```
#include "combinePredictor.h"

combinePredictor::combinePredictor()
{
}

//
combinePredictor::~combinePredictor()
{
}

bool combinePredictor::create()
{
  // create the children class first.
  // make sure both predictors are initialized
  saPredictor.SetSize(byteSize);
  saPredictor.SetBitWidth(bitWidth);
  saPredictor.create();

  glPredictor.SetSize(byteSize);
  glPredictor.SetBitWidth(bitWidth);
  glPredictor.globalHTindex = 0;
  glPredictor.create();

  gsPredictor.SetSize(byteSize);
  gsPredictor.SetBitWidth(bitWidth);
  gsPredictor.globalHTindex = 0;
  gsPredictor.create();

  entryNum = byteSize*8/bitWidth;
  predictionTable = new int[entryNum];
  for(int i=0; i< entryNum ; i++)
   {
     predictionTable[i]=0;
   }

  return 0;
}

bool combinePredictor::stepSimulation(char *address, int branchTaken)
{
  long long int PCAddress = ctoll(address); //type convertion
  iTotalCount++;

  PCAddress = PCAddress >> 2;

  // this mask make sure we got the right bits in PCAdress
  int bitNum = int(log2(entryNum));
  int index = PCAddress & (((0x1<<bitNum)-1));

  glPredictor.globalHTindex = (glPredictor.globalHTindex) & (((0x1<<bitNum)-1));
  gsPredictor.globalHTindex = (gsPredictor.globalHTindex) & (((0x1<<bitNum)-1));


  int predictionFromSa;
   if(saPredictor.predictionTable[index]%4 <2)
     predictionFromSa = 0;
  else
    predictionFromSa = 1;


  int predictionFromGl;
  if(glPredictor.predictionTable[glPredictor.globalHTindex]%4 < 2)
    predictionFromGl = 0;
  else
    predictionFromGl = 1;

  int predictionFromGs;
  if(gsPredictor.predictionTable[gsPredictor.globalHTindex]%4 < 2)
    predictionFromGs = 0;
  else
    predictionFromGs = 1;

  // update the prediction table
  saPredictor.predictionProcess(index, branchTaken);
  glPredictor.predictionProcess(glPredictor.globalHTindex, branchTaken);
  gsPredictor.predictionProcess(gsPredictor.globalHTindex, branchTaken);

  glPredictor.globalHTindex = ((glPredictor.globalHTindex<<1) | branchTaken);
  gsPredictor.globalHTindex = ((gsPredictor.globalHTindex<<1) | branchTaken);

  // we only train the selector when two predictors are different;
  // we do nothing if they are same (both correct and  wrong).
  if((predictionFromGs == predictionFromSa)&&(predictionFromGs == predictionFromGl)
)
  {
    if(predictionFromGs != branchTaken){
      predictFailCount++;
    }
  }else{

    switch(combineMode){
    case 0:
      if(!predictionProcess(index, branchTaken,predictionFromSa,predictionFromGl))
    predictFailCount++;
      break;
    case 1:
      if(!predictionProcess(index, branchTaken,predictionFromGl,predictionFromSa))
    predictFailCount++;
      break;
    case 2:
      if(!predictionProcess(index, branchTaken,predictionFromSa,predictionFromGs))
    predictFailCount++;
      break;
    case 3:
      if(!predictionProcess(index, branchTaken,predictionFromGs,predictionFromSa))
    predictFailCount++;
      break;
    case 4:
      if(!predictionProcess(index, branchTaken,predictionFromGl,predictionFromGs))
    predictFailCount++;
      break;
    case 5:
      if(!predictionProcess(index, branchTaken,predictionFromGs,predictionFromGl))
    predictFailCount++;
      break;
    }
  }
  return 0;
}

// We prefer to firstPredictor here. and right now,
// saPredictor is the firstPredictor, glPredictor is secondPredictor
// 00 for strong secondPredictor 01 for weak secondPredictor
// 11 for strong firstPredictor 10 for weak firstPredicctor
bool combinePredictor::predictionProcess(int index,int branchTaken,
                    int firstPredictor, int SecondPredictor)
{
  bool correct = 1;
  int prediction = predictionTable[index]%4;
  //  printf("prediction = %d \n",prediction);
  // the prefer predictor is same with branchTaken.
  if(branchTaken == firstPredictor)
  {
    switch(prediction){
    case 0:
      predictionTable[index]++;
      correct = 0;// even the prefer predictor is right, however the selector choos
es a wrong predictor
      break;
    case 1:
      predictionTable[index]+=2;
```

```
      correct = 0;// even the prefer predictor is right, however the selector choos
es a wrong predictor
      break;
    case 2:
      predictionTable[index]++;
      break;
    case 3:
      break;
    }
  }else{
    switch(prediction){
    case 0:
      break;
    case 1:
      predictionTable[index]--;
      break;
    case 2:
      predictionTable[index]-=2;
      correct = 0; //  the prefer predictor is wrong, and  the selector chooses thi
s wrong predictor
      break;
    case 3:
      predictionTable[index]--;
      correct = 0;//  the prefer predictor is wrong, and  the selector chooses this
 wrong predictor
      break;
    }
  }
  return correct;
}
```

```
#ifndef combinePredictor_H_
#define combinePredictor_H_

#include "saturaPredictor.h"
#include "globalPredictor.h"
#include "gsharePredictor.h"

class combinePredictor : public saturaPredictor
{
 private:
  class saturaPredictor saPredictor;
  class globalPredictor glPredictor;
  class gsharePredictor gsPredictor;

  bool predictionProcess(int,int,int,int);

public:
  int combineMode; // two combine two.
  bool create();
  bool stepSimulation(char *, int);


  combinePredictor();
  virtual ~combinePredictor();
};

#endif /* combinePredictor_H_ */
```

```
#include "globalPredictor.h"

//construct function
globalPredictor::globalPredictor()
{
  globalHTindex = 0;
}

//destructor function
globalPredictor::~globalPredictor()
{
}

//  To simulating the branch Prediction by line.
//  for a global Predictor, we only need the current
//  branch state.
bool globalPredictor::stepSimulation(int branchTaken)
{

  iTotalCount++;

  // this mask make sure we got the right bits in PCAdress
  int bitNum = int(log2(entryNum));
  globalHTindex = (globalHTindex) & (((0x1<<bitNum)-1));

  if(!predictionProcess(globalHTindex, branchTaken))
    predictFailCount++;

  // update the globalHTindex with the new current branch state
  globalHTindex = ((globalHTindex<<1) | branchTaken);

  return 0;
}
```

```
#ifndef GLOBALPREDICTOR_H_
#define GLOBALPREDICTOR_H_

#include "saturaPredictor.h"

//derived class based on saturaPredictor
//add globalHTindex and redefine the function of stepSimulation

class globalPredictor : public saturaPredictor
{
 private:

public:
  int  globalHTindex;
  bool stepSimulation(int);

  globalPredictor();
  virtual ~globalPredictor();
};

#endif /* GLOBALPREDICTOR_H_ */
```

```
#include "gsharePredictor.h"

gsharePredictor::gsharePredictor()
{
  globalHTindex=0;
}

gsharePredictor::~gsharePredictor()
{
}

// the main function to simulate the prediction
bool gsharePredictor::stepSimulation(char *address, int branchTaken)
{
  iTotalCount++;

  long long int PCAddress = ctoll(address);
  PCAddress = PCAddress >> 2;

 // this mask make sure we got the right bits in PCAdress
  int bitNum = int(log2(entryNum));
  globalHTindex = (globalHTindex) & (((0x1<<bitNum)-1));

  // xor the pc address index and the global history table index
  // the basic of gshare predictor
  int index = (PCAddress & (((0x1<<bitNum)-1)))^globalHTindex;

  if(!predictionProcess(index, branchTaken))
    predictFailCount++;

  //update the global history table index by using the current branch state
  globalHTindex = ((globalHTindex<<1) | branchTaken) ;

  return 0;
}
```

```
#ifndef gsharePredictor_H_
#define gsharePredictor_H_

#include "saturaPredictor.h"

// this class is based on saturaPredictor
// add the globalHTindex and redefine the execute function
class gsharePredictor : public saturaPredictor
{
 private:

public:
  int globalHTindex;  // global history table

  // the main simulating function
  bool stepSimulation(char *, int);

  gsharePredictor();
  virtual ~gsharePredictor();
};

#endif /* gsharePredictor_H_ */
```

```
/*
    CS4431 Branch Preddiction

    -- by Liang Yan
    Computer Science

This file is the main execute file,
all simulation start from here.
 */

#include <stdio.h>
#include <iostream>
#include <string>
#include "saturaPredictor.h"
#include "globalPredictor.h"
#include "gsharePredictor.h"
#include "tournaPredictor.h"
using namespace std;

int main()
{
    int areaSize[3];
    areaSize[0]=2048; //2Kbytes
    areaSize[1]=4096; //4Kbytes
    areaSize[2]=8192; //8Kytes
    int bitWidth =2;

    FILE *file;
    file = fopen("result.txt","a+"); /* add test result file  or create a file if i⤢
t does not exist.*/
    for(int i=0;i<3;i++){

      class saturaPredictor saPredictor;
      saPredictor.SetSize(areaSize[i]);
      saPredictor.SetBitWidth(bitWidth);
      saPredictor.create();

      class globalPredictor glPredictor;
      glPredictor.SetSize(areaSize[i]);
      glPredictor.SetBitWidth(bitWidth);
      glPredictor.create();

      class gsharePredictor gsPredictor;
      gsPredictor.SetSize(areaSize[i]);
      gsPredictor.SetBitWidth(bitWidth);
      gsPredictor.create();

      class tournaPredictor toPredictor;
      // to share the area with three preiction tables, >>2 is ignoring some waste ⤢
here.
      int area = areaSize[i]>>2;
      toPredictor.SetSize(area);
      toPredictor.SetBitWidth(bitWidth);
      toPredictor.create();

      int maxlen = 23;  // to make sure get whole value in a line;

      FILE *fp;
      char filePath[5] = "test";
      char source[8];
      char target[8];
      char buf[23];
      int type=0; // branch = 1, jump =2, others = 3
      int branchState=0;
      if( !(fp = fopen( filePath, "rt" ) ) ){
        printf("could not open the file %s",filePath);
        return 1;
      }
      while (fgets( buf, maxlen, fp ) != NULL) {
    sscanf(buf,"%s %d %d %s",source,&type,&branchState,target);
    if(type ==1){
      // printf("%s %d %d %s \n",source,type,branchState,target);
      saPredictor.stepSimulation(source,branchState);
      glPredictor.stepSimulation(branchState);
      gsPredictor.stepSimulation(source,branchState);
      toPredictor.stepSimulation(source,branchState);
       }
      }
      fclose(fp);
      fprintf(file,"%s,%d,%lld,%lld,%f\n","saturaPredictor",areaSize[i],saPredictor⤢
.GetiTotalCount(),saPredictor.GetpredictFailCount(),saPredictor.GetpredictionHitRat⤢
e());
      fprintf(file,"%s,%d,%lld,%lld,%f\n","globalPredictor",areaSize[i],glPredictor⤢
.GetiTotalCount(),glPredictor.GetpredictFailCount(),glPredictor.GetpredictionHitRat⤢
e());
      fprintf(file,"%s,%d,%lld,%lld,%f\n","gsharePredictor",areaSize[i],gsPredictor⤢
.GetiTotalCount(),gsPredictor.GetpredictFailCount(),gsPredictor.GetpredictionHitRat⤢
e());
      fprintf(file,"%s,%d,%lld,%lld,%f\n","tournaPredictor",areaSize[i],toPredictor⤢
.GetiTotalCount(),toPredictor.GetpredictFailCount(),toPredictor.GetpredictionHitRat⤢
e());
    }
    fclose(file);

    return 0;
}
```

```
CC=g++
CFLAGS=-Wall -std=c++0x

main: main.cpp saturaPredictor.cpp globalPredictor.cpp gsharePredictor.cpp tournaPr
edictor.cpp combinePredictor.cpp
    ${CC} ${CFLAGS} -g -o main main.cpp saturaPredictor.cpp globalPredictor.cpp gsh
arePredictor.cpp tournaPredictor.cpp combinePredictor.cpp

.PHONY: clean stripped cppcheck

stripped:
    ${CC} ${CFLAGS} -static -o main main.cpp
    strip main

cppcheck:
    cppcheck --enable=all --inconclusive *.cpp

clean:
    rm -f main *.o
```

```
#include "saturaPredictor.h"

saturaPredictor::saturaPredictor()
{
  iTotalCount = 0;
  predictFailCount = 0;
  predictionHitRate = 1.0;
  bitWidth=2;
  byteSize=2048;
}

// destructor function
saturaPredictor::~saturaPredictor()
{
}

void saturaPredictor::SetSize(int size)
{
  byteSize = size;
}

int saturaPredictor::GetSize()
{
  return byteSize;
}

void saturaPredictor::SetBitWidth(int width)
{
  bitWidth = width;
}

long long int saturaPredictor::GetiTotalCount()
{
  return iTotalCount;
}

long long int saturaPredictor::GetpredictFailCount()
{
  return predictFailCount;
}

float saturaPredictor::GetpredictionHitRate()
{
  predictionHitRate = (iTotalCount - predictFailCount)/(iTotalCount*1.0);
  return predictionHitRate;
}

bool saturaPredictor::create()
{
  // entryNum is based on the size of prediction table size
  entryNum = byteSize*8/bitWidth;
  predictionTable = new int[entryNum];
  for(int i=0; i< entryNum ; i++)
   {
     predictionTable[i]=0;
   }
  return 0;
}

bool saturaPredictor::stepSimulation(char *address, int branchTaken)
{
  long long int PCAddress = ctoll(address);
  iTotalCount++;
  PCAddress = PCAddress >> 2;// we do not use the first two bits.

  // this mask make sure we got the right bits in PCAdress
  int bitNum = int(log2(entryNum));
  int index = PCAddress & (((0x1<<bitNum)-1));

  // I choose wrong prediction for saving time
  // sine most of the time prediction is right
  if(!predictionProcess(index, branchTaken))
    predictFailCount++;
  return 0;
}

// train the predictor
// return true if predict is right
// also change the value of prediction table
bool saturaPredictor::predictionProcess(int index,int branchTaken)
{
  bool correct = 1;
  int prediction = predictionTable[index]%4; //store two bits in an int, use the fi
rst 4bits.
  if(branchTaken == 1)//0,1 means prediction is wrong
  {
    switch(prediction){
    case 0:
      predictionTable[index]++;
      correct = 0;
      break;
    case 1:
      predictionTable[index]+=2;
      correct = 0;
      break;
    case 2:
      predictionTable[index]++;
      break;
    case 3:
      break;
    }
  }else{// 2,3 means prediction is wrong
    switch(prediction){
    case 0:
      break;
    case 1:
      predictionTable[index]--;
      break;
    case 2:
      predictionTable[index]-=2;
      correct = 0;
      break;
    case 3:
      predictionTable[index]--;
      correct = 0;
      break;
    }
  }
  return correct;
}

// to convert a char[] to a long long int
// to get the integer of instructure address
long long int saturaPredictor::ctoll(char s[])
{
    long long int n = 0;
    for (int i=0; (s[i] >= '0' && s[i] <= '9')
        || (s[i] >= 'a' && s[i] <= 'z')
    || (s[i] >= 'A' && s[i] <= 'Z'); ++i)
    {

        if (tolower(s[i]) > '9')
        {
            n = 16 * n + (10 + tolower(s[i]) - 'a');
        }
        else
        {
            n = 16 * n + (tolower(s[i]) - '0');
        }
    }
    return n;
}
```

```
#ifndef SATURAPREDICTOR_H_
#define SATURAPREDICTOR_H_

#include <stdio.h>
#include <string>
#include <math.h>

class saturaPredictor
{
 protected:
  int  bitWidth; // the bit width, here is 2 bit.
  int  byteSize; // the size of Precditor, here is 2K 4K and 8K.
  long long int iTotalCount;  // the total number of branch instructers
  long long int predictFailCount;  // the total number of wrong prediction of branc
h
  float predictionHitRate;  // the right prediction rate of this predictor

  long long int ctoll(char s[]);  // type change

public:
  int entryNum;

  int *predictionTable;

  int GetSize();
  void SetSize(int size);

  int GetBitWidth();
  void SetBitWidth(int width);

  long long int GetiTotalCount();
  long long int GetpredictFailCount();
  float GetpredictionHitRate();

  bool predictionProcess(int,int);

  // to create the predictor talbe and initilize it.
  virtual bool create();

  // the main simulation function
  virtual bool stepSimulation(char *, int);

  saturaPredictor();
  virtual ~saturaPredictor();
};

#endif /* SATURAPREDICTOR_H_ */
```

```
#include "tournaPredictor.h"

tournaPredictor::tournaPredictor()
{
}

tournaPredictor::~tournaPredictor()
{
}

bool tournaPredictor::create()
{
  // create the children class first.
  // make sure both predictors are initialized
  saPredictor.SetSize(byteSize);
  saPredictor.SetBitWidth(bitWidth);
  saPredictor.create();

  glPredictor.SetSize(byteSize);
  glPredictor.SetBitWidth(bitWidth);
  glPredictor.globalHTindex = 0;
  glPredictor.create();

  entryNum = byteSize*8/bitWidth;
  predictionTable = new int[entryNum];
  for(int i=0; i< entryNum ; i++)
   {
     predictionTable[i]=0;
   }

  return 0;
}

bool tournaPredictor::stepSimulation(char *address, int branchTaken)
{
  long long int PCAddress = ctoll(address); //type convertion
  iTotalCount++;

  PCAddress = PCAddress >> 2;

  // this mask make sure we got the right bits in PCAdress
  int bitNum = int(log2(entryNum));
  int index = PCAddress & (((0x1<<bitNum)-1));

  glPredictor.globalHTindex = (glPredictor.globalHTindex) & (((0x1<<bitNum)-1));

  //get the prediction value from their own prediction table.
 int predictionFromSa;
   if(saPredictor.predictionTable[index]%4 <2)
    predictionFromSa = 0;
  else
    predictionFromSa = 1;

  int predictionFromGl;
  if(glPredictor.predictionTable[glPredictor.globalHTindex]%4 < 2)
    predictionFromGl = 0;
  else
    predictionFromGl = 1;

  //update their own prediction table
  saPredictor.predictionProcess(index, branchTaken);
  glPredictor.predictionProcess(glPredictor.globalHTindex, branchTaken);

  glPredictor.globalHTindex = ((glPredictor.globalHTindex<<1) | branchTaken);

  // we only train the selector when two predictors are different;
  // we do nothing if they are same (both correct and  wrong).
  if(predictionFromGl == predictionFromSa)
  {
    if(predictionFromSa != branchTaken){
      predictFailCount++;
    }
  }else{
    if(!predictionProcess(index, branchTaken,predictionFromSa,predictionFromGl))
      predictFailCount++;// save execute time
  }
  return 0;
}

// We prefer to firstPredictor here. and right now,
// saPredictor is the firstPredictor, glPredictor is secondPredictor
// 00 for strong secondPredictor 01 for weak secondPredictor
// 11 for strong firstPredictor 10 for weak firstPredicctor
bool tournaPredictor::predictionProcess(int index,int branchTaken,
                    int firstPredictor, int SecondPredictor)
{
  bool correct = 1;
  int prediction = predictionTable[index]%4;

  // the prefer predictor is same with branchTaken.
  if(branchTaken == firstPredictor)
  {
    switch(prediction){
    case 0:
      predictionTable[index]++;
      correct = 0;// even the prefer predictor is right, however the selector choos⇗
es a wrong predictor
      break;
    case 1:
      predictionTable[index]+=2;
      correct = 0;// even the prefer predictor is right, however the selector choos⇗
es a wrong predictor
      break;
    case 2:
      predictionTable[index]++;
      break;
    case 3:
      break;
    }
  }else{
    switch(prediction){
    case 0:
      break;
    case 1:
      predictionTable[index]--;
      break;
    case 2:
      predictionTable[index]-=2;
      correct = 0; //  the prefer predictor is wrong, and  the selector chooses thi⇗
s wrong predictor
      break;
    case 3:
      predictionTable[index]--;
      correct = 0;//  the prefer predictor is wrong, and  the selector chooses this⇗
 wrong predictor
      break;
    }
  }
  return correct;
}
```

```
#ifndef tournaPredictor_H_
#define tournaPredictor_H_

#include "saturaPredictor.h"
#include "globalPredictor.h"
#include "gsharePredictor.h"

// this class is based on saturaPredictor
// add two new predictor class
// redefine the initial function create(),
//the execute function stepSimulation()
// and the train function predictionProcess()
class tournaPredictor : public saturaPredictor
{
 private:
  class saturaPredictor saPredictor;
  class globalPredictor glPredictor;

  bool predictionProcess(int,int,int,int);

public:
  bool create();
  bool stepSimulation(char *, int);


  tournaPredictor();
  virtual ~tournaPredictor();
};

#endif /* tournaPredictor_H_ */
```