Обзор методов описания встраиваемой аппаратуры и построения инструментария кросс-разработки

В.В. Рубанов

Аннотация. Статья посвящена обзору методов описания расширяемых встраиваемых систем и построения соответствующих инструментов кросс-разработки (симулятор, ассемблер, дисассемблер, компоновщик, отладчик и т.п.). Рассматривается общий процесс проектирования встраиваемых систем и описывается роль инструментария кросс-разработки. Обсуждаются языки для описания моделей встраиваемых систем и методы получения инструментария кросс-разработки на основе таких описаний. Проводится сравнительный анализ рассмотренных решений.

Введение

В современном мире все бо́льшее распространение получают системы на основе *встраиваемых процессоров*, предназначенных для эффективного выполнения узкого класса задач в условиях жестких ограничений на соотношение производительности, энергопотребления, размера и стоимости изготовления кристалла. Такие системы можно встретить практически в каждом электронном устройстве, начиная от бытовой техники и кончая самолетами и военными комплексами. При этом большую популярность приобретает подход к построению встраиваемых систем на основе *расширяемых* процессоров, включающих некоторое базовое микропроцессорное ядро (soft core), которое дополняется в процессе проектирования специфическими для конкретной системы расширениями в виде сопроцессоров и/или дополнительных функциональных блоков, расширяющих систему команд и подсистему памяти ядра. При таком подходе одно и то же ядро повторно используется в системах различного назначения, существенно сокращая затраты на проектирование. При этом использование специализированных для каждой системы расширений обеспечивает высокую техническую эффективность в смысле баланса указанных выше показателей.

В процессе создания встраиваемых систем важнейшую роль играет *инструментарий кросс-разработки*, позволяющий выполнять разработку, отладку и профилирование программ для целевой системы с использованием инструментальной машины с отличной от целевой архитектурой. Основными компонентами такого инструментария являются ассемблер, компоновщик,

симулятор, отладчик и профилировщик. В качестве инструментальной машины, как правило, выступает обычная рабочая станция. В отличие от производства реальных микросхем, для построения кросс-инструментария достаточно некоторого высокоуровневого описания целевой системы – прежде всего структуры памяти/регистров и системы команд с временны́ми характеристиками исполнения. Это делает возможным раннее создание инструментария кросс-разработки еще в процессе проектирования аппаратуры. Использование кросс-инструментария на этом этапе играет ключевую роль при решении следующих задач:

1. *Прототипирование целевой аппаратуры* и исследование проектных альтернатив (design space exploration) – разработка набора типовых тестов (т.е. программ для целевой машины), их запуск и профилирование на различных вариантах аппаратуры позволяет получать оценки эффективности того или иного проектного варианта и принимать решения о выработке новых улучшений, например, оптимизации системы команд ядра, добавлении / удалении тех или иных функциональных блоков, регистров и сопроцессоров.
2. *Раннее создание приложений* – программное обеспечение для целевой платформы должно быть создано и предварительно отлажено еще до появления реальной аппаратуры. Это необходимо для сокращения времени выхода на рынок полного решения в виде «аппаратура + программы».
3. *Верификация спецификаций аппаратуры* – использование построенного кросс-симулятора позволяет проводить его взаимную верификацию с симуляторами, полученными на основе точной VHDL/Verilog спецификации целевой системы (после того, как такая спецификация будет создана на позднем этапе проектирования). Такая верификация играет важную роль в процессе финального обеспечения качества перед запуском аппаратуры в производство.

Конечно, важно, чтобы после завершения проектирования аппаратуры полученные кросс-инструменты были пригодны для собственно производственного применения при дальнейшей разработке реальных приложений.

В данной статье будут рассмотрены различные современные средства описания моделей аппаратуры, пригодные для построения на основе таких описаний соответствующих кросс-инструментов. При рассмотрении таких методов создания кросс-инструментария будем иметь в виду следующие «идеальные» требования.

1. Получаемый кросс-инструментарий должен обладать *высокой скоростью* работы (десятки миллионов модельных тактов в секунду на современных рабочих станциях) и *потактовой точностью* моделирования.

2. В процессе построения должен обеспечиваться *быстрый цикл внесения согласованных изменений* в кросс-инструменты для отражения различных вариантов аппаратной системы, возникающих как в процессе проектирования ядра, так и в процессе разработки расширений и выборе конфигурации полной системы.

3. В случае расширяемой аппаратуры необходима возможность *разделения разработки* базового инструментария (для базового ядра) и соответствующих модулей/инструментов для различных расширений с возможностью комбинации соответствующих компонентов при построении расширенного инструментария для полной системы [1].

Применение такого «идеального» метода позволило бы эффективно решать поставленные выше задачи прототипирования расширяемой аппаратуры с потактовой точностью, верификации VHDL/Verilog моделей и собственно разработки реальных приложении как на этапе проектирования, так и на этапе эксплуатации аппаратуры.

Статья состоит из введения, трех разделов и заключения. Во втором разделе рассматривается процесс проектирования встраиваемых систем и описывается роль инструментария кросс-разработки. В разделе 3 дается обзор языков для описания моделей встраиваемых систем и соответствующих методов получения инструментария кросс-разработки на основе таких описаний. В четвертом разделе проводится сравнительный анализ рассмотренных решений. В заключении подводятся итоги и предлагаются направления создания новых методов.

1. Проектирование встраиваемых систем

В СССР первыми встраиваемыми компьютерными системами можно считать специализированные бортовые вычислительные машины для военных и космических отраслей. Первые такие машины начали разрабатывать в конце 1950-х (см. [1]-[3]) на базе появившихся тогда сплавных транзисторов, и в начале 1960-х их уже стали применять на практике. В [1] в качестве одних из первых таких систем упоминаются передвижные компьютеры для нужд ПВО (1960-1962), обеспечивавшие управление зенитно-ракетными комплексами с сопровождением многих десятков целей. В зарубежных источниках [4] первой широко известной встраиваемой системой называют бортовой компьютер космического корабля Apollo (середина 1960-х).

Долгое время основной областью применения встраиваемых систем были именно задачи космического и военного назначения. В современном мире

[1] Дело в том, что за разработку базового процессора и за разработку расширений могут отвечать разные компании, причем каждая из них, как правило, помимо разграничения ответственности, желает сохранить детали конструкции соответствующей аппаратуры в тайне.

встраиваемые системы можно найти в самых различных областях от той же военной индустрии до бытовых устройств. Особенностью проектирования классической встраиваемой системы (см., например [5], [8]) является изначальное построение программно-аппаратного комплекса «в целом» под заранее известный набор фиксированных задач. При этом проектирование встраиваемой системы состоит в построении спецификаций ее аппаратных и программных компонентов, пригодных для производства реальных устройств и выполняющих заданные функции в рамках определенных ограничений (обычно быстродействие, энергопотребление, размер и стоимость изготовления кристаллов). В качестве конечной спецификации программной части системы выступает образ начального содержимого памяти системы (firmware), представляющий собой двоичные коды программ (машинные команды) и начальные данные. Спецификацией аппаратуры является описание на некотором языке, пригодное для дальнейшего полностью автоматического синтеза технологических спецификаций для производства реальных микросхем.

1.1. Обобщенная схема проектирования встраиваемых систем

Рассмотрим известную (см. например [5-6]) обобщенную схему проектирования встраиваемой системы (рис. 1).

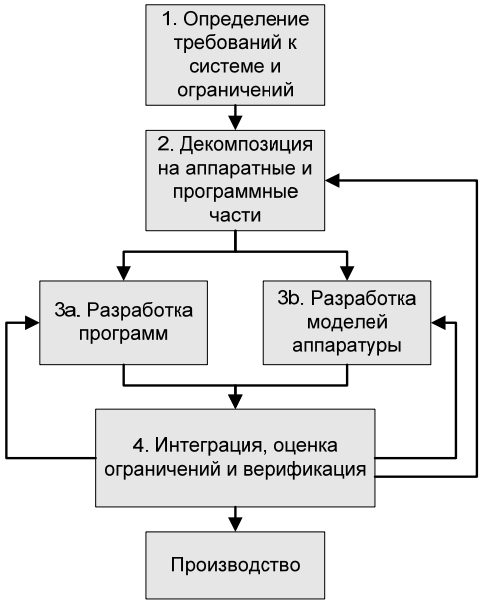

Рис. 1. Обобщенная схема проектирования встраиваемой системы

На первом этапе происходит определение требований к системе. Определяются необходимые функциональные характеристики системы и задаются ограничения. Типовыми ограничениями являются быстродействие, энергопотребление, размер и стоимость изготовления кристаллов в рамках заданного технологического процесса производства микросхем.

На следующем этапе выполняется декомпозиция системы на аппаратные и программные компоненты (HW/SW partitioning). Принимаются решения об общей структуре системы (в первую очередь, число и характеристики вычислительных блоков) и выполняется отображение требуемой функциональности на программные и аппаратные части.

Далее процесс разделяется на две ветви – для проектирования программных и аппаратных компонентов. Выходом аппаратной ветви являются модели аппаратуры. В этой ветви принимаются решения об архитектуре выделенных в системе аппаратных вычислительных устройств. Для программируемых компонентов определяется состав функциональных блоков (включая внешние модули расширений для специфических вычислений), структура памяти (включая регистры) и система команд. Результатом проектирования программной части являются модели программных компонентов, совместимые с соответствующими аппаратными моделями.

Процесс носит итеративный характер, и точность описания моделей на каждой итерации постепенно повышается от высокоуровневых функциональных описаний до синтезируемых спецификаций аппаратуры и машинных кодов программ, соответствующих этой аппаратуре. Каждая итерация заканчивается интеграцией результатов программной и аппаратных ветвей, моделированием полученной системы, проверкой функциональной корректности и сбором соответствующих оценок ключевых параметров для их анализа с целью дальнейшей оптимизации. На основании полученных таким образом оценок принимаются решения о пересмотре декомпозиции между программными и аппаратными компонентами, о конкретных изменениях программ и аппаратуры, например, добавлении/удалении вычислительных блоков или оптимизации системы команд. Цикл повторяется до получения конкретных спецификаций программ и аппаратуры, которые совместно задают встраиваемую систему, удовлетворяющую всем заданным требованиям. Затем проводится верификация спецификаций, и цикл проектирования завершается подготовкой отчуждаемого продукта, пригодного для интеграции в более крупные проекты «систем на чипе» (SoC) или для запуска в отдельное производство. Такой продукт обычно включает в себя:

1) синтезируемые RTL (register transfer level) описания аппаратуры (обычно на VHDL/Verilog);

2) исходные (на C/ассемблере) и машинные (firmware) коды базовых системных и прикладных программ для целевой аппаратуры;

3) набор инструментов кросс-разработки (среда программирования, см. подраздел 1.2) для создания и отладки новых программ;

4) документацию для программистов (справочники по архитектуре системы, по системе команд, по поставляемому системному программному обеспечению и различным библиотекам, по среде программирования).

Существует много различных методов и средств автоматизации проектирования аппаратуры (см., например, обзор [7]). В случае встраиваемых систем огромное внимание уделяется задаче оптимального разбиения системы на аппаратные и программные компоненты (HW/SW partitioning and codesign) – см. [8-13]. Однако в данной статье мы ограничимся только рассмотрением создания и использования (кроме собственно основного назначения для разработки реальных программ) кросс-инструментария как средства получения дополнительных данных для поддержки принятия проектных решений в процессе проектирования аппаратуры (см. следующие разделы). Конкретные методы использования таких данных выходят за рамки данной статьи.

1.2. Разработка программ с помощью кросс-инструментария

Целью использования кросс-инструментов является создание на инструментальной машине файла с двоичным образом (firmware) начального содержимого памяти (как машинные команды, так и данные) для целевой аппаратной системы. Такой файл затем используется для загрузки в конкретные целевые устройства.

В качестве языков программирования встраиваемых систем выступают обычно C (а также его расширенные подмножества типа Embedded C [14]) и ассемблер. Ассемблер используется как в виде вставок в код на языке C, так и в виде отдельных ассемблерных модулей. В случае встраиваемых систем программирование на ассемблере остается важной составляющей создания программ ввиду, как правило, жестких требований к высокой производительности и малому объему программного кода.

На рис. 2 представлена типовая схема разработки программ с помощью кросс-инструментария.

Обычно процесс взаимодействия программиста с кросс-инструментами происходит через визуальную *интегрированную среду разработки* (IDE) со встроенным редактором, механизмами поддержки проектов, различными средствами управления и анализа результатов работы отдельных инструментов.

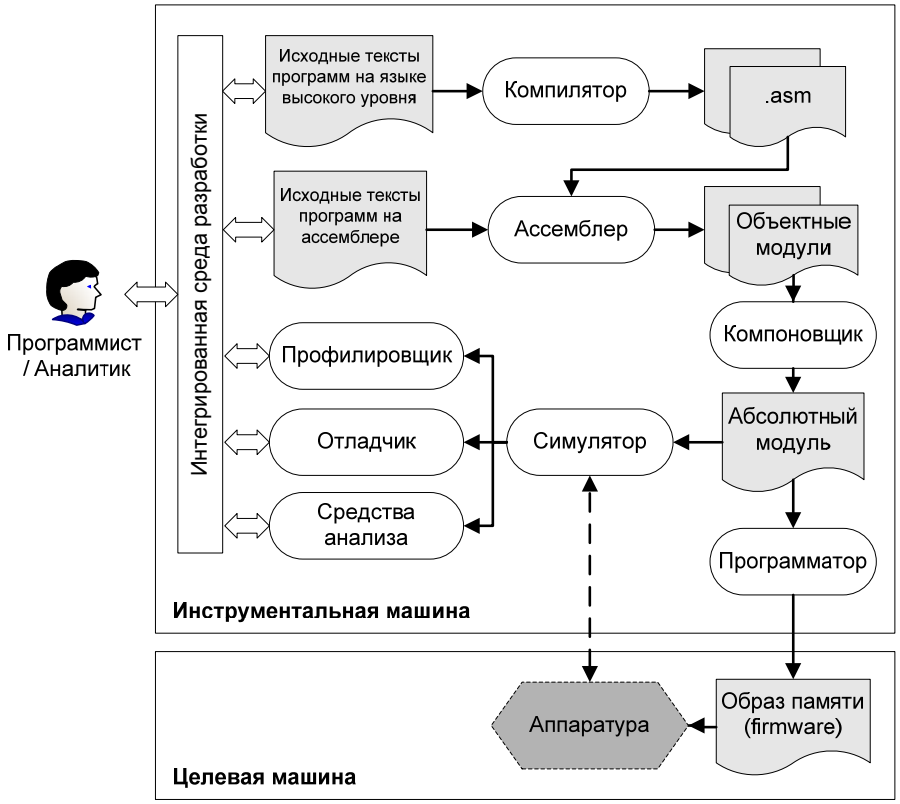

Рис. 2. Разработка программ с помощью кросс-инструментов.

Реальные программы обычно состоят из нескольких модулей, каждому из которых соответствует файл с исходными текстами программ (на языках высокого уровня или ассемблера). *Компилятор* транслирует модули на языке высокого уровня в промежуточные ассемблерные модули. *Ассемблер* отвечает за преобразование ассемблерных модулей (как написанных вручную, так и сгенерированных компилятором) в объектные модули с машинными кодами и данными для целевой аппаратуры. В качестве формата объектных модулей обычно используется ELF [15-16], включающий в себя различные секции (например, секции исполняемого кода, секции данных, секции с информацией о символах и секции с отладочной информацией). *Компоновщик* выполняет сборку нескольких объектных модулей в один абсолютный модуль с объединением соответствующих секций входных объектных файлов. При этом выполняются перемещения символов по абсолютным адресам памяти (автоматически или в соответствии с заданной программистом картой памяти)

с соответствующими исправлениями зависящих от них кодов команд и значений данных. На этом заканчивается этап сборки программы. Полученный абсолютный модуль можно преобразовать в образ памяти для непосредственной загрузки в целевую аппаратуру с помощью *программатора*.

Для программиста взаимодействие с исполняемой моделью аппаратуры осуществляется через *отладчик*, который позволяет просматривать состояние модели (содержимое памяти, регистров, шин, сигналов) и осуществлять управляемое (в том числе, пошаговое) выполнение целевой программы на уровне отдельных команд или строчек исходного кода. Совместно с отладчиком используются различные виды *профилировщиков* и *средств анализа*, визуализирующих необходимые характеристики модели (как статические, так и времени исполнения) и соответствующие статистики. В качестве примеров можно привести:

1) подсчет числа тактов и количества раз исполнения для каждой строчки программы (на уровне исходных кодов языка программирования высокого уровня, на уровне ассемблерных текстов и на уровне дисассемблированных команд процессора);
2) визуализация графа вызовов функций и статистика затраченных для каждого узла тактов и количества раз исполнения;
3) список функций программы, их размеров в программном коде и отражение количества вызовов и суммарных затрат (тактов) на их выполнение (с учетом и без учета вызова потомков);
4) статистика доступа к различным областям памяти;
5) статистика используемого объема памяти;
6) различные статистики на уровне операционной системы (в терминах задач и примитивов синхронизации, зарегистрированных в системе).

В качестве модели целевой системы для кросс-отладчика чаще всего выступает *симулятор*, позволяющий моделировать целевую аппаратуру полностью на инструментальной машине. Существуют симуляторы различного уровня абстракции от функционального симулятора на уровне всей системы до симуляторов на уровне системы команд (в том числе потактово-точных) и симуляторов, эмулирующих точную структуру аппаратуры на уровне функциональных блоков и конкретных вентилей. В качестве симулятора в составе инструментария кросс-разработки обычно используется симулятор уровня системы команд (в том числе, с учетом конвейерных эффектов с потактовой точностью). Также отладчик может поддерживать отладку непосредственно аппаратной модели в виде реального чипа или модели в ПЛИС (FPGA), подключаемой к инструментальной машине, что может иметь место на финальных стадиях проектирования и на

стадии эксплуатации. Использование настоящей аппаратуры позволяет запускать программы в режиме реального времени, однако программный симулятор предоставляет гораздо больше возможностей для отладки и анализа программ.

1.3. Прототипирование на основе кросс-инструментария

Уже на этапе проектирования для прототипирования и верификации целевой системы часто используется инструментарий кросс-разработки (см. например [17-19]). Это позволяет эффективно выполнять тонкую оптимизацию системы (как аппаратуры, так и программ для нее) с использованием системных и прикладных целевых программ, близких к реальным. Благодаря этому, дальнейший переход к выпуску промышленного набора результатов проектирования происходит бесшовно, то есть последняя итерация проектирования и верификации оканчивается результатами, готовыми к производственному применению (спецификацией аппаратуры и программами для нее).

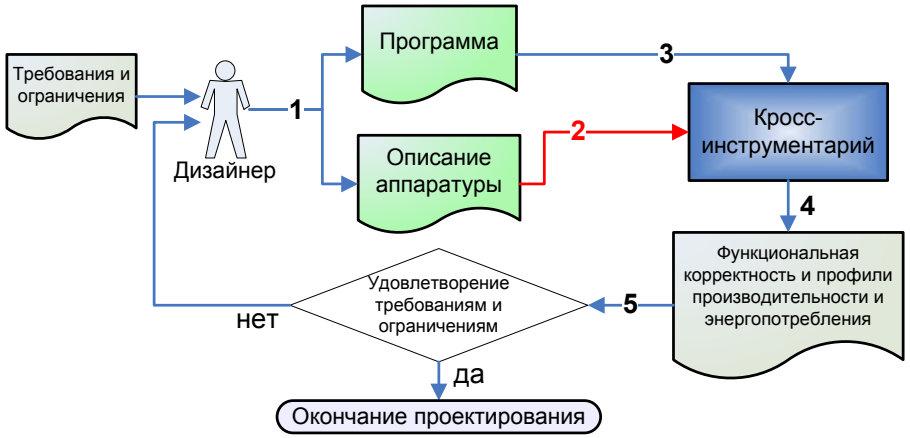

Рис. 3. Прототипирование системы с помощью кросс-инструментов.

Общий процесс прототипирования аппаратуры на основе кросс-инструментария показан на рис. 1.3. Роль инструментария кросс-разработки в этом процессе заключается в валидации функциональной корректности предполагаемой системы и получении достаточно точных оценок ее характеристик (в первую очередь, профилей производительности и энергопотребления, а также размеров требуемой памяти). Для этого на каждой итерации создается конкретная целевая программа и задается описание модели аппаратуры (1). Затем инструментарий кросс-разработки адаптируется под заданную модель аппаратуры (2). С помощью полученных кросс-

инструментов выполняется сборка, отладка, запуск и профилирование целевой программы (3). Получаются (4) значения необходимых характеристик системы, которые используются (5) для оценки удовлетворения требованиям/ограничениям и принятия решения о путях дальнейшей оптимизации системы (изменения программы/системы команд процессора, добавление/удаление аппаратных расширений, регистров и т.п.).

2. Языки описания моделей аппаратуры и соответствующие методы построения кросс-инструментов

В данном разделе рассматриваются существующие языки описания моделей целевой аппаратуры и соответствующие методы построения инструментария кросс-разработки на основе таких описаний. Выделяются три группы средств описания:

1) языки описания аппаратуры с возможностью синтеза реальных спецификаций для производства микросхем;
2) языки ADL (Architecture Description Languages) для высокоуровневого описания аппаратуры;
3) языки программирования общего назначения.

2.1. Языки синтезируемого описания аппаратуры

Рассмотрим три основных языка **HDL** (Hardware Definition Language) [20], которые используются современными разработчиками для описания моделей аппаратуры, пригодных для дальнейшего автоматического синтеза спецификаций для производства реальных чипов. Это **VHDL**, **Verilog** и **SystemC**. Первые два языка были разработаны в середине 80х и до сих пор являются наиболее популярными классическими HDL-языками; SystemC – относительно современная разработка, находящаяся в стадии развития, но стремительно набирающая популярность.

На верхнем уровне эти языки очень схожи – модель аппаратуры описывается в виде взаимодействующих модулей (блоков), для каждого из которых определяется *интерфейс* и *реализация*. Интерфейсы модулей описывают входные, выходные и двусторонние порты, с помощью которых модули соединяются друг с другом для обмена данными и управляющими сигналами. Реализация задает элементы внутреннего состояния и порядок вычисления значений выходных интерфейсов на основе этого состояния и значений входных портов, а также правила обновления внутреннего состояния. Вычисление новых значений выходных интерфейсов и внутреннего состояния инициируется событиями в виде изменения уровней входных сигналов. Важную роль в описании аппаратуры с использованием языков HDL играет понятие времени, которое моделируется целочисленной величиной с

возможностью привязки к физическому времени. Для описания причинно-следственных отношений между событиями в рамках одной единицы модельного времени используется понятие *дельта-задержки* (*delta delay*), которая разделяет последовательно выполняющиеся события, происходящие в одну и ту же единицу модельного времени. При описании реализации модуля могут использоваться выполняющиеся параллельно *процессы*, которые обычно представляют собой бесконечные циклы ожидания наступления некоторых заданных событий (называемых *списком чувствительности процесса*) с их последующей обработкой и возвратом к ожиданию новых изменений.

Рассмотрим каждый из языков более подробно.

3.1.1. VHDL

VHDL [21-24] был разработан в недрах Министерства Обороны США, изначально предназначаясь для облегчения унифицированного описания микросхем, которые включались сторонними поставщиками в различные решения для этого ведомства. Первая официальная версия VHDL появилась в 1987 году в виде стандарта IEEE 1076-1987 [24]. Многие семантические и синтаксические элементы VHDL заимствованы из языка Ada. Подобно Ada, VHDL – это строго типизированный язык, не чувствительный к регистру символов. В дополнение к стандартным базовым возможностям Ada, VHDL включает расширенные логические операции (например, `nand` и `nor`), двунаправленную индексацию массивов, а также дополнительные типы, такие как `time`, `bit`, `bit_vector`, `character`, `string`. Позже в VHDL ввели понятие 9-значной (U,X,0,1,Z,W,H,L,-) логики (см. IEEE Std 1164 [25]) и понятие знаковых/беззнаковых типов (см. IEEE standard 1076.3 [26]). Принципиальной особенностью VHDL является поддержка конструкций для задания параллелизма, свойственного аппаратуре, а именно модулей и процессов. Интерфейс модуля задается с помощью ключевого слова `entity`, ключевое слово `architecture` обозначает описание реализации, которое заключается между `begin` и `end`. Внутри такого блока могут задаваться константы (`constant`), сигналы (`signal`) и собственно поведение в виде набора операторов, в том числе, сгруппированных в виде параллельно выполняющихся процессов (с помощью ключевого слова `process`). Внутри процессов могут объявляться переменные (`variable`). Важным различием переменных и сигналов является то, что значение переменной меняется сразу после выполнения соответствующего оператора (понятие времени не ассоциируется с понятием переменной), а значение сигнала меняется только после окончания текущей итерации выполнения процесса.

Пример 1 иллюстрирует реализацию на языке VHDL простого мультиплексора (см. рис. 4).

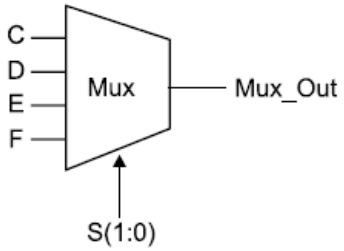

Рис. 4. Простой мультиплексор

```
entity mux is
     port (c, d, e, f:     in std_logic;
           s:              in std_logic_vector(1
downto 0);
           mux_out:  out std_logic);
end mux;

architecture mux_impl of mux is
begin
     mux1: process (s, c, d, e, f)
     begin
          case s is
                when "00" => mux_out <= c;
                when "01" => mux_out <= d;
                when "10" => mux_out <= e;
                when others => mux_out <= f;
          end case;
     end process mux1;
end mux_impl;
```

Пример 1. Простой мультиплексор на VHDL.

3.1.2. Verilog

Язык **Verilog** [21], [28-30] был разработан компанией Gateway Design Automation в 1985 году для целей проектирования интегральных схем на логическом уровне. В 1989 году компания Gateway Design Automation была куплена Cadence, которая сделала этот язык публичным. В 1995 году Verilog стал стандартом IEEE 1364 [29].

Verilog наследует многое из языка C, поддерживает конструкции препроцессора C и большинство основных управляющих конструкций, таких

как “if”, “while” и т.п. Verilog полностью поддерживает операторы языка C, но также обладает дополнительными возможностями для удобной обработки двоичных данных (например, операциями ~^ для XNOR или >>> для арифметического сдвига с заполнением знаковым битом). Однако в Verilog не поддерживаются пользовательские типы, структуры, указатели и рекурсивные функции. Типы данных в Verilog имеют явную битовую ширину. В отличие от VHDL, Verilog является слабо типизированным языком, что позволяет смешивать присвоения элементов разного типа за счет неявного преобразования типов.

Как и в случае VHDL, принципиальным отличием Verilog от своего языка-прототипа является поддержка конструкций, выполняющихся параллельно. Модули в Verilog определяются с помощью ключевого слова `module` и могут быть вложенными. Не происходит явного разделения интерфейса и реализации, как в VHDL. В декларативной части модуля определяются входные (`input`), выходные (`output`), двунаправленные (`inout`) порты, внутренние сигналы (`wire`) и переменные (`reg`).

```
module bcircuit (a, b, av, bv, w, wv);
      input a, b;
      output w;
      input [7:0] av, bv;
      output [7:0] wv;
      wire d;
      wire [7:0] dv;
      reg e;
      reg [7:0] ev;
      . . .
endmodule
```

Пример 2. Описание модуля и его интерфейсов в Verilog

В процедурной части модуля определяется его поведение. Для этого могут использоваться конструкции вентильного уровня (определяется структура модуля в виде соединенных между собой вентилей определенных типов), параллельные присвоения (`assign`) или процедурные блоки (`always` или `initial`). Процедурные блоки аналогичны процессам VHDL. Конструкции языка в рамках процедурного блока выполняются последовательно, в то время как все процедурные блоки выполняются параллельно. Интересно отметить наличие оператора неблокирующего присваивания <=, который может использоваться для присваивания новых значений в рамках процедурного блока. В отличие от обычного присваивания =, изменение значения, присвоенного оператором <=, происходит только в конце текущего кванта времени.

Важным отличием Verilog от VHDL является подверженность недетерминированному поведению (race conditions) модели на основе описания Verilog и отсутствие таких эффектов в VHDL. Более подробный сравнительный обзор VHDL и Verilog можно найти в [34].

Пример 3 иллюстрирует реализацию простого мультиплексора (см. рис. 4) на языке Verilog.

```
module mux (c, d, e, f, s, mux_out);
      input       c, d, e, f;
      input       [1:0] s;
      output      mux_out;
      reg         mux_out;

      always @ (c or d or e or f or s)
      begin
         case (s)
           2'b00: mux_out = c;
           2'b01: mux_out = d;
           2'b10: mux_out = e;
           default: mux_out = f;
         endcase
      end
endmodule
```

Пример 3. Простой мультиплексор на Verilog.

3.1.3. SystemC

Работа над языком **SystemC** [30-31] была начата в середине 1990-х в качестве внутреннего проекта компании Synopsis, и язык был открыт сообществу в 1999 году. В 2000 году был учрежден консорциум Open SystemC Initiative [31], в который вошли такие заинтересованные компании, как Mentor Graphics, Cadence, ARM, CoWare. Образование этого консоциума позволило проводить работы по развитию SystemC в интересах всего сообщества. Первая версия SystemC 1.0 вышла в этом же году и сменилась следующей в 2001 году. В 2005 году для SystemC появился стандарт IEEE Std 1666 [32]. В настоящее время ведутся работы над созданием новой версии SystemC 3.0.

На самом деле, SystemC не является самостоятельным языком. Фактически, это библиотека классов, типов и макросов C++, которая позволяет удобно описывать программно-аппаратную систему на различных уровнях абстракции. Благодаря хорошим возможностям языка C++ для определения пользовательских типов, описания на SystemC выглядят достаточно выразительным образом, что позволяет эффективно расширять семантику

базового языка. К основным расширениям SystemC, явно отсутствующим в C++, относятся:

1) понятие модельного времени;
2) возможности описания параллельно выполняющихся вычислений;
3) дополнительные типы данных, отражающие специфику проектирования аппаратуры (в первую очередь, многозначная логика 1, 0, X, Z).

Основным преимуществом SystemC над языками VHDL и Verilog является возможность использовать одно и то же окружение и язык для описания системы от самых высокоуровневых моделей на C++ до структурных синтезируемых моделей уровня RTL. Теоретически это позволяет постепенно детализировать описание системы в процессе проектирования. Однако пока инструменты поддержки процесса разработки с использованием SystemC уступают инструментарию классических языков VHDL и Verilog. Это приводит к тому, что SystemC в основном используется как «улучшенный» C++ для описания только высокоуровневых моделей системного уровня с последующим переходом на VHDL или Verilog для описания оптимизированной RTL модели для реального синтеза аппаратуры.

Так же, как и в VHDL и Verilog, основным строительным блоком в SystemC является модуль, который объявляется с использованием ключевого слова `SC_MODULE`. В модуле могут определяться порты (`sc_in`, `sc_out`, `sc_inout`), данные модуля (включая ссылки на другие модули) и методы. В каждом модуле обязательно должен присутствовать метод-конструктор, который в SystemC обозначается `SC_CTOR`. Также может объявляться деструктор с использованием обычного синтаксиса C++. Остальные методы делятся на *параллельные* и *обычные* (вспомогательные). Обычные методы применяются, как повторно используемые функции для упрощения реализации параллельных методов. Параллельные методы SystemC аналогичны процессам VHDL и Verilog и служат основным средством для описания деталей поведения модели. В SystemC выделяют два основных типа параллельных методов – `SC_METHOD` и `SC_THREAD`. Тип метода (обычный или подтип параллельного) указывается в конструкторе модуля. Основное различие между `SC_METHOD` и `SC_THREAD` заключается в порядке их исполнения в рамках модели. `SC_METHOD` запускается в ответ на события из своего списка чувствительности столько раз, сколько раз срабатывает эти события. При этом каждый запуск независим друг от друга в смысле сохранения значений локальных переменных, и выполняется он до конца метода или до явного `return`. В `SC_METHOD` нельзя применять функцию `wait` для динамического контроля над приостановкой процесса. `SC_THREAD` же запускается только один раз в начале исполнения модели, поэтому обычно реализация `SC_THREAD` представляет собой бесконечный цикл `while(1)`. Принципиальной особенностью `SC_THREAD` является возможность

использовать функцию `wait` для приостановки исполнения процесса до наступления заданных условий. При этом управление передается ядру симулятора и возвращается только при наступлении этих условий. Выполнение продолжается со следующей после `wait` конструкции с сохранением предыдущих значений локальных переменных метода. С точки зрения эффективности симуляции `SC_METHOD` является более быстрым вариантом `SC_THREAD`.

В SystemC вводятся следующие основные дополнительные типы данных:

- `sc_string` – строковое представление чисел в различных форматах (например, число -2 может быть представлено как -0d2 (десятичный), 0b1110 (4-х битный двоичный знаковый), 0b0010 (4-х битный двоичный беззнаковый));
- `sc_int<N>`, `sc_uint<N>` – знаковые и беззнаковые целочисленные типы заданной битовой ширины N;
- `sc_bigint<N>`, `sc_biguint<N>` – знаковые и беззнаковые целочисленные типы заданной битовой ширины N, превышающей длину значений типа `int`, который поддерживается на инструментальной машине;
- `sc_fixed`, `sc_ufixed` и др. – типы данных с фиксированной точкой
- `sc_logic` и `sc_lv<N>` – 4-х значный бит (1, 0, X, Z) и вектор таких битов соответственно.

Для облегчения описания взаимодействия модулей, кроме портов, в SystemC предлагаются более сложные каналы `sc_mutex`, `sc_fifo` и `sc_semaphore`.

Пример 4 иллюстрирует реализацию простого мультиплексора (см. рис. 1.4) на языке SystemC.

В отличие от VHDL и Verilog, в которых может использоваться интерпретатор описания для моделирования системы, в SystemC задается полная исполняемая модель, и описание на SystemC компилируется в исполняемый файл на инструментальной машине обычным компилятором C++ с использованием специальных библиотек SystemC. В начале симуляции модели, аналогично методу `main` в C, в SystemC управление передается в метод `sc_main(argc, argv)`, в котором происходит инициализация модулей системы и, в конечном итоге, запуск параллельных процессов инициализированных модулей с помощью функции `sc_start()`.

```
SC_MODULE (mux) {
public:
     sc_in  <sc_logic>     c, d, e, f;
     sc_in  <sc_lv<2>>     s;
     sc_out <sc_logic>     mux_out;

     SC_CTOR (mux) {
         SC_METHOD (do_mux);
         sensitive << c << d << e << f << s;
     }

     void do_mux () {
        sc_uint<2> temp_s = s.read();
        switch (temp_s) {
           case 0:  mux_out = c.read(); break;
           case 1:  mux_out = d.read(); break;
           case 2:  mux_out = e.read(); break;
           default: mux_out = f.read();
        }
     }
}
```

Пример 4. Простой мультиплексор на SystemC.

2.2. Кросс-инструменты поддержки HDL языков

Поскольку спецификации аппаратуры на языках VHDL, Verilog и SystemC принципиально не содержат явного описания системы команд, кросс-инструменты поддержки разработки прикладных программ на уровне языка ассемблера (ассемблер, дисассемблер, компоновщик) для описанной таким образом аппаратуры не могут быть получены автоматизированным образом. Поэтому главным доступным кросс-инструментом для описанной на этих языках аппаратуры является симулятор.

Для VHDL и Verilog кросс-симуляторы представляют собой программы для инструментальной машины, которые автоматически настраиваются для моделирования целевой аппаратуры на основе HDL-описания. Для VHDL и Verilog используются как подходы с интерпретацией таких описаний, так и с компиляцией модели в код инструментальной машины. Для SystemC-моделей характерен подход с компиляцией. Для заданного начального состояния полученные модели способны моделировать поведение системы с очень высокой точностью (состояние отдельных регистров, буферов, сигналов, шин и т.п. с квантованием по тактам или даже по отдельным событиям). Каждый

крупный производитель средств автоматизированной разработки аппаратуры (EDA) поддерживает собственные симуляторы для различных HDL-языков. Synopsis предлагает уже ставшую классической среду VCS [35], способную симулировать модели (в том числе смешанные) на всех трех основных языках - VHDL, Verilog и SystemC. Аналогичные смешанные симуляторы предлагаются компаниями Mentor Graphics (ModelSim [36]) и Cadence (NC-Sim [37], Incisive Verification Platform [38]). Все эти симуляторы поддерживаются интегрированными средами разработки и отладки моделей, в которых одним из основных средств визуализации отладки являются диаграммы изменения сигналов (waveforms).

Основной проблемой HDL-симуляторов является низкая скорость работы порядка 10-50 тыс. модельных тактов в секунду на современных рабочих станциях, в то время как типовой цикл моделирования алгоритмов цифровой обработки сигналов требует миллиарды тактов для обработки репрезентативного тестового сигнала. Такая скорость обусловлена высокой точностью моделирования на слишком низком уровне. Таким образом, скорость работы и отсутствие средств поддержки программирования на уровне системы команд делает применение HDL-симуляторов неприемлемым для эффективной разработки прикладных программ. Тем не менее, такие симуляторы играют важную роль в процессе дизайна и верификации самой аппаратуры.

2.3. ADL языки

Изначально ADL (Architecture Description Language) языки (см. обзоры [39-42]) начали появляться в начале 90-х в ответ на потребность в описании модели вычислительной аппаратуры на уровне явной системы команд для обеспечения раннего прототипирования встраиваемых микропроцессоров. Основу описания аппаратуры на ADL-языке составляет спецификация элементов состояния системы (регистры, память) и соответствующей системы команд, осуществляющей вычислительные операции над этими элементами. Основным фактором эффективного использования ADL в процессе проектирования аппаратуры является возможность автоматической генерации необходимых кросс-инструментов (прежде всего, симулятора и ассемблера) для обеспечения моделирования заданных тестовых программ на том или ином варианте аппаратуры. Процесс конструктивного исследования различных проектных альтернатив при проведении предварительного дизайна аппаратуры является наиболее типичным местом эффективного применения ADL-решений. Исходя из прикладных требований, архитектор выполняет разбиение задач на аппаратно и программно реализуемые части (SW/HW partitioning – см. разд. 1), то есть решает, какие функции следует заложить в виде аппаратной реализации, а какие – в виде прикладных программ. Программная реализация функций обеспечивает бо́льшую гибкость и минимизирует аппаратные ресурсы (что приводит к меньшей площади кристалла, меньшему энергопотреблению и стоимости), однако, с другой

стороны, существуют ограничения на производительность реального времени, которую во многих случаях программная реализация обеспечить не может (а повышение тактовой частоты ведет к ухудшению упомянутых параметров). Ясно, что теоретически существует некий оптимальный баланс между программной и аппаратной реализацией для конкретного класса алгоритмов, нахождением которого собственно и занимается архитектор на этапе проектирования. Однако ввиду того, что процесс разбиения задач между аппаратурой и программами носит эвристический характер на основе грубых оценок, необходима экспериментальная проверка корректности и эффективности различных вариантов. Для этой цели используют симуляцию тестовых программных задач на подразумеваемом аппаратном обеспечении (в виде симулятора). Вот почему наличие быстро обновляемого инструментария кросс-разработки очень важно на этом этапе, и ADL-решения наиболее подходят для достижения упомянутой цели.

Процесс использования ADL-решения начинается с описания общей архитектуры предполагаемой системы на формальном языке с фокусом на системе команд. Можно построить данный процесс на основе имеющихся библиотек решений, которые дизайнер может лишь слегка изменять вместо того, чтобы создавать заново. Инструменты поддержки должны уметь проверять целостность и корректность созданного описания. При корректном описании автоматически генерируется инструментарий кросс-разработки, с помощью которого конструктор может оценить ключевые параметры производительности предполагаемой системы и решить, насколько текущая архитектура удовлетворяет необходимым критериям. Среди критериев, анализируемых путем выполнения, профилировки и отладки программ, можно выделить семантическую корректность результата, производительность, энергопотребление, степень использования системы команд и внутренних аппаратных компонентов (регистров, функциональных единиц, шин) во время выполнения заданных приложений. Если какие-то параметры неудовлетворительны, то на основе полученной информации выявляются узкие места и принимаются решения по изменению аппаратуры (изменению системы команд и пересмотру баланса между аппаратурой и программным обеспечением). Данные изменения вносятся в ADL-описание и программы, и процесс повторяется. Таким образом, основными преимуществами ADL являются быстрота описания модели системы (за счет использования более высокоуровневых абстракций, чем в HDL) и автоматическая генерация инструментария кросс-разработки, что обеспечивает быструю оценку параметров различных вариантов аппаратуры на этапе проектирования.

Если все критерии соблюдены, то происходит детализация описания системы и начинается процесс детальной разработки. Некоторые ADL-решения поддерживают синтез шаблонов на HDL-языках, обеспечивая плавный переход к производственной доработке аппаратуры в системах проектирования аппаратного обеспечения.

Рассмотрим наиболее успешные ADL-языки и соответствующие инструменты более подробно.

3.3.1. nML

Язык nML [43-44] изначально был разработан в Техническом университете Берлина в 1991 году и является пионером в области ADL-решений, ориентированных на высокоуровневое описание системы команд. В этом университете nML использовался в качестве способа описания аппаратуры для настраиваемого симулятора SIGH/SIM и компилятора CBC (с языка ALDiSP). Язык nML получил дальнейшее развитие в бельгийском научно-исследовательском центре микроэлектроники IMEC, где в рамках дочерней компании Target Compiler Technologies была создана коммерческая среда разработки [45], ориентированная на DSP-архитектуры. В эту среду входят компилятор CHESS (с языка C), симулятор CHECKERS, ассемблер, дисассемблер и компоновщик. Также поддерживается синтез шаблонов VHDL-описания.

В nML выделяются определения типов (`type`), элементов хранения данных (`mem`) и собственно иерархическое описание системы команд, задаваемое с помощью атрибутных грамматик в виде OR и AND-правил. Элементами грамматики служат *операции* (`op`) и *режимы адресации* (`mode`). Для операций обязательные атрибуты включают в себя описание поведения на расширенном подмножестве C (`action`), ассемблерного синтаксиса (`syntax`) и отображения в машинные коды (`image`). Для режимов адресации – только синтаксис и двоичный код. Корневая операция имеет фиксированное имя `instruction`.

Базовые типы данных nML включают в себя:

- `int(n)` – тип знаковых целых n-битных чисел в дополнительном коде;
- `card(n)` – тип n-битных беззнаковых чисел;
- `float(n,m)` - тип знаковых чисел с плавающей точкой с `n`-битной мантиссой и `m`-битным основанием, в соответствии со стандартом IEEE-754;
- `fix(n,m)` – тип знаковых чисел с фиксированной точкой, с n битами до и m битами после двоичной точки;
- `bool` - булевский тип; предопределены две константы true и false; при приведении к целому true получает значение -1, a false значение 0.

Пример 5 иллюстрирует описание в nML тривиального процессора с шестнадцатью 16-битными регистрами и парой команд сложения и вычитания:

```
type word = card(16)      \ 16-битные данные
type index = card(4)      \ индекс для адресации 16 регистров

mem  RF[16, word]  \ 16 регистров размера word
mem  PC[1, word]          \ регистр-счетчик команд

mode REG(i:index)=RF[i]   \ режим прямой регистровой адресации
syntax = format("R%d",i) \ синтаксис вида R0, R1, .., R15
image  = format("%4b",i) \ тривиальное отображение номера
                          \ регистра в 4 бита машинного слова

op instruction (x:instr_action)
action = {
     PC = PC + 1;
     x.action;
}
syntax = x.syntax
image  = x.image

op instr_action = add_op | sub_op             \ OR-правило

op add_op (dst:REG, src:REG)                  \ AND-правило
action = { dst = dst + src; }
syntax = format("ADD %s, %s", dst.syntax, src.syntax)
image  = format("0000 %s %s", dst.image, src.image)

op sub_op (dst:REG, src:REG)
action = { dst = dst - src; }
syntax = format("SUB %s, %s", dst.syntax, src.syntax)
image  = format("0001 %s %s", dst.image, src.image)
```

Пример 5. Тривиальный процессор в nML

Существенным ограничением nML является отсутствие механизмов описания многотактовых функциональных единиц и конвейеров. Кроме того, в nML поддерживаются только команды фиксированной длины, не поддерживается описание межкомандных зависимостей, и производительность симулятора, опубликованная в [17], невысока.

3.3.2. ISDL

Язык ISDL был разработан группой CAA (Computer-Aided Automation) университета MIT, США [46] и впервые представлен на конференции по автоматизированному дизайну DAC [47] в 1997 году. Основной специализацией ISDL является описание VLIW-архитектур. Изначально язык задумывался для поддержки настраиваемых компилятора, ассемблера и симулятора, а также генератора Verilog-описаний на основе ISDL-спецификаций. Как и nML, ISDL, главным образом, позволяет на основе

использования атрибутной грамматики описывать систему команд процессора, включающую в себя семантику поведения, синтаксис ассемблера, машинные коды, а также описание ресурсных конфликтов. К важным достоинствам языка можно отнести возможность специфицировать задержки и конфликтные ситуации для параллелизма уровня команд (Instruction Level Parallelism, ILP) в виде логических правил, хотя явное описание конвейера отсутствует. Описание ISDL состоит из следующих секций:

- `Format` – формат машинного слова (разбиение на именованные битовые поля);
- `Global_Definitions` – глобальные определения лексем (`Token`) и нетерминальных символов (`Non_Terminal`);
- `Storage` – элементы-хранилища (регистры, память, стек, управляющие и специальные регистры);
- `Instruction_Set` – спецификация системы команд в виде набора операций, описание каждой из которых включает следующие атрибуты:
  - мнемоника;
  - параметры в виде лексем и нетерминальных символов;
  - бинарное кодирование в машинном слове;
  - поведение в виде RTL описания над ресурсами-хранилищами;
  - время выполнения и другие параметры стоимости (например, энергопотребление):
  - задержки;
- `Constraints` – ограничения на совместимость различных операций, как в рамках одной инструкции, так и между соседними инструкциями, а также ограничения в ассемблерном синтаксисе; ограничения записываются в виде логических правил, включающих в себя ссылки на параметры, константы и логические операции.

Описание тривиального процессора в ISDL приведено в примере 6.

```
SECTION Format
IW = OPF[4], DSTF[4], SRCF[4];

SECTION Global_Definitions
Token   "R"[0..15] REG {[0..15];};
Non_Terminal DST:  REG {$$ = REG;} {RF[REG]};
Non_Terminal SRC:  REG {$$ = REG;} {RF[REG]};

SECTION Storage
RegFile RF = 16, 16    // 16 16-битных регистров
ProgramCounter PC = 16 // 16-битный счетчик команд

SECTION Instruction_Set
Field ALU_OP:
     ADD DST, SRC            // ассемблерный синтаксис
```

```
    { IW.OPF = 0x0;
      IW.DSTF = DST;
      IW.SRCF = SRC; }       // двоичное кодирование
    { DST <- DST + SRC; }    // поведение
    {}                       // побочные эффекты
    { cycle = 1; size = 1; }// длительность и размер
    { latency = 1; }         // задержка

    SUB DST, SRC             // ассемблерный синтаксис
    { IW.OPF = 0x1;
      IW.DSTF = DST;
      IW.SRCF = SRC; }       // двоичное кодирование
    { DST <- DST - SRC; }    // поведение
    {}                       // побочные эффекты
    { cycle = 1; size = 1; }// длительность и размер
    { latency = 1; }         // задержка
```

Пример 6. Тривиальный процессор в ISDL

Для описания поведения используется собственный C-подобный язык с включенной библиотекой функций и расширенных операций работы над данными на битовом уровне. Базовые типы языка ограничены только знаковым и беззнаковым целыми, а также числами с плавающей точкой с параметрами, зависящими от инструментальной платформы, на которой работают инструменты ISDL.

Анализируя возможности ISDL, можно выявить следующие недостатки:

- нет механизмов описания поведения операции на конкретных тактах / стадиях конвейера, что делает невозможным потактово-точное моделирование;
- каждую операцию можно привязать только к одному функциональному «полю» (field), что затрудняет корректное описание использования ресурсов много-тактовыми командами;
- нет механизмов описания глобальных аспектов архитектуры таких, как прерывания, аппаратные циклы, конвейер;
- имеется лишь ограниченное число базовых типов (например, нет поддержки строк и чисел с фиксированной точкой).

Кроме того, к сожалению, отсутствуют в доступном виде реальные инструментальные средства, поддерживающие ISDL, так как инициаторы проекта ограничились только реализацией ассемблера, некоторых модулей симулятора GENSIM и кодогенератора для компилятора в качестве диссертационных работ MIT.

3.3.3. EXPRESSION

Язык EXPRESSION разрабатывался в Университете Калифорнии (University of California, Irvine, США) [48] и был впервые представлен на конференции DATE в 1999 году [49]. Этот язык поддерживает широкий класс встраиваемых систем с ILP и иерархиями памяти от RISC, DSP, ASIP до VLIW. EXPRESSION позволяет создавать интегрированное описание структуры и поведения подсистемы процессор-память. Спецификация на EXPRESSION состоит из шести секций (первые три отвечают за поведение, последние три за структуру):

- `OP_GROUP` – спецификация операций (элементарных команд (`OP_CODE`) с описанием параметров и поведения);
- `INSTR` – описание формата команды в виде набора ячеек (`SLOTS`), имеющих определенное положение и ширину в командном слове и ответственных за определенный функциональный модуль;
- `OP_MAPPING` – отображение общих (`generic`) операций компилятора на машинные операции, описанные в первой секции; данное описание используется при генерации кодогенератора компилятора;
- описание структурных компонентов – функциональные устройства (`UNIT`), элементы памяти (`STORAGE`), шины (`CONNECTION`) и порты (`PORT`); задаются связи между ними, а также некоторые свойства (например, типы операций, которое может выполнять устройство, и количество параллельно выполняемых за такт операций);
- описание конвейера (`PIPELINE`) в виде упорядоченных именованных стадий, связанных с функциональными устройствами; секция также содержит описание каналов передачи данных (`DTPATHS`).
- `STORAGE PARAMETERS` – описание свойств элементов памяти: тип (регистровая память, кэш, SRAM, DRAM), количество и размерность ячеек, ассоциативность кэша, адресное пространство, время доступа.

Пример 7 содержит описание тривиального модельного процессора на языке EXPRESSION.

```
(OP_GROUP alu_ops
     (OP_CODE add
           (OP_TYPE DATA_OP)
           (OPERANDS (DST reg) (SRC1 reg) (SRC2 reg))
           (BEHAVIOR DST = SRC1 + SRC2)
     )
     (OP_CODE sub
           (OP_TYPE DATA_OP)
           (OPERANDS (DST reg) (SRC1 reg) (SRC2 reg))
           (BEHAVIOR DST = SRC1 - SRC2)
     )
)
```

```
(VAR_GROUPS (reg RF) )

(INSTR
     (WORDLEN 12)
     (SLOTS ((TYPE DATA) (BITWIDTH 12) (UNIT ALU)) )
)

(SUBTYPE UNIT ExUnit)
(SUBTYPE STORAGE RegFile)

(ExUnit ALU
     (CONNECTIONS AluRfConn)
     (OPCODES alu_ops)
     (CAPACITY 1)
)

(RegFile RF (CONNECTIONS AluRfConn) )

(PIPELINE FETCH DECODE EX)
(EX: ALU)
(DTPATHS (TYPE BI (ALU RF AluRfConn) ))

(STORAGE PARAMETERS
     (RF
          (TYPE REGFILE)
          (SIZE 16)
          (WIDTH 16)
     )
)
```

Пример 7. Тривиальный процессор в EXPRESSION

На основе описания EXPRESSION автоматически генерируются компилятор EXPRESS и симулятор SYMPRESS [50-51]. Также существуют дополнительные утилиты для исследования и оценки использования иерархий памяти (MEMOREX) и визуальное средство для автоматизации процесса оценки и анализа различных архитектурных решений в процессе дизайна аппаратуры (V-SAT). Однако опубликованная скорость работы симулятора SYMPRESS является недостаточной для интерактивного процесса обработки миллиардов тактов в типичном цикле разработки современных алгоритмов цифровой обработки сигналов.

К сожалению, как сам язык EXPRESSION, так и поддерживающие его средства нацелены только на обеспечение процесса исследования проектных

альтернатив (DSE) и не рассчитаны на создание кросс-инструментов для разработки реальных прикладных программ. Поэтому полностью отсутствует промежуточный ассемблерный уровень – сценарий использования предполагает компиляцию программы на C во внутреннее промежуточное представление, которое непосредственно используется симулятором для получения оценок производительности, но не позволяет вести пошаговую отладку программы и использовать алгоритмы на ассемблере. Соответственно, в языке отсутствуют средства описания ассемблерного синтаксиса и двоичного кодирования команд. Также, ввиду наличия детальной структурной составляющей, начальное создание спецификации EXPRESSION является относительно трудоемким по сравнению с nML и ISDL. Кроме того, необходимость согласовывать поведенческие и структурные части описания делает неудобными и чреватыми ошибками изменения на уровне системы команд. В этом смысле EXPRESSION стоит между чистыми поведенческими ADL-решениями и структурными описаниями HDL-уровня.

2.4. Языки программирования общего назначения

Еще одним методом описания модели аппаратуры и построения соответствующего инструментария кросс-разработки является непосредственное использование высокоуровневых языков программирования общего назначения для реализации симулятора, ассемблера, дисассемблера, отладчика и прочих необходимых инструментов без использования каких либо специализированных для аппаратного обеспечения формальных описаний. Основными языками для этой цели обычно служат C и C++.

На практике большинству производителей встраиваемых процессоров (см. например, [52-57]) и многим компаниям, специализирующимся на разработке кросс-инструментов под заказ (например [58-65]), приходится использовать именно этот подход, чтобы получить кросс-инструментарий производственного качества как по уровню производительности, так и по удобству и богатству функциональности. Это становится возможным благодаря универсальным возможностям указанных языков программирования для реализации различных решений, оптимизированных под конкретную целевую аппаратуру.

Однако это достается дорогой ценой в терминах трудоемкости и календарного времени на создание кросс-инструментов, так как подразумевает большой объем ручного программирования. В случае разработки «с чистого листа» затраты на построение производственной версии полного набора кросс-инструментария составляют около 6-9 месяцев, и для этого требуются команды, включающие около 10 квалифицированных программистов. Адаптация уже существующих для подобной аппаратуры собственных (знакомых конкретным программистам) решений и повторное использование соответствующих библиотек позволяет сократить эти сроки и трудозатраты в несколько раз. Однако главным ограничением такого подхода все равно

остается требование к наличию устоявшегося описания целевой аппаратуры, так как адаптация построенных вручную инструментов даже к незначительным изменениям в спецификации аппаратуры требует существенных затрат и чревата ошибками. Например, как правило, несколько дней требуется для полного цикла проектирования, кодирования и тестирования необходимых изменений для отражения смены кодировки, синтаксиса и поведения всего одной команды. Данное ограничение не позволяет использовать такой подход на этапе проектирования аппаратуры, оставляя его возможным для применения только при построении инструментов для разработки прикладных программ уже на этапе серийного производства чипов (в условиях устоявшейся спецификации).

3. Анализ существующих подходов

В предыдущих разделах были рассмотрены различные средства описания аппаратуры и соответствующие методы построения кросс-инструментов. В данном разделе проводится итоговый сравнительный анализ рассмотренных решений.

Итак, качественные характеристики описанных выше классов языков для описания аппаратуры и соответствующих методов построения кросс-инструментария представлены в табл. 1.

На основе данных, приведённых в табл. 1, можно сделать следующие выводы (см. краткую сводку в табл. 2):

1. HDL-языки описания аппаратуры позволяют описывать наиболее точные модели аппаратуры и автоматически получать соответствующий симулятор, однако их использование в качестве основы для построения кросс-инструментария для прототипирования аппаратуры и разработки прикладных программ не представляется целесообразным ввиду низкой скорости работы симулятора, высокой трудоемкости построения модели аппаратуры и сложности внесения в нее изменений на уровне системы команд, а также из-за отсутствия в языке конструкций для задания информации, необходимой для построения остальных кросс-инструментов в дополнение к симулятору.

2. ADL-языки предоставляют наиболее удобные возможности для описания моделей аппаратуры на уровне системы команд, а соответствующий метод автоматического построения кросс-инструментария позволяет эффективно использовать получаемые инструменты для прототипирования аппаратуры. Однако использование существующих ADL-решений не позволяет получать инструменты качества, достаточного для разработки реальных программ, из-за неточности получаемой модели, невысокой скорости работы и ограниченной функциональности получаемых инструментов.

Метод / Характеристика	На основе HDL-языков (VHDL, Verilog, SystemC)	На основе ADL-языков (nML, ISDL, EXPRESSION, …)	На основе языков программирования общего назначения (C, C++, …)
Уровень детализации описания модели аппаратуры	Детальное описание точной структуры на уровне регистровых передач (RTL).	Высокоуровневое описание на уровне системы команд и иерархии памяти с элементами структурного описания в некоторых языках.	Специальных возможностей для описания аппаратуры нет, но общие возможности языков программирования позволяют задать модель на любом уровне.
Поддержка спецификации расширяемых архитектур	Только на уровне модулей	Отсутствует (кроме EXPRESSION, но и там без возможности разделения описаний)	Специальных возможностей нет, но можно реализовать вручную.
Внесение изменений в спецификацию аппаратуры на уровне системы команд	Трудоемко из-за структурного характера описания (система команд явно не описывается)	Относительно легко и согласованным образом. На основе измененной спецификации инструменты обновляются автоматически.	Трудоемко и чревато несогласованностями, так как одно и то же свойство (например, код операции) может быть запрограммировано в нескольких местах разных инструментов.
Построение симулятора	Генерируется автоматически	Генерируется автоматически	Программируется вручную
Точность симуляции	Потактово-точная	Потактовая точность обычно не достигается	Возможно достижение потактовой точности
Скорость симуляции	Низкая (~0,05 – 0,1 MIPS)	Средняя (~1-5 MIPS)	Высокая (~10-20 MIPS)
Построение ассемблера,дисассемблера, компоновщика, отладчика и т.д.	Невозможно из-за отсутствия необходимой информации в спецификации	Генерируются автоматически. Однако функциональность и скорость работы, как правило, невысокого уровня.	Программирование всех необходимых инструментов вручную. Возможно достижение высокого качества по скорости работы и функциональности.

Таблица 1. Характеристики языков описания аппаратуры и соответствующих методов построения кросс-инструментов.

3. Ручная реализация необходимых кросс-инструментов на языках программирования общего назначения на сегодняшний день является

практически единственным методом построения полного набора кросс-инструментария производственного качества (с высокой скоростью работы, достаточной точностью модели и удобной функциональностью, адаптированной под конкретную аппаратуру). Однако этот способ практически исключает возможность использовать такой кросс-инструментарий на этапе прототипирования аппаратуры из-за высокой трудоемкости и временны*ы*х затрат для создания начальной версии инструментария и для последующих циклов его обновления для отражения различных вариаций проектируемой аппаратуры.

Отдельно стоит отметить, что ни один из указанных методов явно не поддерживает расширяемые архитектуры в смысле обеспечения разделения технической ответственности и интеллектуальной собственности в получаемых для полной системы инструментах между производителями отдельных компонентов.

Область применения	**На основе HDL-языков**	**На основе ADL-языков**	**На основе C, C++, …**
Для прототипирования аппаратуры	Ограничено	Эффективно	Практически невозможно
Для производственной разработки прикладных программ	Практически невозможно	Ограничено	Эффективно

Таблица 2. Эффективность применения кросс-инструментов, получаемых рассмотренными методами на основе различных языков.

Таким образом, наиболее близким (хотя и с существенными недостатками) из рассмотренных методов построения кросс-инструментов для решения поставленных задач является метод автоматической генерации кросс-инструментов на основе спецификации на некотором ADL-языке. Рассмотрим более подробно сравнительные характеристики описанных выше ADL языков и соответствующих инструментальных средств поддержки (см. табл. 3).

Таким образом, ни один из существующих ADL-языков не позволяет описывать потактово-точные модели аппаратуры, включающие в себя все типичные для встраиваемых архитектур возможности: конвейер, прерывания, циклы с нулевой задержкой, периферию, зависимости между командами и отдельными операндами (для диагностики конфликтов). Поддержка расширяемой аппаратуры присутствует только в EXPRESSION, но и там соответствующие описания могут быть созданы только в рамках единой спецификации всей системы, что не позволяет разделять описания отдельных компонентов. Кроме того, ни одно из существующих инструментальных решений на основе ADL-языков не поддерживает генерацию полного набора

кросс-инструментов, а скорость работы и удобство использования отдельных получаемых компонентов остаются на низком уровне.

Характеристика	**nML**	**ISDL**	**EXPRES-SION**
Общие возможности			
Описание синтаксиса ассемблера и бинарного кодирования команд	+	+	-
Описание потактового поведения команд	-	-	неявно
Описание иерархии памяти	+	+	+
Описание структуры функциональных модулей	-	-	+
Поддержка описания различных особенностей аппаратуры			
Машинное слово переменной длины	-	+	-
Межкомандные зависимости	-	+	?
Ограничения на комбинации операндов команд	+	+	-
Конвейер	-	-	+
Прерывания	-	-	-
Циклы с нулевой задержкой	-	-	-
Сопроцессоры	-	-	+
Периферийные устройства	-	-	-
Возможности инструментальной поддержки			
Генерация симулятора	+	+	+
Потактово-точная симуляция	-	-	+-
Скорость симулятора (модельных тактов в сек.)	10^6	?	10^6
Генерация ассемблера	+	+	-
Генерация дисассемблера	+	-	-
Генерация компоновщика	+	+	-
Генерация пошагового символьного отладчика	-	-	-
Интегрированная среда разработки (IDE)	-	-	-

Таблица 3. Возможности основных современных ADL-языков и соответствующих инструментальных средств поддержки.

Таким образом, ни один из существующих ADL-языков не позволяет описывать потактово-точные модели аппаратуры, включающие в себя все типичные для встраиваемых архитектур возможности: конвейер, прерывания, циклы с нулевой задержкой, периферию, зависимости между командами и отдельными операндами (для диагностики конфликтов). Поддержка расширяемой аппаратуры присутствует только в EXPRESSION, но и там соответствующие описания могут быть созданы только в рамках единой спецификации всей системы, что не позволяет разделять описания отдельных компонентов. Кроме того, ни одно из существующих инструментальных решений на основе ADL-языков не поддерживает генерацию полного набора кросс-инструментов, а скорость работы и удобство использования отдельных получаемых компонентов остаются на низком уровне.

4. Заключение

В статье была рассмотрена задача построения инструментария кросс-разработки для расширяемых встраиваемых систем. Показана важность раннего решения этой задачи уже на этапе проектирования аппаратуры для обеспечения тонкой оптимизации проектных решений в процессе прототипирования системы, для верификации HDL-моделей и, наконец, для разработки реальных целевых программ. Были сформулированы требования к «идеальному» методу создания кросс-инструментов, и в призме этих требований были рассмотрены и проанализированы существующие средства описания аппаратуры, пригодные для автоматизированного построения кросс-инструментов на их основе. Выделено три класса таких средств – HDL языки синтезируемого описания аппаратуры (VHDL, Verilog, SystemC), ADL-языки (nML, ISDL, EXPRESSION) и языки программирования общего назначения (C/C++).

К сожалению ни одно из существующих решений не удовлетворяет сформулированным требованиям, не позволяя эффективно строить кросс-инструментарий с необходимыми свойствами. Именно поэтому, по мнению автора, перспективным направлением является исследование и разработка новых методов автоматизированного построения кросс-инструментов для расширяемых встраиваемых систем на основе комбинированных описаний моделей аппаратуры. В частности, интересным направлением кажется объединение преимуществ высокоуровневого описания системы команд на языках типа ADL и эффективного описания деталей управляющей логики и периферии на языках программирования общего назначения. Такая комбинации позволила бы получать кросс-инструментарий с достаточными скоростью работы и точностью моделирования, обеспечивая при этом возможность быстрого внесения согласованных изменений для отражения различных вариаций аппаратуры, возникающих в процессе проектирования (как на уровне изменений ядра, так и на уровне изменений расширений и их состава в полной системе). Именно такой кросс-инструментарий был бы

пригоден для эффективного решения поставленных задач прототипирования встраиваемой системы, верификации HDL-моделей и разработки реальных программ.

Литература

[1] Я.А. Хетагуров. Из истории развития специализированных бортовых вычислительных машин. http://www.computer-museum.ru/histussr/special.htm.

[2] В.В. Липаев. Из истории развития отечественной вычислительной техники для военных систем управления в реальном времени. http://www.computer-museum.ru/histussr/16.htm.

[3] К. Колпаков. История развития авиационных бортовых цифровых вычислительных машин в России. http://www.computer-museum.ru/histussr/stpc.htm.

[4] http://en.wikipedia.org/wiki/Apollo_Guidance_Computer.

[5] А. Бухтев. Проектирование встроенных систем: от концепции до кристалла. Журнал «Электронные компоненты», 2007, №1.

[6] A. Parker, etc. System-Level Design. The VLSI Handbook—2nd ed. CRC Press, 2007.

[7] Donald R. Cottrell. Design Automation Technology Roadmap. The VLSI Handbook—2nd ed. CRC Press, 2007.

[8] Software-Hardware Codesign. // IEEE Design & Test of Computers, January-March 2000. pp.92-99.

[9] M L Vallejo, J C Lopez, ”On the hardware-software partitioning problem: System Modeling and partitioning techniques”, ACM TODAES, V-8, 2003.

[10] K Ben Chehida, M Auguin, ”HW/SW partitioning approach for reconfigurable system design”, CASES 2002.

[11] J Henkel, R Ernst, ”An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation Techniques”, IEEE Transactions on VLSI, V-9, 2001.

[12] R Ernst, J Henkel, T Benner. ”Hardware-software co-synthesis for microcontrollers”, IEEE Design and Test,V-10, Dec 1993.

[13] Wayne Wolf. Embedded Computing Systems and Hardware/Software Co-Design. The VLSI Handbook—2nd ed. CRC Press, 2007.

[14] Embedded C. Стандарт ISO/IEC TR 18037:2004.

[15] Executable and Linking Format in Wikipedia. http://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

[16] Generic ELF Specification. http://www.linux-foundation.org/spec/book/ELF-generic/ELF-generic/book1.html.

[17] M. Hartoog, J. Rowson, P. Reddy. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. Design Automation Conference (DAC) 1997.

[18] Lin Yung-Chia. Hardware/Software Co-design with Architecture Description Language. Programming Language Lab. NTHU. 2003.

[19] Д.Ю. Булычев. Разработка программно-аппаратных систем на основе описания макроархитектуры. Сборник Системное программирование. Санкт-Петербург, 2004.

[20] Z. Navabi. Languages for Design and Implementation of Hardware. The VLSI Handbook—2nd ed. CRC Press, 2007.

[21] А.К. Поляков. Языки VHDL и VERILOG в проектировании цифровой аппаратуры. // М.: Солон-Пресс, 2003. 320 с.
[22] П.Н. Бибило. Синтез логических схем с использованием языка VHDL. // М.: СОЛОН-Р, 2002. 384 с.
[23] Volnei A. Pedroni. Circuit Design with VHDL. // MIT Press, 2004.
[24] IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-1987.
[25] IEEE Standard Multivalue Logic System for VHDL Model Interoperability. IEEE Std 1164.
[26] IEEE Standard VHDL Synthesis Packages. IEEE Std 1076.3-1997.
[27] M. Rofoue, Z. Navabi. RT Level Hardware Description with VHDL. The VLSI Handbook—2nd ed. CRC Press, 2007.
[28] Weng Fook Lee. Verilog Coding for Logic Synthesis. // John Wiley & Sons, 2003.
[29] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language. IEEE Std 1364-2005.
[30] Z. Navabi. Register Transfer Level Hardware Description with Verilog. The VLSI Handbook—2nd ed. CRC Press, 2007.
[31] Open SystemC Initiative. http://www.systemc.org.
[32] IEEE Standard System C Language Reference Manual. IEEE Std 1666-2005.
[33] S. Mirkhani and Z. Navabi. Register-Transfer Level Hardware Description with SystemC. The VLSI Handbook—2nd ed. CRC Press, 2007.
[34] Stephen Bailey. Comparison of VHDL, Verilog and SystemVerilog. Model Technology White Paper.
[35] Synopsys VCS http://www.synopsys.com/products/simulation/simulation.html http://www.synopsys.com/products/simulation/vcs_ds.pdf
[36] Mentor Graphics ModelSim. http://www.mentor.com/products/fv/digital_verification/index.cfm.
[37] Cadence NC-Sim. http://www.cadence.com/datasheets/4492C_IncisiveVerilog_DSfnl.pdf
[38] Cadence Incisive Simulators. http://cadence.com/products/functional_ver/simulation/index.aspx
[39] P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. // IEEE Proceedings Computers and Digital Techniques., Vol. 152, No. 3, May 2005.
[40] W. Qin, and S. Malik. Architecture description languages for retargetable compilation. // The Compiler Design Handbook, CRC Press, 2002.
[41] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, A. Nicolau. Architecture Description Languages for Systems-on-Chip Design. // Proc. Asia Pacific Conf. on Chip Design Language, 1999, pp. 109–116.
[42] Rainer Leupers. Retargetable Code Generation for Digital Signal Processors. Kluwer Academic Publishers, 1997.
[43] M. Freericks. The nML Machine Description Formalism. Technical Report 1991/15, TU Berlin, Fachbereich Informatik, 1991.
[44] A. Fauth, J. Van Praet, M. Freericks. Describing instruction set processors using nML. In Proc. of ED&TC, 1995.
[45] Chess/Checkers Products. Target Compiler Technology. http://www.retarget.com/.
[46] ISDL Project Homepage. http://caa.lcs.mit.edu/caa/home.html.
[47] G. Hadjiyannis, S. Hanono, S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. Design Automation Conference (DAC) 1997.
[48] EXPRESSION Homepage. http://www.cecs.uci.edu/~aces/index.html.

[49] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt and Alex Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability, DATE 99.
[50] P. Mishra, A. Shrivastava, N. Dutt. ADL-driven Software Toolkit Generation for DSE. ACM Transactions on Design Automation of Electronic Systems, pp. 1-31, 2006.
[51] M. Reshadi, N. Dutt, P. Mishra. A Retargetable Framework for Instruction-Set Architecture Simulation. ACM Transactions on Embedded Computing Systems, Vol. 5, No. 2, pp. 431–452, May 2006.
[52] Analog Devices Processor Development Tools. http://www.analog.com/processors/platforms/processorDevTools.html.
[53] Texas Instruments Tools & Software Overview. http://focus.ti.com/dsp/docs/dspfindtoolswbytooltype.tsp?sectionId=3&tabId=2088&toolTypeId=1&familyId=44.
[54] Freescale CodeWarrior Development Tools. http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=012726&tid=FSH.
[55] LSI DSP Products. http://www.lsi.com/networking_home/networking_products/dsps/index.html.
[56] NXP (Philips Semiconductors) Development Tools for Microcontrollers. http://www.nxp.com/products/microcontrollers/support/development_tools/.
[57] ARM RealView Development Tools. http://www.arm.com/products/DevTools/.
[58] TASKING - Embedded Software Development Tools. http://www.tasking.com/.
[59] Raisonance Embedded Development Tools. http://www.raisonance.com/.
[60] Signum Embedded Development Tools. http://signum.com/.
[61] Nohau (ICE Technology) Development Tools. http://www.icetech.com/.
[62] Keil Embedded Development Tools. http://www.keil.com/.
[63] Green Hills MULTI Integrated Development Environment. http://www.ghs.com/products/MULTI_IDE.html.
[64] IAR Embedded Development Tools. http://www.iar.com/.
[65] iSystem Solutions for Embedded System Development. http://www.isystem.com/.