Архитектура вычислительных систем. Домашнее задание №3. Вариант №1.

Комиссаров Данил Андреевич

March 2025

1 Полный отчет

Первый вариант задания. Программа на RISC-V ассемблере.

Здесь будем работать с инструкциями ассемблера RISC-V. Теперь не придётся заниматься проектированием железа, но разбираться с устройством работы моделей предсказаний. Что само по себе является достаточно обширной темой. Предисловия не будет, со всеми темами будем разбираться на местах.

Это должно быть интересно.

Пункт 1

Выдан код на ассемблере

```
1  .text
2  main:
3      li      t0, 10000
4      li      a3, 1
5      li      a4, 0
6      li      a1, 0
7  loop:
8      beq     a4, zero, magic_br_1 # branch #1
9      addi    a4, a4, 0
10 magic_br_1:
11     beq     a3, zero, magic_br_2 # branch #2
12     addi    a3, a3, 0
13 magic_br_2:
14
15 # ****** ADD HERE ******
16 # your code for task 4
17 # **********************
18
19     addi    a1, a1, 1
20     bne     a1, t0, loop
21
```

Удобно, что при наведении на команду подсвечивается её значение:

1. **li t0, 10000** - Load Immediate - загрузить 10000 в t0, то есть просто запись констант в регистры
2. **beq a4, zero, magic_br_1** - Branch if EQual - прыгнуть по метке magic_br_1 если a4 == zero (то есть равенство регистра a4 нулю), то есть команда означает, если первый операнд равен второму, то происходит прыжок по метке, записанной третьим параметром.

3. **addi a4, a4, 0** - ADDiotion Immediate - то есть назначить первому операнду значению, равное сумме второго и третьего, вторым операндом может быть знаковый регистр, а третьим immediate (то есть так называемый непосредственный, далее все immediate будем называть *константы*. Их значения доступны непосредственно из инструкции и не требуют доступа к регистру или памяти).

4. **nop** - No OPeration - операция, которая заставляет просто стоять в холостую данный такт. Иногда заменяется на

   $$addi\ a0,\ a0,\ 0$$

   то есть инструкция-затычка.

5. **bne a1, t0, loop** - Branch if Not Equal - то же самое, что и *beq*, но вместо условия равенства, стоит условие неравенства.

Вроде все понятно, конечно, этот материал пересекается с вторым семестром информатики и изучения ассемблера x86, поэтому особых вопросов возникнуть не должно.

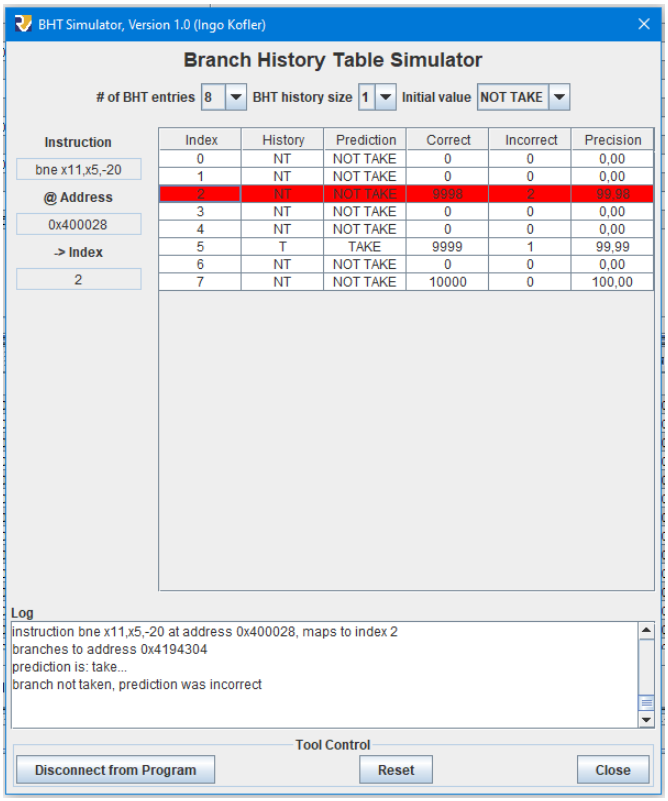
Теперь можно пристальнее присмотреться к коду и заметить, что он выполняет довольно странные действия (впрочем, очевидно, что такой искусственный код написан только для оперирования с модулем предсказания).

А в частности то, что цикл **loop** выполняется 10000 раз, внутри которого не меняется ни одна переменная. Переписывая этот код на C, можно представить его так:

```
1   int main() {
2       int t0 = 10000;
3       int a3 = 1;
4       int a4 = 0;
5       int a1 = 0;
6
7   loop:
8       if (a4 == 0)
9           goto magic_br_1;
10      addi_a4: a4 = a4 + 0;
11
12  magic_br_1:
13      if (a3 == 0)
14          goto magic_br_2;
15      addi_a3: a3 = a3 + 0;
16
17  magic_br_2:
18      a1++;
19      if (a1 != t0) goto loop;
20
21      return 0;
22  }
23
```

Понятно, что от этого кода ожидается только вызывать реакцию predict-модуля.

В задании требуется определить точность предсказаний для перехода по magic_br_1 и magic_br_2. Делая все по инструкции, получаем следующую таблицу:



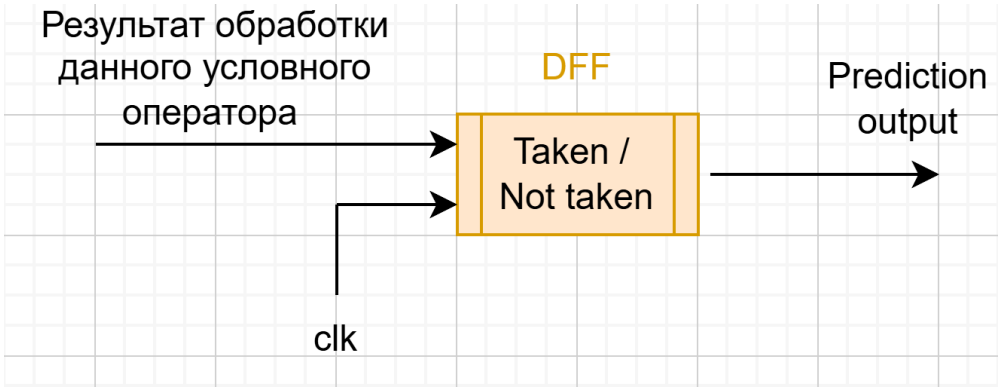
Интересненько. В задании требуется определить точность предсказаний, но здесь не подписано, какой индекс предиктора соответствует тому или иному условному оператору. Судя по всему, в этом и заключается сложность этого пункта: в сопоставлении этих параметров.

Можно пойти простым путем и посмотреть в окошке логов, какой адрес инструкции соответствует какому индексу. Или же прокрутить этот код в голове и догадаться, что если значение по-умолчанию - NOT TAKE, то очевидно, что 2 раза угадать неправильно можно было только цикл loop: при входе в него и при выходе, одной ошибке соответствует условный оператор, связанный с меткой magic_br_1, и соответственно определить последний условный оператор. Но это слишком просто) Предлагаю посмотреть на эту задачу с другой стороны.

Что такое вообще модуль предсказания?

Очевидно, предсказывает результат условного оператора. Это очень важно для оптимизации работы конвейерных процессоров, сейчас не могу сильно вдаваться в это, и так опаздываю сдать отчет =).

Другой вопрос: как реализовать?



Суть конечного автомата только в том, чтобы сохранить текущее состояние, поэтому его реализацией являются любые ячейки памяти, сохраняющие n бит, в которых зашифровано

состояние автомата. В самом простом случае для каждого состояния назначен свой DFF, который сохраняет 1, если автомат работает в этом режиме. Закручено написал, однако...

Итак. То есть модуль предсказания - это элемент оптимизации системы на уровне железа, а не на уровне программ. Понятно.

Разобрались. Далее: для каждого условного оператора нужно выделить предиктор, но так как это *hardware* оптимизация, то динамически выделить на каждый условный оператор не выйдет: сколько на заводе спаяли модулей предсказания, столько их и будет, что, очевидно, несравнимо меньше, чем условных операторов даже в самой простой программе.

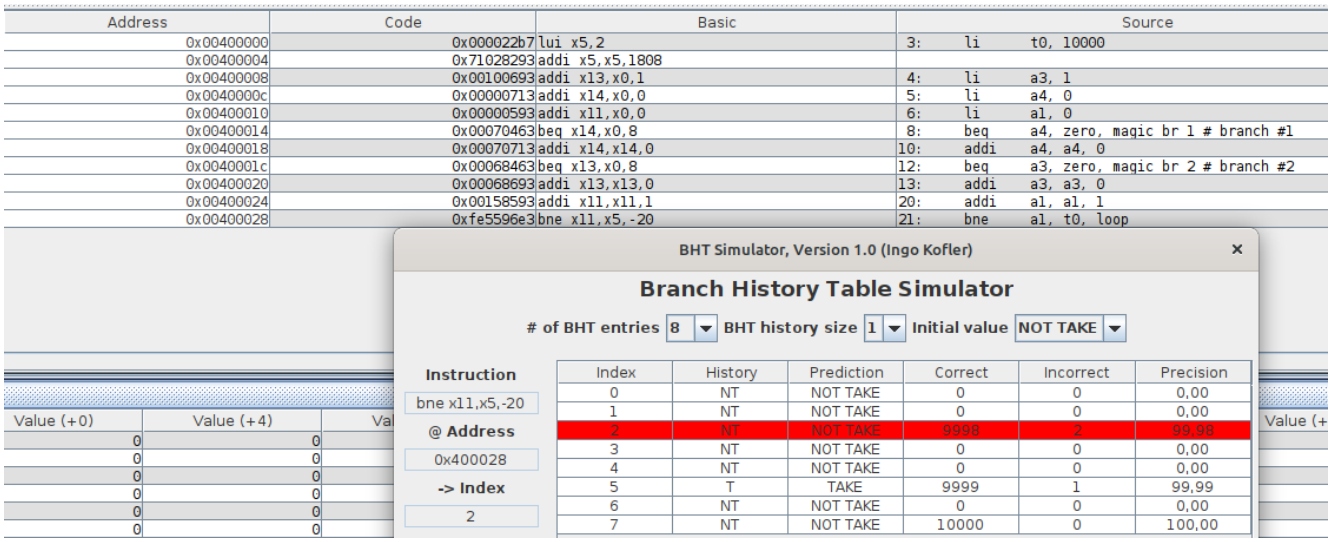
Поэтому следует их как-то распределять по нескольким операторам на модуль, желательно с наименьшей коллизией. И самый простой способ - это распределение по адресу инструкции, то есть

индекс модуля = (*номер инструкции* / *размер инструкции*) (mod) *количество модулей*.

Собственно, это можно и проследить в RARS.

Выявим принцип распределения модулей

Возьмем неизменнеую программу из задания и прокомпилируем:



Address	Code	Basic	Source
0x00400000	0x000022b7	lui x5,2	3: li t0, 10000
0x00400004	0x71028293	addi x5,x5,1808	
0x00400008	0x00100693	addi x13,x0,1	4: li a3, 1
0x0040000c	0x00000713	addi x14,x0,0	5: li a4, 0
0x00400010	0x00000593	addi x11,x0,0	6: li a1, 0
0x00400014	0x00070463	beq x14,x0,8	8: beq a4, zero, magic br 1 # branch #1
0x00400018	0x00070713	addi x14,x14,0	10: addi a4, a4, 0
0x0040001c	0x00068463	beq x13,x0,8	12: beq a3, zero, magic br 2 # branch #2
0x00400020	0x00068693	addi x13,x13,0	13: addi a3, a3, 0
0x00400024	0x00158593	addi x11,x11,1	20: addi a1, a1, 1
0x00400028	0xfe5596e3	bne x11,x5,-20	21: bne a1, t0, loop

BHT Simulator, Version 1.0 (Ingo Kofler)

Branch History Table Simulator

# of BHT entries 8 | BHT history size 1 | Initial value NOT TAKE

Instruction: bne x11,x5,-20

@ Address: 0x400028

-> Index: 2

Index	History	Prediction	Correct	Incorrect	Precision
0	NT	NOT TAKE	0	0	0,00
1	NT	NOT TAKE	0	0	0,00
2	NT	NOT TAKE	9998	2	99,98
3	NT	NOT TAKE	0	0	0,00
4	NT	NOT TAKE	0	0	0,00
5	T	TAKE	9999	1	99,99
6	NT	NOT TAKE	0	0	0,00
7	NT	NOT TAKE	10000	0	100,00

Value (+0)	Value (+4)
0	0
0	0
0	0
0	0
0	0
0	0

Рис. 1: Результат отработки неизмененной программы

То есть, проверим первый **beq**, который встречается в коде: он имеет адрес *0x00400014*, длина инструкции составляет *4* байта, а количество модулей *8*:

$$0x00400014/0x00000004 = 0x00100005$$

$$0x00100005(mod)0x00000008 = 0x00000005$$

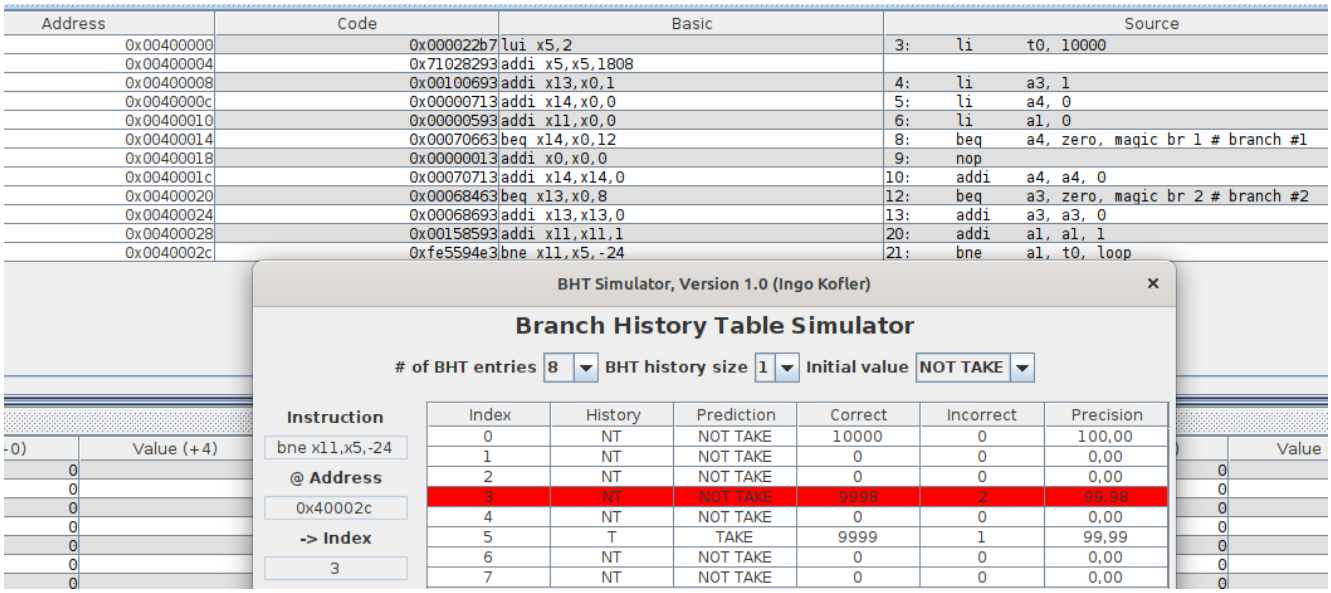
Вот как получается. Проводя соответствующие вычисления приходим к выводу, что второму **beq** соответствует индекс **7**, а инструкции **bne** - индекс **2**.

Впрочем. Это ведь всего лишь предположение, что индексы распределяются именно так, может нам просто повезло: но и это можно проверить. Достаточно лишь попередвигать адреса инструкций условных операторов: для того чтобы передвигать адреса, не ломая всю остальную программу идеально подходят, показавшиеся на первый взгляд бесполезными, инструкции **nop**.

```
1   .text
2   main:
3       li      t0, 10000
4       li      a3, 1
5       li      a4, 0
6       li      a1, 0
7   loop:
8       beq     a4, zero, magic_br_1 # branch #1
9       nop
10      addi    a4, a4, 0
11  magic_br_1:
12      beq     a3, zero, magic_br_2 # branch #2
13      addi    a3, a3, 0
14  magic_br_2:
15
16  # ****** ADD HERE ******
17  # your code for task 4
18  # **********************
19
20      addi    a1, a1, 1
21      bne     a1, t0, loop
```

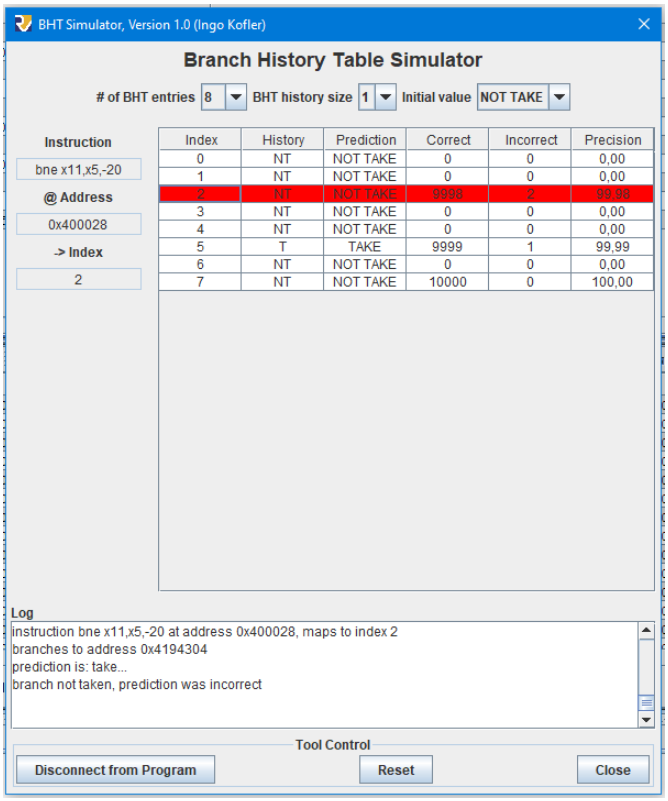
Рис. 2: Один сдвиг инструкций

Вставили один **nop**, по предположению, последние два условных оператора должны быть назначены модулям с увеличенным на единицу номером, а первый останется на том же месте.



Действительно. Так и есть, индексы сдвинулись циклически. Добавляя и другие инструкции и в другие строки, теория подтверждается.

Значит, что теперь можем наконец-то с чистой душой смело ответить:
Точность **bagic_br_1** составляет 99,99%.
Точность **bagic_br_2** составляет 100%.

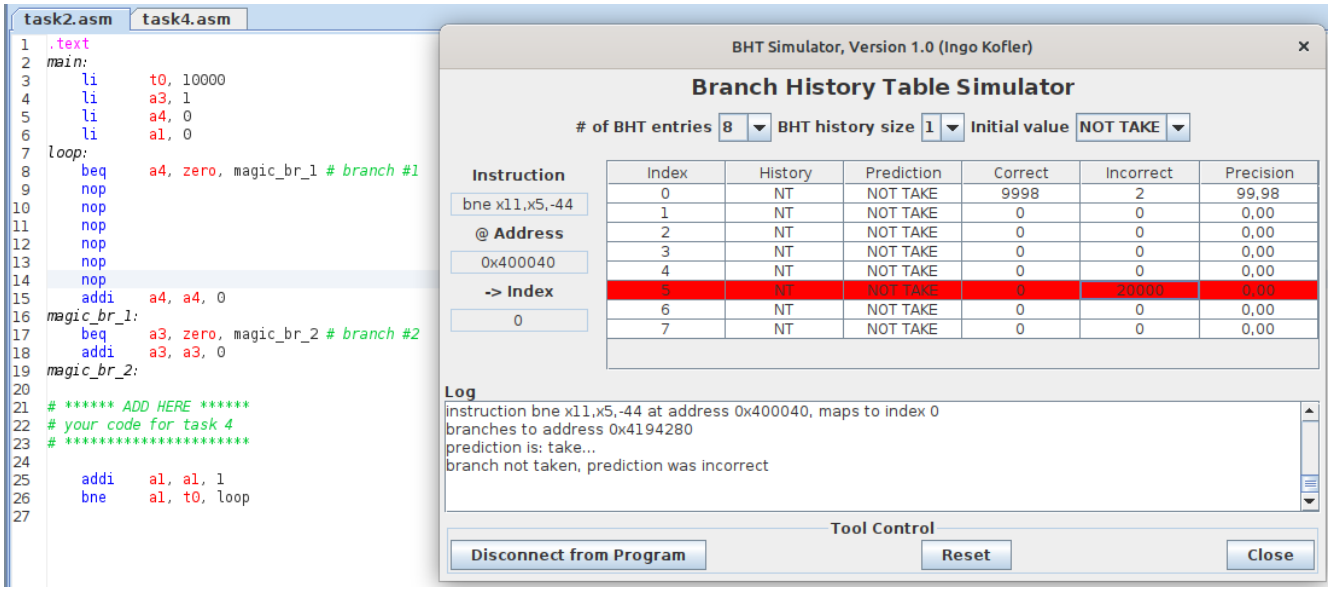


Пункт 2

Теперь уже требуется от нас обмануть модуль предсказания. Сейчас в настройках установлен автомат с двумя состояниями, он плохо справляется со случаем, когда результат условного оператора меняется через одного, попеременно выдавая результаты *taken* - *not taken*.

Тогда имеет смысл совместить модули обоих **beq**, тем более, что уже ранее было выяснено, что первый всегда выдает *taken*, а второй - *not taken*, главное, чтобы сюда не попал третий оператор.

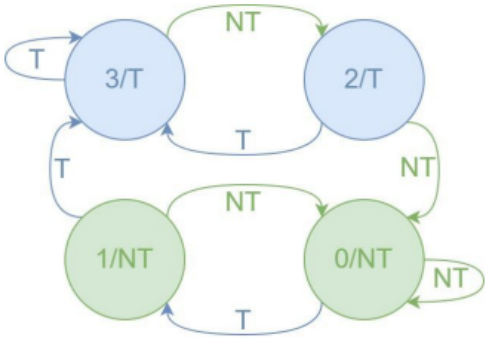
Так и подгоняем их адреса:



Впрочем, ничего сложного.

Пункт 3

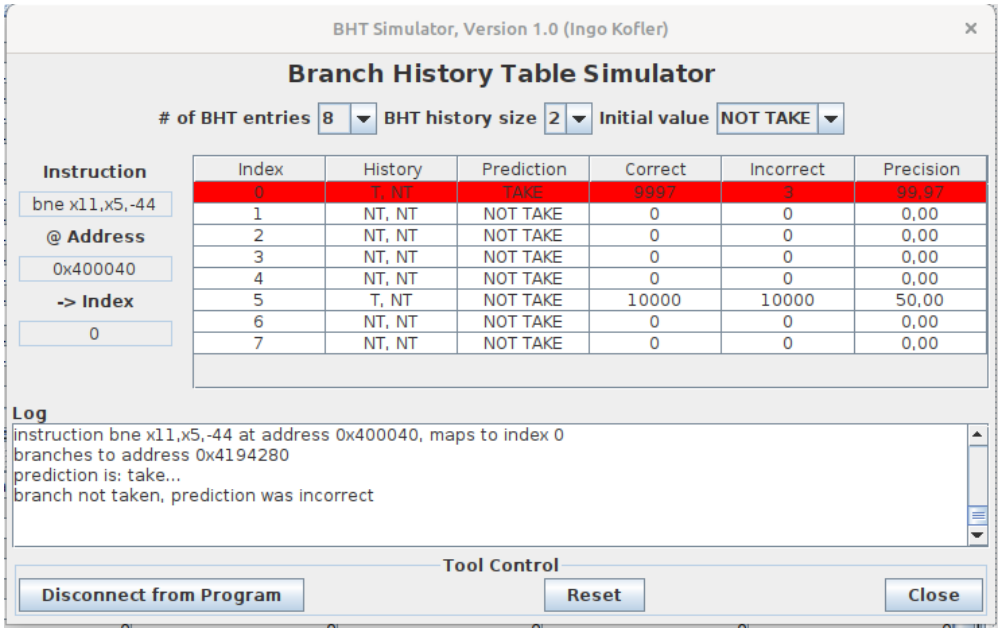
Здесь от нас требуется посмотреть на работу конечного автомата с 4 состояниями, который является подвидом бимодального предсказателя.



History	Prediction	State
NT, NT / T, NT	NOT TAKE	0/NT
NT, T	NOT TAKE	1/NT
T, NT	TAKE	2/T
T, T / NT, T	TAKE	3/T

Отличается этот от того, что был рассмотрен на лекции тем, что при выходе из *weak taken* переходит сразу в *strong not taken* и наоборот: *weak not taken -> strong taken*. Это отличие позволяет не попадаться в ловушку попеременного поведения *taken - not taken - taken - not taken ...* Выдавая хотябы половину правильных предсказаний, но все же не спасает от *taken - taken - not taken - not taken ...*

От нас требуется лишь посмотреть на его поведение в условиях выше написанной программы:



Вот здесь он и выдал в пятом индексе половину правильных предсказаний.
Точность - 50%.

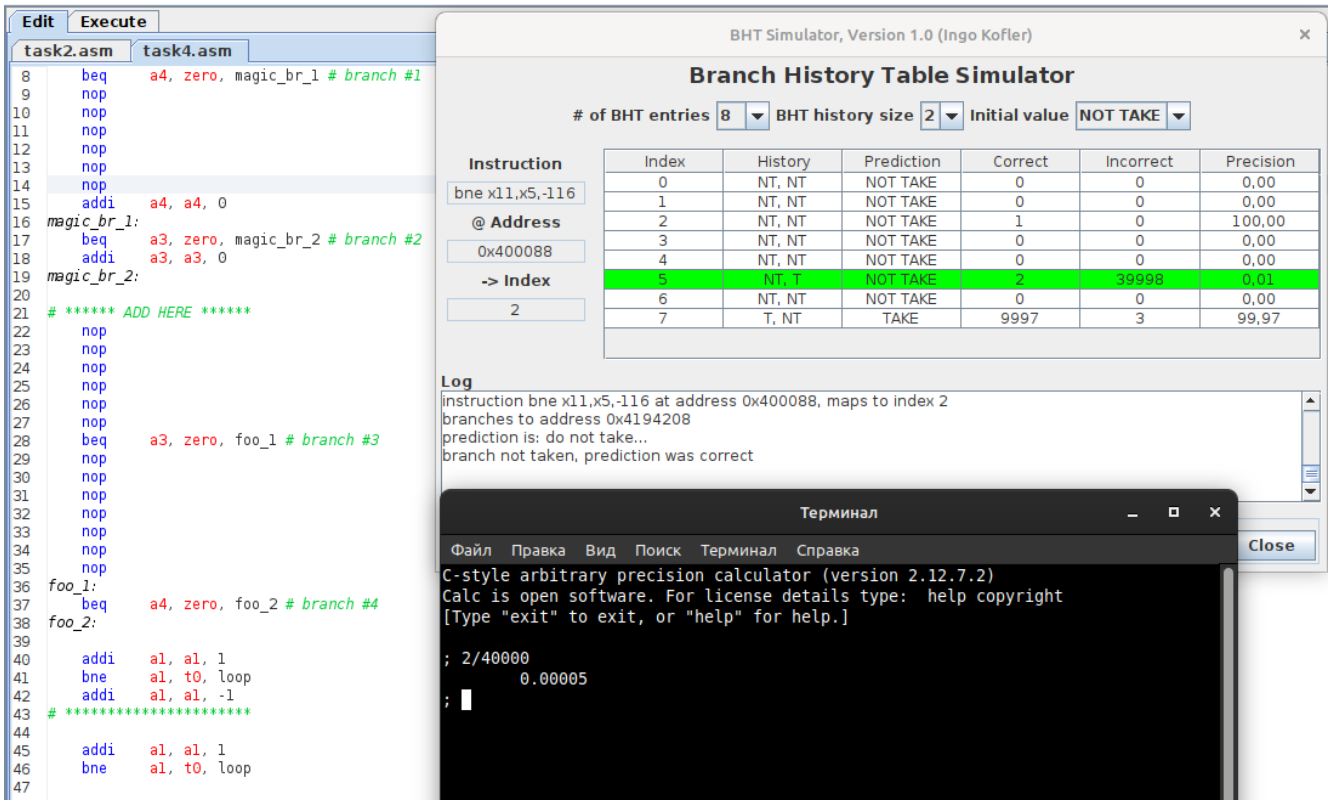
Пункт 4

Теперь уже требуется обмануть бимодальный предиктор. Впрочему, как это сделать, было описано выше.

Тогда реализуем попеременное поведение инструкций: *taken - taken - not taken - not taken ...*

Так и добьемся 0% предсказаний.

Разрешено писать только в отведенном месте, но в конце концов на ассемблере пишем, поэтому даже внутри ограниченного пространства можем перелопатить программу до неузнаваемости.

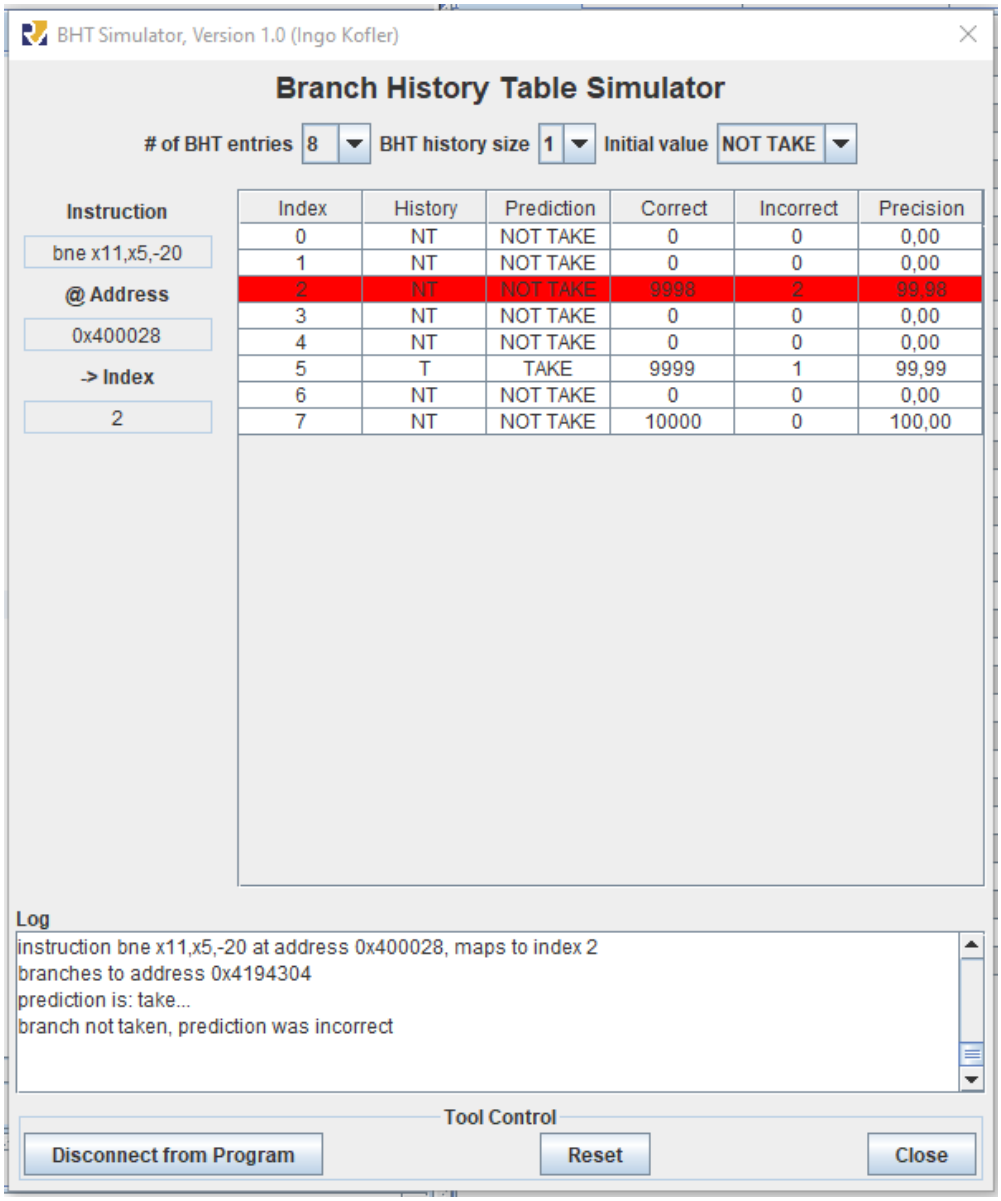


Здесь подогнали условные операторы так, чтобы специально поломать предиктор, впрочем так и получилось. Процент предсказания уже не прописывается в программе корректно, считаем отдельно в калькуляторе. Увеличивая начальное значение **t0**, уменьшается процент предсказания, но я не стал увеличивать, так как комп не тянет.

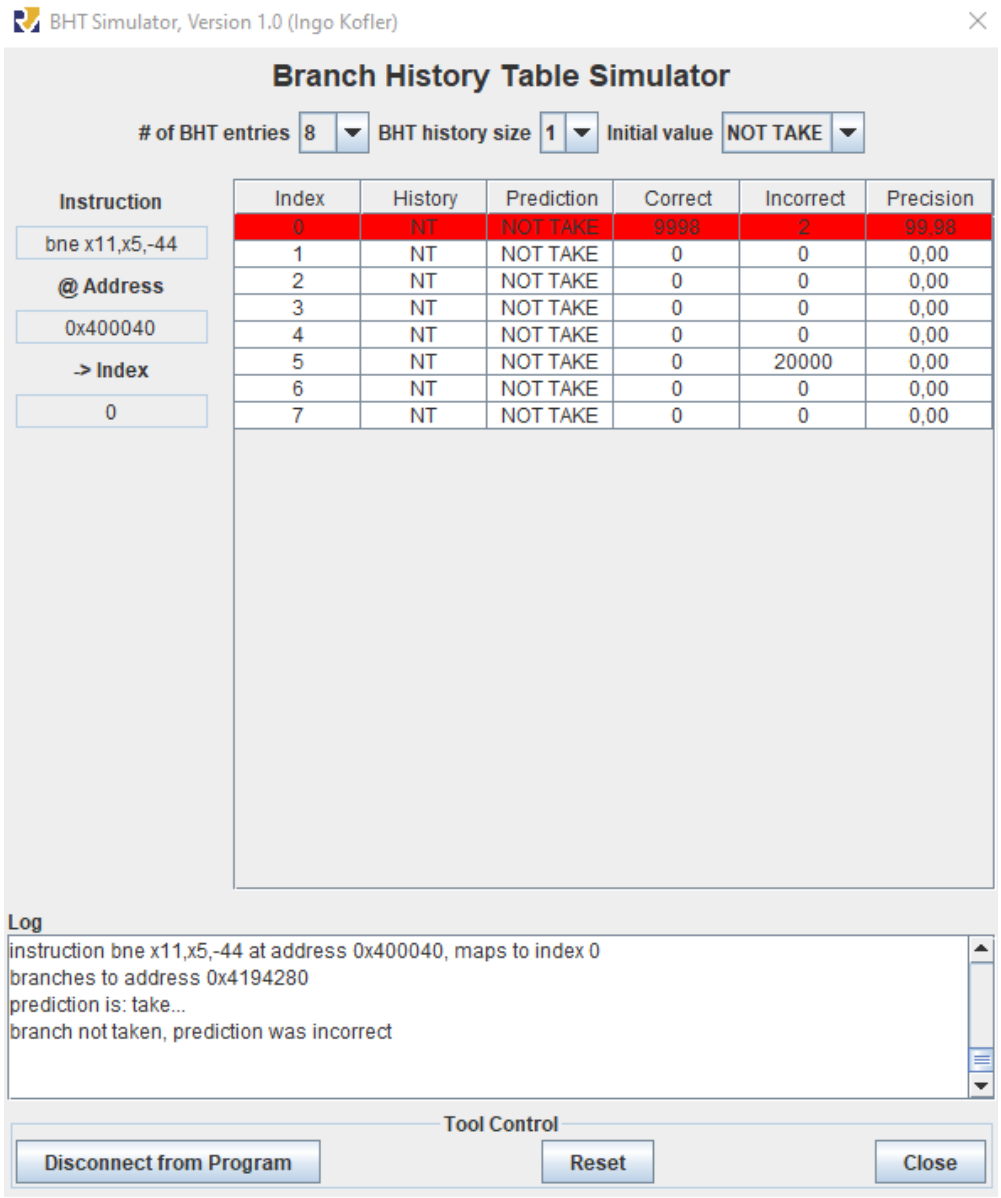
Когда все аспекты рассмотрены, можно начать написание "формального отчета".

2 Формальный отчет

1. Выполнил: Комиссаров Данил Андреевич.
2. Студент группы Б01-304.
3. Первый вариант задания.
4. Контакты: komissarov.da@phystech.edu
5.

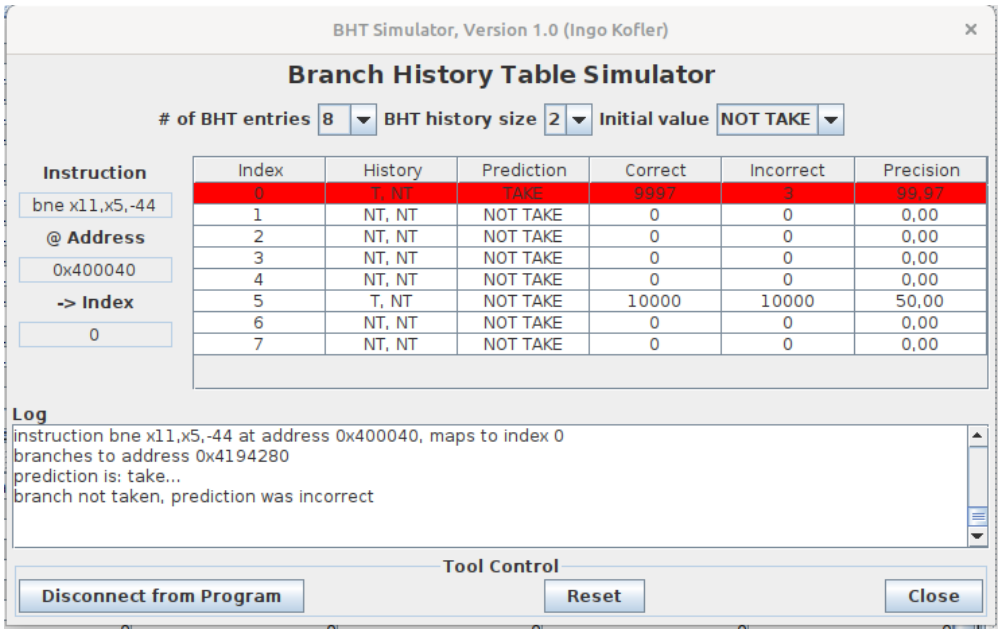


1. Точность **bagic_br_1** составляет 99,99%.
   Точность **bagic_br_2** составляет 100%.

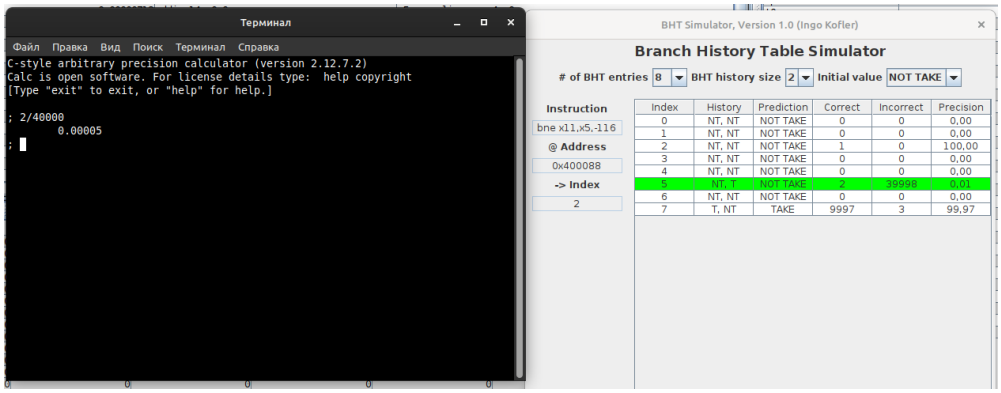

BHT Simulator, Version 1.0 (Ingo Kofler)
Branch History Table Simulator
# of BHT entries 8
BHT history size 1
Initial value NOT TAKE
Instruction
bne x11,x5,-44
@ Address
0x400040
-> Index
0
Index History Prediction Correct Incorrect Precision
0 NT NOT TAKE 9998 2 99,98
1 NT NOT TAKE 0 0 0,00
2 NT NOT TAKE 0 0 0,00
3 NT NOT TAKE 0 0 0,00
4 NT NOT TAKE 0 0 0,00
5 NT NOT TAKE 0 20000 0,00
6 NT NOT TAKE 0 0 0,00
7 NT NOT TAKE 0 0 0,00
Log
instruction bne x11,x5,-44 at address 0x400040, maps to index 0
branches to address 0x4194280
prediction is: take...
branch not taken, prediction was incorrect
Tool Control
Disconnect from Program
Reset
Close

```
1  .text
2  main:
3      li      t0, 10000
4      li      a3, 1
5      li      a4, 0
6      li      a1, 0
7  loop:
8      beq     a4, zero, magic_br_1 # branch #1
9      nop
10     nop
11     nop
12     nop
13     nop
14     nop
15     addi    a4, a4, 0
16 magic_br_1:
17     beq     a3, zero, magic_br_2 # branch #2
18     addi    a3, a3, 0
19 magic_br_2:
20
21 # ****** ADD HERE ******
22 # your code for task 4
23 # **********************
24
25     addi    a1, a1, 1
26     bne     a1, t0, loop
27
```

2. Индексы модулей предсказания для **beq branch1** и **branch2** совпали, а так как в настройках предиктора выставлен конечный автомат с двумя состояниями, то обманывать его можно, попеременно выдавая результаты *taken* - *not taken*. Так и добиваемся точности работы предиктора равной нулю.



3. В отличие от предиктора из первого задания, сейчас установлен предиктор с 4 состояниями, и причем он построен так, чтобы вместо **not taken** выдавать **taken**, необходимо ошибиться 2 раза, что и происходит в цикле *loop*. Так этот предиктор ошибается на 1 раз больше в цикле.




```
 8      beq     a4, zero, magic_br_1 # branch #1
 9      nop
10      nop
11      nop
12      nop
13      nop
14      nop
15      addi    a4, a4, 0
16  magic_br_1:
17      beq     a3, zero, magic_br_2 # branch #2
18      addi    a3, a3, 0
19  magic_br_2:
20
21  # ****** ADD HERE ******
22      nop
23      nop
24      nop
25      nop
26      nop
27      nop
28      beq     a3, zero, foo_1 # branch #3
29      nop
30      nop
31      nop
32      nop
33      nop
34      nop
35      nop
36  foo_1:
37      beq     a4, zero, foo_2 # branch #4
38  foo_2:
39
40      addi    a1, a1, 1
41      bne     a1, t0, loop
42      addi    a1, a1, -1
43  # ************************
44
45      addi    a1, a1, 1
46      bne     a1, t0, loop
47
```

4. Действительно, так и сделаем: цикл **taken - taken - not taken - not taken**. В конце концов на ассемблере пишем, даже внутри ограниченного пространства можем перелопатить программу до неузнаваемости. Корректно процент предсказания уже не прописывается в программе, считаем отдельно в калькуляторе. Увеличивая начальное значение **t0**, уменьшается процент предсказания, но я не стал увеличивать, так как комп не тянет.

На этом пожалуй можно и закончить. Наверное я уделил слишком много внимания несущественным аспектам, в то время, когда некоторые фундаментальные принципы были упущены, но это было интересно.