Matt Bennett
Partner: Adam Ziel

ELEC 3275 - Computer Architecture
Assignment 8 - Branch Predictor

Description

Branching behavior of processor instructions can be guessed using various algorithmic methods of prediction in order to allow later processes in a pipelined architecture to continue while minimizing necessary stalling. In order to create a basic branching prediction algorithm, a system was development in which two branch outcome histories are stored. One history stores up to one hundred previous branching behaviors with the most recent behavior at the first index and the first instruction at the largest occupied index. The second history is stored in an integer and is calculated from the first history. This second history represents the last three branching outcomes based on the three bit binary number corresponding to each of the eight pattern possibilities. In binary, these integers most significant bit, the fours place, represents the most recent outcome stored, the twos place the next latest outcome, and the ones place the latest of the three. Predictions are made by iterating over the larger history and comparing each three element sequence of bits to the smaller three bit history. If an occurrence of the three bit most recent history is found in the same sequence in the larger overall history, the first outcome that followed that sequence is predicted. If the three bits cannot be found in the overall history or the overall history has not yet been build to a length long enough for comparisons, the last recorded outcome is predicted. The first instruction branching prediction is always predicted to be zero since there is not any available data to make a more educated guess.

Results/Verification

The algorithms hundred bit outcome storing, three bit outcome storing, and first occurrence lookup were verified using a basic test function to print the values. The total history was verified by comparing the last resulting total history value printed to the console to the data file. The recent history was confirmed by using the integer values to index into an array of the numbers binary representation. These binary strings were then compared to both the first three indexes of the total history and the last three outcomes from the input files. Finally, the three bit pattern lookup for prediction was verified by determining the first occurrence in the total stored outcomes, and verifying that the outcome that followed the pattern in the history was the predicted outcome.

Figure 1: The total history value from the final instruction of the program following a run with file input BranchInput2.txt is can be observed to directly match the reverse of the files values. The highlighted zeros precede the programs recorded history since the total history stores more values then the BranchInput2 file provides and is initialized to contain all zero elements.

```
Total history:
{{ 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1
1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }}
```

Figure 2: The current pattern is identified as 011. This pattern is looked up in the total history to find most recent pattern with a following outcome to predict. The first 011 sequence occurs at indices zero to two,

however, this is skipped since no outcome can be guessed based on this data. The next most recent occurrence of the pattern is found at indices eight to ten and the outcome that followed, index seven, the outcome that followed this sequence is predicted.

```
Prediction: 1    Input: 1        Correct: True
Recent history: 011
 Total history:
 {{ 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }}
```

Figure 3: Current pattern is determined to be 101, and verified by observing previous three input values from file. Again, the first occurrence of 101 is ignore because meaningful values only begin at three bit sequences with the starting least significant bit at the third index. The next occurrence of the 101 sequence can be seen from indices two to four. The outcome that occurred following this sequence is the first index of the total history, a zero or not taken value. Since the last time this sequence resulted in no branch, the algorithm predicts branching is not taken and returns a zero value to represent this.

```
Prediction: 0    Input: 0        Correct: True
Recent history: 101
 Total history:
 {{ 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0 }}
```

Code

```
/*
 * File:   main.c
 * Author: Matt Bennett
 *         Adam Ziel
 *
 * Description: Predicts branching behavior based on past branching behavior
 *              of a provided series of branching and non branching values.
 */

#include <stdio.h>
#include <stdlib.h>
#include "string.h"

int counter;
int startValue = 0; // startValue is the first prediction guess since no other
                    // data is available from which to predict values
int currentPattern; // Updated to reflect last three values of outcomes
int history[ 100 ]; // Array to store the value of last 100 branching outcomes
char* binary[] = { "000", "001", "010", "011", "100", "101", "110", "111" };
                    // String array to represent the decimal pattern in binary
                    // Used to verify the pattern was working during testing.

/**
 * Uses a starting index of history representing the LSB of a three bit binary
 * number with bit significance increasing with decreasing index. Returns the
 * decimal representation of this binary number.
 *
 * @param startingIndex     earliest index history in pattern to find
```

```
 * @return                  representation of the patterns state based on the
 *                          past three branch outcomes with the most recent as
 *                          the MSB of a 3-bit binary number
 */
int getPattern( int startingIndex ){
    return history[ startingIndex-2 ]*4 + // MSB; Least index/oldest value
           history[ startingIndex-1 ]*2 +
           history[ startingIndex ];      // LSB; Greatest index/newest value
}

/**
 * Moves each element of the history array to the next index and places the
 * given value at the starting index.
 *
 * @param value  value to be added as most recent outcome in history
 */
void push( int value ){
    for ( int i = 99; i >= 0; i-- ){ // For each index of array
                                                        // from last index to first
        history[ i ] = history[ i - 1 ];                // Move previous element
    }                                                   // to current index
    history[ 0 ] = value;                               // Set first element to
                                                        // value of argument
    currentPattern = getPattern( 2 );                   // Update the outcome
                                                        // pattern to reflect
}                                                       // added value

/**
 * Looks at history array to determine if the pattern of the last three outcomes
 * has occurred before in the same order and takes the next outcome that
 * occurred. If the history array is empty, startValue is predicted, otherwise
 * the outcome of the previous instruction is used.
 *
 * @return  prediction of next branching instruction (0, not taken; 1, taken)
 */
int getPrediction(){
    int historyPattern;
    if ( counter <= 0 ){            // Check if first instruction
        return startValue;          // Guess value
    }else if( counter <= 3) { // Check if too few instructions to
                                        // look for a pattern
        return history[ 0 ];            // Predicts most recent outcome
    }else {
        currentPattern = getPattern( 2 ); // Determine most recent three
                                        // outcomes and represents as integer
        for ( int i = 3; i <= 99; i++ ) { // Each prior outcome
            historyPattern = getPattern( i );           // Finds pattern
            if ( historyPattern == currentPattern ){    // Compares to current
                return history[ i - 3 ];  // Guesses next value that occurred if
                                        // patterns are the same
            }
        }                                // Could not find prior occurrence
        return history[ 0 ];             // Predicts most recent outcome
    }
}
/**
```

Matt Bennett
Partner: Adam Ziel

```
 * Print three bit and full histories to console to verify algorithm behavior.
 */
void testing(){
    printf( "Recent history: %s \n", binary[ currentPattern ]);
    printf( " Total history: \n {{ %d", history[ 0 ]);
    for ( int i = 1; i < 100; i++ ){
        printf( ", %d", history[ i ]);
    }
    printf( " }} \n");
}

/**
 * Reads binary input (1 bit per line) from a text file representing instruction
 * branching and predicts if the next instruction will need or not need to
 * branch. Information on the actual and predicted values are printed to the
 * console along with general accuracy information.
 *
 * @return      success
 */
int main(){
    FILE *fp, *fpw;
    fp = fopen("BranchInput2.txt", "r");        // Open and read text file
    fpw = fopen("Output.txt", "w");             // Open and write to text file
    int outcome, prediction;                    // Initialize variables for
                                                // Actual and predicted outcomes
    int correct = 0, incorrect = 0;             // Initialize to 0 a counter
    counter = 0;                                // for number of instructions
                                                // and result of predictions
    char* boolString;                           // String to store boolean T/F

    while ( 1 ) {                               // Until break
        prediction = getPrediction();           // Predict taken or not taken
        printf( "Prediction: %d", prediction ); // Print prediction to console

        fscanf(fp, "%d", &outcome);             // Read next outcome from file
        printf( "\t Input: %d", outcome);       // Print outcome from file
        if ( outcome == prediction ){           // If correctly predicted the
                                                // Outcome
            correct++;                          // Increase correct counter
            boolString = "True";                // Set string to TRUE
            // Print prediction success to file
            fprintf(fpw, "Prediction %d was correct \n", counter);
        } else{                                 // If incorrect prediction
            incorrect++;                        // Increase incorrect counter
            boolString = "False";               // Set string to FALSE
            // Print prediction failure to file
            fprintf(fpw, "Prediction %d was incorrect \n", counter);
        }
        printf(" \t Correct: %s \n", boolString);//Print T/F value of prediction

        //testing();                            // Output basic history info
                                                // to verify program behavior

        push( outcome );                        // Add outcome to history array

        counter++;                              // Increase instruction count
```

Matt Bennett
Partner: Adam Ziel

```
        if ( feof( fp ) )                       // Check if end of file
            break;                              // If end of file, leave loop
    }

    // Print to console and file # of  correct and incorrect predictions
    // and overall accuracy
    printf("\n Correct: %d; \t Incorrect: %d", correct, incorrect);
    printf("\n \t  %.2f%% accuracy \n", (double) correct / counter * 100);
    fprintf(fpw, "Correct: %d; \t Incorrect: %d \n", correct, incorrect);
    fprintf(fpw, "%.2f%% accuracy \n", (double) correct / counter * 100);
    fclose(fp);
    fclose(fpw);
    return 1;
}
```