J.Thi-Qar Sci. Vol.2 (1) Jan./2010

ISSN 1991- 8690 الترقيم الدولي ١٩٩١ - ٨٦٩٠
website: http://jsci.utq.edu.iq Email: utjsci@utq.edu.iq

CODE DENSITY OPTOMIZATION AND IMPLEMENTATION USING LZW APPROACH BASED ON 32-BIT RISC PROCESSOR

Abdullah A. Hussain Hussein A. Kh. Al-Eidane

Basrah University - College of Education - Computer Science Dept.

ABSTRACT

Considering the worldwide used of mobile to connect internet, and code compression technology that played significant role in saving space and power. Our study is focusing on how many items will be stored in the dictionary and how many bits long for every item, which will determine the optimum dictionary size to the compression process .

Focusing on the dictionary size, one can gain the key for improving the compression ratio on interesting Benchmark. The results will hold on variant size of Benchmarks to investigate the proper dictionary size for variant cases. The average result of compression ratio was 62.34% for a dictionary with 9 bits code and 256 items, where the maximum value of compression ratio was 72.89%.
If we use another dictionary with more than 10 bits, the compression ratio will vary a little.

KEYWORD: Mobile communication, Code Density, Compression Technology, Dictionary Optimization, Decompression unit, hardware implementation.

Introduction:

Compression technology is bursting to the surface impressively, because of worldwide usage for mobile technology in internet connection. On other hand, embedded systems are occupying important market area as well. The common parameter between the two issues, that is, the critical necessity of optimum utilizing for resources in terms of size area, weight, and power. In general, these systems use the RISC general-purpose processor. The low-density of processor instruction and the compression technology are playing contradictory roles.
Code compression is a key technique used for increasing the code density, which leads the reduction of memory size, bus, and power. Moreover, saving memory area helps the designers to offer more functions or tasks especially costumers which requesting for further services in their devices.
This paper, suggest an idea is to store the execution program in a compressed format, that is a compacted offline, and then retrieved to its original format for processor demand. The decompressed procedure implemented by a Decompressor hardware, which installed between cache memory and the untouched 32-Bit processor. The performance degradation of the system accepted in case of no exceeding system tolerance .
Designing the Decompressor engine with the chosen algorithm, that is, lossless LZW algorithm should develop the compression ratio on the system, reducing power, and upholding processor real-time. This lossless LZW algorithm is choose [1] among many algorithms [2,3,4] because of its simplicity; high-speed in compressing, and decompressing that support our investigate and implementation goal. The investigation concentrates on the proper dictionary size with different size Benchmark to improve compression ratio on the system, which illustrated in Equation 1 below.

$$\text{Compression ratio} = \frac{\text{Compressed size} + \text{Dictionary size}}{\text{Original size}} \qquad (1)$$

Related Works:

In 1992, Wolf and Chanian [5] developed the compressed code RISC processor using the Huffman encoding as method and a cache Decompressor memory approach. They have achieved compression ratio 73% with ignoring effected in performance.

Game and Booker 1998 [6], they have used similar technique and algorithm for previous study with the PowerPC processor instruction, which instructions are compressed in 64-byte blocks. The blocks decompressed when loaded to the cache memory. There are two look-up tables; each of these has the size of two Kbytes, which used for holding token expansions. These tables used for the two parts of the instruction; first is for high 16 bits and the second one for the lower 16 bits. They achieved a compression ratio between 0.60 and 0.65.

Lekatsas et al 2001[8], they have used a dictionary-based method with Decompressor engine for decompressing one instruction per cycle utilizing processor name Xtensa 1040, he reported an average 25% speed-up and 65% compression ratio.

Lekatsas et al 2002, [7] they have classified instructions into for types; instructions with immediate, branches, fast dictionary, and uncompressed each of which required a particular compression method.

Benini et al.2002, [10] has applied a dictionary-based method in the DLX processor to build the dictionary from the most frequently executed instructions, their results showed an average of 72% compression ratio and 28% speed-up.

Nikolova et al. 2003 [9], presented a code compression scheme for improving SoC performance. They have designed a hardware Decompressor based on dictionary technology with studied case of changing flow. The compression ratio that achieved is among 57% .

Das et al. 2005 [11] applied a code compression on variable length instruction set processors whose encodings are already optimized to a certain extent with respect to their usages. They developed a dictionary-based algorithm that utilizes unused encoding space of an instruction set architecture to encode code-words, and addresses issues arising out of variable length instructions. Their compression achieved by the machine shows a (10-30)% compression for the various benchmarks.

Yuan Xie et al 2006, [12] present a class of code compression techniques called variable-to-fixed code compression (V2FCC), which uses variable-to-fixed coding schemes based on either Tunstall coding or arithmetic coding. These techniques are suitable for both reduced instruction set computer (RISC) and very long instruction word (VLIW) architectures. They favor VLIW architectures that require a high-bandwidth instruction pre-fetch mechanism to supply multiple operations per cycle, and fast decompression, which is critical to overcome the communication bottleneck between memory and CPU. Their experimental results for a VLIW embedded processor TMS320C6x show that the compression ratios using memory-less V2FCC and Markov V2FCC are around 82.5% and 70%, respectively. Decompression unit designs for memory-less V2FCC and Markov V2FCC implemented in TSMC 0.25- m technology.

Choosing the compression algorithm to design hardware Decompressor demands improved compression on the instruction size with small amount of dictionary size as shown in Equation 1. The goal of this research is to achieve a Decompressor design and addressing the required dictionary size for improving compression ratio using the Lossless LZW algorithm. The proper dictionary size with variant size of Benchmark will show how to control extending of the dictionary and the effectiveness of dictionary size on achieving compression ratio in terms of saving hardware area.

Methodology:

In our method, we compile the Benchmark program with 32-Bit ARM processor. Studying Benchmark with binary format should be shown the low code density of outcome file after compression process with the chosen LZW algorithm. The compressed code will store in the memory named compressed memory. The untouched processor will have the retrieved instruction after decompressed by the hardware Decompressor engine. The Decompressor engine has been designed according to the same algorithm therefore the retrieved instruction will be exactly same as the original instruction.

How many items will be stored in the dictionary and how many bits long for every item will determine required dictionary size for doing the compression process. Focusing on the dictionary size will be the key for improving the compression ratio on interesting Benchmark. The results will be taken on variant size of Benchmarks to investigate the proper dictionary size for variant cases. The Decompressor engine is illustrated in the figures 1a, 1b, and 2.

The structure of our proposal Decompressor unit is showing in the Fig.1a. This module include is mainly constituted by FSM Fig. 1b, code ram control, stack ram control and bit to byte conversion logic. FSM is the core control logic; code ram control logic will read the coded code from the outside IN BUF and generate the address for code ram and char ram; stack ram control logic will generate the address to the stack ram; bit to byte logic will write the decompressed binary string in byte mode to the OUT BUF.

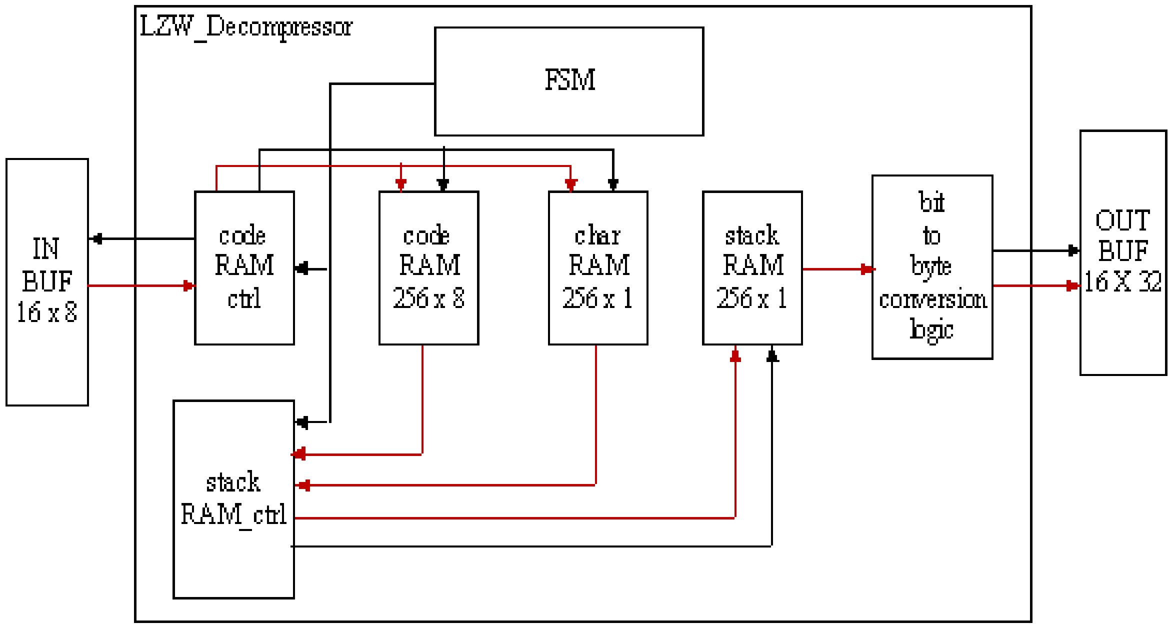


Fig. 1a: Architecture of Decompressor Unit

FSM explanation will be illustrated in this section as follow;

IDLE: FSM is idle when in this state. When detect the decode _ena then the decode process will be started then FSM will go to the RD_DATA state.

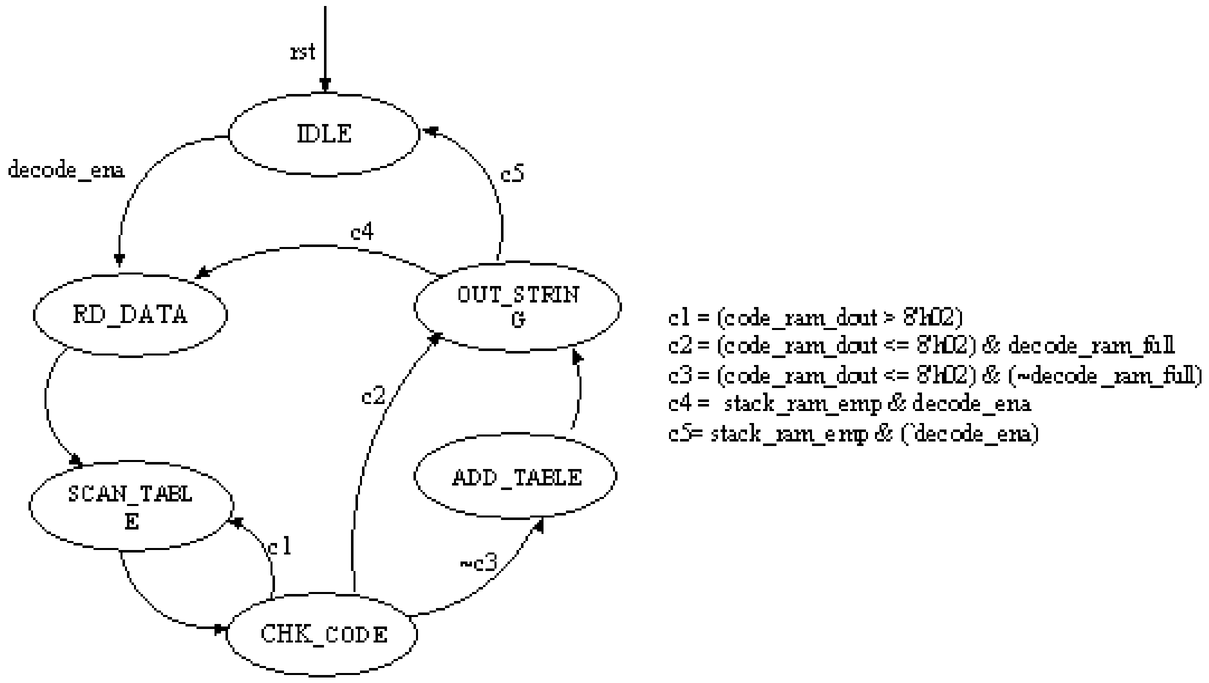
RD_DATA: In this state, the code ram control logic will read one code from the outside IN BUF then FSM will go to the SCAN_TABLE state.

SCAN_TABLE: In this state, the code ram control logic will scan the code ram and he char ram according to the input code. Then FSM will go to the CHK_CODE state .

CHK_CODE: In this state FSM will check the result of scan. If the code read from the code ram is great than 8’h02 then the scan should be continued with the code form the code ram as the new scan address. Until the code from the code ram is less or equal 8’h02 the decode process of current code is completed. During this scan the char from the char ram will be stored into the stack ram one by one. Then FSM will go to the ADD TABLE state.

ADD_TABLE: In this state FSM will add the previous code into the and the first character of current code into the char ram. Then FSM will go to the OUT_STRING state.

OUT_STRING: In this state FSM will read stack ram in reverse order and generate the decompressed binary string. When the stack ram is empty then current decode is completed. If the is active then another code in is needed to be decompressed and FSM will go to RD_DATA state or FSM will go to IDLE state and this module will stop work.



c1 = (code_ram_dout > 8'h02)
c2 = (code_ram_dout <= 8'h02) & decode_ram_full
c3 = (code_ram_dout <= 8'h02) & (~decode_ram_full)
c4 = stack_ram_emp & decode_ena
c5= stack_ram_emp & (`decode_ena)

Fig. 1b: Finite State Machine

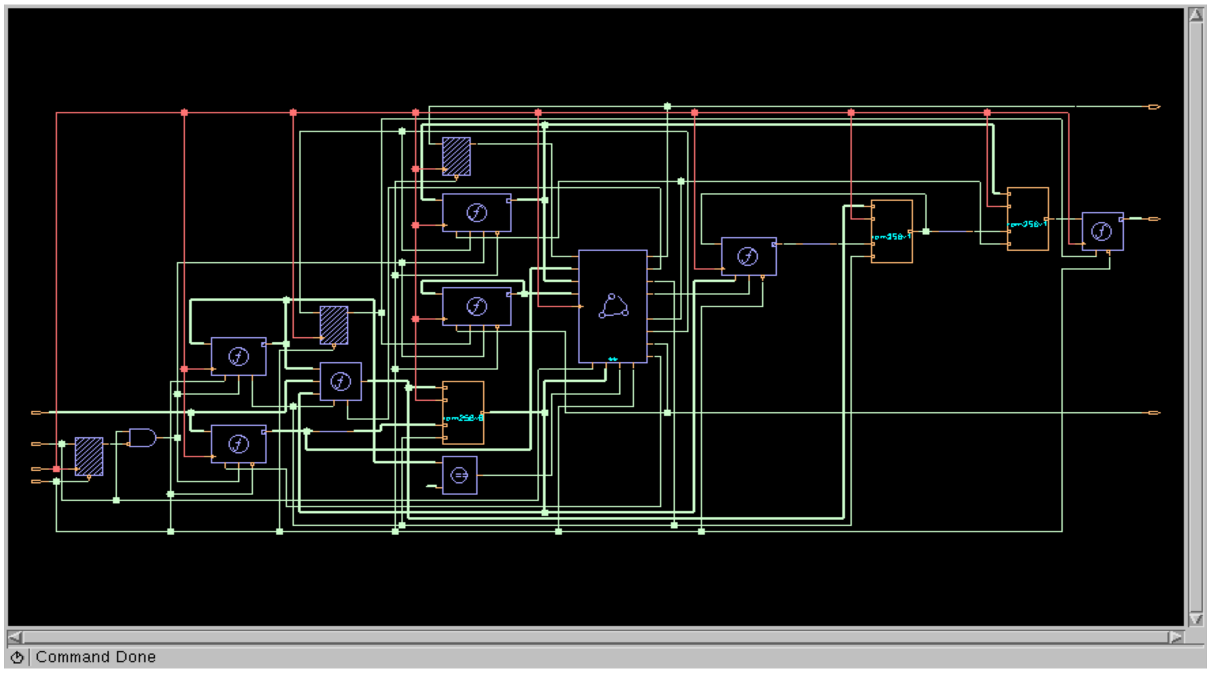


Fig. 2: Circuit of Decompressor Unit

After implementation the Decompressor unit, we have gotten some data that are shown in tables 1, 2.

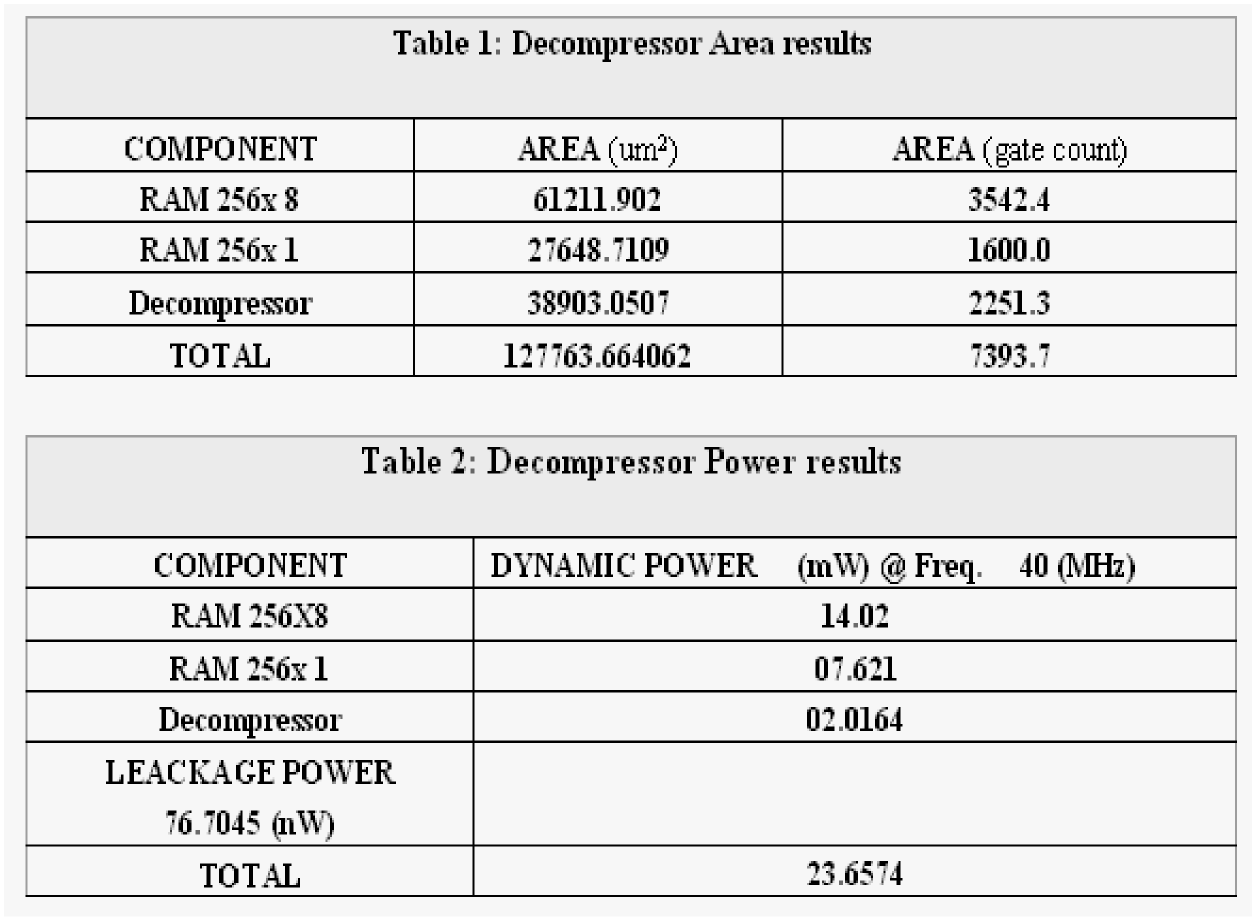


Table 1: Decompressor Area results

COMPONENT	AREA (um²)	AREA (gate count)
RAM 256x 8	61211.902	3542.4
RAM 256x 1	27648.7109	1600.0
Decompressor	38903.0507	2251.3
TOTAL	127763.664062	7393.7

Table 2: Decompressor Power results

COMPONENT	DYNAMIC POWER (mW) @ Freq. 40 (MHz)
RAM 256X8	14.02
RAM 256x 1	07.621
Decompressor	02.0164
LEACKAGE POWER 76.7045 (nW)	
TOTAL	23.6574

From the above power's results, we conclude the consumption power by Decompressor at specific frequency is small, beside the most of total power spend on the memory work. However, the study is showing our design could be running in frequency more than 250-500 MHz, which is showing the successful selection of LZW algorithm to implement as hardware in such system.

E. R. Bello [13] et al has designed Decompressor in Dictionary-Base for SPARC processor. He reported his Engine in 40MHz consume power 791mW and Compression Ratio 78.631 for specific Benchmark. From the above power's results, we conclude the consumption power by our Decompressor at same frequency 40MHz is small 23.6574 mW. The most of total power spend on the memory work. In other hands, the idea of reducing memory size defiantly will lead for reducing power overall system .

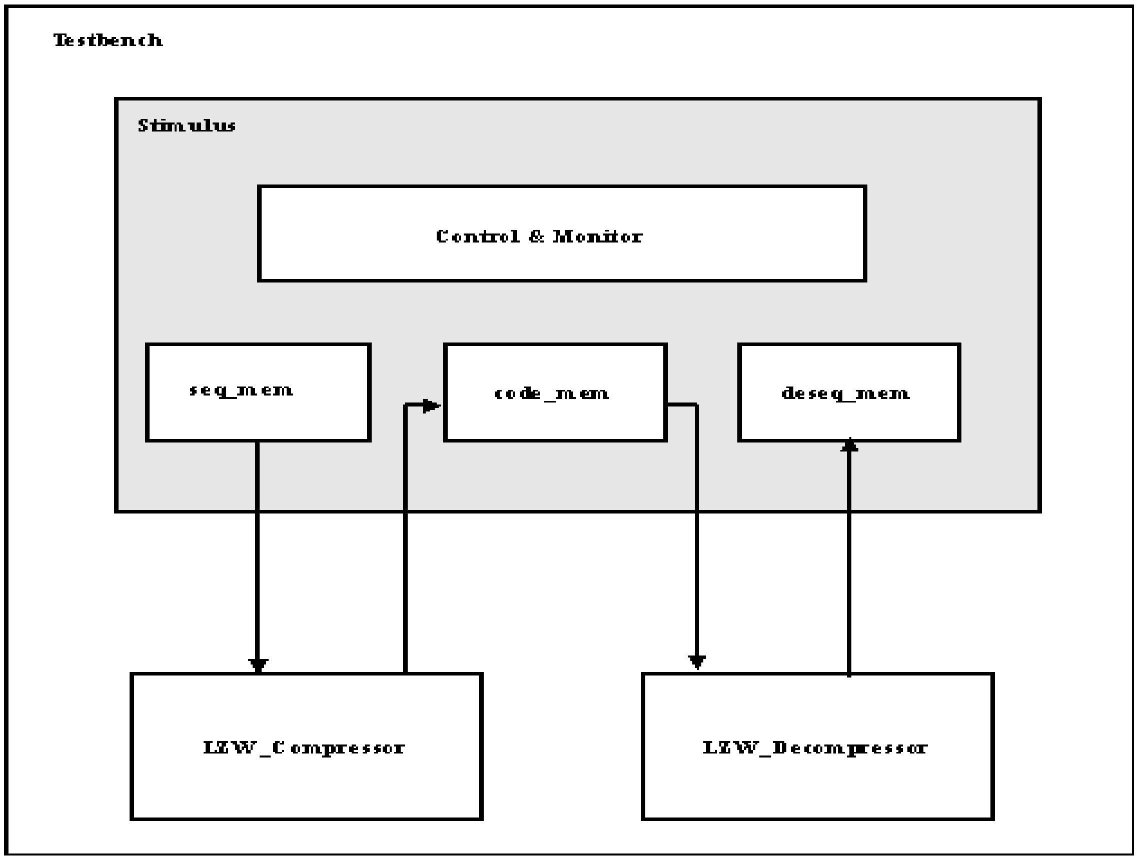


Fig.3: Block diagram illustrate the designed Testbench

The Testbench that is designed is presented in figure 3. In the Testbench one task named seq_gen() will generate a random binary string and store it in the seq_mem which size is 8192x32 bits. The another task named compress_gen() will start LZW_ENCODER module to start the compression. LZW_ENCODER module will read the data need to be compressed from the seq_mem one by one. The compression result will be stored into the code_mem which size is 20000x8 bits. After all the data have been compressed one task, named decode_gen will start the LZW_DECODER to the decompressing operation. LZW_DECODE module read the code one bye one from the code_mem and do the decompressing. The result will be stored into the deseq_mem which size is 8192x32 bits. Compare the result in the seq_mem and deseq_mem we can verify the LZW_ENCODER and LZW_DECODER module. During the verification, some raw data and result data should be logged into some log file. We can directly compare the log files too.

Experimental Results and Verification:

Study of Benchmarks of binary format with variant samples rate will provide obvious details about proper dictionary size. Sample rate with 9 bits long and dictionary size is 256 items. Compressing the record is given the results shown in table 3.

Table 3: Compressing Record

Program name	Original size (bits)	Size after compressing (bits)	Compression ratio (%)
armex	1,564	1,000	63.94
subrout	1,608	1,038	64.55
thumbsub	1,684	1,079	64.07
jump	1,800	1,174	65.22
strcpy	1,828	1,210	66.19
word	1,932	1,218	63.04
blocks	2,068	1,283	62.04
tblock	2,075	1,286	61.98
loadcon	5,908	1,240	20.99
adrlabel	9,672	1,259	13.02
ldrlabel	9,736	1,276	13.11
strtest	43,716	30,954	70.81
swi	46,464	32,729	70.44
shapes	51,256	36,340	70.90
bmw	61,100	41,392	67.74
sorts	61,492	43,353	70.50
newtst	71,372	52,025	72.89
dhrystone	96,272	68,607	71.26
966dhry	103,324	73,625	71.26
920dhry	104,108	73,980	71.06
946dhry	104,760	74,268	70.89
926dhry	105,580	74,884	70.93
primes	428,628	301,402	70.32
strmtst	477,444	335,341	70.24
fft_v5TE	1,274,052	912,892	71.65
fft_v4	1,274,236	913,455	71.69
The average compression ratio			62.34

We have done the same process for different sample rate and we gotten different dictionary size. The results are presented in figure 4, which shows the fact that dictionary size is not bigger to give better results to get. The dictionary of 11 bits with 1792 items is given better compression ratio. However, under hardware's considerations, we demand smaller dictionary size than better compression ratio. Therefore, dictionary of 10 bits with 768 dictionary items seem completely better and spent hardware price is not too high for both cases.

In the figure 5, it illustrated the compression ratio as a function to variant Benchmarks. We can conclude as following:

(a) Although we have gotten different compression ratio with different benchmarks, but distinctly curves trend is still constant.

(b) Considering file size, there is unlike repose for variant cases. It can be seen from the left side of groove, the file size less than 10 KB with 9 bits and 10 bits item long is effective. For the groove when the file is at 10-50 KB with 11-12 bits item long has a best result. On the right side of the groove, the file is above 50 KB with 13 -14 bits item long takes the best result. For the file above 1M, because the sample is less the rest part of the curvilinear cannot be taken into consideration.

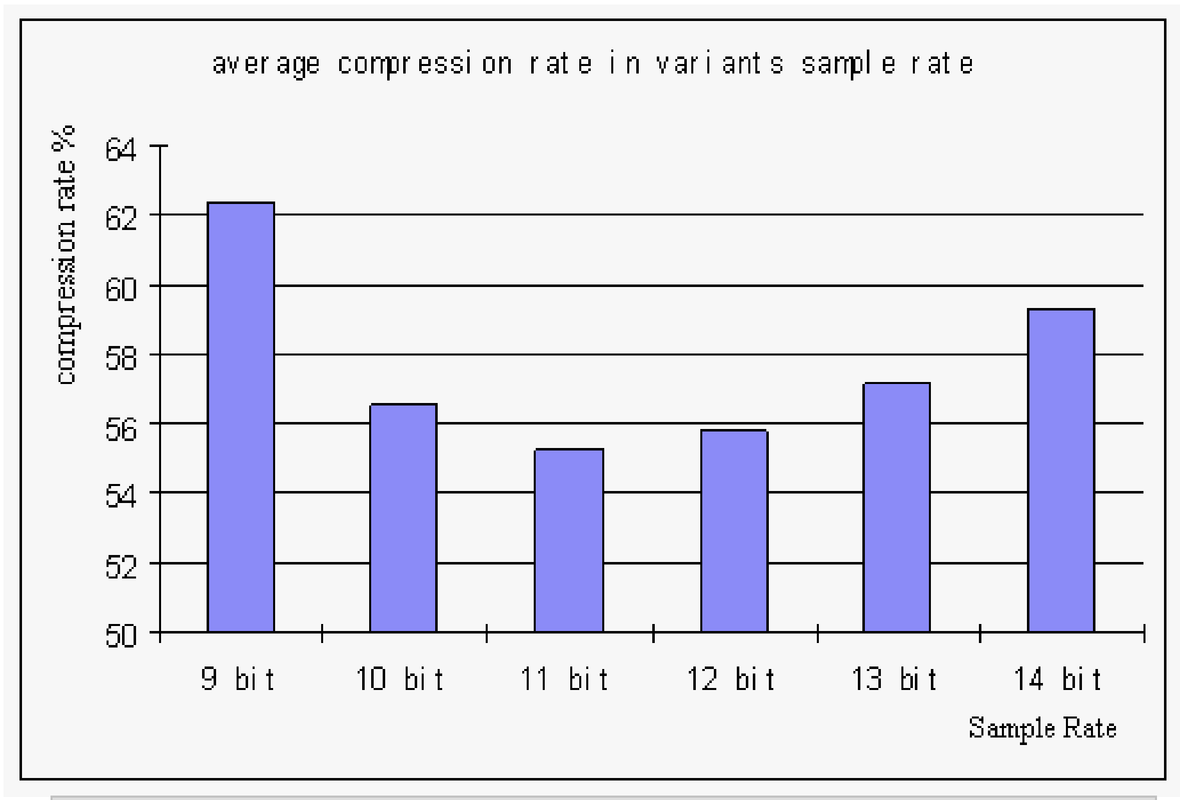


Fig 4: Dictionary Size with Compression Ratio

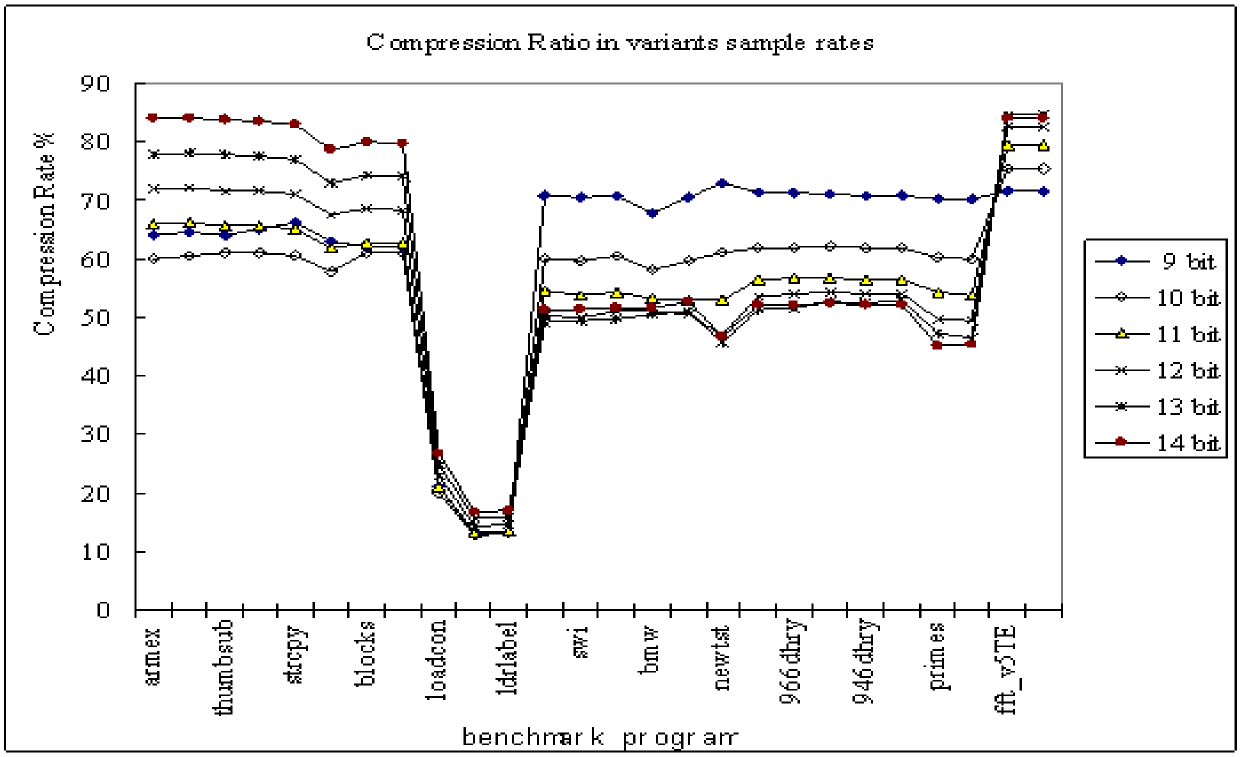


Fig 5: Compressed Ratio with different Sample

Conclusion Future work:

The technique of Code Compression with specific algorithm will deal with the binary program and compressed it in offline case which stored in program-memory then retrieve its instruction by using hardware (Decompressor) in order to processor's demand, we suppose not to touch the processor architecture. The Code Compression is a technique to reduce the code redundancy and to increase the code density, and it is effective for application in the field of memory storage and code transmission in terms of effective utilization of size area and power consumption. The code compression technique is taking the complier output then process the compression on it, the compressing output will store in the memory and retrieve again by using some hardware Decompressor on demand. The code compressions need no change on the complier or ISA.

We have chosen the algorithm LZW as efficient code compression technique to compress program in binary format. The compression is completed in offline case. We have designed hardware Decompressor according to the same algorithm in order to retrieve the compressing small size program to the processor on demand. We have verified our design by designing a Testbench. We have placed the all related design reporting the dissertation. However, we have used variety size Benchmark to confirm the compression ratio outcome by the LZW scheme. Our experimental results show that the compression ratio is varies according to the size of original file. The average compression ratio is 62.34% for a dictionary with 9 bits code and 256 items, where the maximum value of compression ratio is 72.89%. If we use another dictionary with more than 10 bits, the compression ratio will vary a little. When is used dictionary with 9 bits code, the dictionary size can be limited less than 4 KB.

Applications of our research is about to install the Decompressor which designed by us in the embedded system that employ in network area. This technique will help to reduce the memory size and power consume by the system. In the next of our pan is research the Decompressor performance.

References

1. T. A. Welch. 1984, A Technique for High Performance Data Compression. IEEE Computer. Vol. 17; 8~19.
2. D. A. Huffman. 1952, A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the I. R. E., Vol. 40 (9); 1098~1101.
3. J. J. Rissanen and G. G., Mar. 1979, Langdon. Arithmetic Coding. IBM J. Research Develop. Vol. 23, No. 2: 146~162
4. J. Ziv, A. Lempel. 1978, Compression of Individual Sequences via Variable-Rate Coding. IEEE Transactions on Information Theory. Vol. IT-24, No. 5: 530~536
5.A. Wolf and A, Chanian. 1992, Executing compressed programs on an embedded RISC architetecture. In Proceeding Int'l Symp. On Microarchitecture., 81~91
6.M. Game and A. Booker. 1998, Codepack: Code compression for PowerPC Processors. International Business Machines (IBM) Corporation..
7.H. Lekatsas, J. Henkel, and W. Wolf. 2001, Design and Simulation of pipelined decompression architecture for embedded systems. In Proc. ACM/IEEE Int'l Symp. on System System Synthesis. 63~68
8.H. Lekatsas, J. Henkel, and W. Wolf. 2000, Code Compression for low power embedded system design. Proc. ACM/IEEE Design Automation Conference., 294-299
9.E. G. Nikolova, D. J. Mulvaney, V. A. Chouliaras, and J. L. Nunez-Yanez. 2003, A Code Compression Scheme for Improving Soc Performance. International Proceedings Symposium on System-on-Chip, 35~40
10.L. Benini, A. Macii, and A. Nannarelli. 2002, Code compression for cache energy minimization in embedded systems. IEE proceedings on computers and digital techniques. 149(4): 157-163
11.D. Das,R. Kumar, and P. P. Chakrabarti. 2005, Dictionary based code compression for variable length instruction encoding. 18th International Conference on VLSI Design. 545~550
12.Yuan Xie, W. Wolf, and H. Lekatsas. 2006, Code compression for embedded VLIW processors using variable-to-fixed coding.IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 14(5): 525~536
13.E. Billo, R. Azevedo, G. Araujo, and E. Wanderley Netto. 2005, Design of a Decompressor Engine on a SPARC Processor. SBCCI'05 Florianopolis, Brazil..

CODE DENSITY OPTOMIZATION AND IMPLEMENTATION USING LZW APPROACH BASED ON 32-BIT RISC PROCESSOR

عبدالله عبد الامير حسين اليحيى حسـين علي خضر العيداني

جامعة البصرة – كليـة التربيـة – قسـم علوم الحاسبات

الخلاصة

الاستخدام الواسع للهواتف الخلوية بغية الاتصال بالانترنت ، وبوجود تقنيات الضغط والتشفير والتي تلعب دوراً مهماً في حفظ المساحة الخزنية للذاكرة والبيانات في نفس الوقت والقدرة المستهلكة في المعالجتها ، فقد ركزت دراســتنا على تحديد وصياغة كم عدد المقاطع (Items) وطولها بالبت (bit) بالنسبة للبيانات المستخدمة والتي سوف تخزن في القاموس ، اذ ان معالجتنا حددت افضل قاموس من حيث الحجم الخاص بعمليات الضغط. فان التركيز على حجم القاموس سيكسب المعالجة الحصول على مفاتيح لتطوير نسبة الضغط (Compression Ratio) .

ان نتائج دراستنا وضحت وبينت احجام مختلفة من Benchmarks وذلك للبحث عن افضل حجم للقاموس ، اذ ان معدل النتائج يشير الى ان نسبة الضغط كانت ٦٢,٣٤ % لقاموس بحجم 9 bit و 256 مقطع ، في حين كانت القيمة العظمى للضغط 72.89 % لنفس القاموس المستخدم وبنفس عدد المقاطع ايضاً. بينما استخدمنا قاموس اخر بطول اكثــر من 10 bit فأن نسبة الضغط سوف تتغير قليلاً.