

# Autonomous Instruction Memory Equipped with Dynamic Branch Handling Capability

Hui-Chin Yang and Chung-Ping Chung  
Dept. of Computer Science, National Chiao Tung University  
1001 Ta Hsueh Road, Hsinchu, Taiwan 300, ROC  
{huichin, cpchung}@cs.nctu.edu.tw  
Phone: 886-3-5731828 Fax: 886-3-5724176

**Abstract**—Memory accesses have always been a speed-limiting factor, and memory bandwidth has always been an intensively contended scarce resource. Nevertheless, with recent pervasive emergence of portable information appliances, the extraordinary power consumption ratio of memory accesses promotes importance of efficient memory system design to an ultimate. We address the following issues: how to minimize memory bandwidth requirement for instruction accesses, and how to minimize memory access delay, again for instruction accesses. Then we propose to move dynamic branch handler (e.g., branch target buffer) from CPU to the instruction memory side (i.e., bind BTB and instruction memory in the same bus module). We present such a design which can help us achieve the following goals: 1. To greatly reduce instruction address bus traffic, which saves much bus power; 2. To autonomously pipe instructions to the CPU, without having to wait for the addresses from CPU, which saves much delay. Using programs in MediaBench as test programs, our design shows that the instruction address bus traffic reduction is 83%, even by accounting for repeated and redundant target address traffic in instruction decode stage. Furthermore, we suggest using very simple techniques to eliminate transmission of most of these target addresses.

**Index Terms**—branch target buffer, dynamic branch handling, instruction address bus, instruction memory, low-power

## 1. INTRODUCTION

Ever since the emergence of the ENIAC at the Moore School of the University of Pennsylvania by Professors Eckert and Mauckly, computer designers have constantly been striving for some fundamental goals: 1. higher execution speed, 2. more functionalities, 3. less hardware requirements, and 4. lower operational costs. These goals often times conflict with each other, and few design techniques can satisfy multiple goals without overwhelming

complexity. On the other hand, several cost factors are involved in the pursuit of those design goals. Almost all of the design techniques mentioned above require extra silicon area. Most techniques require much extra power. And lastly, many design techniques, such as virtual memory, pipelining, superscalar processing, and VLIW, even rely their performance on extensive off-line processing.

In this research, we are particularly interested in the following performance enhancement techniques: dynamic branch handling and its role with both CPU and instruction memory. We study the nature of these techniques, and look into the interaction/organization of such techniques in many of the diversified application scenarios. Further, we propose a design that integrates the dynamic branch handler and the top-level instruction memory into the same bus module. The goal of such a design is twofold in its merit:

1. *Reduce instruction address bus traffic between the processor and the top-level instruction memory to a minimum:*

Through observation, we believe that only the following instruction addresses are absolutely required for the processor to send to its instruction memory: 1. the original starting address of the program; 2. the instruction addresses which will cause abrupt program execution flow; and 3. those instruction addresses which are not continuous with its preceding instruction. Proposed design in this paper will reveal that even for these instruction addresses, their required amount of information can be much reduced.

2. *Reduce the instruction memory read latency to a minimum:*

Memory accesses are believed to be a speed limiting factor in computer design, especially for pipelined designs. Processes involved in memory accesses include addressing mode resolution, interfacing of logic and memory to bus, bus arbitration, bus protocol transformation, and memory read/write operations within the memory. Many of these processes are very

time-consuming in their own rights; worse yet, all of them must take place in sequential order. In coping with these hurdles, many designs, such as the Harvard-like cache system, allowable addressing modes in RISC, etc., have been widely exercised. We further suggest that if instruction address bus traffic can be reduced, then many of these sequential-in-nature processes can take place in parallel, or even be skipped.

The result of this effort is briefed in the following: we propose to remove dynamic branch handler from the CPU, and place it into the bus module containing the top-level (instruction) memory. With this architectural change, the instruction memory module will become self-activated most of the time, bringing a number of significant advantages. Design details, system modeling and evaluation are to be given in this paper. And finally, we summarize some of the advantages due to this design: Instruction reads can be accelerated, address bus signal switching for instruction reads can be minimized, multiplexing of instruction and data address buses becomes more feasible, and the memory bus guzzles much less power.

The paper organization is as follows: The next section presents background and some related works. The third section presents our designs. Section four presents experiments and results. And the last section is conclusions and future research.

## 2. BACKGROUND AND RELATED WORK

We review the evolution of program flow control mechanisms. In very early days of electronic computing, we allowed the program flow control to be very flexible: All instructions are primarily given capability in specifying where the instruction to be executed next lies in. This idea is still preserved in most micro-programming paradigm. Later, we thought that since most instructions were likely to be executed in a sequential fashion, the “go to next instruction wherever it is” capability could be trimmed to “go to the instruction right after me” except for one category of instructions—the flow control instructions. This greatly simplified the instruction design and promoted program execution efficiency. As pipelined execution emerged and soon became a must for almost all computer designs, control flow instructions was discovered to be a great pipeline streaming hurdle. For example, for 20% branch instructions in dynamic instruction count, if 75% of these are taken branches, then given a pipeline design

and if the branch delay slots count is three, the execution time accounting for branch delay overhead is

$$100\%+20\%*75\%*3 = 145\%$$

of the execution time without branch delay. About one half of the ideal execution time is nonproductive! To cope with this shortcoming, various branch handling techniques have been developed. Among them, the most noticeable is the dynamic branch prediction, which offers a very high branch prediction accuracy rate: Experimentally up to about 98% and in commercial designs typically 90~95% accuracy, were reported. This effectively eliminated almost all run time impediments due to control flow instructions.

We further this trend and raise the following question: Since the program execution flow tracking has been so well handled automatically by hardware, is it possible that we design such capability into the memory module, and let the memory module deliver information to the CPU, autonomously, most of the time, without having CPU send memory addresses to memory? To answer this question, we boldly propose to move the dynamic branch handler from the bus module containing CPU, to the bus module containing instruction memory. Our reasoning is as follows: The amount information needed between CPU and the dynamic branch handler is much less than that between the dynamic branch handler and the instruction memory. For this statement to be true, we have made a necessary assumption: We are now assigning the task of “instruction fetch” to the dynamic branch handler, and switch this task back to CPU only upon dynamic branch handler faults. It is believed that the outcome will be very encouraging.

Some related researches are described below. The goals of these works are only similar to ours, but they are by no means the same. As a preliminary, we start by stating the general goals of these researches: to reduce memory accesses and bus traffic between CPU and memory.

Intelligent RAM [1] merges processing and memory into a single chip to lower memory latency and increase memory bandwidth. The difference of our design from Intelligent RAM is that we do not change the existing computing model, but propose new system architecture to reduce large amount of instruction address bus traffic.

Another related design technique [2, 3, 4, 5] called bus encoding also deserves some attention. Bus encoding aims at two major goals: To reduce bus traffic, and to reduce bus line toggles (for reduced bus power consumption). And instruction address

sequence is the stream of bus data most suitable for encoding, since most subsequent addresses are sequential, and the others are caused by branch points, which can be well predicted. Famous encodings include T0 [2, 3, 4] and T0+BTB [5]. Our design is orthogonal to bus encoding/decoding algorithms. We target at eliminating most instruction address bus traffic, and can work in harmony with bus encoding to further achieve information amount reduction benefit. Note that the bus encoding attempts to reduce information amount for every bus transaction, whereas our design attempts to eliminate a bus transaction entirely whenever possible.

Bray and Flynn proposed a technique which integrates BTB into I-cache [6], but the prediction accuracy was concerned in their work. Our design goal here is instruction address bus traffic elimination, whether the dynamic branch handler and instruction memory are designed into one circuit or not.

### 3. DESIGNS

In this design, we use BTB (branch target buffer) as our dynamic branch handler. We first introduce system block diagram of a conventional design using BTB, as in Figure 1.

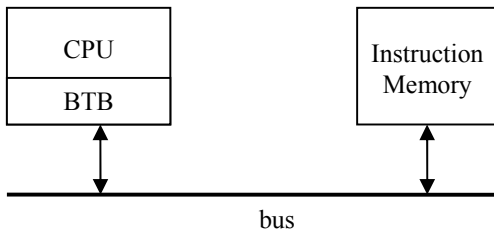


Fig. 1 The block diagram of a conventional [CPU+BTB+instruction memory] design

We then show the system diagram of our design in Figure 2.

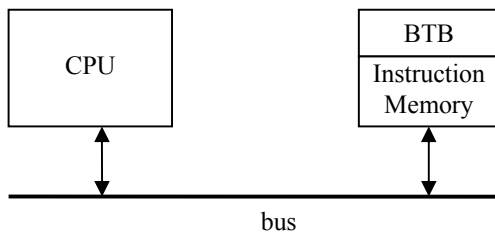


Fig. 2 The block diagram of our proposed [CPU+BTB+instruction memory] design

#### 3.1 Design Overview and System Organization

Figure 2 shows that BTB is moved from CPU to instruction memory side. With this configuration, CPU

only needs to send information to BTB at :

- i. The beginning of program execution
- ii. The decoding of a branch instruction
- iii. The resolution of the branch instruction

#### 3.2 The Algorithms

In this set of algorithms, we assume that: 1. the dynamic branch handler is a BTB; 2. the CPU is five-stage pipelined with IF (instruction fetch), ID (instruction decode), EXE (execution), MEM (data memory access), and WB (result write back) stages. These set of algorithms requires only minor modifications if the dynamic branch handler or the pipeline is otherwise.

Each of the algorithms is described as follows:

##### 3.2.1 The main algorithm

The main algorithm is straightforward: it defines the necessary variables, and puts CPU and BTB to work. It also checks for all exception conditions, and if there is any, then the process will halt.

##### 3.2.2 The CPU\_actions algorithm

The CPU algorithm activates [BTB+IM] by sending it the PC only at initialization. It also needs to provide the [BTB+IM] with the following information: 1. Whether the decoded instruction is a branch. 2. If it is a branch, what the target address is (we assume that the target address is calculated in the ID stage). 3. If a branch instruction results in a branch (we assume that the branch condition is resolved in the EXE stage). To confirm if a branch prediction is correct and to cope with branch mispredictions, this algorithm should record the PC values and the BTB prediction history for instructions up to their EXE stage. Furthermore, it also checks for exception conditions for all cases. Finally, since this algorithm is somewhat tedious, we present it by partitioning the actions into five categories according to in what stage they belong. Note that all these actions should take place in every cycle.

##### 3.2.3 The BTB\_actions algorithm

The BTB algorithm is responsible for the following tasks: 1. It predicts the next fetch instruction address for all instruction reads. For all current instructions that are not found in the BTB, the next fetch instruction address is assumed a fall-through address; hence BTB should be capable of address+1. 2. It creates a BTB entry for all branch instructions, as dictated by a CPU signal B. 3. It updates the predictor status (we assume a popular two-bit saturation counter predictor with strongly/weakly taken/not-taken states; the state transitions can be at your choice) for all entries in BTB, as dictated by a

CPU signal TK. 4. Whether a branch instruction has been captured in the BTB, if its prediction misses, BTB should autonomously restore the correct instruction flow by sending out correct instruction streams to CPU, beginning from the misprediction point. Similarly to the CPU algorithm, to confirm if a branch prediction is correct and to cope with branch mispredictions, this algorithm should record the PC values and the BTB prediction history for instructions up to their EXE stage.

#### ✱The main algorithm

```

initialization:
    BTB reset;
integer
    PC := starting address of program;
    cycle:=0; //# of cycles for program execution//
    branch#:=0; //# of branches executed in program//
    mispredict#:=0; //# of BTB mispredictions//
    BTBPC[3]; //BTBPC buffers PC values of instructions in
                IF, ID, and EXE; used by BTB for instruction
                read and misprediction recovery//
    CPUPC[3]; //CPUPC buffers PC values of instructions in
                IF, ID, and EXE; used by CPU for
                misprediction recovery//
boolean
    CPUtrackpred[3]:= [false,false,false]; //track branch
                predictions; assume up to three unresolved
                predicted branches in pipeline: IF, ID, and
                EXE//
    BTBtrackpred[3];
    PRED; //a control line from [BTB+IM] to CPU, indicating
                BTB prediction outcome//
    B; //a control line from CPU to [BTB+IM], indicating a
                decoded instruction is a branch//
    TK:=false; //a control line from CPU to [BTB+IM],
                indicating a branch is taken//
    ECPT:=false; //a control line from CPU to [BTB+IM],
                indicating an exception occurs in CPU//

main{
    repeat{
        CPU_actions;
        BTB_actions;
        cycle++;}
    until
        exception or exit;}

```

Following facts about these algorithms should be noted: First, the control lines between CPU and [BTB+IM] have been presented in their individual form, for clarity reason. They include the B, TK, and PRED. Attempts to simplify them can be exploited. Second, the target address is sent to BTB every time a branch instruction is decoded, causing much wasted bus traffic. This may be improved, too. Third, we track a number of PC values in CPU, for our purposes.

This storage overhead is not extra in fact, since the PCs are already present in the pipestage latches in a pipelined CPU. Some other data structures we used may be of the same sort.

#### ✱The CPU actions algorithm

```

CPU_actions{
    for IF:
        if initial or after exception
            PC→[BTB+IM];
            CPUPC[3]:=CPUPC[2];
            CPUPC[2]:=CPUPC[1];
            CPUPC[1]:=PC;
            PC++;
        accept instruction from bus and latch instruction in
        instruction register;
        for ID:
            if instruction is a branch
                B:=true, →[BTB+IM];
                branch#++;
            target address→[BTB+IM]; //target address sent to
                [BTB+IM] every time,
                unnecessary; improvement
                opportunity!//
            lasttargetaddr:=target address; //this instruction
                could have been
                mispredicted not taken//
        else
            B:=false, →[BTB+IM]
        if undefined instruction
            ECPT:=true;
    for EXE:
        if instruction is a branch and taken
            TK:=true, →[BTB+IM];
        else
            TK:=false, →[BTB+IM];
            CPUtrackpred[3]:=CPUtrackpred[2];
            CPUtrackpred[2]:=CPUtrackpred[1];
            CPUtrackpred[1]:=accept PRED from
                [BTB+IM];
            if TK≠CPUtrackpred[3] //BTB has mispredicted//
                mispredict#++;
                flush instructions in IF and ID;
                if TK
                    PC:=lasttargetaddr;
                    //a taken branch was mispredicted//
                else
                    PC:=CPUPC[3]+1;
                    //a fall-through branch was
                    mispredicted//
            if execution exception
                ECPT:=true;
    for MEM:
        if data memory exception
            ECPT:=true;
    for WB:
        nil;
    cleric:
        if external interrupt
            ECPT:=true;}

```

### ✖The BTB actions algorithm

```
BTB_actions{
  BTBPC[3]:=BTBPC[2];
  BTBPC[2]:=BTBPC[1];
  for branch prediction and PC generation:
    if initialization or after exception
      BTBPC[1]:=address bus value;
      PRED:=false; //meaning a not taken branch or
                  not a branch instr.//
    else
      if (BTBPC[1] found in BTB AND prediction
is taken)
        BTBPC[1]:=target address;
        PRED:=true; //true means predict that
                  branch is taken//
      else
        BTBPC[1]:=BTBPC[1]+1;
        PRED:=false;
        BTBtrackpred[3]:=BTBtrackpred[2];
        BTBtrackpred[2]:=BTBtrackpred[1];
        BTBtrackpred[1]:=PRED;
  for BTB entry creation:
    if B //CPU has decoded a branch and sent B out in
beginning of EXE//
      if BTBPC[3] entry not found in BTB //and
BTB has no record//
        create entry for BTBPC[2],
        read target address from address bus,
        and set redictor to weakly taken;
  for BTB predictor update:
    if BTBPC[3] entry found in BTB
//this is a branch in its EXE stage//
      if TK
        predictor of BTB entry BTBPC[3] ++;
      else
        predictor of BTB entry BTBPC[3] --;
  for BTB prediction check and misprediction recovery:
    if (TK ≠ BTBtrackpred[3])
      if BTBtrackpred
        BTBPC[1]:=BTBPC[3]+1;
      else BTBPC[1]:=BTBPC[3];
  if instruction memory exception
  ECPT:=true;}
```

## 4. EXPERIMENT RESULTS AND DISCUSSION

### 4.1 Experiment Results

To validate our design and show its usefulness, the following tasks are accomplished. First, we wrote a

simulator to exercise the algorithms, and used pieces of short codes, representing various branch cases, to ensure that the algorithms are correct. Then, we used MediaBench as the test program suite, and collected a set of program runtime profile statistics, with branch behaviors emphasized, to calculate the performance data of a system with this design.

From the experimental data and assuming a 90% BTB accuracy rate, we are able to calculate the amounts of information transferred within and among different bus modules. We call these amounts of information the “internal”, and “external bus” traffic amounts. Table 1 shows the detailed information amounts sent internally and externally. And Figure 3 shows the comparisons of the internal and external, instruction address related information traffic loads, in a pictorial form. Both internal and external bus loadings due to our design are less than those of the conventional design. And note also that the internal information amount of our design should be exactly the same as the external bus traffic of the conventional design, which has been confirmed in the collected data. Since the proposed design has proven itself to be very effective in autonomously providing right instruction stream to the CPU, we are interested in finding out the following fact: Will this design be immune to the accuracy rate variation of the BTB, and provide a quality-assured service to the CPU? To answer this question, we depict the internal, external\_bus, and aggregated traffic reductions of our design versus the conventional design. Results are encouraging: Figure 4 shows that these information reduction rates are relatively stable over a wide range of BTB accuracy rates.

### 4.2 Discussion

This design can find its usefulness in many scenarios. The most noticeable effect of this design is that it relieves the need for instruction address bus most of the time. To name a few application scenarios for this technique to be useful, we note the following: Almost all high-performance computer designs employ instruction cache. On the other hand, most cost-sensitive designs use separate instruction and data memory modules (and for embedded systems, many of the instruction memories are ROMs). In cases in which the CPU has only one memory interface, such as the ARM7 CPU case, as long as we can isolate memory modules for instructions from the others, we can then equip the instruction memory partition with our design, and gain the expected benefits.

Table 1 The Bus traffic Comparisons between traditional design and [BTB+IM] design. In this table, I\_addr stands for instruction address (32 bits) and Branch\_info includes B (1bit), target\_address (32 bits), TK (1 bit) and PRED (1 bit). All units are in millions of instructions on the left half of the table, and millions of bits on the right half.

Benchmark			Information type	Traffic amount			
				Conventional design		Auto [BTB+IM] design	
Program	Total instructions	branch instructions		Internal CPU→BTB	Ext_bus CPU→IM	Internal BTB→IM	Ext_bus CPU→BTB
Adpcm	15.4	1.4	I_addr	15.4*32	15.4*32	15.4*32	<0.0001
			Branch_info	1.4*(32+2)	1.4*10%*32	1.4*10%*32	1.4*(32+2)
Epic	565	85.8	I_addr	565*32	565*32	565*32	
			Branch_info	85.8*(32+2)	85.8*10%*32	85.8*10%*32	85.8*(32+2)
g721	964.5	173.4	I_addr	964.5*32	964.5*32	964.5*32	
			Branch_info	173.4*(32+2)	173.4*10%*32	173.4*10%*32	173.4*(32+2)
Gsm	291.8	18.4	I_addr	291.8*32	291.8*32	291.8*32	
			Branch_info	18.4*(32+2)	18.4*10%*32	18.4*10%*32	18.4*(32+2)
Jpeg	27.8	3.7	I_addr	27.8*32	27.8*32	27.8*32	
			Branch_info	3.7*(32+2)	3.7*10%*32	3.7*10%*32	3.7*(32+2)
Mpeg2	1382.1	247.4	I_addr	1382.1*32	1382.1*32	1382.1*32	
			Branch_info	247.4*(32+2)	247.4*10%*32	247.4*10%*32	247.4*(32+2)
Average	541.1	88.35		20319.1	17597.92	17597.92	3003.9

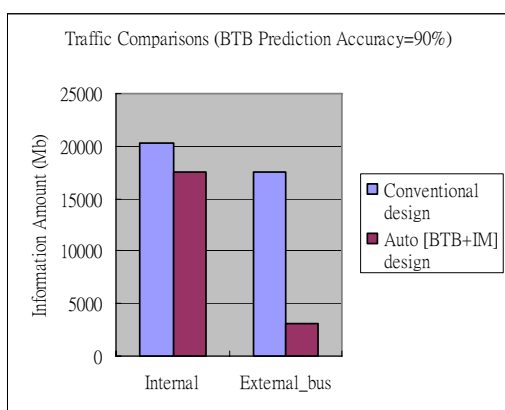


Fig. 3 The reduced bus traffic comparison between conventional and proposed designs

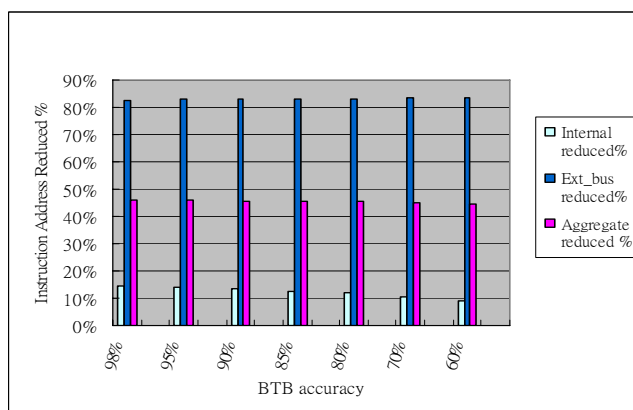


Fig. 4 The various reduced information amounts versus BTB accuracy

Our design is superior to bus encoding technique in that: We attempt to not only reduce the bus traffic, but even eliminate the need for bus to transfer instruction addresses most of the time. This brings two advantages: First, the bus can be freed for any other possible needs; and secondly, since buses are typically much slower than other circuits and pipeline must insert multiple stall cycles to tolerate the low bus speed, eliminating the need for bus implies a big saving in time.

In summary, several goals/benefits have been achieved during the course of this work:

1. Bus congestion can be greatly relieved with this proposed design. Instruction fetches account for about two thirds of all memory fetches in a typical program, and almost all instruction address transfers over the bus can

be eliminated with this design.

2. System power can be greatly lowered, too. Most system power is spent over memory bus. And with this design, this power factor can be greatly reduced.
3. Program run time can be effectively shortened. If the memory system is substantially slower than the CPU, this effect is apparent. On the other hand, even if the pipeline design is well balanced and the instruction fetch does not cause any extra delay, there is still benefit. Most performance designs [7, 8, 9] explore instruction level parallelism. With minor changes to this design, the instruction memory can provide the CPU with multiple instructions in a cycle, aiding the CPU to explore instruction parallelism.

## 5. CONCLUSIONS AND FUTURE RESEARCH

### 5.1 Conclusions

This paper has depicted the picture of how to design a dynamic branch handler into the instruction memory module. The vision of this research is: We want to show that with this architectural change, we are able to manipulate specific system parameters. The design details are presented, including the system block diagram and the operational algorithms for both CPU and the dynamic branch handler. Current status of this work is still primitive, but the future looks encouraging. We will continue to finish the running tasks, and explore further studies.

### 5.2 Future Research

More elaborated evaluation and proof is still in demand. And further in-depth works are being undertaken by our research team. To extend this work, we are also working on the following research topics:

#### 1. *An intelligent and autonomous instruction memory module design*

In the proposed design presented above, the [branch handler+instruction memory] module is fully self-motivated in delivering instructions to the CPU, but still in a traditional way. We view this as a computer architectural change. We suggest pushing the idea one step further, and make the branch handler be able to do the following: identify branch instruction, and calculate PC+offset. We support these viewpoints as follows: while decoding a full set of instructions is tedious, identifying only one (or a few) instruction(s) belonging to the same category is trivial—merely an AND gate is needed. And almost all branch instructions use the PC relative addressing mode, due to program relocation. With these two capabilities, the branch handler—perhaps it should be renamed a program flow tracer—can indeed self-motivated to automatically provide a very precise instruction stream to the CPU, except at initialization or branch mispredictions. We believe that this innovation can be a computer architectural innovation.

#### 2. *An autonomous data memory module design*

Using the same idea of predicting next instruction address, next data address may also be predicted. Data accesses during program execution have many differences in nature: First, they do not occur in every clock cycle, even in a pipelined design. And secondly, consecutive data accesses are less likely to be sequential even for array accesses. These difficulties should be explicitly addressed.

## REFERENCES

- [1] David Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent RAM," *IEEE Micro*, pp. 34-44, April 1997
- [2] Tsung-Hsi Weng, Wei-Hao Chiao, Jean Jyh-Jiun Shann, Chung-Ping Chung, and Jimmy Lu, "Low-Power Data Address Bus Encoding Method", *Proc. of CDES'05*, Las Vegas, USA, pp.204-210, June 2005
- [3] Peter Petrov and Alex Orailoglu, "Low-Power Instruction Bus Encoding for Embedded Processors" *IEEE Transactions on VLSI Systems*, Vol. 12, NO. 8, August 2004
- [4] L. Benini, G. DeMicheli, E. Macii, D. Sciuto, and C. Silvano, "Asymptotic zero-transition activity encoding for address busses in low-power microprocessor-based systems" *GLS-VLSI-97: IEEE 7th Great Lakes Symposium on VLSI*, pp. 77-82, Urbana-Champaign, IL, March 1997.
- [5] Yau-Chong Hu, Wei-Hao Chiao, Jean Jyh-Jiun Shann, Chung-Ping Chung, and Wen-Feng Chen, "Low-Power Branch Prediction", *Proc. of CDES'05*, Las Vegas, USA, pp.211-217, June 2005
- [6] Brian K. Bray and M.J. Flynn "Strategies for branch target buffers," *Technical Report CSL-TR-91-480*, Stanford University, CA, 1991.
- [7] Jih-Ching Chiu, I-Huan Huang and Chung-Ping Chung, 2000, "Design of Instruction Stream Buffer with Trace Support for X86 Processors," *IEEE International Conference on Computer Design*, Austin, Texas, Sep. 17-20, 2000.
- [8] Jih-Ching Chiu, R-Ming Shiu, Shyh-An Chi, and Chung-Ping Chung, "Instruction Cache Prefetching Directed by Branch Prediction," *IEE Proceedings: Computers and Digital Techniques*, Vol. 146, No. 5, pp. 241-246, September 1999.
- [9] Jih-Ching Chiu, Michael Jin-Yi Wang, and Chung-Ping Chung, "Design of Instruction Address Queue for High Degree x86 Super Scalar Architecture," *Journal of Information Science and Engineerin*, Vol. 18, No. 3, pp. 393-409, May 2002.