УДК 004.05

Комбинаторная генерация тестовых программ для микропроцессоров на основе формальных спецификаций системы команд

А.Д. Татарников

Институт системного программирования РАН, andrewt@ispras.ru

***Аннотация* — Генерация тестовых программ и анализ результатов их симуляции на проектной модели являются основным подходом к функциональной верификации микропроцессоров. Верификация – крайне трудоемкий процесс. По некоторым оценкам затраты на нее составляют около 70% от общих трудозатрат на разработку микропроцессора. Это связано с тем, что логика работы современных микропроцессоров содержит огромное количество состояний, и для того, чтобы обеспечить их полное покрытие, требуются значительные усилия. В данной работе рассматривается подход к генерации тестовых программ, который позволяет повысить эффективность тестирования за счет использования комбинаторных методов для генерации тестовых программ. Основная идея метода состоит в построении тестовых воздействий путем комбинаторного перебора инструкций микропроцессора и ситуаций в их работе, условия возникновения которых заданы в виде ограничений. Знание о системе команд микропроцессора автоматически извлекается из формальных спецификаций.**

***Ключевые слова* — микропроцессоры, функциональная верификация, тестирование, генерация тестовых программ, формальные спецификации, комбинаторные методы.**

I. Введение

Задача *функциональной верификации* заключается в проверке того, что поведение микропроцессора во всех возможных ситуациях соответствует спецификации [1, 2]. Эта задача решается путем анализа результатов выполнения команд микропроцессора, поскольку его функциональность определяется реализуемой им системой команд. Для этого на проектной *модели уровня регистровых передач* (*RTL, Register Transfer Level*) симулируется огромное количество тестовых программ, которые должны обеспечить максимальное покрытие возможных ситуаций в работе микропроцессора. Создание таких программ является нетривиальной задачей. Подходы к созданию тестовых программ можно разделить на следующие категории:

1) ручная разработка;
2) случайная генерация;
3) генерация на основе ограничений.

Ручная разработка обладает ограниченной применимостью ввиду трудоемкости. В настоящее время она применяется в основном для проверки так называемых крайних случаев. Случайная генерация позволяет создать большое количество тестов, покрывающих разнообразную функциональность. Направление тестирования можно задавать при помощи настроек (область значений, распределение вероятностей и. т.д.). К сожалению, такой подход не гарантирует полноты покрытия, т.к. некоторые нетривиальные ситуации недостижимы при случайной генерации. Генерация на основе ограничений применяется для построения тестов, нацеленных на конкретные ситуации или классы ситуаций. Условия возникновения ситуаций задаются в виде ограничений на входные данные и состояние микропроцессора, а тестовые программы строятся на основе *шаблонов тестовых программ*, которые указывают, какие ограничения должны быть применены к инструкциям в тестовой программе для воспроизведения нужных ситуаций. Данный подход позволяет проверить поведение микропроцессора во многих сложных ситуациях, но он также не гарантирует полноты покрытия, т.к. при ручной разработке тестовых шаблонов некоторые «интересные» ситуации могут быть пропущены. Поэтому перспективным видится подход, который дополняет генерацию на основе ограничений, обеспечивая автоматическое построение тестовых программ посредством комбинаторного перебора инструкций тестируемого микропроцессора и ситуаций в их работе.

В данной работе рассматривается подход к генерации тестовых программ, который использует формальные спецификации системы команд микропроцессора в качестве источника знания о тестовых ситуациях и обеспечивает их покрытие при помощи комбинаторных методов. Также на основе формальных спецификаций строится симулятор уровня системы команд, позволяющий отслеживать состояние микропроцессора в процессе генерации. Направление генерации задается при помощи тестовых шаблонов, абстрактных описаний, которые задают, какие команды и какие ситуации будут использоваться при построении тестов. Данный подход позволяет улучшить качество тестирования и снизить его трудоемкость.

Статья организована следующим образом. В разделе II делается обзор существующих методов и средств генерации тестовых программ для

микропроцессоров. В разделе III описываются основные концепции предложенного подхода. В разделе IV рассказывается, как знание о тестируемом микропроцессоре может быть задано при помощи формальных спецификаций. В разделе V описывается архитектура среды генерации. В разделе VI описывается язык тестовых шаблонов и варианты его применения для построения комбинаторных тестов. В разделе VII приводятся метрики, полученные в процессе испытания метода. В разделе VIII делается заключение и рассказывается о направлениях дальнейших исследований.

II. Обзор существующих работ

Методы генерации тестовых программ являются приоритетным направлением исследований в течение нескольких последних десятилетий. Одним из лидеров данного направления является компания IBM. Ее подразделение IBM Research занимается разработкой генераторов тестовых программ с начала 1980-х годов. Применяемые ими методы эволюционировали от полностью случайной генерации до генерации на основе ограничений, которые задаются моделью микропроцессора, созданной на специальном декларативном языке. Таким образом, ядро инструмента отделено от конфигурации конкретного микропроцессора, что позволяет решать задачи генерации в общем виде. Данный подход был реализован в генераторе тестовых программ Genesys-Pro [3], который использует шаблоны для описания целей тестирования в терминах ограничений. При этом используются ограничения двух видов: «жесткие», взятые из модели, и «мягкие», основанные на экспертном знании и предназначенные для обеспечения лучшего покрытия интересных с точки зрения тестирования аспектов функциональности. Ограничения второго вида могут быть проигнорированы, если решателю ограничений не удастся найти решение. К достоинствам инструмента можно отнести выразительный язык описания тестовых шаблонов и удобную среду моделирования. Главный недостаток состоит в том, что при ручной разработке тестовых шаблонов не гарантируется полнота покрытия, т.к. некоторые важные ситуации в работе микропроцессора могут быть пропущены.

Другой известный инструмент генерации тестовых программ – это RAVEN (Random Architecture Verification Machine) [4], разработанный компанией Obsidian Software, которая была поглощена компанией ARM в 2011-м году. Этот инструмент генерирует случайные и нацеленные тесты на основе тестовых шаблонов. Для достижения лучшего покрытия используются матрицы тестового покрытия и специальные эвристики на основе экспертного знания. Тестовые шаблоны формулируют цели тестирования в терминах матрицы тестового покрытия и используют ограничения для выбора случайных значений, удовлетворяющих заданным целям. К сожалению, сложно сделать вывод насколько данный инструмент позволяет обеспечить полноту покрытия, т.к. доступная документация не освещает в достаточной степени его возможности.

Также компанией Samsung была разработана среда RDG (Random Diagnostics Generator) [4], предназначенная для случайной генерации тестовых программ для реконфигурируемых микропроцессоров. Среда использует в качестве входных данных описание синтаксиса инструкций тестируемого микропроцессора и тестовые шаблоны на языке C++. Тестовые шаблоны указывают какие инструкции будут использованы в тестовой программе и описывают ограничения, накладываемые на входные значения этих инструкций. При данном подходе среда генерации не владеет информацией о семантике инструкций и не осуществляет симуляцию тестовых программ с целью предсказания результатов. Это позволяет с минимальными усилиями добавлять поддержку новых инструкций и обеспечить высокую скорость генерации. Однако при этом для генерации нацеленных тестов требуется описывать ограничения вручную и отсутствует возможность использовать информацию о состоянии микропроцессора при разрешении ограничений. Это может отрицательно сказаться на качестве тестового покрытия.

Кроме того, исследования в области генерации тестовых программ проводятся в Университете Флориды и Калифорнийском университете в Ирвайне. Подход, предложенный в работе [5], использует детальные спецификации микропроцессоров на языке EXPRESSION [6]. Спецификации описывают структуру (компоненты и соединения) и поведение (семантику инструкций) микропроцессора, а также взаимосвязь между ними. На основе этой информации строится модель на языке SMV (Symbolic Model Verifier) [7]. Ключевой частью метода является *модель ошибок*, описывающая типичные ошибки проектирования. На основе модели ошибок для модели микропроцессора строятся формулы, которые задают условия возникновения конкретных ошибок для данного микропроцессора. Для этих формул при помощи инструмента SMV, который использует метод проверки моделей, стоятся тестовые примеры (контрпримеры для отрицания формул), на основе которых генерируются тестовые программ. По мнению авторов, метод не масштабируется на сложные микропроцессоры, поэтому предлагается комбинировать его с методом на основе тестовых шаблонов. Шаблоны создаются вручную и содержат описания цепочек инструкций, которые вызывают определенные ситуации в поведении микропроцессора (прежде всего, конвейерные конфликты). К недостаткам данного подхода можно отнести сложность разработки детальных спецификаций и модели ошибок.

В работах Института системного программирования РАН [8, 9] описывается метод построения генераторов тестовых программ на основе формальных спецификаций, реализованный в инструменте MicroTESK. [10] Этот метод позволяет упростить создание генераторов тестовых программ

для определенных архитектур. Также в работе [11] рассказывается о методе комбинаторной генерации на основе моделей микропроцессоров, разработанных на языке Java. Текущая работа продолжает эти исследования, объединяя оба метода.

III. Основные концепции

Тестовая программа представляет собой набор *тестовых воздействий*, предназначенных для проверки корректности поведения микропроцессора при определенных условиях. Каждое тестовое воздействие задается в виде последовательности инструкций, выполнение которых приводит к возникновению определенных событий в работе микропроцессора, называемых *тестовыми ситуациями*. Поведение микропроцессора в процессе выполнения последовательности инструкций зависит от значений, переданных инструкциям в виде аргументов, и от текущего состояния микропроцессора. Поскольку каждая инструкция изменяет состояние микропроцессора, то между инструкциями существуют *зависимости*, которые связаны с совместным использованием регистров, памяти и конвейера инструкций.

Для генерации тестовых воздействий, нацеленных на проверку определенных сценариев работы микропроцессора, используются *тестовые шаблоны*. Они описывают тестовые воздействия в виде абстрактных последовательностей инструкций, в которых значения входных аргументов представлены в виде *ограничений*, которые задают условия возникновения тестовых ситуаций.

Анализ статистики нахождения ошибок в микропроцессорах показывает, что многие ошибки могут быть обнаружены при помощи тестовых воздействий, состоящих из 2-5 инструкций. Такие последовательности можно простроить при помощи комбинаторных методов. Построение осуществляется путем перебора инструкций и их тестовых ситуаций с учетом зависимостей.

Комбинаторные последовательности описываются при помощи *обобщенных тестовых шаблонов*, которые задают длину последовательностей, критерии перебора, состав используемых инструкций и зависимости между ними. Состав инструкций может быть описан в терминах *групп инструкций*, которые объединяют инструкции по некоторым общим признакам (арифметика, ветвления, чтение/запись в память и т.д.). Зависимости задаются путем указания совместно используемых регистров и выборе групп инструкций, предполагающих определенные варианты совместного использования конвейера. Зависимости по памяти не рассматриваются в данной работе.

В общем виде построение тестовых воздействий осуществляется следующим образом:

1) на основе обобщенного тестового шаблона строятся всевозможные комбинации инструкций фиксированной длины заданной структуры, составленные из инструкций заданных групп;

2) для каждой комбинации инструкций строятся всевозможные комбинации тестовых ситуаций, на основе которых строятся тестовые шаблоны, описывающие отдельные тестовые воздействия;

3) для каждого тестового шаблона генерируются тестовые данные путем разрешения ограничений, заданных для отдельных инструкций и соответствующих выбранным тестовым ситуациям;

4) на основе тестовых данных строится *инициализирующий код*, который размещает тестовые данные в нужных регистрах или по нужным адресам памяти.

5) инициализирующий код и код тестового воздействия интерпретируются на *эталонной модели*, построенной на основе формальных спецификаций.

Количество возможных комбинаций может быть весьма значительным. Для его сокращения могут применяться специальные эвристики, такие как объединение схожих инструкций в *классы эквивалентности*. Также стоит отметить, что при построении тестов, нацеленных на определенные сценарии работы микропроцессора, речь не идет о полном переборе, т.к. набор используемых инструкций и тестовых ситуаций будет ограничен некоторым подмножеством.

Другая важная оговорка состоит в том, что при разрешении ограничений некоторые из них могут не иметь решения из-за противоречий с ограничениями, накладываемыми предыдущими инструкциями. Такие тестовые примеры считаются недостижимыми и отбрасываются.

Интерпретация инструкций в процессе построения тестовых воздействий позволяет убедиться в корректности генерируемых тестовых программ (предусловия инструкций не нарушены, отсутствуют зацикливания и т.д.). Кроме того, на основе информации о текущем состоянии микропроцессора, полученной из эталонной модели, можно создавать самопроверяющие тестовые программы, которые включают в себя проверки корректности состояния микропроцессора после выполнения каждого тестового воздействия.

В последующих разделах стадии построения тестовых последовательностей будут рассмотрены более подробно на примерах.

IV. Формальные спецификации

Информация об архитектуре тестируемого микропроцессора задается при помощи формальных спецификаций на языке nML [12]. Это высокоуровневый язык из семейства языков описания архитектуры (Architecture Description Language, ADL), который позволяет описать синтаксис и семантику инструкций, абстрагируясь от деталей их реализации.

Спецификации на языке nML описывают используемые константы, типы данных, регистры,

режимы адресации, инструкции, память и временные переменные. Например, в приведенном ниже фрагменте кода описывается статическая константа WORD_SIZE и целочисленные типы данных BIT, BYTE (беззнаковые) и WORD (знаковый).

```
let WORD_SIZE = 32
type BIT = card(1)
type BYTE = card(8)
type WORD = int(WORD_SIZE)
```

Регистры одного типа описываются в виде массивов переменных, доступ к которым осуществляется при помощи режимов адресации. Это специальные абстракции, которые инкапсулируют логику обращения к элементам массива, а также задают ассемблерный (атрибут *syntax*) и двоичный (атрибут *image*) форматы регистров. Например, регистры общего назначения микропроцессора MIPS [13] и режим адресации для доступа к ним можно описать следующим образом.

```
reg GPR [32, WORD]
mode REG (i: card(5)) = GPR[i]
  syntax = format("$%d", i)
  image  = format("%5b", i)
```

Память описывается так же, как и группы регистров, в виде массива. В приведенном ниже фрагменте кода массив M интерпретируется как физическая память, состоящая из 2^{30} 32-битных слов (или 2^{32} байт). Механизмы виртуальной памяти, такие как трансляция адресов и кэширование, описываются в отдельных спецификациях особого формата, которые не рассматриваются в данной статье.

```
mem M [2 ** 30, WORD]
```

Помимо регистров и памяти, в nML можно объявлять временные переменные для хранения промежуточных результатов вычислений. Например, ниже объявлена переменная temp для хранения результатов арифметических операций.

```
var temp[int(33)]
```

Инструкции описываются в виде иерархической структуры *операций*, каждая из которых описывает отдельную часть логики. Логика, общая для всех инструкций (например, инкрементирование счетчика команд), помещается в корневой узел. Операции, уникально характеризующие отдельные инструкции, помещаются в терминальных узлах. Эти операции принимают в качестве параметров режимы адресации, которые описывают входные и выходные параметры инструкции. Операции содержат три атрибута: *action*, *syntax* и *image*. Первый описывает логику работы инструкций в терминах присваивания значений регистров. Последние два задают ассемблерный и двоичный форматы. Например, уникальная часть операции сложения микропроцессора MIPS может быть описана следующим образом.

```
op add (rd: REG, rs: REG, rt: REG)
  syntax = format("add %s,  %s, %s",
                  rd.syntax, rs.syntax, rt.syntax)
  image = format("000000%s%s%s00000100000",
                  rs.image, rt.image, rd.image)
  action = {
    temp = rs<31>::rs + rt<31>::rt;
    if temp<32> != temp<31> then
      mark("overflow");
      exception("IntegerOverflow");
    else
      mark("normal");
      rd = temp<31..0>;
    endif;
  }
```

У данной инструкции два возможных пути выполнения: (1) сложение с переполнением, приводящее к исключению IntegerOverflow, и (2) сложение без переполнения (так называемое *нормальное* выполнение). Оба случая образуют тестовые ситуации, описанные при помощи ограничений на значения входных регистров.

nML позволяет объединять режимы адресации и операции в группы. Если в объявлении используется группа, то это означает, что в этом месте применим любой из ее элементов. При этом обязательное требование для комбинаторной генерации, чтобы сгруппированные объекты имели одинаковый список параметров. Можно описывать несколько вариантов группировки инструкций, руководствуясь разными критериями. Например, группа из операций битовой арифметики может быть объявлена следующим образом.

```
op bitwise = and | or | nor | xor
```

На основе nML спецификаций строится модель микропроцессора, состоящая из следующих компонентов: (1) *метамодели*, которая содержит список имен инструкций, их параметров и связанных с ними ситуаций; (2) *симулятора уровня системы команд*, который позволяет выполнить заданные инструкции, получить их текстовое представление и отслеживать текущее состояние микропроцессора; (3) *модели тестового покрытия*, которая содержит ограничения, описывающие возможные пути выполнения инструкций и соответствующие тем или иным ситуациям.

V. Среда генерации

Модель используется средой генерации в качестве источника знания о тестируемом микропроцессоре. Задачи генерации, описываемые при помощи тестовых шаблонов, формулируются в терминах модели. Другими словами, среда оперирует абстрактными сущностями, предоставленными моделью, по правилам, заданными тестовыми шаблонами.

Упрощенно схему генерации можно описать следующим образом. Тестовые шаблоны анализируются обработчиком тестовых шаблонов, который строит их внутреннее представление. Дальше внутреннее представление последовательно

обрабатывается генератором последовательностей и генератором данных к внутреннему представлению.

Генератор последовательностей создает последовательности инструкций на основе информации из тестовых шаблонов и метамодели. Он позволяет комбинировать инструкции и их тестовые ситуации, используя расширяемый набор методов, который включает случайные и комбинаторные методы. Также реализованы методы для объединения последовательностей, построенных по различным правилам.

Генератор тестовых данных разрешает ограничения для заданных ситуаций и вставляет в последовательности инициализирующий код. Ограничения разрешаются при помощи универсальных решателей, таких как Z3 [14] и CVC4[15]. На завершающем шаге построенные последовательности выполняются на симуляторе и печатаются в виде тестовых программ. Упрощенная схема среды генерации изображены на рис. 1.

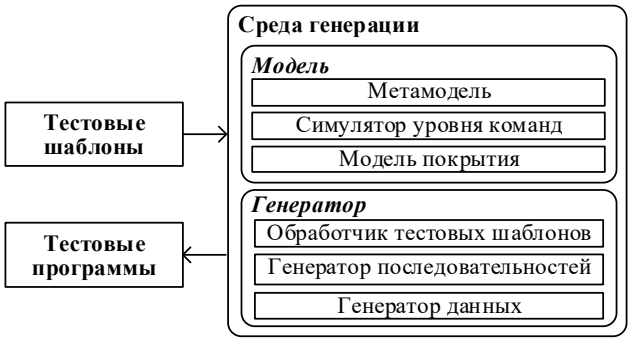

Рис. 1. Архитектура среды генерации

VI. Тестовые шаблоны

Тестовые шаблоны задают структуру тестовых программ и применяемые методы генерации. Для их создания используется язык Ruby [16], расширенный библиотеками, которые реализуют дополнительные конструкции для описания задач генерации.

Последовательности инструкций описываются при помощи блоков, которые задают наборы используемых инструкций и методы построения последовательностей. Ниже приведен пример элементарной последовательности.

```
sequence {
  add t1, t2, t3 do situation('normal') end
  add t1, t4, t5 do situation('overflow') end
}
```

Данный пример описывает последовательность из двух инструкций сложения, первая из которых будет выполнена без переполнения, а вторая вызовет ситуацию целочисленного переполнения. Все последовательности из этих двух инструкций, задающие все возможные комбинации ситуаций в их работе, можно описать при помощи следующего обобщенного шаблона.

```
sequence(:situations => 'product') {
  add t1, t2, t3 do situation('all-paths') end
  add t1, t4, t5 do situation('all-paths') end
}
```

Зависимости по регистрам могут описываться как в явном виде (например, в приведенных выше примерах задана зависимость по выходному регистру t1), так и в абстрактном виде. В последнем случае используемые регистры выбираются в зависимости от заданных критериев. Например, приведенный ниже шаблон описывает последовательности, использующие три типа зависимостей по данным: (1) чтение после записи (read after write, RAW), (2) запись после чтения (write after read, WAR) и (3) запись после записи (write after write, WAW). При этом конкретные регистры будут выбраны случайным образом в соответствии с заданными критериями.

```
sequence(:registers => [:raw, :war, :waw]) {
  add reg(_), reg(_), reg(_)
  add reg(_), reg(_), reg(_)
}
```

Дальнейшее обобщение тестовых шаблонов предполагает перебор инструкций с совпадающим списком параметров, объединенных в группы. Для этого вместо имен инструкций необходимо указать имена их групп, а в параметрах блока задать метод выбора инструкций. Например, следующий обобщенный шаблон перебирает все пары инструкций из группы arithm, которая включает в себя инструкции add, sub, sllv и srlv.

```
sequence(:groups=>'product', :situations=>'product') {
  arithm t1, t2, t3 do situation('all-paths') end
  arithm t1, t4, t5 do situation('all-paths') end
}
```

Инструкции, у которых отличаются списки параметров, можно перебирать при помощи итерирования по спецификациям их вызовов. Например, следующая конструкция задает итерирование по инструкциям ветвления.

```
iterate {
  beq t0, t1, :target
  bgez t0, :target
  b :target
}
```

Также последовательности, построенные при помощи различных методов, можно объединять, используя специальные объединяющие блоки. Например, следующая конструкция позволяет построить последовательности на основе Декартова произведения вложенных последовательностей.

```
block(:combinator => 'product') {
  sequence { … }
  sequence { … }
  sequence { … }
}
```

Описанные методы построения последовательностей можно комбинировать. Это позволяет описывать сложные последовательности. Естественно, полный перебор всех возможных вариантов сочетаний инструкций и ситуаций представляет собой NP-полную задачу. Поэтому он применим только для шаблонов ограниченной длины, использующих ограниченный набор инструкций. При помощи таких шаблонов можно создавать тесты для покрытия определенных аспектов функциональности.

Для сокращения количества вариантов схожие инструкции можно объединять в классы эквивалентности. При этом конкретные инструкции будут выбираться случайным образом из заданных групп. Также в тех случаях, когда полный перебор невозможен из-за комбинаторно взрыва, его можно заменить случайными выборками. Такой подход не дает гарантий относительно качества покрытия, однако позволяет сгенерировать значительное количество тестов, покрывающих нетривиальные ситуации.

VII. Результаты экспериментов

В рамках экспериментальной апробации описанный метод был применен для генерации тестовых программ для архитектуры ARMv8 [17]. Для этого были разработаны спецификации системы команд на языке nML. Трудоемкость разработки составила около 9 человеко-месяцев. Метрики приведены в таблице 1.

Таблица 1

Метрики nML спецификаций для ARMv8

Всего строк кода	9340
Всего инструкций	394
Инструкции ветвления	12
Инструкции работы с памятью	128
Инструкции операций над непосредственными операндами	72
Инструкции операций над регистровыми данными	109
Инструкции умножения и деления	21
Инструкции плавающей арифметики	24
Системные инструкции	28
Средняя длина инструкций (строки кода)	23
Общее число ветвлений в инструкциях	387
Максимальное число ветвлений на инструкцию	12
Максимальное число путей выполнения на инструкцию	144

Как можно увидеть, для последовательностей инструкций ограниченной длины (2-3 инструкции) число возможных комбинаций ситуаций будет конечным. Число комбинаций инструкций может быть значительным, но его можно сократить, объединив схожие инструкции в классы эквивалентности.

Также была проведена оценка производительности. При построении последовательностей из трех инструкций, входные данные для которых генерируются путем разрешения ограничений, производительность генератора составляет около 75 инструкций в секунду. Другими словами, 1000 тестовых воздействий будет сгенерирована за 2 минуты 15 секунд. Производительность может быть улучшена за счет оптимизации взаимодействия с внешним решателем ограничений, которое сейчас осуществляется через файл.

VIII. Заключение

В данной статье был предложен метод генерации тестовых программ для микропроцессоров, который использует формальные спецификации системы команд для получения информации о ситуациях в работе инструкций. Эти ситуации описываются в виде ограничений. На основе полученной информации при помощи комбинаторных методов строятся тестовые воздействия для покрытия всевозможных сочетаний инструкций и ситуаций. Тестовые данные генерируются путем разрешения соответствующих ограничений. Описанный подход был реализован в среде MicroTESK (ИСП РАН) [10].

В настоящий момент ведутся работы по доработке инструмента. Важной задачей является повышение производительности. Прорабатываются эвристики для устранения избыточности при построении комбинаций. Идет работа по созданию специализированного решателя ограничений.

Другим направлением исследований в рамках данного проекта является генерация тестовых программ для подсистемы управления памятью [18], в основе которой лежат те же идеи.

Также в дальнейших планах стоит поддержка спецификаций конвейера команд. На основе этих спецификаций будут разработаны методы генерации, позволяющие строить тестовые воздействия для покрытия различных вариантов совместного выполнения инструкций.

Поддержка

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 15-07-03834.

Литература

[1] Камкин А.С., Коцыняк А.М., Смолов С.А., Сортов А.А., Татарников А.Д., Чупилко М.М. Средства функциональной верификации микропроцессоров // Труды ИСП РАН, Т. 26, В. 1, 2014, С. 149-200

[2] Bergeron J. Writing Testbenches: Functional Verification of HDL Models. Springer, 2003. 478 p.

[3] 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 & Test of Computers, Volume 21, Issue 2, 2004, P. 84–93.

[3] URL: http://www.slideshare.net/DVClub/introducing-obsidian-software-and-ravengcs-for-powerpc (дата обращения: 18.03.2016).

[4] Seonghun Jeong, Youngchul Cho, Daeyong Shin, Changyeon Jo, Yenjo Han, Soojung Ryu, Jeongwook Kim, and Bernhard Egger. Random Test Program Generation for Reconfigurable Architectures // 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, 6 p.
[5] Mishra P. and Dutt N. Specification-Driven Directed Test Generation for Validation of Pipelined Processors // ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 13, Issue 3, 2008, P. 1–36.
[6] Grun P., Halambi A., Khare A., Ganesh V., Dutt N. and Nicolau A. EXPRESSION: An ADL for System Level Design Exploration. Technical Report 1998-29, University of California, Irvine, 1998.
[7] URL: http://www.cs.cmu.edu/~modelcheck/smv.html (дата обращения: 18.03.2016).
[8] Kamkin A., Tatarnikov A.. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors // Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2012. P. 64-69.
[9] Камкин А.С., Сергеева Т.И., Смолов С.А., Татарников А.Д., Чупилко М.М. Расширяемая среда генерации тестовых программ для микропроцессоров // Программирование, № 1, 2014. С. 3-14.
[10] URL: http://forge.ispras.ru/projects/microtesk (дата обращения: 18.03.2016).
[11] Камкин А.С. Комбинаторная генерация тестовых программ для микропроцессоров на основе моделей // Препринт Института Системного Программирования РАН, Т. 21, 2008, 15 С.
[12] Freericks M. The nML Machine Description Formalism. Techical Report, TU Berlin, FB20, Bericht 1991/15.
[13] MIPS64T M Architecture For Programmers. Volume II: The MIPS64T M Instruction Set. Document Number: MD00087, Revision 2.00, June 9, 2003.
[14] Moura L. and Bjørner N. Z3: An Efficient SMT Solver // Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, P. 337–340.
[15] URL: http://cvc4.cs.nyu.edu (дата обращения: 18.03.2016).
[16] URL: http://www.ruby-lang.org (дата обращения: 18.03.2016).
[17] ARM Architecture Reference Manual. ARM DDI 0487A.f, ARM Corporation, 2015. 5886 p.
[18] Kamkin A., Protsenko A., Tatarnikov A. An Approach to Test Program Generation Based on Formal Specifications of Caching and Address Translation Mechanisms // Proceedings of the Institute for System Programming Volume 27 (Issue 3). 2015. P. 125-138.

Combinatorial Test Program Generation for Microprocessors Based on Formal Specifications of Instruction Set Architecture

A.D. Tatarnikov

Institute for System Programming of the Russian Academy of Sciences, andrewt@ispras.ru

***Keywords* — microprocessors, functional verification, testing, test program generation, combinatorial methods.**

ABSTRACT

Test program generation and simulation is the most widely used approach to functional verification of microprocessors. Functional verification is a quite time consuming process. According to various estimates, it accounts for more than 70% of overall resources spent on designing a new microprocessor. This can be explained by the fact that modern hardware designs have an enormous state space and covering all the states demands significant efforts. Most of modern test program generation tools create test stimuli either using random methods or by resolving constraints that specify conditions to be hold to reach certain states. Both approaches do not guarantee that all states will be covered since they are targeted at random or predefined situations. The present work proposes an approach to test program generation that helps improve test coverage by strengthening constraint-based generation with combinatorial methods. The key idea is to construct fixed-length instruction combinations and to apply various combinations of constraints to them. Information used as a basis for creating tests is automatically extracted from formal specifications of the instruction set architecture.

REFERENCES

[1] Kamkin A., Kotsynyak A., Smolov S., Sortov A., Tatarnikov A., Chupilko M. Tools for Functional Verification of Microprocessors // Proceedings of the Institute for System Programming Volume 26 (Issue 1). 2014, P. 149-200 (in Russian).
[2] Bergeron J. Writing Testbenches: Functional Verification of HDL Models. Springer, 2003. 478 p.
[3] 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 & Test of Computers, Volume 21, Issue 2, 2004, P. 84–93.
[3] URL: http://www.slideshare.net/DVClub/introducing-obsidian-software-and-ravengcs-for-powerpc (accessed 18.03.2016).
[4] Seonghun Jeong, Youngchul Cho, Daeyong Shin, Changyeon Jo, Yenjo Han, Soojung Ryu, Jeongwook Kim, and Bernhard Egger. Random Test Program Generation for Reconfigurable Architectures // 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, 6 p.
[5] Mishra P. and Dutt N. Specification-Driven Directed Test Generation for Validation of Pipelined Processors // ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 13, Issue 3, 2008, P. 1–36.
[6] Grun P., Halambi A., Khare A., Ganesh V., Dutt N. and Nicolau A. EXPRESSION: An ADL for System Level

Design Exploration. Technical Report 1998-29, University of California, Irvine, 1998.
[7] URL: http://www.cs.cmu.edu/~modelcheck/smv.html (accessed 18.03.2016).
[8] Kamkin A., Tatarnikov A.. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors // Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2012. P. 64-69.
[9] Kamkin A.S., Sergeeva T.I., Smolov S.A., Tatarnikov A.D., Chupilko M.M. Extensible Environment for Test Program Generation for Microprocessors // Programming and Computer Software, 40(1), 2014. P. 1-9.
[10] URL: http://forge.ispras.ru/projects/microtesk (accessed 18.03.2016).
[11] Kamkin A.S. Combinatorial Model-Based Test Program Generation for Microprocessors // Preprint of Institute for System Programming of RAS, 2008, 13 P.
[12] Freericks M. The nML Machine Description Formalism. Techical Report, TU Berlin, FB20, Bericht 1991/15.
[13] MIPS64T M Architecture For Programmers. Volume II: The MIPS64T M Instruction Set. Document Number: MD00087, Revision 2.00, June 9, 2003.
[14] Moura L. and Bjørner N. Z3: An Efficient SMT Solver // Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 337–340.
[15] URL: http://cvc4.cs.nyu.edu (accessed 18.03.2016).
[16] URL: http://www.ruby-lang.org (accessed 18.03.2016).
[17] ARM Architecture Reference Manual. ARM DDI 0487A.f, ARM Corporation, 2015. 5886 p.
[18] Kamkin A., Protsenko A., Tatarnikov A. An Approach to Test Program Generation Based on Formal Specifications of Caching and Address Translation Mechanisms // Proceedings of the Institute for System Programming Volume 27 (Issue 3). 2015. P. 125-138.