Implementation of Branch Predictor and Instruction Prefetch for Multithreaded Superscalar RISC Processor

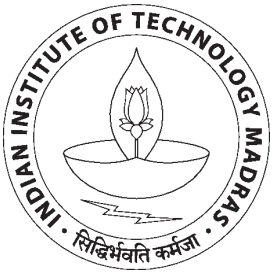
A Project Report

submitted by

KRISHNA CHAITANYA K

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

MAY 2014

THESIS CERTIFICATE

This is to certify that the thesis entitled **Implementation of Branch Predictor and Instruction Prefetch for Multithreaded Superscalar RISC Processor**, submitted by **KRISHNA CHAITANYA K**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V. Kamakoti
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I Would like to express my deepest gratitude to my guide, ***Dr. V. Kamakoti*** for his valuable guidance, encouragement and advice. His immense motivation helped me in making firm commitment towards my project work.

My special thanks to ***Mr. G.S. Madhusudan*** for his encouragement and motivation through out the project. His valuable suggestions and constructive feedback were very helpful in moving ahead in my project work.

I would like to thank my faculty advisor ***Dr. Deleep R.Nair*** who patiently listened, evaluated, and guided us through out our course.

My special thanks to project team members Neel, Rishi Naidu, Arjun ,Naveen, Chidambaranathan, Sachin, Sarath, Senthil, Keerthi for their help and support.

ABSTRACT

KEYWORDS: TAgged GEometric History length, Simultaneous Multithread, Branch History Guided Pefecthing

The techniques of Instruction Level Parallelism (ILP) and pipeline have been used well to speed up the execution of instructions. The conditional branches are the critical factor to the effectiveness of a deep pipeline since the branch instructions can always break the flow of instructions through the pipeline and result in high execution cost. Branch prediction in simultaneous multithreaded processors is difficult because multiple independent threads interfere with shared prediction resources. TAgged GEometric history length (TAGE) predictor is implemented by combining several prediction algorithms. It relies on (partial) hit-miss detection as the prediction computation function.

Instruction cache misses stall the fetch stage of the processor pipeline and hence affect instruction supply to the processor. Instruction prefetching has been proposed as a mechanism to reduce instruction cache (I-cache) misses. A hardware-based instruction prefetching mechanism, Branch History Guided Prefetching (BHGP) is implemented to improve the timeliness of instruction prefetches.

The entire project work is implemented in Bluespec System Verilog and synthesized in Xilinx ISE.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

BHGP	Branch History Guided Prefetching
RISC	Reduced Instruction Set Computer
BSV	Bluespec System Verilog
BTB	Branch Target Buffer
TAGE	TAgged GEometric history length
O-GEHL	Optimized Geometric History Length
PPM	Prediction by Partial Match
BHR	Branch History Registor
PC	Program Counter
HDL	Hardware Description Language
ISA	Instruction set Architecture
ILP	Instruction level Parallelism
SMT	Simultaneous Multithreading
PT	Prefetch Table
BHQ	Branch History Queue
MRM	Most Recently Missed

CHAPTER 1

INTRODUCTION

1.1 Overall Architecture

The Processor design team of Reconfigurable and Intelligent Systems Engineering (RISE) Lab in the Computer Science Dept. of IIT Madras has been actively involved in building a Superscalar processor for Server applications. The proposed processor is a 64 bit, single core, quad threaded superscalar processor. The processor strictly follows RISC-V Instruction set Architecture (ISA). The entire design of the processor is done using a Hardware Description Language (HDL) called Bluespec System Verilog (BSV).

The CPU core is based on the Tomasulo Algorithm. The Microarchitecture of the CPU core is an Out of order microprocessor that is capable of fetching, decoding and issuing 4 instructions every clock cycle. A Centralized Reservation station is implemented to utilize the reservation station entries in a more efficient manner. The Execution units are duplicated to support for Quad issue. The Common Data Bus (CDB) used for Operand forwarding can forward the results of at most 4 execution units. The Reorder buffers are designed to commit maximum of four instructions in every clock cycle. My project work involves the design of Branch Predictor and Instruction Prefetching unit related to the CPU core.

1.2 Objective

1)To design Branch Predictor which can handle 4 threads.

- Gshare
- Tournament
- TAGE

2)To design Instruction Prefetching Unit

1.3 Organization of the thesis

Chapter 2 describes about the need for branch Predictor, overview of basic branch predictors. It also describes about the HDL called Bluespec System Verilog, its key features, BSV constructs. It finally tells about the proposed microarchitecture of the superscalar processor based on the Tomasulo algorithm.

Chapter 3 starts with detailed description of Branch Prediction, components involved in prediction and design of the gshare, tournament and TAGE branch predictors. It also compares the various configurations of converting the above predictors to handle multithread.

Chapter 4 describes about need for instruction prefetching and it describes about design and implementation of BHGP instruction prefetching scheme in the processor.

Chapter 5 describes about how the branch predictor design is verified in Bluespec System Verilog. It also describes about the test cases generated to verify the design.

Chapter 6 concludes with a short description about the future work.

CHAPTER 2

BACKGROUND

This chapter deals about the need for branch predictor. It also deals about the key features of Bluespec System Verilog (BSV), why Bluespec is used, how to build a design in the BSV.

2.1 Need for Branch Predictor

The need for branch prediction arises from the use of pipelining in modern microarchitectures. The goal of pipelining is to utilize the hardware to the fullest possible extent all the time, it is necessary to make sure that each stage of the pipeline contains an instruction as often as possible. If there are no changes in program control flow, then the solution is simple, just make sure that instructions are read from memory quickly enough to keep all the stages full all the time. However, when branches cause the program to behave in ways that the processor does not expect, the solution becomes much more complicated. A branch is a change in the control flow of a program which breaks sequentiality.

Imagine that a branch instruction has moved through the fetch and decode stages and is now being executed. This execution stage is the first time that the processor knows whether or not the branch will be taken. In general the result of this decision is based on a compare between two other data elements. The problem arises because until this comparison occurs, the processor does not know the next correct instruction to execute.

The stages prior to the execution cycle have already begun speculatively processing instructions that follow the branch, yet if the branch is taken, these are not the correct instructions. Therefore, all the stages before the execution cycle must be flushed and instruction fetch must precede from the target location of the taken branch.

This flushing of the pipeline wastes many cycles of execution time, thereby decreasing the performance of the processor. In an effort to save these wasted cycles, processor designers try to predict the direction of each branch instruction before the next instruction is fetched from memory. If the prediction is correct, the next instruction after the branch executes will be the correct instruction to execute next. If the prediction is incorrect, however, the pipeline must be flushed, and the correct instruction read into the pipeline. This incorrect prediction is known as a misprediction.

Branch instructions are executed by the branch functional unit. For a conditional branch, it is not until it exits the branch unit and when both the branch condition and the branch target address are known that the fetch stage can correctly fetch the next instruction. This delay in processing conditional branches incurs a penalty in fetching the next instruction.

The primary aim is to minimize the number of such stall cycles and/or to make use of these cycles to do potentially useful work. The current dominant approach to accomplish this is via branch prediction.

Branch prediction research focuses on improving the performance of pipelined microprocessors by accurately predicting ahead of time whether or not a change in control flow will occur.

2.2 Overview of basic Branch Predictors

2.2.1 Static Branch Prediction

Static branch prediction algorithms tend to be very simple and by definition do not incorporate any feedback from the run-time environment. This characteristic is both the strength and weakness of static prediction algorithms. By not paying any attention to the dynamic run-time behavior of a program, the branch prediction is incapable of adapting to changes in branch outcome patterns. These patterns may vary based on the input set for the program or different phases of a program's execution.

The simplest branch prediction strategy is to predict that the direction of all branches will always go in the same direction (always taken or always not taken). Older pipelined processors, such as the Intel i486 used the always-not-taken prediction algorithm.

A variation of the single-direction static prediction approach is the backwards taken/-forwards not-taken (BTFNT) strategy. A backwards branch is a branch instruction that has a target with a lower address (i.e., one that comes earlier in the program). The rationale behind this heuristic is that the majority of backwards branches are loop branches, and since loops usually iterate many times before exiting, these branches are most likely to be taken. For example, the Intel Pentium 4 processor uses the BTFNT approach as a backup strategy when its dynamic predictor is unable to provide a prediction.

Although static branch predictors have poor prediction accuracy, their major advantages are that they can be accessed instantly, very simple to implement and they require very little hardware resources. Static branch predictors are of less interest in the context of future-generation, large transistor budget, very large-scale integration (VLSI) processors because the additional area for more effective dynamic branch predictors can be afforded.

2.2.2 Profile Static Branch Prediction

Profile-based static branch prediction involves executing an instrumented version of a program on sample input data, collecting statistics, and then feeding back the collected information to the compiler. The compiler makes use of the profile information to make static branch predictions that are inserted into the final program binary as branch hints. The compiler inserts branch hints corresponding to the more frequently observed branch directions during the sample executions. If during the profiling run, a branch was observed to be taken more than half of the time, then the compiler would set the branch hint bit to predict-taken. The advantage of profile-based prediction techniques and the other static branch prediction algorithms is that they are very simple to implement in hardware. One disadvantage of profile-based prediction is that once the predictions are made, they are forever ”set in stone” in the program.

2.2.3 Dynamic Branch Prediction

Dynamic branch prediction algorithms take advantage of the run-time information available in the processor, and can react to changing branch patterns. Dynamic branch predictors typically achieve high branch prediction rates.

There are some branches that static prediction approaches cannot handle, but the branch behavior is still fundamentally very predictable. Consider a branch that is always taken during the first half of the program, and then is always not taken in the second half of the program. Profiling will reveal that the branch is taken half of the time, and any static prediction will result in a prediction accuracy of 50 percent.

On the other hand, if we simply predict that the branch will go in the same direction as the last time we encountered the branch, we can achieve nearly perfect prediction, with

only a single misprediction at the halfway point of the program when the branch changes directions.

2.2.3.1 Smith's Algorithm

Smith's algorithm is one of the earliest proposed dynamic branch direction prediction algorithms, and one of the simplest. The predictor consists of a table that records for each branch whether or not the previous instances were taken or not taken. Each counter tracks the past branch directions. The branch address program counter (PC) is hashed down to i bits. Each counter in the table has a width of k bits. The most significant bit of the counter is used for the branch direction prediction. If the most-significant bit is a one, then the branch is predicted to be taken; if the most significant bit is a zero, the branch is predicted to be not-taken.

After a branch has resolved and its true direction is known, the counter is updated depending on the branch outcome. If the branch was taken, then the counter is incremented only if the current value is less than the maximum possible. If the branch was not taken, then the counter is decremented if the current value is greater than zero. This simple finite state machine is also called a saturating k-bit counter. The counter will have a higher value if the corresponding branch was often taken in the last several encounters of this branch. The counter will tend toward lower values when the recent branches have mostly been not taken.

Figure: 2.1 illustrates the hardware for Smith's algorithm.

The case of Smith's algorithm when k = 1 simply keeps track of the last outcome of a branch that mapped to the counter.

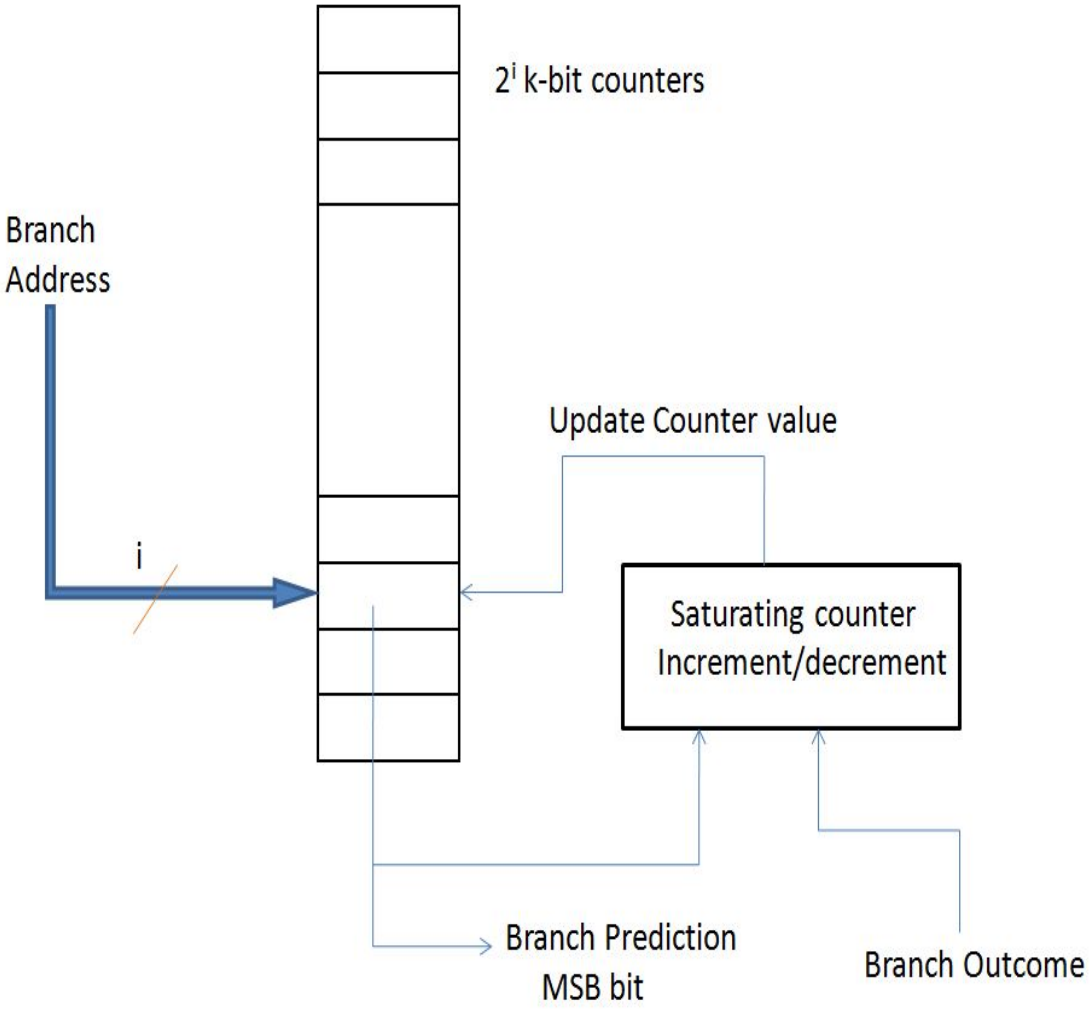


Figure 2.1: Smith Predictor with Saturating k-bit Counters

The 1-bit saturating counter (1bC) has two states: 0,1 where the state 0 is called not-taken and it indicates that the branch is not-taken, while state 1 is called taken and it reflects that the branch is taken. Some branches are predominantly biased toward one direction. A branch at the end of a for loop is usually taken, except for the case of the loop exit. This one exceptional case is called an anomalous decision. The outcomes of several of the most recent branches to map to the same counter can be used if k greater than 1. By using the histories of several recent branches, the counter will not be thrown off by a single anomalous decision. Figure: 2.2 shows hardware implementation of 1-bit saturating counter.

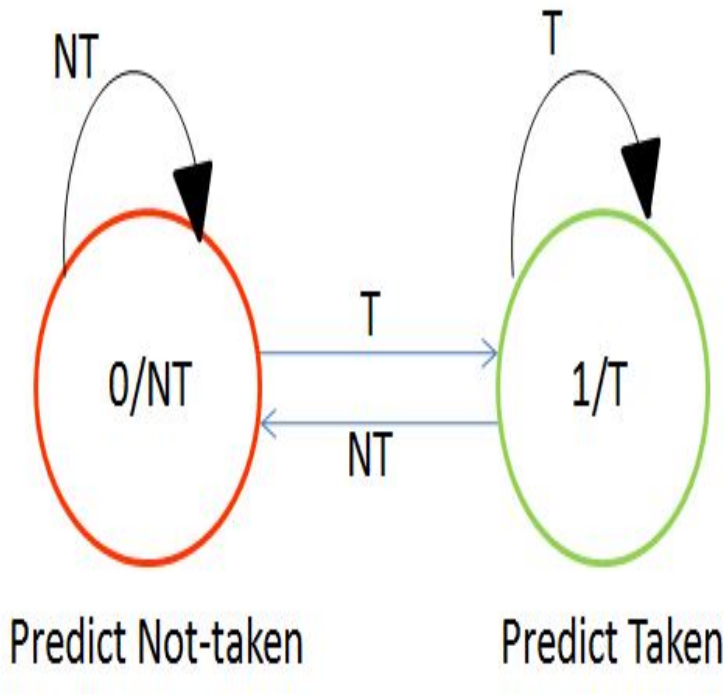


Figure 2.2: 1-bit saturating counter

The 2-bit saturating counter (2bC) is used in many branch prediction algorithms. There are four possible states: 00, 01, 10, 11. States 00 and 01, called strongly not-taken (SN) and weakly not-taken (WN), respectively, provide predictions of not-taken. States 10 and 11, called weakly taken (WT) and strongly taken (ST), respectively, provide a taken-branch prediction. The reason states 00 and 11 are called ”strong” is that the same outcome must have occurred multiple times to reach that state.

Prior to the anomalous decision, both 1-bit and 2-bit branch predictors predict the branches accurately. On the first anomalous decision, 1-bit branch predictor mispredicts because it only remembers the most recent branch and predicts in the same direction. This occurs despite the fact that the vast majority of prior branches were taken. On the other hand, 2-bit branch predictor makes the correct decision because its prediction is influenced by several of the most recent branches instead of the single most recent branch.

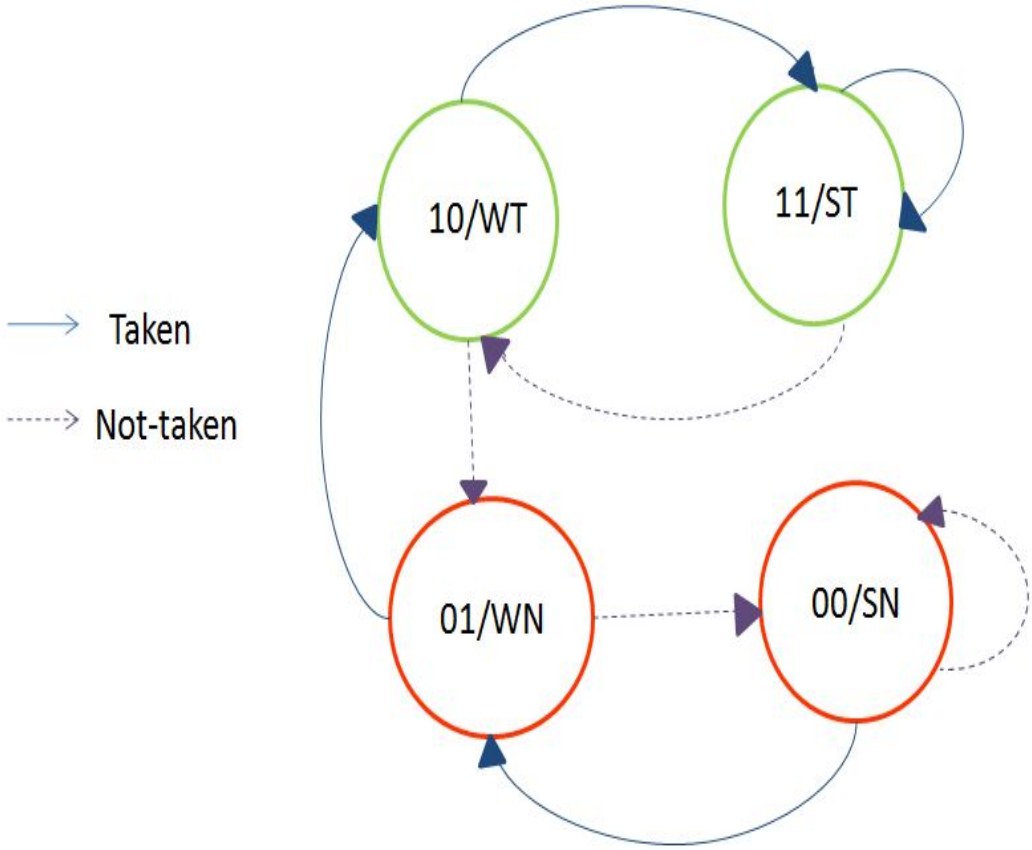


Figure 2.3: 2-bit saturating counter

For tracking branch directions, 2-bit counters provide better prediction accuracies than 1-bit counters due to the additional hysteresis. Adding a third bit only improves performance by a small increment. In many branch predictor designs, this incremental improvement is not worth the 50 percent increase in area of adding an additional bit to every 2-bit counter.

2.3 Bluespec System Verilog

Bluespec System Verilog(BSV) is a a high-level functional hardware description programming language for chip design and electronic design automation. The justification behind writing chip designs in Bluespec is that it leads to shorter, more abstract, and verifiable source code, as well as type-checked numeric code. Because of its expressive power(comparable to most advanced programming languages), full synthesizability, and quality of synthesis, Bluespec is fundamentally changing long-held assumptions about design flow.

2.3.1 Key Features of BSV

- Powerful parameterization and 'generate'.
- High-level of abstraction.
- Fully synthesizable at all levels of abstraction.
- Advanced clock management.
- Powerful static checking

2.3.2 Overview of the BSV build process

The following are the steps involved in building a BSV design:

1. A designer writes a BSV program. It may optionally include Verilog, SystemVerilog, VHDL, and C components.

2. The BSV program is compiled into a Verilog or Bluesim specification. This step has two distinct stages:

- pre-elaboration - parsing and type checking
- post-elaboration - code generation

3. The compilation output is either linked into a simulation environment or processed by a synthesis tool.

Once the Verilog or Bluesim implementation is generated, the workstation provides the following tools to help analyze your design:

- Interface with an external waveform viewer with additional Bluespec-provided annotations, including structure and type definitions.
- Schedule Analysis viewer providing multiple perspectives of a modules schedule.
- Scheduling graphs displaying schedules, conflicts, and dependencies among rules and methods.

2.3.3 Bluespec SystemVerilog Constructs

2.3.3.1 Rules

Rules are used to describe how data is moved from one state to another, instead of the Verilog method of using always blocks. Rules have two components:

- Rule conditions: Boolean expressions which determine when the rule is enabled.
- Rule body: a set of actions which describe state transitions

2.3.3.2 Modules

A module consists of three things: state, rules that operate on that state, and an interface to the outside world (surrounding hierarchy). A module definition specifies a scheme that can be instantiated multiple times.

2.3.3.3 Interfaces

Interfaces provide a means to group wires into bundles with specified uses, described by methods. An interface is a reminiscent of a struct, where each member is a method. Interfaces can also contain other interfaces.

2.3.3.4 Methods

Signals and buses are driven in and out of modules using methods. These methods are grouped together into interfaces. There are three kinds of methods:

- Value Methods: Take 0 or more arguments and return a value.
- Action Methods: Take 0 or more arguments and perform an action (side-effect) inside the module.
- ActionValue Methods: Take 0 or more arguments, perform an action, and return a result.

2.3.3.5 Functions

Functions are simply parameterized combinational circuits. Function application simply connects a parameterized combinational circuit to actual inputs.

2.3.3.6 Application Areas of Bluespec System Verilog

- Modeling for Software development
- Modeling for Architecture Exploration
- Verification
- IP creation

2.3.4 Building a design in Bluespec System Verilog

The various steps involved in building a design in BSV is shown in Figure: 2.4.

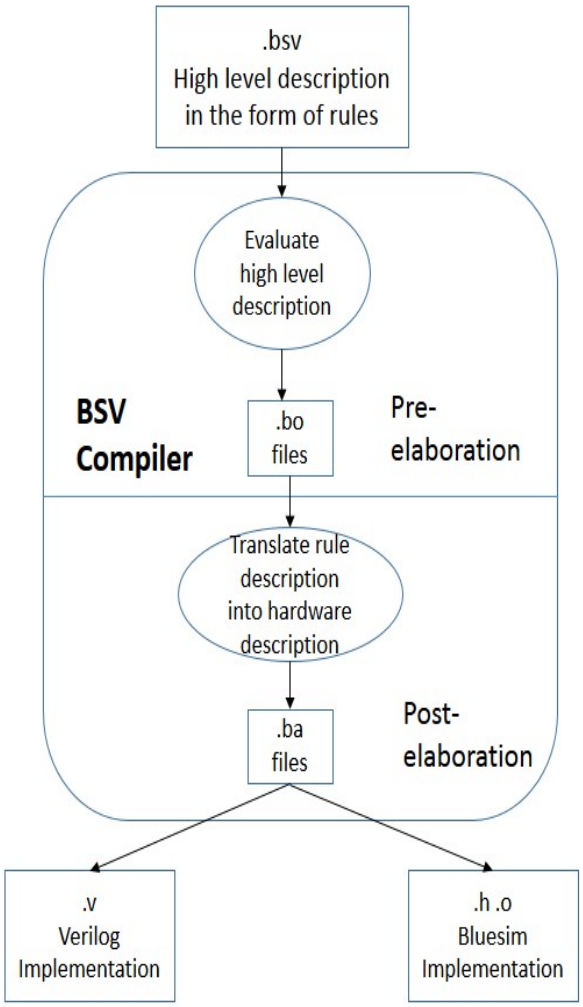


Figure 2.4: Building a design in BSV

1. The designer writes the BSV code and it may contain Verilog, VHDL and C components.
2. The BSV code is compiled into a Verilog or a Bluesim specification. This step has 2 stages:
   - Pre-elaboration parsing and type checking.
   - Post-elaboration code generation.
3. The compiled output is either linked to a simulation environment or processed by synthesis tool.

2.4 Microarchitectural description of the Superscalar Processor

The Microarchitectural description of the proposed Superscalar Processor based on the Tomasulo Algorithm is as shown in Figure: 2.5.

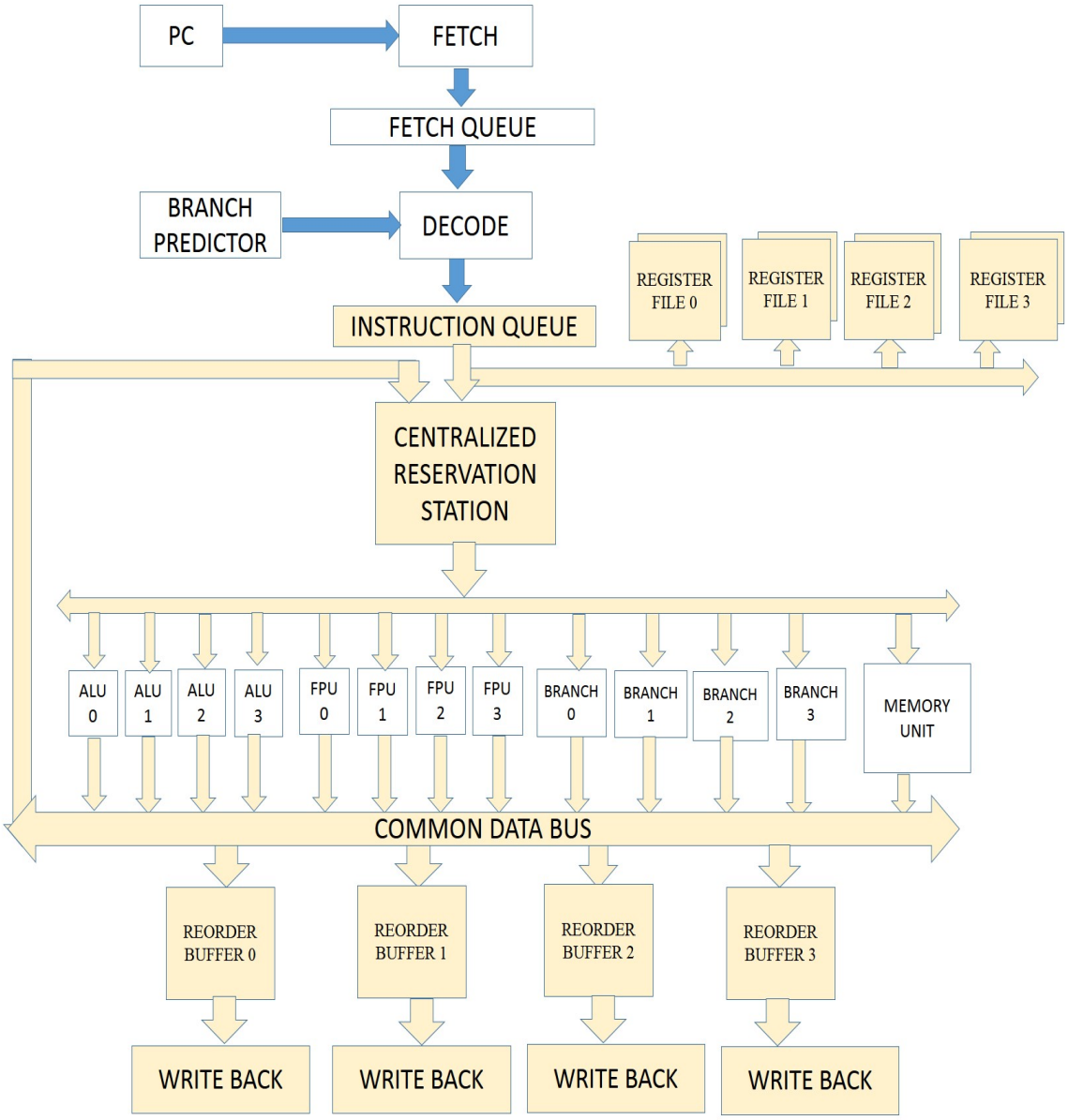


Figure 2.5: Microarchitectural description of the Superscalar Processor

The Microarchitecture is a 64bit, Quad threaded, Out of order microprocessor that is capable of fetching, decoding and issuing 4 instructions each clock cycle. The operands are fetched during issue of the instructions. The instructions after getting issued are stored in the centralized reservation station. The Reservation station can dispatch a maximum of 12 instructions: 4 of ALU, 4 of FPU, 4 of BRANCH in the same clock cycle.

There are 8 Register files: Integer register file, Floating register file for each thread. There are 4 Reorder Buffers (ROB): One Reorder Buffer for each thread. The Execution units are of ALU, FPU, BRANCH, MEMORY type. The Execution units are duplicated 4 times to avoid waiting of the ready instructions in Reservation station. The results from the execution units are forwarded to the reservation station to update the operands with the data. The Common Data Bus (CDB) used for Operand forwarding can forward the results of at most 4 execution units. Selection of the execution units which need to drive the bus is done by prioritization schemes. The Reorder Buffer can commit 4 instructions in a clock cycle. Committing the instruction can be to either register file or memory unit.

CHAPTER 3

BRANCH PREDICTION

This chapter starts with detailed description of Branch Prediction, components involved in prediction and design of the gshare, tournament and TAGE branch predictors It also compares the various configurations of converting the above predictors to handle multi-thread.

The behavior of branch instructions is highly predictable. A key approach to minimizing branch penalty and maximizing instruction flow throughput is to speculate on both branch target addresses and branch conditions of branch instructions. As a static branch instruction is repeatedly executed at run time, its dynamic behavior can be tracked. Based on its past behavior, its future behavior can be effectively predicted.

3.1 Components of Branch Prediction

There are two fundamental components of branch prediction:

- Branch Target Speculation
- Branch Condition Speculation

With any speculative technique, there must be mechanisms to validate the prediction and to safely recover from any mispredictions.

3.1.1 Branch Target Speculation

Branch target speculation involves the use of a branch target buffer (BTB) to store previous branch target addresses. BTB is a small cache memory accessed during the instruction fetch stage using the instruction fetch address (PC). Each entry of the BTB contains two fields: the branch instruction address (BIA) and the branch target address (BTA). When a static branch instruction is executed for the first time, an entry in the BTB is allocated for it. Its instruction address is stored in the BIA field, and its target address is stored in the BTA field. Assuming the BTB is a fully associative cache, the BIA field is used for the associative access of the BTB. The BTB is accessed concurrently with the accessing of the I-cache. When the current PC matches the BIA of an entry in the BTB, a hit in the BTB results. This implies that the current instruction being fetched from the I-cache has been executed before and is a branch instruction. When a hit in the BTB occurs, the BTA field of the hit entry is accessed and can be used as the next instruction fetch address if that particular branch instruction is predicted to be taken.

Figure: 3.1 shows the Branch Target Buffer.

If the branch predictor predicts not-taken, the target is simply the next sequential instruction. If the branch predictor predicts taken and there is a hit in the BTB, then the BTB's prediction is used as the next instruction's address. It is also possible that there is a taken-branch prediction, but there is a miss in the BTB. In this situation, the processor may stall fetching until the target is known. If the branch has a PC-relative target, then the fetch only stalls for a few cycles to wait for the completion of the instruction fetch from the instruction cache, the target offset extraction from the instruction word, and the addition of the offset to the current PC to generate the actual target. Another approach is to fall back to the not-taken target on a BTB miss.

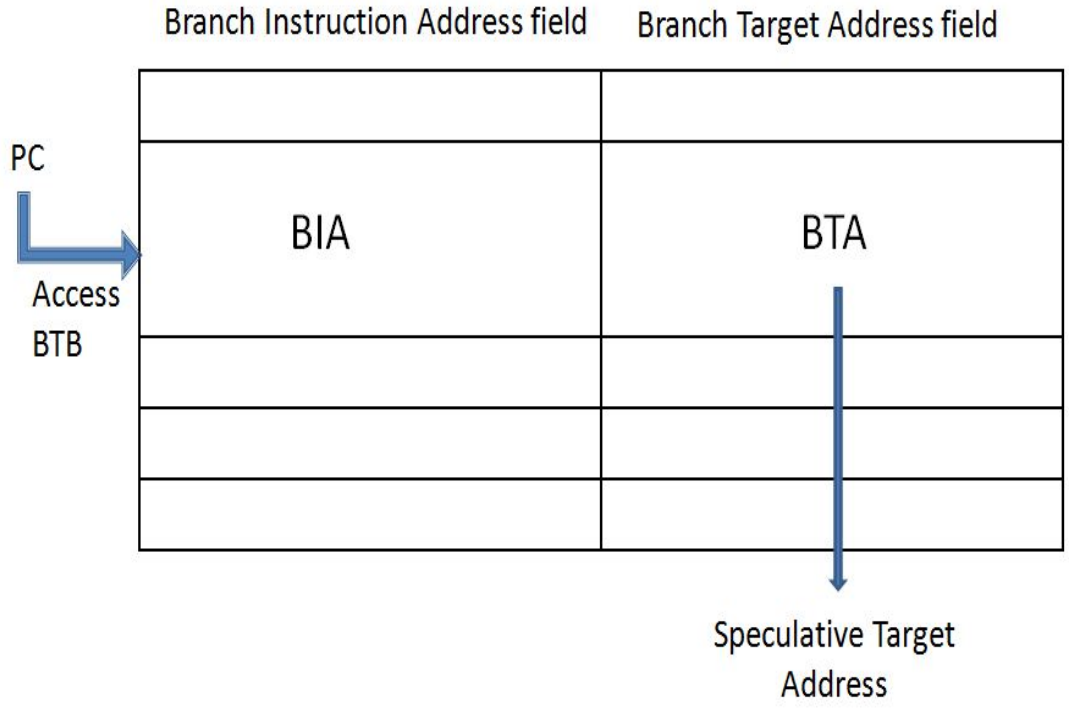


Figure 3.1: Branch Target Speculation using Branch Target Buffer

By accessing the BTB using the branch instruction address and retrieving the branch target address from the BTB all during the fetch stage, the speculative branch target address will be ready to be used in the next machine cycle as the new instruction fetch address if the branch instruction is predicted to be taken. If the branch instruction is predicted to be taken and this prediction turns out to be correct, then the branch instruction is effectively executed in the fetch stage, incurring no branch penalty.

The nonspeculative execution of the branch instruction is still performed for the purpose of validating the speculative execution. The branch instruction is still fetched from the I-cache and executed. The resultant target address and branch condition are compared with the speculative version. If they agree, then correct prediction was made; otherwise, mis-prediction has occurred and recovery must be initiated. The result from the nonspeculative execution is also used to update the content, i.e., the BTA field, of the BTB.

If the branch is discovered later on to have been mispredicted, actions are taken to recover the state of the processor to the point before the mispredicted branch, and execution is resumed along the correct path. The penalty associated with mispredicted branches in modern pipelined processors has a great impact on performance. The performance penalty is increased as the pipelines deepen.

3.2 Branch Condition Speculation

Dynamic branch predictors may require a significant amount of chip area to implement, especially when more complex algorithms are used. For small processors, such as older-generation CPUs or processors targeted for embedded systems, the additional area for these prediction structures may simply be too expensive. For larger, future-generation, wide-issue superscalar processors, accurate conditional branch prediction is critical. Furthermore, these processors have much larger chip areas, and so considerable resources may be dedicated to the implementation of more sophisticated dynamic branch predictors. An additional benefit of dynamic branch prediction is that performance enhancements can be realized without profiling all the applications that one wishes to run, and recompilation is not needed so existing binary executables can benefit.

The main idea behind the majority of dynamic branch predictors is that each time the processor discovers the true outcome of a branch (whether it is taken or not taken), it makes note of some form of context so that the next time it encounters the same situation, it will make the same prediction. dynamic branch predictors make note of context (in the form of branch history), and then make their predictions based on this information.

3.2.1 Gshare Branch Predictor

The two-level predictor employs two separate levels of branch history information to make the branch prediction. The gshare two-level predictor uses a history of the most recent branch outcomes. These outcomes are stored in the branch history register (BHR). The BHR is a shift register where the outcome of each branch is shifted into one end, and the oldest outcome is shifted out of the other end and discarded. The branch outcomes are represented by zeros and ones, which correspond to not-taken and taken, respectively. Therefore, an h-bit branch history register records the h most recent branch outcomes. The branch history is the first level of the gshare two-level predictor. The second level of the gshare two-level predictor is a table of saturating 2-bit counters (2bCs). This table is called the pattern history table (PHT). The PHT is indexed by hashing of the branch address with the contents of the BHR.

The hashing function used is a bit-wise exclusive-OR operation. The combination of the BHR and PC tends to contain more information due to the non-uniform distribution of PC values and branch histories. This is called index sharing. Indexing the BHT with the XOR of the branch history and address eliminates a significant amount of the aliasing that occurs using the traditional 2-bit prediction scheme and also takes advantage of recent branch history information.

Figure: 3.2 shows the hardware organization of a sample gshare predictor.

The counter in the indexed PHT entry provides the branch prediction in the same fashion as the Smith predictor (prediction is determined by the most-significant bit of the counter). Updates to the counter are also the same as for the Smith predictor counters: saturating increment on a taken branch, and saturating decrement on a not-taken branch.

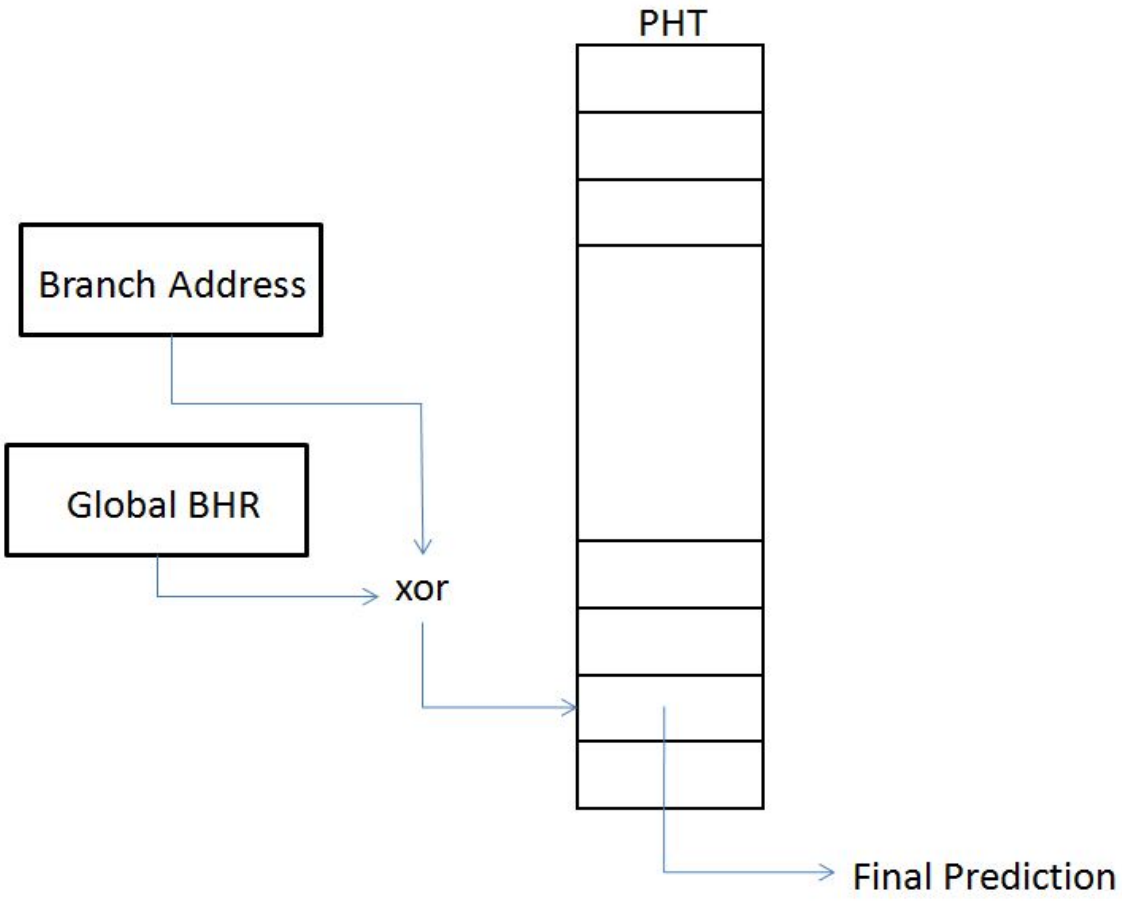


Figure 3.2: gshare predictor

When using only m bits of branch address (where m is less than the total width of the PC), the branch address must be hashed down to m bits, similar to the Smith predictor. If the number of global history bits used h is less than the number of branch address bits used m, then the global history is XORed with the upper h bits of the m branch address bits. The reason for this is that the upper bits of the PC tend to be sparser than the lower-order bits.

The intuition behind using the global branch history is that the behavior of a branch may be linked or correlated with a different earlier branch. The size of the gshare two-level predictor depends on the total available hardware budget. In general, for an X K-byte budget, the PHT will contain 4X counters.

3.2.1.1 Reasons for Branch Mispredictions

Branch mispredictions can occur for a variety of reasons. Some branches are simply hard to predict. Other mispredictions are due to the fact that any realistic branch predictor is limited in size and complexity. There are several cases where a branch is fundamentally unpredictable. The first time the predictor encounters a branch, it has no past information about how the branch behaves, and so at best the predictor could make a random choice and expect a 50 percent prediction rate. A similar situation occurs any time the predictor encounters a new branch history pattern. A predictor needs to see a particular branch (or branch history) a few times before it learns the proper prediction that corresponds to the branch (or branch history). During this training period, it is unlikely that the predictor will perform very well.

If the program enters a new phase of execution (for example, a compiler going from parsing to type-checking), branch behaviors may change and the predictor must relearn the new patterns.

The physical constraints on the size of branch predictors introduces additional sources of branch mispredictions. For example, if a branch predictor has a 128-entry table of counters, and there are 129 distinct branches in a program, then there will be at least one entry that has two different branches mapped to it. If one of these branches is always taken and the other is always not taken, then they will interfere with each other and cause branch mispredictions. Interference is also called aliasing because both branches are aliases for the same predictor entry.

For capacity problems, the only solution is to increase the size of the predictor structures. This is not always possible due to die area, latency, and/or power constraints. For aliasing, a wide variety of algorithms have been developed to address this problem, couple of those predictors are to be followed.

3.2.2 Tournament Branch Predictor

Different branches in a program may be strongly correlated with different types of history. Because of this, some branches may be accurately predicted with global history-based predictors, while others are more strongly correlated with local history. Programs typically contain a mix of such branch types, and for example, choosing to implement a global history-based predictor may yield poor prediction accuracies for the branches that are more strongly correlated with their own local history.

Figure: 3.3 illustrates the hardware for the tournament selection mechanism with two generic component predictors P0 and P1

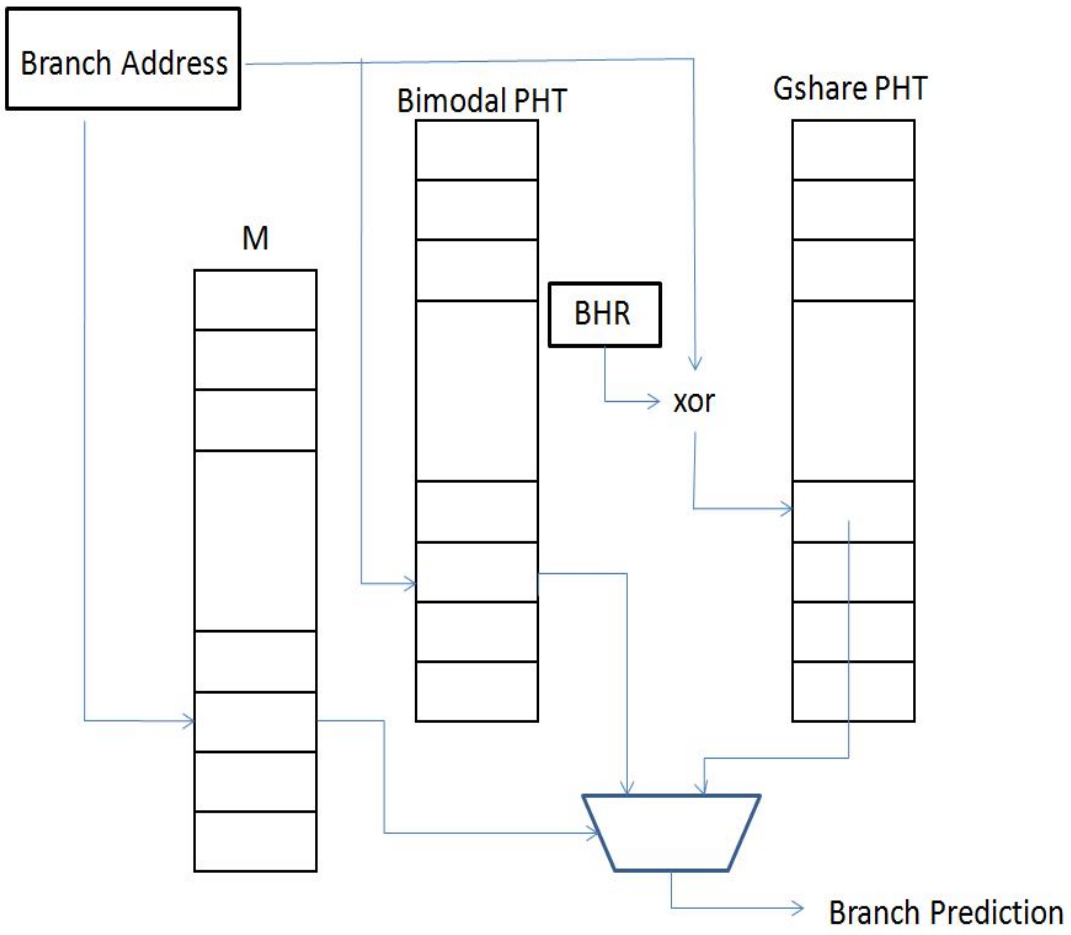


Figure 3.3: Tournamnet predictor

To address this issue, a proposed multischeme branch predictor is the tournament algorithm. The predictor consists of two component predictors P0 (bi-modal predictor) and P1 (gshare prredictor) and a meta-predictor M. The component predictors can be any of the single-scheme predictors.

The meta-predictor M is a table of 2-bit counters indexed by the low-order bits of the branch address. This is identical to the lookup phase of 2-bit predictor , except that a (meta-prediction of zero indicates that P0 should be used, and a (meta-)prediction of one indicates that P1, should be used (the meta-prediction is made from the most-significant bit of the counter). The meta-predictor makes a prediction of which predictor will be correct.

Figure: 3.4 lists the state transitions of the tournament meta- predictor

P0 Correct ?	P1 Correct?	Modification to M
0	0	Do Nothing
0	1	Saturating Increment
1	0	Saturating Decrement
1	1	Do Nothing

Figure 3.4: Tournamnet meta-predictor update rules

After the branch outcome is available, P0 and P1, are updated according to their respective update rules. Although the meta-predictor M is structurally identical to 2-bit predictor, the update rules (i.e., state transitions) are different. When P1's prediction was correct and P0 mispredicted, the corresponding counter in M is incremented, saturating at a maximum value of 3. Conversely, when P1, mispredicts and P0 predicts correctly, the counter is decremented, saturating at zero. If both P0 and P1, are correct, or both mispredict, the counter in M is unmodified.

The prediction lookups on P0, P1 and M are all performed in parallel. When all three predictions have been made, the meta-prediction is used to drive the select line of a multiplexer to choose between the predictions of P0 and P1.

Either or both of the two components of a tournament hybrid predictor may themselves be hybrid predictors. By recursively arranging multiple tournament meta-predictors into a tree, any number of predictors may be combined.

3.2.3 TAGE predictor

State-of-art conditional branch predictors exploit several different history lengths to capture correlation from very remote branch outcomes as well as very recent branch history. Hybrid predictors were initially relying on meta-predictors to select a prediction from a few different predictors.

The Optimized GEometric History Length (O-GEHL) predictor is able to exploit very long history lengths in the hundreds bits range. It achieved state-of-art branch prediction accuracy for storage budgets in the range of 32K bit-1M bit range. It uses a medium N of prediction tables (ex. 4 to 8) and limited hardware logic for prediction computation and it is the most storage-effective reasonably implementable conditional branch predictor.

The O-GEHL predictor relies on an adder tree as the final prediction computation function, but its main characteristic is the use of a geometric series as the list of the history lengths. This allows the O-GEHL predictor to exploit very long history lengths as well as to capture correlations on recent branch outcomes.

TAGE stands for TAgged GEometric history length. TAGE is derived from tagged PPM-like (Prediction by Partial Match) predictor. It relies on a default tagless predictor backed with a plurality of (partially) tagged predictor components indexed using different history lengths for index computation. These history lengths form a geometric series. The prediction is provided either by a tag match on tagged predictor component or by default predictor. In case of multiple hits, the prediction is provided by the tag matching table with the longest history. The main contributions of the conditional branch predictor are the use of geometric history length series in PPM-like predictors and a new and efficient predictor update algorithm. TAGE predictor outperforms the O-GEHL predictor at equal storage budgets and equivalent predictor complexity (number of tables, computation logic, etc.)

TAGE is more cost effective solution for this prediction combination functions for predictors relying on several predictor components indexed with different history lengths. Using geometric history length as the O-GEHL predictor, the TAGE predictor uses (partially) tagged components as the PPM-like predictor. TAGE relies on (partial) hit-miss detection as the prediction computation function. TAGE provides state-of-art prediction accuracy on conditional branches.

The quality of the prediction scheme is very dependent on the choice of the final prediction computation function, but also on the careful design of the predictor update policy. With the current update policy, partial tagging appears to be more efficient than adder tree for final prediction computation.

3.2.3.1 Index Computation

Branch predictor behavior might be sensitive to the initialization state of the predictor. In order to approach a realistic initialization point, the predictor counters are in reset state. For computing indexes for global history predictors, hashing the branch history with the branch address is done (similar to gshare index computation).

While computing index, the global history has to be folded in order to access the tagged structures. A possible way to implement history folding would be to use a tree of XORs. For example, for the 40-bit history, h[0 : 9] xor h[10 : 19] xor h[20 : 29] xor h[30 : 39] requires a depth-2 tree (assuming 2-input XORs). For the 80-bit history, this requires a depth-3 tree. In practice, history folding can be implemented by taking advantage of the fact that we are not folding a random value, but a global history value derived from the previous history value. In a similar fashion, tag can also be generated by folding PC and global history together. The TAGE predictor relies on using a very long global branch history (in hundred bits range). This global history is speculatively updated and must be

restored on misprediction. This can be implemented using shift registers to store global branch history.

3.2.3.2 Predictor Structure

Geometric history length prediction was introduced in O-GEHL predictor. It features M distinct prediction tables Ti, indexed with hash functions of the branch address and global history. Distinct history lengths are used for computing the index of distinct tables. Table T0 is indexed using the branch address. The history lengths used for computing the indexing functions for tables Ti, are of the form

L(i)=apow(i-1)*L(1)

i.e., the lengths L(i) form a geometric series.

Using a geometric series allows to use very long history lengths for indexing some prediction tables, while still dedicating most of the storage space to predictor tables using short history lengths. For example, in the current structure of 5-component predictor, the first component is default predictor, it is indexed by branch address (PC) and the remaining components are indexed using a =2 and L(1) = 10 leads to the following series 10, 20, 40, 80.

Figure: 3.5 illustrates a TAGE predictor. The TAGE predictor features a base predictor T0 in charge of providing a basic prediction and a set of (partially) tagged predictor components Ti. These tagged predictor components Ti, are indexed using different history lengths that form a geometric series

The base predictor used in TAGE is a simple PC-indexed 2-bit counter bimodal table. An entry in a tagged component consists a counter ctr which provides the prediction, a (partial) tag and a useful counter u, both u and ctr are 2-bit counters.

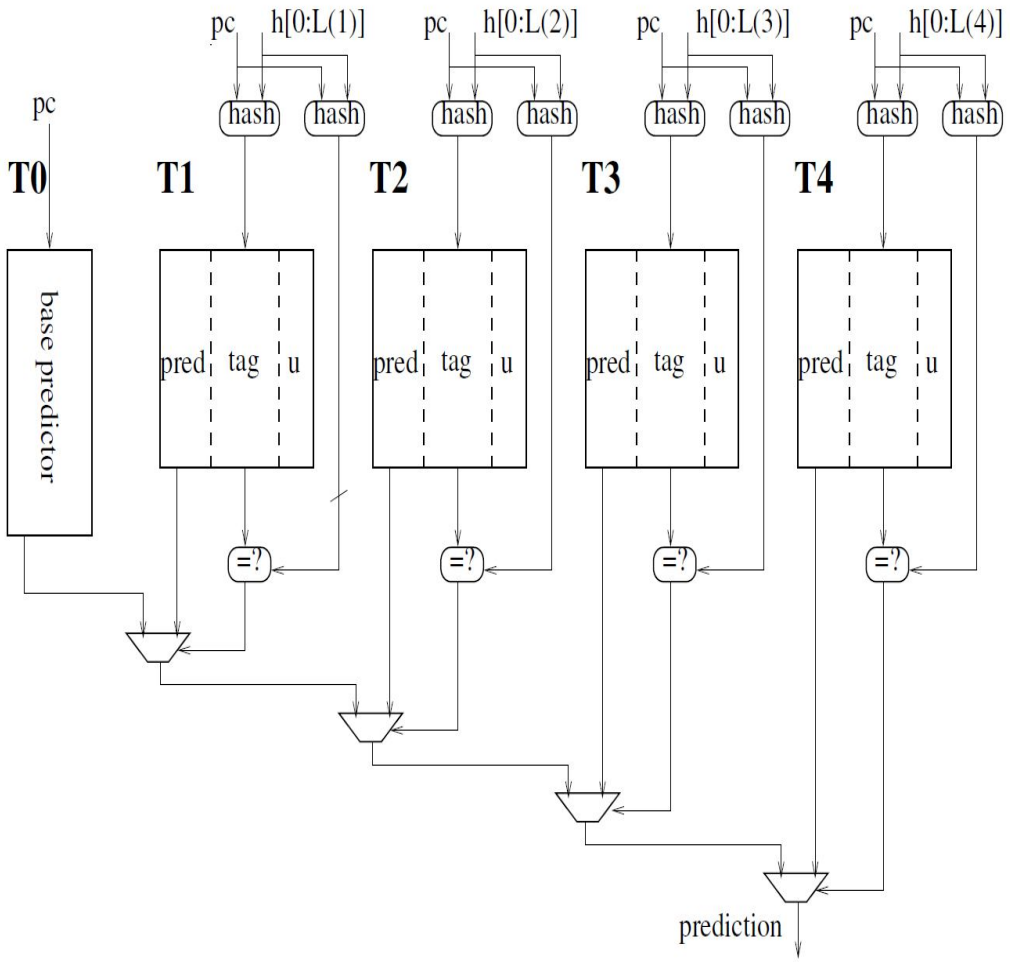


Figure 3.5: 5-component TAGE predictor

3.2.3.3 Prediction Computation

At prediction time, the base predictor and the tagged components are accessed simultaneously. The base predictor provides a default prediction. The tagged components provide a prediction only on a tag match. The final prediction is provided by the hitting tagged predictor component that uses the longest history, in case of no matching tagged component, the default prediction in used.

In the remainder of the thesis, the provider component means the predictor component that ultimately provides the prediction and alternate prediction (altpred) as the prediction that would have occurred if there had been a miss on the provider component.

For example, if there are tag hits on T2 and T4 and tag misses on T1 and T3, then T4 is the provider component and T2 provides altpred. If there is no hitting component then altpred is the default prediction.

3.2.3.4 Predictor Updation

The useful counter u of the provider component is updated when the alternate prediction altpred is different from the final prediction pred. u is incremented when the actual prediction is correct and decremented otherwise. Moreover, the useful counter is also used as an age counter and is gracefully reset. Periodically, the whole column of the most significant bits of the u counters is reset to zero, then the whole column of the least significant bits are reset. If the graceful resetting of useful counters is missing, some entries will be marked useful almost indefinitely.

On correct prediction, the prediction counter of provider component is updated (incremented). If the overall prediction is incorrect, first we update (decrement) the provider component prediction counter. As a second step, if the provider component Ti is not the component using longest history, we try to allocate an entry on a predictor component Tk using a longer history than Ti. Atmost a single component is allocated. The allocation process is described below.

All uk counters are read from predictor components Tk, we apply these rules:

1) Priority for allocation

- If there exists some k, such that entry uk = 0 then the component Tk is allocated Else
- The u counters from the components Tk, are all decremented (no new entry is allocated)

2) Avoiding ping-pong phenomenon: If two components Tj and Tk, j less than k can be allocated (i.e., uj = uk = 0) then Tj is chosen with a higher probability than Tk. This can be implemented in hardware using a simple linear feedback register.

3) Initializing the allocated entry: The allocated entry is initialized with the prediction counter set to weakly taken (01). Counter u is initialized to 0 (i.e., strongly not useful)

This new entry delivers the prediction on the occurrence of the same (history, PC) pair.

The update policy is designed to minimize the perturbation induced by a single occurrence of a branch, in order to minimize it at most one tagged entry is allocated on a misprediction. The selection of the entry to be allocated is managed through the useful counter u. We manage u with two objectives, the value of u is high only if there was some benefit since the last allocation of the entry: rule (1) guarantees that entries that were recently used are not reallocated. Second, by decreasing the u counter when the entry is not selected, and its graceful reseting, we try to mimic a pseudo least recently used policy. The useful counter u is set to strong not useful, until the entry effectively helps to provide a correct prediction, it is the natural target for next replacement. This property could induce ping-pong phenomena on branches competing for a single entry, rule (2) was used to avoid such ping-pongs.

Updating the predictor is not on the critical path. Therefore, complex update policy may be applied. While studies have detailed update policies in branch predictors, the importance of a careful design of the update policy has been rarely pointed out.

Most of the storage budget in the TAGE predictor is dedicated to the storage of the (partial) tags. Therefore the main task is to correctly dimension the tag width. Using a large tag width leads to waste part of the storage while using a short tag width leads to false tag match detections, which may result in a misprediction that may trigger a new entry allocation. This new entry may eject some useful prediction.

3.2.3.5 Implementation Issues

Prediction response time:

The Prediction response time on most global predictors involves three components: the index computation, the predictor table read and the prediction computation logic. Very simple indexing functions using two stages of 3-entry exclusive-OR gates can be used for indexing the predictor components without impairing the prediction accuracy. The prediction table read delay depends on the size of the tables. In TAGE predictor, (partial) tags are needed for the prediction computation. The tag computation may span during the index computation and table read without impacting the overall prediction computation time. The last stage in the prediction computation on the TAGE predictor consists of the tag match followed by the prediction selection. The tag match computations are performed in parallel on the tags flowing out from the tagged components.

At equal storage budgets, the response time is slightly longer than that of more conventional gshare and tournament predictors for which the prediction computation is simpler.

Update implementation

The predictor update must be performed at commit time. On correct prediction, only the prediction counter ctr and the useful counter u of the matching component must be updated i.e., single predictor component is accessed. On a misprediction, a new entry must be allocated in a tagged component. Therefore a prediction can potentially induce up to three accesses to the predictor, i.e., read of all prediction tables at prediction time, read of all prediction tables at commit time and write of two predictor tables at update time.

3.3 SMT Branch Predictor Design

3.3.1 SMT Overview

In general, multithreading permits additional parallelism in a single processor by sharing the functional units across various threads of execution. In addition to the sharing of functional units, each thread must have a copy of the architected state. To combat this overhead, many other processor resources end up being shared by multiple threads. Accomplishing this sharing of resources also requires that the processor be able to perform switching between the threads efficiently, so as to not drastically affect performance. In fact, one can hide many of the memory stalls and control changes by switching threads at those occurrences.

Simultaneous multithreading (SMT) processors are able to use resources that are provided for extracting instruction-level parallelism (e.g. rename registers, out-of-order issue) to also extract thread-level parallelism. An SMT processor uses multiple-issue and dynamic scheduling resources to schedule instructions from different threads that can operate in parallel. Since the threads execute independently, one can issue instructions from these various threads without considering dependencies between them.

SMT is a form of multithreading that seamlessly interleaves the processing of instructions from multiple threads of execution in a single, shared processor pipeline. In a typical SMT system, a single set of branch predictors is shared between several concurrently running threads.

Branch prediction in simultaneous multithreaded processors is difficult because multiple independent threads interfere with shared prediction resources. This interference can be positive or negative. As threads with very different control behavior interfere with

shared prediction resources leading to negative interference within the branch predictor. The interference that occurs due to the sharing of branch predictors between threads could in fact be positive aliasing, where the leading thread trains the predictor for the trailing threads with similar control flow.

3.3.2 Exploring Efficient SMT Branch Predictor Design

There are four possible configurations for branch prediction in simultaneous multithreaded processors:

- Shared Configuration
- Split Branch Configuration
- Split Branch Table Configuration
- Split History Configuration

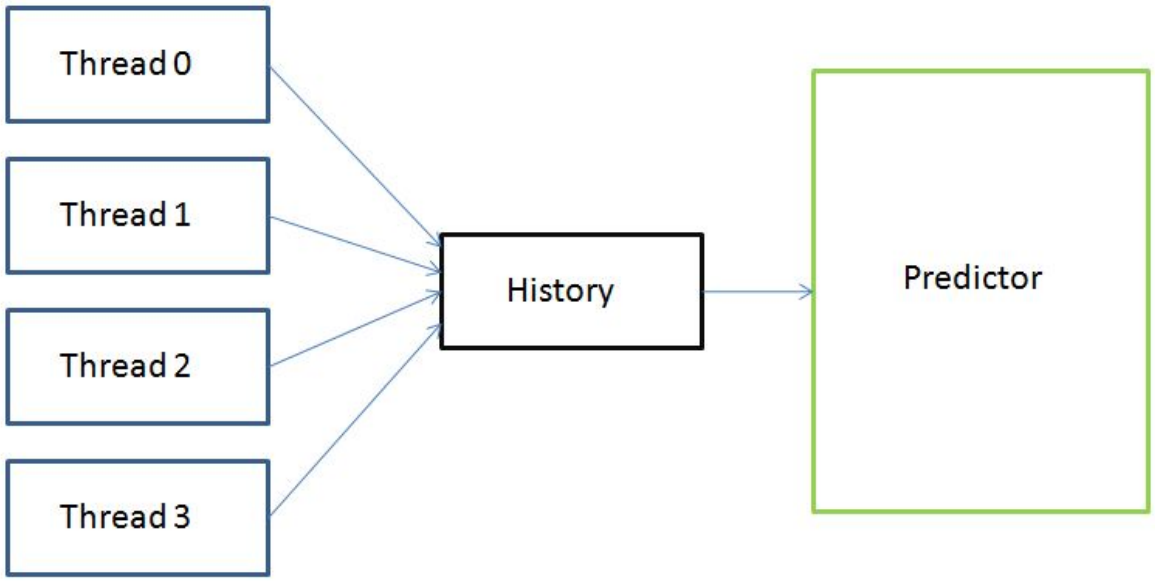


Figure 3.6: Shared Configuration

3.3.2.1 Shared Configuration

The most resource conservative branch predictor configuration for an SMT is a totally shared predictor (Figure: 3.6). In this case, each thread shares both the history register and BHT. This configuration allows for the most interference between threads. This interference can occur both in the history register and in the BHT. This interference reduces the accuracy. This configuration allows most interference and requires least hardware.

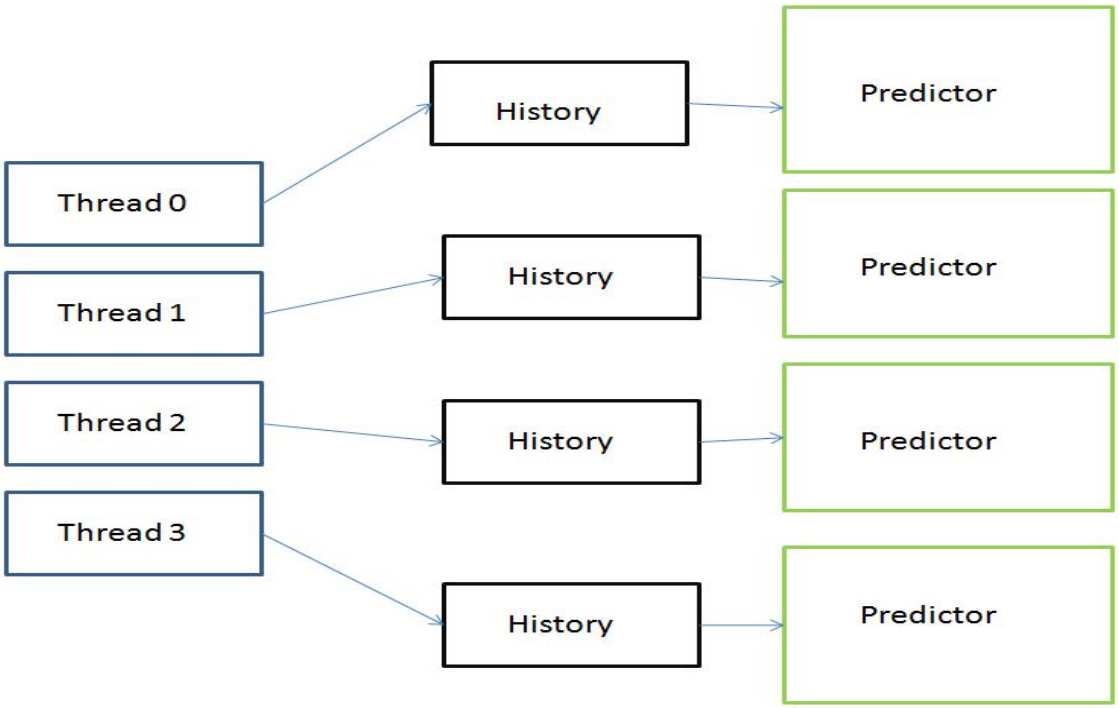


Figure 3.7: Split Branch Configuration

3.3.2.2 Split Branch Configuration

The next logical configuration was providing each thread with its own predictor (Figure: 3.7). This configuration completely eliminates interference between threads. In this case, the predictor acts exactly as it would in a single-threaded environment. This configuration requires most hardware. This configuration provides highest prediction accuracy.

3.3.2.3 Split Branch Table Configuration

The third configuration is a partial split of the branch predictor (Figure: 3.8). In this case, each thread accesses a common branch history register, but then indexes into its own predictor. This configuration allows interference only in the branch history register.

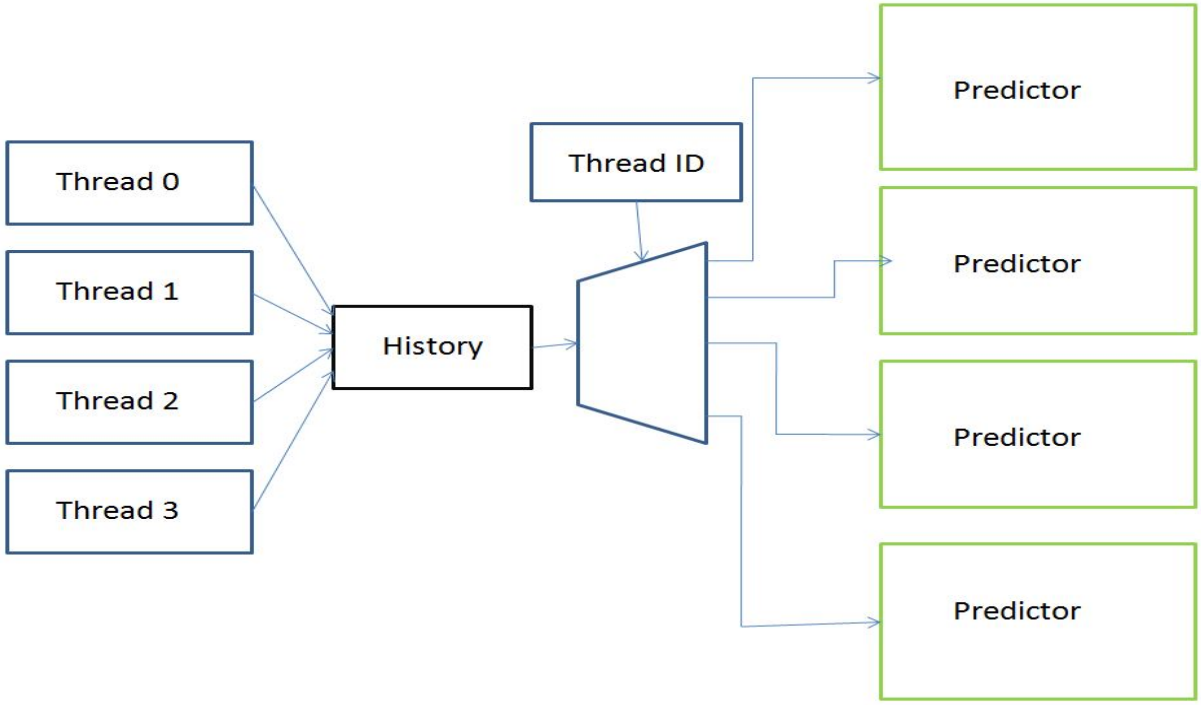


Figure 3.8: Split Branch Configuration

3.3.2.4 Split History Configuration

The final branch predictor configuration is the opposite of partial split of predictor resources (Figure: 3.9). This configuration allots each thread its own history register while indexing into a common predictor. This configuration again only allows interference at one of the two possible places, in the predictor. By only replicating the branch history register, a small resource, instead of the predictor, a much larger resource, the split history configuration eliminates one of the sources of interference in a much more cost and space efficient manner than the split branch table configuration. The hope of this configuration is

that interference in the larger predictor will have less effect than interference in the smaller branch history register.

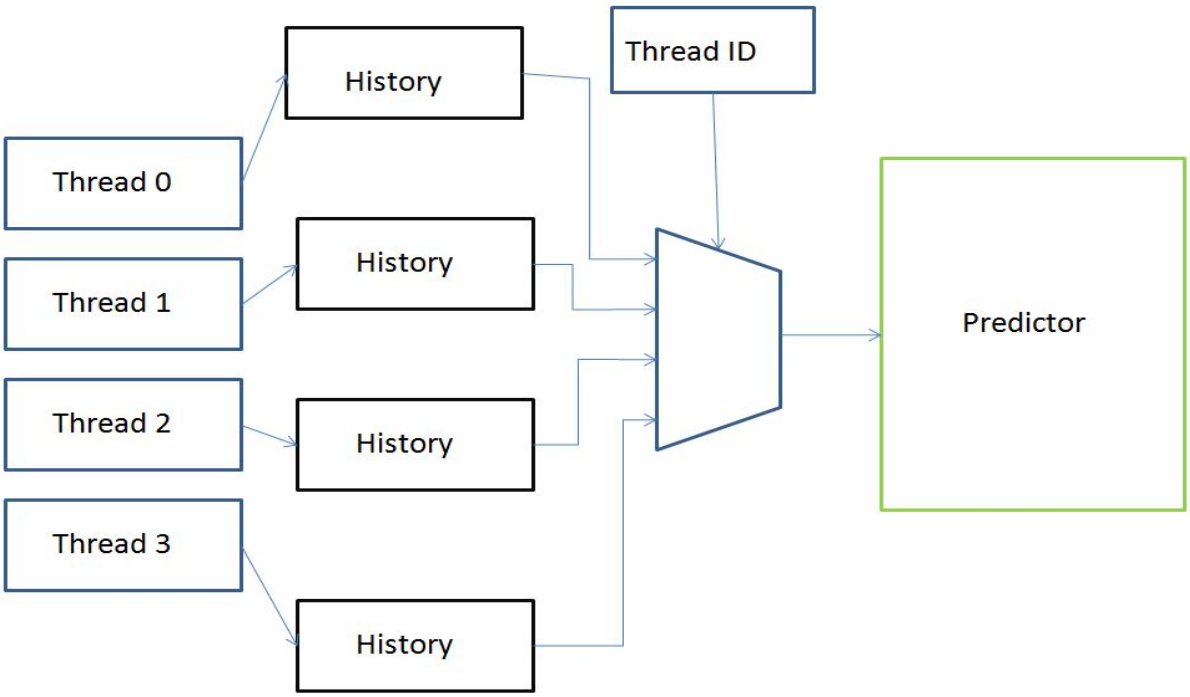


Figure 3.9: Split History Configuration

Considering all the facts like hardware complexity, reduction in the accuracy by sharing resources, the fourth configuration split history turns out to be the best way because of the relative unimportance of branch prediction accuracy in a multithreaded environment, we conclude that valuable development time and on-chip resources should be applied to other more important issues.

All the three branch predictors discussed in this chapter (gshare, tournament and TAGE) are implemented for SMT using this split history configuration technique discussed above.

CHAPTER 4

INSTRUCTION PREFTCHING

4.1 Need for Prefetching

As processor speeds have increased over the past few decades, the gap between memory access latency and the processor cycle time has steadily increased. Consequently, the performance penalty of an I-cache miss has increased. In addition, current superscalar processors fetch and issue multiple instructions per cycle which compounds the miss penalty. Reducing I-cache misses is therefore critical for restoring a balance between the fetch and the execution stages of these wide issue processors.

The common solution for reducing I-cache misses is to increase the cache size. However, the size of the I-cache is limited by timing and area considerations, and may also be reaching a point of diminishing returns. Prefetching would be a more viable alternative, if an effective predictor of instruction miss addresses were available.

With the current trends toward wider issue processors and an increasing gap between the CPU cycle time and the memory access latency, to avoid a fetch bottleneck it is critical to issue prefetches earlier.

4.2 Branch History Guided Prefetching

Branch History Guided Prefetching (BHGP), that uses branches as trigger points to initiate prefetches to candidate blocks. The prefetch candidate blocks contain instructions that resulted in I-cache misses (N - 1) branches after some previous execution of the triggering branch. BHGP selects its prefetch candidate block without needing a branch predictor to predict the outcome of each of these N branches.

Generally, the target address of a branch instruction is the beginning of a basic block which may span multiple cache lines. BHGP maintains both the address and the length of prefetch candidate blocks, so that entire blocks can be prefetched in a timely fashion.

4.2.1 BHGP Structure

Branch History Guided Prefetching (BHGP) exploits a correlation between the execution of a branch instruction and later I-cache misses. This correlation is expected since control flow changes caused by branches lead to many I-cache misses. BHGP identifies those branches that are followed by I-cache misses at an appropriate later time and exploits the regularity of this correlation to prefetch some candidate block of instructions.

For example, a branch instruction (Br-1) will be associated with candidate block (BB-I) if there is an I-cache miss to BB-1 exactly (N - 1) branches after Br-1 is executed. When Br-1 is next executed, a prefetch for BB-1 is initiated.

The prefetch hardware consists of 5 structures: Prefetch Table (PT), Branch History Queue (BHQ), and three registers: BB, L, and M. The PT is a small associative cache. Each entry of the PT contains the address of a branch instruction, the beginning address of its associated prefetch candidate block, and the length of the block (in cache lines). The BHQ is maintained as a FIFO buffer and always holds the addresses of the most recent N

branches executed (N = 5 in Figure 1). Whenever a branch is executed by the processor it is enqueued to the tail of the BHQ and the entry at the head of the BHQ is pushed out (dequeued). The BB register holds the address of the instruction that followed the most recently executed branch (potentially the beginning address of the prefetch candidate block for the branch at head of the BHQ).

For example, Br 5 is the most recent branch and BB-18 is the address of the instruction executed after Br5; BB-18 may become the beginning address of the next prefetch candidate block to be associated with the branch at the head of the BHQ, namely, Br-1 (presently BB-12 is associated with Br-I). The L register stores the number of I-cache lines referenced since the most recent branch and will be used as the length of the prefetch candidate block. The M register is a 1 -bit flag which is reset to 0 whenever a branch instruction is executed and is set to 1 whenever there is an I-cache miss.

Figure: 4.1 illustrates the operation of BHGP using a high level block diagram.

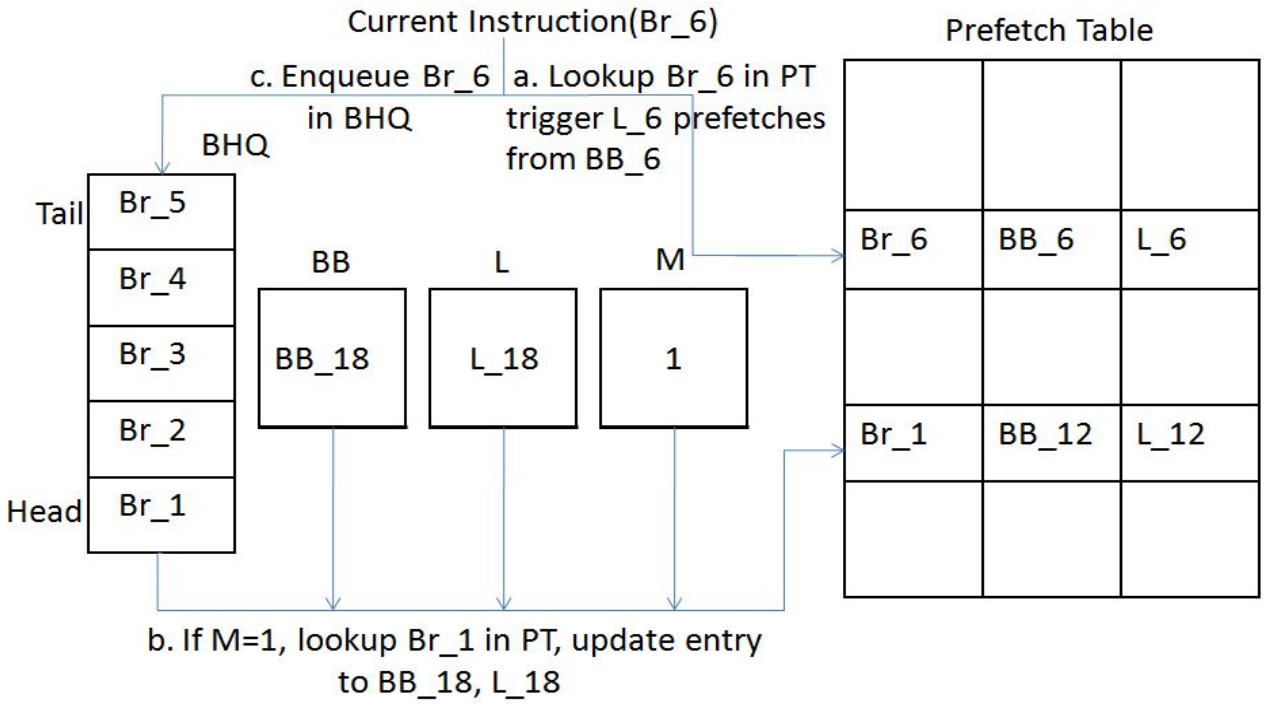


Figure 4.1: BHGP Operation

4.2.2 BHGP Operation

Consider the sequence Br-l, Br-2, Br-3, Br-4, Br-5, Br-6 of six branches in an execution. Each branch may be taken or not taken and the BHQ may contain multiple instances of the same branch. Figure 4.1 shows the state of BHQ some time after Br5 is executed, but before Br-6; suppose that the current state of PT is as shown. BB shows that BB-18 is the block that was entered after Br5, the L register is counting up the number of cache lines in BB-18, and M = 1 indicates that there was an I-cache miss when referencing some instruction in BB-18. Eventually branch Br-6 (the final instruction of BB-18) becomes the current instruction and the following three events occur:

a) The PT is searched for a Br-6 entry. If there is a match, a prefetch is initiated for the entire candidate block (BB-6 of length L-6 lines in Figure 4.1).

b) If M = 1, there was an I-cache miss after Br-5 and the corresponding prefetch candidate block in the BB register (BB-18) and its length in the L register (L-18) need to be associated with the branch at the head of the BHQ, Br-1 in Figure 4.1. If a PT entry for Br-1 already exists, it is updated to BB-18 and L-18. Otherwise, a new entry is created with this information and replaces some other PT entry.

c) Br-6 is enqueued to the tail of the BHQ and Br-1 is pushed out of the BHQ. In addition, the L register is set to 1, M is reset to 0, and the BB register is updated with the address of the first instruction after Br-6.

It is important to note that if no cache miss occurs after Br5 is executed, then M is still 0 when Br-6 occurs, and the PT entry for Br-1, if any, is left unchanged. Each PT entry thus associates a branch with its most recently missed (MRM) candidate block, i.e., the block that experienced a miss most recently while that branch resided at the BHQ head.

To limit the number of useless prefetches, BHGP prefetches the MRM candidate block only if it is not already present in the cache and its confirmation bit is set to 1. Each line in the next level of memory hierarchy (L2 cache) has a confirmation bit that is used to track whether the line was referenced while it resided in L1 after it was last prefetched. The confirmation bit of a line is initialized to 1 and is reset to 0 whenever the line is prefetched and replaced in L1 without being used; it is set to 1 again only when the line experiences a demand miss. Prefetch requests are squashed if the lines confirmation bit is 0 in the L2 directory.

CHAPTER 5

VERIFICATION

5.1 Verification Setup

The set up for verification process is shown in fig: 5.1.

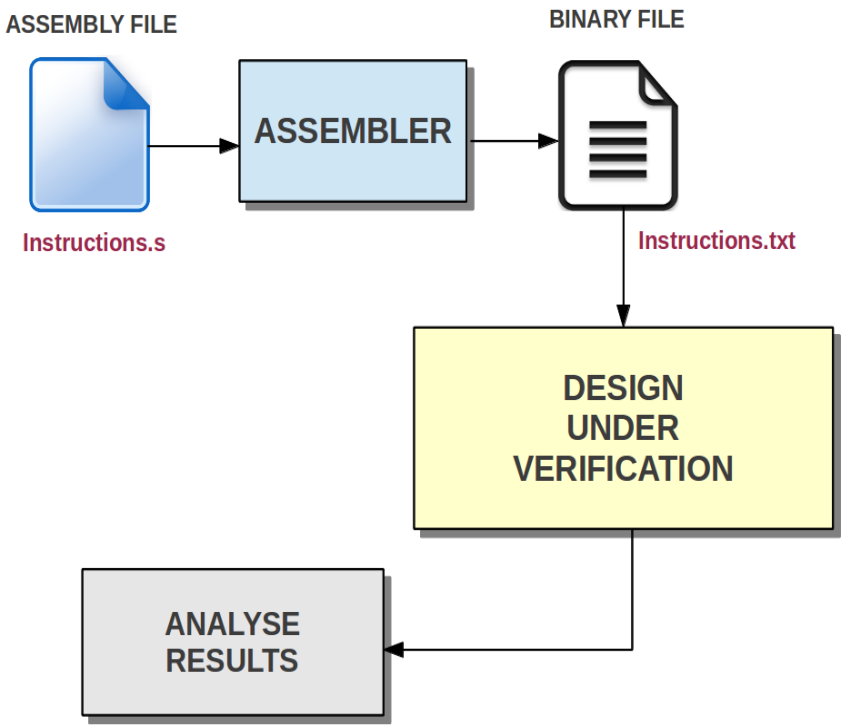


Figure 5.1: Verification Setup

Bluespec Compiler converts the design in BSV to synthesizable Verilog code. Questa simulator was used to verify the various functionalities of the design.

The functionality of the design was verified using a test bench which provides a set of instructions composed of various test scenarios to the main module. ***display*** statements were embedded/included in the code to monitor the functionality of the modules in various clock cycles. The design was simulated using Questa simulator (Modelsim).

The test instructions in mnemonics form (assembly code) are given to the ABACUS assembler. The assembler gives the instructions in binary form. These are stored in a data file and are used in the test bench for verifying the design.

5.2 Verification Strategy

The verification process mainly focused on verifying functional behaviour of the design. The main challenge involved in the verification of the branch predictor was that it gets the feed back from the execution unit on whether the prediction was correct or not. So the test bench was so modified that it would also process the instruction along with the branch predictor and produce the results after required number of cycles. These results are then used to update the branch history tables.

So, the test scenarios were generated to verify the following aspects of the design.

- Filling up of branch history tables.
- Updation of prediction bits.
- Branch Predictor SMT behaviour.
- Address conversions to point to the correct BTB.
- Updation of Global BHR.

5.3 Synthesis Report

The design has been synthesized using Xilinx ISE for Virtex 6 XC6VLX240T-FF1156. All default settings were used. The design strategy was set to optimization for speed. The slice utilization and timing summary are provided below:

Device Utilization Summary :

Slice Logic Utilization:

Number of Slice Registers : 1142 out of 301440

Number of Slice LUTs : 662 out of 150720

Number used as Logic : 662 out of 150720

Number of fully used LUT-FF pairs: 488 out of 11923

Timing Summary :

Minimum period: 3.581ns

Maximum Frequency: 279.252MHz

Minimum input arrival time before clock: 1.186ns

Maximum output required time after clock: 0.779ns

CHAPTER 6

CONCLUSION AND FUTURE WORK

Gshare, Tournament and TAGE branch predictors are successfully implemented in Bluespec System Verilog. In order to achieve high accuracy conditional branch predictors must exploit several different history lengths to capture correlation from very remote branch outcomes as well as very recent branch history. Branch prediction accuracy is less important in an SMT system than in a traditional superscalar processor. This phenomenon occurs because other threads fill in the processing void left when a long latency hazard is encountered by one. BHGP causes each branch execution to trigger a prefetch of its MRM candidate block. In this technique branch instructions are used to trigger the prefetch candidate blocks occurring (N - 1) branches later.

Future Work:

Simple tuning of TAGE branch predictor for allowing slightly better usage of storage budget

- First to enhance the behaviour of the bimodal base predictor.
- Second, using slightly different tag widths on different tagged components make a better usage of the storage space. More precisely, width of the tag should increase with history length: a false match is more harmful on the components using longest history.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture (5th Edition)*. Morgan Kaufmann, 2012.

[2] J. P. Shen and M. H. Lipasti, *Modern Processor Design - Fundamentals of Superscalar processors*. TATA McGraw-Hill Publishing Company Private Limited, 2005.

[3] R. S. Nikhil and K. Czeck, *BSV by Example*. Bluespec, Inc, 2010.

[4] Bluespec, Inc, *Bluespec System Verilog Reference Guide*, revision: 17 ed., 2012.

[5] User Level RISC-V ISA, *University of California, Berkeley*, revision: 2 ed., 2014.

[6] Proceedings of the 32nd Annual International Symposium on Computer Architecture, *Analysis of the O-GEHL branch predictor.*, 2005.

[7] Journal of Instruction Level Parallelism, *A case for (partially)-tagged geometric history length predictors*, 2006.