Bit-Mask Based Compression of FPGA Bitstreams

S.Vigneshwaran, S.**Sreekanth**

Abstract— In this paper, bitmask based compression of FPGA bit-streams has been implemented. Reconfiguration system uses bitstream compression to reduce bitstream size and memory requirement. It also improves communication bandwidth and thereby decreases reconfiguration time. The three major contributions of this paper are; i) Efficient bitmask selection technique that can create a large set of matching patterns; ii) Proposes a bitmask based compression using the bitmask and dictionary selection technique that can significantly reduce the memory requirement iii) Efficient combination of bitmask-based compression and G o l o m b coding of repetitive patterns.

Index Terms—Bitmask-based compression, Decompression engine, Golomb coding, Field Programmable Gate Array (FPGA).

I. INTRODUCTION

Field-Programmable Gate Arrays (Fpga) is widely used in reconfigurable systems. The description of the logic circuit is entered using a hardware description language such as VHDL or Verilog. The logic design is drawn using a schematic editor. Logic synthesizer program is used to transform HDL or schematic into netlist. A netlist is a description of various logic gates in the design and their interconnections. The implementation tool is used to map the logic gates and interconnections into FPGA. The configurable logic block (CLB) in the FPGA contains look up tables (LUT's) which performs the logic operations. The mapping tool collects netlist gates into groups that fit into the LUTs and then the place & route tool assigns the gate collections to specific CLBs while opening or closing the switches in the routing matrices to connect the gates together. When the implementation phase is complete, a program extracts the state of the switches in the routing matrices and generates a bitstream where the ones and zeroes correspond to open or closed switches. Since the configuration information for FPGA has to be stored in internal or external memory as bit-streams, the limited memory size, and access bandwidth become the key factors in determining the different functionalities that a system can be configured and how quickly the configuration can be performed. It is quite costly to employ memory with more capacity and access bandwidth, bitstream compression technique lessen the memory constraint by reducing the size of the bitstream. The compressed bit-streams stores more configuration information using the same memory.

Manuscript received on March 2013.

S.Vigneshwaran, VLSI Design, Anna University/ SNS College of Technology, Coimbatore, INDIA.

S.Srikanth, VLSI Design, Anna University/SNS College of technology, Coimbatore, INDIA.

The efficiency of bitstream compression is measured using Compression Ratio (CR). It is defined as the ratio between the compressed bitstream size (CS) and the original bitstream size (OS) (ie CR=CS/OS). A smaller compression ratio implies a better compression technique. Among various compression techniques that has been proposed compression [5] seems to be attractive for bitstream compression, because of its good compression ratio and relatively simple decompression scheme. This approach combines the advantages of previous compression techniques with good compression ratio and those with fast decompression.

II. RELATED WORK

The existing bitstream compression techniques can be classified into two categories based on whether they need special hardware support during decompression. Some approaches require special hardware features to access the configuration memory, like wildcard register, partial reconfiguration or frame readback, which are provided only by certain FPGAs. For example, the wildcard compression scheme is developed for the Xilinx XC6200 series FPGA, which support wildcard registers. Using these registers, the same logic configuration can be written to multiple cells by a single operation. *Pan et al.* [1] used frame reordering and active frame readback to achieve better redundancy. The difference between consecutive frames (difference vector) is encoded using either Huffman-based run length encoding or LZSS-based compression. Such sophisticated encoding schemes can produce excellent compression. However, the decompression overhead in [1], is a major bottleneck in reconfigurable systems.

In contrast, many bitstream compression techniques only access the configuration memory linearly during decompression, and therefore can be applied to virtually all FPGAs. The basic idea behind most of these techniques is to divide the entire bitstream into many small words, then compress them with common algorithms such as Huffman coding, arithmetic coding or dictionary-based compression. LZSS based. For instance, Xilinx [9] introduced a bitstream compression algorithm based on LZ77 which is integrated in the System ACE controller. *Huebner et al* proposed an LZSS-based technique for Xilinx Virtex XCV2000E FPGA. The decompression engine is designed carefully to achieve fast decompression. *Stefan et al.*[11] observed that simpler algorithms like LZSS successfully maintain decompression overhead in an acceptable range but compromises on compression efficiency. On the other hand, compression techniques using complex algorithms can achieve significant compression but incurs considerable hardware overhead during decompression. Unfortunately, the authors did not model the buffering circuitry of the decompression engine in their work. Hence the hardware overhead presented for some

variable-length coding techniques may be inaccurate.

To increase the decompression throughput of complex compression algorithms, parallel decompression can be used. Nikara *et al.* [12] improved the throughput employing speculative parallel decoders. Qin *et al.* [13] introduced a placement technique of compressed bit-streams to enable parallel decompression. However, since the structure of each decoder and buffering circuitry are not changed, the area overhead is also multiplied. Most importantly, this approach does not reduce the speed overhead introduced by the buffering circuitry for VLC bitstream. In contrast, our proposed approach will significantly improve the maximum operating frequency by effectively addressing the buffering circuitry problem.

III. BACKGROUND AND MOTIVATION

In this section, we briefly analyze the decompression hardware complexity of common variable-length compression techniques. This analysis forms the basis of our approach. In the following discussion, we use the term symbol to refer to a sequence of uncompressed bits and code to refer to the compression result (of a symbol) produced by the compression While compression efficiency is straightforward and widely used criteria to evaluate compression techniques, the complexity of decompression hardware determines whether an algorithm with promising compression ratio can be applied to commercial FPGAs. Interestingly, our study shows that the complexity of the decompression algorithm is not the only determining factor of the hardware complexity.

I.DECODE-AWARE BITSTREAM COMPRESSION

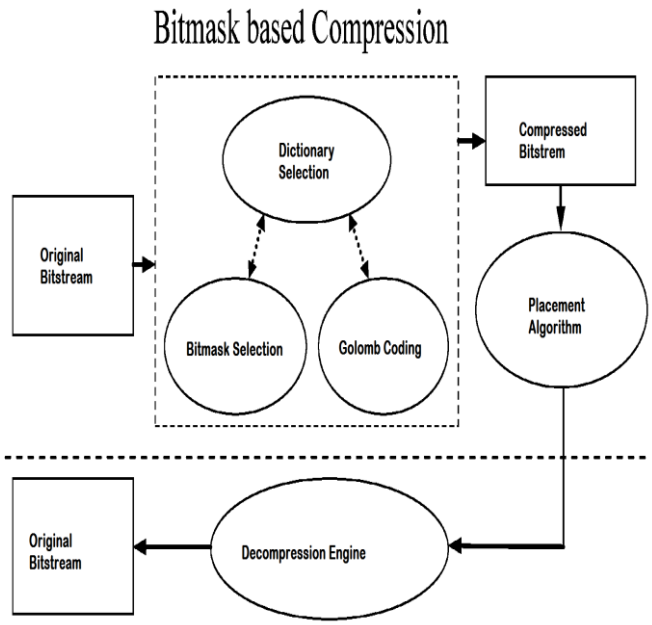

Figure 1. Decode-aware bitstream compression.

Fig.1 [9] shows decode-aware bitstream compression framework. On the compression side, FPGA configuration bitstream is analyzed for selection of profitable dictionary entries and bitmask patterns. The compressed bitstream is then generated using bitmask-based compression and Golomb coding. Next, our decode-aware placement algorithm is employed to place the compressed bitstream in the memory for efficient decompression. During run-time, the compressed bitstream is transmitted from the memory to the decompression engine, and the original configuration bitstream is produced by decompression.

Memory and communication bus are designed in multiple of bytes (8 bits), storing dictionaries or transmitting data other than multiple of byte size is not efficient. Therefore, we restrict the symbol length to be multiples of eight in our current implementation. Since the dictionary for bitstream compression is smaller compared to the size of the bitstream itself, we use to $d=2^i$ to fully utilize the bits for dictionary indexing, where i is the number of indexing bits.

A. Bitmask Selection

Bitmask is a pattern of binary values which is combined with some value using bitwise AND with the result that bits in the value in positions where the mask is zero are also set to zero. A bitmask might also be used to set certain bits using bitwise OR, or to invert them using bitwise exclusive or. This approach tries to incorporate maximum bit changes using mask patterns without adding significant cost (extra bits) such that the CR is improved. Our compression technique also ensures that the decompression efficiency remains the same compared to that of the existing techniques. Fig 3 [5] below represents compression using bitmask selection. The bit-streams which cannot be compressed using dictionary selection are compressed by bitmask selection. The selection of bitmask plays an important role in bitmask-based compression.

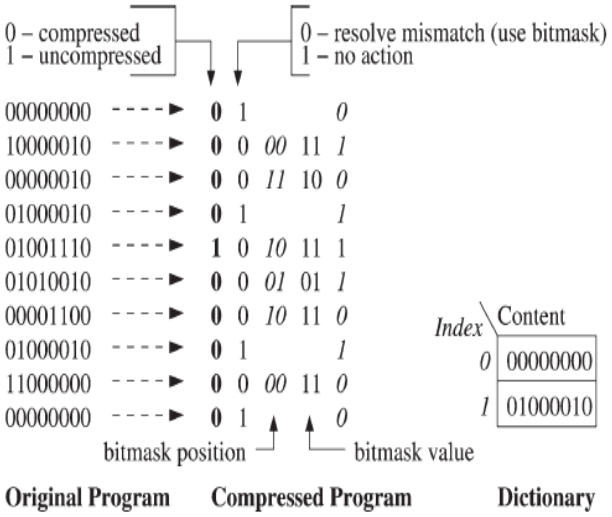

Figure 2. Bitstream compression using bitmask selection approach.

B. Dictionary Selectin

Dictionary-based code-compression techniques provide compression efficiency as well as fast decompression mechanism. The basic idea is to take commonly occurring instruction sequences by using a dictionary. The repeating occurrences are replaced with a code word that points to the index of the dictionary that contains the pattern. The compressed program consists of both code words and uncompressed instructions. Fig.2 [11] shows an example of dictionary based code compression using a simple program binary. The binary consists of ten 8-b patterns, i.e., a total of 80-b. The dictionary has two 8-b entries. The compressed bit-streams requires 62 b, and the dictionary requires 16 b. In this case, the CR is 97.5%. The bit-stream CR for dictionary selection is large therefore it does not yield a better compression technique. Therefore the bit-streams which cannot be compressed using dictionary selection can be compressed by bitmask selection which yields a smaller compression ratio.

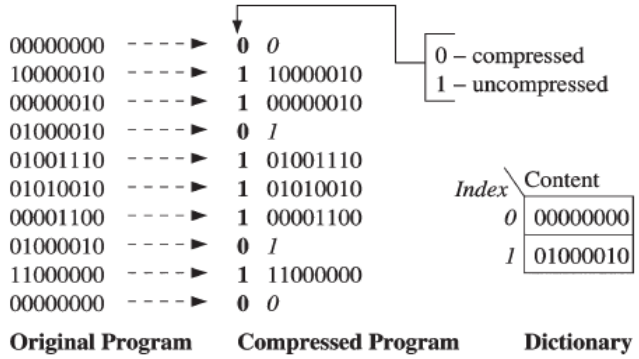

Figure 3. Bitstream compression using dictionary selection

C. Golomb Coding

This section covers the details regarding Golomb Coding basic background information. In Golomb Coding, the group size, m, defines the code structure. Thus, choosing the m parameter decides variable length code structure which will have direct impact on the compression efficiency [10]. Once the parameter m is decided, a table which maps the runs of zeros until the code is ended with a one is created. Determination of the run length is shown as in Figure 3. A run length of multiples of m are grouped into A_k and given the same prefix, which is $(k-1)$ number of one's followed by a zero. A tail is given for each members of the group, which is the binary representation of zero until $(m-1)$. The codeword is then produced by combining the prefix and the tail. An example of the table is in Figure 4.

Data set	01 0000001 0001 000001 001 1					
Subset	01	0000001	0001	000001	001	1
Run-length	1	6	3	5	2	0

Figure 3. Determination of run-length

Using Figure 4, binary strings can be divided into subsets of binary strings and replacing the subsets with the equivalent codeword as shown in Figure 5.

Group	Run-length	Group prefix	Tail	Codeword
A1	0	0	00	000
	1		01	001
	2		10	010
	3		11	011
A2	4	10	00	1000
	5		01	1001
	6		10	1010
	7		11	1011
A3	8	110	00	11000
	9		01	11001
	10		10	11010
	11		11	11011

Figure 4. Golomb coding example with parameter m = 4

Data set	01 0000001 0001 000001 001 1					
Subset	01	0000001	0001	000001	001	1
Encoded	001	1010	011	1001	010	000

Figure 5. Golomb coding example with parameter m = 4

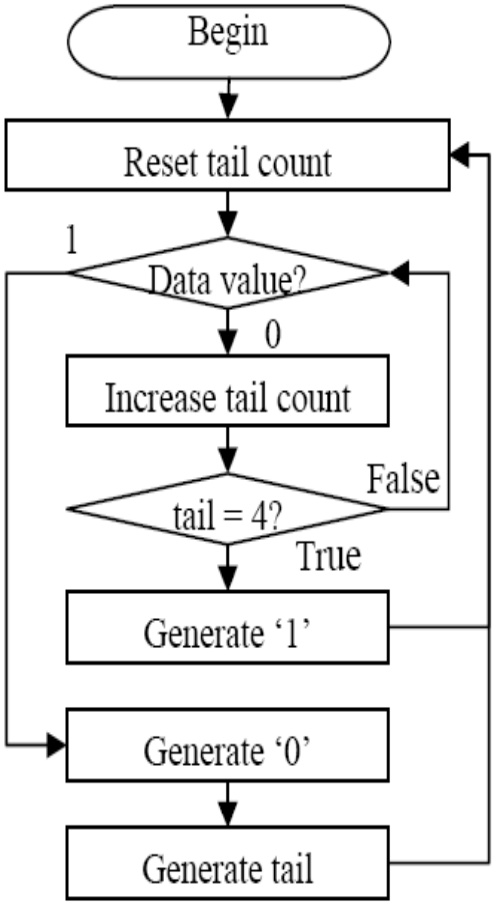

Figure 6. Golomb encoder algorithm (parameter $m = 4$)

The Golomb encoder model can be described in the flow chart as shown in Figure 6. The tail count is controlled by the number of '0's in the input data. If '0's are read, then the tail count will be increased proportionally until it reaches the m parameter where a '1' will be generated as its output data. If the input data is '1', the algorithm will generate a '0' which acts as the divider between the prefix and the tail, and output the current tail count as the tail of the encoded string. The algorithm will then reset the tail count and waits for the next input data. The Golomb decoder model can be described in the flow chart as in Figure 7. The system will firstly detect the value of the prefix, if it is a '1', the system will generate 4 '0's and wait for the next value. If a '0' is detected, the system will acknowledge that the end of prefix has been met and detects the value of the first tail bit. If the value of the first tail bit is '1', the system will generate another 2 '0's, otherwise, wait for the next tail bit. If the last tail bit is '1', another extra '0' will be generated and followed by a '1' which marks the end of a subgroup of the original data. The system will then return to the status of waiting for the next subgroup prefix data.

IV. FPGA IMPLEMENTATION

The Golomb encoder algorithm shown in Figure 6 is not yet a complete design for hardware implementation. Few modifications must be made in order to allow the algorithm to function properly. The encoded binary data will not necessarily be in multiples of bytes, thus padding bits must be placed to make the encoded binary data to be in multiples of 8.

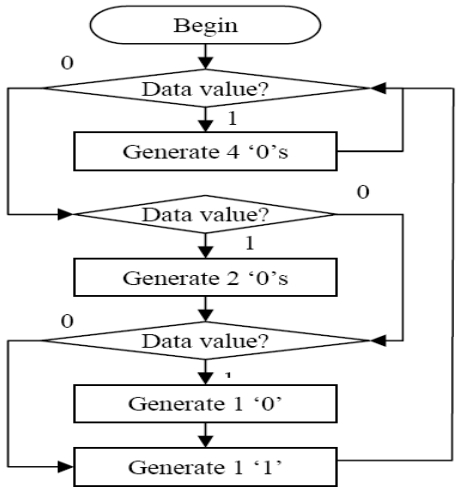

Figure 7. Golomb decoder algorithm (parameter $m = 4$)

Other than that, the previous design of the algorithm was made with the assumption of that the input data string will be terminated with a '1' but however in reality, this will not always be the case because the input data string may also be terminated with a '0'. In order to overcome this problem, the algorithm must be capable of detecting the end of data and if the last bit is a '0', then additional '1' must be added in order to complete the encoding process. However this will also cause another problem of original data being modified due to the additional '1' added, and the decoding process will not work properly, resulting in different set of data being reproduced after the decoding process. In order to solve this matter, header bytes will be added in the beginning of the encoded data which will give the information of the original file size, thus allowing the decoding process to be done correctly by terminating the decoding process after the data size of the original file size was reached during decoding process. The modified Golomb encoder algorithm is shown in Figure 8.

After the system has been reset, the system will move into State 1, generating the original file size as its output, and move into State 2 where the system will wait for input. If the system detects that the total encoded bytes is already reached, the system will proceed to State 5. If the data input is '0', the tail count will increase and return back to State 2.

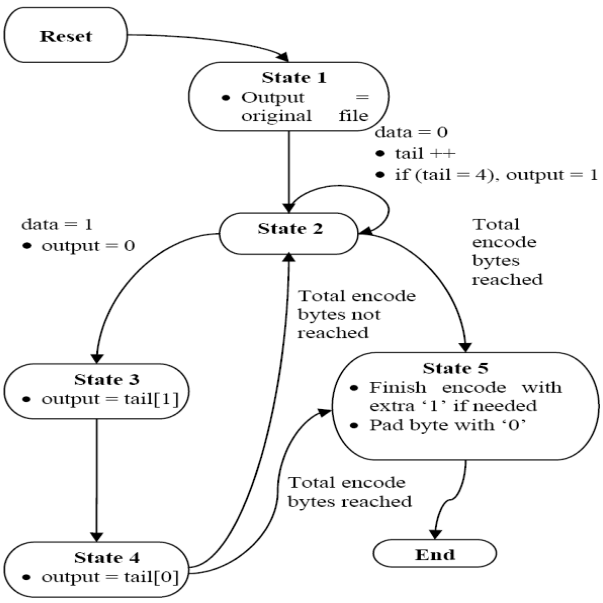

Figure 8. Golomb encoder algorithm on FPGA with Parameter $m = 4$

After the system has been reset, the system will move into State 1, generating the original file size as its output, and move into State 2 where the system will wait for input. If the system detects that the total encoded bytes is already reached, the system will proceed to State 5. If the data input is '0', the tail count will increase and return back to State 2. If the tail count reaches the value 4, the system will generate an output of '1'. On the other hand, if the data input is '1', the system will generate an output of '0' which is the separator between the prefix and the tail and move on to State 3. At State 3, the system will generate the first tail bit and continued by second tail bit at State 4. If the total encoded byte is met, the system will move to State 5, otherwise, back to State 2 where the system will wait for the next input data. At State 5, if the system detects that there is a need to finish the encode with an extra '1' due to the input binary string did not terminate with a '1', then extra encoding will be done with the extra '1' added and then pad the last byte of data with '0' if the encoded data is not in bytes (multiples of 8). The system will then stop handling any process until a Reset signal is given. The Golomb decoder algorithm shown in Figure 7 is not yet a complete design for hardware implementation, like its Golomb Encoder algorithm counterpart. The last byte of the encoded data may be padded with '0's, thus will give wrong decoding output.

However, this had been solved by introducing original file length header into the encoded data as in discussed previously. By knowing the original file size before encoding, the decoder will decode the encoded data until the encoded data reaches the Original file size and terminate the decoding process. The modified Golomb decoder algorithm is as shown in Figure 9.

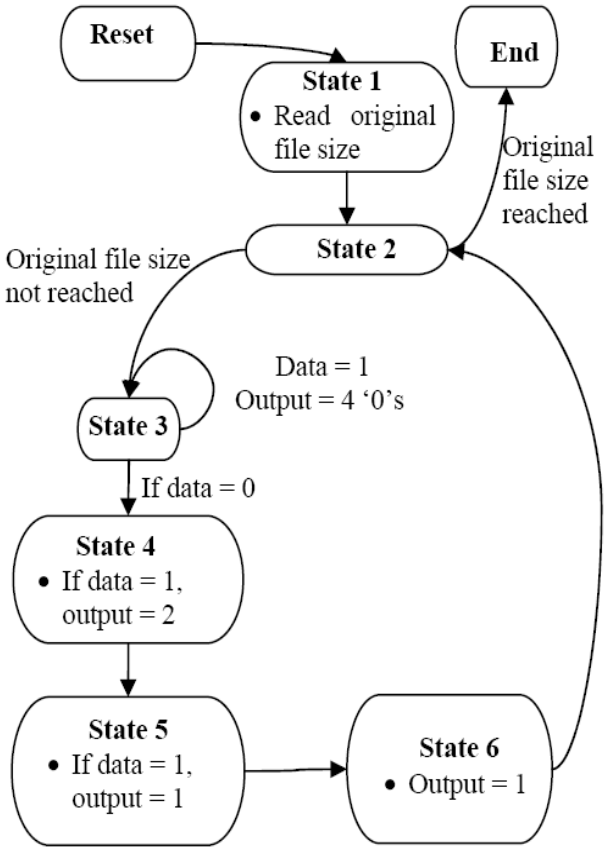

Figure 9. Golomb decoder algorithm on FPGA with Parameter $m = 4$

After the system reset, the system will move into State 1 unconditionally where the system will retrieve the original file size from the encoded data and store it in memory before proceeding to State 2. State 2 will always check whether the total decoded data size has already met the target of the original file size or not. If the target is met, the process will terminate and the system will be idle until Reset signal is given. If the target is not met, the system will move to State 3 where the system will wait for incoming data. If the data received is a '1', the system will generate 4 '0's and return back to State 3. If a '0' is received, the system will acknowledge that the marker between prefix and tail has already been met and proceed to tail decoding process. For the first tail bit at State 4, if the '1' is received, the system will generate 2 '0's as the output and move to State 5 where the system will decode the second tail bit. If a '1' is received, the system will generate 1 '0' as the output and move to State 6 where the system will generate a '1' to mark the end of a subgroup. The system will then return back to State 2 to check whether the total decoded size has already reached the original file size or not and the process cycle continues.

D. Decompression Engine

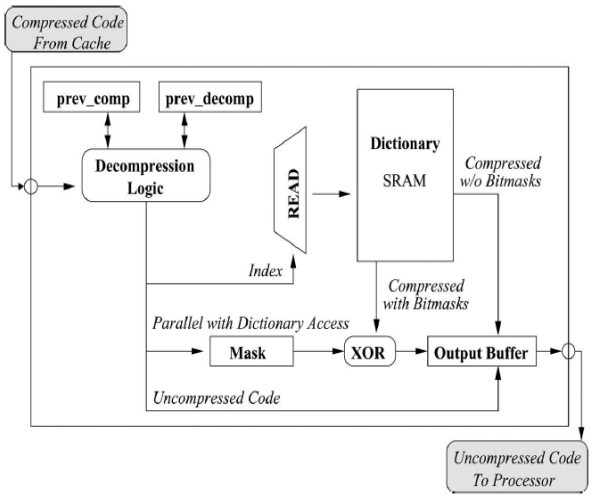

Figure 10. Decompression Engine

The decompression engine is a hardware component used to decode the compressed configuration bitstream and feed the uncompressed bitstream to the configuration unit in FPGAs. A decompression engine usually has two parts: the buffering circuitry is used to buffer and align codes fetched from the memory, while decoders perform decompression operation to generate original symbols .

The design of a decompression engine (DCE), shown in Fig.10 [11] can easily handle bit masks and provide fast decompression. The most important feature of decompression engine is the introduction of XOR gate in addition to the decompression scheme for dictionary based compression. The decompression engine generates a test data length bitmask, which is then XOR ed with the dictionary entry. The test data length bit mask is created by applying the bitmask on the specified position in the encoding. The generation of bit mask is done in parallel with dictionary access, thus reducing additional penalty. The DCE can decode more than one compressed data in one cycle. The decompression engine takes the compressed vector as input. It checks the first bit to see whether the data is compressed. If the first bit is "1" (implies uncompressed), it directly sends the uncompressed data to the output buffer.

On the other hand, if the first bit is a "0", it implies this is a compressed data. Now, there are two possibilities in this scenario. The data may be compressed directly using dictionary entry or may have use bit masks.

V.CONCLUSION

The existing compression algorithms either provide good compression with slow decompression or fast decompression at the cost of compression efficiency. In this paper, we proposed a decoding-aware compression technique that tries to obtain both best possible compression and fast decompression performance. The proposed compression technique analyzes the effect of parameters on compression ratio and chooses the optimal ones automatically. We also exploit Golomb Coding of consecutive repetitive patterns efficiently combined with bitmask-based compression to further improve both compression ratio and decompression efficiency.

REFERENCES

[1] J. H. Pan, T. Mitra, and W. F. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," in Proc. Int. Conf. Comput.-Aided Des., 2004, pp. 766–773.
[2] S. Hauck and W. D. Wilson, "Runlength compression techniques for FPGA configurations," in Proc. IEEE Symp. Field-Program. Custom Comput. Mach., 1999, pp. 286–287.
[3] A. Dandalis and V. K. Prasanna, "Configuration compression for FPGA-based embedded systems," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 13, no. 12, pp. 1394–1398, Dec. 2005.
[4] D. Koch, C. Beckhoff, and J. Teich, "Bitstream decompression for high speed FPGA configuration from slow memories," in Proc. Int. Conf. Field-Program. Technol., 2007, pp. 161–168.
[5] S. Seong and P. Mishra, "Bitmask-based code compression for embedded systems," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 27, no. 4, pp. 673–685, Apr. 2008.
[6] S. Hauck, Z. Li, and E. Schwabe, "Configuration compression for the Xilinx XC6200 FPGA," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 18, no. 8, pp. 1107–1113, Aug. 1999.
[7] D. A. Huffman, "A method for the construction of minimum-redundancy codes," Proc. IRE, vol. 40, no. 9, pp. 1098–1101, 1952.
[8] A. Moffat, R. Neal, and I. H. Witten, "Arithmetic coding revisited," in Proc. Data Compression Conf., 1995, pp. 202–211.
[9] Xiaoke Qin, Chetan Muthry, and Prabhat Mishra, "Decoding Aware Compression of FPGA Bitstreams," in Proc. Data Compression Conf., 2011, pp. 411–419.
[10] S. W. Golomb, "Run Length Encodings," *IEEE Transactions on Information Theory,* vol. 12, pp. 399-401, 1966.
[11] Quartus II development software literature, available at http://www.altera.com/literature/lit-qts.jsp.

Mr.S.Vigneshwaran received his B.E. degree in Instrumentation and Control Engineering from Dr.Mahalingam College of Engineering and Technology, Pollachi in the year 2009, under Anna University, Chennai, and currently Pursuing M.E., degree in VLSI Design at SNS College of Technology, Coimbatore. His area of interest includes VLSI Design Techniques and Electronics.

S.Srikanth received his B.E. degree in electronics and communication engineering from S.N.S College Of Technology, Coimbatore and M.E degree from Sri Ramakrishna Engineering college, Coimbatore. He is Currently working as assistant Professor in S.N.S College of Technology, Coimbatore. His area of interest is VLSI signal processing and Computer Architecture.