implements some shifting logic and merging and unmerging of data to generate a byte of data.

Once LZW encoder has finished its processing of data, and there is less than a byte of data left in the output register than the LZW state machine asks the output register to flush out this data to the output RAM.

6.3 LZW Control State Machine

The LZW state machine is implemented in the `lzw_cntrl.h` file. The state machine interacts with the input/output RAM's, Hash generation block, and the data path of the LZW block. The state machine implementation is shown in the following diagram.

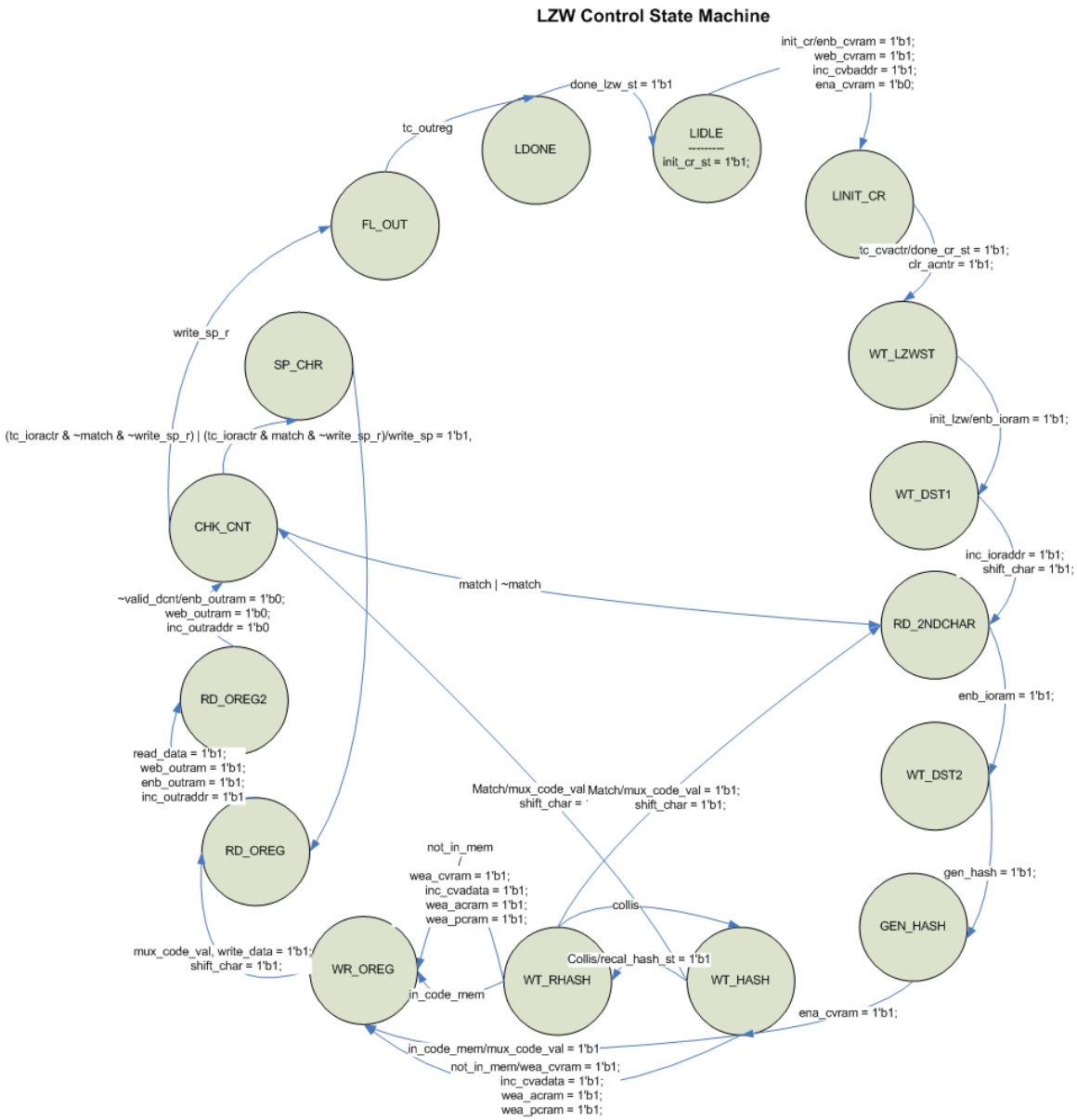


Figure 4 LZW State Machine Diagram

After reset the state machine is in the **LIDLE** state. The state machine transitions out of the **LIDLE** state when it receives the **init_cr** signal from the main state machine into the **LINIT_CR** (LZW initialize code RAM) state. In the **LNIT_CR** state the LZW state machine initializes the entire code RAM block to the FFF value. The state machine transitions out of the **LNIT_CR** state to the **WT_LZWST** (wait for LZW) state when the **tc_cvactr** (terminal count code value RAM address counter) signal. The assertion of this signal indicates that the address counter has reached its maximum value and all the values of the code value RAM have been initialized. The control signals of the code value RAM are also generated from this state.

In the **WT_LZWST** (wait for LZW) state the state machine is waiting for the main state machine to generate the **init_lzw** signal which signals that the LZW encoding function start. Once the **init_lzw** signal is received a byte of data is read from the input RAM (`ioram.h` block). The state machine than transitions to the **WT_DST1** (wait for Data 1) state and is stored in the **string_reg** register in the (`hashl.h`) block while the address t the input is incremented by 1.

The state machine than transitions from the **WT_DST1** (wait for Data 1) state unconditionally to the **RD_2NDCHAR** (Read 2nd character) state. The control signals to the input RAM are generated to read a second word of data from the input RAM, and the state machine transitions to the **WT_DST2** (wait for Data 2) state unconditionally.

From the **WT_DST2** (wait for Data 2) state the state machine unconditionally transitions to the **GEN_HASH** (generate hash) state, while generating the **gen_hash** signal which triggers the (`hashl.h`) block to start performing the hash function on the data read from the input RAM and the character stored in the **string_reg** register. The output of the hash function is used as an address to the code value RAM from which the data is suppose to be read out.

The state machine than waits in the **WT_HASH** (wait for hash result), and depending on the output results of the (`hashl.h`) block transitions into different states. If the (`hashl.h`) block generates the **not_in_mem** signal than it means the character is not in the code RAM (and if it is not in code RAM than it is also not in the dictionary), and the state machine transitions to the **WR_OREG** (write to output register) state, during this transition the state machine also generates the control signal to the dictionary RAM, and writes the character to the dictionary RAM, as well as the output forming logic (`outreg.h`) block. If the **match** signal is generated by the hash block than it means the character already exists in the dictionary and the character is not written to the output forming logic or the dictionary, and the state machine transitions to the **CHK_CNT** (check count) state. If the **collis** (collision) signal is generated by the hash block than it means the hash block has calculated an address which shows that the character already exists in the dictionary, but the character read from the dictionary does not match the input character thus causing a collision, in this case the state machine transitions to the **WT_RHASH** (wait for recalculate hash) state which instructs the hash block to recalculate the hash.

The state machine than waits in the **WT_RHASH** (wait for recalculate hash result), and the same output results of the (`hashl.h`) block are checked in this state and transitions to the same states as in the **WT_HASH** (wait for hash result). If the **collis** (collision) signal is generated by the hash block again, than the state machine transitions to the **WT_HASH** (wait for hash result) to recalculate the hash.

In the **WR_OREG** (write to output register) state the state machine generates the control signals to write the character into the output register if the character does not match in the dictionary, otherwise in case of a match it writes the string from the code value RAM into

the output register. The state machine transitions unconditionally to the **RD_OREG** (read from output register) state.

In the **RD_OREG** (read from output register) state the state machine generates the control signals to read 8-bits of data from the output register and write the 8-bits into the output RAM. The state machine transitions unconditionally to the **RD_OREG2** (read from output register 2) state.

In the **RD_OREG2** (read from output register 2) state the state machine checks to see if the output register (`outreg.h`) block still has more than 8-bits of data or not. If the **valid_dcnt** signal is generated it means that there is 8-bits or more of valid data, and the state machine generates the control signals to read 8-bits of data from the output register and write the 8-bits into the output RAM. The state machine will continue to do this until the **valid_dcnt** signal remains asserted, and will also remain in the current state. Once the **valid_dcnt** signal is de-asserted the state machine transitions to the **CHK_CNT** (check count) state.

The state machine in the **CHK_CNT** (check count) state checks if all the data input RAM has been processed or not. If the data has been all processed than it transitions into the **SP_CHAR** (special character) state, otherwise it transitions to the **RD_2NDCHAR** (Read 2nd character) state to start processing the next character from the input RAM.

In the state machine in the **CHK_CNT** (check count) state checks if all the data input RAM has been processed or not. If the data has been all processed than it transitions into the **SP_CHAR** (special character) state, and a special character is written to the output RAM. This special character is 1FFF, and it basically tells the LZW decoder that the end of encoded data has been reached, and it can stop decoding the input file. The state machine transitions to the **RD_OREG** (read from output register) state to properly write out this13-bits of data to the output register and output RAM. Once, this special character has been totally written out the state machine transitions to the **FL_OUT** (flush out data) state.

The state machine transitions to the **FL_OUT** (flush out data) state from the **CHK_CNT** (check count) state. In this state the state machine is basically asking the output register to flush out any remaining data which is less than 8-bits, and write it to the output RAM. Once the output register asserts the **tc_outreg** (terminal count for output register) signal indicating that it has no more data, the state machine than transitions to the **LDONE** (LZW Done) state.

The state machine transitions unconditionally from the **LDONE** (LZW Done) state back to the **LIDLE** state indicating that all LZW encoder processing has finished while also generating the **lzw_done** signal to the main state machine block.

This block in addition to the state machine also implements the counters which generate the address to the various RAMs which are implemented in the LZW encoder block.

These RAMs are the code value RAM, output RAM as well as the input RAM in the top block.

6.4 LZW Data Path

The LZW data path is shown in the following figure, and consists of the input RAM (an instance of the `ioram.h` block), output RAM (an instance of the `ioram.h` block), the code value RAM (`code_value_ram.h`) block, and the dictionary RAM consisting of the (`prefix_code_ram.h` and `append_char_ram.h`). All these RAM blocks are instanced by using Xilinx Block RAM's. The appropriate number of block RAM's have been instanced to get the correct amount of storage data.

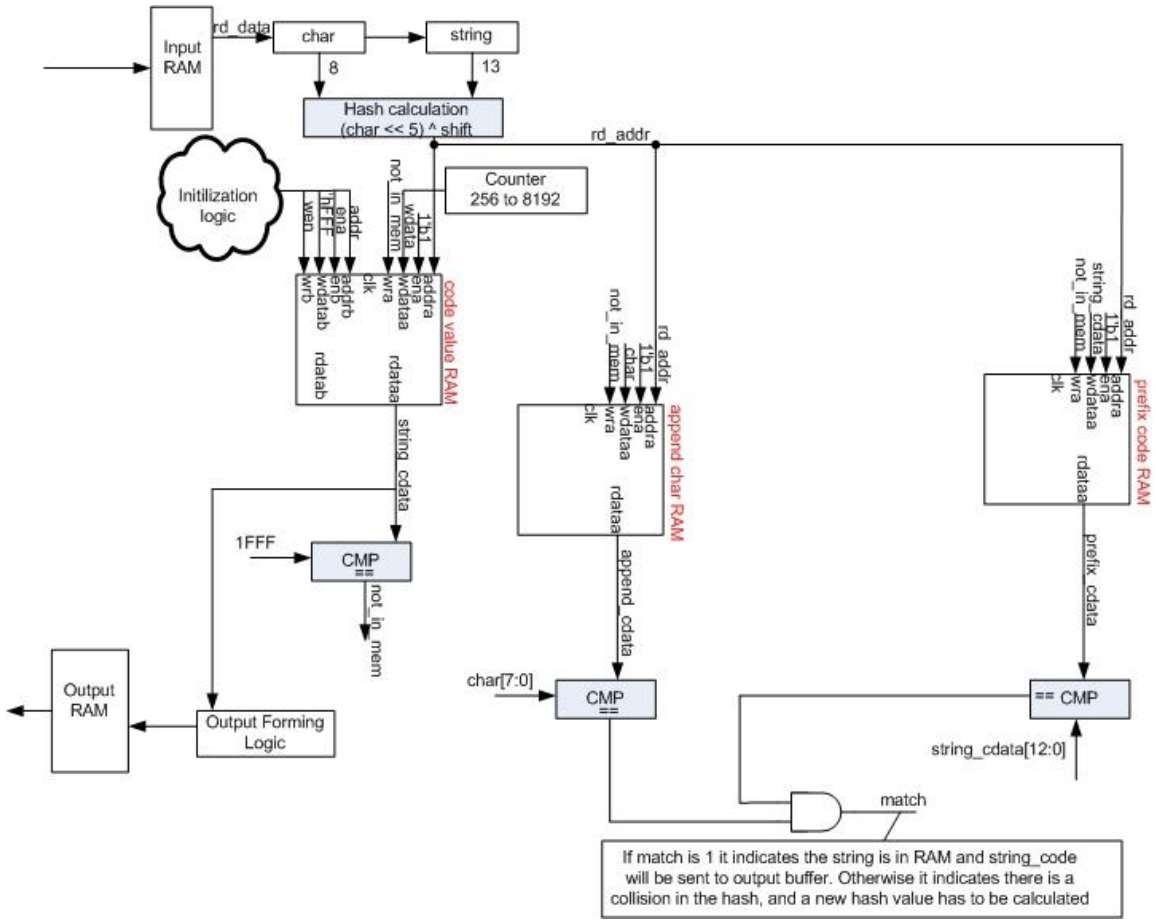


Figure 5 LZW Datapath Diagram

6.4.1 Input/Output RAM

The input/output (`ioram.h`) block is instanced twice, and it stores the uncompressed input data, as well as the compressed data in each instance. The `ioram.h` block can

store 4Kbytes of data, and thus 8 instances of the Xilinx Block RAM are instanced in this module. The lower three bits of the address input signal are used to select to which particular instance of the Block RAM the data will be written, while in case of read the data from the particular RAM is transmitted to the other logic based on the lower 3 address bits.

In the input RAM case the serial port writes to the input RAM on one port, while the LZW encoder block reads the data from the other port. In the output RAM case the LZW encoder writes the data into the RAM from one port while the serial port reads the data from the other port.

Other than Block RAM instances and some muxing and decoding as explained above there is not much logic in this module.

6.4.2 Code Value RAM

The code value RAM (`code_value_ram.h`) block is used to indicate whether the dictionary RAM has valid entries or not and this is the hash table of the LZW algorithm. The `code_value_ram.h` block can store 4Kbytes of data, and thus has 8 instances of the Xilinx Block RAM are instanced in this module. The lower 3 bits of the address input signal are used to select to which particular instance of the Block RAM the data will be written, while in case of read the data from the particular RAM is transmitted to the other logic based on the lower 3 address bits.

After reset the LZW encoder state machine (`lzw_ctrl.h`) block initializes the entire RAM to 1FFF values to indicate that all entries are invalid. Over time as each input character is read from the input RAM and based on the address calculated by the hash block the code RAM block is

Other than Block RAM instances and some muxing and decoding as explained above there is not much logic in this module.

6.4.3 Dictionary RAM

The dictionary RAM consists of 2 memories the append character RAM (`append_char_ram.h`), and the prefix_code RAM (`prefix_code_ram.h`). The dictionary RAM is built as each input character is read from the input RAM, while LZW encoding process is going on. An 8Kbyte of dictionary RAM is implemented by instancing the appropriate number of Block RAM.

The dictionary RAM is only written to and read by the LZW encoder state machine (`lzw_ctrl.h`) block.

7 Serial Port

A serial port is implemented in the design (`ser.h`) with separate buffers for receiving uncompressed data from the PC, and for transmission of compressed data to the PC. The buffers (`ioram.h`) are implemented using dual port Block RAM's available in Xilinx. One side of the dual port RAM is hooked up to the serial port while the other side is hooked up to the LZW encoder block. it for storage of data. The serial port is used to communicate with the PC, from where it receives data that needs to be compressed, and at the end of the compression routine returns the compressed data back to the PC where it can be used to compare against the output of the LZW implemented in a C software program. This comparison is used to show that the hardware implementation of LZW is working properly.

Since the serial port is just for demonstration purposes of the LZW encoder on a FPGA board, therefore it will support only a fixed amount of features:

- Data Format will be: 1 start, 8 data bits, and 1 stop bit.
- Baud rate supported will be 115,200.
- Storage capacity of 4Kbytes of data for input and output RAM's separately
- Separate input and output RAM.

The complete serial port is implemented in the (`ser.h`) file. The sclk_cntr is used to divide the system clock to generate the 16*baud rate clock. In order to keep the interface between the serial port and the LZW synchronous the 16*baud rate is used as a clock enable, instead of a separate clock.

7.1 Data reception

The data reception logic starts whenever a start bit is detected on the receive line of the serial port. The start bit is detected whenever the receive line **sin** makes a transition from high to low. The receive logic only considers this transition to be a valid start bit if low period is detected for 16 clock cycles of the serial port clock. Once a valid start bit is detected each received data bit is loaded into the receive data register **byte_in** (byte in). Each received bit is sampled for 16 serial clocks as per the serial port specifications. Once nine data bits (8 data bits and one stop bit) have been received by the serial port a receive done **rcv_done** signal is generated to the main state machine (`top_ctrl.h`). The main state machine than writes the received 8-bits of data to the (`ioram.h`) block, while receive logic of the serial port goes back to the idle state to wait for the next valid start bit.

7.2 Data transmission

The transmit logic starts to work once the **start_xmt** signal is received from the main control state machine. The transmit logic loads the byte read from the output ram (outram instance of the `ioram.h` block) into the transmit register **bout** (byte out), along with the start and stop bit appended to the byte of data. The serial transmit register shifts out a bit of data whenever the **tc_xmt_cnt** (terminal count of the transmit counter) pulse is

received. The transmit counter has been implemented to make sure that each transmitted bit of data is held for 16 clocks as per specification of a serial port.

7.3 Baud rate generator

The baud rate generator will divide the system clock by the correct divider to generate the required baud rate of 115,200. In this case the correct divider value is 8'd17. The output of this block will be a clock signal which is running at the rate of 115,200*16, and this logic is implemented by the **sclk_cntr**.

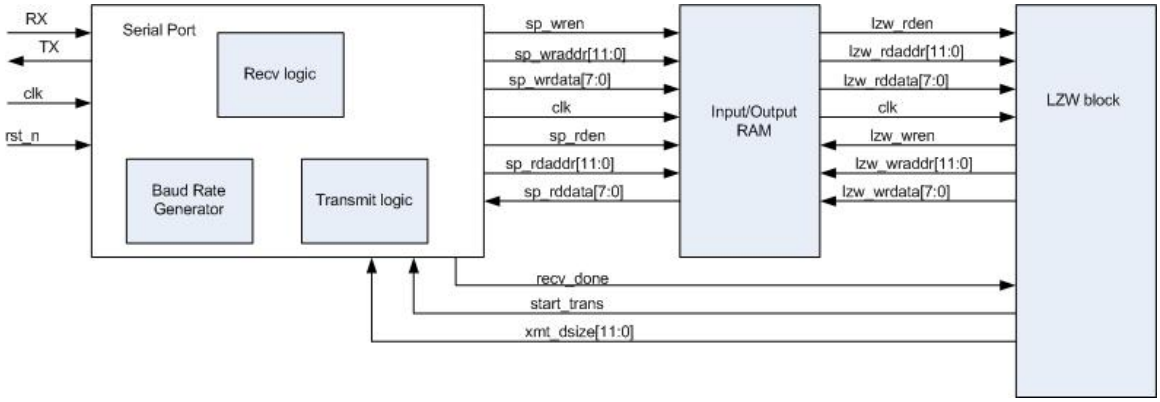


Figure 6 Block Diagram of the Serial Port

8 Test Plan

A test bench and test vectors have been developed to verify the implementation of the LZW RTL. A block diagram of the test bench is shown below.

Two test benches have been developed, one in SystemC and one in Verilog. The SystemC test bench is used to verify the SystemC code. Than the SystemC code is converted to Verilog using an open source program (sc2v). The Verilog output is than verified using the Verilog test bench, in a thorough manner. Once the design has been verified at both levels, it is than tested on the FPGA board.

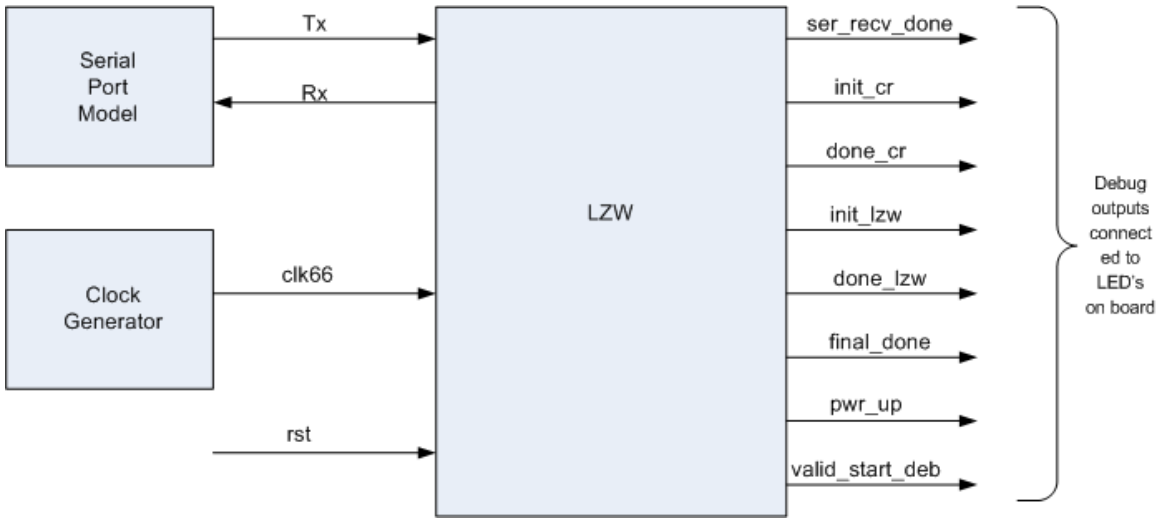


Figure 7 Block Diagram of Test Bench

In order to demonstrate the LZW encoder implementation on the FPGA board a serial port has been added in front of the LZW encoder. The serial port is used to transfer data to the input block from where the LZW encoder picks up the data and encodes it. The encoded data is than stored in the output RAM from where it is then transferred to the PC through the serial port.

In order to verify the design in its entirety a model of the serial port was developed. The serial port model is task based. You can call a task and input the byte of data to it. This byte of data will then be transferred serially to the LZW block. The serial port works at the 115200 bps data rate with an 8-bit data, and one start & stop bit.

In addition to the serial port a clock generator generates the required clock frequency to the LZW block. As there is a 66MHZ clock oscillator on the FPGA board, the clock generator in the test bench also generates the same clock frequency.

In order to verify that the LZW block is functioning properly, a C program implementation of the LZW encoder and decoder has been found from the net. An input file is fed to this C program and the compressed output is generated by the C program in a hex format. The hex format is converted to ASCII and observed. The same input file is also fed to our Verilog test bench after converting the data into ASCII code. The compressed output generated by the LZW RTL is then received by the serial port model which is then stored in a file. The output of the Verilog test bench is than compared with the C program output. This method has been used to debug the LZW RTL. The output of the C program and the LZW RTL must generate the same amount of data bytes, as well as the same ASCII code output.

8.1 Test Vectors

In order to test the LZW block the following three test vectors have been input to SystemC, the Verilog converted RTL, as well as the FPGA board.

8.1.1 Test.v (SystemC testbench.v)

This test vector inputs the following 20 byte of random data to the LZW block, and checks that the repeated characters are compressed.

c1828200013e00000100

The ASCII code for the above 20 characters is input to the LZW block as the test vector in this file. The output of the test bench is then compared with the output of the C program byte by byte.

8.1.2 Test1.v (SystemC testbench1.v)

This test vector inputs the character “c” 4096 times into the LZW block to get the maximum compression.

The ASCII code for the character “c” is input to the LZW block and the input RAM is completed filled with nothing but this character to get the maximum compression. The output of the test bench is then compared with the output of the C program byte by byte.

8.1.3 Test2.v (SystemC testbench2.v)

This test vector inputs the following 52 byte of selected data to the LZW block. The input data has been selected so that no compression should happen, and the output size should be the same as the input size.

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

The ASCII code for the above 52 characters is input to the LZW block as the test vector in this file. The output of the test bench is then compared with the output of the C program byte by byte.