Design of a G-Share Branch Predictor for EISC Processor

InSik Kim[1], JaeYung Jun[2], Yeoul Na[2], and Seon Wook Kim[2]

[1] Department of Automotive Convergence, Korea University / Seoul, South Korea kiscj@korea.ac.kr
[2] Department of Electrical and Computer Engineering, Korea University / Seoul, South Korea
{cool92-3, rapidsna, seon}@korea.ac.kr

* Corresponding Author: Seon Wook Kim

Received October 12, 2015; Accepted October 20, 2015; Published October 31, 2015

* *Short Paper*

***Abstract*:** This paper proposes a method for improving a branch predictor for the extendable instruction set computer (EISC) processor. The original EISC branch predictor has several shortcomings: a small branch target buffer, absence of a global history, a one-bit local branch history, and unsupported prediction of branches following *LERI*, which is a special instruction to extend an immediate value. We adopt a G-share branch predictor and eliminate the existing shortcomings. We verified the new branch predictor on a field-programmable gate array with the Dhrystone benchmark. The newly proposed EISC branch predictor also accomplishes higher branch prediction accuracy than a conventional branch predictor.

***Keywords*:** Microarchitecture, Branch predictor, Digital logic circuits

1. Introduction

An addition of pipeline stages is one easy method to increase an operational clock frequency; thus, modern microprocessors have been developed to have deeper pipeline stages. However, as the pipeline becomes deeper, the branch miss penalty gets worse. For this reason, modern microprocessors include very advanced branch predictors [1].

EISC is the abbreviation for extendable instruction set computer. This processor extends the immediate value of instructions by using a special instruction called LERI [2], and therefore, the processor can compact code size because the immediate value is not large, in most cases (i.e., even LERI is not needed). The EISC processor has a simple branch predictor that delivers poor performance due to a small branch target buffer (BTB), absence of branch correlation, and a limit of a one-bit local branch history. There are several solutions for these problems, such as expanding the size of the BTB, adding a saturation counter, implementing a pattern history table (PHT), and using a global history register (GHR). A G-share branch predictor has all of these components, and thus, it shows high branch prediction accuracy with a relatively small area [3].

In this paper, we implement a G-share branch predictor in the EISC processor to leverage its high prediction accuracy. However, the performance of the G-share branch predictor implemented in the EISC processor was worse than expected due to the following two problems.

Prediction accuracy between G-share branch predictors decreases when the distance of a pair of branches is so close that the following branch makes a prediction even before the previous branch history is updated in the GHR.

A distant branch target address that exceeds an immediate bit field of the EISC branch instruction can be expressed in combination with LERI instructions. The current EISC branch predictor does not support a branch instruction that is preceded by LERI instructions.

By solving these problems, we make main contributions in this paper as follows.

We propose a method to improve the G-share predictor by updating the GHR speculatively with a previous prediction history.

We propose logic that can predict branch instructions following LERI instructions.

We verified our new branch predictor on a Xilinx VIRTEX5 field-programmable gate array (FPGA) [4] with the EISC processor and achieved a 136% improvement in branch prediction accuracy compared to a conventional bimodal branch predictor.

The rest of the paper is organized as follows. Section 2 covers the background of EISC and its shortcomings. Section 3 describes the implementation of the proposed new EISC branch predictor, and its performance is

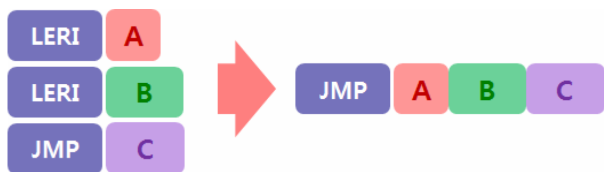


Fig. 1. Behavior of *LERI* Instruction.

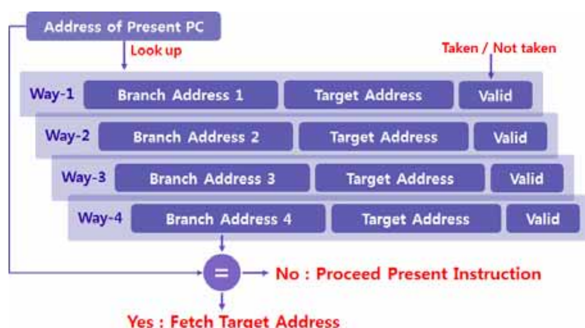


Fig. 2. Original EISC Branch Predictor.

evaluated in Section 4. Finally, Section 5 concludes the paper.

2. Background and Motivation

EISC instruction set architecture (ISA) has a special instruction, called LERI to extend immediate values of instructions. The EISC instructions are able to handle at maximum 32-bit immediate values by using LERI up to three times. As shown in Fig. 1, it can also be used to specify a target address of a branch instruction so that a program can jump anywhere in memory. Without LERI, indirect branches should be used, which degrades performance.

2.1 Analysis of an Original EISC Branch Predictor

This section describes how an original EISC branch predictor works and why its accuracy is low. A BTB consists of a four-entry, fully associative cache for predicting a branch target address, as shown in Fig. 2. Also, no additional storage exists to store a history of branches for predicting the direction of a branch. Instead, the branch predictor stores only branches taken. Any branches in the BTB are always predicted as taken. Because of this simple mechanism, the branch predictor can be implemented with a small area, and its latency becomes very short.

However, there are four performance issues in this simple branch predictor. First, the original EISC branch predictor can store only four branches. Second, this branch predictor does not consider a global history of branches, e.g., branch instructions in nested loops. Third, as the local branch history has one-bit storage, all branches are predicted as taken if there is a matched entry in the BTB. In this case, if a direction of a branch alternates between taken and not-taken states, the branch is continuously mispredicted. Fourth, branch instructions following *LERI* instructions are not supported by the BTB. Thus, those unpredicted branches frequently cause pipeline stalls, because there are so many branches accompanying *LERI* instructions that jump a long distance, such as function call routines. In this paper, we improve branch prediction performance by redesigning the EISC branch predictor to solve these four shortcomings.

```
for (i=0; i<100; i++)
    for (j=0; j<3; j++)
```

Fig. 1. Nested Loop Code for Testing G-share Branch Predictor.

2.2 G-share Branch Predictor

G-share is a two-level branch predictor based on the bi-modal branch predictor [5]. With the program code shown in Fig. 3, the original EISC branch predictor and the bi-modal predictor miss the branch instruction in the last iteration of an inner loop. However, G-share can predict most branches correctly, because it uses a global history register (GHR). The GHR is a shift register that stores resolved directions of recent branches. Different phases of a branch instruction can be distinguished by accessing a pattern history table (PHT), XORing program counter (PC) and GHR. Thus, the local history of a branch is stored as multiple PHT entries according to the history of previous branches.

As noted, four shortcomings of the original EISC branch predictor are 1) a small BTB, 2) absence of a global history, 3) a one-bit local branch history, and 4) lack of prediction of branches following *LERI* instructions. The low accuracy of the branch prediction due to these shortcomings results in larger stalls in pipelines at deeper pipelines.

A BTB can be expanded to more than four entries in G-share, and its PHT stores two-bit local branch history information with a two-bit saturation counter. As a result, issues 1), 2), and 3) are solved by adopting the G-share branch predictor.

2.3 Improved G-share Branch Predictor

Branch instructions are fetched consecutively with the program code shown in Fig.. 4. In this case, branch B is fetched into a pipeline before branch A arrives at the execute stage. Consequently, when branch B is predicted by reading its PHT entry, the GHR does not include the history of branch A. On the other hand, when branch B arrives at the execute stage and updates its PHT entry, the GHR now includes the history of branch A. With these different GHR states, single branch instruction B accesses the different PHT entries when reading and updating the PHT, which degrades accuracy of the branch predictor. In addition, as the number of stages for fetch and decode increases, this situation can occur more frequently, because it takes more cycles to resolve a branch instruction after it

```
.L307():
    76aa:   e0 c9        cmpq 0x0,%R9
    76ac:   0c d5        jz   76c6      <.L311>
    76ae:   4a e4        lea  (%R10),%R4
    76b0:   00 a1        ldi  0x0,%R1
    76b2:   b3 bf        cmp  %R3,%R11
    76b4:   03 d7     A  jc   76bc      <.L314>
    76b6:   03 d4     B  jnz  76be      <.L315>
    76b8:   a2 bf        cmp  %R2,%R10
    76ba:   01 d6        jnc  76be      <.L315>
```

Fig. 4. Continuous Branch Instructions.

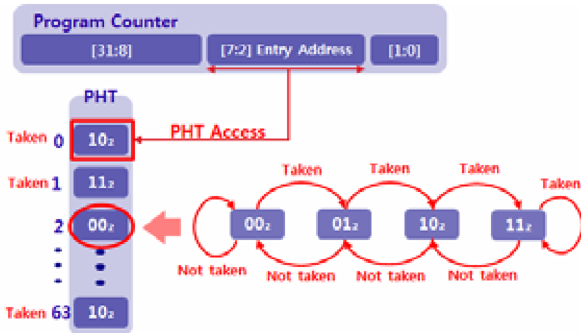


Fig. 5. Saturation Counter and Pattern History Table.

is fetched. We solve this problem by updating the GHR speculatively in the fetch stage. Lastly, the issue of predicting branches that follow *LERI* instructions can be solved by implementing a logic that can recalculate an actual branch instruction address.

3. Implementation

In this section, implementation of our new G-share branch predictor is described.

3.1 Pattern History Table

Fig. 5 describes an overall structure of our pattern history table (PHT). Each PHT entry is a two-bit saturation counter, which increases with “taken” and decreases with “not-taken.” It is used to predict the direction of a branch. With 00_2or 01_2, the branch is predicted as not-taken, whereas for 10_2 or 11_2, it is predicted as taken. We can make the bi-modal branch predictor by using a PHT and a BTB [5].

3.2 Global History Register and G-share Branch Predictor

We implemented a global history register where size depends on the PHT entry size:

$$\log\ (\text{PHT size}) = \#\ \text{bit of GHR} \qquad (1)$$

Taken/not-taken information of resolved bran-ches is inserted into the least significant bit (LSB) of the GHR, as

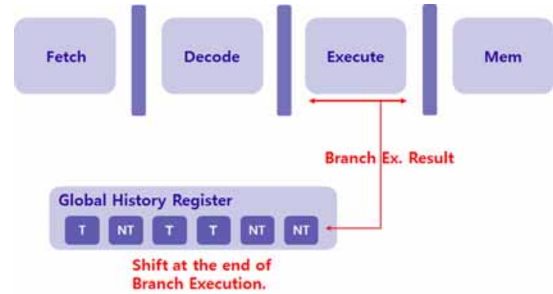


Fig. 6. Global History Register.

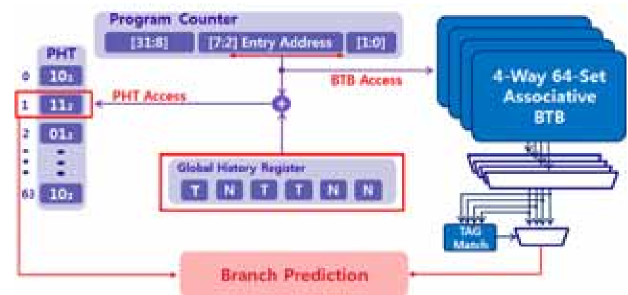


Fig. 7. G-share Branch Predictor.

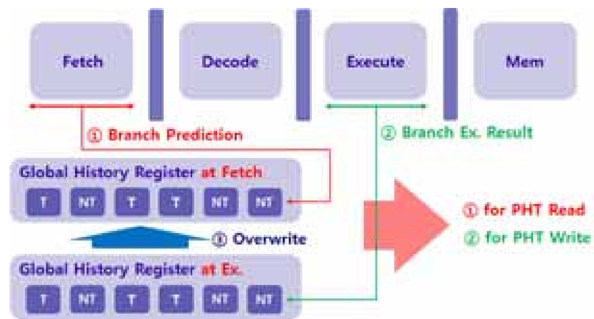


Fig. 2. Advanced Global History Register.

shown in Fig. 6. When a branch is fetched into a pipeline, a part of PC and GHR is XORed to read its corresponding entry in the PHT. After that, the branch instruction is resolved, whether it is taken or not at the execute stage. PHT and GHR are updated at this time for the next branch prediction..

3.3 Improved G-share Branch Predictor

Fig. 7 shows an original G-share branch predictor architecture. Due to deep pipelines, the GHR may not include the history of preceding branches when a branch needs to be predicted at the fetch stage. To solve this problem, we implement another GHR at the fetch stage. It speculatively stores a predicted branch direction at the fetch stage, unlike the original GHR, which stores resolved branch directions at the execute stage. However, a GHR at the fetch stage can be updated with the wrong direction due to misprediction of a preceding branch. To recover this, the GHR at the fetch stage is overwritten with the original GHR whenever a branch is mispredicted. As a result, the

new GHR is used to read the PHT, and the original GHR is used to write the PHT, as shown in Fig. 8.

3.4 Prediction Logic for Branch Instruction Following *LERI*

The original EISC branch predictor does not predict branches that are fetched right after *LERI*, and therefore, such branches are always predicted as not-taken. The reason is that *LERI* folding logic at the fetch stage combines a *LERI* instruction and the following non-*LERI* instruction into a single instruction, then passes it to the decode stage. Therefore, *LERI* is not actually executed at the execute stage but is folded with its following instruction, which improves instruction per cycle (IPC). An instruction goes through a pipeline with its PC and, for a folded instruction, its PC is determined with the PC of the first *LERI* instruction because of PC relative addressing and exception. In the original architecture, the PC of a branch following a *LERI* instruction cannot be known after it is folded. For this reason, a branch with a *LERI* instruction cannot be inserted into the BTB. To calculate its PC, we implement the *LERI* folding logic at the fetch stage to count the number of folded *LERI* instructions. The fetch stage passes the value of the *LERI* counter as well as an instruction, a folded PC and an extended immediate value to the decode stage. At the decode stage, the actual PC of a branch is calculated by adding the folded PC, i.e., the PC of the first *LERI* instruction with the value of the *LERI* counter. This logic enables prediction of branches following the *LERI* instructions.

4. Evaluation

In Section 4, we evaluate the performance of our new branch predictor with the results of various test codes and the Dhrystone benchmark program.

4.1 Performance with Conventional G-share Branch Predictor

Fig. 9 shows example code of correlated branches. In the figure, the branch of the inner loop can be predicted, exploiting the global history. We executed this code with various branch predictors. BTB-based and bi-modal branch predictors show the same miss rate, 20.76%, because neither of them considers the global history. On the other hand, the G-share branch predictor, which considers the global history, shows a 1.2% miss rate.

4.2 Performance with Improved G-share Branch Predictor

The conventional G-share branch predictor does not work normally with program code shown in Fig. 10. When a branch instruction is predicted at the fetch stage, the GHR may not have been updated with the history of all previous branches, which negatively impacts prediction accuracy. Therefore, we improved G-share by adding another GHR, as shown in Section 3.3. Performance of our

Table 1. Performance of G-share Branch Predictor.

Branch Predictor	# of branch instructions	# of miss prediction	Miss rate
BTB based	501	104	20.76%
Bi-modal		104	20.76%
G-share		6	1.20%

Table 2. Performance of Advanced G-share Branch Predictor.

G-Share	# of branch instructions	# of miss prediction	Miss rate
Original	500	103	20.60%
Enhanced		7	1.4%

```
for(i = 0; i < 100; i++)
{
    br_1++;
    for(j = 0; j<4; j++)
        br_2++;
}
```

Fig. 10. Test Code for the Advanced G-share Branch Predictor.

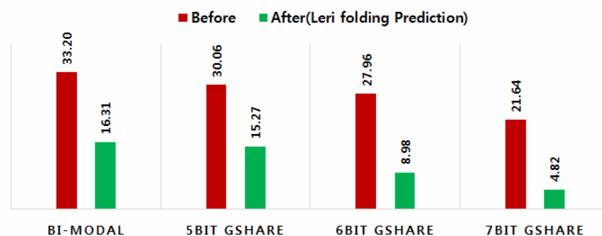


Fig. 11. Performance of the G-share branch predictor supporting branches following *LERI* instructions.

enhanced G-share is shown in Table 2. The miss rate was reduced from 20.6% to 1.4%.

4.3 Performance with Prediction for Branches Preceded by *LERI*

To evaluate the prediction logic for branches preceded by LERI instructions, Dhrystone [6] was used as a benchmark program. We tested branch predictors with various configurations, and the results are shown in Fig. 11. After the branch predictor is able to predict the branch following LERI, the bi-modal miss rate was reduced from 33% to 16%. With a five-bit G-share, the miss rate was lowered from 30% to 15%. In particular, the miss rate of a seven-bit G-share was decreased to approximately 1/4. In addition, because the branch predictor has a larger PHT and GHR, the effect of prediction logic for a branch following a LERI instruction becomes greater. Our six-bit G-share branch predictor has storage of an eight-byte PHT (2 bits * 64), 12-bit GHRs (6 bits * 2), a 1.8KB BTB and additional logic for addressing and synchronizing two

GHRs. Therefore, with a small area overhead, it shows great improvement with respect to prediction accuracy.

5. Conclusion

In this paper, we proposed the design of a new G-share branch predictor for the EISC processor. We identified the shortcomings of the original EISC branch predictor and enhanced it by adopting a G-share branch predictor. Also, we addressed the performance issues of the adopted EISC G-share branch predictor and presented feasible design solutions. The proposed branch predictor was synthesized and tested on an FPGA (Xilinx VIRTEX5) with the Dhrystone benchmark. The proposed EISC branch predictor that has a six-bit GHR and a 64-entry PHT shows prediction accuracy of 91.02% (with an 8.98% miss rate). Moreover, it improved prediction accuracy by 136% compared to a bi-modal branch predictor that has a 64-entry PHT without prediction logic for a branch following LERI.

Acknowledgement

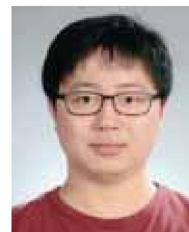
This work (Grants No. C0217499) was partially supported by Business for Cooperative R&D between Industry, Academy, and Research Institute funded Korea Small and Medium Business Administration in 2014. This work was also supported by the IT R&D program of MOTIE/KEIT. [10052716, Design technology development of ultra-low voltage operating circuit and IP for smart sensor SoC] .

References

[1] Sprangle, E. and Carmean, D., "Increasing processor performance by implementing deeper pipelines," Proc. 29th Annual Int. Symp. on Computer Architecture, 2002.
[2] Article (CrossRef Link)
[3] Advanced Digital Chips Inc., "Extenable Instruction Set Computer," Article (CrossRef Link)
[4] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," In Proc. of the fifth int. conf. on Architectural support for programming languages and operating systems, 1992.
[5] Article (CrossRef Link)
[6] Xilinx, "Virtex-5 Family Overview," Article (CrossRef Link)
[7] Tse-Yu Yeh and Yale N. Patt, "Two-level adaptive training branch prediction," In Proc. of the 24th annual int. symp. on Microarchitecture, 1991.
[8] Article (CrossRef Link)
[9] Reinhold P. Weicker, "Dhrystone: a synthetic systems programming benchmark," Magazine Communications of the ACM, vol. 27, issue 10, pp. 1013-1030, October 1984.
[10] Article (CrossRef Link)

Copyrights © 2015 The Institute of Electronics and Information Engineers

InSik Kim received his B.Eng. from the Department of Electronics and Radio Engineering at Kyung Hee University, Yong-in, Korea, in 2014. Since 2014, he has been in the master's course in the Department of Automotive Convergence at Korea University, Korea. His research interests include microarchitecture, digital design and embedded systems.



JaeYung Jun received his B.Eng. from the Department of Electrical Engineering, Korea University, Seoul, Korea, in 2012, and is working on his PhD in the same department. His research interests include microarchitecture, system on chip, and digital design.

Yeoul Na received her B.Eng. in Electrical Engineering from Korea University, Seoul, Korea, in 2008 and received a PhD in Electrical and Computer Engineering from the same university in 2015. She is now a post-doctoral researcher in Prof. SeonWook Kim's laboratory at Korea University. Her research interests include parallelization, JavaScript engines, compiler construction and microarchitecture.

SeonWook Kim received a B.Eng. in Electronics and Computer Engineering from Korea University, Seoul, Korea, in 1988. He received an MSc in Electrical Engineering from Ohio State University, Columbus, Ohio, USA, in 1990, and a PhD in Electrical and Computer Engineering from Purdue University, West Lafayette, Indiana, USA, in 2001. He was a senior researcher at the Agency for Defense Development from 1990 to 1995, and a staff software engineer at Inter/KSL from 2001 to 2002. Currently, he is a Professor with the School of Electrical and Computer Engineering at Korea University and is Associate Dean for Research at the College of Engineering. His research interests include compiler construction, microarchitecture, and SoC design. He is a senior member of ACM and IEEE.