Генерация тестовых программ для микропроцессоров на основе шаблонов конвейерных конфликтов

Д.Н. Воробьев, А.С. Камкин
{vorobyev, kamkin}@ispras.ru

Аннотация. В работе рассматривается методика автоматизированного построения тестовых программ для верификации управляющей логики микропроцессоров. Методика основана на формальной спецификации системы команд и описании шаблонов конфликтных ситуаций возможных в работе конвейера тестируемого микропроцессора. Использование формальных спецификаций позволяет автоматизировать разработку генератора тестовых программ и систематично протестировать управляющую логику. В то же время, поскольку подход основан на высокоуровневых описаниях, не учитывающих потактовое функционирование конвейера, разработанные спецификации и шаблоны, а также сгенерированные по ним тестовые программы допускают повторное использование при изменении микроархитектуры. Это позволяет применять методику на ранних стадиях разработки микропроцессоров, когда возможна частая переработка проектных решений.

1. Введение

Работа современного микропроцессора с конвейерной архитектурой организована очень сложным образом. Одновременно он может обрабатывать несколько инструкций, которые, к тому же, могут взаимодействовать через общие ресурсы. На каждом такте микропроцессор принимает решения, связанные с разрешением конфликтов между инструкциями, изменением потока управления и обработкой исключений. Набор механизмов микропроцессора, отвечающих за управление выполнением инструкций на конвейере, называется *управляющей логикой*. Образно говоря, управляющая логика — это «нервная система» микропроцессора, без которой не возможна согласованная работа его «органов» — модулей и подсистем.

Управляющая логика является ключевым компонентом микропроцессора, и, соответственно, ее проектирование и тестирование должно проводиться с

особой тщательностью. На практике же создание тестов[1], как правило, осуществляется вручную или с использованием случайной генерации. Очевидно, что ручная разработка является непродуктивным подходом, используя который невозможно разработать качественный набор тестов за приемлемое время. Случайная генерация, хотя и позволяет быстро обнаружить многие ошибки в проекте, не дает никаких гарантий относительно полноты тестирования. В последнее время стали появляться методы автоматизации тестирования, использующие потактовые модели управляющей логики. Такие методы нацелены на детальную верификацию управляющих механизмов микропроцессора, но они не приживаются в инженерной среде из-за сложности разработки и поддержки моделей.

В настоящей работе рассматривается подход к генерации тестовых программ для проверки управляющей логики, который «расположен» между случайной генерацией и тестированием на основе точных моделей. В основе подхода лежит формальная спецификация системы команд, описывающая по отдельности инструкции микропроцессора, не вдаваясь в особенности их совместного выполнения на конвейере. Цель генерации задается с помощью тестового покрытия, определяющего шаблоны различных взаимодействий, возникающих на конвейере. Модели системы команд, безусловно, менее информативны по сравнению с точными моделями управляющей логики, однако у них есть ряд практических преимуществ. Прежде всего, такие модели существенно проще для разработки, и, кроме того, они могут быть повторно использованы даже при кардинальных изменениях микроархитектуры.

Оставшаяся часть статьи организована следующим образом. Во втором разделе говорится об управляющей логике микропроцессора, в нем также определяется понятие конвейерного конфликта. Третий раздел посвящен анализу существующих подходов к тестированию управляющей логики. В четвертом разделе рассматриваются основные концепции комбинаторной генерации тестовых программ, на которых основана предлагаемая методика, — сама методика изложена в пятом разделе. Шестой раздел описывает опыт практической апробации подхода. Наконец, в седьмом разделе делается заключение, и очерчиваются направления дальнейших работ.

2. Управляющая логика микропроцессора

Под *управляющей логикой микропроцессора* (*control logic*) понимается его внутренняя функциональность, отвечающая за управление выполнением инструкций на конвейере. Для обозначения подсистемы микропроцессора,

[1] Тесты для проверки управляющей логики микропроцессора обычно имеют форму программ в соответствующей системе команд. Программы, созданные специально для тестирования, называются *тестовыми программами*. В данной статье везде под *тестами* понимаются именно тестовые программы.

реализующей управляющую логику, обычно используют термин *устройство управления* (*control unit*). Последний термин, однако, следует трактовать достаточно широко — функции управления могут и не оформляться в отдельное устройство, а быть распределенными между разными модулями микропроцессора.

Управляющая логика имеет сложную иерархическую структуру, на разных уровнях которой решаются разные задачи управления конвейером микропроцессора. Например, на верхнем уровне может осуществляться трансляция инструкций микропроцессора во внутренние микроинструкции ядра, их распределение по каналам конвейера; на следующем уровне — декомпозиция микроинструкций на еще более мелкие операции; наконец, на уровне отдельных модулей — управление обработкой модульных операций. В рамках данной статьи нас в первую очередь интересует управляющая логика верхнего уровня.

Рассмотрим кратко устройство конвейера микропроцессора и особенности его работы. Классический конвейер состоит из пяти стадий[2]: *выборка инструкции* (*IF*, *Instruction Fetch*), *декодирование* (*ID*, *Instruction Decode*), *выполнение* (*EX*, *EXecute*), *доступ к памяти* (*MEM*, *MEMory access*) и *запись результата* (*WB*, *Write Back*) [1]. В идеальном случае выполнение инструкций на конвейере осуществляется, как показано на Рис. 1. На каждом такте осуществляется выборка очередной инструкции из памяти, при этом стадия *IF* инструкции с номером *N* выполняется параллельно со стадией *ID* инструкции *N*-1, стадией *EX* инструкции *N*-2, стадией *MEM* инструкции *N*-3 и стадией *WB* инструкции *N*-4.

№ инструкции (↓)	Стадии конвейера								
1	IF	ID	EX	MEM	WB				
2		IF	ID	EX	MEM	WB			
3			IF	ID	EX	MEM	WB		
4				IF	ID	EX	MEM	WB	
5					IF	ID	EX	MEM	WB
№ такта (→)	1	2	3	4	5	6	7	8	9

Рис. 1. Выполнение инструкций на конвейере.

[2] В современных микропроцессорах длина конвейера может превышать 20-30 стадий, например, в микропроцессоре Pentium 4 (микроархитектуры Prescott или Cedar Mill) конвейер состоит из 31 стадии.

Такая ситуация возможна, только если выполняемые инструкции являются независимыми. В случае же если, например, результаты одной инструкции подаются на вход другой (то есть имеет место *зависимость по регистрам* типа «запись-чтение»), может произойти *приостановка* (*stalling*) выполнения второй инструкции (а также инструкций, следующих за ней) до тех пор, пока не будет вычислен результат первой[3]. Другим примером нарушения нормального выполнения инструкций на конвейере является обработка *переходов* — инструкций, изменяющих *поток управления*. После выборки инструкции перехода загрузка конвейера приостанавливается до тех пор, пока не будет вычислен адрес перехода и значение истинности условия (если переход условный)[4].

Ситуации, в которых происходят приостановки или сбросы конвейера, называются *конвейерными конфликтами* (*hazards*). Конфликты, в которых две инструкции пытаются обратиться для чтения или записи к одним и тем же данным, называются *конфликтами по данным* (*data hazards*). Рассмотрим их подробнее. Пусть две инструкции *i* и *j* обращаются к общим данным, причем инструкция *i* расположена перед инструкцией *j* по потоку управления. Паттерсон и Хеннесси приводят следующую классификацию конфликтов по данным [2]:

- *чтение после записи* (*RAW*, *Read After Write*) — инструкция *j* пытается прочитать данные до того, как они записаны инструкцией *i*;
- *запись после записи* (*WAW*, *Write After Write*) — инструкция *j* пытается записать данные до того, как они записаны инструкцией *i*;
- *запись после чтения* (*WAR*, *Write After Read*) — инструкция *j* пытается записать данные до того, как они прочитаны инструкцией *i*.

Помимо конфликтов по данным возможны и другие типы конвейерных конфликтов, например, *структурные конфликты* (*structural hazards*), связанные с использованием общих модулей микропроцессора, и *конфликты по управлению* (*control hazards*), возникающие при обработке переходов.

[3] Возможны и более сложные схемы аппаратного разрешения зависимостей, когда происходит переупорядочивание инструкций с тем, чтобы минимизировать число остановок конвейера.

[4] Возможна и другая организация конвейера, когда после выборки инструкции перехода загрузка конвейера не прекращается, но все инструкции, загруженные после нее, *сбрасываются* (*flushing*), если принимается решение о выполнении перехода. В современных микропроцессорах для постоянной загрузки конвейера (минимизации числа остановок и сбросов) применяется *прогнозирование переходов* (*branch prediction*).

Разрешение разного типа конфликтов является важной функцией управляющей логики микропроцессора[5].

Еще одной задачей, за решение которой отвечает управляющая логика, является *обработка исключений*. Исключением называется событие, сигнализирующее о возникновении нештатной ситуации во время выполнения инструкции. При возникновении исключения значение счетчика адреса (адреса текущей инструкции) сохраняется в специальном регистре микропроцессора, а управление передается на программу обработки исключения, при этом все инструкции, загруженные на конвейер после инструкции, вызвавшей исключение, сбрасываются. В широком понимании, исключение является специфической разновидностью конвейерного конфликта, поскольку препятствует нормальному выполнению инструкций на конвейере.

Настоящая работа сфокусирована на тестировании механизмов управляющей логики, отвечающих за разрешение конфликтов (включая, в частности, обработку исключений). Несмотря на то, что в статье рассматривается построение тестов для разных типов конвейерных конфликтов, в том числе конфликтов по управлению, более подробно вопросы тестирования механизмов обработки переходов (по сути, разрешения конфликтов по управлению) изложены в работе [3], опубликованной в этом же сборнике.

3. Обзор существующих работ

В работе [4] описываются две взаимодополняющие техники: генерация тестов на основе *проверки моделей* (*model checking*) и генерация тестов на основе *шаблонов* (*template-based procedures*). Исходной информацией для обоих подходов является спецификация микропроцессора на языке EXPRESSION [5]. На основе спецификации автоматически строится обобщенная структурно-событийная модель на SMV [6], для которой разработчик тестов определяет *модель ошибок* (набор свойств, описывающих классы ошибок). Для каждого элемента модели ошибок строится отрицание соответствующего свойства, а для полученного отрицания с помощью SMV генерируется контрпример. Как отмечают авторы, подход на основе проверки моделей не масштабируется на сложные микропроцессоры, поэтому в качестве дополнения к нему они предлагают технику на основе шаблонов. Шаблоны разрабатываются вручную и описывают последовательности инструкцией, приводящие к возникновению определенных ситуаций в работе конвейера (прежде всего, конфликтов). На основе шаблонов конструируются более сложные тестовые программы, покрывающие модель ошибок. Генерация осуществляется на основе графовой модели, извлеченной из спецификации микропроцессора. Подход на основе шаблонов предполагает

[5] Некоторые микропроцессорные архитектуры возлагают разрешение конфликтов на программистов или компиляторы.

больший объем ручного труда, но при этом лучше масштабируется. Заметим, что обе техники используют детальные спецификации управляющей логики микропроцессора, поэтому их лучше применять на поздних стадиях проектирования. При изменении тестируемой модели микропроцессора необходимо переделывать описание на EXPRESSION, что требует значительных затрат.

В подходе, предложенном в статье [7], формально специфицируется структура конвейера в форме автоматной модели, называемой OSM (Operation State Machine). OSM моделирует управляющую логику микропроцессора на двух уровнях: *операционном* (*operational*) и *аппаратном* (*hardware*). На операционном уровне с помощью расширенных конечных автоматов описывается «движение» инструкций через стадии конвейера (каждая операция описывается отдельным автоматом). На аппаратном уровне моделируются ресурсы микропроцессора в форме так называемых *менеджеров маркеров* (*token managers*). Автомат, описывающий отдельную инструкцию, переходит из одного состояния в другое путем захвата и освобождения маркеров. Модель конвейера представляет собой композицию автоматов операций и автоматов ресурсов. Целью тестирования является проход по всем переходам совокупной автоматной модели. Данный подход требует разработки детальных OSM-спецификаций, что является трудоемкой работой, поэтому, как и предыдущие техники, его нецелесообразно применять на ранних этапах проектирования.

В работе [8] рассказывается об инструменте Genesys-Pro, предназначенном для генерации тестовых программ на основе *шаблонов* (*templates*). Генератор состоит из двух основных компонентов: независимого от целевого микропроцессора *ядра* (*engine*) и *модели* (*model*), описывающей тестируемый микропроцессор на уровне инструкций. Инженер-верификатор разрабатывает на специальном языке шаблоны, которые задают структуру тестовых программ и описывают свойства, которым она должна удовлетворять. Genesys-Pro транслирует каждый шаблон в систему ограничений и строит тестовую программу, используя техники *разрешения ограничений* (*constraint solving*). Преимуществами инструмента является его переносимость на разные микропроцессорные архитектуры и богатый язык описания шаблонов. Однако, поскольку разработка шаблонов осуществляется вручную, поддержка разработанных тестов достаточна сложна, а качество тестирования напрямую зависит от квалификации инженеров-верификаторов.

Статья [9] описывает метод генерации тестов на основе *обхода графа состояний* автоматной модели конвейера. В одном из своих этапов метод использует генератор Genesys (который впоследствии развился в Genesys-Pro). Суть метода заключается в следующем. Разработчик тестов создает модель микропроцессора на SMV. После этого с помощью инструмента CFSM строится множество путей (так называемых *абстрактных тестов*), покрывающих все дуги в графе состояний конечного автомата, извлеченного

из модели. Абстрактные тесты транслируются в шаблоны генератора Genesys, который на их основе генерирует тестовые программы. Метод позволяет достичь хорошего покрытия управляющей логики, но у него есть два основных недостатка. Во-первых, необходим опытный эксперт для разработки модели микропроцессора. Во-вторых, для возможности отображения абстрактных тестов в шаблоны нужно создать достаточно сложное описание в Genesys.

Рассмотренные выше подходы к тестированию управляющей логики микропроцессора можно разбить на два класса: *методы на основе точных моделей* [4,7,9] и *методы на основе шаблонов* [8]. Методы на основе *точных моделей*, то есть моделей, описывающих управляющую логику с потактовой или почти потактовой точностью, позволяют добиться очень высокого качества тестирования. Главным недостатком таких подходов является невозможность или нецелесообразность их использования на ранних этапах проектирования, когда управляющая логика микропроцессора не полностью определена или определена, но часто меняется. Методы генерации тестов на основе шаблонов лишены этого недостатка, но у них есть другой серьезный изъян — они лишены систематичности и не позволяют адекватно оценивать качество тестирования.

Подход, предлагаемый в статье, является компромиссом между этими двумя категориями методов. С одной стороны, для автоматизации построения тестов в нем используются модели. С другой стороны, используемые модели не фиксируют жестко управляющую логику микропроцессора — она описывается неявно в виде обобщенных шаблонов конвейерных конфликтов. За счет этого достигается масштабируемость подхода и возможность его применения в условиях частых изменений в проекте.

4. Основные понятия предлагаемого подхода

Подход основан на методе *комбинаторной генерации тестовых программ*, детальное описание которого доступно в работе [10]. В основе метода лежит *формальная спецификация* системы команд, описывающая отдельные инструкции микропроцессора безотносительно того, как они обрабатываются на конвейере. Описание каждой инструкции включает ее мнемонику, список операндов с указанием их типов, множество вызываемых исключений, предусловие, длительность обработки в тактах и семантику в императивной форме. Кроме того, формально в форме *тестовых ситуаций* и *зависимостей* описываются ситуации на обработку инструкций на конвейере. Генерация тестовых программ осуществляется автоматически путем комбинирования тестовых ситуаций и зависимостей для последовательностей инструкций ограниченной длины. Рассмотрим основные понятия метода комбинаторной генерации, которые важны для описания предлагаемого подхода.

4.1. Структура тестовой программы

Тестовая программа представляет собой последовательность *тестовых вариантов* (*test cases*). Каждый тестовый вариант содержит *тестовое воздействие* — специально подготовленную цепочку инструкций, предназначенную для создания определенной ситуации в работе конвейера микропроцессора. Перед тестовым воздействием помещаются *инициализирующие инструкции*, а после него — *тестовый оракул* — набор инструкций, проверяющих корректность состояния микропроцессора после выполнения тестового воздействия.

Таким образом, структуру тестовой программы можно описать с помощью формулы $Test = \{\langle Pre_i, Action_i, Post_i\rangle\}_{i=0,n-1}$, где Pre_i — это инициализирующие инструкции, $Action_i$ — тестовое воздействие, $Post_i$ — тестовый оракул. В простейшем случае каждая тестовая программа состоит из одного тестового варианта, то есть $Test = \langle Pre, Action, Post\rangle$, кроме того, тестовые оракулы часто опускаются (оценка правильности поведения осуществляется на основе сравнения с эталонной моделью) — $Test = \langle Pre, Action\rangle$.

Ниже в качестве примера приведен фрагмент тестовой программы в системе команд MIPS [11], содержащий один тестовый вариант.

```
// Инициализация инструкции sub[0]: IntegerOverflow=true
// s5[rs]=0xffffffffc1c998db, v0[rt]=0x7def4297
lui s5, 0xc1c9
ori s5, s5, 0x98db
lui v0, 0x7def
ori v0, v0, 0x4297

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

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

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

// Тестовое воздействие: 2010
sub a1, s5, v0 // IntegerOverflow=true
add t7, a0, s3 // Exception=false
div a2, a1     // DivisionByZero=true
```

4.2. Тестовый шаблон

Важным понятием подхода, которое следует рассмотреть подробнее, является понятие *тестового шаблона*. Тестовым шаблоном называется абстрактная форма представления тестового воздействия, в котором вместо конкретных значений операндов инструкций указываются ограничения (*тестовые ситуации* и *зависимости*), которым они должны удовлетворять. Кроме того, вместо определенных инструкций могут быть заданы ограничения на них (обычно в форме *классов эквивалентности* инструкций). По сути, каждый тестовый шаблон задает некоторую *цель* — ситуацию в работе конвейера, которую нужно проверить при тестировании. Задача генерации тестовых программ сводится к построению представительного множества тестовых шаблонов.

Ниже приведен один из возможных тестовых шаблонов для рассмотренного выше примера. Шаблон состоит из трех инструкций: первые две относятся к классу эквивалентности `IADDInstruction`, а третья принадлежит классу `IDIVInstruction`. Для первой инструкции задана тестовая ситуация *IntegerOverflow=true* (возникновение исключения целочисленного переполнения), для второй инструкции ситуацией является *Exception=false* (отсутствие исключений), а для третьей — *DivisionByZero=true* (деление на нуль). Кроме того, между первой и третьей инструкциями имеется зависимость (первый регистр первой инструкции должен совпадать со вторым регистром третьей инструкции).

```
IADDInstruction R, ?, ? @ IntegerOverflow=true → sub a1, s5, v0
IADDInstruction ?, ?, ? @ Exception=false      → add t7, a0, s3
IDIVInstruction ?, R    @ DivisionByZero=true  → div a2, a1
```

Выполнение описываемого этим шаблоном тестового воздействия начинается с первой инструкции (в рассматриваемом примере это `sub`). Данная инструкция вызывает исключение целочисленного переполнения. Предполагается, что программа обработки исключений устроена таким образом, что управление передается на следующую инструкцию (ей является инструкция `add`). После обработки второй инструкции выполняется третья инструкция (инструкция `div`), в которой происходит деление на нуль.

Заметим, что тестовые шаблоны допускают *параметризацию*. В качестве параметров могут выступать классы эквивалентности инструкций, тестовые ситуации и зависимости. В этом случае каждому шаблону соответствует не одна цель тестирования, а целое семейство целей, для каждой из которых строится свое тестовое воздействие. Описание семейства осуществляется с помощью *итераторов* — компонентов генератора, перебирающих допустимые комбинации значений параметров. Ниже приведен пример шаблона с четырьмя параметрами: `$FirstInstruction` (класс

эквивалентности первой инструкции тестового воздействия), `$Situation` (тестовая ситуация первой инструкции), `$ThirdInstruction` (класс эквивалентности третьей инструкции) и `$Dependency` (множество зависимостей третьей инструкций от других инструкций тестового воздействия).

```
$FirstInstruction @ $Situation
IADDInstruction   @ IntegerOverflow=false
$ThirdInstruction @ $Dependency
```

Настоящая работа сфокусирована на тестировании механизмов управляющей логики, связанных с разрешением конфликтов, поэтому нас в первую очередь интересует описание и построение *шаблонов конвейерных конфликтов* — тестовых шаблонов, вызывающих конфликтные ситуации в работе конвейера. Рассмотрим, какие тестовые ситуации и зависимости для этого используются.

4.3. Тестовые ситуации

Для тестирования управляющей логики основной интерес представляют тестовые ситуации, связанные с выполнением инструкций на конвейере. Как правило, обработка любой инструкции происходит одинаковым образом для всех значений операндов (конечно, если не брать в расчет исключения). Таким образом, для инструкции, которая может вызвать N исключений, обычно определяется N+1 тестовая ситуация: *Exception=false*, $Exception_0$=*true*, …, $Exception_{N-1}$=*true*. Для инструкций, обработка которых зависит от значений операндов, возможна дополнительная детализация тестовых ситуаций.

Отдельно рассматриваются инструкции перехода. Тестовые ситуации для них описывают положение адреса (метки) перехода в тестовом воздействии и выполнимость условий (если переход условный). В общем случае тестовая ситуация для инструкции перехода имеет вид *Target=Label*, *Trace*=$\{C_0, \ldots, C_{M-1}\}$, где *Label* — метка инструкции тестового воздействия, на которую осуществляется переход, C_i — выполнимость условия перехода при i-ой обработке инструкции. В некоторых случаях положение метки может быть обобщено до направления перехода (вперед или назад). Более подробно тестирование механизмов обработки переходов рассматривается в статье [3].

4.4. Зависимости между инструкциями

Зависимостям между инструкциями отводится ключевая роль в создании конвейерных конфликтов. Они бывают двух основных типов: *по регистрам* и *по адресам*. Регистровые зависимости выражаются через совпадение регистров, использующихся в качестве операндов двух инструкций тестового воздействия, и бывают следующих типов:

- *чтение-чтение* — обе инструкции осуществляют чтение из одного регистра;

- *чтение-запись* — первая по потоку управления инструкция осуществляет чтение из регистра, вторая — запись в него;
- *запись-чтение* — первая инструкция осуществляет запись в регистр, вторая — чтение из него;
- *запись-запись* — обе инструкции осуществляют запись в один регистр.

Зависимости по адресам имеют более сложную структуру и связаны с внутренним устройством подсистемы управления памятью (организацией буфера трансляции адресов, кэш-памяти и основной памяти). Примеры зависимостей по адресам приведены ниже:

- *VAEqual* — совпадение виртуальных адресов;
- *TLBEqual* — совпадение записей в буфере трансляции адресов;
- *PAEqual* — совпадение физических адресов;
- L_i*RowEqual* — совпадение строк в кэш-памяти L_i;

Более подробно типы зависимостей по адресам и другие вопросы, связанные с тестированием подсистем управления памятью, рассматриваются в работе [12].

5. Предлагаемый подход

Предлагаемый подход к построению тестовых программ состоит в следующем. На основе анализа документации выделяется набор «интересных» ситуаций в управляющей логике микропроцессора. Основной упор делается на разрешение конфликтов (включая обработку исключений). Для каждого типа конфликта разрабатывается его *обобщенная спецификация*[6] — параметризованный шаблон, позволяющий создать соответствующую конфликтную ситуацию. Такие шаблоны также называются *шаблонами конвейерных конфликтов* или *базовыми шаблонами*. Базовые шаблоны имеют небольшой размер, поскольку конфликты между инструкциями возможны, только если они расположены достаточно близко друг к другу. Для построения тестов задаются *итераторы* значений параметров базовых шаблонов, после чего генератор строит тестовые программы, используя разные значения параметров шаблонов и комбинируя шаблоны между собой.

[6] Спецификация называется обобщенной, поскольку она основывается на общих сведениях о микропроцессоре и не содержит информации о потактовом выполнении инструкций. Управляющая логика определяется неявно путем описания возможных взаимодействий инструкций на конвейере.

5.1. Спецификация конфликтов

Рассмотрим общую схему спецификации конфликтных ситуаций в работе конвейера. Все выделенные на этапе анализа требований ситуации классифицируются по типам. Основными типами являются *исключения*, *конфликты по данным*, *структурные конфликты* и *конфликты по управлению* (см. Рис. 2).

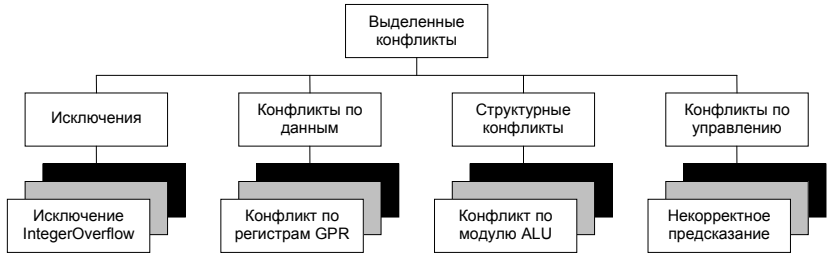

Рис. 2. Выделение конфликтных ситуаций.

Как правило, все конфликтные ситуации, относящиеся к одному типу, описываются одним базовым шаблоном. Различие в их спецификации заключается в разных ограничениях на допустимые значения параметров шаблона (см. Рис. 3). Кроме того, множества значений параметров шаблона разбиваются разработчиком тестов на классы эквивалентности, что служит основой для дальнейшего построения итераторов.

Рис. 3. Спецификация конфликтной ситуации.

Рассмотрим подробнее, как описываются конфликтные ситуации для каждого из указанных типов: исключений, конфликтов по данным, структурным конфликтов и конфликтов по управлению.

5.1.1. Спецификация исключений

Исключением называется событие, сигнализирующее о возникновении нештатной ситуации во время выполнения инструкции. При возникновении

исключения управление передается на специальный обработчик, при этом все инструкции, загруженные на конвейер после инструкции, вызвавшей исключение, сбрасываются. Типичными ошибками в управляющей логике, связанными с обработкой исключений, являются некорректная установка сигнала исключения (например, когда сигнал не устанавливается, хотя условия для исключения выполнены) и некорректная отмена инструкций, загруженных на конвейер.

В тестовых программах, в которых возможны исключения, используются две основные стратегии передачи управления после обработки исключения: управление передается на следующую инструкцию тестового воздействия или управление передается на тестовый оракул, минуя оставшиеся инструкции. Для проверки того, что при возникновении исключения, с конвейера сбрасываются все инструкции, следующие за инструкцией, вызвавшей исключение, предпочтительно использовать вторую стратегию.

Обобщенная спецификация исключения задается базовым шаблоном, представленным ниже.

```
$PreInstructions
$ExceptionInstruction @ $ExceptionType
$PostInstructions
```

В шаблоне используются следующие параметры:

- `$PreInstructions` — цепочка инструкций, предшествующих исключению, (инструкции не должны вызывать исключений);
- `$ExceptionInstruction` — инструкция, вызывающая исключение;
- `$ExceptionType` — тип вызываемого исключения;
- `$PostInstructions` — цепочка инструкций, следующих за исключением, (инструкции, которые должны быть сброшены с конвейера после возникновения исключения).

Для разных типов исключений интерес могут представлять разные типы инструкций, предшествующих исключению и следующих за ним. Ниже приведен пример конкретного тестового воздействия, соответствующего данному базовому шаблону.

```
$PreInstructions                    → dadd   r25, r30, r7
$ExceptionInstruction @ ExceptionType→ lb     r22, 0(r4) //
TLBInvalid=true
$PostInstructions                   → daddiu r5,  r18, 13457
```

Цепочка `$PreInstructions` состоит из одной инструкции `dadd`. В качестве `$ExceptionInstruction` выступает инструкция `lb`,

вызывающая исключение *TLBInvalid* (`$ExceptionType`). Цепочка `$PostInstructions` состоит из одной инструкции `daddiu`.

5.1.2. Спецификация конфликтов по данным

Конфликты по данным возникают, когда разные инструкции пытаются обратиться для чтения или записи к одним и тем же данным, причем хотя бы одна инструкция обращается к ним на запись. Таким образом, для описания конфликтов по данным основной интерес представляют зависимости типа «чтение-запись», «запись-чтение» и «запись-запись», а также их комбинации. Для разных конфликтов могут использоваться разные типы зависимостей. Типичной ошибкой, связанной с разрешением конфликтов по данным, является некорректная реализация блокировок в конвейере, в результате чего происходит нарушение целостности потока данных.

Обобщенная спецификация конфликта по данным задается базовым шаблоном, приведенным ниже.

```
$PreInstructions
$FirstInstruction
$InnerInstructions
$SecondInstruction @ $Dependency
$PostInstructions
```

В шаблоне используются следующие параметры:

- `$PreInstructions` — цепочка инструкций, предшествующих зависимости, (инструкции не должны вызывать исключений);
- `$FirstInstruction` и `$SecondInstruction` — пара инструкций, вступающих в конфликт по данным;
- `$Dependency` — зависимость, приводящая к конфликту по данным;
- `$InnerInstructions` — цепочка инструкций между зависимыми инструкциями (инструкции не должны вызывать исключений и содержать конфликты);
- `$PostInstructions` — цепочка инструкций, следующих за зависимостью, (инструкции, которые могут быть приостановлены вместе с зависимой инструкцией).

Конкретный тип конфликта по данным определяется типами зависимых инструкций и типом зависимости между ними. Ниже приведен пример тестового воздействия, соответствующего данному базовому шаблону.

```
$PreInstructions                →
$FirstInstruction               →madd.s $f18, $f6,  $f28, $f10
$InnerInstructions              →add.s  $f8,  $f17, $f3
$SecondInstruction @ $Dependency→ceil.l.s $f2,  $f18  //
Конфликт по данным
$PostInstructions               →div.s    $f23, $f13, $f24
```

В качестве пары инструкций, вступающих в конфликт, выступают `madd.s` (`$FirstInstruction`) и `ceil.l.s` (`$SecondInstruction`). Они связаны зависимостью типа «запись-чтение» по регистру `$f18`. Цепочка `$PreInstructions` является пустой, `$InnerInstructions` состоит из одной инструкции `add.s`, `$PostInstructions` — из одной инструкции `div.s`.

5.1.3. Спецификация структурных конфликтов

Структурные конфликты возникают, когда несколько инструкций пытаются одновременно обратиться к одному модулю микропроцессора, который не допускает параллельной обработки операций. Как правило, структурные конфликты связаны с близким расположением в коде двух однотипных инструкций, выполнение которых занимает более одного такта (имеется в виду длительность выполнения стадии *EX*). В некоторых случаях для создания конфликта дополнительно требуется наличие зависимости между инструкциями, например, при обращении двух инструкций к кэш-памяти можно выделить в отдельный тип структурного конфликта ситуацию, когда совпадают используемые инструкциями строки кэш-памяти. Типичные ошибки, связанные с разрешением структурных конфликтов, — это, как и для конфликтов по данным, ошибки в реализации блокировок.

Обобщенная спецификация структурного конфликта задается точно таким же базовым шаблоном, что и обобщенная спецификация конфликта по данным. Ниже приведен пример тестового воздействия, в котором создается структурный конфликт.

```
$PreInstructions                →
$FirstInstruction               →  div.s $f11, $f27, $f3
$InnerInstructions              →  add.s $f28, $f7,  $f30
$SecondInstruction @ $Dependency→  div.d $f23, $f1,  $f20 //
Структурный конфликт
$PostInstructions               →  add.d $f18, $f2,  $f25
```

В данном примере все инструкции задействуют модуль арифметики с плавающей точкой (FPU, Floating Point Unit), однако структурный конфликт возникает только между инструкциями деления `div.s` (`$FirstInstruction`) и `div.d` (`$SecondInstruction`) — их выполнение, в отличие от инструкций сложения `add.s` и `add.d`, занимает более одного такта. Цепочка `$PreInstructions` является пустой,

`$InnerInstructions` состоит из одной инструкции `add.s`, `$PostInstructions` — из одной инструкции `add.d`.

5.1.4. Спецификация конфликтов по управлению

Конфликты по управлению связаны с обработкой инструкций перехода. В зависимости от организации микропроцессора выполнение перехода может приводить к приостановке конвейера (когда после выборки инструкции перехода загрузка конвейера приостанавливается до тех пор, пока не будет принято решение о переходе) или сбросу конвейера (когда все инструкции, загруженные на конвейер после инструкции перехода, сбрасываются, если принимается решение о выполнении этого перехода). Ошибки в разрешении конфликтов по управлению могут быть связаны с блокировками конвейера, предсказанием переходов и другими механизмами обработки переходов.

Обобщенная спецификация конфликта по управления задается базовым шаблоном, приведенным ниже.

```
$PreInstructions
$BranchInstruction @ $Target, $Trace
$DelaySlots
$PostInstructions
```

В шаблоне используются следующие параметры:

- `$PreInstructions` — инструкции, предшествующие переходу;
- `$BranchInstruction` — инструкция перехода;
- `$Target` — адрес перехода (указывает на одну из инструкций шаблона);
- `$Trace` — трасса выполнения перехода (последовательность из значений истинности условия перехода);
- `$DelaySlots` — инструкции в слотах задержки[7];
- `$PostInstructions` — инструкции, следующие за переходом.

Подробнее вопросы тестирования модулей обработки переходов рассматриваются в работе [3]. Ниже приведен пример тестового воздействия, соответствующего данному базовому шаблону.

```
$PreInstructions                      →  L:addi r1, r1,  1
$BranchInstruction @ $Target, $Trace→     beq  r1, r0,  L //
Target=L, Trace={1, 0}
$DelaySlots                           →     dadd r7, r12, r23
$PostInstructions                     →
```

[7] Инструкции, следующие за инструкцией перехода, которые всегда выполняются микропроцессором при обработке перехода (не зависимо от того, выполнено условие перехода или нет). Число слотов задержки переходов является постоянной величиной для каждой микропроцессорной архитектуры, например, в MIPS имеется один слот задержки.

В качестве инструкции перехода `$BranchInstruction` выступает инструкций `beq`. Переход осуществляется на первую инструкцию тестового воздействия, помеченную меткой `L` (`$Target`). При первом выполнении перехода условие выполнено, при втором — нет (`$Trace`). Цепочка `$PreInstructions` состоит из одной инструкции `addi`. В слоте задержки перехода находится инструкция `dadd` (`$DelaySlots`). Цепочка `$PostInstructions` является пустой.

5.2. Построение тестовых программ

Теперь рассмотрим, как на основе базовых шаблонов (шаблонов конвейерных конфликтов) строятся тестовые воздействия, используемые в тестовых программах. Сразу отметим, что тестовые воздействия делятся на два типа: *простые* (соответствующие одному базовому шаблону) и *составные* (соответствующие композиции нескольких базовых шаблонов). В соответствии с этим описание методики разбито на две части.

5.2.1. Построение простых тестовых воздействий

Простые воздействия, как правило, нацелены на создание одной конфликтной ситуации в работе конвейера. Методика их построения является простой и основана на использовании *базовых шаблонов* и *итераторов* (см. Рис. 4). Для каждой выделенной ситуации, вообще говоря, строится несколько тестовых воздействий, различающихся значениями параметров (эти значения перебираются в итераторах). Множество воздействий строится путем комбинирования значений параметров (обычно используются все возможные комбинации).

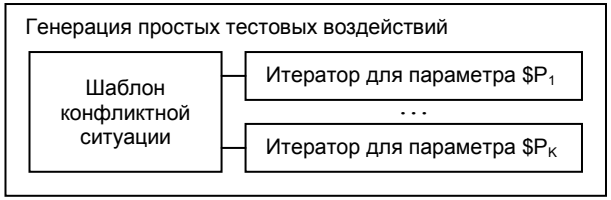

Рис. 4. Генерация простых тестовых воздействий для одной конфликтной ситуации.

Рассмотрим, например, структурный конфликт по модулю FPU, который описывается следующим базовым шаблоном:

```
$PreInstructions
$FirstInstruction
$InnerInstructions
$SecondInstruction @ $Dependency
$PostInstructions
```


Для этого конфликта определены некоторые ограничения на значения параметров. Например, конфликт возможен только между инструкциями, выполняющимися более одного такта, при этом длина последовательности `$InnerInstructions` не должна превышать уменьшенной на два такта длительности выполнения инструкции `$FirstInstruction`. Кроме того, могут быть заданы классы эквивалентности зависимых инструкций. Если, например, блок умножения реализован отдельно от блока деления, инструкции умножения и деления могут и не вызывать структурного конфликта, поэтому следует рассматривать два конфликта (между инструкциями умножения и между инструкциями деления).

Пусть для рассматриваемого конфликта определены следующие классы эквивалентности инструкций и итераторы значений параметров:

```
FMULInstruction    : {mul.s, mul.d}
FDIVInstruction    : {div.s, div.d}
IADDInstruction    : {add, sub}

$PreInstructions   : {}
$FirstInstruction  : FMULInstruction, FDIVInstruction
$SecondInstruction : FMULInstruction, FDIVInstruction
$Dependency        : class($FirstInstruction) ==
class($SecondInstruction)
$InnerInstruction  : {IADDInstruction}
$PostInstructions  : {}
```

Тогда будут построены два тестовых воздействия следующего вида:

```
$PreInstructions     →                                  →
$FirstInstruction    →   mul.d $f12, $f3,  $f21   →
div.s $f18, $f28, $f4
$InnerInstructions   →   sub   r6,   r15,  r3     →
add   r25,  r13,  r27
$SecondInstruction   →   mul.s $f9,  $f23, $f7    →
div.s $f5,  $f12, $f10
$PostInstructions    →                                  →
```

5.2.2. Построение составных тестовых воздействий

Целью составных тестовых воздействий, в отличие от простых, является создание нескольких «одновременных» конфликтов в работе конвейера. Составные воздействия позволяют проверить сложные ситуации в управляющей логике микропроцессора. К таким ситуациям относятся *параллельные конфликты* (когда, например, структурный конфликт для пары инструкций возникает одновременно с конфликтом по данным), *вложенные конфликты* (когда один конфликт происходит на фоне другого), *параллельные исключения* (когда за счет того, что исключения в разных инструкциях могут возникать на разных стадиях, они происходят одновременно) и многие другие.

Построение составных воздействий осуществляется путем композиции нескольких базовых шаблонов. Под композицией понимается как простая конкатенация шаблонов, так и более сложная их комбинация, при которой они могут иметь общие части, в частности, при которой один шаблон может быть вложен в другой.

Обозначим буквой T (возможно с индексом) шаблон произвольного типа, T_E — шаблон для исключения, T_H — шаблон для конфликта по данным или структуре, T_C — шаблон для конфликта по управлению. Основные операции композиции шаблонов приведены в следующей таблице[8].

Операция	***Обозначение***	***Семантика***
Наложение	$T=T_{H1}\|T_{H2}$	$T_H.PreInstructions = T_{H1}.PreInstructions \equiv T_{H2}.PreInstructions$ $T_H.FirstInstruction = T_{H1}.FirstInstruction \equiv T_{H2}.FirstInstruction$ $T_H.SecondInstruction = T_{H1}.SecondInstruction \equiv T_{H2}.SecondInstruction$ $T_H.Dependency = T_{H1}.Dependency \,\&\, T_{H2}.Dependency$ $T_H.InnerInstructions = T_{H1}.InnerInstructions \equiv T_{H2}.InnerInstructions$ $T_H.PostInstructions = T_{H1}.PostInstructions \equiv T_{H2}.PostInstructions$
Сдвиг	$T_H=T_{H1}\downarrow T_{H2}$	$T_H.PreInstructions = T_{H1}.PreInstructions$ $T_H.FirstInstruction = T_{H1}.FirstInstruction$ $T_H.SecondInstruction = T_{H1}.SecondInstruction$ $T_H.Dependency = T_{H1}.Dependency \,\&\, T_{H2}.Dependency$ $T_H.InnerInstructions = \{T_{H1}.InnerInstructions, T_{H2}.FirstInstruction, T_{H2}.InnerInstructions\}$ $T_H.PostInstructions = \{T_{H1}.PostInstructions, T_{H2}.SecondInstruction, T_{H2}.PostInstructions\}$ $T_{H1}.PostInstructions \equiv T_{H2}.PreInstructions$
Конкатенация	$T=T_1\rightarrow T_2$	$T.PreInstructions = T_1.PreInstructions$ $T.MainParameters = T_1.MainParameters$[9] $T.PostInstructions = T_2$ Тип шаблона T совпадает с типом шаблона T_1
Вложение	$T_H=T_{H1}[T]$	$T_H.FirstInstruction = T_{H1}.FirstInstruction$ $T_H.SecondInstruction = T_{H1}.SecondInstruction$ $T_H.Dependency = T_{H1}.Dependency$ $T_H.PreInstructions = T_{H1}.PreInstructions$ $T_H.InnerInstructions = T$ $T_H.PostInstructions = T_{H1}.PostInstructions$

Для того чтобы прояснить семантику составных шаблонов и способ построения по ним тестовых воздействий, рассмотрим примеры.

[8] Набор операций композиции является открытым и может быть расширен.

[9] Под *MainParameters* понимается набор параметров шаблона за исключением *PreInstructions* и *PostInstructions*.

Наложение шаблонов:

```
$PreInstructions1,2                    →   add.s  $f28, $f7,  $f30
$FirstInstruction1,2                   →   div.s  $f11, $f27, $f3
$InnerInstructions1,2                  →   dsub   r25,  r30,  r7
$SecondInstruction1,2 @ $Dependency1 & $Dependency2→   div.d
$f23, $f11, $f20
$PostInstruction1,2                    →   lb     r22,  0(r4)
```

Сдвиг шаблонов:

```
$PreInstructions1                      →   add.s  $f28, $f7,  $f30
$FirstInstruction1                     →   div.s  $f11, $f27, $f3
$InnerInstructions1                    →   dsub   r25,  r30,  r7
    $FirstInstruction2                 →   sub    r13,  r5,   r10
    $InnerInstructions2                →   add    r8,   r11,  r3
$SecondInstruction1 @ $Dependency1→    div.d  $f23, $f1,  $f20
    $PostInstrucions1 ≡ $PreInstructions2→ lb     r22,  0(r4)
    $SecondInstruction2 @ $Dependency2→   div    r12,  r13
    $PostInstructions2                 →   daddiu r5,   r18,  54
```

Конкатенация шаблонов:

```
$PreInstructions1                      →   add.s  $f28, $f7,  $f30
$FirstInstruction1                     →   div.s  $f11, $f27, $f3
$InnerInstructions1                    →   dsub   r25,  r30,  r7
$SecondInstruction1 @ $Dependency1→    div.d  $f23, $f1,  $f20
    $PostInstruction1 ≡ $PreInstructions2→lb    r22,  0(r4)
    $FirstInstruction2                    →sub   r13,  r5,   r10
    $InnerInstructions2                   →add   r8,   r11,  r3
    $SecondInstruction2 @ $Dependency2    →div   r12,  r13
    $PostInstructions2                    →daddiu r5,  r18,  54
```

Вложение шаблонов:

```
      $PreInstructions1                   →add.s $f28, $f7, $f30
      $FirstInstruction1                  →div.s $f11, $f27, $f3
          $PreInstructions2               →lb    r22,  0(r4)
          $FirstInstruction2              →sub   r13,  r5,   r10
          $InnerInstructions2             →add   r8,   r11,  r3
          $SecondInstruction2 @ $Dependency2→div   r12,  r13
          $PostInstructions2              →daddiu r5,  r18,  54
      $SecondInstruction1 @ $Dependency1→div.d $f23, $f1, $f20
      $PostInstructions1                  →nop
```

Интуитивно понятно, как осуществляется итерация разных тестовых воздействий для заданного составного шаблона. Некоторые параметры разных базовых шаблонов отождествляются, после этого для полученного множества параметров задаются итераторы (как правило, используются итераторы, заданные для простых воздействий). Тонким моментом является перебор

меток переходов. Если ограничить положение меток только внутренними инструкциями базовых шаблонов получится достаточно ограниченный набор ситуаций, поэтому допускается перебор меток за пределами базовых шаблонов. Генерация составных шаблонов осуществляется путем перебора различных синтаксических структур из небольшого числа базовых шаблонов, связанных операциями композиции.

6. Практическая апробация подхода

Рассматриваемый подход был применен для верификации управляющей логики двух сопроцессоров: сопроцессора вещественной арифметики (CP1) и сопроцессора комплексной арифметики (CP2). Сопроцессоры имеют общий поток инструкций с центральным микропроцессором и используют три канала выполнения (функциональных конвейера):

- канал арифметики с плавающей точкой (вещественной и комплексной);
- канал обращения к основной памяти;
- канал обращений к накристальной памяти.

Управляющая логика микропроцессора поддерживает разрешение конфликтов разных типов. В обоих сопроцессорах могут возникать исключения при доступе к основной памяти. Кроме того, в сопроцессоре CP1 возможны исключения при выполнении арифметических инструкций. В микропроцессоре реализована возможность статического предсказания переходов и спекулятивного выполнения инструкций.

В тестовых программах использовались воздействия из четырех инструкций разных типов (применялись как простые, так и составные воздействия). Структура используемых тестовых ситуаций и зависимостей близка к описанной в статье, однако были учтены некоторые микроархитектурные особенности сопроцессоров путем введения дополнительных зависимостей по данным.

Следующая таблица показывает объем разработанных спецификаций, шаблонов и других компонентов генератора тестовых программ для разных ревизий микропроцессора[10]; число реализуемых сопроцессорами инструкций; число инструкций, затронутых изменениями в соответствующей ревизии[11] и долю исходного кода генератора, который пришлось изменить[12].

№ ревизии	*Объем кода (в строках)*	*Число инструкций*	*Число затронутых инструкций*	*Объем измененного кода (в строках)*
Сопроцессор CP1				
8	28500	113	—	—
20	28650	114	94	485 (1.7%)
Сопроцессор CP2				
8	4950	15	—	—
20	11550	59	5	45 (0.9%)
29	14350	100	18	165 (1.4%)

Таб. 1. Данные по апробации предлагаемого подхода.

Как видно из таблицы, при модификации микропроцессора изменяется очень небольшой объем кода генератора. В нашем случае изменению подвергались только спецификации инструкций (мнемоника, число тактов выполнения, набор операндов и, частично, семантика). Для всех указанных ревизий правка занимала не более получаса, а генерация тестовых программ потребовала от 30 минут до 5 часов (в зависимости от детальности используемых зависимостей между инструкциями).

В результате верификации было найдено значительное число ошибок в обоих сопроцессорах, которые не были обнаружены с помощью тестовых программ, сгенерированных случайно. Ошибки были найдены как в эталонном симуляторе, так и в RTL-модели микропроцессора.

7. Заключение

Тестирование управляющей логики микропроцессоров является нетривиальной задачей, которую практически невозможно решить без применения методов автоматизации. В статье рассмотрена методика автоматизированной генерации тестовых программ на основе шаблонов конфликтных ситуаций возможных в работе конвейера. В отличие от распространенных на практике подходов (включая ручную разработку и случайную генерацию), предлагаемая методика имеет высокий уровень автоматизации и позволяет на систематичной основе проверить управляющую логику микропроцессора. В то же время, методика отличается от подходов на основе точных моделей управляющей логики тем, что она может применяться

[10] Разработка тестов началась с ревизии №8. В таблице отражены только те ревизии, которые потребовали изменения компонентов генератора тестовых программ.

[11] Число затронутых инструкций не включает в себя инструкции, добавленные в соответствующей ревизии.

[12] Объем измененного кода не учитывает добавление нового кода, описывающего добавленные инструкции.

на ранних этапах разработки, когда структура конвейера не является стабильной, а проект подвержен частым изменениям.

Использование потактовых моделей управляющей логики целесообразно на поздних этапах проектирования. Здесь, за счет большей информативности точных моделей, такие подходы имеют преимущества перед другими методами и потенциально позволяют обнаружить самые сложные ошибки. Кроме того, за счет использование точных моделей возможно построение более компактного набора тестов. В будущем мы планируем расширить генератор тестовых программ средствами разработки точных спецификаций конвейера и построения на их основе автоматных моделей. Таким образом, на ранних этапах разработки можно использовать генерацию на основе шаблонов, а на более поздних, когда стабилизируется управляющая логика, — тесты на основе обхода конечных автоматов.

Литература

[1] Википедия (http://en.wikipedia.org), статья *Instruction pipeline*.

[2] D. Patterson, J. Hennessy. Computer Organization and Design: The Hardware-Software Interface, 2nd edition, 1997.

[3] А.С. Камкин. Некоторые вопросы автоматизации построения тестовых программ для модулей обработки переходов микропроцессоров. Труды ИСП РАН, 2010 (этот же сборник).

[4] P. Mishra, N. Dutt. *Specification-Driven Directed Test Generation for Validation of Pipelined Processors*. ACM Transactions on Design Automation of Electronic Systems, 2008.

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

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

[7] T.N. Dang, A. Roychoudhury, T. Mitra, P. Mishra. *Generating Test Programs to Cover Pipeline Interactions*. Design Automation Conference, 2009.

[8] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. *Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification.* Design and Test of Computers, 2004.

[9] S. Ur, Y. Yadin. Micro-Architecture Coverage Directed Generation of Test Programs. Design Automation Conference, 1999.

[10] А.С. Камкин. Генерация тестовых программ для микропроцессоров. Труды ИСП РАН, 2008.

[11] MIPS64™ Architecture For Programmers. Revision 2.0. MIPS Technologies Inc., 2003.

[12] Д.Н. Воробьев, А.С. Камкин. Генерация тестовых программ для подсистемы управления памятью микропроцессора. Труды ИСП РАН, 2009.