Relatório Laboratório 05

Maria Eduarda Teixeira Costa e Sofia Gazolla da Costa Silva

Departamento de Informática e Estatística - Universidade Federal de Santa Catarina

Organização de Computadores I

Professor Marcelo Daniel Berejuck

Setembro 2025

1 Introdução

Nesse quinto laboratório, a primeira atividade proposta era implementar dois códigos fornecidos em C em Assembly. Já para a segunda atividade proposta, o simulador de Branch History Table, do Mars, deveria ser utilizado para realizar diversas simulações para ambos os códigos, variando a quantidade de entradas, o tamanho da BHT e o valor inicial. Além disso, algumas considerações deviam ser feitas sobre as simulações.

2 Exercício 01

2.1 Desenvolvimento

A primeira atividade consistia na implementação em Assembly de dois códigos que nos foram previamente fornecidos, na linguagem C.

(a) O primeiro código pedia um valor limite ao usuário. Um contador iniciado em 0 verificaria a paridade dos números de 1 em 1 até o limite, imprimindo os números pares após juntamente com a frase "Contador: ".

Na implementação, declaramos na seção `.data` as mensagens de entrada e saída e uma string de quebra de linha.

No `main`, solicitamos e lemos o valor limite do usuário, armazenando-o em `$s0`. Além disso, inicializamos o contador (`count=0`) e a variável de iteração do loop `i` (registrador `$t0`) em zero.

No laço, a instrução `bge` verifica se `i` já atingiu o limite, encerrando o programa com `syscall 10` em `loop_fim` caso ele tenha atingido. O número 2 é carregado e a instrução `div` é usada para dividir `i` por 2. O resto é verificado com `mfhi` e a instrução `bne` desvia para o incremento de `i` se o resto for diferente de zero (ímpar). Se o resto for zero (par), o programa continua, incrementando o contador, imprimindo a frase `"Contador: "`, o número atual e uma quebra de linha.

(b) Já para o segundo código, o usuário insere um tamanho `x` para um vetor, seguido de `x` inteiros. Depois, insere um número a ser procurado. A saída será `"Numero encontrado"` ou `"Numero nao encontrado."`.

Inicialmente, mensagens de entrada/saída e a variável `array_label`, que aloca espaço para até 100 inteiros na memória, foram declaradas em `.data`. No `main`, solicitamos o tamanho do vetor, armazenando esse valor. Um `jal` para `inicio_loop_leitura` é feito para o loop onde o usuário preenche o vetor.

No laço de leitura, um contador `i` é inicializado em 0 e o endereço base do array é carregado. O laço compara `i` com o tamanho do vetor. Se não for igual, a `syscall` lê o inteiro do usuário. Um `sll` com 2 deslocamentos é feito para lidar com bytes. O endereço base é somado ao deslocamento e `sw` armazena o valor no array. O `jump` repete o loop.

Após a leitura, a instrução `bge` leva a `leitura_loop_fim`, retornando a execução ao `main`.

De volta ao `main`, solicitamos o número a ser procurado, que é armazenado em um registrador, seguido por um `jal` para `inicio_loop_busca`. Nesta função, a flag `found` é inicializada (seu valor será 0 se o número não for encontrado e 1 se for), e um novo contador `i` é inicializado para este laço.

No laço de busca, compara-se o contador `i` ao tamanho do vetor. Se não forem iguais, o deslocamento é feito para acessar os elementos, utilizando `lw`. O elemento carregado é comparado ao valor a ser procurado (`$a1`). Se forem iguais, o programa

salta para a função `encontrado`. Caso não sejam, o programa continua. Se o loop terminar sem encontrar, o `bge` inicial leva a `fim_loop_busca`.

Se o número for encontrado, a função `encontrado` atribui o valor 1 à flag `found`, e o programa segue para `fim_loop_busca`.

Em `fim_loop_busca`, uma instrução `beqz` verifica se `found` é 0. Se for, o programa desvia para `nao_encontrado`, onde a string `"Numero nao encontrado."` é impressa, e o programa é encerrado. Caso `found` seja 1, o desvio não ocorre, a string `"Numero encontrado"` é impressa, e o programa retorna ao `main` com `jr $ra`, onde ele é encerrado.

2.2 Resultados:

Os resultados obtidos para todos os testes de ambos os códigos foram corretos, não havendo nenhum erro.

(a) **Primeiro programa:**

```
Digite o limite do contador: 5
Contador: 0
Contador: 2
Contador: 4

-- program is finished running --
```

Figure 1: Teste com 5

```
Digite o limite do contador: 10
Contador: 0
Contador: 2
Contador: 4
Contador: 6
Contador: 8

-- program is finished running --
```

Figure 2: Teste com 10

```
Digite o limite do contador: 1010
Contador: 0
Contador: 2
Contador: 4
Contador: 6
Contador: 8
Contador: 10
Contador: 12
Contador: 14
```

```
Contador: 1000
Contador: 1002
Contador: 1004
Contador: 1006
Contador: 1008

-- program is finished running --
```

Figure 3: Parte do teste com 1010

(b) **Segundo programa:**

```
Digite o tamanho do vetor: 5
Digite 5 numeros para o vetor:
50
28
403
3
10
Digite o numero a procurar: 11
Numero nao encontrado.

-- program is finished running --
```

```
Digite o tamanho do vetor: 5
Digite 5 numeros para o vetor:
50
28
403
3
10
Digite o numero a procurar: 403
Numero encontrado

-- program is finished running --
```

Figure 4: Vetor de 5 elementos

```
Digite o tamanho do vetor: 10
Digite 10 numeros para o vetor:
50
3
26
19
32
98
100
14
5
267
Digite o numero a procurar: 98
Numero encontrado

-- program is finished running --
```

```
Digite o tamanho do vetor: 10
Digite 10 numeros para o vetor:
50
3
26
19
32
98
100
14
5
267
Digite o numero a procurar: 268
Numero nao encontrado.

-- program is finished running --
```

Figure 5: Vetor de 10 elementos

3 Exercício 02

Para o segundo exercício do laboratório, que envolvia o uso da ferramenta BHT simulator nos códigos que desenvolvemos na primeira parte, realizamos diversas simulações, para ambos os códigos, variando o número de entradas do BHT, o tamanho do BHT History e o valor inicial do programa, como solicitado.

No primeiro código, haviam duas instruções de desvio condicional e, no segundo, haviam quatro. Entretanto, cinco dessas seis instruções estão dentro de laços, então o número de vezes que elas se repetem depende da entrada inserida pelo usuário. Para as análises feitas a seguir, foram utilizados o primeiro programa tendo como limite o número 10 e um vetor de 10 elementos para o segundo programa, como foi testado na seção de Resultados, para o exercício anterior. Em um geral, o número ser encontrado no vetor ou não, não foi um parâmetro levado em consideração na

análise do desempenho do programa, pois isso depende exclusivamente da entrada do usuário e não poderia ser otimizado no desenvolvimento do programa.

3.1 Diferentes abordagens de BHT

Sobre o parâmetro de quantidade de entradas da BHT, nossas opções eram 8, 16 e 32. Como não haviam muitas instruções a serem mapeadas, o número de entradas não tinha diferença nesse sentido, mas, mesmo nesses casos, é importante testá-los para evitar que ocorresse conflito (aliasing). Essa situação acontece quando duas instruções de desvio diferentes são mapeadas para o mesmo slot na tabela e sobrescrevem o histórico de previsão uma da outra, diminuindo a precisão da previsão. Embora haja menos chances de isso ocorrer com menos instruções branch, quanto menos entradas BHT houverem, mais provável é de isso acontecer. Entretanto, para todos os testes que realizamos, isso não ocorreu nenhuma vez, com o número das entradas não influenciando de maneira alguma os nossos resultados.

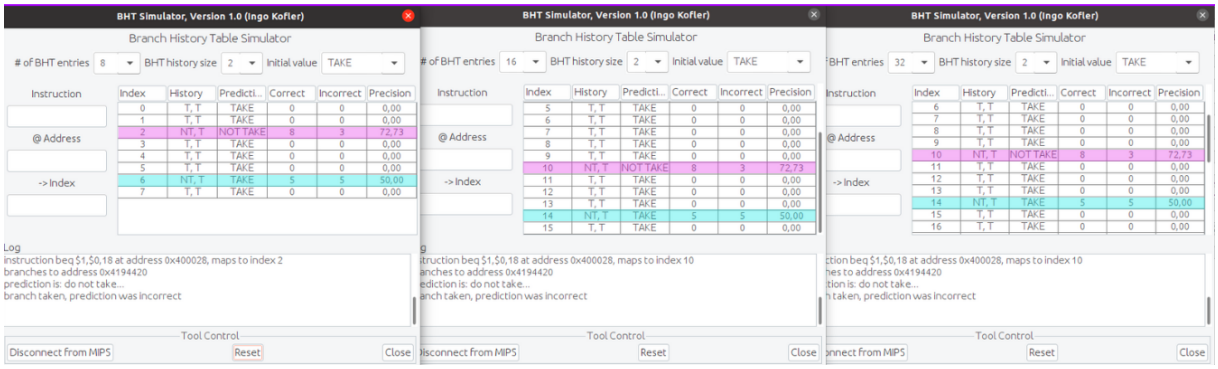


Figure 6: Entradas diferentes com os mesmos parâmetros para o primeiro programa

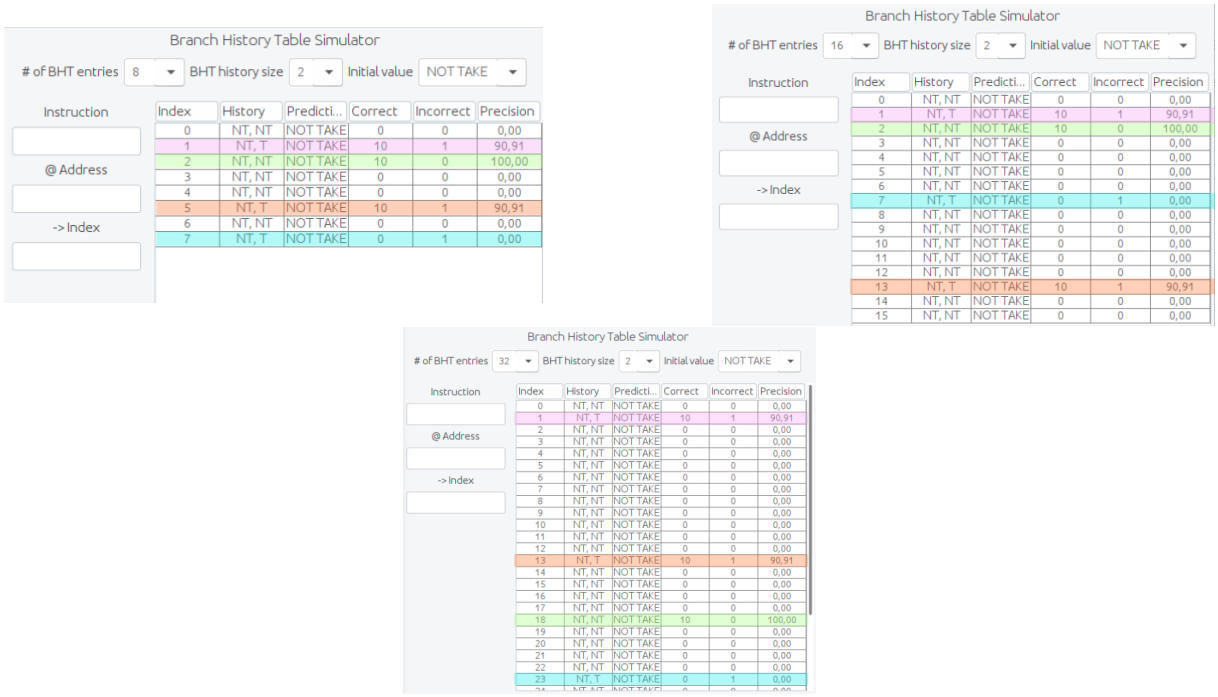


Figure 7: Entradas diferentes com os mesmos parâmetros para o segundo programa

Uma grande diferença pode ser notada, entretanto, ao alterar os valores de BHT History Size. Só haviam duas opções para esse parâmetro, elas sendo o preditor de 1 e o de 2 bits. O preditor de 1 bit armazena apenas o resultado do último desvio, ou seja a predição para o próximo desvio é simplesmente a mesma do último resultado. Se o último desvio foi `TAKE`, ele irá prever `TAKE` novamente. A ineficácia desse preditor pôde ser percebida principalmente para o primeiro programa, onde um dos desvios era alternado: acontecia em uma iteração e, na seguinte, não. Nesse caso, o primeiro valor do desvio era sempre o mesmo (não desvia), então se o programa iniciasse com `NOT TAKE`, ele acertaria 1 dos desvios. Caso ele iniciasse com o `TAKE`, todas as vezes, sem exceção, ele teria uma precisão de 0%.

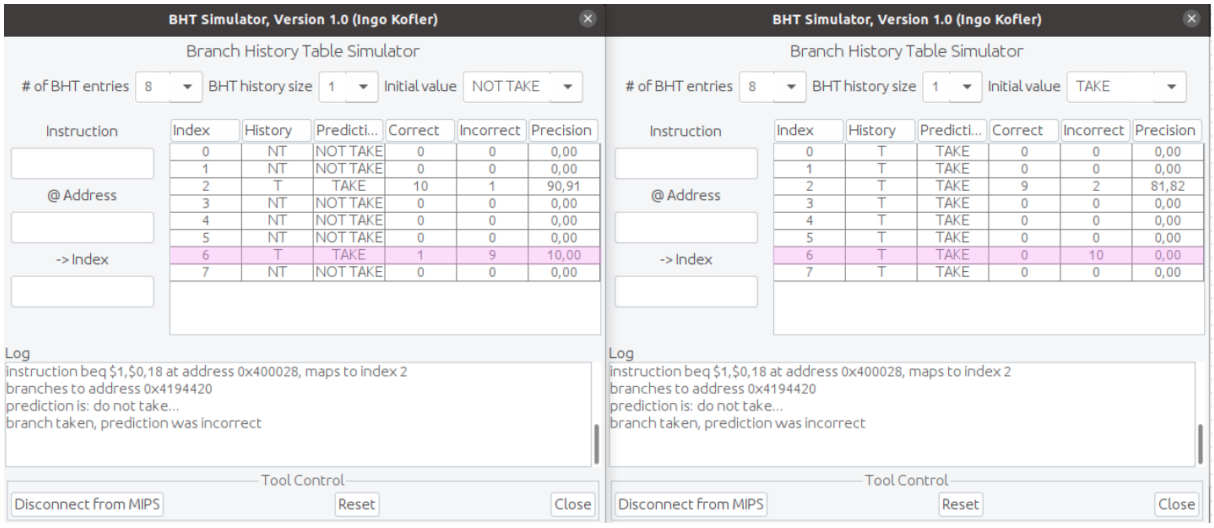


Figure 8: Preditor de 1 bit

Já para o preditor de 2 bits, como há 4 possíveis combinações de bits, podemos ter um resultado mais preciso. A combinação 00 significa que há grandes chances de o desvio não ocorrer, 01 que há chances de o desvio não ocorrer, 10 significa que há chances de o desvio ocorrer e 11 que há grandes chances de o desvio ocorrer. O preditor se atualiza toda vez que um branch é executado, pois se ele é tomado, o contador incrementa 1 e, se ele não é tomado, ele decrementa 1. Os valores nunca ultrapassam do limite, mas se ele for 11 e um desvio ocorrer, ele seguirá em 11, mostrando que o padrão que está sendo seguido é os desvios ocorrendo. Se ele for 11 e o desvio não ocorrer, entretanto, o valor descerá para 10, indicando que, se o próximo desvio também não ocorrer, a predição será trocada de `TAKE` para `NOT TAKE`.

Para o primeiro código, por exemplo, onde a ocorrência dos desvios é alternada, há uma precisão de 50%, pois os valores do BHT History ficam alternando entre 01 e 10. Por conta disso, nota-se resultados muito mais precisos quando se utiliza o preditor de 2 bits. Apesar de uma taxa de 50% não ser ideal, é muito melhor que uma taxa de 0%.

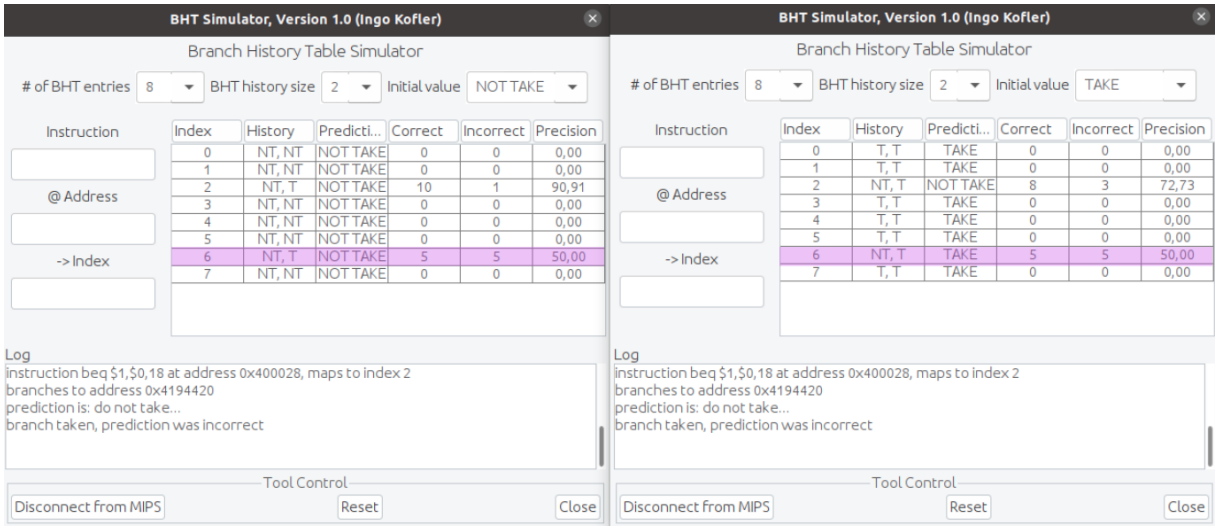


Figure 9: Preditor de 2 bits

Entretanto, no caso do segundo programa, que representa um valor de desvio se mantendo o mesmo por muito tempo, já que cada desvio só acontece uma ou nenhuma vez, quando o valor inicial do preditor é o errado (começa com `TAKE` mas o desvio só irá ocorrer uma vez, no final do programa, por exemplo), o preditor de 1 bit irá ter uma maior precisão, pois ele irá demorar menos tempo para ir ao valor correto. Com 1 bit, haverá um erro e o valor irá alterar de 1 para 0, nesse caso. Já se tivermos 2 bits, ele irá começar em 11, errar uma vez, ir para 10, errando novamente e somente aí o valor irá para 01 e ser `NOT TAKE`, tendo uma precisão menor que quando o programa já altera a previsão na próxima execução, já que ele demora duas execuções ao invés de uma para trocar de "opinião".

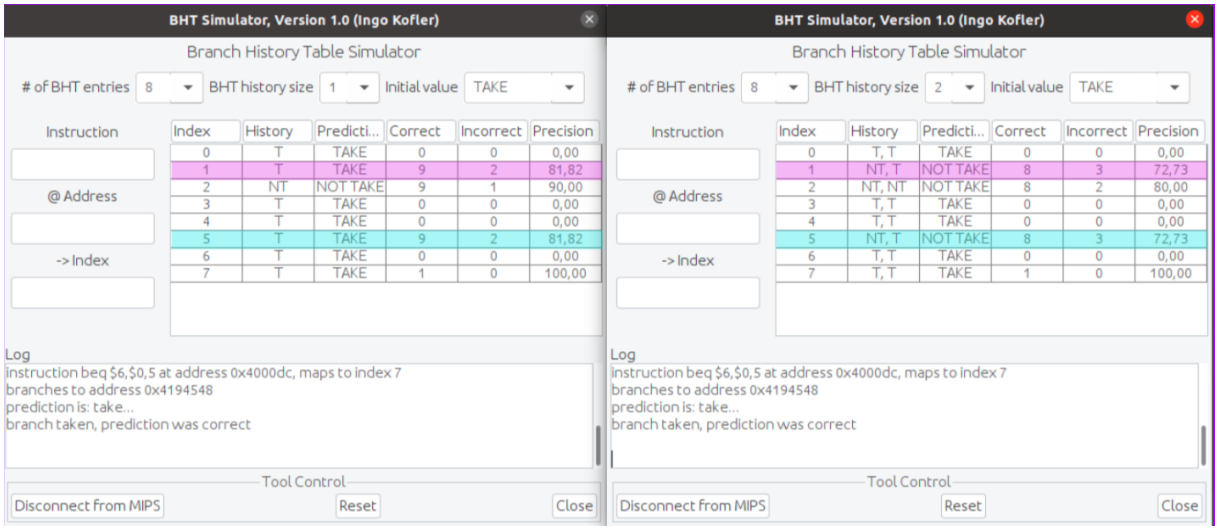


Figure 10: Comparação de preditores com valor inicial TAKE

No caso de começarmos com o valor correto (`NOT TAKE`), ambos os preditores tem exatamente a mesma precisão, pois ambos iniciam acertando e seguem nesse padrão e ambos irão errar quando o desvio de fato ocorrer.

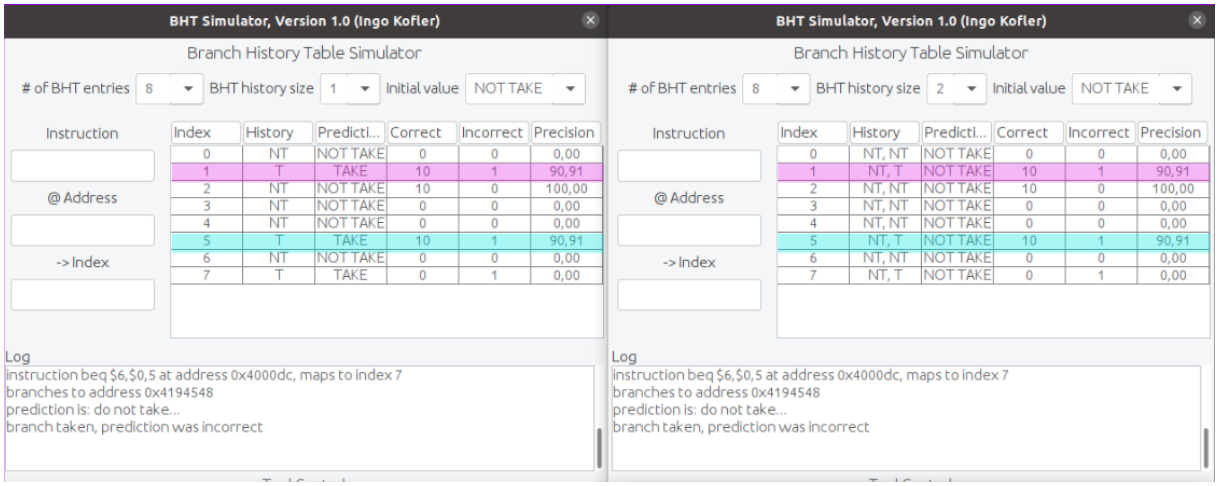


Figure 11: Comparação de preditores com valor inicial NOT TAKE

Finalmente, o último parâmetro que podíamos alterar, o valor inicial, também influenciou significativamente as simulações. Isso se deve ao fato de o valor inicial ter um alto impacto nas predições iniciais, enquanto o preditor ainda está sendo treinado, no caso do preditor de 2 bits. Já para o preditor de 1 bit, ele tem um impacto maior ainda, como visto anteriormente, no caso representado na figura 8. Para a outra instrução de desvio do primeiro programa, há 10 situações em que a instrução não desvia e o desvio ocorre somente na 11ª execução da instrução. Por conta disso, quando o valor inicial é `NOT TAKE`, há uma taxa muito maior de acerto, como o programa mantêm esse padrão durante toda a execução e erra somente na última, quando o desvio ocorre. Já quando o valor inicial é `TAKE` (11), ele erra uma vez, colocando o valor em 10 (ainda é `TAKE`) e mais uma vez, baixando o valor para 01 (`NOT TAKE`) e, aí, começa a acertar as predições. Isso exemplifica muito bem o impacto do valor inicial no treinamento do preditor de 2 bits.

Quando testamos os diferentes parâmetros para o valor inicial para o segundo programa, o resultado foi similar ao da segunda instrução de desvio do primeiro programa, que é percorrida diversas vezes mas só desvia em uma delas. Se o programa iniciava com o valor correto (`NOT TAKE`), a taxa de acerto era maior. Já se ele iniciava com o valor incorreto (`TAKE`), a taxa de acerto diminuia um pouco (no caso do nosso

teste em aproximadamente 10%, como estávamos lidando com um vetor de tamanho 10). Já para a instrução que ocorria apenas 1 vez, que verificava se o número foi encontrado ou não e que não está em um loop, o preditor tem uma taxa de 100% de acerto ou erro, que depende completamente do input do usuário.

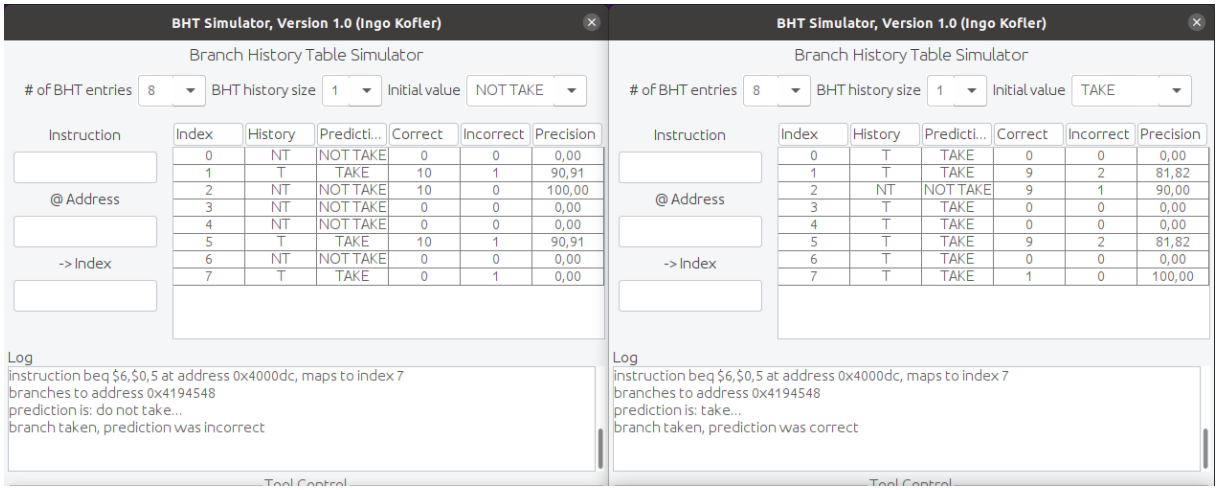


Figure 12: Diferentes valores de entrada

3.2 Pontos críticos

Para identificar os pontos críticos de cada programa, analisamos as simulações realizadas para achar a instrução de desvio que mais "confundia" o processador e, portanto, gerava uma maior taxa de erro.

Para o primeiro programa, a área que percebemos como mais crítica é a que realiza a verificação de paridade do número. Isso se deve ao fato de ela gerar um padrão para as instruções branch que é muito difícil para o processador acertar. Como vimos anteriormente, para o preditor de 1 bit, se o valor inicial for o errado, ele não vai acertar absolutamente nenhuma vez. Para o preditor de 2 bits, a taxa está em 50%, o que é melhor, mas continua muito baixo.

```
loop_for:

        bge     $t0, $s0, loop_fim

        li      $t1, 2
        div     $t0, $t1
        mfhi    $t2

        bne     $t2, $zero, incremento_i
```

Figure 13: Ponto crítico do primeiro programa

Já para o segundo programa, a área mais difícil de prever era a que identificava se o número havia sido encontrado, comparando-o ao número atual. Essa é a área mais crítica pois não existe um padrão identificável para a sua execução, visto que o seu resultado depende completamente da entrada do usuário e da posição onde o número está, se ele sequer estiver no vetor.

```
loop_busca:

        bge     $t0, $s0, fim_loop_busca

        mul     $t2, $t0, 4
        add     $t3, $t1, $t2
        lw      $t4, 0($t3)

        beq     $t4, $a1, encontrado

        addi    $t0, $t0, 1

        j       loop_busca
```

Figure 14: Ponto crítico do segundo programa

4 Otimização dos programas

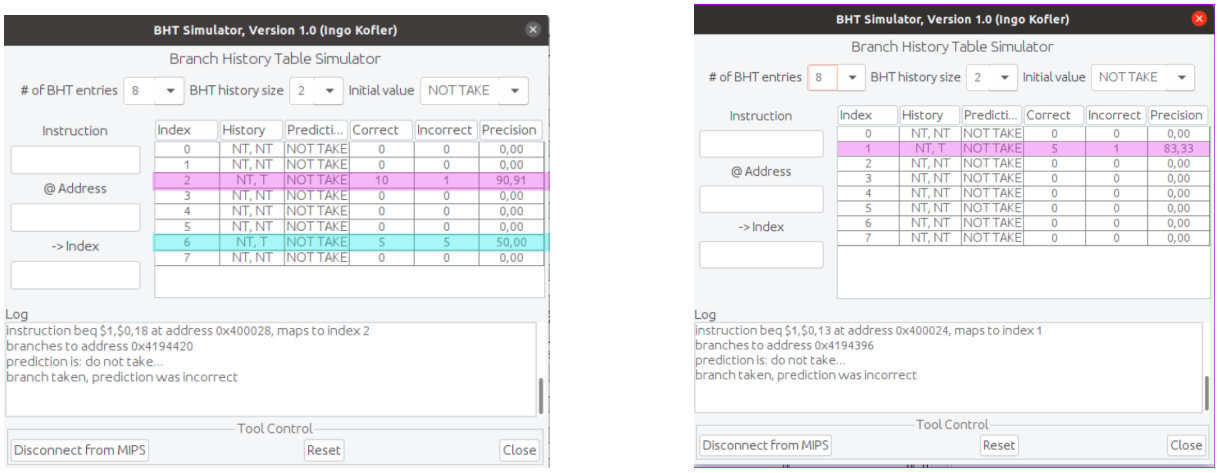
Após analisar os pontos que mais causam erros de predição nos programas, fizemos alterações nos dois para tentar otimizá-los.

Para otimizar o primeiro programa, há uma abordagem muito mais simples para o objetivo do programa, que reduz o número de instruções de desvio de duas para uma.

Para imprimir todos os números pares de 0 até o limite estabelecido pelo usuário, podemos simplesmente criar um laço que inicia em zero e realiza incremento de dois em dois, removendo completamente a verificação da paridade dos números. Sobre a instrução branch restante, que verifica se todo o loop já foi percorrido, nenhuma alteração foi realizada, pois, da maneira que ela ocorre, o desvio vai acontecer apenas uma vez, ao contador atingir o limite definido pelo usuário.

Mesmo com apenas uma instrução sendo alterada e, nesse caso, removida, como era ela quem estava causando uma redução significativa na precisão, a sua remoção otimiza o programa em um geral, levando a uma taxa maior de precisão se a média for feita entre as orecisões das duas instruções. Analisando o parâmetro sozinho para o mesmo valor utilizado nos outros testes, que é 10, a taxa de precisão é um

pouco menor, por haverem menos iterações do laço, mas, como foi mencionado, se considerarmos ambas as precisões da primeira versão, essa segunda continua sendo melhor. O registrador que fazia o papel da variável count também foi removido, pois a variável não impactava em nada o programa.



(a) Resultados anteriores

(b) Resultados otimizados

Figure 15: Comparação entre os resultados anteriores e otimizados.

Para otimizar o segundo programa, alteramos o que tínhamos identificado como a área crítica, `loop_busca`, que tem duas das instruções de branch do programa. A primeira tem uma precisão muito boa, pois é a instrução que verifica se o vetor todo foi percorrido. O problema está na segunda instrução, que identifica se o número procurado foi encontrado ou não, pois ela depende muito da posição que o número está no vetor ou se ele está no vetor mesmo.

Essa instrução era um `beq` que comparava o número atual com o número que queríamos encontrar e, se fosse ele, levava o programa para a função `encontrado`, que estabelecia o valor 1 à variável `found`, indicando que o número havia sido encontrado. Para otimizar o código, trocamos essa instrução por um `seq`, que, ao invés de ir para `encontrado` e definir `found ($a2)` como 1 lá, define a flag como 1 ali mesmo, no caso de o conteúdo dos dois registradores ser igual, indicando que encontramos o número. Usamos `$t5` como uma variável temporária, para armazenar o resultado de `seq` e, após isso, é feito um `or` desse resultado com o valor anterior de `$a2`, para garantir que, se em qualquer momento o valor da flag passar a ser 1, ele continue sendo 1. O resultado do `or` é armazenado em `$a2`, garantindo que esteja tudo pronto para a próxima iteração do loop. Com essa simples substituição, a predição que era mais

difícil para o processador realizar é removida, aumentando significamente a precisão geral do programa. Após essa alteração, também deletamos a função `encontrado`, que não seria mais utilizada nessa nova implementação do programa.

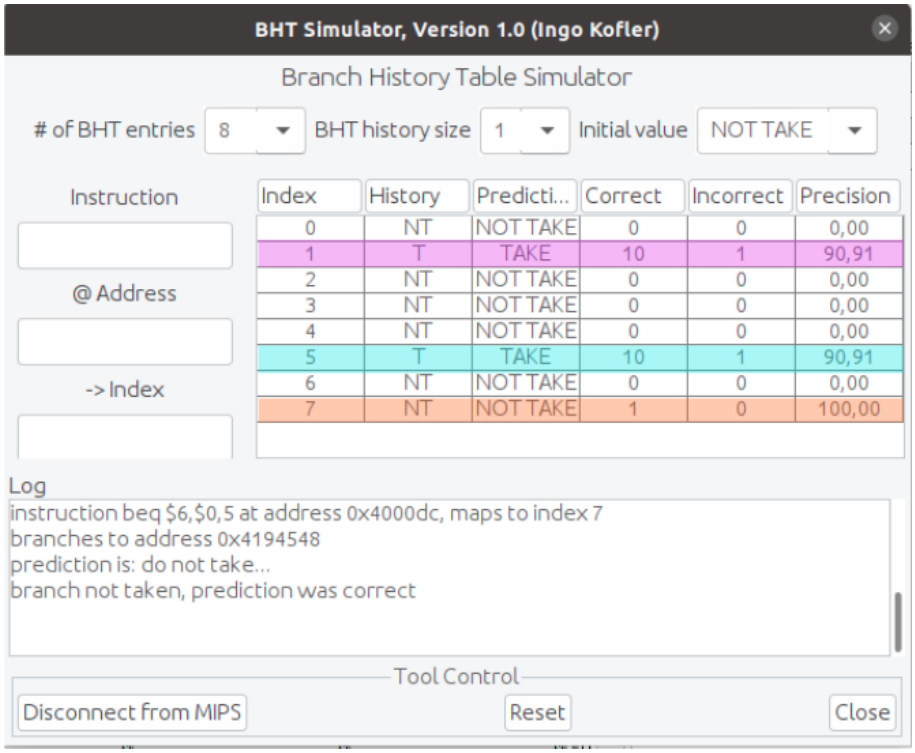


Figure 16: Caso otimizado

Nesse caso específico, a instrução mapeada tem uma taxa de 100% de acerto pois o número foi encontrado e o desvio de `beqz` que verifica se a flag é zero não ocorreu.

5 Materiais e métodos

Para o desenvolvimento dessa atividade foi utilizada a linguagem `Assembly` e a IDE `MARS`, na qual foi realizada toda a escrita, execução e depuração de todos os programas implementados. Além disso, foi utilizada a ferramenta Branch History Table Simulator (BHT Simulator), que faz parte da IDE `MARS`.

6 Dificuldades

As principais dificuldades enfrentadas nesse laboratório foram relacionadas ao segundo exercício, que envolvia o uso da ferramenta BHT simulator, do Mars. Apesar

de não ser uma ferramenta necessariamente complexa, inicialmente foi difícil entender como ela funcionava e como deveríamos interpretar os seus resultados. Entretanto, após pesquisar sobre e realizar diversas simulações alterando os parâmetros, ficou mais simples de entender o que exatamente deveríamos fazer, qual era a importância da ferramenta e como os seus resultados devem ser utilizados para entender melhor o fluxo do nosso programa e, possivelmente, otimizá-lo.

A primeira atividade foi relativamente simples, pois o código em C fornecia uma base muito boa de como realizar a implementação em Assembly. O segundo código a ser implementado foi um pouco mais complicado pois envolvia o uso de arrays e cálculo de deslocamento, mas também não tivemos problemas para realizá-lo.

7 Conclusão

A realização desse laboratório foi muito boa, não necessariamente para fazer novas descobertas e praticar algo diferente em Assembly, mas sim para conhecer a ferramenta BHT Simulator do Mars e entender a grande ajuda que ela pode fornecer quando programando em linguagem de montagem. Uma parte de como a ferramenta auxilia tanto é porque ela permite que entendamos melhor como o processador funciona e segue o seu caminho pelo programa. Além disso, ela ajuda a saber mais sobre a eficiência do nosso código, permitindo que saibamos quais as partes que estão causando uma maior dificuldade de predição nele e, portanto, podem ser otimizadas, permitindo a elaboração de um programa mais eficaz.