ALBORZ: Address Level Bus Power Optimization

Yazdan Aghaghiri
University of Southern California
yazdan@sahand.usc.edu

Farzan Fallah
Fujitsu Labs of America, Inc.
farzan@fla.fujitsu.com

Massoud Pedram
University of Southern California
pedram@ceng.usc.edu

Abstract

In this paper we introduce a new low power address bus encoding technique, and the resulting code, named ALBORZ. The ALBORZ code is constructed based on transition signaling the limited-weight codes and, with enhancements to make it adaptive and irredundant, results in up to 89% reduction in the instruction bus switching activity at the expense of a small area overhead.

1. Introduction

With the rapid increase in the complexity of chips and the popularity of hand held and mobile devices, power consumption has become one of the main design criteria. In a processor, a considerable amount of power is consumed in the highly capacitive and wide memory buses. Because the capacitance of external lines is usually several orders of magnitude higher than the capacitance of transistors, it is desirable to add some logic to the integrated circuits to encode the data before sending it over the bus, and in this way, decrease the switching activities on external buses. In this paper we use this approach to decrease the switching activity of an address bus. We introduce a new address bus encoding technique, and the resulting code, named ALBORZ. Our method is based on the limited weight encoding and transition signaling. We show how ALBORZ can be made adaptive to make the required encoder/decoder hardware smaller and achieve higher reduction in the switching activities. We also present an irredundant ALBORZ code that is very attractive in cases that adding an extra bit to a bus is not possible.

The rest of this paper is organized as follows. In section 2 we will look at the previous works. Section 3 explains the main idea behind the ALBORZ code. In section 4, we present the vanilla ALBORZ code and its fixed and adaptive versions. An irredundant version of the ALBORZ code is described in section 5, while the experimental results are presented in section 6. The conclusion is presented in section 7.

2. Previous Work

In this section we will review similar works in bus encoding and compare various encoding techniques. The following notation will be used throughout this paper,

b(t): Address value to be sent on the bus at time *t* (sourceword)

B(t): Encoded value on the bus lines at time *t* (codeword)

D: Address offset (b(t)-b(t-1))

In [1] Stan and Burleson proposed the *Bus-Invert* method as explained next. Consider an *N*-bit (non-multiplexed) bus. The idea is that if the hamming distance between two consecutive patterns is larger than *N/2*, then the second pattern can be inverted so as to reduce the inter-pattern Hamming distance to below *N/2*. One redundant bit is needed to distinguish between the original and inverted patterns that are transmitted on the bus. The Bus-Invert method tends to perform well when transmitting random patterns, which is often the case on data busses. However, this method is largely ineffective on address buses, which tend to exhibit some degree of sequentiality.

In [2] Benini et al. proposed *T0* code, which exploits data sequentiality to reduce the switching activity on the address bus. The observation is that addresses are sequential except when control flow instructions are encountered or exceptions occur. T0 adds a redundant bus line, called *INC*. If the addresses are sequential, the sender freezes the value on the bus and sets the *INC* line. Otherwise, *INC* is de-asserted and the original address is sent. On average 60% reduction in address bus switching activity is achieved by T0 coding [2].

Ikeda et al. [3] proposed using codebooks in sender and receiver. For every address, the code with minimum hamming distance to it is found in the codebook and then the selected code identifier along with the hamming distance between the address and the selected code is sent over the bus. They extended their method by using adaptive codebook in [4]. Thus the codes in the codebook can be replaced. As the program executes, only those codes will remain in the codebook that the program is accessing addresses around them.

There is another class of encoding techniques that avoids the use of redundant bits. These techniques exploit the decorrelating characteristic of the Exclusive-Or function as follows. Since when using Exclusive-Or, the codewords are transition-signaled over the bus, in every position where there is a 1 in the codeword, the bus will toggle and a switching will occur. This will convert the original problem to that of finding the codewords with the smallest average number of 1's in. The most efficient one of these codes is *T0-XOR*, which was proposed in [5] by Fornaciari et al. The encoder works as follows:

$$B(t) = b(t) \oplus (b(t-1) + S) \oplus B(t-1)$$

T0-XOR reduces the switching activity of instruction address bus up to 74% [5]. It can be easily seen that when the addresses are sequential, no switching activity occurs (similar to the case of T0 code). In the same work, the authors proposed another encoding technique, which is called *Offset-XOR* code. The encoder works as follows:

$$B(t) = (b(t) - b(t-1)) \oplus B(t-1)$$

In [7] the authors have proposed a coding framework that is used for analyzing different bus encoding approaches. Most of the previous methods are included in their proposed framework.

3. Approach

ALBORZ code exploits the optimality of limited-weight-codes when they are used with transition signaling. Consider a streamlined program code trace (i.e., there is no control transfer instruction inside the trace). In this case it is possible to calculate the current address by simply incrementing the previous address. In practice, however, one out of every seven instructions is a control transfer instruction [8]. Hence, it is not always possible to calculate the current address by incrementing the previous one.

Consider Offset-XOR, we compute the offset (i.e. the current address minus the previous address). We then calculate XOR of the offset with the previous value on the bus and send the result over the bus. This reduces the switching activity, because offsets are typically small numbers.

In ALBORZ code, to further decrease the switching activity on the address bus, we change the encoding of the offset. For example, if there is a branch instruction whose offset is `ECh`, we encode it to a value with fewer number of 1's, say `04h`. This results in a 5-1=4 unit decrease in the switching activity on the bus.

It is worth noting that offsets are typically small numbers. As seen

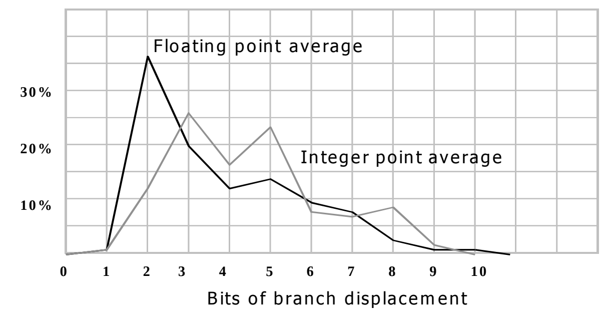


Figure 1- Occurrence frequencies for control transfer instructions in terms of their offsets

in Figure 1, more than 95% of the offsets can be represented with less than 10 bits [8]. This suggests that a large reduction in the switching activity can be achieved by concentrating on small offsets. In the next sections, we show different forms of realizing the ideas explained above.

4. ALBORZ Code

The ALBORZ encoder employs a *codebook* to encode offsets. Each offset is looked up in this codebook. If the offset is present in the codebook (i.e., there is a *hit*), then it is encoded based on the value stored in the codebook; otherwise, no encoding takes place (i.e., there is a *miss*) and the original address is sent over the bus. To distinguish between the two cases when the actual addresses or the encoded offsets are sent over the bus, an extra bit is added to the bus. We call the extra bit `CODEON`. It is desirable to maximize the hit ratio in the codebook. This means that the codebook must include the small offsets since they tend to occur frequently.

When an offset is found in the codebook, a limited-weight-code (LWC) associated with that offset is extracted from the codebook. In other words, the offset is mapped to a LWC. Instead of sending the offset, the corresponding LWC is XOR'ed with the previous value on the bus. Additionally, the `CODEON` bit is set to one. On the decoder side, if the `CODEON` is zero, the value on the bus is used as the address value. Otherwise, XORing the current bus value with the previous value produces the LWC for the offset. Next, the LWC is looked up in the decoder codebook and the corresponding offset is used to calculate the new address. Figures 2 and 3 show the ALBORZ encoder and decoder block diagrams, respectively.

Every entry of the encoder codebook consists of two fields:

D	LWC

The following terminology and notation are used in the remainder of this section:

`LWC(d)`: `Limited weight code of the entry with D = d`

`CODES`: `set of limited weight codes in the codebook`

`OFFSET(lwc)`: `Offset of the entry with LWC = lwc`

`OFFSETS`: `set of offsets in the codebook.`

ALBORZ encoder is formally described as follows:

```
if ( d ∈ OFFSETS )
        B(t) = B(t-1) ⊕ LWC(d)
        set CODEON
else
        B(t) = b(t)
        Reset CODEON
```

The ALBORZ decoder is described as:

```
if ( CODEON )
        lwc = B(t) ⊕ B(t-1)
        b(t) = b(t-1)  + OFFSET(lwc)
else
        b(t) = B(t)
```

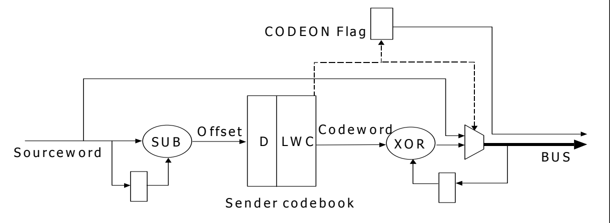


Figure 2- ALBORZ Encoder

The offsets that are mapped to LWCs can be either fixed or they can change at runtime. If the offsets are fixed, then the codebook is called a *fixed codebook.* Otherwise, the codebook entries can be updated during execution of programs; in this case, the codebook is called an *adaptive codebook.* We explain these two cases in more detail below.

Fixed Codebook

In this case the mapping between offsets and limited-weight-codes does not change, and the codebook can be implemented using a ROM or using some combinational logic. Consider a 32-bit address bus. To avoid any transitions when two consecutive addresses are sequential (i.e., assuming that the difference between two sequential addresses is one), the first entry of the codebook has to be as follows:

+1	**00000000h**

For a 32-bit bus, there exist 32 1-limited-weight-codes. An *N*-limited-weight-code or N-LWC is a code with exactly *N* 1's in it. Thus, LWCs of the next 32 entries of the codebook have exactly one 1. The offsets of these entries have to be the most frequently encountered offsets in any program. Therefore, these entries have to be used for small negative and positive offsets. The following table shows the mapping for offsets +2 to +17 and –1 to –16.

+2	**00000001h**
…	**…**
+17	**00008000h**
-1	**00010000h**
…	**…**
-16	**80000000h**

The number of 2-limited-weight-codes is 496 and the same method as that described above is used to map offsets +18 to +265 and -17 to –264.

It is possible to add more entries to the codebook, but this will substantially increase the area penalty; it may also increase the power consumption. The maximum size of the codebook that we can use depends on the number of required transistors, the ratio of switched capacitance inside the codebook to the switched capacitance on the bus, and the ratio of internal (codebook logic)

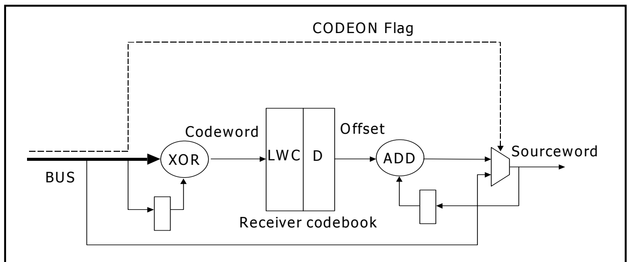


Figure 3- ALBORZ Decoder

to external (bus driver) power supply voltages. With the trend to scale down the transistor feature sizes and power supply voltage levels, the maximum size of codebook can be increased. Therefore, higher reduction in the switched capacitance of the overall bus subsystem (including the encoder/decoder logic overheads) can be achieved.

It is possible to design a fixed codebook in a way that the offsets are used to index the codebook entries. Consider the previous table, which included positive offsets from +1 to +17. If +16 and +17 are eliminated from the table, the four LSB bits of the positive offsets can be used to index the codebook while the MSB bits are zero. This will significantly reduce the codebook's hardware cost.

Adaptive Codebook

In the adaptive codebook, the offset column is implemented using a read/write memory. When an offset lookup takes place, if it is present in the codebook, the corresponding limited-weight-code is read. If a miss occurs, the actual address is sent over the bus and its offset replaces one of the offsets in the codebook. This guarantees that the next time the offset is looked up it will be present in the codebook (assuming that it is not replaced by subsequently-generated offsets).

In this way offsets that are most commonly encountered in a program are gradually loaded into the codebook; this will increase the hit rate and will result in a higher reduction in the bus switching activity. Notice that the same policy has to be used to update codebooks of the encoder and the decoder to ensure the coherence. Every time the CODEON bit is reset, the decoder realizes that a replacement has occurred in the encoder's codebook. Thus the decoder updates its codebook. Since the decoder and the encoder follow the same *eviction policy*, the same entries will be replaced in both codebooks with the new offset. This guarantees that the same offsets exist in both codebooks.

By using an adaptive codebook instead of a fixed codebook, the number of entries in the codebook can be significantly reduced while maintaining the same level of switching activity savings on the bus. However to determine if an offset is present in the codebook, it has to be compared to all offsets in the codebook. This kind of fully associative comparison is usually costly from hardware and power consumption viewpoints. To simplify the codebook hardware, the replacement (or eviction) policy can be changed to use direct or set associative mappings. With these policies each offset can be placed in certain entries of the codebook; therefore, to identify a hit, only a small number of the entries have to be searched.

5. Irredundant ALBORZ Code

Fixed and adaptive ALBORZ are both redundant codes. In other words, they need an extra bit to signal that the encoding has been performed on the current bus value. Adding one extra line to a bus requires modifications in pin-outs of both memories and processors. It will also require changes in the printed circuit boards. In practice, these modifications are not allowed in many systems.

It is possible to alter the ALBORZ code to suppress the redundant bit. Recall that the CODEON bit is required because the encoder has to inform the decoder if the actual address or its encoded version is sent. We modify our coding scheme so the actual address is never sent over the bus; instead the offset or its encoded version is XOR'ed with the previous value of the bus. When there is a hit in the codebook, we XOR the output of the codebook. When there is no hit, we XOR the offset itself. Ambiguity occurs when the offset itself is a LWC. Changing our replacement policy to prevent the LWCs from entering the codebook can solve this problem. After this modification in the replacement policy if the decoder receives a LWC, it knows that there should be a hit in the table. This guarantees that the offsets that are equal to one of the LWCs present in the codebook will always miss. However we still need to find a way to transmit these offsets without any ambiguity. Our solution is to map these LWCs to the offsets that are present in the first column of the codebook. The elimination of the

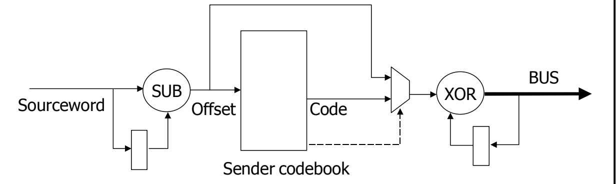


Figure 4 - Irredundant ALBORZ Encoder

`CODEON` bit comes at the expense of a marginally more complex encoding logic. Figure 4 illustrates the block diagram of the irredundant ALBORZ encoder.

The codebook for irredundant ALBORZ is slightly different from that of the redundant code as explained next. As before each entry of the codebook consists of an offset associated with a LWC. The main difference is that there is no offset equal to any of the LWCs in the codebook. In other words, there is no number in common between the OFFSETS column and CODES column. When an offset is looked up in the codebook, it is compared with both the OFFSETS and the CODES; if there is a match in the OFFSETS column, the corresponding LWC will be output. Otherwise, if there is a match in the CODES column, the offset corresponding to the LWC is used as the codebook output. The output is then XOR'ed with the previous value on the bus. If the offset is not equal to any of the OFFSETS or CODES, then a miss occurs and the offset itself is XOR'ed with the previous value on the bus.

As an example, consider a single-entry codebook with the following values,

+18	00000040h

If the offset is `+18`, then `00000040h` or `+64` is XOR'ed with the previous value of the bus. If the offset is `+64` then `+18` is XOR'ed with the bus. If the offset is neither `+64` nor `+18`, say it is `+11`, the offset, i.e., `+11`, is XOR'ed with the previous value on the bus.

The codebook effectively implements a one-to-one mapping between the address offsets and the LWCs. This can result in a reduction in switching activity as long as the probability of having a match in the offset column is higher than that of having a match in the limited-weight-code column.

The Irredundant ALBORZ encoder is formally described as follows:

```
if ( D ∈ OFFSETS )
        B(t) = B(t-1) ⊕ LWC(D)
else if ( D ∈ CODES )
        B(t) = B(t-1) ⊕ OFFSET(D)
else
        B(t) = B(t-1) ⊕ D
```

Note that (OFFSETS $\cap$ CODES = $\emptyset$), otherwise it may not be possible to correctly decode the address in the decoder.

The Irredundant ALBORZ decoder is described as follows:

```
D = B(t) ⊕ B(t-1)
if ( D ∈ CODES )
        b(t) = b(t-1) + OFFSET(D)
else if ( D ∈ OFFSETS )
        b(t) = b(t-1) + LWC(D)
else
        b(t) = b(t-1) + D
```

As before, if a miss occurs and the codebook is adaptive, one of the codebook entries may be replaced with the new offset. To further improve the performance of the Irredundant ALBORZ we can do a simple optimization on the displacements. In Irredundant ALBORZ if a miss happens the offset itself is transition-signaled over the bus. In many cases this offset can be a small negative number. Small negative numbers tend to have many 1's in their binary representation. Thus, they cause many bits to switch each time they miss the codebook. By means of the following simple function we map small negative numbers to large negative numbers and vise versa and avoid the above problem. We invert all bits of negative numbers except their MSBs, and positive numbers unchanged. We call this function LSB-Inv. Following table shows LSB-Inv(x) for a few sample numbers.

Original offset	Modified offset
FFFFFFFF, (-1)	80000000
FFFFFFFE, (-2)	80000001
FFFFFFF5, (-10)	80000009
80000000	FFFFFFFF

6. Experimental Results

We have used the **simplescalar** architectural simulator [9] to evaluate our methods. We have generated instruction address traces for six of the SPEC 2000 benchmark programs. Each trace consists of 15,000,000 instructions and all the simulations have been done using the SPEC 2000 test input parameters. The used benchmark programs are **vortex**, **parser**, **equake**, **gcc**, **vpr** and **art**.

The Fixed codebook was implemented using ROMs or combinational logic. Based on the codebook size, a certain number of LSB bits of the offset is used to index into the codebook. Number of entries that can be used can vary based on the amount of extra hardware that can be tolerated which itself depends on to the technology, system specifications, etc. In Figure 5 we have compared the performance of this scheme for different number of

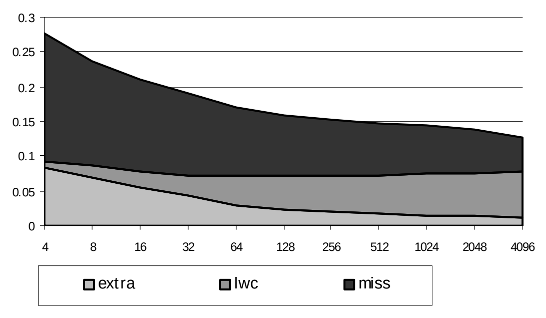


Figure 5- Total activity of the bus compared to the base case for Fixed-redundant-ALBORZ

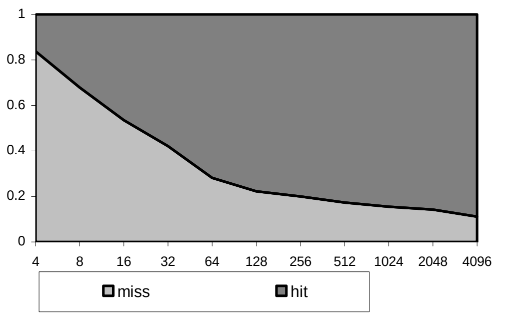


Figure 6- Ratio of codebook misses and hits

codebook entries. Vertical axis shows the average ratio of total activity of the bus to the original activity for the considered traces and horizontal axis shows the number of entries of the table. Switching activity has been partitioned into three segments. **Extra** denotes the activity of the redundant or extra bus line. **LWC** denotes the activity (of all lines except extra line) when there is a hit in the codebook and an LWC is transition signaled over the bus. **Miss** denotes the activity (of all lines except extra line) when there is a miss and the new address itself is put on the bus.

Figure 6 shows percentage of hits and miss. Codebooks in this example are implemented so that half of the entries are filled with positive offsets and the other half by negative ones. The improvement of the technique for over 256 entries is marginal. To reduce the hardware overhead and reduce the size of the codebook significantly, the following technique is used in hardware design. First LSB-Inv(offset) is calculated. After this mapping, all bits except MSB are used to lookup into the codebook. Thus, for example, 2 and –3 will hit in the same row of the codebook cause they only differ in MSB. After the code is looked up, then if the

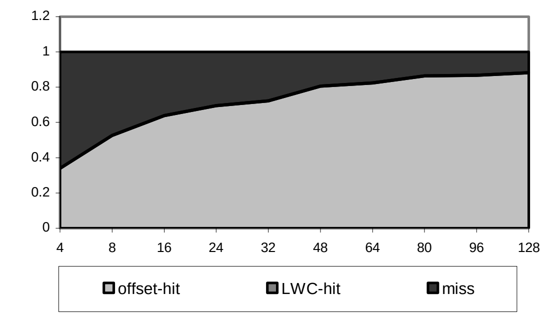


Figure 7- Total activity compared to the base case for Adaptive-Redundant

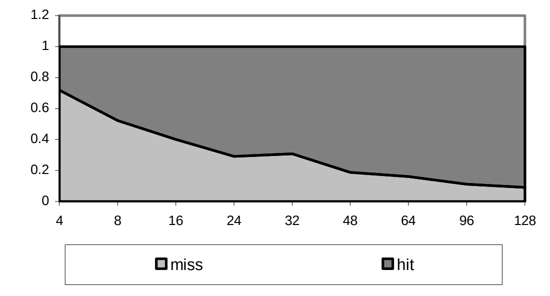


Figure 8- Ratio of codebook misses and hits

MSB is equal to one, the code is flipped. Suppose 00000100 is extracted for both 2 and –3. This code is used when the original number is 2 but for –3 the code will be flipped to generate a new code or 00100000. By this easy technique half of the codebook entries can be eliminated which will reduce the hardware overhead significantly.

For the Redundant Adaptive ALBORZ similar figures (Figure 7 and figure 8) have been drawn. The numbers of entries are much less, since the entries are read/write memory and thus they tend to consume more energy. In our implementation, the offsets are replaced using direct mapping scheme. An offset equal to one, i.e. sequential instructions should be mapped to all zeros as their output code, causing no transition. This special entry is designed in a fashion so that it cannot be replaced. A saturating counter can be added to each entry, this can be used to prevent entries with substantial hits get evicted. In our implementation we used a single-bit saturating counter. If one entry has more than one hit, this bit is set. On the other hand when a new offset tries to replace an entry, this bit is checked and if it is equal to one, is reset. Otherwise, the entry is replaced.

Irredundant Adaptive ALBORZ was evaluated for different number of entries (figure 9 and figure 10). Again the number of entries should be much less than the fixed version. Switching activity can have three different sources, the first one is when there is a hit in the offset column (**Offset-hit**) and an LWC is transition signaled over the bus, we call it **LWC-transition**. Second one is when there is a hit in the LWC Column (**LWC-hit**) and the output is a branch offset. We call this **Offset-transition**. And the last one is when there is no hit in either of the columns (**Miss**) and the offset itself is transition signaled over the bus, we call this **Miss-transition**. As it was mentioned earlier, in this scheme, the LWCs are fixed and the branch offsets can be replaced. Since each generated offset is compared with both columns the way the LWCs are put in the second column has a great effect on the overall performance. Because we map LWCs to offsets, every time there is a hit in LWC column, many bits may switch. In a program, the number of small branches is typically large. This suggests eliminating small LWCs from the table will increase the overall performance. By a simple analysis of our traces we decided to arrange the second column as follows. Among the 1-LWCs the first four are omitted, i.e 0x1, 0x2, 0x4 and 0x8 and the remaining 28 1-LWCs fill the first 28 entries.

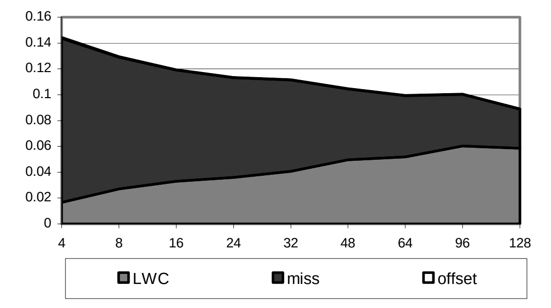


Figure 9- Total activity compared to the base case for Adaptive-Irredundant

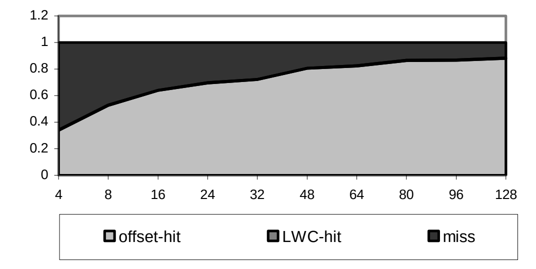


Figure 10– Ratio of codebook misses and hits

The rest of the entries are filled with 2-LWCs sorted in decreasing order. There are many large numbers among the total 496 2-LWCs, which are very unlikely to be offsets. With this arrangement of the codebook, offsets rarely cause any hit in the LWC column as one can see in figure 10.

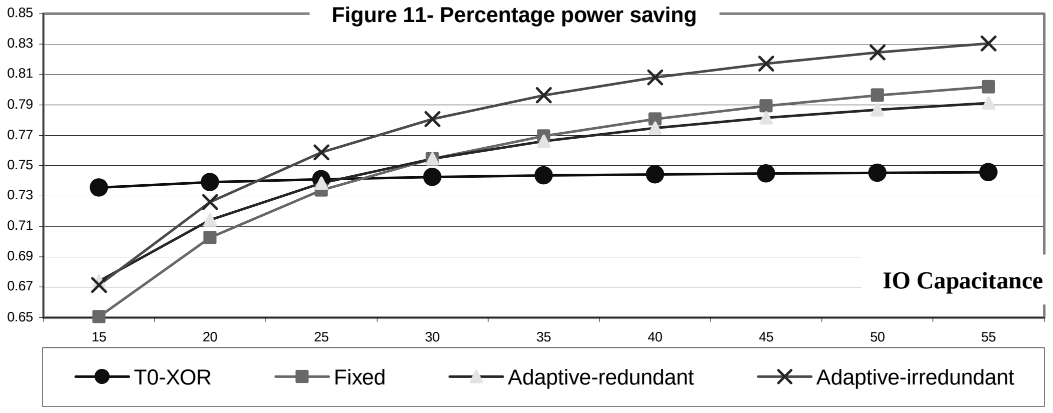
Next, we analyzed the power dissipation overhead of the encoder/decoder logic. Each of the above Encoders/Decoders was designed and the netlist of the encoder/decoder circuit was written in *Berkeley Logic Interchange Format* (BLIF). The netlists were optimized using the SIS *script.rugged* and mapped to a 1.5-volt, 0.18-micron CMOS library using the SIS technology mapper. Instruction addresses of the benchmark programs were then fed into a gate-level logic simulation program named *sim-power* to estimate the power consumption of the encoders. The power calculation was done in 100 MHz. The results are summarized in table 1. Figure 11 shows the power saving which can be achieved by using ALBORZ in comparison to T0-Xor method when the IO supply voltage is 3.3v for different values of external capacitance C. As one can see for large values of C our method outperforms T0-XOR method.

7. Conclusion

In this paper we presented several bus encoding techniques to decrease switching activity on an instruction address bus of a processor. Our techniques use limited weight encoding with transition signaling to reduce switching activity. Experimental results of running SPEC2000 programs show that our techniques can achieve up to 89% reduction in switching activity. Our experiments show that our encoders need about 800 gates and consume around 2mW.

Using some techniques like shutting down parts of the circuit or using transmission gates to implement multiplexors can reduce power dissipation of the encoders. This will make our method more attractive even when the bus capacitance is low.

Figure 11- Percentage power saving



	T0-XOR	Fixed ALBORZ 512-Entry Redundant	Adaptive ALBORZ 32-Entry Redundant	Adaptive ALBORZ 32-Entry Irredundant
Number of literals	**440**	**1750**	**1615**	**2146**
Area of Encoder	**334.82**	**766.67**	**817.64**	**797.53**
Number of gates	**306**	**818**	**870**	**827**
Power dissipated by encoder (mW)	**0.13**	**2.32**	**1.65**	**2.18**

Table 1- Comparison of different encoders

8. References

1. M. R. Stan, W. P. Burleson, "Bus-Invert Coding for low-Power I/O," IEEE Transactions on Very Large Scale Integration Systems, Vol.3, No. 1, pp. 49-58, Mar. 1995.

2. L. Benini, G. De Micheli, E. Macii, D. Sciuto, C. Silvano, "Asymptotic Zero-Transition Activity Encoding for Address Buses in Low-Power Microprocessor-Based Systems," GLS-VLSI-97: IEEE 7 th Great Lakes Symposium on VLSI, Urbana, IL, pp. 77-82, Mar. 1997.

3. M. Ikeda, K. Asada, "Bus Data Coding with Zero Suppression for Low Power Chip Interfaces", International Workshop on Logic and Architecture Synthesis, pp.267-274, Dec. 1996.

4. Komatsu, M. Ikeda, K. Asada, " Low Power Chip Interface based on Bus Data Encoding with Adaptive Code-book Method", Ninth Great Lakes Symposium, pp368-371, 1999.

5. W. Fornaciari, M. Polentarutti, D.Sciuto, and C. Silvano, "Power Optimization of System-Level Address Buses Based on Software Profiling," CODES 2000, San Diego,CA, pp. 29-33, 2000.

6. M. R. Stan, W. P. Burleson, "Low power encodings for global communication in CMOS VLSI," IEEE Transactions on Very Large Scale Integration Systems, Vol.5, No. 4, pp. 444-455, Dec. 1997.

7. S. Ramprasad, N. Shanbhag, I. N. Hajj, " A Coding Framework for Low-Power Address and Data Busses", IEEE transactions on Very Large Scale Integration Systems, Vol 7, No. 2, June 1999

8. *Computer Architecture, A Quantitative Approach*, Henessey and Patterson, 1996.

9. www.simplescalar.org