СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность. Предсказатель переходов (далее ПП) является модулем, значительно ускоряющим работу процессора. Благодаря его предсказаниям модуль выборки команд может спекулятивно загрузить инструкцию из памяти и начать выполнять её до того, как выполнится инструкция ветвления, и будет известно, какие инструкции стоит выполнять далее. Разработка данного модуля – это дорогой и трудоёмкий процесс, так как ПП имеет несколько алгоритмов предсказания, которые эффективны в разных ситуациях. Разработчику приходится изучить, реализовать и протестировать несколько из них, чтобы выяснить подходящий. Эффективность алгоритмов зависит от большого количества факторов, что усложняет процесс выбора. В данном исследовании будут разобраны и проанализированы некоторые алгоритмы предсказания. Выводы исследования помогут в выборе алгоритма и упростят разработку ПП. Важно понимать, что эффективность предсказателя зависит от большого количества «плавающих» параметров кода: количество циклов, условий и их связности, адресов инструкций ветвления. Это вносит варьирование результата при одном и том же алгоритме. Поэтому выводы данного исследования необходимо воспринимать, как ориентир при выборе ПП.

Степень теоретической разработанности темы. Эффективность алгоритмов предсказателей переходов очень чувствительна к параметрам процессора, поэтому исследования этой области устаревают так же быстро, как быстро меняется структура конвейера. Большая часть разбираемых алгоритмов разработаны в 90-х годах прошлого века, поэтому имеющиеся данные требуют регулярной перепроверки.

Также существуют статьи, сравнивающие несколько алгоритмов, но они носят частных характер, так как выводы сделаны в контексте конкретного процессора.

В данной работе будет разобран алгоритм TAGE, с которым и будут сравниваться его исторические предшественники, так как этот алгоритм самый

новый из разбираемых. Данное исследование анализирует алгоритмы в разных процессорных моделях, что позволяет сделать общие выводы об алгоритмах.

Цель и задачи. Цель работы - повышение эффективности подсистемы предсказания переходов в процессоре за счет её адаптации к микроархитектуре процессорного ядра. Сложность предсказателя переходов должна соответствовать сложности самого процессорного ядра, но нет "стандартного" порядка выбора конфигурации предсказателя. В работе необходимо предложить метод получения эффективных структур предсказателей для ядер с конкретной структурой конвейера. Для этого необходимо разработать несколько классических реализаций предсказателя. Для тестирования необходимо разработать тестовое окружение, содержащее считывание тестов, преобразование их к интерфейсам предсказателя, запуск алгоритмов и подсчёт статистики. Также необходимо протестировать предсказатель на реальном окружении с разными конфигурациями. Проанализировать полученные данные и предложить метод выбора предсказателя.

1 Проблема синтеза эффективной структуры ПП

Данное исследование несёт цель в формировании некоторого принципа, который позволит ускорить процесс разработки предсказателя переходов в зависимости от необходимых параметров, а также сформировать предсказатель переходов близким к оптимальному. Данный принцип предлагает метод, следуя пунктам которого, можно получить оценку эффективного алгоритма и его конфигурации. Под эффективностью понимается величина, пропорциональная полученной от предсказателя точности, а также обратно пропорциональна количеству памяти, используемой предсказателем. Причём точность гораздо важнее используемой памяти, исключая случаев бюджетных ограничений.

Для формирования данного метода предлагается рассмотреть и проанализировать наиболее популярные алгоритмы предсказания, а также связанные с ними статьи. Для рассмотрения были выбраны:

- static
- last decision
- counter
- global history (gh)
- path history (ph)
- GShare
- Tage

1.1 Существующие алгоритмы ПП

ПП ориентирован на работу с инструкциями, классифицируемыми, как инструкции ветвления. Из них состоят циклы, условия, функции, возвраты из функций, switch case-ы, jump-ы. В ходе выполнения таких инструкций выявляется дальнейший ход программы: либо результатом будет notTaken(nT) - процессор продолжит выполнять команды, следующие за инструкцией ветвления, либо результатом будет Taken(T) – процессор сменит адрес выполняемых инструкций на адрес, вычисленный из инструкции ветвления.

Предсказатель переходов принимает адрес инструкции ветвления, запускает алгоритм предсказания, кладет адрес в очередь, после выполнения инструкции забирает адрес из очереди и выполняет обучение, используя «правильный» ответ.

ПП часто решает и другие вопросы, не разбираемые в данной работе: разделение инструкций ветвления и не инструкций ветвления, выявление адреса перехода, сохранение адреса возвращения из функций. Исследование сконцентрировано на анализ алгоритмов предсказания, их сравнения и формирование алгоритма выбора предсказателя по структуре конвейера.

1.1.1 Static prediction

Первой алгоритм действует без обучения. Концепция static prediction заключается заранее заданном предсказании. Это может быть постоянное Tkn, постоянное nTkn, решение, основанное на «подсказках» компилятора. Эти решения хорошо подходят для заранее заданных программ. Например, конструкция for очень часто выдает Tkn, и всего один раз, когда цикл заканчивается, nTkn. В программе, состоящей в основном из циклов, можно воспользоваться статическим предсказателем. Однако статические предсказатели неэффективны, если код не известен. При случайном угадывании можно рассчитывать лишь на 50% успешных предсказаний. Лучшей вариацией статического предсказателя является always Tkn, так как циклы почти всегда оказываются Tkn. «Подсказки» компилятора также не дают хороших прогнозов, так как компилятор должен сделать вердикт ещё до того, как будет выполнен код. Некоторые решения запускали уменьшенную версию теста, собирали историю выполнения инструкций ветвления, оставляли «подсказки» в коде и запускали полный тест со статическим предсказателем, который основывался на предсказаниях [2]. Однако реализация «подсказок» компиляторов – довольно дорогая разработка для дешёвого статического метода предсказаний. Данная идея нашла своё применение в алгоритмах, использующих сложную и дорогую структуру [12].

Тем не менее, статический алгоритм довольно важен, так как может использоваться в паре с другими (даже мощными предсказателями). В исследовании [13] упоминаются ряд инструкций ветвления, которые ввиду контекста или своей структуры всегда или почти всегда выдают один и тот же результат. Если позволять предсказателям со счётчиками сохранять информацию о них в своих таблицах, то это может привести к ухудшению точности из-за лишних коллизий. Подобные случаи стоит обрабатывать статическим предсказателем, что и описывается в статье.

1.1.2 Last decision

Алгоритм последнего решения требует внутреннюю кэш-память. Индексация по этой памяти осуществляется по адресу инструкций, а ячейки содержат один бит, который показывает последнее решение – Tkn или nTkn. То есть, получая на вход предсказателя адрес, алгоритм выбирает из кэш-памяти ячейку по этому адресу и делает решение, зависящее от значения ячейки: 0 – nTkn, 1 – Tkn. При обучении меняем значение в случае ошибочного предсказания. Данный алгоритм хорошо справляется с задачами, типа пузырька. В таком алгоритме есть условие, которое много итераций подряд выдает один результат, а после другой. Когда элемент «всплывает» по уже отсортированной части массива, он будет меняться с соседом, пока он больше него, то есть Tkn. Когда элемент найдет своё место, все дальнейшие проверки на перестановку значениями соседних ячеек будут выдавать nTkn. Таким образом мы ошибемся 1 или 2 раза за один проход внутреннего цикла – в самом начале и в моменте, когда Tkn сменится на nTkn. Такой предсказатель даёт больший процент успешных решений, чем статическое предсказание, однако он всё еще не подходит для универсального выполнения задач, так как подобные случаи однотипного поведения условий довольно редки.

1.1.3 Bim counter

Бимодальные счётчики – это логическое развитие предыдущего алгоритма. Предлагается хранить не один, а более бит. Причём при обучении увеличивать значение ячейки на 1 при Tkn и уменьшать при nTkn.

Инкрементация и декрементация производится с насыщением, то есть увеличение не происходит при максимальном значении счётчика, а декрементация не происходит при минимальном значении счётчика. Тогда решением будет старший бит такого счётчика. То есть если в последнее время было много Tkn, тогда и сейчас мы выберем Tkn. Исследования показывают, что максимально эффективным будет счётчик из двух бит.

Однако такой предсказатель имеет значительный недостаток. Предположим, что в процессе выполнения предсказаний в счётчике находится 0. Тогда, если дальнейшие условия будут периодично выдавать Tkn и nTkn, тогда алгоритм периодично будем инкрементировать и декрементировать счётчик, что приведёт к постоянному ответу nTkn, так как старший бит не меняется.

1.1.4 Global history (GH)

GH и PH относятся к двухуровневым предсказателям, то есть состоят из двух частей, одной из которых является вычисление индекса, а второй – счётчики. Двухуровневые предсказатели могут дать значительный прирост производительности, что подробно описывается в статье [16].

Идея глобальной истории основывается на том, что условия могут быть связаны между собой.

```
if (a == 5)
    b = 3
if (b == 3)
    ...
```

Рисунок 1 – Пример связанных условий

В данном примере (рисунок 1) выполнение второго условия напрямую зависит от первого. Хотелось бы учитывать предыдущие Tkn или nTkn для текущего решения. Алгоритм «глобальной истории» использует бимодальные счётчики, но индексация происходит не только по адресу инструкции перехода, но и по регистру глобальной истории. Регистр глобальной истории –

это сдвиговый регистр, который запоминает последние N выполнений инструкций ветвления (рисунок 2).

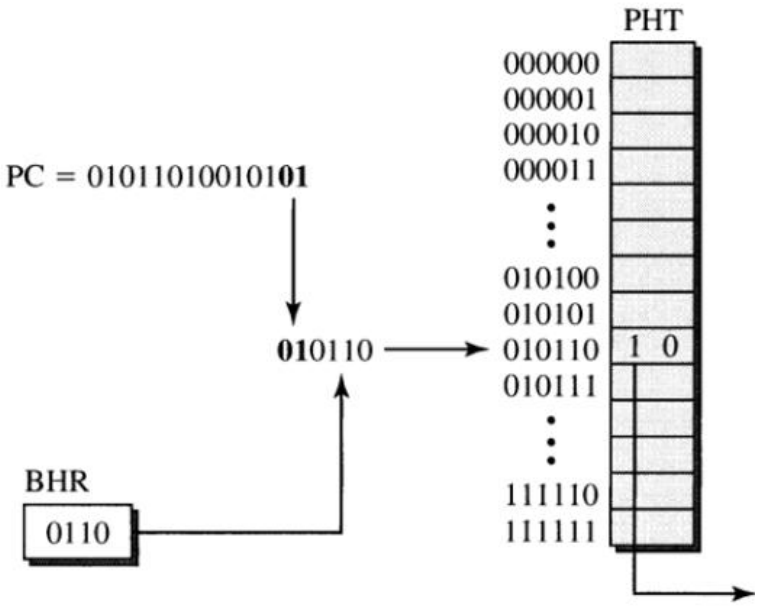

Рисунок 2 – Global history (GH)

Но важно понимать, что есть мы не хотим увеличивать количество ячеек бимодальных счётчиков, значит придется использовать меньшее количество бит адреса инструкции. А если мы используем не полный адрес инструкции, то рискуем получить коллизии – использование разными инструкциями перехода один бимодальный счётчик. Стандартный вариант уменьшения коллизий – это увеличение памяти, однако такой вариант не всегда возможен, и приходится прибегать к различным оптимизациям [11].

Подытожив, глобальная история помогает найти шаблоны выполнения программ.

1.1.5 Path history (PH)

Аналогичные GH рассуждения можно привести по отношению к инструкции перехода самой по себе. Напомним, что бимодальные счётчики плохо справляются с цикличными Tkn, nTkn. То есть бимодальные счётчики никак не учитывают последние выполнения данной инструкции. Они лишь знают, кого за всё время выполнения было больше (Tkn или nTkn). Алгоритм «локальной истории» предполагает добавление сдвиговых регистров (аналогичные gh), но для каждого адреса свой.

Снова мы вынуждены использовать для индексации несколько бит, которые ранее использовались битами счётчика команд, и это приводит к повышению коллизий.

Данный алгоритм (рисунок 3) требует намного больше ячеек памяти, чем алгоритм «глобальной истории», однако он может дать полезен в ряде случаев, в которых инструкции ветвления ведут себя по определённом шаблону.

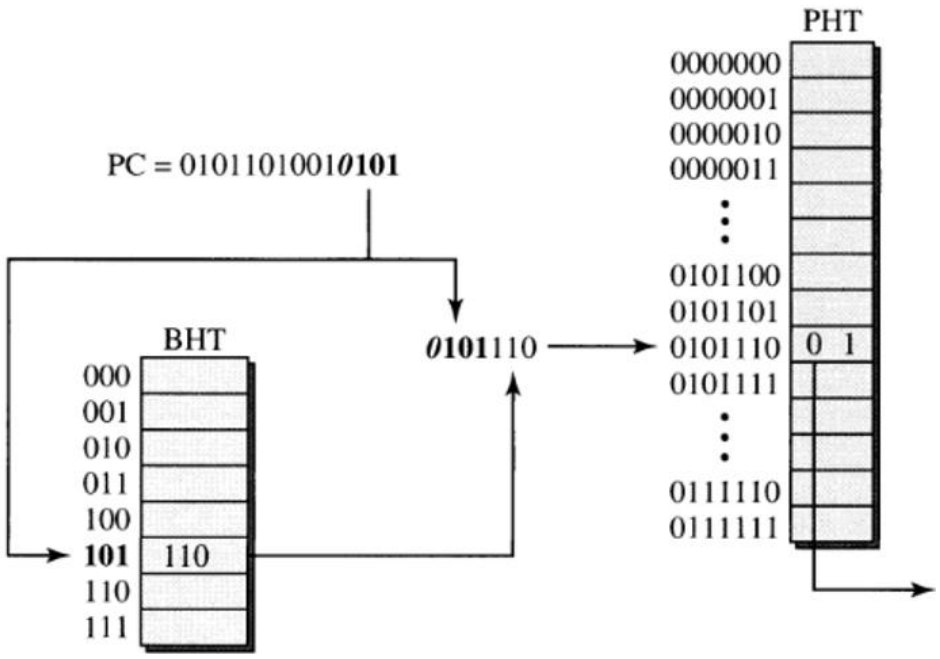

Рисунок 3 – Path history (PH)

1.1.6 GShare

GH и PH имеют большой недостаток: индексация в них приводит к повышению коллизий, а также пустых ячеек. Например, ячейка с адресом 0x0000 может встретиться в крайне редком случае, когда адрес инструкции ветвления – 00, а история была также 00. Другие ячейки наоборот будут встречаться чаще. Алгоритм «GShare» (рисунок 4) предлагает не конкатенацию частей для индексации, а их хеширование через операцию исключающего или. Это приводит к равномерному распределению по ячейкам, однако коллизии в ряде случаев могут возрасти. Исследования показывают, что операция исключающего или приводит к возрастанию точности [6].

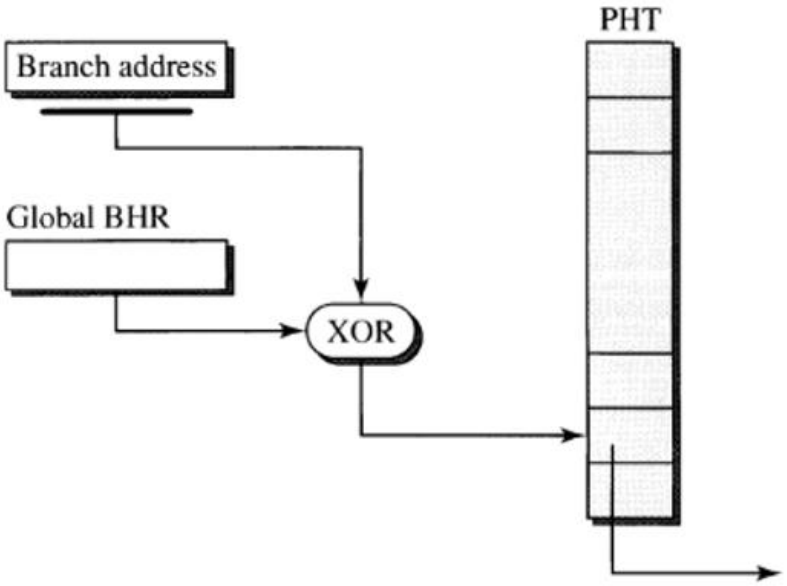

Рисунок 4 – GShare

Исследование [14] показывает, что использование и локальной, и глобальной истории вместе приводит к лучшему результату, чем использование одного из алгоритмов отдельно.

1.1.7 Tage (Tagged Geometric)

У алгоритмов, использующих историю есть ещё один недостаток: они используют историю конкретного размера. Если шаблон повторяется каждые 4 раза, то в алгоритмах с историями достаточно истории из n бит. Если будет использовано больше, то излишние биты не несут никакой пользы. И наоборот, если используемая история содержит меньше бит, чем необходимо для распознавания шаблона, то алгоритм вообще не выполнит свою задачу. Причём длина шаблона у каждой инструкции может быть разная.

Алгоритм «TAGE» несколько таблиц с бимодальными счётчиками, каждая из которых индексируется разным количество бит. Пронумеруем таблицы от нуля до N включительно. Каждая ячейка помимо бимодальных счётчиков содержит бит валидности и уровень достоверности. Уровень достоверности показывает – как часто эта ячейка ошибается. Чем уровень достоверности выше, тем чаще данный бимодальный счётчик предсказывает верно, следовательно тем больше стоит доверять именно данной таблице. При возникновении запроса на предсказание алгоритм опрашивает все таблицы и

сравнивает их предсказания и уровни достоверности. В классической реализации для результата выбирается таблица с наименьшей длиной истории и с наивысшим уровнем достоверности. Также в реализацию данной алгоритма входит аллокация по тэгу. Чтобы не выбирать предсказания из других таблиц с другими историями, которые не относятся к данному случаю, добавим каждой ячейке тэг. Бит валидности обозначает занятость ячейки. Если предсказание было ошибочным, то алгоритм пытается аллоцировать новую ячейку в другой таблице. Если все таблицы уже содержат данный адрес инструкции, то аллокации не происходит. Освобождение ячейки происходит, если уровень достоверности оказался ниже заданного предела – ошибается слишком часто. Под аллокацией подразумевается использование хэш-функции из адреса инструкции, глобальной и локальной истории, сохранение результата этой хэш-функции в ячейку, называемую тэгом, и сравнивание данного тэга с каждым новым запросом. Работа с занятой ячейкой реализуется только при совпадении тэгов. Чтобы ячейка не стала вечно занятой, реализуется механизм устаревания. Добавим счётчик, который будет увеличиваться при очередном неуспешном аллоцировании и обнуляться при успешном. При достижении определённого предела все ячейки освобождаются. Чтобы алгоритм запустился, так как вначале никакая ячейка не содержит тэгов, нулевая таблица не содержит тэгов и всегда выдаёт результат. Нулевая таблица аналогична обычному алгоритму с историей. При ошибках нулевой таблицы запустится основной алгоритм аллокации в таблицах с тэгом.

Исследования показывают [7], что для данного алгоритма следует использовать геометрическое возрастание количества бит, используемого для индексации.

Данный алгоритм (рисунок 5) сильно превышает необходимое количество ячеек памяти по сравнению с остальными разобранными алгоритмами. Он использует порядка 10 таблиц, аналогичных другим алгоритмам, но в каждой ячейке также хранятся тэг, валидность и уровень

достоверности. Однако такие потребления памяти могут быть уместны в ряде случаев, так как TAGE имеет значительный прирост количества успешных предсказаний.

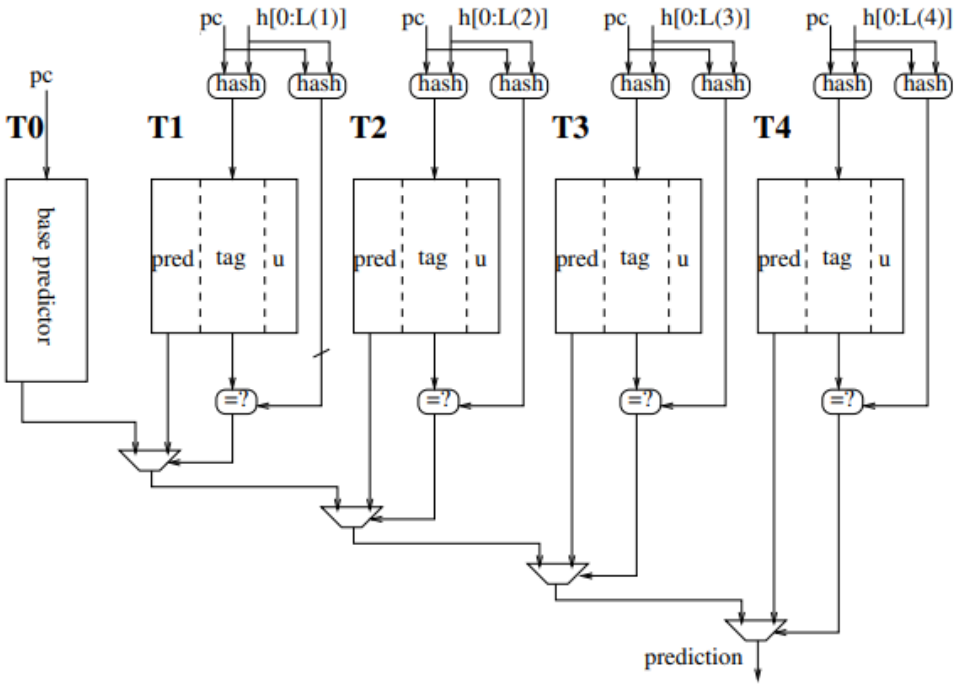

Рисунок 5 – TAGE

TAGE является тяжеловесным и дорогим решением. В случае, если требования к производительности процессора высоки, производитель вынужден использовать данной алгоритма, так как он даёт одни из лучших показателей.

Алгоритм TAGE является довольно востребованным, так как может получить один из лучших показателей точности, поэтому имеет множество исследований. Так как данное исследование рассматривает TAGE, как ориентир на лучшую достижимую точность, подробного исследования по конфигурации TAGE не проводилось ввиду трудоёмкости данного алгоритма. Структура и конфигурация были сформированы на основе исследований [19], [20].

Вторым стандартом для высокопроизводительных процессоров является алгоритм «прецептрон» или однослойная нейронная сеть, который часто упоминается во многих исследованиях ([22], [23], [24], [30]). Данный алгоритм также может показать высокую точность, и его нужно иметь в виду при выборе

предсказателя переходов, однако данное исследование направлено на сравнение более дешёвых решений.

1.2 Анализ существующих алгоритмов ПП

1.2.1 Точность

Предсказатель переходов является узким местом для производительности современных процессоров (упоминается в исследовании [18]), поэтому повышение точности – одна из ключевых задач разработчика.

Важной причиной к использованию предсказателя переходов является конвейеризация процессора. Команды разделяются на подзадачи, которые можно выполнять параллельно в разных модулях ядра. Выполнение такого параллелизма возможно только при условии того, что известна последовательность выполняемых инструкций. Сравнение последовательного и конвейерного процессора обсуждается в исследовании [15]. В нём сделан вывод, что предсказатель переходов привносит большой вклад в производительность процессора при использовании конвейеризации.

Результаты предсказателя переходов сильно зависят от теста, на котором он используется. Даже дорогие алгоритмы могут точечно показать результат хуже, чем более дешёвые алгоритмы, поэтому невозможно сформировать точный процент успешных предсказаний для какого-то алгоритма, но можно воспринимать результаты разбираемых статей, как ориентиры. Здесь будут перечислены некоторые данные, полученные по исследованиям алгоритмов предсказателя переходов.

По результатам исследования [3] можно заметить (рисунок 6), что при запуске одного и того же предсказателя на разных бенчмарках получаются разные результаты (одни – около 1% неверных предсказаний, а другие – 10%). Причём бенчмарки подразумевают большое количество тестов, что исключает возможность маленькой выборки. Выводом данного исследования является то, что любой алгоритм может дать большую вариацию результатов, зависимых от самих тестов. Например, PH тратит гораздо больше памяти, а на большинстве тестов показывает результаты, сравнимые с GH, который использует гораздо

меньше памяти. Однако использование обоих алгоритмов покрывает большой процент случаев, который не смог бы покрыть каждый из методов сам по себе.

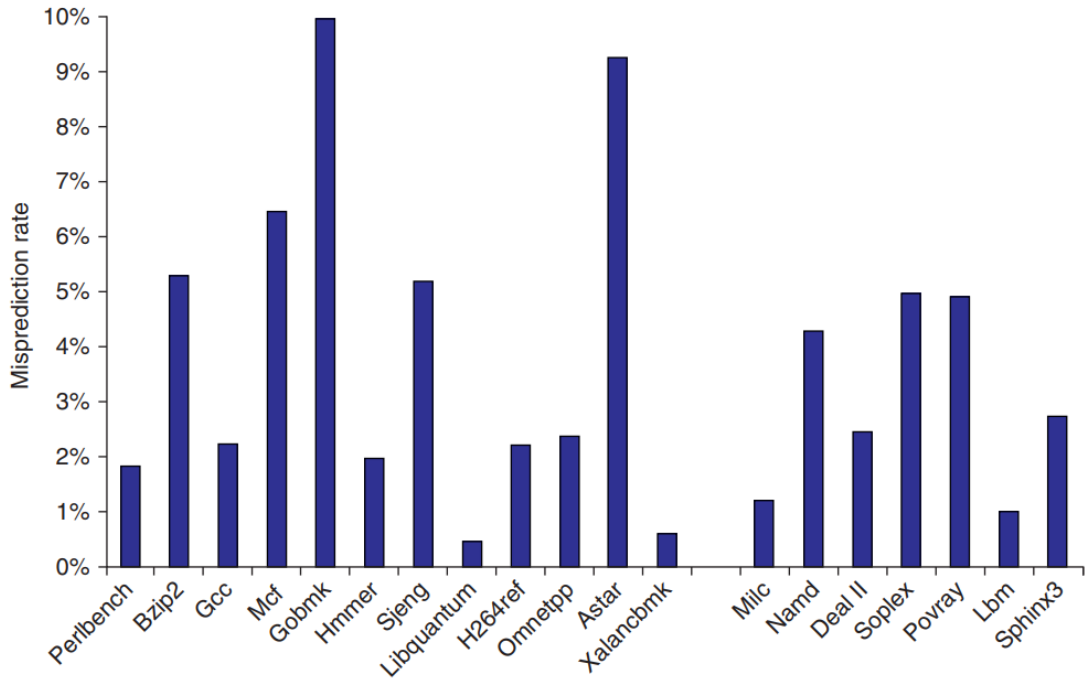

Рисунок 6 – Зависимость процента ошибок от теста

В таблице 1 перечислены проценты успешных предсказаний различными алгоритмами. Снова можно заметить разные цифры у одних и тех же алгоритмов. Во-первых, они могли запускаться на разных тестах; во-вторых, они могли иметь отличительные детали реализации. Например, у статического предсказателя есть несколько подвидов:

– Всегда отвечать Tkn;

– Всегда отвечать nTkn;

– Если jump вперед – nTkn, иначе Tkn;

– Отвечать так, как «подсказывает» компилятор.

Таблица 1 –Точность предсказаний, приведенная из разных источников

	static	last	bim	gh	ph	gshare	tage
[1]	70	85	90	93	94	94	95+
[2]	55-80	79-96	83-97	95–			–

Подобные различия в реализациях существуют и в других алгоритмах, поэтому сравнивать показатели алгоритма от разных источников не стоит.

Важно сравнивать результаты разных алгоритмов предсказателей, которые выполнялись в одном и том же исследовании, то есть рассматривать каждую строчку отдельно.

В [2] исследовании сравнивались возможные шаблоны изменения счётчика по результату инструкции перехода (Tkn – жирная стрелка, nTkn – тонкая стрелка) (рисунок 7). Всего 2^{20} вариантов изменения счётчика. Исследование показывает, что классические инкрементация и декрементация с насыщением показывают наилучший результат, причём начальная позиция не имеет большого значения.

Benchmark	Optimal	"Counter"
spice2g6	97.2	97.0
doduc	94.3	94.3
gcc	89.1	89.1
espresso	89.1	89.1
li	87.1	86.8
eqntott	87.9	87.2

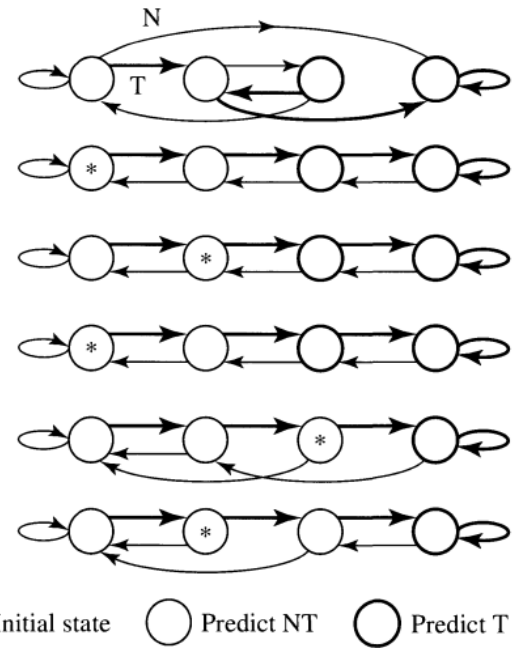

Рисунок 7 – Варианты бимодальных счетчиков

Слабой стороной представленных алгоритмов является графовые вычисления. В статье [21] упоминается ряд ключевых проблем, связанных с графами, которые возможно исправить другими технологиями. Данную проблему стоит иметь в виду при выборе предсказателя переходов.

1.2.2 Память

Память – это важный параметр ПП, так как он влияет на цену, энергопотребление, тепловыделение. В исследовании [31] рассматриваются различные оптимизации предсказателя для экономии памяти.

Важно понимать, что конкретные показатели память в исследованиях не имеют большого значения. В частных случаях разработчик может добавить

используемую память (увеличить количество бит от PC для индексации), ожидая уменьшение коллизий и увеличения процента верных предсказаний, но такое изменение может изменить предсказание в какой-то момент теста, из-за чего всё дальнейшее «течение» предсказателя переходов изменится, а так как любой алгоритм имеет вероятностный результат, реализация с увеличенной памятью может получить меньший процент успешных предсказаний.

В [3] исследовании сравнивается предсказатель, содержащий 4096 ячеей бимодальных счётчиков, «бесконечное» количество бимодальных счётчиков (достаточно проиндексировать все возможные адреса инструкций), 1024 ячейки с двухуровневой реализацией (GH) (рисунок 8). Увеличение количество ячеек почти не показало прироста успешных предсказаний, однако уменьшение количества ячеек и «улучшение» алгоритма привело к заметному уменьшению ошибочных предсказаний. Интересно здесь то, что такой результат не во всех тестах. Некоторые из них показали, что использовать больше ячеек – выгоднее, чем лучший алгоритм. Это опять возвращает к мысли о том, что конкретный алгоритм может быть дороже и эффективнее в целом, но хуже в конкретном случае.

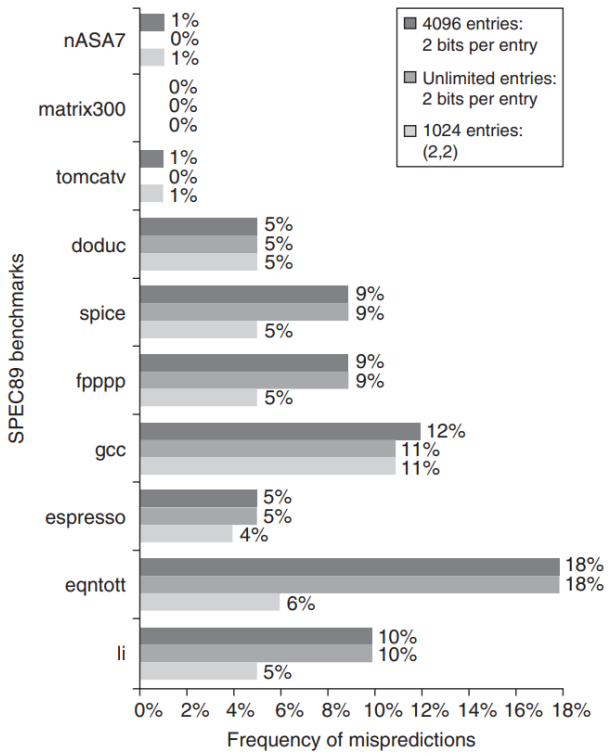

Figure 3.3 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

Рисунок 8 – Результаты ошибок предсказаний на разных тестах в зависимости от конфигурации предсказателя

Также исследования [3] показывают, что увеличение количества бит в счётчике носит логарифмических характер, то есть каждый следующий добавочный бит принесёт меньше пользы, чем предыдущий (рисунок 9).

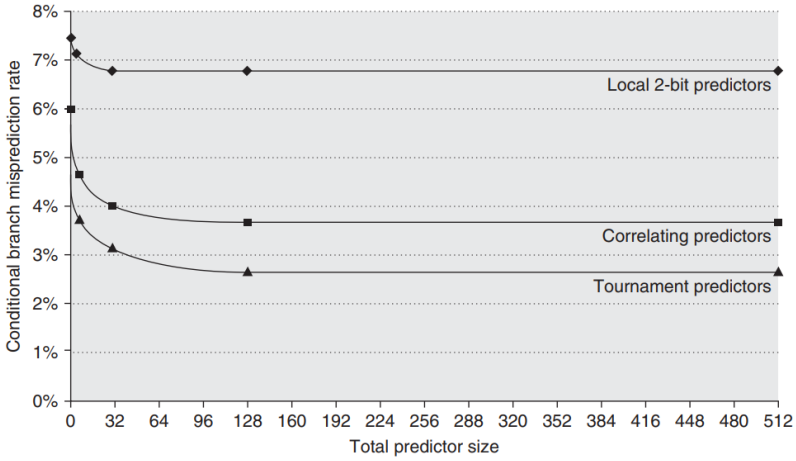

Figure 3.4 The misprediction rate for three different predictors on SPEC89 as the total number of bits is increased. The predictors are a local 2-bit predictor, a correlating predictor that is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar behavior, perhaps converging to the asymptotic limit at slightly larger predictor sizes.

Рисунок 9 – Зависимость процента ошибок предсказаний от размера кэша (памяти) предсказателя

Опираясь на данную информацию [2] об используемой памяти, сформируем относительные показатели для всех алгоритмов (Таблица 2).

Таблица 2 – Память алгоритмов

Алгоритм	Используемая память	Комментарий
static	не использует память	–
n-counter	$n \times 2^X$	каждый счётчик содержит n бит, а всего счётчиков 2^X
gh	$n \times 2^X + h$	помимо счётчиков используется глобальная история на h бит
ph	$n \times 2^X + h \times 2^X$	для каждого адреса, помимо самого счётчика, используется локальная история,

		каждая из которых содержит h бит
GShare	используется столько же, сколько и алгоритм внутри (GH, PH, GH+PH)	
TAGE	$10 \times ((n + 1 + 2 + T) \times 2^X + h \times 2^X)$	в данном алгоритме используется порядка 10 таблиц, каждая из которых содержит $(\mathrm{n} + \mathrm{v} + \mathrm{ctrl} + \mathrm{tag}) \times 2^X + h \times 2^X$, где v – валидность, ctrl – достоверность, tag – тэг

Говоря об используемой памяти, важно помнить о проблемах, которая эта память приносит: цена, энергопотребление и тепловыделение. Проблема с энергопотреблением можно быть довольно острой в случаях встраиваемых систем. Исследование [5] стремится разными способами уменьшить данный показатель, а исследование [10] пытается отключать разные части предсказателя, если считает их неэффективными, поэтому стоит использовать как можно меньше памяти, если целевая точность достигнута.

Количество используемой памяти также влияет и на резервное копирование. В статье [25] перечисляются возможные решения в системах с повышенным риском перебоев энергии, что может привести к сбросу состояния. Данные решения основываются на резервном копировании, сложность которого пропорционально объёму памяти. Повышенный объём памяти может стать критическим фактором в подобных системах.

1.2.3 Использование в процессорах

В таблице 3 приведены модели процессоров, использующих рассматриваемые алгоритмы. Если сравнить год изготовления данных процессоров, можно заметить, чем процессор новее, тем мощнее он использует алгоритм предсказания. Информации по TAGE обнаружено мало, что по использованию в процессорах, что по сравнению с другими процессорами. Связано это с тем, что TAGE самый новый из рассматриваемых алгоритмов, большинство статей, связанных с ним, датируются прошлым десятилетием, когда информацию о других алгоритмах можно найти в статьях 20-го века. Большинство статей, описывающих TAGE, носят частных характер, запускались на маленьком количестве тестов, не имеют сравнительной статистики с другими алгоритмами. Но показанной информации достаточно, чтобы сделать выводы об эффективности TAGE, так как все результаты оказываются чуть меньше 100% успешных предсказаний. Также стоит сказать о том, что TAGE более других алгоритмов имеет ряд значительных изменений в реализации, таких, как LTAGE[7], BATAGE[8], ITTAGE[9] и др. Такие тонкости связаны с тем, что передовые процессоры гонятся за каждой долей процента успешных предсказаний, так как длинный конвейер современных процессоров имеет большую цену неверного предсказания, поэтому производители готовы потратить много ресурсов на точечную разработку TAGE. Поэтому TAGE будет восприниматься, как самый дорогой и самый эффективный предсказатель. Сравнивать его с другими алгоритмами в дорогих реализациях смысла нет, так как его показатели значительно выше. Однако не всё так однозначно становится при менее затратных задачах. В случаях короткого конвейера ошибка не будет сильно уменьшать производительность. Или же если производительность в конкретной задаче не так важна, как цена. Именно в подобных случаях стоит задуматься о относительно простых реализациях предсказателя, поэтому результаты TAGE в данном исследовании стоит воспринимать, как максимум, а другие алгоритмы будут сравниваться по принципу – насколько дешевле, но насколько хуже.

Таблица 3 – Применение алгоритмов в различных процессорах

Ссылки на исследования	static	last	bim	Gh	ph	gshare	tage
[1]	PPC 601, PPC 603	DEC EV4, MIPS R8000	LLNL S-1, Burroughs B4900, Intel Pentium, PPC 604, DEC EV45	Pentium MMX	Pentium Pro, Pentium II, Pentium III	MIPS R12000, UltraSPARC-3	Intel Haswell
[2]	Intel i486, Motorola 88110			AMD/NexGen Nx686 (не уточняется, какой конкретно двухуровневый предсказатель)	Intel P6	IBM Power4, DEC Alpha 21264	
[3]				Intel core i7			

Также важно отметить тот факт, что разные алгоритмы используют разное количество тактов. Статические алгоритмы выполняются за 1 такт, одноуровневые (bim) – за 2, двухуровневые – за 3 (gh, ph, gshare), а сложные предсказатели, которые использую результаты двухуровневых, за более чем 3 (tage – 4, tournament – 5, например, как в [4]).

1.2.4 Проблема синтеза

Для проектирования предсказателя переходов необходимо: реализовать несколько распространённых алгоритмов предсказателей; сформировать набор тестов, содержащих большое количество различных шаблонов

предсказания для получения более честного результата; разработать модуль процессора, собирающий статистику предсказаний; провести анализ конфигураций и подобрать наиболее эффективные; провести тестирование и собрать статистику. Необходимо разбирать разные алгоритмы из-за того, что они имеют разные конфигурации, то есть разное количество используемой памяти, что влияет на цену, энергопотребление, тепловыделение, количество тактов на предсказание. Однако нет стандартного метода выбора предсказателя, так как большое количество факторов влияют на результат.

Такая цепочка разработки предсказателя состоит из большого количества специалистов (разработчики, тестировщики, аналитики), а также может дать неверный результат из-за неправильно выбранной конфигурации, ошибки разработчика или нечестной (содержащие повторяющиеся шаблоны) выборки тестов. Такой процесс может быть довольно дорогой. Отказываясь от полного перебора данных алгоритмов ради экономии, производитель может получить неэффективный предсказатель.

В контексте предсказателя переходов важно упомянуть внеочередное выполнение (далее ВВ) инструкций. Современные суперскалярные системы повсеместно используют технологию ВВ для повышения производительности. Данное исследование может быть особенно полезно в разработке процессоров с ВВ, так как размер других модулей в таких системах значительно превышает процессоры с очередным выполнением, что обязывает использовать при проектировании как можно меньше памяти. Из-за использования большого количества подмодулей площадь и энергопотребление таких процессоров стоят значительных вложений. Предсказатель переходов может быть одним из крупнейших по памяти модулей, поэтому его эффективная реализация – это критический фактор в процессорах с ВВ. К тому же внеочередное выполнение – это технология, которая позволяет гораздо сильнее поднять производительность процессора, чем ПП, поэтому производитель в первую очередь реализует именно ВВ. Сегодня сложно встретить ПП, который реализован для процессора с очередным исполнением, потому что от

ускорения короткого и простого конвейера почти нет смысла. Таким образом, при проектировании ПП важно понимать структуру ВВ и связанные с этой структурой проблемы. Использование данной технологии в паре с предсказателем переходов приводит к ряду проблем, которые необходимо решать отменой последних действий. Подобные решения обязывают к потреблению ещё большего количества логики, памяти и, как следствие, потребления энергии. В исследовании [17] перечисляются возможные оптимизации для уменьшения потребления энергии в случае откатов предсказаний. Подобные проблемы также усложняют проектирование ПП, а также выбор алгоритмов и конфигураций.

Предсказатель переходов – важный модуль процессора, который обеспечивает большой рост производительности ([26], [27], [28], [29]), поэтому улучшение процесса синтеза – это актуальная задача.

Подобные исследования регулярно проводятся, однако они имеют либо частный характер алгоритмов ([32] – выявление оптимальной конфигурации TAGE), либо ориентированы на сравнение дорогих реализаций, как нейронные предсказатели [33]. Данное исследование ориентировано на рассмотрение двухуровневых предсказателей и выявления близких к оптимальным конфигураций, так как нет стандартного метода их определения в процессорах, не требующих таких мощных алгоритмов, как TAGE или perceptron.

2 Предлагаемый метод синтеза

Для ускорения синтеза структур предсказателей предлагается метод, в рамках которого:

1) выделяются требования к предсказателю (формируются трассы вычислений для некоторой целевой нагрузки, фиксируется структура конвейера процессора, из этого получаем требования по точности предсказания);

2) исследуется набор распространенных предсказателей на неком стандартном тестовом наборе, из которого выявляются закономерности влияния тех или иных конфигураций на точность предсказания;

3) вычисляются близкие к оптимальным структуры предсказателей для целевой нагрузки и процессора.

Метод ускорения синтеза включает в себя следующие шаги (рисунок 10):

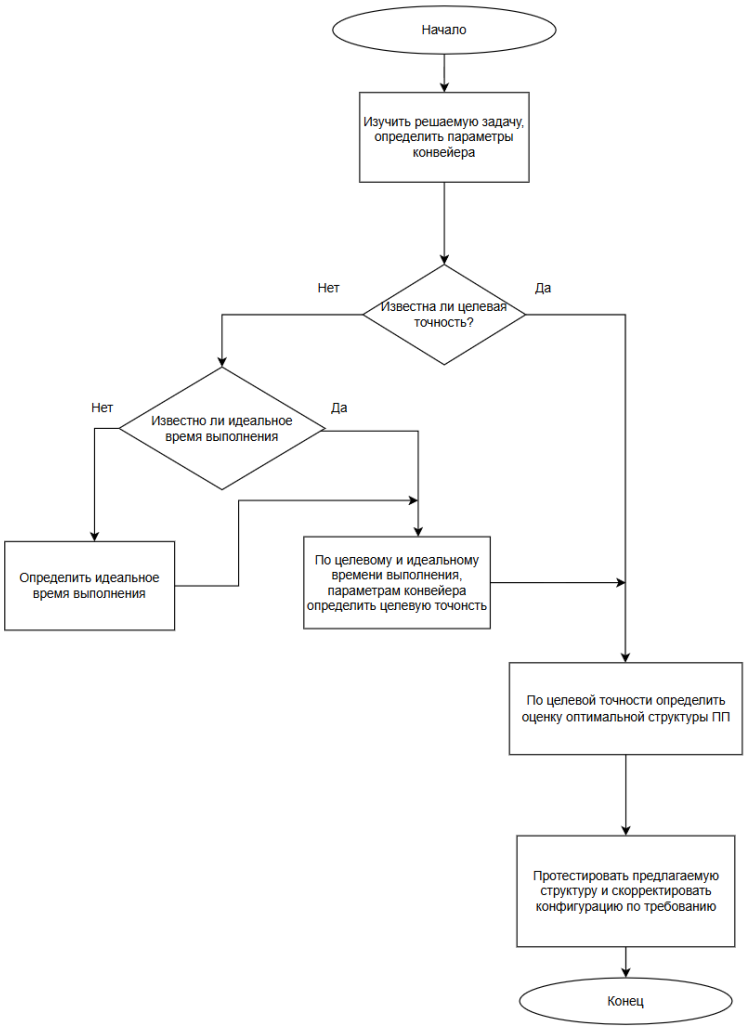

Рисунок 10 – Диаграмма метода

1) При определении целевого времени выполнения программы (максимально возможное время выполнения, исходящее из требований к процессору) необходимо сформировать набор тестов под конкретную задачу или воспользоваться стандартными наборами тестов. Если целевой стоит точность, то перейти к шагу 5.
2) Запуск теста без предсказателя и сбор статистики по инструкциям ветвления.
3) Запуск теста с предсказателем, который «знает» все правильные ответы, то есть произвести запуск без ошибок предсказателя и замерить идеальное время выполнения.
4) По параметрам конвейера, идеальному и целевому времени выполнения определяется целевая точность.
5) Используя целевую точность предсказателя, обратиться к таблице, содержащей результаты тестирования алгоритмов, а также к выводам, полученным на основе анализа алгоритмов.
6) Реализовать предлагаемый алгоритм и конфигурацию, выполнить тестирование. При несовпадении ожидаемого результата с полученным ориентироваться на результаты таблицы, как пропорциональные (многое зависит от конкретной реализации и решаемой задачи).

То есть план сводится к задачам определения целевой точности и выбора предсказателя на основе целевой точности по рекомендациям данного исследования. Для формирования таблицы, которая упоминается в методе, необходимо реализовать тестирование анализируемых алгоритмов, выявить близкие к оптимальным конфигурации, определить формулу перевода параметров конвейера и выполняемых трасс к целевой точности.

2.1 Реализация алгоритмов и тестового окружения

2.1.1 Тестовое окружение и тесты

Предсказатель переходов является модулем процессора. Процессор может из-за своей реализации вносить изменение работы предсказателя (задержки, переполнения буферов, очистки(flush) от других модулей и т.д..).

Своё исследование я провожу в виде модели предсказателя переходов, получающего на вход адрес инструкции, выдающего предсказание, а через несколько тактов получающий результат выполнения инструкции в отрыве от реализации остальных частей процессора. Таким образом запуск стандартных бенчмарков в данном исследовании не подходит, так как невозможно «по-честному» выполнять инструкции. Для исследования необходимо сформировать тестовое окружение, являющееся заглушками интерфейсов предсказателя, а также в тестовом окружении будет подсчитываться статистика предсказаний. Для тестов также необходимо разработать некоторые трассы, основанные на стандартных тестах, пригодных для работы данного предсказателя. Вся работа исполнена через system Verilog кода из синтезируемых конструкций, а также python-скриптов для анализа результатов.

Для получения трасс тестов были взяты классические integer benchmarks, которые были прогнаны через кластер процессора. На стадии выборки инструкции выясняется адрес очередной инструкции. На стадии декодирования инструкции становится ясна информация о том, является ли данная инструкция инструкцией ветвления. На стадии выполнения инструкции выясняется результат ветвления, если оно было. На стадии завершения выясняется фактический исход инструкции (реально ли она выполнилась или же обрабатывалась предыдущими стадиями спекулятивно). В код были добавлены триггеры, которые срабатывали на появление новых инструкций в конвейере и записывали информацию о инструкции в специальный .trace файл. Информация сохранялась, только если инструкция являлась инструкцией ветвления, а с финальной стадии было зафиксировано её выполнение. Файл с тестом был сформирован в виде:

адрес : результат ветвления

адрес : результат ветвления

…

По такой схеме были сформированы порядка трёхсот тестов, которые будут использоваться для анализа предсказателя.

Тестовое окружение состоит из следующих подмодулей: предсказатель переходов, модуль интерфейсов предсказателя и драйверов, модуля чтения тестов и записи результатов в .log файлы, подсчёта статистики, а также модули детального анализа работы алгоритмов.

Под драйверами подразумеваются структуры, в которые будут складываться запросы с адресами и «правильными» ответами на запрос, после чего ожидается результат проделанной работы предсказателя, как «решение совпало или нет». Реализовано через потактовый опрос готовности, а по готовности отправляется новый запрос, а результат предыдущего отправляется в модуль подсчёта статистики.

Модуль подсчёта статистики получает результат предсказаний (какое было предсказание, и верно оно или нет) и по выполнению теста выдаёт следующие результаты:

– Сколько инструкций выполнено;

– Сколько из них было Tkn, а сколько nTkn;

– Сколько из Tkn было предсказано верно, а сколько было верно предсказано из nTkn;

– Какой процент успешных предсказаний.

По данной статистике можно оценить общую результативность алгоритма, а также выяснить влияние количества Tkn и nTkn на результат.

Модуль детального анализа алгоритмов подразумевает собой сохранение состояний предсказателя по определённым событиям. Например, логика TAGE очень сложна, и правильность работы тяжело оценить по классическому способу работы с system Verilog кодом – временными формами. Поэтому события TAGE фиксировались в специальные отладочные файлы и содержали информацию вида:

– Такт : событие: на что повлияло

Такие слушатели были сформированы для большинства алгоритмов, и с их помощью были найден большой процент ошибок реализации.

2.1.2 Предсказатель переходов

Предсказатель переходов имел 4 базовые интерфейса (адрес инструкции, результат предсказания и сигналы их валидности), а также несколько промежуточных сигналов валидности. Например, TAGE выполняется более, чем за 1 такт, поэтому предсказатель выполнен в конвейерном стиле. На каждом этапе предсказатель подтверждает свою работу, что даёт понимание тестовому окружению о валидной работе предсказателя. В случае беспрерывного принятия запросов внутренние модули могут переполниться (о переполнении очереди далее), что приводит к остановке принятия запросов. В такой ситуации промежуточные сигналы валидности перестанут появляться на интерфейсах, что сигнализирует о необходимости повторной отправки запроса, пока все стадии не будут пройдены. Общая структура предсказателя приведена на диаграмме.

На рисунке 11 приведена самая полная конфигурация из анализируемых, для других конфигурация излишние блоки отключаются. Из запроса адрес инструкции попадает в TAGE и GShare, а в GH и PH попадает сам факт запроса для того, чтобы информация из них продвинулась далее по конвейеру. На следующий такт информация из GH, PH и GShare попадает в TAGE. На следующий такт TAGE вычисляет попадания по таблицам, вычисляет таблицу для предсказания. На последний четвертый такт готовое предсказание попадает на вывод предсказателя. Также это предсказание и адрес инструкции попадает в очередь для обучения. Когда результат будет известен и попадёт на вход предсказателя, модуль заберет из очереди предсказание и обучит алгоритмы. Подразумевается, что запросы не могут попасть в ложном порядке на обучение, следовательно реализация очереди уместна. Данная структура является классической для предсказателя переходов, так как слишком дорого прокидывать информацию о предсказании через весь конвейер процессора. Гораздо выгоднее лишь получить факт того, было предсказание верным или

нет, а всю необходимую для обучения информацию достать из очереди. При обучении сдвигаются необходимые регистры в GH, PH, пересчитываются счётчики в и таблицы TAGE, а также аллоцируются новые ячейки при ошибочном предсказании. Очередь может переполниться, если запросы приходили чаще, чем обучения. В таком случае стоит остановить принятие запросов, то есть не посылать сигналы валидности работы предсказателя на каждом такте.

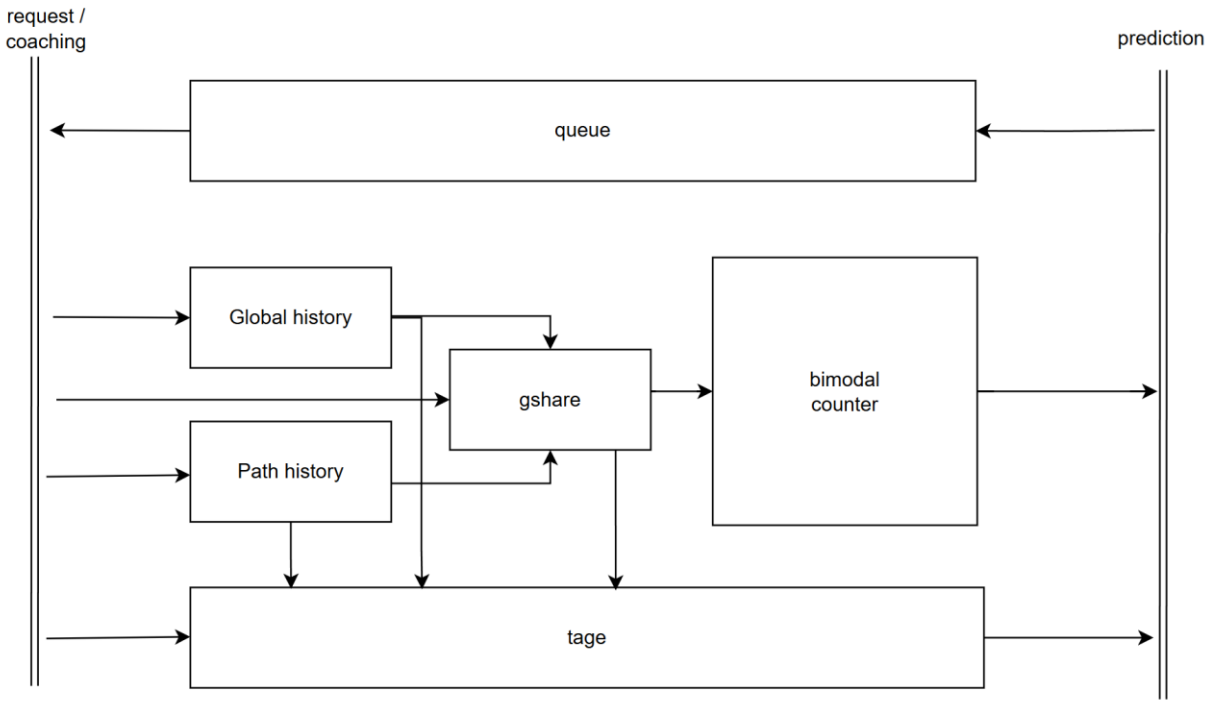

Рисунок 11 – Реализованный ПП

2.1.3 Ограничения

TAGE подразумевает большое количество памяти, что сложно реализовать в виде классических массивов system Verilog. При увеличении размера свыше предельной для языка симулятор перестает собирать информацию во временные формы, а также выполняет код слишком долго, так как пытается индексировать их напрямую (то есть добавить мультиплексоры на каждую из строчек массива, что сильно увеличивает минимально возможное время между тактами). Для решения этой проблемы необходимо либо реализовать подмодуль памяти с развязками, либо использовать небольшое количество памяти. В рамках данного исследования был выбран

второй вариант, из-за чего результат TAGE стоит анализировать с оговоркой маленького количества памяти.

В данном исследовании не были учтены инструкции вызова и возврата из функции, которые являются инструкциями ветвления, а также циклы, с которыми хорошо справляется предсказатель циклов. Предсказатель циклов, который после прохождения очередного цикла запоминает количество итераций, а при следующем появлении данного адреса выдает это количество Tkn, после чего выдает один nTkn. Количество итераций циклов часто превышает возможную длину историй, следовательно в нашем случае циклы будут плохо предсказываться (из-за большого количества Tkn адрес цикла будет всегда восприниматься, как Tkn, что приведёт к ошибке предсказания на каждый конец цикла). Вызовы и возвраты из функций также могут плохо повлиять на Во многих реализациях предсказателя за подобные инструкции отвечает отдельный модуль, который запоминает информацию о том, что они – инструкции безусловного ветвления, а на последующих выполнениях выдают Tkn, а также адрес возврата из функции с помощью стека. В данном исследовании не была рассмотрена проблема адреса возврата, поэтому данный модуль не был включен в рассмотрение, однако данные вызовы могут влиять на коллизии, занимать ячейки в таблицах TAGE, то есть ухудшать предсказания.

2.2 Сбор статистики и анализ

В данном разделе будут перечислены полученные численные показатели алгоритмов при различных конфигурациях, произведён анализ, предложен принцип выбора предсказателя и конфигурации. Запуск проводились на всех имеющихся тестах, но в таблицы для их удобочитаемости будут приведены только средние показатели, а также важные для анализа результаты.

2.2.1 Static

Алгоритм использует 1 бит памяти.

Как и ожидалось, тесты, состоящие в основном из циклов, выдают большое количество верных предсказаний (таблица 4), так как циклы выдают

nTkn только один раз за свой круг, что позволяет получать большую точность, не используя дополнительных кэшей.

Таблица 4 – Точность предсказаний

Максимум	**Минимум**	**Среднее**
81%	33%	61%

Средние показатели точности доказывают факт того, что код чаще содержит Tkn инструкции-ветвления в ходе выполнения.

Однако на ряде тестов точность падает слишком сильно, что значительно уменьшит производительность процессора, поэтому такое решение не подойдет в случаях, где неизвестна структура выполняемой задачи, а также где производительность имеет важное значение.

Важно отметить, что в данной реализации не использовались «подсказки» компилятора. С такими подсказками можно было бы повысить средний показатель точности предсказателя, однако это требует дополнительных разработок функций компилятора, которые бы анализировали код и оставляли эти «подсказки».

2.2.2 Last decision

Алгоритм использует 1 бит памяти.

Данный предсказатель имеет также один бит памяти, но используемый алгоритм выдает большее квадратическое отклонение, что увеличивает разброс результатов. Это происходит из-за того, что ряд задач, содержащих большое количество циклов или, например, сортировку пузырьком будут хорошо предсказываться данным алгоритмом, предсказываться даже лучше, чем статическим предсказателем. Однако мало тестов содержат длинные линии одинаковых результатов ветвления, что ведет к уменьшению точности предсказаний у большей части этих тестов. Данные результаты (таблица 5) говорят о том, что данный алгоритм стоит использовать только в тех случаях, где точно известен шаблон выполнения задачи, как волнообразный (подряд идут nTkn, потом подряд Tkn и т.д.). Тогда мы можем получить результаты выше, чем статический предсказатель, снова не используя кэш. Однако такие

знания о задаче ещё уже, то есть такое решение ещё хуже подходит на роль универсального предсказателя.

Таблица 5 – Точность предсказаний

Максимум	**Минимум**	**Среднее**	**Квдр. Откл.**
87%	21%	56%	Выше, чем у статики

2.2.3 Bim

Алгоритм использует память в размере:

$$c \times 2^x,$$

где $x -$ кол-во используемых бит от адреса инструкции, $c -$ количество бит у счётчика

В случае использования 1 бита алгоритм напоминает «последнее решение», только используется отдельный бит на каждый адрес. В ходе данных тестов использовались максимальное количество бит для адресации, чтобы исключить влияние коллизии на результат.

Посмотрев на статистику (таблица 6), можно заключить, что бимодальный счётчик является самым эффективным, считая эффективность, как добавленная память, делённая на полученный прирост. Важно заметить то, что при 4 битах точность падает. Некоторые шаблоны выполнения насыщают счётчики на значения, далёкие от середины. После таких случаев, если начинались короткие «волны» (Tkn, nTkn, Tkn, nTkn), тогда они все предсказывались одинаково, так как значение счётчика не опускалось до середины, а колебалось у значения, которое было при начале «волны». В случае счётчиков с количеством бит, меньших четырёх, такие «волны» предсказывались лучше.

Таблица 6 – Точность счётчиков в зависимости от используемого количества бит

Кол-во бит в счётчике	1	2	3	4
Средняя точность	63%	68%	69%	68%

Второй замер (таблица 7) рассматривает бимодальные счётчики (как лучшие из предыдущего анализа) и сравнивает количество бит адреса инструкций, используемых для индексации, то есть количества бимодальных счётчиков. В данном тесте всего 916 адресов инструкций.

Таблица 7 – «Bim» с разным количеством счётчиков

Ячейки	**Точность**	**Количество используемых ячеек**
4 бита	54.6%	8
6 бит	65.6%	32
8 бит	71.6%	128
10 бит	73.0%	440
12 бит	73.2%	777
14 бит	73.3%	891
16 бит	73.3%	916

Если использовать не все биты адреса инструкции, возникают коллизии, то есть последняя часть адресов некоторых инструкций может совпасть, что приведёт к ухудшению предсказаний. На графике (рисунок 12) прослеживается зависимость: если было взято совсем мало бит из адреса – это приведёт к почти 100% количеству коллизий (все инструкции будут попадать в одни и те же счётчики и портить предсказания друг другу); при увеличении количества счётчиков уменьшаются и коллизии. Однако важно заметить, что график нелинейный. Это даёт выявить важную точку в 4096 ячеек, то есть 12 бит адреса после которой значительное увеличение памяти не приносит больших результатов.

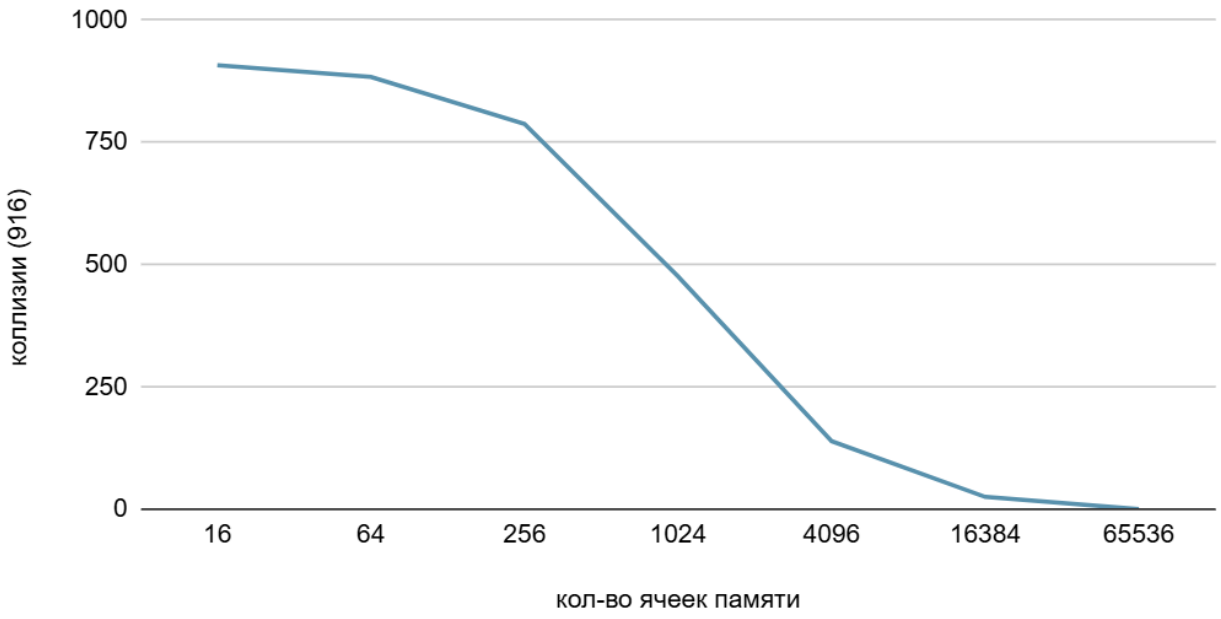

Рисунок 12 – График зависимости коллизий от количества используемых ячеек

Результаты, полученные на данном алгоритме, будут взяты далее, как оптимум:

– 2 бита на счётчик (бимодальные)

– 12 бит на индексацию таблиц.

2.2.4 Global history

Алгоритм использует память в размере:

$$2 \times 2^x + h,$$

где $x -$ кол-во бит адреса инструкции, $h -$ длина истории.

Здесь использовалось 12 бит для индексации. Так как классический алгоритм глобальной истории подразумевает использования конкатенации части адреса инструкции и истории, а всего количество бит ограничено – увеличение длины истории ведёт к уменьшению количества используемых бит для индексации от адреса инструкций. Это может привести к повышениям количества коллизий. Этот эффект мы и видим на результатах тестов, приведённые в таблице 8. 68% - результат, полученный при прошлом тестировании, где не используется история. При увеличении количества бит истории увеличивается и эффективность. Это говорит о том, что старшие биты адреса инструкций меньше влияют на точность (хотя без них коллизий

становится больше, что уменьшает точность, как было выявлено в предыдущем тесте), чем несколько бит истории. Однако если использовать слишком много бит истории, то это приводит к уменьшению точности, так как большая часть адреса уже не используется, а это приводит к большому количеству коллизий.

Таблица 8 – Точность предсказаний и использованное количество бит для истории

h	0 бит	2 бита	4 бита	6 бит	8 бит
Кол-во бит адреса инструкции	12 бит	10 бит	8 бит	6 бит	4 бита
Точность ср.	68%	75%	77%	77%	73%

Результатом данного пункта является то, что история важна в контексте предсказаний. Используя несколько дополнительных бит памяти, можно повысить точность на несколько процентов. Однако в случаях, где инструкции ветвления никак не связаны между собой (довольно редкие случаи), эта история может испортить предсказания, так как её биты используются вместо бит адреса инструкций, что может повысить коллизии в ряде случаев.

2.2.5 Path history

Алгоритм использует память в размере:

$$2 \times 2^x + h + 2^x,$$

где $x -$ количество бит, используемое для индексации, $h -$ длина истории.

PH очень похож по решаемой задаче с GH, однако он использует значительно больше памяти. В результатах (таблица 9) также можно заметить прирост точности, даже выше, чем у глобальной истории. Однако прирост на несколько процентов выше, а потребление памяти возросло порядка в h раз. Данной тест показывает, что локальная история может дать чуть больше, чем глобальная, но использует гораздо больше ресурсов. Может показаться, что тогда от данного алгоритма нет смысла, но этот алгоритм привносит большую пользу в алгоритмы gshare и tage, которые будут разобраны далее.

Здесь лишь стоит сделать пометку, что при сравнении классических реализаций gh и ph стоит использовать gh из-за явного преимущества по стоимости.

Таблица 9 – Точность предсказаний и длина локальной истории

h	0 бит	2 бита	4 бита	6 бит	8 бит
Кол-во бит адреса инструкции	12 бит	10 бит	8 бит	6 бит	4 бита
Точность ср.	68%	79%	82%	82%	81%

2.2.6 GShare

Алгоритм использует память в размере:

$$2 \times 2^x + h + (h \times 2^x),$$

где $x -$ количество бит от адреса инструкции, используемые для индексации, $h -$ длина истории. Второе и третье слагаемое используется или не используется в зависимости от конфигурации (с gh, с ph, с gh и ph)

В данном тесте используется 12 бит адреса инструкции (теперь не нужно уменьшать количество бит адреса, так как используется не конкатенация, а исключающее или), а также одинаковая длина истории для gh и ph. Результаты (таблица 10) показывают ещё больший прирост, чем показывал ph. Это происходит из-за того, что на индексацию влияет и адрес инструкции (все 12 бит), и глобальная, и локальная истории, а само исключающее или позволяет снизить количество коллизий.

Таблица 10 – Точность предсказаний и длина истории

h	2 бита	4 бита	6 бит	8 бит	10 бит	12 бит
Точность ср.	76%	79%	83%	85%	87%	88%

Результаты (таблица 11) данного теста показывают эффективность gshare по сравнению с конкатенацией. Причём эффективность ph возрастает значительно сильнее по сравнению с вариантом без исключающего или, что показывает важность использования ph в паре с данным алгоритмом. Важно заметить, что ранее мы не могли полноценно использовать и gh, и ph, так как

это забирало слишком много бит для индексации у адреса инструкций и приводило к большому количеству коллизий. GShare позволяет использовать до 12 бит всех трёх составляющих, что ещё сильнее позволяет увеличить точность предсказателя.

Таблица 11 – Точность предсказаний и количество используемых ячеек из 916 различных адресов

	Без истории	**gh(4)**	**Gshare gh**	**(4)ph**	**Gshare ph**	**(2)gh+ (2)ph**	**Gshare gh+ph**
Точность ср.	68%	77%	79%	79%	87%	76%	88%
Количество ячеек	777	791	840	814	882	790	893

2.2.7 TAGE

Алгоритм использует память в размере:

$$T \times (2^x \times (2 + 2 + 1 + A)) + h + (h \times 2^x),$$

где $x -$ количество бит, используемое для индексации, $h -$ длина истории, $A -$ средняя длина тэга для аллокации, $T -$ количество таблиц.

Сразу можно заметить значительное увеличение количества памяти, что делает данный алгоритм значительно дороже. Его структура и конфигурация значительно сложнее предыдущих алгоритмов. В данном исследовании не был проведён анализ данного алгоритма ввиду трудоёмкости, и, как уже было сказано, TAGE является самым точным алгоритмом, поэтому результаты (таблица 12), полученные от него, будут являться лишь ориентиром.

Таблица 12 – Точность предсказаний

Максимум	**Минимум**	**Среднее**
89%	97%	95%

Можно увидеть значительное увеличение точности алгоритма даже без детальной его настройки. В тесте использовались 10 таблиц, первая из которых не имела тэга, а все последующие использовали значения геометрического возрастания.

2.3 Метод ускорения синтеза ПП

В таблице 13 перечислены результаты, полученные в ходе тестирования различных алгоритмов. Стоит отметить, что с усложнением алгоритма растёт его точность и потребляемая память (кроме last decision, как уже было сказано, он применим только в случаях с известной структурой выполняемой программы). Прирост потребляемой памяти гораздо выше, чем прирост производительности, что говорит об уменьшении эффективности каждого отдельного бита памяти с усложнением алгоритма. Также стоит отметить о выводах, полученных в ходе тестирования:

– Статические предсказатели могут быть эффективнее двухуровневых (gh, ph, gshare), если известна структура исполняемой программы.

– Оптимальные параметры для счётчиков – 12 бит на адресацию, по 2 бита на сам счётчик (то есть бимодальный); если не использовать gshare, то 4 бита – это оптимальная длина истории.

– Ph не стоит использовать без связки с gshare, так как данный алгоритм дает совсем небольшой прирост по сравнению с gh, однако потребляет гораздо больше памяти.

– Tage может превышать потребление памяти в десятки раз, что делает его самым дорогим алгоритмом, но за эту цену предсказатель достигает самого высокого из рассматриваемых показателя точности, причём разброс минимума и максимума меньше всех, что делает этот алгоритм самым стабильным и точным.

Таблица 13 – Точность алгоритмов

Алгоритмы	**Точность**	**Память**
Static	61%	1
Last decision	56%	1
Bim	68%	8 Кб
Gh	77%	8 Кб+4
Ph	79%	24 Кб
GShare	88%	24 Кб+4
Tage	95%	>100 Кб

Теперь необходимо рассчитать зависимость потребностей от предсказателя переходов к его алгоритмам и конфигурациям. Предсказатель переходов защищает процессор от долгих простоев, пока требуемая инструкция загрузится из памяти, а также позволяет выполнять некоторые стадии инструкций спекулятивно (то есть выполнять, например, стадию декодирования, пока процессор ещё даже не уверен в том, что эту инструкцию нужно будет выполнять).

На рисунке 13 приведён ход выполнения некоторой программы. Слева направо – время, сверху вниз – выполняемые инструкции. У каждой инструкции есть различные стадии выполнения. Эти стадии можно начать выполнять заранее, если получить предсказание предсказателя переходов. При его ошибке процессор получит очистку (flush), то есть работу, выполненную спекулятивно, нужно отменить и выполнить правильную ветвь программы. Такой участок изображен на картинке, как неяркие прямоугольники. Последней инструкций, выполненной перед такой очисткой, была инструкция ветвления. Её адрес – 0x40656c (размер одной инструкции = 4). После данной инструкции процессор начал выполнять инструкции, находящиеся далее в памяти (0x406570, 0x406574 …). В один из дальнейших таков стало известно, что выполнялась не та ветвь, она очистилась, и процессор перешёл к правильной ветви по адресу 0x406558. Таким образом процессор потерял большое количество таков, в которые он ничего не делал. Чтобы их посчитать, нужно умножит ширину конвейера (сколько инструкций могут одновременно обрабатываться процессором) на его длину (среднюю длину одной инструкции в тактах), то есть неяркий кусок на картинке.

Рисунок 13 – Конвейер

Далее необходимо произвести несколько несложных вычислений:

Штраф за очистку:

$$F = s \times d,$$

где $s -$ ширина конвейера, $d -$ длина конвейера - сколько тактов в среднем выполняется инструкция.

Кол-во инструкций, которые вызовут очистку:

$$J \times (100\% - A),$$

где $J -$ среднее кол-во инструкций ветвления в задачах, $A -$ точность, которую необходимо получить от предсказателя переходов.

Кол-во тактов, которые процессор потеряет из-за очистки:

$$J \times (100\% - A) \times F.$$

Тогда

$$J \times (100\% - A) \times F + t_0, \leq t$$

где $t_0 -$ время, за которое выполняется задача без очистки, $t -$ максимальное время выполнения задачи, которое удовлетворяет запрос к производительности процессора.

Итоговое выражение для определения необходимой точности ПП

$$A \geq 1 - \frac{(t - t_0)}{J \times d \times s}. \quad (1)$$

Таким образом, чтобы выяснить необходимую точность предсказателя, необходимо запустить программу, сохранить все результаты ветвления, перезапустить программу с полученной информацией и замерить время выполнения, при котором все ветви угадываются верно. Далее подставить в формулу (1) все параметры процессора, желаемое время выполнения и идеальное время выполнения и получить необходимую точность. Далее можно воспользоваться статистикой, полученной в исследовании и на её основе выбрать предсказатель переходов. Однако этот вариант не будет однозначно верным, так как точность предсказателя зависит от задачи, тонкостей реализации, а также очистки могут происходить и не по вине неверного предсказания. Поэтому стоит воспользоваться результатами данного исследования как ориентиром, после чего перепроверить точность на решаемой задаче. При несовпадении точности воспринимать результаты, как пропорциональные к полученным (рисунок 14).

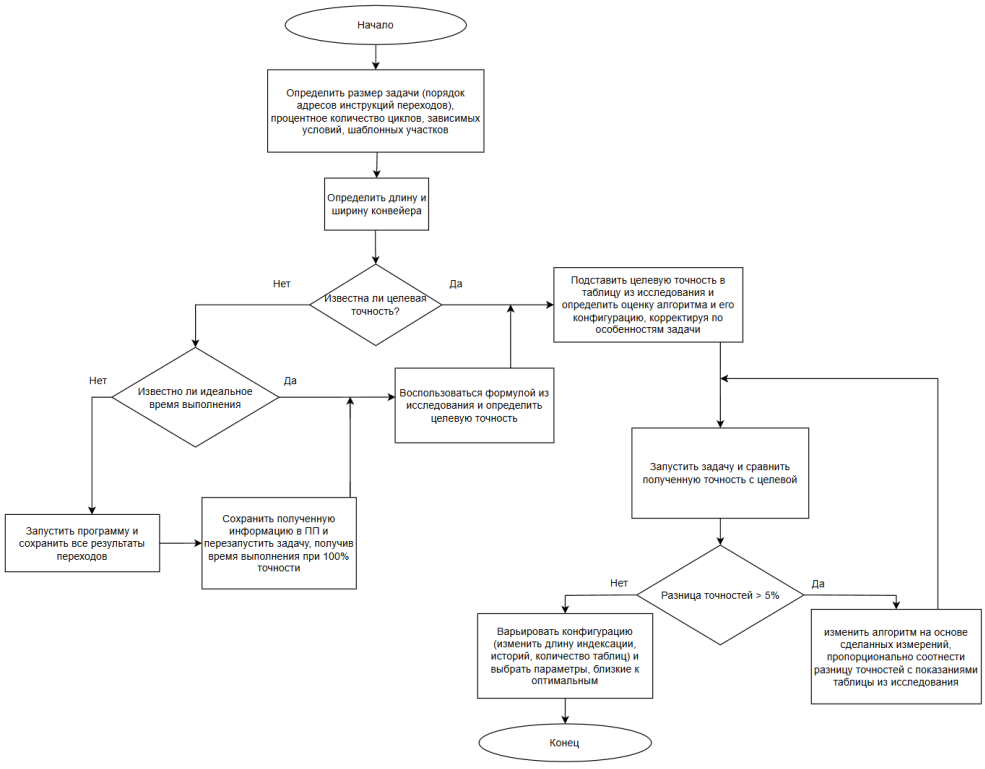

Рисунок 14 – Блок-схема метода

3 Экспериментальное подтверждение метода

Для проверки метода рассматривается набор тестов, не участвовавший в сборе статистики. Некоторые из них имеют явную предрасположенность для некоторых алгоритмов (содержание большого количества связанных условий и др.). В таблицах тестирования будут перечислены алгоритмы, используемые конфигурации алгоритмов и полученные точности. Для счётчиков рассматривается количество бит для индексации таблицы, а для историй рассматривается длина истории.

3.1 Первый тест

Рассматривается тест random_jump.trace. Данный тест не содержит предсказуемых шаблонов, так как прыжки зависят от псевдослучайных параметров (размер памяти предсказателей слишком мал, чтобы распознать закономерность псевдослучайных закономерностей). Исходя из особенностей теста, можно сразу сделать выбор в пользу статического алгоритма. Однако проверим результаты (таблица 14) на всех алгоритмах.

Таблица 14 – Алгоритмы и их точности

Алгоритмы	**Конфигурация**	**Точность**
Static		51%
Last decision		54%
Bim	10	58%
	12	59%
	14	59%
Gh	2	59%
	4	59%
	6	57%
Ph	2	58%
	4	59%
	6	56%
GShare		61%
Tage		61%

В данном случае все предсказатели показали результаты около 60%. Прирост обусловлен циклами (количество итераций также случайно), которые лучше предсказываются дорогими предсказателями, а может быть обусловлен шумом. Однако этот прирост невелик относительно прироста используемой

памяти, поэтому в данном случае стоит использовать статический предсказатель. Этот пример показывает, что использование рекомендаций из данного исследования экономят время, позволяя не реализовывать и не анализировать алгоритмы ПП для некоторых случаев.

3.2 Второй тест

Далее был рассмотрен integer benchmark, то есть обширный набор инструкций по работе с целочисленными данными. Данный тест по своей структуре не даёт чётких рекомендаций к выбираемому алгоритмы, поэтому стоит воспользоваться таблицей. Предположим, что целевая точность равна 70%. Ближайший результат на основе таблицы 15 – это bim (68%). Стоит воспользоваться рекомендуемыми параметрами, а именно 12 бит на адресацию, то есть 4096 ячеек.

Таблица 15 – Алгоритмы и их точности

Алгоритмы	**Конфигурация**	**Точность**
Static		48%
Last decision		53%
Bim	10	65%
	12	68%
	14	70%
Gh	2	70%
	4	71%
	6	67%
Ph	2	72%
	4	74%
	6	73%
GShare		82%
Tage		90%

В ходе тестирования данного алгоритма трассой int_bench.trace было получено 68% точности, что недостаточно для целевой точности. Тогда стоит воспользоваться алгоритмом, который даст результат немного точнее, чем bim. Gh с рекомендуемыми 4 битами истории уже показывает 71% точности, что подходит под целевую, а значит стоит воспользоваться данным алгоритмом. Остается лишь варьировать конфигурацию алгоритма (количество бит для

индексации таблиц, длина истории), чтобы получить минимально возможное количество памяти. Тестирование с различными параметрами показали, что использовать 12 бит адресации, из которых 2 бита – история, дают 70%. В ходе данных рассуждений была полученная целевая точность при использовании близким к оптимальным структурам предсказателя. Такую же точность даёт счётчик с 14 битами индексации, однако такая конфигурация использовала бы больше памяти.

Этот пример показывает, что, воспользовавшись таблицей из данного исследования, можно получить близкую к оптимальной структуре ПП.

3.3 Третий тест

Воспользуемся методом ускорения синтеза для трассы large_test.trace, в ходе выполнения которой встречается очень большое количество инструкций ветвления (400000). Данная характеристика говорит о том, что в ходе выполнения будут происходить постоянные коллизии. Проведём тестирование на основе целевого и идеального времени выполнения, а также параметров конвейера.

Частота процессора:

$$v = 2\text{Ггц},$$

то есть 2 млрд. тактов в секунду.

Время, за которое выполняется задача без очистки в ходе тестирования:

$$t_0 = 491{,}95 \times 10^{10} \text{ тактов}.$$

Максимальное время выполнения задачи, которое удовлетворяет запрос к производительности процессора:

$$t = 41 \text{ (минут)}$$

или

$$t = 41 \times 60 \times 2 \times 10^9 = 492 \times 10^{10} \text{ (тактов)}.$$

Длина конвейера (сколько тактов в среднем выполняется инструкция):

$$d = 1000(\text{тактов}).$$

Ширина конвейера:

$$s = 5 \text{ (линии)}.$$

Воспользуемся раннее выведенной формулой (1), и получим целевую точность

$$A \geq \frac{(492 - 491{,}95) * 10^{10}}{4 * 10^5 * 10^3 * 5},$$

тогда

$$A \geq 0{,}75.$$

Ориентируясь на таблицу 13, получен алгоритм Gh. Проведя тестирование данного алгоритма, получено 64% точности, чего не хватит, чтобы тест выполнился быстрее целевого времени. Зная контекст того, что тест производит много коллизий, улучшение алгоритма до ph или изменение конфигураций не даст необходимого прироста, так как даже пропорции это не даст больше 10% прироста. Алгоритм gshare с рекомендуемыми параметрами даёт 74%, что снова недостаточно, однако отрыв в 1% можно исправить изменением конфигурации. Количество коллизий можно снизить, увеличив количество бит для адресации. При использовании 13 бит для адресации программа выдает 75.1% точности, что входит в допустимый диапазон. Полученный алгоритм и конфигурация близки к оптимальным, однако результат можно улучшать, проведя дополнительные исследования. В данном тестировании был проведён дополнительный замер GShare с 13 битами, так как по условию задачи требовалось поднять точонтьс всего на 1%, что предполагает более детального анализа.

Выполнив тестирование на всех предсказателях (таблица 16), можно заключить, что, воспользовавшись формулой (1) и таблицей 13 из данного исследования, можно получить близкую к оптимальной структуру ПП.

Таблица 16 – Алгоритмы и их точности

Алгоритмы	**Конфигурация**	**Точность**
Static		35%
Last decision		47%
Bim	10	58%
	12	61%
	14	61%
Gh	2	62%

	4	64%
	6	63%
Ph	2	64%
	4	70%
	6	69%
GShare	12	74%
	13	75,1
Tage		88%

3.4 Результаты тестирования

Анализ полученных оценок и различных результатов алгоритмов показывает, что использование метода позволяет ускорить процесс разработки предсказателя переходов, не прибегая к перебору большого количества вариантов алгоритмов и их конфигураций. В экспериментах было достаточно реализовать предлагаемый ПП, а после лишь подобрать параметры под конкретную задачу, что гораздо проще полноценной разработки всех вариантов.

Важным вывод тестирования является выигрыш трудозатрат разработки. При использовании метода необходимо провести анализ, спроектировать 1 (или 2 в случае ошибочной оценки) алгоритма ПП и варьировать конфигурации для получения близкого к оптимальному ПП. В данной последовательности присутствует всего одно обязательно проектирование, а также одно дополнительное в случае, если точность отличается более, чем на 5 % (только в труднопрогнозируемых шаблонах ветвления), а также тестирование каждого из них и единоразовый подбор оптимальной конфигурации. Полный перебор обязывает каждый алгоритм спроектировать, протестировать, перебрать конфигурации и выбрать оптимальную структуру. То есть при анализе семи алгоритмов пришлось бы выполнить ориентировочно в 7 раз больше работы, чем с использованием метода при успешной оценке, и в 3.5 (7/2) раза при неуспешной оценке. Причём второй случай оценивается в 30% вероятности (основано на проведённых тестах). Таким образом, метод даёт на 86% меньше трудозатрат при успешной оценке

и 72% при ошибочной. При учёте вероятности успешной и неуспешной оценки общий выигрыш трудозатрат по сравнению с полным перебором равен 81%. В более правдоподобном случае, если разработчик провёл общий анализ задачи, то приходится перебирать около трёх алгоритмов. При успешной оценке это в 3 раза больше (уменьшение трудозатрат на 66%), а при ошибочной оценке в 1,5 раза (на 33%). Таким образом, выигрыш метода по сравнению с подходом анализа составляет 55% трудоёмкости.

ЗАКЛЮЧЕНИЕ

Данное исследование предлагает метод ускорения синтеза предсказателя переходов для процессоров с внеочередным исполнением инструкций. В рамках анализа были разобраны популярные алгоритмы, проанализированы некоторые статьи по предсказателю переходов, определены близкие к оптимальным конфигурации структур. Ориентируясь на результаты тестирования и сформулированные шаблоны выбора предсказателя переходов, можно выполнить проектирование, не используя полный перебор подходящих алгоритмов. Полученная методика позволяет получить оценку подходящего алгоритма и конфигурации для разных случаев, ускоряя процесс синтеза ПП.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. История предсказания переходов с 1 500 000 года до н.э. по 1995 год // Хабр URL: https://habr.com/ru/articles/337000/ (дата обращения: 05.03.2025).

2. Shen J. P., Lipasti M. H. Modern Processor Design: Fundamentals of Superscalar Processors. – Long Grove, Illinois: Waveland Press, Inc., 2013. – 830 с. – ISBN 978-1-4786-0783-0.

3. Hennessy J. L., Patterson D. A. Computer Architecture: A Quantitative Approach. Fifth Edition. – Amsterdam: Morgan Kaufmann, 2012. – 856 с. – ISBN 978-0-12-383872-8.

4. A. Choudhury, S. V. Siddamal and J. Mallidue, "An optimized RISC-V processor with five stage pipelining using Tournament Branch Predictor for efficient performance," 2022 International Conference on Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER), Shivamogga, India, 2022, pp. 57-60, doi: 10.1109/DISCOVER55800.2022.9974891.

5. S. Kim, E. Jo and H. Kim, "Low Power Branch Predictor for Embedded Processors," 2010 10th IEEE International Conference on Computer and Information Technology, Bradford, UK, 2010, pp. 107-114, doi: 10.1109/CIT.2010.59.

6. T. Jiang, N. Wu, F. Zhou, L. Zhao, F. Ge and J. Wen, "Design of a High Performance Branch Predictor Based on Global History Considering Hardware Cost," 2021 IEEE 4th International Conference on Electronics Technology (ICET), Chengdu, China, 2021, pp. 422-426, doi: 10.1109/ICET51757.2021.9451111.

7. Seznec A. A 256 Kbits L-TAGE branch predictor // IRISA/INRIA/HIPEAC. – 2007. – Presented at the 2nd Championship Branch Prediction (CBP-2).

8. T. Jiang, N. Wu, F. Zhou, L. Zhao, F. Ge and J. Wen, "Design of a High Performance Branch Predictor Based on Global History Considering Hardware Cost," 2021 IEEE 4th International Conference on Electronics Technology (ICET), Chengdu, China, 2021, pp. 422-426, doi: 10.1109/ICET51757.2021.9451111.

9. T. Jiang, N. Wu, F. Zhou, L. Zhao, F. Ge and J. Wen, "Design of a High Performance Branch Predictor Based on Global History Considering Hardware Cost," 2021 IEEE 4th International Conference on Electronics Technology (ICET), Chengdu, China, 2021, pp. 422-426, doi: 10.1109/ICET51757.2021.9451111.

10. A. Baniasadi and A. Moshovos, "Branch predictor prediction: a power-aware branch predictor for high-performance processors," Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors, Freiburg, Germany, 2002, pp. 458-461, doi: 10.1109/ICCD.2002.1106813.

11. T. Chen, P. Pan, G. Jiang and M. Ye, "Record Branch Prediction: An Optimized Scheme for Two-level Branch Predictors," in High Performance Computing and Communication \& IEEE International Conference on Embedded Software and Systems, IEEE International Conference on, Liverpool, United Kingdom United Kingdom, 2012, pp. 1526-1533, doi: 10.1109/HPCC.2012.223.

12. D. Grunwald, D. Lindsay and B. Zorn, "Static methods in hybrid branch prediction," Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192), Paris, France, 1998, pp. 222-229, doi: 10.1109/PACT.1998.727254.

13. D. Gope and M. H. Lipasti, "Bias-Free Branch Predictor," 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 2014, pp. 521-532, doi: 10.1109/MICRO.2014.32.

14. A. S. Fong and C. Y. Ho, "Global/Local Hashed Perceptron Branch Prediction," Fifth International Conference on Information Technology: New Generations (itng 2008), Las Vegas, NV, USA, 2008, pp. 247-252, doi: 10.1109/ITNG.2008.258.

15. S. A. I. Quadri and M. Z. Jahangir, "Design, Implementation and Performance Comparison of Different Branch Predictors on Pipelined-CPU," 2017 International Conference on Computer, Electrical & Communication Engineering (ICCECE), Kolkata, India, 2017, pp. 1-7, doi: 10.1109/ICCECE.2017.8526196.

16. Tse-Yu Yeh and Y. N. Patt, "A Comparison Of Dynamic Branch Predictors That Use Two Levels Of Branch History," Proceedings of the 20th Annual

International Symposium on Computer Architecture, San Diego, CA, USA, 1993, pp. 257-266, doi: 10.1109/ISCA.1993.698566.

17. B. Gregg and C. Teuscher, "Against the Current: Introducing Reversibility to Superscalar Processors via Reversible Branch Predictors," 2024 IEEE 15th International Green and Sustainable Computing Conference (IGSC), Austin, TX, USA, 2024, pp. 135-141, doi: 10.1109/IGSC64514.2024.00033.

18. M. Li, R. Xu, H. Zhang, L. Li and Y. Li, "CMA-BP: A Clustered Multi-Task Learning and Branch Attention Based Branch Predictor," 2024 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Kuching, Malaysia, 2024, pp. 1370-1376, doi: 10.1109/SMC54092.2024.10831163.

19. X. Yang, S. Mai and R. Bao, "MispredTable: A Side Branch Predictor to TAGE in Multithreading Processors," 2023 IEEE International Symposium on Circuits and Systems (ISCAS), Monterey, CA, USA, 2023, pp. 1-5, doi: 10.1109/ISCAS46773.2023.10181591.

20. D. J. Schlais and M. H. Lipasti, "BADGR: A practical GHR implementation for TAGE branch predictors," 2016 IEEE 34th International Conference on Computer Design (ICCD), Scottsdale, AZ, USA, 2016, pp. 536-543, doi: 10.1109/ICCD.2016.7753338.

21. A. Samara and J. Tuck, "The Case for Domain-Specialized Branch Predictors for Graph-Processing," in IEEE Computer Architecture Letters, vol. 19, no. 2, pp. 101-104, 1 July-Dec. 2020, doi: 10.1109/LCA.2020.3005895.

22. L. Zhang, N. Wu, F. Ge, F. Zhou and M. R. Yahya, "A Dynamic Branch Predictor Based on Parallel Structure of SRNN," in IEEE Access, vol. 8, pp. 86230-86237, 2020, doi: 10.1109/ACCESS.2020.2992643.

23. J. Fang, Y. He, J. Guo, M. Cai and Y. Hou, "Design and Implementation of Perceptron-Based Branch Predictor," 2023 4th International Conference on Computer, Big Data and Artificial Intelligence (ICCBD+AI), Guiyang, China, 2023, pp. 286-290, doi: 10.1109/ICCBD-AI62252.2023.00054.

24. A. Yin, T. Zhong, H. Li, S. Tang and Z. Zhao, "Language Model is a Branch Predictor for Simultaneous Machine Translation," ICASSP 2024 - 2024

IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Seoul, Korea, Republic of, 2024, pp. 9976-9980, doi: 10.1109/ICASSP48485.2024.10447486.

25. M. Zhao et al., "Branch Predictor Design for Energy Harvesting Powered Nonvolatile Processors," in IEEE Transactions on Computers, vol. 73, no. 3, pp. 722-734, March 2024, doi: 10.1109/TC.2023.3339977.

26. A. Ambashankar, G. Chandrasekar, C. A. R and S. S, "Exploration of Performance of Dynamic Branch Predictors used in Mitigating Cost of Branching," 2022 Third International Conference on Intelligent Computing Instrumentation and Control Technologies (ICICICT), Kannur, India, 2022, pp. 574-579, doi: 10.1109/ICICICT54557.2022.9917915.

27. L. Zhang, N. Wu, F. Ge, F. Zhou and M. R. Yahya, "A Dynamic Branch Predictor Based on Parallel Structure of SRNN," in IEEE Access, vol. 8, pp. 86230-86237, 2020, doi: 10.1109/ACCESS.2020.2992643.

28. N. Adiga, J. Bonanno, A. Collura, M. Heizmann, B. R. Prasky and A. Saporito, "The IBM z15 High Frequency Mainframe Branch Predictor Industrial Product," 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 2020, pp. 27-39, doi: 10.1109/ISCA45697.2020.00014.

29. D. Schall, A. Sandberg and B. Grot, "The Last-Level Branch Predictor," 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO), Austin, TX, USA, 2024, pp. 464-479, doi: 10.1109/MICRO61859.2024.00042.

30. A. D. Halke and A. A. Kulkarni, "CPU Branch Prediction Using Perceptron," 2021 IEEE 8th Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering (UPCON), Dehradun, India, 2021, pp. 1-5, doi: 10.1109/UPCON52273.2021.9667663.

31. M. Das, A. Banerjee, M. Chaudhuri and B. Sardar, "Shared Pattern History Tables in Multicomponent Branch Predictors With a Dealiasing Cache," in IEEE Embedded Systems Letters, vol. 12, no. 3, pp. 95-98, Sept. 2020, doi: 10.1109/LES.2019.2957512.

32. Q. Zhai, Z. Zhang and R. Xiao, "LLM Based End-to-end Branch Predictor Optimization Generator," 2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP), Hong Kong, Hong Kong, 2024, pp. 214-216, doi: 10.1109/ASAP61560.2024.00050.

33. I. Healy, P. Giordano and W. Elmannai, "Branch Prediction in CPU Pipelining," 2023 IEEE 14th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON), New York, NY, USA, 2023, pp. 0364-0368, doi: 10.1109/UEMCON59035.2023.10316163.

ПРИЛОЖЕНИЕ А

Листинг разработанного предсказателя переходов

```
module bpu
(
  input  logic                                    clk,
  input  logic                                    rst_n,

  input  logic                                    bpu_req_i,
  input  logic [PC                       -1:0] bpu_addr_i,

  input  logic                                    bpu_flush_i,

  output logic                                       bpu_b1_val_o,
  output logic                                       bpu_b2_val_o,
  output logic                                       bpu_b3_val_o,
  output logic                                 bpu_b4_val_o,
  output logic                                       bpu_b4_pred_val_o,

  output logic [TAGE_IND                        -1:0] bpu_b3_tage_index_o,

  input  logic                                       bpu_update_i,
  input  logic [TAGE_IND                     -1:0] bpu_tage_ind_i,
  input  logic                                       bpu_taken_i,
  input  logic [PC                       -1:0] bpu_pc_i
);

int fd_tage, fd_result, fd_bim, fd_entrop;

initial fd_tage = $fopen("tage","w");
initial fd_result = $fopen("result","w");
initial fd_bim = $fopen("bim","w");
initial fd_entrop = $fopen("entrop","w");

localparam PART = 14;
localparam INDEX_PHR_PART = PART;
localparam INDEX_GHR_PART = PART;
localparam INDEX_ADDR_PART = PART;
localparam PHR_SIZE = 2**INDEX_PHR_PART;
localparam MEM_ADDR = PART;
localparam MEM_SIZE = 2**MEM_ADDR;
localparam TAGE_NUM = 12;
localparam TAGE_U_AGE = 2;
localparam TAGE_ADDR = PART;
localparam TAGE_TAB_SIZE = 2**PART;

int time_g;

logic b1,b2,b3,b4;
logic direction_b4;
logic [PC          -1:0] b1_addr,b2_addr,b3_addr,b4_addr;
logic [TAGE_IND  -1:0] bpu_tage_ind_ff, bpu_tage_ind_ff_2;

logic [1:0] mem [MEM_SIZE];
logic [INDEX_GHR_PART-1:0] ghr;
logic [INDEX_PHR_PART-1:0] phr [PHR_SIZE];
logic [MEM_SIZE-1:0] entrop;
logic addresses [longint];
int entrop_counter = 0;
int addresses_counter = 0;

logic [PC     -1:0] addr, addr_coach, addr_coach_ff;
logic [PC     -1:0] addr_b1, addr_b2;
```

```
logic [PC     -1:0] addr_tag, addr_tag_coach, addr_tag_coach_ff;
logic [INDEX_GHR_PART-1:0] buf_ghr [$];
logic [INDEX_GHR_PART-1:0] buf_ghr_head;
logic [INDEX_PHR_PART-1:0] buf_phr [$];
logic [INDEX_PHR_PART-1:0] buf_phr_head;
logic [INDEX_PHR_PART-1:0] phr_cur, coach_phr;

logic bpu_update_ff, bpu_update_ff_2;
logic bpu_taken_ff, bpu_taken_ff_2;
logic [PC     -1:0] bpu_addr_ff, bpu_addr_ff_2;
logic [PC     -1:0] bpu_pc_ff, bpu_pc_ff_2;
logic bpu_flush_ff, bpu_flush_ff_2;


initial time_g = 0;

always @(posedge clk, negedge rst_n) begin
  time_g++;
  if (!rst_n) begin
    b1 <= 0;
    b2 <= 0;
    b3 <= 0;
    b4 <= 0;

    b1_addr <= 0;
    b2_addr <= 0;
    b3_addr <= 0;
    b4_addr <= 0;

    ghr <= '0;

    entrop <= {MEM_SIZE{0}};

    for(int i=0; i<PHR_SIZE; i++) phr[i] <= '0;
    for(int i=0; i<MEM_SIZE; i++) mem[i] <= '0;

    bpu_update_ff <= 0;
    bpu_update_ff_2 <= 0;
    bpu_taken_ff <= 0;
    bpu_taken_ff_2 <= 0;
    bpu_pc_ff <= 0;
    bpu_pc_ff_2 <= 0;
    bpu_flush_ff <= 0;
    bpu_flush_ff_2 <= 0;
    bpu_tage_ind_ff <= 0;
    bpu_tage_ind_ff_2 <= 0;
    bpu_addr_ff <= 0;
    bpu_addr_ff_2 <= 0;

  end else begin
    b2 <= b1;
    b3 <= b2;
    b4 <= b3;

    b2_addr <= b1_addr;
    b3_addr <= b2_addr;
    b4_addr <= b3_addr;

    bpu_update_ff <= bpu_update_i;
    bpu_update_ff_2 <= bpu_update_ff;
    bpu_taken_ff <= bpu_taken_i;
    bpu_taken_ff_2 <= bpu_taken_ff;
    bpu_pc_ff <= bpu_pc_i;
    bpu_pc_ff_2 <= bpu_pc_ff;
```

```
    bpu_flush_ff <= bpu_flush_i;
    bpu_flush_ff_2 <= bpu_flush_ff;
    bpu_tage_ind_ff <= bpu_tage_ind_i;
    bpu_tage_ind_ff_2 <= bpu_tage_ind_ff;
    bpu_addr_ff <= bpu_addr_i;
    bpu_addr_ff_2 <= bpu_addr_ff;

  end
end

task saturation_inc (int i);
  if (mem[i]<2'b11) mem[i]++;
endtask

task saturation_dec (int i);
  if (mem[i]>2'b00) mem[i]--;
endtask


always @(posedge clk) begin
  if (bpu_req_i) begin
    b1 <= 1;
    b1_addr <= addr;
    buf_ghr.push_back(ghr);
    buf_phr.push_back(phr_cur);
  end
  else begin
    b1 <= 0;
  end
end

always @(posedge clk) begin
  buf_ghr_head = buf_ghr[0];
  buf_phr_head = buf_phr[0];
  if(bpu_update_i) begin
    buf_ghr.pop_front();
    buf_phr.pop_front();
  end
end

assign phr_cur = phr[$unsigned(INDEX_PHR_PART'(bpu_addr_i))];
assign coach_phr = phr[$unsigned(INDEX_PHR_PART'(bpu_pc_i))];

assign addr = $unsigned(INDEX_ADDR_PART'(bpu_addr_i))
                ^($unsigned(INDEX_GHR_PART'(ghr)))
                ^ $unsigned(INDEX_PHR_PART'(phr_cur));

assign addr_tag = addr;


assign addr_coach = $unsigned(INDEX_ADDR_PART'(bpu_pc_i))
                  ^($unsigned(INDEX_GHR_PART'(buf_ghr_head)))
                  ^ $unsigned(INDEX_PHR_PART'(buf_phr_head));

assign addr_tag_coach = addr_coach;

always @(posedge clk) begin
  if (bpu_update_i) begin
    $fwrite(fd_entrop, "%d:%h %h %d %d %b\n", time_g, bpu_pc_i, addr_coach,
entrop_counter, addresses_counter,
      (entrop[addr_coach] != 0 && !addresses.exists(bpu_pc_i)));
    if(entrop[addr_coach] == 0) entrop_counter++;
    entrop[addr_coach] <= 1;
    if(!addresses.exists(bpu_pc_i)) begin
```

```
      addresses_counter++;
    end
    addresses[bpu_pc_i] = 1;

    if (bpu_taken_i) saturation_inc(addr_coach);
    else saturation_dec(addr_coach);

    ghr <= {bpu_taken_i, ghr[INDEX_GHR_PART-1:1]};
    phr[$unsigned(INDEX_PHR_PART'(bpu_pc_i))]         <=         {bpu_taken_i,
coach_phr[INDEX_PHR_PART-1:1]};
  end
end


logic [TAGE_NUM-1:0] hit;
logic [TAGE_U_AGE-1:0] u_match [TAGE_NUM];
logic [1:0] bi_match [TAGE_NUM];
logic [1:0] bi [TAGE_NUM];

logic [1:0] bimodal_0 [TAGE_TAB_SIZE];
logic [TAGE_U_AGE-1:0] u_0 [TAGE_TAB_SIZE];

logic [TAGE_NUM-1:0] misspred;

assign hit[0] = 1;
assign u_match[0] = u_0[TAGE_ADDR'(addr)];
assign bi[0] = bimodal_0[TAGE_ADDR'(addr_b1)];

int index, index_ff;

int ret_index;


always @(posedge clk, negedge rst_n) begin
  if (!rst_n) begin
    ret_index         <= 0;
    index_ff          <= 0;
    addr_b1           <= 0;
    addr_b2           <= 0;
    addr_coach_ff     <= 0;
    addr_tag_coach_ff <= 0;
    for(int i=0; i<TAGE_TAB_SIZE; i++) begin
      u_0[i] <= 1;
      bimodal_0[i] <= '0;
    end
  end else begin
    index_ff <= index;
    ret_index <= bpu_tage_ind_i;

    addr_b1 <= addr;
    addr_b2 <= addr_b1;

    addr_coach_ff     <= addr_coach;
    addr_tag_coach_ff <= addr_tag_coach;

    if (bpu_req_i) bi_match[0] <= bimodal_0[TAGE_ADDR'(addr)];

    if (bpu_update_i) begin
      if(bpu_taken_i)           bimodal_0[TAGE_ADDR'(addr_coach)]           <=
bimodal_0[TAGE_ADDR'(addr_coach)] < 3 ? bimodal_0[TAGE_ADDR'(addr_coach)] + 1
: bimodal_0[TAGE_ADDR'(addr_coach)];
      else                                bimodal_0[TAGE_ADDR'(addr_coach)] <=
bimodal_0[TAGE_ADDR'(addr_coach)] > 0 ? bimodal_0[TAGE_ADDR'(addr_coach)] - 1
: bimodal_0[TAGE_ADDR'(addr_coach)];
```

```
      $fwrite(fd_bim, "ret %d:addr_coach %h addr_coach_sliced %h tkn %b ghr %b
phr %b\n",
        time_g,
        addr_coach,
        $unsigned(INDEX_ADDR_PART'(addr_coach)),
        bpu_taken_i,
        buf_ghr_head,
        buf_phr_head);

      if (bpu_taken_i == bimodal_0[TAGE_ADDR'(addr_coach)][1]) begin
        misspred[0] <= 0;
      end else begin
        misspred[0] <= 1;
      end
    end

    if (bpu_update_ff) begin
      u_0[TAGE_ADDR'(addr_coach_ff)] <= !misspred[0]
                                       ? u_0[TAGE_ADDR'(addr_coach_ff)]!=2'b11
                                         ? u_0[TAGE_ADDR'(addr_coach_ff)] + 1
                                         : u_0[TAGE_ADDR'(addr_coach_ff)]
                                       : u_0[TAGE_ADDR'(addr_coach_ff)]!=0
                                         ? u_0[TAGE_ADDR'(addr_coach_ff)] - 1
                                         : u_0[TAGE_ADDR'(addr_coach_ff)];

    end
  end
end

for(genvar i = 1; i<TAGE_NUM; i++) begin: gen_tage

  localparam val = i;
  logic [val-1       :0]  tag     [TAGE_TAB_SIZE];
  logic [1           :0]  bimodal [TAGE_TAB_SIZE];
  logic [TAGE_U_AGE-1:0]  u       [TAGE_TAB_SIZE];

  logic ret_hit;

  assign bi[i] = bimodal[TAGE_ADDR'(addr_coach_ff)];

  always_ff @(posedge clk, negedge rst_n) begin
    if (!rst_n) begin
      ret_hit <= 0;
      misspred[i] = 0;
      hit[i] <= 0;
      for (int j=0; j<TAGE_TAB_SIZE; j++) begin
        u[j] <= 0;
        bimodal[j] <= '0;
      end
    end else begin

      if (bpu_update_i) begin
        if    (u[TAGE_ADDR'(addr_coach)]>0    &&    tag[val'(addr_coach)]    ==
val'(addr_tag_coach)) begin
          ret_hit <= 1;
          if(bpu_taken_i)          bimodal[TAGE_ADDR'(addr_coach)]          <=
bimodal[TAGE_ADDR'(addr_coach)] < 3 ? bimodal[TAGE_ADDR'(addr_coach)] + 1 :
bimodal[TAGE_ADDR'(addr_coach)];
          else                                bimodal[TAGE_ADDR'(addr_coach)] <=
bimodal[TAGE_ADDR'(addr_coach)] > 0 ? bimodal[TAGE_ADDR'(addr_coach)] - 1 :
bimodal[TAGE_ADDR'(addr_coach)];
          if (bpu_taken_i == bimodal[TAGE_ADDR'(addr_coach)][1]) begin
```

```
            misspred[i] <= 0;
          end else begin
            misspred[i] <= 1;
          end
        end
        else begin
          ret_hit <= 0;
        end
      end

      if (bpu_update_ff) begin
        if(misspred[ret_index] && u[TAGE_ADDR'(addr_coach_ff)]==0) begin
          tag [val'(addr_coach_ff)] <= addr_tag_coach_ff;
          bimodal [TAGE_ADDR'(addr_coach_ff)] <= bi[ret_index];
          u[TAGE_ADDR'(addr_coach_ff)] <= 2;
        end
        if (ret_hit)
          u[TAGE_ADDR'(addr_coach_ff)] <= !misspred[i]
                                ? u[TAGE_ADDR'(addr_coach_ff)]!=2'b11
                                  ? u[TAGE_ADDR'(addr_coach_ff)] + 1
                                  : u[TAGE_ADDR'(addr_coach_ff)]
                                : u[TAGE_ADDR'(addr_coach_ff)]!=0
                                  ? u[TAGE_ADDR'(addr_coach_ff)] - 1
                                  : u[TAGE_ADDR'(addr_coach_ff)];
      end

      if (bpu_req_i) begin
        if(tag[val'(addr)] == val'(addr_tag) && u[TAGE_ADDR'(addr)]!=0) begin
          hit[i]      <= 1;
          u_match [i] <= u       [TAGE_ADDR'(addr)];
          bi_match[i] <= bimodal [TAGE_ADDR'(addr)];
        end
        else hit[i] <= 0;
      end
    end
  end

end: gen_tage

logic [TAGE_U_AGE-1:0] u_hit_max;

string u_str, u_temp;

always @(posedge clk) begin
  if (b2) begin
    $fwrite(fd_tage,"%d: addr: 0x%h table_index: %2d counter: %b ghr: %b phr:
%b\n",
      time_g,
      addr_b2,
      index,
      bi_match[index],
      ghr,
      phr[INDEX_PHR_PART'(bpu_addr_ff_2)]);
  end
  if (bpu_update_ff_2) begin
    u_str = "";
    for (int i=TAGE_NUM-1; i>=0; i--) begin
      if(hit[i]) begin
        u_temp.itoa(u_match[i]);
        u_str = {u_str, " ", u_temp};
      end
      else u_str = {u_str, " ?"};
    end
```

```
    $fwrite(fd_result, "%d: addr: 0x%h tn: %b mispred: %b index:%d hit: %b u:
%s\n",
      time_g,
      bpu_pc_ff_2,
      bpu_taken_ff_2,
      bpu_flush_ff_2,
      bpu_tage_ind_ff_2,
      hit,
      u_str);
  end
  if (b1) begin
    u_hit_max = 0;
    for(int i=0; i<TAGE_NUM; i++) begin
      if(hit[i] && u_match[i]>=u_hit_max) begin
        u_hit_max = u_match[i];
        index = i;
      end
    end
  end
end

always @(posedge clk, negedge rst_n) begin
  if (!rst_n) begin
    direction_b4 <= 0;
  end else begin
    // direction_b4 <= bi_match[index_ff][1]; // in case of tage test
    if (b3) begin
      $fwrite(fd_bim,    "bpu    %d:%h    %h    %b\n",    time_g,    b3_addr,
$unsigned(INDEX_ADDR_PART'(b3_addr)),
mem[$unsigned(INDEX_ADDR_PART'(b3_addr))]);
    end
    direction_b4 <= (mem[$unsigned(INDEX_ADDR_PART'(b3_addr))][1]);
  end
end

assign bpu_b1_val_o = b1;
assign bpu_b2_val_o = b2;
assign bpu_b3_val_o = b3;
assign bpu_b4_val_o = b4;

assign bpu_b4_pred_taken_o = direction_b4;
assign bpu_b3_tage_index_o = index_ff;

endmodule : bpu
```

ПРИЛОЖЕНИЕ Б

Листинг тестового окружения

```
`define PC 64
`define TAGE_IND 4
`include "tb_drv.sv"


`timescale 1ns/1ns
module tb ();

      logic clk;
      initial begin
            clk = '0;
            forever #(0.5) clk = ~clk;
      end

      logic                                          rst_n;

ret_if ret_iface(clk);
bpu_if bpu_iface(clk);
in_if  in_iface(clk);

bpu inst_bpu
      (
            .clk                                (clk),
            .rst_n                              (rst_n),

            .bpu_req_i              (in_iface.bpu_req_i),
            .bpu_addr_i             (in_iface.bpu_addr_i),
            .bpu_flush_i           (in_iface.bpu_flush_i),

            .bpu_b1_val_o                (bpu_iface.b1_val_o),
            .bpu_b2_val_o                (bpu_iface.b2_val_o),
            .bpu_b3_val_o                (bpu_iface.b3_val_o),
            .bpu_b4_val_o                (bpu_iface.b4_val_o),
            .bpu_b4_pred_taken_o         (bpu_iface.b4_pred_taken_o),
            .bpu_b3_tage_index_o         (bpu_iface.b3_tage_index_o),

            .bpu_update_i                (ret_iface.update_i),
            .bpu_tage_ind_i              (ret_iface.tage_ind_i),
            .bpu_taken_i                (ret_iface.taken_i),
            .bpu_pc_i                   (ret_iface.pc_i),
      );


logic [112:0] ghr;
logic [97:0] phr;

task update_hist(logic dir, Block bl);
    ghr = {ghr[111:0], dir};
    phr = {phr[96:0], bl.cfi_address[1]};
endtask


ret_driver drv_ret;
bpu_driver drv_bpu;
in_driver drv_in;

Block blocks [$];

    task load_trace( string path );
```

```
        int fd;
        Block bl;

        bl = new();
        fd = $fopen(path,"r");
        while (!$feof(fd)) begin
            bl.parse_from_log(fd);
            blocks.push_back(bl.copy());
        end
        $fclose(fd);
    endtask

Block bl_in_q [$];
Block bl_out_q [$];

bpu_t pred_in_q [$];
bpu_t pred_out_q [$];

int cfi_commited = 0;
int cfi_taken    = 0;
int cfi_not_taken = 0;

int predicted_taken = 0;
int predicted_not_taken = 0;
int predicted = 0;
int misp_dir = 0;

task print_statistics();
    real prd, mis_pred;
    prd  = real'(predicted);
    mis_pred = real'(predicted + misp_dir);
    $display("");
    $display("cfi_commited      %d   vs   %d",   cfi_commited,   cfi_taken   +
cfi_not_taken);
    $display("cfi_taken     %d", cfi_taken);
    $display("cfi_not_taken %d", cfi_not_taken);
    $display("predicted     %d", predicted);
    $display("predicted_t   %d", predicted_taken);
    $display("predicted_nt  %d", predicted_not_taken);
    $display("misp_dir      %d", misp_dir);
    $display("all           %d", predicted + misp_dir);
    $display("acuracy       %f", prd * 100 / mis_pred);
endtask

int tage_trace_fd;
int fd_out;

task flush(logic [PC -1:0] addr, logic req);
    if (req) begin
        drv_in.send_ret_flush();
        @(posedge clk);
        drv_bpu.flush();
        drv_in.send_addr(addr);
        @(posedge clk);
        pred_in_q.delete();
        pred_out_q.delete();
    end
endtask

task run();
    int commited = 0;
    logic [63:0] start_address;

    Block bl, curr_bl;
```

```
    bpu_t pred, pred_old;

    ghr = '0;
    phr = '0;

    @(posedge clk);
    curr_bl = blocks.pop_front();
    drv_in.send_addr(curr_bl.cfi_address);
    bl_in_q.push_back(curr_bl);

    forever begin
        @(posedge clk);
        if (bl_out_q.size() > 0 && pred_out_q.size() > 0) begin
            bl   = bl_out_q.pop_front();
            pred = pred_out_q.pop_front();
            cfi_commited += 1;
            predicted += bl.cfi_taken == pred.cfi_taken;
            misp_dir += bl.cfi_taken != pred.cfi_taken;
            if (bl.cfi_taken != pred.cfi_taken) drv_in.send_ret_flush();
            if (bl.cfi_taken) begin
                cfi_taken += 1;
            end else begin
                cfi_not_taken += 1;
            end
            if (pred.cfi_taken && bl.cfi_taken == pred.cfi_taken) begin
                predicted_taken += 1;
            end else if (~pred.cfi_taken && bl.cfi_taken == pred.cfi_taken)
begin
                predicted_not_taken += 1;
            end
            drv_ret.send(bl, pred, 1'b0, 1'b0);
            commited += 1;

            if (blocks.size() == 0) begin
                break;
            end

            curr_bl = blocks.pop_front();
            drv_in.send_addr(curr_bl.cfi_address);
            bl_in_q.push_back(curr_bl);
        end

        if (drv_bpu.box.num() > 0) begin
            drv_bpu.box.get(pred);
            pred_in_q.push_back(pred);
        end
    end
endtask

initial begin
    Block bl;
    bpu_t pred;
    forever begin
        @(posedge clk);
        if (pred_in_q.size() > 0 && bl_in_q.size() > 0) begin
            bl = bl_in_q.pop_front();
            pred = pred_in_q.pop_front();
            bl_out_q.push_back(bl);
            pred_out_q.push_back(pred);
        end

    end
end
```

```
string arr [$];
string tmp;
int fd_f;


initial begin
    rst_n <= 1'b0;
    drv_ret = new(bpu_iface);
    drv_bpu = new(bpu_iface);
    drv_in = new(bpu_iface);
    fork
        drv_ret.run();
        drv_bpu.run();
        drv_in.run();
    join_none
    #10
    rst_n <= 1'b1;
    #50
    fd_f = $fopen("files", "r");
    while (!$feof(fd_f)) begin
        $fscanf(fd_f, "%s", tmp);
        arr.push_back(tmp);
    end
    $fclose(fd_f);

    for (int i = 0; i < 1; i++) begin
        load_trace("trace/test.trace");
        run();
        $display("iteration %d", i);
    end
    print_statistics();

    $finish;
end


initial begin
    $vcdplusfile("dump.vpd");
    $vcdpluson(0, "tb");
    $vcdplusmemon();
end


endmodule : tb
```

ПРИЛОЖЕНИЕ В

Листинг драйверов тестового окружения

```
class Block;
    int            id;
    logic [PC-1:0] cfi_address;
    logic          cfi_taken;

    function Block copy;
        copy = new ();
        copy.cfi_address    = cfi_address;
        copy.cfi_taken      = cfi_taken;
        copy.id             = id;
        return copy;
    endfunction

    function void parse_from_log(int fd);
        $fscanf(fd, "ca=0x%h t=%b id=%d"
            , this.cfi_address, this.cfi_taken, this.id);
    endfunction
endclass : Block


typedef struct packed {
    logic cfi_taken;
    logic [TAGE_IND-1:0] tage_ind;
} bpu_t;


interface ret_if(input logic clk);
    logic                                    update_i;
    logic                       [TAGE_IND-1:0] tage_ind_i;
    logic                                    taken_i;
    logic                     [PC -1:0] pc_i;

    clocking mc @(posedge clk);
        output update_i, tage_ind_i, taken_i, pc_i;
    endclocking
endinterface


interface in_if(input logic clk);
    logic bpu_req_i;
    logic [PC -1:0] bpu_addr_i;
    logic bpu_flush_i;
    clocking mc @(posedge clk);
        output bpu_req_i, bpu_addr_i, bpu_flush_i;
    endclocking
endinterface


interface bpu_if(input logic clk);
    logic                                    b1_val_o;
    logic                                    b2_val_o;
    logic                                    b3_val_o;
    logic                                    b4_val_o;
    logic                                    b4_pred_taken_o;
    logic                       [TAGE_IND-1:0]   b3_tage_index_o;
    clocking mc @(posedge clk);
        input   b1_val_o,   b2_val_o,   b3_val_o,   b4_val_o,   b4_pred_taken_o,
b3_tage_index_o;
    endclocking
endinterface : bpu_if
```

```
class bpu_trans;
    Block bl;
    bpu_t  pred;
    bit        misp;
    bit        flush;

    function bpu_trans copy;
        copy = new();
        copy.bl = bl;
        copy.pred = pred;
        copy.misp = misp;
        copy.flush = flush;
        return copy;
    endfunction

endclass



class ret_driver;
   virtual bpu_if vif;

   mailbox #(bpu_trans) mbox;

    function new(virtual ret_if if0);
        vif = if0;
        mbox = new();
    endfunction

    task run();
        bpu_trans trans;
        int res;
        int timeout = 0;
        forever begin
            @(vif.mc);

            res = mbox.try_peek(trans);

            vif.mc.update_i           <= 1'b0;
            vif.mc.tage_ind_i           <= '0;
            vif.mc.taken_i              <= '0;
            vif.mc.pc_i                 <= '0;

            if (res == 1) begin
                vif.mc.update_i           <= 1'b1;
                vif.mc.tage_ind_i         <= trans.pred.tage_ind;
                vif.mc.taken_i       <= trans.bl.cfi_taken;
                vif.mc.pc_i               <= trans.bl.cfi_address;
                mbox.get(trans);
                timeout = 0;
            end else begin
                timeout += 1;
            end
        end
    endtask : run

    task send(Block bl, bpu_t pred, bit misp, bit flush);
        bpu_trans t = new();
        t.pred = pred;
        t.bl = bl;
        t.misp = misp;
        t.flush = flush;
        mbox.put(t.copy());
```

```
    endtask : send
endclass : ret_driver

class in_driver;
    virtual in_if vif;

    mailbox #(logic [PC -1:0]) force_box;
    mailbox #(bit) ret_flush_box;

    function new(virtual bpu_if if0);
        vif = if0;
        force_box = new();
        ret_flush_box = new();
    endfunction

    task run();
        int res;
        bit dummy;
        bit cfi_taken;
        logic [PC -1:0] force_addr;

        forever begin
            @(vif.mc);

            vif.mc.bpu_req_i <= 1'b0;
            vif.mc.bpu_flush_i <= 1'b0;

            res = force_box.try_get(force_addr);
            if (res == 1) begin
                vif.mc.bpu_req_i     <= 1'b1;
                vif.mc.bpu_addr_i    <= force_addr;
            end

            res = ret_flush_box.try_get(dummy);
            if (res == 1) begin
                vif.mc.bpu_flush_i <= 1'b1;
            end
        end
    endtask

    task send_force_addr(logic [PC -1:0] addr);
        force_box.put(addr);
    endtask

    task send_ret_flush();
        ret_flush_box.put(1'b1);
    endtask

endclass : in_driver


class bpu_driver;

    virtual bpu_if vif;
    mailbox #(bpu_t) box;
    mailbox #(bit) flush_box;
    bpu_t b1, b2, b3, b4, b5;
    logic p1, p2, p3, p4, p5;

    function new(virtual bpu_if if0);
        vif = if0;
        box = new();
        flush_box = new();
    endfunction
```

```
    task run();
        bit flush;

        forever begin
            @(vif.mc);
            flush = 0;
            flush_box.try_get(flush);
            if (flush == 1) begin
                {p1, p2, p3, p4, p5} <= '0;
                box = new();
            end else begin
                p2 <= p1; p3 <= p2; p4 <= p3; p5 <= p4;

                p1 <= vif.mc.b1_val_o;

                b1 <= '0;
                b2 <= b1;
                b3 <= b2;
                b4 <= b3;
                b5 <= b4;

                if (vif.mc.b1_val_o) begin
                    b1.cfi_taken        <= 'x;
                end

                if (vif.mc.b2_val_o) begin
                    p2 <= 1'b1;
                end else begin
                    p2 <= 1'b0;
                end

                if (vif.mc.b3_val_o) begin
                    p3                  <= 1;
                    b3.tage_ind         <= vif.mc.b3_tage_index_o;
                end else begin
                    p3 <= 1'b0;
                end

                if (vif.mc.b4_val_o) begin
                    b4.cfi_taken        <= vif.mc.b4_pred_taken_o;
                end

                if (p5) begin
                    box.put(b5);
                end
            end
        end
    endtask : run

    task flush();

        flush_box.put(1'b1);
    endtask : flush
endclass : bpu_driver
```