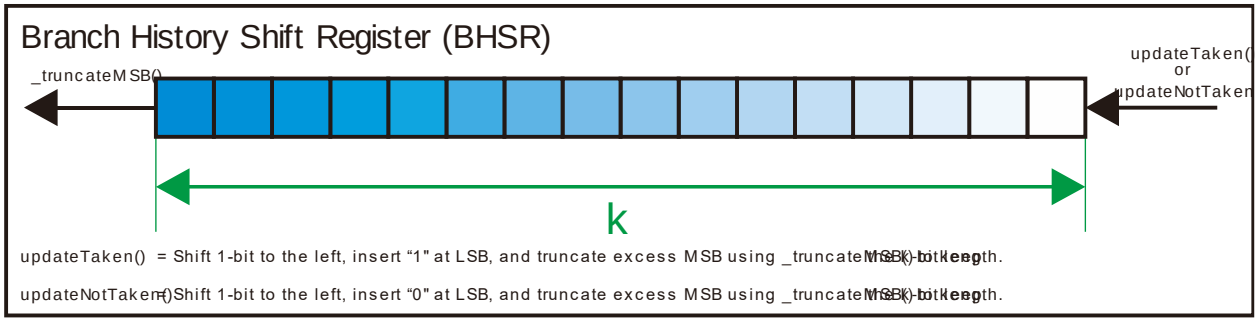
One-Level and Two-Level Branch Predictor Simulator

In this assignment, we will implement a program to simulate branch predictor that supports one-level and two-level branch predictor schemes. The program will be built on C++ and should be compatible with both UNIX and Windows platform. To begin the simulation, the program will be given a trace file that contains the branch address and the actual outcome of the branch (i.e., taken or not taken) and the configuration of the branch predictor. Then, the program will read the trace file entry one by one and give the statistics on how good the accuracy of the branch predictor is. Let's begin with two very basic elements of our branch predictor simulator.

A. Basic Elements

There are two basic elements of our branch predictor simulator. This basic elements store the history of the branches in the form of shift register (branch history shift register) or in the form of two-bit saturating counter (pattern history). The implementation of both basic elements are described below.

1. The Branch History Shift Register (BHSR)



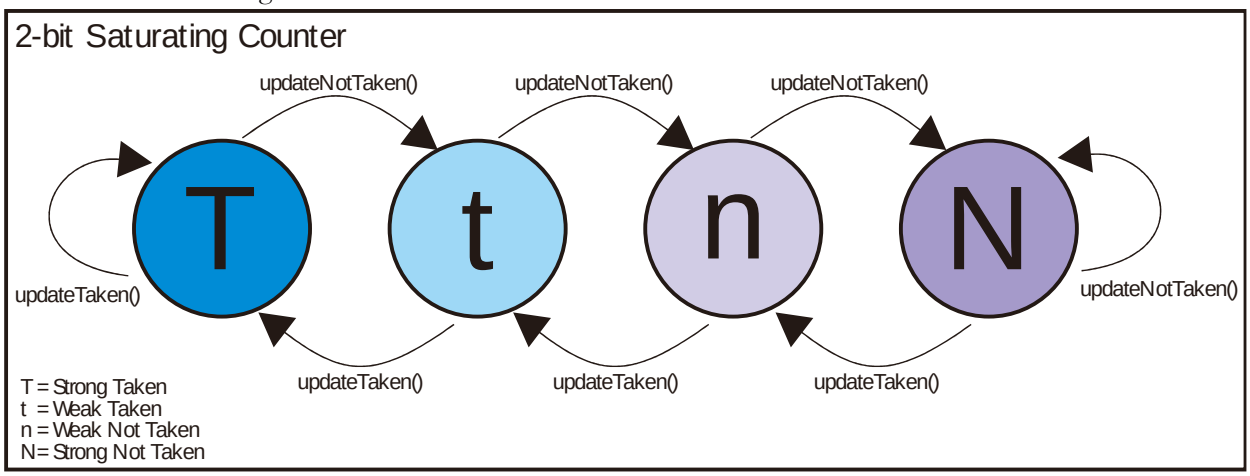
The BHSR is basically an k-bit shift register that stores the history of the outcome from the last k branches. If a new branch is encountered and the outcome is "taken", then BHSR is shifted to the left and bit "1" is inserted as the new LSB. Likewise, if the new branch's outcome is "not taken", the BHSR is shifted to the left and bit "0" is inserted as the new LSB. The excess MSB is automatically truncated, keeping the length of BHSR constant. The object BHSR is defined in the `class branch_history_shift_register` and the prototype of the class is listed in the file `branch_history_shift_register.hpp` as follows.

```
class branch_history_register
{
public:
    branch_history_register();
    branch_history_register(unsigned int length);
    branch_history_register(unsigned int initial, unsigned int length);
    branch_history_register(unsigned long long initial, unsigned int length);
    void updateTaken();
    void updateNotTaken();
    void reset();
    void reset(unsigned int length);
    void reset(unsigned int initial, unsigned int length);
    void reset(unsigned long long initial, unsigned int length);
```

```
    unsigned long long getLongValue();
    unsigned int getIntValue();
    unsigned int getLength();
    std::string getBinaryString();
private:
    unsigned long long _value;
    unsigned int _length;
    void _truncateMSB();
};
```

Note that there are four constructors that can be chosen from. It allows flexibility for the user to set-up the length of BHSR (i.e., the k-bit) as well as the initial value of the BHSR. If the user does not specify the length, it will be initialized as 1 (i.e., 1-bit BHSR). The default initial value of the BHSR is 0.

2. The Two-Bit Saturating Counter



The Two-Bit Saturating Counter is a two-bit finite state machine that indicates the outcome pattern of the branches. There are four states indexed using two-bit: strong taken (T), weak taken (t), weak not taken (n), and strong not taken (N). The state can be updated by calling appropriate method as shown on the state diagram above. Then, the outcome of the prediction will be based on the position of the state machine. It will predict taken if the state is T or t and it will predict not taken if the state is n or N. This object is defined in `class saturating_counter` and the prototype of the class is listed in the file `saturating_coutner.hpp` as follows.

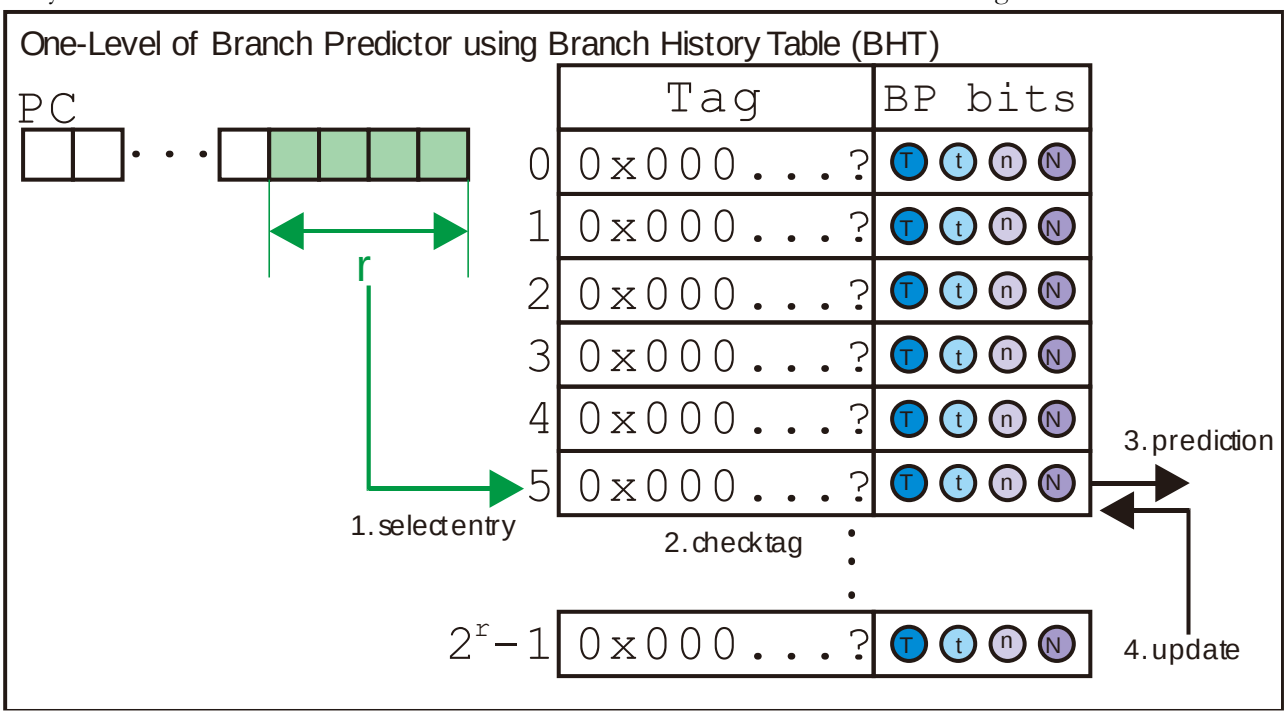
```
enum counter_status
{
        strong_taken,
        weak_taken,
        weak_nottaken,
        strong_nottaken,
};

class saturating_counter
{
public:
    saturating_counter();
    saturating_counter(counter_status initial);
    void updateTaken();
    void updateNotTaken();
    void reset();
    void reset(counter_status initial);
    bool isTaken();
    bool isNotTaken();
    counter_status getCounterStatus();
    std::string getCounterStatusString();
```

```
private:
    counter_status _status;
};
```

Using default constructor, the saturating counter is initialized using weak not taken state although the user can choose another state as initial state.

B. One-Level Branch Predictor with Branch History Table

The first branch predictor that our program can simulate is one-level branch predictor with branch history table. The diagram of this branch predictor is shown below. The Branch History Table (BHT) is similar to the Decode History Table (DHT) but with additional field to store the address of the branch instructions[1]. Therefore, each entry of the BHT consists of the address of a branch instruction and a two-bit saturating counter.



1. The Branch History Table Entry.
   By default, the two-bit saturating counters are initialized into weak not taken state. If there is an aliasing on an entry of BHT (i.e., the address of the branch stored in tag field is missmatch), then the saturating counter will be automatically reset and the tag will be replaced by new address of the branch. Therefore, it will predict not taken during entry replacement. Each of the BHT entry is defined in `class branch_history_table_entry` and the prototype of the class is listed on the file `branch_history_table_entry.hpp` as follows.

```
class branch_history_table_entry
{
public:
    branch_history_table_entry();
    branch_history_table_entry(counter_status initState);
    branch_history_table_entry(unsigned int initAddress);
    branch_history_table_entry(unsigned int initAddress, counter_status initialState);
```

[1] http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/

```
    void updateTaken();
    void updateNotTaken();

    bool isTaken(unsigned int address);
    bool isNotTaken(unsigned int address);
    bool isTakenAutoReplace(unsigned int address);
    bool isTakenAutoReplace(counter_status initState, unsigned int address);
    bool isNotTakenAutoReplace(unsigned int address);
    bool isNotTakenAutoReplace(counter_status initState, unsigned int address);

    void reset();
    void reset(counter_status initState);
    void reset(unsigned int initAddress);
    void reset(unsigned int initAddress, counter_status initialState);

    saturating_counter getCounter();
    unsigned int getCurrentAddress();
    std::string getEntryStringDecimal();
    std::string getEntryStringHex();
private:
    unsigned int       _address;
    saturating_counter _counter;
};
```

2. The Branch History Table.
   As stated earlier, the Branch History Table consists at least one Branch History Table Entry. The user can specify the number of BHT entries as an argument of the program as follows. Note that the argument given to the program is the $\log_2 \#BHT\ Entries$. The argument should be larger than or equal to 0 where the maximum number may be limited by the amount of available system memory.

```
./sim [tracefile] [r]
./sim [tracefile] [log2(number of BHT entry)]
e.g. ./sim trace.in 4         Initialize BHT with 16 entries.
     ./sim trace.in 10        Initialize BHT with 1024 entries.
```

Whole branch history table is defined in `class branch_history_table` and the prototype of the class is listed on the file `branch_history_table.hpp` as follows.

```
class branch_history_table
{
public:
    branch_history_table(unsigned int numofEntryinLogTwo);
    branch_history_table(unsigned int numofEntryinLogTwo, counter_status initState);
    branch_history_table(unsigned int numofEntryinLogTwo, unsigned int initAddress);
    branch_history_table(unsigned int numofEntryinLogTwo, unsigned int initAddress,
counter_status initState);

    void updateTaken(unsigned int address);
    void updateNotTaken(unsigned int address);

    bool isTaken(unsigned int address);
    bool isNotTaken(unsigned int address);
    bool isTakenAutoReplace(unsigned int address);
    bool isTakenAutoReplace(counter_status initState, unsigned int address);
    bool isNotTakenAutoReplace(unsigned int address);
    bool isNotTakenAutoReplace(counter_status initState, unsigned int address);

    void reset(unsigned int numofEntryinLogTwo);
    void reset(unsigned int numofEntryinLogTwo, counter_status initState);
    void reset(unsigned int numofEntryinLogTwo, unsigned int initAddress);
    void reset(unsigned int numofEntryinLogTwo, unsigned int initAddress, counter_status
initState);

    branch_history_table_entry getEntrybyAddress(unsigned int address);
    branch_history_table_entry getEntrybyIndex(unsigned int index);
```

```
    unsigned int getNumofEntry();
    unsigned int getnumofEntryinLogTwo();
    void printTableDecimal();
    void printTableHex();

private:
    unsigned int                _numofEntryinLogTwo;
    unsigned int                _numofEntry;
    branch_history_table_entry* _bht;
    unsigned int calculateIndex(unsigned int address);
};
```

3. Simulation Result

   This is the output of the simulator if the simulation finishes successfully.

```
Hanin Two-Level Branch Predictor Simulator v0.1 (2019 April 01)
(C) 2019 Bagus Hanindhito (hanindhito@bagus.my.id)
=====================================================
One Level Branch Predictor Simulation
Model                                 : BHT
Number of Entry                       : 1024
Hardware Cost (excluding tag fields)  : 2048
=====================================================
Simulation Statistics
Number of Predicted Branch            : 15556
Number of Correct Prediction          : 13708
Number of Wrong Prediction            : 1848
Branch Prediction Accuracy            : 88.1203%
=====================================================
```

Here is the simulation result for various BHT size using the provided `trace.in` file.

# BHT Entries	Hardware Cost	Total Prediction	Correct Prediction	Wrong Prediction	Accuracy
1	2	15,556	9,746	5,810	62.65%
2	4	15,556	9,849	5,707	63.31%
32	64	15,556	12,273	3,283	78.90%
64	128	15,556	12,854	2,702	82.63%
128	256	15,556	13,280	2,276	85.37%
1,024	2,048	15,556	13,708	1,848	88.12%
2,048	4,096	15,556	13,713	1,843	88.15%
4,096	8,192	15,556	13,712	1,844	88.15%
8,192	16,384	15,556	13,714	1,842	88.16%

C. Two-Level Branch Predictor with Branch History Shift Register (BHSR) Table and Pattern History Table (PHT)

The second branch predictor, and probably the most difficult one, that our program can simulate is two-level branch predictor with Branch Shift Register Table and Pattern History Table. Since the type of this branch predictor is highly reconfigurable, there are nine possible configurations of it which makes the implementation of the simulator a little bit challenging.

1. Possible Configuration

   The two-level branch predictor is highly configurable using four arguments provided during launching the executable of the simulator.

```
./sim [tracefile] [i] [j] [k] [s]
./sim [tracefile] [log2(# BHSR)] [log2(# PHT)] [BHSR Length] [log2(# Set)]
```

```
e.g. ./sim trace.in 0 0 8 0    Initialize GAg with 8-bit BHSR length.
     ./sim trace.in 4 0 16 0   Initialize PAg with 16-bit BHSR length.
```

Here is the possible configuration of the two-level branch predictor.

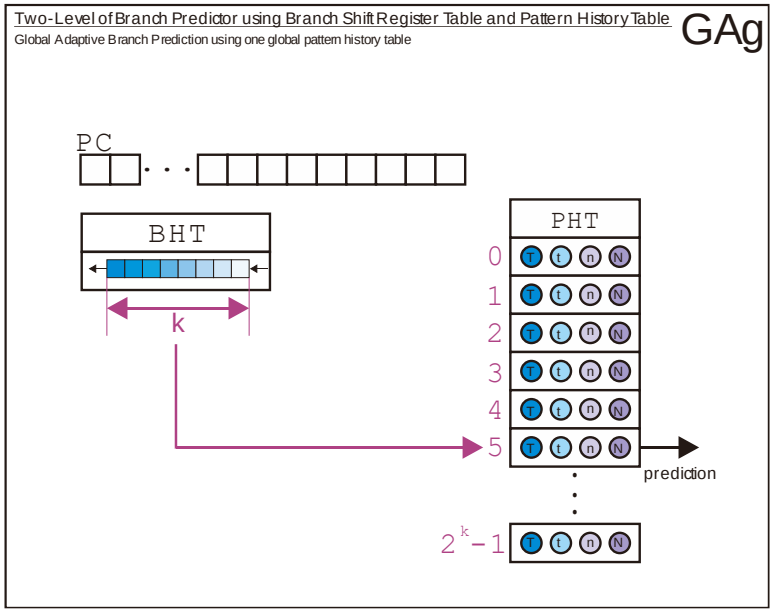
First Level (Branch History)	Second Level (Pattern History)	Name
Kept Globally	Kept Globally	GAg
Kept Globally	Kept per Set	GAs
Kept Globally	Kept per Address	GAp
Kept per Set	Kept Globally	SAg
Kept per Set	Kept per Set	SAs
Kept per Set	Kept per Address	SAp
Kept per Address	Kept Globally	PAg
Kept per Address	Kept per Set	PAs
Kept per Address	Kept per Address	PAp

a. GAx Type

This type of branch predictor is indicated by owning only one BHSR on the first level predictor. There are three subtypes determined by the configuration of its second level branch predictor as follows.

- GAg Subtype

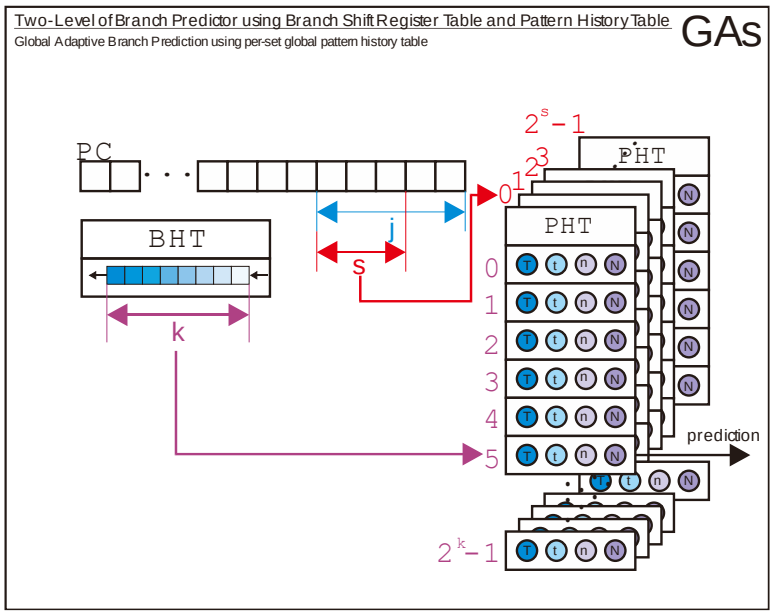
  This subtype is the simplest of all configuration, having only one k-bit BHSR on the first level and one PHT on the second level. The number of entries in the PHT is determined by the length of BHSR.



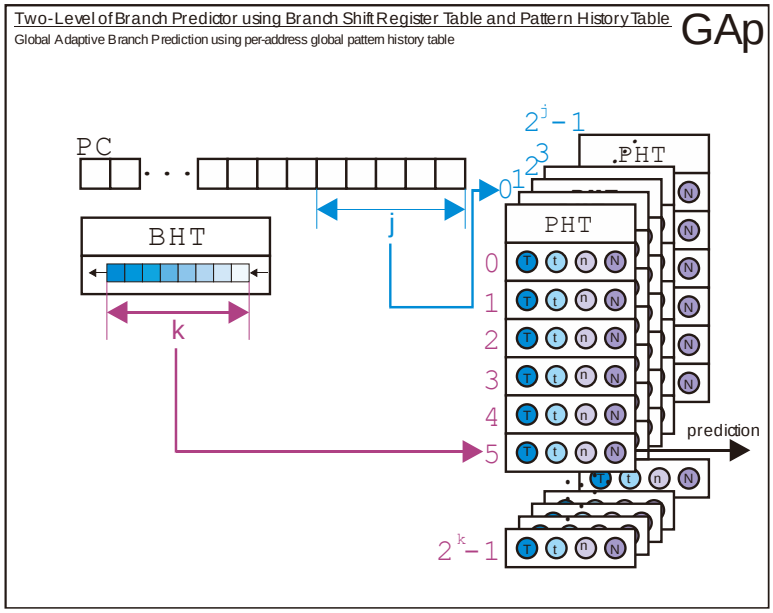
- GAs Subtype

  This subtype perhaps the most unique configuration. Since the first level consists only of one BHSR, we cannot directly derive the number of set for the second level. Instead, we use the s-bit directly to determine the number of set for the second level. The accuracy should be between GAg and GAp since the number of PHT will be less than those in GAp.

+1-737-346-5110 | hanindhito@bagus.my.id | https://www.bagus.my.id

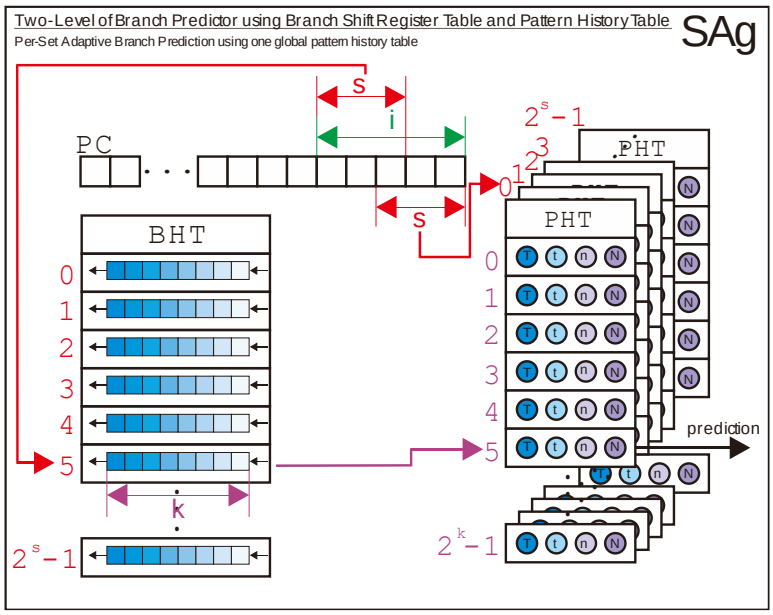


- GAp Subtype
  This subtype having more than one PHT on its second level. The number of PHTs should be more than the GAs thus we can expect better accuracy on it.

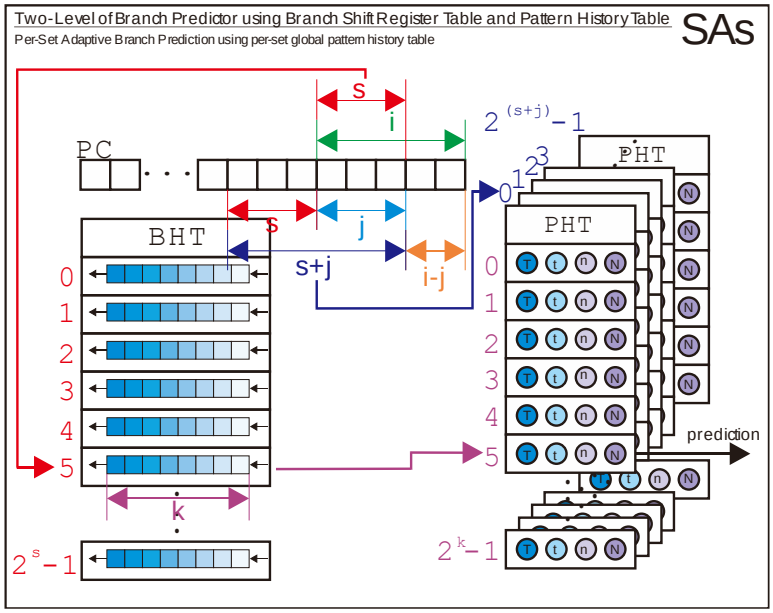


b. SAx Type
This type of branch predictor is indicated by having more than one sets of BHSR on the first level predictor followed by multiple sets of PHT on the second level predictor. This type has the most expensive second level implementation compared to other XAg, XAs, and XAp. The number of BHSR should be less than PAx type thus we can expect middle accuracy for first level predictor compared to GAx and PAx. There are three subtypes determined by the configuration of its second level branch predictor as follows.

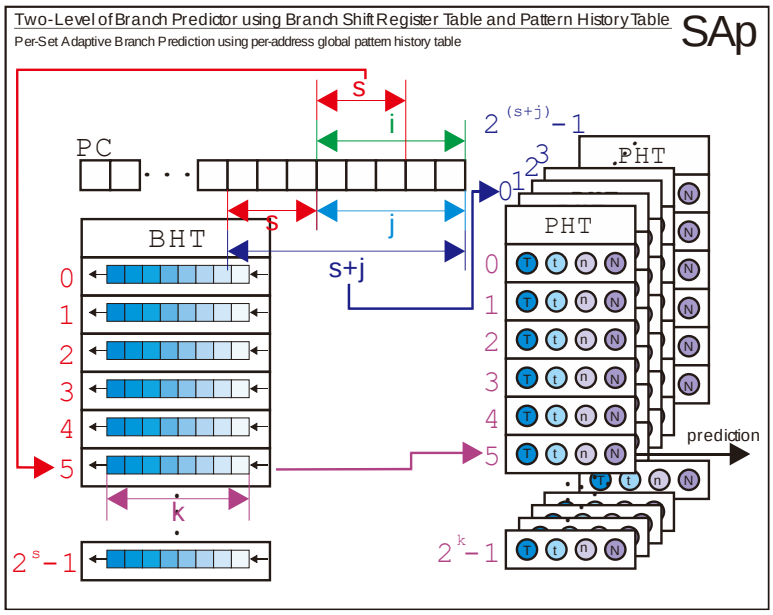
- SAg Subtype
  Because it has multiple sets on its first level, the total number of PHTs will be the same as the number of sets on its first level. It has global PHT for each sets thus every sets is mapped into one PHT.



- SAs Subtype
  The same as SAg, this subtype will have multiple sets of BHSRs. Each sets will be directly mapped into a set of PHTs on the second level.



- SAp Subtype
  The same as SAg, this subtype will have multiple sets of BHSRs. Each sets will be directly mapped into some PHTs on the second level.

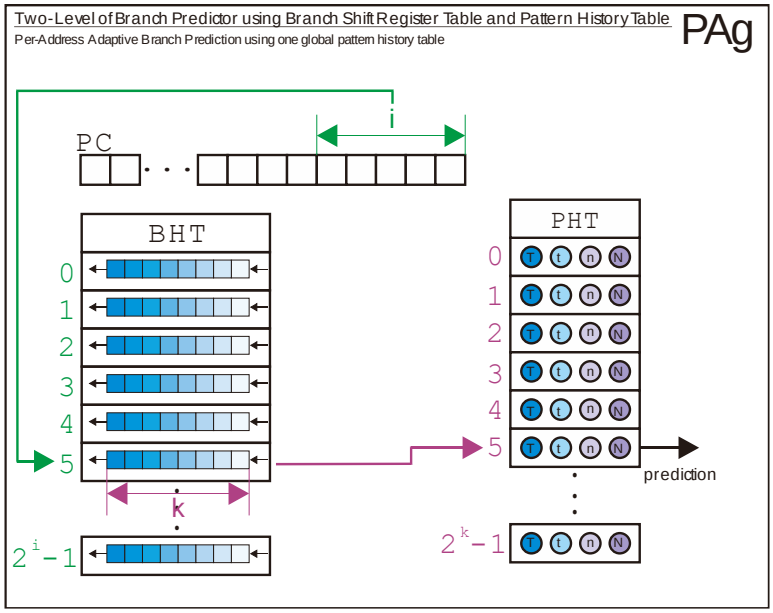


c. PAx Type

This type of branch predictor is indicated by having more than one BHSR on the first level predictor.

- PAg Subtype

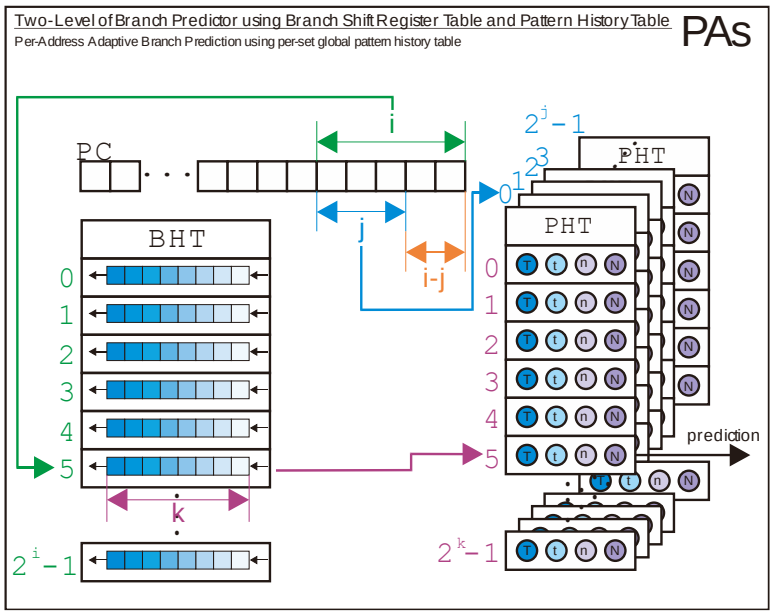
  This subtype has multiple number of BHSR in its first level. All of them are pointing to the same PHT on the second level.



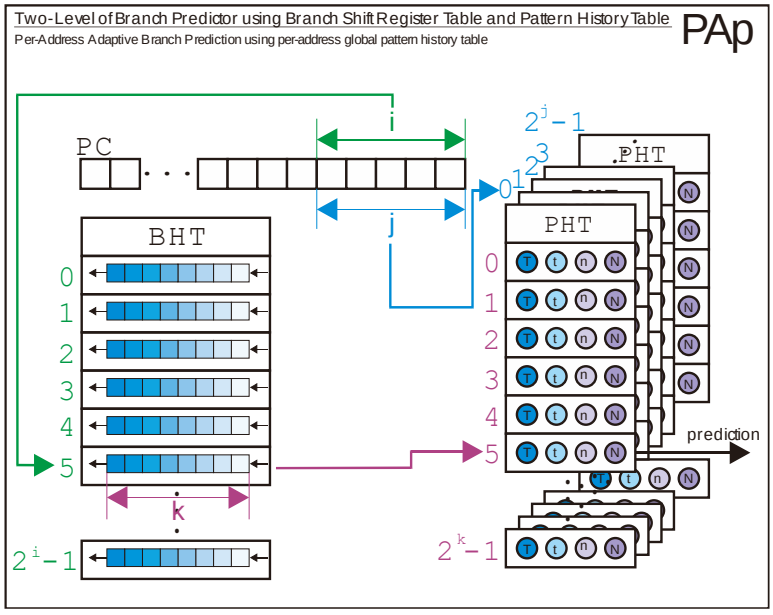
- PAs Subtype

  This subtype has multiple number of BHSR in its first level as well as some sets of PHT on the second level.

+1-737-346-5110 | hanindhito@bagus.my.id | https://www.bagus.my.id



- PAp Subtype
  This subtype has multiple number of BHSR in its first level as well as multiple number of PHT in its second level.



Finally, we define the requirement for arguments `i`, `j`, `k`, and `s` for constructing each possible configuration as follows.

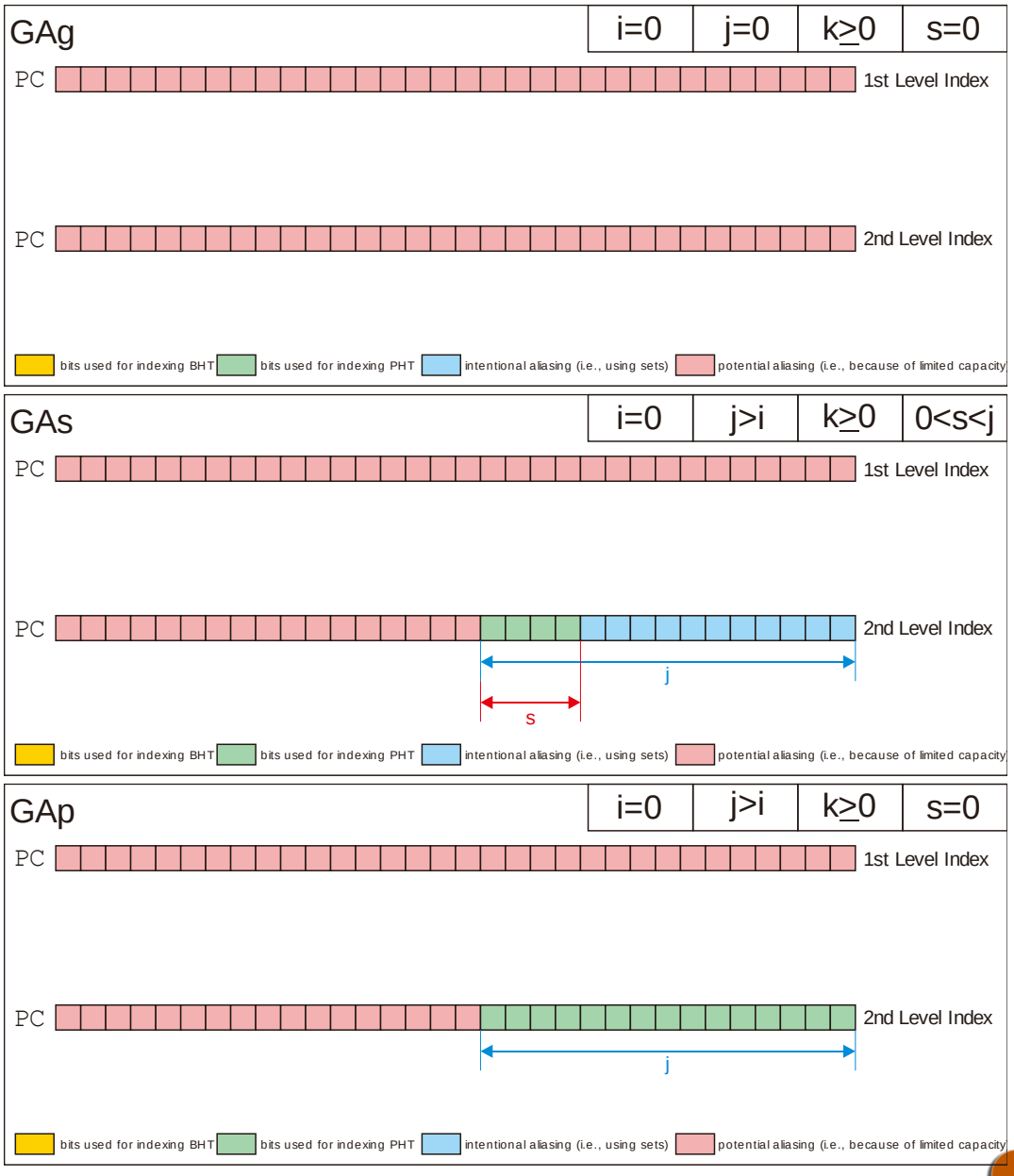
Name	i	j	k	s
GAg	$i = 0$	$j = 0$	$k \geq 0$	$s = 0$
GAs	$i = 0$	$j > i$	$k \geq 0$	$0 < s < j$
GAp	$i = 0$	$j > i$	$k \geq 0$	$s = 0$

+1-737-346-5110 | hanindhito@bagus.my.id | https://www.bagus.my.id

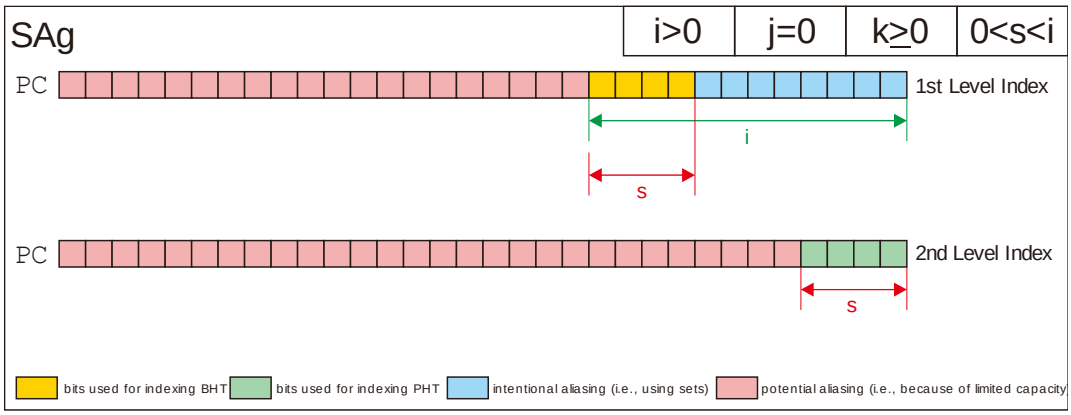
SAg	$i > 0$	$j = 0$	$k \geq 0$	$0 < s < i$
SAs	$i > 0$	$0 < j < i$	$k \geq 0$	$0 < s < i$
SAp	$i > 0$	$j \geq i$	$k \geq 0$	$0 < s < i$
PAg	$i > 0$	$j = 0$	$k \geq 0$	$s = 0$
PAs	$i > 0$	$0 < j < i$	$k \geq 0$	$s = 0$
PAp	$i > 0$	$j \geq i$	$k \geq 0$	$s = 0$

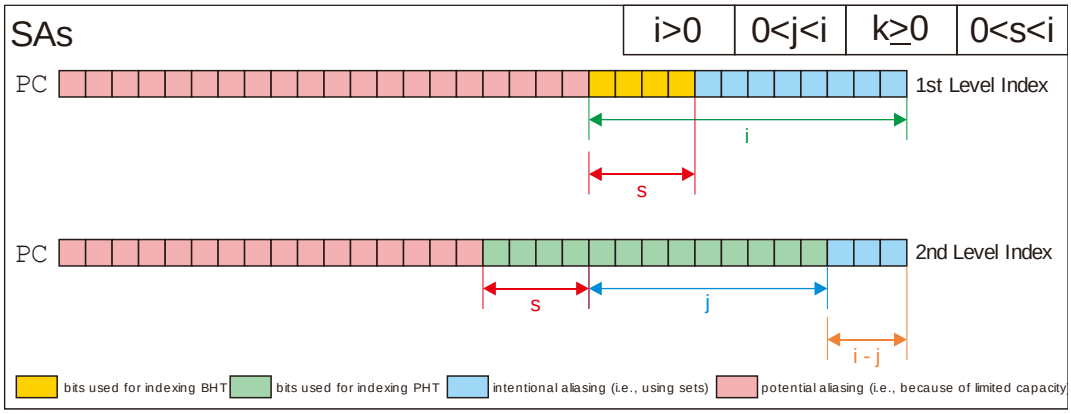
2. Indexing Scheme

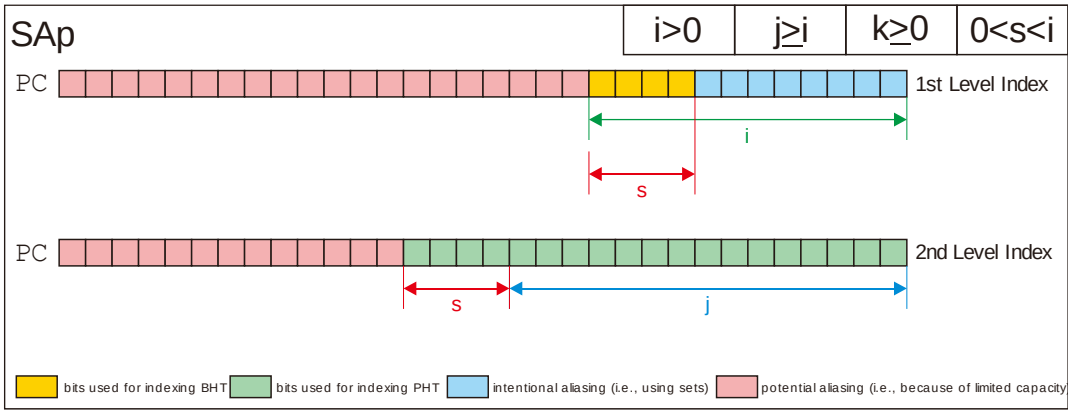
We use the address of branch instruction to index both Branch History Shift Register Table and Pattern History Table. Because of the differences on how each level of the branch predictor is accessed, the use of address of the branch instruction bit for indexing may also be different. Here is the diagram on how we use the address bit of branch instruction to index both level of branch predictor.

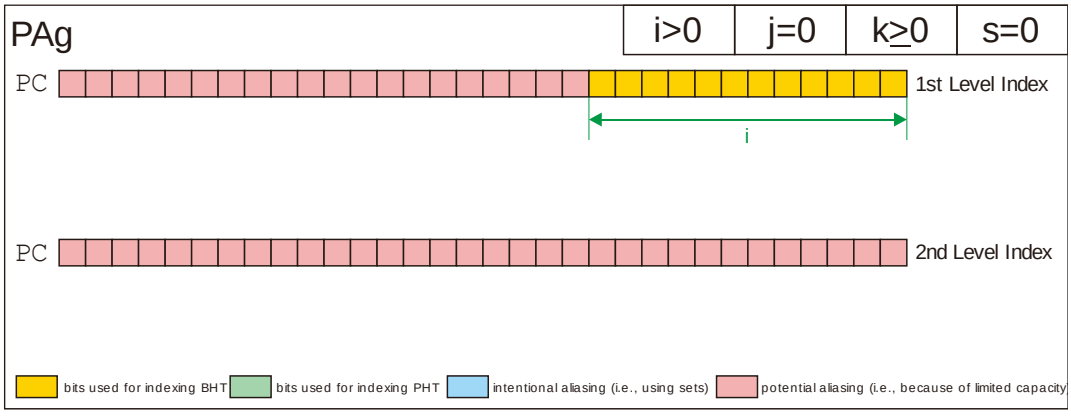


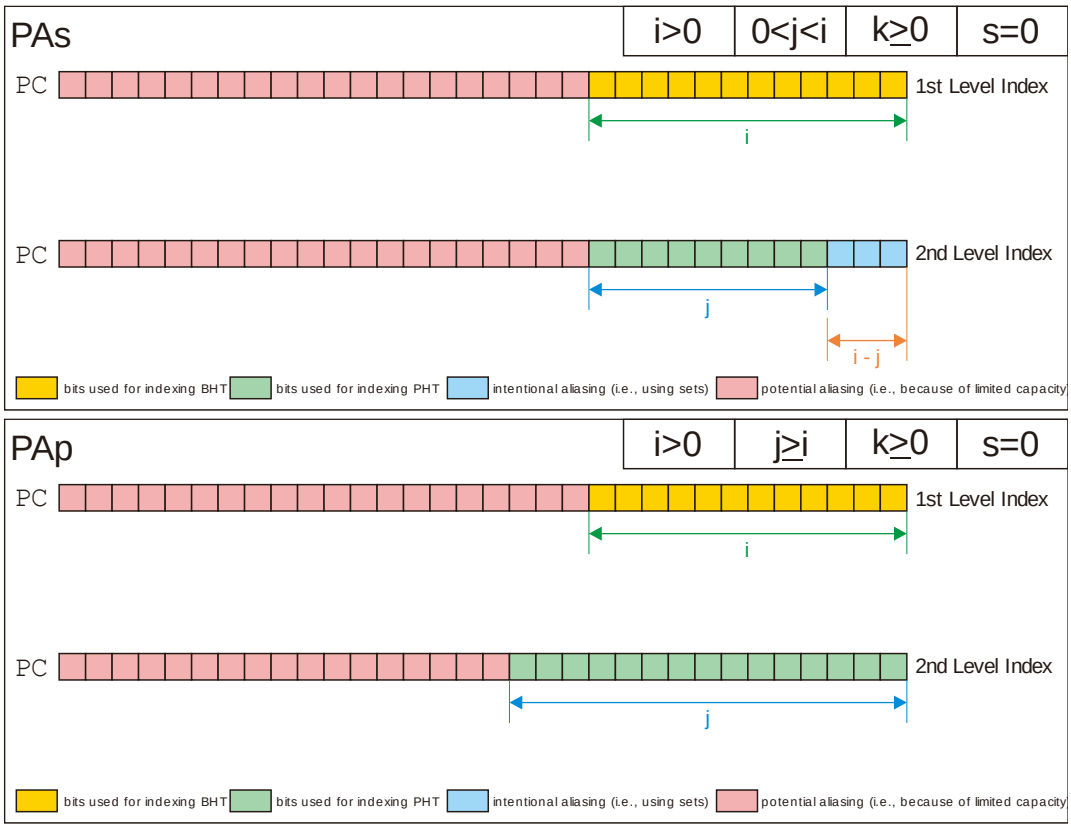
+1-737-346-5110 | hanindhito@bagus.my.id | https://www.bagus.my.id


SAg
i>0
j=0
k≥0
0<s<i
PC
1st Level Index
i
s
PC
2nd Level Index
s
bits used for indexing BHT
bits used for indexing PHT
intentional aliasing (i.e., using sets)
potential aliasing (i.e., because of limited capacity


SAs
i>0
0<j<i
k≥0
0<s<i
PC
1st Level Index
i
s
PC
2nd Level Index
s
j
i - j
bits used for indexing BHT
bits used for indexing PHT
intentional aliasing (i.e., using sets)
potential aliasing (i.e., because of limited capacity


SAp
i>0
j≥i
k≥0
0<s<i
PC
1st Level Index
i
s
PC
2nd Level Index
s
j
bits used for indexing BHT
bits used for indexing PHT
intentional aliasing (i.e., using sets)
potential aliasing (i.e., because of limited capacity


PAg
i>0
j=0
k≥0
s=0
PC
1st Level Index
i
PC
2nd Level Index
bits used for indexing BHT
bits used for indexing PHT
intentional aliasing (i.e., using sets)
potential aliasing (i.e., because of limited capacity



3. Hardware Cost Calculation

   Based on the configuration of each possible branch predictor, the table below lists the hardware cost to implement each possible configurations in terms of bits.

Name	First Level Cost	Second Level Cost	Total Cost
GAg	k	$2^k \cdot 2$	$k + 2^k \cdot 2$
GAs	k	$2^s \cdot 2^k \cdot 2$	$k + 2^s \cdot 2^k \cdot 2$
GAp	k	$2^j \cdot 2^k \cdot 2$	$k + 2^j \cdot 2^k \cdot 2$
SAg	$2^s \cdot k$	$2^s \cdot 2^k \cdot 2$	$2^s \cdot k + 2^s \cdot 2^k \cdot 2$
SAs	$2^s \cdot k$	$2^s \cdot 2^j \cdot 2^k \cdot 2$	$2^s \cdot k + 2^s \cdot 2^j \cdot 2^k \cdot 2$
SAp	$2^s \cdot k$	$2^s \cdot 2^j \cdot 2^k \cdot 2$	$2^s \cdot k + 2^s \cdot 2^j \cdot 2^k \cdot 2$
PAg	$2^i \cdot k$	$2^k \cdot 2$	$2^i \cdot k + 2^k \cdot 2$
PAs	$2^i \cdot k$	$2^j \cdot 2^k \cdot 2$	$2^i \cdot k + 2^j \cdot 2^k \cdot 2$
PAp	$2^i \cdot k$	$2^j \cdot 2^k \cdot 2$	$2^i \cdot k + 2^j \cdot 2^k \cdot 2$

4. The Branch History Register Table

   The first level of branch predictor consists of branch history register table whose number of branch history shift registers can be specified by the user. Whole branch history shift register table is defined in `class branch_history_register_table` and the prototype of the class is listed on the file `branch_history_register_table.hpp` as follows.

```
class branch_history_register_table
{
public:
    branch_history_register_table(unsigned int numofEntryinLogTwo);
    branch_history_register_table(unsigned int numofEntryinLogTwo, unsigned int
registerLength);
    branch_history_register_table(unsigned int numofEntryinLogTwo, unsigned int
registerInitial, unsigned int registerLength);
    branch_history_register_table(unsigned int numofEntryinLogTwo, unsigned long long
registerInitial, unsigned int registerLength);

    void updateTaken(unsigned int tableIndex);
    void updateNotTaken(unsigned int tableIndex);

    void reset(unsigned int numofEntryinLogTwo);
    void reset(unsigned int numofEntryinLogTwo, unsigned int registerLength);
    void reset(unsigned int numofEntryinLogTwo, unsigned int registerInitial, unsigned int
registerLength);
    void reset(unsigned int numofEntryinLogTwo, unsigned long long registerInitial, unsigned
int registerLength);

    branch_history_register getEntrybyIndex(unsigned int tableIndex);
    unsigned int getNumofEntry();
    unsigned int getnumofEntryinLogTwo();
    void printTableBinary();
    void printTableDecimal();
    void printTableHex();

private:
    unsigned int               _numofEntryinLogTwo;
    unsigned int               _numofEntry;
    branch_history_register* _bhsrt;
};
```

5. The Pattern History Register Table

   The second level of branch predictor consists of pattern history table which contains some two-bit saturating counters. The number of saturating counters follows the length of branch history shift registers on the first-level. Whole pattern history table is defined in `class pattern_history_table` and the prototype of the class is listed on the file `pattern_history_table.hpp` as follows.

```
class pattern_history_table
{
public:
    pattern_history_table();
    pattern_history_table(unsigned int numofEntryinLogTwo);
    pattern_history_table(unsigned int numofEntryinLogTwo, counter_status counterInitial);

    void updateTaken(unsigned int tableIndex);
    void updateNotTaken(unsigned int tableIndex);

    void reset(unsigned int numofEntryinLogTwo);
    void reset(unsigned int numofEntryinLogTwo, counter_status counterInitial);

    saturating_counter getEntrybyIndex(unsigned int tableIndex);
    unsigned int getNumofEntry();
    unsigned int getnumofEntryinLogTwo();
    void printTable();

private:
    unsigned int               _numofEntryinLogTwo;
    unsigned int               _numofEntry;
    saturating_counter*        _pht;
};
```

6. The Two-Level Branch Predictor

   Finally, we have the main two-level branch predictor that is highly configurable. The class itself has comprehensive parameter check before the two-level branch predictor can be constructed. Moreover, it also

adjusts the addressing mode based on the configuration. This class manages one branch history shift register table with adjustable entries and length and a configurable number of pattern history table whose number of saturating counters can be adjusted. Here is the prototype of `class two_level_predictor` as listed in header file `two_level_predictor.hpp`.

```
enum two_level_predictor_type
{
        GAg,
        GAs,
        GAp,
        SAg,
        SAs,
        SAp,
        PAg,
        PAs,
        PAp,
        NAn,
};

class two_level_predictor
{
public:
    static bool isParameterValid(int i, int j, int k, int s);
    static std::string getParameterErrorInfo(int i, int j, int k, int s);
    static std::string getPredictorTypeString(int i, int j, int k, int s);
    static two_level_predictor_type getPredictorType(unsigned int i, unsigned int j, unsigned
int k, unsigned int s);

    two_level_predictor(unsigned int i, unsigned int j, unsigned int k, unsigned int s);
    void reset(unsigned int i, unsigned int j, unsigned int k, unsigned int s);
    void updateTaken(unsigned int address);
    void updateNotTaken(unsigned int address);
    bool isPredictTaken(unsigned int address);
    bool isPredictNotTaken(unsigned int address);
    unsigned int getNumofBHSREntryinLogTwo();
    unsigned int getNumofPHTinLogTwo();
    unsigned int getLengthofBHSR();
    unsigned int getNumofSetinLogTwo();
    unsigned int getNumofPHTEntryinLogTwo();

    unsigned int getNumofBHSREntry();
    unsigned int getNumofPHT();
    unsigned int getNumofPHTEntry();
    unsigned int getNumofSet();

    two_level_predictor_type       getPredictorType();
    pattern_history_table          getPatternHistoryTable(unsigned int phtindex);
    branch_history_register_table  getBranchHistoryRegisterTable();

    unsigned int calculateHardwareCostinBit();
    void printBranchPredictorContents();


private:
    unsigned int _numofBHSREntryinLogTwo; //i
    unsigned int _numofPHTinLogTwo; //j
    unsigned int _lengthofBHSR; //k
    unsigned int _numofSetinLogTwo; //s

    unsigned int _numofBHSREntry;
    unsigned int _numofPHT;
    unsigned int _numofPHTEntry;
    unsigned int _numofSet;

    unsigned int _i;
    unsigned int _j;
    unsigned int _k;
    unsigned int _s;

    two_level_predictor_type _predictorType;
    unsigned int  calculateBHRTIndex(unsigned int address);
```

```
    unsigned int _calculatePHTSIndex(unsigned int address);

    pattern_history_table*        _phts;
    branch_history_register_table _bhrt;
};
```

7. Simulation Result

This is the output of the simulator if the simulation finishes successfully. The trace input is the provided mini trace `test_trace.in`.

```
Hanin Two-Level Branch Predictor Simulator v0.1 (2019 April 01)
(C) 2019 Bagus Hanindhito (hanindhito@bagus.my.id)
=====================================================
Two Level Branch Predictor Simulation
Model                                   : GAg
Number of Branch History Shift Registers : 1
Length of Branch History Shift Registers : 3
Number of Pattern History Tables         : 1
Number of PHT Entries                    : 8
Hardware Cost                            : 19
=====================================================
Simulation Statistics
Number of Predicted Branch               : 16
Number of Correct Prediction             : 9
Number of Wrong Prediction               : 7
Branch Prediction Accuracy               : 56.25%
=====================================================
```

Here is the simulation result for various configuration using the provided `trace.in` file.

Model	# BSHR	# PHT	# PHT Entry	Hardware Cost	Total Prediction	Correct Prediction	Wrong Prediction	Accuracy
GAg	1	1	8,192	16,397	15,556	13,284	2,272	85.39%
GAg	1	1	262,144	524,306	15,556	13,093	2,463	84.17%
GAp	1	4	65,536	524,304	15,556	13,212	2,344	84.93%
GAp	1	8	32,768	524,303	15,556	13,290	2,266	85.43%
GAp	1	16	16,384	524,302	15,556	13,346	2,210	85.79%
GAp	1	32	8192	524,301	15,556	13,395	2,161	86.11%
GAp	1	131,072	2	524,289	15,556	13,866	1,690	89.17%
GAs	1	4	65,536	524,304	15,556	13,211	2,345	84.93%
PAg	32,768	1	16,384	491,520	15,556	13,409	2,147	86,20%
PAg	65,536	1	256	524,800	15,556	13,537	2,019	87.02%
PAg	131,072	1	16	524,320	15,556	13,424	2,132	86,29%
PAp	1,024	1,024	256	532,480	15,556	13,082	2,474	84.10%
PAp	16	8,192	32	524,368	15,556	13,391	2,165	86.08%
PAp	2	131,072	2	524,290	15,556	13,721	1,835	88.20%
PAs	131,072	32,768	4	524,288	15,556	13,476	2,080	86,63%
PAs	262,144	65,536	2	524,288	15,556	13,593	1,963	87.38%
SAg	1,024	1,024	256	532,480	15,556	13,282	2,274	85,38%
SAp	32	8,192	32	524,448	15,556	13,324	2,232	85.65%
SAs	32	2,048	128	524,512	15,556	13,075	2,481	84.05%

D. Source Code

Source code are available on Github https://github.com/hibagus/TwoLevelBPSimulator