Разработка программно-аппаратных систем на основе описания макроархитектуры

Д.Ю.Булычев
db@tepkom.ru
Санкт-Петербургский государственный университет
198504, Университетский пр., 28
Санкт-Петербург, Россия

Аннотация

Несмотря на впечатляющий прогресс в развитии средств разработки программно-аппаратных систем, решение некоторых задач в этой области все еще не поддержано соответствующими инструментами. Одна из наиболее важных проблем — быстрая настройка компилятора и оптимизатора на новую аппаратную платформу — также остается открытой. В статье описывается подход к разработке программно-аппаратных систем, основанный на понятии *макроархитектуры* целевого процессора, и показывается, как данная проблема может быть решена путем одновременного синтеза системы команд и машинного кода для целевого приложения. Кроме того, описывается иерархия задач, возникающих при таком подходе, и возможные способы их решения.

Введение

Основной задачей разработки программно-аппаратных систем является построение "подходящей" реализации некоторого набора алгоритмов. Использованный здесь неформальный термин "подходящий" объединяет много факторов, обычно принимаемых во внимание, например, производительность, энергопотребление, стоимость, вес и т.д. Мы ограничимся случаем, когда нужный набор алгоритмов представлен в виде программ на языке высокого уровня, поскольку этот способ описания является наиболее практичным среди всех возможных.

Одним из способов построения систем обсуждаемого типа является разработка специфических для данного приложения процессоров (ASIP — Application-Specific Instruction Set Processor) [22]. Несмотря на то что современные технологии разработки процессоров позволяют реализовать на уровне оборудования практически произвольно сложные алгоритмы, программируемые процессоры, реализующие сравнительно простые системы команд, продолжают широко использоваться.

Существуют две основные причины этого. Во-первых, программируемость позволяет достичь гибкости. Система, построенная на программируемом процессоре (интерпретаторе) дает возможность исполнять потенциально неограниченный набор программ. Во-вторых, система команд обеспечивает возможность переиспользовать мощности процессора. Поскольку гибкость и универсальность в интересующей нас области отходят на второй план (в силу ограниченности множества программ, исполняемых на встроенном устройстве), остановимся подробнее на переиспользовании.

Пусть нас интересует реализация какого-либо алгоритма в виде непрограммируемого устройства, обладающего таким же поведением, как и некоторая программа. Простейшим путем его реализации является синтез из поведенческой спецификации на каком-либо языке описания оборудования (например VHDL). При прямолинейном написании такой спецификации синтезированное устройство практически всегда окажется сильно избыточным. Например, разные вхождения одного и того же выражения в описание поведенческой модели приведут к синтезу разных устройств, его реализующих. Во избежание этого придется разбить исходную спецификацию на похожие фрагменты и реализовать соответствующую диспетчеризацию между ними — т.е. фактически, разработать систему команд и написать в ее терминах программу[1].

Подобная факторизация на уровне микроархитектуры (т.е. аппаратной реализации) в свою очередь ведет к сложной проблеме — настройке компилятора и оптимизатора. Для ее решения

[1]Современные методы порождения высокоэффективного устройства по поведенческому описанию включают в себя стадию оптимизации, которая может существенно снизить избыточность порождения. Однако глубина и состав этих оптимизаций не позволяют рассматривать их как средство выделения системы команд.

необходимо либо поддерживать соответствие между программой и факторизуемой микроархитектурой, либо научиться автоматически извлекать кодогенератор из поведенческого описания процессора. Обе эти задачи рассматриваются в современной специальной литературе.

В данной статье мы опишем технологию разработки встроенных систем, основанную на понятии *макроархитектуры* устройства — его описании на промежуточном уровне, позволяющем отделить задачу выбора подходящей системы команд или настройки компилятора от оптимизации собственно устройства на нижнем уровне.

1 Общие подходы к разработке программно-аппаратных систем

Несмотря на большое разнообразие конкретных подходов и инженерных решений в данной области, все они так или иначе сходятся в своих основных чертах. Мы будем далее опираться на подход, описанный в [5], поскольку исторически именно он является предшественником наших текущих исследований.

В основе этого подхода лежит представление реализации встроенной системы в виде декомпозиции на параметризованную базовую архитектуру и набор выделенных аппаратных объектов, реализующих критические с точки зрения производительности части вычислений. Выделение аппаратных объектов предполагается производить на основе анализа программной реализации системы (профилирования). Необходимая производительность системы достигается путем оптимизации на многих уровнях: оптимизации собственно машинного кода, параметризации базовой архитектуры, оптимизации выделенных аппаратных блоков и интерфейсов между ними и т.д.

При разработке данного подхода основной упор был сделан на исследование способа выделения аппаратных объектов; что же касается программной части системы, то предполагалось, что выбор базовой архитектуры и настройка компилятора яв-

ляются технической задачей в силу наличия большого количества подходов для ее решения. Однако в дальнейшем оказалось, что на этом пути стоят препятствия как технического, так и принципиального характера.

Прежде всего, изучение существующих подходов к настройке компиляторов на другую аппаратную платформу выявило их незрелость применительно к исследуемой области [6]. Именно, все эти подходы обеспечивали настраиваемость только при условии значительного объема ручных изменений и, следовательно, сравнительно большого времени настройки. Все они, кроме того, требовали описания системы команд процессора на специальном формализме, непригодном ни для чего иного, кроме порождения кодогенератора. Изучение возможности порождения такого описания из низкоуровневого описания устройства [16] показало заметное падение качества кодогенератора.

С другой стороны, оказалось, что подход, основанный на предварительном выборе базовой архитектуры с последующей настройкой компилятора, страдает от любопытного принципиального недостатка: несмотря на то что базовая архитектура выбирается с тем, чтобы целевое приложение компилировалось в нее наиболее естественным образом, сложность задачи генерации кода при этом не уменьшается, а возрастает. Иными словами, сперва должны быть затрачены некоторые усилия, чтобы определить основные черты базовой архитектуры, а затем — чтобы настроить на нее компилятор, при этом последняя задача может оказаться весьма сложной в силу того, что прагматические соображения, принятые во внимание при выборе архитектуры, недоступны компилятору. Наконец, приведем еще одно принципиальное соображение: в то время как свойства базовой архитектуры определяются исходя из “интересов” конкретного целевого приложения, компилятор для этой архитектуры должен по существу порождать эффективный код для *любой* программы. Задача настройки компилятора, таким образом, является обобщением исходной задачи, причем обобщением “сверх необходимого”.

Итак, оказалось, что дальнейшее развитие технологии раз-

работки встроенных систем требует существенного пересмотра той ее части, которая основывалась на понятии настройки компилятора и выбора целевой аппаратной платформы. Далее будет изложен результат этого пересмотра.

2 Прототипирование и оптимизация

Под *прототипом* встроенной системы мы будем понимать совокупность исполняемого машинного кода и модели вычислительного устройства при условии, что данная модель исполняет этот код корректно, но, возможно, неэффективно. Кроме того, мы будем требовать, чтобы переход от модели устройства к его аппаратной реализации осуществлялся автоматически. Таким образом, прототип превращается в готовую систему тогда, когда его эффективность удовлетворяет заданным характеристикам задачи.

Неформально говоря, понятие прототипа отделяет стадию разработки встроенной системы от стадии ее оптимизации. В зависимости от обстоятельств в разработку могут входить такие этапы, как первоначальный анализ задачи, проектирование, моделирование (т.е. разработка модели в каком-либо формализме и изучение ее поведения), разбиение (отделение частей, реализуемых аппаратно, от программной части системы и выбор аппаратной платформы для ее исполнения), реализация (написание кода программной части и разработку специализированных аппаратных блоков) и, наконец, сборка — получение исполняемого кода для выбранной аппаратной платформы. Не все эти этапы могут присутствовать в процессе разработки конкретной системы, равно как и необязательно именно в таком порядке (например, разработка может начаться с написания реализации на языке высокого уровня, а дальнейшая работа будет проводиться именно с этим кодом). Тем не менее в общем случае таков набор задач, которые должны быть решены для достижения результата.

Основным отличием разработки от оптимизации является то, что при оптимизации уже существует некоторый функционально полный и корректно работающий вариант системы — эталонная реализация. В идеальном случае требуемая эффективность достигается далее набором локальных и поддержанных

средствами разработки изменений, хотя не исключен и вариант, когда свойства прототипа настолько плохи, что приходится вернуться к предыдущим этапам разработки (это означает, в частности, что на них были допущены ошибки). Формально говоря, эталонной реализацией можно считать и работающий код на инструментальной машине, однако, поскольку автоматический синтез целевой платформы из описания инструментальной обычно невозможен, мы не считаем реализацию на инструментальной машине прототипом.

Мы рассмотрим способ автоматизированного получения прототипа, основанный на извлечении макроархитектуры аппаратной платформы из реализации на языке высокого уровня. Такой подход позволяет исключить стадию настройки средств разработки на целевую платформу, поскольку система команд и код реализации для нее порождаются одновременно. Кроме того, помимо собственно прототипирования мы рассмотрим задачи оптимизации, решение которых может быть автоматизировано в рамках данного подхода.

3 Макро- и микроархитектура устройства

Под *макроархитектурой* мы понимаем архитектуру процессора с точки зрения компилятора с языка высокого уровня. Соответственно, *микроархитектура* есть реализация макроархитектуры на языке описания оборудования. Здесь уместна аналогия между данными уровнями спецификации устройства и реализацией языков программирования: в то время как макроархитектура аналогична языку, микроархитектура соответствует его интерпретатору. В такой ситуации язык описания оборудования есть язык реализации этого интерпретатора.

Граница между макро- и микроархитектурой, следовательно, может перемещаться между двумя крайними случаями, одним из которых является ситуация, когда целевая программа написана на языке описания оборудования (и, следовательно, макроархитектура совпадает с микроархитектурой), а другим — когда процессор является интерпретатором исходного языка (например аппаратная Java-машина). Поэтому в каждом конкретном случае необходимо заранее договориться, где именно проходит эта граница. Как было сказано выше, мы считаем, что исход-

ная программа реализована на императивном языке высокого уровня (например C). В такой ситуации на уровень макроархитектуры устройства попадают следующие элементы:

- набор программно доступных ресурсов (регистров, памятей) и их свойства (разрядность, количество экземпляров, принцип агрегирования и т.д.);
- режимы адресации;
- набор команд, их семантика, правила порождения двоичного кода и ассемблерного представления.

Этот выбор обусловлен тем, что только вышеперечисленные свойства существенны для генерации кода с языка высокого уровня. Более тонкие детали реализации процессора (например конвейеризация и параллелизм на уровне инструкций) не только не оказывают существенного влияния на порождение кода[2], но вообще могут быть еще не известны на момент кодогенерации в силу того, что финальные характеристики устройства пока не ясны.

Излагаемая точка зрения во многом противоречит общепринятой, в соответствии с которой поведенческое описание *системы команд* рассматривается как загрубление поведенческого описания *устройства в целом*, при этом считается, что чем более полно описано поведение устройства, тем лучше. Описание деталей поведения устройства обычно оправдывается тем, что это в некоторых случаях позволяет получить более эффективный код с языка высокого уровня благодаря планированию и использованию суперскалярных возможностей. Однако общеизвестно [17], что описание этих возможностей допускает существенно более простую параметризацию, чем параметризация, необходимая для настройки кодогенератора. Наконец, очевидно, что одна и та же макроархитектура может в качестве своих реализаций иметь несколько совершенно различных микроархитектур (например при аппаратном реинжиниринге). Все это приводит нас к мысли, что описание на уровне системы команд имеет смысл тогда, когда предполагается программирование в

[2]Точнее, существует простой конструктивный критерий качества выбора инструкций и распределения регистров с точки зрения планирования: порожденный код не должен содержать антизависимостей [17].

терминах данной системы команд (либо вручную на ассемблере, либо с использованием компилятора). Только те детали поведения устройства, которые существенны с такой точки зрения, должны попасть на уровень макроархитектуры. Это, в частности, означает, что описание макроархитектуры, вообще говоря, зависит от входного языка высокого уровня — если в нем, например, присутствуют средства явного управления параллелизмом, то они должны получить адекватное отражение на уровне макроархитектуры, иначе невозможно гарантировать семантическую корректность.

4 Технология разработки встроенных систем

Схема получения прототипа системы приведена на рисунке 1. Рассмотрим основные этапы предлагаемого подхода более подробно.

Прежде всего, по тексту реализации системы на языке высокого уровня порождается описание макроархитектуры процессора и ассемблерный код (рис. 1, *A*). Этот процесс является критическим для описываемого подхода, поскольку порождение "неудачной" архитектуры или неэффективного кода является недопустимым. Вместе с тем, существуют весомые аргументы в пользу того, что данная задача может быть решена более успешно, чем задача автоматической настройки компилятора на целевую платформу.

Действительно, основной сложностью при настройке компилятора на платформу для встроенного приложения является использование расширенных возможностей этой платформы — нестандартных ресурсов, режимов адресации и команд. Эти возможности вводятся разработчиками оборудования с целью достижения большей производительности. Но поскольку окончательный набор программ, работающих на данной платформе, ее разработчику недоступен, правильное их использование в каждом конкретном случае может превращаться в нетривиальную задачу. С другой стороны, даже небольшой анализ текста реализации позволяет определить, какие конкретно расширенные возможности целевой платформы необходимы для синтеза эффективного кода. Подобного рода анализ, проводимый с целью выбора целевого процессора среди множества уже готовых, яв-

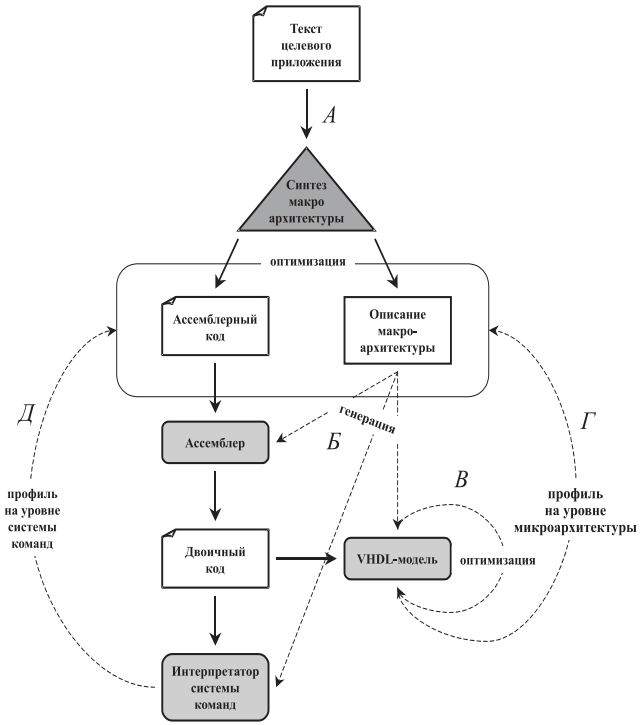

Рис. 1: Схема технологии прототипирования

ляется обычным делом. Таким образом, сложность задачи настройки компилятора происходит не из-за сложности приложения, а из-за несогласованности расширенных возможностей универсальной платформы с его потребностями.

В современной литературе рассмотрено несколько способов получения системы команд, приспособленных для эффективного исполнения данного приложения [8, 12, 14, 21], однако все они в той или иной степени двигаются в направлении, противоположном описанному нами: система команд порождается путем группировки микроинструкций для данной микроархитектуры. Таким образом, задача компиляции не исключается целиком из процесса разработки, поскольку исходный микрокод должен быть как-то сгенерирован. В силу этого автором разработан новый алгоритм, позволяющий осуществить синтез си-

стемы команд вместе с ассемблерным кодом непосредственно по абстрактному синтаксическому дереву программы. Изложение алгоритма выходит за рамки данной статьи; некоторые подробности можно почерпнуть в [1].

Следующим этапом разработки является автоматическое получение ассемблера, дизассемблера и интерпретатора системы команд (рис. 1, *Б*). Формализм описания макроархитектуры должен позволять проводить синтез этих средств без привлечения дополнительных усилий.

Ассемблер получает на вход сгенерированный на предыдущем этапе ассемблерный код и порождает объектный модуль для данной архитектуры. Дизассемблер производит обратное преобразование; кроме того, модуль дизассемблера используется в интерпретаторе системы команд.

Наконец, из описания макроархитектуры автоматически порождается спецификация микроархитектуры на одном из языков описания оборудования (например VHDL).

После порождения всех этих средств прототип составляется из загрузочного модуля, полученного из порожденного ассемблерного текста с помощью ассемблера, и модели микроархитектуры. Видно, что процесс получения прототипа является полностью автоматическим. Вместе с тем, технология должна предоставлять возможности его настройки в двух ключевых местах: при синтезе макроархитектуры из исходного кода и при синтезе микроархитектуры из макроархитектуры.

После порождения прототипа возможны две ситауции: если полученная система удовлетворяет критериям эффективности, то цель достигнута. В противном случае прототип должен подвергнуться оптимизации. Ниже мы рассмотрим возможные способы оптимизации, находящиеся в рамках данной технологии.

Все способы оптимизации прототипа разумно разделить на две группы.

В первую группу попадают преобразования, ортогональные по отношению к системе команд:

- Оптимизация кодирования, т.е. снижение накладных расходов на хранение и декодирование программы (рис. 1, *Г*). Данное преобразование можно проводить по *статическому* или *динамическому* профилям. В первом случае имеется в виду статическое распределение инструкций в порожденной

программе, во втором — динамические частоты их исполнения на наборе тестовых данных. В первом случае достигается минимизация размера кода, во втором — минимизация времени работы. Кроме того, возможна смешанная оптимизация по двум профилям.

- Оптимизация микроархитектуры (рис. 1, *В*), т.е. преобразование реализации устройства, которое не меняет систему команд (например введение дополнительного кэша и т.д.).

- Планирование команд (*instruction scheduling*) — преобразование ассемблерной программы с целью более эффективного использования внутреннего параллелизма процессора. Заметим, что такой параллелизм появляется только при генерации микроархитектуры, поэтому хорошо спланированная программа не может появиться без принятия во внимание ее свойств. Построение хорошего планировщика по информации, извлекаемой в момент генерации микроархитектуры, является открытой задачей.

Во вторую группу попадают преобразования собственно системы команд (рис. 1, *Д*). Заметим, что при ее изменении должен согласованно изменяться и ассемблерный код реализации, поскольку в рамках данной технологии компилятор отсутствует как таковой. Ассортимент преобразований системы команд представляется весьма широким; ниже мы перечислим те из них, которые кажутся наиболее полезными.

- Разбиение команды по значению операнда. Например, команда сложения с непосредственным операндом (константой) может быть разбита на две команды: сложение с единицей и сложение с константой, отличной от единицы.

- Слияние команд по значению операнда — преобразование, обратное предыдущему.

- Разбиение команды по семантике. Например, команда сложения и перехода по нулю может быть разбита на команду сложения и команду перехода по нулю.

- Слияние команд по семантике — преобразование, обратное предыдущему.

- Удаление неиспользуемой команды.

В настоящее время изучение перечня этих преобразований и способов их реализации является темой дальнейших исследований. Теоретически представляется возможным перенести на данный этап процедуру выделения аппаратных блоков, поскольку каждый такой блок можно рассматривать как выделенную инструкцию со сложным поведением.

5 Существующие разработки

В интересующей нас области существует большое количество подходов, призванных автоматизировать задачу разработки целевой системы. Ниже мы рассмотрим наиболее полно освещенные в публикациях результаты.

Язык описания устройства MIMOLA [15] и построенная на его основе технология являются, видимо, исторически первым успешным опытом в данной области. MIMOLA предназначена для структурного описания устройства. В результате этого настройка компилятора и порождение аппаратной реализации происходят на основе одной и той же спецификации. С другой стороны, такой подход требует решения задачи выделения системы команд из структурного описания, что, во-первых, не всегда возможно, а, во-вторых, потенциально ведет к низкому качеству кода.

CHESS/Checkers [13] — технология, базирующаяся на языке nML [7], который предназначен для поведенческого описания устройства. В состав технологии входят настраиваемый компилятор CBC и настраиваемый симулятор Checkers. Основные усилия при исследовании этого подхода были направлены на получение эффективного компилятора для данной архитектуры, вопросы же разработки самой архитектуры для заданного множества программ не рассматриваются. Эта технология является одной из немногих, доведенных до состояния коммерческого продукта; с другой стороны, ее использование налагает ограничения на характеристики целевого устройства (например, не поддерживается работа с многотактовыми командами).

Аналогичной разработкой является FlexWare [18], в состав которой входит настраиваемый компилятор CODESYN и настраи-

ваемый симулятор INSULIN. Особенностью данного подхода является то, что для настройки компилятора и симулятора используются разные формализмы.

Язык EXPRESSION [10] также разработан для задачи автоматической генерации компилятора и симулятора. Его особенность заключается в использовании смешанного структурно-поведенческого способа описания целевой архитектуры. Отдельные усилия были приложены для обеспечения возможности спецификации различных способов организации памяти.

Порождение точных симуляторов для сложных конвейерных архитектур рассматривалось в качестве основной задачи при создании языка описания архитектуры LISA [19, 20]. Разработанный на его основе генератор симуляторов MaxCore порождает быстрые и точные симуляторы для широкого класса аппаратных платформ, включающего DSP, RISC, VLIW и др.

Язык описания системы команд ISDL [9] предназначен для настройки компилятора, ассемблера и симулятора на VLIW-архитектуру. Извлекаемая из описания на этом языке специальная структура данных (Split-Node DAG) описывает возможные способы группировки микроинструкций в длинное машинное слово. На основе данного формализма построен настраиваемый кодогенератор AVIV [11].

Заключение

Оценка перспективности предлагаемой технологии не может быть осуществлена без ее апробации. На настоящий момент разработан язык описания макроархитектуры [2] и генераторы, порождающие все необходимые для работы средства, а также VHDL-модель [3, 4]. Оценка этих средств для существующих систем команд свидетельствует о перспективности данной технологии; ведется работа над схемой извлечения макроархитектуры из приложения и построения настраиваемого планировщика.

Список литературы

[1] Булычев Д.Ю. Разработка алгоритма взаимного синтеза целевой программы и системы команд для реализации

программно-аппаратных систем: Отчет по гранту Министерства образования России. — СПб.: 2003. — 7 с.

[2] Булычев Д.Ю. Язык описания макроархитектуры для технологии совместной программно-аппаратной разработки // Наст. сборник. — С. 23-48.

[3] Симановский А.А., Шапоренков Д.А. Синтез ассемблера и дизассемблера из описания макроархитектуры // Наст. сборник. — С. 49-74.

[4] Смирнов М.Н. Порождение модели микроархитектуры из модели макроархитектуры // Наст. сборник. — С. 75-88.

[5] Barabanov A., Bombana M., Fominykh N. e.a. Reusable Objects for Optimized DSP Design // Embedded Microprocessor Systems. — 1996. — P. 433-442.

[6] Boulytchev D., Lomov D. An Empirical Study of Retargetable Compilers // Proc. of 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics. — 2001. — P. 328-331.

[7] Fauth A., Van Praet J., Freericks M. Describing Instruction Set Processors Using nML // Proc. on the European Design and Test Conference. — 1995. — P. 503-507.

[8] Gschwind M. Instruction Set Selection for ASIP Design // Proc. of the Seventh International Workshop on Hardware/Software Codesign. — 1999. — P. 7-11.

[9] Hadjiyiannis G., Hanono S., Devadas S. ISDL: an Instruction Set Description Language for Retargetability // Proc. of the 34th Annual Conference on Design Automation. — 1997. — P. 299-302.

[10] Halambi A., Grun P., Khare A. e.a. EXPRESSION: a Language for Architecture Exploration through Compiler/Simulator Retargetability // Proc. Design Automation and Test in Europe Conf. — 1999. — P. 485-490.

[11] Hanono S., Devadas S. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator // Proc. of the 35th Annual Conference on Design Automation. — 1998. — P. 510-515.

[12] Huang I-J., Despain A.M. Generating Instruction Sets and Microarchitectures from Applications // Proc. of the IEEE/ACM International Conference on Computer-Aided Design. — 1994. — P. 391-396.

[13] Lanneer D., Van Praet J., Kifli A. e.a. CHESS: Retargetable Code Generation for Embedded Processors // Code Generation for Embedded Processors. — Boston/London/Dordrecht: Kluwer Academic Publishers, 1995. — P. 85-102.

[14] Leupers R., Marwedel P. Instruction Set Extraction From Programmable Structures // Proc. of the European Design Automation Conference. — 1994. — P. 156-161.

[15] Leupers R., Marwedel P. Retargetable Code Generation Based on Structural Descriptions // Design Automation for Embedded Systems. — Vol. 3. № 1. — Boston/London/Dordrecht: Kluwer Academic Publishers, 1998. — P. 1-36.

[16] Leupers R., Marwedel P. Retargetable Generation of Code Selectors from HDL Processor Models // Proc. of the 1997 European Design and Test Conference. — 1997. — P. 140-144.

[17] Muchnik S. Advanced Compiler Design and Implementation. — San Francisco: Morgan Kaufmann Publishers, 1997. — 855 p.

[18] Paulin P., Liem C., Sutarwala S. FlexWare: a Flexible Firmware Development Environment for Embedded Systems // Code Generation for Embedded Processors. — Boston/London/Dordrecht: Kluwer Academic Publishers, 1995. — P. 67-84.

[19] Pees S., Hoffmann A., Zivojnovic V., Meyr H. LISA — Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures // Proc. of the 36th ACM/IEEE Conference on Design Automation. — 1999. — P. 933-938.

[20] Schliebusch O., Hoffmann A., Nohl A. e.a. Architecture Implementation Using the Machine Description Language LISA // Proc. of the 2002 Conference on Asia South Pacific Design Automation/VLSI Design. — 2002. — P. 239-245.

[21] Van Praet J., Goossens G., Lanneer D., de Man H. Instruction Set Definition and Instruction Selection for ASIPs // Proc. of the 7th ACM/IEEE International Symposium on High-Level Synthesis. — 1994. — P. 11-16.

[22] Weaver C., Krishna R., Wu L., Austin T. Application Specific Architectures: a Recipe for Fast, Flexible and Power Efficient Designs // Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. — 2001. — P. 181-185.