Генерация тестовых программ для микропроцессоров

А. С. Камкин
kamkin@ispras.ru

Аннотация. В работе описывается подход к автоматической генерации тестовых программ, предназначенный для систематичного функционального тестирования микропроцессоров. Предлагаемый подход дополняет такие широко распространенные на практике методы, как тестирование с помощью существующего программного обеспечения и тестирование с помощью случайных программ. Генерация тестовых программ осуществляется на основе модели микропроцессора, включающей в себя структурную модель микропроцессора и модель системы команд. Цель генерации задается с помощью критерия тестового покрытия, выделяющего набор тестовых ситуаций для каждой инструкции микропроцессора. Помимо описания методики генерации тестовых программ в статье также описывается устройство генератора, реализующего эту методику, и опыт его использования для тестирования микропроцессоров.

1. Введение

Микропроцессоры являются основой любой компьютерной системы, поэтому от правильности их работы зависит корректность и надежность системы в целом. С ростом использования компьютеров в нашей жизни растет и ответственность разработчиков за качество создаваемых систем. Научно-технические расчеты, военное дело, управление транспортом, медицинские системы — вот далеко не полный перечень приложений микропроцессорных систем, в которых ошибка может иметь тяжелые последствия. Даже если ошибка не представляет угрозы для жизни или здоровья людей, она может иметь очень тяжелые экономические последствия для компании, производящей микропроцессор или системы на его основе — ошибки плохо сказываются на имидже компании и конкурентоспособности ее продукции.

Современные микропроцессоры являются невероятно сложными системами — они состоят из десятков миллионов транзисторов, объединенных в сотни взаимодействующих модулей и подсистем. Для повышения производительности в микропроцессорах используется множество специальных механизмов: конвейер инструкций, суперскалярное выполнение, предсказание переходов, неблокируемое кэширование и многие другие. Все

это делает задачу *функционального тестирования микропроцессоров*[1] чрезвычайно трудоемкой. По различным данным, тестирование микропроцессора занимает около 70-80% от общего объема трудозатрат на его разработку. Поэтому развиваются методы автоматизированного тестирования, которые, с одной стороны, нацелены на повышение качества производимых микропроцессоров, с другой — на сокращение цикла разработки.

Одним из основных способов тестирования микропроцессоров является проверка корректности их работы на некотором наборе программ. Программы компилируются, загружаются в память и выполняются микропроцессором; результаты их работы протоколируются и используются для анализа правильности работы микропроцессора. *Тестовые программы*, то есть программы, которые используются для тестирования, могут быть получены разными способами. Одним из способов является кросс-компиляции *существующего программного обеспечения* (ПО). Программы, полученные таким образом, не дают гарантий относительно качества тестирования — это объемные тесты, которые широко охватывают функциональность микропроцессора, но не достаточно глубоко. Альтернативой тестам на основе существующего ПО являются программы, специально разработанные (или автоматически сгенерированные) для целей тестирования. Широко распространенным методом автоматического построения тестовых программ является *случайная генерация*, но этот метод также нельзя назвать *систематичным*. К таковым мы относим подходы, нацеленные на достижение определенного критерия тестового покрытия и дающие гарантии относительно полноты тестирования с точки зрения этого критерия.

Анализ ошибок в микропроцессоре MIPS R4000 PC/SC (ревизия 2.2) [1], проведенный в работе [2], говорит, что большинство ошибок (93.5%) связано с некорректной реализацией *управляющей логики* (*control logic bugs*). *Управляющей логикой* называется функциональность микропроцессора, отвечающая за планирование и организацию процессов выполнения инструкций: конвейер инструкций, суперскалярное выполнение, предсказание ветвлений и так далее. Для обнаружения большей части таких ошибок необходимо одновременно реализовать несколько условий, например, одна из ошибок микропроцессора проявлялась только в следующей ситуации (см. пункты 4 и 14 [1]).

[1] Тестируется обычно не сам микропроцессор, а его проектная модель *уровня регистровых передач* (*RTL*, *Register Transfer Level*), разработанная на специализированном *языке описания аппаратуры* (*HDL*, *Hardware Description Language*), например, Verilog или VHDL. Такое тестирование называется *имитационным* (*simulation-based validation*), поскольку производится в специальном *симуляторе*, осуществляющем имитационное моделирование работы микропроцессора.

- Инструкция загрузки данных в регистр вызывает промах в кэше данных.
- За ней через одну инструкцию NOP (No Operation, специальная инструкция микропроцессора, которая не производит никаких действий и обычно используется для временных задержек и выравнивания памяти) следует инструкция безусловного перехода по адресу, содержащемуся в загруженном регистре.
- Инструкция перехода — последняя инструкция на странице виртуальной памяти.
- Номер следующей страницы не содержится в буфере трансляции адресов (TLB, Translation Lookaside Buffer).

Заметим, что такие ошибки сложно обнаружить, используя существующее ПО или случайную генерацию, поскольку вероятность появления соответствующей ситуации в тестовой программе очень мала. Предлагаемый в работе подход за счет систематичного перебора и комбинирования тестовых ситуаций для отдельных инструкций позволяет обнаруживать такие ошибки; кроме того, он предоставляет разработчику тестов набор параметров, изменяя которые можно управлять глубиной тестирования. В основу работы положен опыт отдела Технологий программирования Института системного программирования РАН [3,4] разработки тестов и создания автоматических генераторов тестовых данных *на основе моделей*.

Класс ошибок	**Число ошибок**	**Процент ошибок**
Ошибки обработки данных (datapath bugs)	3	6.5%
Ошибки в управляющей логике / одно условие (control logic bugs / single event)	17	37.0%
Ошибки в управляющей логике / несколько условий (control logic bugs / multiple events)	26	56.5%
Всего ошибок	**46**	**100.0%**

Таблица 1. Соотношение разных видов ошибок в микропроцессорах [2].

Оставшаяся часть статьи организована следующим образом. Во втором, следующем за введением, разделе рассказывается о существующих методах построения тестовых программ. В этом разделе описываются распространенные подходы к тестированию микропроцессоров, делается обзор работ, посвященных генерации тестовых программ, анализируются достоинства и недостатки существующих методов. Третий раздел посвящен описанию предлагаемого подхода. В нем вводятся основные понятия подхода,

описывается используемая для генерации модель микропроцессора, рассматривается метод генерации и устройство генератора; изложение иллюстрируется примерами, основанными на системе команд MIPS64 [5]. В четвертом разделе описывается опыт практического применения подхода для тестирования микропроцессоров. Пятый раздел завершает статью и очерчивает направления дальнейших исследований.

2. Методы построения тестовых программ

Методами построения тестовых программ для функционального тестирования микропроцессоров занимаются, пожалуй, с момента появления микропроцессоров. Тем не менее, разработка новых подходов до сих пор привлекает внимание исследователей; более того, интерес к этой проблематике со временем только возрастает. На это есть своя причина — постоянно увеличивается сложность микропроцессоров, развивается архитектура, совершенствуются алгоритмы и схемы их функционирования — старые методы тестирования оказываются непригодными или малопригодными для качественной проверки микропроцессоров нового поколения.

Качество тестирования любой системы (не обязательно микропроцессора) напрямую зависит от используемого тестового набора, от того, насколько полно он охватывает ситуации, возможные в работе системы; какие внутренние взаимодействия между модулями вызывает. Пространство состояний современных микропроцессоров чрезвычайно велико, поэтому разработка высококачественного набора тестов требует значительных затрат. Очевидно, что сегодня полностью ручная разработка тестов практически неприемлема, поскольку на тестирование в этом случае уйдут годы, и не исключено, что к моменту его завершения микропроцессор уже потеряет свою актуальность на рынке.

2.1. Распространенные подходы

В настоящее время в практике тестирования микропроцессоров распространены следующие подходы:

- ручная разработка тестовых программ;
- тестирование с помощью существующего ПО;
- случайная генерация тестовых программ;
- случайная генерация тестовых программ на основе тестовых шаблонов.

Несмотря на сказанное ранее, *ручная разработка тестовых программ* достаточно часто используется для проверки так называемых *крайних случаев* (*corner cases*) в работе микропроцессора. Разработчики таких тестов должны знать детали реализации тестируемого микропроцессора, чтобы создать соответствующие ситуации и проверить правильность поведения

микропроцессора в них. Большая опасность при таком подходе кроется в том, что важный для тестирования случай будет упущен (разработчик тестов может не придать значения тестовому случаю или по невнимательности забыть реализовать нужный тест), поэтому соответствующая ситуация в работе микропроцессора не будет реализована при тестировании [6].

Другим популярным методом функционального тестирования микропроцессоров является *тестирование с помощью существующего ПО*. Такое тестирование широко распространено и всегда проводится для микропроцессоров общего назначения. Как минимум, микропроцессор проверяют на одной или нескольких известных операционных системах. Как отмечают исследователи, программы, полученные таким образом, не дают гарантий относительно качества тестирования — это объемные тесты, которые широко, но не достаточно глубоко охватывают функциональность микропроцессора.

Самым распространенным методом автоматического построения тестовых программ является *случайная генерация*. Программы, построенные таким образом, позволяют быстро обнаруживать простые ошибки. Другое достоинство случайных тестов состоит в том, что они могут создать ситуацию, которую сложно представить, но которая в то же время является интересной для тестирования [6]. Как уже отмечалось во введении, случайную генерацию нельзя назвать систематичным подходом — вероятность возникновения крайнего случая, как правило, очень мала, поэтому соответствующие ситуации скорее всего останутся непроверенными при тестировании.

В настоящее время для построения тестовых программ широко используется *случайная генерация на основе тестовых шаблонов* (*шаблонов тестовых программ*) (*test templates*, *test program templates*). Тестовым шаблоном называется абстрактная форма представления тестовой программы. Шаблоны фиксируют или некоторым образом описывают последовательность инструкций тестовой программы. Вместо конкретных значений входных операндов инструкций в тестовых шаблонах указывается набор ограничений, которым операнды должны удовлетворять перед выполнением соответствующей инструкции. Генератор, используя некоторый механизм разрешения ограничений, строит случайное значение, удовлетворяющее заданным ограничениям. Такой подход также называется *случайной генерацией на основе ограничений* (*constraint-driven random generation*). Использование тестовых шаблонов позволяет значительно сократить трудозатраты, поскольку автоматизируется рутинная работа по подбору значений операндов, требуемых для создания тестовых ситуаций. Однако тестовые шаблоны, как правило, разрабатываются вручную, поэтому, как и при ручной разработке, есть возможность упустить важную для тестирования ситуацию.

Ниже рассмотрены работы по генерации тестовых программ, большинство из которых посвящено методам автоматического построения тестовых шаблонов.

2.2. Обзор работ

Компания IBM использует автоматические генераторы тестовых программ в процессе тестирования своих микропроцессоров, начиная с середины 1980-х [7]. Потребность в общем подходе, 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* (*engine*) и *модель* (*model*), описывающую тестируемый микропроцессор. В 1991 году компанией был разработан генератор тестовых программ *Genesys*, который с 1993 года является основным генератором тестовых программ в IBM [8,9]. Генератор широко использовался как внутри компании, так и за ее пределами. В настоящее время в IBM разработана усовершенствованная версия генератора — *Genesys-Pro* [7].

Генератор тестовых программ Genesys-Pro использует три типа описаний: *ядро*, включающее общие принципы организации микропроцессоров и генерации тестов для них; *модель*, содержащую описание особенностей целевой архитектуры и некоторые знания о способах ее тестирования, и *шаблоны тестовых программ*, описывающие сценарии, которые нужно реализовать в тестовых программах. Генератор сводит задачу построения тестовых программ *к задаче поиска решения системы ограничений* (*CSP*, *Constraint Satisfaction Problem*), для решения которой используется общий механизм разрешения ограничений [7]. К достоинствам Genesys-Pro можно отнести выразительный язык описания шаблонов тестовых программ, который позволяет разработчикам тестов гибко управлять процессом генерации, а также удобную среду моделирования микропроцессорных архитектур. Подход является достаточно масштабируемым и универсальным. Недостатком, на наш взгляд, является ручная разработка тестовых шаблонов — многие шаблоны можно генерировать автоматически на основе описания архитектуры микропроцессора. Хотя этот недостаток можно исправить с помощью надстроек над генератором (см., например, [10]).

Интересный подход к тестированию микропроцессоров с конвейерной архитектурой *на основе проверки моделей* (*model checking*) предлагается Мишрой (Mishra) из Университета Флориды (University of Florida) и Дуттом (Dutt) из Центра встроенных компьютерных систем Калифорнийского университета города Ирвин (Center for Embedded Computer Systems, University of California, Irvine). Этими исследователями (иногда в соавторстве с другими) в 2002-2006 годах написано несколько работ, описывающих предлагаемый подход и результаты его апробации [11-17]. Идея подхода состоит в следующем. На основе текстового описания архитектуры микропроцессора разрабатывается формальное описание на *языке описания архитектуры* (*ADL*, *Architecture Description Language*) EXPRESSION [18]. ADL-описание

транслируется в модель на SMV (Symbolic Model Verifier)[2] [19]. Разработчик тестов указывает набор *свойств* (*properties*) в виде формул темпоральной логики, описывающих различные ситуации в работе конвейера, которые необходимо протестировать. Эти свойства отражают пути передачи управления и пути передачи данных между модулями конвейера. Авторы отмечают, что набор свойств такого вида может быть автоматически сгенерирован из ADL-описания. Модель микропроцессора и отрицания свойств подаются на вход SMV, который для каждого из указанных отрицаний пытается построить *контрпример* — вариант работы конвейера, в котором нарушается отрицание свойства — тем самым достигая описанную в исходном свойстве ситуацию. Построенные по контрпримерам тестовые программы выполняются на симуляторе, который, как и SMV-модель, может быть автоматически получен из ADL-описания. Результаты выполнения программ на симуляторе используются для проверки правильности работы микропроцессора. Для уменьшения времени генерации тестовых программ и затрат памяти, авторы предлагают использовать *ограниченную проверку модели* (*BMC*, *Bounded Model Checking*) [13] и *декомпозицию свойств* [14]. Предложенный подход был апробирован на микропроцессоре DLX[3]. Результаты апробации приведены в техническом отчете [12]. Достоинством подхода является его целенаправленность — один тест для покрытия одного свойства. Слабым местом подхода является необходимость описания всех интересных ситуаций в виде формул темпоральной логики. Конечно, как отмечают авторы, некоторый класс ситуаций может быть получен автоматически из ADL-описания, но это сравнительно небольшое множество крайних случаев. Следует также отметить, что, как и все подходы на основе проверки моделей, он подвержен *комбинаторному взрыву состояний* (*state explosion problem*). Для ограничения числа состояний в примере из технического отчета [12] авторы использовали модель памяти инструкций, состоящую всего из трех слов, а также всего три регистра общего назначения.

Подход к генерации тестовых программ *на основе конечно-автоматной модели конвейера*, использующий техники обхода графа состояний, предлагается Уром (Ur) и Ядином (Yadin) из Исследовательского центра IBM

[2] Язык и одноименный инструмент SMV разработаны в Университете Карнеги-Меллона (Carnegie Mellon University) МакМилланом (McMillan). Инструмент предназначен для проверки того, что система переходов с конечным числом состояний удовлетворяет спецификации, заданной в темпоральной логике.

[3] DLX — микропроцессор, спроектированный Хеннесси (Hennessy) и Паттерсоном (Patterson) — главными разработчиками архитектур MIPS и Berkeley RISC соответственно. Микропроцессор имеет систему команд близкую к MIPS и предназначен преимущественно для образовательных и исследовательских целей.

в городе Хайфе (IBM Haifa Research Lab) [10]. Суть подхода в следующем. Вручную строится конечно-автоматная модель конвейера микропроцессора на языке SMV, в терминах которой определяются критерии тестового покрытия и генерируются *абстрактные тесты*. Абстрактным тестом называется тестовый шаблон, который описывает путь в графе состояний конечного автомата, начинающийся и заканчивающийся в состоянии, соответствующем пустому конвейеру. Абстрактные тесты генерируются с помощью специального инструмента CFSM [20], строящего множество маршрутов, покрывающих все дуги в графе состояний конечного автомата. Абстрактные тесты транслируются в описания тестовых шаблонов генератора Genesys, который по ним строит тестовые программы. Подход был успешно апробирован на микропроцессорах семейства PowerPC[4]. Достоинством подхода является сравнительно небольшое число тестов, достигающих хорошего покрытия управляющей логики микропроцессора. Авторы отмечают два недостатка своего подхода. Во-первых, необходим опытный эксперт для создания конечно-автоматной модели, описывающей работу конвейера с потактовой точностью. Во-вторых, для возможности отображения абстрактных тестов в конкретные последовательности инструкций необходимо создавать достаточно сложное описание в Genesys.

Другой подход к генерации тестовых программ для микропроцессоров с конвейерной архитектурой *на основе конечных автоматов* предлагается Кохно (Kohno) и Мацумото (Matsumoto) из компании Toshiba [21]. Исследователи воплотили свой подход в инструменте *mVpGen*. Единственной входной информацией для mVpGen является спецификация конвейера, подробно описывающая его функционирование на уровне различных *классов инструкций*. На основе такой спецификации инструмент автоматически генерирует *тестовые случаи* (*test cases*) и *конечно-автоматную модель конвейера*. Тестовые случаи представляют собой состояния конвейера микропроцессора, в которых выполняющиеся инструкции вступают в *конфликты чтения/записи* (*data hazards*) или *конфликты использования ресурсов* (*structural hazards*). Переходы в конечно-автоматной модели определяются для классов инструкций и ограничений на значения операндов инструкций. По конечно-автоматной модели вычисляется множество достижимых тестовых случаев. Для каждого достижимого тестового случая строится тестовый шаблон — путь из начального состояния конечного автомата в состояние, соответствующее тестовому случаю. Наконец, по тестовым шаблонам генерируются тестовые программы. Подход был успешно апробирован в компании Toshiba на микропроцессоре MeP[5]. На наш взгляд,

[4] PowerPC — известная микропроцессорная RISC-архитектура, созданная в 1991 году альянсом компаний Apple-IBM-Motorola, известном как AIM.

[5] MeP (Media embedded Processor) — 32-х разрядный конфигурируемый RISC-процессор обработки медиа данных, разрабатываемый компанией Toshiba. Первое ядро MeP (MeP-c1) разработано в 2001 году.

этот подход похож на предыдущий — в обоих подходах используются схожие модели. Основное отличие состоит в разных целях и задачах тестирования — в первом подходе совершается обход графа состояний конечного автомата, моделирующего конвейер, во втором подходе полный обход не совершается, а строятся пути только в те состояния, в которых возникают конфликты между инструкциями.

Принципиально другой подход к генерации тестовых программ, основанный на использовании *генетических алгоритмов*, предлагается Корно (Corno), Кумани (Cumani), Сонца Реорда (Sonza Reorda), Сквиллеро (Squillero) и другими из Туринского политехнического университета (Politecnico di Torino). Этими исследователями в 2000-2005 годах написано большое число статей, описывающих предлагаемый подход, разработанный инструмент μGP и результаты его апробации [22-35]. Генерация программ осуществляется на основе библиотеки инструкций, которая описывает синтаксис языка ассемблера целевого микропроцессора. Программа представляется как ациклический граф, каждая вершина которого содержит ссылку на описание инструкции в библиотеке инструкций и, если это необходимо, значения операндов. Тестовые программы строятся путем мутации структуры графа и значений операндов инструкций внутри отдельных вершин. Оценочная функция, на основе которой производится генерация, в зависимости от целей тестирования может быть разной, например, учитывать *покрытие ошибок на уровне логических вентилей* (*gate-level fault coverage*, *stuck-at fault coverage*) [25], *покрытие инструкций RTL-модели* (*RTL statement coverage*) [34] или *внутренние счетчики производительности микропроцессора* (*performance counters*) [33]. Подход был апробирован на нескольких микропроцессорах, включая микропроцессоры с конвейерной архитектурой DLX/pII [31] и SPARC v8 [26,35]. Предлагаемый подход является достаточно гибким и универсальным, он позволяет достичь высокого уровня тестового покрытия для различных метрик качества тестирования, но ценой большого времени генерации тестовых программ. Как правило, для каждой конкретной метрики существуют более эффективные алгоритмы генерации тестовых программ.

2.3. Выводы из анализа текущего состояния

Подводя итоги, заметим, что многие исследователи видят преимущества использования моделей и метрик качества тестирования на их основе для генерации тестовых программ. Основой вопрос, который встает, — это вопрос о выборе адекватных моделей и адекватных метрик. Также заметим, что большое число работ посвящено тестированию на основе конечно-автоматных моделей, описывающих конвейер микропроцессора с потактовой точностью (будем называть такие модели *точными моделями конвейера*). Такие методы, как правило, предназначены для повышения покрытия, достигаемого существующими тестами [36,37] и для генерации тестов, нацеленных на сравнительно небольшое число крайних случаев [11-17], но они не являются

альтернативой *массивному тестированию*[6], обеспечиваемому тестами на основе существующего ПО и случайной генерацией.

Отметим следующие проблемы использования точных моделей конвейера. Есть два способа ее получения: либо модель извлекается автоматически на основе статического анализа кода RTL-описания микропроцессора [2,6,36,37], либо строится вручную [8,21]. Автоматическое извлечение модели является сложной задачей и требует наличия в коде аннотаций разработчика [2] или привлечения эвристик [36,37]. Для сложных микропроцессоров автоматическое извлечение конечно-автоматной модели управляющей логики практически неосуществимо. При построении модели вручную возникает другая проблема — построенную модель сложно отлаживать [10]. Поскольку на основе такой модели предполагается генерация тестов, важно чтобы модель точно описывала тестируемый компонент, так как в противном случае цели тестирования не будут достигнуты.

Существующие подходы к тестированию микропроцессоров с помощью тестовых программ можно представить в виде следующей таблицы.

Тестирование	**Ручное тестирование**	**Вероятностное тестирование**	**Тестирование на основе моделей**
Массивное	Существующее ПО	Случайная генерация	
Немассивное	Ручная разработка	Тестовые шаблоны	На основе автоматов

Таблица 2. Классификация имеющихся подходов к построению тестов.

Таблица показывает, что для массивного тестирования микропроцессоров отсутствуют методы генерации тестовых программ на основе моделей. Общение с разработчиками микропроцессоров показывает, что потребность в подобных методах достаточно высока. Они привносят систематичность в массивное тестирование и позволяют увеличить уровень достигнутого тестового покрытия. В качестве основы таких методов можно использовать *комбинаторное тестирование* и *случайную генерацию на основе тестовых шаблонов* — разработчик тестов определяет тестовое покрытие для отдельных инструкций в виде наборов тестовых ситуаций, а генератор строит различные тестовые шаблоны, комбинируя тестовые ситуации для инструкций, которые могут повлиять друг на друга при выполнении на конвейере (инструкций, которые в тексте программы расположены близко). Именно к такому классу методов относится предлагаемый подход.

[6] *Массивными тестами* называются тесты, которые имеют большой размер и широко охватывают функциональность тестируемой системы. Как правило, массивные тесты генерируются автоматически.

3. Описание предлагаемого подхода

Предлагаемый в работе подход к генерации тестовых программ нацелен на массивное функциональное тестирование микропроцессоров. В этом смысле подход является альтернативой тестированию на основе существующего ПО и тестированию с помощью случайных программ. С другой стороны, подход можно рассматривать как развитие методов генерации на основе тестовых шаблонов, поскольку он оперирует с шаблонами, но они не разрабатываются вручную, а строятся генератором автоматически.

Для генерации используется модель микропроцессора, включающая в себя структурную модель микропроцессора и модель системы команд. Цель генерации задается с помощью критерия тестового покрытия, выделяющего набор тестовых ситуаций для каждой инструкции микропроцессора. В методе реализуется направленный перебор всевозможных сочетаний тестовых ситуаций для последовательностей инструкций ограниченной длины. Как и все комбинаторные методы, он подвержен “комбинаторному взрыву” — размер тестов резко возрастает при увеличении длины тестируемых последовательностей инструкций, числа тестовых ситуаций и других параметров генерации.

Для сокращения размера тестов используются дополнительные эвристики, например, гипотезы об эквивалентности некоторых инструкций между собой. Эквивалентные инструкции одинаковы с точки зрения управляющей логики микропроцессора — они затрагивают одни и те же модули, а их планирование и выполнение на конвейере осуществляется идентичным образом. Например, инструкции сложения и вычитания чисел одного типа можно считать эквивалентными, поскольку они выполняются одинаково. Какие инструкции являются эквивалентными, а какие нет, определяется на основе экспертных оценок.

Идея подхода основана на предположении, согласно которому поведение микропроцессора (на каждом такте) зависит от множества выполняемых инструкций, зависимостей между ними (в том числе порядка их следования в тексте программы), а также от ситуаций, в которых оказываются выполняемые инструкции. Здесь говорится о *множестве* выполняемых инструкций, а не об *одиночных* инструкциях, поскольку большинство современных микропроцессоров являются *конвейерными*, и процессы выполнения инструкций в них могут пересекаться по времени [38]. Для качественного тестирования микропроцессоров с конвейерной архитектурой нужно по-разному “загружать” конвейер, используя в тестовых программах различные последовательности инструкций.

Генератору на вход подаются описания тестируемых инструкций и тестовых ситуаций для них, возможные типы зависимостей между инструкциями, а также параметры, управляющие генерацией, например, длина генерируемых последовательностей инструкций. В общих словах, построение тестовых

программ осуществляется следующим образом. Генерируются всевозможные последовательности инструкций указанной длины. Для каждой последовательности инструкций строятся всевозможные множества зависимостей между ними. Для каждого множества зависимостей комбинируются всевозможные тестовые ситуации.

Несколько слов об используемой терминологии. Обычно под *значением операнда* инструкции понимается номер регистра, если значение параметра передается через регистр, либо само значение, если оно передается непосредственно. Мы, для удобства, под значением операнда понимаем как номер регистра (если значение передается через регистр), так и значение параметра. Таким образом, значением операнда, в общем случае, является пара, состоящая из номера регистра и значения параметра. Под *программой* понимается произвольная конечная последовательность инструкций микропроцессора на языке ассемблера.

3.1. Основные понятия предлагаемого подхода

В данном разделе рассматриваются основные понятия предлагаемого подхода к генерации тестовых программ: понятия *тестового шаблона*, *зависимости между инструкциями*, *тестовой ситуации* и *тестового воздействия*. Прежде чем определять эти понятия, рассмотрим, как устроены тестовые программы. В рамках предлагаемого подхода тестовые программы имеют вид $\Pi = \pi_{start} \cdot \{\langle \pi_i, x_i(D_i, S_i)\rangle\}_{i=1,n} \cdot \pi_{stop}$, где:

- π_{start} — *инициализирующая программа*:
  содержит вспомогательные инструкции, предназначенные для инициализации микропроцессора;
- $\langle \pi_i, x_i(D_i, S_i)\rangle$ — *тестовый случай* ($i = 1, \ldots, n$):
  - π_i — *программа подготовки тестового воздействия*:
    последовательность инструкций, осуществляющая инициализацию операндов тестируемых инструкций и подготовку состояния микропроцессора перед выполнением тестового воздействия;
  - $x_i(D_i, S_i)$ — *тестовое воздействие*:
    тестируемая последовательность инструкций: между инструкциями определены зависимости D_i, а значения операндов инструкций и состояние микропроцессора (содержимое регистров, кэш-памяти и других подсистем) удовлетворяют ограничениям S_i;
- π_{stop} — *завершающая программа*:
  содержит вспомогательные инструкции, предназначенные для завершения работы микропроцессора;
- n — *размер тестовой программы*:
  параметр генерации, указывающий число тестовых воздействий в одной тестовой программе.

Ключевым понятием подхода является понятие *тестового воздействия*. Тестовое воздействие характеризуется *тестовым шаблоном* (тестируемой последовательностью инструкций), *зависимостями между инструкциями* (как операнды разных инструкций связаны между собой) и *тестовыми ситуациями* (ограничениями на значения операндов инструкций и состояние микропроцессора). Целью генерации является перебор всевозможных тестовых шаблонов небольшого размера и построение для каждого из них всевозможных зависимостей и тестовых ситуаций.

3.1.1. Тестовый шаблон

Большинство современных микропроцессоров имеют *конвейерную архитектуру*. В таких микропроцессорах выполнение инструкций разбито на *стадии*, и возможны ситуации, когда некоторая стадия одной инструкции выполняется параллельно с некоторой стадией другой инструкции [38] (см. Рис. 1). Для качественного тестирования конвейерных микропроцессоров недостаточно изолированно протестировать все инструкции, даже если при этом используются разнообразные значения операндов и разнообразные состояния памяти микропроцессора — помимо этого тесты должны создавать различные *состояния конвейера*.

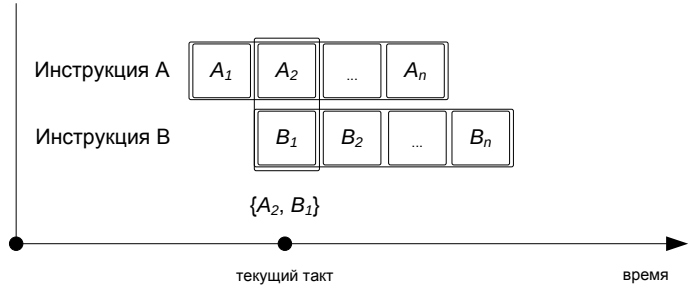

Рис.1. Выполнение двух инструкций на конвейере.

В предлагаемом подходе для создания разнообразных состояний конвейера предназначены так называемые *тестовые шаблоны* (*шаблоны тестовых воздействий*). Тестовым шаблоном называется тестируемая последовательность инструкций без указания конкретных значений операндов. Задача тестового шаблона зафиксировать порядок инструкций в тестовом воздействии. Для одного тестового шаблона обычно генерируются множество тестовых воздействий, которые отличаются друг от друга зависимостями между инструкциями и тестовыми ситуациями.

Ниже приведен пример тестового шаблона, состоящий из двух инструкций: инструкции сложения *ADD* и инструкции вычитания *SUB*:

```
add ... // тестовый шаблон фиксирует порядок инструкций
```



```
sub ... // тестовый шаблон не определяет значений операндов
```

Практика показывает, что большинство ошибок в микропроцессорах обнаруживается на достаточно коротких последовательностях инструкций (обычно для обнаружения ошибки требуется 2–4 инструкции, более длинные последовательности нужны реже), поэтому при генерации тестовых программ можно использовать тестовые шаблоны небольшого размера. Однако даже в этом случае общее число тестовых шаблонов может быть очень велико (число всевозможных троек, составленных из 200 инструкций, равно $200^3 = 8\,000\,000$).

Для сокращения числа тестовых шаблонов используется следующий подход. Близкие с точки зрения управляющей логики инструкции (то есть инструкции, планирование и выполнение которых осуществляется похожим образом) объединяются в *классы эквивалентности*. Тестовые шаблоны, в которых на одинаковых позициях находятся эквивалентные инструкции, считаются *эквивалентными*. Тестовые программы должны содержать все классы эквивалентности тестовых шаблонов; для построения тестового воздействия используется произвольный тестовый шаблон из соответствующего класса эквивалентности.

3.1.2. Зависимости между инструкциями

Использование различных тестовых шаблонов, как правило, не является достаточным условием качественного тестирования микропроцессора. Важным моментом является то, какие *зависимости между инструкциями* использовались в тестах. Микропроцессор может по-разному планировать выполнение инструкций или по-разному осуществлять блокировки в зависимости от того, как связаны между собой инструкции.

В своем подходе мы выделяем два основных типа зависимостей между инструкциями: *зависимости по регистрам* и *зависимости по адресам*. Зависимости по регистрам выражаются в совпадении регистров, использующихся в качестве операндов разных инструкций тестового воздействия. Зависимости по адресам определены для инструкций работы с памятью, таких как *инструкции загрузки и сохранения регистров* (*load*/*store instructions*). Зависимости по адресам являются частным случаем так называемых *зависимостей по содержимому*[7], которые определяются не на основе совпадения или несовпадения используемых регистров, а с помощью ограничений на значения пары входных параметров (зависимого и определяющего).

[7] Зависимости по содержимому имеют смысл, когда значения операндов (или функций от значений операндов) адресуют некоторые разделяемые ресурсы, то есть имеют смысл адреса.

Мы используем две разновидности зависимостей по регистрам: *зависимости типа определение-использование* (*define-use dependencies*), когда выходной регистр одной инструкции является входным регистром следующей за ней инструкции (либо между ними находятся другие инструкции, которые не переопределяют значение этого регистра), и *зависимости типа определение-определение* (*define-define dependencies*), когда выходной регистр одной инструкции также является выходным регистром следующей за ней инструкции (тоже могут быть промежуточные, не затрагивающие этот регистр, инструкции). Возможны и другие типы зависимостей, например, *зависимости типа использование-использование* (*use-use dependencies*), но такие зависимости обычно не влияют на логику работы микропроцессора.

Для иллюстрации зависимостей по регистрам рассмотрим два простых примера. В первом примере показана зависимость типа определение-использование:

```
add r1, r2, r3 // определение регистра r1
sub r4, r1, r5 // использование регистра r1
```

Инструкция *ADD* складывает содержимое регистров *r2* и *r3* и сохраняет результат в регистре *r1*. Следующая за ней инструкция *SUB* использует содержимое регистра *r1* — она вычитает из содержимого регистра *r1* содержимое регистра *r5* и сохраняет результат в регистре *r4*.

В следующем примере проиллюстрирована зависимость типа определение-определение:

```
add r1, r2, r3 // определение регистра r1
sub r1, r4, r5 // переопределение регистра r1
```

В этом примере инструкция *SUB* сохраняет свой результат в том же самом регистре *r1*, что и предшествующая ей инструкция *ADD*.

Зависимости по адресам определяются структурой памяти микропроцессора. Мы выделяем *зависимости по виртуальным адресам* и *зависимости по физическим адресам*. Использование в тестах различных зависимостей по адресам позволяет улучшить качество тестирования подсистемы трансляции адресов (для зависимостей по виртуальным адресам) и подсистемы управления кэш-памятью (для зависимостей по физическим адресам). В качестве примеров зависимостей по адресам можно привести следующие: совпадение виртуальных адресов, используемых различными инструкциями; совпадение физических адресов при несовпадающих виртуальных адресах; попадание в одну и ту же запись буфера трансляции адресов; попадание в одну и ту же строку кэш-памяти.

Проиллюстрируем зависимости по адресам на примере микропроцессоров семейства RM7000 [39]. Рассмотрим зависимость следующего вида — совпадение номера строки в кэше данных L1. Данная зависимость является скорее зависимостью по физическим адресам, хотя ее можно отнести и к зависимостям по виртуальным адресам, поскольку она затрагивает только

младшие 12 бит адреса, которые у виртуального и физического адреса в MIPS64-совместимых микропроцессорах совпадают [5]. Обозначим эту зависимость как *L1RowEqual*.

Физический адрес в микропроцессорах семейства RM7000 состоит из 36 бит и имеет следующую структуру: биты с 0-ого по 2-ой включительно (*byte bits*) определяют позицию байта внутри двойного слова, биты с 3-его по 4-ый (*dword bits*) — позицию двойного слова в строке кэш-памяти, биты с 5-ого по 11-ый (*index bits*) — номер сроки и, наконец, биты с 12-ого по 35-ый (*upper address bits*) — тэг данных, используемый для проверки попадания данных в кэш-память [39].

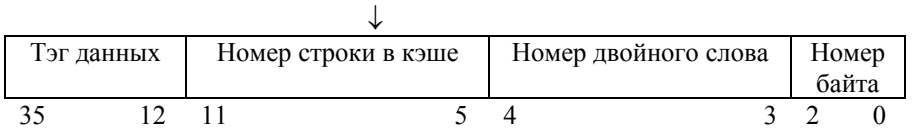

Рис. 2. Формат физического адреса в RM7000.

Таким образом, хранящиеся в памяти данные отображаются в одну строку кэша данных L1 тогда и только тогда, когда биты *index bits* их физических адресов совпадают.

Зависимости между инструкциями описываются набором атрибутов; конкретный набор значений атрибутов фиксирует зависимость рассматриваемого типа. Помимо атрибутов описание зависимости включает в себя следующие компоненты:

- *итератор* — перебирает все допустимые комбинации значений атрибутов зависимости;
- *фильтр* — проверяет множество зависимостей данного типа, относящихся к одному зависимому операнду, на непротиворечивость;
- *конструктор* — конструирует (полностью или частично) значение зависимого операнда по значениям операндов, от которых он зависим через зависимости данного типа;
- *предусловие* — проверяет допустимость использования зависимости данного типа для пары операндов различных инструкций.

Рассмотрим, как устроены указанные выше компоненты на примере зависимости *L1RowEqual*. Зависимость описывается одним атрибутом, который принимает два значения: *true* или *false*. Итератор зависимости перебирает эти значения в некотором порядке. Фильтр проверяет, что в множестве зависимостей существует не более одной зависимости, у которой атрибут имеет значение *true* (в противном случае возникает противоречие или избыточность). Если в множестве зависимостей существует такая, у которой атрибут имеет значение *true*, конструктор присваивает битам *index bits* зависимого операнда значение соответствующих бит адреса, от которого он зависит; если у всех зависимостей атрибут имеет значение *false*, конструктор

генерирует случайное значение, отличное от всех значений *index bits*, используемых в адресах, от которых зависит конструируемый операнд. Предусловие зависимости проверяет, что виртуальные адреса, используемые в операндах зависимых инструкций, могут быть преобразованы в физические — в противном случае, например, когда используются ошибочные адреса, обращения к кэш-памяти (по крайней мере для одной инструкции) не производится, поэтому зависимость не имеет смысла.

Типы зависимостей, которые целесообразно использовать для тестирования, определяются на основе анализа документации по тестируемому микропроцессору. Определение зависимостей по регистрам обычно сводится к разбиению множества регистров на типы: *регистры общего назначения* (*GPR*, *General Purpose Registers*), *системные регистры* (*system registers*), *регистры модуля арифметики с плавающей точкой* (*FPR*, *Floating Point Registers*) — зависимости возможны только между регистрами одного типа. При определении зависимостей по адресам следует обращать внимание на структуру буфера трансляции адресов и кэш-памяти (L1, L2 и других уровней, если они присутствуют).

3.1.3. Тестовые ситуации

Как правило, инструкции микропроцессора имеют несколько *ветвей функциональности*, то есть в зависимости от значений входных операндов и состояния микропроцессора ведут себя по-разному. Для инструкций, вызывающих исключения, примерами ветвей функциональности являются ситуации, в которых возникает то или иное исключение. Обязательным условием качественного тестирования микропроцессора является покрытие всех ветвей функциональности всех реализуемых им инструкций. Далее вместо термина *ветвь функциональности* мы используем более общий термин *тестовая ситуация*.

Под *тестовой ситуацией для инструкции* понимается ограничение на значения входных операндов и состояние микропроцессора перед началом выполнения инструкции. Если рассматривать не отдельную инструкцию, а их последовательность (тестовый шаблон), можно определить *тестовую ситуацию для шаблона* в целом как набор тестовых ситуаций для входящих в него инструкций.

Тестовые ситуации выявляются на основе анализа описания архитектуры микропроцессора и его системы команд. Особое внимание при этом уделяется ветвям функциональности, в частности, условиям на исключения. Рассмотрим пример описания инструкции *ADD* из системы команд MIPS64 [5]:


```
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
```

Предикат *NotWordValue*(*GPR*[*rs*]) ***or*** *NotWordValue*(*GPR*[*rt*]) является отрицанием *предусловия инструкции*. В рассматриваемом примере оба входных регистра должны содержать 32-х битные слова (значения битов с 32-ого по 63-ый включительно совпадают со значением 31-ого бита). Предикат $temp_{32} \neq temp_{31}$ определяет ветвь функциональности, соответствующую исключению *IntegerOverflow*; другая ветвь соответствует нормальному выполнению инструкции.

Также, как и зависимости между инструкциями, тестовые ситуации описываются набором атрибутов, а конкретный набор значений атрибутов фиксирует тестовую ситуацию рассматриваемого типа. Помимо атрибутов описание тестовой ситуации включает в себя следующие компоненты:

- *итератор* — перебирает все допустимые комбинации значений атрибутов тестовой ситуации;
- *фильтр* — проверяет тестовую ситуацию на непротиворечивость тестовым ситуациям предшествующих инструкций с учетом существующих зависимостей между инструкциями[8];
- *конструктор* — конструирует значения операндов инструкции с учетом зависимостей от предшествующих инструкций;
- *компонент подготовки* — строит *программу подготовки тестовой ситуации* для инструкции, то есть программу, которая инициализирует операнды инструкции и подготавливает нужным образом состояние микропроцессора.

Рассмотрим пример. Семейство тестовых ситуаций для инструкции *ADD* (на некотором уровне абстракции) описывается одним атрибутом *IntegerOverflow*, который принимает два значения: *true* или *false*. Если значение атрибута равно *true*, это означает, что содержимое входных регистров инструкции должно быть таким, чтобы при их сложении возникало переполнение; в противном случае переполнения быть не должно.

[8] На практике можно использовать более “слабые” условия фильтрации, например, фильтр может пропускать тестовые ситуации, которые протииворечат зависимостям, если при этом не нарушаются предусловия инструкций.

Фильтр тестовой ситуации проверяет, связаны ли входные регистры инструкции зависимостью типа определение-использование с выходными регистрами предшествующих инструкций: если да, и зависимыми являются оба регистра, проверяется, согласуется ли их содержимое со значением атрибута *IntegerOverflow*; если нет, такая ситуация отбрасывается; если же зависимым является только один регистр, ситуация отбрасывается, только если значение *IntegerOverflow* равно *true*, а значение зависимого регистра — 0.

Если обозначить через x и y значения операндов, а через *temp* результат сложения x и y, расширенных знаковым битом, то есть $(x_{31} \| x_{31..0}) + (y_{31} \| y_{31..0})$, то условие на переполнение примет вид $temp_{31} \neq temp_{32}$. Конструктор тестовой ситуации генерирует случайные значения независимых операндов, которые удовлетворяют этому ограничению, если атрибут *IntegerOverflow* имеет значение *true*, и не удовлетворяют ему, если атрибут *IntegerOverflow* имеет значение *false*.

Компонент подготовки тестовой ситуации для каждого независимого входного регистра добавляет в программу подготовки последовательность инструкций, осуществляющую загрузку в него сконструированного значения.

3.1.4. Тестовые воздействия

Как было отмечено в начале раздела, основным понятием в предлагаемом подходе является понятие *тестового воздействия*. Тестовым воздействием называется последовательность инструкций с заданными значениями операндов, выполнение которой начинается в заданном состоянии микропроцессора. Будем называть тестовые воздействия *эквивалентными*, если у них совпадают тестовые шаблоны, зависимости между инструкциями и тестовые ситуации. Под *обобщенными тестовыми воздействиями* будем понимать классы эквивалентности тестовых воздействий.

Целью генерации является реализация в тестовых программах всех обобщенных тестовых воздействий. Для достижения этой цели решаются две задачи: *перебор обобщенных тестовых воздействий* и *подготовка* (*конструирование*) *тестовых воздействий*. При переборе осуществляется последовательное построение всевозможных тестовых шаблонов, зависимостей между инструкциями и тестовых ситуаций. Подготовка тестового воздействия состоит в построении программы, которая инициализирует операнды инструкций тестового воздействия и подготавливает состояние микропроцессора таким образом, чтобы они соответствовали тестовой ситуации. По сути, на каждом шаге перебора формируется набор ограничений на тестовое воздействие, которые разрешаются на этапе подготовки тестового воздействия.

Для перебора обобщенных тестовых воздействий используются итераторы зависимостей и итераторы тестовых ситуаций, для подготовки тестовых воздействий — конструкторы зависимостей, а также конструкторы и компоненты подготовки тестовых ситуаций. Более подробно вопросы,

относящиеся к генерации, рассмотрены в разделе "*Метод генерации и устройство генератора*".

3.2. Модель микропроцессора

Для генерации тестовых программ используется *модель микропроцессора*. Модель имеет два аспекта: *статический* и *динамический*. Первый из них состоит в том, что модель является статическим описанием архитектуры микропроцессора и его системы команд. Описание, будучи *формальным*, позволяет генератору автоматически анализировать модель и использовать результаты анализа при построении тестовых программ. Второй аспект заключается в том, что в процессе генерации (в динамике) модель отражает текущее состояние микропроцессора. Модельное состояние позволяет контролировать предусловия инструкций и корректно подготавливать тестовые ситуации.

Заметим также, что модельное состояние можно использовать для построения так называемых *самопроверяющих тестов*, то есть тестов, в которых после каждого тестового воздействия располагается *тестовый оракул* — программа, осуществляющая проверку правильности выполнения тестового воздействия микропроцессором.

Модель микропроцессора включает в себя *модель архитектуры* (*структурную модель*), *модель типов данных*, *модель системы команд* и *модель конвейера*. Рассмотрим каждую из перечисленных составляющих.

3.2.1. Модель архитектуры

Модель архитектуры описывает основные подсистемы микропроцессора: регистры, буфер трансляции адресов, кэш-память и другие. Основное назначение модели архитектуры заключается в описании структуры состояния микропроцессора — это фундамент, на основе которого осуществляется описание системы команд, зависимостей между инструкциями и тестовых ситуаций.

Важно отметить, что уровень абстракции, на котором моделируется архитектура микропроцессора, может варьироваться в зависимости от целей тестирования. Например, при моделировании кэш-памяти можно не включать в модель хранящиеся в кэше данные, а моделировать только тэги данных — этого достаточно для того, чтобы проверять и подготавливать тестовые ситуации такие как попадание или промах в кэш-память.

Важной составной частью модели архитектуры является *модель регистров*. Модель регистров вводит на множестве регистров микропроцессора систему типов: *регистры общего назначения*, *системные регистры*, *регистры модуля арифметики с плавающей точкой* и другие. Особая роль этой модели связана с тем, что регистры являются основным способом передачи значений параметров инструкциям и получения результата их выполнения. При описании инструкций указываются типы их входных и выходных регистров;

эта информация используется генератором для построения зависимостей по регистрам.

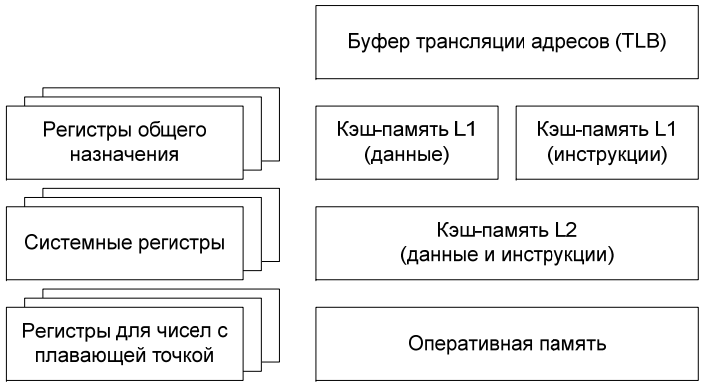

Рис. 3. Основные моделируемые подсистемы микропроцессора.

3.2.2. Модель типов данных

Разные инструкции микропроцессора определены над данными разных типов: одни — над 32-х битными словами, другие — над двойными словами; одни инструкции требуют, чтобы значение операнда являлось числом с плавающей точкой, другие — чтобы оно было виртуальным адресом и так далее. *Модель типов данных* описывает множество типов данных микропроцессора и отражает совместимость различных типов данных между собой.

Как правило, микропроцессоры имеют достаточно простую систему типов: байт, полуслово, слово, двойное слово, числа с плавающей точкой одинарной и двойной точности, виртуальный адрес и некоторые другие.

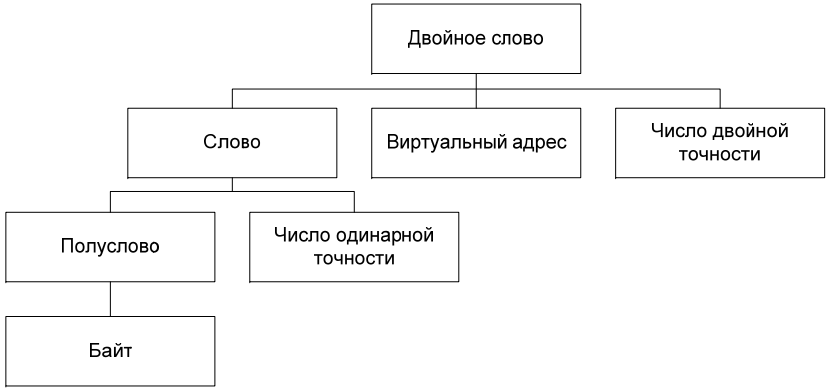

Рис.4. Диаграмма основных типов данных микропроцессора.

При описании инструкций указываются типы данных их операндов. Эта информация используется генератором при построении зависимостей между

инструкциями. В частности, зависимость по регистрам типа определение-использование возможна только в том случае, если типы выходного и входного регистров совместимы. Например, регистр, содержащий результат 32-х битной операции *ADD*, можно использовать в качестве операнда 64-х битной операции *DSUB*, но не наоборот.

3.2.3. Модель системы команд

Модель системы команд содержит основную информацию, с которой работает генератор тестовых программ: *интерфейс*, *семантику* и *ассемблерный формат* инструкций микропроцессора. Описание интерфейса используется генератором при построении зависимостей между инструкциями, на основе описания семантики генератор обновляет модельное состояние микропроцессора; ассемблерный формат используется при отображении внутреннего представления тестовых программ в файлы на языке ассемблера.

Описание отдельной инструкции микропроцессора включает в себя следующие компоненты:

- *интерфейс инструкции* — описывает операнды инструкции: для каждого операнда указывается его имя, способ передачи значения (непосредственно или через регистр, в последнем случае также указывается тип регистра), тип данных и информация о том, является операнд входным или выходным;
- *предусловие инструкции* — определяет ситуации, в которых определено поведение микропроцессора при выполнении данной инструкции. Автоматически генерируемой частью предусловия является проверка совместимости значений входных операндов соответствующим типам данным;
- *функция вычисления значений выходных операндов* — вычисляет значения выходных операндов инструкции по значениям входных операндов, а также исключения, возникающие при выполнении инструкции;
- *функция обновления состояния микропроцессора* — обновляет модельное состояние микропроцессора на основе значений входных операндов инструкции и текущего состояния микропроцессора;
- *отображение в ассемблерный код* — определяет формат записи инструкции и ее операндов на языке ассемблера.

Отметим, что семантика инструкции описывается двумя функциями: функцией вычисления значений выходных операндов и функцией обновления состояния микропроцессора. Первая функция используется, когда при конструировании тестовой ситуации требуется определить значения

зависимых операндов соответствующей инструкции. Эти значения нельзя получить из модельного состояния микропроцессора, поскольку конструирование тестовой ситуации осуществляется до обновления состояния предшествующими инструкциями тестового воздействия (они будут выполнены только после выполнения программ подготовки тестовых ситуаций всех инструкций тестового воздействия). Вторая функция, как видно из названия, используется для обновления состояния микропроцессора.

Модель системы команд микропроцессора разрабатывается на основе анализа документации. Современные руководства содержат детальные описания инструкций, которые могут быть легко формализованы. Для примера, рассмотрим фрагмент описания инструкции *ADD* из системы команд MIPS64 [5].

Format: `ADD rd, rs, rt`

Description: `rd ← rs + rt`

Operation:

```
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
```

Из этого описания видно, что у инструкции три операнда: *rd*, *rs* и *rt*; значения всех операндов передаются через регистры общего назначения (GPRs, General Purpose Registers); тип данных операндов — слово; операнд *rd* является выходным, операнды *rs* и *rt* — входными. Описание также определяет предусловие инструкции, ее семантику и отображение в ассемблерный код.

3.2.4. Модель конвейера

Модель конвейера является необязательной частью модели микропроцессора и предназначена для более детального моделирования процесса выполнения инструкций. Использование модели конвейера позволяет описывать и создавать тестовые ситуации, связанные с параллельным выполнением инструкций на конвейере, например, ситуации, в которых между инструкциями возникают конфликты использования ресурсов микропроцессора. Заметим, что модель не обязательно должна описывать конвейер микропроцессора с потактовой точностью — уровень детализации описания зависит от целей тестирования.

Мы не будем подробно описывать модель конвейера, поскольку она используется только в особых случаях, и ее описание может затруднить общее понимание подхода. В общих словах, модель конвейера представляет собой автомат, который меняет свое состояние в зависимости от поданной на выполнение инструкции. Состояние такого автомата может использоваться для исключения конфликтов использования ресурсов, ослеживания *слотов задержки* (*delay slots*), а также для других целей.

3.3. Метод генерации и устройство генератора

В данном разделе описывается метод генерации и устройство генератора тестовых программ, разработанного в соответствии с описанным подходом. На вход генератору подается модель микропроцессора, описания зависимостей между инструкциями и описания тестовых ситуаций. Из числа инструкций, описанных в модели системы команд, указываются так называемые *тестируемые инструкции*, используемые для составления тестовых воздействий. Также на вход генератору подаются параметры, управляющие построением тестовых программ.

3.3.1. Метод генерации

В общих словах, генерация тестовых программ осуществляется следующим образом. В цикле перебираются тестовые шаблоны. Для каждого шаблона перебираются и конструируются множества зависимостей по регистрам. Для каждого множества зависимостей по регистрам комбинируются тестовые ситуации для инструкций тестового воздействия. Для каждого набора тестовых ситуаций перебираются множества зависимостей по адресам. Последовательно для каждой инструкции тестового воздействия сначала конструируются зависимости по адресам от предыдущих инструкций (если они есть), затем — тестовая ситуация, после чего для тестовой ситуации строится программа подготовки. Из программ подготовки тестовых ситуаций строится программа подготовки тестового воздействия, которая в сгенерированном коде помещается непосредственно перед тестовым воздействием. Процесс генерации продолжается до тех пор, пока не будут перебраны все обобщенные тестовые воздействия.

Генерация осуществляется согласно следующей схеме:

- получить очередной тестовый шаблон
  - *получить очередное множество зависимостей по регистрам*
  - *сконструировать зависимости по регистрам*
  - *получить очередную тестовую ситуацию для шаблона*
    - получить очередное множество зависимостей по адресам
    - для каждой инструкции тестового воздействия:

- проверить предусловие:
  - если предусловие нарушено, перейти к очередной тестовой ситуации для шаблона
- проверить наличие зависимостей по адресам:
  - если зависимости существуют, сконструировать зависимости
- *сконструировать тестовую ситуацию для инструкции*
- проверить тестовую ситуацию для инструкции на противоречивость:
  - *если тестовая ситуация противоречива, перейти к следующей тестовой ситуации для шаблона*
- *получить программу подготовки тестовой ситуации для инструкции*

- построить программу подготовки тестового воздействия
- выполнить программу подготовки тестового воздействия на модели
- выполнить тестовое воздействие на модели

Рассмотрим подробнее, как строится программа подготовки тестового воздействия. На первый взгляд, задача является тривиальной — достаточно последовательно расположить программы подготовки тестовых ситуаций для инструкций тестового воздействия: сначала — программу подготовки тестовой ситуации для первой инструкции, затем для второй и так далее. В большинстве случаев такой способ работает, но не всегда. Бывают ситуации, когда программа подготовки тестовой ситуации влияет на предшествующие инструкции. Мы столкнулись с этой проблемой при тестировании инструкций загрузки и сохранения регистров. Для таких инструкций программа подготовки состоит из двух частей: подготовка буфера трансляции адресов и подготовка кэш-памяти. Программа подготовки буфера трансляции адресов изменяет состояние кэш-памяти, поэтому при построении программы подготовки тестового воздействия целесообразно сначала осуществлять подготовку буфера трансляции адресов для всех инструкций тестового воздействия, а затем — подготовку кэш-памяти (подготовка кэш-памяти не затрагивают буфера трансляции адресов, поскольку для этого используются адреса из неотображаемых сегментов виртуальной памяти).

В общем случае, используемый в генераторе метод построения программ подготовки тестовых воздействий следующий. Разработчик тестов разбивает программы подготовки тестовых ситуаций для тестируемых инструкций на несколько фрагментов. Каждый фрагмент отвечает за инициализацию

определенной подсистемы микропроцессора. Разработчик упорядочивает фрагменты таким образом, чтобы каждый следующий фрагмент не влияет 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.

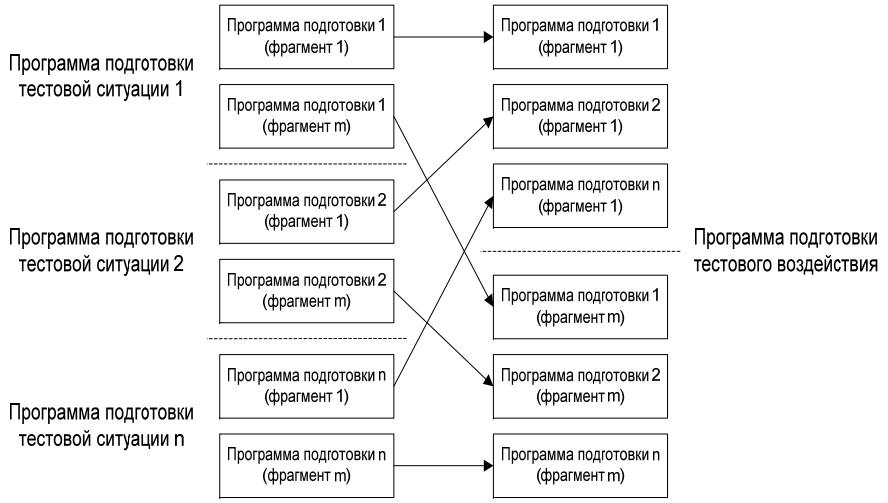

Рис.5. Построение программы подготовки тестового воздействия.

Отметим еще один важный момент. Программа подготовки тестовой ситуации для инструкции тестового воздействия зависит от состояния микропроцессора. Для того чтобы построить ее корректно, необходимо выполнить функции обновления состояния микропроцессора для всех предшествующих инструкций. Среди них есть инструкции, относящиеся к программам подготовки тестовых ситуаций для последующих инструкций тестового воздействия. В свою очередь, для корректного построения этих программ необходимо выполнить рассматриваемую инструкцию, но это возможно лишь после полного построения и выполнения программы подготовки тестового воздействия. Для того чтобы выйти из этого замкнутого круга, используются зависимости. Очевидно, что программу подготовки тестовой ситуации для первой инструкции можно построить корректно. Если инструкция не зависит по адресам от предшествующих (использует другую запись буфера трансляции адресов, другую строку кэш-памяти), при построении программы подготовки тестовой ситуации для нее можно использовать состояние микропроцессора до выполнения тестового воздействия. Если же зависимость по адресам существует, предполагается, что зависимую часть ситуации можно

подготовить, используя состояние микропроцессора до выполнения тестового воздействия и информацию о зависимости.

3.3.2. Архитектура генератора

В генераторе можно выделить следующие основные компоненты: *итератор тестовых шаблонов*, предназначенный для перебора шаблонов тестовых воздействий; *итератор зависимостей*, отвечающий за перебор зависимостей между инструкциями, и *итератор тестовых ситуаций*, задача которого состоит в комбинировании тестовых ситуаций для инструкций тестового воздействия. Итератор тестовых зависимостей в свою очередь состоит из *итератора зависимостей по регистрам* и *итератора зависимостей по адресам*. Помимо основных компонентов, которые формируют *ядро генератора*, генератор содержит несколько вспомогательных *библиотек*.

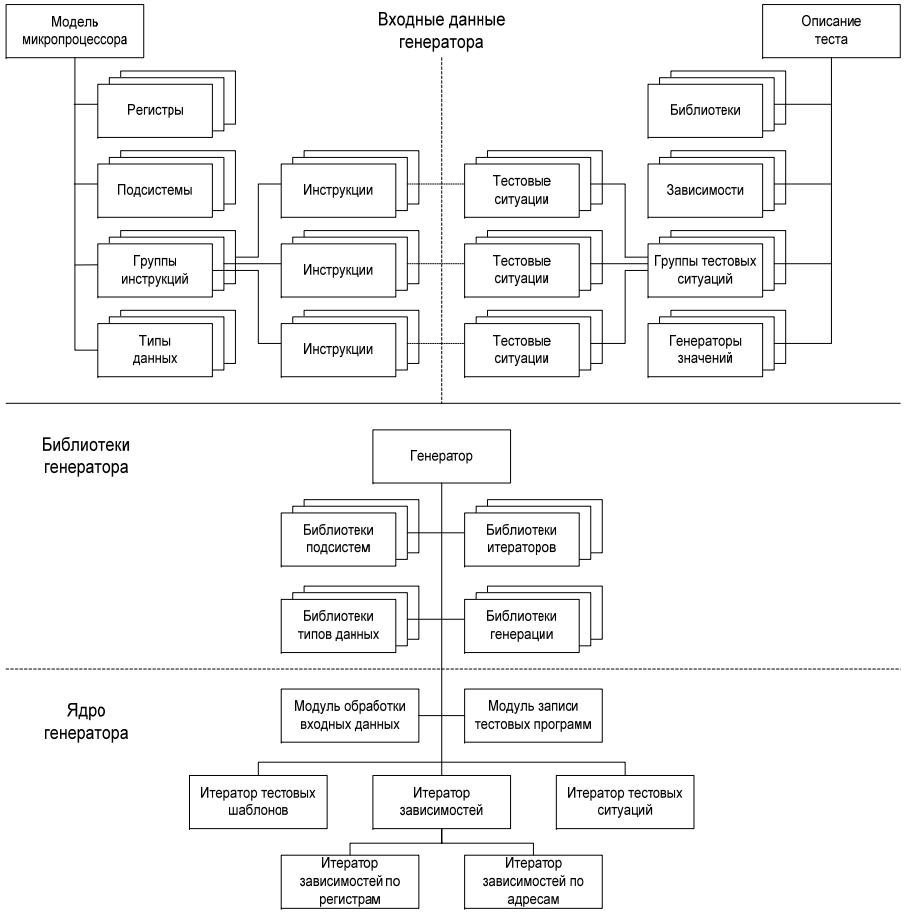

Рис.6. Структура входных данных генератора и его архитектура.

Среди библиотек генератора можно отдельно выделить *библиотеки подсистем*, *библиотеки типов данных*, *библиотеки итераторов* и *библиотеки генерации*. Библиотеки подсистем предназначены для упрощения моделирования основных подсистем микропроцессора, таких как регистры, буфер трансляции адресов, кэш-память. К библиотекам подсистем также относятся *библиотеки арифметики*, которые реализуют основные арифметические операции, включая операции над числами с плавающей точкой. Библиотеки типов данных моделируют основные типы данных, используемые в микропроцессорах. Библиотеки итераторов содержат реализацию итераторов, часто используемых при разработке тестов. Библиотеки генерации отвечают за конструирование значений разных типов данных на основе ограничений.

Заметим, что описания инструкций, которые подаются на вход генератору, могут объединяться в группы. Группы могут иметь иерархическую структуру. Группировка инструкций, как правило, производится на основе подсистем микропроцессора, задействованных для выполнения инструкций. Разбиение инструкций на группы дает генератору дополнительную информацию, которую он может использовать при построении тестовых программ.

3.3.3. Параметры управления генерацией

Как было сказано ранее, на вход генератору помимо модели микропроцессора, описаний зависимостей и описаний тестовых ситуаций подаются параметры, управляющие построением тестовых программ, к числу которых относятся:

- *классы эквивалентности инструкций* — предназначены для сокращения числа тестовых воздействий. Предполагается, что эквивалентные инструкции одинаковы с точки зрения управляющей логики микропроцессора, и выбор той или иной инструкции из одного класса эквивалентности не влияет качество тестирования. Отметим, что эквивалентные инструкции должны иметь одинаковые наборы тестовых ситуаций;
- *размер тестового шаблона* — указывает число инструкций в одном тестовом воздействии. На практике используются тестовые воздействия небольшой длины (из 2–4 инструкций, более длинные тестовые воздействия используются реже);
- *число тестовых воздействий в одной программе* — управляет разбиением тестовых воздействий по тестовым программам. Как только в процессе генерации накапливается требуемое число тестовых воздействий, они записываются в файл.
- *шаблон имени файла* — задает способ именования файлов с тестовыми программами.

3.3.4. Примеры

В данном разделе в качестве иллюстрации ко всему вышесказанному приведены четыре примера. Первые три из них являются фрагментами описаний, которые подаются на вход генератору; четвертый — это часть сгенерированной тестовой программы. Примеры никак не связаны между собой — их можно рассматривать независимо друг от друга.

Сразу следует сказать, что в существующей версии генератора разработка входных данных (модели микропроцессора, зависимостей между инструкциями и тестовых ситуаций) осуществляется на языке программирования Java. В дальнейшем мы планируем использовать специализированные языки описания микропроцессоров и тестов — в этом случае классы, подобные тем, которые мы используем сейчас, будут использоваться для внутреннего представления данных генератора.

Ниже приведен фрагмент описания инструкции сложения *ADD*.

```
// класс, описывающий инструкцию ADD
public class ADDInstruction extends Instruction
{
  public ADDInstruction()
  {
    super("add");

   // операнды инструкции ADD
   addOperand(new Operand(this, "rd", Operand.OUTPUT,
      OperandType.GPR_REGISTER, ContentType.WORD));
   addOperand(new Operand(this, "rs", Operand.INPUT,
      OperandType.GPR_REGISTER, ContentType.WORD));
   addOperand(new Operand(this, "rt", Operand.INPUT,
      OperandType.GPR_REGISTER, ContentType.WORD));
  }

  // функция вычисления значений выходных операндов
  public void calculate(Processor processor)
  {
   Operand rdOperand = getOperand("rd");
   Operand rsOperand = getOperand("rs");
   Operand rtOperand = getOperand("rt");

   int rsValue = rsOperand.getIntegerValue();
   int rtValue = rtOperand.getIntegerValue();

   if(!WordType.isWord(rsValue)||!WordType.isWord(rtValue))
   {
```



```
     setException(new UnpredictableBehavior());
     return;
   }

   int result = add(rsValue, rtValue);

   if(arithmeticOverflow32(result))
   {
     setException(new IntegerOverflow());
     return;
   }

   rdOperand.setIntegerValue(result);
}

// функция обновления состояния микропроцессора
public void execute(Processor processor)
{
  super.execute(processor);

  Operand rdOperand = getOperand("rd");
  Operand rsOperand = getOperand("rs");
  Operand rtOperand = getOperand("rt");

  Register rsRegister = rsOperand.getRegister();
  Register rtRegister = rtOperand.getRegister();
  Register rdRegister = rdOperand.getRegister();

  int result = add(rsRegister.getIntegerValue(),
    rtRegister.getIntegerValue());

  if(arithmeticOverflow32(result))
  { setException(new IntegerOverflow()); }
  else
  { rdRegister.setIntegerValue(result); }
}

// отображение инструкции в ассемблерный код
public String toString()
{
  Operand rd = getOperand("rd");
  Operand rs = getOperand("rs");
  Operand rt = getOperand("rt");

  return "add " + rd.getRegister() + ", " +
```

```
        rs.getRegister() + ", " + rt.getRegister();
    }

  ...
}
```

Следующий пример является описанием зависимости по физическому адресу.

```
//класс,описывающий зависимость по физическому адресу
public class PADependency extends Dependency
{
  // итераторы зависимости
  protected BooleanIterator l1TagEqual =
  new BooleanIterator();
  protected BooleanIterator l1RowEqual =
  new BooleanIterator();

  // предусловие зависимости
  public boolean precondition()
  {
    MemorySituation situationDeps =
    (MemorySituation)getDepsSituation();
    MemorySituation situationFrom =
    (MemorySituation)getFromSituation();

    return situationDeps.canTranslateAddress()
      && situationFrom.canTranslateAddress();
  }

  // конструктор зависимости
  public void construct(Situation ts,
  Dependencies deps,  GeneratorContext context)
  { ... }

  // фильтр зависимости
  public boolean consistent(Operand operand)
  { ... }

  // описание зависимости
  public String toString()
  {
    return "l1TagEqual=" + l1TagEqual.booleanValue()
      + ", l1RowEqual=" + l1RowEqual.booleanValue();
  }
  ...
}
```


Приведенный ниже код описывает тестовую ситуации для инструкции деления *DIV*.

```
// класс, описывающий тестовую ситуацию для
инструкции DIV
public class DIVSituation extends Situation
{
  // итератор тестовой ситуации
  protected BooleanIterator divisinByZero =
  new BooleanIterator();

  // конструктор тестовой ситуации
  public    void    construct(Processor    processor,
  GeneratorContext context)
  {
    Operand rsOperand = getOperand("rs");
    Operand rtOperand = getOperand("rt");


if(!context.isDefinedRegister(rsOperand.getRegister()))
   {rsOperand.setIntegerValue(Utils.random.nextInt());}


if(!context.isDefinedRegister(rtOperand.getRegister()))
    {
      if(divisionByZero.booleanValue())
      { rtOperand.setIntegerValue(0); }
      else
      {
        int rtValue;

        do { rtValue = Utils.random.nextInt(); }
        while(rtValue == 0)

        rtOperand.setIntegerValue(rtValue);
      }
    }
  }

  // фильтр тестовой ситуации
  public boolean isConsistent()
  {
    return true;
  }
```

```
  // функция подготовки тестовой ситуации
  public Program prepare(Processor processor,
  GeneratorContext context)
  {
    Program program = new Program();

    Operand rsOperand = getOperand("rs");
    Operand rtOperand = getOperand("rt");


if(!context.isDefinedRegister(rsOperand.getRegister()))
    {
      program.append(new
      LoadImmediate32Program(processor,
        rsOperand.getRegister(),
        rsOperand.getIntegerValue()));

      context.useRegister(rsOperand.getRegister());
    }


if(!context.isDefinedRegister(rtOperand.getRegister()))
    {
      program.append(
      new LoadImmediate32Program(processor,
        rtOperand.getRegister(),
        rtOperand.getIntegerValue()));

      context.useRegister(rtOperand.getRegister());
    }

    return program;
  }

  // описание тестовой ситуации
  public String toString()
  {
    return
    "DivisionByZero="+divisionByZero.booleanValue();
  }

    ...
}
```


Последний пример — это часть сгенерированной тестовой программы, состоящая из одного тестового воздействия, предваренного программой подготовки.

Несколько слов об используемых в примере инструкциях. Инструкция *LUI* сдвигает 16 битное значение (второй операнд) на 16 бит слева, расширяет полученное 32-х битное значение знаковым битом и записывает результат в регистр, указанный в качестве первого операнда. Инструкция *ORI* осуществляет побитовое ИЛИ значения регистра, указанного в качестве второго операнда, с 16 битным значением (третий операнд), результат записывается в регистр, указанный в качестве первого операнда; *SUB*, *ADD* и *DIV* — это инструкции вычитания, сложения и деления соответственно.

```
...

// Подготовка sub[0]: IntegerOverflow=true
// s5[rs]=0xfffffffc1c998db, v0[rt]=0x7def4297
lui s5, 0xc1c9
ori s5, s5, 0x98db
lui v0, 0x7def
ori v0, v0, 0x4297

// Подготовка add[1]: IntegerOverflow=false
// a0[rs]=0x1d922e27, a1[rt]=0x32bd66d5
lui a0, 0x1d92
ori a0, a0, 0x2e27
lui s3, 0x32bd
ori s3, s3, 0x66d5

// Подготовка div[2]: DivisionByZero=true
// a2[rs]=0x48f, a1[rt]=0x0
lui a2, 0x0
ori a2, a2, 0x48f
lui a1, 0x0

// Зависимости: div[2].rt[1]-sub[0].rd[0]

// Тестовое воздействие: 2008
sub a1, s5, v0 // IntegerOverflow=true
add t7, a0, s3 // IntegerOverflow=false
div a2, a1     // DivisionByZero=true

...
```

4. Опыт практического применения подхода

Описанный подход был использован в двух проектах по тестированию разных MIPS64-совместимых микропроцессоров. В первом проекте тестировалась только *подсистема управления памятью* (*MMU*, *Memory Management Unit*). Для этого в тестовых воздействиях использовались инструкции загрузки и сохранения регистров. Во втором проекте проводилось *системное тестирование* — для построения тестовых воздействий использовались все инструкции микропроцессора.

Мы использовали традиционную процедуру проверки правильности поведения RTL-модели микропроцессора — одни и те же программы выполнялись на тестируемой модели и эталонном симуляторе; результаты выполнения программ сравнивались на соответствие. Результатом выполнения программы является *трасса* — последовательность, содержащая основные события, возникшие при выполнении инструкций программы (результаты выполнения, исключения и другие).

4.1. Тестирование подсистемы управления памятью

Память современных компьютеров представляет достаточно сложную иерархию запоминающих устройств различных объемов, стоимости и времени доступа [38]. Помимо регистров и оперативной памяти в микропроцессорах имеется, по крайней мере, одноуровневая, а чаще двухуровневая кэш-память. Для ускорения преобразования виртуальных адресов в физические в микропроцессорах используются буферы трансляции адресов. Логически связанный набор модулей микропроцессора, отвечающих за организацию работы с памятью, называется *подсистемой управления памятью*.

Мы применили описанный подход для тестирования подсистемы управления памятью MIPS64-совместимого микропроцессора. В качестве тестовых воздействий использовались всевозможные пары, составленные из четырех инструкций: *LB* (загрузка байта), *LD* (загрузка двойного слова), *SB* (сохранение байта) и *SD* (сохранение двойного слова).

Тестовые ситуации для инструкций описывались следующими атрибутами:

- *isMapped* — отображаемый/неотображаемый сегмент виртуальной памяти;
- *isCached* — кэшируемый/некэшируемый сегмент виртуальной памяти;
- *jtlbHit* — попадание/промах в TLB;
- *DVG* — управляющие биты секции TLB;
- *mtlbHit* — попадание/промах в DTLB[9];
- *cachePolicy* — политика кэширования;
- *l1Hit* — попадание/промах в кэш-память L1;

[9] DTLB (Data TLB) — это небольшой буфер, который кэширует обращения к TLB при трансляции адресов данных.

- *l2Hit* — попадание/промах в кэш-память L2.

При генерации тестовых программ использовались следующие зависимости по адресам:

- *vaEqual* — совпадение/несовпадение виртуальных адресов;
- *tlbEqual* — совпадение/несовпадение записи TLB;
- *pageEqual* — совпадение/несовпадение секции в записи TLB[10];
- *paEqual* — совпадение/несовпадение физических адресов;
- *l1RowEqual* — совпадение/несовпадение номера строки в кэш-памяти L1;
- *l2RowEqual* — совпадение/несовпадение номера строки в кэш-памяти L2;
- *mtlbReplace* — совпадение/несовпадение записи TLB, используемой второй инструкцией, с записью, 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 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 первой инструкцией.

В результате тестирования было найдено несколько критических ошибок в подсистеме управления кэш-памятью, которые не были обнаружены с помощью тестовых программ, сгенерированных случайно.

4.2. Системное тестирование микропроцессора

Более масштабным применением описанного подхода является системное тестирование другого MIPS64-совместимого микропроцессора. В качестве тестовых воздействий использовались тройки инструкций, составленные из всех 221 инструкции микропроцессора[11].

[10] В MIPS64-совместимых микропроцессорах запись TLB состоит из двух секций: одна для страницы (VPN, Virtual Page Number) с четным номером, вторая для страницы с нечетным номером. Запись адресуется номером страницы без младшего бита (VPN2), младший бит конкретизирует используемую секцию.

[11] Инструкции, которые отличаются 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.

Инструкции были условно разбиты на 13 групп:

- *целочисленная арифметика* (33 инструкции);
- *логические операции* (8 инструкций);
- *перемещение* (8 инструкций);
- *сдвиги* (15 инструкций);
- *переходы* (20 инструкций);
- *пустые операции* (2 инструкции);
- *загрузка/сохранение* (26 инструкций);
- *исключения* (14 инструкций);
- *сопроцессор CP0* (13 инструкций);
- *арифметика с плавающей точкой* (24 инструкции);
- *перемещение в FPU* (Floating Point Unit, модуль арифметики с плавающей точкой) (26 инструкций);
- *преобразование типов* (26 инструкций);
- *ветвление в FPU* (6 инструкций).

Поскольку число инструкций достаточно велико, для сокращения общего размера тестовых программ мы использовали дополнительную эвристику — тестовые воздействия включали инструкции не более чем из двух различных групп. К моменту написания статьи было выполнено около 70% тестов. В результате было найдено 5 ошибок в RTL-модели микропроцессора и 5 ошибок в эталонном симуляторе. Ниже приводятся некоторые из найденных ошибок.

Рассмотрим пример ошибки, для обнаружения которой важна структура тестового шаблона[12]. Это ошибка была обнаружена в RTL-модели микропроцессора.

```
div  r1, r2
mthi r3
mfhi r4
```

Первая инструкция делит содержимое регистра *r1* на содержимое регистра *r2* и записывает полученный результат (частное и остаток) в регистры *lo* и *hi*. Следующая за делением инструкция сохраняет содержимое регистра *r3* в регистре *hi*. Третья инструкция читает значение регистра *hi* и записывает его значение в регистр *r4*. Ошибка заключается в том, что последняя инструкция вместо значения, записанного второй инструкцией, считывает из регистра *hi* результат деления.

Чтобы обнаружить следующую ошибку, важен как тестовый шаблон, так и тестовая ситуация. Эта ошибка, также как и предыдущая, была обнаружена в RTL-модели микропроцессора.

[12] В данном примере из вида тестового шаблона вытекают некоторые зависимости по регистрам.


```
ctc1  r1, $25
movt  r2, r3, $fcc0
cvt.s $f1, $f2 // Inexact=true
```

Первая инструкция заносит данные в управляющий регистр FPU. Вторая инструкция делает пересылку в зависимости от битов условий (condition codes), содержащихся в FPU. Третья инструкция совершает операцию над числами с плавающей точкой, которая вызывает исключение потери точности *Inexact*. Ошибка возникает при разрешении одновременного выполнения двух инструкций (целочисленной инструкции и инструкции над числами с плавающей точкой), а также разрешении обработки исключения *Inexact* (установлен разряд *Enables.Inexact* в регистре *FCSR*) и проявляется как “зависание” микропроцессора.

Последний пример иллюстрирует ошибку, для обнаружения которой важны тестовый шаблон, зависимости между инструкциями и тестовые ситуации. Это ошибка была обнаружена в симуляторе микропроцессора:

```
add r1, r2, r3 // IntegerOverflow=true
sub r4, r1, r5
```

Первая инструкция складывает содержимое регистров *r2* и *r3* и помещает результат в регистр *r1*. Содержимое регистров *r2* и *r3* должно быть таким, чтобы при их сложении возникало исключение переполнения *IntegerOverflow*. В этом случае, согласно описанию системы команд MIPS64, значение регистра *r1* не должно измениться, поэтому следующая за сложением инструкция вычитания использует значение *r1*, которое было загружено в регистр программой подготовки. Ошибка заключается в том, что симулятор при возникновении исключения записывает в регистр *r1* “мусор”, который не является значением 32-х битного слова. При использовании этого значения в инструкции вычитания симулятор останавливается из-за внутренней ошибки.

4.3. Отладка тестовых программ

Опыт генерации тестовых программ показывает, что следует уделять большое внимание отладке компонентов тестов. Ошибки в описании тестов могут приводить к тому, что сгенерированные тестовые программы содержат не все тестовые воздействия или подготавливают их некорректным образом.

Мы выделяем два уровня отладки тестов: *проверка соответствия тестов модели микропроцессора* и *проверка соответствия модели микропроцессора эталонному симулятору*. Для отладки первого типа мы использовали подход на основе *пред- и постусловий* основных функций, реализуемых компонентами теста: функций конструирования зависимостей и тестовых ситуаций, функций подготовки тестовых ситуаций и других. Это позволяет находить значительное число ошибок, особенно в конструкторах. Отладку второго типа осуществлять значительно сложнее. Грубые ошибки, такие как несовместимость типов или конфликты использования ресурсов, находятся с

помощью прогона тестовых программ в симуляторе. Чтобы обнаружить более сложные ошибки, необходим детальный анализ трассы выполнения тестовых программ.

5. Заключение

В статье был рассмотрен подход к автоматической генерации тестовых программ для микропроцессоров. В отличие от распространенных на практике методов, таких как тестирование на основе существующего ПО и тестирование на основе случайных программ, предлагаемый подход является значительно более систематичным и технологичным. Кроме того, поскольку он основан на использовании моделей, он позволяет формализовать критерии качества тестирования.

Методы тестирования, предлагаемые в исследовательских работах, ориентированы, прежде всего, на достижение сравнительно небольшого числа “крайних случаев”; для этого используются модели, описывающие микропроцессор с потактовой точностью. В отличие от таких узконаправленных методов наш подход использует более простые, понятные разработчикам модели и генерирует при этом массивные тесты, включающие все сочетания инструкций микропроцессора, интересные для тестирования. Это позволяет сократить затраты на разработку тестов, обеспечив при этом приемлемое качество тестирование, которое (что очень важно) может быть улучшено путем детализации модели и критериев тестового покрытия.

Если говорить о направлениях дальнейшей работы, то в первую очередь следует сказать о доработке генератора с уровня прототипа до уровня полноценного инструмента. Есть много пунктов, по которым такую доработку можно осуществить. Например, в существующей версии генератора описания инструкций, зависимостей между инструкциями и тестовых ситуаций делаются на языке Java, хотя было бы удобнее использовать для этих целей специализированные языки. Другой интересной возможностью является повышение уровня автоматизации разработки тестов. К примеру, конструкторы тестовых ситуаций, которые сейчас пишутся вручную, можно попробовать реализовать с использованием системы разрешения ограничений. Это позволит описывать тестовые ситуации декларативно, в форме предикатов, а их построение будет осуществляться полностью автоматически.

Литература

[1] MIPS R4000PC/SC Errata, Processor Revision 2.2 and 3.0. MIPS Technologies Inc., May 1994.

[2] R. Ho, C. Han Yang, M. Horowitz, D. L. Dill. Architecture Validation for Processors. ISCA’1995: International Symposium on Computer Architecture, June 1995.

[3] http://www.ispras.ru.

[4] http://www.unitesk.com.

[5] MIPS64™ Architecture For Programmers. Revision 2.0. MIPS Tecnologies Inc., June 2003.

[6] R. Ho. Validation Tools for Complex Digital Designs. PhD Thesis. November, 1996.

[7] 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.

[8] http://www.haifa.ibm.com/projects/verification/GenesysPresent/index.htm.

[9] L. Fournier, Y. Arbetman, M. Levinger. Functional Verification Methodology for Microprocessors Using the Genesys Test Program Generator: Application to the x86 Microprocessors Family. DATE’1999: Design Automation and Test in Europe, IEEE CS Press, 1999.

[10] S. Ur and Y. Yadin. Micro Architecture Coverage Directed Generation of Test Programs. DAC’1999: Design Automation Conference, 1999.

[11] P. Mishra, N. Dutt. Automatic Functional Test Program Generation for Pipelined Processors Using Model Checking. HLDVT’2002: Proceedings of the 7th IEEE International High-Level Design Validation and Test Workshop, 2002.

[12] P. Mishra, N. Dutt. Architecture Description Language Driven Functional Test Program Generation for Microprocessors Using SMV. CECS Technical Report 02-26, September 2002.

[13] H.M. Koo, P. Mishra. Test Generation Using SAT-based Bounded Model Checking for Validation of Pipelined Processors. ACM Great Lakes Symposium on VLSI, 2006.

[14] H.M. Koo, P. Mishra. Functional Test Generation Using Property Decomposition for Validation of Pipelined Processors. DATE’2006: Design, Automation and Test in Europe, March 2006.

[15] P. Mishra, N. Dutt. Graph-Based Functional Test Program Generation for Pipelined Processors. DATE’2004: Design, Automation and Test in Europe Conference and Exhibition, Volume 1, 2004.

[16] P. Mishra, N. Dutt. Functional Coverage Driven Test Generation for Validation of Pipelined Processors. DATE’2005: Design, Automation and Test in Europe, Volume 2, 2005.

[17] P. Mishra, N. Dutt, N. Krishnamurthy, M.S. Abadir. A Top-Down Methodology for Validation of Microprocessors. IEEE Design and Test, 2004.

[18] 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.

[19] www.cs.cmu.edu/~modelcheck/smv.html.

[20] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, Y. Wolfsthal. Coverage Directed Test Generation Using Symbolic Techniques. FMCAD’1996: Formal Methods in Computer Aided Design, LNCS 1166, Palo Alto, CA, USA, November 1996.

[21] K. Kohno, N. Matsumoto. A New Verification Methodology for Complex Pipeline Behavior. DAC’2001: Design Automation Conference, 2001.

[22] F. Corno, G. Squillero. Exploiting Auto-Adaptive GP for Highly Effective Test Programs Generation. ICES’2003: The 5th International Conference on Evolvable Systems: From Biology to Hardware, Trondheim, Norway, March 2003.

[23] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante. A Genetic Algorithm-based System for Generating Test Programs for Microprocessor IP Cores. ICTAI’2000: The 12th IEEE International Conference on Tools with Artificial Intelligence, Vancouver, British Columbia, Canada, November 2000.

[24] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante. On the Test of Microprocessor IP Cores. DATE'2001: IEEE Design, Automation and Test in Europe Conference, Munich, Germany, March 2001.
[25] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero. Fully Automatic Test Program Generation for Microprocessor Cores. DATE'2003: Design, Automation and Test in Europe, Munich, Germany, March 2003.
[26] E. Sanchez, M. Schillaci, M. Sonza Reorda, G. Squillero, L. Sterpone, M. Violante. New Evolutionary Techniques for Test-Program Generation for Complex Microprocessor Cores. GECCO'2005: Genetic and Evolutionary Computation Conference, Washington, DC, USA, June 2005.
[27] F. Corno, G. Squillero. An Enhanced Framework for Microprocessor Test-Program Generation. EUROGP'2003: The 6th European Conference on Genetic Programming, Essex, UK, April 2003.
[28] G. Squillero. MicroGP – An Evolutionary Assembly Program Generator. Genetic Programming and Evolvable Machines, Volume 6(3), 2005.
[29] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero. Efficient Machine-Code Test-Program Induction. CEC'2002: Congress on Evolutionary Computation, Honolulu, Hawaii, USA, 2002.
[30] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero. Evolutionary Test Program Induction for Microprocessor Design Verification. ATS'2002: IEEE Asian Test Symposium, Guam, USA, November 2002.
[31] F. Corno, M. Sonza Reorda, G. Squillero. Automatic Test Program Generation for Pipelined Processors. SAC'2003: The 18th Annual ACM Symposium on Applied Computing, Melbourne, Florida, USA, March 2003.
[32] F. Corno, E. Sanchez, M. Sonza Reorda, G. Squillero. Code Generation for Functional Validation of Pipelined Microprocessors. Journal of Electronic Testing: Theory and Applications, Volume 20(3), June 2004.
[33] W. Lindsay, E. Sanchez, M. Sonza Reorda, G. Squillero. Automatic Test Programs Generation Driven by Internal Performance Counters. MTV'2004: The 5th International Workshop on Microprocessor Test and Verification, 2004.
[34] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero. Automatic Test Program Generation from RT-Level Microprocessor Descriptions. ISQED'2002: The 3rd International Symposium on Quality Electronic Design, San Jose, California, USA, March 2002.
[35] F. Corno, E. Sanchez, M. Sonza Reorda, G. Squillero. Automatic Test Program Generation – A Case Study. IEEE Design and Test, Special Issue on Functional Verification and Testbench Generation, Volume 21, Issue 2, March-April 2004.
[36] D. Moundanos, J. Abraham, Y. Hoskote. A Unified Framework for Design Validation and Manufacturing Test. ITC'1996: International Test Conference, 1996.
[37] D. Moundanos, J. Abraham, Y. Hoskote. Abstraction Techniques for Validation Coverage Analysis and Test Generation. IEEE Transactions on Computers, Volume 47, 1998.
[38] D. Patterson, J. Hennesy. Computer Organization and Design: The Hardware/Software Interface. 3rd Edition. The Morgan Kaufmann Series in Computer Architecture and Design, 2005.
[39] RM7000 Family User Manual. Issue 1, May 2001.