УДК 004.415.538, 004.434 DOI 10.31114/2078-7707-2018-2-9-15

Система комбинируемых специализированных генераторов тестов для нового поколения VLIW DSP процессоров с архитектурой Elcore50

А.В. Гаращенко[1,2], А.В. Николаев[1], Ф.М. Путря[1], С.С. Сардарян[1]

[1]ОАО НПЦ «ЭЛВИС», г. Москва, г. Зеленоград,

[2]Национальный исследовательский университет «МИЭТ», г. Москва, г. Зеленоград,

ant.gar1@mail.ru

***Аннотация* — Даже если не вдаваться в подробности реализации микроархитектуры, пространство состояний современного ядра астрономически огромно. Оно определяется сочетанием набора команд во VLIW инструкции, динамическим сочетанием команд и зависимостей между ними в конвейере, динамикой исполнения обращений к памяти (которых во VLIW процессоре может исполняться несколько в рамках одной команды), внешними прерываниями, состоянием подсистемы отладки. В данной работе рассмотрено решение на основе системы сочетаемых специализированных генераторов, позволяющих организовывать иерархические вызовы разных генераторов в процессе создания теста, и, таким образом, добиться расширения покрываемых подмножеств глобального пространства состояний процессора, не снижая при этом вероятность формирования краевых ситуаций для отдельных подсистем и подмножеств свойств процессора, на которые нацелены каждый из специализированных генераторов в отдельности.**

***Ключевые слова* — верификация процессоров, широкое командное слово, подсистема памяти, генерация тестов, покрытие.**

I. Введение

В связи с архитектурной сложностью современных многоядерных процессоров, применяемых при разработке систем на кристалле (СнК), более шестидесяти процентов ресурсов проектирования тратится на их верификацию. Это обусловлено высокой комбинаторной сложностью проверки корректности работы как отдельных ядер, так и системы в целом. Помимо этого, существует тенденция усложнения СнК за счет повышения гетерогенности, связанной с необходимостью увеличения скорости выполнения отдельных классов задач. Такие вычислительные системы состоят из ядер общего назначения и специализированных вычислительных ядер. Т.е. стоит задача проверки реализаций разных архитектур вычислительных ядер, входящих в состав проектируемой СнК, которые могут отличаться не только системой команд, но и способом компоновки команд во VLIW-пакет (VLIW - Very Long Instruction Word), его шириной, организацией регистровых файлов, кэшей и многим другим. Вычислительная сложность современных алгоритмов формальной верификации ограничивает область их применения небольшими блоками (типа ALU или отдельных элементов подсистемы памяти) или отдельными свойствами процессора, которые легко локализовать, что резко ограничивает использование формальных методов для тестов уровня полного ядра процессора [1-4]. Таким образом, динамическая верификация пока ещё остается одним из основных методов проверки вычислительных ядер.

Если касаться особенностей верификации именно специализированных микропроцессоров обработки цифровых сигналов (далее DSP), то среди них стоит выделить широкое комбинаторное пространство возможных ситуаций. Зачастую такие процессоры имеют гарвардскую VLIW архитектуру со скалярным и векторным исполнительными каналами, аппаратными циклами и многоканальной памятью. Их верификация требует огромного объема сложных тестов, что становится основной проблемой функциональной верификации. Однако ограничивающим фактором является время, требуемое для разработки полного набора тестов. Ввиду чего не может идти речи о написании всех тестов вручную. Это определяет высокую актуальность создания новых средств для проверки правильности работы подобных структур.

Для увеличения скорости моделирования, а, значит, и проверки правильности работы, каждое процессорное ядро и даже некоторые его модули верифицируются в автономных тестовых окружениях (при условии, что проект процессора написан с учетом требований со стороны задачи декомпозиции верификации [5, 6]), так как скорость моделирования отдельных частей процессора на порядок больше. Однако задача создания всеобъемлющего набора тестовых последовательностей для процессорного ядра остается критической. Далеко не все подсистемы можно локализовать в виде отдельного блока, и даже там, где это возможно, могут иметь место ошибки в протоколе межблочного взаимодействия. Даже если не вдаваться в

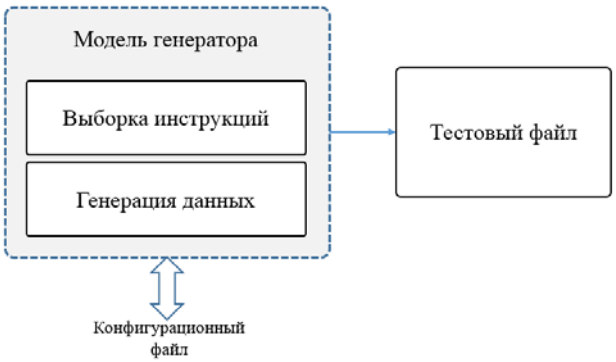
подробности реализации микроархитектуры, пространство состояний современного ядра астрономически огромно. Оно определяется сочетанием набора команд во VLIW-инструкции, динамическим сочетанием команд и зависимостей между ними в конвейере, динамикой исполнения обращений к памяти, которых во VLIW-процессоре может исполняться несколько в рамках одной команды, внешними прерываниями, состоянием подсистемы отладки. Генераторы тестов уже давно служат в качестве основного инструмента для покрытия вычислительных ядер тестами [7-9]. Однако объем пространства состояний такой, что один генератор общего назначения будет чрезмерно сложным и с редким выходом краевых ситуаций, что приведет к неприемлемо долгому процессу генерации и запуску тестов до момента достижения требуемого покрытия. Специализированные генераторы тестов на отдельные подсистемы или подмножества свойств процессора, например, отдельные генераторы тестов на набор команд программного управления, на подсистему памяти, на подсистему прерываний позволяют создать большое число краевых ситуаций для целевой подсистемы и, соответственно, добиться их большего покрытия. Однако, с точки зрения системы, специализированные генераторы покрывают лишь отдельные локализованные подмножества глобального пространства состояний, оставляя большие пробелы между подмножествами в этом пространстве. Примером промежуточного состояния, не покрываемого специализированными генераторами, может быть прерывание в момент исполнения заданной комбинации команд программного управления параллельно с исполнением команд обращения в кэш память, конфликтующих друг с другом при обращении к кэш строке. Полностью случайный поток команд такое состояние создаст через годы моделирования, а направленные специализированные тесты не создадут вовсе, поскольку используют ограничивающие их поведение шаблоны или модели.

В данной работе предложено решение на основе системы сочетаемых специализированных генераторов, позволяющих организовывать иерархические вызовы разных генераторов в процессе создания теста, и, таким образом, добиться расширения покрываемых подмножеств глобального пространства состояний процессора. Например, сочетание генератора на программное управление (генерация исключений) с генератором на подсистему памяти (исключения в условиях длительной блокировки конвейера). Такие тесты выполняют проверку системы в целом, так как одновременная работа многих частей процессорного ядра создаёт критические ситуации, вероятность появления которых при их направленной верификации очень мала. Для их генерации необходимо учесть существующие методики и подходы тестирования различных блоков процессора. Сгенерированные тестовые последовательности позволяют осуществлять проверку взаимодействия различных устройств процессорного ядра в рамках одного теста.

II. Модель генератора потока случайных инструкций

Архитектура модели, позволяющей решить задачу генерации случайных последовательностей, приведена на рис. 1. Тесты для системы команд DSP представляют собой последовательности ассемблерных инструкций. Такое тестирование позволяет проверить эти инструкции на предмет правильности выполнения с поведенческой точки зрения путем тестирования случайными входными данными. Для каждой инструкции случайными данными являются: исходное состояние используемых регистров (как входных, так и выходных), номера этих регистров, состояние и адреса ячеек памяти в форматах, требующих пересылок. Сгенерированная программа предназначается для выполнения на RTL-модели (register transfer level) и симуляторе. Для применения такого генератора при верификации другого микропроцессора необходима минимальная модификация программы, а именно – изменить конфигурационный файл, содержащий мнемоники ассемблерных команд и их форматы.

Рис. 1. Архитектура генератора случайных инструкций



Генерация тестовых данных включает в себя произвольный выбор тестируемой инструкции из списка, получаемого из конфигурационного файла; выбор формата из возможных для данной инструкции; выбор регистров для инструкции в зависимости от формата пересылок; инициализацию регистра состояния. Для заданной инструкции и параметров вычисляется эталонное значение. При инициализации значений 64-х и 128-и разрядных регистров используются промежуточные пересылки через два 32-х разрядных регистра, поэтому они зарезервированы и не используются в качестве источников или приемников вычислительных команд и пересылок.

Возможны два сценария использования данной модели генератора.

По первому сценарию тест состоит из нескольких подтестов, записываемых в PRAM-память процессора. Каждый подтест заключается в инициализации начальных данных (регистров и памяти), выполнении одной тестовой команды, проверке результата и записи результата выполнения в память. По умолчанию программа генерируют тесты по первому сценарию работы.

По второму сценарию тест заключается в инициализации регистрового файла, адресных регистров и соответствующих им ячеек памяти начальными произвольными значениями; выполнении заданного количества произвольных инструкций. Количество генерируемых инструкций ограничено 4000. Поддерживается возможность задания доли базовых инструкций и, соответственно, расширенных. Результатом теста является состояние регистрового файла. Корректность выполнения теста проверяется сравнением архитектурного состояния RTL и эталонной модели процессора.

Особенностью данного генератора является способность создавать VLIW-пакеты, содержащие векторные и скалярные инструкции. Поддерживается генерация пакетов как заданной длины, так и случайной (ограниченной максимальной длиной). Инструкции попадают в один пакет с учетом ограничения количества команд данного типа, которые можно исполнить в рамках одного VLIW-пакета.

III. Модель генератора тестов на программное управление

При верификации устройства управления DSP-процессора следует учитывать особенности его архитектуры. Основным объектом тестирования является конвейер микропроцессора. В основном проверяются ситуации, приводящие его к различным блокировкам. В рамках рассматриваемой модели генератора существует необходимость покрытия пространства блокировок, вызванных программными переходами, подпрограммами, аппаратными циклами и исключениями; а также блокировок, вызванных зависимостью по данным между следующими друг за другом инструкциями.

Рис. 2. Структурная схема генератора на программное управление

Структура, позволяющая решить данную задачу, приведена на рис. 2. Модуль управления генераторами задаёт общие правила генерации теста, описанные в конфигурационном файле. Они поступают на вход генераторов инструкций и данных, которые осуществляют формирование последовательностей, состоящих из исполняемого кода, связей между ними; а также создание зависимых и независимых операндов для инструкций соответственно. Для расчёта значений регистров и памяти полученный код запускается на функциональной модели, помимо этого обеспечивая возможность получения текущего состояния тестируемой системы в целом. Диспетчер запросов отвечает за обработку запросов функциональной модели к памяти за инструкциями и данными.

С помощью такого генератора возможно генерировать тесты вида, который приведен на рис. 3, где доступны программные условные и безусловные переходы по относительным (j-команды) и абсолютным (b-команды) адресам со слотами задержки, а также с сохранением адреса возврата; вызов подпрограмм, в том числе, и из подпрограммы (кроме рекурсии); вложенные аппаратные циклы [10].

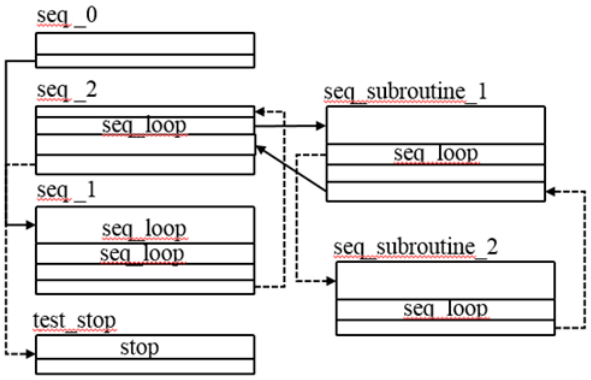


Рис. 3. Структурная схема теста

Основными составляющими секции с переходами являются последовательности двух типов: seq_n, seq_subroutine_n. Seq – часть кода, состоящая из случайных инструкций, связанных условными или безусловными переходами. Seq_subroutine – подпрограммы со своим стеком. Переходы генерируются как с положительным, так и с отрицательным смещением.

Вызов подпрограмм реализован с помощью передачи адреса возврата, а также аргументов через регистры в функцию с дальнейшей записью в память (общий стек). Указатель на верхушку стека размещается в памяти по известному адресу, при этом этот адрес генерируется случайным образом с учетом ограничений карты памяти микропроцессора. Когда происходит запись в память, указатель инкрементируется, при извлечении – декрементируется. После выполнения тела функции адрес читается из стека в регистр с последующим переходом по нему. Возможен вызов подпрограмм из других подпрограмм, при этом стоят ограничения от зацикливаний, учитывая возможную косвенную или прямую рекурсию. Количество вызовов подпрограмм ограничено размером стека.

Реализована генерация программных и аппаратных циклов. Программные циклы сделаны с помощью условных и безусловных переходов. Глубина циклов, как и количество вызовов подпрограмм, задаётся в конфигурационном файле. Однако следует учесть, что

глубина аппаратных циклов ограничена аппаратурой, поэтому при генерации циклов в подпрограммах учитывается их вложенность до этого.

Отличительной особенностью рассматриваемого генератора является возможность создания обработчиков исключений, адреса которых помещаются в специальный регистр. Помимо этого генерируются исключения трех типов:

1) UI (Unknown Instruction) – неизвестная инструкция;
2) BA (Bad Addres) – неверный адрес;
3) SYSCALL – системные вызовы.

Первый тип исключений создаётся с помощью различных прыжков на адреса с несуществующими инструкциями, а также прямой записью в регистр исключений.

Второй тип генерируется путем обращения в память PRAM по запрещенным DSP-процессором адресам или Scatter-Gather- обращением в кэш с одним адресом, не принадлежащим к диапазону PRAM памяти.

Пример:

```
seq_subroutine _1:
   stl r31, (r0)
   addl 4, r0, r0
   do 4 llsub_11_end
   xor r12, r1, r3
   minl r7.l, r4.l, r6.l
      do 6, llsub_12_end
   bs seq_subroutine _2
      llsub_12_end:
   subl 4, r0, r0
   llsub_11_end:
   ldl (r0), r31
   b r31
```

```
seq _0:
   subl r22, r12, r15
   addl r22, r17, r19
   bs seq_subroutine _1
   bs seq_subroutine _1
   stl r0, (r=0x1200000)
   j test_stop
```

В примере выше приведена часть сгенерированного теста, состоящая из одной последовательности типа seq, в которой присутствует два вызова подпрограммы seq_subroutine_1. В подпрограмме seq_subroutine_0 сначала сохраняется адрес возврата, затем указатель стека смещается на 4 единицы вверх, после чего во вложенном цикле идет вызов подпрограммы seq_subroutine_2. Команды разделяются сгенерированными случайными инструкциями. В конце seq_subroutine_0 считывается адрес возврата в регистр 31, указатель на верхушку стека смещается на 4 единицы вниз и происходит переход по адресу в регистре 31.

Несмотря на наличие функциональной модели у генератора, для проверки тестируемого ядра микропроцессора на предмет несоответствия поведения тестируемого процессора и эталонной модели в основном используется режим сравнения результатов работы RTL и симулятора (TLM модели). Преимущество такого подхода состоит в том, что проверка правильности поведения компонента выполняется автоматически.

IV. Модель генератора тестов иерархии кэшей

Конструирование тестов данным генератором осуществляется путем построения графовой модели иерархии кэш-памяти, вершинами которой являются состояния кэша (рис. 4), а ребрами – переходы между состояниями (инструкции, написанные на метаязыке).

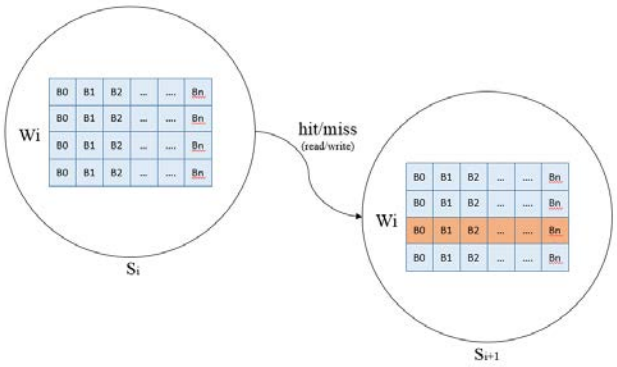


Рис. 4. Графовая модель иерархии кэш памяти

Метаязык дает возможность работать с памятью микропроцессора на более абстрактном уровне, отвязывая тесты от конкретного ассемблера, что позволяет переносить тесты для верификации процессоров с отличной архитектурой. Он содержит функции чтения (read) и записи (write). Предусмотрена векторно-блочная работа с памятью. Для конвертации метаязыка в ассемблерные команды разработан специальный генератор, который поддерживает не более 16 векторно-блочных обращений к памяти.

Между состояниями (вершинами графа) кэша генератором предусмотрены следующие переходы:

1) hit(int mode) – попадание в выбранный кэш по чтению или записи. Параметр mode определяет какой адрес будет использован, если он равен 0, используется случайный закэшированный адрес, в случае если mode равен 1, используется адрес последней исполненной команды.

2) miss() – промах по чтению или записи.

3) parallel_hit_miss(int mode=0) - параллельные операции попадания и промахов по чтению и записи. Параметр mode работает как в случае с hit.

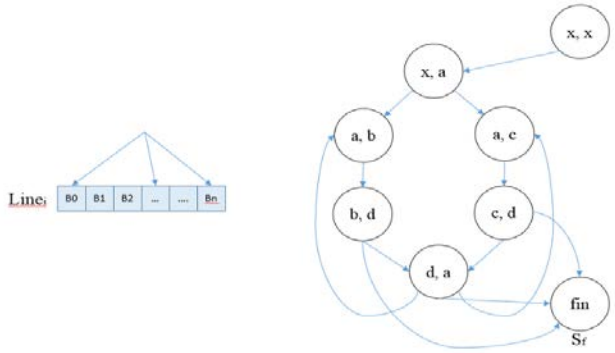
4) replace() - вытеснение строки, использованной в последней исполненной команде.

5) next_line() – чтение или запись в двух соседних строках от строки, использованной в последней исполненной команде, при этом в саму строку обращения нет.

6) next_way() – чтение или запись в случайный ассоциативный путь для строки, использованной в последней исполненной команде, при этом в саму строку обращения нет.

При генерации всех переходов адрес, тип обращения и размер транзакции задаются случайным

образом. Пример графа для строки одного кэша и двух



ассоциативных путей показан на рисунке 5.

Рис. 5. Графовая модель для одной кэш строки

По описанию, приведенному в конфигурационном файле (рис. 6), строится графовая модель. Каждый переход в графе - это отдельный тест. Сначала генерируется преамбула, состоящая из последовательностей команд метаязыка, приводящих систему кэш-памяти к нужному состоянию; далее идет блок фиксации дампа памяти; затем инструкция, отвечающая за переход; блок проверки дампа памяти с

```
L0 {
    ways = 1;
    lines=128;
    lineSize=32;
    group=0;
    lineAdr=0;
    tag=A[31:5];
    byteAtdr=A[4:0];
}

L1 {
    ways = 8;
    lines=256;
    lineSize=64;
    group=A[6];
    lineAdr=A[11:7];
    tag=A[31:12];
    byteAtdr=A[5:0];
}

L2 {
    ways = 16;
    lines=256;
    lineSize=64;
    group=A[6];
    lineAdr=A[11:7];
    tag=A[31:12];
    byteAtdr=A[5:0];
}
```

эталонной моделью.

Рис. 6. Описание иерархии кэш памяти

Возможные критические ситуации для кэша (без учета когерентности), создаваемые рассматриваемым генератором: цепочки запись-чтение, запись-запись-чтение-чтение; обращения по адресам после вытеснения закэшированных по ним данных; вытеснение строчки с дальнейшим чтением из нее; запись или чтение данных, переходящих через границу одной строки в одном ассоциативном пути для вытеснения в нем сразу двух строк; запись или чтение данных, переходящих через границу одного ассоциативного пути, чтобы вытеснять сразу две строки в двух разных путях. Генерация команды инвалидации позволяет создать дополнительную нагрузку на систему кэш-памяти. В дальнейшем предполагается наложить все это на механизм когерентности.

Для генерации тестов для связки L1 и L2 кэшей требуется дополнительно указать в конфигурационном файле следующие моменты: является ли L1 инклюзивным; как происходит вытеснение из L1 и вытеснение сразу из L1 и L2; как происходит промах в L0 с вытеснением в L1.

Отличительной особенностью генератора является поддержка генерации тестовых сценариев для многоканального режима обращения в кэш-память. Вариант по умолчанию — преамбула, состоящая из последовательностей команд метаязыка на одном канале с одновременным переходом в разные состояния по всем допустимым каналам и проверка данных сравнением с эталонной моделью. Также в генерацию тестов добавлены случайные инструкции обращения к внешней и внутренней памяти.

V. Генерация внешних прерываний

Внешнее прерывание, приходящее в общем случае асинхронно к потоку исполнения команд, должно привести к выходу на обработчик, его исполнению, и корректному возврату к процессу исполнения основного потока управления при любом исходном состоянии конвейера. Внешнюю генерацию прерываний для RTL-модели организовать легко. Однако, есть две проблемы.

Первая проблема — нужно комбинаторно перебрать состояния ядра в момент прихода прерывания. Для этих целей в качестве программы жертвы используется программа, генерируемая с помощью всех описанных выше генераторов и их комбинаций.

Вторая проблема — проверка корректности входа в обработчик и возврата из него. Классическим способом проверки является сравнение трасс и состояний в контрольных точках RTL и TLM-моделей. Однако реализовать подачу прерываний в одинаковые моменты времени с точки зрения конвейера моделей разных уровней абстракции крайне сложно. Можно было бы добиться потактовой точности TLM-модели и использовать информацию о внутреннем состоянии конвейера для генерации события прерывания, однако, данный подход требует слишком больших трудозатрат. На данном этапе было решено использовать упрощенный механизм сравнения. Факт выхода в обработчик и корректность его работы проверяется отдельным монитором (при этом совпадение трасс не требуется). В свою очередь трассы сравниваются для всех инструкций, кроме тех, которые исполняются в обработчике прерываний. Сходимость основной программы обеспечивается процедурой восстановления контекста при выходе из обработчика.

VI. Интеграция генераторов

Для создания более сложных, комплексных тестов необходима интеграция генераторов друг в друга, так как взаимодействие разных устройств микропроцессора, в том числе и DSP-процессора, порождает огромное множество возможных ситуаций.

С помощью генератора случайных инструкций создаются VLIW-пакеты, которые случайным образом помещаются в тело подпрограмм, циклов, обработчиков исключений, а также вставляются между программными переходами и командами обращения в память.

Комбинирование генератора на программное управление с генератором на подсистему памяти позволяет создавать критические ситуации для микропроцессора. Например, исключения в условиях

длительной блокировки конвейера, такие как блокировки, связанные с обращениями к памяти, которые, в свою очередь, подразделяются на блокировки арбитража, вызванные конфликтами при обращении к блокам памяти, и блокировки, вызванные задержкой ответа со стороны памяти (это имеет место при промахах кэш, а также некэшируемых обращениях к внешней памяти).

Для обеспечения комбинируемости генераторов в генерируемом коде, выполняющем проверку определенной подсистемы, предусмотрены специальные секции, в которые допускается интеграция кода генераторов потока арифметических команд или тестов доступа к памяти ортогональных по используемым ресурсам к тестовой программе верхнего уровня.

VII. ЗАКЛЮЧЕНИЕ

В работе проведен анализ проблем, возникающих при автоматизации генерации псевдослучайных тестов для вычислительных ядер. Описаны пути их решения с формированием рекомендаций по оптимизации соответствующих архитектурных реализаций на примере разработки специализированных генераторов, таких как: генератор потока случайных инструкций, тестов на программное управление, тестов иерархии кэшей и тестов внешних прерываний. Созданные генераторы были применены при проверке корректности работы VLIW DSP-процессора с архитектурой Elcore50. Также предложен подход к их объединению, который позволил протестировать взаимодействие разных блоков процессора и выявить несколько критичных ошибок в работе DSP-ядра.

По истечении года при помощи описанных моделей была создана система генерации тестов для разрабатываемого процессорного ядра с VLIW-архитектурой, решающих комплексную задачу проверки всех подсистем ядра и их взаимодействия путем комбинирования уже существовавших и вновь разрабатываемых специализированных генераторов на отдельные подсистемы ядра.

ЛИТЕРАТУРА

[1] Vigyan Singhal, Starting Formal Right from Formal Test Plannin, Oski Technology, Verification Academy at DAC-2015. S. 1–10.

[2] David M. Russinoff. Formal Verification of Floating-Point RTL at AMD Using the ACL2 Theorem Prover. In Computer-Aided Reasoning: ACL2 Case Studies, chapter 13. Kluwer Academic Publishers. 2000. S. 1–8.

[3] Камкин А.С., Петроченков М.В. Система поддержки верификации реализаций протоколов когерентности с использованием формальных методов // Вопросы радиоэлектроники. 2014. Т. 5. № 2. С. 5–17.

[4] Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем СПб.: БХВ-Петербург, 2010. 560 с.

[5] Bening, Lionel, Foster, Harry D. Principles of Verifiable RTL Design: A functional coding style supporting verification processes in Verilog Hardcover, Springer – May 31, 2001 g., S. 12–17.

[6] Greene B. and McDaniel M. The Cortex-A15 Verification Story // DVClub, Austin, december 7, 2011 g., S. 1–7.

[7] Камкин А.С., Коцыняк А.М., Смолов С.А., Татарников А.Д., Чупилко М.М., Сортов А.А. Средства функциональной верификации микропроцессоров / Сб. трудов Института системного программирования РАН Т. 26. 2014 С. 149–206.

[8] Мешков А.Н., Рыжов М.П., Шмелев В.А. Развитие средств верификации микропроцессора «Эльбрус-2S» // Вопросы радиоэлектроники. 2014. Т. 4. № 3. С. 5–17.

[9] Путря Ф.М. Применение генераторов случайных программ и случайных фоновых воздействий при функциональной верификации многоядерных систем на кристалле: материалы седьмой международной конференции «Автоматизация проектирования дискретных систем», Минск, 16-17 ноября 2010 г., С. 234–241.

[10] Гаращенко А.В, Гагарина Л.Г., Федотова Е.Л., Высочкин А.В., Зайцев В.В. Разработка генератора верификационных тестов для многоядерных структур // Информатизация и связь. 2017. №4. С. 20–25.

System of Combined Specialized Test Generators for the New Generation of VLIW DSP Processors with Elcore50 Architecture

A.V. Garashchenko[1,2], A.V. Nikolaev[1], F.M. Putrya[1], S.S. Sardaryan[2]

[1]Open Joint-Stock Company Research & Development Center «ELVEES», Zelenograd, Moscow

[2]National Research University of Electronic Technology (MIET), Zelenograd, Moscow,

ant.gar1@mail.ru

***Abstract* — In connection with the architectural complexity of modern multi-core structures, more than 60% of the design resources are spent on verification during the development of the processor. Automatic generation of tests is often used to increase test coverage and reduce overall test time. Therefore, the creation of verification test generators to verify the correct operation of microprocessors is becoming increasingly important.**

This paper describes the technique of development of the several tests generators used for microprocessor verification. The first one is designed for generating VLIW of packets. The second one is for the verification of the control flow. With the help of it creates sequences of assembler instructions are created to check the pipeline. Software and hardware cycles, subprogram calls, conditional and unconditional conversions are possible. The third generator is aimed at checking the cache memory of the processor. It is based on the graph model of the memory subsystem built on its description.

In the suggested approach, the source code of the tests is constructed by using combinatorial techniques, that is all possible combinations of instructions, situations, and dependencies are sorted taking into account the constraints that direct the tests to check certain situations and also exclude the possibility of generating infinite cycles. The generated test sequences allow for various tests.

To create the more complex tests possible integration of generators into each other is considered since the interaction of different devices of the processor generates a large number of critical situations. The proposed approach makes it possible to improve the efficiency of microprocessor testing.

***Keywords* — verification of processors, wide command word, memory subsystem, test generation, coverage.**

REFERENCES

[1] Vigyan Singhal, Starting Formal Right from Formal Test Plannin, Oski Technology, Verification Academy at DAC-2015. S. 1–10.

[2] David M. Russinoff. Formal Verification of Floating-Point RTL at AMD Using the ACL2 Theorem Prover. In Computer-Aided Reasoning: ACL2 Case Studies, chapter 13. Kluwer Academic Publishers. 2000. S. 1–8.

[3] Kamkin A.S., Petrochenkov M.V. Sistema podderzhki verifikacii realizacij protokolov kogerentnosti s ispol'zovaniem formal'nyh metodov (System for supporting the verification of coherence protocol implementations using formal methods) // Voprosy radioehlektroniki. 2014. T. 5. № 2. S. 5–17.

[4] Karpov YU.G. Model Checking. Verifikaciya parallel'nyh i raspredelennyh programmnyh sistem (Verification of parallel and distributed software systems) SPb.: BHV-Peterburg, 2010. 560 s.

[5] Bening, Lionel, Foster, Harry D. Principles of Verifiable RTL Design: A functional coding style supporting verification processes in Verilog Hardcover, Springer – May 31, 2001 g., S. 12–17.

[6] Greene B. and McDaniel M. The Cortex-A15 Verification Story // DVClub, Austin, december 7, 2011 g., S. 1–7.

[7] Kamkin A.S., Kocynyak A.M., Smolov S.A., Tatarnikov A.D., CHupilko M.M., Sortov A.A. Sredstva funkcionalnoj verifikacii mikroprocessorov (Tools for Functional Verification of Microprocessors) / Sb. trudov Instituta sistemnogo programmirovaniya RAN T. 26. 2014 S. 149–206.

[8] Meshkov A.N., Ryzhov M.P., SHmelev V.A. Razvitie sredstv verifikacii mikroprocessora «Elbrus-2S» (The development of the verification tools of the "Elbrus-2s" microprocessor) // Voprosy radioehlektroniki. 2014. T. 4. № 3. S. 5–17.

[9] Putrya F.M. Primenenie generatorov sluchajnyh programm i sluchajnyh fonovyh vozdejstvij pri funkcional'noj verifikacii mnogoyadernyh sistem na kristalle (Application of generators of random programs and random background influences in the functional verification of multi-core systems on a chip): materialy sed'moj mezhdunarodnoj konferencii "Avtomatizaciya proektirovaniya diskretnyh sistem". 16–17 noyabrya 2010 g., Minsk, S. 234–241.

[10] Garashchenko A.V, Gagarina L.G., Fedotova E.L., Vysochkin A.V., Zajcev V.V. Razrabotka generatora verifikacionnyh testov dlya mnogoyadernyh struktur (Development of a verification test generator for multi-core structures) // Informatizaciya i svyaz'. 2017. №4. S. 20–25.