Федеральное государственное бюджетное учреждение науки
Институт системного программирования Российской академии наук

На правах рукописи

Татарников Андрей Дмитриевич

АВТОМАТИЗАЦИЯ КОНСТРУИРОВАНИЯ ГЕНЕРАТОРОВ ТЕСТОВЫХ ПРОГРАММ ДЛЯ МИКРОПРОЦЕССОРОВ НА ОСНОВЕ ФОРМАЛЬНЫХ СПЕЦИФИКАЦИЙ

Специальность 05.13.11 –
математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата технических наук

Научный руководитель
к.ф.-м.н. Камкин Александр Сергеевич

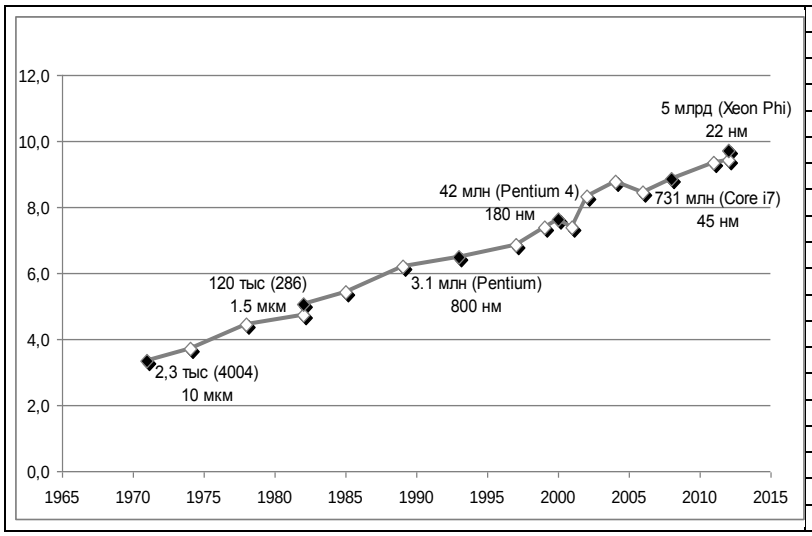
Москва 2017

Содержание

Введение

Актуальность темы

Микропроцессоры [16] лежат в основе большинства электронных устройств. В то время как другие компоненты отвечают за ввод и вывод данных, роль микропроцессора состоит в обработке этих данных: он определяет, какие операции должны быть выполнены над данными и контролирует процесс их выполнения. Микропроцессоры состоят из множества транзисторов, объединенных в единый вычислительный элемент на полупроводниковом кристалле. Число транзисторов увеличивается по закону Мура [17] и в настоящее время достигает нескольких миллиардов [18] (см. рисунок 1).



Год	Микропроцессор	Транзисторы
1971	4004	2 300
1974	8080	5 000
1978	8086	29 000
1982	186	55 000
1982	286	134 000
1985	386	275 000
1989	486	1 180 235
1993	Pentium	3 100 000
1997	Pentium II	7 500 000
1999	Pentium III	24 000 000
2000	Pentium 4	42 000 000
2001	Itanium	25 000 000
2002	Itanium 2	220 000 000
2004	Itanium 2 9M	592 000 000
2006	Core 2 Duo (2 ядра)	291 000 000
2008	Core i7 (4 ядер)	731 000 000
2011	Core i7 (6 ядер)	2 270 000 000
2012	Xeon Westmere-EX (8 ядер)	2 600 000 000
2012	Xeon Phi (62)	5 000 000 000

Рисунок 1. Рост числа транзисторов (на графике — десятичный логарифм) в микропроцессорах компании Intel

Высокая сложность современных микропроцессоров, вызванная оптимизацией производительности и энергопотребления, приводит ошибкам проектирования. Дальнейшее усложнение влечет за собой рост числа ошибок. Например, по данным на август 2008-го года в микропроцессоре Intel Pentium 4 было найдено 104 ошибки, из которых 43 не исправлено и их исправление не запланировано [19]. В то же время в микропроцессоре Intel Core i7-900 по данным на февраль 2015-го года обнаружено 167 ошибок, из которых исправлено только 16 и еще для 2 запланировано исправление [20]. Следует понимать, что в обоих случаях речь идет лишь об известных проблемах, в то время как реальное число ошибок может быть гораздо выше.

Цена ошибок в микропроцессорах очень высока. В отличие от дефектов в программах, которые устраняются сравнительно просто, ошибки в микропроцессорах не могут быть исправлены и для их устранения потребуется повторный выпуск и замена микросхемы или целого блока. Например, в 1994-м году замена продукции из-за ошибки в реализации команды FDIV микропроцессора Pentium обошлась компании Intel в 475 миллионов долларов [21].

Обеспечение функциональной корректности микропроцессоров является фундаментальной проблемой, для решения которой применяется комплекс мер, известный как *функциональная верификация*. Она выполняется параллельно с проектированием, и ее задача заключается в проверке соответствия результатов, полученных на текущем этапе, заданным требованиям и ограничениям. Верификация является весьма трудоёмкой задачей. По некоторым оценкам, затраты на нее достигают 70-80% от общих затрат на проектирование, а число инженеров-верификаторов примерно вдвое превосходит число инженеров-разработчиков [22]. С ростом сложности микропроцессоров ситуация только ухудшается — возможности методов верификации отстают от развития микропроцессоров; соответственно, проверка корректности (и без того являющаяся самым узким местом процесса проектирования) вовлекает в себя все бóльшие объемы ресурсов. Таким образом, задача совершенствования методов и инструментов верификации имеет ключевое значение.

Наиболее часто применяемым на практике подходом к функциональной верификации микропроцессоров является *имитационная верификация* (*simulation-based verification*), также называемая *тестированием*. На *системном уровне* (т.е. на уровне устройства в целом) она осуществляется следующим образом: создаются *тестовые программы* на языке ассемблера; программы запускаются на проектной модели микропроцессора; в результате получаются *трассы исполнения*, содержащие информацию о событиях, которые

произошли в процессе исполнения программ; полученные трассы проверяются на корректность.

Создание тестовых программ осуществляется при помощи специальных программных инструментов, известных как *генераторы тестовых программ*. Они используют различные *техники генерации* для обеспечения максимальной полноты тестирования: от случайной генерации до нацеленной генерации, основанной на формальных методах. Ни одна из них не является универсальным решением для всех задач верификации, поэтому на практике применяется множество дополняющих друг друга техник. Общепринятым подходом является генерация на основе *шаблонов тестовых программ*, описывающих структурные и поведенческие свойства программ. Обработка тестового шаблона состоит в применении той или иной техники генерации для удовлетворения того или иного свойства.

Как правило, генераторы тестовых программ предназначены для конкретных микропроцессорных архитектур и основаны на конкретных техниках генерации. Однако, так как микропроцессорные архитектуры и техники генерации эволюционируют, возникает задача расширения возможностей существующих генераторов. Трудность ее решения заключается в том, что знание об архитектуре микропроцессора зачастую неотделимо от реализации техник генерации. Поддержка новых архитектур и техник генерации требует существенных изменений в реализации генератора. Чтобы этого избежать, инженеры-верификаторы вынуждены одновременно использовать несколько генераторов, каждый из которых решает какую-то отдельную задачу. При этом совместное использование различных техник генерации для решения общей задачи оказывается невозможным.

Таким образом, актуальной является задача разработки метода, позволяющего создавать для любых микропроцессорных архитектур генераторы тестовых программ, интегрирующие в себе разные техники генерации. Перспективным решением данной задачи видится автоматизированное конструирование генераторов тестовых программ на

основе формальных спецификаций архитектуры микропроцессора. При этом формальные спецификации будут выступать в качестве источника знания об архитектуре тестируемого микропроцессора, используемого компонентами генератора, реализующими техники генерации. Генерация будет осуществляться на основе шаблонов, разработанных на специальном языке, позволяющем описывать свойства тестовых программ в терминах формальных спецификаций и задавать техники генерации, применяемые для удовлетворения этих свойств.

Цель и задачи работы

Цель работы — разработка метода автоматизации конструирования генераторов тестовых программ для микропроцессоров. Метод должен быть применимым к широкому спектру микропроцессорных архитектур. Генераторы должны создавать тестовые программы на языке ассемблера по шаблонам, описывающим структурные и поведенческие свойства этих программ. Генераторы должны реализовывать разные техники генерации и быть расширяемыми. Для достижения цели работы были поставлены следующие задачи:

1. Провести анализ существующих методов и средств генерации тестовых программ для микропроцессоров.
2. Разработать метод автоматизации конструирования генераторов тестовых программ для микропроцессоров на основе формальных спецификаций.
3. Разработать язык описания шаблонов тестовых программ, позволяющий описывать их структурные и поведенческие свойства.
4. Разработать архитектуру расширяемого генератора тестовых программ для микропроцессоров, позволяющую интегрировать разные техники генерации.
5. Разработать программный инструмент, реализующий предложенный метод автоматизации конструирования генераторов тестовых программ.
6. Оценить характеристики предложенного метода на основе опыта применения разработанного программного инструмента для

конструирования генераторов тестовых программ для нескольких микропроцессорных архитектур.

Научная новизна работы

Научной новизной обладают следующие результаты работы:

1. Метод автоматизации конструирования генераторов тестовых программ для микропроцессоров на основе формальных спецификаций.
2. Язык описания шаблонов тестовых программ, позволяющий описывать их структурные и поведенческие свойства.
3. Архитектура генератора тестовых программ для микропроцессоров, позволяющая интегрировать разные техники генерации.

Теоретическая и практическая значимость

В работе предложен метод автоматизации конструирования генераторов тестовых программ для микропроцессоров. В основе предложенного метода лежат архитектурно-независимые техники генерации, для применения которых к конкретной микропроцессорной архитектуре используется информация, полученная в результате анализа формальных спецификаций этой архитектуры. Результаты проведенного исследования могут послужить основой для разработки архитектурно-независимых техник генерации тестовых программ и создания программных инструментов, основанных на анализе формальных спецификаций. Кроме этого, эти результаты могут использоваться в исследовательских проектах и учебных курсах по проектированию и верификации микропроцессоров.

На основе предложенного метода разработан программный инструмент MicroTESK, позволяющий автоматизировать на основе формальных спецификаций конструирование генераторов тестовых программ для микропроцессоров. Разработанный инструмент был применен для создания генераторов тестовых программ для архитектур MIPS64, ARMv8, PowerPC и RISC-V. Генераторы тестовых программ для MIPS64 и ARMv8 используются в отечественных и зарубежных компаниях. Помимо этого, разработанный

инструмент может быть использован для создания генераторов тестовых программ для широкого спектра других микропроцессорных архитектур.

Методология и методы исследования

Методологическую основу исследования составляют теория компиляторов, теория формальных языков, теория автоматов, теория множеств, теория графов, теория алгоритмов и математическая логика.

Положения, выносимые на защиту

1. Метод автоматизации конструирования генераторов тестовых программ для микропроцессоров на основе формальных спецификаций.
2. Язык описания шаблонов тестовых программ, позволяющий описывать их структурные и поведенческие свойства.
3. Архитектура генераторов тестовых программ для микропроцессоров, позволяющая интегрировать разные техники генерации и допускающая расширение множества поддерживаемых техник.
4. Программный инструмент, использующий предложенный метод для конструирования генераторов тестовых программ с предложенной архитектурой, осуществляющих генерацию на основе шаблонов на предложенном языке.

Апробация работы

Основные положения работы обсуждались на следующих конференциях и семинарах:

- Международная конференция «Design Automation Conference», выставка University Booth (г. Остин, США, 2-7 июня 2013 г. и г. Сан-Франциско, США, 2-5 июня 2014 г.);
- Международная конференция «Design, Automation & Test in Europe», выставка University Booth (г. Гренобль, Франция, 18-22 марта 2013 г.; г. Дрезден, Германия, 24-28 марта 2014 г.; г. Гренобль, Франция, 10-12 марта 2015 г.; г. Лозанна, Швейцария, 28-30 марта 2017 г.);

- Международный коллоквиум молодых исследователей в области программной инженерии «Spring/Summer Young Researchers' Colloquium on Software Engineering» (г. Пермь, 30-31 мая 2012 г. и д. Красновидово, 30 мая-1 июня 2016 г.);
- Международная конференция «A.P. Ershov Informatics Conference» (г. Москва, 27-29 июня 2017 г.);
- Открытая конференция ИСП РАН (г. Москва, 1-2 декабря 2016 г.);
- Международный симпозиум «IEEE East-West Design & Test Symposium» (г. Ереван, Армения, 14-17 октября 2016 г.);
- Всероссийская научно-техническая конференция «Проблемы разработки перспективных микро- и наноэлектронных систем» МЭС-2016 (г. Москва, 3-7 октября 2016 г.);
- Международная конференция «Новые информационные технологии в исследовании сложных структур» (г. Екатеринбург, 6-10 июня 2016 г.);
- Международная летняя школа молодых ученых «Новые информационные технологии в исследовании сложных структур» (г. Анапа, 8-12 июня 2015 г.);
- Научно-техническая конференция студентов, аспирантов и молодых специалистов НИУ ВШЭ им. Е.В. Арменского (г. Москва, 4 февраля 2015 г.);
- Совместный семинар ЗАО «МЦСТ» и ИСП РАН «Проблемы верификации микропроцессоров» (г. Москва, 10 апреля 2014 г.);
- Семинар Института системного программирования РАН (г. Москва, 2013, 2014 и 2016 гг.).

Публикации

По теме диссертации автором опубликовано 15 работ, в том числе 7 научных статей [1–7] в рецензируемых журналах, входящих в перечень журналов, рекомендованных ВАК РФ. В работе [2] автором описывается архитектура инструмента MicroTESK. В статье [5] вклад автора заключается в

описании средств системной верификации микропроцессоров. В работах [6, 14] автору принадлежит описание концепции и архитектуры расширяемой среды генерации тестовых программ. В работах [7, 10, 11] вклад автора состоит в разработке средств моделирования подсистемы памяти и конструкций языка описания шаблонов тестовых программ, позволяющих создавать тесты на подсистему памяти. В работе [13] автором дается обзор существующих подходов и формулируются требования к системе хранения информации, используемой для построения тестов. В статье [15] автором описываются предлагаемый подход к моделированию архитектуры микропроцессора и концепция языка описания шаблонов тестовых программ.

Личный вклад

Все представленные в диссертации результаты получены лично автором.

Структура и объем диссертации

Работа состоит из введения, четырех глав, заключения, списка литературы (122 наименования) и одного приложения. Основной текст диссертации (без списка литературы и приложения) занимает 148 страниц.

Краткое содержание диссертации

Во введении обосновывается актуальность темы работы, определяются ее цели и задачи, раскрывается теоретическая и практическая значимость.

Первая глава содержит обзор подходов к функциональной верификации микропроцессоров. Основное внимание уделяется верификации при помощи тестовых программ. В главе рассматриваются сильные и слабые стороны распространенных техник генерации тестовых программ, области их применения и варианты их совместного использования. Также делается сравнительный анализ возможностей существующих инструментов, реализующих эти техники. На основе данного анализа делается вывод об отсутствии генератора тестовых программ, который был бы применим для широкого спектра микропроцессорных архитектур и позволил бы

интегрировать основные применяемые на практике техники генерации, и выдвигается тезис об актуальности задачи его создания.

Во второй главе предлагается метод автоматизации конструирования генераторов тестовых программ для микропроцессоров на основе формальных спецификаций их архитектуры. Данный метод предполагает, что информация, извлеченная из формальных спецификаций, будет использоваться для обеспечения целенаправленности тестирования и контроля состояния микропроцессора в процессе генерации. Построение тестовых программ будет осуществляться на основе шаблонов, разработанных на специальном расширяемом языке, позволяющем описывать их структурные и поведенческие свойства. Расширяемая архитектура генераторов позволит интегрировать разные техники генерации. Глава состоит из трех разделов, в которых описывается предлагаемый метод автоматизации конструирования генераторов тестовых программ, язык описания шаблонов тестовых программ и архитектура конструируемых генераторов тестовых соответственно.

В третьей главе описывается реализация предложенного метода автоматизации конструирования генераторов тестовых программ. Метод нашел свое воплощение в инструменте с открытым исходным кодом MicroTESK (Microprocessor TEsting and Specification Kit) версии 2.0, разработанном на языке Java. Данный инструмент на основе формальных спецификаций конструирует генераторы тестовых программ, состоящие из модели микропроцессора и архитектурно-независимого ядра. Глава состоит из двух разделов, в которых описывается реализации инструмента MicroTESK и реализация архитектурно-независимого ядра генераторов тестовых программ соответственно.

В четвертой главе описываются результаты применения предложенного метода автоматизации конструирования генераторов тестовых программ для микропроцессоров MIPS64, ARMv8, PowerPC и RISC-V. В главе оценивается трудоемкость создания генераторов с применением предложенного метода и приводится сравнение с результатами применения других аналогичных

методов. Также в главе обосновывается применимость предложенного метода для конструирования генераторов тестовых программ для микропроцессорных архитектур промышленного масштаба.

В заключении перечисляются основные результаты работы.

Глава 1. Генерация тестовых программ для микропроцессоров

Данная глава содержит обзор техник и инструментов, применяемых для генерации тестовых программ для функциональной верификации микропроцессоров. В начале главы рассказывается об области применения генераторов тестовых программ, а затем делается сравнительный анализ существующих техник и инструментов. На основе данного анализа формулируется основные требования к методу автоматизации конструирования генераторов тестовых программ, предлагаемому в данной работе.

1.1 Проектирование микропроцессоров

Проектирование микропроцессора – сложный процесс, включающий в себя нескольких этапов [16, 25, 26], на каждом из которых создается описание микропроцессора на определенном уровне абстракции. На рисунке 2 показана общая схема процесса проектирования микропроцессора.

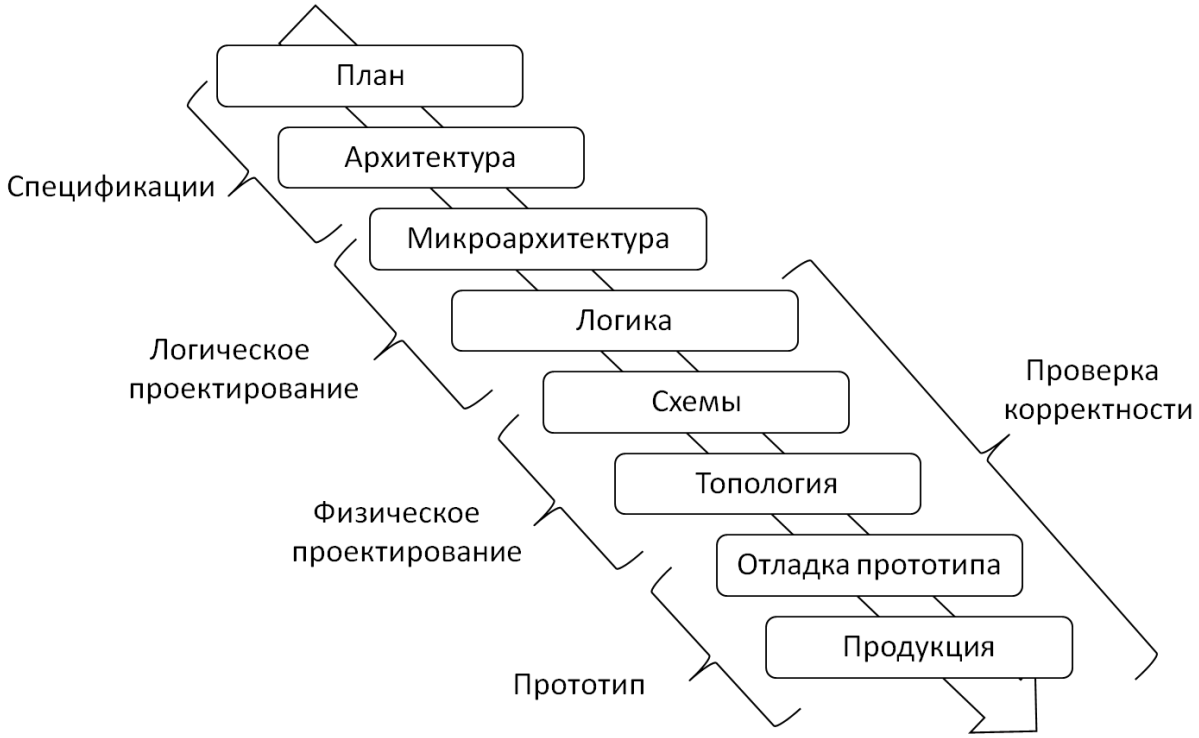


Рисунок 2. Процесс проектирования микропроцессора

Проектирование микропроцессора начинается с составления детального *плана* (*технического задания*), в котором описывается концепция проектируемого устройства, целевая рыночная ниша и основные

характеристики (такие как цена, производительность и энергопотребление). Также определяется на основе каких предыдущих разработок будет создан микропроцессор и какое программное обеспечение будет на нем запускаться. Помимо этого делается оценка времени и ресурсов, отведенных на проектирование, составляется график работ и выбирается общая методология проектирования.

На этапе проектирования *архитектуры* определяется, какие команды будут поддерживаться микропроцессором. На этом этапе уточняются требования и создается спецификация архитектуры. Помимо документов требований и спецификаций архитектуры результатом данного этапа является *программный эмулятор* (*симулятор*), позволяющий интерпретировать программы, написанные для целевого микропроцессора. Он может использоваться для кросс-разработки программного обеспечения или выступать в качестве эталонной модели.

После архитектуры проектируется *микроархитектура*. Она описывает функциональные модули микропроцессора, их взаимодействие и разделение задач между ними. На этом этапе определяются такие характеристики, как организация конвейера команд и иерархии памяти, способ реализации многопоточности и т.д. Обычно результатом проектирования микроархитектуры являются диаграммы взаимодействия между модулями и спецификации, описывающие различные алгоритмы. Кроме того на этапах проектирования архитектуры и микроархитектуры осуществляется *моделирование*. Для этого применяются следующие средства: (1) *языки программирования общего назначения*, (2) *языки системного проектирования* (*SLDL, System-Level Design Languages*) и (3) *языки описания архитектуры* (*ADL, Architecture Description Languages*) [27, 28, 29]. Примеры языков указанных типов представлены в таблице 1. С их помощью создаются модели для проведения различных экспериментов (например, программные эмуляторы или формальные модели), а также средства разработки для проектируемой архитектуры (компиляторы, дизассемблеры, и т.д.).

Таблица 1. Языки, используемые при проектировании микропроцессоров

Тип языка	**Примеры**
Языки программирования общего назначения	C, C++, Perl, Python
Языки описания архитектуры (ADL)	LISA, EXPRESSION, ISDL, nML
Языки системного проектирования (SLDL)	SystemC, SystemVerilog, Bluespec
Языки описания аппаратуры (HDL)	Verilog, VHDL

На следующем этапе при помощи *языков описания аппаратуры* (*HDL, Hardware Description Languages*) описывается логическая структура микропроцессора. Результатом данного этапа является *модель уровня регистровых передач* (*RTL, Register Transfer Level*), называемая также *HDL-моделью* или *HDL-описанием,* которая с потактовой точностью определяет пересылки данных, возникающие при работе устройства. На этапе разработки HDL-модели часто прибегают к *прототипированию (prototyping)* [30]. Оно предполагает создание функционально идентичной реализации, выполненной с использованием *программируемых логических интегральных схем* (*ПЛИС*), с целью проведения различных экспериментов.

После этого на основе HDL-модели строится функционально ей эквивалентная *схема* из логических вентилей в заданном технологическом базисе. Процесс построения данной схемы осуществляется средствами *систем автоматизированного проектирования* (САПР) и называется *логическим синтезом.*

Далее осуществляется построение *топологии*, которая описывает размещение транзисторов и проводников в слоях материала, их которого изготовляется схема на кристалле. Этот этап называется *физическим синтезом* и он, также как и логический синтез, автоматизирован средствами САПР. В результате получается представление (как правило, основанное на фотошаблонах), которое затем используется при производстве интегральных схем.

По завершению физического проектирования осуществляется *выпуск (tape-out)* первого *опытного образца (first silicon)* интегральной схемы. Данный образец проходит несколько циклов отладки, в процессе которых выявляются и

исправляются ошибки, перед тем как, созданный микропроцессор может быть выпущен на рынок.

1.2 Функциональная верификация микропроцессоров

Параллельно с процессом проектирования осуществляется контроль корректности полученных результатов. Для обеспечения функциональной корректности микропроцессоров применяется комплекс мер, известный как *функциональная верификация*. Прежде всего, эти меры нацелены на обнаружение и исправление ошибок, допущенных в процессе проектирования. Хотя функциональная верификация присутствует на всех этапах, особенно она актуальна при создании HDL-модели, поскольку функциональность, описанная на этом этапе, впоследствии не изменяется [26]. Верификация может осуществляться как на уровне отдельных модулей микропроцессора (*модульная* верификация), так и для устройства в целом (*системная* верификация).

Существуют два основных подхода к функциональной верификации: (1) *имитационная верификация* (*simulation-based verification*), также называемая *тестированием*, и (2) *формальная верификация* (*formal method-based verification*) [31]. Первый заключается в проверке корректности реакции проектируемого устройства, работа которого имитируется при помощи программного или аппаратного эмулятора, на *тестовые воздействия*. Второй основан на построении математической (формальной) модели и проверке выполнимости ее свойств. Формальная верификация позволяет осуществить исчерпывающую проверку всех возможных вариантов поведения, заданных моделью. Однако она имеет ряд ограничений (высокая трудоемкость и вычислительная сложность, а также трудности формализации), которые затрудняют ее использование в промышленных проектах. По этой причине на практике основной акцент делается на тестирование [32].

Тестирование микропроцессоров на системном уровне осуществляется путем генерации *тестовых программ* на языке ассемблера, которые представляют собой последовательности команд (инструкций) микропроцессора, и анализа результатов их выполнения с целью убедиться, что

их поведение соответствует спецификации. Такой подход видится наиболее естественным, т.к. система команд определяет функциональность микропроцессора и является единственным доступным интерфейсом, через который пользователи могут с ним взаимодействовать. Верификация при помощи тестовых программ применима на всех этапах проектирования, начиная с момента создания программного эмулятора, но на каждом этапе имеет свои особенности, связанные со скоростью выполнения программ и уровнем детализации полученных результатов. В таблице 2 показаны примерная скорость выполнения тестов и уровень детализации результатов на различных прототипах [33]. Обычно тестирование начинается с HDL-модели и продолжается до выпуска продукции.

Таблица 2. Скорость выполнения тестов и уровень детализации результатов на различных прототипах

Объект тестирования	**Скорость выполнения (инстр./сек.)**	**Уровень детализации**
HDL-эмулятор	10^3	Очень высокая
Эталонный эмулятор на C/C++	$3 * 10^6$	Высокая
ПЛИС-прототип	10^7	Средняя
Опытный образец микропроцессора	$10^8 – 10^9$	Очень низкая

Существуют два основных подхода к функциональной верификации микропроцессоров при помощи тестовых программ: (1) сравнение *трасс выполнения* с эталонными трассами и (2) использование *встроенных проверок* (*self-checks*).

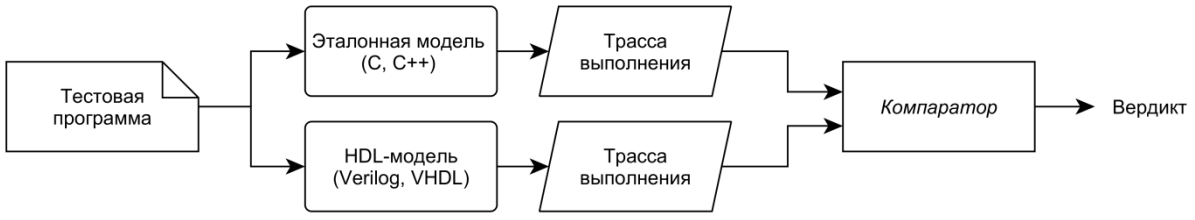


Рисунок 3. Тестирование посредством сравнения трасс

В первом случае (см. рисунок 3) при выполнении программы на HDL-модели или другой программной модели создается трасса выполнения, фиксирующая события, которые происходят в микропроцессоре. Полученная трасса сравнивается с эталонной трассой, полученной в результате выполнения той же программы на *эталонной модели (reference model)*. Для сравнения трасс используются специальные программы, называемые *компараторами*.

Эталонная модель создается независимо от HDL-модели на языке высокого уровня (например, C или C++) и является более абстрактной. Часто в качестве эталонной модели выступает *эмулятор уровня инструкций (instruction-level simulator)*, разработанный на стадии архитектурного проектирования. Более абстрактная модель, как правило, содержит меньшее количество ошибок, а тот факт, что она разрабатывается независимо, уменьшает вероятность повторения ошибок, допущенных при создании HDL-модели. Данный подход позволяет универсальным образом осуществлять проверку корректности поведения микропроцессора при выполнении различных тестовых программ. Таким образом, отпадает необходимость реализовывать проверки для отдельных тестов. К недостаткам подхода, можно отнести то, что трассы, полученные при помощи упрощенной эталонной модели, могут не отражать все события, которые происходят в HDL-модели (и реальном процессоре). Это приводит к трудностям при сопоставлении трасс.

Второй подход предполагает, что код тестовой программы содержит проверки, которые необходимо осуществить в процессе ее выполнения. Такие программы можно выполнять не только на HDL-моделях, но и на аппаратных ускорителях, ПЛИС-прототипах и экспериментальных образцах микросхем (*post-silicon verification*). Это позволяет повысить производительность, так как время выполнения программы на аппаратных эмуляторах существенно меньше, чем на программных. Однако при таком подходе снижается точность проверки, так как множество событий, которые может фиксировать тестовая программа, ограничено.

Решение о завершении верификации принимается на основе набора критериев, который включает в себя следующее [34]:

- выполнение плана верификации;
- отсутствие ошибок при прогоне регрессионных тестов;
- отсутствие ошибок на 10^5 псевдослучайных тестов для каждого из имеющихся тестовых шаблонов;
- достижение заданного уровня покрытия HDL-кода и функционального

покрытия;

- отсутствие изменений в HDL-коде в течение длительного времени (3-4 недели);
- отсутствие новых ошибок в течение определенного времени (2-3 недели);
- истечение отведенного согласно плану времени на разработку и верификацию.

Особого внимания заслуживают критерии, основанные на достижении требуемого уровня покрытия. Достигнутый уровень покрытия оценивается при помощи *метрик тестового покрытия* [35]. Они представляют собой количественные характеристики полноты покрытия выбранной *модели тестового покрытия,* которая описывается как конечный набор *тестовых ситуаций*. Числовое значение метрики покрытия представлено как отношение числа достигнутых при выполнении теста ситуаций к общему числу ситуаций в модели. Выделяют два типа моделей покрытия: (1) основанные на реализации и (2) основанные на функциональных требованиях.

В моделях первого типа тестовые ситуации связаны непосредственно с кодом HDL-модели или с производными от нее структурами. Т.е. метрики покрытия на основе реализации позволяют оценить, в какой мере код HDL-модели был задействован при выполнении набора тестов. Примерами таких метрик являются: покрытие строк кода (line coverage), покрытие операторов (statement coverage), покрытие переходов управления (branch coverage), а также покрытие состояний, дуг и путей (state, arc, transition coverage) конечных автоматов [36]. Большинство современных САПР предоставляют инструментарий для сбора и анализа данных о достигнутом покрытии реализации.

В моделях второго типа, известных как *модели функционального покрытия*, тестовые ситуации задаются в терминах абстракций микроархитектуры или архитектуры микропроцессора, а также абстракций более высокого уровня. Такие модели создаются на основе спецификаций *функциональных требований* к микропроцессору различного уровня. Они могут

создаваться вручную или посредством анализа *формальных спецификаций* [37], разработанных в процессе проектирования архитектуры и микроархитектуры.

Важно отметить, что ни одна из метрик тестового покрытия, взятая в отдельности не даёт ответа на вопрос о полноте набора тестов. Поэтому на практике используются комбинации различных метрик, основанных как на реализации, так и на функциональных требованиях.

Для генерации тестов, предназначенных для достижения требуемого уровня покрытия для выбранных метрик, применяются разнообразные техники автоматической генерации тестовых программ. Инструменты, при помощи которых осуществляется генерация, известны как *генераторы тестовых программ (TPG, test program generators)* или *генераторы потока команд (ISG, instruction stream generators).* Помимо генерации тестовых программ практикуется *ручная разработка* и *использование существующего программного обеспечения* [33], однако применимость этих подходов ограничена. Ручная разработка обладает высокой трудоемкостью. А существующее программное обеспечение не дает гарантий полноты функционально покрытия. Кроме того его запуск на HDL-эмуляторе может занимать много времени. Поэтому основной акцент делается на генерацию.

1.3 Техники генерации тестовых программ

Большинство современных генераторов тестовых программ в той или иной степени полагаются на *случайную генерацию* [38]. Однако применяемые техники генерации могут существенно отличаться степенью *нацеленности*. Под нацеленностью следует понимать ориентированность на покрытие конкретных тестовых ситуаций или классов тестовых ситуаций. Следует заметить, что т.к. генерация тестовых программ предполагает повторяемость результатов, алгоритмы случайной генерации, как правило, основаны на детерминированных генераторах псевдослучайных чисел.

В простейшем случае команды и их входные значения выбираются случайным образом [39]. Инструмент, решающий подобную задачу, можно разработать достаточно быстро, и он позволит сгенерировать набор тестов,

обеспечивающих базовый уровень покрытия. К сожалению, такой подход не является систематическим и не позволит достичь полного покрытия. Подобные инструменты не располагают сведениями о семантике команд и особенностях реализации микроархитектуры, поэтому не гарантируют достижения всех интересных с точки зрения тестирования ситуаций, а также корректности построенной программы (отсутствие зацикливаний, переходов на секцию данных, нарушений инвариантов команд, и т.д.).

Первый шаг в направлении повышения нацеленности случайных тестов – ограничение области допустимых значений для случайного выбора и присвоение *весов* выбираем вариантам [40]. Такой подход предполагает описание задач генерации в виде *шаблонов*, которые задают списки команд для выбора (в том числе иерархические), веса отдельных элементов и области значений их входных аргументов. Это позволит более тщательно протестировать отдельные команды или классы команд. Веса и области значений подбираются на основе полученных значений метрик тестового покрытия.

Следующий шаг в сторону улучшения покрытия предполагает использование техник комбинаторной генерации [41]. Такой подход позволит нацелить генератор на тестирование ситуаций, связанных с совместным выполнением команд на конвейере. Суть подхода заключается в систематическом переборе коротких последовательностей команд (включая зависимости между ними). Помимо этого можно получать входные аргументы команд путем перебора данных из некоторого предопределенного набора. Это позволит достичь лучшего покрытия *пограничных случаев (corner cases)*. Путем совместного использования техник случайной и комбинаторной генерации можно построить более сложные тесты, которые помогут покрыть многие маловероятные и трудновообразимые ситуации в работе микропроцессора.

Для проверки поведения микропроцессора в конкретных ситуациях часто прибегают к *детерминированной эмуляции (deterministic simulation)* [42]. Идея заключается в создании тестовых программ с детерминированным поведением,

включающих встроенные проверки. Изначально такие программы разрабатывались вручную, что делало их создание крайне трудоемкой задачей. Автоматическая генерация таких программ предполагает исполнение команд в процессе генерации и использование полученных результатов для создания проверок. Исполнение команд в процессе генерации также позволяет гарантировать корректность построенных программ. При данном подходе программы стоятся по шаблонам, которые описывают тестовый сценарий. Исполнение команд генератор осуществляет самостоятельно или использует для этого внешние эталонные модели. Построенные таким образом тесты можно считать полностью нацеленными. Основным недостатком данного подхода остается трудоемкость создания тестов, которая сопоставима с ручной разработкой. Как и при ручной разработке, чтобы создать код, покрывающий нетривиальные ситуации в работе микропроцессора, от разработчика требуется глубокое знание особенностей микроархитектуры. Кроме того, есть риск, что некоторые ситуации будут пропущены. С целью улучшения покрытия и уменьшения трудозатрат шаблоны могут использовать рандомизацию или комбинаторный перебор. Например, построение тестовых программ может осуществляться путем перебора разнообразных графов потока управления (*структур переходов*), описанных в шаблоне, и маршрутов в них (*трасс выполнения*) [43].

Построение нацеленных тестов можно упростить при помощи *генерации на основе ограничений* [44]. При таком подходе в шаблонах указываются ограничения, которым должны удовлетворять операнды команд. Ограничения соответствуют тестовым ситуациям, основанным на особенностях реализации микроархитектуры или на функциональных требованиях. В процессе обработки шаблона генератор, используя некоторый механизм разрешения ограничений, который зависит от их типа, строит случайные значения, удовлетворяющие заданным ограничениям. Использование ограничений позволяет значительно сократить трудозатраты, связанные с подбором значений операндов, требуемых для покрытия тестовых ситуаций. Однако, т.к. тестовые шаблоны по-прежнему

разрабатываются вручную, остается риск упустить важную для тестирования ситуацию.

Систематическое тестирование может быть обеспечено путем создания *формальных моделей* тестируемого микропроцессора [45] и применения к ним техник проверки моделей для построения тестов, охватывающих все пространство состояний. Такой подход называют *генерацией на основе моделей*. Т.к. современные микропроцессоры имеют огромное количество состояний и переходов между ними, у этого метода будут высокие вычислительные затраты. Для их сокращения можно использовать тестовые шаблоны, которые будут задавать направление тестирования. К недостаткам данного подхода относятся высокая трудоемкость его реализации (пока ведутся исследования, но нет промышленных инструментов), а также трудности создания формальных моделей.

Таблица 3. Техники генерации и требуем ые ими входные данные

Техника генерации	**Входные данные**
Случайная генерация	Ассемблерный формат команд; Тестовые шаблоны (списки команд, веса)
Комбинаторная генерация	Все выше перечисленное; Тестовые шаблоны (правила комбинирования)
Генерация детерминированных тестов со встроенными проверками	Все выше перечисленное; Тестовые шаблоны (описание тестовых сценариев); Семантика команд (эталонная модель)
Генерация на основе ограничений	Все выше перечисленное; Тестовые шаблоны (задание ограничений); Модели покрытия (наборы ограничений) различного типа
Генерация на основе моделей	Все выше перечисленное; Формальные модели микропроцессора

Как можно заметить, перечисленным техникам требуется разное количество входных данных. В таблице 3 приводится список техник и требуемых для них входных данных в порядке увеличения их количества. Реализации перечисленных техник в различных инструментах генерации могут отличаться. Отдельный инструмент может поддерживать один или несколько техник. Особенности реализации техник в конкретных инструментах будут рассмотрены далее.

1.4 Инструменты генерации тестовых программ

1.4.1 Инструменты НИИСИ РАН

В НИИСИ РАН разрабатывается система INTEG [40, 46], предназначенная для тестирования микропроцессоров MIPS64 [47]. Используемый ею подход к тестированию называется *стохастическим*. Он предполагает генерацию тестовых программ по заданному шаблону с параметризированным случайным выбором команд и их аргументов. После этого осуществляется исполнение сгенерированных программ на HDL-модели и на эталонной модели, в качестве которой выступает программный эмулятор VMIPS [48], и сравнение полученных результатов.

Генератор тестовых программ системы INTEG поддерживает случайные и комбинаторные техники генерации. Тестовые шаблоны для него разрабатываются на специализированном языке, конструкции которого основаны на синтаксисе языка C. Система предоставляет графический интерфейс, который упрощает их разработку. Шаблоны задают последовательность фрагментов кода в тестовой программе, состав входящих в них команд и аргументы этих команд. Все параметры, оставленные свободными, генератор выбирает случайно, в соответствии с заданными для них вероятностями. Основные возможности, предоставляемые языком описания тестовых шаблонов, включают: (1) задание областей памяти для кода и для данных; (2) описание последовательностей команд (в том числе конструкции для описания циклов); (3) выбор команд; (4) выбор регистров, используемых в качестве аргументов команд; (5) задание значений аргументов команд; (6) задание адресов данных и адресов передачи управления; (7) описание правил генерации случайных значений; (8) создание макросов для повторного использования шаблонного кода. При разработке тестовых шаблонов для системы INTEG руководствуются планом тестирования и метриками покрытия код HDL-модели. Для устранения пробелов в покрытии изменяются настройки шаблонов, такие как степень случайности выбора, область допустимых значений, т.д.

В процессе генерации система INTEG отслеживает состояние микропроцессора, используя упрощенный подход. При таком подходе исполнение команд на эмуляторе не осуществляется, а значение регистров в некоторой точке выполнения считается известным или неопределенным. При необходимости генератор обновляет значения регистров и обеспечивает защиту от записи в регистры, часто используемые для чтения.

Система INTEG предоставляет достаточно мощный и удобный инструментарий для создания случайных тестов. Средства разработки тестовых шаблонов, генерации тестовых программ, выполнения тестирования, а также анализа его результатов интегрированы в единую систему. Однако данная система имеет ряд ограничений:

- Поддерживаются только случайная и комбинаторная генерация. Остается неясно, какой уровень функционального покрытия можно обеспечить этими методами и насколько сложно разработать шаблоны, удовлетворяющие требованиям по тестовому покрытию.
- Не предусмотрено добавление пользовательских техник генерации и пользовательских конструкций в язык описания тестовых шаблонов.
- В процессе генерации построенные команды не исполняются на эталонном эмуляторе. Таким образом, генератор не может отслеживать состояния микропроцессора в любой точке выполнения тестовой программы. Эта информация необходима для контроля корректности сгенерированных программ и создания встроенных проверок. Кроме того она требуется техникам генерации, основанным на разрешении ограничений. Таким образом, реализация перечисленных возможностей столкнется со значительными трудностями.
- Поддерживается только архитектура MIPS64. В работе [46] есть упоминание о возможности настройки под другие архитектуры. Однако неясно, каким образом такая настройка будет осуществляться и каких трудозатрат она потребует.

1.4.2 Инструменты ARM

Инструменты RIS

В компании ARM [49] разработано семейство инструментов генерации тестовых программ, получившее название RIS (Random Instruction Sequence) [50, 51]. Это узкоспециализированные инструменты, предназначенные для решения различных задач тестирования микропроцессоров с архитектурой ARM. Решаемые ими задачи включают тестирование таких механизмов, как многоядерность [52] и управление памятью [53]. Настройка инструментов RIS осуществляется при помощи конфигурационных файлов, которые задают используемые команды, их веса, ограничения на их операнды, способы размещения кода и данных в памяти, а также цепочки команд, решающих специальные задачи (вытеснение данных из кэш-памяти и т.п.). Инструменты RIS, описанные в работах [52] и [53], не используют эталонные модели и не осуществляют исполнение команд в процессе генерации. В тех случаях, когда требуются встроенные проверки, тесты выполняются дважды и полученные результаты сравниваются (применимо только для детерминированных результатов). Такой подход упрощает разработку инструмента и делает возможным его использование непосредственно на ПЛИС-прототипе или экспериментальном образце микропроцессора. К сожалению, доступная информация об инструментах RIS не позволяет в полной мере оценить их функциональные возможности. Из того, что известно можно сделать вывод, что основным ограничением инструментов RIS является то, что они жестко привязаны к архитектуре ARM и ориентированы на решение конкретных задач. Поддержка других архитектур и техник генерации в них не предусмотрена.

Инструмент RAVEN

Другим инструментом, используемым в компании ARM, является RAVEN (Random Architecture Verification Machine) [54, 55, 56], разработанный в компании Obsidian Software, поглощенной ARM в 2011 году. Это

универсальное средство, применимое для широкого спектра архитектур, в котором ядро, реализующее логику генерации, отделено от конфигурации для конкретной архитектуры. RAVEN позволяет создавать как случайные, так и нацеленные тесты. В процессе работы инструмент отслеживает состояние микропроцессора путем исполнения построенных программ на внешней эталонной модели. Это позволяет гарантировать корректность построенных программ и использовать информацию о текущем состоянии микропроцессора для генерации тестов.

Конфигурация тестируемого микропроцессора задается в виде XML-файлов. Данные файлы содержат следующую информацию об архитектуре тестируемого микропроцессора: (1) перечень поддерживаемых команд и их групп, организованный в виде дерева; (2) сигнатуры команд (ассемблерный синтаксис, двоичная кодировка); (3) операнды команд, их свойства, используемые ими режимы адресации и правила их группировки; (4) семантика команд, представленная в виде формул; (5) ресурсы микропроцессора, к которым обращаются команды. Кроме того там также описываются специальные условия такие, как исключения, ситуации в работе конвейера инструкций и подсистемы памяти. Эти описания используются генератором для построения тестовых программ.

Другой важный аспект конфигурации – это внешняя эталонная модель, используемая генератором для отслеживания состояния микропроцессора. Она интегрируется в ядро инструмента при помощи специальных библиотек на языке C++. Как правило, модель разрабатывают производители микропроцессора, а интеграцию осуществляют разработчики инструмента. Это требует скоординированных совместных усилий т.к. для успешной интеграции необходимо, чтобы модель удовлетворяла требованиям, предъявляемым инструментом.

Задачи тестирования описываются при помощи шаблонов, которые разрабатывается вручную или создаются при помощи графического интерфейса. В шаблонах задаются используемые команды, вероятности их

появления, правила генерации входных значений, целевые тестовые ситуации т.д. Тестовые шаблоны разрабатываются в соответствии с таблицами тестового покрытия, которые формулируют цели тестирования. На основе метрик покрытия, полученных в результате выполнения построенных тестов, в набор шаблонов вносятся изменения до тех пор, пока требуемый уровень покрытия не будет достигнут.

Тестовые ситуации описываются в виде правил, основанных на ограничениях и различных эвристиках, которые хранятся в специальной базе знаний. Набор правил может пополняться. Это позволяет накапливать знания, используемые для построения тестов, и повторно применять их для тестирования других микропроцессоров.

Список тестовых ситуаций, покрытие которых может быть обеспечено при помощи инструмента RAVEN включает: (1) комбинации следующих друг за другом команд; (2) ситуации в работе операций с плавающей точкой; (3) исключения в работе команд; (4) конвейерные конфликты; (5) различные сценарии обработки запросов к иерархии памяти; (6) ситуации, связанные с совместным использованием памяти несколькими ядрами многоядерного микропроцессора.

К сожалению, информации об инструменте RAVEN, имеющаяся в открытом доступе, недостаточно подробна. Остается неясно, какой уровень нацеленности он позволяет достичь, насколько трудоемко писание тестовых ситуаций и предусматривается ли поддержка новых типов тестовых ситуаций. Также к недостаткам инструмента можно отнести то, что описание микропроцессора, используемое для генерации тестов, и эталонная модель являются дублирующимися представлениями одного и того же знания. Это усложняет поддержку особенно инструмента, если эти представления разрабатываются разными командами.

1.4.3 Инструменты IBM Research

В IBM Research разрабатывается несколько инструментов генерации тестовых программ, которые имеют различное назначение и используются в

различных промышленных проектах. Изначально эти инструменты создавались для верификации микропроцессоров семейства PowerPC, а в дальнейшем применялись и для других архитектур (таких, как ARM и x86).

Инструмент Genesys-Pro

В настоящее время основным средством генерации тестовых программ, используемым в IBM, является Genesys-Pro [32]. Это универсальный инструмент, позволяющий создавать случайные и нацеленные тесты для различных типов микропроцессоров. Данный инструмент разделен на *ядро*, которое реализует методы генерации, применимые для любых архитектур, и *модель*, которая содержит всю информацию о тестируемом микропроцессоре. Задачи тестирования формулируются при помощи *тестовых шаблонов*. Сгенерированные команды исполняются на внешнем программном эмуляторе с целью отслеживания состояния микропроцессора и контроля корректности построенной программы.

Инструмент Genesys-Pro включает в себя специальную среду моделирования, позволяющую создавать модели микропроцессоров на основе набора высокоуровневых блоков. Модели содержат информацию двух видов: (1) декларативное описание микропроцессора, которое включает в себя команды, ресурсы (регистры, память) и некоторые механизмы (например, трансляция адресов) и (2) тестовое знание для данного микропроцессора, которое представляет собой набор эвристик, позволяющих повысить уровень покрытия. Семантика команд описывается в виде ограничений на их входные аргументы. Описание команд также включает их сигнатуры, используемые ими ресурсы, типы данных аргументов и допустимые входные значения. Команды могут объединяться в группы. Среда моделирования имеет некоторые ограничения, не позволяющие описывать семантику некоторых типов команд. В частности для инструкций арифметики с плавающей точкой и команд доступа к памяти используются дополнительные инструменты, о которых будет рассказано в следующих разделах. Кроме того семантику некоторых сложных команд приходится описывать на языке C++.

Шаблоны создаются на специализированном языке [57], предоставляющем конструкции для описания последовательностей команд, распределений вероятностей и ограничений. Последовательности команд могут строиться при помощи техник случайной и комбинаторной генерации. Также язык предоставляет средства для описания последовательностей команд, которые будут выполняться в разных потоках (или разных ядрах). Входные аргументы команд генерируются случайным образом (с заданной вероятностью) или путем решения ограничений. Ограничения, задающие те или иные аспекты поведения инструкций, берутся из модели. Кроме этого существуют универсальные ограничения, которые можно разделить на следующие типы: (1) ограничения на выравнивание адресов; (2) зависимости по ресурсам для команд; (3) события, связанные с работой подсистемы памяти (промахи и попадания в различные буферы). Ограничения могут быть обязательными (hard) и необязательными (soft). Первые имеют более высокий вес при решении систем ограничений и, как правило, основаны на семантике команд. Вторые могут быть проигнорированы, если система ограничений не имеет решения, и обычно основаны на тестовом знании. Для ограничений можно задавать вероятности, с которыми они должны быть применены для выбранных команд. Также язык описания тестовых шаблонов поддерживает условную генерацию. Т.е. можно задавать пред- и постусловия, которые должны выполняться для того, чтобы фрагмент кода был добавлен в тестовую программу.

Генерация тестовых программ включает следующие стадии: (1) построение последовательностей команд; (2) решение ограничений (или генерация случайных значений) для каждой из команд; (3) исполнение команд на эталонной модели. В случае если не удается решить ограничения, инструмент может вносить коррективы в последовательность команд (добавлять дополнительные команд, повторно генерировать предыдущие команд).

Инструмент Genesys-Pro позволяет достичь достаточно высокого уровня нацеленности. Степень случайности тестов задается тестовыми шаблонами, которые разрабатываются в соответствии с планом верификации. Решение относительно полноты набора тестовых шаблонов принимается на основе метрик покрытия, основанных на реализации и функциональных требованиях.

К недостаткам Genesys-Pro можно отнести ограничения среды моделирования, которые требуют использования дополнительных средств для моделирования инструкций арифметики с плавающей точкой и доступа к памяти. Другим слабым местом является использование внешней эталонной модели для эмуляции исполнения команд. Это требует дополнительных трудозатрат на интеграцию и поддержку данной модели. Также непонятно насколько расширяемым является инструмент (возможность добавления новых техник генерации).

Инструмент FPGen

Для верификации модулей арифметики с плавающей точкой IBM Research был разработан инструмент FPGen [58]. Этот инструмент расширяет Genesys-Pro средствами генерации тестовых данных для покрытия всевозможных ситуаций в работе операций с плавающей точкой [59] (требования определены в стандарте IEEE 754 [60]).

Инструмент FPGen генерирует значения входных аргументов команд, разрешая специализированные ограничения. Ограничения определяют значения отдельных входных аргументов, отношения между значениями входных аргументов, результат промежуточных вычислений или конечный результат выполнения команды. Для описания ограничений используется язык XML. Генератор FPGen не привязан к какой-либо конкретной архитектуре, он генерирует наборы данных, которые сохраняются в специальном формате [60]. Эти данные используются Genesys-Pro при генерации тестовых программ, использующих операции арифметики с плавающей точкой. В процессе генерации Genesys-Pro комбинирует результаты решения ограничений, которые он разрешает самостоятельно и которые он разрешает при помощи FPGen.

Инструмент DeepTrans

Еще одно известное расширение Genesys-Pro – это инструмент DeepTrans [62], предназначенный для тестирования механизмов трансляции адресов. Этот инструмент использует спецификации подсистемы памяти, разработанные на специализированном языке моделирования. Данный язык позволяет представить процесс преобразования адреса в виде ориентированного ациклического графа, вершины которого соответствуют стадиям процесса, а дуги — переходам между стадиями. Множество путей от истока до стока задает конкретные варианты преобразования адреса и соответствует тестовым ситуациям. Тестовые ситуации представляются в виде ограничений, на которые можно ссылаться из тестовых шаблонов. За обработку шаблонов отвечает инструмент Genesys-Pro, который разрешает ограничения и трансформирует результаты в последовательности команд. К достоинствам DeepTrans следует отнести развитый язык моделирования механизмов трансляции. Он позволяет достаточно быстро осуществить настройку инструмента для тестирования механизмов трансляции адресов произвольного микропроцессора. Известный недостаток состоит в том, что не поддерживается автоматическое извлечение зависимостей между командами (конфликтов использования устройств); их приходится дополнительно указывать в тестовых шаблонах.

1.4.4 Разработки ИСП РАН

Исследования в области генерации тестовых программ для микропроцессоров ведутся в ИСП РАН с середины 2000-х годов [41]. В рамках этих исследований был разработан прототип инструмента MicroTESK (Microprocessor TEsting and Specification Tool Kit) версии 1.0. Данный инструмент использует *модели микропроцессора* и *тестовые шаблоны*, разработанные на языке Java, и позволяет сгенерировать тесты для широкого спектра микропроцессорных архитектур при помощи случайных и комбинаторных техник генерации, а также техник, основанных на разрешении ограничений. Модель микропроцессора включает в себя программный

эмулятор, который позволяет отслеживать состояние микропроцессора в процессе генерации.

Модель микропроцессора задает семантику команд и структуру подсистемы управления памятью. Тестовые шаблоны позволяют описывать сценарии в терминах тестовых ситуаций, связанных с работой отдельных команд. При этом в тестовом шаблоне задаются используемые команды, их порядок и связанные с ними ситуации. Также инструмент позволяет строить тесты для тестирования работы *модуля предсказания переходов (BPU, branch prediction unit)* путем перебора возможных трасс выполнения в структуре переходов, заданной шаблоном [43]. Тесты для *подсистемы управления памятью (MMU, memory management unit)* описаются в виде коротких последовательностей команд (2-3 команды), для которых строятся комбинации возможных сценариев обработки обращений к памяти с учетом зависимостей (допустимых вариантов совместного использования буферов памяти) [63].

К достоинствам инструмента можно отнести: (1) возможность поддержки различных архитектур, (2) отсутствие зависимостей от внешних эмуляторов, (3) поддержку генерации нацеленных тестов для BPU и MMU. Главным недостатком инструмента является использование языка Java для моделирования и описания тестовых шаблонов. Это усложняет конфигурирование инструмента, т.к. требует знания языка Java и деталей реализации библиотек MicroTESK. При этом эмулятор и модель покрытия необходимо описывать отдельно. Также отсутствует возможность автоматического извлечения информации из описаний – все свойства микропроцессора приходится задавать в явном виде. Таким образом, разработка модели является трудноосуществимой задачей без тесного взаимодействия с разработчиками инструмента. Еще одна проблема, связанная с использованием языка Java, – это то, что обновления библиотек, которые неизбежны в процессе эволюции инструмента, могут повлечь за собой изменения в уже созданных моделях и шаблонах. Другой недостаток заключается в том, что в инструмент не заложена возможность интеграции

новых методов генерации и их совместного использования. Например, нет возможности сгенерировать тестовую программу, решающую задачи тестирования BPU и MMU, на основе единого тестового шаблона.

1.4.5 Другие разработки

Среда RDG

В компании Samsung разработана среда RDG (Random Diagnostics Generator) [64], предназначенная для случайной генерации тестовых программ для реконфигурируемых микропроцессоров (Samsung Reconfigurable Processor, SRP). Эта среда использует в качестве входных данных описание синтаксиса команд тестируемого микропроцессора и тестовые шаблоны на языке C++. Тестовые шаблоны задают, какие команды будут использованы в тестовой программе, и описывают ограничения, накладываемые на входные значения этих команд. Среда RDG не владеет информацией о семантике команд и не осуществляет исполнение тестовых программ с целью предсказания результатов. Такой подход был выбран из-за особенностей тестирования реконфигурируемых микропроцессоров (система команд зависит от конфигурации и их реализация может отличаться). Кроме того это позволяет с минимальными усилиями добавлять поддержку новых команд и обеспечить высокую скорость генерации. Недостаток такого подхода в том, что для генерации нацеленных тестов требуется описывать ограничения вручную и отсутствует возможность отслеживать состояние микропроцессора. Стоит сказать, что инструмент RDG позволяет эффективно решать задачу тестирования реконфигурируемых микропроцессоров Samsung, но он не является универсальным. Поддержка новых микропроцессорных архитектур и техник генерации в него не заложена.

Генератор MA^{2}TG

Исследования в области генерации тестовых программ для функциональной верификации микропроцессоров проводились в Оборонном научно-техническом университете Китая (National University of Defense

Technology) [65]. В рамках этих исследований был разработан прототип генератора тестовых программ MA^{2}TG. Данный генератор применим для различных архитектур и поддерживает случайную генерацию и генерацию на основе ограничений. В качестве входных данных для генератора MA^{2}TG используются спецификации на языке EXPRESSION [66], описывающие архитектуру тестируемого микропроцессора, и шаблоны на специализированном языке, формулирующие задачи генерации. Инструмент транслирует входные в файлы в программу-генератор на языке C++, которая генерирует тесты. В процессе генерации исполнения построенных последовательностей команд не осуществляется.

Спецификации на языке EXPRESSION содержат информацию о структуре и поведенческих свойствах микропроцессора, а также о связи между ними. Для генерации тестовых программ в первую очередь необходима информация о поддерживаемых командах, которая включает текстовый и бинарный формат команд, списки их операндов и семантику команд (условия возникновения тестовых ситуаций). MA^{2}TG строит на основе спецификаций библиотеку шаблонов команд (Instruction Template Library, ITL), которая используется при генерации. Каждая команда описывается классом на языке C++, который содержит методы для ее печати, получения списка аргументов и доступа к ассоциированным с ней ограничениям. Использование формальных спецификаций позволяет относительно просто сконфигурировать инструмент для тестирования микропроцессора с новой архитектурой. Однако подход MA^{2}TG имеет некоторые недостатки. Информация о структуре микропроцессора, которая описывается в спецификациях на языке EXPRESSION, чаще всего не требуется для генерации тестовых программ. Эта информация может быть достаточно объемной, и ее специфицирование требует дополнительных трудозатрат. Кроме того она может быть недоступна на ранних стадиях работы над проектом. А в случаях, когда требуется тестирование микропроцессоров, по-разному реализующих одну и ту же архитектуру, она может препятствовать повторному использованию

спецификаций. Все это увеличивает трудоемкость разработки и поддержки спецификаций. Использование более простого языка, описывающего только поведенческие свойства, могло бы упростить эту задачу. Также, следует заметить, что MA^{2}TG не строит эталонный эмулятор на основе спецификаций, хотя язык EXPRESSION предоставляет для этого достаточное количество информации. Эталонный эмулятор был бы полезен для проверки корректности построенных программ, создания встроенных проверок и разрешения ограничений.

Задачи генерации описываются на специализированном языке. Такие описания называют ограничениями (constraints), хотя, по сути, они являются вариантом тестовых шаблонов. На их основе MA^{2}TG создает программы на языке C++, которые при помощи ITL и внешних библиотек разрешения ограничений, генерируют тесты. Язык описания тестовых шаблонов позволяет задавать следующие параметры: (1) используемые команды, их количество, порядок и вероятность появления; (2) ограничения на значения операндов команд; (3) зависимости по операндам между командами. Данный язык ориентирован на случайную выборку команд и разрешения ограничений для выбранных команд. Это позволяет быстро создавать большое количество небольших тестов для верификации определенных команд. Однако он не позволяет описывать сценарии со сложной структурой переходов. Также не предоставляется возможностей для описания многопоточных сценариев. Т.к. инструмент не осуществляет исполнение команд на эталонной модели, не поддерживается создание встроенных проверок. Кроме того отсутствие информации о состоянии микропроцессора усложняет решение ограничений. Из описания инструмента неясно, какие типы ограничений он поддерживает и насколько сложно в него добавить поддержку новых типов.

Инструмент MA^{2}TG реализован в виде прототипа. Несмотря на то, что его подход имеет ряд преимуществ, его функциональные возможности ограничены. Остается непонятно насколько он соответствует требованиям, предъявляемым инструментам, используемым в промышленных проектах. Из

очевидных недостатков можно перечислить излишне детальные спецификации, отсутствие контроля состояния микропроцессора в процессе генерации, ограниченные возможности языка тестовых шаблонов и ограниченный набор поддерживаемых техник генерации.

Исследования Университета Флориды и Калифорнийского университета в Ирвайне

В Университете Флориды (UFL) и Калифорнийском университет в Ирвайне (UCI) был разработан метод генерации тестовых программ, нацеленных на проверку корректности работы конвейера команд микропроцессора [37]. Данный метод использует спецификации на языке EXPRESSION [66], на основе которых строится модель, описывающая архитектуру микропроцессора в виде графа. Кроме этого разрабатывается модель ошибок, которая описывает типичные ошибки проектирования. На основе модели ошибок для модели микропроцессора строятся формулы, которые задают условия возникновения конкретных ошибок для данного микропроцессора. Для этих формул при помощи инструмента SMV (Symbolic Model Verifier) [66], который использует метод проверки моделей, стоятся тестовые примеры (контрпримеры для отрицания формул), на основе которых генерируются тестовые программы. По мнению авторов, метод не масштабируется на сложные микропроцессоры, поэтому предлагается дополнительно использовать тестовые шаблоны. Они создаются вручную и содержат описания цепочек команд, которые вызывают определенные ситуации в поведении микропроцессора (прежде всего, конвейерные конфликты). К достоинствам данного метода можно отнести то, что он применим для различных типов микропроцессоров и позволяет обеспечить покрытие ситуаций в работе конвейера команд, используя минимальное количество тестов. Главный недостаток метода – сложность разработки детальных спецификаций и модели ошибок. Следует также отметить, что данные исследования носили академический характер и на их основе не были

разработаны инструменты, которые могли бы использоваться в промышленных проектах.

Инструмент μGP

Исследователями Туринского политехнического университета (Politecnico di Torino) был предложен интересный подход к генерации тестовых программ, основанный на использовании генетических алгоритмов [68, 69, 70]. Данный подход был реализован в прототипе инструмента μGP. Основная идея подхода состоит в следующем. Для генератора тестовых программ создается библиотека инструкций, которая описывает синтаксис языка ассемблера целевого микропроцессора. Задачи генерации описываются в виде ациклического графа, который описывает поток выполнения тестовой программы. Каждая вершина графа содержит ссылку на описание команды в библиотеке команд и значения ее операндов. Библиотека команд и задачи генерации описываются на языке XML. Генерация тестовых программ осуществляется путем мутации структуры графа и значений операндов команд внутри отдельных вершин. Генератор пытается построить тестовую программу, которое обеспечит наилучшее покрытие для заданной метрики. Значение метрики получается путем выполнения построенных программ на программном эмуляторе RTL-модели.

Достоинством данного подхода является его гибкость и универсальность. Он позволяет достичь высокого уровня тестового покрытия для различных типов микропроцессоров, используя различные метрики покрытия. Главный недостаток – высокая вычислительная сложность, из-за которой скорость генерации получается достаточно низкой. Инструмент μGP реализует только один метод генерации и не предусматривает поддержку новых методов. Еще один возможный недостаток – это то, что входные форматы конфигурационных файлов не являются интуитивно понятными.

1.5 Выводы

Для всех рассмотренных в этой главе инструментов, независимо от поддерживаемых ими техник, можно выделить два свойства:

(1) *реконфигурируемость* и (2) *расширяемость*. Оба этих свойства показывают, какое количество усилий требуется для адаптации инструмента к решению новых задач. Под реконфигурируемость понимается возможность поддержки новых микропроцессорных архитектур, а под расширяемостью – возможность интеграции компонентов, реализующих различные техники генерации.

Другим важным свойством является *контролируемость генерации*, возможность отслеживать текущее состояние микропроцессора в процессе генерации путем исполнения генерируемых последовательностей команд на эталонной модели, в качестве которой выступает программный эмулятор. Это позволяет гарантировать корректность построенных программ, создавать тесты со встроенными проверками и применять техники генерации, использующие информацию о текущем состоянии микропроцессора. Эталонные модели разрабатываются отдельно от инструмента генерации и интегрируются в него при помощи специальных библиотек. Однако, по сути, они являются частью конфигурации инструмента т.к. используют знание о синтаксисе и семантике команд для их интерпретации.

Основным свойством, характеризующим техники генерации, является *целенаправленность тестирования*. Техники, основанные на случайной генерации, не позволяют систематическим образом обеспечить покрытие функциональных требований. Современные микропроцессоры с конвейерной архитектурой и сложной организацией памяти имеют огромное пространство возможных состояний, которые с малой вероятностью будут достигнуты при использовании случайных тестов. Поэтому для создания тестовых программ, нацеленных на проверку конкретных ситуаций или классов ситуаций, требуются специализированные техники, основанные на разрешении ограничений и проверке моделей.

Еще одним важным свойством современных инструментов верификации является возможность построения тестовых программ для многоядерных микропроцессоров. На базовом уровне она присутствует практически во всех инструментах, т.к. для этого достаточно включить в построенные программы

команды, обеспечивающие разветвление потока выполнения на несколько ядер. Однако техники нацеленной генерации, использующие информацию о текущем состоянии микропроцессора, должны учитывать тот факт, что каждое из ядер имеет свое состояние. Кроме того, инструменты должны обеспечивать покрытие ситуаций, связанных с параллельным выполнением программ.

Таблица 4 сравнивает функциональные возможности рассмотренных ранее инструментов по перечисленным критериям.

Таблица 4. Сравнение возможностей рассмотренных инструментов генерации

Инструмент	Реконфигурируемость	Расширяемость	Контролируемость генерации	Целенаправленность тестирования	Поддержка многоядерности
MicroTESK 1.0	Да	Нет	Да	Высокая	Нет
INTEG	Нет	Нет	Нет	Средняя	Нет
RIS	Нет	Нет	Нет	Средняя	Да
RAVEN	Да	Нет	Да	Средняя	Да
Genesys-Pro, FPGen, DeepTrans	Да	Да (частично)	Да	Высокая	Да
RDG	Да	Нет	Нет	Низкая	Нет
MA^2TG	Да	Нет	Нет	Высокая	Нет
UFL/UCI	Да	Нет	Нет	Высокая	Нет
μGP	Да	Нет	Нет	Средняя	Нет

Как можно заметить, большинство инструментов предназначено для решения конкретных задач и не предполагают расширяемость. Это объясняется тем, что коммерческие компании занимаются верификацией выпускаемых ими микропроцессоров и создают инструменты для решения текущих задач верификации, а исследовательские институты изобретают новые техники верификации и создают инструменты для их апробации. При этом задача интеграции техник, которые не используются в данный момент, не ставится.

Ситуация с реконфигурируемостью несколько сложнее. Несмотря на то, что некоторые инструменты позволяют конфигурирование под различные архитектуры, интеграция внешней эталонной модели сопряжена с трудностями. Большинство из перечисленных реконфигурируемых инструментов не

используют эталонные модели и не осуществляют контроль корректности построенных программ. Однако даже без необходимости интеграции эталонной модели создание пользовательских конфигураций может иметь значительную трудоемкость.

Целенаправленность тестирования и поддержка многоядерности характеризуют техники генерации и зависят от области применения инструмента. При этом они в той или иной степени связаны с расширяемостью и реконфигурируемость: более гибкие инструменты реализуют большее количество техник генерации и предоставляют больше возможностей.

Т.к. применимость отдельных инструментов ограничена, актуальной задачей является создание инструмента, который позволил бы объединить различные методы генерации, и был бы применим для различных типов микропроцессоров. Подобный инструмент должен сочетать все перечисленные выше свойства. Его характеристики могут быть сформулированы следующим образом.

Реконфигурируемость предполагает описание архитектуры тестируемого микропроцессора в простом, удобном и понятном инженеру-верификатору формате. Расширяемость предполагает, что описания, созданные для одних техник генерации, могут быть использованы другими техниками. Т.к. техники генерации требуют разное количество информации, то для более сложных техник будут требоваться описания, которые дополнят информацию, представленную в описаниях, используемых более простыми. Информацию об архитектуре микропроцессора, используемую рассмотренными инструментами, можно разделить на четыре уровня: (1) формат команд (текстовый и бинарный), (2) семантика команд, (3) организация подсистемы памяти и (4) организация конвейера команд. Один из возможных способов представления этой информации – применение формальных спецификаций. Это позволит использовать единое описание для извлечения ограничений, построения формальных моделей и эталонных эмуляторов. Для решения данной задачи требуется формальный язык, на котором можно в зависимости от потребности

описывать различные аспекты архитектуры микропроцессора. Другой вариант решения – использование нескольких языков, расширяющих возможности друг друга.

Еще одна важная характеристика – удобство описания тестовых сценариев. Используемый язык должен быть понятен инженерам-верификаторам. Желательно, чтобы он использовал абстракции цифровой аппаратуры. Он должен позволять описывать сложные тестовые сценарии, использующие комбинации различных техник генерации. Также требования реконфигурируемости и расширяемости предполагают адаптацию к новым архитектурам и добавление новых конструкций. Таким образом, язык должен позволять динамическое добавление и изменение поддерживаемых конструкций.

Глава 2. Автоматизация конструирования генераторов тестовых программ

В данной главе описывается предлагаемый метод автоматизации конструирования генераторов тестовых программ для микропроцессоров. Он предполагает использование формальных спецификаций в качестве источника информации об архитектуре тестируемого микропроцессора. Генерация тестовых программ будет осуществляться на основе шаблонов, разработанных на специальном языке, позволяющем описывать свойства программ в терминах формальных спецификаций и техник генерации, применяемых для удовлетворения заданных свойств. Расширяемая архитектура генераторов позволит интегрировать разные техники генерации. Схема использования инструмента конструирования генераторов тестовых программ, реализующего предлагаемый метод, и генераторов, построенных с его помощью, показана на рисунке 4.

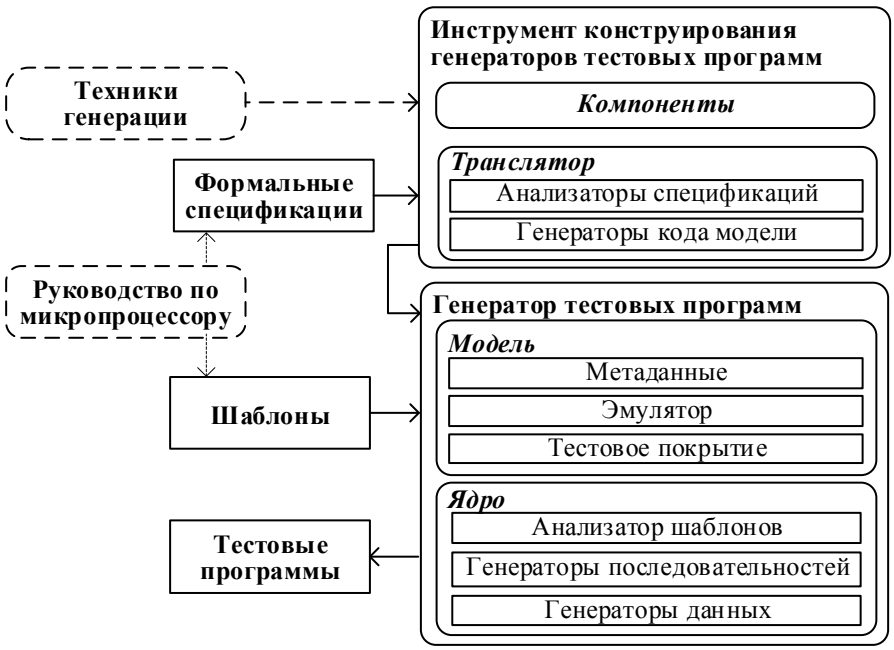


Рисунок 4. Инструмент, реализующий предлагаемый метод, и результат его работы

Глава организована следующим образом. В первом разделе описывается метод автоматизации конструирования генераторов тестовых программ на основе формальных спецификаций. Второй раздел посвящен языку описания шаблонов тестовых программ. В третьем разделе описывается архитектура конструируемых генераторов тестовых программ.

2.1 Метод автоматизации конструирования генераторов тестовых программ

Генераторы тестовых программ используют информацию трех типов: (1) описание архитектуры тестируемого микропроцессора; (2) свойства генерируемых тестовых программ; (3) знание о техниках генерации, применяемых для удовлетворения этих свойств. Для создания генератора, который поддерживал бы любые микропроцессорные архитектуры и позволял бы интегрировать в себе разные техники генерации, необходимо разделение между данными типами информации. При этом архитектура микропроцессора и техники генерации должны быть полностью независимыми друг от друга, а свойства тестовых программ формулироваться на их основе. В таком случае для поддержки новой архитектуры достаточно разработать ее описание, на основе которого можно будет автоматически сконструировать генератор тестовых программ.

Конструируемые генераторы тестовых программ будут состоять из двух частей: (1) *модели*, содержащей информацию об архитектуре микропроцессора, и (2) *ядра*, интегрирующего в себе архитектурно-независимые компоненты, реализующие различные техники генерации. При этом ядро является общим для всех конструируемых генераторов. Генерация будет осуществляться на основе шаблонов тестовых программ, задающих их свойства в терминах элементов архитектуры, описываемых моделью, и техник генерации, реализуемых ядром. Обработка тестовых шаблонов осуществляется ядром и состоит в применении той или иной техники генерации для удовлетворения того или иного свойства.

2.1.1 Использование формальных спецификаций

Для конструирования модели микропроцессора необходимо описание его архитектуры. Разработка подобных описаний будет осуществляться инженерами-верификаторами и не должна требовать знания языков программирования и понимания особенностей реализации конструируемых генераторов тестовых программ. Формат описаний должен быть интуитивно

понятен и построен на абстракциях цифровой аппаратуры. Так как полная информация об архитектуре микропроцессора содержится в руководстве по микропроцессору и другой проектной документации, логичным видится использование *формальных спецификаций*, основанных на этой документации.

Различные техники генерации требуют информацию об архитектуре микропроцессора разного уровня детализации. Практически всем техникам необходима информация о *системе команд* (их синтаксис и семантика). Кроме этого техникам, ориентированным на создание нацеленных тестов для проверки определенных аспектов функциональности, требуется знание об особенностях *микроархитектуры* (организация подсистемы памяти и конвейера команд). Т.к. информация об особенностях микроархитектуры используется лишь некоторыми техниками, для каждой из которых она может иметь свой формат, основная роль отводится спецификациям системы команд. Спецификации микроархитектуры будут служить в качестве дополнительного источника знания. При этом они не должны дублировать уже имеющиеся описания.

2.1.2 Описание архитектуры микропроцессора на языке nML

Для создания формальных спецификаций хорошо подходят *языки описания архитектуры* (ADL-языки) [29, 71]. Они позволяют описывать поведенческие и структурные свойства микропроцессора, абстрагируясь от деталей реализации. Эти языки часто используются для моделирования в процессе проектирования микропроцессора. В этом случае созданные описания могут быть повторно использованы для конструирования генераторов тестовых программ.

По характеру описываемых свойств ADL-языки принято разделять на две группы: *структурные* и *поведенческие*. Языки первой группы ориентированы на специфицирование микроархитектуры (примером такого языка является MIMOLA). Вторая группа концентрируется на описании системы команд микропроцессора (к этой категории можно отнести ISDL и nML). Кроме того, существуют *смешанные языки*, сочетающие возможности языков обеих групп

(например, LISA и EXPRESSION). Для спецификации архитектуры наиболее подходят поведенческие ADL-языки, позволяющие явно описывать синтаксис и семантику команд. Из нескольких поведенческих и смешанных ADL-языков, таких как LISA, EXPRESSION, ISDL и nML, был выбран последний. Данный язык обладает интуитивно понятным синтаксисом, имеет доступную документацию и используется в нескольких зарубежных университетах и коммерческих компаниях.

Язык nML [72] был разработан в начале 1990-х гг. в Берлинском техническом университете (Technische Universität Berlin) и изначально предназначался для создания ассемблеров, дизассемблеров, программных эмуляторов и настраиваемых компиляторов. В середине 1990-х гг. в компании Cadence проводились исследования по созданию на основе nML-спецификаций программного инструментария, включавшего все перечисленные средства, для совместной разработки программного и аппаратного обеспечения [73]. С конца 1990-х гг. nML активно используется Target Compiler Technologies [74] для решения таких задач, как конструирование эмуляторов, компиляторов и RTL-моделей. Эта компания расширила язык nML средствами моделирования конвейера [29]. Другая версия языка nML была разработана в Индийском технологическом институте Каннур (Indian Institute of Technology Kanpur) и получила название Sim-nML [75]. В язык были добавлены средства описания ресурсов (модулей) микропроцессора и модели их использования при выполнении инструкций. В Исследовательском институте информатики в Тулузе (Institut de Recherche en Informatique de Toulouse) для создания эмуляторов микропроцессоров применяется язык nMP, основанный на Sim-nML [76, 77].

С целью упрощения создания спецификаций было решено использовать первоначальную редакцию языка nML, возможности которой ограничиваются описанием синтаксиса и семантики команд. Для специфицирования микроархитектуры предполагается использование дополнительных формальных языков.

Рассмотрим более подробно возможности языка nML. Данный язык основан на атрибутной грамматике и позволяет описать «программистскую» модель системы команд микропроцессора. Спецификации на nML включают описания следующих сущностей: типы данных, константы, регистры, режимы адресации, команды, память и временные переменные.

Типы данных позволяют описывать знаковые и беззнаковые целые числа произвольной длины и числа с плавающей точкой в формате, описанном в стандарте IEEE 754 [60]. В примере 1 приведены объявления типов данных из спецификации архитектуры MIPS64 [47].

Пример 1. Типы данных MIPS64

```
01:   type BIT = card(1)                          Беззнаковые целочисленные
02:   type BYTE = card(8)                         типы размером 1, 8, 16, 32, 64 и
03:   type HWORD = card(16)                       128 бит
04:   type WORD = card(32)
05:   type DWORD = card(64)
06:   type QWORD = card(128)
07:
08:   type SHORT = int(16)                        Знаковые целочисленные типы
09:   type INT = int(32)                          размером 16, 32 и 64 бита
10:   type LONG = int(64)
11:
12:   type FLOAT32 = float(23, 8)                 Типы с плавающей точкой
13:   type FLOAT64 = float(52, 11)                одинарной и двойной точности
```

Константы задаются как целые числа неопределенного размера, которые приводятся к нужному типу в зависимости от контекста.

Регистровые файлы описываются в виде массивов переменных, доступ к которым осуществляется при помощи *режимов адресации*. Это специальные абстракции, которые инкапсулируют логику обращения к элементам массива, а также задают ассемблерный (атрибут *syntax*) и двоичный (атрибут *image*) форматы регистров. Например, регистры общего назначения микропроцессора MIPS64 и режим адресации для доступа к ним можно описать следующим образом:

Пример 2. Регистры общего назначения MIPS64

```
01:   reg GPR [32, WORD]                          Регистровый файл
02:   mode R (i: card(5)) = GPR[i]                Режим адресации
03:     syntax = format("$%d", i)                 Ассемблерный формат
04:     image  = format("%5b", i)                 Двоичный формат
```

Физическая память микропроцессора описывается так же, как и регистры, в виде массива. В приведенном ниже фрагменте спецификации массив *M*

интерпретируется как физическая память, состоящая из 2^{30} 32-битных слов (или 2^{32} байт). nML не предоставляет средств описания механизмов виртуальной памяти, таких как трансляция адресов и кэширование. При обращении к памяти предполагается, что физический адрес равен виртуальному адресу.

Пример 3. Массив физической памяти

```
01:    mem M [2 ** 30, WORD]                    Физическая память
```

Помимо регистров и памяти, в nML можно объявлять временные переменные для хранения промежуточных результатов вычислений. Например, ниже объявлена переменная 33-битная целочисленная *temp33*, используемая для хранения результатов арифметических операций, при которых возможно переполнение.

Пример 4. Временная переменная

```
01:    var temp33[int(33)]                      Временная переменная
```

Команды специфицируются в виде иерархической структуры *операций*, описывающих отдельные части логики. Логика, общая для всех команд (например, обновление значения счетчика команд), помещается в корневой узел. Операции, уникально характеризующие отдельные команды, помещаются в терминальных узлах. Эти операции принимают в качестве параметров режимы адресации, которые описывают входные и выходные параметры команды. Операции содержат три атрибута: *action*, *syntax* и *image*. Первый описывает логику работы команд в терминах присваивания значений регистров, а остальные два задают ассемблерный и двоичный форматы соответственно. Например, спецификация команды *ADD* микропроцессора MIPS64 выглядит следующим образом:

Пример 5. Спецификация команды ADD архитектуры MIPS64 на языке nML

```
01:   op add (rd: R, rs: R, rt: R)
02:     syntax = format("add %s, %s, %s", rd.syntax, rs.syntax, rt.syntax)
03:     image  = format("000000%5s%5s%5s00000100000", rs.image, rt.image, rd.image)
04:     action  = {
05:       if sign_extend(WORD, rs<31>) != rs<63..32> ||
06:          sign_extend(WORD, rt<31>) != rt<63..32> then
07:         unpredicted;
08:       endif;
09:       temp33 = rs<31>::rs<31..0> + rt<31>::rt<31..0>;
10:       if temp33<32> != temp33<31> then
11:         exception("IntegerOverflow");
12:       else
13:         rd = sign_extend(DWORD, temp33<31..0>);
14:       endif;
15:     }
```

Данная спецификация разработана на основе описания команды в руководстве по микропроцессору MIPS64 [47], которое имеет следующий вид:

Пример 6. Описание семантики команды ADD из руководства по архитектуре MIPS64

```
01:   if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
02:     UNPREDICTABLE
03:   endif
04:   temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
05:   if temp32 ≠ temp31 then
06:     SignalException(IntegerOverflow)
07:   else
08:     GPR[rd] ← sign_extend(temp31..0)
09:   endif
```

nML позволяет объединять режимы адресации и операции в *группы*. Использование группы в объявлении означает, что в этом месте применим любой из ее элементов. В примере 7 показаны группы арифметических операций над 32-битными числами из системы команд MIPS64.

Пример 7. Группы арифметических операций над 32-битными числами системы команд MIPS64

```
01:   op Mips32ArithmeticRR     = clo | clz | div | divu | madd | maddu | multu | mult | msub | msubu
02:   op Mips32ArithmeticRRR  = add | addu | div_reg | mod_reg | divu_reg | modu_reg |
03:                                sub | subu | mul | mul_reg | muh_reg | muhu_reg | mulu_reg
04:   op Mips32ArithmeticRRI   = addi | addiu | aui
05:   op Mips32ArithmeticOp     = Mips32ArithmeticRRR | Mips32ArithmeticRR | Mips32ArithmeticRRI
```

Логика обновления счетчика команд, общая для всех команд, описывается как отдельная операция, параметризованная группой, которая включает в себя операции, описывающие логику отдельных команд. В примере 8 показана спецификация операции *instruction*, которая описывает логику управления счетчиком команд микропроцессора MIPS64 со слотом задержки, и используемых ею переменных.

Пример 8. Описание логики обновления счетчика команд, общей для всех команд архитектуры MIPS64

```
01:   let PC = "CIA"                                  Регистр CIA помечен как счетчик команд
02:   reg CIA [DWORD]                                 Регистр, используемый как счетчик команд
03:
04:   mem BRANCH [BIT]                                Переменные слота задержки
05:   mem NEXTPC [DWORD]
06:
07:   var is_delay_slot [BIT]                         Временные переменные
08:   var jump_address [DWORD]
09:
10:   op instruction (operation: Op)                  Группа Op включает все команды MIPS64
11:     syntax = operation.syntax                     Ассемблерный формат наследуется
12:     image  = operation.image                      Бинарный формат наследуется
13:     action = {
14:       is_delay_slot = BRANCH;                     Обновление значений переменных
15:       jump_address = NEXTPC;
16:       BRANCH = 0;
17:       NEXTPC = 0;
18:
19:       operation.action;                           Исполнение команды
20:       if is_delay_slot == 1 then                  Обновление счетчика команд
21:         CIA = jump_address;                       Переход на произвольный адрес
22:       else
23:         CIA = CIA + 4;                            Переход на следующую команду
24:       endif;
25:     }
```

Как можно увидеть, операции, реализующие логику отдельных команд, вызываются общей частью. После этого происходит обновление счетчика команд. Счетчик команд или инкрементируется или, если команда осуществляет переход, ему присваивается новый адрес. Адрес перехода передается через глобальную переменную. В примере 9 показана спецификация команды условного перехода *BEQ* системы команд MIPS64.

Пример 9. Спецификация команды BEQ архитектуры MIPS64 на языке nML

```
01:   op beq (rs: R, rt: R, offset: SHORT)
02:     syntax = format("beq %s, %s, %d", rs.syntax, rt.syntax, offset)
03:     image  = format("000100%5s%5s%16s", rs.image, rt.image, offset)
04:     action = {
05:       if rs == rt then
06:         BRANCH = 1;
07:         NEXTPC = CIA + (sign_extend(DWORD, offset) << 2);
08:       endif;
09:     }
```

Отдельного внимания заслуживают команды доступа к памяти. На языке nML обращения к памяти описываются как доступ к элементам массива физической памяти. При этом физический адрес считается равным виртуальному адресу. В примере 10 показана спецификация команды *LD* системы команд MIPS64.

Пример 10. Спецификация команды LD архитектуры MIPS64 на языке nML

```
01:    var temp_address[DWORD]
02:    op ld (rt: R, offset: SHORT, base: R)
03:      syntax = format("ld %s, %d(%s)", rt.syntax, offset, base.syntax)
04:      image  = format("110111%5s%5s%16s", base.image, rt.image, offset)
05:      action = {
06:        temp_address = base + sign_extend(DWORD, offset);
07:        rt = MEM[temp_address_load >> 3];
08:    }
```

2.1.3 Расширение возможностей языка nML

Для построения тестовых программ, нацеленных на проверку работы подсистемы управления памятью *(MMU, memory management unit)* микропроцессора, генератору информация о данной подсистеме. Для описания ее конфигурации был предложен специальный язык, получивший название MMUSL [2, 7]. Спецификации подсистемы MMU на данном языке включают в себя описания типов адресов, сегментов памяти, буферов памяти и управляющей логики обработки операций чтения и записи. Подробное описание языка MMUSL и техник генерации тестовых программ для подсистемы MMU на основе формальных спецификаций выходит за рамки данной работы. Однако необходимо пояснить каким образом описание системы команд на языке nML может быть дополнено описанием подсистемы MMU. Как показано в примере 10, на языке nML обращения к памяти специфицируются как доступ к массиву, описывающему физическую память (в данном примере он называется *MEM*). При этом физический адрес считается равным виртуальному адресу. Для расширения данной схемы можно разработать дополнительные спецификации на языке MMUSL, описывающие механизмы трансляции адресов и кэширования данных. Эти спецификации включают в себя описания логики обработки операций чтения и записи. Данные операции соответствуют операциям чтения и записи в массив памяти, объявленный в nML-спецификациях. Другими словами, при чтении или записи в массив *MEM* по виртуальному адресу будет задействована соответствующая логика, описанная в MMUSL-спецификациях, которая выполнит трансляцию адреса в физический и осуществит доступ к физической памяти через иерархию буферов.

2.1.4 Архитектура модели микропроцессора

Модель микропроцессора, построенная на основе формальных спецификаций, должна содержать в себе всю информацию, необходимую для генерации тестовых программ для данной архитектуры. При этом доступ к этой информации должен осуществляться независимо от моделируемой архитектуры. Информация об архитектуре микропроцессора требуется генератору тестовых программ для решения следующих задач:

- анализа шаблонов тестовых программ;
- генерации входных данных для команд, вызывающих определенные ситуации в их работе;
- исполнения команд на эталонном эмуляторе;
- печати команд в ассемблерном формате.

Таким образом, конструируемая модель будет включать в себя следующие части: (1) *метаданные*, описывающие сигнатуры поддерживаемых команд; (2) *эмулятор*, позволяющий исполнять и печатать команды микропроцессора; (3) *модель тестового покрытия*, содержащая каталог ситуаций, связанных с командами, и условия их возникновения. Метаданные будут использоваться ядром генератора при анализе шаблонов тестовых программ для распознавания описываемых команд. Запросы к модели на исполнение, печать и получение информации о связанных с командами ситуациях будут также описываться в терминах метаданных. Это позволит организовать ядро генератора в виде набора компонентов, реализующих архитектурно-независимые техники генерации.

Метаданные представляют собой таблицы из поддерживаемых моделью команд, режимов адресации и регистровых файлов. Команды характеризуются именем, списком операндов, набором атрибутов, описывающих их поведенческие свойства и списком связанных с ними тестовых ситуаций. Режима адресации имеют имя, список аргументов и набор поведенческих свойств. Регистровые файлы характеризуются именем, типом данных и количеством элементов.

Эмулятор включает в себя элементы хранения данных (регистры и память) и команды, доступ к которым предоставляется через архитектурно-независимые интерфейсы. Эмулятор позволяет поддерживать несколько копий элементов хранения данных. Это необходимо для моделирования работы многоядерных микропроцессоров и для создания временных контекстов исполнения (для случаев, когда необходимо вернуться к некоторому начальному состоянию). Эмулятор может расширяться компонентом, отвечающим за моделирования работы подсистемы памяти. Исполнение команд на эмуляторе осуществляется пошагово. В любой момент времени можно получить доступ к информации о его текущем состоянии. Схема работы с эмулятором выглядит следующим образом:

- на основе запросов в терминах метаданных, создаются объекты, описывающие вызовы конкретных команд;
- эти команды можно исполнять на одном из ядер эмулятора или печатать в ассемблерном формате;
- команды можно поместить в память эмулятора и считать оттуда;
- эмулятор предоставляет доступ к регистрам на чтение и запись (соответствующие запросы описываются в терминах метаданных);
- для исполнения команд нужно поместить их в память и записать стартовый адрес в регистр, хранящий счетчик команд;
- после этого можно извлекать команды по одной, исполнять их и при этом контролировать состояние регистров и памяти;
- исполнение команд можно осуществлять на разных ядрах и использовать при этом постоянный или временный контексты.

Модель покрытия включает в себя описания тестовых ситуаций в различных форматах. Ситуации, связанные с системой команд, описываются в виде ограничений, выраженных в виде формул, где в качестве переменных выступают аргументы команд. Ситуации, связанные с подсистемой управления памятью, задаются в виде путей на графе, описывающем логику ее работы.

Ситуации различных типов не зависят друг от друга и обрабатываются различными компонентами генератора тестовых программ.

2.2 Язык описания шаблонов тестовых программ

Генерация *тестовых программ* осуществляется на основе шаблонов, которые задают их свойства в абстрактном виде. Для описания шаблонов предложен специализированный язык, который позволяет создавать шаблоны для любых микропроцессорных архитектур и применять расширяемый набор техник генерации для достижения данных свойств. Далее концепции тестовых программ и шаблонов тестовых программ рассмотрены более подробно.

2.2.1 Структура тестовых программ

Тестовая программа представляет собой набор *тестовых воздействий*, предназначенных для проверки определенных аспектов функциональности микропроцессора. Каждое тестовое воздействие задается в виде последовательности команд, выполнение которых приводит к возникновению некоторых событий в работе микропроцессора, называемых *тестовыми ситуациями*. Как правило, для этого требуется, чтобы микропроцессор находился в определенном начальном состоянии. Поэтому тестовые воздействия предваряются *инициализирующим кодом*, который содержит команды, записывающие необходимые значения в регистры и память. После выполнения тестового воздействия может осуществляться проверка корректности полученных результатов. Для этого используются *встроенные проверки (self-checks)*, последовательности команд, сравнивающие значения в регистрах и памяти с эталонными. Тестовое воздействие вместе с инициализирующим кодом и встроенными проверками называют *тестовым примером (test case)*. В простейшем случае тестовые примеры не зависят друг от друга и выполняются последовательно. Однако для решения более сложных задач тестирования может потребоваться выполнение тестовых примеров в произвольном порядке, циклическое выполнение или выполнение на разных ядрах микропроцессора. Для этого в тестовые программы добавляется *код*

диспетчеризации (dispatching code), который содержит команды, осуществляющие переход в другие части программы. При этом тестовые примеры остаются самодостаточными и имеют единственную точку входа, а переход осуществляется между секциями диспетчеризации. Помимо описанных сущностей, каждая тестовая программа включает в себя *пролог*, который осуществляет начальную инициализацию микропроцессора, и *эпилог*, который завершает работу программы. Таким образом, структуру тестовых программ можно описать формулой $П = П_{start} \cdot \{\langle \pi_{dispatch} \rangle \cdot \langle \pi_{start}, x_i, \pi_{stop} \rangle\}_{i=1,n} \cdot П_{stop}$, где:

- $П_{start}$ – пролог тестовой программы;
- $\pi_{dispatch}$ – код диспетчеризации;
- $\langle \pi_{start}, x_i, \pi_{stop} \rangle$ – тестовый пример, включающий в себя:
    - π_{start} – инициализирующий код;
    - x_i – тестовое воздействие;
    - π_{stop} – встроенные проверки;
- $П_{stop}$ – эпилог тестовой программы;
- n – число тестовых примеров в программе.

2.2.2 Описываемые свойства тестовых программ

Шаблоны тестовых программ представляют собой абстрактные описания их свойств. При этом на основе одного шаблона может быть построено множество тестовых программ, удовлетворяющих описанным свойствам. Это позволяет обеспечить покрытие всевозможных комбинаций ситуаций в работе микропроцессора.

Свойства тестовых программ, описываемые шаблонами, можно разделить на два типа: (1) *структурные* и (2) *поведенческие*. Первые задают способы построения последовательностей команд, составляющих тестовые воздействия, а вторые – ситуации, которые должны возникнуть в процессе их исполнения. Также к структурным свойствам относятся описания пролога, эпилога и кода диспетчеризации. Эти части программ описываются в виде фиксированных

последовательностей команд, так как выполняемые ими действия заранее определены.

Рассмотрим более подробно свойства тестовых примеров. Они представляют собой последовательности команд, которые должны обеспечивать покрытие определенных ситуаций в работе микропроцессора. Ситуации могут быть связаны как с отдельными командами, так и с последовательностью в целом. Для достижения целевых ситуаций может потребоваться одновременное выполнение нескольких условий различного типа. Это связано с наличием *зависимостей* между командами, которые возникают при совместном использовании регистров, памяти и конвейера команд. Для удовлетворения необходимых условий последовательности команд должны быть построены определенным образом. Важно заметить, что гарантировать выполнение всех необходимых условий часто не представляется возможным ввиду сложности реализации и высоких вычислительных затрат. Кроме того т.к. часто физически невозможно предусмотреть все возможные зависимости, существует риск, что некоторые важные тестовые ситуации будут пропущены. Поэтому применяется подход, при котором тестовые воздействия описываются путем комбинирования последовательностей команд, нацеленных на выполнение отдельных условий. При этом создается большое количество тестовых примеров, некоторые из которых не соответствуют требуемым условиям или избыточны, но в целом это дает более достаточно высокое качество покрытия при приемлемой вычислительно сложности. Кроме того такой подход позволяет обеспечить покрытие многих ситуаций в работе микропроцессора, которые трудно предусмотреть и формализовать.

Таким образом, описания тестовых примеров в шаблонах тестовых программ должны задавать свойства следующих типов для генерируемых последовательностей команд:

- способы построения последовательностей команд;
- способы выбора регистров, используемых командами;
- способы комбинирования тестовых ситуаций;

- способы генерации входных данных для команд;
- способы построения инициализирующего кода;
- способы построения встроенных проверок.

Для удовлетворения данных свойств применяются разнообразные *техники генерации*, реализуемые генератором.

2.2.3 Концепция языка описания шаблонов тестовых программ

Для создания тестовых шаблонов необходим специализированный *предметно-ориентированный язык (domain-specific language)*. Он должен удовлетворять следующим основным требованиям: (1) быть простым и понятным инженерам-верификаторам; (2) быть применим для любых микропроцессорных архитектур; (3) позволять использовать расширяемый набор техник генерации; (4) быть пригодным для использования в промышленных проектах. На основе данных требований и анализа подходов, применяемых на практике, сформулированы следующие требования к реализации такого языка:

- За основу должен быть взят популярный язык высокого уровня. Здесь хорошо подходят сценарные языки такие, как Ruby [78], Python [79] и Perl [80], часто используемые инженерами-верификаторами для автоматизации различных задач.
- Формат, используемый для описания сущностей языка ассемблера (команды, данные и директивы), должен быть максимально приближен к используемому в настоящих ассемблерных программах.
- Языковые конструкции для описания сущностей языка ассемблера должны динамически настраиваться в соответствии с архитектурой тестируемого микропроцессора.
- Конструкции для задания используемых техник генерации должны быть независимыми от конкретных техник и применимы для любых из поддерживаемых техник.

- Должна поддерживаться возможность описания шаблонов, основанных на совместном использовании различных техник генерации.
- Язык должен позволять описывать отдельные задачи генерации в виде библиотек, на основе которых можно было бы создавать описания более сложных шаблонов.

После анализа различных возможных вариантов реализации *языка описания тестовых шаблонов*, было принято решение взять за основу язык программирования Ruby [78]. Данный язык позволит удовлетворить всем вышеперечисленным требованиям. Его основным преимуществом являются мощные средства *метапрограммирования* [81], которые значительно упрощают создание предметно-ориентированных языков. Язык поддерживает динамическое создание методов, определение блоков кода, интерпретируемых в особом контексте, и перегрузку операторов.

Таким образом, язык описания тестовых шаблонов наследует все возможности языка Ruby, расширяя его дополнительными конструкциями для описания свойств тестовых программ. Предоставляемые им конструкции можно разделить на *статические* и *динамические*. Статические конструкции используются для задания техник генерации, используемых для построения описанной шаблоном программы, и являются частью реализации ядра генератора. Динамические конструкции используются для описания сущностей ассемблера тестируемого микропроцессора и создаются динамически на основе информации, извлеченной из формальных спецификаций и описываемой метаданными, при помощи средств метапрограммирования языка Ruby.

Шаблоны тестовых программ представляют собой классы на языке Ruby, наследующие библиотечный базовый класс Template, который реализует конструкции описания свойств тестовых программ. Структура тестовых шаблонов показана в примере 11.

Пример 11. Структура тестовых шаблонов

```
01:    require ENV['TEMPLATE']
02:    class MyTemplate < Template
03:      def initialize
04:        super
05:        # Настройки генерации
06:      end
07:      def pre
08:        # Пролог тестовой программы
09:      end
10:      def post
11:        # Эпилог тестовой программы
12:      end
13:      def run
14:        # Тестовые примеры и код диспетчеризации
15:      end
16:    end
```

Шаблоны тестовых программ определяют четыре основных метода: (1) *initialize,* который задает настройки генератора; (2) *pre,* который содержит описание пролога тестовой программы; (3) *post,* который содержит описание эпилога тестовой программы; (4) *run,* который содержит описание тестовых примеров и кода диспетчеризации. Помимо перечисленных методов позволяется объявлять дополнительные методы, которые будут вызываться из основных методов. Это позволяет структурировать описания тестовых программ и обеспечить повторное использование кода. Общая стратегия повторного использования кода шаблонов такая же, как и для большинства языков высокого уровня. Ruby позволяет создавать базовые классы, которые могут содержать код, описывающий свойства, наследуемые несколькими шаблонами. Другой подход – создавать классы, предоставляющие утилитные методы, предназначенные для решения отдельных задач и используемые различными шаблонами.

Тестовые примеры описываются при помощи конструкций *block* и ее разновидностей. На основе отдельного блока может быть построено несколько тестовых примеров в зависимости от его структуры и используемых параметров. Также можно задавать сколько раз блок должен быть обработан. Это позволяет получить несколько различных тестовых примеров на основе последовательностей, использующих методы рандомизации. Таким образом описания тестовых примеров имеют следующий формат:

block(*attrs*) { *body* }.*run N*, где *attrs* – набор атрибутов, задающих способы построения последовательностей команд и их обработки, *body* – тело блока, содержащее команды и вложенные блоки, на основе которых строятся последовательности команд, и *N* – необязательный параметр, задающий сколько раз блок будет обработан (по умолчанию равен 1).

2.2.4 Описание последовательностей команд

Последовательности команд, составляющие тестовые примеры, описываются в виде *блоков*, которые можно определить рекурсивно. Таким образом, блок представляет собой или *элементарный блок* или упорядоченный набор *вложенных блоков*. Под элементарным блоком понимается отдельная команда. При этом неэлементарный блок может быть пустым. Блоки описываются при помощи конструкции *block {...}*. Ниже приведен пример вложенной структуры из блоков, использующих команды MIPS.

Пример 12. Блоки, описывающие последовательности команд

```
01:    block {                    Блок верхнего уровня
02:      block {                  Вложенный блок
03:        add t0, t1, t2         Элементарный блок
04:        add t3, t4, t5
05:      }
06:      block {
07:        sub t0, t1, t2
08:        sub t3, t4, t5
09:      }
10:      block {
11:        slt t0, t1, t2
12:        slt t3, t4, t5
13:      }
```

Каждый блок описывает одну или несколько последовательностей команд. Элементарный блок задает одну последовательность, состоящую из единственной команды. Неэлементарный блок задает набор последовательностей, построенных на основе последовательностей, описанных вложенными блоками. Существует три частных случая блоков: (1) *sequence (последовательность)*, (2) *atomic (атомарная последовательность)* и (3) *iterate (блок итерирования)*.

Блок типа *sequence* можно считать элементарным: он возвращает одну последовательность, состоящую из команд, указанных в блоке. Пример такого блока приведен ниже.

Пример 13. Блок, описывающий одну последовательность команд

```
01:    sequence {                                  Одна последовательность
02:      add t0, t1, t2
03:      add t3, t4, t5
04:      add t6, t7, t8
05:    }
```

Блок типа *atomic* является разновидностью блока типа *sequence*. Его итератор также возвращает одну единственную последовательность, состоящую из команд, указанных в блоке. Важное отличие заключается в том, что эта последовательность является неделимой. Т.е. она никогда не будет перемешана с другими командами.

Блок типа *iterate* предназначен для итерирования по последовательностям, которые возвращают вложенные блоки. Число возвращаемых блоком последовательностей равно общему числу всех последовательностей, возвращаемых вложенными блоками. Пример приведен ниже.

Пример 14. Блок, описывающий итерирование по вложенным элементам

```
01:    iterate {                                   Три последовательности из одной команды
02:      add t0, t1, t2
03:      add t3, t4, t5
04:      add t6, t7, t8
05:    }
```

Для неэлементарных блоков задаются атрибуты, которые определяют способ построения последовательностей на основе вложенных блоков. Каждый из атрибутов связан с той или иной техникой построения последовательностей, реализованной генератором. Набор техник и набор атрибутов являются расширяемыми. Пример неэлементарного блока, который описывает последовательности команд, построенные на основе вложенных блоков, приведен ниже.

Пример 15. Блок, описывающий последовательности, построенные на основе вложенных блоков

```
01:    block(                                                    Блок описывает последовательности команд,
02:      :combinator => 'diagonal',                              построенные на основе вложенных блоков:
03:      :permutator => 'trivial',                               {A11, B11, B12, C11, C12, C13},
04:      :compositor => 'catenation',                            {A21, B21, B22, C11, C12, C13},
05:      :rearranger  => 'trivial',                              {A31, B11, B12, C11, C12, C13}
06:      :obfuscator  => 'trivial') {
07:      iterate { A11; A21; A31 }                               Блок описывает: {A11}, {A21}, {A31}
08:      iterate { sequence { B11, B12 }; sequence { B21, B22 } } Блок описывает: {B11, B12}, {B21, B22}
09:      iterate { sequence { C11; C12; C13 } }                  Блок описывает: {C11, C12, C13}
10:    }
```

Техники построения последовательностей команд будут подробно описаны в разделе, посвященном генераторам тестовых программ.

2.2.5 Описание правил выбора регистров

Регистры, используемые командами тестового примера, могут задаваться жестко или выбираться в соответствии с определенной стратегией. Возможность динамического выбора регистров – важное требование для случайной и комбинаторной генерации. При динамическом выборе указывается имя регистрового массива и критерий выбора конкретного регистра. При выборе учитываются три критерия: (1) диапазон возможных значений для номера регистра; (2) история использования регистров; (3) применяемая стратегия выбора. По умолчанию диапазон номеров регистров определяется формальными спецификациями. Его можно изменить, указав подмножество, которое необходимо исключить, или подмножество, из которого необходимо выбрать. История использования регистров предоставляется средой генерации, и ее можно модифицировать при помощи специальных конструкций языка шаблонов. Стратегии выбора определяют, каким образом информация о диапазоне значений и использованных регистрах будет использована. Ниже приведен список поддерживаемых в настоящий момент стратегий, который при необходимости может быть расширен пользовательскими реализациями:

- *random* – выбрать случайный регистр;
- *free* – выбрать неиспользуемый в данном тестовом примере регистр;
- *used* – выбрать регистр, который уже используется;
- *try_free* – выбрать неиспользуемый регистр или, если такового не имеется, вернуть один из используемых.

Рассмотрим, как осуществляется управление выбором регистров в языке тестовых шаблонов. Регистры могут быть жестко заданы, используя их имена или номера. Для динамического выбора регистров, номер должен быть задан как случайное значение (функция *rand*) или как неизвестное значение (символ "_"). Для неизвестного значения стратегия выбора регистра задается при помощи атрибута *select*, параметризованного именем стратегии (по умолчанию используется стратегия *random*). При этом можно использовать дополнительный атрибут *exclude*, чтобы исключить некоторые регистры из диапазона, или атрибут *retain*, чтобы задать пользовательский диапазон. Помимо этого конкретным стратегиям могут быть переданы пользовательские атрибуты, позволяющие настроить их поведение. Т.к. синтаксис сложных стратегий выбора регистров может быть достаточно запутанным, для упрощения кода их описания можно поместить в специализированные Ruby-функции. Пример 16 демонстрирует использование описанных конструкций выбора регистров (архитектура MIPS).

Пример 16. Конструкции языка тестовых шаблонов для выбора регистров

```
01:  add t0, t1, t2                                          Регистры жестко заданы по именам
02:  add r(8), r(9), r(10)                                   Регистры жестко заданы по номерам
03:  add r(rand(0, 31)), r(rand(0, 31)), r(rand(0, 31))      Случайный выбор из указанного диапазона
04:  add r(_), r(_), r(_)                                    Случайный выбор из диапазона, заданного в nML
05:  add r(_ select('free')), r(_ select('free')), t2        Выбор неиспользуемых регистров
06:  add r(_ select('free'), :exclude => [at, v0, v1]), t1, t2   Выбор неиспользуемых регистров кроме указанных
07:  add r(_ select('free'), :retain => [at, v0, v1]), t1, t2    Выбор неиспользуемых регистров из указанных
```

Кроме того язык тестовых шаблонов предоставляет функции *free_register* и *free_all_registers*, позволяющие поместить занятые регистры в список неиспользуемых.

При динамическом выборе регистров часто возникает необходимость задать зависимости по регистрам. Для этой цели могут использоваться Ruby-переменные, при помощи которых можно передать один и тот же объект, описывающий правило динамического выбора регистра, нескольким командам. В таком случае выбор регистра будет осуществляться при первом обращении, а выбранный регистр будет использоваться остальными командами. Другой вариант решения данной задачи – при мощи дополнительного атрибута передавать стратегии выбора уникальный идентификатор. В случае если, выбор

для данного идентификатора уже сделан, стратегия будет возвращать существующее значение, в противном случае будет строиться новое значение. Такой подход полезен для тестовых примеров, построенных путем комбинирования составных частей, описанных в разных шаблонах и независящих друг от друга. Пример 17 иллюстрирует оба подхода.

Пример 17. Задание зависимостей при динамическом выборе регистров

```
01:  add r1=r(_), t1, t2                          Зависимость типа “чтение после записи” задана при
02:  add r(_), r1, t3                             помощи переменой r1
03:  add r(_ select(‘random’), :id=>’r1’), t1, t2 Зависимость типа “чтение после записи” задана при
04:  add r(_), r(_ select(‘random’), :id=>’r1’), t3 помощи атрибута id, значение которого – ‘r1’
```

2.2.6 Описание тестовых ситуаций

Значения входных аргументов команд генерируются в соответствии с тестовыми ситуациями, связанными с этими командами. Тестовые ситуации могут быть различных типов и обрабатываться компонентами, реализующими различные техники генерации. При этом формат их описания в шаблонах остается единым. Для задания привязки тестовой ситуации к команде используется функция *situation*, в которую передается идентификатор ситуации и набор используемых ей атрибутов. В примере 18 демонстрируется использование данной функции.

Пример 18. Задание тестовых ситуаций для команд

```
01:  add t0, t1, t2 do situation(‘zero’) end      Значения входных регистров равны нулю
02:  add t3, t4, t5 do                            Значения входных регистров – случайные числа,
03:     situation(‘random’, :dist=>int_dist)      сгенерированные на основе распределения, заданного
04:  end                                          атрибутом dist
```

Если команда имеет непосредственные аргументы, значения которых должны быть сгенерированы в результате обработки заданной тестовой ситуации, то они задаются при помощи символа “_”.

Если для команды не задана тестовая ситуация и значения ее входных аргументов не определены в данном тестовом примере, для них будут сгенерированы случайные числа из диапазона, соответствующего их типу данных. Кроме этого при помощи функции *set_default_situation* можно задать тестовые ситуации, используемые по умолчанию для отдельных команд или групп команд. Использование данной функции демонстрируется в примере 19.

Пример 19. Задание тестовой ситуации по умолчанию для команды ADD

```
01:   set_default_situation 'add' do                  Для всех команд ADD будет
02:     situation('my_situation')                    использоваться ситуация my_situation
03:   end
```

Последовательность команд с заданными для них тестовыми ситуациями может обрабатываться различными способами. Например, могут выбираться различные комбинации решений для отдельных тестовых ситуаций. Таким образом, одна последовательность команд может порождать несколько тестовых примеров. При этом несколько методов обработки тестовых ситуаций могут использоваться совместно. В качестве примера можно привести задачу построения тестов, покрывающих различные пути в заданной структуре переходов, базовые блоки которой содержат команды, вызывающие различные комбинации событий, связанных с доступом к памяти. Настройки компонентов, реализующих стратегии обработки тестовых ситуаций, задаются при помощи атрибута *engines*. В примере 20 показан блок, задающий настройки для стратегии обработки ситуаций, связанных с потоком управления и памятью.

Пример 20. Задание стратегий обработки тестовых ситуаций

```
01:   sequence(
02:     :engines => {
03:       :combinator => 'diagonal',              Способ объединения результатов работы компонентов
04:       :branch => {:branch_exec_limit => 3,    Настройки для компонента branch
05:                   :trace_count_limit => -1},
06:       :memory => {:classifier => 'event-based',  Настройки для компонента memory
07:                   :page_mask => 0x0fff)}
08:     }) {
09:     ...
10:   }
```

Назначение данных настроек не имеет значения для языка описания тестовых шаблонов т.к. различные реализации этих компонентов будут иметь свои наборы настроек. Язык же предоставляет возможность задавать любые наборы настроек.

2.2.7 Описание инициализирующего кода и встроенных проверок

Тестовые данные, сгенерированные в результате обработки тестовых ситуаций, необходимо поместить в регистры или память. Для этой цели создается специальный *инициализирующий код*. Он строится на основе сгенерированных данных и правил, описанных в шаблонах. Данные правила

называются *препараторами* (preparator). Они определяются в методе *pre* тестового шаблона. Поддерживаются следующие виды препараторов:

- препараторы для регистров;
- препараторы для регистров на основе потоков данных;
- препараторы для памяти.

Набор препараторов может быть расширен путем написания соответствующих библиотек. Далее поддерживаемые препараторы будут рассмотрены более подробно.

Кроме этого тестовые примеры могут включать в себя *встроенные проверки*, которые проверяют состояние микропроцессора после выполнения тестового воздействия. Данные проверки строятся автоматически на основе информации о состоянии микропроцессора, предоставленной эмулятором, и правил, заданных в тестовых шаблонах. Эти правила называются *компараторами* (*comparator*) и содержат описания команд, используемых для создания встроенных проверок. По своим функциональным возможностям и синтаксису компараторы аналогичны препараторам. Два основных отличия заключаются в том, что они используются для сравнения текущих значений с эталонными и в случае несовпадения могут осуществлять переход в обработчику, отвечающему за создание сообщения от ошибке.

Препараторы для регистров

Данный тип препараторов описывает последовательности команд, которые помещают данные в регистры определенного типа, доступ к которым описан некоторыми режимами адресации. Т.к. для устранения избыточности важно, чтобы размер препараторов был минимальным, поддерживается возможность переопределения препараторов для отдельных случаев (определенные номера регистров, маски значений). Препараторы для регистров определяются при помощи конструкции *preparator*, которая использует следующие атрибуты для описания условий, при которых они будут применены:

- *target* – имя режима адресации, описывающего доступ к инициализируемому регистру;
- *mask* (необязательный атрибут) – маска, которой должны соответствовать данные, помещаемые в регистр;
- *arguments* (необязательный атрибут) – значения аргументов режима адресации (например, номер регистра), для которых применим данный препаратор;
- *name* (необязательный атрибут) – имя, которое уникально идентифицирует препаратор и используется для разрешения неоднозначностей в случаях когда, применимы несколько препараторов.

Кроме этого с целью улучшить тестовое покрытие поддерживается возможность определять несколько вариантов одного и того же препаратора, которые будут выбираться случайным образом в соответствии с заданным распределением. Они описываются при помощи конструкции *variant*, использующей два необязательных атрибута: (1) *name*, который задает уникальный идентификатор, используемый для выбора конкретного варианта, и (2) *bias*, который задает вес данного варианта.

Код препараторов использует ключевые слова *target* и *value* для обозначения инициализируемого регистра и записываемого в него значения соответственно. В случаях, когда необходимо использовать отдельные поля значения *value*, диапазон этих полей задается при помощи необязательных параметров. В примере 21 описывается препаратор для архитектуры MIPS32, который инициализирует 32-битрые регистры общего назначения *R* двумя способами (используя непосредственные значения и данные из памяти) и включает два дополнительных особых случая (для значений 0 и -1).

Пример 21. Препаратор для регистров общего назначения архитектуры MIPS32

```
01:   preparator(:target => 'R') {                    Препаратор для регистров R по умолчанию
02:     variant(:bias => 25) {                        Регистр инициализируется при помощи команд
03:       data {                                      чтения данных из памяти. Вес варианта: 25
04:         label :preparator_data
05:         word value
06:       }
07:       la at, :preparator_data
08:       lw target, 0, at
09:     }
10:     variant(:bias => 75) {                        Регистр инициализируется при помощи команд,
11:       lui target, value(16, 31)                   использующих непосредственные аргументы.
12:       ori target, target, value(0, 15)            Вес варианта: 75
13:     }
14:   }
15:   preparator(:target => 'R', :mask => '00000000') {   Особый случай для значения, равного 0
16:     xor target, zero, zero
17:   }
18:   preparator(:target => 'R', :mask => 'FFFFFFFF') {   Особый случай для значения, равного -1
19:     nor target, zero, zero
20:   }
```

Язык тестовых шаблонов позволяет явным образом вставлять инициализирующий код в тестовые программы. Это позволяет сократить объем тестовых шаблонов и создавать сложные препараторы, основанные на повторном использовании имеющихся описаний. Для этой цели используется функция *prepare*, которая принимает в качестве параметров инициализируемый регистр, присваиваемое ему значение, а также необязательные атрибуты *name* и *variant*, позволяющие при необходимости выбрать конкретный вариант конкретного препаратора. Например, чтобы вставить в тестовую программу код, помещающий значение 0xDEADBEEF в регистр t0, можно использовать следующий код:

Пример 22. Вставка препаратора для регистра тестовую программу

```
01:   prepare t0, 0xDEADBEEF          Будет вставлен код, записывающий 0xDEADBEEF в регистр t0
```

Компараторы для регистров

Компараторы для регистров определяется аналогично препараторам. С точки зрения синтаксиса все разница заключается в том, что вместо ключевого слова preparator используется ключевое слово comparator. Ниже приведен пример компаратора для регистров общего назначения архитектуры MIPS32, включающего особый случай для эталонного значения равного нулю.

Пример 23. Компаратор для регистров общего назначения архитектуры MIPS32

```
01:    comparator(:target => ’R’) {
02:       prepare target, value
03:       bne at, target, :check_failed
04:       nop
05:    }
06:    comparator(:target => ’R’, :mask => "00000000") {
07:       bne zero, target, :check_failed
08:       nop
09:    }
```

Препараторы для регистров на основе потоков данных

Препараторов, записывающих значения в регистры, недостаточно для решения некоторых задач. К таким задачам относится построение инициализирующего кода для команд условного перехода, исполняемых в цикле. Данная задача решается при помощи препараторов для регистров на основе потока данных. Они позволяют записывать и читать данные из потоков (массивов, хранящихся в памяти). Эти препараторы используют два регистра: один хранит адрес, используемый для чтения данных, а другой – прочитанные или записанные данные. При таком подходе поток сначала заполняется данными, а потом они считываются в процессе выполнения тестового примера.

Препараторы данного вида описываются при помощи конструкции *stream_preparator*, которая использует атрибуты *data_source* и *index_source* для задания типов регистров, хранящих данные и адрес. Препаратор включает в себя три секции: (1) *init*, которая содержит код инициализации регистра адреса начальным адресом потока; (2) *read*, которая содержит код чтения следующего элемента из потока и инкремента адреса; (3) *write*, которая содержит код записи нового элемента в поток и инкремента адреса. Внутри этих секций используются ключевые слова *start_label*, *index_source* и *data_source* для обозначения метки, указывающей на начало потока данных, и регистров адреса и данных соответственно. Пример 24 демонстрирует описание препаратора на основе потока данных для архитектуры ARMv8.

Пример 24. Препаратор для регистров на основе потока данных для архитектуры ARMv8

```
01:    stream_preparator(:data_source => 'REG', :index_source => 'REG') {
02:      init {
03:        adr index_source, start_label
04:      }
05:      read {
06:        ldar data_source, index_source
07:        add index_source, index_source, 8, 0
08:      }
09:      write {
10:        stlr data_source, index_source
11:        add index_source, index_source, 8, 0
12:      }
13:    }
```

Препараторы для памяти

Еще один тип поддерживаемых препараторов – препараторы для памяти, описывающие код размещения данных по заданному адресу. Язык тестовых шаблонов позволяет определить несколько вариантов препаратора данного типа, которые будет использоваться для блоков данных разного размера. Препараторы для памяти описываются при помощи конструкции *memory_preparator*, которая использует атрибут *size* для задания размера размещаемого в памяти блока данных. В коде препаратора используются ключевые слова *address* и *data* для обозначения адреса размещения и размещаемых данных соответственно. Пример 25 содержит код, размещающий в памяти 64-битные блоки данных, для архитектуры MIPS32.

Пример 25. Препаратор, размещающих 64-битные блоки данных в памяти

```
01:    memory_preparator(:size=>64) {
02:       la t0, address
03:       prepare t1, data(31, 0)
04:       sw t1, 0, t0
05:       prepare t1, data(63, 32)
06:       sw t1, 4, t0
07:    }
```

2.2.8 Описание правил рандомизации

Многие задачи генерации предполагают случайный выбор на основе заданного распределения. Для этой цели язык тестовых шаблонов предоставляет следующие конструкции:

- *range* – описывает выбираемые значения и их веса, которые задаются атрибутами *value* и *bias* соответственно. Значения могут принимать одну

из следующих форм: (1) одиночное значение; (2) диапазон значений; (3) массив значений; (4) распределение значений.

- *dist* – описывает распределения в виде набора значений, описанных конструкцией *range*. Если веса значений не заданы, это означает равномерное распределение.

Пример 26 демонстрирует использование описанных выше конструкций для описания распределения целых чисел.

Пример 26. Описание случайных распределений целых чисел

```
01:    simple_dist = dist(
02:      range(:value => 0, :bias => 25),
03:      range(:value => 1..2, :bias => 25),
04:      range(:value => [3, 5, 7], :bias => 50)
05:      )
06:    composite_dist = dist(
07:      range(:value=> simple_dist, :bias => 80),
08:      range(:value=> [4, 6, 8], :bias => 20)
09:      )
```

Описанные распределения можно использовать для построения значений непосредственных аргументов команд или значений используемых ими регистров. Пример 27 демонстрирует возможные способы их использования.

Пример 27. Использование случайных распределений целых чисел для генерации значений

```
01:    addi t0, t1, rand(simple_dist)                          Непосредственное значение
02:    addi t2, t3, _ do situation('random', :dist => simple_dist) end   Тестовая ситуация
03::   prepare t4, rand(simple_dist)                           Инициализирующий код
```

Случайный выбор на основе вероятностных распределений может использоваться также для выбора тестовых ситуаций, применяемых к командам. Для этой цели описывается распределение, в котором в качестве значений выступают ситуации, и которое связывается с командой при помощи конструкции *random_situation*. Данный подход показан в примере 28.

Пример 28. Случайный выбор тестовых ситуаций на основе заданного распределения

```
01:    sit_dist = dist(range(:value => situation('overflow')),
02:                    range(:value => situation('normal')),
03:                    range(:value => situation('zero')),
04:                    range(:value => situation('random', :dist => int_dist)))
05:    add t1, t2, t3 do random_situation(sit_dist) end
```

Кроме этого случайным образом могут выбираться отдельные команды и целые блоки команд. Случайный выбор команд связан с понятием *групп команд*. Они описываются в формальных спецификациях и используются в тестовых шаблонах, когда необходимо выбрать случайную команду заданной

группы. Группы команд также можно определять в тестовых шаблонах. Для этого описывается распределение для имен команд, на основе которого при помощи функции *define_group* определяется группа. Определение и использование группы команд показано в примере 29.

Пример 29. Определение и использование группы команд

```
01:  alu_dist = dist(range(:value => 'add', :bias => 40),          Описание распределение
02:                  range(:value => 'sub', :bias => 30),
03:                  range(:value => ['and', 'or', 'nor', 'xor'], :bias => 30))
04:  define_group('alu', alu_dist)                                 Определение группы
05:  alu t0, t1, t2                                                Случайная команда из группы
```

Для случайного выбора последовательностей команд определяется распределение, в котором в качестве значений выступают последовательности команд. После этого при помощи функции *random_sequence* задается случайный выбор последовательности. В примере 30 демонстрируется случайный выбор последовательностей команд.

Пример 30. Случайный выбор последовательностей команд

```
01:  seq_dist = dist(range(:value => lambda do sequence{...} end, :bias => 50),
02:                  range(:value => lambda do sequence{...} end, :bias => 35),
03:                  range(:value => lambda do sequence{...} end, :bias => 15))
04:  random_sequence(seq_dist)
```

Описанные выше подходы к описанию правил случайной генерации можно применять в комбинации для построения сложных тестовых примеров.

2.2.9 Описание размещения команд и данных в памяти

Ассемблерные программы содержат директивы, позволяющие задать адреса размещения их данных и команд в памяти. Язык тестовых шаблонов предоставляет аналогичные конструкции. В настоящий момент поддерживаются директивы: (1) *org*, которая задает адрес размещения в виде смещения от базового адреса, и (2) *align*, которая задает выравнивание для адреса размещения. Смещение можно также задавать как смещение от адреса конца предыдущего размещения, используя атрибут *delta*. Помимо этого язык предоставляет функцию *label*, которая позволяет определить метки для адресов тестовой программы. Пример 31 демонстрирует использование директив размещения и меток.

Пример 31. Использование директив размещения и меток

```
01:    org 0x00001000                  Начальное смещение 0x00001000
02:    label :start                    Объявление метки start
03:    add t0, t1, t2
04:    align 4                         Выравнивание адреса на 16 байт
05:    add t0, t1, t2
```

Данные описываются внутри блока *data*. Формат описания данных основан на типах данных, объявленных в формальных спецификациях. Пример 32 иллюстрирует определение блока данных для архитектуры MIPS32.

Пример 32. Определение блока данных

```
01:    data {
02:        org 0x00001000
03:        label :byte_values
04:        byte 1, 2, 3, 4
05:        label :word_values
06:        word 0xDEADBEEF, 0xBAADF00D
07:    }
```

2.2.10 Описание структуры переходов между тестовыми примерами

Сложные сценарии тестирования могут предполагать выполнение тестовых примеров в произвольном порядке, циклическое выполнение или выполнение на разных ядрах микропроцессора. За организацию структуры переходов между тестовыми примерами отвечает *код диспетчеризации (dispatching code)*, который размещается снаружи блоков, описывающих тестовые примеры. Этот код содержит глобальные метки и команды, осуществляющие переход по глобальным меткам, объявленных в других частях тестового шаблона. При этом тестовые примеры имеют единственную точку входа и не осуществляют глобальных переходов. В примере 33 показан код, организующий выполнение двух тестовых примеров на разных ядрах (архитектура ARMv8).

Пример 33. Описание тестовых примеров, выполняемых на разных ядрах

```
01:   mrs x1, mpidr_el1                 Код диспетчеризации:
02:   cmp x1, 0, 0                      передает управление на метку pe0_label для ядра 0 и
03:   b ne, :pe1_label                  на метку pe1_label для ядра 1
04:
05:   label :pe0_label
06:   sequence {                        Тестовый пример для ядра 0
07:     ...
08:   }.run
09:   b_imm :end_label
10:
11:   label :pe1_label
12:   sequence {                        Тестовый пример для ядра 1
13:     ...
14:   }.run
15:   b_imm :end_label
16:
17:   org 0x2100                        Код диспетчеризации:
18:   label :end_label                  сюда выполняется переход для обоих ядер после
19:   nop                               завершения выполнения тестовых примеров
```

2.2.11 Расширяемость языка описания шаблонов тестовых программ

Расширяемость генератора тестовых программ предполагает расширяемость языка, используемого для описания шаблонов. Важно заметить, что расширение функциональных возможностей генератора не обязательно требует добавления новых языковых конструкций. Многие конструкции используют наборы атрибутов, которые задают свойства описываемых сущностей. В таких случаях расширение языка сводится, к добавлению новых атрибутов или использованию новых значений для уже имеющихся атрибутов. Такой подход применим для решения следующих задач:

- поддержка новых способов построения последовательностей команд;
- поддержка новых стратегий выбора регистров;
- поддержка новых типов тестовых ситуаций;
- поддержка новых способов комбинирования тестовых ситуаций;
- поддержка новых способов обработки тестовых примеров.

Однако некоторые задачи могут потребовать создания новых языковых конструкций. К таким задачам относятся:

- поддержка препараторов и компараторов нового типа;
- поддержка возможности описания пользовательских ограничений.

Для решения этих задач необходимо разработать Ruby-классы, реализующие соответствующие методы и добавить их в список классов,

наследуемых базовым классом Template. При этом блочные конструкции реализуются как методы, принимающие в качестве параметра лямбда-функцию, которая содержит код, находящийся внутри блока. Это позволит анализировать его в отдельном контексте независимо от других конструкций языка. Таким образом, новые конструкции не будут конфликтовать с уже имеющимися.

2.3 Архитектура генераторов тестовых программ

Генераторы тестовых программ состоят из архитектурно-независимого *ядра* и *модели* микропроцессора, конструируемой на основе формальных спецификаций. Генерация осуществляется ядром, которое получает всю необходимую для этого информацию из модели и шаблонов тестовых программ. Ядро включает в себя несколько основных компонентов, каждый из которых отвечает за решение отдельной задачи. Архитектура генератора тестовых программ показана на рисунке 5.

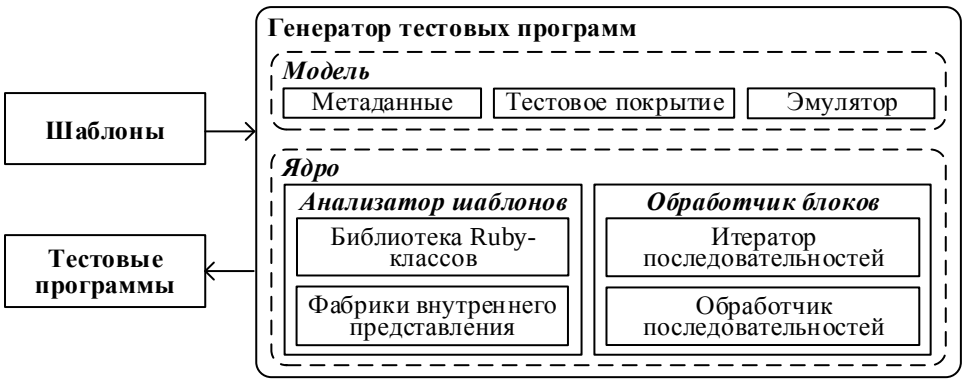


Рисунок 5. Архитектура генератора тестовых программ

Процесс генерации включает в себя две основные стадии: (1) анализ шаблона с целью построения его *внутреннего представления* и (2) обработка внутреннего представления с целью генерации тестовых программ. Эти задачи решаются при помощи *анализатора (template analyzer)* и *обработчика (template processor)* шаблонов соответственно. В процессе обработки внутреннего представления строятся последовательности команд, удовлетворяющие заданным для них структурным и поведенческим свойствам, из которых составляется тестовая программа. За удовлетворение структурных свойств отвечает *итератор последовательностей (sequence iterator)*, который строит

множество последовательностей, соответствующих описанной структуре. Построенные последовательности передаются *обработчику последовательностей (sequence processor)*, задача которого заключается в удовлетворении заданных для них поведенческих свойств. Для этого формируются ограничения на входные данные команд, которые должны быть выполнены для достижения тех или иных тестовых ситуаций. На основе ограничений генерируются данные и строится код, размещающий их регистрах или памяти. Основные компоненты генератора и механизмы их взаимодействия подробно рассмотрены в следующих разделах.

2.3.1 Анализатор шаблонов тестовых программ

Анализатор шаблонов тестовых программ (template analyzer) обрабатывает шаблоны с целью построения их *внутреннего представления*. Так как шаблоны представляют собой классы на языке Ruby, их обработка сводится к исполнению их кода, в результате которого конструируются объекты внутреннего представления. Анализатор шаблонов тестовых программ включает в себя следующие основные компоненты: (1) интерпретатор языка Ruby; (2) библиотеки Ruby-модулей, описывающие конструкции языка; (3) классы, описывающие объекты внутреннего представления; (4) фабрики, вызываемые Ruby-модулями для построения объектов внутреннего представления. Процесс анализа шаблона состоит из следующих шагов:

1. интерпретатор загружает Ruby-класс, описывающий шаблон;
2. в процессе загрузки класса вызывается библиотечный код, создающий на основе метаданных из модели микропроцессора динамические методы, позволяющие описывать свойства элементов конкретной архитектуры;
3. вызываются методы загруженного Ruby-класса, которые содержат код, вызывающий статические и динамические методы библиотечных модулей, отвечающие за создание объектов внутреннего представления;
4. вызываемые методы библиотечных модулей обращаются к фабрикам, которые конструируют объекты внутреннего представления.

Внутреннее представление шаблона отражает структуру генерируемой тестовой программы. Оно включает в себя последовательность объектов, описывающих его части (пролог, эпилог, тестовые примеры и код диспетчеризации), называемых *блоками*. Они соответствуют блочным конструкциям языка шаблонов, используемых для описания тестовых примеров. При этом пролог, эпилог и код диспетчеризации, не использующие блочные конструкции, рассматриваются как элементарные блоки. Помимо блоков, внутреннее представление включает в себя объекты, описывающие препараторы и компараторы.

Конструирование внутреннего представления осуществляются следующими методами класса шаблона, которые вызываются в указанном порядке:

- *pre* – конструирует внутреннее представление пролога тестовой программы, а также всех препараторов и компараторов;
- *post* – конструирует внутреннее представление эпилога тестовой программы;
- *run* – конструирует внутреннее представление тестовых примеров и кода диспетчеризации.

Блоки внутреннего представления содержат в себе списки объектов, описывающих команды или вложенные блоки. Кроме этого каждых блок снабжен атрибутами, задающими способ построения последовательностей команд. Объекты, описывающие команды, которые принято называть *абстрактными командами*. Они используют следующие атрибуты: (1) имя команды; (2) передаваемые ей аргументы; (3) заданная для нее тестовая ситуация; (4) объект метаданных, описывающий ее свойства.

Аргументами команды может быть другая команда (для VLIW-команд), режим адресации (addressing mode) или непосредственное значение (immediate value). Режимы адресации имеют набор атрибутов, аналогичный командам, с тем отличием, что все аргументы являются непосредственными значениями. Непосредственные аргументы команд и режимов адресации могут иметь

фиксированное значение или их значение может генерировать по определенным правилам.

Помимо команд блоки могут содержать в себя следующие типы объектов: метки; секции данных; директивы ассемблера; текстовые сообщения; специальные инструкции для управления работой генератора, и т.д. Они задаются как атрибуты команд. Это позволяет зафиксировать их местоположение относительно соседних команд. Также для них могут создаваться специальные псевдокоманды, содержание только объекты перечисленных типов.

Анализатор является архитектурно-независимым. Он применим к любой архитектуре, описанной моделью микропроцессора (создаваемые им объекты внутреннего представления основаны на матаданных модели). Анализатор также является расширяемым. Это означает, что в язык описания шаблонов можно добавлять конструкции для задания новых свойств генерируемых программ. Для этого необходимо разработать и зарегистрировать следующие компоненты: (1) Ruby-модуль, описывающий новую конструкцию; (2) фабрику, конструирующую соответствующий объект внутреннего представления, и (3) классы, на основе которых будет построен этот объект.

2.3.2 Обработчик внутреннего представления

Обработчик шаблонов тестовых программ (template processor) осуществляет генерацию тестовых программ путем обработки блоков внутреннего представления, построенных анализатором шаблонов. Блоки обрабатываются по отдельности в определенном порядке. В результате их обработки строятся фрагменты кода (последовательности команд), удовлетворяющие заданным структурным и поведенческим свойствам, из которых затем составляются тестовые программы.

Код, построенный в результате обработки блока, исполняются на эмуляторе. Это позволяет отслеживать текущее состояние микропроцессора в процессе генерации. Такая возможность необходима для построения тестовых примеров с учетом текущего состояния микропроцессора, создания встроенных

проверок и контроля корректности построенного кода. Также это позволяет обеспечить обработку блоков в соответствии с порядком исполнения описываемых ими фрагментов кода. Это важно, так как между ними могут быть зависимости по состоянию микропроцессора.

Процесс генерации тестовых программ обработчиком тестовых шаблонов включает следующие стадии:

1. Построение кода пролога, эпилога и секций диспетчеризации (имеющих фиксированные адреса), а также размещение пролога, секций диспетчеризации и всех глобальных данных в памяти эмулятора. При этом фиксируются адреса тестовых примеров, которые будут размещаться следом за построенными секциями кода.
2. Исполнение построенного кода на эмуляторе с целью подготовки начального состояния, необходимого для генерации тестовых примеров. Исполнение приостанавливается при достижении конца построенного кода.
3. Генерация кода тестового примера, адрес размещения которого был достигнут в процессе исполнения построенного ранее кода, и размещение его в памяти эмулятора.
4. Запуск исполнения построенного тестового примера. Исполнение приостанавливается при достижении адреса его конца.
5. Построение кода встроенных проверок на основе информации о текущем состоянии микропроцессора, полученной из эмулятора. При этом построенный код размещается в памяти эмулятора и исполняется.
6. Запуск исполнения с последней точки остановки. При достижении адреса начала еще не построенного тестового примера, выполняется переход к стадии 3. При достижении адреса начала кода диспетчеризации, который еще не был размещен в памяти (адрес не был зафиксирован), он размещается в памяти, а стадия 6 повторяется.
7. Размещение в памяти и исполнение эпилога.
8. Печать сгенерированного кода в виде программы на языке ассемблера.

Построение кода для блоков, описывающих различные части тестовых программ, осуществляется по-разному. По способы обработки блоки можно разделить на два типа: (1) со *статической* структурой и (2) с *динамической* структурой. К первому типу относятся пролог, эпилог и код диспетчеризации. Каждый из них порождает единственную последовательность команд, которая имеет жестко заданную структуру и использует фиксированные значения аргументов команд. Ко второму типу относятся блоки, описывающие тестовые примеры. Они порождают одну или несколько последовательностей команд, удовлетворяющих заданным структурным и поведенческим свойствам.

Для блоков со статической структурой процесс построения последовательности команд выглядит следующим образом. На основе блока строится *абстрактная последовательность команд*, состоящая из абстрактных команд, порядок которых полностью соответствует шаблонному описанию. О такой последовательности говорят, что она имеет фиксированный адрес, если в ее начале располагаются директивы ассемблера (такие как .org), задающие ее смещение. Построенная абстрактная последовательность передается специальной фабрике, реализуемой моделью микропроцессора, которая строит на ее основе *конкретную последовательность команд*. Такая последовательность состоит из *конкретных команд*, реализуемых эмулятором, которые можно исполнить и сохранить в ассемблерном и бинарном форматах.

Обработка блоков с динамической структурой несколько сложнее. На основе блоков данного типа строится одна или несколько абстрактных последовательностей (способы их построения будут рассмотрены в следующем подразделе). Затем осуществляется обработка этих последовательностей, которая включает в себя следующие стадии: (1) выбор регистров, используемых командами; (2) построение ограничений для тестовых ситуаций, связанных с командами; (3) генерация входных данных для команд посредством разрешения ограничений; (4) построение инициализирующего кода для сгенерированных данных. В результате обработки сначала получаются уточненные абстрактные последовательности (их число может быть больше

числа оригинальных последовательностей), а затем на их основе строятся конкретные последовательности. Стадии построения этих последовательностей будут более подробно рассмотрены далее.

Построенные конкретные последовательности помещаются в память эмулятора и исполняются, а после завершения обработки всех блоков печатаются в виде тестовой программы.

2.3.3 Итератор последовательностей команд

Построение абстрактных последовательностей на основе блоков внутреннего представления осуществляется следующим образом. Каждый блок реализует *итератор абстрактных последовательностей команд*: итераторы неэлементарных блоков строятся на основе итераторов вложенных блоков; итератор элементарного блока возвращает одну последовательность, состоящую из единственной команды. Для блоков задаются атрибуты, которые определяют способ построения итератора последовательностей блока из итераторов вложенных блоков. Поддерживаются следующие атрибуты:

- *combinator* – техника комбинирования последовательностей, возвращаемых итераторами вложенных блоков;
- *permutator* – техника модификации комбинации последовательностей;
- *compositor* – техника композиции последовательностей;
- *rearranger* – техника перегруппировки последовательностей;
- *obfuscator* – техника модификации последовательности.

Построение итератора последовательностей для неэлементарного блока осуществляется путем последовательного применения перечисленных техник. Перечисленные техники реализуются в виде компонентов, которые имеют названия *комбинатор*, *пермутатор*, *композитор*, *перегруппировщик* и *обфускатор* соответственно. Может существовать несколько различных реализаций одной и той же техники. Набор компонентов может быть расширен пользовательскими реализациями.

Для демонстрации функций перечисленных техник будет использоваться описание шаблонное блока из примера 34.

Пример 34. Блок, параметризированный методами построения последовательностей

```
01:   # несколько комбинированных последовательностей
02:   block(
03:     :combinator => 'combinator-name',
04:     :permutator => 'permutator-name',
05:     :compositor => 'compositor-name',
06:     :rearranger => 'rearranger-name',
07:     :obfuscator => 'obfuscator-name') {
08:    # 3 последовательности длины 1: {A11}, {A21} и {A31}
09:    iterate { # Блок A
10:      A11,
11:      A21,
12:      A31,
13:    }
14:    # 2 последовательности длины 2: {B11, B12} и {B21, B22}
15:    iterate { # Блок B
16:      sequence { B11, B12 }
17:      sequence { B21, B22 }
18:    }
19:    # 1 последовательность длины 3: {C11, C12, C13}
20:    iterate { # Блок C
21:      sequence { C11, C12, C13 }
22:    }
23:  }
```

Комбинаторы

Комбинатор – это компонент, осуществляющий комбинирование последовательностей, возвращаемых итераторами вложенных блоков. Он принимает на вход упорядоченный набор итераторов последовательностей и выдает на выход итератор по комбинациям, построенным из этих последовательностей. *Комбинацией последовательностей* – это кортеж из нескольких последовательностей (их число совпадает с числом вложенных блоков). Поддерживаются следующие комбинаторы (возможные значения *combinator-name*): (1) *diagonal* (используется по умолчанию); (2) *product; (3) random.*

Комбинатор *diagonal* синхронно итерирует последовательности вложенных блоков. Комбинирование завершается, когда исчерпываются все вложенные итераторы. Каждый раз, когда вложенный итератор исчерпывается, он инициализируется повторно. Число комбинаций, выдаваемых комбинатором *diagonal*, равно $max(N_i,...,N_k)$, где N_i – число последовательностей,

возвращаемых итератором i-ого блока. На рисунке 6 показаны комбинации, возвращаемые комбинатором *diagonal* для блока, описанного в примере 34.

#1	#2	#3
$A11$	$A21$	$A31$
$B11$ $B12$	$B21$ $B22$	$B11$ $B12$
$C11$ $C12$ $C13$	$C11$ $C12$ $C13$	$C11$ $C12$ $C13$

Рисунок 6. Комбинации, возвращаемые комбинатором *diagonal*

Комбинатор *product* строит все возможные комбинации последовательностей, возвращаемых итераторами вложенных блоков. Число комбинаций, выдаваемых комбинатором *product*, равно $N_1 \times ... \times N_k$, где N_i – число последовательностей, возвращаемых итератором i-ого блока. На рисунке 7 показаны комбинации, возвращаемые комбинатором *product.*

#1	#2	#3	#4	#5	#6
$A11$	$A11$	$A21$	$A21$	$A31$	$A31$
$B11$ $B12$	$B21$ $B22$	$B11$ $B12$	$B21$ $B22$	$B11$ $B12$	$B21$ $B22$
$C11$ $C12$ $C13$	$C11$ $C12$ $C13$	$C11$ $C12$ $C13$	$C11$ $C12$ $C13$	$C11$ $C12$ $C13$	$C11$ $C12$ $C13$

Рисунок 7. Комбинации, возвращаемые комбинатором *product*

Комбинатор *random* выдает одну случайную комбинацию последовательностей, возвращаемых итераторами вложенных блоков. Число комбинаций, выдаваемых комбинатором $random$, равно 1. На рисунке 8 показывается один из возможных вариантов комбинации, которую может возвратить комбинатор *random*.

#1
$A31$
$B21$ $B22$
$C11$ $C12$ $C13$

Рисунок 8. Вариант комбинации, возвращаемой комбинатором *random*

Пермутаторы

Пермутатор – это компонент, модифицирующий комбинации, возвращаемые комбинатором путем перестановки некоторых последовательностей. Он принимает на выход итератор комбинаций

последовательностей и отдает на выход итератор модифицированных комбинаций последовательностей. Поддерживаются следующие пермутаторы (возможные значения *permutator-name*): (1) *trivial* (используется по умолчанию); (2) *random.*

Пермутатор *trivial* оставляет каждую комбинацию без изменений. Т.е. пара из комбинатора *product* и пермутатора *trivial* возвратит в точности такие же комбинации, что и просто комбинатор *product.*

Пермутатор *random* меняет порядок последовательностей в комбинации случайным образом. На рисунке 9 показаны комбинации, которые может возвратить пара из комбинатора *product* и пермутатора *random.*

#1	#2	#3	#4	#5	#6
$C11$ $C12$ $C13$	$B21$ $B22$	$C11$ $C12$ $C13$	$A21$	$C11$ $C12$ $C13$	$A31$
$B11$ $B12$	$A11$	$A21$	$B21$ $B22$	$B11$ $B12$	$C11$ $C12$ $C13$
$A11$	$C11$ $C12$ $C13$	$B11$ $B12$	$C11$ $C12$ $C13$	$A31$	$B21$ $B22$

Рисунок 9. Комбинации, возвращаемой комбинатором *product* и пермутатором *random*

Композиторы

Композитор – это компонент, объединяющий (мультиплексирующий) последовательности, входящие в комбинацию, в одну последовательность (с сохранением порядка инструкций внутри каждой последовательности). Он принимает на вход комбинацию последовательностей и выдает на выход объединенную последовательность. Поддерживаются следующие композиторы (возможные значения *compositor-name*): (1) *catenation* (используется по умолчанию); (2) *rotation*; (3) *random.*

Композитор *catenation* объединяет последовательности, располагая их одна за другой.

Композитор *rotation* объединяет последовательности, используя *круговую дисциплину* (*round robin*): в результирующую последовательность по кругу, начиная с первой последовательности, добавляется очередная инструкция очередной последовательности. Если последовательность себя исчерпала, она

исключается из рассмотрения. На рисунке 10 показан результат работы композитора *rotation.*

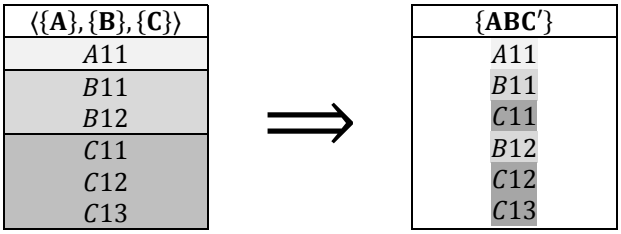


Рисунок 10. Результат работы композитора *rotation*

Композитор *random* объединяет последовательности случайным образом (но с сохранением порядка инструкций внутри каждой последовательности). На рисунке 11 показан результат работы композитора *random.*

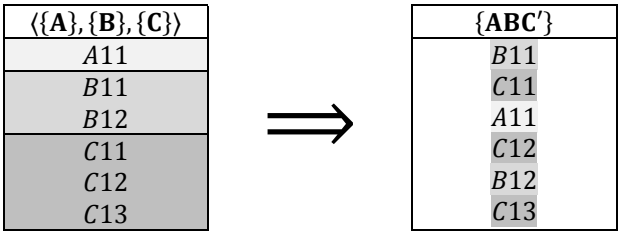


Рисунок 11. Результат работы композитора *random*

Перегруппировщики

Перегруппировщик – это компонент, перегруппирующий последовательности, построенные композитором. Он принимает на вход набор последовательностей и отдает на выход модифицированный набор последовательностей. Поддерживаются следующие перегруппировщики (возможные значения *rearranger-name*): (1) *trivial* (используется по умолчанию); (2) *expand.*

Перегруппировщик *trivial* оставляет набор последовательностей без изменений.

Перегруппировщик *expand* объединяет набор последовательностей в одну единственную последовательность путем их конкатенации. На рисунке 12 показан результат работы перегруппировщика *expand.*

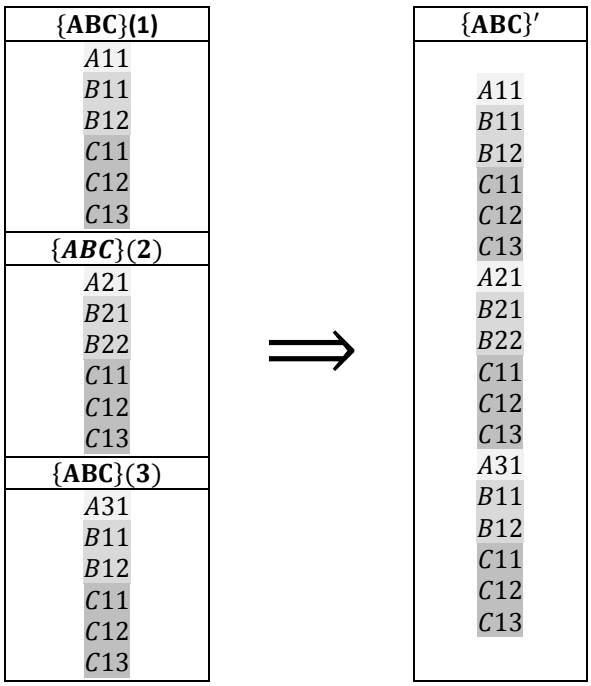


Рисунок 12. Результат работы перегруппировщика *expand*

Обфускаторы

Обфускатор – это компонент, модифицирующий последовательности, возвращаемые перегруппировщиком (путем перестановки некоторых инструкций). Он принимает на вход последовательность и отдает на выход модифицированную последовательность. Поддерживаются следующие обфускаторы (возможные значения *obfuscator-name*): (1) *trivial* (используется по умолчанию); (2) *random.*

Обфускатор *trivial* оставляет последовательность без изменений.

Обфускатор *random* меняет порядок инструкций в последовательности случайным образом. Результат его работы показан на рисунке 13.

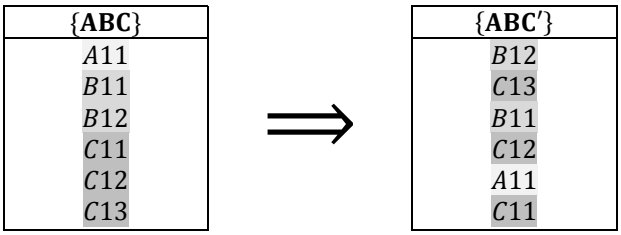


Рисунок 13. Результат работы обфускатора *random*

В настоящее время на использование блоков в шаблонах тестовых программ накладываются следующие ограничения: (1) блоки *sequence* и *atomic* не содержат вложенных блоков; (2) блоки *sequence* и *atomic* поддерживают

только параметр *obfuscator*; (3) блоки *iterate* поддерживают только параметры *rearranger* и *obfuscator*.

Комбинатор diagonal	**Пермутатор random**	**Композитор catenation**	**Перегруппировщик trivial**	**Обфускатор trivial**
Инициализируются итераторы последовательностей вложенных блоков *A*, *B* и *C*. Составляется комбинация из очередных последовательностей блоков *A(1)*, *B(1)* и *C(1)*.	Строится случайная перестановка последова-тельностей	Последова-тельности комбинации конкатенируются	Набор последовательностей не изменяется	Последова-тельность не изменяется
$A11$ $B11$ $B12$ $C11$ $C12$ $C13$	$B11$ $B12$ $A11$ $C11$ $C12$ $C13$	$B11$ $B12$ $A11$ $C11$ $C12$ $C13$	$B11$ $B12$ $A11$ $C11$ $C12$ $C13$	$B11$ $B12$ $A11$ $C11$ $C12$ $C13$
Итератор блока *C* исчерпан; он переинициализируется. Составляется комбинация из очередных последовательностей блоков *A(2)*, *B(2)* и *C(1)*.	Строится случайная перестановка последова-тельностей	Последова-тельности комбинации конкатенируются	Набор последовательностей не изменяется	Последова-тельность не изменяется
$A21$ $B21$ $B22$ $C11$ $C12$ $C13$	$C11$ $C12$ $C13$ $B21$ $B22$ $A21$	$C11$ $C12$ $C13$ $B21$ $B22$ $A21$	$C11$ $C12$ $C13$ $B21$ $B22$ $A21$	$C11$ $C12$ $C13$ $B21$ $B22$ $A21$
Итераторы блоков *B* и *C* исчерпаны; они переинициализируются. Составляется комбинация из очередных последовательностей блоков *A(3)*, *B(1)* и *C(1)*.	Строится случайная перестановка последова-тельностей	Последова-тельности комбинации конкатенируются	Набор последовательностей не изменяется	Последова-тельность не изменяется
$A31$ $B11$ $B12$ $C11$ $C12$ $C13$	$A31$ $B11$ $B12$ $C11$ $C12$ $C13$	$A31$ $B11$ $B12$ $C11$ $C12$ $C13$	$A31$ $B11$ $B12$ $C11$ $C12$ $C13$	$A31$ $B11$ $B12$ $C11$ $C12$ $C13$
Итераторы всех вложенных блоков исчерпаны; построение комбинаций завершается.				

Рисунок 14. Пример построения последовательности

Таким образом, применяя различные типы блоков, использующие различные стратегии построения последовательностей, возможно описывать

сложные тестовые примеры путем комбинирования уже имеющихся описаний. Рисунок 14 иллюстрирует построение последовательностей инструкций для следующих значений параметров: (1) комбинатор – *diagonal*; (2) пермутатор – *random*; (3) композитор – *catenation*; (4) перегруппировщик – *trivial*; (5) обфускатор – *trivial*.

2.3.4 Обработчик последовательностей команд

В построенных абстрактных последовательностях могут быть не определены используемые регистры и входные данные для команд. Задача обработчика *последовательностей команд (sequence processor)* – фиксация всех не определенных в тестовых шаблонах параметров для достижения заданных поведенческих свойств. Обработка последовательностей включает в себя следующие стадии:

1. выбор регистров, используемых командами;
2. построение ограничений для тестовых ситуаций, связанных с командами;
3. генерация входных данных для команд путем разрешения ограничений;
4. построение кода, помещающего данные в регистры и память.

Выбор регистров

Если номер используемого командой регистра не задан жестко, то он помечается как неизвестный. Неизвестные номера регистров снабжаются набором атрибутов, задающих способ их выбора. Выбор номеров регистров осуществляется при помощи компонентов ядра, называемых *стратегиями выбора регистров*. Общая схема выбора регистров выглядит следующим образом. Просматриваются все команды и составляется множество используемых известных регистров. Оно используется некоторыми стратегиями выбора регистров. После этого все команды просматриваются заново и для каждого неизвестного регистра осуществляется выбор номера при помощи стратегии, которая выбирается в соответствии с заданными атрибутами. При выборе номера регистра множество используемых известных регистров обновляется. Кроме этого обновление множества происходит, если

при просмотре последовательности команд встречаются служебные инструкции генератора, влияющие на выбор регистров (резервация номеров регистров, сброс зарезервированных регистров и т.д.).

Построение ограничений

Входные данные для команд генерируются в соответствии с тестовыми ситуациями, заданными для них в шаблоне. Условия достижения тестовых ситуаций задаются при помощи *ограничений* различного вида. При этом тестовые ситуации для отдельных команд могут быть заданы как множество ситуаций одного класса. Между командами одной последовательности существуют зависимости, связанные с совместным использованием конвейера команд и памяти. Поэтому для достижения максимальной полноты тестирования необходимо обеспечить покрытие всевозможных комбинаций ситуаций, связанных с отдельными командами.

Задача комбинирования тестовых ситуаций может решаться различными способами. Важно понимать, что декартово произведение будет избыточно, а случайная композиция не сможет обеспечить должный уровень покрытия. Поэтому необходимо учитывать природу комбинируемых тестовых ситуаций и использовать для каждого класса ситуаций свои стратегии построения комбинаций. Таким образом, будет использоваться подход, при котором комбинации тестовых ситуаций строятся для команд определенных классов и затем объединяются по заданным правилам в последовательности, описывающие тестовые воздействия. При этом при обработке одной абстрактной последовательности будет порождаться несколько тестовых воздействий, состоящих из тех же команд, но использующих разные комбинации ситуаций и аннотированных различными ограничениями.

Компоненты, отвечающие за комбинирования тестовых ситуаций определенных типов и построение соответствующих ограничений, называются *генераторами ограничений*. Они выбирают из обрабатываемой абстрактной последовательности интересующие их команды и строят для них комбинации тестовых ситуаций. Способ комбинирования определяется техникой

комбинирования, реализованной компонентом, и атрибутами, заданными в шаблоне. Затем для каждой ситуации строится ограничение. В результате на выходе получается множество последовательностей, состоящих из одних и тех же команд, но аннотированными разными ограничениями. В некоторых случаях генераторы ограничений могут при этом вставлять в эти последовательности дополнительные команды для вспомогательных целей.

Последовательности команд, получаемые на выходе разных генераторов ограничений, объединяются в последовательности, описывающие тестовые воздействия. При этом строятся всевозможные их комбинации. Для комбинирования используется один из поддерживаемых ядром комбинатор. Используемый комбинатор задается в шаблоне при помощи соответствующего атрибута блока.

Ограничения, которыми аннотируются команды, берутся из модели покрытия или реализуются ядром.

Генерация данных

Генерация входных данных для команд осуществляется путем разрешения ограничений. Ограничения разрешаются в порядке исполнения команд, чтобы учесть зависимости между ними. Для разрешения ограничений каждого типа используются отдельный компонент, называемый *решателем*. Для сгенерированных в результате решения ограничения данных строится инициализирующий код, который добавляется в качестве пролога коду тестового воздействия.

Чтобы обеспечить разрешение ограничений в правильном порядке, код тестового воздействия исполняется на эмуляторе. Для этого используется временный контекст эмулятора, который потом сбрасывается. Это делается потому, что к этому моменту размер инициализирующего кода неизвестен и, следовательно, нет возможности разместить код тестового воздействия по правильному адресу. Таким образом, процесс генерации данных выглядит следующим образом:

- запускается исполнение последовательности на эмуляторе;

- перед первым исполнением каждой команды для нее разрешается ограничение;
- для решения ограничения используется информация о текущем состоянии регистров и памяти, предоставляемая эмулятором;
- для сгенерированных в результате решения ограничения данных строится инициализирующий код и исполняется;
- если обрабатываемая команда использует непосредственные аргументы, для которых были сгенерированы значения, то она обновляется в эмуляторе;
- при следующем исполнении команды генерация данных не происходит;
- переходы потока управления за пределы последовательности игнорируются.

В результате получается конкретная последовательность команд с фиксированными значениями операндов и включающая в себя необходимый инициализирующий код. Вопросы построения инициализирующего кода рассмотрены ниже.

Построение инициализирующего кода

Данные, генерируемые в результате решения ограничений различного типа, могут использоваться по-разному. Это могут быть значения регистров, данные, помещаемые в память, адреса, таблицы трансляции адресов и т.д., для которых требуется создавать инициализирующий код различного вида. Поэтому в шаблонах для каждого типа ограничений задается соответствующий препаратор. Препараторы описываются в виде абстрактных последовательностей, в которые подставляются нужные данные и номера регистров. Для каждого препаратора разрабатывается Ruby-модуль, описывающий используемую языковую конструкцию, классы внутреннего представления и фабрика, позволяющая на основе внутреннего представления и входных данных построить инициализирующий код.

2.3.5 Расширяемость генератора тестовых программ

Описанная архитектура генератора тестовых программ допускает расширение его функциональности пользовательскими реализациями компонентов, решающих следующие задачи:

- построение последовательностей команд (комбинаторы, композиторы, перестановщики, перегруппировщики, обфускаторы);
- выбор регистров (стратегии выбора);
- комбинирование тестовых ситуаций и построение ограничений (генераторы ограничений);
- разрешение ограничений определенного типа (решатели ограничений);
- построение инициализирующего кода определенного вида (препараторы).

Компоненты, отвечающие за обработку тестовых ситуаций определенного типа, объединяются в *расширения*. Для поддержки возможности обработки тестовых ситуаций нового типа, разрабатывается соответствующее расширение и регистрируется в ядре генератора.

2.4 Выводы

В данной главе был предложен метод конструирования генераторов тестовых программ для микропроцессоров на основе формальных спецификаций их архитектуры. Метод позволяет создавать расширяемые генераторы тестовых программ для широкого спектра микропроцессорных архитектур. Генерация тестовых программ осуществляется на основе шаблонов, описывающих их структурные и поведенческие свойства в абстрактном виде, разработанных на предметно-ориентированном языке, основанном на Ruby. Архитектура генераторов позволяет использовать различные техники генерации, набор которых может быть расширен пользовательскими реализациями.

Глава 3. Реализация предложенного метода

В данной главе приводится описание реализации метода автоматизации конструирования генераторов тестовых программ, предложенного в предыдущей главе. Метод нашел свое воплощение в инструменте с открытым исходным кодом MicroTESK (Microprocessor TEsting and Specification Kit) версии 2.0 [82], разработанном на языке Java. Инструмент MicroTESK осуществляет конструирование генераторов тестовых программ для микропроцессоров на основе формальных спецификаций их архитектуры. Сконструированные генераторы состоят из архитектурно-независимого ядра, являющего частью инструмента, и модели микропроцессора, конструируемой на основе формальных спецификаций.

Инструмент MicroTESK разделен на две основные части: (1) *среду моделирования*, которая отвечает за конструирование модели микропроцессора, и (2) *среду тестирования*, которая отвечает за генерацию тестовых программ и включает в себя реализацию архитектурно-независимого ядра. Структура инструмента MicroTESK изображена на рисунке 15.

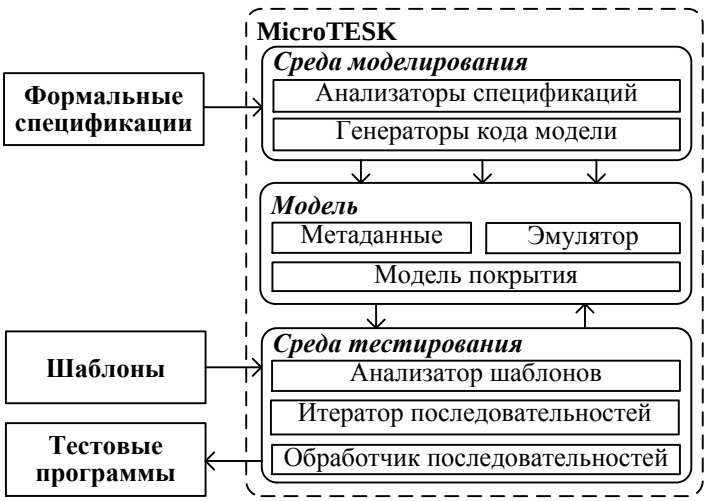


Рисунок 15. Архитектура инструмента MicroTESK

В общих словах, сценарий использования инструмента MicroTESK следующий. Пользователь разрабатывает формальные спецификации, описывающие архитектуру тестируемого микропроцессора. Эти спецификации анализируются средой моделирования, которая осуществляет построение модели микропроцессора. Затем пользователь разрабатывает шаблоны

тестовых программ, которые задают их свойства в терминах модели микропроцессора и техник генерации, поддерживаемых ядром. После этого среда тестирования осуществляет построение тестовых программ на основе шаблонов и модели микропроцессора.

Подробное описание среды моделирования, среды тестирования, а также их компонентов и механизмов их взаимодействия приводятся в последующих разделах.

3.1 Среда моделирования

Задача *среды моделирования* состоит в построении *модели* микропроцессора на основе формальных спецификаций. Модель содержит всю необходимую информацию для генерации тестовых программ для данного микропроцессора. Эта информация извлекается из формальных спецификаций при помощи набора *анализаторов*, каждый их которых отвечает за получение информации определенного типа. Т.к. архитектура микропроцессора может описываться при помощи формальных спецификаций различного типа, анализаторы объединяются в группы. Для поддержки новых типов спецификаций необходимо реализовать соответствующую группу анализаторов при помощи специальных библиотек, которые значительно упрощают эту задачу. Анализаторам спецификаций соответствуют *генераторы кода*, которые отвечают за построение отдельных компонентов модели на основе специальных *библиотек моделирования*. Компоненты модели содержат *точки соединения (connection point)*, которые обеспечивают интеграцию между компонентами различного типа.

3.1.1 Модель микропроцессора

Модель микропроцессора включает в себя три основных части: (1) *метаданные*, хранящие информацию о командах, регистрах и памяти микропроцессора; (2) *эмулятор*, позволяющий программно эмулировать исполнение команд и получать информацию о состоянии микропроцессора;

(3) *модель тестового покрытия*, содержащая информацию о возможных путях исполнения команд.

Метаданные

Задача метаданных – предоставить информацию о модели микропроцессора среде генерации, для которой она является «черным ящиком». Эта информация необходима для конфигурирования языка тестовых шаблонов под заданную архитектуру и для выполнения различных запросов к модели. Метаданные описывают следующие сущности:

- регистры и память (имя, количество элементов, тип данных);
- команды и режимы адресации (имя, параметры, атрибуты).

Регистры и память хранят информацию о текущем состоянии микропроцессора. Поэтому их описание необходимо для получения доступа к информации о состоянии микропроцессора.

Команды являются интерфейсом взаимодействия с микропроцессором (и его моделью). По этой причине их описание являются ключевым элементом метаданных. Оно необходимо для печати и исполнения команд, а также для построения тестов, предполагающих выбор команд, удовлетворяющих тем или иным требованиям. Рассмотрим более подробно структуру метаописаний команд.

Спецификации на языке nML описывают команды в виде иерархической структуры операций. При этом отдельные команды представлены путями из узлов, описывающих общие для всех команд свойства, к терминальным узлам, описывающим свойства уникальные для данных команд. Операции могут объединяться в группы для задания отношений «один ко многим». Параметрами операций являются режимы адресации или непосредственные значения (immediate values). Метаданные повторяют эту структуру, т.к. это необходимо для описания вызовов команды, состоящих из нескольких операций (характерно для архитектур семейства VLIW). Однако описание команд в виде путей в графе являются избыточными для более простых

семейств архитектур таких, как RISC. В таких архитектурах команды уникально идентифицируется операциями, расположенными в терминальных узлах. Например, в спецификации системы команд MIPS, структура которой показана на рисунке 16, команды *ADD* описывается путем из операции *instruction* к операции *add* (другими словами операция *instruction* параметризована операцией *add*), которая входит в группы *Op* и *ALU*. При этом операция *add* уникально идентифицирует команду *ADD*, а остальные операции, составляющие данную инструкцию, выводятся из *контекста*. Таким образом, вызов команды *ADD* можно описывать как вызов операции *add*. Для этого в метаданные вводится сущность, именуемая как *сокращенный вызов (shortcut)*, которая позволяет в зависимости от контекста описывать вызов команды (или некоторой ее части) как вызов операции, уникально идентифицирующей соответствующий путь в графе операций. Для одной операции может существовать несколько вариантов сокращенно вызова: в зависимости от контекста она может использоваться для обозначения всего пути или его части, заканчивающейся в этой операции.

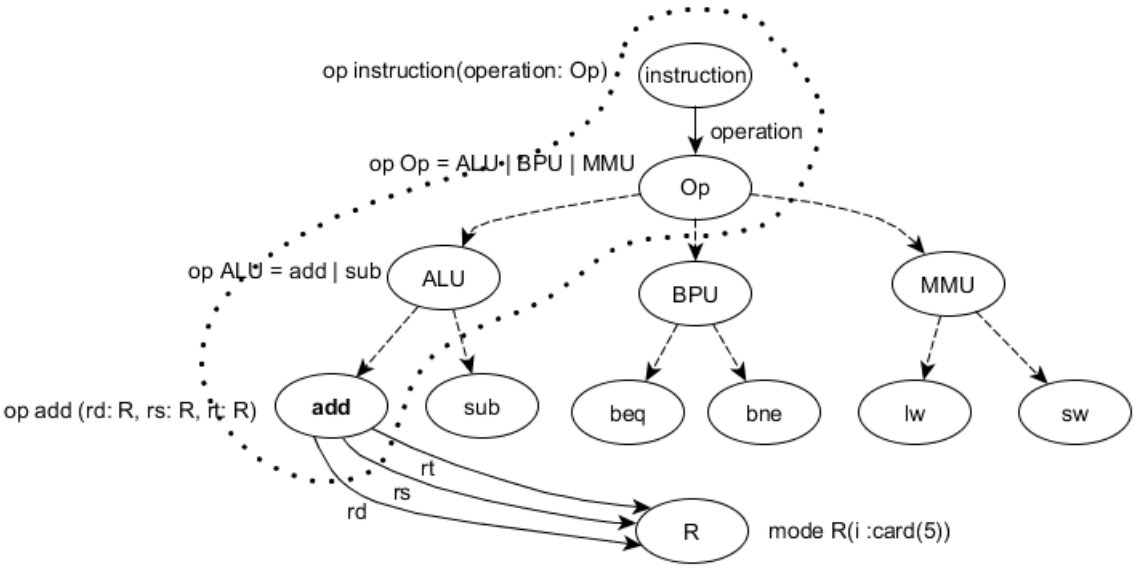


Рисунок 16. Структура системы команд MIPS и описание команды ADD

Метаданные содержат список операций, режимов адресации и их групп. Группы представляют собой коллекции имен входящих в них элементов.

Операции и режимы адресации характеризуются следующими атрибутами: (1) имя, (2) параметры, (3) сокращенные вызовы и (3) свойства.

Параметры описываются следующими атрибутами: (1) имя; (2) тип сущности (операция, режим адресации или непосредственный аргумент); (3) имя сущности (для операций и режимов адресации); (4) тип данных (для режимов адресации и непосредственных аргументов) и (5) тип доступа (чтение, запись).

Сокращенные вызовы характеризуются контекстом, в котором они используются, и списком параметров. Последний может отличаться от списка параметров вызываемой операции, т.к. другие операции на данном пути в графе могут иметь дополнительные параметры, которые необходимо им передавать. Все эти параметры помещаются в сигнатуру сокращенного вызова.

Операции и режимы адресации обладают списком свойств, которые характеризуют некоторые аспекты их поведения и которые могут быть использованы при построении тестов. Эти свойства синтезируются путем анализа спецификаций, и их набор может быть расширен. Набор свойств включает в себя:

- isBranch – показывает осуществляется ли переход (только для операций);
- isConditionalBranch – показывает осуществляется ли условный переход (только для операций);
- canThrowException – показывает может ли быть сгенерировано исключение;
- isMemoryReference – показывает осуществляется ли доступ к памяти (для режимов адресации);
- isLoad – показывает осуществляется ли чтение из памяти;
- isStore – показывает осуществляется ли запись в память;
- getBlockSize – размер блока данных, читаемого или записываемого в память.

Таким образом, метаданные предоставляют среде генерации всю информацию, необходимую для построения тестовых программ для моделируемой архитектуры.

Эмулятор

Эмулятор системы команд (instruction set simulator, ISS), являющейся частью модели, служит в качестве *эталонной модели (reference model),* на которой выполняются генерируемые последовательности команд с целью контроля состояния микропроцессора. Кроме этого он отвечает за печать команд. Эмулятор инкапсулирует в себе всю информацию о синтаксисе и семантике команд, и взаимодействие с ним осуществляется через набор обобщенных интерфейсов. Это позволяет сделать среду генерации полностью независимой от архитектуры тестируемого микропроцессора.

Эмулятор имеет следующие ограничения: (1) он не является потактово-точным, (2) выполнение отдельных команд считается атомарным, (3) параллельное выполнение команд не поддерживается. Эти допущения позволяют упростить реализацию и не оказывают существенного влияния на качество сгенерированных тестов. В тех случаях, когда требуется исполнение многопоточных программ, выполнение команд осуществляется последовательно, но используя разные контексты.

Список задач, решаемых эмулятором, включает в себя:

- печать команд в ассемблерном формате;
- кодирование команд в двоичный формат;
- декодирование команд;
- исполнение команд;
- построение трасс выполнения команд;
- управление контекстами;
- доступ к информации о текущем состоянии регистров и памяти.

Выполнение этих задач осуществляется при помощи набора абстракций. Две основные сущности, определяемые эмулятором: (1) *элементы хранения*

данных (включающие регистры, физическую память и временные переменные), и (2) *команды* (состоящие из операций, режимов адресации и непосредственных значений). Все из перечисленных выше задач решаются при помощи манипуляций над этими сущностями. Кроме этого определены дополнительные абстракции, которые управляют доступом к этим сущностям. Далее они будут рассмотрены более подробно.

К элементам хранения данных относятся регистры, физическая память и внутренние временные переменные. Они задаются в виде массивов, где каждый элемент является битовым вектором фиксированной длины. Регистры и физической памяти определяют текущее состояние микропроцессора, которое также называют *контекстом исполнения.* В процессе работы эмулятора может поддерживаться несколько копий контекстов. Во-первых, это необходимо для эмуляции работы *многоядерных* микропроцессоров, где каждый поток манипулирует собственным набором регистров. Во-вторых, для решения некоторых задач требуется исполнение команд во временном контексте с последующей отменой изменений. У этих двух случаев есть следующие важные отличия. В первом случае для каждого ядра на протяжении всего сеанса работы эмулятора поддерживается отдельный контекст. При этом некоторые элементы контекста, такие как память, могут разделяться между ядрами. Во втором случае временный контекст создается как копия текущего контекста отдельного ядра и существует в течение ограниченного периода. При этом эта копия является уникальной, и все изменения являются временными и не видны другим ядрам. Управление контекстами осуществляется при помощи абстракции, называемой *менеджером контекстов*. Он позволяет задавать количество поддерживаемых контекстов и выбирать текущий контекст (как постоянный или как временный). К текущему контексту предоставляется внешний доступ для осуществления следующих действий: (1) размещение данных в памяти и (2) чтение и запись значений заданных регистров (например, счетчика команд) и ячеек памяти. За решение этих задач отвечают сущности, именуемые *менеджер размещения данных* и *провайдер состояния*.

Память заслуживает отдельного внимания. Т.к. спецификации на языке nML не позволяют описать механизмы управления памятью (включая трансляцию адресов), построенный на их основе эмулятор использует упрощенную схему работы с памятью. Эта схема предполагает, что виртуальные адреса равны физическим и доступ осуществляется напрямую к физической памяти. Однако среда может быть расширена дополнительными средствами моделирования, которые используют язык MMUSL [7] для создания модели подсистемы управления памятью. Эта модель включает в себя отдельный эмулятор, который должен быть интегрирован в эмулятор системы команд. Интеграция осуществляется следующим образом. Эмулятор системы команд имеет точку соединения, которая позволяет зарегистрировать внешний *обработчик запросов к памяти*, который вызывается при каждом обращении к массиву физической памяти. Обработчик переадресует все запросы к эмулятору подсистемы управления памятью, который осуществляет трансляцию виртуальных адресов, кэширование данных и доступ к массиву физической памяти по физическим адресам. Как и эмулятор системы команд, эмулятор подсистемы памяти позволяет поддерживать несколько контекстов, управление которыми осуществляется совместно для обоих эмуляторов.

На рисунке 17 показаны основные компоненты, организующие работу с элементами хранения данных, и связи между ними.

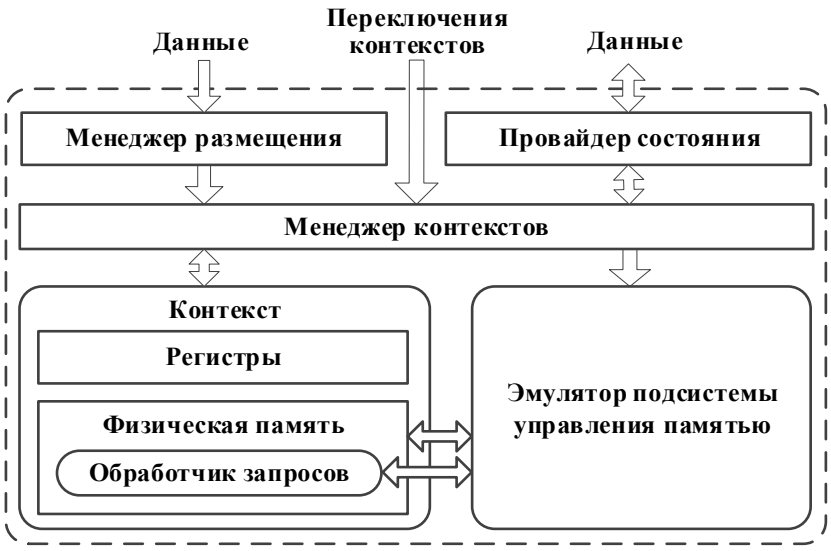


Рисунок 17. Компоненты эмулятора, организующие работу с элементами хранения данных

Команды, как уже было сказано, являются композитными объектами, состоящими из операций, режимов адресации и непосредственных значений, для обозначения которых используется общий термин *примитив*. Конкретный вызов команды создается путем комбинирования примитивов по правилам, заданным в спецификациях и описанных метаданными. Поддерживается два способа создания этих объектов: (1) на основе запросов, составленных из метаданных, и (2) путем декодирования данных в бинарном формате. Эти задачи возложены на компоненты, известные как *конфигуратор вызовов* и *декодер* соответственно. При исполнении команды взаимодействуют с текущим контекстом, предоставляемым им менеджером контекстов, читая и записывая данные. Кроме этого они вносят информацию о событиях, которые произошли во время их выполнения, в *журнал событий*. Поддерживаемые события включают: исполнение команды, запись в регистры и доступ к памяти. Помимо этого каждая инструкция может быть сохранена в текстовом и бинарном формате. Компоненты эмулятора, организующие работу с командами, и их взаимодействие показаны на рисунке 18.

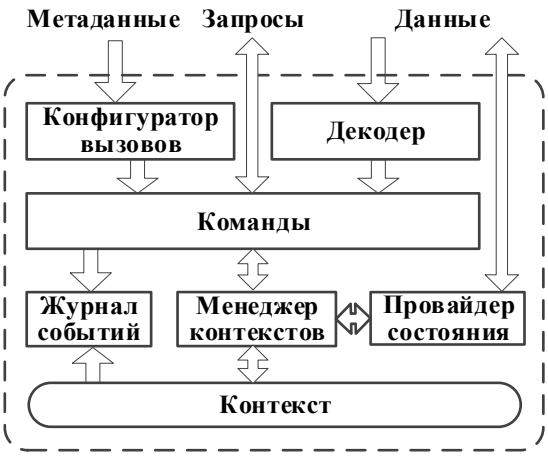


Рисунок 18. Компоненты эмулятора, организующие работу с командами

Необходимо заметить, что эмулятор отвечает только за доступ к элементам хранения данных и выполнение отдельных инструкций. Задача управления эмуляцией (переключение контекстов и выбор команд для выполнения) является независимой от конфигурации микропроцессора и ее решение возложено на компоненты среды генерации, о которых будет рассказано далее.

Модель тестового покрытия

Модель тестового покрытия содержит информацию, необходимую для покрытия различных путей выполнения команд, которую также называют *тестовым знанием*. Это знание является специфичным для конкретного микропроцессора и синтезируется на основе анализа формальных спецификаций. Тестовое знание представлено в виде таблицы *тестовых ситуаций*, соответствующих конкретным командам. Каждая тестовая ситуация ассоциируется с одним из возможных путей исполнения данной команды и описывается в виде ограничения на значения ее входных аргументов, которое должно быть удовлетворено для покрытия данного пути.

Ограничения описываются при помощи библиотеки Fortress [84], разработанной автором. Эта библиотека позволяет задавать ограничения в виде формул, представленных в форме синтаксических деревьев, и вызывать внешние решатели. Ограничения сохраняются в формате XML и загружаются по требованию. Кроме того поддерживается возможность сохранять ограничения в виде файлов на языке SMT-LIB (Satisfiability Modulo Theories Library) [85], который используется в качестве основного средства взаимодействия с внешними решателями ограничений.

В основе представления ограничений лежит *форма статического единственного присваивания* (Static Single Assignment, SSA) [86], построенная на основе nML-спецификаций. Т.к. команды не являются монолитными, а состоят из операций и режимов адресации, SSA форма разбита на части, соответствующие отдельным примитивам. Таким образом, ограничения собираются путем объединения SSA-форм примитивов, составляющих команды. При этом так как ограничению соответствует один из множества возможных путей в графе потока управления, φ-функции сводятся к единственному варианту, а неиспользуемые блоки кода исключаются.

Интерфейс, используемый для доступа к модели тестового покрытия, устроен следующим образом. Он позволяет запрашивать доступ к объектам, описывающим ограничения, по идентификатору команды и идентификатору

ограничения. Имена ограничений соответствуют меткам в коде формальных спецификаций. Кроме этого существуют предопределенные идентификаторы, обозначающие некоторые классы тестовых ситуаций (например, выполнение без исключений или выполнение с исключением определенного типа).

Для расширенных вариантов формальных спецификаций, таких как спецификации подсистемы управления памятью, строятся отдельные модели тестового покрытия. Они могут использовать ограничения других типов и другие форматы их представления. Интеграция в общую модель осуществляется следующим образом. Модель тестового покрытия нового типа регистрируется через точку соединения в модели микропроцессора как отдельная сущность. Эта сущность затем используется компонентами среды генерации, реализующими техники построения тестов на основе данного типа знания.

3.1.2 Анализаторы формальных спецификаций

Модель строится на основе формальных спецификаций. Необходимая для этого информация собирается при помощи *анализаторов формальных спецификаций*. Они реализованы в виде *компилятора переднего плана (front-end compiler)* [87, 88], состоящего из набора применяемых последовательно компонентов. Этот набор включается в себя:

1. лексический анализатор;
2. синтаксический анализатор;
3. семантический анализатор;
4. анализаторы внутреннего представления.

Лексический анализатор преобразует последовательности символов, считанные из файлов формальных спецификаций, в последовательности *лексем*. Затем синтаксический анализатор строит из этих лексем *дерево разбора*. После этого дерево разбора обрабатывается семантическим анализатором, который строит *внутреннее представление (internal representation)*, представляющее собой *абстрактное синтаксическое дерево*. Внутреннее представление обрабатывается набором *анализаторов внутреннего представления*, которые

извлекают из него всю интересующую информацию. Эта информация может быть представлена в отдельном специальном формате или добавлена в объекты внутреннего представления в качестве атрибутов. Обновленное внутреннее представление и другие результаты работы анализаторов передаются генератором кода модели, о которых будет рассказано в следующем подразделе. Схема взаимодействия перечисленных компонентов показана на рисунке 19.

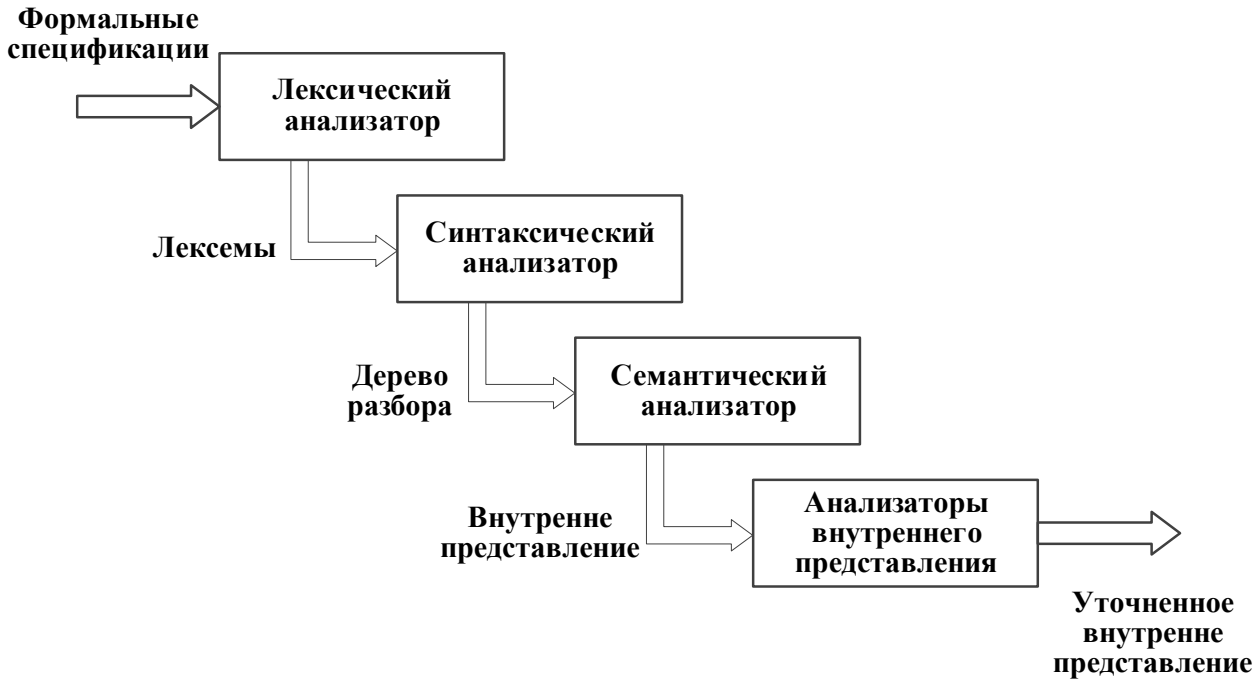


Рисунок 19. Компоненты анализатора формальных спецификаций и их взаимодействие

Перечисленный набор компонентов разрабатывается для каждого из поддерживаемых языков формальных спецификаций, используя общую методологию и набор специальных библиотек. Отдельного внимания заслуживают анализаторы внутреннего представления. Они предназначены для извлечения информации об определенных свойствах сущностей, описанных при помощи спецификаций. По мере расширения набора исследуемых свойств, множество анализаторов данного типа может пополняться пользовательскими компонентами.

Т.к. спецификации системы команд, созданные при помощи языка nML, являются основным источником информации об архитектуре тестируемого микропроцессора, анализатор nML-спецификаций является ключевым

компонентом среды моделирования. Его устройство будет подробно рассмотрено далее. Дополнительные анализаторы, такие как анализатор спецификаций подсистемы управления памятью на языке MMUSL, реализованы аналогичным образом, используя аналогичные компоненты. Эти анализаторы запускаются после анализатора nML-спецификаций и имеют доступ к результатам его работы. Дополнительные анализаторы реализованы в виде *расширений (plug-in)* среды и используют входные файлы определенного типа. Внутренние представления, полученные в результате их работы, дополняют друг друга и интегрируются в единый объект. На рисунке 20 показана схема совместной работы нескольких анализаторов формальных спецификаций.

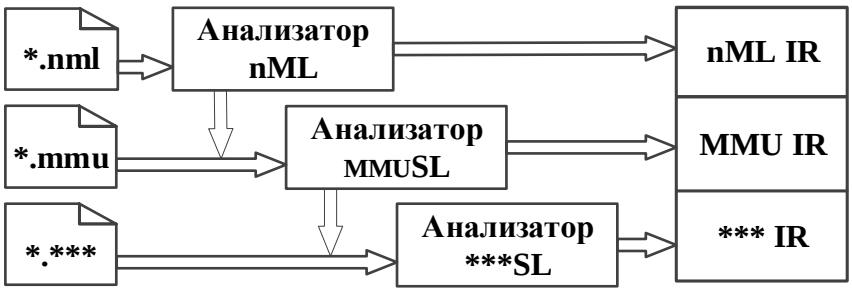


Рисунок 20. Схема совместной работы нескольких анализаторов формальных спецификаций

Анализаторы формальных спецификаций разработаны при помощи инструмента ANTLR [89, 90, 91], который позволяет сгенерировать лексический, синтаксический и семантический анализаторы на основе описаний грамматики языка. Данный инструмент был выбран т.к. он разработан на языке Java и является открытым. Для упрощения разработки анализаторов MicroTESK предоставляет набор библиотек, которые реализуют логику работы препроцессора, обработки ошибок, управления таблицей символов, а также служат строительными блоками для внутреннего представления.

Лексический анализатор построен на основе грамматики ANTLR, которая описывает правила разбиения потока символов на лексемы. Данная грамматика включает правила обработки директив препроцессора, которые позволяют включить или исключить из обработки отдельные фрагменты кода или файлы. Результатом работы данного анализатора является поток лексем. Грамматика лексического анализатора разделена на две части: (1) базовую и

(2) специфическую для nML. Первая содержит описания лексем, общие для многих языков, и правила обработки директив препроцессора. Вторая содержит описания лексем, используемых языком nML. При создании лексических анализаторов для новых языков базовая часть используется повторно и дополняется описаниями лексем, специфичными для данного языка.

Синтаксический анализатор построен на основе грамматики ANTLR, описывающей правила построения дерева разбора. Кроме этого данные правила осуществляют регистрацию идентификаторов в таблице символов и связанные с этим проверки (отсутствие переопределений, применимость идентификаторов определенного типа в данном контексте). Логика управления таблицей символов и правила грамматики, общие для многих языков, такие как правила разбора инструкций (statements) и выражений (expressions), помещаются в отдельные компоненты, которые могут быть повторно использованы при создании синтаксических анализаторов для новых языков.

Семантический анализатор построен на основе грамматики ANTLR, которая описывает правила обхода дерева разбора, построенного на предыдущем этапе. Данные правила обращаются к фабрикам внутреннего представления, которые осуществляют необходимые семантические проверки и конструируют объекты внутреннего представления.

Внутреннее представление включает в себя набор таблиц, которые хранят описания следующих сущностей: (1) константы, (2) типы данных, (3) элементы хранения данных, (4) режимы адресации и (5) операции. Отдельного внимания заслуживают режимы адресации и операции. Они содержат ссылки на другие примитивы (операции, режимы адресации), описывающие типы их аргументов. Каждый примитив имеет набор атрибутов, которые описывают их функции и представлены в виде абстрактных синтаксических деревьев. Тело атрибута может содержать вызовы атрибутов примитивов, переданных текущему примитиву в качестве аргументов. Описание вызова содержит в себе ссылку на вызываемый атрибут и содержащий его примитив. Таким образом, примитивы

образуют ориентированный ациклический граф (directed acyclic graph), истоком которого является операция *instruction*.

Набор сущностей внутреннего представления, построенного семантическим анализатором, в точности соответствует набору сущностей, описанных nML-спецификациями. Однако для построения модели требуется дополнительная информация, которая синтезируется в результате анализа внутреннего представления. Для решения этой задачи разработан набор анализаторов, которые осуществляют обход внутреннего представления и извлекают необходимую информацию. В настоящее время набор включает в себя анализаторы, которые решают следующие задачи:

1. синтез правил построения сокращенных вызовов операций;
2. определение типов аргументов (входной или выходной);
3. определение операций, порождающих исключения;
4. определение операций, осуществляющих передачу управления;
5. определение операций, осуществляющих доступ к памяти;
6. определение формата бинарного представления примитивов;
7. построение набора ограничений для каждого примитивов.

Набор анализаторов может быть расширен пользовательскими анализаторами. Все анализаторы имею общую архитектуру. Общие части вынесены в повторно используемые компоненты. Анализаторы включают в себя два основных компонента: (1) *обходчик (walker)* и (2) *посетитель (visitor)*. Обходчик осуществляет обход внутреннего представления, вызывая методы посетителя для каждого объекта. Посетитель анализирует переданные ему объекты и накапливает информацию об их свойствах, которую сохраняет в атрибутах примитивов и или в дополнительных структурах (например, создает SSA-форму).

3.1.3 Генераторы кода и библиотеки моделирования

На основе внутреннего представления и извлеченной из него дополнительной информации осуществляется конструирование модели, которая представляет собой набор Java-классов, построенных на основе

специальных библиотек моделирования. За конструирования модели отвечают генераторы кода, которые аналогично анализатором внутреннего представления осуществляют его обход и генерирует на основе полученной информации Java-код. Каждый генератор отвечает за построение отдельной части модели. Поддерживаются следующие генераторы:

1. генератор внешнего интерфейса модели;
2. генератор метаданных;
3. генератор эмулятора;
4. генератор декодера;
5. генератор модели покрытия.

Набор генераторов может быть расширен в случае необходимости. Генераторы реализованы на основе библиотеки StringTemplate [92], которая осуществляет генерацию кода по шаблонам. Логика обхода внутреннего представления реализована так же, как в анализаторах внутреннего представления, и использует те же компоненты.

Рассмотрим более подробно библиотеки, используемые для конструирования модели. Набор библиотек включает в себя:

- Библиотеки описания данных и операций над ними. Данные представлены в виде битовых векторов, для хранения которых используются классы библиотеки Fortress [84]. Операции целочисленной арифметики также реализованы в этой библиотеке. Для описания операций над числами с плавающей точкой используется библиотека Java SoftFloat [93], разработанная в ИСП РАН. Данная библиотека была разработана как Java-версия библиотеки Berkeley SoftFloat [94], реализующей стандарт IEEE 754 [60].
- Библиотека описания элементов хранения данных. Включает в себя классы для описания регистров и физической памяти. Данные классы позволяют регистрировать обработчики запросов, которые могут быть использованы для интеграции расширений и регистрации событий.

- Библиотека описания исполняемых команд. Служит основанной для эмулятора системы команд. Позволяет описывать режимы адресации и операции, а также фабрики, отвечающие на создание их экземпляров и конструирование на их основе вызовов команд. Логика работы команд описывается при помощи двух описанных выше библиотек.
- Библиотека компонентов декодера. Позволяет построить компоненты, обеспечивающие построение вызовов команд на основе их бинарного представления.
- Библиотека описания метаданных. Содержит классы, описывающие свойства всех сущностей системы команд.
- Библиотека описания ограничений, основанная на библиотеке Fortress [84]. Позволяет описывать ограничения в виде SMT-формул.
- Библиотека внешнего интерфейса модели. Обеспечивает доступ ко всем частям модели.
- Библиотеки описания эмулятора подсистемы памяти и ее модели покрытия.

Библиотеки могут эволюционировать, и их набор может расширяться по мере необходимости. При этом данные изменения не затронут пользователей инструмента.

3.2 Среда тестирования

Среда тестирования отвечает за построение тестовых программ на основе модели микропроцессора и шаблонов тестовых программ. Она состоит из *анализатора (template analyzer)* и *обработчика (template processor)* шаблонов. Первый строит внутреннее представление шаблонов, а второй генерирует на его основе тестовые программы. Обработка внутреннего представления осуществляется на уровне отдельных блоков. Для каждого из них строятся последовательности команд, которые затем исполняются на программном эмуляторе, предоставляемым моделью микропроцессора. Это позволяет отслеживать состояние микропроцессора и учитывать его в процессе

генерации. Основные компоненты, участвующие в построении последовательностей команд включают в себя:

- итератор последовательностей команд (sequence iterator);
- распределитель регистров (register allocator);
- обработчик последовательностей команд (sequence processor);
- исполнитель последовательностей команд (sequence executor);
- генератор данных (data generator);
- построитель встроенных проверок (self-check maker);

Перечисленные компоненты будут подробно рассмотрены в следующих подразделах.

3.2.1 Анализатор шаблонов тестовых программ

Анализатор тестовых (template analyzer) шаблонов интерпретирует код тестовых шаблонов и строит для них *внутреннее представление*. Для этого используется *интерпретатор* JRuby [95], который интегрирован в анализатор. Этот интерпретатор разработан на языке Java и позволяет в процессе исполнения Ruby-кода загружать Java-компоненты и вызывать их методы. Схема анализатора шаблонов тестовых программ показана на рисунке 21.

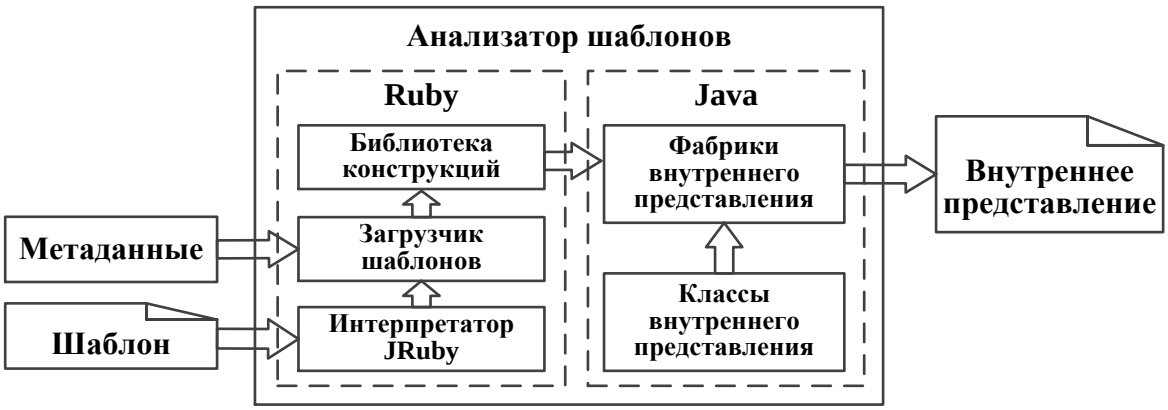


Рисунок 21. Схема анализатора шаблонов тестовых программ

Анализатор шаблонов включает в себя следующие компоненты:

- интерпретатор JRuby;
- загрузчик шаблонов (разработан на Ruby);
- библиотеки для описания конструкций языка (разработаны на Ruby);
- классы внутреннего представления (разработаны на Java);

- фабрики внутреннего представления (разработаны на Java).

Анализ шаблонов тестовых программ осуществляется следующим образом. При запуске анализатора интерпретатор исполняет загрузчик шаблонов, который загружает класс анализируемого шаблона. В процессе загрузки класса шаблона, загрузчик на основе метаданных из модели микропроцессора создает для него методы, отвечающие за конструирование архитектурно-зависимых объектов внутреннего представления (команды, режимы адресации и т.п.). После этого загрузчик вызывает методы класса шаблона, описывающие отдельные части тестовой программы (пролог, эпилог и тестовые примеры). Эти методы, используя различные конструкций языка описания шаблонов, реализованные в Ruby-библиотеках, обращаются к фабрикам внутреннего представления для конструирования соответствующих объектов. После этого внутреннее представление передается для дальнейшей обработки.

Подход, при котором сначала строится внутреннее представление, а затем осуществляется его обработка, позволяет сделать реализацию генератора тестовых программ независимой от языка описания шаблонов. То есть для поддержки нового языка описания шаблонов, при условии наличия для него аналогичного интерпретатора, достаточно разработать загрузчик шаблонов и библиотеку языковых конструкций. Например, можно создать реализацию, в которой шаблоны будут разрабатываться на языке Python [79], а для их анализа будет использоваться интерпретатор Jython. При этом реализация остальных частей генератора тестовых программ не изменится.

3.2.2 Внутреннее представление шаблонов тестовых программ

Внутреннее представление, полученное в результате анализа шаблона, представляет собой набор объектов, описывающих свойства отдельных частей тестовых программ (пролога, эпилога, тестовых примеров и секций диспетчеризации). Каждая из этих частей представлена в виде *блока*, который описывает способ построения ее кода. Блоки снабжены набором атрибутов (пары ключ-значение), которые задают техники, применяемые для обработки

содержимого этих блоков. Блоки содержат в себе список из структурных элементов, на основе которых будет построен код соответствующих частей тестовой программы. В качестве структурных элементов выступают вложенные блоки, команды и элементы специального назначения. Вложенные блоки обрабатываются рекурсивно.

Команды состоят из *примитивов*, описывающих их составные части (операции и режимы адресации). Структура связей между частями соответствует структуре, описанной метаданными. Примитивы снабжены следующими атрибутами: имя, список передаваемых им аргументов и объект метаданных, описывающий их свойства. В качестве аргументов выступают примитивы или непосредственные значения (immediate value). Они могут быть жестко определены или помечены как неопределенные. В последнем случае для них используется специальный объект-заглушка, который заменяется на конкретный объект в процессе дальнейшей обработки. Помимо этого для команды может быть задана связанная с ней тестовая ситуация. Тестовые ситуации содержат набор атрибутов, задающие настройки, которые будут использоваться при ее обработке.

К элементам случайного назначения относятся сущности следующих типов: секции данных; метки; директивы ассемблера (смещения, выравнивания); текст; служебные инструкции генератора и т.д. Они помещаются в один объект вместе со следующей за ними командой, чтобы обеспечить фиксацию их местоположения относительно этой команды. При этом возможно также создавать структурные элементы, содержащие только перечисленные сущности. На рисунке 22 показан структурный элемент, описывающий команду ADDI архитектуры MIPS с неизвестными регистрами и непосредственным аргументом, а также связанные с ней метку и директиву задания смещения.

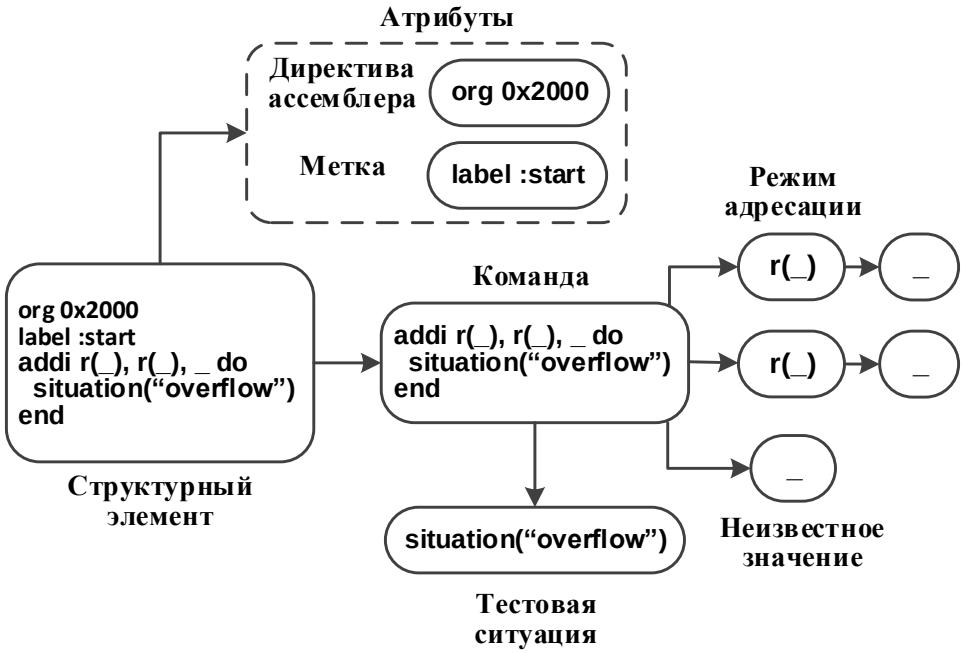


Рисунок 22. Объекты внутреннего представления, описывающие команду

Кроме блоков, которые были описаны выше, внутреннее представление содержит объекты, описывающие различные препараторы и компараторы. Они также из структурных элементов такого же вида. При этом команды могут использовать неизвестные значения и режимы адресации. В процессе построения кода для препараторов и компараторов они будут заменены на объекты, описывающие конкретные значения или примитивы (например, режим адресации, описывающий инициализируемый регистр, и записываемое в него значение).

3.2.3 Обработчик внутреннего представления

Построенное анализатором шаблонов внутреннее представление передается *обработчику внутреннего представления (template processor)*, который осуществляет на его основе построение тестовых программ.

Тестовая программа представляет собой список последовательностей команд, описывающих ее отдельные части (пролог, эпилог, тестовые примеры, секции диспетчеризации). Последовательности команд строятся в результате обработки блоков внутреннего представления, описывающих соответствующие части программы. Построенные последовательности команд исполняются на программном эмуляторе модели микропроцессора. Информация о состоянии

эмулятора используется при построении тестовых примеров и встроенных проверок для них. Поэтому, чтобы обеспечить правильность состояния эмулятора в момент построения конкретного тестового примера, построение кода для отдельных частей осуществляется в порядке их исполнения.

Рисунок 23 иллюстрирует разбиение шаблона тестовой программы на блоки. При помощи стрелок показана структура переходов управления между частями программы, описываемыми блоками. На рисунке заштрихованы блоки, для которых структура кода фиксирована и при его построении не используется информация о состоянии эмулятора. Двойная штриховка обозначает, что адреса, по которым будет располагаться код, заранее определены. Блоки, которые остались не заштрихованными, соответствуют описаниям тестовых примеров.

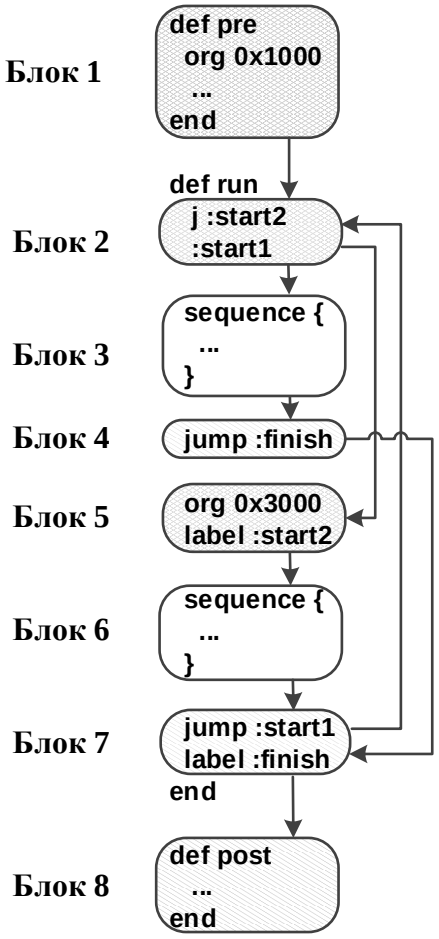


Рисунок 23. Схема разбиения шаблона на блоки и структура переходом между ними

Схема обработки блоков внутреннего представления выглядит следующим образом:

1. Сначала обрабатываются блоки, при построении кода для которых не используется информация о состоянии эмулятора. К таким блокам относятся пролог, эпилог и секции диспетчеризации. Последние содержат переходы друг в друга, поэтому перед началом исполнения их код должен быть построен и размешен в памяти эмулятора. Важным моментом является то, что адрес размещения кода в памяти должен быть заранее известен. Это возможно только, если последовательность команд начинается с директивы, задающей ее смещение (обычно это директива .org), или последовательность размещается следом за другой последовательностью с известным смещением. Для эпилога и некоторых секций диспетчеризации, смещение которых зависит от размера кода тестовых примеров, за которыми они следуют, не может быть определено на этой стадии. Поэтому построенные для них последовательности команд не размещается в эмуляторе, а помечаются как ожидающие размещения.
2. На следующем этапе осуществляется построение кода тестовых примеров и встроенных проверок для них. Для того чтобы выбрать обрабатываемый блок, и привести эмулятор в необходимое начальное состояние, на эмуляторе запускается исполнение построенного ранее кода. Исполнение приостанавливается при достижении адреса конца одной из построенных ранее последовательностей команд, за которым отсутствует размещенный код. После этого выбирается блок, который следует за блоком, на основе которого была построена последняя исполняемая последовательность команд. В результате обработки этого блока строится одна или несколько последовательностей команд, которые сразу же размещаются в памяти эмулятора и исполняются. После исполнения каждой из последовательностей, на основе информации о состоянии эмулятора и описаний компараторов строятся встроенные проверки. После обработки блока, выбирается другой, следующий за ним блок. Если этот блок является секцией диспетчеризации, для которой уже был построен код, то

этот код размещается в памяти эмулятора и исполняется. После этого выбирается блок, который следует за блоком, на котором остановилось исполнение и процесс повторяется заново.

3. Если моделируемый микропроцессор содержит несколько ядер, то исполнение кода запускается по очереди на разных ядрах. При этом синхронизация между ядрами осуществляется на уровне отдельных последовательностей. При обработке блока, выбирается то ядро, для которого исполнение достигло этого блока. Если таких ядер несколько, то выбирается первое из них.
4. После того, как исполнение дойдет до эпилога, генерация завершается. Если при этом некоторые блоки остались не обработанными, так как не были достигнуты в процессе исполнения, то они обрабатываются, используя текущее состояния эмулятора. При этом построенные последовательности не исполняются. После этого весь построенный код печатается в виде программы на языке ассемблера. Если в процессе исполнения построенного кода на эмуляторе произошли ошибки (зацикливания, переходы по невалидным адресам, нарушения инвариантов и т.д.), генерация прерывается с соответствующим сообщением об ошибке.

Обработка блоков заключается в построении последовательностей команд, которое включает в себя распределение регистров. После этого осуществляется обработка тестовых ситуаций, заданных для команд, в процессе которой генерируются тестовые данные и строится инициализирующий код, размещающий их в регистры и память.

3.2.4 Итератор последовательностей команд

Каждый блок внутреннего представления порождает одну или несколько последовательностей команд. Пролог, эпилог и секции диспетчеризации имеют фиксированную структуру и, поэтому, обрабатываются по упрощенной схеме. Построенные для них последовательности команд полностью соответствуют их описанию в шаблонах.

Блоки тестовых примеров могут порождать множество последовательностей в зависимости от их описания и выбранных стратегий построения последовательностей. За построение последовательностей команд отвечает компонент, называемый *итератором последовательностей команд (sequence iterator)*.

Основная идея реализации этого компонента состоит в следующем. Каждый блок содержит список итераторов по последовательностям команд, описываемых при помощи его структурных элементов. Итератор последовательностей команд применяет к этому списку набор компонентов, при помощи которых на основе элементов списка строит новые последовательности команд. Команды, содержащиеся в блоке, эквивалентны итератору по одной последовательности из одного элемента. Для вложенных блоков строится отдельный итератор на основе их содержимого. Компоненты, применяемые для построения последовательностей, включают в себя: compositor, permutator, combinator, rearranger и obfuscator (описаны во второй главе). Для каждого компонента реализован набор стратегий (также описаны во второй главе), определяющих его поведение, который можно пополнять новыми реализациями. В примере 35 показано, как организована совместная работа перечисленных компонентов и как ним применяются стратегии, определяющие их поведение. Как можно увидеть, одни компоненты обрабатывают результаты работы других. При необходимости эта схема может быть расширена компонентами новых типов. Применяемые стратегии выбираются на основе атрибутов блока, соответствующих именам компонентов.

Пример 35. Код построения итератора последовательностей команд на основе списка итераторов

```
01:     public Iterator<List<T>> newIterator(final List<Iterator<List<T>>> iterators) {
02:       if (isSequence) {
03:         return new GeneratorObfuscator<>(
04:             new GeneratorSequence<>(iterators), obfuscatorStrategy);
05:       }
06:
07:       if (isIterate) {
08:         return new GeneratorObfuscator<>(
09:             new GeneratorRearranger<>(new GeneratorIterate<>(iterators), rearrangerStrategy),
10:             obfuscatorStrategy
11:             );
12:       }
13:
14:       return new GeneratorObfuscator<>(
15:           new GeneratorRearranger<>(
16:               new GeneratorCompositor<>(
17:                   new CombinatorPermutator<>(combinatorStrategy, permutatorStrategy),
18:                   compositorStrategy,
19:                   iterators),
20:               rearrangerStrategy),
21:           obfuscatorStrategy
22:           );
23:     }
```

3.2.5 Распределитель регистров

Регистры, используемые командами построенных последовательностей, могут быть не зафиксированы (то есть их индексы помечены как неизвестные значения). Выбор регистров осуществляется в процессе обработки последовательности команд при помощи компонента, называемого *распределителем регистров (register allocator)*. Этот компонент использует набор стратегий, которые реализуют конкретные способы выбора регистров и имеют следующий интерфейс:

Пример 36. Интерфейс стратегий выбора регистров

```
01:     public interface AllocationStrategy {
02:       <T> T next(
03:           final Collection<T> domain,
04:           final Collection<T> exclude,
05:           final Collection<T> used,
06:           final Map<String, String> attributes);
07:
08:       <T> T next(
09:           final Supplier<T> supplier,
10:           final Collection<T> exclude,
11:           final Collection<T> used,
12:           final Map<String, String> attributes);
13:     }
```

Стратегии осуществляют выбор номеров регистров на основе следующей информации: (1) множество допустимый значений (domain); (2) множество значений, исключаемых из выбора (exclude); (3) множество значений, которые

уже были выбраны (used); (4) набор атрибутов, задающих варианты работы для используемой стратегии (attributes).

Множество выбранных регистров включает в себя все жестко заданные регистры. Для того, чтобы они учитывались при выборе, обработка последовательности осуществляется в два этапа. На первом составляется множество жестко заданных регистров. А на втором на основе этой информации осуществляется выбор регистров, которые не были заданы жестко, путем применения соответствующих стратегий выбора.

Распределитель регистров поддерживает список стратегий, в который можно добавлять новые реализации.

3.2.6 Обработчик последовательностей команд

За генерацию входных данных и построение инициализирующего кода для последовательностей команд отвечает компонент, называемый *обработчиком последовательностей команд (sequence processor)*.

Первая задача, которую он решает, состоит в построении ограничений для тестовых ситуаций, связанных с командами. Ограничения, используемые для описания тестовых ситуаций различного типа, имеют разный формат. Поэтому для обработки тестовых ситуаций каждого типа используется отдельный компонент, называемый *генератором ограничений (constraint generator)*. Он просматривает последовательности команд и строит ограничения для связанных с ними тестовых ситуаций определенного типа. При этом для одной и той же тестовой ситуации может быть построено несколько вариантов ограничений. Перед обработкой генераторами ограничений последовательности команд разделяются: для каждого генератора выбираются команды определенного типа. В процессе обработки переданной ему последовательности генератор ограничений обращается к *модели тестового покрытия*. Результатом его работы являются последовательности команд, аннотированные ограничениями, построенными для заданных тестовых ситуаций. Каждая последовательность содержит уникальную комбинацию ограничений. На основе последовательностей, возвращаемых генераторами

ограничений, строятся уточненные последовательности команд, которые описывают тестовые воздействия. Для этого при помощи комбинатора, заданного в шаблоне, строятся комбинации из исходных последовательностей, которые затем конкатенируются. Схема построения ограничений для последовательности команд показана на рисунке 24.

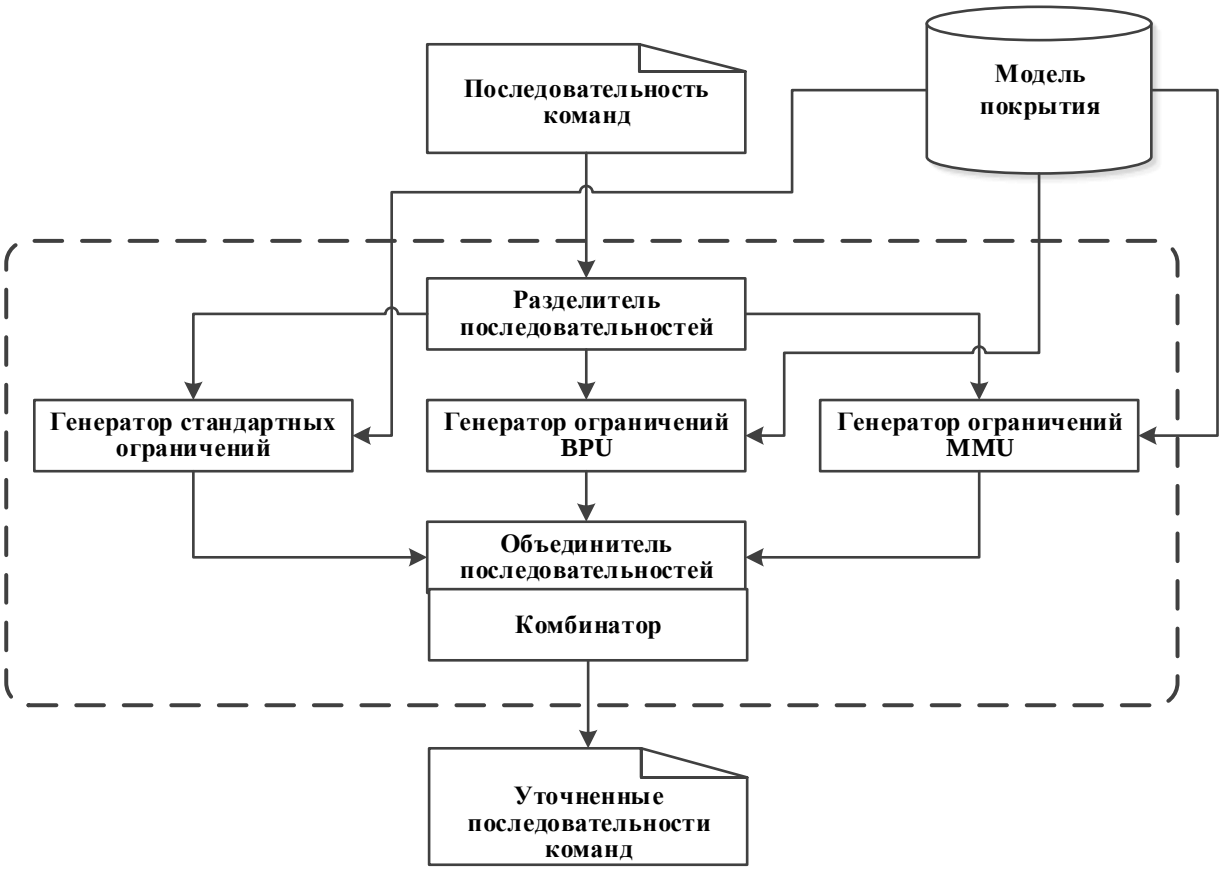


Рисунок 24. Схема построения ограничений для последовательности команд

На основе построенных ограничений осуществляется генерация входных данных для команд, на основе которых затем строится инициализирующий код. В процессе генерации осуществляется исполнение обрабатываемых команд на эмуляторе. Это необходимо, чтобы обеспечить обработку команд в соответствии с порядком их исполнения и использовать информацию о состоянии эмулятора при разрешении ограничений. За исполнение команд отвечает компонент, называемый *исполнителем последовательностей команд (sequence executor)*. За генерацию данных и построение инициализирующего кода отвечает *генератор данных (data generator)*. Эти компоненты подробно рассматриваются в следующих подразделах.

3.2.7 Исполнитель последовательностей команд

Эмулятор, реализованный моделью микропроцессора, предназначен для исполнения отдельных команд. Исполнением целых последовательностей команд управляет компонент, называемый *исполнителем последовательностей команд (sequence executor)*. Он может работать в двух режимах: (1) исполнение последовательности команд с целью обновления состояния эмулятора и (2) исполнение последовательности команд с целью генерации для нее данных и построения инициализирующего кода. В первом случае используется постоянный контекст эмулятора, во втором – временный. Общая схема исполнения последовательности команд состоит из следующих шагов:

1. последовательность команд размещается в памяти эмулятора;
2. создается точка остановки (breakpoint), соответствующая конечному адресу последовательности;
3. в счетчик команд записывается начальный адрес последовательности;
4. из памяти эмулятора извлекается команда, соответствующая счетчику команд;
5. команда исполняется, обновляя значение счетчика команд;
6. если достигнута точка остановки, исполнение приостанавливается, в противном случае повторяется шаг 4.

В режиме генерации данных в эту схему добавляется дополнительный шаг. Перед исполнением команды (шаг 5) проверяется было ли обработано связанное с ней ограничение. Если оно еще не было обработано, то осуществляется генерации данных и построение инициализирующего кода. При этом построенный код размещается в память (следом за последовательностью) и исполняется. После этого счетчику команд присваивается адрес выбранной ранее команды и эта команда исполняется.

Исполняемые команды могут быть описаны при помощи шаблона в явном виде или они могут быть описаны в виде блока данных, на который передается управление. В первом случае для них строятся соответствующие исполняемые объекты, которые кэшируются и используются при каждом

обращении. Во втором случае объект исполняемой команды строится при первом вызове команды при помощи декодера, встроенного в эмулятор.

3.2.8 Генератор данных

Обработка ограничений осуществляется при помощи компонента, называемого *генератором данных (data generator)*. Он решет две следующих задачи:

- разрешения ограничения с целью генерации данных определенного вида;
- построения инициализирующего кода на основе этих данных.

Задача разрешения ограничений решается следующим образом. За разрешение каждого типа ограничений отвечает отдельный компонент, называемый *решателем ограничений (constraint solver)*. Решатели ограничений регистрируется в специальном каталоге, реализованном системой TestBase (разработана в ИСП РАН) [96] . Данная система обрабатывает запросы на генерацию данных, выбирая подходящий решатель ограничений. Таким образом, реализация генератора данных не зависит от типов поддерживаемых им решателей ограничений. Запрос к системе TestBase содержит в себе: имя команды, список ее аргументов (фиксированных и неизвестных), имя используемого решателя и описание ограничения (идентификатор и дополнительные атрибуты). Список поддерживаемых типов ограничений включает в себя:

- ограничения на значения выходных аргументов (случайные значения из заданного диапазона, значения особого вида);
- ограничения на ситуации в работе операций арифметики с плавающей точкой;
- ограничения на пути исполнения команд (на основе nML-спецификаций);
- ограничения на сценарии обработки запросов к памяти (на основе MMUSL-спецификаций);
- ограничения на трассы исполнения в структуре переходов для команд ветвления.

Реализация решателя ограничений зависит от типа ограничения. Решатели ограничений, описанных при помощи модели тестового покрытия (на основе nML- и MMUSL-спецификаций), реализуют логику взаимодействия с соответствующими компонентами этой модели. Например, для решения ограничений на пути исполнения команд, описанных в виде SMT-формул, используются внешние SMT-решатели, такие как Z3 [97] и CVC4 [98]. Логика работа с этими решателями реализуется библиотекой Fortress [84], которая использует общий интерфейс для всех SMT-решателей, что делает компоненты генератора данных независимыми от внешних инструментов.

После того, как были сгенерированы данные, необходимо сгенерировать код, размещающий их в регистрах и памяти. Решение этой задачи возлагается на компоненты, называемые *построителями инициализирующего кода (initializer maker).* Они обеспечивают построение инициализирующего кода для данных определенного формата. Формат данных зависит от типа ограничения. При этом несколько разных решателей ограничений могут использовать один и тот же формат данных. При построении кода осуществляется поиск препаратора соответствующего типа, который должен быть объявлен в шаблоне. После этого по его образу и подобию строится код, который использует номера регистров, значения непосредственных аргументов, также блоки данных, соответствующие результатам решения ограничения.

3.2.9 Построитель встроенных проверок

За построение встроенных проверок для последовательностей команд, описывающих тестовые воздействия, отвечает компонент, называемый *построителем встроенных проверок (self-check maker).* Для решения этой задачи он использует информацию о состоянии микропроцессора, получаемую их эмулятора, и компараторы, описанные в шаблонах тестовых программ. Построитель встроенных проверок работает следующим образом. Он просматривает последовательность команд и составляет множество используемых ею регистров. После этого для каждого регистра создается код

встроенной проверки, основанной на описании соответствующего компаратора, и эталонного значения, полученного из эмулятора.

3.3 Расширяемость инструмента MicroTESK

Под расширяемостью понимается возможность интегрировать в генератор тестовых программ компоненты, реализующие новые техники генерации. Техники генерации могут быть реализованы одним независимым компонентом или группой связанных компонентов, совместно решающих общую задачу. При помощи независимых компонентов решаются следующие задачи:

- построение последовательностей команд (комбинаторы, пермутаторы, композиторы, перегруппировщики и обфускаторы);
- выбор регистров (стратегии распределения регистров).

Также, при выполнении указанных условий, при помощи независимого компонента можно решить следующие задачи:

- построение ограничений (если для них могут использоваться существующие решатели);
- разрешение ограничений (если при этом не требуется дополнительная информация о тестовом покрытии и построители инициализирующего кода нового типа);
- построение инициализирующего кода (при условии использования существующих решателей ограничений и типов данных);
- построение кода встроенных проверок (на основе имеющейся информации о состоянии микропроцессора).

Разработка группы связанных компонентов требуется в случаях, когда необходимо добавить поддержку новых типов ограничений. Компоненты, которые необходимо будет разработать, включают в себя:

- транслятор формальных спецификаций или его компоненты, отвечающие за извлечение ограничений и построение модели покрытия на их основе;

- библиотеки для моделирования новых свойств архитектуры микропроцессора;
- генератор ограничений;
- решатель ограничений;
- построитель инициализирующего кода (включая средства расширения языка описания шаблонов).

Перечисленные компоненты объединяются в *расширения*. Там образом для решения задачи поддержки новых типов ограничений, необходимо разработать расширение, включающее набор перечисленных компонентов. Примером разработки такого набора компонентов является поддержка генерации тестов для подсистемы памяти, использующая формальные спецификации на языке MMUSL.

3.4 Выводы

Инструмент MicroTESK использует предложенный метод для конструирования генераторов тестовых программ с предложенной архитектурой, осуществляющих генерацию на основе шаблонов на предложенном языке. Данный инструмент позволяет применять расширяемый набор техник генерации к широкому спектру микропроцессорных архитектур. Расширение возможностей инструмента осуществляется путем разработки новых компонентов.

Глава 4. Результаты практического применения

В данной главе описывается опыт практического применения инструмента, реализующего предложенный метод автоматизации конструирования генераторов тестовых программ. Инструмент был применен для создания генераторов тестовых программ для архитектур MIPS64, ARMv8, PowerPC и RISC-V. На основе анализа полученного опыта делается вывод о соответствии полученных результатов поставленным целям.

4.1 Генератор тестовых программ для архитектуры MIPS64

При помощи инструмента, реализующего предложенный метод, был сконструирован генератор тестовых программ для архитектуры MIPS64 [47] и ее разновидности Комдив64 [100]. Созданный генератор применяется в НИИСИ РАН для тестирования микропроцессора Комдив64. Особенности архитектуры MIPS64 и опыт разработки ее формальных спецификаций подробно рассмотрены в следующих подразделах.

4.1.1 Архитектура MIPS64

MIPS – это микропроцессорная архитектура, основанная на концепции проектирования RISC (reduced instruction set computer – «компьютер с сокращённым набором команд»), которая была разработана компанией MIPS Technologies (принадлежит компании Imagination Technologies [101] с 2013 года). За последние десятилетия было разработано множество версий данной архитектуры включая MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS32 и MIPS64. В настоящее время актуальными являются MIPS32 (32-битная версия) и MIPS64 (64-битная версия). Архитектура MIPS получила широкое распространение на рынке встраиваемых систем (компьютерные сети, телекоммуникации, игровые приставки, принтеры, цифровые камеры, карманные персональные компьютеры, и т.п.). Среди производителей электроники, использующих данную архитектуру, NEC, Philips и Toshiba. Из отечественных компаний, микропроцессоры, основанные на архитектуре MIPS, разрабатываются в НИИСИ РАН, «ЭЛВИС» и «Байкал Электроникс». Эти

микропроцессоры находят свое применение в военной и космической отраслях [100, 102].

4.1.2 Спецификации архитектуры MIPS64

Архитектура MIPS64 описывается в руководстве «MIPS™ Architecture For Programmers», включающем три тома [47, 103, 104], общий объем которых составляет примерно 1000 страниц. Система команд MIPS64 включает всего около 270 команд различного типа. Всю совокупность команд можно разбить на следующие группы:

- *арифметико-логические операции* (целочисленная арифметика, логические операции, сдвиги, пересылка данных, пустые команды);
- *команды ветвления* (условные и безусловные переходы);
- *команды доступа к памяти* (запись и чтение различных типов данных);
- *операции с плавающей точкой* (арифметика, пересылка данных, преобразование типов, ветвления).
- *системные команды* (исключения, пересылка данных системного сопроцессора CP0, обращения к буферам памяти, системные вызовы).

Команды имеют фиксированный размер 32-бита. Микропроцессоры MIPS64 используют слот задержки размером 4 байта. Архитектура MIPS64 определяет 32 64-битных регистра общего назначения (General Purpose Registers, GPRs), специальный 64-битный регистр для хранения значения счетчика команд (Program Counter, PC) и пару специальных регистров HI и LO для хранения результатов умножения и деления. Кроме того для модуля арифметики с плавающей точкой определены 32 регистра общего назначения (Floating Point General Purpose Registers, FPRs) размером 64-бит и 32 контрольных регистра (Floating Point Control Registers, FCRs) размером 32-бита для хранения флагов, выставляемых операциями с плавающей точкой. Помимо этого определены специальные регистры для сопроцессора CP0, предназначенные для хранения данных, используемых системными вызовами.

Все регистры MIPS64 и около 90% команд были специфицированы на языке nML. В приведенном ниже примере демонстрируется описание регистров общего назначения, режима адресации для доступа к ним и команды целочисленного сложения.

Пример 37. Спецификация регистров общего назначения и команды ADD архитектуры MIPS64

```
01:     // Тип данных для двойного слова
02:     type DWORD = card(64)
03:     // Регистры общего назначения (32 двойных слова)
04:     reg GPR [32, DWORD]
05:     // Временная переменная для хранения промежуточных результатов
06:     var temp33[card(33)]
07:     // Режим адресации для доступа к регистрам общего назначения
08:     mode R (i: card(5)) = GPR[i]
09:       syntax = format("r%d", i)
10:       image  = format("%5s", i)
11:     // Команда сложения с переполнением
12:     op add (rd: R, rs: R, rt: R)
13:       syntax = format("add %s, %s, %s", rd.syntax, rs.syntax, rt.syntax)
14:       image  = format("000000%5s%5s%5s00000100000", rs.image, rt.image, rd.image)
15:       action = {
16:         if sign_extend(WORD, rs<31>) != rs<63..32> ||
17:            sign_extend(WORD, rt<31>) != rt<63..32> then
18:           unpredicted;
19:         endif;
20:         temp33 = rs<31>::rs<31..0> + rt<31>::rt<31..0>;
21:         if temp33<32> != temp33<31> then
22:           exception("IntegerOverflow");
23:         else
24:           rd = sign_extend(DWORD, temp33<31..0>);
25:         endif;
26:       }
```

Помимо спецификаций системы команд, была разработана формальная спецификация подсистему управления памятью на языке MMUSL. Данная спецификация включает в себя описание буферов трансляции адреса JTLB и DTLB, буферов кэш-памяти L1 и L2, сегментов kseg0, kseg1, xkphys и useg, а также логики обработки операций чтения и записи. В примере 38 показан фрагмент описания логики обработки операции чтения подсистемой памяти MIPS64.

Пример 38. Фрагмент описания логики обработки операции чтения подсистемой памяти MIPS64

```
01:     mmu MMU (va: VA) = (data: DATA_SIZE)
02:       var pa: PA;
03:       var line: DATA_SIZE;
04:       var l1Entry: L1.entry;
05:       read = {
06:         pa = TranslateAddress(va);        // Трансляция адреса
07:         if IsCached(pa.cca) == 1 then
08:           if L1(pa).hit then              // Обращение к L1
09:             l1Entry = L1(pa);
10:             line = l1Entry.DATA;
11:           else
12:             line = MEM(pa);
13:             l1Entry.TAG = pa.paddress<...>; // Обновление L1
14:             l1Entry.DATA = line;
15:             L1(pa) = l1Entry;
16:           endif;
17:         else
18:           line = MEM(pa);
19:         endif;
20:         data = line;
21:       }
22:       write = { … }
```

Трудоемкость специфицирования архитектуры MIPS64 составила около 4 человеко-месяцев. Всего было описано 235 команд и структура подсистемы памяти. Размер разработанных спецификаций составил 3999 строк для системы команд (nML) и около 267 строк для подсистемы памяти (MMUSL). В таблице 5 приведены сведения о специфицированных командах.

Таблица 5. Статистика по специфицированным командам MIPS64

Класс команд	**Число специфицированных команд**
Целочисленная арифметика	48
Битовый сдвиг и поворот	22
Логические операции	11
Проверка условий	6
Ветвления	38
Перемещения данных	10
Работа с памятью	11 (из 26)
Операции с плавающей точкой	51 (из 65)
Исключения	14
Системные	24 (из 30)
Всего	**235 (из 270)**

4.1.3 Генерация тестовых программ для архитектуры MIPS64

Для генератора тестовых программ для архитектуры MIPS64 были разработаны шаблоны тестовых программ, демонстрирующие его возможности.

Всего было разработано около 30 шаблонов. Тестовые сценарии, описанные при помощи этих шаблонов, включают в себя:

- детерминированные тесты для команд целочисленной арифметики;
- случайные тесты для команд целочисленной арифметики;
- комбинаторные тесты на различные ситуации в работе команд целочисленной арифметики, описанные при помощи ограничений;
- тесты со встроенными проверками для команд целочисленной арифметики;
- тесты для операций с плавающей точкой (комбинаторные и на основе ограничений);
- комбинаторные тесты на команды ветвления (перебор трасс исполнения);
- комбинаторные тесты для команд доступа к памяти (комбинации различных сценариев обработки запросов к памяти, построенных путем разрешения ограничений);
- комбинированные тесты на команды целочисленной арифметики и ветвления, основанные на разрешении ограничений.

Таким образом, можно сделать заключение, что построенный генератор позволяет комбинировать различные техники генерации и пригоден для использования в промышленных проектах.

4.2 Генератор тестовых программ для архитектуры ARMv8

При помощи инструмента, реализующего предложенный метод, был сконструирован генератор тестовых программ для архитектуры ARMv8. В настоящее время он успешно применяется в известной международной компании, разрабатывающей собственные микропроцессоры архитектуры ARMv8. Особенности данной архитектуры и опыт разработки ее формальных спецификаций подробно рассмотрены ниже.

4.2.1 Архитектура ARMv8

ARM (изначально Acorn RISC Machine, сейчас Advanced RISC Machine) – это семейство микропроцессорных архитектур, разработанных в одноименной

компании [105]. Микропроцессоры этого семейства чрезвычайно популярны: с 1990 года было выпущено более 86 миллиардов чипов [106]; 95% смартфонов базируются на ARM (данные за 2010 год) [107]; микросхемы ARM находят применение во встроенных системах и центрах обработки данных. Разработкой ARM-совместимых ядер занимается не только ARM, но и другие компании, включая Qualcomm, Apple, Samsung, NVIDIA и Huawei [108]; очевидно, что эффективная реализация микропроцессора есть важнейший инструмент конкурентной борьбы. Новая архитектура ARMv8(-A) отличается большим числом команд (около 1000) и сложной организацией виртуальной памяти (VMSA, Virtual Memory System Architecture) [109].

4.2.2 Спецификации архитектуры ARMv8

Архитектура ARMv8 описывается в руководстве «*ARM Architecture Reference Manual*» [109], объем которого составляет примерно 4000 страниц. Описание включает архитектурные особенности (уровни исключений и т.п.), систему команд (около 1000 команд разных типов) и организацию подсистемы памяти. Всю совокупность инструкций можно разбить на следующие группы:

- *команды ветвления* (условные и безусловные переходы, исключения, возврат из обработчика исключений);
- *команды загрузки и сохранения данных* (обычные загрузки/сохранения для различных типов данных, загрузки/сохранения с блокировками);
- *вычислительные команды* (инициализация регистров, целочисленная арифметика и арифметика с плавающей точкой);
- *векторные команды* (SIMD — Single Instruction, Multiple Data),
- *системные команды* (работа с системными регистрами).

Состояние микропроцессора (точнее, одного его ядра) описывается набором значений регистров, соответствующих текущему уровню исполнения команд (EL — Exception Level). Всего поддерживается четыре уровня, переключение между которыми осуществляется специальными системными командами:

- EL0 — уровень пользовательских приложений (непривилегированное исполнение);
- EL1 — уровень ядра операционной системы (привилегированное исполнение);
- EL2 — уровень гипервизора;
- EL3 — уровень защищенного монитора.

Архитектурой определены следующие регистры:

- R0-R30 — 64-битные регистры общего назначения (с доступом ко всем 64 битам или только к младшим 32);
- SP — указатель стека текущего EL;
- PC — счетчик команд (недоступный программно);
- V0-V31 — 128-битные регистры для хранения чисел с плавающей точкой (с поддержкой разных форматов и возможностью векторной интерпретации одного регистра);
- FPCR и FPSR — контрольный и статусный регистры для операций над числами с плавающей точкой;
- несколько сотен системных регистров.

Ниже в качестве иллюстрации приведено описание команды MOVZ (Move Wide with Zero), перемещающей 16-битные данные в регистр и обнуляющей 48 оставшихся битов, на языке nML.

Пример 39. Спецификация команды MOVZ архитектуры ARMv8

```
01:   // Константы, соответствующие разным режимам перемещения
02:   let MoveWideOp_N = 0b00 // Перемещение с инверсией всех битов
03:   let MoveWideOp_Z = 0b10 // Перемещение с обнулением остальных битов
04:   let MoveWideOp_K = 0b11 // Перемещение без изменения остальных битов
05:   // Общая операция (функция) для команд перемещения 16-битных данных
06:   op MovWideImmGeneral (rd: REG, imm: HWORD, shift: card(2), opcode: card(2))
07:     action = {
08:       // Учет режима перемещения (обнуление битов)
09:       if opcode != MoveWideOp_K then rd = 0; endif;
10:       // Выбор поля регистра для записи
11:       if    shift == 0 then  rd<15..0>   = imm;
12:       elif shift == 1 then  rd<31..16> = imm;
13:       elif shift == 2 then  rd<47..32> = imm;
14:       elif shift == 3 then  rd<63..48> = imm;
15:       endif;
16:       // Учет режима перемещения (инверсия битов)
17:       if opcode == MoveWideOp_N then rd = ~rd; endif;
18:     }
19:   // Команда перемещения 16-битных данных с обнулением остальных битов
20:   op movz (rd: REG, imm: HWORD, shift: card(2))
21:     // Ассемблерный синтаксис
22:     syntax = format("movz %s, #0x%x, LSL #%d", rd.syntax, imm, coerce(BYTE, shift) * 16)
23:     // Двоичный формат
24:     image = format("%s10100101%2s%16s%5s", coerce(BIT, 1), shift, imm, rd.image)
25:     // Совершаемые командой действия
26:     action = {
27:       MovWideImmGeneral(rd, imm, shift, MoveWideOp_Z).action;
28:     }
```

Описание виртуальной памяти ARMv8 (VMSAv8-64) занимает около 600 страниц. Значительная его часть посвящена механизму трансляции адресов. В VMSAv8-64 определены четыре режима трансляции: защищенный EL3, незащищенный EL2, EL1&0 в защищенном и незащищенном вариантах. Оба варианта режимов ELl&0 состоят в традиционном преобразовании виртуального адреса в физический. Режимы EL3 и EL2 включают два этапа: сначала виртуальный адрес преобразуется в промежуточный физический, а затем промежуточный физический адрес — в физический. Каждое преобразование вызывает обход таблиц трансляции и может привести к четырем дополнительным обращениям к памяти. Для ускорения этого процесса таблицы трансляции кэшируются в ассоциативном буфере трансляции (TLB, Translation Lookaside Buffer).

Формальные спецификации подсистемы памяти были разработаны на языке MMUSL**.** Были специфицированы TLB, таблицы трансляции, двухуровневая кэш-памяти и логика обработки запросов к памяти во всех режимах. В приведенном ниже примере описывается тип виртуального адреса.

Описание представляет собой структуру, содержащую адрес и дополнительные данные о запросе к памяти.

Пример 40. Описание типа адреса для архитектуры VMSAv8-64

```
01:    address VA (
02:      // Виртуальный адрес
03:      addr : 64,
04:      // Дополнительные данные
05:      acctype: 4, iswrite: 1, wasaligned: 1, size: 6
06:    )
```

Сегменты памяти задают отображение диапазона адресов некоторого типа на множество адресов другого типа. В следующем примере описан сегмент VA_LO_UNMAPPED, преобразующий виртуальные адреса в физические напрямую (без использования TLB и таблиц трансляции).

Пример 41. Описание типа адреса для архитектуры VMSAv8-64

```
01:    segment VA_LO_UNMAPPED (va: VA) = (pa: PA)
02:      // Диапазон отображаемых виртуальных адресов
03:      range = (0x0000000000000000, 0x0000ffffFFFFffff)
04:      // Способ преобразования виртуальных адресов в физические
05:      read = { pa.addrdesc.paddress.physicaladdress = va.vaddress<47..0>; }
```

Ниже приведен фрагмент спецификации буфера TLB. Данный буфер отображается на регистры и доступен из спецификаций системы команд.

Пример 42. Описание буфера TLB архитектуры VMSAv8-64

```
01:    register buffer TLB (va: VA)
02:      // Ассоциативность
03:      ways = 64
04:      // Число множеств
05:      sets = 1
06:      // Формат записи
07:      entry = (addr: 36, nG: 1, contiguous: 1, level: 2, blocksize: 64,
08:                perms: Permissions, addrdesc: AddressDescriptor)
09:      // Предикат, проверяющий попадание в буфер
10:      match = (addr == va.addr<47..12> && addrdesc.fault.type == Fault_None)
11:      // Политика вытеснения данных
12:      policy = LRU
```

Таблица трансляции описывается как буфер, отображаемый на память. Таким образом, обращения к ней вызывают «рекурсивный» доступ в память. Описание таблицы трансляции показано в приведенном ниже примере.

Пример 43. Описание таблицы трансляции VMSAv8-64

```
01:    memory buffer TranslationTable(va: VA)
02:      entry = (IGNORED: 9, XN: 1, PXN: 1, Contiguous: 1, RES0: 4, pa: 36, nG: 1,
03:                AF: 1, SH: 2, AP: 2, NS: 1, AttrIndx: 3, page: 1, valid: 1)
```

Логика загрузки и сохранения данных описывается запросами к сегментам и буферам. Ниже приведен фрагмент спецификации подсистемы памяти VMSAv8-64.

Пример 44. Фрагмент описания спецификации подсистемы памяти VMSAv8-64

```
01:   // Управляющая логика подсистемы памяти
02:   mmu pmem (va: VA) = (data: 64)
03:     var pa: PA;
04:     var l1Entry: L1.entry;
05:     var line: 256;
06:     // Обработка запросов загрузки данных
07:     read = {
08:       if va.acctype != AccType_PTW then
09:         pa.addrdesc = AArch64TranslateAddress(va.vaddress, va.acctype, 0, 1, 8);
10:       else
11:         pa.addrdesc.paddress.physicaladdress = va.vaddress<47..0>;
12:         pa.size = va.size;
13:         pa.acctype = va.acctype;
14:       endif;
15:       if (va.acctype == AccType_AT) then update_PAR_EL1(pa); endif;
16:       if L1(pa).hit then
17:         l1Entry = L1(pa);
18:         line = l1Entry.DATA;
19:       else
20:         if va.acctype != AccType_PTW && va.acctype != AccType_AT then
21:           line = AArch64MemSingleRead(pa.addrdesc, 8, va.acctype, 1);
22:         else
23:           line = M(pa);
24:         endif;
25:         l1Entry.TAG = pa.addrdesc.paddress.physicaladdress<47..12>;
26:         l1Entry.DATA = line;
27:         L1(pa) = l1Entry;
28:       endif;
29:       data = get_word_from_line(va.vaddress<4..3>, line);
30:       if (va.acctype == AccType_IFETCH) then
31:         data<47..0> = pa.addrdesc.paddress.physicaladdress;
32:         data<63..48> = 0;
33:       endif;
34:     }
35:     // Обработка запросов сохранения данных
36:     write = {...}
```

Трудоемкость разработки формальных спецификаций микропроцессора ARMv8 составила 30 человеко-месяцев. Всего были описаны 340 основных команд, 217 команд арифметики с плавающей точкой, 249 команд расширения ARMv8.1 и ARMv8.2, 209 команд векторной арифметики SIMD, а также подсистема памяти. В целом были специфицированы все уровни исключений EL0-EL3. Объем кода составил около 18178 строк для системы команд (nML) и около 2643 строк для подсистемы памяти (MMUSL). В таблице 6 приведены сведения о специфицированных командах.

Таблица 6. Статистика по специфицированным командам ARMv8

Класс команд	**Число специфицированных команд**
(1) Передача управления	12
(2) Сохранение в память и загрузка из памяти	85
(3) Арифметика с непосредственными операндами	68
(4) Арифметика с регистровыми операндами	121
(5) Расширенная арифметика	22
(6) Арифметики с плавающей точкой	217
(7) Команды SIMD	209 (из 477)
(8) Системные команды	32
(a) Расширение v8.1, v8.2	249
Всего	**1015 (из 1283)**

4.2.3 Генерация тестовых программ для архитектуры ARMv8

Для генератора тестовых программ для архитектуры ARMv8 были разработаны шаблоны тестовых программ, демонстрирующие его возможности. Для них был разработан базовый Ruby-класс, описывающий все их общие свойства шаблонов для архитектуры ARMv8. В частности, в этот класс были добавлены дополнительные оберточные функции для доступа к регистрам и командам. Это позволило использовать в шаблонах нотацию, максимально приближенную к используемой языком ассемблера для ARMv8. Всего было разработано около 60 шаблонов. Разработанные шаблоны позволяют создавать тесты следующих типов:

- проверка базовой функциональности всех поддерживаемых команд (самые очевидные варианты использования);
- детерминированные тесты для команд целочисленной арифметики (реализации различных алгоритмов);
- случайные тесты для команд целочисленной арифметики (случайные регистры и случайные значения);
- комбинаторные тесты для команд целочисленной арифметики, покрывающие различные комбинации ситуаций в их работе, описанных при помощи ограничений;
- тесты со строенными проверками для команд целочисленной арифметики;

- тесты для операций с плавающей точкой;
- комбинаторные тесты на команды ветвления (перебор трасс исполнения);
- тесты для подсистемы памяти на основе ограничений;
- комбинаторные тесты для команд доступа к памяти (комбинации различных сценариев обработки запросов к памяти, построенных путем разрешения ограничений);
- тесты для команд целочисленной арифметики и доступа к памяти, исполняющие тестовые воздействия на нескольких ядрах микропроцессора;
- комбинированные тесты на команды целочисленной арифметики и ветвления, основанные на разрешении ограничений различных типов.

Как видим, данный генератор позволяет генерировать тестовые программы, предназначенные для тестирования разнообразной функциональности с применением различных техник генерации. Таким образом, можно сделать заключение, что сконструированный генератор удовлетворяет необходимым требованиям для использования его в промышленных проектах.

4.2.4 Проверка корректности генерируемых тестовых программ

При разработке формальных спецификаций архитектуры микропроцессора возможны ошибки. Подобные ошибки могут выражаться в некорректном ассемблерном формате команд или некорректных результатах их исполнения на эмуляторе, встроенном в генератор тестовых программ. Результатом будут тестовые программы, которые не компилируются, приводят к некорректным результатам и не достигают заданных в шаблонах свойств. Для того чтобы избежать подобных проблем, были приняты следующие меры.

В процессе генерации поострenные последовательности команд исполняются на встроенном в генератор эмуляторе, который создает трассы исполнения. Эти трассы фиксируют такие события как исполнение команды, запись в регистр и обращение к памяти. Построенные программы

компилируются и исполняются на программном эмуляторе QEMU [110, 111], который создает аналогичные трассы. После этого трассы, созданные обоими эмуляторами, сравниваются. Необходимыми условиями корректности генерируемых программ является их компилируемость, исполнение на эмуляторе QEMU без ошибок и совпадение трасс (имеются в виду основные события).

4.3 Генератор тестовых программ для архитектуры PowerPC

Инструмент, реализующий предложенный метод, был применен для создания генератора тестовых программ для архитектуры PowerPC [112]. Разработанный генератор используется в исследованиях по верификации систем реального времени, использующих микропроцессоры этой архитектуры. Особенности архитектуры PowerPC и опыт разработки ее формальных спецификаций подробно рассмотрены ниже.

4.3.1 Архитектура PowerPC

PowerPC (Performance Optimization With Enhanced RISC – Performance Computing, иногда сокращается как PPC) – это микропроцессорная архитектура, основанная на концепции проектирования RISC, которая была разработана в 1991-м году альянсом компаний Apple, IBM и Motorola (известен как AIM). Микропроцессоры PowerPC использовались в персональных компьютерах компании Apple до конца 2000-х годов. В настоящее время одной из важнейших областей применения микропроцессоров PowerPC является авионика.

4.3.2 Спецификации архитектуры PowerPC

Архитектура PowerPC описывается в руководстве «PowerPC Architecture Book», включающем три тома [113, 114, 115], общий объем которых составляет 433 страницы. Микропроцессоры PowerPC могут работать в двух режимах: 32-битном и 64-битном. При этом поддерживается возможность динамического

переключения между режимами. В 32-битном режиме 64-битный микропроцессор может исполнять 32-битный код.

Состояние микропроцессора PowerPC описывается при помощи 32-х регистров общего назначения (GPR0 – GPR31) и такого же количества регистров с плавающей точкой (FPR0 – FPR31). Оба типа регистров имеют размер 64 бита. Кроме этого поддерживаются специальные регистры CR (Condition Register), LR (Link Register), CTR (CounT Register), FPSCR (Floating-point Status and Control Register) и XER (fiXed-point Exception Register).

Команды, поддерживаемые микропроцессорами PowerPC, можно разделить на следующие категории:

- команды перемещения данных (в память, из памяти и между регистрами);
- вычислительные команды целочисленной арифметики (сложение, вычитание, умножение, деление, логические операции, битовые сдвигов и повороты);
- команды сравнения;
- команды ветвления;
- команды операций с плавающей точкой;
- системные команды.

На языке nML было описано небольшое подмножество системы команд PowerPC. Всего было описано 34 команды. Объем кода спецификаций составил 935 строк. Трудоемкость разработки спецификаций составила 1 человеко-месяц. Специфицированные команды включают в себя основные операции целочисленной арифметики, сравнения, ветвления и доступа к памяти.

4.3.3 Генерация тестовых программ для архитектуры PowerPC

Для архитектуры PowerPC были разработаны шаблоны тестовых программ, позволяющие продемонстрировать базовые возможности генератора. На основе созданных шаблонов строятся тесты для проверки базовой функциональности всех описанных команд и строятся случайные тесты со встроенными проверками на операции целочисленной арифметики.

Таким образом, проведенные эксперименты показали, что предложенный метод позволяет сконструировать работоспособный генератор тестовых программ для архитектуры PowerPC.

4.4 Генератор тестовых программ для архитектуры RISC-V

Инструмент, реализующий предложенный метод, был применен для создания генератора тестовых программ для архитектуры RISC-V [116]. Разработанный генератор используется в исследованиях по верификации микропроцессоров с этой архитектурой. Особенности архитектуры RISC-V и опыт разработки ее формальных спецификаций подробно рассмотрены ниже.

4.4.1 Архитектура RISC-V

RISC-V – это открытая микропроцессорная архитектура, основанная на концепции проектирования RISC. Она была создана Университетом Калифорнии в Беркли в 2010-м году, где она применялась в исследовательских и образовательных проектах. В настоящее время развитием данной архитектуры занимается консорциум код названием RISC-V Foundation [117]. Ключевые особенности этой архитектуры – открытость и возможность создания расширений. Создатели RISC-V позиционируют ее как архитектуру широкого спектра применения. Над созданием реализации архитектуры RISC-V работают такие компании как NVIDIA [118] и Samsung [119], а также совместные исследовании в этой области ведутся в Швейцарской высшей технической школе Цюриха и в Болонском университете [120].

4.4.2 Спецификации архитектуры RISC-V

Архитектура RISC-V описывается в руководстве «The RISC-V Instruction Set Manual», состоящем из двух томов [121, 122], общий объем которых составляет 236 страниц. Существует 32-битная и 64-битная версии архитектуры RISC-V, использующие идентичный набор команд. Разница между этими двумя версиями заключается в размере регистров общего назначения и в том, что в 64-битную версию добавляются дополнительные команды для поддержки 32-битных операций. Система команд состоит из базового набора команд и

расширений, которые могут быть стандартными и пользовательскими. Различные реализации архитектуры могут поддерживать разные подмножества системы команд. Подмножества системы команд, описанные в руководстве, включают в себя:

- RV32I, базовый набор, который включает в себя общий для 32- и 64-битной версий архитектуры набор команд целочисленной арифметики, ветвления, доступа к памяти и системных вызовов;
- RV64I (дополнение к RV32I), базовый набор, который включает в себя дополнительные 32-битные команды целочисленной арифметики и доступа к памяти для 64-битной версии архитектуры;
- RV32M, стандартное расширение, которое включает в себя целочисленные операции умножения и деления общие для 32- и 64-битной версий архитектуры;
- RV64M (дополнение к RV32M), стандартное расширение, которое включает в себя дополнительные 32-битные команды умножения и деления для 64-битной версии архитектуры;
- RV32A, стандартное расширение, которое включает в себя команды атомарных операций целочисленной арифметики над данными из памяти для 32- и 64-битной версий архитектуры;
- RV64A (дополнение RV32A), стандартное расширение, которое включает в себя дополнительные команды атомарных операций целочисленной арифметики над 32-битными данными из памяти для 64-битной версии архитектуры;
- RV32F, стандартное расширение, которое включает в себя команды арифметики с плавающей точкой одинарной точности и команды конвертации в целочисленный формат общие для 32- и 64-битной версий архитектуры;
- RV64F (дополнение к RV32F), стандартное расширение, которое включает в себя дополнительные команды конвертации в 32-битный

целочисленный формат для 64-битной версии архитектуры;

- RV32D, стандартное расширение, которое включает в себя команды арифметики с плавающей точкой двойной точности и команды конвертации в целочисленный формат общие для 32- и 64-битной версий;
- RV64D (дополнение к RV32D), стандартное расширение, которое включает в себя дополнительные команды конвертации в 32-битный целочисленный формат для 64-битной версии архитектуры;
- привилегированные системные команды.

Состояние микропроцессоров с архитектурой RISC-V описывается следующими регистрами: счетчик команд PC размером 32 или 64 бита; 32 регистра общего назначения x0 – x31 размером 32 или 64 бита (RV32I и RV64I); 32 регистра с плавающей точкой одинарной точности f0-f31 размером 32 бита (RV32F и RV64F); 32 регистра с плавающей точкой двойной точности f0-f31 размером 64 бита (RV32D и RV64D); управляющий регистр для операций с плавающей точкой FCSR (Floating-Point Control and Status Register) 32 или 64 бита (RV32F, RV64F, RV32D и RV64D); 12-битные управляющие системные регистры CSR (Control and Status Registers), отображаемые на память, в количестве до 4096.

На языке nML были описаны следующие подмножества системы команд RISC-V: RV32I (около 80%), RV64I, RV32M и RV64M. Тип архитектуры (32 или 65 бита) и задаются при помощи директив препроцессора в коде спецификаций. В приведенном ниже примере демонстрируется описание регистров общего назначения, режима адресации для доступа к ним и команды целочисленного сложения.

Пример 45. Спецификация регистров общего назначения и команды ADDI архитектуры RISC-V

```
01:    // Тип данных для слова(зависит от конфигурации)
02:    #ifdef RV64I
03:      let XLEN = 64
04:    #else
05:      let XLEN = 32
06:    #endif
07:    type XWORD = card(XLEN)
08:    // Регистры общего назначения
09:    reg XREG [32, XWORD]
10:    // Режим адресации для доступа к регистрам общего назначения
11:    mode X (i: card(5)) = XREG [i]
12:      syntax = format("x%d", i)
13:      image  = format("%5s", i)
14:    // Команда сложения с регистра и 12-битной константы
15:    op addi(rd: X, rs1: X, imm: card(12))
16:      syntax = format("addi %s, %s, 0x%x", rd.syntax, rs1.syntax, imm)
17:      image  = format("%12s%s000%s0010011", imm, rs1.image, rd.image)
18:      action = {
19:        rd = rs1 + sign_extend(XWORD, imm);
20:       }
```

Трудоемкость разработки спецификаций составила 0.75 человеко-месяца. Объем кода спецификаций составил 816 строк. Всего было описано 62 команды. В таблице 7 приведены сведения о специфицированных командах.

Таблица 7. Статистика по специфицированным командам RISC-V

Класс команд	**Число специфицированных команд**
RV32I (целочисленная арифметика, доступ к памяти, ветвления)	37
RV32I (доступ к управляющим регистрам)	0 (из 6)
RV32I (системные команды)	0 (из 9)
RV64I (целочисленная арифметика, доступ к памяти)	12
RV32M (целочисленное умножение и деление)	8
RV64M (целочисленное умножение и деление)	5
RV32A (атомарные операции)	0 (из 11)
RV64A (атомарные операции)	0 (из 11)
RV32F (плавающая арифметика)	0 (из 26)
RV64F (плавающая арифметика)	0 (из 4)
RV32D (плавающая арифметика)	0 (из 26)
RV64D (плавающая арифметика)	0 (из 6)
Всего	**62 (из 161)**

4.4.3 Генерация тестовых программ для архитектуры RISC-V

Для генератора тестовых программ для архитектуры RISC-V были разработаны шаблоны тестовых программ, демонстрирующие его возможности. Все общие свойства шаблонов для данной архитектуры были описаны в базовом классе, в котором были добавлены дополнительные оберточные

функции для доступа к регистрам и командам. Это позволило использовать в шаблонах нотацию, максимально приближенную к используемой языком ассемблера для RISC-V. Всего было разработано около 10 шаблонов, позволяющих создавать тесты следующих типов:

- детерминированные тесты для команд целочисленной арифметики;
- случайные тесты для команд целочисленной арифметики;
- комбинаторные тесты для команд целочисленной арифметики;
- тесты со строенными проверками для команд целочисленной арифметики;
- комбинаторные тесты на команды ветвления (перебор трасс исполнения);

Таким образом, проведенные эксперименты показали, что предложенный метод позволяет сконструировать работоспособный генератор тестовых программ для архитектуры RISC-V. Важно отметить, что данный генератор, поддерживающий 40% от общего числа команд, был разработан всего за 3 недели.

4.5 Выводы

Предложенный метод автоматизации конструирования генераторов тестовых программ был применен для микропроцессоров MIPS64, ARMv8, PowerPC и RISC-V. Для данных архитектур были разработаны формальные спецификации системы команд (на языке nML) и подсистемы управления памятью (на языке MMUSL), на основе которых были сконструированы генераторы тестовых программ. Общая статистика по разработанным формальным спецификациям приводится в таблице 8.

Таблица 8. Статистика по разработанным формальным спецификациям

Архитектура	**MIPS64**	**ARMv8**	**PowerPC**	**RISC-V**
Число заспецифицированных команд	235	1015	34	62
Размер nML спецификаций в строках кода	3999	18178	935	816
Размер MMUSL спецификаций в строках кода	267	2643	0	0
Общие трудозаты в человеко-месяцах	4	30	1	0.75
Трудоемкость описания одной команды в человеко-днях	0.37	0.65	0.64	0.27

Практический опыт показал, что трудоемкость разработки формальных спецификаций находится в линейной зависимости от количества команд, поддерживаемых микропроцессором. Трудоемкость описания одной команды зависит от сложности системы команд и в среднем составляет 0.48 человеко-дня. Для сравнения: при использовании инструмента MicroTESK 1.0, который предполагает разработку спецификаций на языке Java, трудоемкость описания одной команды архитектуры MIPS64 составляет около 0.8 человеко-дня [11, 99]. Таким образом, предложенный метод позволяет сократить трудозатраты на создание генератора тестовых программ примерно в 2 раза. Сравнение с другими инструментами генерации не приводится, так как данные о трудоемкости их конфигурирования отсутствуют в открытых источниках.

Сконструированные генераторы тестовых программ отвечают основным требованиям, предъявляемым к промышленным инструментам такого рода:

- поддержка различных техник построения тестовых программ (случайные, комбинаторные, на основе ограничений);
- возможность построения тестов со встроенными проверками;
- учет архитектурных и микроархитектурных особенностей (условий возникновения исключений, числа вычислительных ядер и потоков, параметров кэш-памяти и т.п.).

Генераторы тестовых программ для MIPS64 и ARMv8 применяются в отечественных и зарубежных компаниях.

Заключение

Основные научные и практические результаты, полученные в диссертационной работе и выносимые на защиту, состоят в следующем:

1. Разработан метод автоматизации конструирования генераторов тестовых программ для микропроцессоров на основе формальных спецификаций.
2. Разработан язык описания шаблонов тестовых программ, позволяющий описывать их структурные и поведенческие свойства.
3. Разработана архитектура генераторов тестовых программ для микропроцессоров, позволяющая интегрировать разные техники генерации и допускающая расширение множества поддерживаемых техник.
4. Разработан программный инструмент, использующий предложенный метод для конструирования генераторов тестовых программ с предложенной архитектурой, осуществляющих генерацию на основе шаблонов на предложенном языке.

Разработанный программный инструмент применяется в промышленных проектах по верификации микропроцессоров.

Благодарности. Автор выражает благодарность А.С. Камкину за неоценимый вклад в проведенную работу, без энтузиазма и оптимизма которого она бы не состоялась. Отдельную благодарность автор выражает коллегам по Институту, которые участвовали в апробации разработанного инструмента.

Список литературы

[1] Татарников А.Д. *Обзор методов и средств генерации тестовых программ для микропроцессоров.* Труды ИСП РАН, т. 29, в. 1, 2017, С. 167-194.

[2] Камкин А.С., Коцыняк А.М., Проценко А.С., Татарников А.Д., Чупилко М.М.. *Генератор тестовых программ для архитектуры ARMv8 на основе инструмента MicroTESK.* Труды ИСП РАН, т. 28, в. 6, 2016, С. 87-102.

[3] Татарников А.Д. *Построение поведенческих моделей микропроцессоров для генерации тестовых программ.* Известия высших учебных заведений. Физика. Том 59. No 8/2. 2016. С. 97-100.

[4] Tatarnikov A. *An Approach to Instruction Stream Generation for Functional Verification of Microprocessor Designs.* Proceedings of 14th IEEE East-West Design & Test Symposium (EWDTS'2016). 2016. P. 270-273.

[5] Татарников А.Д. *Комбинаторная генерация тестовых программ для микропроцессоров на основе формальных спецификаций системы команд.* Сборник трудов конференция «Проблемы разработки перспективных микро- и наноэлектронных систем». 2016. Часть II. С. 38-45.

[6] Tatarnikov A. *Language for Describing Templates for Test Program Generation for Microprocessors.* Proceedings of the Institute for System Programming Volume 28 (Issue 4). 2016. P. 77-98.

[7] Chupilko M., Kamkin A., Kotsynyak A., Protsenko A., Smolov S., Tatarnikov A. *Specification-Based Test Program Generation for ARM VMSAv8-64 Memory Management Units.* Proceedings of 16th International Workshop on Microprocessor and SOC Test and Verification (MTV 2015), 2015. P. 1-7.

[8] Камкин А.С., Проценко А.С., Татарников А.Д.. *Генерация тестовых программ для микропроцессоров на основе спецификаций подсистем памяти*. Известия высших учебных заведений. Физика. Том 58. No 11/2. 2015. С. 70-74.

[9] Kamkin A., Protsenko A., Tatarnikov A. *An Approach to Test Program Generation Based on Formal Specifications of Caching and Address Translation*

Mechanisms. Proceedings of the Institute for System Programming. Volume 27 (Issue 3) 2015. P. 125-138.

[10] Татарников А.Д. *Инструмент автоматизации разработки генераторов тестовых программ для микропроцессоров на основе формальных спецификаций.* Материалы научно-технической конференции студентов, аспирантов и молодых специалистов НИУ ВШЭ им. Е.В. Арменского. 2015, С. 53-53.

[11] Камкин А.С., Коцыняк А.М., Смолов С.А., Сортов А.А., Татарников А.Д., Чупилко М.М. *Средства функциональной верификации микропроцессоров.* Труды ИСП РАН, т. 26, в. 1, 2014, С. 149-200.

[12] Камкин А.С., Сергеева Т.И., Смолов С.А., Татарников А.Д., Чупилко М.М. *Расширяемая среда генерации тестовых программ для микропроцессоров.* Программирование, №1, 2014, С. 3-14.

[13] Kotsynyak A., Tatarnikov A. *Generic Knowledgebase for Test Generation.* Proceedings of SYRCoSE 2014: 8th Spring/Summer Young Researchers' Colloquium on Software Engineering, 2014. P. 114-117

[14] Kamkin A., Sergeeva T., Tatarnikov A., Utekhin A. MicroTESK: *An Extensible Framework for Test Program Generation.* Proceedings of SYRCoSE 2013: 7th Spring/Summer Young Researchers' Colloquium on Software Engineering, 2013. P. 51-57.

[15] Kamkin A., Tatarnikov A. *MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors.* Proceedings of SYRCoSE 2012: 6th Spring/Summer Young Researchers' Colloquium on Software Engineering, 2012. P. 64-69.

[16] Grant McFarland. *Microprocessor Design: A Practical Guide from Design Planning to Manufacturing Professional Engineering.* McGraw Hill Professional, 2006, 408 p.

[17] Moore G.E. *Cramming More Components onto Integrated Circuits.* Electronics Magazine, vol. 86, no. 1, 1965.

[18] Статистика числа транзисторов в микропроцессорах — http://en.wikipedia.org/wiki/Transistor_count

[19] Intel® Pentium® 4 Processor. Specification Update, August 2008 (http://download.intel.com/design/intarch/specupdt/24919969.pdf)

[20] Intel® Core™ i7-900 Desktop Processor Extreme Edition Series and Intel® Core™ i7-900 Desktop Processor Series. Specification Update, February 2015. (http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/core-i7-900-ee-and-desktop-processor-series-spec-update.pdf)

[21] Beizer B. The Pentium Bug – An Industry Watershed // Testing Techniques Newsletter, TTN Online Edition, September 1995.

[22] Bergeron J. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2000. 354 p. doi:10.1007/978-1-4615-0302-6

[23] Bentley B. *Validating the Intel® Pentium 4® Microprocessor*. Proc. Design Automation Conference (DAC), 2001. pp. 244-248. doi:10.1145/378239.378473

[24] Bentley B. *Validating a Modern Microprocessor.* Proc. International Conference on Computer Aided Verification (CAV), 2005. pp. 2-4. (http://www.cav2005.inf.ed.ac.uk/bentley_CAV_07_08_2005.ppt) (DOI 10.1007/11513988_2)

[25] Matteo Monchiero, Raúl Martínez. *Design Cycle for Microprocessors.* Intel Corporation, 2011, 34 p.

[26] А.С. Камкин. *Верификация микропроцессоров: борьба с ошибками и управление качеством.* Электроника: НТБ, №3, 2010. С. 98-104.

[27] Z. Navabi. *Languages for Design and Implementation of Hardware.* W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.

[28] S. Mikhani, Z. Navabi. *System Level Design Languages.* W.-K. Chen (Ed.). The VLSI Handbook. CRC Press, 2007. 2320 p.

[29] P. Mishra, N. Dutt (Eds.). *Processor Description Languages.* Systems on Silicon. Morgan Kaufmann, 2008. 432 p.

[30] Будылин Ф.К., Полищук И.А., Слесарев М.В., Юрлин С.В. *Опыт прототипирования микропроцессоров компании ЗАО «МЦСТ»*. Журнал «Вопросы РадиоЭлектроники», 2012. с. 132.

[31] W.K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall, 2005. 624 p.

[32] 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.* IEEE Design & Test of Computers, 21(2), 2004. p. 84-93.

[33] Чибисов П.А. *Тестирование микропроцессоров и их RTL-моделей приложениями пользователя под ОС Linux*. Программные продукты и системы, №3, 2012, с. 112-116.

[34] Бобков С.Г., Чибисов П.А. *Повышение качества тестирования высокопроизводительных микропроцессоров методами встречного тестирования с анализом функционального тестового покрытия выделенных приложений*. Информационные технологии, No8, 2013, с. 26-33.

[35] Хисамбеев И.Ш., Чибисов П.А. *Об одном методе построения метрик функционального покрытия в тестировании микропроцессоров.* Проблемы разработки перспективных микро- и наноэлектронных систем - 2014. Сборник трудов под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2014. Часть II. С. 63-68.

[36] Piziali A. Functional Verification Coverage Measurement and Analysis. New York: Kluwer Academic Publishers. 2004. 216 p.

[37] P. Mishra, N. Dutt. Specification-Driven Directed Test Generation for Validation of Pipelined Processors. ACM Transactions on Design Automation of Electronic Systems, 13(3), 2008. P. 1-36.

[38] E.A. Poe. *Introduction to random test generation for processor verification.* Technical report. Obsidian Software, 2002, 7 p.

[39] *Генератор тестовых программ RISU для тестирования эмулятора QEMU* - https://git.linaro.org/people/peter.maydell/risu.git/about/.

[40] Грибков И.В., Захаров А.В., Кольцов П.П. и др. *Стохастическое тестирование в системе INTEG.* Программные продукты и системы. 2007. № 2. с. 22-26.

[41] Камкин А.С. *Генерация тестовых программ для микропроцессоров.* Труды ИСП РАН, том 14, часть 2, 2008, с. 23–63.

[42] N. Sharma, B. Dickman, *Verifying an ARM Core*, EE Times, 2001, 7 p.

[43] А.С. Камкин. *Некоторые вопросы автоматизации построения тестовых программ для модулей обработки переходов микропроцессоров*. Труды ИСП РАН, 18, 2010. С. 129-149.

[44] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, G. Shurek. *Constraint-Based Random Stimuli Generation for Hardware Verification.* AI Magazine, 28(3), 2007. P. 13-30.

[45] P. Mishra and N. Dutt. *Specification-Driven Directed Test Generation for Validation of Pipelined Processors*. ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 13, Issue 3, 2008, pp. 1–36.

[46] Грибков И.В., Захаров А.В., Кольцов П.П. и др. *Развитие системы стохастического тестирования микропроцессоров INTEG*. Программные продукты и системы. 2010. № 2. С. 14–23.

[47] *MIPS64™ Architecture For Programmers. Volume 1: Introduction to the MIPS64™ Architecture. Revision 6.01*. MIPS Technologies Inc. 2014. 148 P.

[48] *Программный эмулятор VMIPS* – http://www.dgate.org/vmips/.

[49] *Сайт компании ARM* — http://www.arm.com.

[50] N. Sharma and B. Dickman. *Verifying an ARM Core. EE Times*. 2001. P 7.

[51] Hrishikesh M.S., Rajagopalan M., Sriram S., Mantri R. *System Validation at ARM — Enabling our Partners to Build Better Systems.* White Paper. April 2016 (http://www.arm.com/files/pdf/System_Validation_at_ARM_Enabling_our_partners_to_build_better_systems.pdf).

[52] Venkatesan D., Nagarajan P. *A Case Study of Multiprocessor Bugs Found Using RIS Generators and Memory Usage Techniques.* Workshop on Microprocessor Test and Verification, 2014. pp. 4-9. DOI: 10.1109/MTV.2014.28.

[53] Hudson J., Kurucheti G. *A Configurable Random Instruction Sequence (RIS) Tool for Memory Coherence in Multi-processor Systems.* Workshop on Microprocessor Test and Verification, 2014. pp. 98-101. DOI: 10.1109/MTV.2014.26.

[54] *Генератор тестовых программ RAVEN* – http://www.slideshare.net/DVClub/introducing-obsidian-software-andravengcs-for-powerpc.

[55] Obsidian Software Inc. "*Raven: Product datasheet*". 6 P.

[56] R.N. Mahapatra, P. Bhojwani, J. Lee, and Y. Kim. *Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No.43 Phase 3 Report.* 2009. 43 P.

[57] M. Behm, J. Ludden, Y. Lichtenstein, M. Rimon, M. Vinov. *Industrial Experience with Test Generation Languages for Processor Verification.* Proceedings of the Design Automation Conference. 2004. pp. 36-40.

[58] M. Aharoni, S. Asaf, L. Fournier, A. Koifman and R. Nagel. *FPgen – A Test Generation Framework for Datapath Floating-Point Verification.* Proceedings of the Eighth IEEE International Workshop on High-Level Design Validation and Test Workshop (HLDVT'03), 2003. pp. 17–22, ISBN 0-7803-8236-6.

[59] M. Aharony, E. Gofman, E. Guralnik, A. Koyfman. *Injecting floating-point testing knowledge into test generators.* Proceedings of the 7th international Haifa Verification conference on Hardware and Software: Verification and Testing (HVC'11), 2011. pp. 234-241, ISBN 978-3-642-34187-8.

[60] *IEEE standard for binary FP arithmetic. An American National Standard, ANSI/IEEEE Std. 754-2008*. 58 P.

[61] *IBM Floating-Point Test Suite for IEEE 754R Standard* - https://www.research.ibm.com/haifa/projects/verification/fpgen/ieeets.html

[62] Adir A., Fournier L., Katz Y., Koyfman A. *DeepTrans – Extending the Model-based Approach to Functional Verification of Address Translation Mechanisms.* High-Level Design Validation and Test Workshop, 2006. pp. 102-110.

[63] Д.Н. Воробьев, А.С. Камкин. *Генерация тестовых программ для подсистемы управления памятью микропроцессоров.* Труды ИСП РАН, т. 17, 2009, С. 119-132.

[64] Seonghun Jeong, Youngchul Cho, Daeyong Shin, Changyeon Jo, Yenjo Han, Soojung Ryu, Jeongwook Kim, and Bernhard Egger. *Random Test Program Generation for Reconfigurable Architectures.* Proceedings of 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, 6 p.

[65] T.Li, D.Zhu, Y.Guo, G.Liu, S.Li. *MA2TG: A Functional Test Program Generator for Microprocessor Verification.* Euromicro Conference on Digital System Design, 2005. pp.176-183.

[66] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt and A. Nicolau. *EXPRESSION: An ADL for System Level Design Exploration.* Technical Report 1998-29, University of California, Irvine, 1998. 26 P.

[67] *Инструмент SMV* – http://www.cs.cmu.edu/~modelcheck/smv.html

[68] 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.

[69] F. Corno et al. *Fully Automatic Test Program Generation for Microprocessor Cores*. Proc. IEEE Design, Automation and Test in Europe (DATE 03), IEEE CS Press, 2003, pp. 1006-1011.

[70] 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, 2004, pp. 102-109.

[71] Рубанов В.В. *Обзор методов описания встраиваемой аппаратуры и построения инструментария кросс-разработки.* Труды ИСП РАН, т. 17, 2008, С. 7-40.

[72] M. Freericks. *The nML Machine Description Formalism*. Technical Report. TU Berlin, FB20, Bericht, 1991/15. 47 p.

[73] M. Hartoog and J. Rowson and P. Reddy and S. Desai and D. Dunlop and E. Harcourt and N. Khullar. *Generation of software tools from processor descriptions for hardware/software codesign.* In Proceedings of Design Automation Conference (DAC), pages 303–306, 1997.

[74] *Страница компании Target Compiler Technologies* — http://www.retarget.com

[75] S. Chandra, R. Moona. *Retargetable Functional Simulator using High Level Processor Models.* VLSI Design, 2000. P. 424-429.

[76] *Страница инструмента GLISS* — http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=54

[77] H. Cassé, J. Barre, R. Vaillant, P. Sainrat. *Fast Instruction-Accurate Simulation with SimNML.* Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO), 2011. P. 8-12.

[78] *Язык программирования Ruby* — http://www.ruby-lang.org

[79] *Язык программирования Python* — https://www.python.org

[80] *Язык программирования Perl* — https://www.perl.org

[81] Flanagan D., Matsumoto Y. *The Ruby Programming Language*. O'Reilly Media, Sebastopol, 2008, 446 p.

[82] *Среда генерации тестовых программ для микропроцессоров* — http://forge.ispras.ru/projects/microtesk

[83] *Язык программирования Java* — https://en.wikipedia.org/wiki/Java_(programming_language)

[84] *Библиотека Fortress* — http://forge.ispras.ru/projects/solver-api

[85] Cok D.R. *The SMT-LIBv2 Language and Tools: A Tutorial*. GrammaTech, Inc., Version 1.1, 2011.

[86] Дроздов А.Ю., Новиков С.В. *Эффективный алгоритм построения формы статического единственного присваивания.* Информационные технологии, № 3, 2005.

[87] Ахо А., Лам М., Сети Р., Ульман Д. *Компиляторы: принципы, технологии и инструментарий*. 2 изд. Вильямс, 2008.

[88] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997, 887 p.

[89] *Инструмент ANTLR* – http://www.antlr.org/

[90] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* The Pragmatic Bookshelf, 2007, 384 p.

[91] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages.* The Pragmatic Bookshelf, 2009, 350 p.

[92] *Библиотека StringTemplate* – http://www.stringtemplate.org/

[93] *Библиотека Java SoftFloat* – http://forge.ispras.ru/projects/jsoftfloat

[94] *Библиотека Berkeley SoftFloat* – http://www.jhauser.us/arithmetic/SoftFloat.html

[95] *Интерпретатор JRuby* – http://jruby.org/

[96] *База знания TestBase* – http://forge.ispras.ru/projects/testbase

[97] L. Moura and N. Bjørner. *Z3: An Efficient SMT Solver. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[98] *Решатель ограничений CVC4* – http://cvc4.cs.nyu.edu

[99] Камкин А.С. *Метод автоматизации имитационного тестирования микропроцессоров с конвейерной архитектурой на основе формальных спецификаций.* Диссертация на соискание ученой степени кандидата физико-математических наук. Москва, 2009.

[100] *Микропроцессоры НИИСИ РАН* – https://www.niisi.ru/devel.htm

[101] *Сайт компании Imagination Technologies* – https://www.imgtec.com/

[102] Дмитрий Мякин. *Армейский компьютер «Восход» на архитектуре КОМДИВ64 готов к экстремальным условиям*, 2016 (https://3dnews.ru/939502).

[103] *MIPS64™ Architecture For Programmers. Volume 2: The MIPS64™ Instruction Set Reference Manual*. Revision 6.04. MIPS Technologies Inc. 2015. 551 P.

[104] *MIPS64™ Architecture For Programmers. Volume 3: MIPS64™/microMIPS64™ Privileged Resource Architecture.* Revision 6.03. MIPS Technologies Inc. 2015. 368 P.

[105] *Сайт ARM* – http://www.arm.com

[106] Mallya H. *The Backstory of How ARM Reached a Milestone of 86 Billion Chips in 25 Years*. July 19, 2016 (https://yourstory.com/2016/07/arm-holdings-story/)

[107] Morgan T.P. *ARM Holdings Eager for PC and Server Expansion. Record 2010, Looking for Intel Killer 2020.* February 1, 2011 (http://www.theregister.co.uk/2011/02/01/arm_holdings_q4_2010_numbers/).

[108] Sims G. *Custom Cores versus ARM Cores, What Is It All About?* January 7, 2016 (http://www.androidauthority.com/arm-cortex-core-custom-core-kryo-explained-664777/).

[109] *ARM Architecture Reference Manual.* ARM DDI 0487A.f, ARM Corporation, 2015. 5886 P.

[110] *Программный эмулятор QEMU* – http://www.qemu.org/

[111] *Программный эмулятор* для ARMv8 на основе *QEMU* – https://forge.ispras.ru/projects/qemu-armv8

[112] *Архитектура PowerPC* – https://en.wikipedia.org/wiki/PowerPC

[113] Wetzel/Poughkeepsie/IBM. *"PowerPC User Instruction Set Architecture"*. Book I. Version 2.02, 28 January 2005. 230 P.

[114] Wetzel/Poughkeepsie/IBM. *"PowerPC Virtual Environment Architecture".* Book II. Version 2.02, 28 January 2005. 68 P.

[115] Wetzel/Poughkeepsie/IBM. *"PowerPC Operating Environment Architecture".* Book III. Version 2.02, 28 January 2005. 135 P.

[116] *Архитектура RISC-V* – https://en.wikipedia.org/wiki/RISC-V

[117] *Сайт RISC-V Foundation* – https://riscv.org

[118] Michael Larabel. *NVIDIA Is Building Its Next-Gen Falcon Controller Using RISC-V*. 26 July 2016. (https://www.phoronix.com/scan.php?page=news_item&px=NVIDIA-RISC-V-Next-Gen-Falcon)

[119] David Manners. *Samsung Defection From ARM to RISC-V*. 28 November 2016. (https://www.electronicsweekly.com/blogs/mannerisms/dilemmas/samsung-defection-arm-risc-v-2016-11)

[120] *Микропроцессор PULPino* (*основан на RISC-V*) – http://www.pulp-platform.org

[121] Andrew Waterman, Krste Asanovic. *The RISC-V Instruction Set Manual. Volume I: User-Level ISA.* Version 2.2. University of California, Berkeley. May 7, 2017. 145 P.

[122] Andrew Waterman, Krste Asanovic. *The RISC-V Instruction Set Manual. Volume II: Privileged Architecture. Version 1.10.* University of California, Berkeley. May 7, 2017. 91 P.

Приложения

«Утверждаю»
Зам. директора по наноэлектронике Федерального государственного учреждения "Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук"
_____ М.С. Горбунов
« 20 » июля 2017 года

АКТ

о внедрении результатов диссертационной работы

Татарникова Андрея Дмитриевича

Комиссия в составе:

председатель:

зав. отделением разработки вычислительных систем к.т.н. Аряшев Сергей Иванович,

члены комиссии:

зав. сектором отдела архитектур высокопроизводительных микропроцессоров к.т.н. Корниленко Александр Владимирович, зав. отделом сопровождения разработок Сидоров Сергей Александрович, зав. группой отдела тестирования микросхем Слинкин Дмитрий Игоревич, составила настоящий акт о том, что разработанный в диссертационной работе Татарникова А.Д. метод автоматизации конструирования генераторов тестовых программ для микропроцессоров на основе формальных спецификаций и реализующий его инструмент MicroTESK использованы при выполнении ОКР «Процессор-5» и «Процессор-6» по созданию микропроцессоров 1890ВМ8Я и 1890ВМ9Я в ФГУ ФНЦ НИИСИ РАН.

Использование указанных результатов диссертации позволяет повысить эффективность тестирования микропроцессоров и их моделей, сократить время, требующееся на проведение работ по функциональной верификации моделей проектируемых микропроцессоров.

Использование генератора тестовых программ, разработанного при помощи инструмента MicroTESK, позволило выявить несколько серьезных ошибок в подсистеме управления памятью и в буфере трансляции адресов в RTL-модели микропроцессора 1890ВМ8Я.

Председатель комиссии: зав. отделением к.т.н. Аряшев С. И.

Члены комиссии:

зав. сектором к.т.н. Корниленко А.В.

зав. отделом Сидоров С. А.

зав. группой Слинкин Д.И.