Генерация тестовых данных для системного функционального тестирования микропроцессоров с учетом кэширования и трансляции адресов

Е.Корныхин

Аннотация

В статье рассматривается задача генерации тестовых данных для системного функционального тестирования микропроцессоров (core-level verification), а именно задача построения тестовой программы по заданной ее абстрактной форме (тестовому шаблону) и начальному состоянию микропроцессора. Для решения этой задачи в работе предложен алгоритм, сводящий ее к задаче разрешения ограничений. При этом учитываются такие особенности микропроцессора, как кэширование и трансляция адресов.

1 Введение

Вычислительные системы играют все большую роль в процессах, от которых зависит здоровье и жизнь людей. Поэтому необходимо, чтобы вычислительные системы работали корректно. Базовым компонентом многих вычислительных систем являются микропроцессоры, выполняющие управляющие функции. Тестирование микропроцессоров является важной задачей, которой и посвящена данная работа.

В данной работе микропроцессор рассматривается как единая система, входными данными для которой являются машинные программы, загруженные в память (далее такие программы будут называться *тестовыми программами*). Эти программы исполняются, процесс исполнения протоколируется и затем анализируется. Для функционального тестирования (которому посвящена данная статья) важно лишь, правильно ли исполнена загруженная программа. Кроме того, такое тестирование можно проводить еще до выпуска микропроцессора с использованием симуляции его модели на языке Verilog или VHDL.

В статье [4] была предложена технологическая цепочка построения тестовых программ на основе модели микропроцессора. Цель генерации тестовых программ задается с помощью критерия тестового покрытия, выделяющего набор тестовых ситуаций для каждой инструкции микропроцессора. Генератору тестовых программ на вход подаются описания тестируемых инструкций и тестовых

ситуаций для них, возможные зависимости между инструкциями, а также параметры, управляющие генерацией (например, длина генерируемых последовательностей инструкций). В общих словах, технологическая цепочка состоит из следующих этапов:

1. генерируются всевозможные последовательности инструкций указанной длины;
2. для каждой последовательности инструкций строятся всевозможные множества зависимостей между ними с точностью до изоморфизма;
3. для каждого множества зависимостей комбинируются всевозможные тестовые ситуации. Получается тестовый шаблон, поскольку начальное состояние микропроцессора и значения параметров инструкций в тестовой программе еще не построены;
4. завершающим этапом генерации тестовой программы является генерация начального состояния микропроцессора (регистров, кэш-памяти, TLB и др. – это и есть *тестовые данные*), на котором реализуются заданные в тестовом шаблоне зависимости и тестовые ситуации.

В работе [4] последний этап был описан достаточно схематично. Организации именно этого этапа (генерации тестовых данных) посвящена данная работа. Практика показывает, что при использовании штатных средств и алгоритмов удается строить тестовые данные для шаблонов небольшого размера (до трех команд), однако для тестирования современных процессоров следовало бы использовать тестовые шаблоны большего размера. Например, при тестировании конвейера приходится использовать тестовые программы, размер которых сопоставим с размером конвейера, а это несколько десятков команд.

Входом генератора тестовых данных является тестовый шаблон и модель тестовых ситуаций. По ним генератор тестовых данных строит начальное состояние микропроцессора (т.е. начальные значения регистров, ячеек кэш-памяти, буфера трансляции адресов и пр.). Генератор тестовых данных использует разрешение ограничений над целыми числами, для которого разработаны эффективные алгоритмы [3]. Причем обычно разрешение таких ограничений удается провести в разумные временные рамки. С использованием такого генератора тестовых данных удается полностью провести технологическую цепочку построения тестовых программ для микропроцессоров. Однако в известных работах [13, 14] описывается генерация ограничений для шаблонов, использующих лишь регистры, а для инструкций работы с памятью приводятся алгоритмы, не учитывающие такие технологии, как кэширование и трансляция адресов (получение физических адресов по виртуальным). По сути в таких работах в виде ограничений кодируется изменений состояний каждого регистра и зависимости между

ними. Наивное применение подобных идей для инструкций работы с памятью приводит к очень сложным ограничениям [1], которые не удается разрешить за приемлимое время. В данной работе предложен алгоритм, который путем преобразует такие тестовые ситуации в очень простую форму – в ограничения вида равенство-неравенство адресов (с использованием также принципа «разделяй и влавствуй»).

Как следует из [4], тестовые шаблоны представляются последовательностью из кода инструкции, параметров и тестовой ситуации (для инструкций работы с памятью – указания попаданий-промахов в кэше, ограничения на вытесняемые и вытесняющие теги, ограничения на физические и виртуальные адреса). Пример описания тестового шаблона :

```
REGISTER ax : 32;REGISTER bx : 32;CONST offset : 16;
ADD ax, bx, bx @ overflow
LW ax, bx, offset @ noexception( l1Miss, l2Hit )
XOR bx, ax, bx @ noexc
```

В этом тестовом шаблоне три команды - ADD, LW и XOR. Шаблон начинается с объявления переменных с указанием их битовых длин. Модификатор (REGISTER или CONST) указывает семантику использования переменной. Тестовая ситуация указывается после знака «@»: тестовая ситуация первой команды - «overflow», второй команды - «noexception(l1Miss, l2Hit)» и третьей - «noexc». Тестовая ситуация для второй команды содержит аргументы - они более детально описывают тестовую ситуацию.

Генерация ограничений для тестового шаблона производится последовательно для каждой тестовой ситуации в нем. Для трансляции тестовой ситуации инструкций работы с памятью приходится использовать и предыдущие тестовые инструкции данного шаблона – в некоторых тестовых ситуациях они должны обращаться в одну «область памяти». Эта область памяти задается значением заданного диапазона бит в адресе (назовем такую область памяти *сетом*; внутри сета адреса определяются уникальным *тегом* [7]). Например, политика замещения LRU (Least Recently Used) формулируется в рамках сета: вытесняемые и вытесняющий теги лежат в одном сете и вытесняемый тег тот, к которому дольше всего не было обращений. Предлагается следующая последовательность шагов при генерации тестовых данных для шаблона:

1. распределить инструкции тестового шаблона по сетам;

[1] даже без учета трансляции адресов для кодирования состояния микропроцессора можно использовать формулу длиной порядка размера памяти ($mem_0 = var0 \wedge mem_1 = var1 \wedge ...$); каждое изменение производится по неизвестному индексу, поэтому при записи нового состояния микропроцессора приходится перебирать все возможные варианты ($mem[i] := x$ приводит к формуле $(i = 0 \wedge mem_0 = x \wedge mem_1 = var1 \wedge ...) \vee (i = 1 \wedge mem_0 = var0 \wedge mem_1 = x \wedge ...) \vee ...$), а если таких изменений несколько, то приходится рассматривать все возможные варианты значений индексов. Получающаяся формула имеет размер порядка $|L| \cdot 2^n$, где $|L|$ – размер памяти, а n – количество изменений памяти. В данной работе предложен метод кодирования изменений, приводящий к формуле размера порядка $|L| + n$.

2. внутри этого сета формулировать ограничения на физические адреса (теги);
3. сформулировать ограничения из трансляции адресов;
4. сформулировать и решить задачу разрешения ограничений для всего шаблона;
5. построить начальные состояния подсистем микропроцессора.

Оставшаяся часть статьи организована следующим образом. Вначале будет дан обзор работ по системному функциональному тестированию микропроцессоров, затем в части 3 будет дан алгоритм для первых двух шагов алгоритма (в рамках которого будет осуществлено преобразование тестовых ситуаций в кэше в равенство-неравенство адресов), в части 4 будет дан алгоритм для остальных шагов.

2 Обзор работ по системному функциональному тестированию микропроцессоров

В настоящее время в практике тестирования микропроцессоров распространены следующие подходы к построению тестовых программ:

- *ручная разработка тестовых программ* хоть и практически неприменима для полного тестирования микропроцессора, всё же может применяться для тестирования особых, крайних случаев.
- *тестирование с использованием кросс-компиляции* применяется часто из-за невысокой сложности его проведения: после согласования спецификации микропроцессора можно начинать делать кросс-компилятор, а код, предназначенный для кросс-компиляции, уже готов. Однако гарантировать полноту такое тестирование не может.
- *случайная генерация тестовых программ* применяется так же часто в силу простоты автоматизации. Такие тестовые программы позволяют быстро обнаружить простые ошибки, однако опять же не гарантируют полноты тестирования. Разрабатываются и более хитрые варианты случайной генерации [8].
- *случайная генерация тестовых программ на основе тестовых шаблонов* предполагает разделение процесса генерации тестовой программы на два этапа: на первом подготавливаются *тестовые шаблоны* – абстрактные представления тестовых программ (вместо указания конкретных параметров инструкций в тестовых шаблонах указывается ограничения на параметры) – а на втором этапе по тестовым шаблонам генерируются тестовые программы.

Второй этап включает в себя *генерацию тестовых данных*, т.е. генерацию начальных значений регистров, ячеек кэш-памяти, строк TLB и т.д.

В решении задачи генерации тестовых данных можно выделить следующие подходы:

- комбинаторные техники;
- с использованием решения задачи ATPG;
- с использованием разрешения ограничений.

Комбинаторные техники применимы в случае простых тестовых шаблонов. Ограничениями на переменные в таких тестовых шаблонах являются указания домена. Все значения в домене равноправны и могут появиться в тестовой программе. Однако данная техника требует доведения всех ограничений в тестовом шаблоне до простой формы (ограничение домена переменной), что удается сделать не всегда. В работе исследователей из Fujitsu Lab. [9] предлагается описывать тестовые программы в виде выражений (Test Specification Expressions, TSE) и описывать инструкции микропроцессора на языке ISDL. Специальный генератор строит тестовые программы, удовлетворяющие TSE. Kohno и Matsumoto [10] рассматривают задачу верификации конвейерных микропроцессоров, сводя ее к генерации тестовых программ. В процессе своей работы генератор строит тестовый шаблон, к которому применяются также комбинаторные техники. Доменами переменных являются множество регистров и множество констант, допустимых в качестве значений параметров инструкций.

Исследователи из Politecnico di Milano [12] предложили генеровать тестовые данные с использованием *техник решения задачи ATPG* (Automatic Test Pattern Generation). ATPG – задача поиска значений входных сигналов («векторов») схемы с целью поиска ее некорректного поведения. ATPG чаще применяется для модульного тестирования, если известна RTL-модель микропроцессора. Задача ATPG известна давно и для ее решения существуют (в том числе коммерческие) инструменты. Для применения ATPG при генерации тестовых программ необходимо, чтобы RTL-модель микропроцессора была готова к моменту генерации тестовых данных. Кроме того, использование такой методики именно для функционального тестирования ограничено, поскольку тесты на функционирование микропроцессора приходится строить с учетом модели спроектированного микропроцессора, которая сама же при этом будет и тестироваться.

Наиболее впечатляющих результатов достигают инструменты, использующие для генерации тестовых данных *разрешение ограничений*. Ограничение с логической точки зрения то же, что и предикат, а задача разрешения ограничений – то же, что и задача выполнимости системы предикатов, но для решения этой задачи применяются специальные алгоритмы [3]. В работе [14] исследователей из

Китайского Национального Университета технологий безопасности описывается инструмент MAATG. Тестовый шаблон для него может содержать лишь ограничения равенства или неравенства значений и указание области значений переменной. Для задания архитектуры микропроцессора используется описание на языке EXPRESSION. Другой инструмент – Genesys-Pro [13] – позиционируется компанией IBM как разработка, впитавшая лучшее из разработок последних 20 лет. Тестовые шаблоны позволяют задавать тестовые программы переменной длины. Для любой инструкции в тестовом шаблоне может быть указана эвристика для выбора значений параметров [6]. Среди возможных эвристик есть и эвристики на события в кэш-памяти и при трансляции адресов. Однако в известных работах не раскрывается содержание таких эвристик, что не дает возможности понять эффективность генерации программ, нацеленных на тестирование памяти. Система команд микропроцессора должна быть описана в виде ограничений (constraint net) на операнды, код операции, что не является естественным описанием поведения инструкции, особенно если в рамках нее выполняется несколько последовательных вычислений на основе параметров инструкции. Для генерации параметров очередной инструкции Genesys-Pro использует уже построенную тестовую программу и состояние микропроцессора, которое известно полностью. Этот подход обеспечил масштабируемость на большие тестовые шаблоны, но и привел к необходимости использования механизма возврата (backtracking), если выбрать параметры для очередной инструкции.

В данной работе при решении задачи генерации тестовых данных также используется разрешение ограничений. В отличие от MAATG тестовые шаблоны могут содержать не просто ограничения равенства или неравенства, а более сложные ограничения, например, кэш-промах. А по сравнению с Genesys-Pro в данной статье делается попытка транслировать тестовый шаблон в ограничения целиком [2]. При этом отпадает необходимость в механизме возврата [3]. Особенностью тестовых шаблонов, получаемых в рамках [4], является фиксация для каждой инструкции регистров-параметров. Для таких шаблонов Genesys-Pro будет работать крайне неэффективно, поскольку теряется возможность с помощью выбора параметров «подогнать» исполнение очередной инструкции под заданные в тестовом шаблоне для нее события. На тестовых шаблонах из [4] Genesys-Pro будет работать следующим образом: выберет некоторое начальное состояние микропроцессора, начнет исполнять тестовый шаблон (поскольку на-

[2]Известно, что задача разрешения ограничений (т.е. задача выполнимости) NP-полна. Это означает, что для больших тестовых шаблонов предлагаемый в данной статье метод может быть не столь эффективным. Однако действительно длинные тестовые шаблоны в практике тестирования микропроцессоров применяются редко.

[3]Из-за этого качественно меняется разрешаемая система ограничений. Genesys-Pro сводит общую задачу к множеству задач, на порядок меньшей сложности. Кроме того, в данной статье предлагается более технологичный метод построения тестовых данных: описание архитектуры микропроцессора может быть получено из стандарта архитектуры микропроцессора и представляет собой понятное для человека императивное задание.

чальное состояние ему известно), но как только дойдет до инструкции, которая будет исполнена не так, как требуется в шаблоне, Genesys-Pro сделает возврат в самое начало, а именно ему придется выбрать другое начальное состояние микропроцессора и весь процесс запустить заново. Такой процесс генерации тестовых данных слишком неэффективен. Для задания схемы трансляции адресов в Genesys-Pro предлагается использовать подход DeepTrans [5]. Однако по имеющимся работам невозможно сделать вывод о том, как такая схема трансляции адресов отображается в ограничения [4]. Кроме того попытка наивного переноса идей из представленных в обзоре инструментов (кодирование изменений состояния каждого регистра и зависимостей между ними в виде ограничений) для инструкций работы с памятью приводит к очень сложным ограничениям [5], которые не удается разрешить за приемлемое время.

3 Преобразования тестовых ситуаций в кэше в равенство - неравенство адресов

Согласно описанному алгоритму генерации тестовых данных сначала происходит распределение тестовых ситуаций инструкций работы с памятью по сетам, т.е. каждой такой тестовой ситуации сопоставить число – номер сета. Сопоставление можно проводить любым способом. Главное – чтобы выбор привел к системе ограничений, имеющей решение. Например, можно сопоставить все тестовые ситуации одному сету или в разные.

После распределения тестовых ситуаций по сетам отдельно для каждого сета проводится следующий алгоритм, дающий ограничения «равенства-неравенства» тегов. Исходными данным для этого алгоритма является последовательность тестовых ситуаций в кэш-памяти, относящихся к одному сету. Для каждой тестовой ситуации указаны 1-2 тега (имеющийся тег либо пара из вытесняющего и вытесняемого тегов). Алгоритм состоит из двух шагов. На первом шаге составляются ограничения на конечные множества тегов, а на втором шаге эти ограничения разрешаются символьно (упрощаются) до искомого вида. Разрешение ограничений можно проводить любым из известных алгоритмов разреше-

[4] Авторы статьи используют при описании способа трансляции адреса элементы массива Memory с неизвестными индексами. Известно, что попытки построения ограничений, описывающих работу с элементами массива при неизвестных индексах, приводит к очень сложным ограничениям, разрешимость которых за приемлимое время можно поставить под сомнение.

[5] Даже без учета трансляции адресов для кодирования состояния микропроцессора можно использовать формулу длиной порядка размера памяти ($mem_0 = var0 \wedge mem_1 = var1 \wedge ...$); каждое изменение производится по неизвестному индексу, поэтому при записи нового состояния микропроцессора приходится перебирать все возможные варианты ($mem[i] := x$ приводит к формуле $(i = 0 \wedge mem_0 = x \wedge mem_1 = var1 \wedge ...) \vee (i = 1 \wedge mem_0 = var0 \wedge mem_1 = x \wedge ...) \vee ...$), а если таких изменений несколько, то приходится рассматривать все возможные варианты значений индексов. Получающаяся формула имеет размер порядка $|L| \cdot 2^n$, где $|L|$ – размер памяти, а n – количество изменений памяти. В данной работе предложен метод кодирования изменений, приводящий к формуле размера порядка $|L| + n$.

ния ограничений [3]. Ниже приведен псевдокод алгоритма. В нем учитывается, что при кэш-промахе происходит вытеснение согласно политике замещения LRU (Least Recently Used), хотя подобная техника применима и к другим политикам замещения. Текущее состояние сета моделируется множеством L. Кэш-попадание описывается принадлежностью тэга этому множеству, а кэш-промах – непринадлежностью вытесняющего тега этому множеству. Политика замещения LRU переформулирована в следующем виде: после последнего обращения к тегу до его вытеснения должны произойти обращения ко всем остальным тегам сета. xs – теги-переменные начального состояния сета. Итоговые ограничения вида равенство-неравенство адресов будут сформулированы на содержимое xs и на вытесняющие теги.

```
  procedure A( tt : test_template_for_set, xs : тег-list )
  returns C : constraint-set
  begin
    C := {}
    var L : тэг-set := {}
    для каждого (тега t из xs)
    begin
      добавить в C ограничение t ∉ L;
      L := L ∪ {t};
    end;
    для каждой (тестовой ситуации τ из tt)
    begin
      если τ есть кэш-попадание тега p, то
        добавить в C ограничение p ∈ L
      иначе если τ есть кэш-промах тега p с вытеснением тега q, то
      begin
        добавить в C ограничение q ∈ L;
        добавить в C ограничение p ∉ L;
        добавить в C ограничение lru( q,L,τ,tt );
        L := L ∪ {p} \ {q};
      end;
    end;
    упростить C;
  end,
  procedure lru( q : тэг, L : тэг-set, τ : тестовая_ситуация,
tt : test_template_for_set ) returns C : constraint
  begin
    C := ⊥;
    для каждого (τ'(p1) : кэш-попадания из tt с начала [6] до τ)
    begin
```

[6] «начало» есть добавление тегов-переменных начального состояния в сет, добавленное перед тестовым шаблоном

```
      var T : тэг-set := множество вытесняющих тегов и тегов попадания в
tt между τ′ и τ невключительно;
      C := C ∨ (q = p1) ∧ (L \ {q} = T);
    end;
  end
```

Приведенный алгоритм можно оптимизировать с целью уменьшения размера C с учетом следующих замечаний:

1. между последним обращением к тегу q и его вытеснением должно быть не менее $N-1$ обращений к любым тегам, где N – ассоциативность кэш-памяти (размер сета), т.е. в цикле процедуры lru можно пропустить кэш-попадания, отстоящие от τ ближе, чем $N-1$
2. порядок последних обращений к тегам повторяет порядок их вытеснения, т.е. в цикле процедуры lru можно пропустить кэш-попадания от начала tt до кэш-попадания тега, вытесняемого предыдущим кэш-промахом из цикла процедуры A
3. последовательность кэш-попаданий в цикле процедуры lru не должна проходить через более чем N вытеснений (в противном случае в этой последовательности обязательно должен был бы появиться вытесняемый тег), т.е. в этом цикле можно пропустить кэш-попадания от начала tt до кэш-промаха, отстоящего от τ ровно на N кэш-промахов
4. в процедуре lru можно генерировать C *ленивым* образом, т.е. для получения C проходить небольшое количество итераций цикла, затем возвращаться в алгоритм A; если такой C не дал решения, вернуться и пройти еще некоторое количество итераций (такой механизм, например, реализован в системах логического программирования с ограничениями [11])
5. если tt начинается с последовательности кэш-промахов, то несложно просчитать без разрешения ограничений, чему равны вытесняемые ими теги (например, первый вытесняемый тег равен первому добавлявшемуся тегу в сет, второй – второму и т.д.)

Рассмотрим пример последовательности тестовых ситуаций в кэше и поведение алгоритма A на этой последовательности (с учетом оптимизации). «hit x» означает кэш-попадание с тегом x, «miss x -> y» означает кэш-промах тега x с вытеснением тега y.

```
 hit x1 //начальное состояние сета
 hit x2 //начальное состояние сета
 hit x3       L = { x1, x2, x3, x4 }
 hit x4 //начальное состояние сета
[ miss x5 -> x6 ]
```

```
 hit x5   // добавленный hit
 hit x7       L1 = ( { x1, x2, x3, x4 } \ { x6 } ) \/ { x5 }
[ miss x8 -> x9 ]
```

и система для 4-х ассоциативной кэш-памяти (ассоциативность равна размеру сета):

$$\begin{cases} x1 \notin \{x2, x3, x4\}, x2 \notin \{x3, x4\}, x3 \neq x4 \\ L = \{x1, x2, x3, x4\} \\ x6 \in L \\ x5 \notin L \\ L1 = (L \setminus \{x6\}) \cup \{x5\} \\ \{x7, x9\} \subseteq L1 \\ x8 \notin L1 \\ \left[\begin{array}{l} \begin{cases} x9 = x3 \\ \{x4, x5, x7\} = L1 \setminus \{x9\} \\ x6 = x1 \\ \{x2, x3, x4\} = L \setminus \{x6\} \end{cases} \\ \begin{cases} x9 = x2 \\ \{x3, x4, x5, x7\} = L1 \setminus \{x9\} \\ x6 = x1 \\ \{x2, x3, x4\} = L \setminus \{x6\} \end{cases} \end{array} \right. \end{cases}$$

Решением такой системы могут быть следующие зависимости между тегами:

$$\begin{cases} \{x1, x2, x3, x4, x5\} - \text{ все разные} \\ x6 = x1 \\ x7 = x2 \\ x9 = x3 \\ x8 \neq x2, x8 \neq x3, x8 \neq x4, x8 \neq x5 \end{cases}$$

или

$$\begin{cases} \{x1, x2, x3, x4, x5\} - \text{ все разные} \\ x6 = x1 \\ x7 = x3 \vee x7 = x4 \vee x7 = x5 \\ x9 = x2 \\ x8 \neq x2, x8 \neq x3, x8 \neq x4, x8 \neq x5 \end{cases}$$

4 Построение ограничений для трансляции адресов

Оставшиеся шаги алгоритма генерации тестовых данных включают генерацию ограничений для тестового шаблона из механизма трансляции адресов, разрешение получившихся ограничений и формирование начального состояния подсистем микропроцессора. В статье рассматривается способ трансляции адресов, принятый в стандарте микропроцессоров MIPS [1].

Инструкция обращения в память в тестовом шаблоне может быть снабжена тестовой ситуацией трансляции адресов. В данной статье рассматривается трансляция адресов с помощью TLB (Translation Lookahead Buffer) [2]. Такая тестовая ситуация может указывать на попадание или промах в кэше TLB или свойства строки TLB, задействованной для трансляции адреса. Модель строки TLB состоит из следующих полей:

- маска – целое число (само поле содержит последовательность единичных битов в количестве, равном удвоенному значению этого поля);
- регион (r) – 2 бита (диапазон бит виртуального адреса с rend до rstart);
- старшие биты номера виртуальной страницы (vpn2) (диапазон бит виртуального адреса с vend до vstart);
- флаг глобальности (g) – 1 бит;
- идентификатор адресного пространства (asid);
- номер физического сегмента (pfn) дважды для четной и нечетной виртуальной страницы (pfnend и pfnstart – диапазон номеров бит физического адреса, в который при трансляции адресов помещается номер физического сегмента);
- другие флаги так же дважды для четной и нечетной виртуальной страницы.

Все строки TLB индексированы. При успешном обращении к TLB должна подходить только одна строка.

Следующий псевдокод описывает шаг формулирования ограничений из трансляции адресов общего алгоритма генерации тестовых данных. На вход он получает тестовый шаблон и информацию о равенстве-неравенстве тегов и сетов с предыдущего шага генерации тестовых данных. На выходе получается набор ограничений. Впоследствии он будет добавлен к ограничениям с предыдущего шага алгоритма генерации тестовых данных и к ограничениям тестовых ситуаций еще не задействованных инструкций (например, тестовых ситуаций арифметических инструкций). Ограничения, которые строит этот алгоритм, сформулированы не только на переменных-регистрах и константах тестового шаблона, но и на некоторых других (маски, некоторые поля строк TLB).

Описанный в псевдокоде алгоритм B можно разбить на три последовательные части. В первой составляется и решается задача на (кэшируемые) индексы строк TLB (поскольку кэш TLB устроен так же, как кэш-память из одного сета), затем выявляются ограничения из тестовых ситуаций инструкций, работающих с одной и той же строкой TLB:

- совпадение битов виртуальных адресов, отвечающих полю «r»
- совпадение битов виртуальных адресов, отвечающих полю «vpn/2» с учетом поля «mask» (например, если поле «vpn/2» расположено с 40 по 13й биты, то $v_{40..13+m} = w_{40..13+m}$, где v и w – виртуальные адреса, а m – целочисленная форма поля «mask», а именно половина количества нулей в поле «mask»)
- ограничение на поля «g» и «asid»
- ограничения на бит четности номера виртуальной страницы, например, если при трансляции адресов двух инструкций происходит обращение к одной строке TLB, но физические адреса различаются (это можно понять на основе равенства-неравенства тегов и сетов), то биты четности разные

Затем для каждой задействованной строки TLB выбирается соответствующая ей инструкция из тестового шаблона. Для нее составляются ограничения с каждой инструкций из тестового шаблона, которой соответствует другая строка TLB. Ограничение выражает факт отличия строк TLB, а именно либо отличие бит виртуальных адресов, соответствующих полю «r» или полю «vpn/2» (с учетом маски), или бит «g» равен 0 с отличием поля «asid» от значения в регистре EntryHi.

В псевдокоде используются обозначения битовых операций:

- получение бита выражения e с индексом i – e_i;
- получение диапазона бит выражения e от индекса i до индекса j – $e_{i..j}$;
- битовая конкатенация выражений $e1$ и $e2$ – $e1||e2$.

Для простоты псевдокод приведен для случая, когда в тестовом шаблоне не встречается инструкция прямой записи в TLB по индексу – TLBWI – и инструкция изменения регистра EntryHi. Если такие инструкции встречаются, то при добавлении ограничений нужно аккуратно ввести новые переменные для измененных значений строк TLB и использовать их после инструкций записи в TLB.

```
  procedure B( tt : test_template_memory_only, tags : инструкция ↦ тэг, sets
: инструкций ↦ сет ) returns C : constraint-set
  begin
    var va: инструкция ↦ выражение;
    var mask: инструкция ↦ переменная;
    var g: инструкция ↦ переменная;
```

```
var asid: инструкция ↦ переменная;
var idxs: инструкция ↦ переменная;
var EntryHi: переменная;
var Ci : constraint-set := A( тестовые_ситуации_на_кэш_TLB( tt ) );
var indexes: инструкция ↦ Nat := resolve( Ci );
C := {};
для каждой( пары инструкций ip и iq из tt : indexes[ip] = indexes[iq] )
begin
  var vp := va[ip];
  var vq := va[iq];
  добавить в C ограничение vp_{rend..rstart} = vq_{rend..rstart};
  var m := mask[ip];      /*что равно mask[iq], т.к. речь идет о масках
одной и той же строки TLB */
  добавить в C ограничение vp_{vend..vstart+m} = vq_{rend..rstart+m};
  var gi := g[ip];      /*что равно g[iq], т.к. речь идет о битах глобальности
одной и той же строки TLB */
  var asidi := asid[ip];
  добавить в C ограничение gi = 1 ∨ gi = 0 ∧ asidi = EntryHi;
  var tp := tags[ip];
  var tq := tags[iq];
  var sp := sets[ip];
  var sq := sets[iq];
  var idxp := idxs[ip];
  var idxq := idxs[iq];
  если «тэг полностью входит в pfn» и tp ≠ tq, то добавить в C ограничение
vp_m ≠ vq_m;
  если «сет полностью входит в pfn» и sp ≠ sq, то добавить в C ограничение
vp_m ≠ vq_m;
  если «тэг частично входит в pfn» и tp = tq, то добавить в C ограничение
vp_m = vq_m;
  если «тэг полностью входит в pfn, а сет частично входит в pfn» и tp =
tq и sp = sq, то добавить в C ограничение vp_m = vq_m;
  если «pfn состоит только из тэга и сета» и tp = tq и sp ≠ sq, то добавить
в C ограничение vp_m ≠ vq_m;
  var D : выражение := разделить_на_части(
(tp||sp||idxp)_{pfnend..pfnstart} = (tq||sq||idxq)_{pfnend..pfnstart} );
  добавить в C ограничение (vp_m = vq_m) ⇔ D;
end
var iva : Nat ↦ инструкция := выбрать_виртуальный_адрес_для_индекса(
indexes, va );
для каждого( виртуального адреса v из va[iva[_] ∪ инструкции из tt с тестовой
ситуацией TLB_Refill или TLB_Invalid ])
begin
  для каждого( виртуального адреса w из vas : индекс инструкции у w ≠
индекс инструкции у v )
  begin
```

```
            var mw := маска строки TLB, с которой работает инструкция адреса
w;
            var gw := поле g строки TLB, с которой работает инструкция адреса
w;
            var asidw := поле asid строки TLB, с которой работает инструкция
адреса w;
            добавить в C ограничение v_{rend..rstart} ≠ w_{rend..rstart} ∨ v_{vpnend..vpnstart+mw} ≠
w_{vpnend..vpnstart+mw} ∨ gw = 0 ∧ asidw ≠ EntryHi ;
        end
    end
  end
```

Запись «тег полностью входит в pfn» означает, что биты физического адреса для поля pfn полностью включают (нестрого) в себя биты тега физического адреса. Запись «тег частично входит в pfn» означает, что либо есть такой бит тега физического адреса, который не входит в биты для поля pfn, либо биты тега и поля pfn совпадают. Аналогично с битами сета и соответствующими записями.

Запись «разделить на части$(t_1||s_1 = t_2||s_2)$» означает преобразование этого выражения к выражению $t_1 = t_2 \vee s_1 = s_2$ в случае, когда t_1 и t_2 имеют одинаковую битовую длину и s_1 и s_2 также имеют одинаковую битовую длину.

Структура iva получается выбором инструкции для каждого задействованного в тестовом шаблоне индекса строки TLB.

Рассмотрим пример небольшого тестового шаблона, на котором проиллюстрируем алгоритм B.

```
REGISTER x, y, z : 64;
CONST of : 16;
LW x, y, of
    @ noexc{ LoadMemory:l1Hit; AddressTranslation:tlbHit }
ADD z, x, y @ noexception
SW x, z, of
    @ noexc{ StoreMemory:l1Miss; AddressTranslation:tlbHit }
```

В тестовом шаблоне три инструкции: LW, ADD и SW. Задействованы 3 регистра (x, y, z) и одна константа (of). Тестовые ситуации noexc обозначают нормальный ход выполнения инструкции: виртуальный адрес вычисляется в виде суммы второго и третьего параметров (регистра и константы), затем делается трансляция адресов (комментарий к ее выполнению приводится после слова AddressTranslation) и, наконец, делается запись или чтение из памяти (комментарий к выполнению этой операции приводится после слова LoadMemory или StoreMemory). Таким образом, при исполнении первой инструкции должны произойти кэш-попадание в кэше первого уровня, а при трансляции адресов кэш-попадание в кэше TLB. При исполнении третьей инструкции должны произойти кэш-промах в кэше первого уровня и кэш-попадание в кэше TLB.

Составим для этого тестового шаблона ограничения согласно предложенным в статье алгоритмам. Назовем тег, соответствующий первой инструкции, t_1, а сет s_1. Соответственно для второй инструкции – t_2 и s_2. Первый шаг – распределить инструкции по сетам. Пусть $s_1 = s_2 = 0$. Следующий шаг – выявление ограничений на теги внутри каждого сета. В данном случае получается $t_1 \neq t_2$. Следующий шаг – выявление значений индексов строк TLB. Назовем индекс строки, с которой работает первая инструкция, i_1, а индекс для второй инструкции – i_2. Т.к. обе инструкции исполняются с кэш-попаданием в кэше TLB, то $\{i_1, i_2\} \subseteq \{$ индексы строк начального состояния кэша TLB $\}$. Пусть $i_1 = i_2 = 1$, а 1-я строка в TLB будет заполняться в последнюю очередь. Теперь выявляем ограничения на x, y, z, of, исходя из равенства сетов, неравенства тегов и равенства индексов строк TLB. Виртуальный адрес первой инструкции задается выражением $y + of$, виртуальный адрес второй инструкции – выражением $z + of$. Пусть микропроцессор таков, что поле r строки TLB занимает биты с 63го по 62й на виртуальном адресе, поле vpn занимает биты с 40го по 13й на виртуальном адресе, поле pfn занимает 30 бит, тег физического адреса занимает 20 бит, сет – 20 бит. Тогда получаются такие новые ограничения $(y+of)_{63..62} = (z+of)_{63..62}$. Пусть переменная m_1 будет обозначать маску первой строки TLB. Тогда получается такое новое ограничение $(y+of)_{40..13+m_1} = (z+of)_{40..13+m_1}$. Пусть переменные g_1 и $asid_1$ будут обозначать флаг глобальности и поле asid первой строки TLB соответственно. Тогда добавляется ограничение $g_1 = 1 \vee g_1 = 0 \wedge asid_1 = EntryHi$. Поскольку тег занимает биты с 63 до 44, а pfn – с 63 по 39, то тег полностью входит в pfn и так как $t_1 \neq t_2$, то добавляется такое ограничение $(y + of)_{m_1} \neq (z + of)_{m_1}$. Далее составляем и упрощаем выражение $(t_1||s_1||idx_1)_{63..39} = (t_2||s_2||idx_2)_{63..39}$, что эквивалентно $t_1||s_{1\{19..14\}} = t_2||s_{2\{19..14\}}$, что эквивалентно $t_1 = t_2 \wedge s_{1\{19..14\}} = s_{2\{19..14\}}$. Первый конъюнкт равен $\perp$, т.к. $t_1 \neq t_2$, а второй равен $\top$, т.к. $s_1 = s_2$. Таким образом, следует добавить ограничение $(y + of)_{m_1} = (z + of)_{m_1} \Leftrightarrow \perp$, которое только что было добавлено. Поскольку другие индексы не задействованы, то больше ограничений согласно приведенному алгоритму выделять не нужно. Остается добавить ограничение, описывающее инструкцию ADD, а именно $z = x + y$ и полученный набор ограничений на переменные – целые числа разрешать любым известным алгоритмом разрешения ограничений. Кстати, для этого тестового шаблона все полученные ограничения можно представить в виде задачи линейного программирования, что дает возможность применить для их решения соответствующие инструменты решения задачи линейного программирования.

5 Заключение

В статье рассматривалась задача генерации тестовых данных для системного функционального тестирования микропроцессоров, а именно задача построения тестовой программы по заданного для нее тестового шаблона. Для решения этой задачи в работе предложен алгоритм, использующий разрешение ограничений [3]. Предложенный алгоритм реализуется на базе системы логического программирования с ограничениями ECLiPSe [11] в качестве решателя ограничений на целые числа и конечные множества целых чисел. Алгоритм применяется в проекте по тестированию промышленного микропроцессора MIPS64 [1]. В будущем предполагается расширить спектр используемых политик кэширования за счет внедрения механизмов их описания.

Список литературы

[1] *MIPS64: Architecture for Programmers Volume II: The MIPS64 Instruction Set*.

[2] *MIPS64: Architecture for Programmers Volume III: The MIPS64 Privileged Resource Architecture*.

[3] Семенов А.Л. Методы распространения ограничений: основные концепции. *PSI'03/ИМРО — Интервальная математика и методы распространения ограничений*, 2003.

[4] Камкин А.С. Генерация тестовых программ для микропроцессоров. *Труды ИСП РАН*, 14(2):23–64, 2008.

[5] Y. Katz A. Koyfman A. Adir, R. Emek. Deeptrans - a model-based approach to functional verification of address translation mechanisms. *Microprocessor Test and Verification: Common Challenges and Solutions, 2003. Proceedings. 4th International Workshop on*, pages 3–6, 2003.

[6] L.Fournier E.Marcus M.Rimon M.Vinov A.Ziv A.Adir, E.Almog. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design and Test of Computers*, 21(2):84–93, Mar/Apr 2004.

[7] J. Hennesy D. Patterson. *Computer Organisation and Design: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design, 3rd edition, 2005.

[8] M. Sonza Reorda G. Squillero F. Corno, E. Sanchez. Automatic test program generation – a case study. *IEEE Design & Test, Special issue on Functional Verification and Testbench Generation*, 21(2):102–109, MArch-April 2001.

[9] K.Takayama F.Fallah. A new functional test program generation methodology. *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 76–81, 2001.

[10] N.Matsumoto K.Kohno. A new verification methodology for complex pipeline behavior. *Proceedings of the 38st Design Automation Conference (DAC'01)*, 2001.

[11] M.Wallace K.R.Apt. *Constraint Logic Programming Using Eclipse*. Cambridge University Press, 2007.

[12] F.Ferrandi D.Sciuto M.Beardo, F.Bruschi. An approach to functional testing of vliw architectures. *Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, pages 29–33, 2000.

[13] Y.Lichtenstein M.Rimon M.Vinov M.Behm, J.Ludden. Industrial experience with test generation languages for processor verification. *Proceedings of the 41st Design Automation Conference (DAC'04)*, 2004.

[14] Y.Guo G.Liu S.Li T.Li, D.Zhu. Maatg: A functional test program generator for microprocessor verification. *Proceedings of the 2005 8th Euromicro conference on Digital System Design (DSD'05)*, 2005.