Software-Only Bus Encoding Techniques for an Embedded System

Wei-Chung Cheng, Jian-Lin Liang and Massoud Pedram
Dept. of EE-Systems
University of Southern California
Los Angeles, CA 90089

Abstract

Microprocessors with built-in Liquid Crystal Device (LCD) controllers and equipped with Flash memory are common in mobile computing applications. In the first part of the paper, a software-only encoding technique is proposed to reduce the power consumption of the processor-memory bus when displaying an image on the LCD. Based on the translation mechanism of the LCD controller, our approach is to start with the palette as a coding table for the pixel buffer and then reassign the codes according to the image characteristics. Experimental results prove the efficacy of this approach; power reduction reaches 29% for text-based and 17% for graphics-based images. In the second part of the paper, another software-only encoding technique is presented to reduce the transitions on the processor-CompactFlash bus. The device driver in Linux operating system is modified to perform Bus-Invert encoding when the data is read from or written to a Compact Flash file system. With minimal software overhead, the transitions on the bus are reduced by up to 25%.

1. Introduction

Mobile computing has evolved as a potent and influential cultural phenomenon. Portable devices such as Personal Digital Assistants (PDA), cellular phones, and GPS navigators are indispensable components of today's high technology society. Because computing power is growing and product size is shrinking, power consumption in microelectronic circuitry has become a critical concern because high degrees of power consumption severely limit product usefulness.

For these kinds of applications, semiconductor vendors offer highly integrated "system-on-chip" (SOC) solutions [1][2][3][4]. These systems integrate a Reduced Instruction Set Computer (RISC) microprocessor with many of the essential peripheral controllers (e.g., memory controller, Direct Memory Access (DMA) controller, LCD controller, PCMCIA controller, etc.) on the same chip. A system and accelerates product introduction. Although these highly integrated micro-controller solutions are quite useful, they tend to restrict designers' ability to perform aggressive optimizations, including attempts to reduce the system power consumption. Most of the hardware-level power saving techniques, such as clock gating and dynamic voltage scaling, cannot be applied to these kinds of systems because of their fixed architecture and interface requirements.

Previous studies have proposed a number of low power techniques for the Active Matrix LCD (AMLCD) [7][8], but they are only applicable at the logic or gate levels and, therefore, are not useful to us. In addition, many low power bus-encoding techniques have been developed [9][10][11][12]. However, these techniques require hardware modification. The interest of this paper is in the application of system-level or software-level techniques (which are adjustable) to reduce power consumption on the data bus and the IO bus.

In this paper, two power-saving techniques for embedded systems that combine a micro-controller with an LCD and a Flash memory are presented. The target system is described in Section 2. In Section 3, a low-power bus encoding technique that uses the LCD frame buffer palette is presented. Section 4 introduces another encoding technique for reducing power consumed on a PCMCIA bus. Section 5 concludes this work.

2. Target System

The target portable system is built around the Intel StrongARM SA-1110/SA-1111 evaluation boards [13]. SA-1110 is a highly integrated micro-controller, including a CPU core, a memory controller, an LCD controller, a PCMCIA interface, and other peripheral controllers. One or more Dynamic RAM chips, used as the main memory, are also included on the board. The microprocessor is a standard off-the-shelf product and cannot be modified. The memory chips are also commodity parts with fixed standard interfaces.

The LCD controller needs a specific memory region to store the displayed image data. This memory region is known as the *frame buffer*, which is also the interface to the application programmer who wants to draw graphical objects on the LCD. Pixel information is stored in the *pixel buffer*. Depending on the required color depth, 1, 2, 3, or even more bytes are used to represent each pixel. For

This work is supported by DARPA PAC/C program under contract award number DAAB07-00-C-L516.

example, to provide a 32-bit color depth, each pixel needs 4 bytes of memory. Therefore, the size of the pixel buffer depends on the total number of pixels and the color depth to be displayed. The LCD controller reads the pixel information from the pixel buffer at an acceptable rate (based on the refresh rate) through the data bus (*D_bus* in Figure 1), interprets the pixel, and sends the video signals (via *L_bus* in Figure 1) to LCD.

The PCMCIA interface provides control functions for accessing external CompactFlash (CF) memory, which is used as non-volatile memory in the system. The data transfer between the micro-controller and the CF memory card takes place via the IO bus (*P_bus* in Figure 1).

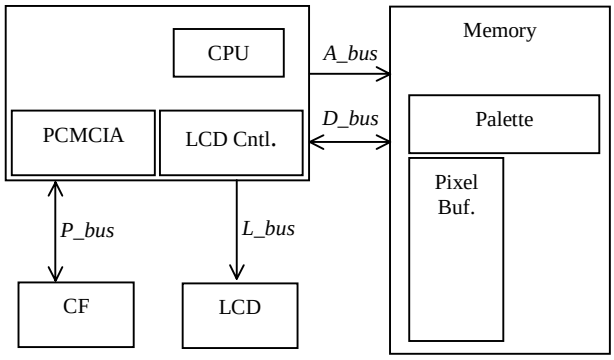

Figure 1. The target system

3. Encoding for the Memory Bus

3.1 LCD background

As stated earlier, the video information for images that are to be displayed on the LCD is stored in the frame buffer in a bitmapped manner. The video controller reads the information from the frame buffer and drives the LCD to display the corresponding image. Consider an LCD that is capable of displaying 320 by 240 pixels in a 24-bit color depth. If the refresh rate is 60Hz, the data rate between the frame buffer and the controller will be 320*240*24*60=111 M bits per second (bps)! This is an excessive amount of data. Thus, in practice, a *palette* is used to relieve the data traffic.

A palette is simply a one-dimensional index table, which is stored both on the main memory and the micro-controller. An entry in the palette with index *i* (*i=0,…, 255*) is initialized with c_i, which in turn represents a 24-bit color. The controller simply reads the color indices for pixels from the pixel buffer via the D_bus and decodes them by using its local copy of the palette. For a 256-color palette, the data rate between the memory and the controller is 320*240*8*60=37 M bps, which amounts to a 66% reduction in memory-controller traffic. The tradeoff, of course, is that in this scheme, out of 2^{24} colors, at most 256 different colors can be displayed on the LCD at any time. Furthermore, some memory is required on the controller side to store the palette. The color index decoding performed by the LCD controller provides the context for our software bus encoding approach.

During each LCD refresh cycle, power dissipation on the D_bus is proportional to the total bit-level activity on the bus when the color indices from the pixel buffer are sent to the LCD controller. When the LCD controller is in the burst mode, it sequentially fetches the index values stored in the pixel buffer. This data transfer causes the bit-level activity, which is in turn proportional to the expected Hamming distance between color indices of consecutive index values.

We can think of the palette as an encoding table that assigns an 8-bit color index to each of the 256 24-bit colors. By changing this encoding table and rewriting the pixel buffer values, we can reduce the activity on the D_bus by minimizing the expected Hamming distance between consecutive index values fetched on the bus. Note that the displayed image will not change as a result of this palette reordering and the corresponding pixel buffer rewriting

3.2 Problem formulation

In this section, we propose a technique for reducing the transitions on the memory data bus that are generated when the LCD controller reads the data in the pixel buffer. From the previous description of the palette, we can see that the palette and the pixel buffer can be reassigned to compose the same image while reducing the switching activity related to accessing the frame buffer.

Consider a palette containing 2^k colors $C=\{c_0, c_1 ... c_{2^k-1}\}$. Assuming that the data bus width is *k*, the data stream that is present on the data bus when reading the pixel data from the pixel buffer can be represented by a sequence of binary values: $X = x_0, x_1 ... x_{l-1}$, where $x_i \in C$ and *l* denotes the pixel count. The total switching activity of *X* is calculated as the sum of the Hamming distances between consecutive binary values x_i and x_{i+1} denoted as $H(x_i, x_{i+1})$. We attempt to find an optimal palette assignment (i.e., a color permutation) $\pi : C \to C$ such that the following objective function is minimized: $\sum_{i=0}^{l-1} H(\pi(x_i), \pi(x_{i+1}))$. We call this optimization problem the *Palette Assignment (PA) problem.*

If the bus width is *4k*, then the objective function becomes $\sum_{i=0}^{l-4} H(\pi(x_i), \pi(x_{i+4}))$. This is because four pixels are transmitted at the same time, therefore, the bit-level transitions occur between codes for pixels *i* and *i+4*. Formulating and solving the corresponding problem is straightforward. Details are omitted.

3.3 Palette encoding algorithm

Another way of stating the PA problem is to construct a state transition graph *G*, where each color c_i is represented

by a node n_i in the graph. There exists an edge e_{ij} in the graph if the two colors c_i and c_j are fetched on the bus consecutively. The weight of edge e_{ij}, which is denoted by w_{ij}, is equal to the transition probability tp_{ij} between c_i and c_j in the sequence of binary values X. With this terminology and notation, the PA problem can be restated as that of assigning index values (codes) to the colors so as to minimize $\sum_{i=0}^{2^k}\sum_{j>i} tp_{ij}.w_{ij}$. Notice that this problem is identical to the problem encountered during low-power state assignment [14][15]. This problem is known to be NP-hard. It has been solved using a simulated annealing (SA) algorithm. The solution quality can be very good, but the excessive runtime of an SA-based algorithm can be a concern. In this particular case, however, the SA algorithm is run only once per image to create the color-mapped image that will be stored in memory. At runtime, the LCD controller simply fetches the pixel color indices from the previously processed and optimized image file. This scheme works for image files that are statically generated.

Given a color, its brightness (for example, as the average of the R, G, and B values in an RGB color space) is calculated. Next, the 256 colors that are present in the palette are sorted by their brightness and use Gray code [16] to encode the sorted list of colors. For example, brightness levels 0, 1, 2, and 3 are assigned the Gray code sequence 0, 1, 3, and 2. This heuristic is based on the observation that the color brightness is likely to change continuously because of the reflective surface of subjects such as the human skin. This method works very well for black and white images where the only relevant distance metric in the color space is the brightness.

For color images, a more general distance metric can be defined based on the chrominance and luminance values and used for sorting the palette colors.

3.4 Experimental results

The images displayed in an embedded system are usually from two sources: (1) the graphics user interface (GUI), such as a window system, or (2) image files. For example, a navigation system displays map images, which are pre-produced and then downloaded. The proposed approach is to modify the GUI or the image files in advance. The following heuristic algorithm is used to solve the PA problem.

Figure 2 shows the histograms, which provide a plot of the occurrence frequency vs. brightness for two images ("win1" from Figure 3 and "elaine" from Figure 4). "win1" is generated by a GUI and exhibits a sparse distribution, whereas "elaine" is from an image file and exhibits a continuous distribution function. Based on this, we classify the images into "text mode" and "graphics mode."

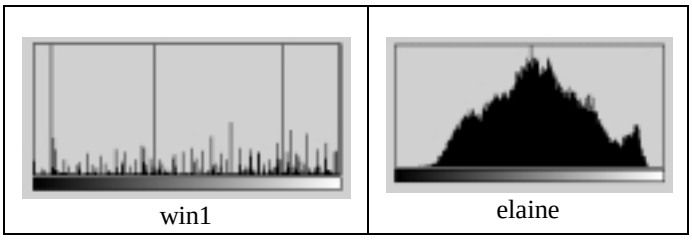

Figure 2. Text mode vs. graphics mode histogram

Based on the techniques described in Section 3.3, a software tool called *Palladia* (*Pal*ette *A*ssignment *Dia*gnostician) was developed to achieve the desired frame buffer encoding. The program implements two options: a Simulated Annealing (SA) based algorithm and a Palette-Sorting (PS) based heuristic.

The reduction in transition counts on the D_bus for two GUI images depicted in Figure 3 is shown in Table 1. The SA-based and heuristic algorithms reduce the switching activity by 15% to 29%. We can see that the heuristic algorithm is quite effective in spite of its much faster runtime and lower computational requirement.

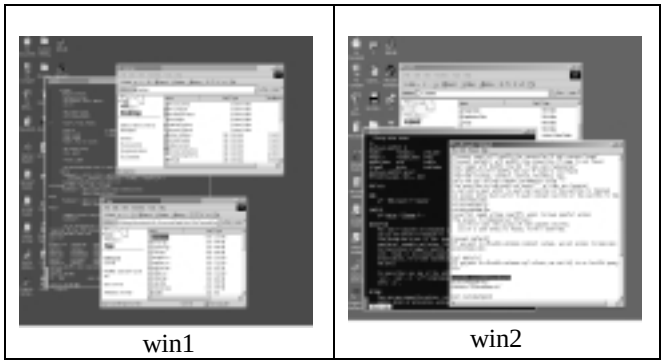

Figure 3. GUI benchmark images (a) win1 and (b) win2

Table 1. Results for SA-based and PS-based heuristic algorithms applied to win1 and win2

Image	Transition Count Saving %	
	SA-based Algorithm	PS-based Heuristic
win1	29.3	25.4
win2	17.9	15.7

Table 2 shows the test results of the heuristic algorithm running on the image files shown in Figure 4. Results for an 8-bit and a 32-bit D_bus are provided. Transition count savings are slightly higher for the 8-bit bus.

Figure 4. Photo/graphics benchmark images

Table 2. PS-based heuristic algorithm results for the photo/graphics images

Image	Transition Count Saving %	
	8-bit	32-bit
lena	7.7	6.6
elaine	17.6	15.8
pentagon	18.8	17.7
bellagio	12.9	6.6
paris	11.7	4.9
car	15.2	13.9
nyny	12.6	8.0
montecarlo	10.7	7.6
mgm	12.6	7.8

4. Encoding for the Compact Flash Bus

4.1 Compact Flash background

In this section, a software encoding technique for an IO bus is presented. The power dissipated on IO busses in embedded systems was drawn little attention previously because it was small compared to the power consumed by external devices (e.g., LCDs, hard drives, and network interface cards).

Mobile applications such as digital cameras, MP3 players, and PDAs require small, lightweight, and power-conserving devices for data storage. Therefore, solid-state storages such as Flash memory have emerged as an alternative to hard drives. Flash memory products are available to consumers in a card form such as a PC card [17] or a CompactFlash (CF) card [18]. Inside the card, are the Flash memory chips and a controller, which takes care of the PCMCIA/CF interfacing. An advanced process technology is used to fabricate these memory and controller chips. Compared to the internal power dissipation of the card, the IO bus tends to consume a lot of power, as is explained below.

Take CF as an example. Its form factor is quite small. The 50 signals of CF are compatible to (a subset of) the 68 PCMCIA signals. The latter is more mature and is well supported by mobile computers. As a result, most embedded computer systems support PCMCIA cards but not CF cards. For a computer that does not support CF, a CF-to-PCMCIA adaptor (defined in the CF specification) can be used. Consider the power consumption of the IO bus in this scenario: the voltage is fixed at 3.3V (to comply with the CF and PCMCIA standards), the capacitance is high (the wires inside the credit-card size PCMCIA adaptor card are physically long), and the frequency is high because of the speed of solid-state memory. Hence, the power dissipation on the bus can be quite high.

According to the CF specification [18], a CF-card has to support three operation modes: CF-Memory, CF-IO, and CF-ATA (*AT Attachment*). The first two modes define the card as a memory component attached to the PCMCIA bus. To operate the CF-card in these two modes, the OS has to be aware of this special memory device and control it with a specific device driver. For this reason, the implementation of the third mode, CF-ATA, is mandatory in order to provide backward compatibility. In this mode, the card emulates an Integrated Drive Electronics (IDE) hard drive, which is virtually standard equipment for every IBM-compatible PC. In the target system – as well as many other embedded systems – the CF is treated as a hard drive, and therefore the CF-ATA mode is in use.

The IDE interface protocol is defined in the ATA specification [20]. Briefly, it provides seven 8-bit command registers and one 16-bit data register. The IDE device is controlled by a program (i.e., the device driver) that writes different commands and parameters into the command registers. These commands include read, write, erase sector, format track, standby, idle, recalibrate, etc., while the parameters include the sector number and the sector count. In the read/write operation, after the command word is written into the command registers the actual data can then be read/written through the 16-bit data register. For example, to read sectors, first the sector number, sector count, and read command are written to the command registers and then one or more sectors (512k bytes per sector) are read from the data register. The controller inside the CF-card internally translates the sector addresses into memory addresses and performs the corresponding operations to the Flash memory chips. The translation is transparent to the software, so any computer without any additional software can use CF-cards as long as it supports

IDE devices. The software (i.e., the IDE device driver) can access the hard drives by using either DMA or Program IO (PIO). DMA relieves the CPU load while PIO is used for high-speed devices or when DMA is unavailable.

4.2 PCMCIA bus encoding

A software bus encoding technique used to reduce the transitions on the PCMCIA bus is presented here. The goal of this technique is to analyze the data (files) on the CF-card off-line. According to the analysis results, the data in the files are processed by a bus-invert algorithm [11] and then stored back to CF. The processing steps are as follows. The file is divided into blocks of data. Each block is analyzed to derive the "best" inversion pattern. The 16-bit wide inversion pattern is exclusive-or'ed with all the 16-bit wide words in the block, and the results are written back to the file. Of course, the inversion pattern for each block is also stored in the file so that performing another exclusive-or operation with the encoded value can restore the original value. When the file is loaded into the main memory, the device driver of the CF-card recovers the data by performing the necessary exclusive-or operation on the fly.

The idea is simple but different from a typical bus invert technique in the following ways. (1) The CPU performs the inversion without needing extra hardware. (2) The calculation of the inversion pattern per block of data is made at the compilation time by solving an optimization problem (which we call the inversion pattern selection problem or the IPS problem). (3) The redundant information is carried in the file header instead of being sent along with the data (i.e., using extra bus signals). (4) The inversion pattern is changed for each block of data to adapt to the specific characteristics of the block.

Implementation details. If the sectors are read into memory in the PIO mode, the data will stay in the data cache. For example, the StrongARM SA-1110 has an 8KB main data cache, which is large enough to cover a 4KB page in ARM Linux. The inversion operations can be performed efficiently at a maximum clock frequency of 206 MHz by a loop consisting of five instructions (load the word, conditional XOR, store the word, rotate the invert vector, and branch). In addition, the remaining unused components can be put into sleep mode during the inversion period to reduce the overhead.

For ARM Linux, we modified the IDE device driver (i.e., linux/drivers/ide/ide-disk.c) so that it inverts the data after (before) a sector is read from (written to) the CF-card. Notice that the encoding occurs during not only the regular file read/write operations but also the page in/out events when the CF-card is used as a swap device.

IPS problem formulation. The problem of finding the optimal inversion pattern (α) can be formulated as the following minimization problem:

$$\sum_{i=0}^{l} MIN\{|x_i \otimes x_{i+1}|, |x_i \otimes x_{i+1} \otimes \alpha|\}$$

where x_i denotes the codes for the words in a block of data. This problem, which is also known as the partial bus-invert problem in the literature, has been shown to be an intractable problem [19]. A number of heuristic algorithms have been proposed to solve the IPS problem. The goal is to reduce the power consumption of a mobile system in which the applications are pre-determined. Therefore, the optimization problem can be solved off-line without worrying about power dissipation overhead for solving this optimization problem. In practice, a branch-and-bound algorithm is used to find the best possible inversion pattern. Since the number of words in a block is rather small and the bus width is limited, this approach is practical.

Determining the block size. The standard bus-invert technique uses a fixed inversion pattern α, which is a pattern of all 1's. Better results can be achieved by using different patterns for different blocks of data (pages or sectors). A smaller block size increases power savings on the bus but also tends to increase the overhead in terms of the space for inversion pattern storage and the time spent on performing inversion during the decoding phase (the encoding phase is not of concern since it is done on a non-power-constrained processor).

Preserving data cache entries. In the encoding algorithm described above, it is necessary to perform an exclusive-or operation on all the data entries that are written to or read from the CF. Assume that the data cache is not flushed upon entering a system call and that DMA is used for data IO. The CF data coding will potentially flood the cache with block data and change the cache behavior. To solve this problem, we can disable (freeze) the cache prior to accessing the main memory to perform the exclusive-or operation. After the encoding/decoding process, cache is unfrozen and the normal computational processes will continue their execution on a cache that is kept "hot".

4.3 Experimental results

The SPEC95 integer benchmarks were used to generate the data traces. Consider a 16-bit PCMCIA bus. First, two software applications were used to examine the effect of smaller block sizes. Data is divided into blocks. In each block, the optimal inversion pattern was found. The numbers of transitions in different block sizes are reported in Table 3. As expected, the results improve monotonically as the block size becomes smaller, but the marginal improvement is not impressive. This suggests the use of a larger window size of 1024 or even 4096 bytes in order to minimize the storage space and computation time requirements. The 4KB size is more desirable for us because it matches the page size in Linux. Eight benchmarks with a page size of 4 KB were tried. Results are reported in Table 4.

Table 3. Results as a function of block sizes

Block Size	256	512	1024	4096
compress	81.5%	80.1%	78.9%	76.7%
ijpeg	87.6%	85.7%	85.1%	83.1%

Table 4. Results of comparing adaptive partial bus invert, partial bus invert w/ fixed inversion pattern, and full bus invert

	Length	Opt-4k	93e5	BI
compress	88k	0.82	0.90	0.94
gcc	1,286k	0.75	0.81	0.90
go	500k	0.71	0.75	0.87
ijpeg	167k	0.78	0.84	0.93
li	98k	0.75	0.79	0.91
m88ksim	177k	0.79	0.87	0.93
perl	286k	0.76	0.80	0.89
vortex	813k	0.69	0.74	0.88

The first column lists the data trace lengths for each of the benchmarks. The remaining three columns report the ratio of the bit-level transition count with the bus-invert encoding to the count without the encoding. Thus a smaller ratio reveals a larger power saving. In the second column, we used the adaptive partial bus-invert algorithm per a block size of 4KB. In the third column, "93e5," the fixed pattern 0x93e5 (hex) is used for all the files. This particular pattern was chosen because it happens to be the most common optimal-inversion pattern in all of the applications that were tried. The last column, BI, lists the original bus-invert results for reference.

Experimental results imply that with a globally fixed inversion pattern, the transitions on the IO bus can be reduced by between 10-26%. Because the pattern does not change from block to block, there is no need to store the inversion patterns. In other words, by a simple modification of the device driver, the power dissipation can be reduced with zero hardware cost and minimal software overhead (which is due to dynamic encoding and decoding of the data that is written to and read from the CF). Higher savings can be obtained by changing the inversion pattern per blocks of size 4096. This is a tradeoff that system developer needs to consider.

5. Conclusions

Two software approaches to reduce the power consumption of the memory and IO busses in an embedded system equipped with LCD and Flash memory were proposed. For the memory bus, the palette, which is the translation mechanism of the LCD controller, was used as a coding table and reassigned the palette and rewrote entries in the pixel buffer according to their correlation to the image. We formulated the problem as a state assignment problem and presented an SA-based and an efficient heuristic algorithm for solving it. Experimental results proved the efficiency of this approach: there is up to a 29% power reduction for the text-mode images and a 17% reduction for the graphics-mode images. For the PCMCIA bus, up to 26% of the transitions can be eliminated by a very simple modification of the OS kernel.

6. References

[1] Intel StrongARM SA-1110. http://developer.intel.com.

[2] Motorola M683XX DragonBall. http://www.motorola.com.

[3] Sharp 77790. http://www.sharpmeg.com/lh77/lh77.html.

[4] Hitachi H8/300L.http://semiconductor.hitachi.com/h8.html.

[5] Palm PalmPilot. http://www.palm.com/products.

[6] PocketPC. http://www.microsoft.com/pocketpc.

[7] J. S. Kim, D. K. Jeong, and G. Kim, "A multi-level multi-phase charge-recycling method for low-power AMLCD column drivers," *IEEE Journal of Solid-State Circuits*, Vol. 35.1, pp. 74–84, Jan. 2000.

[8] S. W. Lee, J. S. Kim, and C. H. Han, "Pixel arrangement for low-power dot inversion liquid crystal display panels," *Electronics Letters*, Vol. 34. 16, pp. 1604–1606, Aug. 1998.

[9] L. Benini, G. DeMicheli, E. Macii, M. Poncino, and S. Quer, "System-level power optimization of special purpose applications: The beach solution," *ISLPED-97: ACM/IEEE International Symposium on Low Power Electronics and Design*. Monterey, CA., pp. 24-29, Aug. 1997.

[10] E. Musoll, T. Lang, and J. Cortadella. "Exploiting the locality of memory references to reduce the address bus energy," *ISLPED-97: ACM/IEEE International Symposium on Low Power Electronics and Design*, Monterey, CA., pp. 202-207, Aug. 1997.

[11] M. R. Stan and W. P. Burleson, "Bus-invert coding for low-power I/O," *IEEE Transactions on VLSI Systems*, Vol. 3, No. 1, pp. 49-58, 1995.

[12] W. C. Cheng and M. Pedram, "Power-optimal encoding for DRAM address bus," *ISLPED-00: ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 250-252, 2000.

[13] Intel StrongARM processors SA-1110/SA-1111, Development Platform, http://developer.intel.com/.

[14] K. Roy and S. Prasad, "Syclop: Synthesis of CMOS logic for low power application," *Proceedings of the International Conference on Computer Design*, pp. 464-467, Oct. 1992.

[15] C. Y. Tsui, M. Pedram, and A. M. Despain, "Low power state assignment targeting two and multilevel logic implementations," IEEE Trans. on Computer Aided Design, Vol. 17. No. 12, pp. 1281-1291, Dec. 1998.

[16] C. L. Su, C. Y. Tsui, and A. M. Despain, "Saving power in the control path of embedded processors," IEEE Design and Test of Computers, Vol. 11, No. 4, pp. 24-30, 1994.

[17] PCMCIA, *PC Card Standard*, 1995. http://www.pc-card.com/.

[18] CompactFlash Association, *CF+ and CompactFlash Specification*, July 1999. http://www.compactflash.org

[19] Y. Shin, S. I. Chae, and K. Choi, "Partial bus-invert coding for power optimization of system level bus," *ISLPED-98: ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 127-129, 1998.

[20] American National Standards Institute, *AT Attachment Interface Document, X3.221-1994*. http://www.ansi.org.