國立交通大學

電機學院與資訊學院 資訊學程

碩 士 論 文

智慧型自主指令記憶體設計

Intelligent Autonomous Instruction Memory Design

研 究 生：王立銘

指導教授：鍾崇斌　教授

中 華 民 國 九 十 五 年 七 月

智慧型自主指令記憶體設計
Intelligent Autonomous Instruction Memory Design

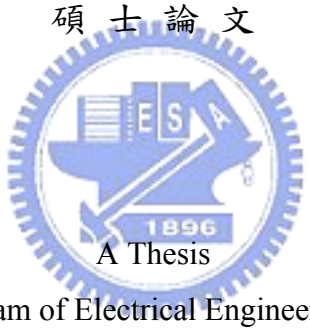
研 究 生：王立銘　　Student：Li-Ming Wang

指導教授：鍾崇斌　　Advisor：Chung-Ping Chung

國 立 交 通 大 學

電機學院與資訊學院專班 資訊學程

碩 士 論 文



A Thesis

Submitted to Degree Program of Electrical Engineering and Computer Science

College of Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science

July 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年七月

智慧型自主指令記憶體設計

學生：王立銘　　指導教授：鍾崇斌 博士

國立交通大學 電機學院與資訊學院 資訊學程（研究所）碩士班

摘　　要

智慧型自主指令記憶體的主要概念是將動態分支預測器併入最上層的指令記憶體使後者具備“程式流程追蹤” 能力。藉著動態分支預測器的協助，指令記憶體在多數時間可以不需 CPU 核心提供指令位址而知道要到那個位址去擷取下一道指令。這個概念的目的是要將 CPU 與指令記憶體之間的指令位址傳輸量降到最低。實作出這樣的概念或許可以比許多已知的指令位址匯流排編碼技術要節省更多的能源。當動態分支預測器從 CPU 移到指令記憶體，新增輔助硬體與一套溝通 CPU 與指令記憶體之間有效率的控制匯流排傳輸協定對維持程式流程的正確性以及原本動態分支預測器的運作是不可或缺的。運用上述概念的一個簡單設計會先提出來，接著提出配備具有解碼分支指令並計算其分支目標位址能力的部份指令解碼器的一個強化設計。最後提出的是配備部份指令解碼器與返回堆疊的更強化設計。實驗結果顯示這三個設計比起傳統的架構分別減少 97.71%，98.49% 與 99.99%的指令位址傳輸以及 84.99%，86.54%與 92.01%的總位元變化量。以上提出的設計都勝過 T0 編碼技術許多。第三個設計略勝 T0 DAT(128 筆)編碼技術。

智慧型自主指令記憶體設計
Intelligent Autonomous Instruction Memory Design

student：Li-Ming Wang　　Advisors：Dr. Chung-Ping Chung

Degree Program of Electrical Engineering and Computer Science
National Chiao Tung University

ABSTRACT

Main concept of Intelligent Autonomous Instruction Memory (iAIM) is to equip top-level instruction memory with "program flow tracing" capability by incorporating dynamic branch predictor into top-level instruction memory. With help of dynamic branch predictor, instruction memory can know where to fetch the next instruction without instruction address supplied by CPU most of the time. The purpose of such concept is to reduce instruction address traffic between CPU and instruction memory to a minimum. The realization of such concept may conserve more energy on instruction address bus than many known instruction address bus encoding techniques. While dynamic branch predictor is removed from CPU to instruction memory, additional auxiliary hardware and an efficient control bus communication protocol between CPU and instruction memory are essential to maintain program flow correctness and original dynamic branch predictor operation. A simple design of iAIM that makes use of the above concept is proposed first, followed by an enhanced design that equips iAIM with a partial instruction decoder capable of calculating branch target address by decoding branch instruction. A more enhanced design that equips iAIM with a partial instruction decoder and a return stack is proposed finally. The experiment results show three proposed designs can reduce instruction address transmission to 97.71%, 98.49% and 99.99% and reduce total bit transitions to 84.99%, 86.54% and 92.01% compared with conventional architecture respectively. All these designs greatly outperform T0 encoding technique. The third design outperforms T0 DAT with 128 entries technique slightly.

誌　　謝

首先感謝指導教授 鍾崇斌老師在 2002 年某次公司舉辦的教育訓練課程中，讓我重新拾起對計算機組織與架構方面的學習興趣，以及產生在學習上努力不懈的理念。這樣的理念促使我 2003 年開始到在職專班修學分，並在 2004 年考取專班資訊組後的修課與研究過程中獲益匪淺。同時也要感謝實驗室另一位老師 單智君教授過去幾個月對研究提出的指正與建議。再者感謝口試委員 邱舉明教授與邱日清教授在口試時的寶貴意見，使論文得以修改得更完整。除此之外，感謝博士班的喬偉豪、翁綜禧學長及楊惠親學姊在研究過程中所給與的協助。

就一個在職生來說，我必須感謝公司主管吳國樑副處長、許致榮經理最近一年在工作上的安排，以及鄭昆霖、林宜賢同事在工作上卓越的表現，使我在最後這一年得以順利地完成學業。感謝公司主管黃英惠資深處長、第一次修學分班(計算機網路)的授課老師簡榮宏教授在兩年前要考專班時惠賜推薦書，並感謝交大的在職專班讓我有再一次回到校園學習的機會。

過去時常因為考試、作業、研究而無法回高雄善盡照顧家庭的義務，在這裏衷心感謝家人的體諒與支持。

王立銘 2006.9.3

Table of Contents

摘　　要 i
ABSTRACT ii
誌　　謝 iii
Table of Contents iv
List of Figures v
List of Tables vi
Chapter 1 Introduction 1
1.1 Background 1
1.2 Research Motivation 7
1.3 Research Objective 7
1.4 Organization of this Thesis 8
Chapter 2 Design of Proposed Architecture 9
2.1 Challenges in Design 9
2.2 Key Ideas in Design 9
2.3 Proposed Design of iAIM 16
2.4 Proposed Design of Enhanced iAIM with Partial Decoder 24
2.5 Proposed Design of Enhanced iAIM with Partial Decoder and Return Stack 32
2.6 Design Restriction and Execution Examples 41
Chapter 3 Evaluation and Discussion 51
3.1 Evaluation Methodology 51
3.2 Evaluation Metrics 52
3.3 Experimental Environment 53
3.4 Experimental Benchmark 55
3.5 Experimental Results 58
3.6 Discussion 61
Chapter 4 Conclusion and Future Works 65
4.1 Conclusion 65
4.2 Future Works 65
References 67
Vita 68

List of Figures

Figure 1.1 Diagram of T0 encoding 6
Figure 1.2 Diagram of T0 DAT encoding 7
Figure 2.1 Block diagram of conventional architecture 10
Figure 2.2 Block diagram of iAIM design 10
Figure 2.3 Automatic instruction address generator inside iAIM 11
Figure 2.4 Control line that iAIM uses to inform CPU 11
Figure 2.5 S-Indicate control lines that CPU uses to inform iAIM 13
Figure 2.6 Buses between CPU and iAIM 16
Figure 2.7 Additional circuit in iAIM 18
Figure 2.8 Action timing of partial decoder 25
Figure 2.9 Additional circuit in enhanced iAIM with partial decoder 26
Figure 2.10 Action timing and algorithm for procedure call handling 32
Figure 2.11 Action timing and algorithm for procedure return handling 33
Figure 2.12 Additional circuit in enhanced iAIM with partial decoder and return stack 34
Figure 2.13 Scenario of return stack overflow in a finite depth push-down stack 40
Figure 3.1 Simulation flowchart 54
Figure 3.2 Percentage of reduced instruction address bus active cycles 59
Figure 3.3 Percentage of reduced bit transitions 60
Figure 3.4 Percentage of bit transitions on address bus and control line(s) 60
Figure 3.5 Percentage of bit transitions on address bus and control lines S-Indicate, P-Taken 61
Figure 4.1 Unified instruction and data memory system 66

List of Tables

Table 3.1 Instruction counts and maximum procedure call depth for selected benchmarks.....57
Table 3.2 Simulation Results.....59

Chapter 1 Introduction

Main concept of Intelligent Autonomous Instruction Memory (iAIM) is based on the following arguments : 1. Dynamic branch predictor is smart enough to trace program flow with over 90% accuracy. 2. Dynamic branch predictor rarely needs CPU intervention. 3. Should dynamic branch predictor be moved from CPU to instruction memory?

After equipping top-level instruction memory with “program flow tracing” capability by incorporating dynamical branch predictor, instruction memory can know where to fetch the next instruction without instruction address supplied by CPU most of the time.

Throughout this thesis, a classic MIPS five-stage pipeline is assumed. The five stages are Instruction fetch (IF) stage, Instruction decode/register fetch (ID) stage, Execution/effect address (EX) stage, Memory access (MEM) stage and Write-back (WB) stage respectively. Figure A.24 in [1] shows the implementation of MIPS data path adopted in this thesis.

1.1 Background

1.1.1 How bus consumes Power

The amount of power dissipated due to switch activity is related to the voltage level on the bus, the capacitance between the bus and the ground which is called self-capacitance and the capacitance between the adjacent bus lines which is called coupling-capacitance. The general power consumption equation of bus [2] is shown as follows :

$$P_{csc} = \frac{1}{2} \cdot (\mathbf{SBT} + \lambda\, \mathbf{CBTr} + 4\, \lambda\, \mathbf{CBTg}) \cdot Cs \cdot V_{dd}^2$$

- Cs : self-capacitance
- Cc : coupling-capacitance
- λ : Cc / Cs
- **SBT** : self-bus bit transitions
- **CBTr** : coupling 1-bit transitions

- **CBTg** : coupling bit toggles
- V_{dd} : supply voltage

There are 2 kinds of bus in computer system : on-chip bus and off-chip bus.

For off-chip bus system, coupling-capacitance is negligible compared with self-capacitance. Power consumed on off-chip bus is nearly power consumed by charging and discharging self-capacitances of all individual bus lines. Because capacitance driven by I/O nodes is three orders of magnitude larger than that on the internal nodes of the processor [3], reducing self-bus bit transitions will reduce the same percentage of bus power as well. Off-chip bus occupies non-trivial portion of power in a system (power consumption of Intel Celeron core at 266 MHz is 16W, while its off-chip bus operating at 133 MHz consumes 3.3W [4]).

For on-chip bus system, coupling-capacitance can not be ignored. In very deep submicron process, power consumption due to coupling-capacitance dominates (the ration of coupling-capacitance to self-capacitance is 2.4 in 130 nm process and is 3.4 in 45 nm process). On-chip bus occupies considerable portion inside a processor (on-chip buses account for 15% and 30% of total power in Alpha 21064 and Intel 80386, respectively [5]).

1.1.2 Characteristics of Program Execution

Program execution can be classified into 2 categories :

1. Sequential execution :

   This kind of execution occupies about 85-90% portion of program execution.

2. Execution of taken branches :

   This kind of execution occupies about 10-15% portion of program execution. Taken branches can be further classified into 2 classes.

   Fixed target branches : most taken branches are fixed target branches.

Dynamic branch predictor like branch target buffer can handle fixed target branches with target expressed in immediate field of the instruction.

Changing target branches : it includes procedure return, some other special uses that load pc from a register other than link register (e.g., function table, switch conditional statement). Procedure return can be handled by return stack.

1.1.3 What is dynamic branch predictor and its operation

About 15% of instructions in typical programs are branches. Branch instructions can reduce the performance of pipelines by interrupting the normal sequence of program execution, as known as control hazards. While almost all most modern processors use pipelining to achieve high performance, control hazards may cause greater and greater performance loss in proportion to the degree of pipelining.

Dynamic branch predictor is used to help processor resolve the outcome of branch early, thus preventing control dependences from causing stalls [1]. The typical case of dynamic branch predictor is branch target buffer (BTB) [6]. Branch target buffer is used as dynamic branch predictor in this thesis.

Branch target buffer is a branch prediction cache and is designed to reduce branch penalty by predicting the path of the branch and storing information about the branch. The major information stored in each entry of branch target buffer consists of :

1. Valid bit : to tell whether the entry is empty or not.
2. Branch instruction address (branch tag) : the current program counter (PC) is compared to branch instruction address field to determine if there is a “hit”.
3. Branch target address : If there is a hit and the branch is predicted taken, the program counter is loaded with this value and instruction fetching continues from this point.
4. Branch prediction bits (predictor) : 2-bit prediction scheme is most commonly used [1].

For the classic MIPS five-stage pipeline, when the current program counter is sent to instruction memory to fetch the current instruction, this current program counter is also sent to branch target buffer to see if there is a “hit”.

If there is a “miss”, that means there is no valid entry whose branch tag equals the

program counter, the instruction fetcher in CPU will update the program counter to the next sequential PC by adding a word size to the PC. There are 2 scenarios under a "miss" :

1. At the end of the ID stage for the branch instruction, it turns out that this is a not-taken branch :

   The branch processing unit in CPU will not enter a new entry into branch target buffer for this branch, while the instruction fetcher in CPU will keep fetching the subsequent instruction.

   The branch penalty in this case is 0 clock cycle.

2. At the end of the ID stage for the branch instruction, it turns out that this is a taken branch :

   The branch processing unit in CPU will enter a new entry into branch target buffer for this branch, while the instruction fetcher in CPU will kill fetched instruction at IF pipe stage, and start fetching the calculated branch target address at the start of the next clock cycle.

   The branch penalty in this case is 1 clock cycle.

   The detail of entering a new entry into branch target buffer is as follows :

   If the position of the new entry is occupied, some replacement algorithm (e.g., Least Recently Used or Random algorithm) is used to discard an existing entry to make room for this new one.

   In this new entry, valid bit field is set to 1, branch instruction address (branch tag) field is set to the branch instruction address (that is exactly the program counter 1 clock cycle ago), branch target address field is set to the value calculated at the end of ID stage, branch prediction bits field is set to the initialized value according to the adopted n-bit prediction scheme.

If there is a "hit", that means there is a valid entry whose branch tag equals the program counter. There are 4 scenarios under a "hit" :

1. Branch prediction bits predicts this branch instruction is a taken branch, and at the end of ID stage for the branch instruction, it turns out to be a taken branch :

   At the end of the IF stage, branch target buffer will supply the branch target address to update the program counter. At the start of the next clock cycle, this corrected PC that is the branch target address is sent to instruction memory. At the end of the ID stage for the branch instruction, it turns out that the prediction 1 clock cycle ago is correct. The branch processing unit in CPU will update branch prediction bits in branch target buffer, while the instruction fetcher in CPU will

keep fetching the subsequent instruction.

The branch penalty in this case is reduced to 0 clock cycle.

2. Branch prediction bits predicts this branch instruction is a taken branch, but at the end of ID stage for the branch instruction, it turns out to be a not-taken branch :

   At the end of the IF stage, branch target buffer will supply the branch target address to update the program counter. At the start of the next clock cycle, this corrected PC that is the branch target address is sent to instruction memory. At the end of the ID stage for the branch instruction, it turns out that the prediction 1 clock cycle ago is incorrect. The branch processing unit in CPU will update branch prediction bits of the branch entry in branch target buffer, while the instruction fetcher in CPU will kill fetched instruction at IF pipe stage, and start fetching the fall-through address after the branch instruction at the start of the next clock cycle.

   The branch penalty in this case is 1 clock cycle.

3. Branch prediction bits predicts this branch instruction is a not-taken branch, and at the end of ID stage for the branch instruction, it turns out to be a not-taken branch :

   At the end of the IF stage, branch target buffer will do nothing, the instruction fetcher in CPU will update the program counter to the next sequential PC by adding a word size to the PC. At the end of the ID stage for the branch instruction, it turns out that the prediction 1 clock cycle ago is correct. The branch processing unit in CPU will update branch prediction bits of the branch entry in branch target buffer, while the instruction fetcher in CPU will keep fetching the subsequent instruction.

   The branch penalty in this case is 0 clock cycle.

4. Branch prediction bits predicts this branch instruction is a not-taken branch, and at the end of ID stage for the branch instruction, it turns out to be a taken branch :

   At the end of the IF stage, branch target buffer will do nothing, the instruction fetcher in CPU will update the program counter to the next sequential PC by adding a word size to the PC. At the end of the ID stage for the branch instruction, it turns out that the prediction 1 clock cycle ago is incorrect. The branch processing unit in CPU will update branch prediction bits of the branch entry in branch target buffer, while the instruction fetcher in CPU will kill fetched instruction at IF pipe stage, and start fetching the calculated branch target address at the start of the next clock cycle.

The branch penalty in this case is 1 clock cycle.

Summary for the interrelationship among BTB, instruction memory and CPU :

1. When an entry is found in branch target buffer and its prediction is taken, branch target buffer will update the program counter.
2. When there is no entry found in branch target buffer or an entry is found but its prediction is not-taken, the instruction fetcher in CPU will update the program counter to the next sequential PC by adding a word size to the PC.

When a branch instruction is resolved at the end of ID stage, the branch processing unit in CPU will do the following things : updating branch target buffer (including entering a new entry or updating an existing entry) if necessary, flushing the instructions in the wrong path and updating the program counter when the current path is wrong.

1.1.4 Zero-Transition (T0) Bus Encoding Technique

T0 encoding technique [7] makes use of the characteristic of program sequential execution to reduce switch activity on instruction address bus. T0 adds a control line called INC (see Figure 1.1). If the address is consecutive to the previous one, sender asserts INC line and freezes the bus. Otherwise, sender de-asserts INC line and address is transmitted on the bus.

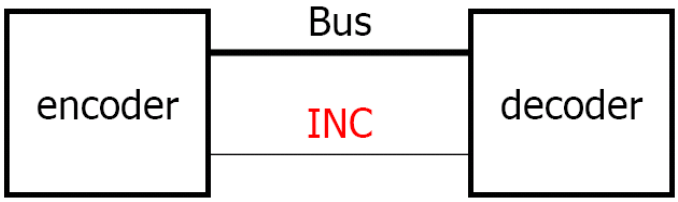


Figure 1.1 Diagram of T0 encoding

1.1.5 T0 with Discontinuous Address Table Bus Encoding Technique [8]

This approach is based on T0 encoding technique and adds a discontinuous address table in both encoder and decoder to record the address pairs that are sent in sequence but with discontinuous values. When two instruction addresses to be transmitted are found in DAT table or not found but consecutive, sender asserts INC line and freezes the

bus. Otherwise, sender de-asserts INC line and address is transmitted on the bus (see Figure 1.2). This approach reduces most of address transmission for taken branch execution, but Content-Addressable-Memory (CAM) is required in both encoder and decoder.

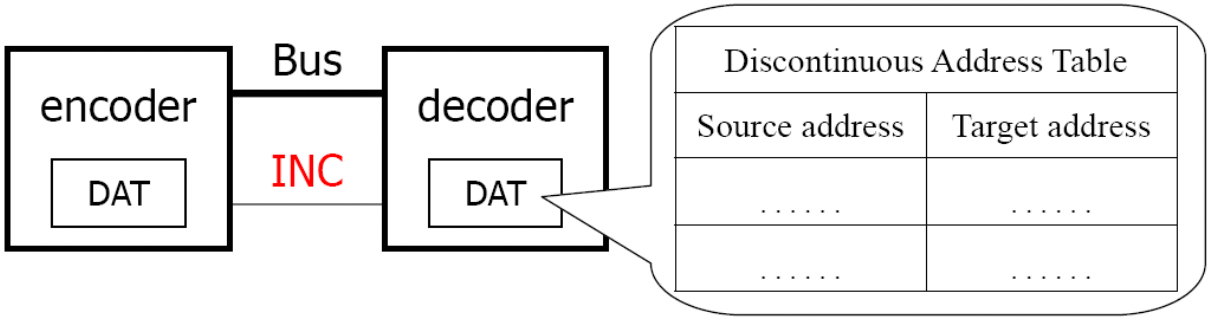


Figure 1.2 Diagram of T0 DAT encoding

1.2 Research Motivation

What we observe from BTB operation is as follows :

1. PC sent to instruction memory is certified by BTB : Branch target or PC+1 word size
2. Action of PC+1 word size is not necessary to be done in CPU
3. When program has been executed for a while, BTB will become steady. Under such “BTB warm up” situation, BTB rarely needs CPU intervention

That is to say, the useful information that CPU passes on to BTB is the result of branch instruction while the useful information that BTB passes on to instruction memory is program counter certified by BTB. The former information is much less than the latter one which comes up every clock cycle.

Under such observation, it may be reasonable to move dynamic branch predictor from CPU to instruction memory side. Such brand-new instruction memory concept can be named “Intelligent Autonomous Instruction Memory” (or called “iAIM” for brevity) due to its program flow tracing capability.

1.3 Research Objective

After applying iAIM concept to conventional computer system design, total execution time, BTB accuracy and Reduction in bus traffic (it includes percentage of reduced instruction address bus active cycles and percentage of reduced bit transitions)

are evaluation metrics on achievement in objective.

Energy conservation on instruction address bus will be evaluated from “bus active cycles” and “bit transitions on bus” metrics indirectly. The evaluation results of conventional design with T0 encoding and with T0 DAT encoding technique under the same BTB organization are given as contrasts.

1.4 Organization of this Thesis

The rest of this thesis is organized as follows. Chapter 2 explains the design detail of iAIM and two other enhanced designs. Chapter 3 presents evaluation methodology, experiment results and discussion. Conclusion and future works are then provided in Chapter 4.

Chapter 2 Design of Proposed Architecture

The design of Intelligent Autonomous Instruction Memory is discussed in this chapter. Section 2.1 and section 2.2 introduce challenges and key ideas in design. Section 2.3 shows the detail of proposed design. Section 2.4 and section 2.5 shows two other enhanced designs.

2.1 Challenges in Design

When branch target buffer is removed from CPU to instruction memory, problems of program flow and BTB maintenance are introduced and need to be solved :

1. How CPU can know branch prediction is correct or not in iAIM.
2. How to enter, update BTB entries in iAIM, that means BTB maintenance can not be handled in CPU directly.
3. How iAIM can know when to use self-generated address and when to use the value on instruction address bus prepared by CPU due to wrong branch prediction or changing target branch.
4. How iAIM can know to pipeline stall happens and keep fetching the same instruction used in previous clock cycle.

2.2 Key Ideas in Design

In conventional architecture, BTB will update program counter in CPU when a predictive taken branch is found. On the contrary, because BTB is inside iAIM, it can not update program counter of CPU. Therefore, it is necessary to add at least one control signal line that iAIM uses to inform CPU of its branch prediction. Similarly, when branch prediction of BTB in iAIM is wrong or some situation like procedure return happens, CPU needs at least one control signal line to inform iAIM of actual branch result and provide correct PC value so that iAIM can supply correct instruction to CPU and do BTB maintenance. Figure 2.1 and Figure 2.2 show block diagrams of conventional architecture and iAIM design respectively.

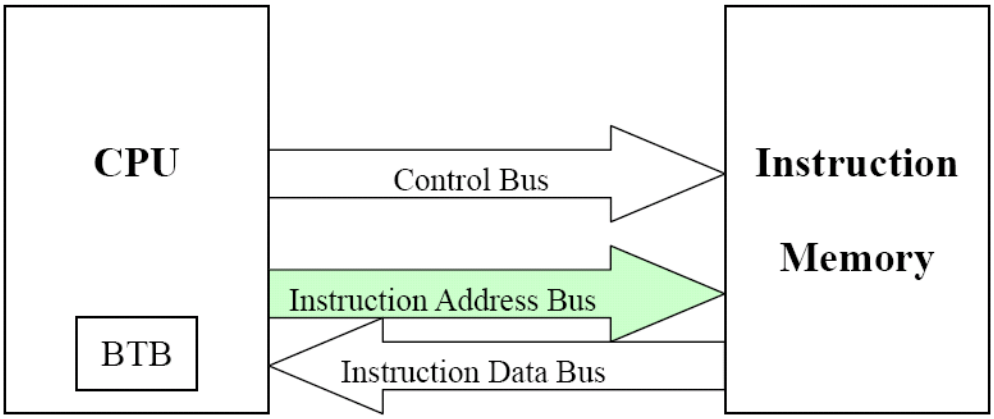


Figure 2.1 Block diagram of conventional architecture

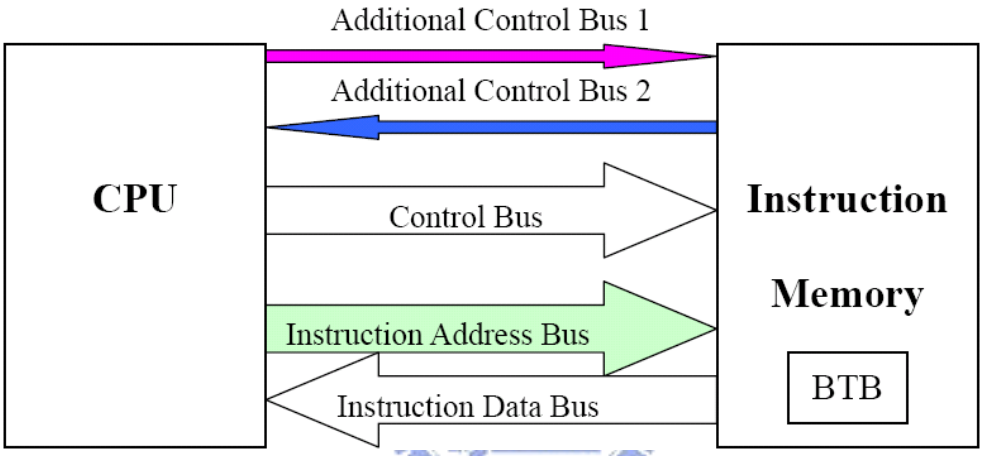


Figure 2.2 Block diagram of iAIM design

Key ideas to implement iAIM design are discussed as follows.

Firstly (Idea 1), iAIM must have instruction address automatic generation mechanism.

Because the philosophy of iAIM is to reduce instruction address traffic between CPU and instruction memory to a minimum, iAIM always tries to generate the next fetched instruction address by itself.

With help of BTB inside iAIM, the branch target address is supplied by BTB when a branch entry is found in BTB and its prediction is taken; otherwise, current used program counter value plus a word size is used at the next coming clock cycle. Therefore, a PC incrementer that that adds a word size to the current PC value is necessary. Figure 2.3 shows automatic instruction address generator inside iAIM.

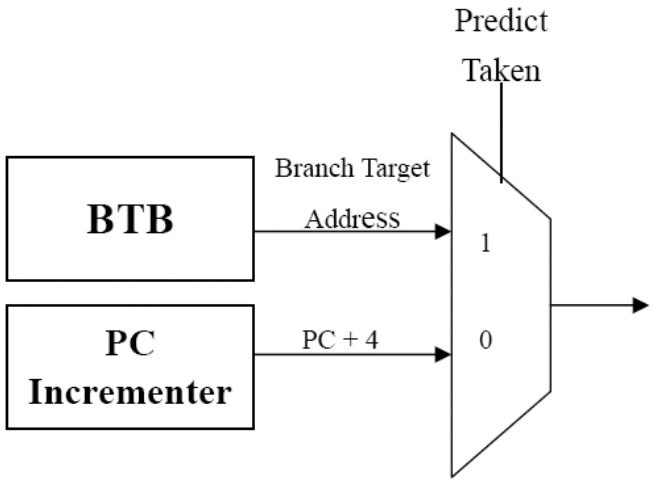


Figure 2.3 Automatic instruction address generator inside iAIM

Secondly (Idea 2), iAIM needs to inform CPU of branch prediction result.

In MIPS pipeline, when a branch entry is found in BTB of iAIM and its prediction is taken, on the next coming clock cycle, iAIM needs to assert a signal to inform CPU that the instruction address used is already replaced by branch target address, not the next sequential PC. At the end of the next coming clock cycle, CPU will resolve the result of branch and know the prediction is correct or not. If the prediction is not correct, CPU needs to take some action to force iAIM to use correct instruction address.

The proposed signal that iAIM uses to inform CPU is one control line called "Predict Taken", or "P-Taken" control line for brevity hereafter :

When a branch entry is found at current clock cycle, this signal is set to 1 at the next coming clock cycle; otherwise, it is set to 0. Figure 2.4 shows this control line that iAIM uses to inform CPU.

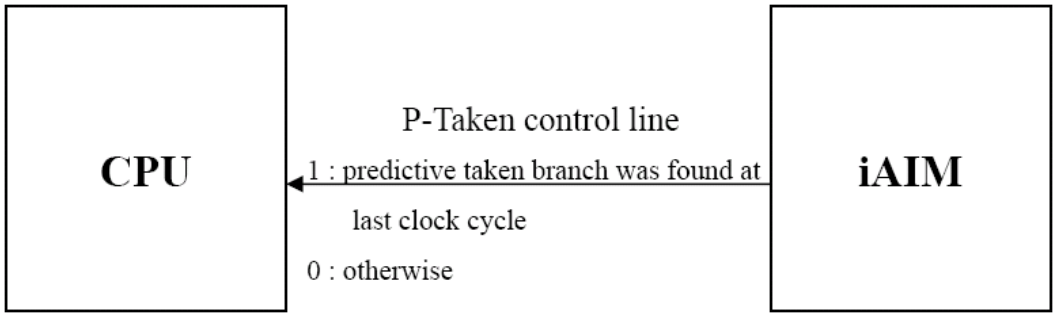


Figure 2.4 Control line that iAIM uses to inform CPU

Thirdly (Idea 3), CPU needs to force iAIM to use correct instruction address

when iAIM's branch prediction is wrong.

In MIPS pipeline, when a branch instruction is resolved at ID stage in CPU, CPU will check if iAIM asserts "P-Taken" line to 1 or not at current clock cycle :

If the prediction is wrong, at the next coming clock cycle CPU will prepare correct instruction address on instruction address bus and inform iAIM of "Wrong Prediction" situation to indicate that branch prediction 2 clock cycles ago is wrong and the instruction address on instruction address bus should be used.

Fourthly (Idea 4), CPU always forces iAIM to use correct instruction address after changing target branch is resolved.

After changing target branch is resolved, CPU will prepare correct instruction address on instruction address bus and inform iAIM of "Compulsory" situation to indicate iAIM to use the instruction address on instruction address bus at current clock cycle.

Fifthly (Idea 5), CPU needs to inform iAIM of the pipeline stall situation.

When pipeline stall happens in conventional architecture, the same instruction address as the one used at last clock cycle will be sent to instruction memory. The reason that CPU needs to inform iAIM of "Pipeline Stall" situation is iAIM has its own instruction address auto-generation mechanism. Such mechanism should cease functioning when pipeline stall happens.

Sixthly (Idea 6), Idea 3, Idea 4 and Idea 5 deal with the situations that iAIM can not use the instruction address generated by its instruction address auto-generation mechanism. CPU needs inform iAIM of "Autonomous" situation to indicate iAIM to use the instruction address generated by its instruction address auto-generation mechanism. This situation also help do BTB maintenance when CPU finds branch prediction in iAIM.

Summarized from idea 3 to idea 6, there are 4 kinds of situations that CPU uses to inform iAIM. In situations of Idea 3 and Idea 4, CPU prepares the instruction address on instruction address bus, and iAIM is forced to use the instruction address on instruction address bus on. In situation of Idea 5, CPU freezes instruction address bus and iAIM uses the same the instruction address as

the one used at last clock cycle. In situation of Idea 6, CPU freezes instruction address bus and iAIM uses the instruction address generated by its instruction address auto-generation mechanism. Two control lines (called “Situation Indication” or “S-Indicate” control lines for brevity hereafter) can be used for CPU to inform iAIM of one of 4 kinds of situations at the beginning of every clock cycle :

00 for “Autonomous” situation,

01 for “Pipeline Stall” situation,

10 for “Wrong Prediction” situation,

11 for “Compulsory” situation.

Figure 2.5 shows S-Indicate control lines that CPU uses to inform iAIM.

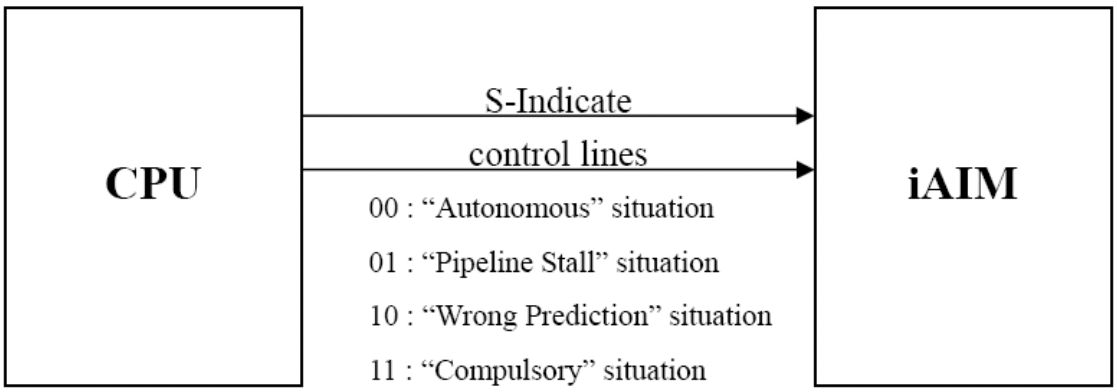


Figure 2.5 S-Indicate control lines that CPU uses to inform iAIM

Seventhly (Idea 7), in order to maintain original BTB operation, two additional 34-bit registers organized as FIFO are necessary :

1. First 34-bit register that store information in iAIM 1 colck cycle ago consists of the following fields :

   32-bit field that stores PC used 1 clock cycles ago (called “PCt-1” for brevity),

   1bit field that stores branch entry found in BTB or not 1 clock cycle ago (called “InBTBt-1” for brevity),

   1bit field that stores taken branch predicted by BTB or not 1 clock cycle ago (called “PTakent-1” for brevity).

2. Second 34-bit register that store information in iAIM 2 colck cycle ago consists of the following fields :

   32-bit field that stores PC used 2 clock cycles ago (called “PCt-2” for brevity),

   1bit field that stores branch entry found in BTB or not 2 clock cycle ago (called “InBTBt-2” for brevity),

   1bit field that stores taken branch predicted by BTB or not 2 clock cycle ago (called “PTakent-2” for brevity).

If a branch instruction enters IF stage at the first clock, CPU will inform iAIM of either “Autonomous” or “Wrong Prediction” situation at the third clock cycle. BTB operation in iAIM is the same as the description of section 1.1.3 in Chapter 1 :

When CPU informs iAIM of “Wrong Prediction” situation at the third clock cycle, there are 2 cases :

Case 1 : InBTBt-2 is 1,

Use PCt-2 as index to do searching in BTB and update its “predictor” field according to PTakent-2 :

If PTakent-2 is 1, update this field toward not-taken direction.

If PTakent-2 is 0, update this field toward taken direction.

Case 2 : InBTBt-2 is 0,

It means no such entry exists in BTB. Enter a new entry into BTB with its initial values listed as below :

“valid bit” field is set to 1,
“branch instruction address” field is set to PCt-2,
“branch target address” field is set to the value on instruction address bus,
“predictor” field is set to the initialized value according to adopted n-bit prediction scheme (it may be weakly-taken in 2 bit prediction scheme).

When CPU informs iAIM of “Autonomous” situation at the third clock cycle, there are 2 cases :

Case 1 : InBTBt-2 is 1,

Use PCt-2 as index to do searching in BTB and update its “predictor” field according to PTakent-2 :

If PTakent-2 is 1, update this field toward taken direction.

If PTakent-2 is 0, update this field toward not-taken direction.

Case 2 : InBTBt-2 is 0,

Do nothing in BTB. Because a not-taken branch will not be entered into BTB if it does not exits before.

2.3 Proposed Design of iAIM

On the basis of key ideas discussed in 2.2, the minimum indispensable elements of Intelligent Autonomous Instruction Memory Design can be derived.

1. Additional control bus between CPU and iAIM

   1) One control line for iAIM to inform CPU of predicting taken(called "Predict Taken" or "P-Taken" control line)

   2) Two control lines for CPU to inform iAIM of one of 4 kinds of situations(called "Situation Indication" or "S-Indiacte" control lines) :

      00 for "Autonomous" situation,

      01 for "Pipeline Stall" situation,

      10 for "Wrong Prediction" situation,

      11 for "Compulsory" situation.

   Figure 2.6 shows buses between CPU and iAIM.

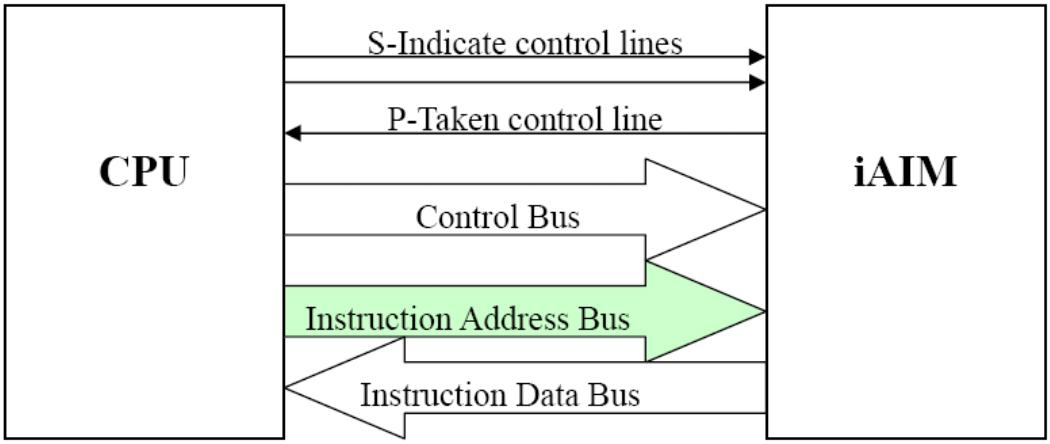


Figure 2.6 Buses between CPU and iAIM

2. Additional circuit in iAIM

   1) A incrementer called "PC Incrementer" that add PC used at last clock cycle by a word size is used to generate next sequential instruction address.

   2) A multiplexer called "PC MUX" is used to select one of 4 kinds of instruction address sources :

      i. Last PC plus a word size for sequential execution,

      ii. Branch target address for BTB's taken branch prediction,

iii. Last PC for pipeline stall,

iv. Compulsory PC address sent from CPU.

3) Two 34-bit Registers are organized as FIFO as follows :

i. First 34-bit register that store information in iAIM 1 colck cycle ago consists of the following fields :

32-bit field that stores PC used 1 clock cycles ago (called “PCt-1” for brevity),

1bit field that stores branch entry found in BTB or not 1 clock cycle ago (called “InBTBt-1” for brevity),

1bit field that stores taken branch predicted by BTB or not 1 clock cycle ago (called “PTakent-1” for brevity).

ii. Second 34-bit register that store information in iAIM 2 colck cycle ago consists of the following fields :

32-bit field that stores PC used 2 clock cycles ago (called “PCt-2” for brevity),

1bit field that stores branch entry found in BTB or not 2 clock cycle ago (called “InBTBt-2” for brevity),

1bit field that stores taken branch predicted by BTB or not 2 clock cycle ago (called “PTakent-2” for brevity).

Figure 2.7 shows additional circuit in iAIM.

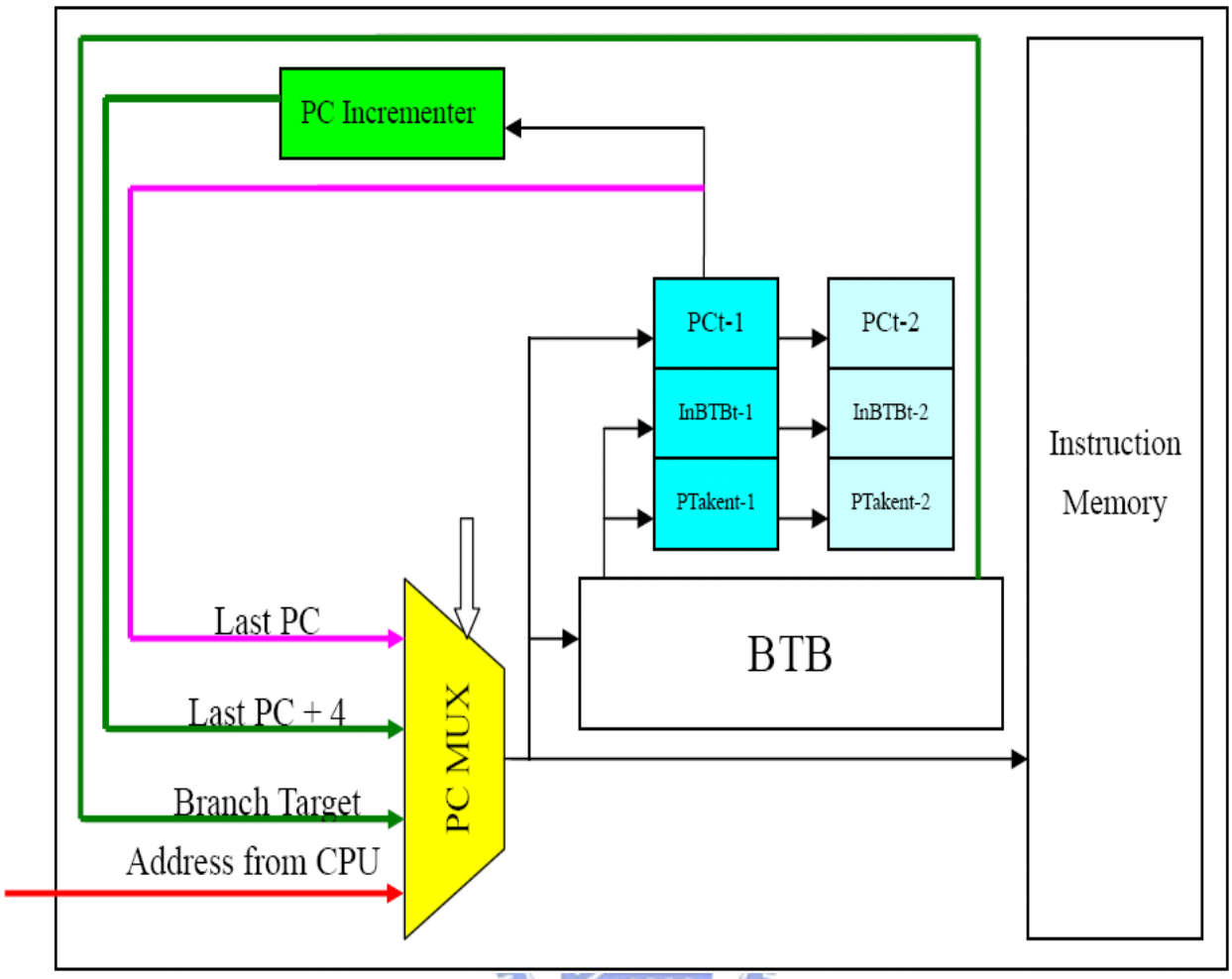


Figure 2.7 Additional circuit in iAIM

3. Control signal description

1) From iAIM to CPU :

There is a “P-Taken” control line used to inform CPU of taken branch prediction in iAIM.

The algorithm of its sending timing is described as below.

```
P-Taken sending algorithm {
    unsigned int iAIM_PC = PC used in iAIM at current clock cycle, P-Taken;
    Use iAIM_PC as index to do searching in BTB of iAIM;
    if (branch entry is found) {
       if (predictor field predicts taken) {
          P-Taken = 0b1 at next clock cycle;
       } else {
          P-Taken = 0b0 at next clock cycle;
       }
    } else {
       P-Taken = 0b0 at next clock cycle;
    }
}
```

2) Form CPU to iAIM :

There are 4 kinds of situations “Autonomous”, “Pipeline Stall”, “Wrong Prediction”, “Compulsory” used to inform iAIM of various situations detected in CPU.

```
S-Indicate sending algorithm {
    unsigned int force_PC = PC value used at current clock cycle, S-Indicate;
    unsigned int target = target address resolved at current clock cycle;
    unsigned int fall_through = fall-through address resolved at current clock cycle;
    unsigned int P-Taken = value of P-Taken control line at current clock cycle;
    CPU detects the following situations :
    Case 1 : when changing target branch is resolved in ID stage
        S-Indicate = 0b11 ("Compulsory") at next clock cycle;
        Instruction address bus at current clock cycle = force_PC;
    Case 2 : when jump instruction(except J, JAL) is resolved in ID stage
        S-Indicate = 0b11 ("Compulsory") at next clock cycle;
        Instruction address bus at next clock cycle = target;
    Case 3 : when CPU detects pipeline stall
        S-Indicate = 0b01 ("Pipeline Stall") at current clock cycle;
        Instruction address bus is frozen at current clock cycle;
    Case 4 : when branch instruction(including J, JAL) is resolved in ID stage
            if (the result is a taken branch) {
               if (P-Taken = 0b1 at the start of current clock cycle) {
                  S-Indicate = 0b00 ("Autonomous") at next clock cycle;
                  Instruction address bus is frozen at next clock cycle;
               } else {
                  S-Indicate = 0b10 ("Wrong Prediction") at next clock cycle;
                  Instruction address bus at next clock cycle = target;
               }
            } else {
               if (P-Taken = 0b1 at the start of current clock cycle) {
                  S-Indicate = 0b10 ("Wrong Prediction") at next clock cycle;
                  Instruction address bus at next clock cycle = fall_through;
               } else {
                  S-Indicate = 0b00 ("Autonomous") at next clock cycle;
                  Instruction address bus is frozen at next clock cycle;
               }
            }
    Case 5 : when CPU can not detect any one of the above situations
        S-Indicate = 0b00 ("Autonomous") at next clock cycle;
        Instruction address bus is frozen at next clock cycle;
}
```

4. Action algorithm of CPU

```
CPU action algorithm {
    unsigned int force_PC = PC value used at current clock cycle;
    unsigned int target = target address resolved at current clock cycle;
    unsigned int fall_through = fall-through address resolved at current clock cycle;
    unsigned int S-Indicate; /* value on S-Indicate control line */
    unsigned int P-Taken = value on P-Taken control line at current clock cycle;
    if (pipeline stall is detected) {
      S-Indicate = 0b01 ("Pipeline Stall") at current clock cycle
      Instruction address bus is frozen at current clock cycle;
    } else if (changing target branch is resolved in ID stage) {
      S-Indicate = 0b11 ("Compulsory") at next clock cycle;
      Instruction address bus at current clock cycle = force_PC;
    } else if (when jump instruction(except J, JAL) is resolved in ID stage) {
      S-Indicate = 0b11 ("Compulsory") at next clock cycle;
      Instruction address bus at next clock cycle = target;
    } else if (when branch instruction(including J, JAL) is resolved in ID stage) {
      if (the result is a taken branch) {
        if (P-Taken = 0b1 at the start of current clock cycle) {
          S-Indicate = 0b00 ("Autonomous") at next clock cycle;
          Instruction address bus is frozen at next clock cycle;
        } else {
          S-Indicate = 0b10 ("Wrong Prediction") at next clock cycle;
          Instruction address bus at next clock cycle = target;
        }
      } else {
        if (P-Taken = 0b1 at the start of current clock cycle) {
          S-Indicate = 0b10 ("Wrong Prediction") at next clock cycle;
          Instruction address bus at next clock cycle = fall_through;
        } else {
          S-Indicate = 0b00 ("Autonomous") at next clock cycle;
          Instruction address bus is frozen at next clock cycle;
        }
      }
    } else {
      S-Indicate = 0b00 ("Autonomous") at next clock cycle;
      Instruction address bus is frozen at next clock cycle;
    }
}
```

5. Action algorithm of iAIM

```
iAIM action algorithm {
    unsigned int force_PC = PC value on instruction address bus used at current clock cycle;
    unsigned int btb_target = target address found in BTB 1 clock cycle ago;
    unsigned int S-Indicate = value on S-Indicate control lines at current clock cycle;
    unsigned int P-Taken; /* value on P-Taken control line */
    unsigned int iAIM_PC; /* PC used in iAIM at current clock cycle */
    /* PC used 1 clock cycle ago, 2 clock cycles ago */
    unsigned int PCt-1, PCt-2;
    /* branch entry found in BTB or not 1 clock cycle ago, 2 clock cycles ago*/
    unsigned int InBTBt-1, InBTBt-2;
    /* taken branch predicted by BTB or not 1 clock cycle ago, 2 clock cycles ago */
    unsigned int PTakent-1, PTakent-2;
    if (S-Indicate = 0b01 ("Pipeline Stall")) {
      iAIM_PC = PCt-1;
    } else if (S-Indicate = 0b11 ("Compulsory")) {
      iAIM_PC = force_PC;
    } else if (S-Indicate = 0b10 ("Wrong Prediction")) {
      iAIM_PC = force_PC;
      If (InBTBt-2 = 0b1) {
        Update the predictor field of entry in BTB where branch address is PCt-2:
        If (PTakent-2 = 0b1)
          Update predictor toward not-taken;
        Else
          Update predictor toward taken;
      } else {
        Enter an entry into BTB :
        branch address = PCt-2;
        branch target address = force_PC;
        Other fields are set to default values;
      }
    } else {
      /* To be continued on next page */
}
```

```
iAIM action algorithm (continued) {
        if (IB1 = 0b1) {
           iAIM_PC = btb_target;
        }
        else {
           iAIM_PC = PCt-1 + 4;
        }
        if (InBTBt-2 = 0b1) {
           Update the predictor field of entry in BTB where branch address is PCt-2:
           If (PTakent-2 = 0b1) {
              Update predictor toward taken;
           }
           Else {
              Update predictor toward not-taken;
           }
        }
     }
     if (S-Indicate = 0b01 ("Pipeline Stall"))
     {
        PCt-2, InBTBt-2, PTakent-2 remain unchanged;
     } else {
        PCt-2 = PCt-1; InBTBt-2 = InBTBt-1; PTakent-2 = PTakent-1;
     }
     PCt-1 = iAIM_PC;
     Using iAIM_PC as index to do searching in BTB;
     if (branch entry is found) {
        if (predictor field predicts taken) {
           P-Taken = 0b1 at next clock cycle; InBTBt-1 = 0b1; PTakent-1 = 0b1;
        } else {
           P-Taken = 0b0 at next clock cycle; InBTBt-1 = 0b1; PTakent-1 = 0b0;
        }
     } else {
        InBTBt-1 = 0b0; PTakent-1 = 0b0;
        P-Taken = 0b0 at next clock cycle;
     }
}
```

2.4 Proposed Design of Enhanced iAIM with Partial Decoder

In design proposed in section 2.3, when CPU finds branch prediction in iAIM is wrong, it needs to prepare corrected address on instruction address bus at the next clock cycle. This does increase bit transitions on instruction address bus and can be avoided if the design proposed in section 2.3 is further enhanced with a partial decoder.

The design idea of partial decoder is described as below :

- Partial decoder is capable of identifying branch instruction (including J, JAL) and calculating its branch target address and fall-through address by associating simple logics.
- When iAIM is instructed by CPU with "S-Indicate" control lines equaling 0b10 ("Wrong Prediction"), it will check PTakent-2 value :

  If PTakent-2 equals 0b1, iAIM uses fall-through address calculated 2 clock cycles ago. Otherwise, it uses branch target address calculated 2 clock cycles ago.

1. Additional circuit in enhanced iAIM with partial decoder

   1) A partial decoder (called "PD" for brevity): After instruction is fetched by instruction memory, this instruction is not only sent to instruction data bus bus also sent to PD. PD is capable of identifying branch instruction (including J, JAL) and calculating its branch target address and fall-through address by associating simple logics before the end of clock cycle.

   2) Two registers are required to stores calculated branch target address and fall-through address as follows :

      i. A register stores branch target address (called "Target" for brevity),

      ii. A register stores fall-through address (called "FallThru" for brevity).

      Figure 2.8 shows action timing of partial decoder.

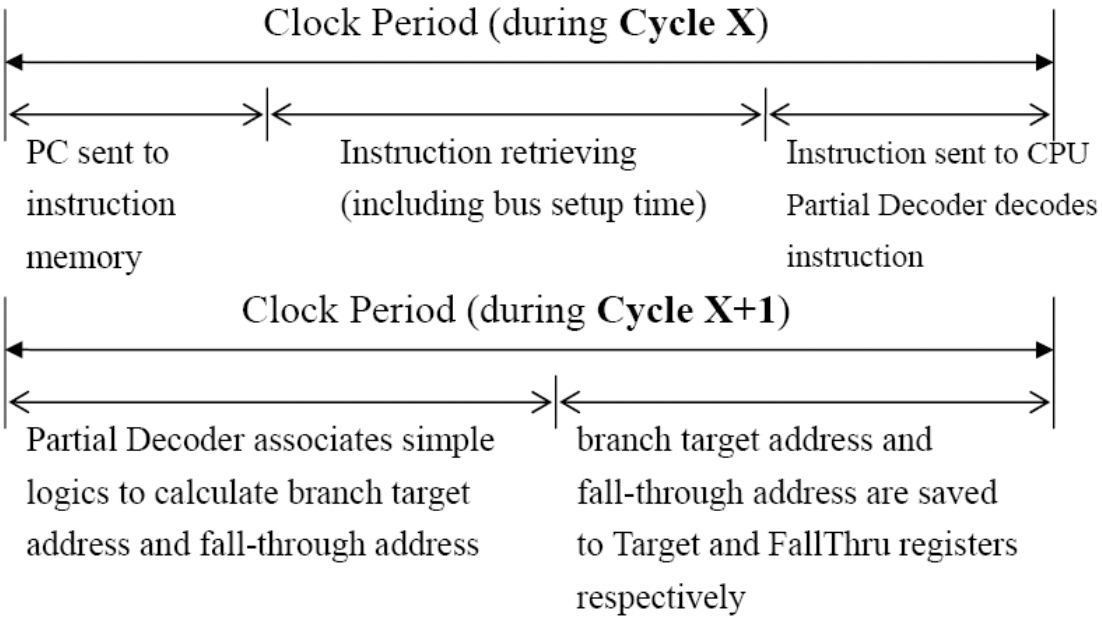


Figure 2.8 Action timing of partial decoder

3) A multiplexer called "PC MUX" is augmented to select additional 2 kinds of instruction address sources :

   v. Target,

   vi. FallThru.

Figure 2.9 shows additional circuit in enhanced iAIM with partial decoder

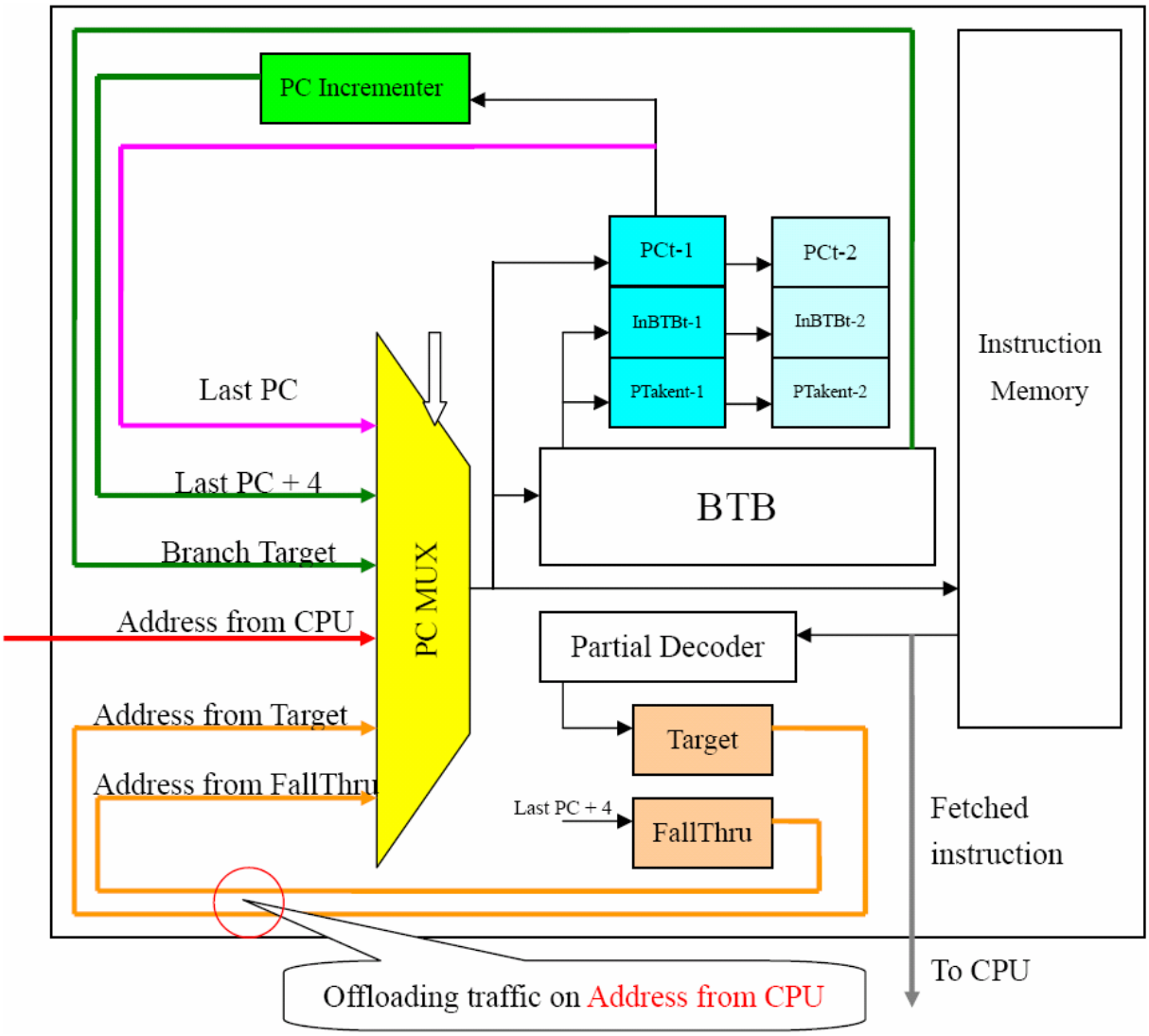


Figure 2.9 Additional circuit in enhanced iAIM with partial decoder

2. Control signal description

1) From iAIM to CPU :

The same as the one in section 2.3.

2) Form CPU to iAIM :

The same as the one in section 2.3 except underscored words in case 4.

```
Case 4 : when branch instruction(including J, JAL) is resolved in ID stage
        if (the result is a taken branch) {
           if (P-Taken = 0b1 at the start of current clock cycle) {
              S-Indicate = 0b00 ("Autonomous") at next clock cycle;
              Instruction address bus is frozen at next clock cycle;
           } else {
              S-Indicate = 0b10 ("Wrong Prediction") at next clock cycle;
              Instruction address bus is frozen at current clock cycle;
           }
        } else {
           if (P-Taken = 0b1 at the start of current clock cycle) {
              S-Indicate = 0b10 ("Wrong Prediction") at next clock cycle;
              Instruction address bus is frozen at current clock cycle;
           } else {
              S-Indicate = 0b00 ("Autonomous") at next clock cycle;
              Instruction address bus is frozen at next clock cycle;
           }
        }
```

3. Action algorithm of CPU

The same as the one in section 2.3 except underscored words.

```
CPU action algorithm {
    unsigned int force_PC = PC value used at current clock cycle;
    unsigned int target = target address resolved at current clock cycle;
    unsigned int fall_through = fall-through address resolved at current clock cycle;
    unsigned int S-Indicate; /* value on S-Indicate control line */
    unsigned int P-Taken = value on P-Taken control line at current clock cycle;
    if (pipeline stall is detected) {
      S-Indicate = 0b01 ("Pipeline Stall") at current clock cycle
      Instruction address bus is frozen at current clock cycle;
    } else if (changing target branch is resolved in ID stage) {
      S-Indicate = 0b11 ("Compulsory") at next clock cycle;
      Instruction address bus at current clock cycle = force_PC;
    } else if (when jump instruction(except J, JAL) is resolved in ID stage) {
      S-Indicate = 0b11 ("Compulsory") at next clock cycle;
      Instruction address bus at next clock cycle = target;
    } else if (when branch instruction(including J, JAL) is resolved in ID stage) {
      if (the result is a taken branch) {
        if (P-Taken = 0b1 at the start of current clock cycle) {
          S-Indicate = 0b00 ("Autonomous") at next clock cycle;
          Instruction address bus is frozen at next clock cycle;
        } else {
          S-Indicate = 0b10 ("Wrong Prediction") at next clock cycle;
          Instruction address bus is frozen at next clock cycle;
        }
      } else {
        if (P-Taken = 0b1 at the start of current clock cycle) {
          S-Indicate = 0b10 ("Wrong Prediction") at next clock cycle;
          Instruction address bus is frozen at next clock cycle;
        } else {
          S-Indicate = 0b00 ("Autonomous") at next clock cycle;
          Instruction address bus is frozen at next clock cycle;
        }
      }
    } else {
      S-Indicate = 0b00 ("Autonomous") at next clock cycle;
      Instruction address bus is frozen at next clock cycle;
    }
}
```

4. Action algorithm of iAIM

The same as the one in section 2.3 except underscored words.

```
iAIM action algorithm {
    unsigned int force_PC = PC value on instruction address bus used at current clock cycle;
    unsigned int btb_target = target address found in BTB 1 clock cycle ago;
    unsigned int S-Indicate = value on S-Indicate control lines at current clock cycle;
    unsigned int P-Taken; /* value on P-Taken control line */
    unsigned int iAIM_PC; /* PC used in iAIM at current clock cycle */
    /* PC used 1 clock cycle ago, 2 clock cycles ago */
    unsigned int PCt-1, PCt-2;
    /* branch entry found in BTB or not 1 clock cycle ago, 2 clock cycles ago*/
    unsigned int InBTBt-1, InBTBt-2;
    /* taken branch predicted by BTB or not 1 clock cycle ago, 2 clock cycles ago */
    unsigned int PTakent-1, PTakent-2;
    /* branch target address calculated in iAIM */
    unsigned int Target;
    /* fall-through address calculated in iAIM*/
    unsigned int FallThru;
    if (S-Indicate = 0b01 ("Pipeline Stall")) {
      iAIM_PC = PCt-1;
    } else if (S-Indicate = 0b11 ("Compulsory")) {
      iAIM_PC = force_PC;
    } else if (S-Indicate = 0b10 ("Wrong Prediction")) {
      If (PTakent-2 = 0b1) {
        iAIM_PC = FallThru;
      } else {
        iAIM_PC = Target;
      }
      If (InBTBt-2 = 0b1) {
        Update the predictor field of entry in BTB where branch address is PCt-2:
        If (PTakent-2 = 0b1)
          Update predictor toward not-taken;
        Else
          Update predictor toward taken;
      } else {
        Enter an entry into BTB :
        branch address = PCt-2;
        branch target address = Target;
        Other fields are set to default values;
      }
    } else {
      /* To be continued on next page */
}
```

```
iAIM action algorithm (continued) {
        if (InBTBt-1 = 0b1) {
           iAIM_PC = btb_target;
        }
        else {
           iAIM_PC = PCt-1 + 4;
        }
        if (InBTBt-2 = 0b1) {
           Update the predictor field of entry in BTB where branch address is PCt-2:
           If (PTakent-2 = 0b1) {
              Update predictor toward taken;
           }
           Else {
              Update predictor toward not-taken;
           }
        }
     }
     if (S-Indicate = 0b01 ("Pipeline Stall"))
     {
        PCt-2, InBTBt-2, PTakent-2 remain unchanged;
     } else {
        PCt-2 = PCt-1; InBTBt-2 = InBTBt-1; PTakent-2 = PTakent-1;
     }
     PCt-1 = iAIM_PC;
     Using iAIM_PC as index to do searching in BTB;
     if (branch entry is found) {
        if (predictor field predicts taken) {
           P-Taken = 0b1 at next clock cycle; InBTBt-1 = 0b1; PTakent-1 = 0b1;
        } else {
           P-Taken = 0b0 at next clock cycle; InBTBt-1 = 0b1; PTakent-1 = 0b0;
        }
     } else {
        InBTBt-1 = 0b0; PTakent-1 = 0b0;
        P-Taken = 0b0 at next clock cycle;
     }
     /* To be continued on next page */
}
```

```
iAIM action algorithm (continued) {
    Use partial decoder PD to decode fetched instruction from instruction memory;
    if (fetched instruction is branch instruction(including J, JAL)) {
        /* The following actions will be completed before the end of next clock cycle */
        Target = branch target address calculated in iAIM;
        FallThru = fall-through address calculated in iAIM;
    }
}
```

2.5 Proposed Design of Enhanced iAIM with Partial Decoder and Return Stack

The design proposed in section 2.4 can be further enhanced by implementing return stack inside iAIM. The purpose of equipping iAIM with return stack is to eliminate target address traffic due to procedure return instructions which occupy most portion of changing target branches.

The design idea of return stack is described as below :

- Partial decoder is augmented to be capable of identifying procedure call instructions (JAL and JALR) and procedure return instructions (JR to r31).
- When a procedure call instruction is resolved in partial decoder, the instruction address following the procedure call instruction is pushed into return stack.
- When a procedure return instruction is resolved in partial decoder, the instruction address used at the next clock cycle is popped from return stack.

The following design has an assumption that the size of return stack is big enough to accommodate the maximum depth of procedure call for all applications running on it. In reality, return stack can not be infinite. In the end of this section, one of a workable mechanism to deal with finite return stack will be proposed.

Figure 2.10 shows action timing and algorithm for procedure call handling. Figure 2.11 shows action timing and algorithm for procedure return handling.

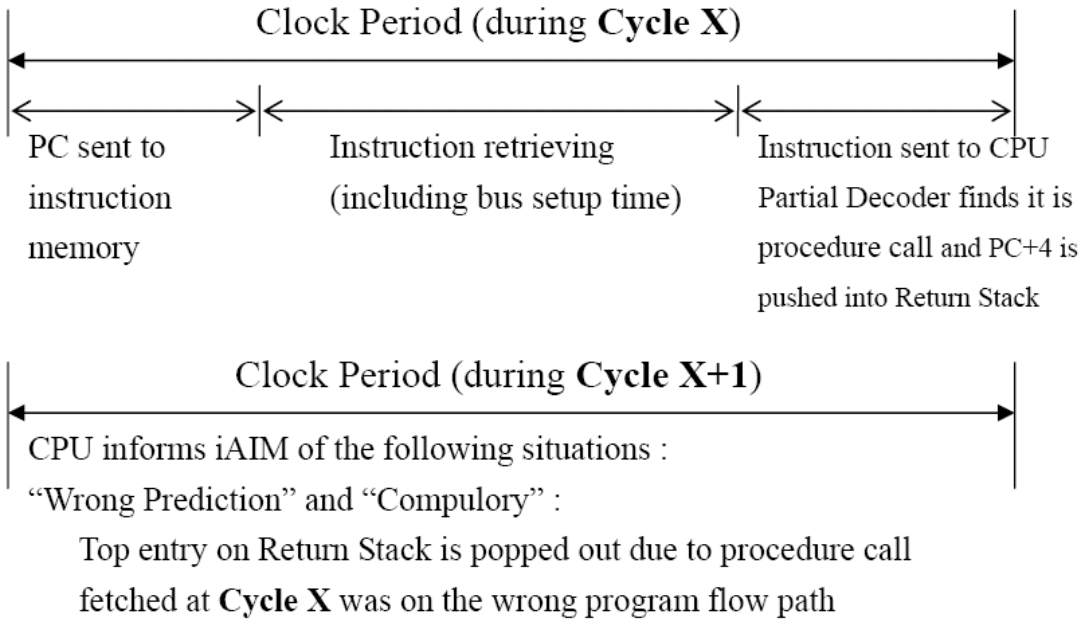


Figure 2.10 Action timing and algorithm for procedure call handling

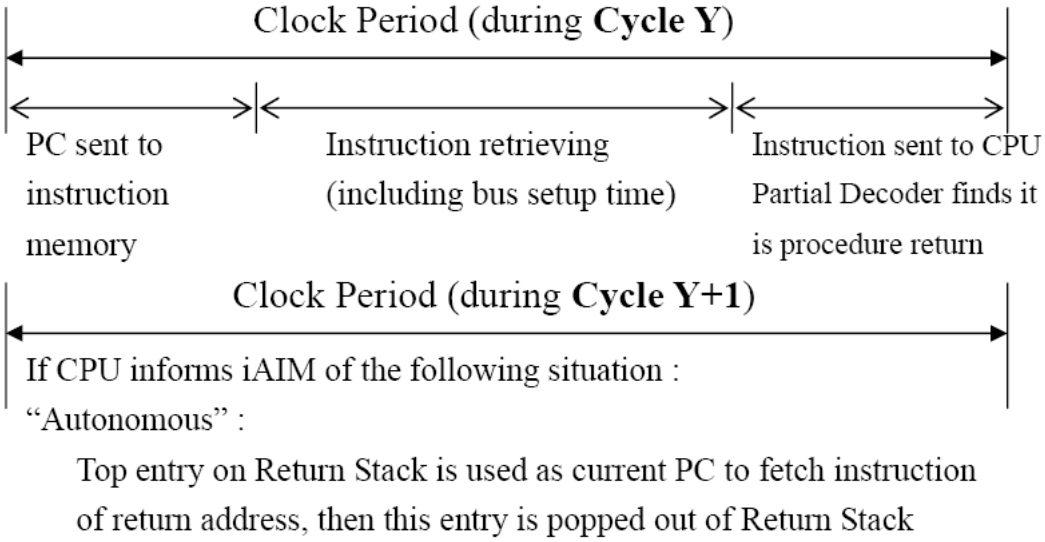


Figure 2.11 Action timing and algorithm for procedure return handling

1. Additional circuit in enhanced iAIM with partial decoder and return stack
   1) A return stack (called "RS" for brevity)
   2) A partial decoder (called "PD" for brevity) described in section 2.4 is enhanced : After instruction is fetch by instruction memory, this instruction is sent to PD. PD is capable of identifying procedure call instructions (JAL and JALR) and procedure return instructions (JR to r31) . When a procedure call instruction is resolved, PD pushes the instruction address following the procedure call instruction into RS. When a procedure return instruction is resolved, the instruction address used at the next clock cycle is popped from RS.
   3) A multiplexer called "PC MUX" is augmented to select additional 1 kind of instruction address source :

      vii. top entry of RS

Figure 2.12 shows additional circuit in enhanced iAIM with partial decoder and return stack.

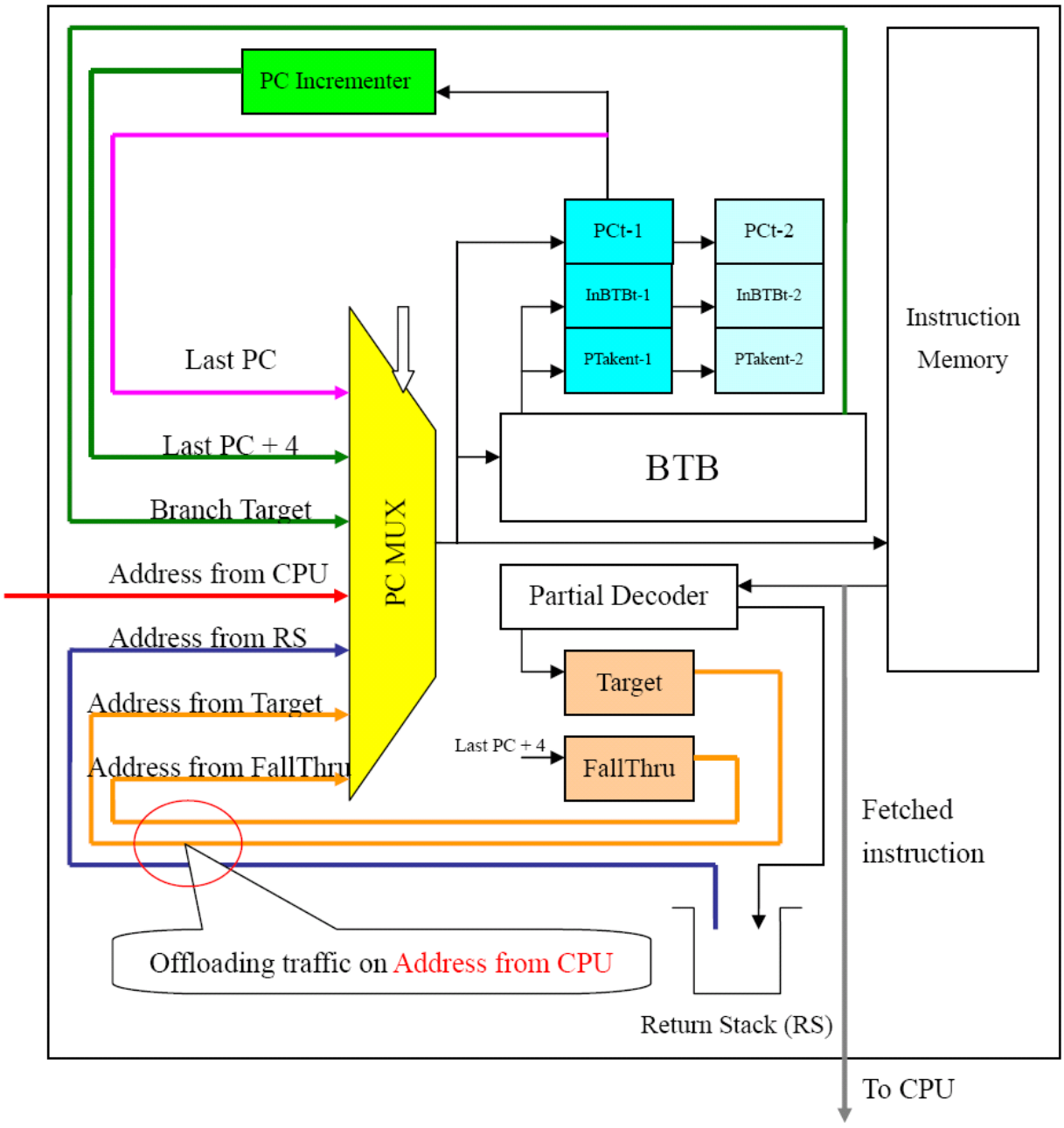


Figure 2.12 Additional circuit in enhanced iAIM with partial decoder and return stack

2. Control signal description

1) From iAIM to CPU :

The same as the one in section 2.3.

2) Form CPU to iAIM :

The same as the one in section 2.4 except underscored words in case 2.

Case 2 : when jump instruction(except J, JAL) is resolved in ID stage

if (jump instruction is JR to r31) {

S-Indicate = 0b00 ("Autonomous") at next clock cycle;

Instruction address bus is frozen at next clock cycle;

} else {

S-Indicate = 0b11 ("Compulsory") at next clock cycle;

Instruction address bus at next clock cycle = target;

}

3. Action algorithm of CPU

The same as the one in section 2.4 except underscored words.

```
CPU action algorithm {
    unsigned int force_PC = PC value used at current clock cycle;
    unsigned int target = target address resolved at current clock cycle;
    unsigned int fall_through = fall-through address resolved at current clock cycle;
    unsigned int S-Indicate; /* value on S-Indicate control line */
    unsigned int P-Taken = value on P-Taken control line at current clock cycle;
    if (pipeline stall is detected) {
      S-Indicate = 0b01 ("Pipeline Stall") at current clock cycle
      Instruction address bus is frozen at current clock cycle;
    } else if (changing target branch is resolved in ID stage) {
      S-Indicate = 0b11 ("Compulsory") at next clock cycle;
      Instruction address bus at current clock cycle = force_PC;
    } else if (when jump instruction(except J, JAL) is resolved in ID stage) {
      if (jump instruction is JR to r31) {
        S-Indicate = 0b00 ("Autonomous") at next clock cycle;
        Instruction address bus is frozen at next clock cycle;
      } else {
        S-Indicate = 0b11 ("Compulsory") at next clock cycle;
        Instruction address bus at next clock cycle = target;
      }
    } else if (when branch instruction(including J, JAL) is resolved in ID stage) {
      if (the result is a taken branch) {
        if (P-Taken = 0b1 at the start of current clock cycle) {
          S-Indicate = 0b00 ("Autonomous") at next clock cycle;
          Instruction address bus is frozen at next clock cycle;
        } else {
          S-Indicate = 0b10 ("Wrong Prediction") at next clock cycle;
          Instruction address bus is frozen at next clock cycle;
        }
      } else {
        if (P-Taken = 0b1 at the start of current clock cycle) {
          S-Indicate = 0b10 ("Wrong Prediction") at next clock cycle;
          Instruction address bus is frozen at next clock cycle;
        } else {
          S-Indicate = 0b00 ("Autonomous") at next clock cycle;
          Instruction address bus is frozen at next clock cycle;
        }
      }
    } else {
      S-Indicate = 0b00 ("Autonomous") at next clock cycle;
      Instruction address bus is frozen at next clock cycle;
    }
}
```

4. Action algorithm of iAIM

The same as the one in section 2.4 except underscored words.

```
iAIM action algorithm {
    unsigned int force_PC = PC value on instruction address bus used at current clock cycle;
    unsigned int btb_target = target address found in BTB 1 clock cycle ago;
    unsigned int S-Indicate = value on S-Indicate control lines at current clock cycle;
    unsigned int P-Taken; /* value on P-Taken control line */
    unsigned int iAIM_PC; /* PC used in iAIM at current clock cycle */
    /* PC used 1 clock cycle ago, 2 clock cycles ago */
    unsigned int PCt-1, PCt-2;
    /* branch entry found in BTB or not 1 clock cycle ago, 2 clock cycles ago*/
    unsigned int InBTBt-1, InBTBt-2;
    /* taken branch predicted by BTB or not 1 clock cycle ago, 2 clock cycles ago */
    unsigned int PTakent-1, PTakent-2;
    /* branch target address calculated in iAIM */
    unsigned int Target;
    /* fall-through address calculated in iAIM*/
    unsigned int FallThru;
    if (S-Indicate = 0b01 ("Pipeline Stall")) {
      iAIM_PC = PCt-1;
    } else if (S-Indicate = 0b11 ("Compulsory")) {
      iAIM_PC = force_PC;
    } else if (S-Indicate = 0b10 ("Wrong Prediction")) {
      If (PTakent-2 = 0b1) {
        iAIM_PC = FallThru;
      } else {
        iAIM_PC = Target;
      }
      If (InBTBt-2 = 0b1) {
        Update the predictor field of entry in BTB where branch address is PCt-2:
        If (PTakent-2 = 0b1)
          Update predictor toward not-taken;
        Else
          Update predictor toward taken;
      } else {
        Enter an entry into BTB :
        branch address = PCt-2;
        branch target address = Target;
        Other fields are set to default values;
      }
    } else {
      /* To be continued on next page */
}
```

```
iAIM action algorithm (continued) {
        if (procedure return instruction (JR to r31) was resolved in partial
           decoder PD at last clock cycle) {
           iAIM_PC = return address popped from return stack RS;
        } else {
           if (InBTBt-1 = 0b1) {
              iAIM_PC = btb_target;
           }
           else {
              iAIM_PC = PCt-1 + 4;
           }
           if (InBTBt-2 = 0b1) {
              Update the predictor field of entry in BTB where branch address is PCt-2:
              If (PTakent-2 = 0b1) {
                 Update predictor toward taken;
              }
              Else {
                 Update predictor toward not-taken;
              }
           }
        }
     }
     if (S-Indicate = 0b10 ("Wrong Prediction") or S-Indicate = 0b11 ("Compulsory")) {
        if (procedure call instruction (JAL, JALR) was resolved in partial
           decoder PD at last clock cycle) {
           /* due to wrong branch prediction or changing target branch */
           Popped a return address out of return stack RS;
        }
     }
     /* To be continued on next page */
}
```

```
iAIM action algorithm (continued) {
    if (S-Indicate = 0b01 ("Pipeline Stall"))
    {
      PCt-2, InBTBt-2, PTakent-2 remain unchanged;
    } else {
      PCt-2 = PCt-1; InBTBt-2 = InBTBt-1; PTakent-2 = PTakent-1;
    }
    PC1 = iAIM_PC;
    Using iAIM_PC as index to do searching in BTB;
    if (branch entry is found) {
      if (predictor field predicts taken) {
        P-Taken = 0b1 at next clock cycle; InBTBt-1 = 0b1; PTakent-1 = 0b1;
      } else {
        P-Taken = 0b0 at next clock cycle; InBTBt-1 = 0b1; PTakent-1 = 0b0;
      }
    } else {
      InBTBt-1 = 0b0; PTakent-1 = 0b0;
      P-Taken = 0b0 at next clock cycle;
    }
  Use partial decoder PD to decode fetched instruction from instruction memory;
  if (fetched instruction is branch instruction(including J, JAL)) {
    /* The following actions will be completed before the end of next clock cycle */
    Target = branch target address calculated in iAIM;
    FallThru = fall-through address calculated in iAIM;
  }
  if (fetched instruction is procedure call instruction(JAL, JALR)) {
    (iAIM_PC + 4) is pushed into return stack RS;
  } else if (fetched instruction is procedure return instruction(JR to r31)) {
    return address will be popped from return stack RS at next clock cycle;
  }
}
```

For research purpose, the above design assumes return stack is big enough to accommodate the maximum procedure depth of applications running on the processor. In reality, return stack has limited entries, more procedure calls than return stack entries can

corrupt return stack, which may be implemented as a finite depth push-down stack. Figure 2.13 shows scenario of return stack overflow in a finite depth push-down stack. Thereafter, corresponding procedure returns cause underflow by popping empty stack (see [9]). Some researches have proposed the backup storage solution to augment limited return stack size to very large number. And some research (in [9]) has proposed the protection mechanism to prevent "underflow" problem either in return stack and backup storage.

A simple proposed solution to deal with "underflow" in proposed design of section 2.5 with finite return stack size is described as below :

1. iAIM needs to inform CPU by some control signal when a procedure return instruction is resolved but the return stack is empty. "P-Taken" control line can be used because branch instruction and procedure return instruction are mutually exclusive. When return stack underflow happens, iAIM asserts "P-Taken" control line to 1 at the next clock cycle.
2. When CPU finds iAIM asserts "P-Taken" control line to 1 and at the same clock cycle a procedure return instruction is resolved, CPU will set "S-Indicate" control lines to indicate "Compulsory" situation and prepare the return address on instruction address at the beginning of the next clock cycle.
3. iAIM will use the return address on instruction address when CPU sets "S-Indicate" control lines to indicate "Compulsory" situation.

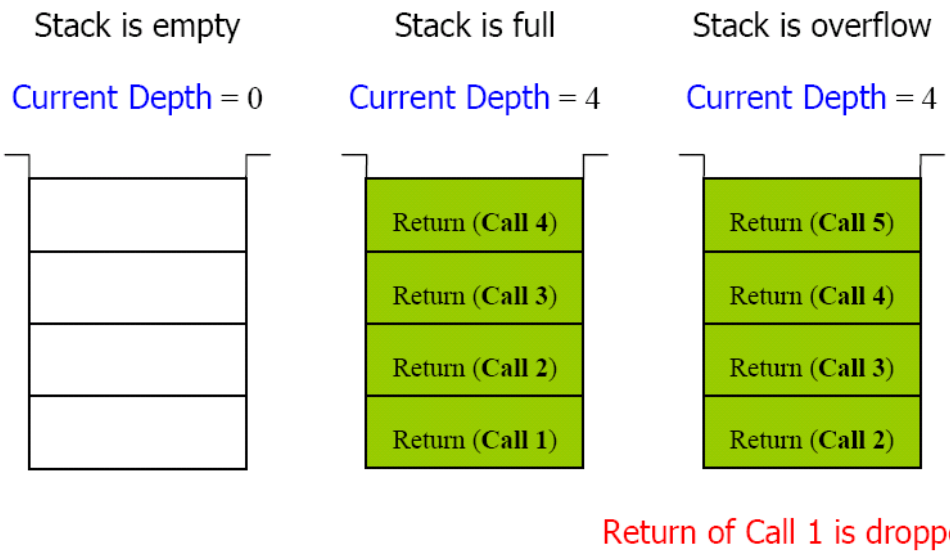


Figure 2.13 Scenario of return stack overflow in a finite depth push-down stack

2.6 Design Restriction and Execution Examples

Top-level instruction memory in proposed designs is assumed to have the same clock rate with CPU. Although not all sorts of memory are clock-aware, all self-managed multi-power mode memories are now equipped with clock signals. For example, DRAMs are clocked always. The only restriction in iAIM design is how to synchronize memory clock with CPU's.

In order to illustrate the validity of iAIM design, a representative scenario of instruction execution is taken as an example :

There are a part of instructions in a program which comprise 2 branch instructions B1, B2 and other instructions S1, S2, S3, … In this execution scenario, B1 is not taken and B2 is taken.

B1 (address : 0x80000400, branch target : 0x80000a00, not-taken in this scenario)

B2 (address : 0x80000404, branch target : 0x80000800, taken in this scenario)

S1 (address : 0x80000408)

…

S2 (address : 0x80000800)

S3 (address : 0x80000804)

…

S4 (address : 0x80000a00)

…

There are 4 possible cases of instruction execution flow in iAIM design depending on BTB's prediction :

1. B1 is predicted not-taken, B2 is predicted taken :

   In this case, both B1 and B2 are correctly predicted by BTB in iAIM.

2. B1 is predicted not-taken, B2 is predicted not-taken :

   In this case, B1 is correctly predicted but B2 is incorrectly predicted by BTB in iAIM. Penalty of 1 clock cycles is incurred.

3. B1 is predicted taken, B2 is predicted taken :

   In this case, B1 is incorrectly predicted but B2 is correctly predicted by BTB in iAIM. Penalty of 1 clock cycles is incurred.

4. B1 is predicted taken, B2 is predicted not-taken :

   In this case, B1 and B2 are all incorrectly predicted by BTB in iAIM. Penalty of 2 clock cycles is incurred.

Execution detail of iAIM design in the first case (B1 is predicted not-taken, B2 is predicted taken) is shown below :

Clock Cycle	PC used In CPU	PC used in iAIM	Control Signal S-Indicate	Control Signal P-Taken	Actions taken in CPU / Actions taken in iAIM
X	0x80000400	0x80000400	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x800003FC) is not a branch instruction, CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB predicts current PC is a not taken branch. iAIM will set P-Taken to 0 at next clock cycle. 3) Next self-generated PC in iAIM will be current PC plus 4.
X+1	0x80000404	0x80000404	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000400) is a not-taken branch. Because iAIM doesn't set P-Taken to 1 at current clock cycle, CPU finds BTB prediction in iAIM at last clock cycle was correct. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB predicts current PC is a taken branch. iAIM will set P-Taken to 1 at next clock cycle. 3) Next self-generated PC in iAIM will be branch target address.

X+2	0x80000408	0x80000800	**0 0**	**1**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000404) is a taken branch. Because iAIM sets P-Taken to 1 at current clock cycle, CPU finds BTB prediction in iAIM at last clock cycle was correct. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to branch target address plus 4.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) iAIM updates an entry in BTB when a not-taken branch was found in BTB 2 clock cycles ago : source is PC used in iAIM 2 clock cycles ago, target is address on instruction address bus, direction is not-taken. 3) Before the end of clock cycle, BTB predicts current PC is not a branch. iAIM will set P-Taken to 0 at next clock cycle. 4) Next self-generated PC in iAIM will be current PC plus 4.
X+3	0x80000804	0x80000804	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000800) is not a branch. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) iAIM updates an entry in BTB : source is PC used in iAIM 2 clock cycles ago, target is address on instruction address bus, direction is taken. 3) Before the end of clock cycle, BTB finds current PC is not a branch. iAIM will set P-Taken to 0 at next clock cycle. 4) Next self-generated PC in iAIM will be current PC plus 4.

Execution detail of iAIM design in the second case (B1 is predicted not-taken, B2 is predicted not-taken) is shown below :

Clock Cycle	PC used In CPU	PC used in iAIM	Control Signal S-Indicate	Control Signal P-Taken	Actions taken in CPU Actions taken in iAIM
X	0x80000400	0x80000400	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x800003FC) is not a branch instruction, CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB predicts current PC is a not taken branch. iAIM will set P-Taken to 0 at next clock cycle. 3) Next self-generated PC in iAIM will be current PC plus 4.
X+1	0x80000404	0x80000404	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000400) is a not-taken branch. Because iAIM doesn't set P-Taken to 1 at current clock cycle, CPU finds BTB prediction in iAIM at last clock cycle was correct. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB predicts current PC is a not-taken branch. iAIM will set P-Taken to 0 at next clock cycle. 3) Next self-generated PC in iAIM will be current PC plus 4.
X+2	0x80000408	0x80000408	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000404) is a not-taken branch. Because iAIM doesn't set P-Taken to 1 at current clock cycle, CPU finds BTB prediction in iAIM at last clock cycle was incorrect. CPU will set S-Indicate to 0b10 at next clock cycle.

					2) Next PC used in CPU will be updated to branch target address.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) iAIM updates an entry in BTB when a not-taken branch was found in BTB 2 clock cycles ago : source is PC used in iAIM 2 clock cycles ago, target is address on instruction address bus, direction is not-taken. 3) Before the end of clock cycle, BTB predicts current PC is not branch. iAIM will set P-Taken to 0 at next clock cycle. 4) Next self-generated PC in iAIM will be current PC plus 4.
X+3	0x80000800	0x80000800	**1 0**	**0**	1) iAIM starts fetching correct address at current cycle, there is no instruction to be decoded in ID stage of CPU. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus address plus 4.
					1) Because S-Indicate is set to 0b10, iAIM uses address on instruction address bus as its PC. 2) iAIM inserts an entry into BTB when PC used in iAIM 2 clock cycles ago was not found in BTB, updates a entry in BTB otherwise : source is PC used in iAIM 2 clock cycles ago, target is address on instruction address bus, direction is taken. 3) Before the end of clock cycle, BTB predicts current PC is not a branch. iAIM will set P-Taken to 0 at next clock cycle. 4) Next self-generated PC in iAIM will be current PC plus 4.
X+4	0x80000804	0x80000804	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000800) is not a branch. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.

					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB finds current PC is not a branch. iAIM will set P-Taken to 0 at next clock cycle. 3) Next self-generated PC in iAIM will be current PC plus 4.

Execution detail of iAIM design in the third case (B1 is predicted taken, B2 is predicted taken) is shown below :

Clock Cycle	PC used In CPU	PC used in iAIM	Control Signal S-Indicate	Control Signal P-Taken	Actions taken in CPU Actions taken in iAIM
X	0x80000400	0x80000400	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x800003FC) is not a branch instruction, CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4. 1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB predicts current PC is a taken branch. iAIM will set P-Taken to 1 at next clock cycle. 3) Next self-generated PC in iAIM will be branch target address.
X+1	0x80000404	0x80000a00	**0 0**	**1**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000400) is a not-taken branch. Because iAIM sets P-Taken to 1 at current clock cycle, CPU finds BTB prediction in iAIM at last clock cycle was incorrect. CPU will set S-Indicate to 0b10 at next clock cycle. 2) Next PC used in CPU will be the same with current PC.

					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB predicts current PC is not a branch. iAIM will set P-Taken to 0 at next clock cycle. 3) Next self-generated PC in iAIM will be current PC plus 4.
X+2	0x80000404	0x80000404	**1 0**	**0**	1) iAIM starts fetching correct address at current cycle, there is no instruction to be decoded in ID stage of CPU. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.
					1) Because S-Indicate is set to 0b10, iAIM uses address on instruction address bus as its PC. 2) iAIM updates an entry in BTB : source is PC used in iAIM 2 clock cycles ago, target is address on instruction address bus, direction is not-taken. 3) Before the end of clock cycle, BTB predicts current PC is a taken branch. iAIM will set P-Taken to 1 at next clock cycle. 4) Next self-generated PC in iAIM will be branch target address.
X+3	0x80000408	0x80000800	**0 0**	**1**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000404) is a taken branch. Because iAIM does set P-Taken to 1 at current clock cycle, CPU finds BTB prediction in iAIM at last clock cycle was correct. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to branch target address plus 4.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB predicts current PC is not a branch. iAIM will set P-Taken to 0 at next clock cycle. 3) Next self-generated PC in iAIM will be current PC plus 4.

X+4	0x80000804	0x80000804	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000800) is not a branch. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) iAIM updates an entry in BTB : source is PC used in iAIM 2 clock cycles ago, target is address on instruction address bus, direction is taken. 3) Before the end of clock cycle, BTB finds current PC is not a branch. iAIM will set P-Taken to 0 at next clock cycle. 4) Next self-generated PC in iAIM will be current PC plus 4.

Execution detail of iAIM design in the fourth case (B1 is predicted taken, B2 is predicted not-taken) is shown below :

Clock Cycle	PC used In CPU	PC used in iAIM	Control Signal S-Indicate	Control Signal P-Taken	Actions taken in CPU Actions taken in iAIM
X	0x80000400	0x80000400	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x800003FC) is not a branch instruction, CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB predicts current PC is a taken branch. iAIM will set P-Taken to 1 at next clock cycle. 3) Next self-generated PC in iAIM will be branch target address.
X+1	0x80000404	0x80000a00	**0 0**	**1**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000400) is a not-taken branch. Because iAIM sets P-Taken to 1 at current

					clock cycle, CPU finds BTB prediction in iAIM at last clock cycle was incorrect. CPU will set S-Indicate to 0b10 at next clock cycle. 2) Next PC used in CPU will be the same with current PC.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB predicts current PC is not a branch. iAIM will set P-Taken to 0 at next clock cycle. 3) Next self-generated PC in iAIM will be current PC plus 4.
X+2	0x80000404	0x80000404	**1 0**	**0**	1) iAIM starts fetching correct address at current cycle, there is no instruction to be decoded in ID stage of CPU. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.
					1) Because S-Indicate is set to 0b10, iAIM uses address on instruction address bus as its PC. 2) iAIM updates an entry in BTB : source is PC used in iAIM 2 clock cycles ago, target is address on instruction address bus, direction is not-taken. 3) Before the end of clock cycle, BTB predicts current PC is a not-taken branch. iAIM will set P-Taken to 0 at next clock cycle. 4) Next self-generated PC in iAIM will be current PC plus 4.
X+3	0x80000408	0x80000408	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000404) is a taken branch. Because iAIM doesn't set P-Taken to 1 at current clock cycle, CPU finds BTB prediction in iAIM at last clock cycle was incorrect. CPU will set S-Indicate to 0b10 at next clock cycle. 2) Next PC used in CPU will be updated to branch target address.

					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB predicts current PC is not a branch. iAIM will set P-Taken to 0 at next clock cycle. 3) Next self-generated PC in iAIM will be current PC plus 4.
X+4	0x80000800	0x80000800	**1 0**	**0**	1) iAIM starts fetching correct address at current cycle, there is no instruction to be decoded in ID stage of CPU. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.
					1) Because S-Indicate is set to 0b10, iAIM uses address on instruction address bus as its PC. 2) iAIM inserts an entry into BTB when PC used in iAIM 2 clock cycles ago was not found in BTB, updates a entry in BTB otherwise : source is PC used in iAIM 2 clock cycles ago, target is address on instruction address bus, direction is taken. 3) Before the end of clock cycle, BTB predicts current PC is not a branch. iAIM will set P-Taken to 0 at next clock cycle. 4) Next self-generated PC in iAIM will be current PC plus 4.
X+5	0x80000804	0x80000804	**0 0**	**0**	1) Before the end of clock cycle, CPU resolves the instruction(address 0x80000800) is not a branch. CPU will set S-Indicate to 0b00 at next clock cycle. 2) Next PC used in CPU will be updated to current PC plus 4.
					1) Because S-Indicate is set to 0b00, iAIM uses self-generated address as its PC. 2) Before the end of clock cycle, BTB finds current PC is not a branch. iAIM will set P-Taken to 0 at next clock cycle. 3) Next self-generated PC in iAIM will be current PC plus 4.

Chapter 3 Evaluation and Discussion

Proposed designs in Chapter 2 are evaluated by trace-driven simulator. The benchmark suit is a subset of MiBench [10], which is a benchmark suite for embedded programs. The results are evaluated by four metrics : total execution cycles, BTB accuracy, percentage of reduced instruction address bus active cycles, percentage of reduced bit transitions.

3.1 Evaluation Methodology

Since proposed designs in Chapter 2 are system-level innovation in computer architecture, behavioral simulation like trace-driven simulator can be a suitable approach to prove how many benefits such innovation gains compared with conventional architecture.

Proposed designs are evaluated by a trace-driven simulator. Since proposed designs in this thesis are based on classic MIPS five-stage pipeline, my simulator uses MIPS I instruction trace as key input.

My trace-driven simulator accepts the following parameters as its input :

1. Architecture : conventional architecture, conventional architecture plus T0 encoding, proposed design of iAIM, proposed design of enhanced iAIM with partial decoder, and proposed design of enhanced iAIM with partial decoder and return stack.

2. BTB configuration : Perfect BTB (it consists of 2 properties. First, after a taken branch is first allocated into BTB, its prediction afterwards will be always correct. Second, any allocated entry in BTB will never be replaced.), 2048/4way/LRU (it means 2048 entries in 4-way set-associative BTB with Least Recently Used replacement algorithm), and 32/4way/LRU (it means 32 entries in 4-way set-associative BTB with Least Recently Used replacement algorithm).

3. MIPS I instruction trace of benchmark program.

My trace-driven simulator will records bit transitions for every line of instruction address bus and addition control lines (conventional architecture has no additional control line;

conventional architecture plus T0-encoding and conventional architecture plus T0-DAT encoding have one additional control line, our 3 proposed designs of iAIM has 3 addition control lines) at every clock cycle during MIPS I instruction trace is being fed.

After finishes execution, my simulator will output the following data :

1. Total execution cycles

2. BTB accuracy

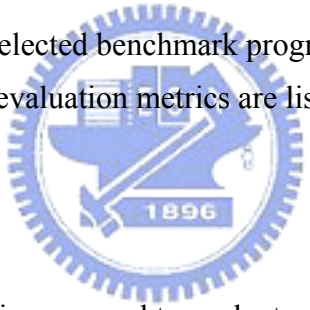
3. Instruction address bus active cycles.

4. Total bit transitions on instruction address bus and control line(s).

5. Total bit transitions on instruction address bus.

6. Total bit transitions on control line(s).

After collecting all statistics of selected benchmark programs, we have enough data to do evaluation on proposed designs. The evaluation metrics are listed in next section.



3.2 Evaluation Metrics

In this thesis, the following metrics are used to evaluate proposed designs of iAIM:

- Total execution cycles

  This metric is used to indicate whether proposed designs suffer performance loss due to longer execution time compared with conventional architecture.

- BTB accuracy

  This metric is used to indicate whether proposed designs suffer loss in branch prediction accuracy due to poorer BTB accuracy compared with conventional architecture.

- Percentage of reduced instruction address bus active cycles

  This value is defined as :

(Total execution cycles－Instruction address bus active cycles) / Total execution cycles

If this value is high, it means instruction address bus is frozen most of the time. This metric can effectively be used to evaluate bus power consumption indirectly due to coupling capacitance.

- Percentage of reduced bit transitions

  This value is defined as :

  (Total bit transitions in conventional architecture with the same BTB configuration －Total bit transitions) / (Total bit transitions in conventional architecture with the same BTB configuration)

  If this value is high, it means number of total bit transitions is small. This metric can effectively be used to evaluate bus power consumption indirectly due to self-capacitance.

3.3 Experimental Environment

The experimental toolset MIPS SDE / MIPS FGT 5.02.02 [11] is used to generate MIPS I instruction trace for benchmark programs :

- Install MIPS SDE / MIPS FGT 5.02.02.
- Use command “sde-make SBD=GSIM1B” to build MIPS I code (*benchmark_ram*) of benchmark program for GNU simulator platform.
- Use command “sde-run --trace-insn=on --trace-file *trace_filename benchmark_ram*” to generate MIPS I instruction trace file.

Since delay branch slot is always applied in GNU simulator platform, the generated trace file needs to be modified to remove delay branch slot for all branch and jump instructions.

The modified trace file is then fed into trace simulator by specifying various parameters like BTB configuration (perfect BTB or not, the number of entries/set-associativity/replacement algorithm of BTB), return stack configuration (return stack is used or not), and selected design (conventional architecture, conventional

architecture with T0 encoding, conventional architecture with T0-DAT encoding, proposed design of iAIM, proposed design of iAIM with partial decoder and proposed design of iAIM with partial decoder and return stack). Figure 3.1 shows the flowchart of simulation.

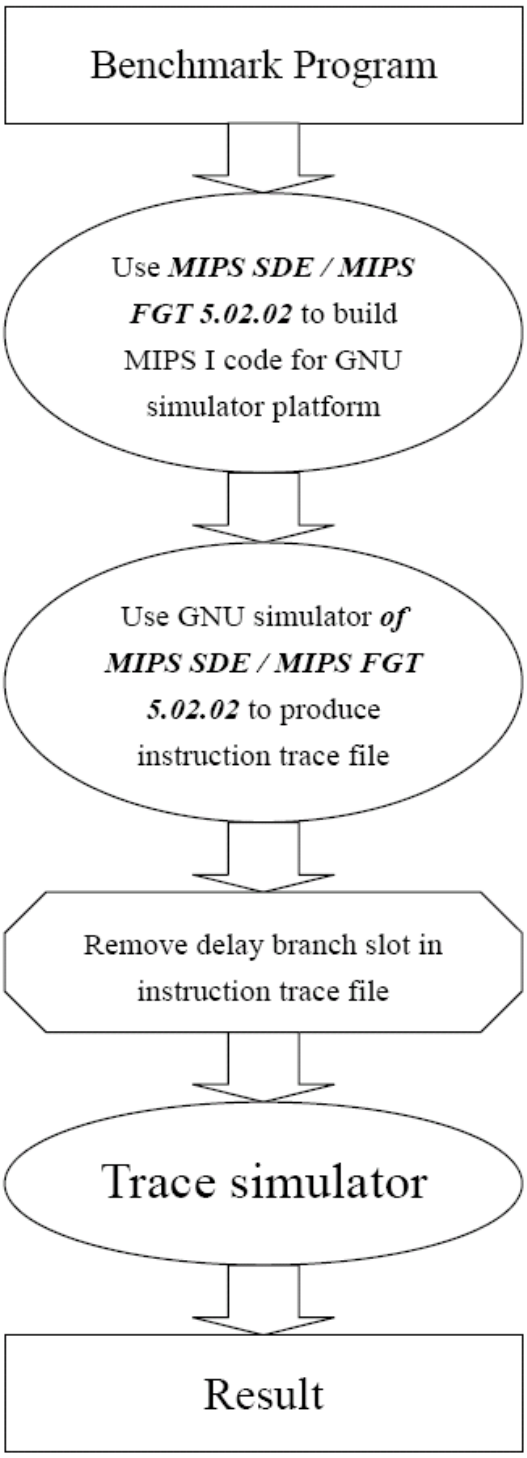


Figure 3.1 Simulation flowchart

3.4 Experimental Benchmark

The benchmark programs selected are a subset of MiBench [10], which is a benchmark suite consisting of commercially representative embedded programs. MiBench consists of 6 categories including Automotive and Industrial Control, Network, Security, Consumer Devices, Office Automation, and Telecommunications. In each category, at least one benchmark is chosen as experimental benchmark. All chosen benchmarks are listed as below :

- In the category of Automotive and Industrial Control

  basicmath : it performs simple mathematical calculations that often don't have dedicated hardware support in embedded processors.

  bitcount : it tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers.

- In the category of Network

  dijkstra : it constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm.

- In the category of Security

  sha : it is the secure hash algorithm that produces a 160-bit message digest for a given input. It is often used in the secure exchange of cryptographic keys and for generating digital signatures.

  rijndael encrypt/decrypt : Rijndael was selected as the National Institute of Standards and Technologies Advanced Encryption Standard (AES). It is a block cipher with the option of 128-, 192-, and 256-bit keys and blocks.

- In the category of Consumer Devices

  jpeg encode/decode : JPEG is a standard, lossy compression image format. It is a representative algorithm for image compression and decompression and is commonly used to view images embedded in documents.

  lame : it is a GPL'ed MP3 encoder that supports constant, average and variable bit-rate encoding. It uses small and large wave files for its data inputs.

- In the category of Office Automation

stringsearch : it searches for given words in phrases using a case insensitive comparison algorithm.

- In the category of Telecommunications

FFT/IFFT : it performs a Fast Fourier Transform and its inverse transform on an array of data. Fourier transforms are used in digital signal processing to find the frequencies contained in a given input signal.

ADPCM encode/decode : Adaptive Differential Pulse Code Modulation (ADPCM) is a variation of the well-known standard Pulse Code Modulation (PCM). A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1.

CRC32 : it performs a 32-bit Cyclic Redundancy Check (CRC) on a file. CRC checks are often used to detect errors in data transmission.

Table 3.1 shows the instruction counts and maximum procedure call depth for selected benchmarks.

Benchmark	Number of total executed instructions	Number of executed branch instructions	Number of executed procedure return instructions	Maximum procedure call depth
basicmath	59,681,183	8,022,682 (13.4426 %)	1,179,026 (1.9755 %)	17
bitcount	46,804,277	4,816,087 (10.2898 %)	1,050,913 (2.2453 %)	17
dijkstra	72,216,243	11,012,478 (15.2493 %)	621,782 (0.8610 %)	26
sha	11,402,685	258,052 (2.2631 %)	11,156 (0.0978 %)	17
rijndael encrypt	35,905,204	1,047,215 (2.9166 %)	206,351 (0.5747 %)	17
rijndael decrypt	35,723,715	1,045,648 (2.9270 %)	205,997 (0.5766 %)	17
jpeg encode	34,746,120	4,070,029 (11.7136 %)	66,230 (0.1906 %)	17
jpeg decode	8,471,825	373,111 (4.4041 %)	13,078 (0.1544 %)	17
lame	191,301,926	14,830,125 (7.7522 %)	1,224,499 (0.6401 %)	17
stringsearch	211,681	35,075 (16.5697 %)	2,892 (1.3662 %)	17
FFT	18,571,659	2,215,564 (11.9298 %)	299,462 (1.6125 %)	17
IFFT	16,056,034	1,913,372 (11.9168 %)	272,061 (1.6944 %)	17
ADPCM encode	36,515,266	7,614,881 (20.8540 %)	23,512 (0.0644 %)	17
ADPCM decode	29,296,742	6,587,106 (22.4841 %)	23,495 (0.0802 %)	17
CRC32	92,343,488	12,427,418 (13.4578 %)	5,498,790 (5.9547 %)	17

Table 3.1 Instruction counts and maximum procedure call depth for selected benchmarks

3.5 Experimental Results

Table 3.2 shows simulation results. The abbreviations on table 3.2 are listed as below.

There are 6 kinds of designs :

- Original : stands for conventional architecture that BTB is in CPU.
- T0 : means conventional architecture with T0 encoded instruction address bus.
- T0 DAT128 : means conventional architecture with T0 with Discontinuous Address Table of 128 entry encoded instruction address bus.
- Proposed I : stands for design of iAIM proposed in section 2.3.
- Proposed II : stands for design of iAIM with partial decoder proposed in section 2.4.
- Proposed III : stands for design of iAIM with partial decoder and return stack proposed in section 2.5.

There are 3 kinds of BTB configurations:

- Perfect BTB : it consists of 2 properties. First, after a taken branch is first allocated into BTB, its prediction afterwards will be always correct. Second, any allocated entry in BTB will never be replaced.
- 2048 (4way, LRU) : means 2048 entries in 4-way set-associative BTB with Least Recently Used replacement algorithm.
- 32 (4way, LRU) : means 32 entries in 4-way set-associative BTB with Least Recently Used replacement algorithm.

BTB Entries / Accuracy	Design	Total Execution Cycles	Address Bus Active Cycles	% of Address Bus Frozen Cycles	Total Bit Transitions on Address Bus and additional Control Line(s)	% of Reduced Bit Transitions compared with Original	Bit Transitions on Address Bus	Bit Transitions on Control line(s)	Bit Transitions on Control lines S-Indicate	Bit Transitions on Control line P-Taken
Perfect BTB / 99.99 %	Original	700,787,910	700,608,868	0.0255	1,541,728,133	0	1,541,728,133	-	-	-
	T0	700,787,910	55,756,114	92.0438	402,288,302	73.9067	269,057,737	133,230,565	-	-
	T0 DAT128	700,787,910	18,983,157	97.2912	118,672,679	92.3026	82,351,940	36,320,739	-	-
	Proposed I	700,787,910	10,671,756	98.4772	199,721,333	87.0456	43,428,773	156,292,560	45,780,724	110,511,836
	Proposed II	700,787,910	10,661,282	98.4787	199,667,193	87.0491	43,374,633	156,292,560	45,780,724	110,511,836
	Proposed III	690,088,666	80,668	99.9883	114,530,065	92.5713	1,034,481	113,495,584	2,983,748	110,511,836
2048 (4way, LRU) / 91.49 %	Original	707,266,049	707,087,007	0.0253	1,558,928,091	0	1,558,928,091	-	-	-
	T0	707,266,049	63,322,107	91.0469	427,284,280	72.5911	287,714,443	139,569,837	-	-
	T0 DAT128	707,266,049	25,462,209	96.3999	138,944,871	91.0872	98,308,542	40,636,329	-	-
	Proposed I	707,266,049	16,970,558	97.6005	240,765,665	84.5557	69,851,873	170,913,792	58,737,002	112,176,790
	Proposed II	707,266,049	10,661,282	98.4926	214,288,425	86.2541	43,374,633	170,913,792	58,737,002	112,176,790
	Proposed III	696,566,805	80,668	99.9884	129,151,297	91.7154	1,034,481	128,116,816	15,940,026	112,176,790
32 (4way, LRU) / 82.82 %	Original	713,879,097	713,700,055	0.0251	1,569,816,155	0	1,569,816,155	-	-	-
	T0	713,879,097	62,862,831	91.1942	424,843,478	72.9367	285,907,171	138,936,307	-	-
	T0 DAT128	713,879,097	31,880,727	95.5342	168,259,225	89.2816	121,421,584	46,837,641	-	-
	Proposed I	713,879,097	23,631,314	96.6897	272,600,469	82.6349	101,872,745	170,727,724	71,963,098	98,764,626
	Proposed II	713,879,097	10,661,282	98.5066	214,102,357	86.3613	43,374,633	170,727,724	71,963,098	98,764,626
	Proposed III	703,179,853	80,668	99.9885	128,965,229	91.7847	1,034,481	127,930,748	29,166,122	98,764,626

Table 3.2 Simulation Results

Figure 3.2 and Figure 3.3 show reduction ratios in instruction address bus active cycles and bit transitions for 5 different designs respectively.

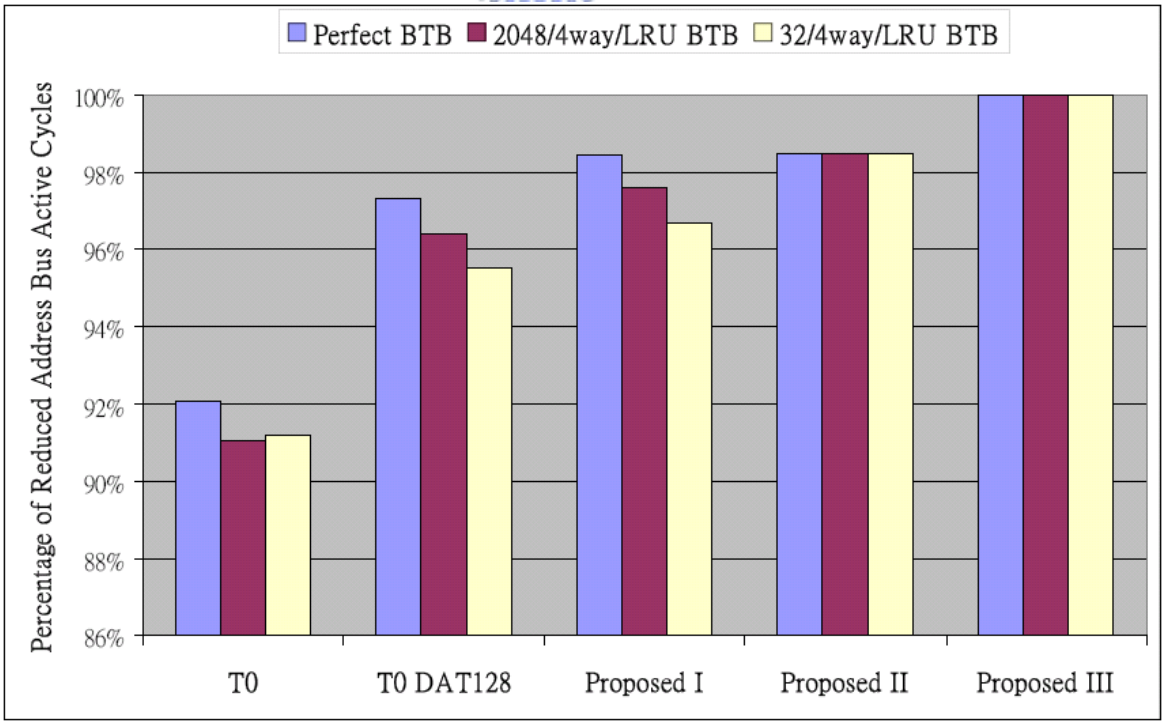


Figure 3.2 Percentage of reduced instruction address bus active cycles

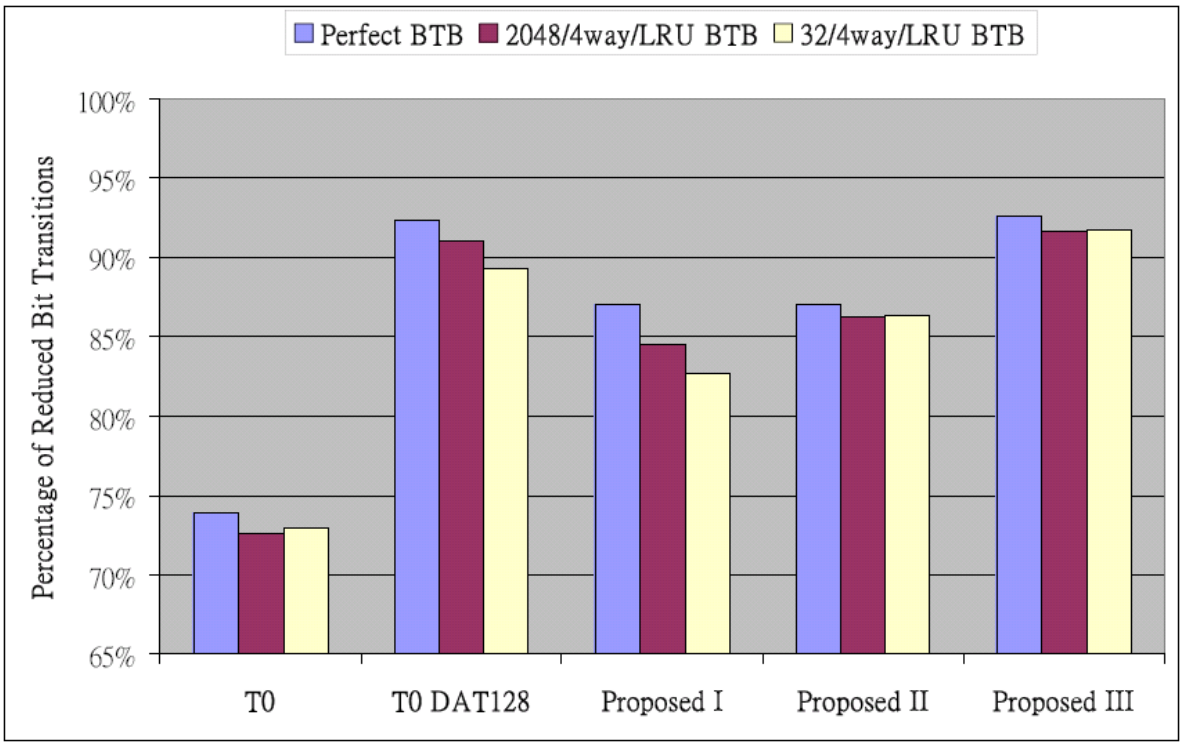


Figure 3.3 Percentage of reduced bit transitions

Figure 3.4 shows percentage of bit transitions on instruction address bus and control line(s) for 5 different designs. Figure 3.5 shows percentage of bit transitions on instruction address bus and control lines S-Indicate, P-Taken for proposed iAIM designs.

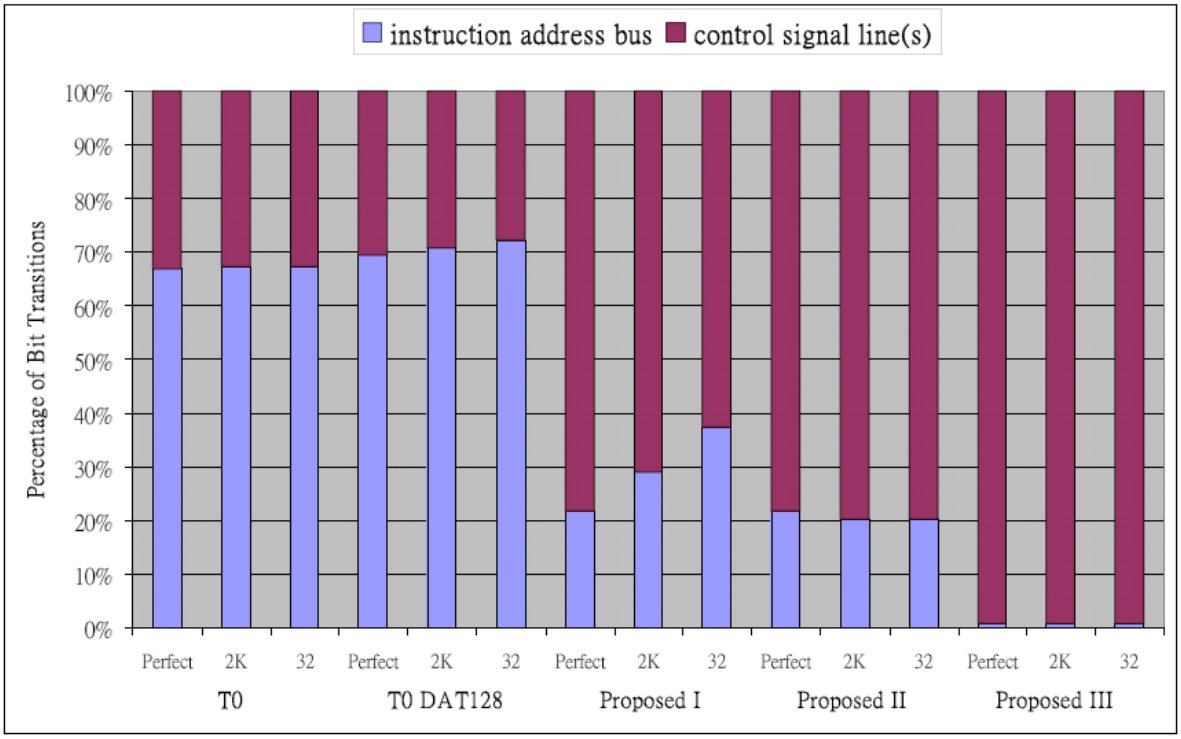


Figure 3.4 Percentage of bit transitions on address bus and control line(s)

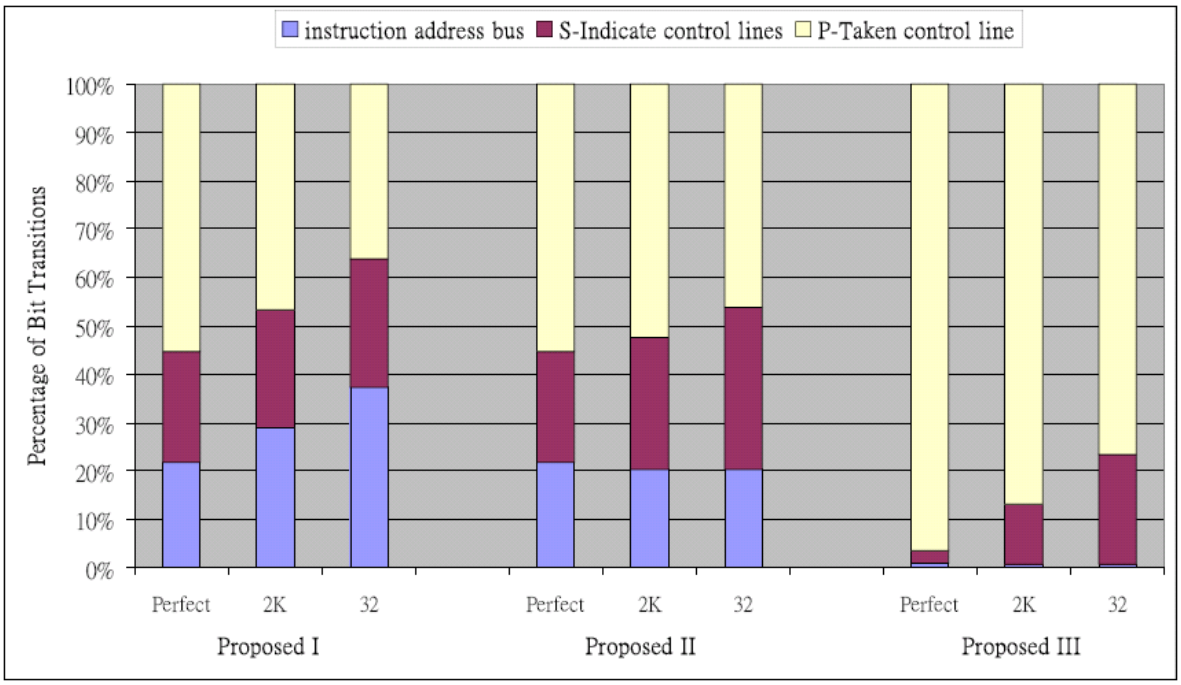


Figure 3.5 Percentage of bit transitions on address bus and control lines S-Indicate, P-Taken

3.6 Discussion

3.6.1 Experimental Results for Five Evaluation Metrics

Simulation results for five evaluation metrics are summarized as below :

- Total execution cycles in conventional architecture, conventional architecture with T0 encoded instruction address bus, proposed design I and II of iAIM are exactly the same. Total execution cycles in proposed design III of iAIM is slightly less than in all other designs because return stack reduces penalty for procedure return instructions.
- BTB accuracy for the same BTB configuration in conventional architecture, conventional architecture with T0 encoded instruction address bus, proposed design I and III of iAIM are exactly the same. Although iAIMs update BTB one cycle later than conventional architecture does, such one cycle delay does not harm BTB accuracy.
- T0 encoded instruction address bus reduces 91.43 % of instruction address bus active cycles and 73.14 % of bit transitions on instruction address bus and one control line (“INC” [7]) on average.

Bit transitions on control line occupy about 33 % of total bit transitions.

- T0 DAT with 128 entries encoded instruction address bus reduces 96.41 % of instruction address bus active cycles and 90.89 % of bit transitions on instruction address bus and one control line ("INC" [8]) on average.

  Bit transitions on control line occupy about 29 % of total bit transitions.

- iAIM proposed in section 2.3 reduces 97.59 % of instruction address bus active cycles and 84.75 % of bit transitions on instruction address bus and three control lines ("P-Taken" and "S-Indicate") on average.

  Bit transitions on control lines occupy about 71 % of total bit transitions.

- iAIM with partial decoder proposed in section 2.4 reduces 98.50 % of instruction address bus active cycles and 86.55 % of bit transitions on instruction address bus and three control lines ("P-Taken" and "S-Indicate") on average.

  Bit transitions on control lines occupy about 79 % of total bit transitions.

- iAIM with partial decoder and return stack proposed in section 2.5 reduces 99.99 % of instruction address bus active cycles and 92.02 % of bit transitions on instruction address bus and three control lines ("P-Taken" and "S-Indicate") on average.

  Bit transitions on control lines occupy about 99 % of total bit transitions.

3.6.2 Comparisons among Bus Encoding Techniques and 3 iAIM Designs

Although basic design philosophies are different, bus encoding techniques (like T0 encoding) and iAIM have the same purpose – reducing bus traffic on instruction address bus. The cause that iAIM can reduce much more bus traffic is it equips instruction memory with program flow tracing capability. With program flow tracing capability, iAIM is capable of eliminating the need for bus to transfer instruction addresses most of the time. Such capability makes iAIM more intelligent and autonomous than bus encoding techniques.

T0 DAT encoding is a special bus encoding technique that makes use of not only the characteristic of program sequential execution but also the characteristic of taken branch execution. Only the third proposed iAIM design can outperform it slightly in all 2 metrics of bus traffic reduction. Since Content-Addressable-Memory (CAM) is required in both encoder and decoder in T0 DAT, it adds additional time to the existing delay time due to CPU to memory latency. Such an increase affects the clock rate and then

harms the performance of processor accordingly.

As the constituents of bit transitions are considered, bit transitions on instruction address bus hold the greater part in both T0 and T0 DAT encoding technique while bit transitions on control lines occupy the most majority in iAIM designs. This fact reveals the proportion of bit transitions on instruction address bus in iAIM is insignificant. In other words, the true overhead for iAIM is bit transitions on control lines. Therefore, a communication protocol that uses least control lines to convey minimum control signals between CPU and iAIM is necessary.

Among 3 proposed iAIM designs, performances of Proposed II and Proposed III are more insensitive to BTB prediction accuracy than Proposed I. The cause of this phenomenon is Proposed II/III record information of recently resolved branch instruction so that branch recovery becomes easy.

3.6.3 Benefits and Drawbacks in iAIM Designs

The benefits in iAIM are listed as below :

1. Reduction in bus traffic to spare bandwidth :

   This can be proved from the above experiment results.

2. Reduction in power consumption :

   For off-chip application, it can be deduced indirectly from experiment results since most traffic on instruction is reduced.

3. Reduction in delay time due to possible high CPU to memory latency

   Since iAIM can reduce delay time when instruction address is self-generated, total instruction fetch time can be shortened if two address generation mechanisms (from bus or iAIM internal) can take different cycles.

The drawbacks in iAIM are listed as below :

1. Although BTB is merely removed from CPU to instruction memory side and only some simple logic like partial decoder is added into instruction memory, it does incur addition overhead in conventional computer architecture.

2. Although iAIM reduces almost all traffic on instruction address bus, there are more additional bus traffic appearing on the additional internal buses needed in iAIM. These additional internal buses make on-chip iAIM application less

useful.

3.6.4 iAIM Application in Real Computer System Environments

The effects of applying iAIM concept to real computer system environments are discussed as follows :

- CPU and top-level Instruction Memory reside on different Chips

  Under this environment, instruction address bus between CPU and top-level instruction memory is external bus.

  As mentioned in section 1.1.1, power consumed on external bus due to relatively high self-capacitance is several order larger than energy than internal bus insides CPU or instruction memory. Though BTB power in different chip is different due to different process and iAIM incurs more internal buses and additional logics, it still conserves more power than conventional architecture does.

- CPU and top-level Instruction Memory reside on the same Chip

  Under this environment, Instruction address bus between CPU and top-level instruction memory is internal bus.

  Power consumed on internal bus is dominant by coupling capacitance. iAIM can greatly reduce power of coupling capacitance on instruction address bus since it freezes bus most of the time.

  If internal buses added by iAIM are not dedicate buses, iAIM also gains benefit in this environment. Otherwise, iAIM is useless.

Chapter 4 Conclusion and Future Works

4.1 Conclusion

The meaning that BTB is placed inside CPU in conventional architecture needs to be rethought because it incurs too much unnecessary traffic on instruction address bus. Proposed designs in Chapter 2 prove iAIM concept not only feasible but also effective. Some functions in CPU costs little and can be duplicated in iAIM by using extremely few logics (e.g., partial decoder). The mechanism that how dynamic branch predictor like BTB predicts branch instruction has already been mature for a long time and is not invented by us. What we do is only to move dynamic branch predictor to instruction memory side. The choice of depth of return stack proposed in section 2.4 can be based on simulate application programs on simulator to define a proper value.

The underlying design philosophy for three iAIM designs proposed in this thesis is to equip top-level instruction memory with program flow tracing capability. Such design philosophy is an innovation in computer architecture. Such computer architecture change looks promising in simulation results. And the increase on additional circuit cost seems a small amount since most function blocks are merely moved from CPU to instruction memory.

4.2 Future Works

In this thesis, BTB, partial decoder and return stack are incorporated into instruction memory one after another to form iAIM designs of Proposed I, II and III respectively. There are still several works in such design philosophy.

- Does there exist a similar design philosophy that is also applicable to data memory? That means an intelligent autonomous data memory design may be another practicable research direction.

- iAIM design does require additional internal buses inside instruction memory module. For application environments with off-chip instruction memory system, power consumption due to additional internal buses inside iAIM is negligible compared with power saved on external instruction address bus. Nonetheless, for application environments with on-chip instruction memory system, if additional

internal buses inside iAIM are dedicated to iAIM, power consumed on additional internal buses inside iAIM may cancel out power saved on external instruction address bus. Because iAIM can relieve address traffic on instruction address bus greatly, many systems that use unified instruction and data memory (see Figure 3.6) may benefit by iAIM concept. When iAIM design is applied to the mixed instruction/data address bus in unified memory system, one additional control line to distinguish instruction address stream and data address stream is enough. When this control line indicates data address stream occupies the address bus at current clock cycle, iAIM's instruction address handling mechanism inside unified memory may treat it as "Pipeline Stall" situation while CPU can make use of "S-Indicate" control lines to apply one of data address bus encoding techniques (like BI [12], T0_BI [13], T0_BI_1 [14], …) to reduce bus power. This future work is practicable and deserves elaborate design and extensive evaluation.

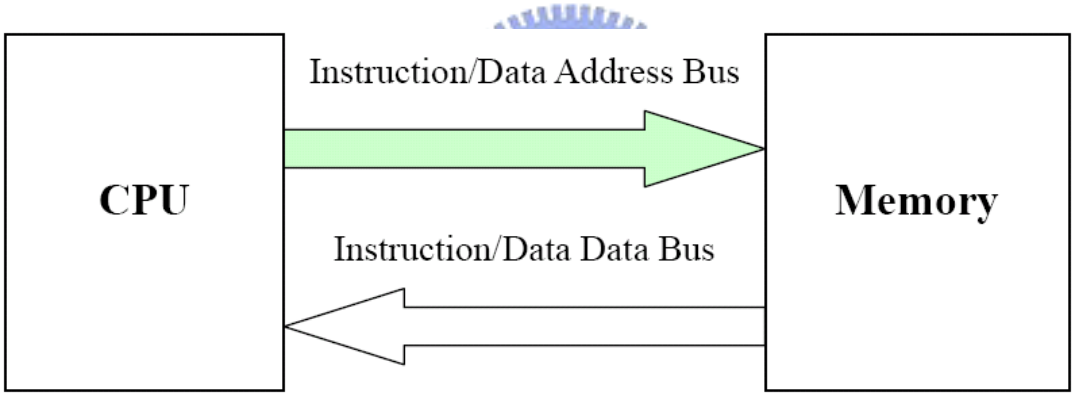


Figure 4.1 Unified instruction and data memory system

References

[1] J. L. Hennessy and D. A. Patterson, “Computer Architecture - A Quantitative Approach”, 3rd ed. Morgan Kaufmann Publishers, 2003.

[2] P. Petrov, A. Orailoglu, “Low-power Instruction Bus Encoding for Embedded Processors,” in IEEE Transactions on VLSI (TVLSI), July, 2004.

[3] H. B. Bakoglu, Circuits, Interconnections and Packaging for VLSI, Addison-Wesley, 1990.

[4] K. Basu, A. Choudhary, and M. Kandemir, “Power protocol : reducing power dissipation on off-chip data buses”, In Proc. the 35th Annual International Symposium on Microarchitecture, Istanbul, Turkey, November 2002.

[5] Edwin Naroska, Shanq-Jang Ruan, and Feipei Lai, “On Optimizing Power and Crosstalk for Bus Coupling Capacitance Using Genetic Algorithm”, IEEE International Symposium on Circuits and Systems, Bangkok, Thailand, May 2003

[6] C. H. Perleberg and A. J. Smith, “Branch target buffer design and optimization,” IEEE Transactions on Computers, 42(4), 1993.

[7] L. Benini, G. De Micheli, E. Marcii, 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, Urbanana-Champaign, IL, March 1997.

[8] Tsung-His Weng, “Low-Power Address Bus Encoding,” Master’s Thesis, Department of Computer Science and Information Engineering, Nation Chiao Tung University, Taiwan, R.O.C., June 2005.

[9] Y. Park and Gyungho Lee, “Return Address Stack Management for Protection from Buffer Overflow Attacks,” Proc. the ACM Frontiers of Computing, Ischia, Italy, Apr. 2004.

[10] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, “MiBench: A Free, Commercially Representative Embedded Benchmark Suite,” Proceedings of the 4th International Workshop on Workload Characterization, 2001, pp. 3–14.

[11] MIPS Technologies, Inc., “MIPS SDE / MIPS FGT 5.02.02 Programmers’Guide,” February 17, 2003.

[12] M.R Stan and W.P Burleson, “Bus Invert Coding for Low Power I/O,” IEEE Transactions VLSI systems, pp 49-58, March 1995.

[13] L. Benini, G. DeMicheli, E. Macii, D. Sciuto, and C. Silvano, “Address bus encoding techniques for system-level power optimization,” DATE-98: IEEE Design Automation and Test in Europe. Paris, France, February 1998, pages 861-866.

[14] Tsung-Hsi Weng, Wei-Hao Chiao, Jean Jyh-Jiun Shann, and Chung-Ping Chung, Jimmy Lu, “Low-Power Data Address Bus Encoding Method,” 2005 International Conference on Computer Design (CDES’05).

Vita

Li-Ming Wang (王立銘)

A. Personal History

Birth place : Kaohsiung City, Taiwan Birth date : April 4, 1973

Residence : Hsinchu or Kaohsiung City, Taiwan

E-mail address : livius@ms56.hinet.net

B. Educational History

1. Kaohsiung Senior High School, Kaohsiung, Taiwan, 1991 (高雄中學)
2. National Cheng Kung University, Tainan, Taiwan
   Degree: Bachelor of Electrical Engineering, 1995 (成功大學電機系 84 級)
3. National Chiao Tung University, Hsinchu, Taiwan
   Degree: Degree Program of Electrical Engineering and Computer Science College of Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science, 2006 (交通大學電資學院在職專班資訊學程碩士 2006)

C. Professional Positions

1. Associate Engineer in Process Control Computer Section of China Steel Aluminum Corp. (中鋼鋁業) from Sep 1997 to Jun 2002
2. Engineer, Senior Engineer, Associate Project Manager in System Development Div. of D-Link Corp. (友訊科技) and Alpha Networks Inc. (明泰科技) from Jun 2002