REDUCTION OF BUS TRANSITION FOR COMPRESSED CODE SYSTEMS

S. R. Malathi, R. Ramya Asmi

Department of Computer Science and Engineering,
Sri Venkateswara College of Engineering, Chennai, India
malathiraj@svce.ac.in, rramyaasmi@gmail.com

ABSTRACT

Low power VLSI circuit design is one of the most important issues in present day technology. One of the ways of reducing power is to reduce the number of transitions on the bus. The main focus here is to present a method for reducing the power consumption of compressed-code systems by inverting the bits that are transmitted on the bus. Compression will generally increase bit-toggling, as it removes redundancies from the code transmitted on the bus. Arithmetic coding technique is used for compression /decompression and bit-toggling reduction is done by using shift invert coding technique. Therefore, there is also an additional challenge, to find the right balance between compression ratio and the bit-toggling reduction. This technique requires only 2 extra bits for the low Power coding, irrespective of the bit-width of the bus for compressed data.

KEYWORDS

Low power VLSI, Bus transition reduction, Arithmetic coding, Compressed Code systems.

1. INTRODUCTION

Designs of portable consumer electronic devices such as mobile phones, PDAs, video games, and other systems are increasingly demanding low power consumption to maximize the battery life, reduce weight and improve reliability. These types of power sensitive devices are usually equipped with microprocessors as the processing elements and memories as the storage units. With current CMOS technology, a large portion of power consumption is in the form of dynamic power, which in turn is determined by the bit switching (bit-toggling) and the switched load capacitance. Since the microprocessor fetches instructions over the memory bus every clock cycle and bus lines to memory are often much longer than buses within the processor, the power consumed by the bus due to instruction fetch is significant.

So far, research for the instruction data bus switching reduction has generally concentrated on code compression. The compressed code causes less memory access, thus reducing the bus activity. Compression requires complicated compression/decompression units, which reside in the critical path and can considerably affect the overall system performance. Apart from the memory optimization aspect of code compression, it is desirable to minimize bit-toggling on the bus, since the energy consumed on the bus is proportional to the number of bit-toggles. If **h** denotes the total number of toggles on one bus-line, $C_{eff\text{-}line}$ is the bus capacitance taking into account cross talk (interaction between other lines) and V is the voltage difference between high and low then the energy consumed on one bus-line is given by

$$Energy_{bus\text{-}line} = 0.5 * \mathbf{h} * C_{eff\text{-}line} * V^2$$

DOI : 10.5121/vlsic.2013.4110

This explains reason for reducing the number of bit-toggles. So we concentrate only on bit-toggling minimization and therefore our results show improvement only on bus power consumption.

In this paper, we investigate a different approach—bus encoding [11]. Most of existing bus encoding schemes are effective for address or data memory buses and mainly utilize correlations of transferred data. For example, T0 [1] and Gray encodings [2] use the temporal correlation of data on address buses, while the bus-invert encoding [3] exploits the spatial transition correlation among the data bits. Our investigation is on the instruction data buses and found that the bit switching behavior of the instruction data bus is different from those of the other types of buses. Figure 1 shows an experimental result of the bit transition probability for three different memory buses: instruction memory address bus (*imab*) instruction memory data bus (*imdb*) and data memory data bus (*dmdb*) .As can be seen from Figure 1, switching activity on the instruction address bus concentrates on the low section of bits, largely due to the sequential access of instruction memory. For the data memory data bus, the switching activity spreads over all bus bits with almost 50% switching probability. But for the instruction data bus, the switching probability is not evenly distributed. Some bits show very low switching activity. Therefore, most of existing encodings for address buses and data memory data buses do not suit for encoding of the instruction data buses.

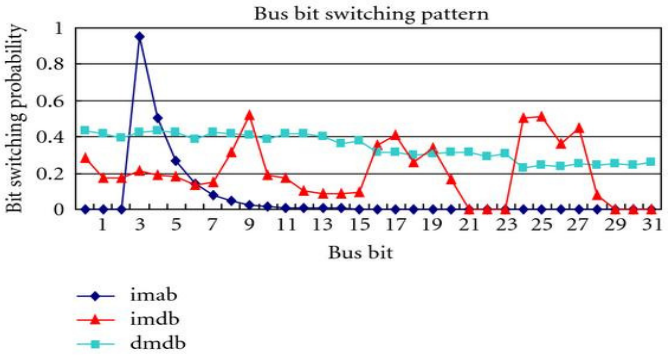

Figure 1. Bit switching probability of different buses.

2. EXISTING WORK

Bus encoding techniques for low power consumption have been studied in the last couple of decades. The Gray encoding [2] was proposed for the instruction address bus where binary addresses are converted into Gray code for bus transmission. Another approach [1] for address bus encoding is the asymptotic zero-transition activity encoding, known as T0. For the instructions of a program to be executed sequentially without any branches, T0 can ideally achieve zero bus switching. In [4], Henkel and Lekatsas presented an adaptive address bus encoding (A^2BC) for low power address buses in the deep submicron design, where the coupling effects of bus lines were considered.

Stan and Burleson [3] proposed the bus-invert (BI) coding. This method uses either the original or the inverted value to encode the data bus. If the current value to be sent over the bus causes more than half of the bus bits to switch, its inverted value will be transferred on the bus. For the wide data bus without evenly distributed random data, the same authors proposed a partitioned bus-invert coding, partitioning the wide bus into several narrow sub-buses and applying the BI encoding to each sub-bus. This partitioning approach improves the switching reduction at the cost of extra invert control lines. The partitioned bus-invert approach has been modified and proposed

as partial bus invert (PBI) coding [5] for the address bus. In the same paper, they extended this approach to multi way partial bus-invert (MPBI), where highly correlated bus lines were clustered into multiple sub-buses and each of them was encoded independently.

A dictionary-based approach to reduce data bus power consumption has been introduced in [6]. This approach exploits frequent data patterns detected from the application trace and uses two synchronized dictionaries on both sides of the bus. The dictionaries cache recently transferred data so that the same data that can be accessed in the local dictionary will not be transferred on the bus to reduce bus switching activity.

For instruction bus power reduction, most previous researchers have focused on code compression. The pioneer work by Wolfe and Chanin [7] mainly aimed for program memory reduction. With their approach, the total bus switching activity can be reduced via compressed codes that are transferred over the bus. A decompression unit is required to restore each instruction before execution. Scheme in [8] also compresses instructions and compacts more compressed instructions into one bus word to reduce the total number of memory access, hence the total number of bus switches. This code compression scheme was extended in [9] to further reduce switching between consecutive instruction words.

Petrov and Orailoglu [10] introduced an instruction bus encoding, where the major loops are encoded and stored in the memory so that when they are fetched, the switching activity on the bus is minimized. This approach can achieve good switching reduction but requires a complex code transformation and control in the decoding logic.

In [15], codeword assignment through heuristics is used to reduce the overall power consumption. In [16] the authors use a reconfiguration mechanism, called Instruction Re-map Table, to re-map the instructions to shorter length code words. Using this mechanism, frequently used set of instructions are compressed. This reduces code size and hence the cost. The same mechanism is used to target power reduction by encoding frequently used instruction sequences to Gray codes. Such encodings, along with instruction compression, reduce the instruction fetch power.

3. PROPOSED WORK

3.1 Proposed Method for Reducing Bus Transition

In this paper, we apply shift-invert coding technique onto the compressed data using Arithmetic Coding technique which reduces the bit-toggling to a large extent.

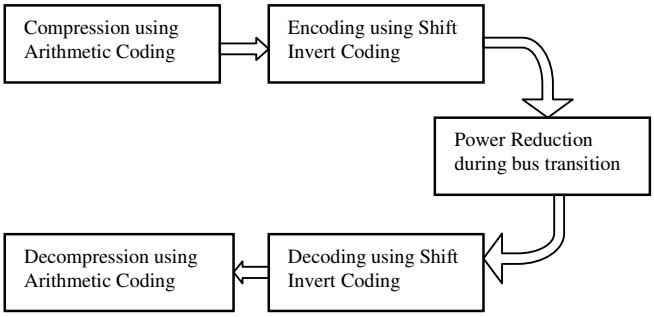

Figure 2. Flow diagram of the proposed method

3.2 Overview of Arithmetic Coding

Shannon [14] showed that for the best possible compression code (in the sense of minimum average code length), the output length contains a contribution of -lg p bits from the encoding of each symbol whose probability of occurrence is p. If we can provide an accurate model for the probability of occurrence of each possible symbol at every point in a file, we can use arithmetic coding to encode the symbols that actually occur; the number of bits used by arithmetic coding to encode a symbol with probability p is very nearly -lg p, so the encoding is very nearly optimal for the given probability estimates. It is used to encode an entire file into single interval. The most important advantage of arithmetic coding is its flexibility: it can be used in conjunction with any model that can provide a sequence of event probabilities. This advantage is significant because large compression gains can be obtained only through the use of sophisticated models of the input data. Models used for arithmetic coding may be adaptive, and in fact a number of independent models. The other important advantage of arithmetic coding is its optimality. Arithmetic coding is optimal in theory and very nearly optimal in practice, in the sense of encoding using minimal average code length.

3.2.1 Arithmetic coding algorithm and its implementation

The algorithm for encoding a file using arithmetic coding works conceptually as follows. With given probabilities of symbols, the algorithm works in three steps.

1) Starts with a "current interval" [H, L) set to [0, 1).
2) For each symbol of the input file, perform steps (a) and (b).
   (a) Subdivide current interval into subintervals, one for each symbol.
   (b) The size of a subinterval is proportional to the probability that the symbol will be the next symbol in the file. Then select the subinterval corresponding to the symbol that actually occurs next and make it the new current interval.
3) Output enough bits to distinguish the final current interval from all other possible final intervals

The length of the final subinterval is clearly equal to the product of the probabilities of the individual symbols, which is the probability p of the particular sequence of symbols in the file. The final step uses almost exactly -lg p bits to distinguish the file from all other possible files. The end of the file is indicated using either a special end-of-file symbol coded just once, or some external indication of the file's length.

A. Pseudo code for arithmetic encoding is as follows:

```
Set lower bound = 0
Set upper bound = 1
While there are still symbols to encode
        Current range = upper bound - lower bound
        Upper bound = lower bound + (current range × upper bound of new symbol)
        Lower bound = lower bound + (current range × lower bound of new symbol)
end while
```

B. Pseudo code for arithmetic decoding is as follows:

```
Encoded value = encoded input;
While string is not fully decoded,
        identify the symbol containing encoded value within its range;
        Current range = upper bound of new symbol – lower bound of new symbol
        Encoded value = (encoded value - lower bound of new symbol) ÷ current range
end while
```

Example 1: Encoding the file containing symbols "bbb" using arbitrary fixed probability estimates P (a) = 0.4, P (b) = 0.5, and P (EOF) = 0.1[12] where P(a),P(b) and P(EOF) be the probability of a, b and EOF. Encoding proceeds as follows:

Table 1. Probability and Range of the symbols

Symbol	Probability	Range
a	0.4	[0.0,0.4)
b	0.5	[0.4,0.9)
EOF	0.1	[0.9,1.0)

Range is calculated by using cumulative frequency as shown in Figure 3;

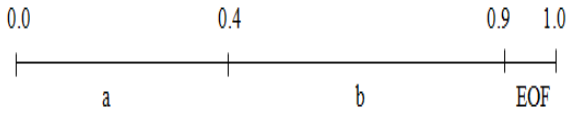

Figure 3. Range calculation

The result of encoding the symbol "bbb" using the pseudo code for encoding is shown in the Figure 4. The symbol "bbb" till end-of-file is encoded as [0.8125, 0.825). To decode, the lower bound of the target interval [0.8125, 0.825) is taken. The lower bound is 0.8125. Apply arithmetic decoding algorithm till all input symbols are met. The encoded input is lower bound of the target interval. The output of the decoding is "bbb". The idea of arithmetic coding originated with Shannon in his seminal 1948 paper on information theory [14].

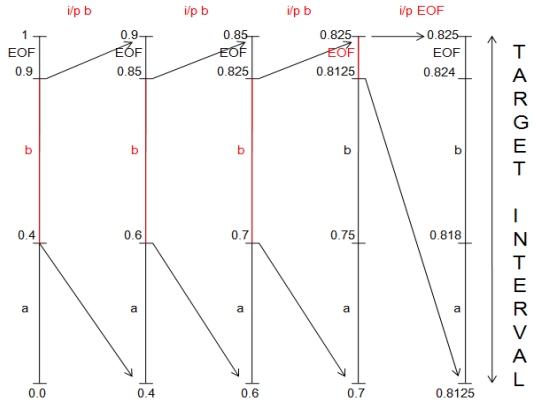

Figure 4. Arithmetic encoding of symbol 'bbb'

The basic implementation of arithmetic coding has two major difficulties: the shrinking current interval requires the use of high precision arithmetic, and no output is produced until the entire file has been read. The most straightforward solution to both of these problems is to output each leading bit as soon as it is known, and then to double the length of the current interval so that it reflects only the unknown part of the final interval. Witten, Neal, and Cleary [13] added a clever mechanism for preventing the current interval from shrinking too much when the endpoints are close to ½ but straddle ½. In that case we do not yet know the next output bit, but we do know that whatever it is, the following bit will have the opposite value; we merely keep track of that fact, and expand the current interval symmetrically about ½. This follow-on procedure may be repeated any number of times, so the current interval size is always longer than ¼. The coding

and interval expansion working [12] is described in detail. This process takes place immediately after the selection of the subinterval corresponding to an input symbol. The following steps are repeated as many times as possible:

a) If the new subinterval is not entirely within one of the intervals [0, ½), [¼, ¾), or [½, 1), stop iterating and return.

b) If the new subinterval lies entirely within [0, ½), output 0 and any 1s left over from previous symbols; then double the size of the interval [0, ½), expanding toward the right.

c) If the new subinterval lies entirely within [½, 1), output 1 and any 0s left over from previous symbols; then double the size of the interval [½, 1), expanding toward the left.

d) If the new subinterval lies entirely within [¼, ¾), keep track of this fact for future output; then double the size of the interval [¼, ¾), expanding in both directions away from the midpoint.

Example 2: The detail of encoding the same file in example 1 is continued. The "follow" (in Table 2) output in the sixth line indicates the follow-on procedure. We keep track of our knowledge that the next output bit will be followed by its opposite; this opposite" bit is the 0 output in the ninth line. The encoded file is 1101000, as before. Clearly the current interval contains some information about the preceding inputs; this information has not yet been output, so we can think of it as the coder's state. If 'a' is the length of the current interval, the state holds- lg a bit not yet output. In the basic method (illustrated by Example 1) the state contains all the information about the output, since nothing is output until the end. In the implementation illustrated by Example 2, the state always contains fewer than two bits of output information, since the length of the current interval is always more than ¼. The final state in Example 2 is [0; 0.8), which contain -lg 0.8 ≈ 0.322 bits of information.

Table 2. Illustration of Interval expansion procedure

Current interval	Action	subintervals			Input
		a	b	EOF	
[0.00,1.00)	Subdivide	[0.00,0.40)	[0.40,0.90)	[0.90,1.00)	b
[0.40,0.90)	Subdivide	[0.40,0.60)	[0.60,0.85)	[0.85,0.90)	b
[0.60,0.85)	Output 1 Expand [½,1)				
[0.20,0.70)	Subdivide	[0.20,0.40)	[0.40,0.65)	[0.65,0.70)	b
[0.40,0.65)	Follow Expand [¼,¾)				
[0.30,0.80)	Subdivide	[0.30,0.50)	[0.50,0.75)	[0.75,0.80)	EOF
[0.75,0.80)	Output 10 Expand [½,1)				
[0.50,0.60)	Output 1 Expand [½,1)				
[0.00,0.20)	Output 0 Expand [0,½)				
[0.00,0.40)	Output 0 Expand [0,½)				
[0.00,0.80)	Output 0				

3.4 Overview of Shift-invert coding

The main idea in this technique is to optionally shift the data bits by one bit position (either left-shift or right-shift) if the shifting reduces the number of bus transitions [11]. And it requires only 2 extra bits for the low power coding, regardless of the bit-width of the bus and does not assume anything about the nature of the data.

The terminologies used are

w = Width of the string
k = Time instant
α = Data yet to be transmitted.
β = Data transmitted on the bus.
$\alpha^k = \alpha_{w-1}{}^k, \alpha_{w-2}{}^k, ...\alpha_0{}^k$ represents a binary data string at any time instant k.
β^k = Data transmitted on the bus at time k.

Note that the bit width w' of the bus (i.e., the encoded data that gets transmitted on the bus) could be greater than w depending on the coding scheme used. For instance, in default Bus Invert Coding [3], w' = w + 1. For variations of Bus Invert Coding, w' can be greater than w+1. In any coding scheme based on bus inversion, the value of the i^{th} bit, b_i on the bus will be either the data value α_i or $1-\alpha_i$. Thus, for all i, $0 \leq i < w'$,

$b_i = \alpha_i$, uninverted bit OR
$b_i = 1 - \alpha_i$, inverted bit.

There are four encoding schemes presented in shift –invert coding. These encoding schemes are defined as follows:

(1) *Default*: No encoding is done (just same bits are passed). The default operation on a w-bit data is defined as $\alpha_i^{(DEFAULT)} = \alpha_i$, $0 \leq i < w'$.
(2) *Left-shift*: Data bits are circularly left-shifted by one position. The Left-shift operation on a w-bit data is defined as, $\alpha_i^{(LS)} = \alpha_{i-1}$; $0 \leq i < w'$, and $\alpha_0^{(LS)} = \alpha_{w'-1}$.
(3) *Right-shift*: Data bits are circularly right-shifted by one position. The Right-shift operation on a w-bit data is defined as, $\alpha_i^{(RS)} = \alpha_{i+1}$; $0 \leq i < w'-1$, $\alpha_{w'}^{(RS)} = \alpha_0$.
(4) *Invert*: Inverts all bits. The invert operation on w-bit data is defined as, $\alpha_i^{(INV)}=1-\alpha_i$; 0 ,$\leq i < w'$.

Example 3: Consider the following example. (Ignore the extra bits used in the encoding scheme for a moment).
Let β^k = 01100101 (assume a 8-bit bus) and the new data at time k+1, α^{k+1}= 10110011;
Now let us see how it reduces the bit-toggling in the following Table 3.

Table 3. Number of bit-toggling in each encoding scheme

Data at time - k	β^k	01100101	No. Of bit-toggling
Data at time – k+1	α^{k+1}	10110011	5
Data at time – k+1(LS)	$\alpha^{k+1(LS)}$	01100111	1
Data at time – k+1(RS)	$\alpha^{k+1(RS)}$	11011001	5
Data at time – k+1(INV)	$\alpha^{k+1(INV)}$	01001100	3

In this example, the number of transitions N between βk and αk+1is 5. In the case of Bus Invert Coding [3], let us see whether it is beneficial (i.e., whether the number of 0 to 1 and 1 to 0 transitions are reduced) to send αk+1(INV) over the bus. The number of transitions NINV between βk and αk+1(INV) is 3. Since NINV< N/2, in the Bus Invert Coding [3] technique, αk+1(INV) will be sent over the bus at time k+1. Now, let us see the number of bit toggles when we left-shift the data αk+1 once, as defined above. The left-shifted data at time k+1 becomes, αk+1(LS)=01100111.Comparing this data with βk = 01100101, the number of transitions NLS between a βk and αk+1(LS) is just 1, which is better than the 3 transitions occurring from the inverted data αk+1(INV) .If data is right-shifted then NRS is 5. Thus, in this case, it is clear that by sending the left-shifted data, we can reduce the number of transitions even further than the reduction in bit toggle obtained from sending the inverted data. For each new data that needs to be sent over the bus, we evaluate the transitions N, NINV , NLS and NRS between βk and αk+1, αk+1(INV), αk+1(LS), and αk+1(RS) respectively. We then choose the encoding that results in the least number of transitions. The pseudo code of shift-invert coding [11] is shown as follows;

```
Procedure ShiftInv ()
{
Input  : α^(k+1), β^k
Output: β^(k+1) (SHIFT_INV)
N       ←  num_trans (α^(k+1), β^k)
N^INV   ←  num_trans (α^(k+1(INV)), β^k)
N^LS    ←  num_trans (α^(k+1(LS)), β^k)
N^RS    ←  num_trans (α^(k+1(RS), β^k)
β^(k+1) ←  one of (α^(k+1), α^(k+1(INV)), α^(k+1(LS)), α^(k+1(RS)))
           depending on min (N, N^INV, N^LS, N^RS)
}
```

The procedure num_trans (α, β) returns the number of bit-positions in which the passed in vectors α and β differ. Note that the data that gets sent over the bus, βk+1 can be one of αk+1, αk+1(INV), αk+1(LS), αk+1(RS). Thus, we need to tag the bus with 2 additional bits that indicate the encoding that was used. This will be used to decode the bus value appropriately at the receiving end. Thus, in Shift-Invert coding, the width of the bus w' = w + 2, where w is the width of the data vector. 2 additional bits as compared to 1 additional bit in default Bus Invert Coding [3] is used in Shift Invert Coding [11].

4. HARDWARE MODEL

One way of hardware realization for the Shift-Invert coding is shown in Figure 5. The inputs to the encoder are αk+1 and βk where k denotes the time instance. For illustration purposes, block-diagram of Shift Invert coding for an 8-bit data is shown. Thus, the bit width of αk+1 is 8 and the bit width of βk is 10 which includes the two control signals (C1 C0) used to indicate the mode of encoding. The blocks named "Default", "Left-Shift", "Right-Shift" and "Invert" indicate the 4 encoding schemes used in Shift Invert coding technique. XOR blocks are used to compare and capture the transition between the 8 bits of βk and αk+1. Each of the XOR blocks take two 10-bit inputs and generate one 10-bit output. One of the two 10-bit inputs is the signal βk on the bus at time instant k. The other input to the XOR blocks is the 8-bit data signal αk+1- in some encoded form (depending on the encoding scheme). Note that the bit width of αk+1 is only 8, and the other 2-bits labeled SHIFT-INVk are the 2 additional bits used to indicate the scheme used at time instance k+1.Table 4 shows a bit representation to indicate the encoding scheme used. For example, if the input data is left-shifted before we send it over the bus, the two control bits will be assigned the value 01. Likewise, the other bit assignments are shown in the table.

Table 4. Encoding for Shift Invert coding

Default(no encoding)	00
Left-shift	01
Right-shift	10
Invert	11

The 10:4 Adder-Counter generates an output that can be anywhere in the range [0...10]. This implies that 10:4 Adder counter will generate a 4-bit output. These 4-bits indicate the total number of transitions (i.e. total no. of bit-toggling) on the bus for each type of encoding. Note that in this example, the data bus is assumed to be 8-bits wide and we used an Adder counter block that takes 10 input bits and generats a 4- bit output. In general, for a w-bit data, we will need a (w+2) - bit Adder-Counter block that generates a $[\log_2 (w + 2)]$ bit output. The outputs from the four 10:4 Adder-Counter blocks are sent to a 4-way 4-bit comparator which compares and finds the encoding scheme that has the least number of transitions. The comparator then generates the two control bits C_1C_0 that indicate the encoding scheme chosen for decoding at time instant k+1. The encoded data that has the least number of transitions and the control bits C_1C_0 are then sent over the bus as β^{k+1}.

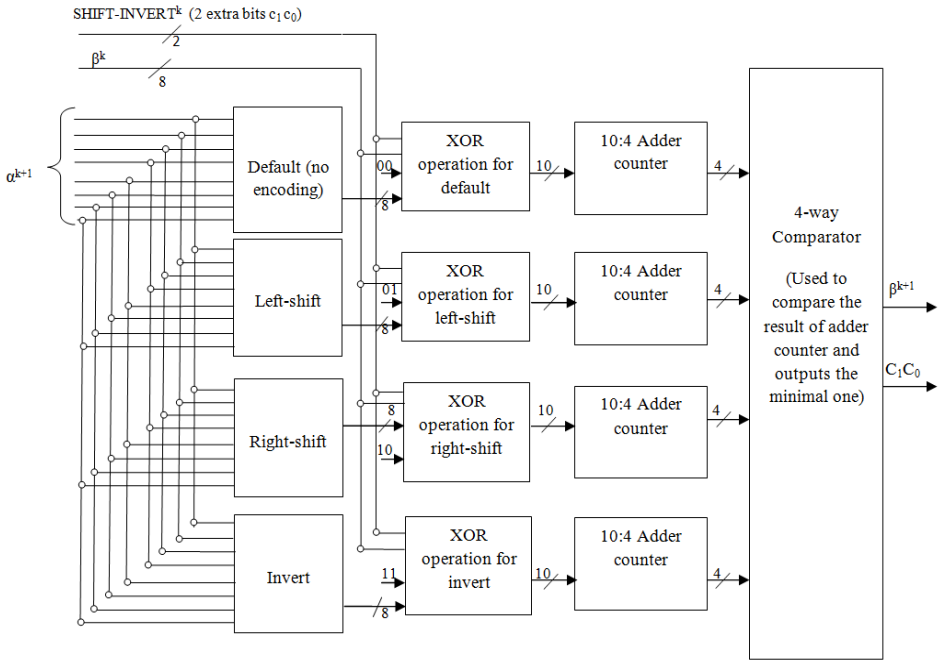

Figure 5. Hardware model for shift-invert coding for bit bus

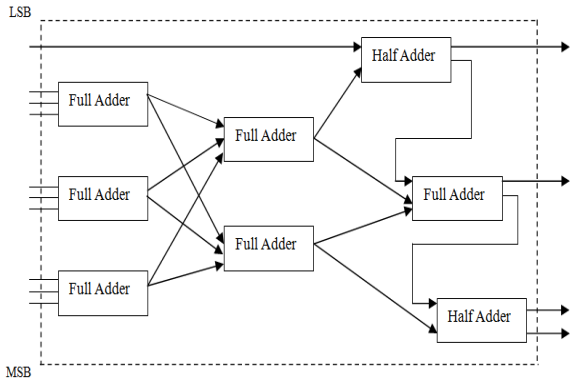

Figure 6. 10:4 Adder counter circuit

A 4:3 Adder counter circuit will be needed for counting number of 1's if the data bus size is 32 bits. Suppose a 32+2 bit, where 32 bit is data and 2 bit is the encoding bits with a total of 34 bit. To count the number of 1's in 34 bit, three 10 bits can be given as a input to 10:4 adder counter (10+10+10=30) and the remaining 4 bits are given as a input to the 4:3 adder counter. Finally, adding all the values of adder counter will result in number of bit-toggling in the two 34 bits compared.

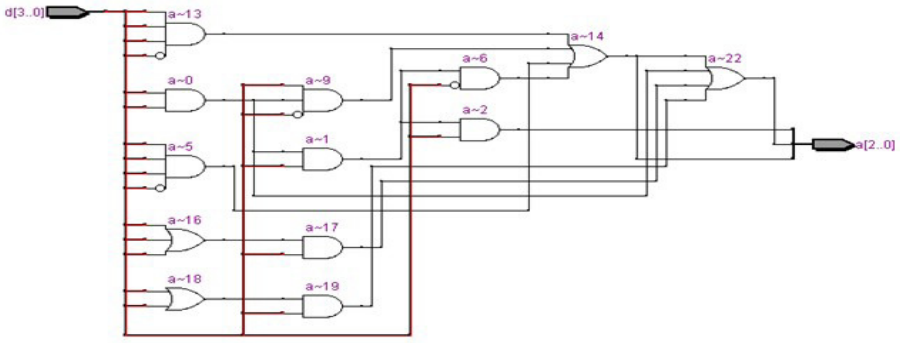

Figure 7. A 4:3 Adder-Counter Circuit

5. EXPERIMENTAL RESULTS

As shown in the architecture of the proposed method, Shift Invert coding technique [9] was applied on data compressed using Arithmetic Coding technique. The implementation was done using C++ followed by Modelsim. Compression using Arithmetic coding was done using C++. The compressed binary data generated was given as input test bench to the simulated hardware model discussed here and timing wave generated using Modelsim to study the bus transition. The bus width was varied as an integer power of 2 and for each width of the bus 100000 simulation cycles were performed. The average number of bus transitions for bus widths of 8, 16 and 32 for the Bus Invert Coding and Shift Invert Coding techniques for the uncompressed and compressed data is listed in Table 5 and Table 6. From Table 5 it can be observed that Shift invert coding reduces the bus transition by an average of 5% than Bus Invert Coding for any random data on a given bus width. When the same encoding techniques are applied on compressed data having more number of transitions due to removal of redundant information, Shift Invert Coding performs much better than Bus Invert Coding as shown in Table 6. The bus transition increases by 53% for Bus Invert Coding and only 4.5% for Shift Invert Coding when applied on compressed data.

Table 5. Average number of bus transitions for random data

	Average number of Transitions per cycle for bus width in 2^n		
	8	16	32
Default Data (no encoding) (DEF)	4.00	8.00	16.00
Bus Invert coding applied on Random Data (BIC)	3.27	6.83	14.23
Shift Invert Coding applied on Random Data (SINV- RD)	3.17	6.60	13.80

Table 6. Average number of bus transitions for compressed data

	Average number of Transitions per cycle for bus width in 2^n		
	8	16	32
Data compressed using Arithmetic Coding (AC-COMP)	5.00	10.00	22.00
Bus Invert coding applied on Data compressed using Arithmetic Coding (AC-BIC)	4.9	9.80	18.85
Shift Invert Coding applied on Data compressed using Arithmetic Coding (AC-SINV)	3.21	6.88	14.50

6. CONCLUSIONS

The proposed method is a combination of Arithmetic Coding and Shift-invert Coding for reducing bus transitions and thereby reduce the power consumed in embedded systems. Arithmetic coding is used to reduce the redundancy in data and thereby reduce the memory access. It also gives high compression ratio. Since it reduces identical data, most of the data in the file is unique .This unique data causes more bit-toggling during data access. To reduce this bit toggling, Shift-invert coding is used on the compressed code. From the experiments conducted it is been proved that Shift Invert Coding reduces bus transition significantly than Bus Invert Coding. Comparison of the results with the work done on similar platform could not be done due to unavailability of such application. As a future work Shift Invert coding can be applied to other compression techniques available for various systems and studied.

REFERENCES

[1] L. Benini, G. de Micheli, E. Macii, D. Sciuto, and C.Silvano, (1997) "Asymptotic zero transition activity encoding for address busses in low-power microprocessor-based systems," in Proceedings of the 7th IEEE Great Lakes Symposium on VLSI, pp. 77–82.

[2] C.-L. Su, C.-Y. Tsui, and A. M. Despain, (1994) "Saving power in the control path of embedded processors," IEEE Design and Test of Computers, vol. 11, no. 4, pp. 24–30.

[3] M. R. Stan and W. P. Burleson, (1995) "Bus-invert coding for low power i/o," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 3, no. 1, pp. 49–58.

[4] J. Henkel and H. Lekatsas, (2001) "A^2BC : adaptive address bus coding for low power deep sub-micron designs," in Proceedings of the 38th Annual Design Automation Conference (DAC '01), pp. 744–749.

[5] Y. Shin, S.-I. Chae, and K. Choi, (2001) "Partial bus-invert coding for power optimization of application-specific systems," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 9, no. 2, pp. 377–383.

[6] T. Lv, J. Henkel, H. Lekatsas, and W. Wolf, (2003) "A dictionary based en/decoding scheme for low-power data buses," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 11, no. 5, pp. 943–951.

[7] A. Wolfe and A. Chanin, (1992) "Executing compressed programs on an embedded RISC architecture," in Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO '92), pp. 81–91.

[8] H. Lekatsas, J. Henkel, and W. Wolf, (2000) "Code compression for low power embedded system design," in Proceedings of the 37th Design Automation Conference (DAC '00), pp. 294–299.

[9] H. Lekatsas, J. Henkel, and W. Wolf, (2005) "Approximate arithmetic coding for bus transition reduction in low power designs," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 13, no. 6, pp. 696–706.

[10] P. Petrov and A. Orailoglu, (2004) "Low-power instruction bus encoding for embedded processors," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 12, no. 8, pp. 812– 826.

[11] Jayapreetha Natesan and Damu Radhakrishnan, (2004) "Shift Invert coding (SINV) for low power VLSI," Proceedings of EUROMICRO Systems on Digital System Design (DSD'04), pp 190-194.

[12] P. G. Howard and J. S. Vitter, (1992) "Practical implementations of arithmetic coding," Image and Text Compression. Norwell, MA: Kluwer Academic, pp. 85–112.

[13] I. H. Witten, R. M. Neal, and J. G. Cleary, (1987) "Arithmetic coding for data compression," Commun. ACM, vol. 30, no. 6, pp. 520–540.

[14] C.E. Shannon, "A Mathematical Theory of Communication," Bell Syst. Tech. J.27 (July 1948), 398-403.

[15] Balaji Vaidyanathan, Yuan Xie, (2006) "Crosstalk-Aware Energy Efficient Encoding for Instruction Bus through Code Compression", In proc. of IEEE International SOC Conference. Pp193-196.

[16] Subash Chandar, Mahesh Mehendale & R. Govindarajan, (2006) "Area and Power Reduction of Embedded DSP Systems using Instruction Compression and Re-configurable Encoding", Journal of VLSI Signal Processing Systems, Volume 44, Issue 3, pp 245 – 267.