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 для проверки выполнимости семантики достаточно задать поведение инструкций (иными словами, эта семантика не должна содержать особенности конкретной реализации в конкретном микропроцессоре, она должна быть неизбыточной).

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.

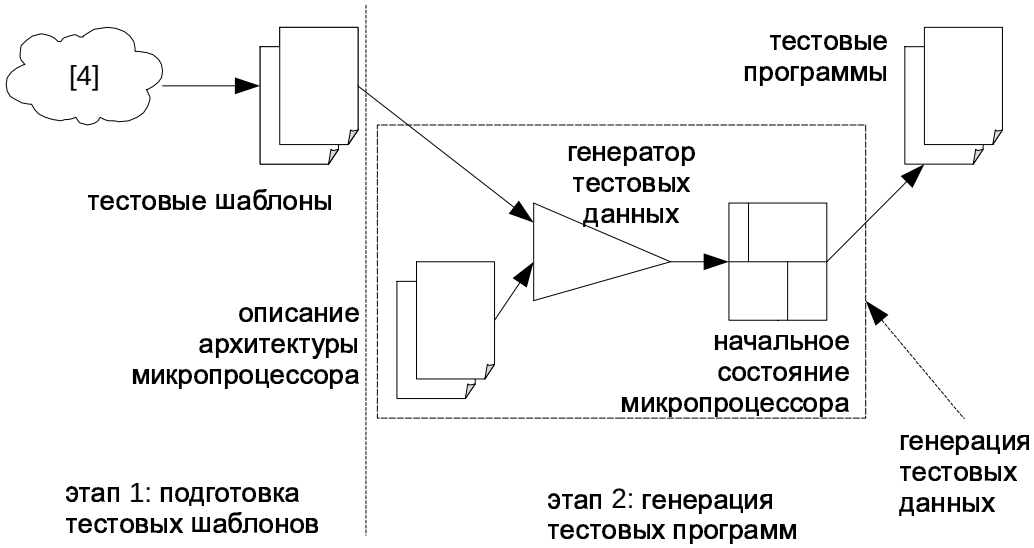


Рис. 1: Генерация тестовых программ на основе тестовых шаблонов

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 @ l1Hit
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 информация о том, как должна быть исполнена инструкция: с переполнени-

ем (overflow), с кэш-попаданием (l1Hit) или без исключений (normal). 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 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 [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. 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 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 действительно длинные тестовые шаблоны в практике тестирования микропроцессоров применяются редко). При этом 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 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.

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 удается разрешить за приемлемое время. 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 $|L| + n$.

3 Генерация тестовых данных для операций с памятью

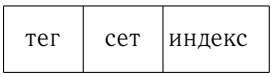
Под *генерацией тестовых данных* понимается генерация начального состояния микропроцессора (регистров, ячеек кэш-памяти, строк TLB и т.п.) для заданного тестового шаблона. При исполнении тестового шаблона из сгенерированного начального состояния микропроцессора все инструкции шаблона должны быть исполнены в соответствии с требованиями шаблона. В качестве требований к исполнению инструкции используется указание *тестовой ситуации* у инструкции. Для операций с памятью тестовыми ситуациями могут быть кэш-промах, кэш-попадание, ограничение на адрес, с которым работает команда, ограничение на строку TLB, с которой работает трансляция адресов в данной инструкции. 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. Он используется при трансляции виртуального адреса в физический. Аналогично тестовая ситуация третьей команды составлена из промаха в кэш-памяти второго уровня (L2) и попадания в буфере TLB.

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* (вычисление физического адреса по виртуальному). 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 командами инициализации начального состояния микропроцессора на основе вычисленного начального состояния его подсистем.

[1] Например, 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.



физический адрес

Рис. 2: Структура физического адреса

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. Сопоставление можно проводить любым способом. Главное – чтобы этот выбор впоследствие давал решение. 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 (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 разрешения ограничений [3]. Именно использование символьных упрощений и множественная интерпретация кэш-памяти [2] позволило отказаться от кодирования изменений состояния кэш-памяти, что упростило задачу на разрешение ограничений.

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 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

[2] в 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

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 : тег-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 : тег, L : тег-set, τ : тестовая_ситуация,
tt : test_template_for_set ) returns C : constraint
  begin
    C := ⊥;
    для каждого (τ'(p1) : кэш-попадания из tt с начала [3] до τ)
    begin
      var T : тег-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

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

(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 [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 (Translation Lookahead Buffer), принятый в стандарте микропроцессоров MIPS [2]. 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 кэше 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 битов виртуальных адресов, отвечающих полю «г»

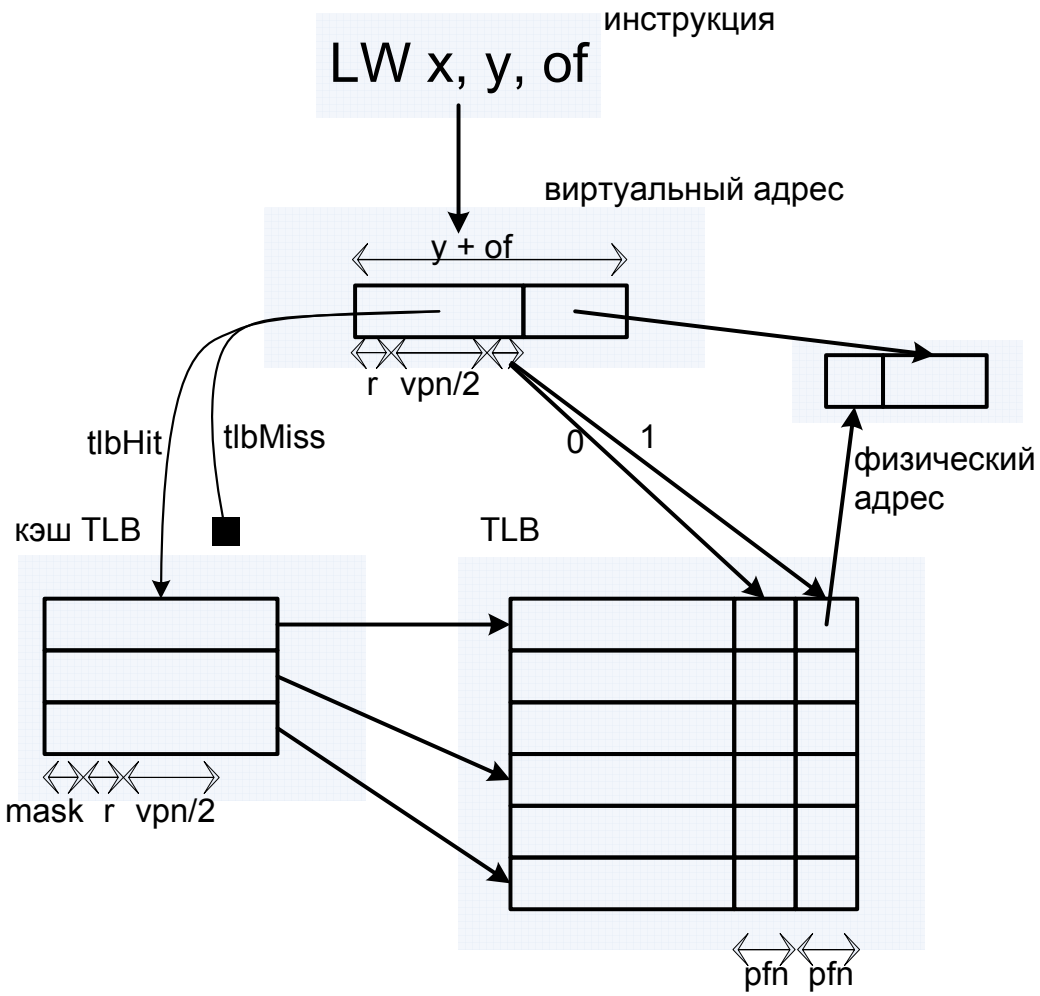


Рис. 3: Трансляция адреса в микропроцессорах стандарта MIPS

- 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.

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 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' Colloqium 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.