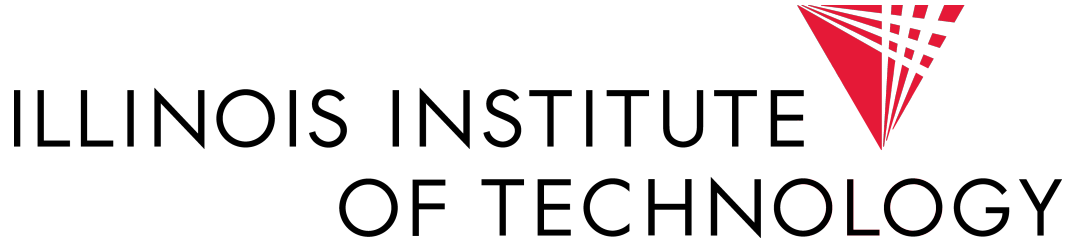

IIT

ECE585

Advanced Computer Architecture

Branch Predictor Implementation

Author:
Maxime Descos

Abstract

This report will cover how three different branch predictors were implemented in VHDL and tested. This predictors were created using small generic blocs and implement an improvement in the pipeline of a processor for conditional branches. The three units will be compared and their algorithms explained.

This project concludes the course Advanced Computer Architecture and allows to put in practice the knowledge acquired and to go deeper in the comprehension by implementing ourselves one key component of modern processors.

Contents

1 Introduction

The goal of this project is to implement in VHDL three branch predictors: 1-bit branch predictor, 2-bit branch predictor, (4,2) correlating branch predictor. The project is concluding the course Advanced Computer Architecture given by Dr. Oruklu. This course gives the keys to understand how modern processors work.

The project was split over three different designs of branch predictors. Each design is an extension of the previous one to gradually achieve a complex system. But the first predictor was split in two in order to create step by step.

The method used for this project was the following:

- Review of the concept of branch predictor using the textbook (2)
- Design of the first 1-bit predictor using an array instead of counters to simplify the first step.
- Implementation in VHDL on ModelSim of the design
- Design of the 1-bit predictor using only basic units such as counters, multiplexer, comparator and demultiplexer.
- Design of the 2-bit counter
- Implementation of the 2-bit predictor using the previous created blocs and the 2-bit saturating counter.
- Review of correlating predictor
- Design of the (4,2) correlating predictor
- Implementation using previous blocks and the new ones needed, such as global branch history shift register.

Each implementation of a unit was following by a testbench to test the functionality of the block before adding it to the system. For the first two predictors, they were tested with a small testbench before using the test program.

Some issues came up during the course of this project and were overcome by finding new way to test a unit or through review of designs and predictor concept.

The first issue was how to start the first predictor. The solution was to split it into two. The first design was using a table to store the predictions and only a comparator and a counter for the misses. This simple design allows to test the comparator and the counter, but mostly as a small step into the project.

In a second time, the first predictor was finished using counters, multiplexer and demultiplexer instead of the high-level array.

After the implementation of the 2-bit saturating counter and the following predictor were straight-forward.

The design of the third predictor was also a bit challenging but the design of the first two led naturally to the last one. Its implementation however was not so easy. Each block was tested before but the smaller simulation than the test program didn't allow to see all the cases and when the whole system was made, some blocks such as the demultiplexer 1:16 were wrong. The issue with this predictor was the amount of signals which confused the analyze of the waves. To overcome this issue, a do file was made to select only the signals needed and a overlook of the waves for particular cases like the start or the end was enough to find the unit causing a problem.

However, the principal solution for each predictor was to find the theoretical limit of each one. This limit is the number of misses each predictor should have given the test program. This limits were the goals to achieve for the predictors and it was the main test to be sure that the implementation has an optimal behavior or something was wrong somewhere in the system.

In the next part, the algorithm behind each predictor will be discussed.

2 Background

Branch predictor are used in modern processor to overcome the inherent issue of conditional branches in a pipeline.

In the pipeline, the instruction i+1 will be launched before the instruction i is finished. But with a branch, the whole problem is to know what will be the next instruction to launch it. If the processor has to wait, a latency will occur and the processor will be slowed by that.

The solution is to implement a branch predictor to find if the branch is taken or not taken and launch the next instruction following this prediction. If the prediction is right, the processor will continue without any latency. In the case of a misprediction, the next instruction is invalidated and the processor will start the right instruction on the next clock cycle, there is in this case a latency but a latency unavoidable because of the structure of a branch.

The project instruction (1) propose to implement three different branch predictor but basically, two different concepts.

First of all, all the predictors are dynamic predictors. The content of the prediction will change during the execution of the program. An example of static predictor will be to consider all the branches as predicted taken. The static predictors can be obviously more complex than that but they will never adapt to the program executed and so, will always be less reliable.

2.1 Local predictor

The first and second predictor are local predictors. Each branches will have an history kept and when the branch will come up during the execution, the processor will go to the history of the branch to look what was the last result(s) and will predict the result over this.

This concept is based on branches behavior. When a branch is taken there is chances than the next one will be as well. One can use the example of the loop.

The loop is going to do certain number of iteration before stopping. There is a branch at the beginning of each iteration to be sure that the loop doesn't have to stop. But for all the iteration before, the branch is almost not necessary and can be predicted easily.

For example, if the program has the following piece of code:

for(int i=0; i < 1000; i++)

There is 1000 iterations, so 1000 branches. 999 branches are taken and the last one is not taken. With a 1-bit local predictor, there will be only two mispredictions. One at the beginning to initialize the history on "taken" and one at the end, when the loop is over.

The first predictor is a 1-bit predictor. That means that each branch will have a history of one bit, so only the last outcome for the branch.

The 2-bit branch predictor follows the following scheme (from the project instruction(1)):

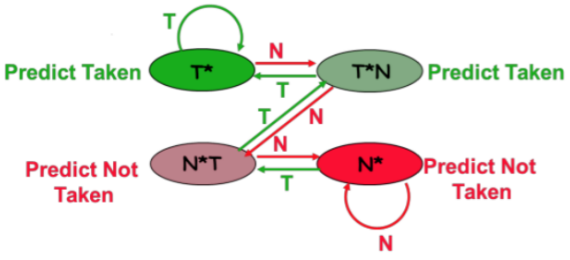

Figure 1: 2-bit saturating counter

To explain it simply, to go from a taken prediction to not taken one (the two extremes), the two next outcomes have to be "not taken".

This allow the prediction to be more stable. The prediction will not change for one mistake. Usually, this type of counter makes less mispredictions but in some particular program, it can have more than the 1-bit predictor.

2.2 Global predictor

The third predictor to implement is a global predictor. That means the prediction will be done using the outcomes of the previous branches and not the history of the

particular branch used. It is based on the principle that the result of a branch is often due to the context around, so the other branches.

To show this, one can take a look at the following code:

j = 0;

if(i < 5) j++;

if(j >0)

If the first branch is taken, the next one will be taken too. In most programs, the result of branches affect the outcomes of the other branches.

With this concept, another type of predictor can be done. So the history of all the branches will be kept in a register. The predictor to implement kept the outcomes of the four previous branches. This four bits will select one 2-bit counter over 16 (there is 16 possibilities for 4 bits). So for each branch, there is 16 counters for each possibility over the 4 previous outcomes.

The use of a 2-bit counter is for the same reason than before, to make it more reliable.

3 Architectural exploration of dynamic predictors

In this part, the implementation of each predictor will be discussed.

Each predictor is made from different generic blocs, so the design can be synthesizable. The order of the implementation makes it simpler to create the units on one design and to reuse the elements for the next system.

3.1 1-bit predictor

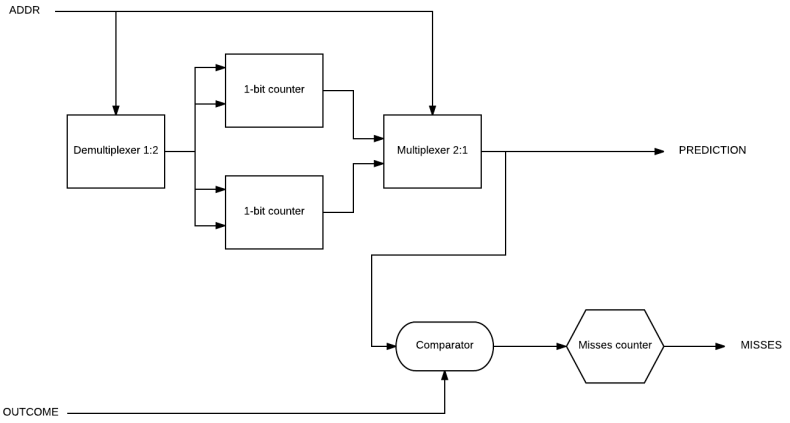

Figure 2: 1-bit branch predictor

The picture above is the architecture of the 1-bit branch predictor. It is made of counters to store the local history, one multiplexer to select the prediction, a comparator to check if there is a miss or no, a counter to determine the number of misprediction and a demultiplexer to change the value of the counter.

As said before, this predictor is fairly basic and there is no interesting about it. The only important features are:

- A clock for the counter to be sure that the value is changed on a clock cycle
- The comparator is made using a XOR gate between the OUTCOME and the PREDICTION
- An INIT signal initialize the 1-bit counters to 0 and the misses counter to 0
- 4 inputs: CLOCK, INIT, ADDR (address of the branch) and OUTCOME
- 2 outputs: PREDICTION (to be used by the processor to fetch the next instruction) and MISSES (to determine the number of mispredictions)

3.2 2-bit predictor

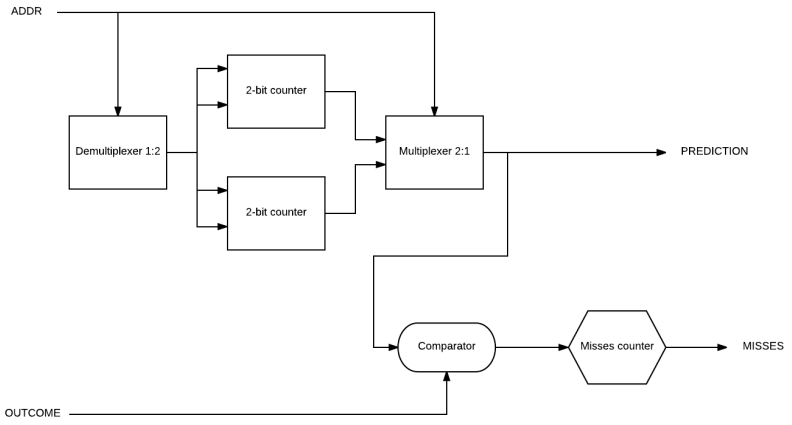

Figure 3: 2-bit branch predictor

As said before, no real differences with the previous one. The 2-bit counter is a saturating counter from 0 to 3 using the scheme seen before. If the OUTCOME is 1, the counter is incremented (if the value is not 3 already), if the OUTCOME is 0, the counter will be decremented (if the value is not 0 already). This will be done on a clock cycle (synchronous element) and if the demultiplexer said, by putting the

enable signal to 1, to this counter to change. At the end, the counter is divide by two and the quotient of the integer division is either 0 or 1 and will be the prediction.

3.3 (4,2) correlating predictor

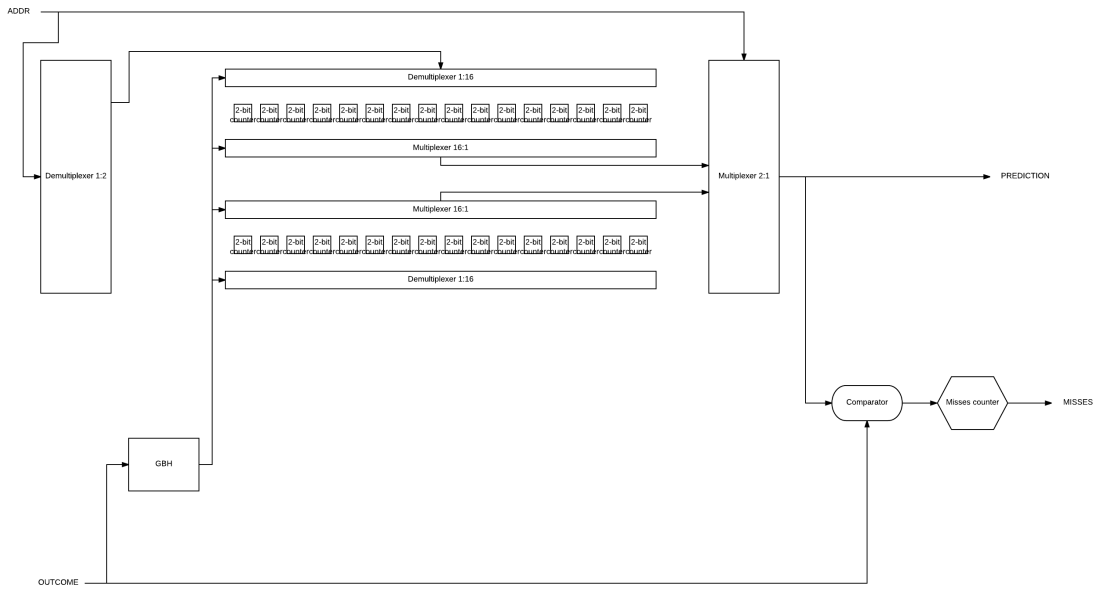

Figure 4: (4,2) branch predictor

Each branch will have 16 2-bit saturating counters for every possibility from the 4 previous outcomes. The principle is the same has the previous predictors on this part. The demultiplexer 1:2 and multiplexer 2:1 will choose one branch using the address given. The demultiplexer 1:16 and multiplexer 16:1 will choose one counter using the 4 bits from the Global Branch History.

The Global Branch History is a simple 4-bit shift register. Each outcome will be added on the least significant bit and the others bit will be shifted on the left. This block is also synchronous and the action will be done on every clock cycle.

The implementation of each predictors was not the most complicated of the project. After the design was made, creating and adding each elements was straightforward. The debugging and validation part of each block was the most time consuming part.

4 Functional validation and verification

Each unit was tested using testbenches. The testbenches were implementing several cases to be sure the element was functioning accordingly. But to be sure that the whole system was working well, the theoretical number of mispredictions for the test program has to be known. This section will discuss the limit for each predictor and the simulation will help to validate this result.

4.1 Test program

The MIPS code of the test program can be found in the appendix.

This code is simple but has two loop one into the other, so the predictors have to adapt and it is more realistic than loops without intrication.

In a second time a script in C was made to reproduce the behaviour of the MIPS code and to write a text file with the id of the loop and the outcome: taken/not taken.

4.2 1-bit predictor

The test program is made of two loops, one inside the other.

The inner loop is done with a counter from 0 to 3, so there will be 4 iterations of the inner loop every time. This loop will have the following behaviour: T, T, T, NT. Indeed, the last iteration will be the end of the loop and so, the branch will be not taken. Since, the system has a 1-bit counter, the prediction will be the outcome of the previous iteration, hence NT, T, T, T.

Finally, there is so two mispredictions for the inner loop.

The outer loop is made from 0 to 999. First of all, there will be so 1000*2 = 2000 misprediction due to the inner loop. Now, the behaviour of the outer loop will be: T, T, T, T, ..., NT. The counter is initialize on Not Taken, so there will be one misprediction at the beginning and one at the end when the loop is over.

Finally, this predictor should have 2002 mispredictions.

Now, the simulation using the test program gives the same result on the MISSES output (2002 in hexadecimal is 07D2)

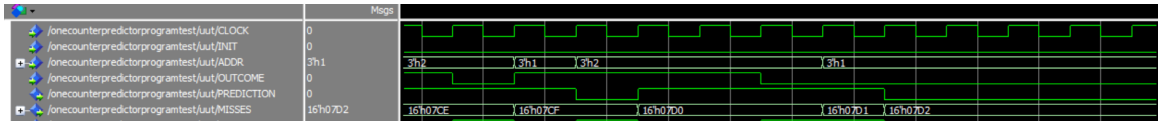

Figure 5: 1-bit predictor simulation

4.3 2-bit predictor

The outcomes is still the same but the behaviour of this predictor will be a bit different.

For the first iteration of the inner loop will be predicted as: NT, NT, T, T. The 2-bit counter is initialize at 0, so it will take two iteration to go to 2, so a prediction of Taken.

All the other iterations of the inner loop will be taken, because even if the last iteration of the inner loop is not taken, the counter was saturation to 3, so it would take one more NT to go back to a NT prediction. So the behaviour is T, T, T, T. The inner loop will so have 1 misprediction and 3 at the beginning. Hence, 1*999 + 3 = 1002 mispredictions.

The outer loop will take two times to have the counter on taken, so two mispredictions at the beginning and one at the end.

Finally, this counter will have 1005 mispredictions (3ED).

The simulation gives the same result.

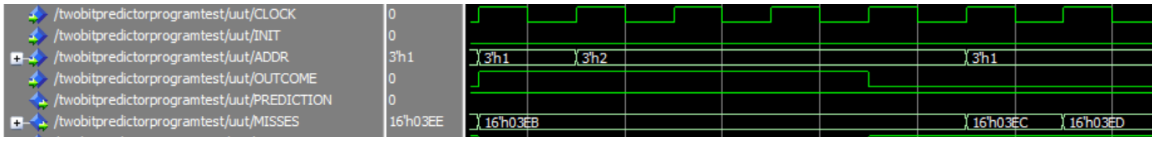

Figure 6: 2-bit predictor simulation

Therefore, the predictor can be validate. To be sure that it follows the theoretical behaviour, the simulation of the beginning is given and one can see that the mispredictions from the analyze are the same.

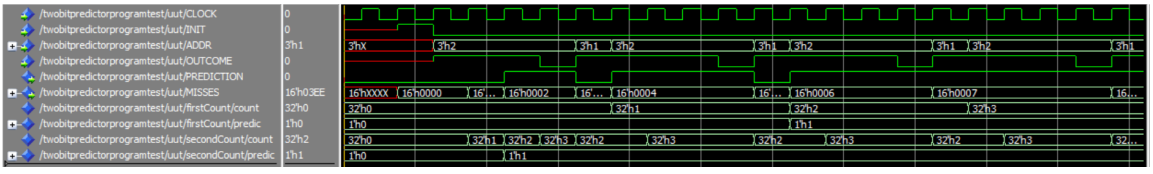

Figure 7: 2-bit predictor initialization simulation

4.4 (4,2) correlating predictor

The counter selected will be from the 4 previous outcomes, so the behaviour of the Global Branch History is (with the hexadecimal and the decimal):

Initialization: 0000(0-0), 0001(1-1), 0011(3-3), 0111(7-7), 1110(E-14), 1101(D-13)

After: 1011(B-11), 0111(7-7), 1111(F-15), 1110(E-14), 1101(D-13)

During the functioning, the inner loop will only use the 2-bit counters 13, 11, 7 and 15. Indeed, the 1110 will be for the outer loop since, the last outcome of the inner loop is not taken, so 0.

The counters 13,11 and 7 will rapidly (exactly after two inner loops) saturate on 3, so on taken. There will so be 6 mispredictions at the beginning to saturate this counters.

The counter 15 will also saturate because the inner loop is always not taken on this iteration since it is the last iteration, but the saturation will be on 0, not taken. There will be no mispredictions since the counter is initialized on 0 already.

For the outer loop, as said, only one counter will be used, the 14th. This counter will rapidly (after two) saturate on 3, so there is two misprediction at the beginning. But also one at the end after the last iteration. So there is 3 misprediction from the outer loop.

Moreover, there is some counters used only at the beginning. It is the counter 0 for the inner loop, the counter 1 for the inner loop and the counter 3 for the inner loop. This three counter will add three more mispredictions since they are initialized on 0 and the branch will be taken.

Finally, this predictor should have 6+3+3 = 12 mispredictions.

The simulation gives the same results, so the system can be validate.

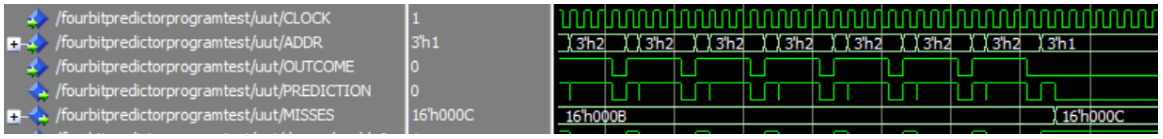

Figure 8: (4,2) predictor simulation

As before, the initialization should be also checked to be sure that the predictor has a normal behaviour. The simulation shows actually the same behaviour.

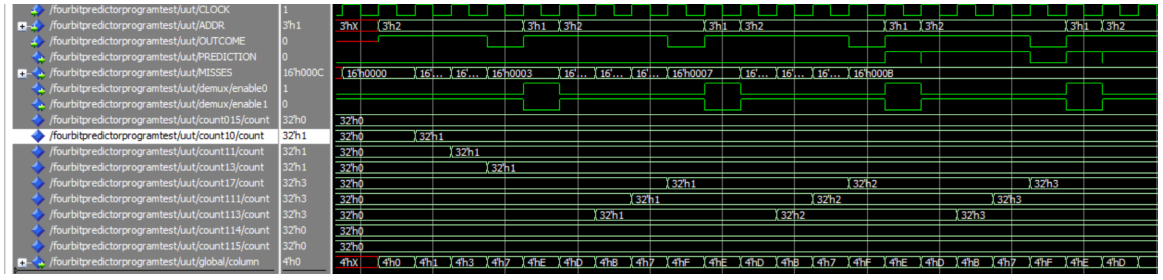

Figure 9: (4,2) predictor initialization simulation

4.5 Other test programs

In order to better evaluate each predictor, three other programs were made with different specifications.

As seen, the first test program put the (4,2) correlating predictor has the best predictor.

The second program (branchCounter2) is the same has the first one but with a another loop inside the second one with 20 iterations. That will influence on the last predictor since the Global Branch History only use 4 bits, so was perfectly made for the first program.

The third test program (branchCounter3) is made of two loop intricated but the second loop is done randomly. There is a if statement with a random condition.

The fourth program (branchCounter4) was finally 4 loops intricated.

5 Results

The performance can be analyzed using the two following charts:

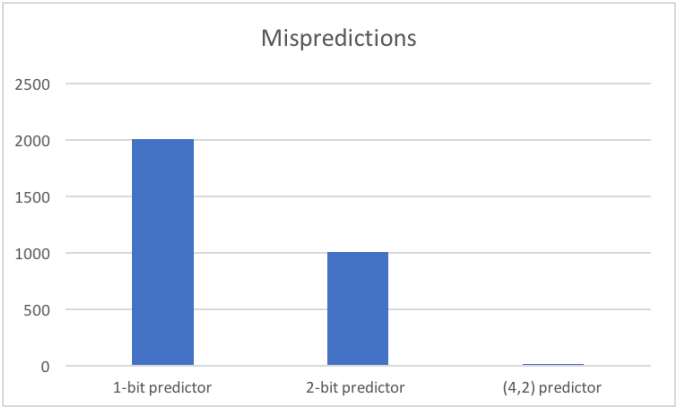

Figure 10: Mispredictions (linear scale)

The first chart gives the difference between the two first predictors. The 2-bit predictor is twice more reliable than the 1-bit. This is easily explained by the 2-bit saturating counter which going to take into account more outcomes than the simple 1-bit counter.

However, the 1-bit counter is useful as a base comparison. Even if it results is not good (4000 predictions, around 2000 misses, so a 50% of good prediction).

Now, the (4,2) correlating predictor is 100 times more reliable than the 2-bit predictor and 200 times than the 1-bit predictor. This also is easily explained by the test program used. The test program is predictable and after a few misses due

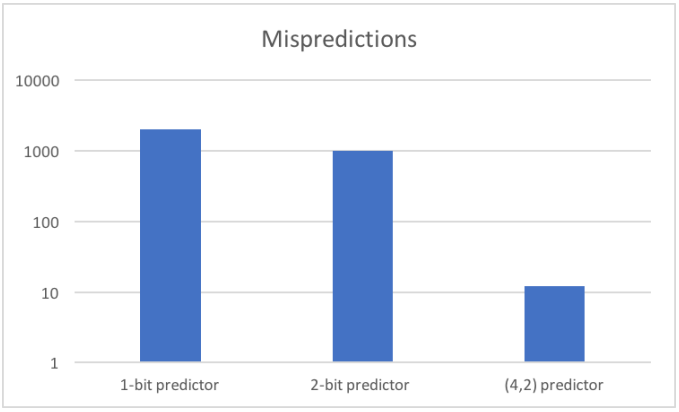

Figure 11: Mispredictions (logarithmic scale)

to the initialization (impossible to remove), the behaviour is easily known and the predictor will so not mispredict anymore.

According to this project, the (4,2) correlating predictor is an excellent predictor : $12/4000 = 0.3\%$ of misprediction. However, one has to pay attention to the test program. In order to really test these predictors, different test programs should be used.

On the other hand, the last predictor is more expensive than the two others. Each branch needs 16 counters and there is two demultiplexers and two multiplexers to add also. Finally, there is 16+16+2+2+1+1+1+1 = 40 elements for that predictor, comparing to the 2-bit predictor and its 6 elements.

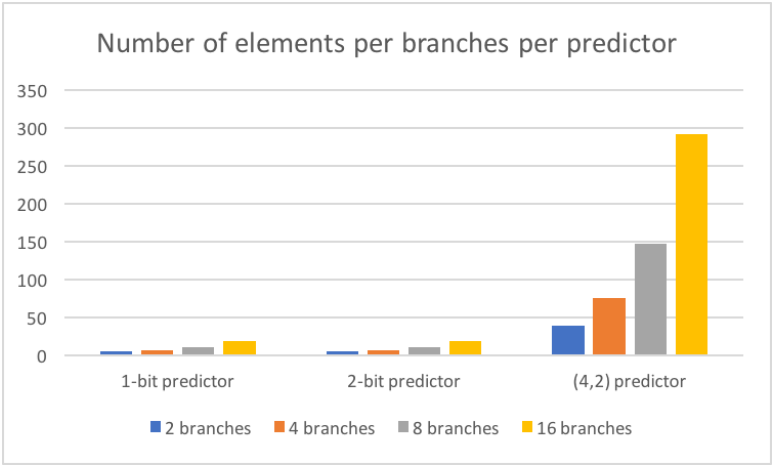

Figure 12: Number of elements per predictor

The figure shows the exponential scale between the correlating predictor and the

local history predictors.

Finally, the (4,2) predictor is much more reliable than the others but also more expensive. The implementation was only for two branches but for each new branches, you need 18 more units and 1 for the 2-bit predictor. Processor architect have to make a trade-off between performance and cost.

Now, with the optional test program, the (4,2) predictor is not as good as the first program shows.

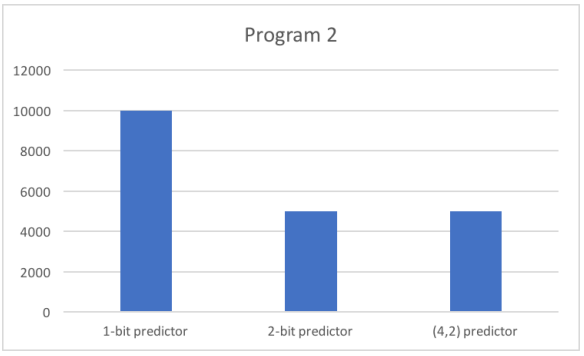

Figure 13: Mispredictions on program 2

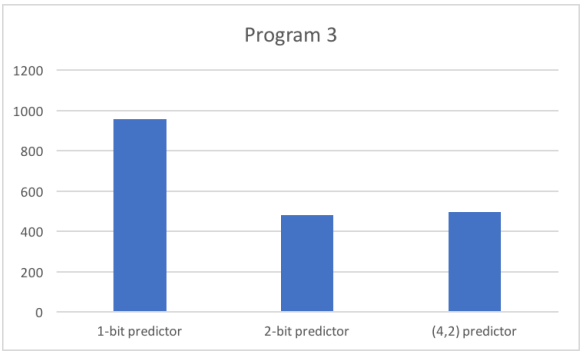

Figure 14: Mispredictions on program 3

Finally, on this other programs, the correlating predictor is not even as good as the 2-bit local predictor. This is explained because the first program was made with an inner loop of 4 iterations, so adapted to the 4 bit of the global history, but with different number of iterations, the global history will have no advantage and the number of misses will be directly due to the 2-bit counter. The bigger number of misses between the 2-bit counter and the correlated is due at the more important initialization stage because of the greater number of counters in the former predictor.

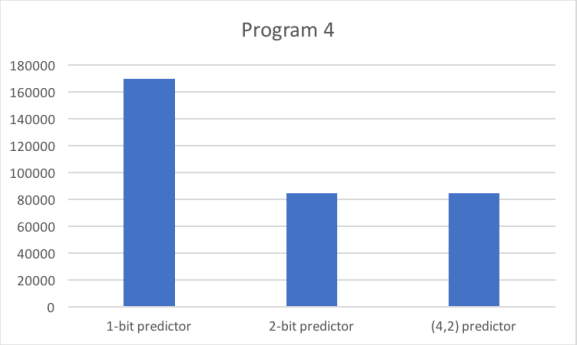

Figure 15: Mispredictions on program 4

6 Conclusion and future works

The predictors implemented have a fairly good behavior and specially the last predictor almost never missed (0.3%) for the original test program but not for the others.. The project makes me realize the importance of predictor and that some not so complex predictor can still make a big difference in the pipeline.

The next step for this project will be to implemented a new predictor. In particular, a tournament predictor could be interesting. This predictor uses correlation and local history and make a choice over thus two predictions.

Finally, I really enjoyed to spend time on this project. It allowed us to implement some modern processors concept and it was not too difficult, so we could finish it.

7 Appendix

The HDL codes are not put in the report but are given with it.

```
# $s0 == c, $t0 == i, $t1 == j, $t2 == 1000, $t3 == 4

move $s0, $zero   # initialize c to 0
li $t0, 0     # initialize i to 0
li $t2, 1000    # store the end value of i in a register for comparison

B1: beq $t0, $t2, end # loop test
addi $s0, $s0, 1  # c++
addi $t0, $t0, 1  # i++
b B1

end:

# $s0 == c, $t0 == i, $t1 == j, $t2 == 1000, $t3 == 4

ADD $s0, $zero, $zero   # initialize c to 0
ADD $t0, $zero, $zero   # initialize i to 0
ADD $t2, $zero, $zero   # store the end value of i in a register for
    comparison
ADDI $t2, $t2, 1000
ADD $t1, $zero, $zero   # initialize j to 0
ADD $t3, $zero, $zero   # store the end value of j in a register for
    comparison
ADDI $t3, $t3, 4

B1: BEQ $t0, $t2, end # loop 1 test
ADDI $t0, $t0, 1  # i++
ADD $t1, $zero, $zero # reinitialize j at 0

B2: BEQ $t1, $t3, B1  # loop 2 test
ADDI $s0, $s0, 1  # c++
ADDI $t1, $t1, 1  # j++
J B2

end:
```

Listing 1: MIPS.s

```
//Script to generate a list of branches and their outcome following the
    format above
//First column: the id or address of the branch
//Second column: the outcome of the branch
// 0 means the branch is not taken (0 = NT)
// 1 means the branch is taken (1 = NT)

#include <stdio.h>
```

```
8
9  int main() {
10   FILE *fptr;
11     fptr = fopen("vhdl_input.txt", "w"); //open the file to store the
     data
12
13     for(int i =0; i<1000; i++){
14       //fprintf(fptr, "1 T\n");
15       if(i != 0)  //first time in the loop, no branch
16           fprintf(fptr, "1 1\n");
17
18       for(int j=0; j<4; j++){
19         //fprintf(fptr, "2 T\n");
20         if (j != 0) //first time in the loop, no branch
21               fprintf(fptr, "2 1\n");
22       }
23
24       //fprintf(fptr, "2 NT\n");
25         fprintf(fptr, "2 0\n");
26     }
27
28     //fprintf(fptr, "1 NT\n");
29     fprintf(fptr, "1 0\n");
30
31   return 0;
32 }
```

Listing 2: branchCounter.c

```
1  //Script to generate a list of branches and their outcome following the
       format above
2  //First column: the id or address of the branch
3  //Second column: the outcome of the branch
4  // 0 means the branch is not taken (0 = NT)
5  // 1 means the branch is taken (1 = NT)
6
7  #include <stdio.h>
8
9  int main() {
10   FILE *fptr;
11     fptr = fopen("vhdl_input2.txt", "w"); //open the file to store the
     data
12
13     for(int i =0; i<1000; i++){
14       //fprintf(fptr, "1 T\n");
15       if(i != 0)  //first time in the loop, no branch
16           fprintf(fptr, "1 1\n");
17
18       for(int j=0; j<4; j++){
```

```
19          //fprintf(fptr, "2 T\n");
20          if (j != 0) //first time in the loop, no branch
21                fprintf(fptr, "2 1\n");
22
23              for(int k=0; k<20; k++){
24          //fprintf(fptr, "2 T\n");
25            if (k != 0) //first time in the loop, no branch
26                  fprintf(fptr, "3 1\n");
27          }
28          //fprintf(fptr, "2 NT\n");
29            fprintf(fptr, "3 0\n");
30        }
31
32        //fprintf(fptr, "2 NT\n");
33          fprintf(fptr, "2 0\n");
34      }
35
36      //fprintf(fptr, "1 NT\n");
37      fprintf(fptr, "1 0\n");
38
39    return 0;
40  }
```

Listing 3: branchCounter2.c

```
1  //Script to generate a list of branches and their outcome following the
       format above
2  //First column: the id or address of the branch
3  //Second column: the outcome of the branch
4  // 0 means the branch is not taken (0 = NT)
5  // 1 means the branch is taken (1 = NT)
6
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 int main() {
11   FILE *fptr;
12     fptr = fopen("vhdl_input3.txt", "w"); //open the file to store the
       data
13
14     for(int i =0; i<1000; i++){
15       //fprintf(fptr, "1 T\n");
16       if(i != 0)  //first time in the loop, no branch
17           fprintf(fptr, "1 1\n");
18
19         if(rand()%2 == 0) {
20             for(int j=0; j<4; j++){
21                 if (j != 0) //first time in the loop, no branch
22                     fprintf(fptr, "2 1\n");
```

```
            }
            fprintf(fptr, "2 0\n");
        }

    }

    fprintf(fptr, "1 0\n");

  return 0;
}
```

Listing 4: branchCounter3.c

```
//Script to generate a list of branches and their outcome following the format above
//First column: the id or address of the branch
//Second column: the outcome of the branch
// 0 means the branch is not taken (0 = NT)
// 1 means the branch is taken (1 = NT)

#include <stdio.h>

int main() {
  FILE *fptr;
    fptr = fopen("vhdl_input4.txt", "w"); //open the file to store the data

    for(int i =0; i<1000; i++){
      //fprintf(fptr, "1 T\n");
      if(i != 0)  //first time in the loop, no branch
          fprintf(fptr, "1 1\n");

      for(int j=0; j<4; j++){
        //fprintf(fptr, "2 T\n");
        if (j != 0) //first time in the loop, no branch
              fprintf(fptr, "2 1\n");

            for(int k=0; k<20; k++){
                if (k != 0) //first time in the loop, no branch
                    fprintf(fptr, "3 1\n");

                for(int l=0; l<20; l++){
                    if (l != 0) //first time in the loop, no branch
                        fprintf(fptr, "4 1\n");
                }
                fprintf(fptr, "4 0\n");
            }
            fprintf(fptr, "3 0\n");
      }
```

```
35
36          //fprintf(fptr, "2 NT\n");
37            fprintf(fptr, "2 0\n");
38        }
39
40        //fprintf(fptr, "1 NT\n");
41        fprintf(fptr, "1 0\n");
42
43      return 0;
44  }
```

Listing 5: branchCounter4.c

References

[1] Dr. E. Oruklu, *HDL Implementation of Dynamic Branch Predictors - Project Instructions*, v1.0, April 2017.

[2] J. L. Hennesy and D. A. Patterson *Computer Architecture, A Quantitative Approach*, 5th edition.