DOI: 10.22184/1992-4178.2021.204.3.58.65

Специализированные процессоры ASIP и способы их верификации

Т. Грёткер[1], С. Белоусов[2]

УДК 004.31 | ВАК 05.13.12

В статье рассматриваются основные преимущества разработки специализированных процессоров (Application-specific instruction-set processor – ASIP) с помощью инструментов, входящих в платформу ASIP Designer от компании Synopsys, и подробно обсуждаются подходы верификации полученного процессора: проверка функциональной модели процессора, тестирование программного обеспечения и профилирование, а также верификация RTL.

ASIP Designer [1] от компании Synopsys – это уникальная программная платформа, позволяющая автоматизировать процесс разработки, оптимизации, а также верификации ASIP, обеспечивая тем самым построение эффективных каналов передачи данных, команд управления и архитектуры памяти для реализации требуемого параллелизма на уровне инструкций, данных и задач. Два подхода – Compiler-in-the-Loop и Synthesis-in-the-Loop – позволяют получить оптимальные результаты за короткое время, снижая риски и сокращая затраты на разработку СнК. Кроме того, ASIP Designer совместим с существующими сегодня на рынке системами непрерывной интеграции и верификации.

Верификация процессора на стадии проектирования требует больших временных затрат. Однако с помощью ASIP Designer можно существенно сократить время выполнения верификации за счет автоматизации и упрощения тестового окружения, а также использования динамических тестов и формальных методов проверки устройства.

Под тестируемой моделью процессора в рамках статьи понимается полный комплект средств разработки (SDK), генерируемый средствами ASIP Designer. Рассматриваемая проверка RTL включает три этапа: проверку ядра процессора, его взаимодействия с периферийными устройствами и, наконец, интеграции процессора в систему.

ВВЕДЕНИЕ В ASIP DESIGNER

На рис. 1 представлен интерактивный процесс разработки специализированных процессоров с помощью платформы ASIP Designer.

Процесс разработки начинается с определения модели процессора, которая состоит из трех основных частей:

1. Система команд, специфика микроархитектуры, организация конвейера, регистры и память. Описание данных характеристик выполняется в виде nML-кода вместе с заголовочным файлом, в котором определяются примитивы типов данных и функций.
2. Примитивы с определенной функциональностью и временны́ми характеристиками, описанные с помощью C-кода, называемого PDG, который нужен для проведения моделирования с точностью до такта.
3. Заголовочный файл компилятора, который содержит отображение типов и операторов C/C++ из алгоритмического исходного кода в примитивы процессорного ядра.

На основе данной модели процессора генерируется SDK (рис. 1 (1)), компилятор C/C++, ассемблер и компоновщик. Помимо этого, создается высокоуровневая модель ISS, отладчик и профилировщик для оценки разработанного кода и используемых в нем алгоритмов.

Обычно процесс работы с ASIP Designer начинается с запуска примеров моделей процессоров, доступных в платформе. При этом благодаря автоматической генерации SDK можно сразу же начинать исследование архитектуры (рис. 1 (2)). Корректность работы модели процессора оценивается путем профилирования программ, реализующих ключевые для него алгоритмы.

Генерация аппаратного представления VHDL/Verilog (рис. 1 (3)) происходит на основе nML/PDG. После этого применяется подход Synthesis-in-the-Loop, который позволяет автоматически получать информацию о площади, быстродействии и потребляемой мощности будущего устройства. ISS и RTL, полученные из одного инструмента, по сути своей являются эквивалентным представлением одного и того же процессора.

[1] Консультант по бизнесу и технологиям, groetker@biztеccon.com.

[2] Компания Synopsys, ведущий инженер по применению IP-блоков, sergey.belousov@synopsys.com.

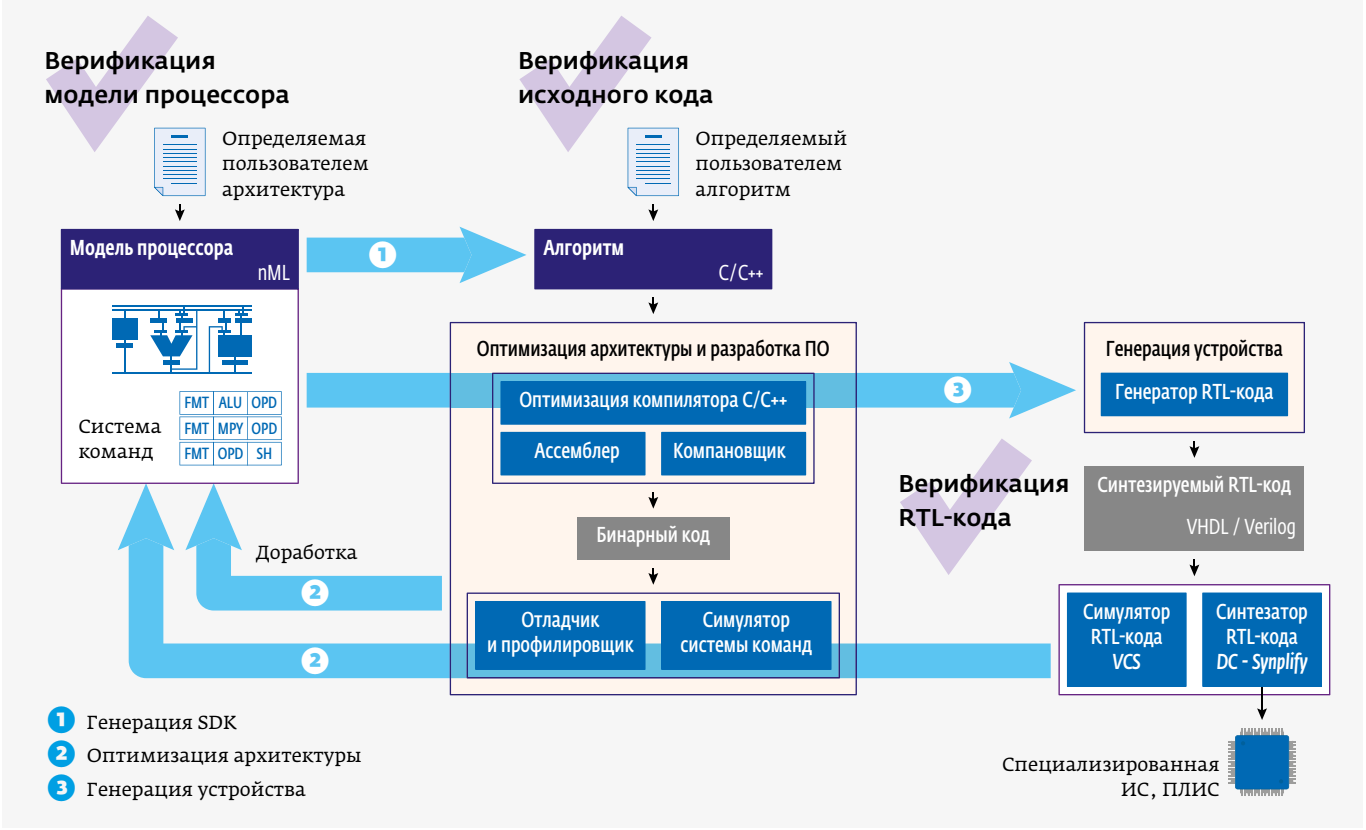

Рис. 1. Маршрут разработки в ASIP Designer

ФАЗЫ ВЕРИФИКАЦИИ

Далее мы рассмотрим три основных этапа верификации, показанные на рис. 1.

На первом этапе – *верификации модели процессора* – проверяется функциональная корректность процессорного ядра для генерации на его основе пакета SDK.

Второй этап – *верификация исходного кода* – связан с исходным кодом приложения, которое используется для оптимизации архитектуры процессора. Иными словами, архитектура процессора выбирается на основе ПО, которое будет на нем выполняться. Этот подход иначе называют Compiler-in-the-Loop.

И последним этапом является *верификация RTL-кода*. Основная задача этого типа верификации – проверить, что полученный средствами ASIP Designer код соответствует исходным требованиям спецификации. В первую очередь необходимо проверить, что RTL-код функционально соответствует модели процессора, его периферия работает без ошибок и он правильно интегрирован в систему.

Верификация модели процессора

Статическая верификация

Статические проверки процессорного ядра выполняются непосредственно на этапе генерации SDK. Они включают в себя проверки на связанность (ограничения на передачу данных между переменными различных типов), аппаратные конфликты (одновременное использование одной области памяти двумя параллельными вычислительными потоками), проблемы с конвейером, неиспользуемые инструкции и т. д.

Результатом статических проверок является отчет с диагностическими данными. На рис. 2 приведены два таких отчета. Отчет по связанности показывает, для каких областей памяти данные могут быть перемещены, а для каких нет. В отчете о проблемах с конвейером указано, какие проблемы в нем присутствуют. Компилятор C, встроенный в ASIP Designer, может устранить указанные проблемы в ПО, как только разработчик добавит соответствующие правила в nML-модели процессора.

Кроме того, автоматически генерируются однострочные тестовые программы на C для проверки полноценной поддержки этого языка. На рис. 3 продемонстрированы примеры однострочных тестов, сгенерированных средствами IDE ASIP Designer.

Динамические тесты: сравнение с эталонным исполнением инструкций

Динамические тесты играют ключевую роль в процессе верификации в платформе ASIP Designer. В пакете ASIP

Designer доступны примеры готовых процессоров с набором таких тестов.

Стандартная практика проверки подразумевает разработку на основе С базовых повторно используемых программных тестов. Эти программы должны использовать специфичные для конкретных применений типы данных и операции. Для этих целей ASIP Designer создает стандартный файл заголовка с описанными классами C++ и перегружаемыми операциями. Программы тестов, написанные на С, изначально запускаются на хостовом процессоре, например x86, и компилируются с использованием сторонних компиляторов C++, таких как g++ или VisualC++. В дальнейшем результаты таких тестов могут использоваться в качестве эталонных. Например, возможно их применение для проверки потактовой модели процессора или финальной версии RTL-кода с использованием HDL-симуляторов.

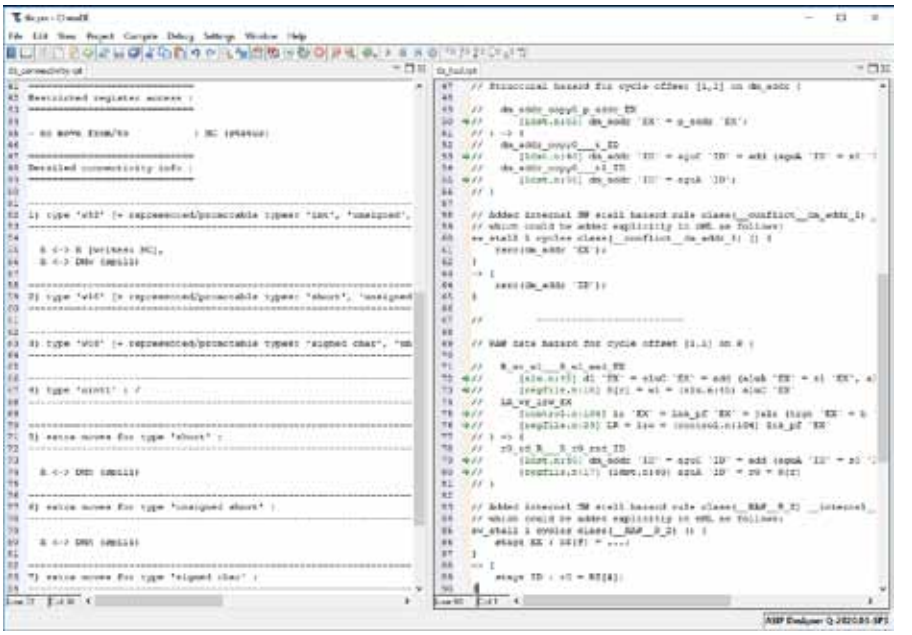

Рис. 2. Пример отчетов, сгенерированных в ASIP Designer

Основной набор динамических тестов, который рекомендуется для проверки модели процессора, включает следующее:

а) *Юнит-тесты*. Независимо от того, разрабатывается ли процессор с нуля или модифицируется существующий, любые изменения в ISA или микроархитектуре должны быть покрыты соответствующими тестами. Это включает в себя проверку работы арифметических операций, инструкций записи-чтения, переходов, режимов адресации и работы конвейера. Нужно отметить, что тесты должны включать в себя проверку функциональности в пограничных условиях. Кроме того, в случае любого изменения архитектуры новые или измененные функции должны сразу же покрываться соответствующими юнит-тестами.

б) *Программно-функциональное тестирование*. Преимуществом использования платформы ASIP Designer является ранняя и непрерывная верификация с использованием программных тестов. Возможность положиться на эффективный компилятор C/C++ обеспечивает более эффективную разработку тестового окружения в сравнении с тестами на базе низкоуровневых языков.

Тесты в целом можно использовать повторно без существенных изменений, даже если ISA или микроархитектура подвергаются изменениям: исполняемый код проверяет не только работу процессора, но и корректность работы компилятора, и самого по себе программного обеспечения.

В качестве примера в таблице представлены программные тесты, которые компания Synopsys поставляет вместе с учебной моделью ASIP Tmicro.

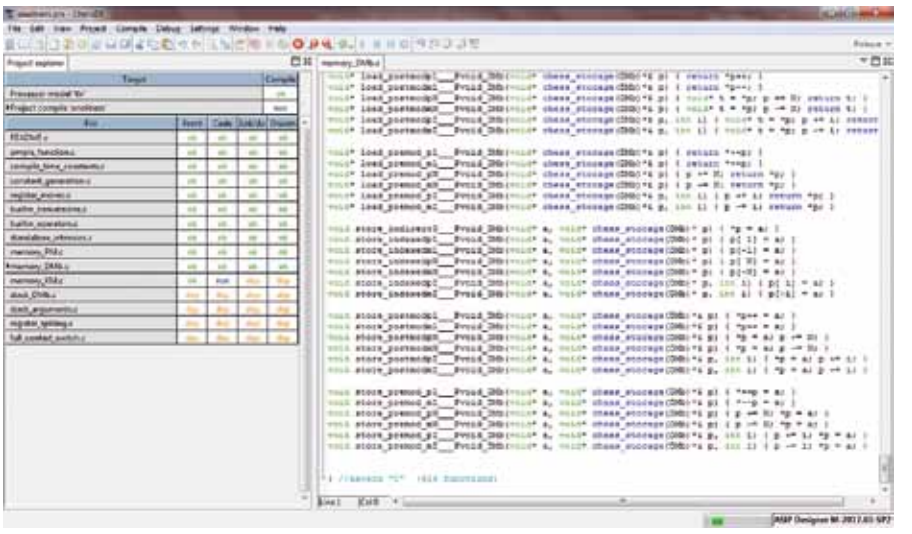

Рис. 3. Пример однострочных тестов, полученных в ASIP Designer

Программно-функциональные тесты в примере Tmicro

Низкоуровневые тесты	
A00_report	Выполнение функции report
A01_increment	Выполнение последовательного кода
A02_jumps	Операции перехода
A03_call	Операции вызова и возврата
A04_control	Выполнение операций управления
A06_loop	Аппаратные циклы
A07_jump_loop	Операция перехода в конце аппаратного цикла
A08_call_loop	Операция вызова в конце аппаратного цикла
A21_jump_at_high_pc	Безусловный переход при большом значении PC
A22_cjump_at_high_pc	Условный переход при большом значении PC
A23_call_at_high_pc	Операция вызова при большом значении PC
A26_loop_at_high_pc	Аппаратный цикл при большом значении PC
A30_reserved	Чтение-запись регистров
Базовые тесты C	
C00_simple_report	Выполнение функции report
C01_simple_call	Выполнение вызова функции
C02_simple_if	Операции условного перехода
C03_simple_loop	Простой цикл с постоянным или переменным количеством итераций
C10_int_expr	Операции над типом short int
C11_dint_expr	Операции над типом long int
C12_ptr_arith	Арифметические операции с указателями
C13_logical_expr	Логические операции
C14_short_div	Деление типа short int
C15_long_div	Деление типа long int
C16_dint_manifest	Умножение типа long int с явными операндами
C20_conversions	Конверсионные операции
C21_ref_cast	Референсные преобразования типов
C30_function_ptr	Неявный вызов функции
C31_loop_call	Аппаратные циклы только с функцией вызова
C32_many_arguments	Передача аргументов функции через регистры или стек
C33_stack_frame	Фрейм стека: область построения, область аргументов и т. д.
C34_manifest_return	Явный вызов return при сложном условии if
C35_factorial	Функция факториал (разовая рекурсия)
C36_inline_static	Статические переменные
C40_collapse_if	Свертка последовательных if
C41_big_if	Функция перехода с типом long (> 128 и >256)
C42_goto_trafos	Goto-трансформация
C43_switch_trafos	Switch-трансформация
C50_doloop	Цикл do
C51_nested_loop	Вложенные циклы
C52_loop_cond	Циклы со сложными условиями
C53_do_with_if	Цикл do с условным оператором
C54_loop_count	Цикл со счетчиком и эффектом по его значению
C55_loop_trafos	Трансформация циклов
C56_unroll	Директивы chess_unroll и chess_flatten
C60_aggregate	Инициализация массивов через aggregate
C61_union	Объединения
C62_struct	Структуры
C63_softw_bitfield	Битовые поля
C70_algebraic_optim	Алгебраическая оптимизация
C71_induction	Индукционный анализ и построение цепочек
C72_bool_opt	Булева оптимизация
Библиотечные тесты	
L01_ctype	Проверка функций ctype.h
L02_stdarg	Проверка функций stdarg.h
L03_string	Проверка функций string.h
L04_stdlib	Проверка функций stdlib.h
Тесты по работе с памятью	
M02_pm_access	Проверка операций load / store
Программы и тестовые сценарии	
P00_eratosthenes	Сито Эратосфена (вычисление простых чисел)
P01_heap_sort	Сортировка в куче
P02_matrix_mult	Умножение матриц
P03_dhrystone	Тест DhryStone, версия C / 1.1, 12 / 01 / 84
P04_iir	БИХ-фильтр
P05_qsort	Полиморфная сортировка массива
P06_fibonacci	Функция Фибоначчи (глубокая / многократная рекурсия)
P07_ackermann	Функция Аккермана (глубокая / непримитивная рекурсия)
P08_compander	Программа с большим количеством switch
Специализированные тесты ASIP	
S00_lmul	Внутренний LMUL

в) *Тестирование компилятора*. Тестирование компилятора по сути представляет собой проверку целостности и корректности nML-модели и заголовочного файла. Существует четыре способа проверки компилятора:
   1. Процессорно-специфичные однострочные программы проверяют, что все ресурсы для компиляции C доступны.
   2. Юнит-тесты, полученные для заголовочного файла, проверяют отображение и оптимизацию конструкций языка C / C++.
   3. Одинаковые результаты, полученные при исполнении тестов на хостовом процессоре (например, x86) и на программной модели ASIP доказывают эквивалентность GCC или иного применяемого компилятора и специализированного компилятора ASIP.
   4. Непрерывное увеличение числа специфических тестов обеспечивает покрытие ими всё большей функциональности.

г) *Тестирование прерываний*. Необходимо, чтобы модель процессора обрабатывала прерывания корректно. Помимо соответствующих юнит-тестов, рекомендуется также перезапускать обычные программные тесты с включенными прерываниями. Время их срабатывания необходимо в данном случае указывать случайным или псевдослучайным. Самый простой способ сделать это – добавить генератор импульсов в PDG-код модели процессора, как показано в примерах применения Tmicro [2, 3].

Верификация исходного кода

При оптимизации архитектуры с помощью ASIP Designer обычно используют подход Compiler-in-the-Loop, который подразумевает компиляцию и профилирование ядер приложений, описанных на C / C++. Поскольку архитектурные изменения кодируются в nML / PDG-модели процессора, они обычно идут рука об руку с модификацией исходного кода приложения. Чтобы обеспечить целостность проекта, каждое изменение кода должно быть проверено как можно раньше в среде непрерывной интеграции (CI), то есть каждое изменение в исходном коде сначала исполняется на хостовом процессоре и в ISS, после чего результаты этих двух проверок сравниваются. Это мы и определяем как верификацию исходного кода.

Рассмотрим пример трансформации исходного кода: введение специфичных для приложения типов данных и операций, векторизация, трансформация циклов для оптимизации использования памяти, преобразование потока управления, чтобы предоставить компилятору больше параллелизма. На рис. 4 представлено, как такая трансформация исходного кода помогает улучшить быстродействие при разработке архитектуры средствами ASIP Designer.

По оси абсцисс на рис. 4 расположены даты итеративного изменения кода, по оси ординат – количество тактов на выполнение задачи.

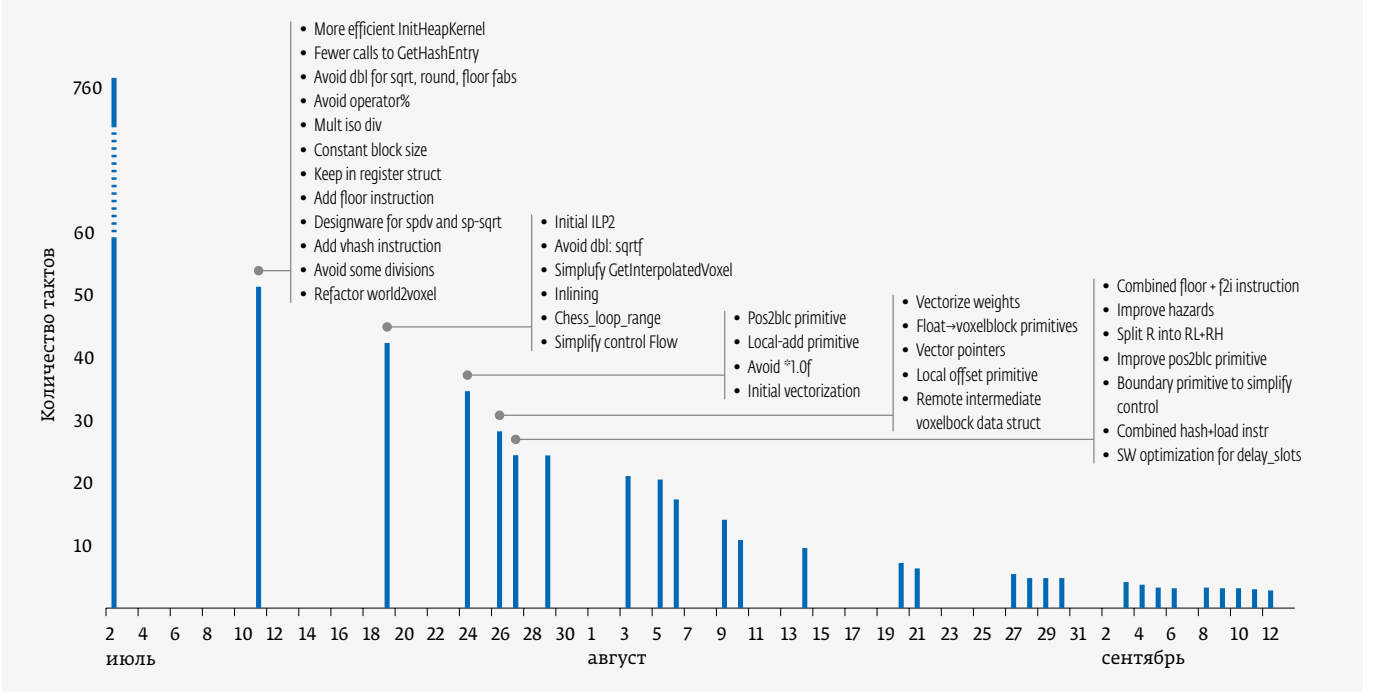

Рис. 4. Трансформация исходного кода C в задаче распознавания образов

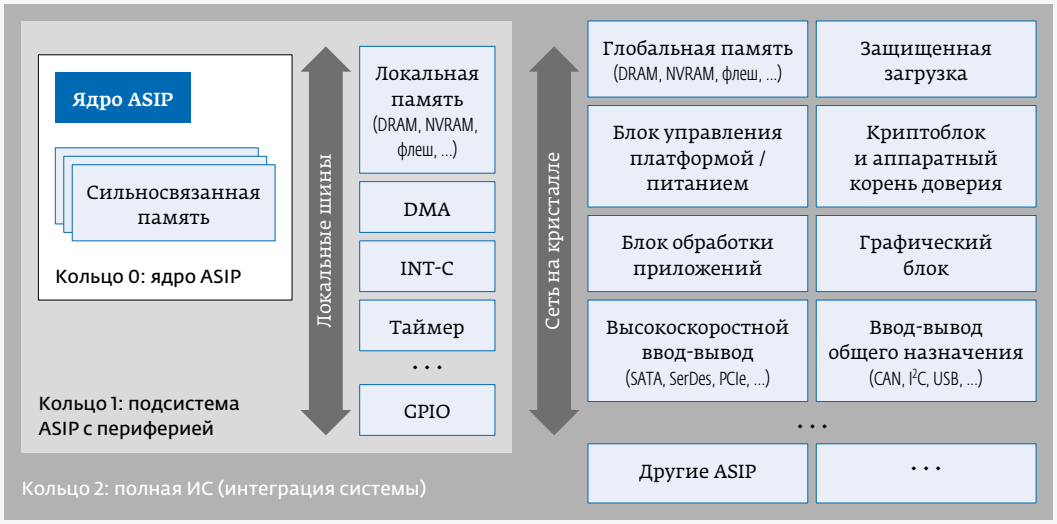

Рис. 5. Различные уровни абстракции при верификации (кольца верификации)

Верификация RTL-кода

Основные инструменты

Для обеспечения непрерывного маршрута, построенного на базе SystemVerilog и совместимого с различными компонентами UVM, рекомендуются четыре основные технологии:

1. Повторное использование как можно большего количества программно-функциональных тестов.
2. Генерация случайных последовательностей инструкций с помощью инструмента Risk [4].
3. Конфигурируемый SystemVerilog-код, включающий в себя тестовое окружение, классы для последующей генерации случайных последовательностей импульсов и групп покрытия для ISA.
4. Интерфейсы между ASIP Designer и виртуальными/аппаратными системами прототипирования.

RTL-верификация

Выше мы говорили о росте количества программных тестов от инструкций, алгоритмических ядер и до конечного приложения. Точно также в RTL-верификации в первую очередь нужно начать проверку с процессорного ядра. Далее увеличивается периметр проверки и добавляется различная периферия. В процессе такого масштабирования мы приходим к законченной системе, в которой присутствует разработанный средствами ASIP Designer процессор.

Такое разделение устройства на подблоки для проверки называется кольцами верификации (рис. 5).

Процессорное ядро

В первую очередь мы должны обеспечить корректность имплементации процессорного ISA, и нашей ключевой метрикой будут степень покрытия инструкций сгенерированными SystemVerilog-мониторами плюс построчное покрытие RTL-кода. Также нужно проверить эквивалентность ISS и RTL с точки зрения функциональности и временны́х характеристик.

Платформа для такой проверки автоматически генерируется инструментом ASIP Designer RTL generator [5]. Она включает в себя генераторы синхросигналов, сигналов сброса, а также модели памяти для программ и данных. Стандартные тестовые программы могут быть запущены на такой платформе «из коробки» без дополнительной их модификации. Кроме того, процесс сравнения на эквивалентность ISS и RTL в ASIP Designer выполняется автоматически.

Разработка корректных, значимых тестовых программ, которые бы обеспечивали хорошую степень покрытия, является задачей, требующей для большинства процессоров больших временных затрат. В ASIP Designer данная проблема решается с помощью специального инструмента Risk, представляющего собой генератор псевдослучайных низкоуровневых тестовых программ (рис. 6).

Примеры, доступные в инструменте Risk, содержат в себе арифметические проверки, проверки перемещения данных, а также различные варианты проверок управляющих операций [2].

Необходимо разделять два типа собираемых метрик покрытия: те, которые отвечают за сравнение RTL и ISS, и те, которые отвечают на вопрос: «Действительно ли процессор выполняет требуемые операции?». Тестовые программы, которые генерирует Risk, в первую очередь предоставляют первую метрику. Если процессорная модель была дополнена утверждениями SystemVerilog, то тесты, полученные из Risk, могут также использоваться для проверки функциональности и быстродействия процессора или периферии.

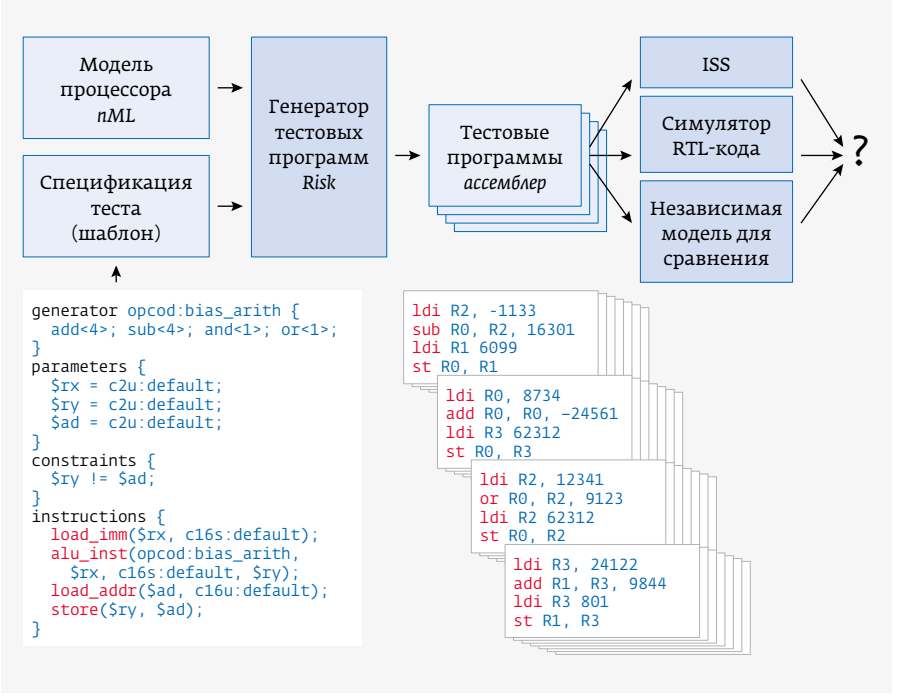

Рис. 6. Генерация случайных инструкций средствами инструмента Risk

```
isa_coverpoints: alu_instr load_store_instr control_instr div_instr;
isa_cross_coverpoint: reg_MC; // => cross coverage (instr x MC status register)
isa_cross_coverpoint_bins: “00xxxxxx” “01xxxxxx” “10xxxxxx” “11xxxxxx”;
```

Рис. 7. Генерация SystemVerilog-кода для перекрестного покрытия

Создание SystemVerilog-мониторов может конфигурироваться под конкретные нужды проекта. На рис. 7 показан пример, показывающий, как ASIP Designer генерирует SystemVerilog-код для перекрестного покрытия тестами различных типов инструкций и двух крайних левых битов регистра MC.

Периферия

В ASIP Designer присутствует механизм для автоматической генерации случайной последовательности инструкций, представляющих собой специальные SystemVerilog-классы. Они же могут использоваться для проверки периферийных блоков.

Со смещением фокуса верификации с ядра, полученного средствами ASIP Designer, в сторону процессорной подсистемы необходимы дополнительные генераторы сигналов и блоки проверки. Обычно это подразумевает использование сторонних верификационных блоков (VIP) для проверки блоков памяти и коммуникационных каналов.

На рис. 8 показан SystemVerilog-класс, полученный для очень простой модели процессора. Этот класс представляет собой высокоуровневое представление ISA, содержащее поле enum, необходимое для инструкции. Методы get_image() и get_syntax() обеспечивают доступ к инструкциям и их синтаксическим представлениям.

Интеграция системы

Когда речь идет о проверке интеграции на уровне системы, необходимо учитывать также условия загрузки, поскольку различные процессоры могут считывать программный код ASIP из флеш-памяти, проверять его, декомпозировать перед тем, как наше ASIP-ядро начнет работать. Часто также следует учитывать DMA-контроллеры, которые переносят данные из памяти DDR в SRAM-память системы. Помимо этого, в системе могут присутствовать блоки управления питанием (PMU), контроллеры сенсоров, с которыми необходимо выполнять итерации. На этом этапе сложность устройства многократно возрастает, что влечет за собой также усложнение тестового окружения. Это в свою очередь оказывает негативное влияние на быстродействие средств моделирования RTL-кода.

```
// Encoded instruction
typedef struct {
    logic[0:13] bits;
    int width; // #bits used
} image_type;

// Example of ALU instruction with
// op code and two operand registers
class core$alu_opn;
  rand core$alu_op P$op;  // operation
  rand core$eR P$a; // operand reg A
  rand core$eR P$b; // operand reg B
  ...
endclass

class core$core; // Top-level “OR rule”
  rand enum { // Enum used to select sub-rule
    S$alu_opn,          // Simple example:
    S$compare_opn,      // flat hierarchy,
    S$load_sp_indexed, // sub-rules refer
    ...                 // to instructions.
  } sub;
  rand core$alu_opn R$alu_opn;
  rand core$compare_opn R$compare_opn;
  rand core$load_sp_indexed R$load_sp_indexed;
  ...
  function image_type get_image();
  ... // Extract final instruction
  function string get_syntax();
  ... // Assembly syntax (log/debug)
endclass
```

Рис. 8. Сгенерированные ASIP Designer классы

Быстродействующие системы, такие как аппаратные эмуляторы [6], системы прототипирования, построенные на базе FPGA [7], или виртуальные прототипы позволяют проводить верификацию на уровне системы. ASIP Designer поддерживает все три метода, которые указаны выше, за счет поддержки аппаратных платформ ZeBu, HAPS и системы виртуального прототипирования Virtualizer. ASIP Designer автоматически генерирует необходимые скрипты, программный код и окружение для них. Прототипы от сторонних производителей поддерживаются также по умолчанию, поскольку код, полученный из ASIP Designer, полностью соответствует стандартам HDL, SystemC и поддерживает интерфейсы JTAG.

ЗАКЛЮЧЕНИЕ

Системы, построенные на базе ASIP Designer с мощным SDK, совмещают в себе программируемость на C/C++ с мощностью и производительностью специализированного оборудования. Продуктовые линейки, созданные на основе ASIP, обычно обладают гибкостью, которая позволяет удовлетворить требования различных сегментов рынка, используя одни и те же продукты. Кроме того, они хорошо себя зарекомендовали в проектах, где используется программно-функциональная верификация, а также в случае, когда необходимо внесение различных модификаций с минимально возможными задержками реализации конечного продукта.

ЛИТЕРАТУРА

1. ASIP Designer: Design Tool for Application-Specific Instruction-Set Processors Datasheet
2. ASIP Designer: Verification of the Tmicro Core
3. ASIP Designer: Implementing Interrupts on the Tmicro Core
4. ASIP Designer: Risk manual
5. ASIP Designer: Go User Manual
6. https://www.synopsys.com/verification/emulation.html
7. https://www.synopsys.com/verification/prototyping/haps.html
8. **Willems M., Cox S.** Softening Hardware: Using Application-Specific Processors to Optimize Modern SoC Designs // Synopsys white paper.
9. ASIP Designer: Chess Compiler Processor Modeling Manual
10. ASIP Designer: Chess Compiler User Manual
11. ASIP Designer: Example Processor Models
12. Designing Application-Specific Processors for Smart Vision Systems: A SLAM Case Study // Synopsys webinar, April 2020.
13. https://www.synopsys.com//verification/virtual-prototyping.html
14. ASIP Designer: Checkers ISS Interface Manual