Finding low activity op-code sets using genetic computing

Mohammad Dastjerdi-Mottaghi
Nanoelectronics Center of Excellence,
School of Electrical and Computer Engineering
University of Tehran, Tehran, Iran
Email:
m.mottaghi@ece.ut.ac.ir

Mohammad Riazati
Nanoelectronics Center of Excellence,
School of Electrical and Computer Engineering
University of Tehran, Tehran, Iran
Email:
m.riazati@ece.ut.ac.ir

Masoud Daneshtalab
Nanoelectronics Center of Excellence,
School of Electrical and Computer Engineering
University of Tehran, Tehran, Iran
Vali Asr Educational Institude,
Email:
m.daneshtalab@ece.ut.ac.ir

Zainalabedin Navabi
Departement of Electrical and Computer Engineering
Northeastern University, Boston, U.S.A.
Email:
navabi@ece.neu.edu

ABSTRACT

In this paper, we propose a genetic algorithm for finding the optimum op-code sequence for instruction set of a given processor. The sequence, which we look for, raises the least possible average signal transitions on the address bus of the given processor. The algorithm takes the probability of each instruction pair. Then randomly generates some op-code sequence as the initial population. Afterwards it iteratively uses some problem specific heuristics to generate a better population based upon the existing population and the table of pair probabilities, in this manner better and better populations are generated until (after about 200000 iterations) no better op-code sequence can be generated at which time the algorithm stops. Results, for MIPS-R4000, show that the proposed algorithm reduces the average switching activity of the address bus by 44%.

Keywords: Chromosome, Genetic Algorithm, High Class, Instruction coding, Low Class, Low power, Middle Class, Op-code sequence, Switching activity

I. INTRODUCTION

The increasing trend of today technology to portable and battery operated systems on the one hand and numerous problems posed by large and extensive circuitry in IC's, such as cooling on the other hand leads the designers to low-power circuits. Studies have shown that power dissipation in static CMOS circuits is directly related to their sa (switching activity) as the following equation also tells us the same fact:

$E_{avg} = sa * C * V^{2}$

A new technique for switching activity reduction in fetch unit of a generic processor is presented in this paper. What we are looking for is an optimum op-code sequence for instructions of a processor which results in the least possible switching activity (in the fetch unit) in the CPU while executing different programs.

One of the inevitable functions of a CPU is *fetch*. Fetch unit is the ever active unit of almost all CPU's. A dominant source of switching activity in this unit is the bit transitions caused by instruction words. Each instruction word is composed of an op-code (operational code) part plus some other parts. Minimizing the bit transitions caused by op-code part of instruction words can noticeably reduce the total switching activity of the fetch unit; this is the target of this paper.

For the rest of the paper we have introduced related works in section II. Section III talks about optimum instruction coding and the related topics. In section IV the details of the proposed algorithm are given. Results and summary are the topics of sections V and VI, respectively.

II. RELATED RESEARCHES

Bit transition reduction in data and address busses of processors has attracted the attention of many research groups recently. T0 method [2] proposes adding an extra line to address bus which signals the regularity (augmented one by one) of addresses. Whenever the cited line is '1' memory control circuitry simply adds 1 to the previous address and generates the new address. [3] achieves the same goal without using the extra line; in this technique by inspecting the address bus whenever no change is made to it, it is added by one, otherwise the new address is used. Locality of data and address references is used by [4] in which addressing is done through the use of *offsets* and *bases*. Although these techniques give efficacious solutions to activity reduction of address unit they do not reduce switching activities caused by instructions themselves. Unfortunately instructions do not follow any regularity. The encoding technique proposed in [5] uses self-organized lists to obtain an optimum encoding for accessing frequently used addresses. Introduced in [6] is a power reduction technique based on association of suitable op-codes to instructions which considers instruction op-codes and the movement among instructions as a state machine. In this state machine instructions are modeled to states and fetches are considered as state transitions. Then minimum average switching activity in the state machine is looked for using the proposed techniques in [7] and [8]. The corresponding state machine in [7] which works based on integral linear programming has too many states and consumes too much time to give the optimum op-code sequence and practically is not usable. [8] is similar to [7] except that using some heuristics tries to speed up the calculations. The technique is less accurate and due to too many sates of its state machine is practically useless.

III. OPTIMUM INSTRUCTION CODING

Associated to each instruction of a CPU is an op-code; this is what we mean by instruction coding.

A. HOW INSTRUCTION CODING AFFECTS ACTIVITIES IN FETCH UNIT

Frequencies of different instructions of a CPU are not identical; i.e. in a typical program, instructions such as *load*, *jump* and *move* are more frequent than instructions like *divide* and *multiply*. This fact helps us devise more efficacious op-code sets for instructions. Especially when some combinations of instructions are more probable to appear in a typical program, we can assign such op-codes to these instructions that the resulting switching activity becomes as few as possible. As an example consider the op-code sequence of Table I. As you see the most probable instruction pair in this table is (LD, ST); it means that in typical programs 11.64% (see Table I) of two consecutive instructions are *loads (LD)* followed by *stores (ST)*. Therefore instruction coding {LD=00, ST=01} is better than {LD=00, ST=11}; since the former causes just 1 bit transition whereas the latter causes 2 bit transitions.

Current Instruction	Next Instruction	Probability (P)	Cost (C)	C * P
LD(00)	LD(00)	0.095251	0	0
LD(00)	ST(01)	0.116411	1	0.116411
LD(00)	ADD(10)	0.00521	1	0.00521
LD(00)	SUB(11)	0.088735	2	0.17747
ST(01)	LD(00)	0.08958	1	0.08958
ST(01)	ST(01)	0.086719	0	0
ST(01)	ADD(10)	0.084454	2	0.168908
ST(01)	SUB(11)	0.042384	1	0.042384
ADD(10)	LD(00)	0.078568	1	0.078568
ADD(10)	ST(01)	0.039663	2	0.079326
ADD(10)	ADD(10)	0.050523	0	0
ADD(10)	SUB(11)	0.004492	1	0.004492
SUB(11)	LD(00)	0.036448	2	0.072896
SUB(11)	ST(01)	0.078474	1	0.078474
SUB(11)	ADD(10)	0.015107	1	0.015107
SUB(11)	SUB(11)	0.08798	0	0
AHD (Average Hamming Distance)				0.928826

Table I - AHD for an instruction set of 4 instructions and op-code sequence of {LD=00, ST=01, ADD=10, SUB=11}

Top

B. COST OF AN INSTRUCTION CODING

As implied in the previous section knowing the appearance probabilities of different combinations of instructions is necessary to associate an optimum op-code to each instruction. In this paper we use frequencies (appearance probabilities) of *pairs* of instructions. To obtain the frequencies of instruction pairs we have first compiled benchmark programs (including compressors, word processors and compilers) and then in the resulting assembly program we have counted all pairs of consecutive instructions and in this way we have calculated the frequency of instruction pairs (for MIPS-R4000 processor).

To each instruction pair we associate a *cost*. The Hamming distance between op-codes of each pair is considered as its cost. As you may know the Hamming distance between two bit patterns is the number of bits where the patterns do not have similar bits; for example the Hamming distance between 0**01**010 and 0**10**010 is 2 since there are 2 bits (bolded) in these patterns that are not the same.

The cost of an instruction coding is defined to be the weighted average Hamming distance (cost) of all pairs of op-codes. It is mathematically expressed by the following formula:

$$AHD = \sum_f \sum_s HD(I_f, I_s) \times P(I_f, I_s)$$

In the above formula, *HD* is the Hamming distance and *P* is the probability associated with each instruction pair. I_f and I_s are first and second instructions respectively.

Table I, shows the information used to calculate the cost of the following instruction coding:

```
ocSeq1 = {LD=00, ST=01, ADD=10, SUB=11}
```

In this table there are 16 pairs of instructions in conjunction with their frequencies and costs. To obtain AHD (average Hamming distance) of sequence *ocSeq1*, $C \times P$ is calculated for each pair. Then these values are added up to give AHD which is by definition the cost of *ocSeq1*.

C. ALGORITHM OUTLINE

Now we want to give you a general view of how the proposed algorithm works: It is an iterative optimization algorithm which looks for the optimum op-code sequence (instruction coding). In each iteration, using genetic computing, a new op-code sequence (population) is generated. Then the AHD (fitness) of each op-code sequence is computed and according to the obtained AHD's a better population is generated. This iteration is repeated until no better op-code sequence is found. In fact in this process, AHD is minimized and the op-code sequence with the least AHD is found.

The search space in this problem is very vast; for a processor with *i* instructions a total of about *i*! permutations exist among which one (or more) has the least AHD (i.e. what we are looking for). For a processor with 15 instructions the search space size is 1,307,674,368,000 while typical instruction count of today processors is about 80 or so.

1. **Start:** Generate random population of *n* chromosomes (*n* op-code sequences).
2. **Fitness:** Evaluate the fitness (AHD(*x*)) of each chromosome *x* in the population.
3. **New population:** Create a new population by repeating the following steps until the new population has the least AHD
   - 3.1. **Selection:** Select two parent chromosomes from the population
   - 3.2. **Crossover:** With a crossover probability cross over (recombine) the parents to form new offspring (op-code sequence). If no crossover was performed, offspring is the exact copy of parents.
   - 3.3. **Mutation:** With a mutation probability mutate new offspring at each locus (position in chromosome).
   - 3.4. **Accepting:** Place new offspring in the new population
4. **Replace:** Use newly generated population for a further run of the algorithm
5. **Test:** If the end condition is satisfied, **stop**, and return the best solution in current population
6. **Loop:** Go to step 2

Table II – The outline of the genetic algorithm used in our solution

As you see an ordinary search technique can not find the answer in a reasonable time period. Recently genetic algorithms are extensively deployed in such problems to find the optimum (or often sub-optimum) answer. Genetic computing can perform better since it benefits from problem specific heuristics to ignore some cases and move faster toward global optimum(s). For problems with vast search spaces, a well designed genetic algorithm is often (not always) faster and more successful than other search techniques, since it does not examine all cases.

IV. FINDING THE OPTIMUM OPCODE SEQUENCE USING GENETIC COMPUTING

We deployed genetic computing to solve the problem; in Table II the outline of the genetic algorithm which we have used in our solution is shown. The key factors which cause a genetic algorithm to converge to the answer are *encoding (representation)*, *selection*, *crossover* and *mutation.* These factors drastically affect the performance of a genetic algorithm and here we want to discuss each of them.

A. Encoding (Representation)

A genetic algorithm (GA) performs its search with some chromosomes a group of which is called *population*. This is a key question how a chromosome is related to the problem and how it is represented. In this problem we consider a chromosome to be an op-code sequence. Therefore a population is a group of op-code sequences based upon their fitness (AHD) a new generation (population) is created in each iteration. As said before in the problem of instruction coding we are given *n* instructions to each of which a *unique* op-code from interval [0, *n*) is to be associated. Hence we can represent a chromosome with a sequence of *n* numbers (op-codes) each of which corresponds to some instruction; in the jargon of genetic computing it is called *permutation* encoding. The output of our algorithm is the best (fittest) chromosome.

B. Selection

GA (genetic algorithm) is not different from random search unless we can make sure that a better generation is created from the existing generation. In creation of a better generation *crossover* and *selection* play very important roles. What differentiates GA from random search is *crossover* which is problem dependant and its efficacy is very much influenced by *selection*. Domain (problem) knowledge helps us devise better and more efficient schemes for *crossover* and *selection*.

Before crossing over two chromosomes we should first *select* them from the population. To do so after the fitness of each chromosome is evaluated we categorize the population into three categories of *High Class*, *Middle Class* and *Low Class* from each of which we select differently. The categorization is done based upon the difference between the fitness of the best and the worst chromosome. Specifically we form an interval, called the *fitness range*, the lower bound of which is the least AHD (the best fitness) and its upper bound is the greatest AHD. Then those chromosomes whose fitness values lie in the lower 10% of the fitness range are classified as *High Class* (good) chromosomes and those whose fitness values lie in the upper 10% of the fitness range, as *Low Class* and the rest are considered as *Middle Class* chromosomes.

We use *roulette wheel* to select from High Class chromosomes; i.e. the fitter the chromosome is the more probable it is to be selected. Middle Class chromosomes are *randomly* selected. But Low Class (bad) chromosomes are selected using *inverse roulette wheel*; i.e. the fitter the chromosome is the *less* it is probable to be selected. This is because we select chromosomes to generate better offspring; hence worse chromosomes should be bettered sooner.

C. Crossover

The objective of crossover is to use the results of previous searches (populations) to generate a better generation. Domain (problem) specific knowledge helps us devise efficient schemes to move toward a better population and get closer to the optimum answer. For the problem of optimum instruction coding we propose 3 crossover schemes for the different classes of chromosomes.

C.1. Low class chromosomes

In Figure 1 the proposed crossover scheme for low class chromosomes is shown. The idea behind the scheme is to find the genes which are responsible for the badness of the chromosomes and to correct these bad genes by replacing them with their peers in a good chromosome (step 5 in Figure 1). We define bad genes to be those which are common to two bad (low class) chromosomes.

As seen in Figure 1, two low class chromosomes are first selected; then the intersection of the two chromosomes is found. Then a good chromosome is selected from high class chromosomes. Finally the original (bad) chromosome is copied exactly to the offspring except for the common loci (positions in chromosome) the values of which are copied from the high class chromosome (step 5). In step 6 repeated genes, which make the chromosome (op-code sequence) invalid, are replaced with bad genes which were discarded in step 5.

C.2. Middle class chromosomes

In Figure 2 the proposed crossover scheme for middle class chromosomes is shown. The idea behind the scheme is to find the genes which are responsible for the goodness of two high class chromosomes and to inject them to a middle class chromosome (step 5 in Figure 2) hoping that this injection makes the middle class chromosome fitter. We define good genes to be those which are common to two good (high class) chromosomes.

As seen in Figure 2, two high class chromosomes are first selected; then the intersection of the two chromosomes is found. Then a middle class chromosome is selected and is copied exactly to the offspring (step 4). In step 5 good genes (common to good chromosomes) are injected into the offspring. Validation is done in step 6 where repeated genes, which make the offspring (op-code sequence) invalid, are replaced with genes which were discarded in step 5.

C.3. High class chromosomes

In Figure 3 the proposed crossover scheme for high class chromosomes is shown. The idea behind the scheme is to associate distant op-code pairs (with greater Hamming distances) to less frequent instruction pairs, hoping that distant pairs have less chance to raise signal transitions on the address bus and final AHD becomes less.

As seen in Figure 3, a high class chromosome is first selected and copied exactly to the offspring (step 2). Then using roulette wheel a *rare* instruction pair (with least possible frequency) is selected from the corresponding table; this is step 3 in Figure 3 and the selection is such that pairs with lower frequencies are more likely to be selected. In the figure pair (=, <) is selected which has the frequency of 0. In step 4 roulette wheel is used to select a distant op-code pair (i.e. the more distant the more likely to be selected). Now the selected distant pair is associated to the rare instruction pair selected in step 3. Finally validation is performed in step 6 which removes repeated genes (op-codes).

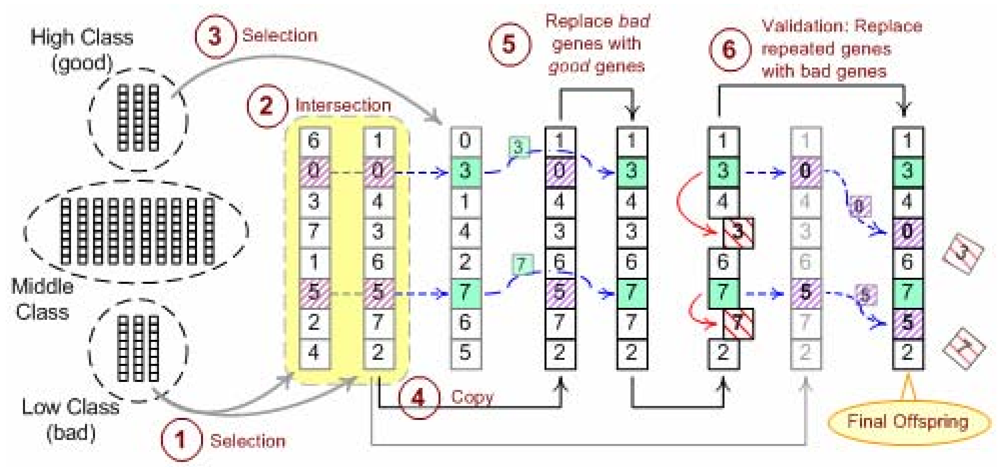

Figure 1- The proposed crossover scheme for *low class* (bad) chromosomes – The final offspring which is placed in the next population is the op-code sequence of (1, 3, 4, 0, 6, 7, 5, 2)

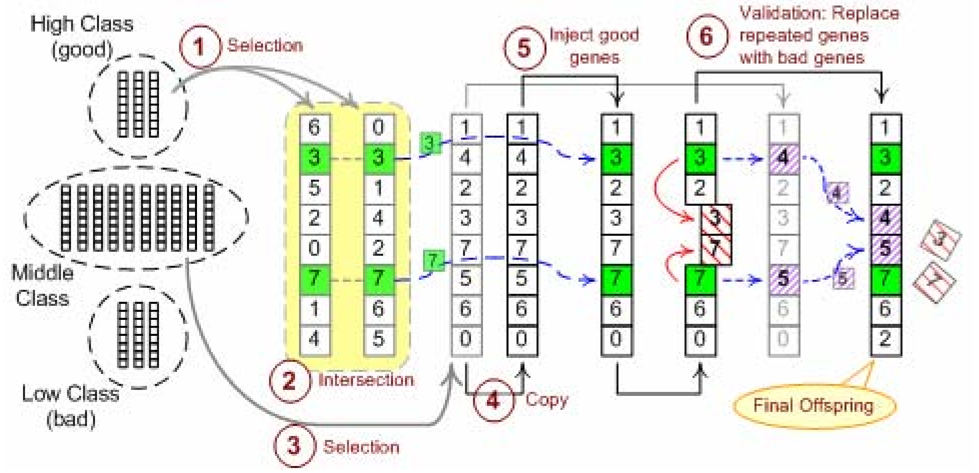

Figure 2- The proposed crossover scheme for *middle class* chromosomes – The final offspring which is placed in the next population is the op-code sequence of (1, 3, 2, 4, 5, 7, 6, 2)

Top

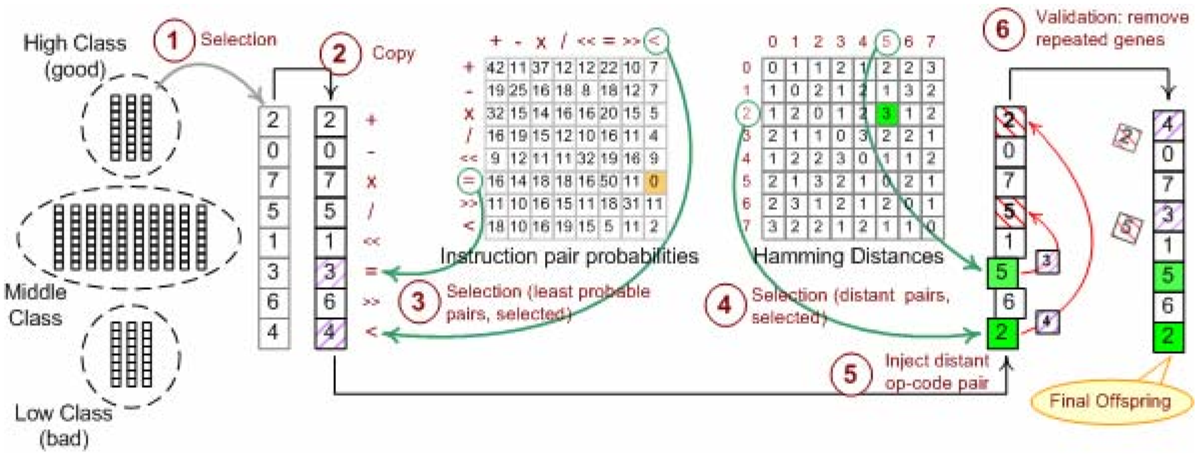
High Class (good)
Middle Class
Low Class (bad)
1 Selection
2 Copy
3 Selection (least probable pairs, selected)
Instruction pair probabilities
Hamming Distances
4 Selection (distant pairs, selected)
5 Inject distant op-code pair
6 Validation: remove repeated genes
Final Offspring

Figure 3- The proposed crossover scheme for *high class* (good) chromosomes – The final offspring which is placed in the next population is the op-code sequence of (4, 0, 7, 3, 1, 5, 6, 2) - Probabilities are per 1000 instructions.

D. *Mutation*

Mutation is performed to prevent GA from getting stuck in local optima. For this problem we propose to randomly select two loci (positions in chromosome) and exchange their values. This causes the chromosome to randomly jump to a new area in the search space. We propose to perform mutation with 1% of probability (i.e. on average, one out of 100 chromosomes is mutated). Notice that if the mutated chromosome is not a good one, it becomes extinct after some generations.

V. RESULTS AND DISCUSSION

The proposed technique can be applied to any processor the op-codes of which are of the same size (like RISC processors). [6] has reported his results for MIPS-R4000. To show the efficacy of our technique we too, applied our proposed technique to MIPS-R4000 and obtained the optimum instruction coding (op-code sequence). Then we ran 8 different applications with the resulted op-code sequence and measured the average switching activity of each program. [6] has done the same procedure for his results as we have done (the preceding steps). We used PCSpim [10] to simulate the execution of programs on MIPS-R4000. This processor has a total of 64 instructions (hence 6-bit op-codes) therefore the chromosomes had 64 genes. The results are shown in Figure 4.

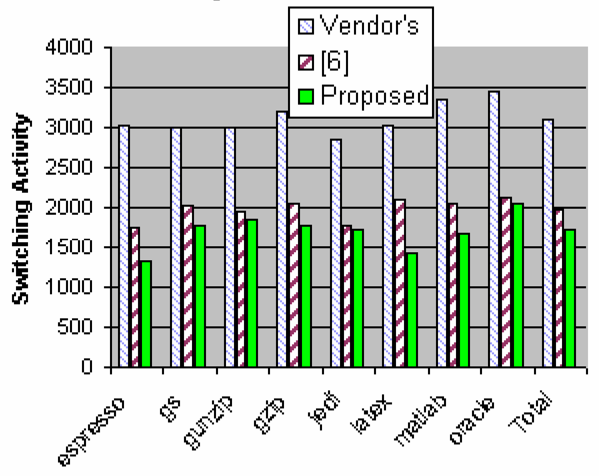

Figure 4- Switching Activities of different programs run with 3 op-code set: Vendor's op-code set, [6] and the proposed technique

As seen in the figure the op-code set, obtained by the proposed technique, reduces the total switching activity of the address bus by **44%** as compared with the original op-code set of MIPS-R4000 (Vendor's), this is **12%** better than of [6].

VI. SUMMARY AND CONCLUSION

The objective of this paper was to find the optimum instruction coding (op-code sequence) for a given processor. The optimum instruction coding is such arranged that raises the least signal transitions while running different applicationc.

Chromosome	Op-code sequence
Population size	$20+10\times n$
Selection	High → roulette wheel
	Middle → Random
	Low → inverse roulette wheel
Encoding	permutation
Crossover type	High → Figure 3
	Middle → Figure 2
	Low → Figure 1
Crossover rate	90%
Mutation scheme	Exchange of two randomly selected loci
Mutation rate	1%

Top

Table III – summary of the proposed genetic algorithm

We presented a genetic algorithm which looked for the optimum op-code set for the instruction set of a given processor. We were motivated to optimize the op-code set since some power saving could be achieved by minimizing the signal transitions raised by instruction words. We deployed genetic computing to find the solution since the search space was very vast (for a processor with n instructions there are a total of $n!$ different op-code sets one (or more) of which raises the least signal transitions on average).

The proposed algorithm takes two inputs: the list of all instructions and an $n \times n$ table each entry of which gives the frequency of its corresponding *instruction pair*. The output of the algorithm is the optimum op-code sequence.

The outline of the proposed algorithm is given in Table II. We proposed three crossover schemes, depicted in figures 1 to 3, which efficiently guide the population toward the optimum op-code sequence. The proposed GA is summarized in Table III. Results for MIPS-R4000 show that, the proposed technique raises 44% less switching activity as compared to the original instruction set of MIPS-R4000. The reduction is 12% better than that of [6].

REFERENCES

[1] Wei-Chung Cheng; Pedram. M, "Power-Optimal Encoding for a DRAM Address Bus," *IEEE Transaction on VLSI,* Vol. 10, Issue 2, pp. 109-118, April 2002

[2] L. Benini, G. De Micheli, E. Macii, D. Sciuto, and C. Silvano, "Asymptotic zero-transition activity encoding for address busses in low-power microprocessor-based systems," in *Proc. IEEE 7th Great Lakes Symp. VLSI*, Mar. 1997, pp. 77–82.

[3] Y. Aghaghiri, F. Fallah, and M. Pedram, "Irredundant address bus encoding for low-power," in *Proc. IEEE Int. Symp. Low-Power Electronics and Design*, Aug. 2001, pp. 182–187.

[4] E. Musoll, T. Lang, and J. Cortadella, "Working-zone encoding for reducing the energy in microprocessor address buses," *IEEE Trans. VLSI Syst.*, vol. 6, pp. 568–572, Dec. 1998.

[5] M. Mamidipaka, D. Hirschberg, and N. Dutt, "Low power address encoding using self-organizing lists," in *Proc. IEEE Int. Symp. Low-Power Electronics and Design*, Aug. 2001, pp. 188–193.

[6] L. Benini, G. De Micheli, A. Macii, E. Macii, and M. Poncino, "Reducing power consumption of dedicated processors through instruction set encoding," in *Proc. IEEE 8th Great Lakes Symp VLSI*, Feb. 1998, pp. 8–12.

[7] L. Benini, G. De Micheli, "State Assignment for Low Power Dissipation," *IEEE Journal of Solid State Circuits*, Vol. 30, No. 3, pp. 258-268, March 1995.

[8] G. D. Hachtel, M. Hermida, A. Pardo, M. Poncino, F. Somenzi, "Re-Encoding Sequential Circuits to Reduce Power Dissipation," *ICCAD-94: IEEE/ACM International Conference on Computer-Aided Design*, pp. 70-73, San Jose, CA, November 1994.

[9] J. Heinrich, MIPS R4000 Microprocessor User's Manual, Second Edition, MIPS Technologies, Mountain View, CA, 1994.

[10] SPIM, a MIPS32 simulator, Larus James, http://www.cs.wisc.edu/~larus/spim.html, retrieved on 2006-04-05.

Top