```
#include "btBuffer.h"

btBuffer::btBuffer()
{
  iTotalCount = 0;
  missHitCount = 0;
  wrongTargetCount = 0;
  wrongDirectionCount = 0;
  byteSize=2048;
}
// destructor function
btBuffer::~btBuffer()
{
  if(btb!=NULL)
   delete[] btb;
}

// size of btb
void btBuffer::SetSize(int size)
{
  byteSize = size;
}
int btBuffer::GetSize()
{
  return byteSize;
}

// set current prediction from tournament predictor
void btBuffer::SetCurrentPrediction(int value)
{
  currentPrediction = value;
}
// get current prediction from tournament predictor
int btBuffer::GetCurrentPrediction()
{
  return currentPrediction;
}

// the total count of jump and branch instructions
long long int btBuffer::GetiTotalCount()
{
  return iTotalCount;
}
long long int btBuffer::GetMissHitCount()
{
  return missHitCount;
}
float btBuffer::GetMissHitRate()
{
  missHitRate = (missHitCount)/(iTotalCount*1.0);
  return missHitRate;
}
long long int btBuffer::GetWrongTargetCount()
{
  return wrongTargetCount;
}
float btBuffer::GetWrongTargetRate()
{
  wrongTargetRate = (wrongTargetCount)/(iTotalCount*1.0);
  return wrongTargetRate;
}
long long int btBuffer::GetWrongDirectionCount()
{
  return wrongDirectionCount;
}
float btBuffer::GetWrongDirectionRate()
{
  wrongDirectionRate = (wrongDirectionCount)/(iTotalCount*1.0);
  return wrongDirectionRate;
}

/*-----
update the value of entry line in btb
entry line structure:
0000 0000 0000 0000 0000 0000 0000 0000
   tag (22,23,24 bits)             |->

0000 0000 0000 0000 0000 0000 0000     000         0
32bit for target address       ->|     type    valid
------*/

unsigned long long  btBuffer::SetBTBEntry(btbData btbEntry)
{
  unsigned long long btbLine = 0;
  // add pc tag to btb entry line
  btbLine += btbEntry.pcTag;
  btbLine = btbLine << 36;

  // add target address to btb entry line
  unsigned long long temp = 0;
  temp += btbEntry.targetAddress;
  temp = temp << 4;
  btbLine += temp;

  // add type
  temp=0;
  temp += btbEntry.type;
  temp = temp << 1;
  btbLine += temp;

  //add valid
  btbLine += btbEntry.valid;

  return btbLine;
}

/*-----
get the value from the btb entry line
entry line structure:
0000 0000 0000 0000 0000 0000 0000 0000
   tag (22,23,24 bits)             |->

0000 0000 0000 0000 0000 0000 0000     000         0
32bit for target address       ->|     type    valid
------*/
btbData btBuffer::GetBTBEntry(unsigned long long btbLine)
{

  btbData btbEntry;
  unsigned long long temp = btbLine;

  //get the valid
  btbEntry.valid= temp & 0x1; // get the last 1 bit for

  //get the type
  temp=btbLine;
  temp = temp >> 1 ;
  btbEntry.type=temp & (0x7);

  //get the target address
  temp=btbLine;
  temp = temp >> 4 ;
  btbEntry.targetAddress=temp & (0xFFFFFFFF);

  // get the tag
  temp=btbLine;
  btbEntry.pcTag = temp >> 36;

  return btbEntry;
}
// construct the branch target buffer
```

```
bool btBuffer::create()
{
  // entryNum is based on the size of BTB
  entryNum = byteSize >> 3;
  bitNum  = int(log2(entryNum)) ;
  btb = new unsigned long long[entryNum];
  for(int i=0; i< entryNum ; i++)
   {
     btb[i] = 0;
   }
  return 0;
}

bool btBuffer::stepSimulation(char *address, char *targetAddress, int branchTaken,
int type)
{
  iTotalCount++;

  unsigned int PCAddress = ctoll(address);
  unsigned int target = ctoll(targetAddress);

  PCAddress = PCAddress >> 2;// we do not use the first two bits.
  unsigned int temp = PCAddress;

  int index = temp & ((0x1 << bitNum)-1);
  int tag = PCAddress >> bitNum;

  currentBTBEntry = GetBTBEntry(btb[index]);
  btbData realEntry;
  realEntry.valid = 1;
  realEntry.type = type;
  realEntry.targetAddress = target;
  realEntry.pcTag = tag;

  if(!currentBTBEntry.valid){
    btb[index] = SetBTBEntry(realEntry);
  }else{
    // miss hit in BTB
    if((type == 2 && currentBTBEntry.pcTag != realEntry.pcTag)||
       (type == 1 && branchTaken == 1 && currentPrediction ==1 && currentBTBEntry.p
cTag != realEntry.pcTag))
         missHitCount++;

    // wrong target address
    if((type == 2 && currentBTBEntry.pcTag == realEntry.pcTag)
       || (type == 1 && currentPrediction ==1 && branchTaken == 1 && currentBTBEntr
y.pcTag == realEntry.pcTag) )
      if(currentBTBEntry.targetAddress != target)
    wrongTargetCount++;

    // wrong direction
    if(type == 1 && currentBTBEntry.targetAddress == target && currentBTBEntry.pcTa
g == realEntry.pcTag)
      if(currentPrediction == 0 && branchTaken == 1)
    wrongDirectionCount++;

    // update the btb entry line
    if(type == 2 || (type == 1 && branchTaken == 1))
      btb[index] = SetBTBEntry(realEntry);
  }
  return 0;
}

// to convert a char[] to a long long int
// to get the integer of instructure address
unsigned int btBuffer::ctoll(char s[])
{
    unsigned 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 BTBUFFER_H_
#define BTBUFFER_H_

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

struct btbData
{
  int  valid;
  unsigned int pcTag;
  unsigned int targetAddress;
  int  type;
} ;

class btBuffer
{
 private:
  int currentPrediction; // current prediction from predictor;
  btbData currentBTBEntry;

  int  byteSize; // the size of btb, here is 2K 4K and 8K.
  long long int iTotalCount;  // the total number of jump and branch instructers

  long long int missHitCount;
  float missHitRate;  // the miss rate hit of this btb
  long long int wrongTargetCount;
  float wrongTargetRate;  // the wrong target rate of this btb
  long long int wrongDirectionCount;
  float wrongDirectionRate;  // the wrong direction rate of this btb

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

public:
  int entryNum;
  int bitNum;


  btbData GetBTBEntry(unsigned long long);
  unsigned long long SetBTBEntry(btbData);

  unsigned long long* btb;

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

  int GetCurrentPrediction();
  void SetCurrentPrediction(int value);

  long long int GetiTotalCount();
  long long int GetMissHitCount();
  float GetMissHitRate();
  long long int GetWrongTargetCount();
  float GetWrongTargetRate();
  long long int GetWrongDirectionCount();
  float GetWrongDirectionRate();


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

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

  btBuffer();
  ~btBuffer();
};

#endif /* BTBUFFER_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))+1;
  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_ */
```

```
/*
    CS4431 Branch Target Buffer Simulation

    -- 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 "btBuffer.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;
    int sizePredictor=8192;

    FILE *fp;
    int maxlen = 23;  // to make sure get whole value in a line;
    char buf[23];

    char filePath[5] = "test";
    char source[8];
    char target[8];
    int type=0; // branch = 1, jump =2, others = 3
    int branchState=0;
    if( !(fp = fopen( filePath, "rt" ) ) ){
      printf("could not open the file %s \n",filePath);
      return 1;
    }

    for(int i=0;i<3;i++){
      tournaPredictor* toPredictor = new tournaPredictor();
      // to share the area with three preiction tables, >>2 is ignoring some waste here.
      toPredictor->SetSize(sizePredictor>>2);
      toPredictor->SetBitWidth(bitWidth);
      toPredictor->create();

      btBuffer* btb = new btBuffer();
      btb->SetSize(areaSize[i]);
      btb->create();

      rewind(fp); // back to the head of the file
      while (fgets( buf, maxlen, fp ) != NULL) {
    sscanf(buf,"%s %d %d %s",source,&type,&branchState,target);
    if(type ==1 || type == 2){
      // since only jump and branch could reach here, we assume predition equals 1, it only
          // changes for branch
      int prediction = 1;
      if(type == 1){
        toPredictor->stepSimulation(source,branchState);
        prediction = toPredictor->getCurrentPrediction();
      }
          btb->SetCurrentPrediction(prediction);
          btb->stepSimulation(source, target,branchState,type);
       }
      }
      // save test data
      FILE *file;
      file = fopen("result.txt","a+"); /* add test result file  or create a file if it does not exist.*/
      fprintf(file,"%s,%d,%lld,%lld,%lld,%lld,%f,%f,%f\n","BTB",areaSize[i],btb->GetTotalCount(),btb->GetMissHitCount(),
          btb->GetWrongTargetCount(),btb->GetWrongDirectionCount(),btb->GetMissHitRate(),btb->GetWrongTargetRate(),
          btb->GetWrongDirectionRate());
      fclose(file);

      if(btb!=NULL)
    delete btb;
      if(toPredictor!=NULL)
    delete toPredictor;
    }
    fclose(fp);
   return 0;
}
```

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

main: main.cpp saturaPredictor.cpp globalPredictor.cpp tournaPredictor.cpp  btBuffe↗
r.cpp
    ${CC} ${CFLAGS} -g -o main main.cpp saturaPredictor.cpp globalPredictor.cpp tou↗
rnaPredictor.cpp btBuffer.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 result.txt
```

```
#include "saturaPredictor.h"

saturaPredictor::saturaPredictor()
{
  iTotalCount = 0;
  predictFailCount = 0;
  predictionHitRate = 1.0;
  bitWidth=2;
  byteSize=2048;
}
saturaPredictor::~saturaPredictor()
{
  //  if(predictionTable != NULL)
  //    delete[] predictionTable;
}

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 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))+1;
  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
unsigned int saturaPredictor::ctoll(char s[])
{
    unsigned 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

  unsigned 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()
{
  if(predictionTable!=NULL)
    delete[] predictionTable;
}

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;
}

int tournaPredictor::getCurrentPrediction()
{
  return currentPrediction;
}

bool tournaPredictor::stepSimulation(char *address, int branchTaken)
{
  iTotalCount++;

  unsigned int PCAddress = ctoll(address); //type convertion
  PCAddress = PCAddress >> 2;

  // this mask make sure we got the right bits in PCAdress
  int bitNum = int(log2(entryNum))+1;
  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;

  // get the value from the tourament predictor
  int selector = predictionTable[index]%4;
  if (selector < 2) // weak select
    currentPrediction = predictionFromGl;
  else // strong select
    currentPrediction = predictionFromSa;

  //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 selector = predictionTable[index]%4;

  // the prefer predictor is same with branchTaken.
  if(branchTaken == firstPredictor)
  {
    switch(selector){
    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(selector){
    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"

// 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;
  int currentPrediction;

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

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

  int getCurrentPrediction();

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

#endif /* tournaPredictor_H_ */
```