Комбинаторная генерация тестовых программ для микропроцессоров на основе моделей

А.С. Камкин
Институт системного программирования РАН
109004, Россия, Москва, Б. Коммунистическая, 25
E-mail: kamkin@ispras.ru

Аннотация. В статье описывается метод автоматизированной генерации тестовых программ, предназначенный для систематичной функциональной верификации микропроцессоров. Метод дополняет такие широко распространенные на практике подходы, как верификация на основе существующего программного обеспечения и случайная генерация. В предлагаемом методе конструирование тестовых программ базируется на модели микропроцессора, которая включает в себя структурную модель и модель системы команд. Цель генерации определяется с помощью тестового покрытия, заданного на уровне инструкций. Тестовые программы конструируются путем комбинирования тестовых ситуаций для различных последовательностей инструкций.

1. Введение

Постоянно увеличивающаяся сложность микропроцессоров и сокращающееся время от начала проектирования до выхода продукта на рынок сделали функциональную верификацию ключевой составляющей процесса разработки микропроцессоров. Потребность в методах автоматизации верификации в настоящее время широко признана в полупроводниковой промышленности. Значительный прогресс в этой области был достигнут в результате использования формальных методов, однако, такие методы по-прежнему ограничены проверкой сравнительно небольших блоков или нацелены на решение очень специальных задач [1]. При верификации микропроцессоров на системном уровне основную роль, как и прежде, играют техники имитационного тестирования.

Системная верификация микропроцессоров обычно осуществляется с помощью тестовых программ. Тестовые программы могут быть получены разными способами, например, с помощью кросс-компиляции существующего программного обеспечения. Другой широко распространенной техникой построения тестовых программ является случайная генерация. Следует отметить, что подобного рода методы не являются систематичными и не дают гарантий относительно качества верификации.

Анализ ошибок в микропроцессоре MIPS R4000 PC/SC показывает, что большинство из них (56.5%) связаны с одновременными взаимодействиями между несколькими подсистемами микропроцессора [2, 3]. Для того чтобы создать такие ситуации, необходимо обеспечить соблюдение нескольких ограничений на разные инструкции тестовой программы. Ошибки такого рода сложно найти, используя тесты, написанные вручную, поскольку число тестовых вариантов может быть огромным. Программы, сгенерированные случайно, в принципе, могут их обнаружить, но вероятность выполнения всех ограничений, как правило, настолько мала, что создание соответствующих ситуаций может потребовать очень больших затрат по времени [3].

Предлагаемый в данной работе метод направлен на обнаружение ошибок вышеописанного типа. Следует сразу отметить, что метод не призван заменить другие подходы к верификации микропроцессоров, он занимает свою собственную нишу и может быть рассмотрен как дополнение к существующим техникам. В соответствии с предлагаемым методом мы разработали генератор тестовых программ TestFusion 4M. Это генератор на основе моделей, который использует тестовое покрытие, заданное на уровне инструкций, и комбинаторные техники построения разнообразных последовательностей

инструкций. Генератор был успешно применен в ряде проектов. В настоящей статье описываются два из них.

Оставшаяся часть статьи организована следующим образом. Во втором разделе делается обзор существующих подходов к генерации тестовых программ. Третий раздел посвящен описанию предлагаемого метода. В этом разделе также рассматривается организация модели микропроцессора и структура тестового покрытия. Четвертый раздел очерчивает направления дальнейших исследований. Они касаются вопросов автоматического извлечения тестового покрытия и построения тестовых шаблонов на основе анализа модели микропроцессора. В пятом разделе описываются два примера использования метода. Наконец, шестой раздел завершает статью.

2. Методы генерации тестовых программ

В настоящее время распространены следующие методы построения тестовых программ:

- ручная разработка тестовых программ;
- кросс-компиляция существующего программного обеспечения;
- случайная генерация тестовых программ;
- генерация тестовых программ на основе шаблонов.

Ручная разработка тестовых программ достаточно часто используется для проверки так называемых крайних случаев. Инженеры-верификаторы, разрабатывающие такие тесты, должны знать детали реализации верифицируемого микропроцессора, чтобы создать соответствующие ситуации и проверить правильность поведения микропроцессора в них. Очевидно, что такой подход является низко продуктивным. Еще один недостаток подхода заключается в том, что важный для тестирования случай может быть упущен [4].

Другим популярным методом является тестирование с помощью существующего программного обеспечения. Такое тестирование всегда проводится для микропроцессоров общего назначения. Как минимум, микропроцессор проверяют на одной или нескольких операционных системах. Следует отметить, что программы, полученные таким образом, не дают гарантий относительно качества тестирования. Это объемные тесты, которые широко, но не достаточно глубоко охватывают функциональность микропроцессора.

Самым простым методом автоматизированного построения тестовых программ является случайная генерация. Программы, построенные случайным образом, позволяют быстро обнаруживать сравнительно простые ошибки. Другое достоинство метода состоит в том, что с его помощью можно создать ситуации, которые сложно представить, но которые в то же время интересны для тестирования [4]. Однако, полностью случайная генерация практически бесполезна для создания сложных ситуаций в работе микропроцессора.

На наш взгляд, самым перспективным способом автоматизированного построения тестовых программ является генерация на основе шаблонов. Тестовым шаблоном называется абстрактное представление тестовой программы или ее фрагмента, которое, во-первых, фиксирует или ограничивает последовательность инструкций, а во-вторых, задает ограничения на значения операндов инструкций и состояние микропроцессора. При построении тестовой программы генератор находит случайное решение соответствующей системы ограничений. Использование тестовых шаблонов позволяет уменьшить трудозатраты на верификацию, но если тестовые шаблоны разрабатываются вручную, сохраняется возможность пропустить важный тестовый вариант.

2.1. Обзор работ

Компания IBM использует автоматические генераторы тестовых программ, начиная с середины 1980-х [5]. Потребность в общем методе, пригодном для широкого класса

микропроцессорных архитектур, привела компанию к подходу на основе моделей. В рамках этого подхода генератор разбивается на два основных компонента: независимое от целевого микропроцессора ядро и модель, описывающую тестируемый микропроцессор. К настоящему времени компанией IBM разработан генератор на основе шаблонов Genesys-Pro [5]. Достоинствами Genesys-Pro являются выразительный язык описания тестовых шаблонов и удобная среда моделирования микропроцессорных архитектур.

Интересный метод верификации микропроцессоров с конвейерной архитектурой предлагается П. Мишрой из Университета Флориды и Н. Дуттом из Центра встроенных компьютерных систем Калифорнийского университета г. Ирвин. Авторы используют язык EXPRESSION для описания конвейера микропроцессора. Разработанное описание транслируется в модель на входном языке инструмента SMV [8]. Инженер-верификатор задает набор темпоральных свойств, описывающих различные ситуации в работе конвейера (пересылки данных, передачи управления и т.п.). После этого с помощью SMV строятся контрпримеры для отрицаний заданных свойств, используя техники проверки моделей. Построенные контрпримеры отображаются в тестовые программы. Важной чертой подхода является его целенаправленность (один тест для покрытия одного свойства). Однако предлагаемая методология не масштабируется на сложные промышленные проекты.

Ш. Ур и Я. Ядин из Исследовательской лаборатории IBM в г. Хайфе предлагают метод генерации тестов на основе обхода графа состояний конечно-автоматной модели микропроцессора. Суть метода заключается в следующем. Инженер-верификатор разрабатывает модель микропроцессора на входном языке SMV. После этого с помощью инструмента CFSM строится множество путей (абстрактных тестов), покрывающих все дуги в графе состояний конечного автомата, извлеченного из модели. Абстрактные тесты транслируются в описания тестовых шаблонов генератора Genesys, который на их основе генерирует тестовые программы. Метод позволяет достичь хорошего покрытия управляющей логики микропроцессора, но у него есть два основных недостатка. Во-первых, необходим опытный эксперт для разработки модели микропроцессора. Во-вторых, для возможности отображения абстрактных тестов в конкретные последовательности инструкций, необходимо создавать достаточно сложное описание в Genesys.

Другой метод, основанный на конечно-автоматных моделях, предлагается К. Кохно и Н. Мацумото из компании Toshiba [11]. Исследователи воплотили свой подход в инструменте mVpGen. Единственной входной информацией для этого инструмента является спецификация конвейера. На основе такой спецификации mVpGen автоматически генерирует тестовые варианты и конечно-автоматную модель микропроцессора. Тестовые варианты представляют собой состояния автомата, в которых выполняющиеся инструкции вступают в конфликты. Для каждого достижимого тестового варианта строится тестовый шаблон — путь из начального состояния автомата в состояние, соответствующее тестовому варианту. Наконец, по тестовым шаблонам генерируются тестовые программы.

Принципиально другой метод, основанный на использовании генетических алгоритмов, предлагается Ф. Корно, Ж. Кумани и др. из Туринского политехнического университета. Генерация осуществляется на основе библиотеки инструкций, которая описывает синтаксис языка ассемблера целевого микропроцессора. Тестовая программа представляется как ациклический граф, каждая вершина которого соответствует инструкции программы. Вершина графа содержит ссылку на описание инструкции в библиотеке и, если это необходимо, значения операндов. Тестовая программа строится путем мутации структуры графа и значений операндов инструкций внутри отдельных вершин. Предлагаемый подход является достаточно гибким и универсальным, он

позволяет достичь высокого уровня тестового покрытия для различных метрик качества тестирования, но ценой значительного времени генерации.

2.2. Ниша методов комбинаторной генерации

Подводя итог, отметим, что многие исследователи считают, что генерация тестовых программ на основе моделей имеет много преимуществ. Основой вопрос, который возникает, связан с выбором адекватных моделей. Большое число работ посвящено генерации тестовых программ на основе потактовых моделей. Такие методы предназначены для увеличения уровня тестового покрытия, достигнутого существующими тестами [13, 14], а также для генерации тестов, нацеленных на решение специфических задач [6].

Отметим следующие проблемы, связанные с использованием потактовых моделей. Есть два способа получения модели — либо она извлекается автоматически на основе анализа описания микропроцессора уровня регистровых передач [2, 4, 13, 14], либо разрабатывается вручную [9, 11]. Автоматическое извлечение модели является сложной задачей и требует наличия в коде аннотаций [2] или привлечения эвристик [13, 14]. Для сложных микропроцессоров автоматическое извлечение модели управляющей логики практически неосуществимо. При построении модели вручную возникает другая проблема — построенную модель сложно отлаживать [9]. Следует также отметить, что имеется сложность с использованием потактовых моделей на ранних стадиях разработки микропроцессора, поскольку потактовое функционирование еще точно не определено.

В настоящей работе рассматривается генерация тестовых программ на основе моделей уровня инструкций. Такие модели используются для генерации основной массы тестов для микропроцессоров. Генерация на уровне инструкций имеет два полюса: полностью случайная генерация и генерация на основе сценариев. Первый способ не требует больших затрат и позволяет генерировать большой объем ненаправленных тестов. Второй способ является более сложным. Это разновидность генерации на основе тестовых шаблонов, в которой используются сложные целенаправленные шаблоны, называемые сценариями. Этот подход поддерживается в промышленных инструментах, таких как Genesys-Pro [5].

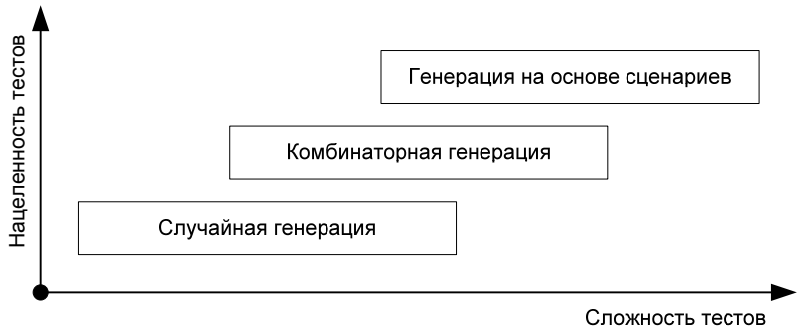

Рисунок 1. Ниша комбинаторных методов генерации тестовых программ.

Общение с инженерами-верификаторами показывает, что существует определенный пробел между случайной генерацией тестовых программ и генерацией на основе сценариев. Мы полагаем, что этот пробел может быть заполнен комбинаторными методами (см. Рис. 1). Такие методы используют тестовое покрытие, заданное на уровне инструкций, и комбинаторные техники для автоматического построения тестовых шаблонов. Построенные комбинаторным способом шаблоны не такие сложные и осмысленные как сценарии, разработанные вручную, но они генерируются автоматически, причем систематичным образом. Наш опыт показывает, что это позволяет обнаружить

дополнительные ошибки, которые пропускаются при использовании случайной генерации и генерации на основе сценариев.

3. Описание метода

Идея предлагаемого метода основана на предположении, что поведение микропроцессора зависит от множества выполняемых инструкций (состояние конвейера), зависимостей между ними (через регистры или память) и ситуаций (событий), возникающих при выполнении инструкций (исключения, попадания/промахи в кэш и т.п.).

3.1. Основные понятия метода

В данном разделе рассматриваются основные понятия предлагаемого метода: понятия *тестового шаблона*, *зависимости*, *тестовой ситуации* и *тестового воздействия*. Прежде всего, рассмотрим структуру генерируемых тестовых программ. Ее можно описать формулой $\pi = \pi_{start} \cdot \{\langle \pi_i, x_i[s_i, d_i]\rangle\}_{i=1,n} \cdot \pi_{stop}$, где:

- π_{start} — *инициализирующая программа*

  префикс тестовой программы, который содержит инструкции, предназначенные для инициализации микропроцессора;
- $\langle \pi_i, x_i[s_i, d_i]\rangle$ — *тестовый вариант* ($i = 1, \ldots, n$):
  - π_i — *программа подготовки тестового воздействия*

    последовательность инструкций, осуществляющая инициализацию регистров и памяти микропроцессора;
  - $x_i[s_i, d_i]$ — *тестовое воздействие*

    специально подготовленная последовательность инструкций, предназначенная для тестирования микропроцессора, где s_i — *множество тестовых ситуаций*, а d_i — *множество зависимостей*;
- π_{stop} — *завершающая программа*

  постфикс тестовой программы, который содержит инструкции, предназначенные для завершения работы микропроцессора;
- n — *размер тестовой программы*

  число тестовых воздействий в программе.

Ключевым понятием предлагаемого метода является понятие *тестового воздействия*. Тестовое воздействие описывается с помощью *тестового шаблона* (представляющего собой последовательностью инструкций без конкретных значений операндов), *множества зависимостей* (которые определяют, как операнды различных инструкций связаны друг с другом) и *тестовых ситуаций* (которые ограничивают значения операндов и состояние микропроцессора). Целью генерации является систематичный перебор тестовых шаблонов, зависимостей и тестовых ситуаций.

Тестовый шаблон. Под тестовыми шаблонами в данной статье понимаются последовательности инструкций без конкретных значений операндов. Они предназначены для создания различных состояний конвейера микропроцессора. Для каждого тестового шаблона обычно генерируется множество тестовых воздействий, которые отличаются друг от друга зависимостями между инструкциями и тестовыми ситуациями.

Анализ статистики показывает, что большое число ошибок (конечно, не все) могут быть обнаружены с помощью коротких последовательностей инструкций (2–5 инструкции), поэтому мы предлагаем использовать относительно короткие шаблоны в рамках комбинаторной генерации тестовых программ. Очевидно, что даже при небольшой длине общее число тестовых шаблонов может быть значительным. Для сокращения числа тестов

следует использовать специальные эвристики. Например, схожие инструкции могут быть объединены в классы эквивалентности. Тестовые шаблоны, в которых на одинаковых позициях находятся эквивалентные инструкции, можно считать эквивалентными. Класс эквивалентности тестовых шаблонов называется *обобщенным тестовым шаблоном*.

Для построения тестовых программ обычно используется перебор всех обобщенных тестовых шаблонов ограниченной длины. Например, мы часто используем пары или тройки инструкций. Следует отметить, что генератор TestFusion 4M поддерживает и другие способы комбинирования инструкций, называемые *комбинаторами*. Он также позволяет декомпозировать тестовые шаблоны на слабо связанные секции, каждую из которых можно строить с помощью собственного комбинатора.

Зависимость. Использование различных тестовых шаблонов, как правило, не является достаточным условием для качественной верификации микропроцессора. Микропроцессор может по-разному выполнять инструкции в зависимости от того, как они связаны между собой. В предлагаемом методе используются два типа зависимостей — *зависимости по регистрам* и *зависимости по адресам*. Зависимости по регистрам выражаются с помощью равенств и неравенств регистров, использующихся в качестве операндов инструкций тестового воздействия. Зависимости по адресам тесно связаны с иерархией памяти микропроцессора. В качестве примера можно рассмотреть следующие зависимости: совпадение виртуальных адресов, совпадение физических адресов, совпадение страниц виртуальной памяти, совпадение используемых строк кэша.

Проиллюстрируем зависимость по адресам на примере микропроцессора RM7000 [15]. Рассмотрим зависимость следующего вида — совпадение номера строки в кэше данных L1 для двух инструкций загрузки/сохранения. Обозначим эту зависимость символом `L1RowEqual`. Физический адрес RM7000 состоит из 36 бит: биты [11:5] используются для индексации одной из 128 строк, внутри каждой из которых находятся четыре 64-х битных двойных слова; биты [4:3] определяют номер одного из четырех двойных слов; биты [2:0] задают позицию байта внутри двойного слова; биты [35:12] содержат тэг. Таким образом, данные отображаются в одну и ту же строку кэше L1 тогда и только тогда, когда биты [11:5] их адресов совпадают.

В общем случае зависимость между инструкциями (точнее семейство однотипных зависимостей) описывается с помощью набора атрибутов. Конкретные значения атрибутов фиксируют зависимость рассматриваемого типа. Помимо атрибутов описание зависимости включает в себя следующие компоненты:

- *итератор* — перебирает допустимые комбинации значений атрибутов;
- *предусловие* — проверяет допустимость использования зависимости для пары операндов различных инструкций;
- *конструктор* — конструирует (полностью или частично) значение зависимого операнда по значениям операндов, от которых он зависим через зависимости данного типа.

Рассмотрим зависимость `L1RowEqual`. Эта зависимость описывается одним булевым атрибутом. Итератор перебирает значения `{true, false}` в некотором порядке. Предусловие зависимости проверяет, могут ли используемые виртуальные адреса быть транслированы в физические (в противном случае, когда, например, используются некорректные значения адресов, доступа в кэш-память не происходит). Если существует зависимость, чей атрибут имеет значение `true`, конструктор копирует биты [11:5] адреса из определяющего операнда в зависимый; если атрибуты всех зависимостей имеют значение `false`, конструктор генерирует случайное значение бит адреса операнда, отличное от уже используемых.

Тестовая ситуация. Обычно инструкции по-разному выполняются в зависимости от значений операндов и состояния микропроцессора. Например, инструкция, которая может вызвать исключение, имеет два альтернативных пути выполнения: нормальное выполнение и выполнение, вызывающее исключение. В этой статье мы используем термин *тестовая ситуация*, который означает один из возможных путей выполнения инструкции. Формально, тестовой ситуацией называется ограничение на значения операндов и состояние микропроцессора. Тестовые ситуации разрабатываются на основе анализа функционального описания системы команд. Рассмотрим, например, описание инструкции `add` из руководства по MIPS64 [16]:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← sign_extend(temp31..0)
endif
```

В этом примере первая ветвь соответствует отрицанию предусловия инструкции — требуется, чтобы оба регистра `rs` и `rt` были инициализированы 32-х битным словами. Предикат $\texttt{temp}_{32} \neq \texttt{temp}_{31}$ определяет тестовую ситуацию, соответствующую исключению целочисленного переполнения; альтернативная ситуация соответствует нормальному выполнению инструкции.

Как и зависимость, тестовая ситуация (точнее семейство однотипных тестовых ситуаций) описывается набором атрибутов, а конкретные значения атрибутов фиксируют тестовую ситуацию рассматриваемого типа. Помимо атрибутов описание тестовой ситуации включает следующие компоненты:

- *итератор* — перебирает допустимые значения атрибутов;
- *конструктор* — конструирует значения операндов инструкции;
- *препаратор* — строит программу подготовки тестовой ситуации, которая инициализирует операнды инструкции и состояние микропроцессора.

Рассмотрим инструкцию `add`. Тестовая ситуация для этой инструкции (на некотором уровне абстракции) параметризуется одним булевым атрибутом `IntegerOverflow`. Если атрибут имеет значение `true`, это означает, что значения операндов должны быть такими, чтобы происходило целочисленное переполнение; в противном случае переполнения быть не должно. Конструктор генерирует случайные значения независимых операндов такие, что условие переполнения истинно тогда и только тогда, когда `IntegerOverflow` имеет значение `true`. Препаратор для каждого независимого операнда добавляет в программу подготовки тестовой ситуации специальные инструкции, загружающие сконструированное значение в соответствующий регистр.

Тестовое воздействие. Тестовым воздействием называется последовательность инструкций с заданными значениями операндов, выполнение которой начинается в заданном состоянии микропроцессора. Будем называть тестовые воздействия эквивалентными, если у них эквивалентны тестовые шаблоны, зависимости между инструкциями и тестовые ситуации. Под *обобщенными тестовыми воздействиями* будем понимать классы эквивалентности тестовых воздействий.

Целью генерации является конструирование в тестовых программах всех обобщенных тестовых воздействий. Для достижения этой цели решаются две задачи: перебор

обобщенных тестовых воздействий и конструирование тестовых воздействий. На каждом шаге перебора, по сути, формулируется набор ограничений на тестовое воздействие, которые разрешаются на стадии конструирования. Для перебора обобщенных тестовых воздействий используются итераторы, для конструирования тестовых воздействий — конструкторы и препараторы.

3.2. Модель микропроцессора

Предлагаемый метод генерации тестовых программ основан на использовании модели микропроцессора. Модель содержит статическую и динамическую информацию о верифицируемом микропроцессоре. Статическая составляющая включает структуру подсистем микропроцессора, описания инструкций и другую информацию, которую обычно называют архитектурой. Динамическая часть модели является абстрактным представлением состояния микропроцессора. Генератор интерпретирует инструкции, добавляемые в тестовую программу, путем изменения модельного состояния. Это позволяет контролировать предусловия инструкций и корректно подготавливать тестовые ситуации. Следует отметить, что генерация на основе модели позволяет создавать самопроверяющие тестовые программы, которые содержат встроенные проверки корректности состояния микропроцессора после выполнения каждого тестового воздействия.

Основной частью модели является описание системы команд микропроцессора. Описание отдельной инструкции включает следующие компоненты:

- *интерфейс инструкции* — описывает операнды инструкции. Определение каждого операнда содержит имя, тип (непосредственный или регистровый), тип данных (слово, число с плавающей точкой и т.п.) и направление потока данных (входной, выходной или одновременно входной и выходной);
- *предусловие инструкции* — определяет ситуации, в которых выполнение инструкции является предсказуемым;
- *функция выполнения инструкции* — вычисляет значения выходных операндов инструкции и обновляет модельной состояние микропроцессора;
- *ассемблерный формат* — определяет запись инструкции на языке ассемблера.

3.3. Генератор TestFusion 4M

Разработанный нами генератор тестовых программ TestFusion 4M в качестве входной информации берет модель микропроцессора, описания тестового покрытия (тестовых ситуаций и зависимостей), а также параметры генерации. Часть инструкций помечается как “верифицируемые” — эти инструкции используются для составления тестовых воздействий.

Генерация тестовых программ осуществляется следующим образом. Перебираются тестовые шаблоны, зависимости и тестовые ситуации. Сначала генератор распределяет регистры в соответствии с регистровыми зависимостями. Далее для каждой инструкции тестового воздействия он конструирует адресные зависимости и тестовые ситуации, после чего создает программу подготовки тестовой ситуации. Из программ подготовки тестовых ситуаций генератор конструирует суммарную программу подготовки тестового воздействия. Процесс продолжается до тех пор, пока не будут перебраны все обобщенные тестовые воздействия. Ниже представлена упрощенная схема генерации:

- получить очередной тестовый шаблон
    - получить очередное множество зависимостей по регистрам
    - сконструировать зависимости по регистрам

- получить очередное множество тестовых ситуаций*
  - получить очередное множество зависимостей по адресам
  - для каждой инструкции тестового воздействия:
    - проверить предусловие:
      если предусловие нарушено, перейти к *
    - проверить наличие зависимостей по адресам:
      если зависимости существуют, сконструировать их
    - сконструировать тестовую ситуацию для инструкции
    - получить программу подготовки тестовой ситуации
  - сконструировать программу подготовки тестового воздействия
  - проинтерпретировать программу подготовки
  - проинтерпретировать тестовое воздействие

На первый взгляд, задача построения суммарной программы подготовки тестового воздействия является тривиальной — достаточно последовательно объединить программы подготовки тестовых ситуаций для всех инструкций тестового воздействия. Обычно такой способ работает, но не всегда. Бывают ситуации, когда программа подготовки тестовой ситуации влияет на предшествующие инструкции. В таких ситуациях, инженер-верификатор разбивает программы подготовки тестовых ситуаций на несколько фрагментов. Каждый фрагмент отвечает за инициализацию определенной подсистемы микропроцессора. Разработчик упорядочивает фрагменты таким образом, чтобы каждый следующий фрагмент не влиял на подсистемы, подготовленные предыдущими фрагментами. Чтобы такое упорядочивание было возможно, в графе, отражающем влияние подготовки одной подсистемы на состояние других подсистем не должно быть циклов; в противном случае следует объединять компоненты сильной связности графа в более крупные подсистемы и строить программы подготовки для таких объединенных подсистем.

Генератор TestFusion 4M имеет гибкую архитектуру, отражающую основные концепции метода. Основные компоненты генератора отвечают за перебор тестовых шаблонов, зависимостей и тестовых ситуаций. Эти компоненты образуют ядро генератора. Пользователь может настраивать эти компоненты, выбирая подходящие значения параметров. Помимо ядра в генераторе есть набор библиотек, упрощающих разработку моделей микропроцессоров, которые также содержат готовые к использованию компоненты, такие как комбинаторы, генераторы тестовых данных и др. Для увеличения удобства использования TestFusion 4M обладает графическим интерфейсом (см. Рис. 2).

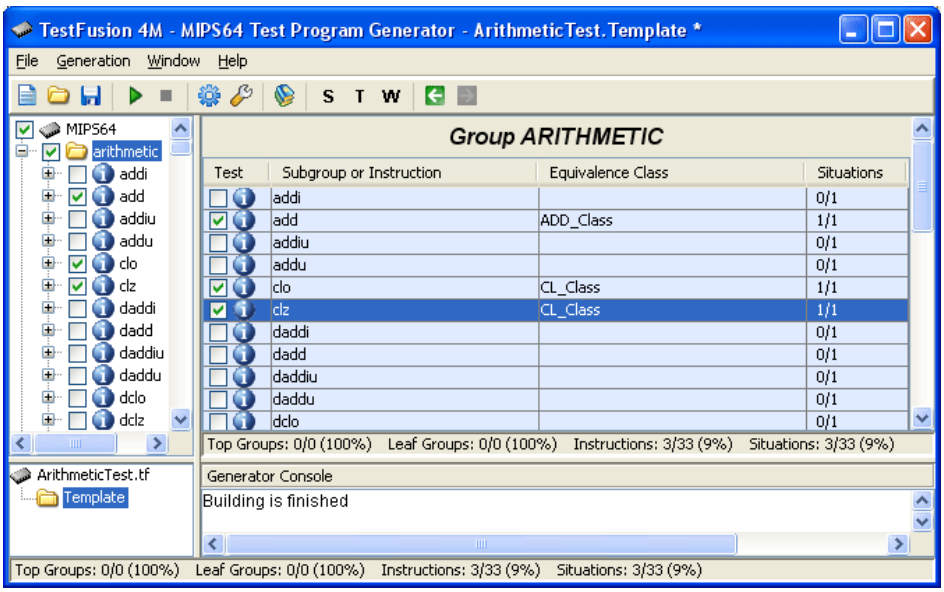

Рисунок 2. Графический интерфейс TestFusion 4M.

4. Дальнейшие исследования

В данном разделе рассматриваются некоторые идеи, над которыми мы планируем работать в ближайшее время. Эти идеи касаются вопросов автоматического извлечения тестового покрытия (тестовых ситуаций и зависимостей) из модели микропроцессора, а также вопросов автоматизированного построения тестовых шаблонов. Модель микропроцессора содержит описания инструкций и подсистем, которые, будучи формальными, могут быть использованы для автоматического извлечения тестовых ситуаций и зависимостей между инструкциями.

Семантика отдельной инструкции может быть представлена с помощью *дерева выполнения*. Вершины дерева выполнения соответствуют подсистемам микропроцессора, а дуги описывают пути передачи управления. Каждая дуга снабжена предикатом, который определяет условие соответствующего перехода (см. Рис. 3). Деревья выполнения могут использоваться для факторизации инструкций. Например, две инструкции можно считать эквивалентными, если у них одинаковые деревья выполнения или одинаковые префиксы деревьев выполнения. Подобная факторизация позволяет уменьшить суммарный объем генерируемых тестовых шаблонов. Что является еще более важным, деревья выполнения можно использовать для автоматического извлечения тестовых ситуаций. Каждая ситуация соответствует пути (ветви) в дереве выполнения инструкции. Например, фрагмент дерева выполнения на Рис. 3 определяет четыре тестовые ситуации: `{TLB[Hit, Valid], L1[Hit]}`, `{TLB[Hit, Valid], L1[Miss]}`, `{TLB[Hit, Invalid]}` и `{TLB[Miss]}`.

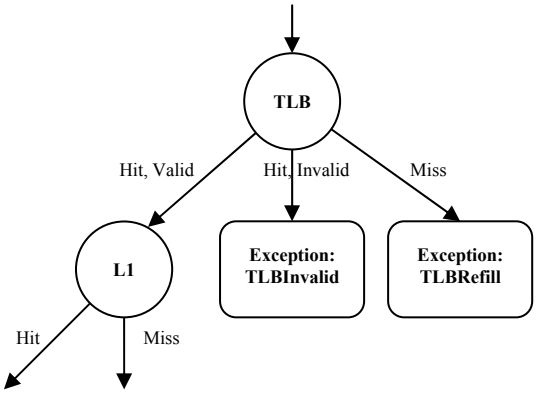

Рисунок 3. Фрагмент дерева выполнения инструкций загрузки/сохранения.

Для возможности автоматического извлечения зависимостей между инструкциями, описания подсистем должны включать в себя всю необходимую информацию: тип ресурса (прямого отображения, частично-ассоциативный, ассоциативный и др.), число элементов, структура элементов, функция вычисления тэга (для буферов) и др. Например, описание TLB могло бы выглядеть следующим образом:

```
associative buffer TLB<TLB_ENTRY, 64> {
   VIRTUAL_PAGE_NUMBER tag(VIRTUAL_ADDRESS va) { ... }
}

structure TLB_ENTRY {
   VIRTUAL_PAGE_NUMBER  vpn;
   PHYSICAL_PAGE_NUMBER pfn;
}
```

Из этого описания можно автоматически извлечь зависимость типа TLB_ENTRY_EQUAL, которая описывает доступ к одной ячейке TLB из двух разных инструкций загрузки/сохранения.

Теперь рассмотрим конструирование тестовых шаблонов. Один из возможных сценариев взаимодействия с генератором может быть следующим. Инженер-верификатор выбирает подсистемы, которые он или она хочет проверить. Генератор анализирует деревья выполнения инструкций и определяет, какие из них используют выбранные подсистемы. Затем он факторизует эти инструкции, если это необходимо, и извлекает тестовые ситуации и зависимости для них. Наконец, он запускает генерацию тестовых шаблонов. Основной вопрос, который здесь возникает — как конструировать тестовые шаблоны для заданного множества инструкций. Один из обещающих подходов основан на обходе графа состояний конечно-автоматной модели.

Поскольку мы используем высокоуровневые модели, невозможно автоматически извлечь потактовую конечно-автоматную модель микропроцессора. Однако можно построить *абстрактную автоматную модель конвейера* путем добавления в вершины деревьев выполнения специальных атрибутов (время выполнения, способность параллельной обработки инструкций и т.п.). Используя эти атрибуты, можно настраивать временны*е* аспекты выполнения инструкций на конвейере. Состояния абстрактного автомата могут быть следующего типа $\{(branch_i, node_i, time_i)\}_{i=1,n}$, где $branch_i$ — ветвь в дереве выполнения, $node_i$ — текущая вершина, $time_i$ — время обработки инструкции в текущей вершине (см. Рис. 4).

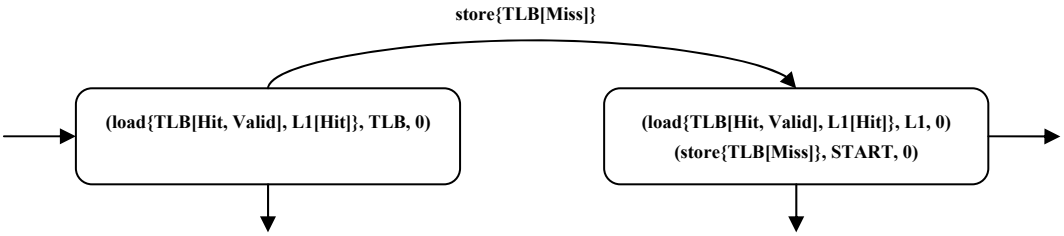

Рисунок 4. Фрагмент абстрактной автоматной модели конвейера.

Целью генерации является обход графа состояний абстрактной автоматной модели конвейера. На каждом шаге генератор выбирает инструкцию для добавления в тестовый шаблон, тестовую ситуацию (ветвь в дереве выполнения) и зависимости между этой инструкцией и инструкциями, выполняемыми в настоящее время (которые присутствуют в состоянии). Для уменьшения суммарного размера конструируемых шаблонов генератор может использовать определенные эвристики, например, он может ограничить максимальное число зависимостей между инструкциями.

Еще одним интересным направлением для дальнейших работ является интеграция в генератор TestFusion 4M возможностей, реализуемых генераторами OTK [17] и Pinery [18], которые предназначены для построения тестовых данных со сложной структурой и которые, также как TestFusion 4M, разработаны в Институте системного программирования РАН. Эти возможности, в частности, можно использовать для построения разнообразных графов потоков управления и потоков данных тестовых программ.

5. Примеры использования метода

Генератор TestFusion 4M был использован в двух промышленных проектах: верификация подсистемы управления памятью MIPS64-совместимого микропроцессора и системная верификация другого MIPS64-совместимого микропроцессора. Следует отметить, что в обоих проектах помимо комбинаторной генерации были использованы и другие методы верификации, такие как случайная генерация, генерация на основе сценариев и ручная разработка тестов. Тестовые программы, сгенерированные TestFusion 4M, обнаружили дополнительные ошибки, которые были пропущены другими методами.

5.1. Верификация подсистемы управления памятью

Тестовые воздействия на подсистему управления памятью были организованы как пары инструкций загрузки/сохранения: `lb` (загрузка байта), `ld` (загрузка двойного слова), `sb` (сохранение байта) и `sd` (сохранение двойного слова). Тестовые ситуации для инструкций были параметризованы следующими атрибутами:

- `isMapped` — отображаемый/неотображаемый сегмент виртуальной памяти;
- `isCached` — кэшируемый/некэшируемый сегмент виртуальной памяти;
- `tlbHit` — попадание/промах в TLB;
- `DVG` — управляющие биты секции TLB[1];

[1] В MIPS64-совместимых микропроцессорах ячейка TLB состоит из двух секций — первая предназначена для страницы виртуальной памяти с четным номером, вторая — для страницы с нечетным номером.

- `dtlbHit` — попадание/промах в DTLB[2];
- `cachePolicy` — политика кэширования;
- `l1Hit` — попадание/промах в кэш-память L1;
- `l2Hit` — попадание/промах в кэш-память L2.

Зависимости между инструкциями описывались в помощью следующих атрибутов:

- `vaEqual` — совпадение/несовпадение виртуальных адресов;
- `tlbEqual` — совпадение/несовпадение ячеек TLB;
- `pageEqual` — совпадение/несовпадение секций в ячейке TLB;
- `paEqual` — совпадение/несовпадение физических адресов;
- `l1RowEqual` — совпадение/несовпадение номеров строк в кэш-памяти L1;
- `l2RowEqual` — совпадение/несовпадение номеров строк в кэш-памяти L2;
- `dtlbReplace` — совпадение/несовпадение ячейки TLB, используемой второй инструкцией, с ячейкой, вытесненной из DTLB первой инструкцией;
- `l1Replace` — совпадение/несовпадение тэга кэш-памяти L1, вычисленного по физическому адресу, используемого во второй инструкции, с тэгом данных, который был вытеснен из кэш-памяти L1 первой инструкцией;
- `l2Replace` — совпадение/несовпадение тэга кэш-памяти L2, вычисленного по физическому адресу, используемого во второй инструкции, с тэгом данных, который был вытеснен из кэш-памяти L2 первой инструкцией.

Мы обнаружили одну критическую ошибку в подсистеме управления памятью, которая возникала, только когда были выполнены определенные ограничения на инструкции и зависимости между ними.

5.2. Верификация MIPS64-совместимого микропроцессора

Самым масштабным применением описанного подхода является системная верификация другого MIPS64-совместимого микропроцессора. В этом проекте мы в качестве тестовых воздействий использовали тройки инструкций. Общее число инструкций 221[3]. Все инструкции были разбиты на 13 групп:

- `arithmetic` (33 инструкции);
- `logic` (8);
- `move` (8);
- `shift` (15);
- `branch` (20);
- `nop` (2);
- `memory` (26);
- `interrupt` (14);
- `system` (13);

---

[2] DTLB (Data TLB) — небольшой буфер, кэширующий обращения к TLB при трансляции адресов данных.

[3] Инструкции, которые отличались форматом операндов, например, `add.s` (сложение чисел с плавающей точкой одинарной точности) и `add.d` (сложение чисел с плавающей точкой двойной точности), при подсчете считались разными инструкциями.

- `fpu.arithmetic` (24);
- `fpu.move` (26);
- `fpu.convert` (26);
- `fpu.branch` (6).

Для сокращения общего размера тестовых программ мы использовали дополнительную эвристику — тестовые воздействия включали инструкции не более чем из двух различных групп. Тестовые ситуации и зависимости, используемые для инструкций загрузки/сохранения полностью аналогичны тем, которые были описаны выше. Тестовые данные для арифметических инструкций были направлены на создание исключительных ситуаций, а также использовали граничные значения операндов. Инструкции ветвления описывались значением истинности условия перехода (для условных переходов) и направлением перехода (вперед или назад). Было найдено 9 ошибок в описании микропроцессора уровня регистровых передач и 6 ошибок в симуляторе микропроцессора.

6. Заключение

В статье был рассмотрен метод автоматизированной генерации тестовых программ для микропроцессоров. В отличие от широко распространенных на практике методов, таких как верификация на основе существующего программного обеспечения и случайная генерация, комбинаторная генерация тестовых программ на основе моделей является более систематичной и технологичной. Наш опыт показывает, что предлагаемый метод позволяет находить такие ошибки, которые часто пропускают другие методы. Мы рассматриваем наш метод как необходимое дополнение к существующим подходам. Поскольку описание комбинаторных тестов не требует значительных трудозатрат, такое тестирование может проводиться перед применением продвинутых методов на основе сценариев. В соответствии с описанным подходом мы разработали генератор тестовых программ TestFusion 4M, который был успешно использован в ряде проектов. В дальнейшем мы планируем реализовать в генераторе возможности автоматического извлечения тестового покрытия и конструирования тестовых шаблонов на основе анализа модели микропроцессора.

Литература

1. M. Behm, J. Ludden, Y. Lichtenstein, M. Rimon, M. Vinov. *Industrial Experience with Test Generation Languages for Processor Verification*. Design Automation Conference, 2004.
2. *MIPS R4000PC/SC Errata, Processor Revision 2.2 and 3.0*. MIPS Technologies Inc., May 10, 1994.
3. R. Ho, C. Han Yang, M. Horowitz, D.L. Dill. *Architecture Validation for Processors*. International Symposium on Computer Architecture, 1995.
4. R. Ho. *Validation Tools for Complex Digital Designs*. PhD Thesis. November, 1996.
5. A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. *Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification*. Design and Test, 2004.
6. P. Mishra, N. Dutt. *Automatic Functional Test Program Generation for Pipelined Processors Using Model Checking*. IEEE International High-Level Design Validation and Test Workshop, 2002.

7. P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, A. Nicolau. *EXPRESSION: An ADL for System Level Design Exploration*. Technical Report 1998-29, University of California, Irvine, 1998.
8. www.cs.cmu.edu/~modelcheck/smv.html.
9. S. Ur and Y. Yadin. *Micro Architecture Coverage Directed Generation of Test Programs*. Design Automation Conference, 1999.
10. D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, Y. Wolfsthal. *Coverage Directed Test Generation Using Symbolic Techniques*. Formal Methods in Computer Aided Design, 1996.
11. K. Kohno, N. Matsumoto. *A New Verification Methodology for Complex Pipeline Behavior*. Design Automation Conference, 2001.
12. F. Corno, M. Sonza Reorda, G. Squillero, M. Violante. *A Genetic Algorithm-Based System for Generating Test Programs for Microprocessor IP Cores*. IEEE International Conference on Tools with Artificial Intelligence, 2000.
13. D. Moundanos, J. Abraham, Y. Hoskote. *A Unified Framework for Design Validation and Manufacturing Test*. International Test Conference, 1996.
14. D. Moundanos, J. Abraham, Y. Hoskote. *Abstraction Techniques for Validation Coverage Analysis and Test Generation*. IEEE Transactions on Computers, Vol. 47, 1998.
15. *RM7000 Family User Manual*. Issue 1, May 2001.
16. *MIPS64™ Architecture For Programmers*. Revision 2.0. MIPS Tecnologies Inc., June 9, 2003.
17. A.S. Kossatchev, A.K. Petrenko, S.V. Zelenov, S.A. Zelenova. *Application of Model-Based Approach for Automated Testing of Optimizing Compilers*. International Workshop on Program Understanding, 2003.
18. А.В. Демаков, С.В. Зеленов, С.А. Зеленова. *Генерация тестовых данных сложной структуры с учетом контекстных ограничений.* Труды Института системного программирования РАН, 2006.