

Генерация тестовых данных для тестирования механизмов кэширования и трансляции адресов микропроцессоров

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). Для решения этой задачи в работе предложен алгоритм, сводящий ее к задаче разрешения ограничений. В данной статье рассматриваются вопросы тестирования инструкций работы с памятью (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.

Как и для программных систем, для микропроцессоров может быть проведено *функциональное тестирование*, а именно тестирование, целью которого является проверка выполнения микропроцессором функциональных требований. Под функциональными требованиями *на системном уровне* (в противовес *модульному уровню*) понимается выполнение инструкциями микропроцессора семантики, заявленной в архитектуре микропроцессора [1]. При этом проверяемая семантика не должна содержать особенностей конкретной реализации в конкретном микропроцессоре, она не должна быть избыточной.

В рамках системного тестирования микропроцессор рассматривается как единая система, входными данными для которой являются машинные программы, загруженные в память (далее такие программы будут называться *тестовыми программами*). В рамках функционального тестирования эти программы исполняются, процесс исполнения протоколируется и затем анализируется. При этом важно лишь, правильно ли исполнена загруженная программа.

Тестовые программы могут быть построены на основе модели микропроцессора. При этом сначала тестовые программы систематически строятся в абстрактном виде (в виде *тестового шаблона*) – без конкретных параметров инструкций и без необходимой инициализации внутреннего состояния микропроцессора. Затем тестовые шаблоны уточняются до тестовых программ, при этом происходит выбор параметров инструкций и построение инструкций инициализации состояния микропроцессора перед исполнением тестового шаблона. Решение этой задачи для инструкций, затрагивающих лишь регистры, хорошо известно [16, 5, 9]. Однако модель состояния микропроцессора сильно усложняется, если тестовый шаблон включает инструкции, работающие с кэш-памятью [15], что усложняет и построение тестовых программ. Этой проблеме посвящена данная статья.

2 Обзор работ по системному функциональному тестированию микропроцессоров

В настоящее время в практике системного функционального тестирования микропроцессоров можно выделить 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. Сгенерированные таким образом тестовые программы позволяют быстро обнаружить простые ошибки, однако не гарантируют полноты тестирования. Разрабатываются и более сложные варианты случайной генерации [8];
- *случайная генерация тестовых программ на основе тестовых шаблонов* предполагает разделение процесса генерации тестовой программы на два этапа (см. рис. 1): на первом подготавливаются тестовые шаблоны – абстрактные представления тестовых программ (в тестовых шаблонах для параметров инструкций вместо значений указываются ограничения на значения) – а на втором этапе по тестовым шаблонам генерируются тестовые программы.

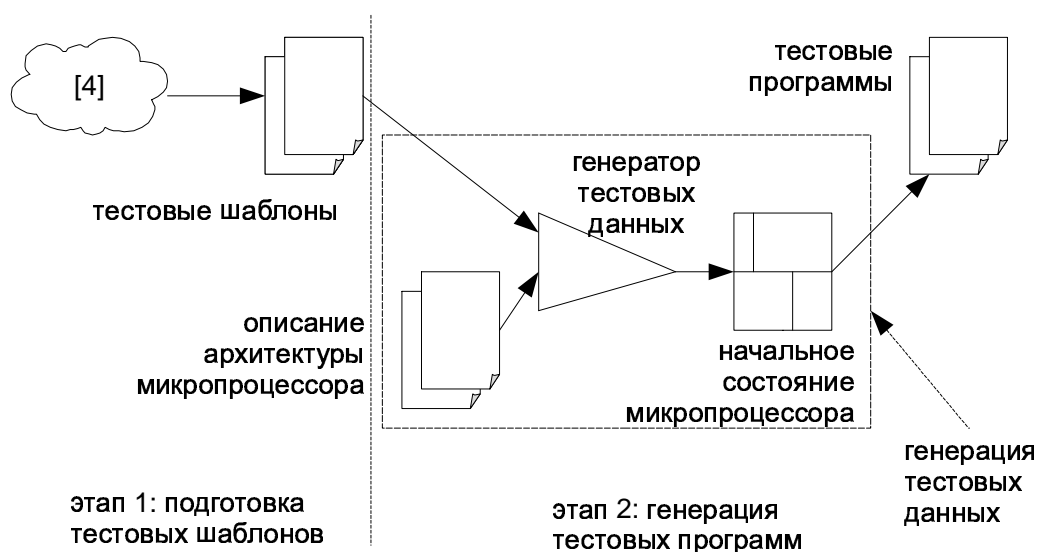


Рис. 1: Генерация тестовых программ на основе тестовых шаблонов

Второй этап включает в себя *генерацию тестовых данных*, т.е. генерацию параметров инструкций (параметров-констант) и начальных значений регистров, ячеек кэш-памяти, строк TLB и т.д. Иногда выбор регистров для инструкций задается в тестовом шаблоне, а иногда выбор регистров может сделать генератор тестовых данных.

Тестовые шаблоны описывают последовательность инструкций, параметры инструкций с указанием происходящих событий (например, переполнение, промахи или попадания в кэш-памяти). В качестве параметров инструкции могут быть как явно указаны регистры и константы, так и предоставлена возможность инструменту самому подобрать регистры. Последовательность инструкций чаще всего задана явно. Необходимость в тестовом шаблоне обычно возникает тогда, когда тестирование проводится нацеленным образом и эта цель выражена последовательностью инструкций, каждая из которых должна быть исполнена заданным образом. Пример тестового шаблона:

```
REGISTER ax:32; REGISTER bx:32; REGISTER cx:32;
ADD ax, ax, bx @ overflow
LW cx, ax, 0 @ 11Hit
MUL ax, cx, bx @ normal
```

В этом тестовом шаблоне 3 инструкции – ADD, LW и MUL. У каждой инструкции указаны параметры – или 3 регистра, или 2 регистра и константа – и после знака '@' информация о том, как должна быть исполнена инструкция (ограниче-

ние на значения параметров инструкции и состояние микропроцессора): overflow – с переполнением при целочисленном сложении (сумма параметров либо слишком мала, либо слишком велика, чтобы точно записать ее в регистр-результат), l1Hit – с кэш-попаданием (при загрузке из памяти данные по требуемому физическому адресу в момент исполнения инструкции должны присутствовать в кэш-памяти) и noMal – без исключений (инструкция выполняется полностью, регистр-результат содержит точное значение). Чтобы получить тестовую программу по этому шаблону, достаточно задать начальные значения регистров ax, bx и cx и той части кэш-памяти, с которой работает вторая инструкция (LW) (это и будут тестовые данные для данного шаблона). Это можно сделать, добавив в начало тестового шаблона инструкции инициализации состояния микропроцессора. Полученную тестовую программу можно исполнить и проверить, совпадает ли поведение каждой инструкции с тем, что было заявлено в тестовом шаблоне.

Обратимся к задаче генерации тестовых данных. Среди известных работ можно выделить следующие методы ее решения:

- комбинаторные техники;
- решение задачи ATPG;
- разрешение ограничений.

Комбинаторные техники применимы в случае простых тестовых шаблонов. Такие тестовые шаблоны включают лишь простые ограничения, а именно указание области значений переменной. Причем все значения этой области в тестовой программе равноправны. Техника хоть и простая, но довольно ограниченная в применении, поскольку не всегда получается привести ограничения на переменные к такому простому виду. В работе исследователей из Fujitsu Lab. [10] предлагается описать тестовые программы в виде выражений (Test Specification Expressions, TSE), а инструкции микропроцессора – на языке ISDL. Специальный генератор строит тестовые программы, удовлетворяющие TSE. Kohno и Matsumoto [11] рассматривают задачу верификации конвейерных микропроцессоров, используя для этого генерацию тестовых программ с помощью тестовых шаблонов. Области значений переменных в таких шаблонах складываются из регистров и числовых констант.

Исследователи из Politecnico di Milano [13] предложили генерировать 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 про-

грамм необходимо, чтобы RTL-модель микропроцессора была готова к моменту генерации тестовых данных. Кроме того, использование такой методики именно для функционального тестирования ограничено, поскольку нарушено требование неизбыточности задания семантики инструкций – наличие RTL-модели является лишним, дополнительным требованием к семантике и требует дополнительных трудозатрат на свою подготовку.

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 [16] 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 [14] – 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 невозможно.

В данной работе при решении задачи генерации тестовых данных также используется разрешение ограничений. Выполняется эта работа в рамках технологической цепочки построения тестовых программ на основе модели микропроцессора [4]. Однако для тестовых шаблонов, получаемых в рамках этой технологической цепочки, инструмент MAATG неприменим, поскольку шаблоны могут содержать не только ограничения равенства или неравенства регистров, но и более сложные ограничения, например, кэш-промах. В данной статье по сравнению

с Genesys-Pro тестовый шаблон транслируется в ограничения целиком (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 практика показывает, что ошибки в управляющей логике обнаруживаются на тестовых программах небольшой длины). При предлагаемой организации построения ограничений отпадает необходимость в механизме возврата. 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.

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 начальное 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 [7]. Однако по имеющимся работам невозможно сделать вывод о том, как такая схема трансляции адресов отображается в ограничения. Авторы статьи используют при описании способа трансляции адреса элементы массива Мемогу 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.

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, которые не удастся разрешить за приемлемое время. Даже без учета трансляции адресов для кодирования состояния микропроцессора можно использовать формулу длиной порядка размера памяти ($mem_0 = var_0 \wedge mem_1 = var_1 \wedge \dots$); каждое изменение производится по неизвестному ин-

дексу, 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 $|L| + n$.

3 Генерация тестовых данных для операций с памятью

Под *генерацией тестовых данных* понимается генерация начального состояния микропроцессора (регистров, ячеек кэш-памяти, строк TLB и т.п.) для заданного тестового шаблона. При исполнении тестового шаблона из сгенерированного начального состояния микропроцессора все инструкции шаблона должны быть исполнены в соответствии с требованиями шаблона. В качестве требований к исполнению инструкции используется указание *тестовой ситуации* у инструкции (ограничения на значения параметров и состояние микропроцессора).

При исполнении инструкций загрузки и сохранения значений в памяти могут быть задействованы такие подсистемы микропроцессора, как кэш-память и TLB (Translation Lookaside Buffer) [15]. Параметры инструкции формируют *виртуальный адрес*, с которым работает инструкция. Затем по виртуальному адресу (возможно, с помощью TLB) микропроцессор вычисляет *физический адрес*. И, наконец, с использованием физического адреса (возможно, с помощью кэш-памяти) осуществляется необходимая операция с оперативной памятью. Если данные по нужному физическому адресу присутствуют в кэш-памяти (такая ситуация называется *кэш-попаданием*), они используются инструкцией и обращение в оперативную память не производится. Если данные по нужному физическому адресу отсутствуют в кэш-памяти (такая ситуация называется *кэш-промахом*), то происходит обращение в оперативную память за требуемыми данными, которые после этого помещаются в кэш-память. Поскольку размер кэш-памяти ограничен, некоторые данные при этом должны быть *вытеснены* из кэш-памяти.

TLB состоит из набора строк, задающих соответствие значений бит виртуального адреса, отвечающих номеру виртуальной страницы, значениям бит физического адреса, отвечающих номеру физической страницы. Кроме этого, TLB содержит небольшую собственную кэш-память – *буфер TLB* – для ускорения поиска нужной строки TLB. Поэтому, в отношении TLB возможны такие ситуации, как *кэш-попадание в буфере TLB* и *кэш-промах в буфере TLB*.

Тестовыми ситуациями для операций с памятью могут быть кэш-промах, кэш-попадание, ограничение на адрес, с которым работает инструкция, ограничение на строку TLB. Например, в следующем тестовом шаблоне:

```
REGISTER ax:32; REGISTER bx:32; CONST c:16;  
LW ax, bx, c @ l1Hit, tlbMiss  
ADD ax, ax, bx @ overflow  
SW ax, bx, c @ l2Miss, tlbHit
```

в качестве тестовой ситуации первой инструкции указано `l1Hit, tlbMiss`. Идентификатор `l1Hit` означает попадание в кэш-памяти первого уровня (L1). Идентификатор `tlbMiss` означает промах в буфере TLB. Тестовая ситуация третьей команды составлена из промаха в кэш-памяти второго уровня (L2) – `l2Miss` – и попадания в буфере TLB (`tlbHit`).

В данной работе предлагается метод генерации тестовых данных для тестовых шаблонов с инструкциями, работающими с памятью. Тестовые ситуации для таких инструкций можно разделить на *тестовые ситуации в кэш-памяти* и *тестовые ситуации при трансляции адресов*. Поскольку тестовые ситуации в кэш-памяти разных инструкций связаны друг с другом ¹, то сначала в предлагаемом методе надо сгенерировать ограничения для тестовых ситуаций в кэш-памяти (см. часть 4 данной статьи). Так как тестовые ситуации при трансляции адресов связаны друг с другом (там тоже используется кэширование), то следующим шагом метода будет генерация ограничений для тестовых ситуаций при трансляции адресов (см. часть 5 данной статьи). В результате получится система ограничений на начальные значения регистров, физические адреса и строки TLB, после разрешения которой остается дополнить тестовый шаблон инструкциями инициализации начального состояния микропроцессора на основе вычисленного начального состояния его подсистем.

4 Упрощение тестовых ситуаций в кэш-памяти

Каждая инструкция работы с памятью оперирует с некоторым физическим адресом. Физический адрес можно представить в виде трех битовых полей – *тег* (старшие биты адреса), *сет* и *индекс в строке кэш-памяти* [15] (см. рис. 2).

Упрощение тестовых ситуаций в кэш-памяти начинается с *распределения тестовых ситуаций шаблона по сетам*, т.е. каждой такой тестовой ситуации следует сопоставить число – номер сета его физического адреса. Сопоставление

¹Например, кэш-промах в одной инструкции означает особый выбор физических адресов для предыдущих инструкций с кэш-попаданием, поскольку при кэш-промахе вытесняемый адрес есть один из тех адресов, к которым до этого были кэш-попадания.

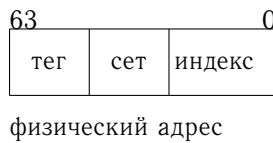


Рис. 2: Структура физического 64-битного адреса

надо проводить таким образом, чтобы оно впоследствии давало совместную систему ограничений. Например, можно сопоставить все тестовые ситуации одному сету или разным.

После этого отдельно для каждого сета проводится следующий *алгоритм получения ограничений «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 (для кэш-попадания это тег физического адреса, для кэш-промаха это 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 символично (упрощаются) до искомого вида. Разрешение ограничений можно проводить любым из известных алгоритмов разрешения ограничений [3]. Именно использование символических упрощений и множественная интерпретация кэш-памяти ² позволило отказаться от кодирования изменений состояния кэш-памяти, что упрощает задачу на разрешение ограничений.

Ниже приведен псевдокод алгоритма построения ограничений, описывающих последовательность тестовых ситуаций в кэш-памяти тестового шаблона. В этом алгоритме в качестве политики замещения используется LRU (Least Recently Used), хотя подобная техника построения ограничений применима и к другим политикам замещения. Текущее состояние сета моделируется множеством L . Кэш-попадание описывается принадлежностью тега 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 : ter-list )
returns C : constraint-set

```

²в противовес списочной (векторной) интерпретации сета кэш-памяти; по сути порядок элементов, который задается политикой замещения в сете кэш-памяти, кодируется не в состоянии сета, а в последовательности тестовых ситуаций в кэш-памяти

```

begin
  C := {};
  var L : тег-set := {}
  для каждого (тега t из xs)
  begin
    добавить в C ограничение  $t \notin L$ ;
    L := L  $\cup$  {t};
  end;
  для каждой (тестовой_ситуации  $\tau$  из tt)
  begin
    если  $\tau$  есть кэш-попадание тега p, то
      добавить в C ограничение  $p \in L$ ;
    иначе если  $\tau$  есть кэш-промах тега p с вытеснением тега q, то
      begin
        добавить в C ограничение  $q \in L$ ;
        добавить в C ограничение  $p \notin L$ ;
        добавить в C ограничение lru( q, L,  $\tau$ , tt );
        L := L  $\cup$  {p} \ {q};
      end;
    end;
    упростить C;
  end,
  procedure lru( q : тег, L : тег-set,  $\tau$  : тестовая_ситуация,
tt : test_template_for_set ) returns C : constraint
begin
  C :=  $\perp$ ;
  для каждого ( $\tau'(p1)$  : кэш-попадания из tt с начала 3 до  $\tau$ )
  begin
    var T : тег-set := множество вытесняющих тегов и тегов попадания в
tt между  $\tau'$  и  $\tau$  неключительно;
    C := C  $\vee$  ( $q = p1$ )  $\wedge$  ( $L \setminus \{q\} = T$ );
  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

³ «начало» есть добавление тегов-переменных начального состояния в сет, добавленное перед тестовым шаблоном

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 [12])
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.)

5 Построение ограничений для трансляции адресов

Оставшиеся шаги алгоритма генерации тестовых данных включают генерацию ограничений для механизма трансляции адресов, разрешение получившихся ограничений и формирование начального состояния подсистем микропроцессора. В статье рассматривается способ трансляции адресов с помощью TLB, принятый в архитектуре микропроцессоров MIPS [2], хотя подобные идеи применимы для трансляции адресов в других RISC-архитектурах. Процесс трансляции адресов и основные структуры показаны на рис. 3.

Для генерации ограничений для трансляции адресов используется тестовый шаблон и информация о сетах и тегах с этапа упрощения тестовых ситуаций в кэш-памяти. Первым этапом генерации ограничений является упрощение тестовых ситуаций в буфере TLB (это делается аналогично тому, как описано в части 4 – сет один и тот же, в качестве тегов выступают индексы строк TLB). Затем для каждой пары инструкций, обращающихся к одной строке TLB, выделяются ограничения на их виртуальные адреса (выраженные в терминах параметров инструкции):

- совпадение бит виртуальных адресов, отвечающих полю «г»;
- совпадение бит виртуальных адресов, отвечающих полю « $vrp/2$ » с учетом поля «mask» (например, если поле « $vrp/2$ » расположено с 40 по 13й биты,

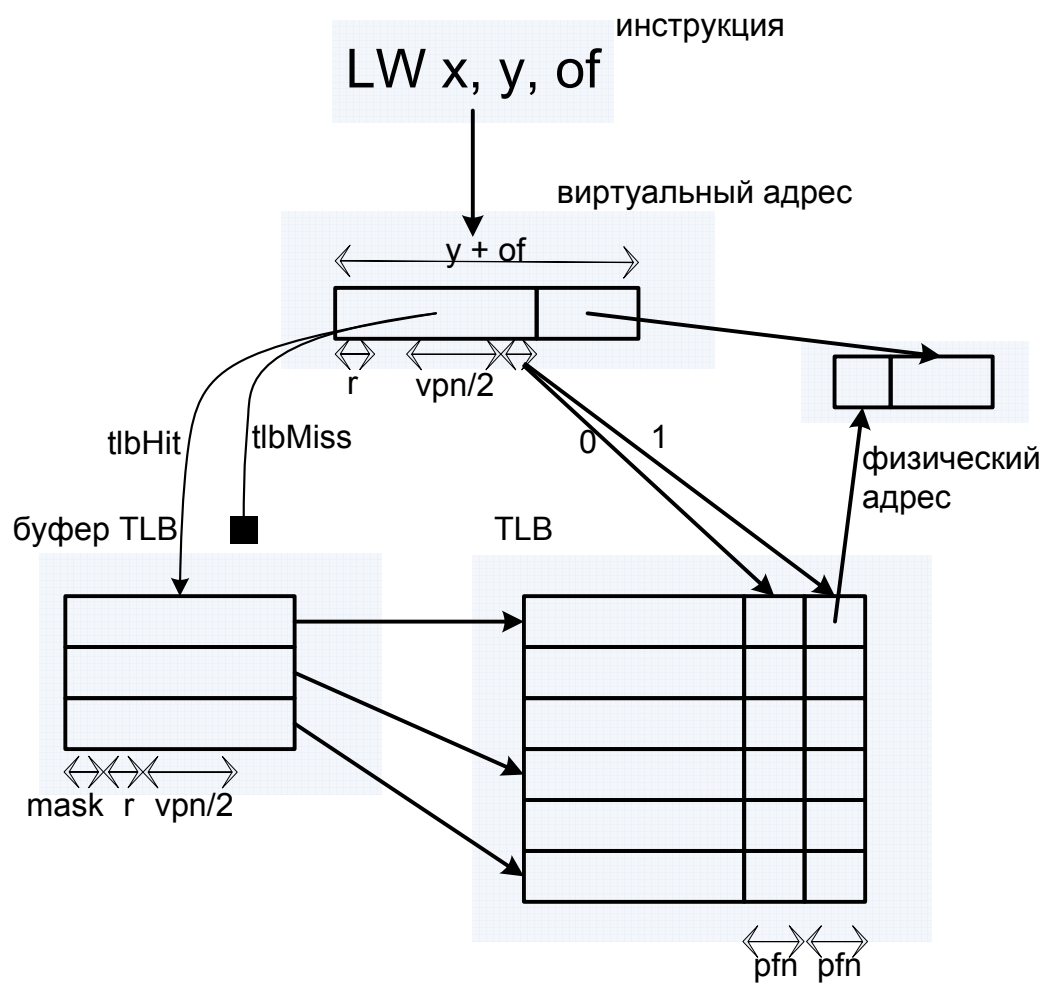


Рис. 3: Трансляция адреса в микропроцессорах архитектуры MIPS

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;
- соответствие бит виртуального и физического адресов, отвечающих смещению в странице; например, если поле «vpn/2» расположено с 40 по 13й биты, то $v_{11+m..0} = phys_{11+m..0}$, где v – виртуальный адрес инструкции, $phys$ – физический адрес инструкции, m – целочисленная форма поля «mask».

Тестовый шаблон работает со страницами (виртуальной памяти) одного размера – того размера, который задает операционная система. Поэтому для всех строк TLB, с которыми работает тестовый шаблон, значение поля «mask» известно и задается перед генерацией тестовых данных. Это упрощает получающиеся ограничения, поскольку устраняет взятие диапазонов бит с переменными границами.

Затем для каждой задействованной строки TLB выбирается произвольная инструкция из тестового шаблона. Для каждой пары таких инструкций (пару образуют инструкции, относящиеся к разным строкам TLB) составляется ограничение, которое выражает факт отличия их строк TLB. Для трансляции адресов, принятой в MIPS, это может быть дизъюнкция из отличия бит виртуальных адресов, соответствующих полю «g», бит, соответствующих полю «vpn/2» (с учетом маски), из равенства 0 бита «g» с отличием поля «asid» от значения в регистре EntryHi (его значение так же, как и поле «mask») задается перед построением ограничений, поскольку это параметр функционирования операционной системы, не меняющийся во время исполнения тестового шаблона.

6 Заключение

В статье рассматривалась задача генерации тестовых данных для системного функционального тестирования микропроцессоров, а именно задача построения тестовой программы по заданному для нее тестовому шаблону. Для решения этой задачи в работе предложен алгоритм, использующий разрешение ограничений [3]. Предложенный алгоритм реализуется на базе системы логического программирования с ограничениями ECLiPSe [12] в качестве решателя ограничений на целые числа и конечные множества целых чисел. Алгоритм применяется в проекте по тестированию промышленного микропроцессора архитектуры

MIPS64 [?]. 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] Корныхин Е.В. Генерация тестовых данных для тестирования арифметических операций центральных процессоров. *Труды ИСП РАН*, 15:107–117, 2008.
- [6] A.Adir, E.Almog, L.Fournier, E.Marcus, M.Rimon, M.Vinov, and A.Ziv. 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] A. Adir, R. Emek, Y. Katz, and A. Koyfman. 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.
- [8] F. Corno, E. Sanchez, M.S. Reorda, and G. Squillero. 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] Kornikhin E. Test data generation for arithmetic subsystem of cpus mips64. *Proceedings of the Second Spring Young Researchers' Colloquium on Software Engineering*, 2:43–46, 2008.
- [10] F.Fallah and K.Takayama. A new functional test program generation methodology. *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 76–81, 2001.
- [11] K.Kohno and N.Matsumoto. A new verification methodology for complex pipeline behavior. *Proceedings of the 38st Design Automation Conference (DAC'01)*, 2001.

- [12] K.R.Apt and M.Wallace. *Constraint Logic Programming Using Eclipse*. Cambridge University Press, 2007.
- [13] M.Beardo, F.Bruschi, F.Ferrandi, and D.Sciuto. 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.
- [14] M.Behm, J.Ludden, Y.Lichtenstein, M.Rimon, and M.Vinov. Industrial experience with test generation languages for processor verification. *Proceedings of the 41st Design Automation Conference (DAC'04)*, 2004.
- [15] D. Patterson and J. Hennesy. *Computer Organisation and Design: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design, 3rd edition, 2005.
- [16] T.Li, D.Zhu, Y.Guo, G.Liu, and S.Li. Maatg: A functional test program generator for microprocessor verification. *Proceedings of the 2005 8th Euromicro conference on Digital System Design (DSD'05)*, 2005.