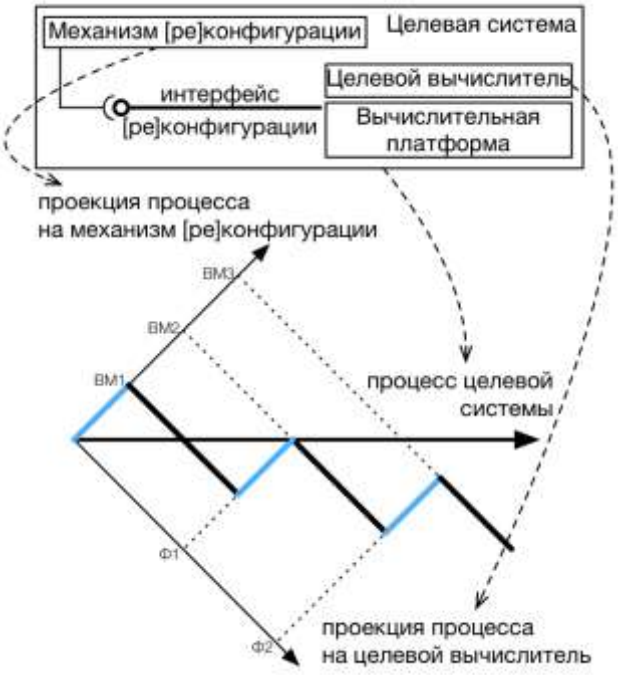
УНИВЕРСИТЕТ ИТМО

С.В. Быковский, Я.Г. Горбачев, А.О. Ключев, А.В. Пенской, А.Е. Платунов

СОПРЯЖЁННОЕ ПРОЕКТИРОВАНИЕ ВСТРАИВАЕМЫХ СИСТЕМ (HARDWARE/SOFTWARE CO-DESIGN)

Часть 1



Санкт-Петербург

2016

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

С.В. Быковский, Я.Г. Горбачев, А.О. Ключев,
А.В. Пенской, А.Е. Платунов

СОПРЯЖЁННОЕ ПРОЕКТИРОВАНИЕ ВСТРАИВАЕМЫХ СИСТЕМ (HARDWARE/SOFTWARE CO-DESIGN)

Часть 1

Учебное пособие

УНИВЕРСИТЕТ ИТМО

Санкт-Петербург

2016

Быковский С.В., Горбачев Я.Г., Ключев А.О., Пенской А.В., Платунов А.Е. Сопряжённое проектирование встраиваемых систем (Hardware/Software Co-Design). Часть 1. Учебное пособие. – СПб.: Университет ИТМО, 2016. – 108 с.

В учебном пособии представлены обзорные и оригинальные материалы по прогрессивным методологиям высокоуровневого проектирования встраиваемых вычислительных систем.

Для подготовки магистров по направлению 09.04.01 «Информатика и вычислительная техника» по программе «Проектирование встраиваемых вычислительных систем и систем на кристалле» и магистров по направлению 09.04.04 «Программная инженерия» по программе «Программное обеспечение мобильных и встраиваемых систем».

Рекомендовано к печати ученым советом мегафакультета КТиУ Университета ИТМО, протокол № 6 от 21.06.2016.

Университет ИТМО — ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО — участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО — становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Учебное пособие является введением в высокоуровневое проектирование специализированных, в частности – встраиваемых, вычислительных систем, и содержит обзорные материалы по направлению, в англоязычной литературе носящему название Hardware/Software Co-Design. В отечественной литературе нет общепринятого названия данного направления, для его обозначения используют такие термины, как «аппаратно-программный кодизайн», «кодизайн», «сопряжённое проектирование аппаратуры и программного обеспечения», «совместное проектирование аппаратуры и программного обеспечения».

Необходимость высокоуровневого проектирования подобных систем и применения новых методологий обусловлены рядом факторов. Вычислительная техника со времени создания первых программируемых ЭВМ уже проделала огромный путь и не прекращает развиваться. Современные компьютеры далеко обогнали своих предшественников, с каждым годом предоставляемые ими возможности растут. Увеличиваются вычислительные мощности отдельных процессоров, создаются всё более сложные гетерогенные вычислительные системы, объединяющие различные вычислительные ядра. Всё большее распространение получают постоянно усложняющиеся распределённые системы. Программное обеспечение также становится всё более сложным, современные продукты насчитывают миллионы строк кода – и эти цифры растут.

Одновременно, возрастают потребности в специализированных вычислительных системах. Сегодня сложно представить нашу жизнь без электроники во всех сферах человеческой деятельности. Бытовые приборы, блоки управления автомобилями, кораблями и самолётами, охранные системы, электронные ключи, прогрессивные системы вооружений, связь – везде используются специализированные компьютеры, так называемые встраиваемые системы. Фактически, в составе парка вычислительных систем они составляют абсолютное большинство.

Требования, предъявляемые к таким системам, по определению жёстче, чем к ЭВМ общего назначения. В общем случае встраиваемые системы должны быть надёжными, малогабаритными, дешёвыми, с низким энергопотреблением и с предсказуемым временем реакции на внешние воздействия (должны удовлетворять требованиям реального времени). Ввиду этого, зачастую использование существующих аппаратных решений оказывается неэффективным, и помимо написания программного обеспечения – как делается обычно в системах общего назначения – встаёт задача создания также и специализированной аппаратуры.

Таким образом, при проектировании встраиваемых систем разработчик вынужден проектировать аппаратуру и программное обеспечение с учётом множества разноплановых аспектов, и зачастую вынужден работать одновременно в нескольких областях инженерной деятельности. Условия рыночной экономики добавляют к вышеперечисленному жёсткие временные ограничения, накладываемые на процесс разработки, требования к качеству конечного продукта и необходимость учёта рисков. Немалое внимание приходится уделять также человеческим ресурсам и правильной организации команды разработчиков. Кроме того, как говорилось ранее, элементная база и потенциальные возможности постоянно совершенствуются, предоставляя всё более совершенные и эффективные средства для создания специализированных вычислительных систем.

Исходя из всего вышеперечисленного, проектирование встраиваемых систем представляет собой очень трудоёмкую и многоплановую задачу. Самым простым и очевидным подходом для борьбы со сложностью является применение шаблонных решений и подходов, заимствование готовых элементов. При этом довольно часто это производится без рассмотрения всех возможных способов реализации, а некоторые важные аспекты просто не учитываются. Подавляющее большинство встраиваемых систем сегодня создаётся именно таким путём, что очень редко позволяет создавать действительно эффективные и оптимальные решения.

Направление кодизайна объединяет многочисленные исследования в области параллельного и скоординированного проектирования программного обеспечения и аппаратуры. Рассмотрение абстрактной задачи и алгоритма её решения сначала на системном уровне, безотносительно к реализации, и последующее распределение подзадач между программными и аппаратными компонентами даёт большую свободу в поиске компромиссов, позволяет лучше настраивать систему под конкретную задачу. В контексте проектирования специализированных вычислителей термин «кодизайн» фактически используется как синоним прогрессивных направлений системного, высокоуровневого или архитектурного проектирования, и представляет собой набор подходов и инструментальных средств для решения актуальных проблем проектирования специализированных вычислительных систем.

В первой части пособия объясняются особенности встраиваемых систем и описываются проблемы их проектирования. Даётся определение киберфизических систем, перспективного класса систем автоматики. Рассматриваются понятия программного и аппаратного обеспечения и приводятся некоторые перспективные подходы к организации современных вычислителей, сочетающих в себе программируемые

микропроцессорные ядра и аппаратно реализуемые функциональные блоки. Описываются методы и типовые решения, существующие в области высокоуровневого проектирования специализированных вычислительных систем.

1 ВВЕДЕНИЕ В ПРОЕКТИРОВАНИЕ ВСТРАИВАЕМЫХ СИСТЕМ

1.1 Встраиваемые и киберфизические системы

Изначально, электронные вычислительные машины (ЭВМ) создавались как инструмент для выполнения громоздких и трудоёмких вычислений. Однако с течением времени сфера применения ЭВМ расширилась на многие другие области человеческой деятельности, поддающиеся формализации и алгоритмическому описанию. В том числе, в 60-е годы двадцатого века появились так называемые информационно-управляющие системы (ИУС), функцией которых стало управление реальными объектами.

В конце 70-х годов, благодаря появлению интегральных схем и микропроцессоров, стало возможным приближать ИУС непосредственно к объектам управления или даже встраивать их в эти объекты. Так появилось новое понятие — встраиваемые, или встроенные, системы (ВсС). Бурное развитие вычислительной техники в течение последних десятилетий, заключавшееся, в частности, в миниатюризации электронных устройств и в росте их вычислительных мощностей, привело к увеличению количества подобных систем. Фактически, ВсС теперь представляют собой абсолютное большинство среди всех существующих вычислительных систем (ВС).

В литературе нет общепризнанного и окончательного мнения, что можно относить к ВсС, а что — нет, и границы этого понятия довольно размыты. Достаточно сравнить некоторые примеры определений из литературы [1]:

«Встроенной системой можно считать любую вычислительную систему, которая не является ПК, портативным компьютером (laptop) или большим универсальным компьютером (mainframe computer)».

«Устройство, которое включает в себя программируемый компьютер, но не является при этом компьютером общего назначения».

«Сложно определить. Практически любая вычислительная система, не являющаяся настольным компьютером».

«Система обработки информации, встроенная в какой-либо продукт».

Наиболее обще ВсС можно определить, как специализированную ВС, которая в силу решаемой задачи непосредственно взаимодействует с физическими объектами и процессами [2].

Следующим после появления ВсС этапом эволюции ИУС и их естественным развитием стали пространственно-распределённые встраиваемые системы, в которых связи между компонентами являются

слабыми, или, иначе говоря, интенсивность обмена данными в рамках одного вычислительного процесса значительно выше интенсивности обмена данными между разными вычислительными процессами.

Такие системы стали доступными по мере удешевления элементной базы, увеличения степени её интеграции, а также роста надёжности вычислительных устройств. Благодаря перечисленным факторам появилась возможность устанавливать отдельные, относительно независимые ЭВМ в разные места объекта управления, объединяя все вычислительные узлы в единую контроллерную сеть.

При этом многие современные ВсС по факту являются распределенными, и в зарубежной научно-технической литературе достаточно четко декларируется, что ВсС – это не только малогабаритные или моноблочные, одномодульные устройства, но и подобные пространственно и/или архитектурно распределенные системы, которые непосредственно сопряжены с объектами большого масштаба, пространственно распределенными объектами и т.д.

Диапазон реализаций ВсС очень велик. В него попадают и простые устройства, выполняющие какие-либо элементарные функции, и сложнейшие распределенные иерархические системы, в том числе и управляющие критически важными объектами. ВсС находят широкое применение в бытовой электронике, промышленной автоматике, на транспорте, в телекоммуникационных системах, медицинском оборудовании, в военной и аэрокосмической технике, в других областях. Сфера применения ВсС постоянно расширяется и в том или ином виде эти системы в ближайшее время проникнут во все области деятельности человека.

При этом вопреки широкой распространённости ВсС, основным объектом изучения науки в компьютерной области долгое время была обработка информации в стандартных ВС общего назначения. Только относительно недавно ВсС получили достаточно внимания со стороны исследователей. И только недавно обнаружилось, что технологии и инженерные методы, которые требуются для проектирования и анализа таких систем, сильно отличаются от тех, что применяются для систем общего назначения.

Несмотря на то что ВсС использовались с 1970х, в течение основной части своей истории они рассматривались просто как маленькие компьютеры. Принципиальная инженерная проблема понималась как необходимость справляться с ограниченными ресурсами (ограниченная вычислительная мощность, ограниченные энергетические ресурсы, малый объём памяти и т.д.), исходя из чего, основной задачей была типовая оптимизация. Так как любые изделия и проекты выигрывают от своей

оптимизации, направление ВсС чётко не выделялось в компьютерной науке, просто предполагалось, что требуется более агрессивное использование обычных оптимизирующих техник.

Недавно, как утверждают некоторые авторы [3], появилось понимание, что принципиальные проблемы во встроенных системах происходят из-за их взаимодействия с физическими процессами, а не из-за ограниченных ресурсов. В 2006 году был введён термин «киберфизические системы» (CPS), подразумевающий интеграцию вычислений с физическими процессами. Понятие отражает ожидающийся в ближайшем будущем качественный переход в восприятии ВсС и методов их проектирования. Суть CPS состоит в том, что проектирование объекта управления и системы управления для этого объекта должны выполняться в едином ключе, в едином комплексе, тесно взаимодействующих инструментальных средств [1]. Кроме того, концепция CPS, как правило, предполагает объединение в сеть с другими устройствами, тесное взаимодействие и обмен информацией с этими устройствами.

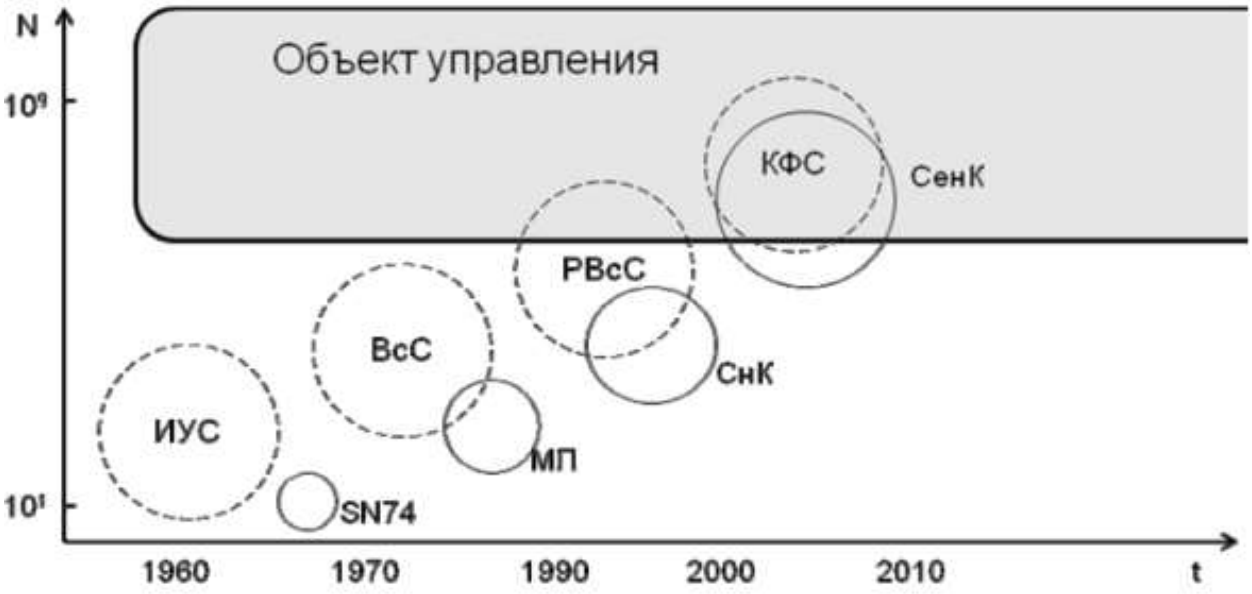


Рис. 1. Эволюция понятия ИУС. ВсС – встраиваемые системы, МП – микропроцессоры, РВсС – распределённые встраиваемые системы, СнК – системы на кристалле, КФС – киберфизические системы, СенК – сети на кристалле, N – степень интеграции [3].

Таким образом, с течением времени эволюция понятия «встраиваемые вычислительные системы» менялась следующим образом (Рис. 1):

1) Информационно-управляющие системы, 60-е годы (УВК, БЦВМ).
2) Встроенные вычислительные системы (embedded systems) – ВсС (ES), конец 70-х годов.

3) Распределенные встроенные системы управления (networked embedded control systems / распределенные информационно-управляющие системы) – NECS / РИУС, конец 90-х годов.
4) Cyber Physical Systems – CPS (киберфизические системы), примерно с 2006г.

Сам по себе термин кибернетика (от др.-греч. κυβερνητική — «искусство управления») обозначает науку об общих закономерностях получения, хранения, передачи и преобразования информации в сложных управляющих системах, будь то машины, живые организмы или общество. Термин был введён Норбертом Винером, американским математиком, который во время Второй Мировой войны впервые применил технологию для автоматического прицеливания и стрельбы из зенитных орудий. Виннер описывал своё видение кибернетики как объединение управления и коммуникаций с управлением на основе обратной связи. Хотя механизмы, которые он использовал, не включали в себя вычислительные машины, принципы аналогичны использующимся сейчас в компьютерных системах управления.

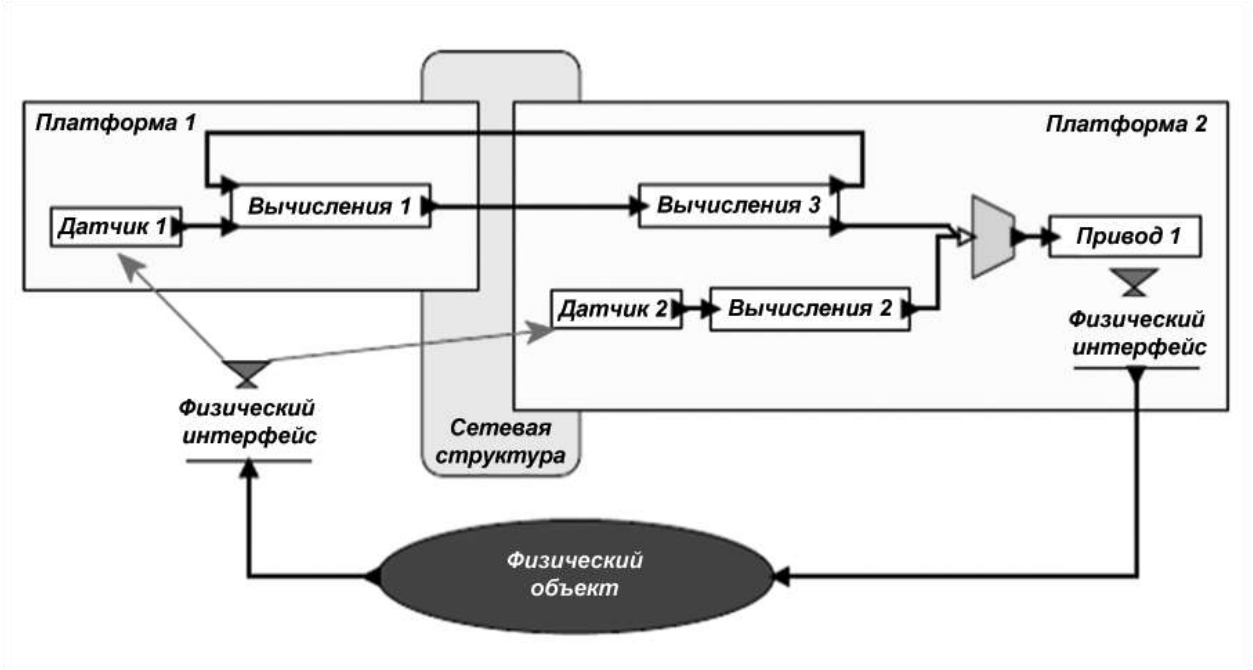


Рис. 2. Пример структуры киберфизической системы [3].

На рисунке (Рис. 2) приведён пример типичной структуры CPS. Она состоит из объекта управления, «физической» части — той части, которая не реализуется вычислительной техникой или цифровыми сетями, и может включать механику, биологические или химические процессы, даже людей-операторов. Управление этой частью CPS осуществляют одна или несколько платформ, которые состоят из датчиков, исполнительных механизмов, одного или нескольких процессоров и, возможно, одной или

нескольких операционных систем. Сетевая структура обеспечивает механизмы взаимосвязи платформ, образуя вместе с ними «кибернетическую» часть CPS.

1.2 Особенности проектирования встраиваемых систем

Изучение и проектирование ВсС влечёт за собой некоторые определенные проблемы, которые являются редкими в универсальных вычислениях и обычных настольных компьютерах. Например, в универсальном программном обеспечении время, которое требуется, чтобы выполнить задачу, является всего лишь показателем производительности, а не корректности написания ПО. Не является некорректным, если больше временных ресурсов тратится на решение задачи — это просто менее удобно.

В ВсС время, которое требуется, чтобы выполнить задачу, может быть критичным для правильного функционирования системы, ведь происходящие в реальном мире физические процессы редко являются процедурными, и промедление в обработке каких-то событий для критических областей может повлечь непоправимые последствия. Принципиальное отличие информационных систем от систем реального времени, к которым, в большинстве случаев, относятся ВсС, состоит в трактовке параметра «реакция вход-выход»: «The right answer late is wrong» («Правильный ответ поздно – неправильный»).

Функции ВсС, как и любой системы – будь то биологический организм, человеческое общество или искусственно созданный с какой-либо целью объект – можно свести к некоей основной базовой задаче или набору таковых. Система должна успевать обрабатывать изменения значений на своих входах и выставлять актуальные значения управляющих выходов в соответствии с заложенными алгоритмами, влияя тем самым на окружающую среду и подчинённый объект (объекты) управления. Таким образом, осуществляется управление динамикой процессов в окружающей среде по принципу обратной связи: за счёт измерений состояния контролируемых процессов и организации действий, которые влияют на данные процессы через некоторые исполнительные механизмы.

Кроме того, может присутствовать необходимость одновременного выполнения нескольких действий, каждое из которых является важным – ведь процессов, которые необходимо контролировать, часто бывает больше одного. Исходя из этого – параллелизм внутренне присущ ВсС. Подход, когда каждая задача ждёт своей очереди, и производится простой последовательный перебор по порядку, в большинстве случаев неприемлем в алгоритмах управления.

Исходя из этого, при проектировании ВсС необходимо добиваться максимально быстрого и параллельного с точки зрения внешней среды

поведения ВС, несмотря на последовательную природу большинства современных компьютеров. Желательно, чтобы ВсС функционировала так, как будто является полностью параллельной системой. Многие из технических проблем при разработке и анализе встроенного программного обеспечения микропроцессорных систем как раз и происходят вследствие потребности соединять последовательные вычисления с непрерывно и одновременно изменяющимися во внешнем мире процессами.

Но временные характеристики не являются единственной отличительной способностью ВсС. Зачастую такие системы применяются в ответственных областях, что влечёт за собой повышенные требования к надёжности и всесторонней защите проектируемых устройств. Если выходит из строя бортовой компьютер автомобиля или самолёта, медицинское оборудование, блок управления лифтом, светофором или каким-нибудь станком – это может вызвать причинение вреда людям, окружающей среде, чьему-нибудь имуществу. Даже зависающий телефон может повлечь за собой отказ потребителей от его приобретения, и, как одно из вероятных последствий – банкротство выпускающей данные телефоны фирмы.

Поэтому для обеспечения достаточной степени надёжности работы устройств нужно учитывать множество факторов: как природные, естественные воздействия, так и последствия человеческой деятельности. Причём последнее включают в себя случайные факторы и целенаправленные вредоносные воздействия – вандализм, попытки несанкционированного доступа, взлома устройств. Учитывая, что набирающий силу киберфизический подход к созданию ВсС подразумевает в большинстве случаев подключение к сети интернет, а также то, что понятие «интернет вещей», когда каждое устройство подключается к глобальной сети и становится доступным извне, уже становится реальностью – появляется всё больше мишеней для вероятных атак извне и всё острее встаёт проблема обеспечения безопасности.

Защита от естественных внешних воздействий предполагает в разных ситуациях меры по сохранению работоспособности в условиях повышенной влажности, повышенной и/или пониженной температуры, при вибрациях, ударах, мощных электромагнитных помехах и т.д. – набор неблагоприятных факторов может меняться в зависимости от конкретной задачи. Как правило, противодействие внешним условиям предполагает стандартные меры, например, обеспечение защиты корпусов от влаги, придание им (и всем находящимся внутри деталям) достаточной механической прочности, надёжного крепления элементов между собой и т.д. Однако, зачастую, подобными мерами пренебрегают, что влечёт за собой выход из строя некачественно спроектированных систем.

Кроме внешних факторов, влияющих на работу системы уже во время её функционирования, обязательно следует учитывать и возможность возникновения ошибок во время процесса создания продукта. Любой, даже самый опытный инженер, может допускать ошибки, и чем сложнее изделие, тем их вероятность больше. Это могут быть баги в коде, неправильная трассировка дорожек на плате, неверно подобранные электронные элементы, и не в последнюю очередь концептуальные ошибки всего проекта в целом. Бывают ошибки в инструментариях для разработки и в поставляемых производителями библиотеках, закупленные электронные компоненты могут оказаться бракованными. Проблемы могут возникать на этапе сборки устройств, при их монтаже на объектах и т.д.

Поэтому процессу создания ВсС в широком смысле – учитывающем сначала разработку, затем изготовление – обычно сопутствуют процессы верификации и тестирования. Первые, опытные образцы, проверяются на соответствие требованиям технического задания. Потом при налаженном производстве производится уже лишь быстрая проверка работоспособности новых изделий по некоторым характерным признакам.

Концептуально, сама по себе задача обеспечения надёжности и защищённости неотделима и напрямую следует из задачи обеспечения функциональности вычислительной системы, т.е. того, что обычно прописывается в техническом задании явно. Грубо говоря – система должна работать. Какой бы ни было её предназначение, в чём бы не проявлялась эта работа – управление неким объектом, сбор информации с датчиков, просто мигание лампочками – система должна штатно функционировать во всём допустимом диапазоне внешних воздействий, не только в идеальных условиях. Если удалось добиться правильного поведения системы и заставить её сработать один раз, это не значит, что получено работоспособное устройство, удовлетворяющее требованиям.

Известное и естественное свойство человеческого разума заставляет нас видеть желаемое там, где его на самом деле нет. Часто экспериментальные данные трактуются в пользу теории, при этом на несоответствия закрываются глаза. Несмотря на это, оценка результатов работы – собственной или коллектива – должна быть максимально критичной и беспристрастной. Следует повторять опыты снова и снова, меняя исходные данные и внешние условия, тщательно фиксируя отклонения от нормы, чтобы убедиться в том, что полученное изделие полностью работоспособно.

Также, если говорить о надёжности, всегда стоит помнить: вероятность безотказной работы системы (в случае если для работоспособности всей системы в целом требуется работоспособность

всех составляющих, что чаще всего верно для BcC) равна произведению вероятностей безотказной работы каждого её элемента.

Это влечёт за собой два следствия. Первое – то, что максимально надёжное является максимально простым. Нет более безотказного инструмента, чем лом. При любом наращивании функциональности – за счёт увеличения объёма кода, добавления новых элементов и т.д. – одновременно увеличивается вероятность отказа. Так, например, в ПО, применяющемся в некоторых классах устройств для критических областей (например, в авиационном оборудовании), запрещается использование операционных систем, как лишних ненадёжных элементов, потенциальных источников отказов. Это не значит, что вести наращивание функциональности нельзя принципиально – сложные и достаточно надёжные системы создаются за счёт резервирования и добавления избыточности разными способами и на разных уровнях. Но это является самостоятельной и очень серьёзной задачей, что должно ясно пониматься и обязательно учитываться в планировании процесса проектирования.

Второе следствие – то, что BcC, состоящие из большого количества тесно взаимодействующих «элементов» (в качестве которых можно выделить ПО, в том числе операционные системы, библиотеки и драйвера, интегральные схемы, все электронные элементы, устанавливаемые на платах, со связями между ними, датчики, исполнительные механизмы, соединительные кабели и т.д.), априори имеют большой потенциал для возникновения ошибок при проектировании и отказов – уже в процессе работы.

Как правило, достаточно сложными являются уже хотя бы отдельно ПО и аппаратная часть, при их комбинировании – идёт взрывной рост сложности. Даже при полностью отлаженных отдельно ПО и аппаратуре, в их совместной работе могут возникать непредвиденные эффекты, вызванные взаимодействием этих базовых элементов. Ко всему этому ещё добавляются завязки на объект управления, влияние внешней среды, механические и другие составляющие, если они имеются.

Зачастую проявляются связи между, казалось бы, совершенно не связанными явлениями, которые невозможно или очень сложно предугадать заранее. Хорошим тоном считается после любого незначительного изменения в какой-то сложной системе, добавления хоть сколь угодно независимого и самостоятельного объекта – заново проводить полное тестирование всей системы в целом. Ярким примером может служить Linux-сообщество, где можно проследить подобные процессы: почти после каждого обновления системы обнаруживаются баги, проявляющиеся при редких сочетаниях аппаратуры и ПО, в сложно моделируемых ситуациях и т.д.

Рис. 3. Взаимное влияние компонентов ВсС и внешних факторов, сказывающееся на сложности и надёжности.

Так как ВсС, согласно определению и назначению, часто встраиваются в объекты управления – на них чаще всего накладываются многочисленные ограничения. Например, в качестве дополнительных условий могут быть заданные максимальные размеры или даже форма исполнения конструкции устройства, температурные режимы работы, максимально допустимое энергопотребление и т.д. Снижение энергопотребления вообще является одним из наиболее часто встречающихся пунктов многокритериальной оптимизации, которая производится при проектировании ВсС.

Помимо этого, всегда стоит задача минимизации стоимости производства конечных устройств. Так как ВсС в большинстве случаев являются не стандартными, а заказными, производящимися на заказ устройствами, снижение стоимости комплектующих, технологий изготовления, накладных расходов на проектирование напрямую влияют на стоимость и, следовательно, конкурентоспособность конечных устройств, выставляемых на рынок или предоставляемых заказчику.

Так как проектирование ВсС, как правило, производится силами некоторого ограниченного коллектива специалистов, занимающегося

решением достаточно похожих и близких по смыслу задач, крайне желательным также является повышение степени повторного использования решений. Аккумуляция опыта и наработок позволяет ускорить разработку будущих продуктов, но требует накладных расходов на создание документации и пригодных для повторного использования IP-блоков, модулей, архитектурных и других решений.

Сама по себе разработка ВсС представляет собой результат междисциплинарного проектирования, в котором условно можно выделить три основных составляющих [2]:

- Этап решения задачи на прикладном уровне, когда необходимо найти правильные методы и алгоритмы без деталей реализации. Это сфера деятельности прикладных специалистов из соответствующих областей (физика, энергетика, медицина, лингвистика, биология и др.).
- Процесс проектирования, в ходе которого требуется отобразить полученное прикладное решение на технологическую базу информатики и вычислительной техники (ВТ). Это работа специалистов из области информатики, сегодня все чаще ее называют архитектурным, высокоуровневым или системным проектированием.
- Фаза реализации, в ходе которой инженеры, программисты и прикладные специалисты обеспечивают выполнение ранее сформулированных требований, таких как необходимая функциональность, динамика поведения, надежность и безопасность функционирования, габариты, энергопотребление, стоимость и технологичность при тиражировании.

После этапов разработки следуют обязательные шаги по верификации и валидации. Иногда они постоянно сопутствуют процессу проектирования, позволяя оценивать промежуточные результаты до получения конечного продукта, что позволяет вовремя отследить отклонения от нормы и допускаемые в процессе ошибки. Помимо этого, в процесс проектирования обязательно входят шаги по анализу возможных решений

После этапов разработки следуют обязательные шаги по верификации и валидации. Иногда они постоянно сопутствуют процессу проектирования, позволяя оценивать промежуточные результаты до получения конечного продукта, что позволяет вовремя отследить отклонения от нормы и допускаемые в процессе ошибки. Помимо этого, в процесс проектирования обязательно входят шаги по анализу возможных решений.

Таким образом, разработка ВсС требует обширных знаний в разных областях – понимания вычислительных систем, программного обеспечения, сетей и физических процессов, умения оценивать их совместную динамику. Чтобы заставить ВсС работать с физическими процессами, требуется технически сложное, низкоуровневое проектирование. Разработчики встраиваемого ПО, если система реализуется на микропроцессорной базе, вынуждены бороться с контроллерами прерываний, архитектурами памяти, программированием на уровне ассемблера (для использования специализированных функций или для точного контроля времени выполнения), разработкой драйверов устройств, сетевых интерфейсов и политик планирования. Разработчики аппаратуры должны учитывать многочисленные ограничения, контролировать реальные задержки сигналов, огромное количество ресурсов тратится на верификацию полученных продуктов. Кроме того, очень важно иметь четкое представление о предмете проектирования, доступных методах и средствах его создания. На системном уровне требуется контролировать множество факторов, координировать усилия отдельных разработчиков, что тоже является отдельной и важной задачей.

Разнообразие задач автоматизации и способов их решения порождает огромное число вариантов ВсС. С учетом существующих технических ограничений и выделяемых финансово-временных бюджетов, выбор варианта реализации может превращаться для разработчика в сложную научно-техническую задачу. Поиск оптимальных вариантов реализации, называемый исследованием пространства проектных решений, может занимать значительную часть времени разработки.

Проектируя ВсС, разработчик всегда создает *специализированную вычислительную систему* независимо от степени соотношения готовых и заново создаваемых решений. В сферу его анализа попадают все уровни организации системы. Он имеет дело не с созданием приложения в готовой операционной среде при наличии мощных и удобных инструментальных средств, а с созданием новой специализированной ВС в условиях жестких ограничений самого разного плана.

1.3 Современные проблемы проектирования

В настоящее время многими специалистами отмечается постоянный *рост сложности современных ВсС* ([4–6]), с чем связана одна из основных проблем проектирования подобных систем. Одной из причин этого являются всё возрастающие требования к функциональности и характеристикам проектируемых электронных изделий, другой — развитие элементной базы, предоставляющей все больше возможностей по интеграции и потенциально доступным вычислительным мощностям.

Рис. 4. Увеличение количества транзисторов на кристалле [7].

Одновременно с увеличением количества транзисторов на кристаллах интегральных схем уменьшается стоимость каждого отдельного транзистора (Рис. 5). Благодаря этому постоянно растёт доступность сложных аппаратных решений. Сегодня уже стали реальностью гетерогенные СнК с интегрированными на одном чипе микропроцессорами различных типов, массивами памяти, цифровыми и аналоговыми IP-блоками.

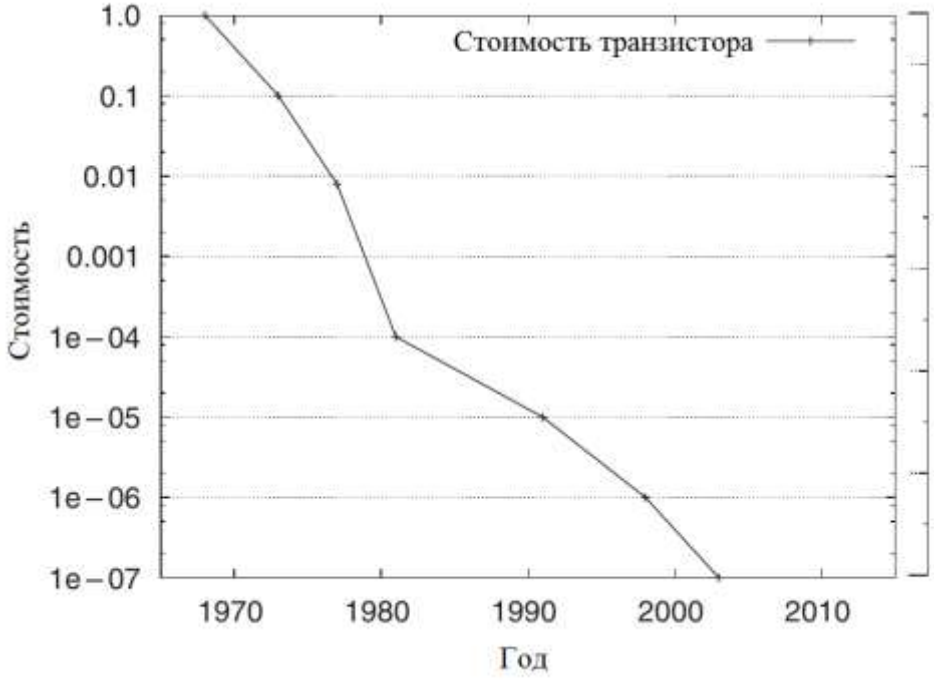


Рис. 5. Уменьшение стоимости транзистора [7].

Согласно так называемому «Закону Мура», наблюдению, изначально сделанному в 1965 году и остававшемуся верным на протяжении последних пятидесяти лет, количество размещаемых на кристалле транзисторов удваивается каждые 2 года. Некоторые специалисты предрекают достижение предела, когда этот закон перестанет действовать — из-за ограничений на уменьшение физических размеров, роста энергопотребления, невозможности отвода тепла, неоправданных затрат на организацию производства и прочих факторов. Но тем не менее на данный момент этот процесс идёт именно с предсказанной скоростью (Рис. 4).

Однако при таком обилии возможностей фактически используется лишь малая часть доступного потенциала (Рис. 6). Реальный прирост производительности вычислений не пропорционален увеличению количества транзисторов. Человечество просто не научилось пока справляться с задачами по созданию настолько сложной аппаратуры, чтобы использовались одновременно и эффективно все доступные ресурсы.

Рис. 6. Кризис сложности [8].

Ведущие производители микропроцессоров стремятся использовать достижения полупроводниковых технологий за счёт выпуска многоядерных процессоров, количество вычислительных узлов в которых постоянно растёт, всё больше утилизируя предоставляемые возможности (Рис. 7). Подобный подход позволяет использовать большое количество элементарных логических ячеек и реализует крупно-гранулярный параллелизм уровня процессов. При этом действительно эффективное использование полученных вычислительных мощностей требует также полного пересмотра подхода к программированию, без которого большинство ядер оказываются бесполезным и излишним балластом. По

сути, многопроцессорные системы являются всего лишь потенциальной возможностью, они никак не гарантируют, что разработчики ПО смогут её использовать.

К тому же применяемые в современных процессорах различные вариации фон-неймановской архитектуры имеют врождённый недостаток — так называемое «бутылочное горлышко архитектуры фон-Неймана». Он заключается в том, что даже при высокой производительности процессора его результирующая пропускная способность сильно ограничивается гораздо более медленным доступом к памяти. В случае наращивания количества ядер вопрос доступа к памяти не исчезает, а, наоборот, встаёт ещё более остро.

Максимально полно ресурсы кристалла можно было бы использовать при создании заказных микросхем, реализующих параллельные — например, так называемые потоковые — модели вычислений. Этот подход позволил бы достичь высокого параллелизма вычислений и доступа к памяти, а следовательно — и большой производительности. Но значительная часть работы по проектированию подобных микросхем до сих пор производится вручную, несмотря на обилие инструментов по автоматизации проектирования. За счёт этого, такие вычислители имеют некоторые ограничения из-за растущей с увеличением размера проектов сложности, что проявляется даже при условии интенсивного повторного использования готовых блоков, и из-за высокой стоимости разработки.

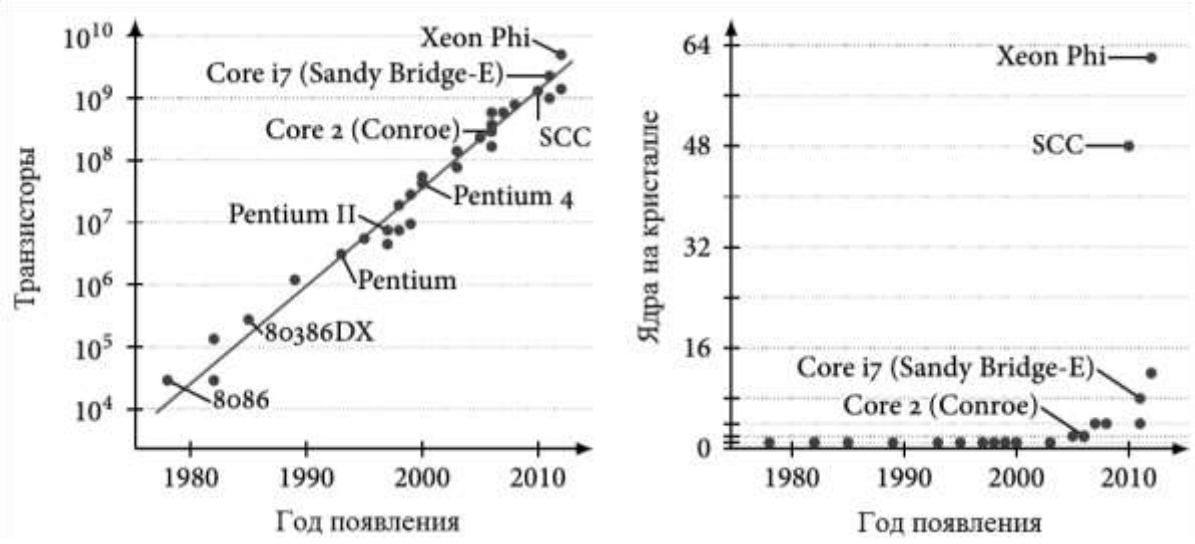


Рис. 7. Эволюция процессоров Intel. Изменение количества транзисторов и процессорных ядер [9].

Кроме того, заказные аппаратные решения по своей сути являются узкоспециализированными, могут решать только ограниченное количество задач. Фактически при таком подходе для каждого приложения требуется проектировать отдельную микросхему. Это экономически оправдано

только для некоторых, наиболее часто встречающихся, и не настолько масштабных задач (шифрование, цифровая обработка сигналов и т.д.). Занять нишу между заказной, жёстко заданной аппаратурой и процессорными программируемыми реализациями призваны реконфигурируемые системы. Но это направление пока только развивается, тоже предоставляя лишь потенциальные возможности, которыми нужно научиться пользоваться.

Сложность аппаратуры современных электронных устройств проявляется не только в технологии СнК, когда множество вычислительных элементов взаимодействуют на одном чипе, но и на уровне распределенных ВсС, которые состоят из взаимодействующих вычислительных устройств — таких, как электронные блоки управления (ЭБУ) современных автомобилей. На одном транспортном средстве имеются десятки взаимосвязанных самостоятельных блоков, каждый из которых реализует отдельную функцию и взаимодействует с другими. Таким образом, имеется тенденция усложнения ВсС, заключающаяся в том, что большинство систем реализуются в виде многопроцессорных распределенных ВС или контроллерных сетей.

В программном обеспечении ВсС сложность возрастает ещё быстрее. Количество строк кода, например, в авионике, в последнее время росло — и продолжает расти — экспоненциально (Рис. 8). То же самое верно и для автомобильной промышленности, мобильных устройств, других областей. Миллионы и даже сотни миллионов строк кода должны выполняться одновременно на связанных между собой устройствах, обеспечивая функционирование сложных распределённых систем.

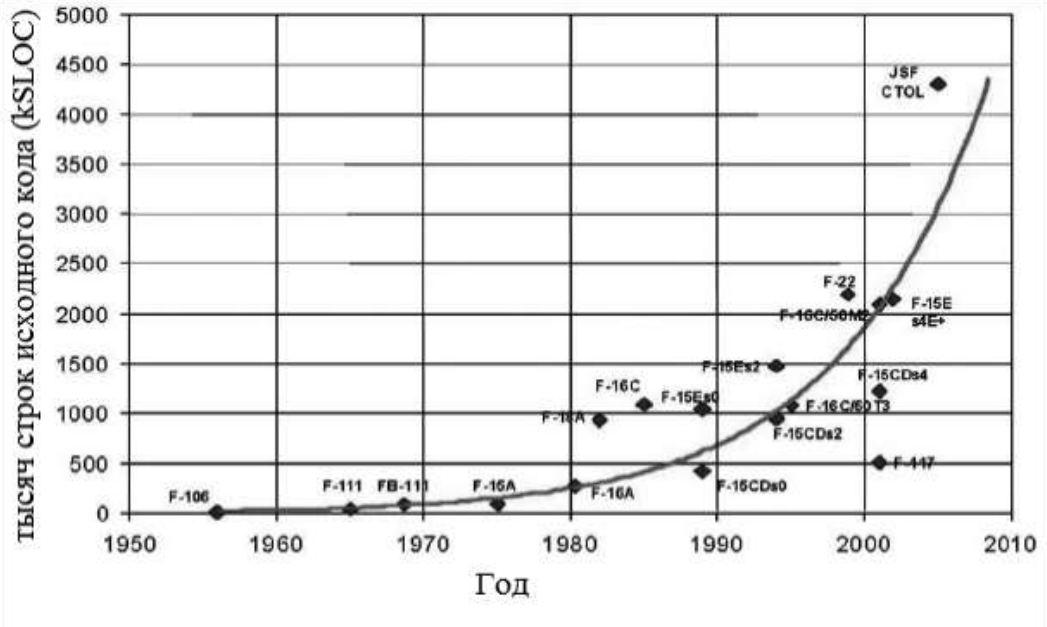


Рис. 8. Рост сложности программного обеспечения в авионике [4].

Кроме этого, всё большее количество компаний идёт по пути выбора фиксированной аппаратной платформы, отлаженной и оттестированной, функциональность которой наращивается и изменяется за счёт ПО и реконфигурации. Это позволяет избежать рисков и затрат на проектирование аппаратуры, но ещё больше работы смещается в область программирования. Фактически львиная часть бюджетов разработки ВсС сегодня приходится именно на разработку ПО.

Этот огромный объём программного кода современных ВсС требует соответствующих затрат на его отладку и верификацию. Создание действительно надёжных и безопасных, отвечающих заданным критериям качества устройств, требует всё больше внимания уделять оценке корректности применяемого в ВсС ПО.

Помимо проблем создания сложных систем «с нуля», остро стоят вопросы *повторного использования* и *интеграции*. Применение готовых блоков и модулей, в том числе, созданных сторонними компаниями, может значительно сократить время выхода на рынок, но это сопряжено с определёнными трудностями и рисками. Использование законченных подсистем сторонних производителей требует их всестороннего и тщательного тестирования и проверки — ведь в случае обнаружения, что подсистема в чём-то критично не соответствует возлагаемым на неё задачам или конфликтует с остальными частями проектируемой ВсС, могут потребоваться огромные затраты на попытки разобраться в её устройстве и на переделку под конкретные нужды (если таковая, вообще, возможна). Подобный подход значительно облегчило бы наличие стандартов и соглашений по обмену готовыми программными и аппаратными решениями, которые позволили бы добавлять новые устройства по типу «plug-and-play».

Кроме вышеперечисленных проблем проектирования современных ВсС, относящихся больше к технической области, имеет место также и ряд моментов, обусловленных влиянием человеческого фактора. Ведь инструменты для автоматизации проектирования являются лишь вспомогательными средствами, полностью исключить роль человека сможет лишь полноценный искусственный интеллект. А мышление человека имеет некоторые характерные особенности, которые необходимо учитывать при планировании процесса проектирования.

Так, например, очень часто у разработчиков ВсС отмечается *суженное пространство проектных решений*, излишне *шаблонное проектирование*. Сужение проектного пространства выражается в неявном выборе того или иного решения или шаблона, без рассмотрения и анализа альтернативных вариантов. Как правило, это вызвано [10,11]:

1) Зависимостью пространства проектных решений от ситуации и исполнителя.
2) Фиксацией решения на уровне технического задания заказчиком, вызванной текущими тенденциями и сложившимися/привычными практиками.

Сужение пространства приводит к принятию неэффективных и не оптимальных проектных решений. Это наиболее характерно при проектировании архитектуры системы, так как работа на данном уровне требует высокой квалификации исполнителя, разностороннего рассмотрения (не только целевой системы, но и процесса разработки, доступных кадров, перспективности технологий и многого другого), и при этом имеет наименее формализованные средства генерации и анализа решений. Кроме того, стоимость исправления ошибок архитектурного уровня многократно превышает стоимость исправления ошибок реализации, а ценность этих решений видится низкой, ввиду отдалённости результата во времени, в связи с чем, разработчик не склонен тратить на принятие этих решений серьёзные ресурсы.

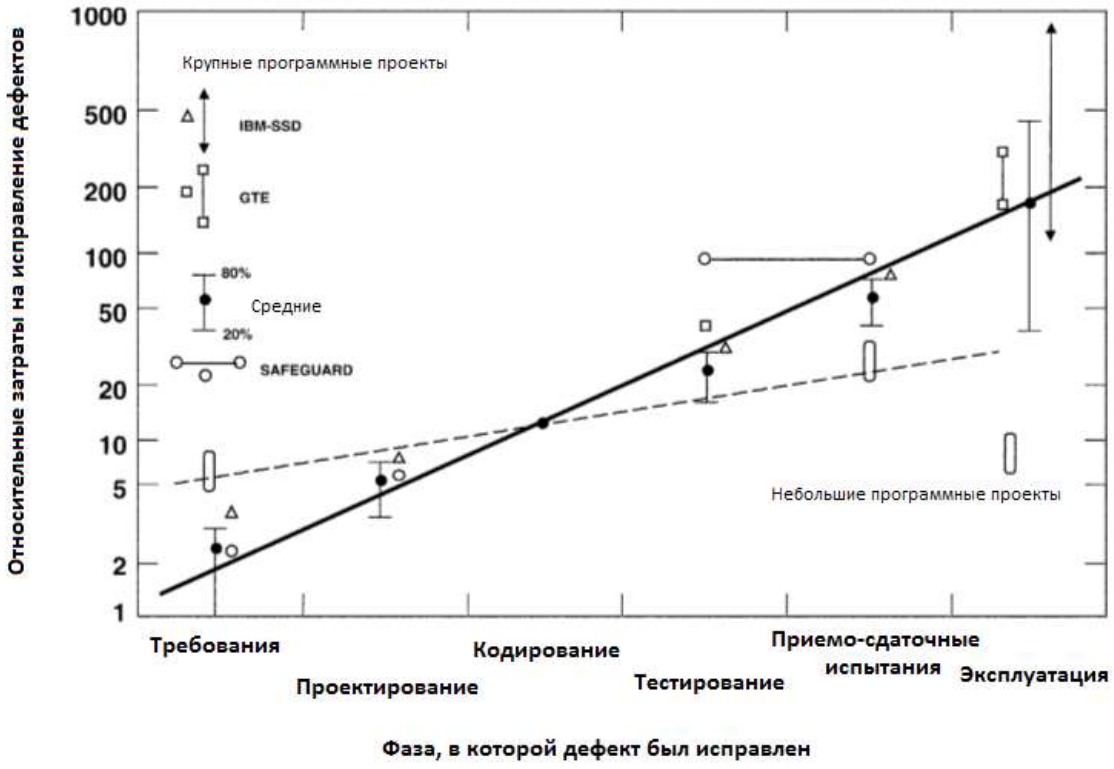


Рис. 9. Зависимость стоимость исправления ошибки от стадии, на которой она обнаружена.

Кроме того, отмечается проблема *сохранения* и *передачи накопленного опыта* в области высокоуровневых решений. Ведь наиболее общие и значимые решения в проектировании ВсС принимаются именно на

архитектурном уровне, но при этом, к сожалению, процесс принятия архитектурных решений и сами архитектурные решения имеют довольно низкий уровень формализации, что обуславливается:

1) высокой размерностью задачи, выраженной в количестве анализируемых объектов, методологических проблемах анализа их различий и многогранности вариантов рассмотрения;
2) отсутствием прямых критериев оценки эффективности принятых решений;
3) высокой сложностью установления корреляции между принятыми архитектурными решениями и интегральными характеристиками проекта — корреляция легко устанавливается для ошибочных решений, в отличие от «правильных», оптимальных (ярким примером является так называемая «ошибка выживших»).

Сегодня высокий уровень формализации достигнут для описания ряда технических вопросов организации ВсС, с которыми сталкивается разработчик ежедневно. В первую очередь это структурные описания вычислительной системы (компонентные архитектурные стили), структуры данных, классификации, процессы взаимодействия в системе и с системой. Работа с менее распространёнными или более оригинальными способами представления системы практически сразу наталкивается на дефицит соответствующих инструментов (что можно видеть на примере аспектного программирования [12]). Это формирует проблему сохранения и передачи проектного опыта, решение которой позволило бы значительно повысить эффективность индустрии.

Основными существующими инструментами, в некоторой степени решающими данную проблему, являются шаблоны проектирования (design patterns, [13]) и языки архитектурного описания (Architectural Description Language, [13]), в которых аккумулируется накопленный опыт и практики проектирования.

При этом наличие открытых исходных кодов проектов или документации на них (даже архитектурного уровня) имеет крайне низкую эффективность, так как:

1) Работа с исходным кодом имеет крайне высокую стоимость восстановления информации «архитектурного уровня» об организации системы.
2) Зафиксированная информация об архитектуре системы крайне редко содержит информацию о том, как формировалась данная архитектура и оценку её эффективности, что не позволяет эффективно применять полученный опыт.

Ещё одной проблемой является *смешение интересов*. Ведь работа над ВсС в целом, единовременно, на сегодня не представляется возможной из-за размера и сложности разрабатываемых систем, а также высокого уровня специализации разработчиков. По этой причине, пространство проектных решений имеет сегментацию с точки зрения гранулярности/общности рассматриваемых объектов, а также с точки зрения интересов различных групп задействованных в проекте специалистов. Выбор сегмента проектного пространства определяет сложность работы с ним, что обусловливает качество и сроки проектирования. Избыточность сегмента приводит к рассеиванию внимания и росту сложности, недостаточность сегмента – к сужению пространства проектных решений.

В рамках одного проекта используется множество сегментов проектного пространства, работа с которыми производится поэтапно, а принятые решения воплощаются в виде ВсС. Такой подход к проектированию получил название «принципа разделения интересов» (separate of concerns [14]) и широко используется в различных методологиях и инструментальных средствах.

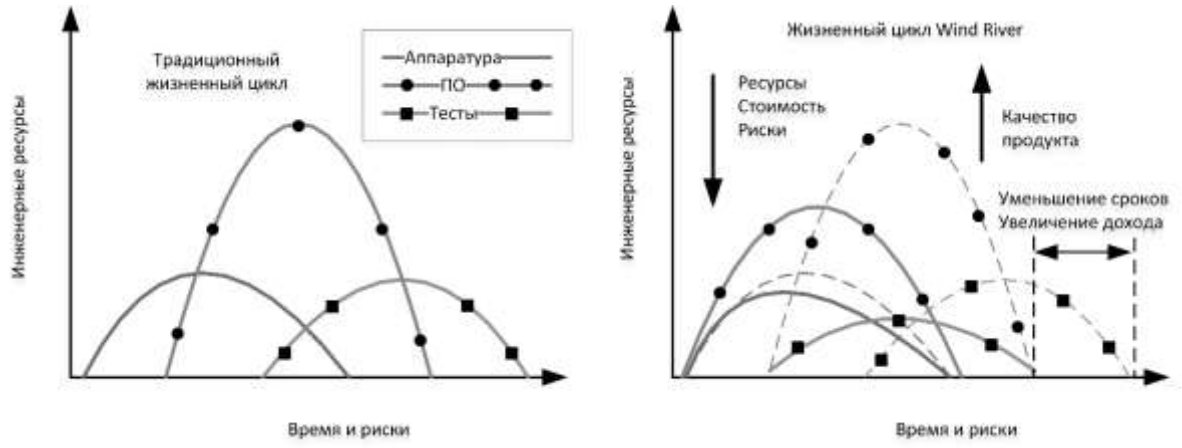


Рис. 10. Ранняя интеграция компонентов ВсС, реализованных в рамках различных ВПл.

Несколько искусственным может показаться разговор об уровневой организации вычислительных систем, как о проявлении принципа разделения интересов, но по всем признакам это так: существует множество взаимосвязанных описаний единого объекта (вычислительного процесса), каждое из которых адресовано к своей группе интересов. Впечатление искусственности вызвано наличием строгой взаимосвязи между уровнями организации вычислительного процесса, в рамках которой вышележащие уровни реализуются нижележащими. Это находит отражение в структуре процесса разработки.

Совместная работа с представлениями ВсС, показанная в правой части Рис. 10, является одной из актуальных задач, так как последовательная

разработка уровней снизу-вверх не только занимает больше времени, но и повышает интеграционные риски [15,16].

Данная проблема особенно актуальна для ВсС ввиду требований реального времени и ограниченных ресурсов, приводящих к смешению уровней между собой в сильно связанные конструкции [17].

Как было сказано ранее, традиционный жизненный цикл ВсС подразумевает разработку программной и аппаратной составляющей ВС, которые, как правило, производятся различными специалистами. Реальное число специальностей, необходимых для разработки типовой ВсС, больше, и может варьироваться в зависимости от используемых технологий (к примеру, от выбора производителей ПЛИС или языка программирования).

Поэтому одним из довольно серьёзных человеческих факторов, которые необходимо учитывать, является *специализация разработчиков*. Привлечение специалистов разных специальностей приводит к необходимости организации и координации их совместной работы на всех этапах жизненного цикла системы, разграничение зон ответственностей, управление формальными и неформальными требованиями.

Таким образом, на данный момент существует большое количество актуальных проблем в проектировании ВсС, и сам этот процесс представляется многогранной и сложно формализуемой задачей. Постоянно развивающиеся и совершенствующиеся подходы высокоуровневого проектирования призваны облегчить решение этой задачи.

2 ОРГАНИЗАЦИЯ ВСТРАИВАЕМЫХ СИСТЕМ

2.1 Программное и аппаратное обеспечение

Как правило, ВсС представляют собой полностью самодостаточные устройства. Это цельные ВС, функционирующие практически независимо и обладающие возможностью влиять на своё окружение тем или иным образом (или хотя бы реагирующие на события внешнего мира изменениями собственного внутреннего состояния). В подавляющем большинстве случаев такие системы реализуются как электронные компоненты, смонтированные на печатных платах и связанные при помощи проводящих дорожек определённым образом, с некоторыми интерфейсами для связи с объектами управления и внешним миром. Иногда ВсС выполняются в корпусах и зачастую имеют собственные автономные источники питания.

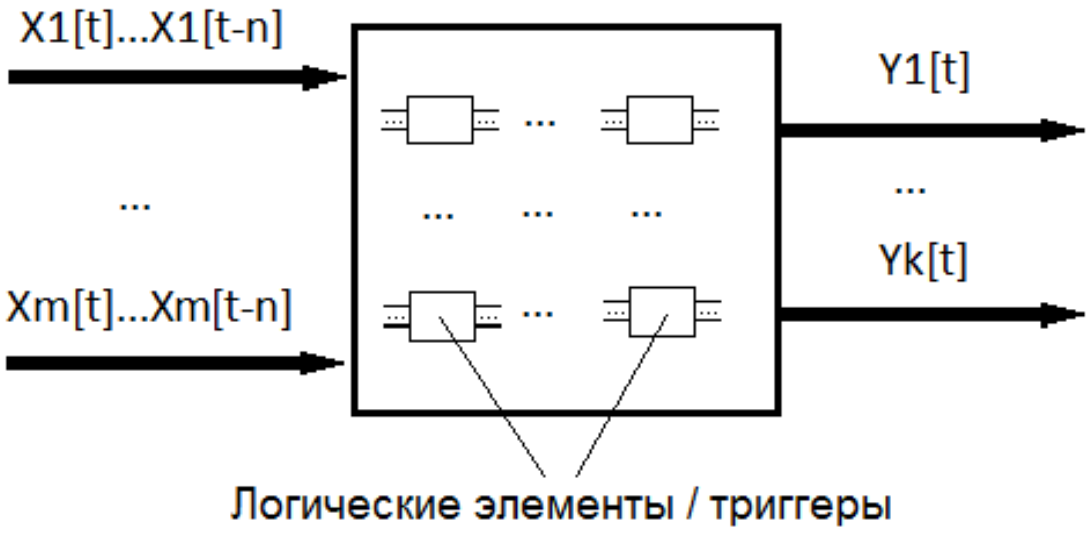


Рис. 11. Схематичное представление некоторой электронной схемы.

Любые ВС вообще являются набором связанных между собой по определённым законам логических и запоминающих элементов, которые производят обработку последовательностей входных сигналов, изменение внутренних состояний и генерацию выходных сигналов. Это схематически изображено на Рис. 11. Значения входных воздействий (*X1...Xm*) в течение *n* тактов до наблюдаемого момента времени (*t*) полностью определяют выходные значения *Y1...Yk* в этот момент времени. При этом в контексте ВС чаще встречаются варианты, когда нет ограничения на *n* тактов – т. е. на актуальное состояние системы может влиять событие, происшедшее хоть годы назад (например, когда была нажата какая-то кнопка, изменяющая режим функционирования устройства).

Теоретически алгоритмы любой сложности можно отображать на комбинационных схемах и триггерах, возможно – с обратными связями, таким образом, чтобы на вход подавались исходные данные, и на выходе через какое-то время появлялся результат. Это так называемая потоковая обработка данных, когда данные обрабатываются вычислительными узлами сразу же, как только поступают на их входы. Но практически реализация такого рода применяется только в некоторых случаях, когда это действительно оправдано. Это происходит из-за трудоёмкости проектирования подобных схем, необходимости использования большого количества транзисторов на кристалле для реализации всех её базовых элементов, и отсутствия гибкости, т.е. возможности изменять функциональность в широких масштабах для решения различных задач.

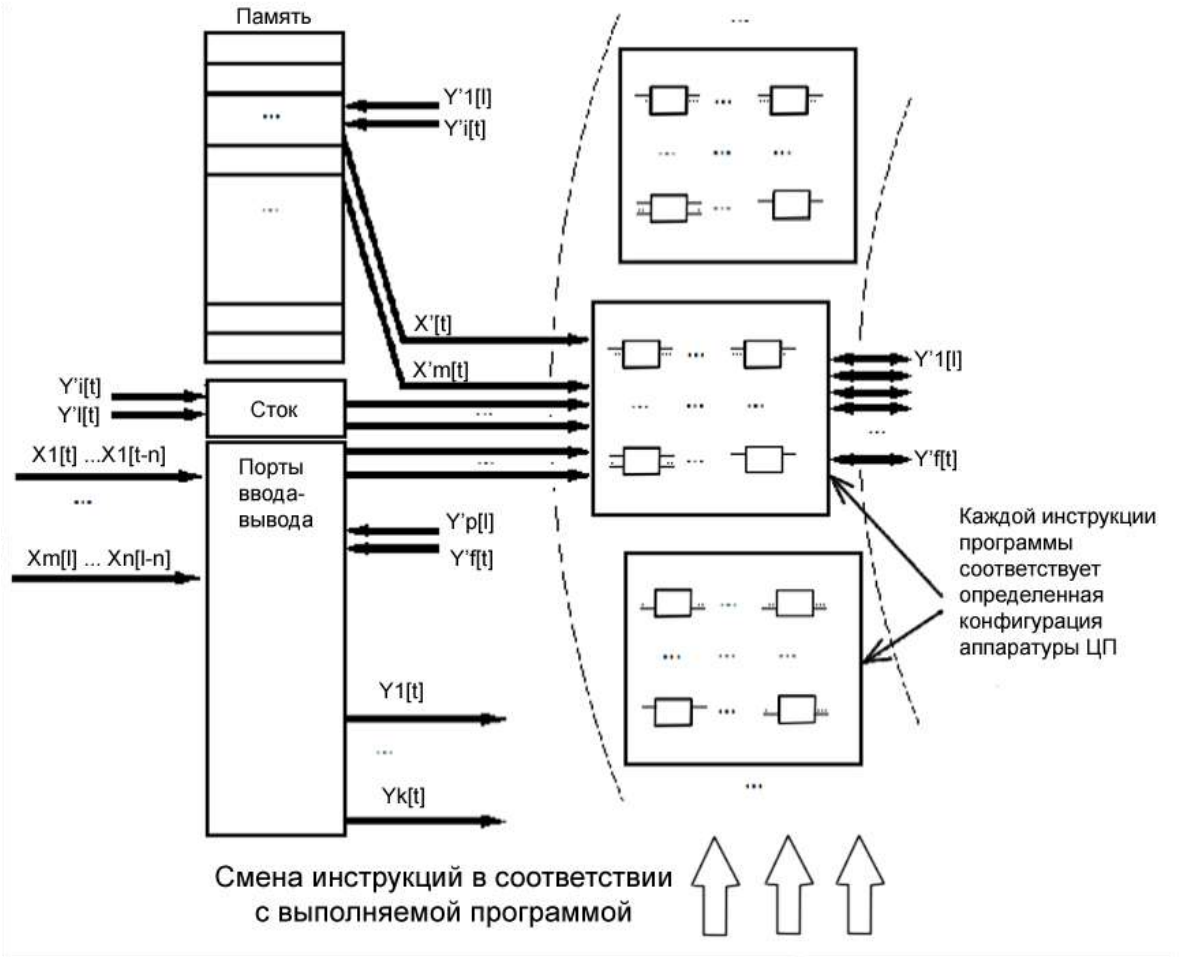


Рис. 12. Схематичное представление фон-Неймановской архитектуры.

Более распространёнными в современной вычислительной технике являются ВС с использованием фон-Неймановской архитектуры. Простейшие ВС, функционирующие на этих принципах, можно представить, как небольшую фиксированную универсальную схему, которая способна выполнять несколько простейших действий (операции

сложения, записи в память и чтения из памяти, логические операции и т.д.). Выбор конкретного действия из всех возможных определяется поданной на вход управляющей последовательностью. Значения управляющей последовательности в заданном порядке выгружаются из памяти, оттуда же берутся также состояния входов схемы и туда записываются результаты выполнения действий с выходов схемы. Координация вычислений и всего функционирования схемы производится управляющей логикой, контроллером.

Набор управляющих последовательностей, изменяющих состояние схемы, называется программой. Выполнение программы можно представить, как вращающееся рывками колесо (Рис. 12), на каждом рывке (такте процессора) подставляющее остальной системе входы и выходы одной из нескольких возможных схем. Колесо может прокручиваться, в том числе и в обратную сторону — отрабатывая команды условных и безусловных переходов, вызовов функций или входя в прерывания — но при нормальном выполнении программы оно проворачивается за раз на одну «схему», в одну сторону.

За счёт такого механизма происходит сворачивание в пространстве схемы, описанной в первом примере: параллельное одновременное выполнение алгоритма, когда данные обрабатываются вычислительными узлами сразу же, как только становятся им доступны, превращается в последовательный, растянутый во времени перебор всех вычислительных узлов с сохранением промежуточных результатов в памяти. Это приводит к значительной экономии транзисторов и, соответственно, места на кристалле, и значительному упрощению задачи создания ВС. Основным недостатком описанной архитектуры является долгая по сравнению с полностью параллельными электронными схемами обработка входных данных, что ещё больше усугубляется необходимостью постоянных обращений к более медленной, чем сам процессор, памяти.

Из приведённых выше примеров можно вывести понятия аппаратного и программного обеспечения. В широком смысле, к аппаратному обеспечению (hardware) относят любые ВС — и различные разновидности фон-Неймановских машин, и аппаратные блоки, реализующие потоковую обработку данных, и возможные промежуточные и гибридные варианты, которые, на самом деле, являются комбинациями этих двух. В контексте существующих технологий создания ЭВМ можно сказать, что это синхронные цифровые электронные схемы, реализованные при помощи комбинационных логических схем и элементов памяти (триггеров). Также, можно дать и ещё более простое и интуитивное определение аппаратного обеспечения — это вещественно существующие части ВС, то, что реализовано в физическом мире, в отличие от ПО, которое представляет собой просто набор данных.

Помимо этого, под аппаратурой в более узком смысле иногда понимают вычислительные блоки, реализующие потоковую обработку данных. В этом нет противоречия, так как в той же фон-Неймановской машине можно явно выделить такие элементы, как управляющий контроллер и тракт обработки данных. Последний в момент выполнения каждой инструкции прогоняет данные сквозь себя, реализуя фактически ту же потоковую обработку данных, просто в меньших масштабах. Т.е. отличие фон-Неймановской машины лишь в том, что в ней есть дополнительная надстройка, управляющая логика, разбивающая глобальные задачи на выполняемые поочерёдно действия.

Программное обеспечение (software) относится уже только к программируемым компьютерам. Хотя многие цифровые схемы, даже реализующие потоковую обработку данных, могут конфигурироваться и в некоторых пределах изменять свои свойства (например, настройка контроллера UART, настройка аудио/видео кодеков и т.д.), это не считается за ПО и обычно затрагивает лишь параметрические изменения. Под понятие ПО подпадают именно последовательно выполняемые инструкции (а заодно, и вообще все те данные, которые, загруженные в программируемый компьютер, определяют его функционирование — например, таблицы рассчитанных заранее значений). К ПО относят также документацию, позволяющую его использовать.

Как правило, в современных разработках ПО для ВсС пишется на языках высокого уровня (ЯВУ), в основном на C и C++. Такое представление более понятно человеку, чем машинные коды и язык ассемблера, которые исторически использовались раньше. При использовании ЯВУ многие детали реализации автоматизируются при помощи интеллектуальных компиляторов, освобождая время и силы программистов на написание и отладку именно алгоритмов. Современные технологии компиляции позволяют генерировать достаточно эффективные исполняемые коды, а в случае, когда этого не хватает и всё же требуется оптимизировать некоторые части программы вручную — например, с целью ускорение выполнения или снижения энергопотребления — есть возможность использовать ассемблерные вставки, не сильно ухудшающие читаемость программ.

Тем не менее, несмотря на отработанные технологии и сложившиеся традиции написания ПО для встраиваемых систем, ведутся поиски путей ещё большего увеличения уровня абстракции. Так, например, пакет Simulink, входящий в состав Matlab от Mathworks, позволяет генерировать код из блок-схем алгоритмов. В некоторых случаях бывает оправдано использование виртуальных машин, напрямую поддерживающих какой-либо ЯВУ и скрывающих детали платформы и низкоуровневой реализации.

В процессе компиляции ПО из понятной человеку формы на ЯВУ транслируется в форму, понятную компьютеру, т.е. в набор кодов инструкций, своим выполнением позволяющих реализовывать на данной аппаратуре заданный программистом алгоритм. Между высокоуровневым описанием и полученным исполняемым кодом нет взаимно однозначного соответствия: из одной и той же программы на ЯВУ, используя разные компиляторы и настройки оптимизации, можно получать совершенно разные исполняемые файлы. И, с другой стороны, по коду исполняемого файла не всегда возможно восстановить исходное высокоуровневое описание.

Очень часто в качестве вычислительных ядер ВсС, берущих на себя основную нагрузку по обработке информации и управлению, используются так называемые микроконтроллеры — относительно маломощные и достаточно стандартные процессоры с некоторым набором периферийных устройств и встроенной памятью, выполненные в виде одной микросхемы и способные решать некоторые простые задачи. Для многих понятие ВсС является синонимом вычислительной системы именно на этой базе, хотя современные технологии позволяют создавать более эффективные по многим параметрам решения.

Существует некоторое число различных вариантов реализаций функциональности проекта, которые могут быть использованы для любой части некоторого алгоритма. Каждая функция системы может быть выполнена как:

1) программа для виртуальной машины на произвольном процессоре;
2) программа на процессоре общего назначения (CPU);
3) программа на процессоре с расширенным набором инструкций для конкретного алгоритма (ASIP);
4) программа на цифровом сигнальном процессоре (DSP);
5) микрокод на специализированном процессоре с очень длинными командами (VLIW);
6) конфигурируемая аппаратура на программируемых логических схемах (FPGA);
7) алгоритм, полностью реализованный в аппаратуре (ASIC).

Каждый из данных подходов к реализации имеет свои характеристики энергопотребления, производительности и гибкости. При разделении функциональности проектируемой системы на части, которые используют эти целевые платформы, или при отображении всей задачи на какую-то выбранную конкретную платформу — ищется баланс между

производительностью и энергопотреблением с одной стороны, и возможностью изменять поведение при помощи программирования с другой. Немаловажную роль также играют такие факторы, как стоимость изготовления, трудоёмкость проектирования и другие.

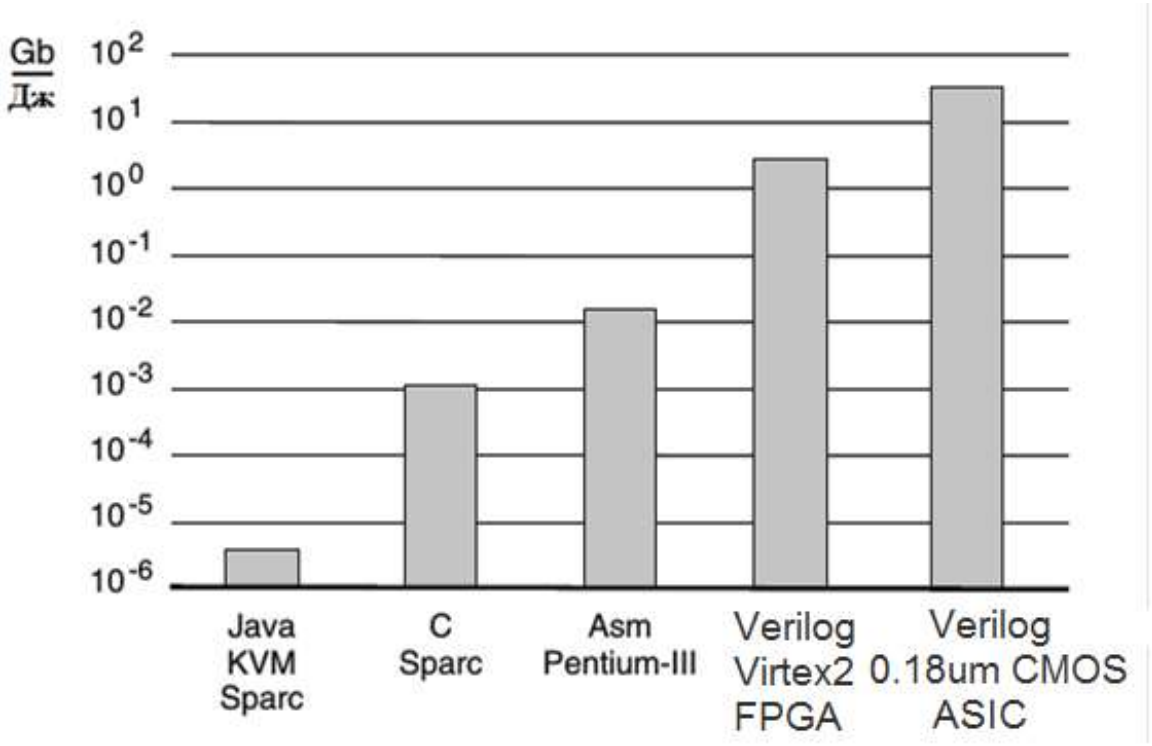


Рис. 13. Энергетическая эффективность [18].

Рис. 13 даёт представление, насколько могут различаться по эффективности разные подходы к реализации. На нём приведены характеристики энергопотребления для реализации одного и того же приложения — шифрования по стандарту AES на различных целевых платформах. Слева направо это: Java, запущенная на Java-машине, которая запущена на микроконтроллере; C-программа на микроконтроллере; оптимизированный код Ассемблера на процессоре Pentium-III; код на языке Verilog, запущенный на Virtex-II FPGA; ASIC-реализация. Гибкость реализации изменяется слева направо, от очень легко и быстро программируемой Java-машины до трудоёмкой в проектировании и неизменяемой после изготовления заказной микросхемы. Ось Y показывает число гигабит информации, которые могут быть закодированы каждой из платформ, используя один Джоуль энергии. Масштаб логарифмический, и можно видеть, что разница в эффективности различается на порядки.

Из приведённого примера ясно, что характеристики конечного продукта очень сильно зависят от выбора технологии, с помощью которой он реализуется. Нельзя однозначно сказать, что какой-то из вариантов

лучше других. У всех подходов есть свои преимущества, и все они могут быть эффективны для конкретных задач.

2.2 Системы на кристалле

Система на кристалле (System on Chip, SoC, СнК) — система, построенная на едином кристалле, в которой интегрируются такие элементы, как процессор (процессоры, в том числе специализированные), некоторый объем памяти, ряд периферийных устройств и специализированных вычислительных блоков, и их соединения. Всё вышеперечисленное составляет оптимальный набор для некоторого заранее известного приложения — например, обработки и передачи видеоданных. Выражение "система на кристалле" не является, строго говоря, термином. Это понятие отражает общую тенденцию к повышению уровня интеграции за счёт интеграции функций.

СнК часто используются в качестве компонентов ВсС и в предельных случаях фактически ими же и являются, для ВсС и СнК используются одни методы проектирования. Реализация в виде СнК исключает необходимость применения многочисленных отдельных интегральных схем и реализации интерфейсов связи между ними. От обычных микроконтроллеров СнК отличается спецификой организации, направленной на решение конкретных задач, в то время как микроконтроллер реализует универсальный набор функций. Также отличием от микроконтроллеров является то, что в СнК обычно используются более мощные процессоры, пригодные для запуска тяжеловесных операционных систем (Linux, Windows), которые зачастую требуют для эффективной работы больших банков внешней памяти и развёрнутую периферию. В некоторых реализациях микросхемы СнК имеют контакты как на нижней своей части для соединения с платой, так и на верхней — для установки дополнительных блоков памяти.

Типовая встраиваемая система, построенная на базе SoC, может содержать такие наборы интерфейсов и контроллеров, как:

- системная шина и контроллеры шин LPC/ISA, PCI, PCMCIA;
- контроллеры управления NOR/NAND Flash, SDRAM, SRAM, DDR;
- контроллер Ethernet;
- последовательные интерфейсы UART, SPI/SSP/uWire, RS-232, RS-422/RS-485, CAN;
- беспроводные интерфейсы WiFi/IEEE802.11, ZigBee, Bluetooth, IrDA;

- интерфейсы поддержки Flash-карт памяти: SD/MMC, CompactFlash, MemoryStick;
- контроллер LCD STN/TFT/OLED;
- контроллер матричной клавиатуры;
- модули беспроводной передачи данных GSM/GPRS, CDMA;
- модули приема сигналов спутниковых навигационных систем GPS, Glonass;
- аппаратные поддержки плавающей точки, шифрования, DRM и т.п.;
- аудио- и видео- интерфейсы;
- источники опорной частоты;
- регуляторы напряжения и стабилизаторы питания.

Преимущества "систем на кристалле" перед классическими "системами на печатной плате":

1) Миниатюризация. Как правило, устройство, созданное на базе SoC, состоит из одной (максимум 2-х) СБИС и ограниченного набора дискретных компонент, которые по технологическим причинам не могут быть интегрированы внутрь ИС.
2) Снижение потребляемой мощности. СБИС типа "система на кристалле" изготавливаются по технологии "глубокого субмикрона" (DSM — Deep Submicron) 0.35 мкм и ниже, что позволяет снизить напряжение питания и, как следствие, существенно уменьшить потребляемую мощность.
3) Повышение надежности. Объединение нескольких компонент (IP-блоков) на одной пластине кремния позволяет существенно уменьшить число паяных соединений.
4) Снижение стоимости для больших партий. ВсС, выполненных на базе СнК, при налаженном производстве являются гораздо более дешёвыми решениями, чем обычные системы на платах.
5) Упрощение монтажа. При использовании технологии СнК требуется устанавливать меньше корпусов на плату — как правило, сам вычислительный элемент.

Также, необходимо помнить, что процессу проектирования аппаратуры обычно сопутствует процесс создания ПО, драйверов, позволяющих «оживить» создаваемую микросхему. Кроме того, производительность приборов класса СнК в значительной мере зависит от эффективности взаимодействия всех встроенных компонентов и от

эффективности их взаимодействия с внешним, относительно прибора, миром. В первую очередь это связано с различием в быстродействии встроенных компонентов, в особенности организации интерфейсов.

При всех достоинствах технологии проектирование и отладка СнК как единой сложной системы является гораздо более трудоёмким и сложным процессом, чем проектирование её компонентов в виде отдельных микросхем. Исходя из этого, одной из основных проблем при разработке СнК является борьба со сложностью и трудности анализа протекающих внутри микросхемы процессов.

Помимо этого, учитывая высокую стоимость исправления ошибок в конечных реализациях, с одной стороны, значительную долю процесса проектирования СнК всегда занимает верификация и всестороннее тестирование получаемого продукта, а с другой — ищутся пути внесения изменений в законченное и уже выпущенное с производства изделие. Последний вариант привлекателен ещё и тем, что может увеличить гибкость устройства, позволяет лучше настраивать его под конкретное приложение.

Существует отдельный класс СнК, представляющих однокристальное конфигурируемое или программируемое решение, которое допускает оперативное изменение своей внутренней аппаратной структуры и конечного предназначения на этапе производства или в полевых условиях, непосредственно в проекте. Такие интегральные схемы были отнесены к группе изделий системного уровня интеграции, но получили другое название — Configurable System on a Chip или CSoC. Поскольку термин CSoC не стандартизован, то существуют и другие названия изделий этого класса — System on Programmable Chip (SoPC), Programmable System on a Chip (PSoC) или просто SoC, что определяется вкусом и желаниями конкретного производителя микросхем.

Помимо этого, перспективным на данный момент направлением развития СнК считаются сети на кристаллах — СнК, в которых соединения между отдельными компонентами реализуются не в виде шин и прямых соединений, а в виде сетей. Это позволяет улучшить масштабируемость однокристальных решений, а также, в больших проектах — энергопотребление. Кроме того, ведутся исследования по использованию беспроводных соединений для связи в рамках чипа, с целью обхода ограничений традиционных электрических соединений, связанных с задержками передачи сигналов, их рассинхронизацией и энергетическими потерями в цепях.

2.3 Многоядерные системы

Развитие компьютерных архитектур в настоящий момент тормозится наличием следующих основных системных проблем:

1) Энергетическая стена – транзисторы дешевы, энергия дорога;
2) Стена памяти – памяти много, но доступ к памяти сравнительно медленный;
3) Стена параллелизм**а** – комплексная проблема, начинающая с параллелизма на уровне команд и кончающаяся распараллеливанием на уровне процессорных ядер;
4) Стена образования – классическое высшее образование с достаточно жесткой привязкой к архитектуре Фон-Неймана делает научно-техническое сообщество более инертным и мешает развивать и внедрять новые архитектуры систем.

Первая стена не позволяет увеличивать дальше тактовую частоту процессорных ядер, а остальные три стены затрудняют увеличение количества ядер. Кроме того, увеличение количества ядер не всегда приводит к пропорциональному приросту производительности, вследствие следующих ограничений:

1) Закон Амдала: «В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента» (см. рис. 14);
2) Неэффективная схема связей между ядрами, вытекающая из технологических ограничений по организации связей между узлами сетей на кристалле или кристаллами на печатной плате и не позволяющая добиться нужной степени распараллеливания для данной задачи;
3) Сложность программирования.

Тем не менее, несмотря на описанные выше проблемы, многоядерные системы являются основным и едва ли не единственным на данный момент реально осуществимым подходом к созданию высокопроизводительных, «high-end» процессоров. Причём, число вычислительных ядер в последних процессорах исчисляется десятками (Рис. 14). То, что такие компании-гиганты, как Intel и Amd, применяют именно эту технологию в своих перспективных разработках, говорит само за себя. Активно используются многоядерные системы и в области ВсС – поэтому, следует коснуться этого вопроса хотя бы обзорно.

Концепция многоядерных систем подразумевает как минимум три аспекта [19]:

1) наличие нескольких вычислительных ядер;
2) наличие коммуникаций между этими ядрами;

3) наличие связи с внешним миром – памятью, периферийными устройствами и т.д.

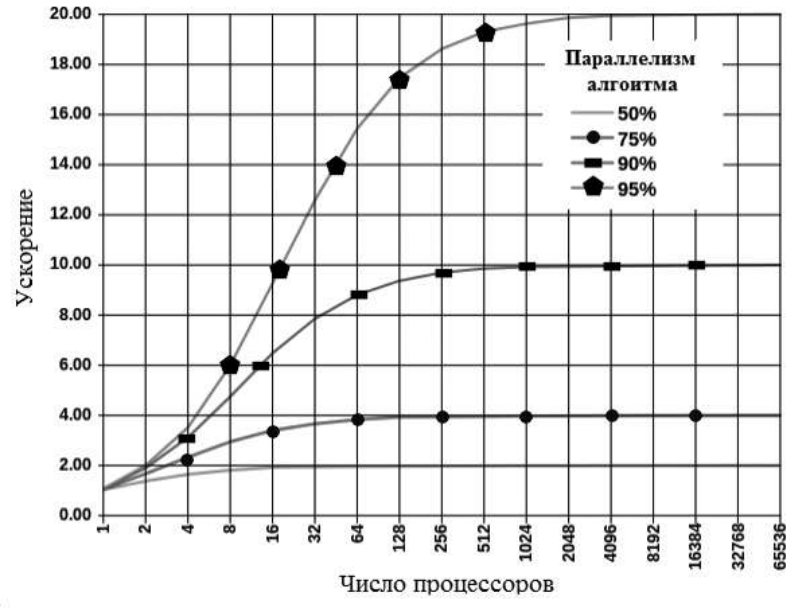


Рис. 14. Закон Амдала.

Процессоры, которые образуют многоядерные системы, могут быть гомогенными – когда на всех ядрах могут запускаться одни и те же бинарные файлы. Такие процессоры имеют одинаковые наборы команд и производительность. При этом современные архитектуры могут управлять частотой тактирования для каждого ядра, что помогает в вопросах энергосбережения или, наоборот, даёт возможность временного ускорения какого-то приложения. Вообще, большинство многоядерных процессоров общего назначения являются гомогенными. Кроме этого, в большой их части реализовано глобальное адресное пространство, общее для всех ядер, и полная когерентность кэш-памяти (когерентность предполагает, что у каждого процессора свой кэш, но в него отображаются все изменения в глобальной памяти). Таким образом, с программной точки зрения, одно гомогенное ядро практически ничем не отличается от другого. Гетерогенные архитектуры включают как минимум два разных типа ядер, различающихся по набору инструкций, функциональности или производительности.

Кроме физической многоядерности, на каждом процессоре может использоваться технология SMT, позволяющая запускать одновременно несколько вычислительных потоков, на самом деле выполняя их поочерёдно. Это достигается за счёт дополнительных наборов регистров и контроллеров прерываний при общих остальных ресурсах. Благодаря этому, в момент простоя одного потока вычислений управление передаётся другому — например, в момент ожидания освобождения какого-то

ресурса, или завершения долгой, многотактовой команды. В процессорах производства Intel эта технология, называемая «Hyper-threading», позволяет получать два логических процессора для каждого физического, в процессорах Sun UltraSPARC — восемь. Однако, это не является многоядерностью в прямом смысле слова.

В отношении коммуникаций между ядрами внутри одного чипа первоначально использовался подход с общей шиной, разделяемой между всеми индивидуальными процессорами, и имеющей выход к контроллеру памяти. Для снижения трафика в шине каждому процессору назначалась кэш-память, обычно двухуровневая, устанавливаемая между вычислительным ядром и шиной. Но даже несмотря на это, общая шина ограничивает пропускную способность. При передаче данных разделение трафика между всеми подключёнными к шине ядрами не может не сказываться негативно, и, кроме того, при подключении многих ведомых устройств к шине она имеет достаточно большие задержки из-за своих электрических свойств (большой длины проводников и растущих с количеством подключённых устройств ёмкостей).

Поэтому в более современных реализациях разработчики процессоров стараются отойти от такого подхода, соединяя ядра между собой и с памятью более оптимальными способами. Так, например, расположение кэш-памяти между процессорами и общей шиной позволяет снизить трафик в шине. А при подходе, называемом «неравномерный доступ к памяти (NUMA, Non-Uniform Memory Access), во все вычислительные ядра интегрируются контроллеры памяти. Благодаря этому каждый процессор получает непосредственный доступ к некоторой части физической памяти, доступной через него для других ядер. В итоге, получается, что при данном подходе память физически распределена, но логически общедоступна — иначе говоря, доступ к различным областям памяти может занимать разное количество времени.

Обеспечение непротиворечивости памяти при возможности доступа к ней нескольких параллельно работающих ядер находится в числе фундаментальных проблем создания многоядерных систем. Это приводит к необходимости введения методов синхронизации, обеспечивающих когерентность использующихся несколькими ядрами данных. При этом механизмы блокировки или синхронизации могут стать причиной конфликта блокировок между несколькими потоками или процессами, конкурирующими за доступ к данным (или другому ресурсу).

Сообщения об изменении одним из ядер значения переменной, хранящейся в памяти, передаются по внутренним интерфейсам другим ядрам — их кэши прослушивают внутренние шины. Впоследствии, при обращении к изменённой переменной эти ядра будут вынуждены читать

значения из памяти, даже если у них есть их копии в собственном кэше. Сам же по себе доступ к разделяемым ресурсам может быть обеспечен, например, если все запросы к этим ресурсам от разных ядер складываются в простое FIFO, заполняемое последовательно.

Рис. 15. Структура типового многоядерного процессора [20].

В некоторых реализациях для коммуникаций между ядрами может применяться специально выделенная разделяемая память. Этот подход редко используется в высокопроизводительных процессорах, но его можно встретить в гетерогенных системах, использующих, например, кроме основного ЦП ускоряющий определённые задачи DSP-сопроцессор.

С точки зрения создания ПО для многоядерных систем базовыми задачами является разделение приложений на параллельно выполняемые потоки и обеспечение синхронизации между ними. Реализация чисто программной синхронизации сложна, но, с другой стороны, аппаратные механизмы синхронизации не масштабируются. Поэтому в основном используются комбинированные методы, реализованные операционными системами с частичной аппаратной поддержкой.

Одной из важных концепций, относящихся к поддержанию целостности совместно используемой разными ядрами памяти и других ресурсов, являются атомарные (неделимые) операции сохранения данных. Запись в идеале должна происходить мгновенно, и информация об этом должна доходить до всех ядер. В качестве аппаратной поддержки синхронизации современные процессоры предоставляют так называемые механизмы «read-modify-write», позволяющие совершить все операции с общими данными максимально быстро.

Таким образом, многоядерные системы представляют собой очень мощный и при этом чрезвычайно сложный инструмент. Это проявляется как в их использовании, с точки зрения программистов, так и в проектировании, с точки зрения разработчиков микропроцессоров. Остаются нерешёнными задачи оптимизации доступа к памяти, снятия ограничений масштабируемости многоядерных систем, обеспечения соединений между ядрами. При этом в большинстве случаев современное ПО пишется в стандартной, последовательной манере, без учёта возможностей многопоточного исполнения.

2.4 Реконфигурируемые системы

Реконфигурируемые вычислительные системы (РВС) — это системы, имеющие возможность менять свою модель вычислений, или, иначе говоря, позволяющие вносить существенные изменения в свою аппаратную часть. РВС занимают нишу между двумя парадигмами создания ВС: чисто программными системами на некотором процессоре с фиксированной аппаратной архитектурой и специализированными аппаратными, реализующими, как правило, потоковую модель вычислений.

Необходимость таких систем обусловлена тем, что классическая вычислительная система на базе архитектуры фон-Неймана имеет множество недостатков. Одна из её проблем – взаимодействие процессора с памятью, так как современная память работает значительно медленнее процессора. Несмотря на наличие механизмов, ускоряющих этот процесс (многоуровневая кэш память, конвейерное исполнение команд, предсказание ветвлений), в большинстве случаев АЛУ процессора простаивает, что снижает эффективность системы в целом.

Эффективность работы процессора немного возрастает при использовании механизма гипертредининга, когда к одному блоку памяти обращается два потока, работающие с одной и той же областью памяти в рамках одного процесса. Но, тем не менее рост производительности, необходимый для высокопроизводительных вычислений, и экономичность, важная во ВсС, плохо достижимы в рамках классических архитектур. Чем больше разрыв между скоростью работы памяти и процессора, тем ниже фактически достигаемая эффективность работы вычислительной системы. Кроме этого, 95% времени выполнения инструкции (см. Рис. 16) приходится на всевозможные вспомогательные и подготовительные действия, и лишь 5% на саму вычислительную операцию – то, ради чего всё, собственно, и организуется.

Рис. 16. Примерное соотношение времени работы различных вспомогательных действий при выполнении инструкции микропроцессора с архитектурой Фон-Неймана [21].

С другой стороны, фон-Неймановская машина является универсальным вычислителем, способным решать практически любую задачу в зависимости от заложенной программы. Аппаратные реализации потоковой модели вычислений, представляющие собой её полную противоположность и другую крайность среди реализаций ВС, не имеют необходимости обращаться к памяти за данными и инструкциями перед каждым действием. Поэтому они представляют собой, как правило, максимально оптимальное вычислительное устройство, которое можно реализовать на заданных ресурсах, но при этом решаемая задача фиксируется жёстко, и имеются ограничения на её размерность.

Использование РВС позволяет достигать производительности вычислений большей, чем при использовании ПО и фиксированного процессора (но всё же, обычно, меньшей, чем с аппаратно реализованными потоковыми вычислениями), при этом — с некоторой степенью гибкости и возможностью изменения исполняемых функций в определённых рамках, ограниченных заданными ресурсами. Таким образом получается, фактически, изменяемый аппаратный блок, обладающий преимуществами аппаратной реализации и в дополнение возможностью перестраиваться под разные приложения.

Впервые концепция реконфигурируемости была описана Estrin G. в "Organization of Computer Systems — The Fixed Plus Variable Structure Computer" в 1960 году. Автор предлагал использовать стандартный процессор с дополнительной реконфигурируемой надстройкой, управляемой процессором и использующейся для выполнения специфических задач (таких, например, как обработка сигналов). Однако, развитие идея получила только спустя 20 лет, в 80-90е годы.

Наиболее актуальны РВС в области создания суперкомпьютеров и встраиваемых систем. В первом случае, реконфигурация позволяет получать выигрыш в виде высокой производительности при решении различных специфических ресурсоёмких задач, где уже достигнут предел повышения производительности, полученный стандартными экстенсивными методами, или же — обходится слишком дорого. Во втором — обеспечивается необходимая вычислительная мощность в, наоборот, небольших вычислительных комплексах, которые работают в рамках постоянного дефицита ресурсов: габаритных размеров, массы, энергии, стоимости, что усугубляется также требованиями реального времени. То есть, это два граничных случая для всех ВС.

С помощью принципов реконфигурируемости можно решать различные задачи. На первом месте обычно стоит повышение производительности, функциональная гибкость и адаптивность, улучшение надёжности, сокращение энергопотребления и размеров. Другие достоинства реконфигурируемости требуют для понимания анализа жизненного цикла изделия и возможных подходов к его проектированию: удовлетворение нужд потребителей, быстрая разработка и выведение на рынок, повышение качества продукта, дифференциация продуктовой линии, адаптация к новым стандартам, снижение стоимости разработки.

Стандартным подходом к созданию РВС является реконфигурация при помощи записи значений в некоторые регистры, которые влияют на работу системы, для чего используется электричество (другие способы изменения структуры аппаратуры, например, устанавливаемые «джамперы», к вопросу рассмотрения не относятся). Записанные значения могут храниться после выключения питания, а могут исчезать, в зависимости от реализованной технологии их хранения. Грубо говоря, в предельном случае любую память, используемую в современных компьютерах, можно назвать реконфигурируемой системой.

Аппаратные вычислительные элементы РВС могут быть реконфигурированы однажды, когда система выпускается, раз в какое-то количество лет, для устранения багов, а также для обновлений (статическая реконфигурация), или же раз в несколько часов, чтобы

адаптироваться под текущую задачу (динамическая/run-time реконфигурация). Часто реализуется частичная реконфигурация — динамическая реконфигурация некоторой части аппаратуры при неизменной структуре остальной аппаратуры.

Обычно реконфигурируемый блок взаимодействует с некоторой фиксированной аппаратурой, под управлением какого-нибудь процессора. Существует две категории взаимодействий процессорных ядер и реконфигурируемых компонентов: реконфигурируемая матрица рассматривается контроллером процессора как функциональный модуль, или же она представляет собой независимый сопроцессор. В обоих случаях, матрица должна иметь возможность перепрограммирования, желательно «на лету». Часто, для ускорения этого процесса используются кэши конфигураций.

Использование реконфигурируемого вычислителя как функционального модуля процессора, с разделением некоторого регистрового файла между этими двумя вычислителями, имеет значительное преимущество в простоте по сравнению с использованием автономного сопроцессора. При этом на производительности негативно сказывается использование прямого доступа к памяти через стандартный процессорный конвейер, являющегося традиционным узким местом процессоров общего назначения.

Кроме того, РВС классифицируются по степени гранулярности: как крупногранулярные, которые обычно состоят из цельных реконфигурируемых элементов, реализующих некоторые ограниченные базовые задачи по обработке данных, и мелкогранулярные, которые реализуются на основе матрицы программируемых логических блоков и могут изменять свои свойства вплоть да отдельных вентилей.

Мелкогранулярная реконфигурация реализуется, как правило, при помощи ПЛИС, в которых можно программировать операции, выполняемые базовыми блоками, и соединения между ними. Как правило, все значения, управляющие функционированием подобных систем, генерируются высокоуровневыми инструментами из поведенческих или RTL описаний, кодируются в «битовый поток» и прошиваются при помощи интерфейса стандарта JTAG. Известен способ построения мелкогранулярных вычислительных платформ, называемый «вычисления с памятью» – когда все значения некоторых функций хранятся в реконфигурируемой памяти. Соответственно, реконфигурация производится простой перезаписью этих значений, а вычисления – обращениями к ним.

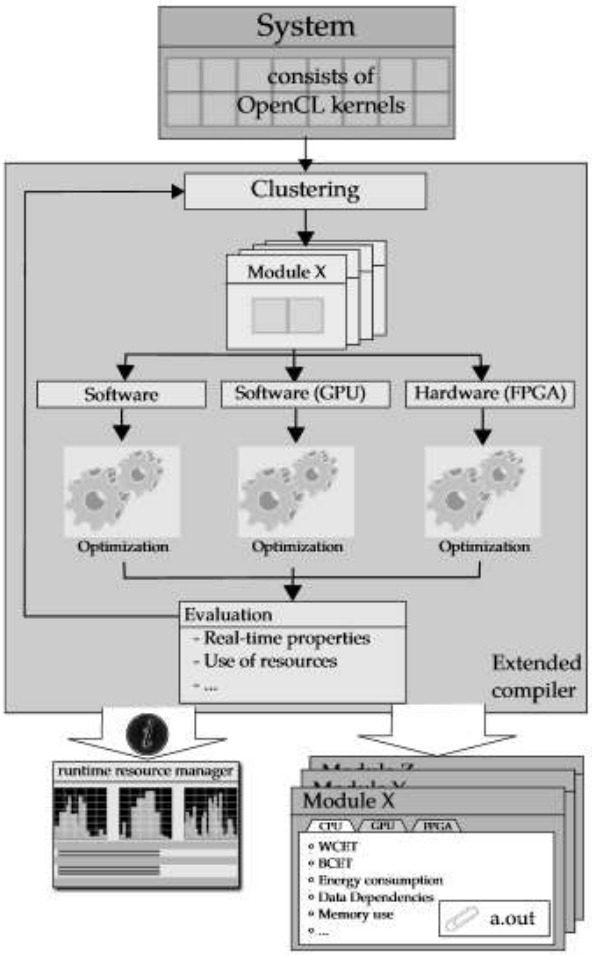


Рис. 17. Организация динамической реконфигурации с компиляцией вычислительных ядер перед выполнением для всех аппаратных блоков [22].

Крупногранулярных архитектур бесчисленное множество (MorphoSys, ADRES и т.д.), они представляют собой системы из блоков с меняющимися связями между ними. В качестве примера крупногранулярной РВС в следующем подразделе рассматривается система NL3, разработанная на кафедре Вычислительной Техники университета ИТМО.

Вообще, РВС чрезвычайно разнообразны по архитектуре и требуют сложных технологий создания конфигурационного обеспечения (configware), что делает задачу их проектирования нетривиальной. Это подтверждается опытом создания большого числа как экспериментальных, так и серийных ВсС с реконфигурируемой архитектурой.

Важными задачами, требующими решения, являются также: определение частей алгоритма, которые требуют оптимизации, и выделение их из изначального вычислительного процесса. Один из подходов [22] предполагает наличие менеджера ресурсов, который перераспределяет вычислительные задачи между различными блоками

(CPU, DSP, GPU, FPGA) в зависимости от требуемой для решаемой задачи вычислительной мощности.

Методология основывается на использовании расширенного компилятора, который разделяет исходный код приложения на модули, генерирует исполняемый код (или конфигурации для аппаратуры), и анализирует модули. Система реализуется на OpenCL и состоит из отдельных ядер. Каждое из этих ядер (модулей) компилируется и оптимизируется для различных вычислительных ресурсов (CPU, DSP, GPU, FPGA), после чего производится анализ его производительности. На выходе компилятора получается информация для менеджера ресурсов и набор модулей с дополнительной информацией, необходимой для управления ими. Впоследствии, управляющая логика может легко перемещать части задачи из одних ресурсов в другие, увеличивая/уменьшая производительность, энергопотребление, и т.д.

Динамическое профилирование совместно с декомпиляцией выполняемого приложения и генерацией аппаратуры для реконфигурируемых блоков может быть использовано для выявления и выделения из основного вычислительного процесса, реализуемого на стандартном процессоре, частей кода, перспективных для перемещения в FPGA модуль. Некоторые аспекты этой опции реализованы в процессоре WARP [23].

Самые часто используемые участки кода декомпилируются для выделения существенного параллелизма. Для этого используются подходы, базирующиеся на анализе индексов массивов, разворачиваются циклы, конвейеризируется доступ к памяти, вычисления, и передача заданных массивов из памяти во внутренние регистровые файлы. За этим следует шаг, определяющий лучший участок кода для реализации в аппаратуре – это могут быть целые функции и циклы. При помощи упрощённой версией FPGA синтезатора, подъёмной для встроенного микропроцессора, производится быстрый синтез прямо в чипе, размещение и трассировка, после чего полученное ядро загружается в аппаратный блок. Описанный подход позволяет получить ускорение на один порядок, правда, с издержками на генерацию прошивки для ПЛИС.

Основной интерес данного метода заключается в том, что проблема разделения на оптимизированные аппаратные и медленные программные части решается уже после развёртывания системы, удовлетворяет насущным требованиям. Это является преимуществом в мире беспроводных соединений – там, где вычисления могут динамически перераспределяться для лучшего использования аккумуляторов, данный аспект может быть решающим фактором. RTOS будущего могут очень хорошо использовать этот механизм как один из доступных методов

снижения энергопотребления, наряду с динамическим изменением напряжения, частоты тактирования и т.д.

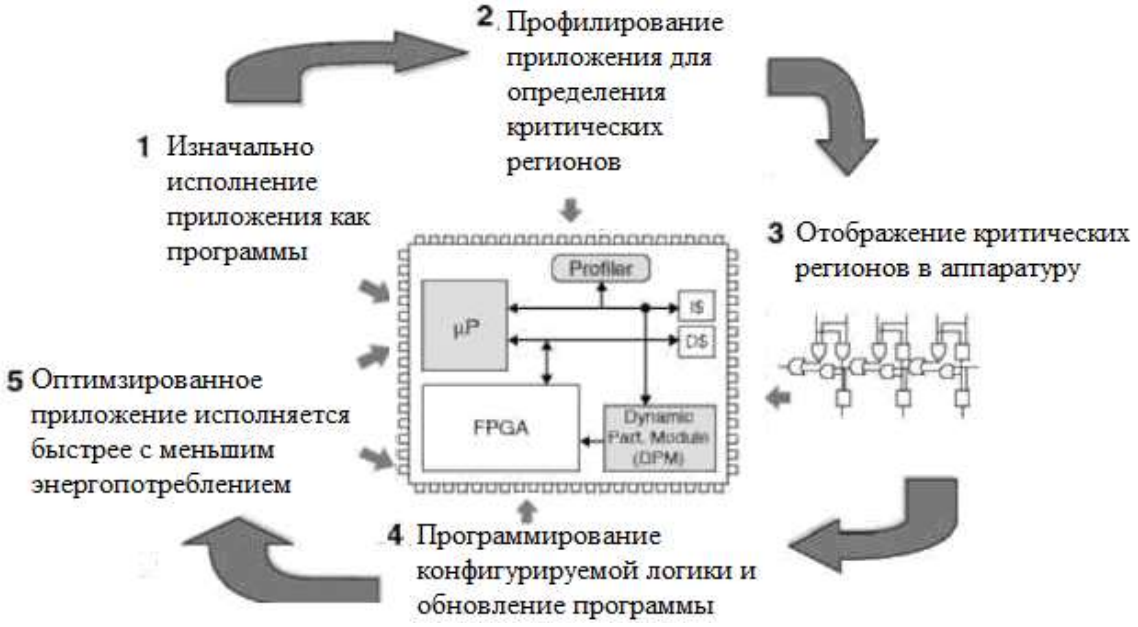


Рис. 18. WARP процессор [23].

Подводя итоги, можно вывести основные задачи, решаемые при создании РВС. Такими задачами являются:

- создание аппаратных блоков, обладающих свойством реконфигурируемости;
- выявление участков приложения, которые требуется оптимизировать;
- отображение оптимизируемых частей приложения на реконфигурируемую аппаратуру (создание оптимизированных аппаратных реализаций);
- реконфигурация – способ записи новой конфигурации в аппаратные реконфигурируемые блоки;
- включение реконфигурируемых блоков в вычислительный процесс.

Ключевая проблема, связанная с реконфигурируемой аппаратурой – сложность её программирования. Традиционные процессы разработки: для аппаратуры – синтез, размещение и трассировка, для ПО – компиляция, запуск и отладка – не поддерживают реконфигурируемые системы. Динамическое реконфигурирование может быть рассмотрено как гибрид между ПО, где ЦПУ "реконфигурируется" с выполнением каждой

инструкции, а объём памяти большой, но пропускная способность доступа к ней ограничена, и аппаратурой, где реконфигурация происходит редко (например, записью в конфигурационные регистры UART), памяти мало, но доступ к ней имеет потенциально высокую пропускную способность. Объединение процессора общего назначения с реконфигурируемым модулем требует создания модели программирования и решения проблемы пропускной способности коммуникаций.

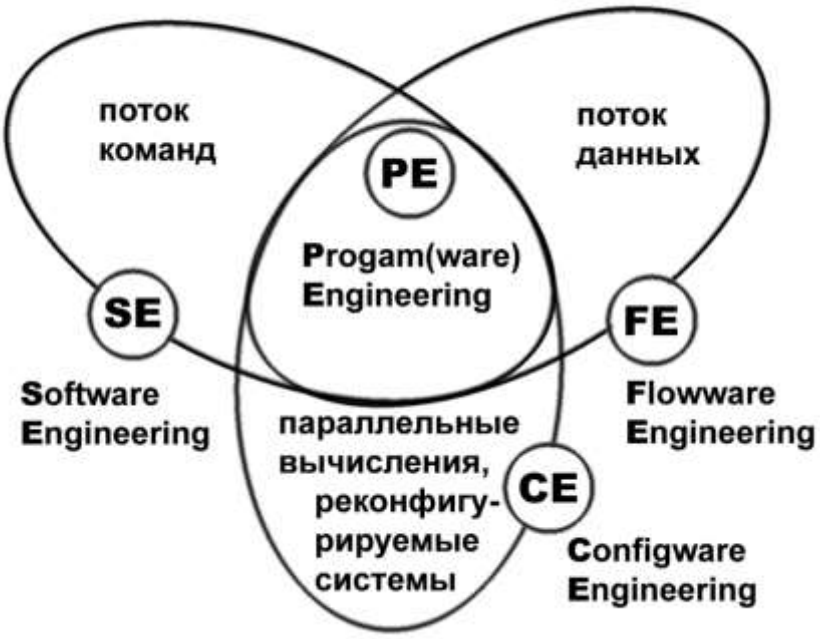


Рис. 19. Современное представление вычислительной техники [21].

При всей перспективности РВС нет единой универсальной методологии, решающей все вышеперечисленные задачи достаточно эффективно. Поэтому исследования и поиск эффективных и комплексных подходов активно ведутся до сих пор.

2.4.1 Пример: архитектура NL3

Рассмотрим организацию и функционирование крупногранулярной РВС на примере системы NL3. Предназначение системы – управление входящим в исследовательский и обучающий комплекс зондовым микроскопом и обработка получаемых с него данных. Одной из особенностей зондовой микроскопии является необходимость высокоскоростной цифровой обработки сигналов (ЦОС) в реальном времени (использование таких алгоритмов, как фильтрация, ПИД-регулирование, синхронное детектирование), что требует эффективных и высокопроизводительных реализаций вычислений. Дополнительным условием, связанным с высокой сложностью прикладной области и использованием комплекса в исследовательских задачах, является

необходимость обеспечения возможности пользовательского программирования комплекса. Таким образом, реализация в виде РВС является напрашивающимся и естественным решением.

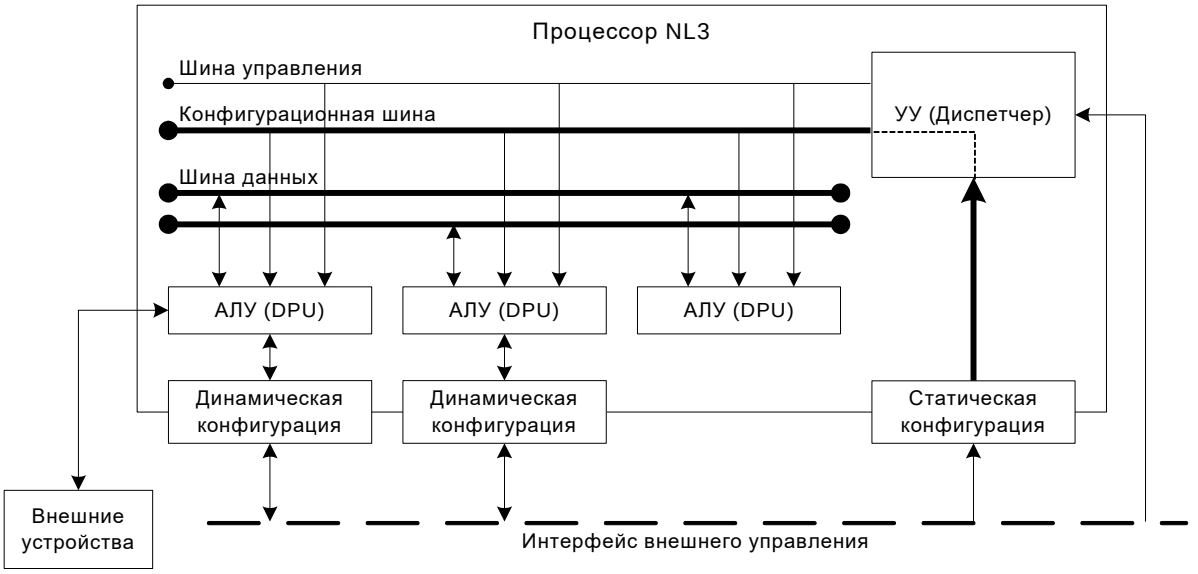


Рис. 20. Организация вычислителя NL3.

Процессор NL3 сочетает в себе элементы суперскалярной и VLIW архитектуры, основной обрабатывающей единицей выступает DPU (Data Processing Unit) – реализация функционального блока модели вычислений. Типовой состав DPU аналитического прибора (может изменяться) включает:

- блоки цифровой обработки (вычислитель поворота координат; коррелятор; КИХ/БИХ-фильтр);
- блоки общего назначения (сумматор; табличная выборка; умножитель; память; мультиплексор);
- специализированные блоки (приемопередатчики внешних устройств; осциллограф; отладчик).

Организация вычислений производится посредством статического и динамического конфигурирования композиции DPU, для чего разработан прототип САПР, включающий графическую систему подготовки net-листов, компилятор на основе системы неравенств, библиотеки логических ФБ, Design-Time и Run-Time библиотеки драйверов DPU.

Основу архитектуры NL3 (Рис. 20) составляют DPU (Рис. 21), которые выполняют задачи вычисления, обмена данными, обмена данными с шинами данных, загрузки данных с конфигурационной шины, обмен данными с ВУ. DPU имеют порты – буферы для ввода/вывода данных. Они

различаются по направлению: входные и выходные, и по типу: для данных и конфигурационные. DPU функционируют независимо друг от друга.

Любые действия в вычислителе выполняются по команде. Типами команд являются:

- Загрузка значения во входной порт;
- Запуск вычислений;
- Выгрузка значения выходного порта.

Команды передаются по конфигурационной шине и шине управления.

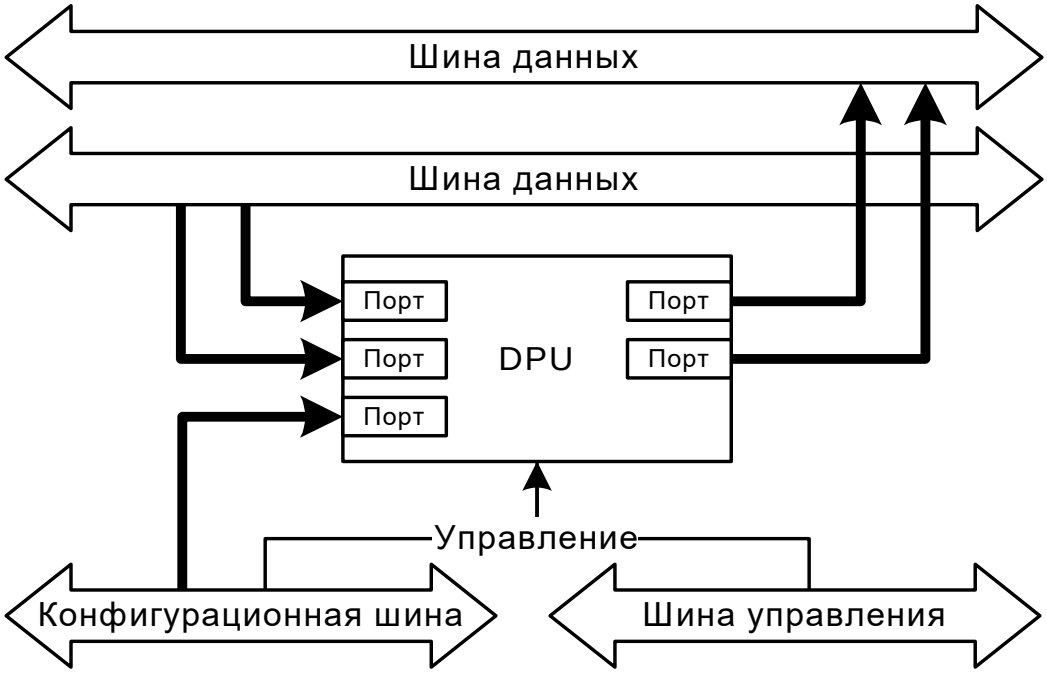


Рис. 21. Data Processing Unit (DPU).

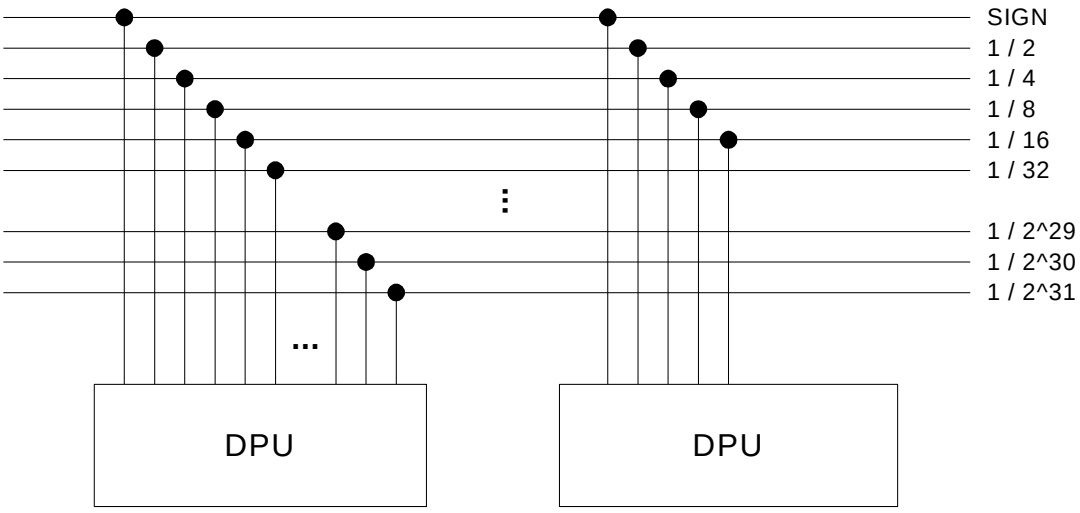


Рис. 22. Организация шины данных.

Шина данных выполняет две задачи – загрузку и выгрузку данных DPU, и организована образом, показанным на Рис. 22. В один момент времени шина может выгружать только один DPU, а загружать – много DPU. Данные кодируются как 32-битные вещественные числа с фиксированной запятой с диапазоном значений: [-1, +1]. К DPU могут быть подключены не все линии шины данных. Выравнивание производится на старшие разряды, уменьшение разрядности подключения приводит к понижению точности, на неподключенных линиях шины данных удерживается ноль. Каждый DPU подключен к одной или двум шинам данных.

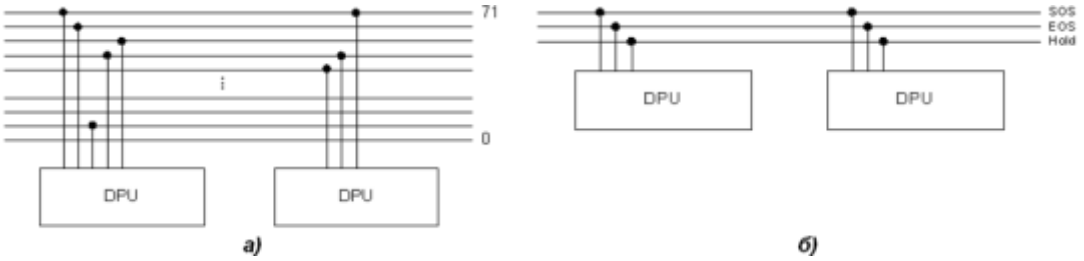


Рис. 23. Конфигурационная шина (а), шина управления (б).

Конфигурационная шина и шина управления изображены на Рис. 23. Конфигурационная шина передает индивидуальные команды и константы, имеет разрядность 72 бита. К каждому DPU подключены только те линии, которые нужны для управления и передачи данных, возможно совместное использование

Шина управления передает широковещательные команды, имеет разрядность 3 бита, и к каждому DPU подключены все линии шины.

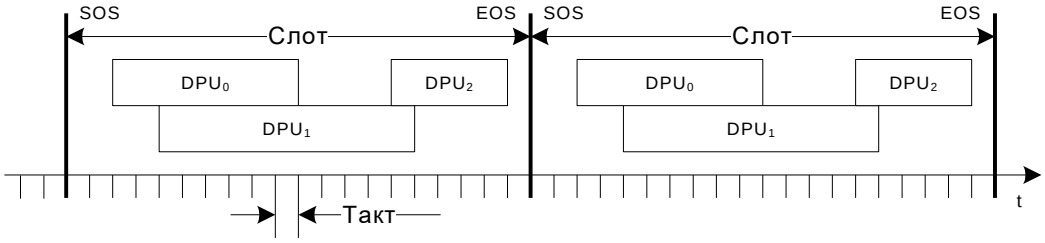


Рис. 24. Диаграмма работы DPU под управлением диспетчера.

Типы команд:

- Start of Time Slot (SOS)
- End of Time Slot (EOS)
- Hold

Важно эффективное подключение DPU к конфигурационной шине.

Диаграмма работы DPU под управлением диспетчера приведена на

Рис. 24. Диспетчер функционирует соответственно следующим правилам. Слот – максимальная длительность обработки входных данных, определяется FSR. Статическая конфигурация (программа) – последовательность значений (инструкций), выставляемых на конфигурационную шину, Размер = Fдиспетчер / FSR, не меняется от слота к слоту, изменения требует остановки диспетчера. Задачи диспетчера: управление конфигурационной шиной в течение слота и формирование команд на шине управления в начале и в конце слота.

Динамическая конфигурация DPU – область памяти, в которой находятся параметры работы и текущие данные DPU. Она может быть изменена во время работы диспетчера, доступна на чтение и запись для DPU и интерфейса внешнего управления и снабжена механизмами синхронизации.

Интерфейс внешнего управления NL3 осуществляет запуск и остановку программ на NL3 (загрузку статической конфигурации диспетчера, управление диспетчером, инициализацию динамических конфигураций DPU); управление и контроль работы DPU (запись и чтение динамических конфигураций DPU в время работы NL3). Интерфейс внешнего управления не является необходимым элементом архитектуры NL3, при его отсутствии просто отсутствует конфигурирование (динамическое управление).

Так как часть свойств NL3 выходит за рамки суперскалярной и VLIW архитектур, поясним эти моменты:

- У DPU различных типов отличается время работы. Вместе с тем время работы однотипных DPU тоже может отличаться. В суперскалярных и VLIW архитектурах активация исполнительного блока – это инструкция. NL3 поддерживает более низкоуровневое программирование. Для активации DPU требуется несколько команд: загрузка, выгрузка и начало вычислений. Можно варьировать количество тактов между этими командами и благодаря этому время работы даже однотипных DPU может отличаться.
- Длинные инструкции передаются по конфигурационной шине. К каждому DPU подключается только часть линий шины. Одни и те же линии могут быть подключены к нескольким DPU, что соответствует наложению команд для разных DPU в рамках одной инструкции. Если в VLIW архитектуре каждый фрагмент длинной инструкции предназначен для отдельного

исполнительного блока, то у NL3 эти фрагменты могут пересекаться.

- DPU поддерживают конвейеризацию. Вычисления начинаются по специальной команде, так что в DPU можно загружать входные данные до выгрузки результатов предыдущих вычислений. Загрузка входных данных требует определенного времени, но она не портит значения, вычисленные ранее.

2.5 Проблемно-ориентированные процессоры

Проблемно-ориентированный процессор, заказной процессор — это специализированный процессор, проектируемый под конкретную задачу. Вообще, под названием «application-specific processor (ASP)» в англоязычной литературе понимаются и ASIC, реализующие решение задачи на жёсткой логике, и цифровые сигнальные процессоры (DSP), и процессоры с расширенными системами команд (application-specific instruction-set processor, ASIP). Однако, в русскоязычной литературе под названием «проблемно-ориентированный процессор» понимаются именно специализированные процессоры, имеющие не стандартную систему команд, которая вместе с поддерживающей её аппаратурой создаётся для решения некоторой заранее известной и специализированной задачи.

ASIP являются компромиссом между универсальностью и программируемостью процессоров общего назначения, производительностью и энергетической эффективностью ASIC-решений, которые, обычно, серьёзно оптимизируются, но под конкретные приложения. Как правило, данный подход используется в технологии СнК. Архитектура ASIP включают статическую логику, в которую входит некоторый набор стандартных инструкций, и конфигурируемую логику для расширений, изменяемую в процессе функционирования или при синтезе архитектуры. Таким образом, запущенное на ASIP ПО может использовать как стандартные возможности процессоров общего назначения, так и улучшенные аппаратно.

Благодаря этому появляется возможность получать оптимальные (например, высокопроизводительные или энергетически эффективные) решения для конкретных, известных задач при невысокой стоимости конечного изделия. В отличие от аппаратных ускорителей на жёстко заданной логике, решения на основе ASIP обладают значительно большей гибкостью и лучше приспособлены для повторного использования. Правда, это достигается за счёт более трудоёмкого процесса проектирования, поэтому данный подход эффективен при выпуске больших партий устройств.

Задача проектирования ASIP не ограничивается созданием набора инструкций процессорного ядра и оптимизацией их аппаратных

реализаций. Из-за специфичности получаемого вычислительного ядра требуется также создавать и вспомогательный инструментарий, позволяющий компилировать для полученного процессора предназначенное для запуска на нём ПО, а крайне желательно — ещё и симулятор, позволяющий отлаживать как ПО, так и саму аппаратуру.

Проектирование ASIP включает следующие шаги [24]:

1) Исследование пространства возможных архитектурных решений;
2) Реализация архитектуры;
3) Создание ПО для полученной архитектуры;
4) Системная интеграция.

Поиск архитектуры для реализации является итеративным процессом, который требует наличия таких инструментов, как ассемблер, линкер, компилятор и симулятор. Сутью задачи является профилирование приложения на различных архитектурах и поиск оптимального или стремящегося к нему варианта. При этом весь инструментарий полностью завязан на архитектуру и должен проектироваться заново после каждого её изменения. Вариант, когда инструменты для разработки ПО и процессорная архитектура создаются раздельно, могут повлечь за собой критические ошибки и несоответствия. Поэтому ASIP в большинстве случаев моделируются при помощи более абстрактных языков описания архитектуры (Architecture Description Languages, ADL), которые содержат информацию не только об архитектуре аппаратуры процессора, но и о его наборе инструкций, модели памяти и других аспектах, важных для программирования.

Более детально применяемые при создании ASIP методы и подходы можно рассмотреть на конкретном примере — архитектуре и технологии NISC [25] и поддерживающем её инструментарии, который включает, в том числе, и язык описания архитектуры GNR. При этом хоть NISC-процессоры, с одной стороны, и нельзя отнести к ASIP — ведь в них отсутствуют инструкции процессора как таковые, и они являются скорее ASP в широком смысле — вся технологическая цепочка фактически повторяет типичную цепочку создания ASIP, просто убирается один промежуточный шаг.

2.5.1 Пример: технология NISC и язык GNR

No-Instruction-Set-Computer (NISC, [25,26]) – статически планируемая горизонтально-программируемая вычислительная архитектура. Вместе с поддерживающим её инструментарием и языком описания архитектуры Generic Netlist Representation (GNR, [27,28]) разрабатывалась в Центре для Встраиваемых Компьютерных Систем (CECS) в Калифорнийском

университете в Ирвайне, в процессе поиска сбалансированного подхода к проектированию ASP, который обеспечивал бы высокую эффективность как синтеза аппаратуры, так и компиляции программ. Для этого было решено отказаться от использования в процессорах инструкций, в результате чего и была создана технология NISC.

Она состоит в следующем: для описания тракта обработки данных и управляющего им контроллера используется формальный язык GNR; потактовый компилятор заменяет собой декодер инструкций — при помощи имеющейся у него детальной информации о структуре тракта данных он генерирует управляющее слова для каждого такта процессора, полностью контролируя его поведение; наконец, скомпилированные программы записываются в память в сжатом состоянии, что позволяет уменьшить их объём до 16% в сравнении с подобными программами для RISC архитектуры.

Необходимость такого подхода его создатели обуславливают слишком медленным внедрением технологий ASIP, в основном из-за комплексности и сложности задачи создания подобных процессоров. Помимо нахождения наборов инструкций, позволяющих увеличить производительность, для каждой из инструкций необходимо учитывать её компиляцию и эффективную аппаратную реализацию. Язык описания аппаратуры (ADL) должен автоматизировать как можно большую часть этой работы, при этом без потерь в качестве и удобстве. Большинство существующих ADL фокусируются либо только на поведенческом аспекте, либо только на структурном. Поведенческие ADL удобны для компиляции (подробно описывается поведение каждой инструкции) и одновременно создают затруднения при оптимизации аппаратуры. Например, разделение ресурсов и выбор путей перенаправления (forwarding path customization) осложнены или вообще невозможны при подобном подходе. При использовании структурных ADL детали реализации контроллера (декодера инструкций) и тракта обработки данных разделены, и это даёт широкие возможности по оптимизации аппаратуры, но при этом описание декодера инструкций усложняет подобные языки и влечёт за собой ошибки. Кроме того, затруднено извлечение поведения конкретных инструкций для использования в языках высокого уровня. Таким образом, в традиционных подходах либо структура для синтеза генерируется из поведений инструкций, либо эти поведения для компиляции извлекаются из структурных описаний. Основные же причины, по которым в процессорах используются инструкции — переносимость исполняемых кодов, уменьшение их размеров и упрощение разработки компиляторов — не критичны при создании специализированных процессоров, использовании ADL для синтеза компиляторов и при архивировании программ. Поэтому от

инструкций можно отказаться вообще, убрав из цикла разработки промежуточную и не обязательную для достижения конечного результата ступень.

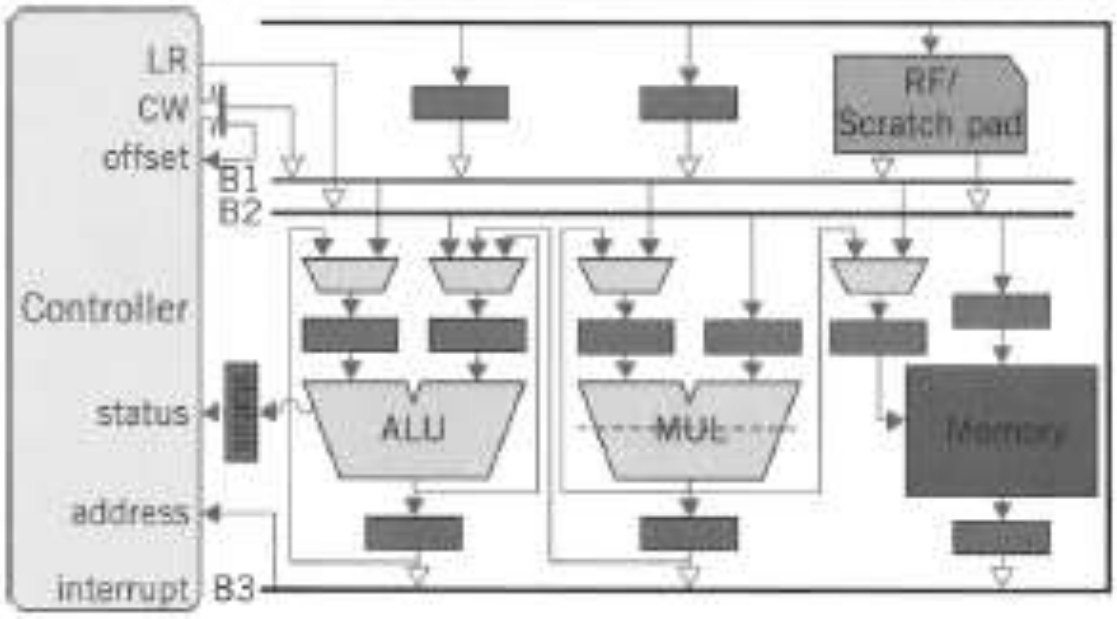


Рис. 25. Простейшая NISC архитектура [25].

Архитектура NISC является вариацией микрокодируемой архитектуры (microcoded architecture), при этом в отличие от неё, описываются не просто операции передачи данных между регистрами, а вообще поведение каждого низкоуровневого компонента тракта данных. Реализация эффективной декомпрессии добавляется в контроллер в ходе его синтеза. NISC архитектура похожа на архитектуру VLIW, но гораздо проще — например, в TMS320C6x загрузка, распаковка и декодирование инструкции занимает шесть ступеней конвейера, в то время, как подобная NISC реализация требует только одну ступень, заключающуюся в разархивации.

Архитектура NISC состоит из контроллера и тракта данных. Контроллер на каждом такте выдаёт заранее рассчитанные компилятором управляющие сигналы для всех аппаратных компонентов. И контроллер, и тракт данных могут быть конвейеризованы.

NISC компилятор генерируют значения «0», «1» или «X» (любое). «X» используются для уменьшения размера кода или для оптимизации энергопотребления, и означает, что данному элементу в текущем цикле может быть присвоено любое значение без влияния на поведение программы.

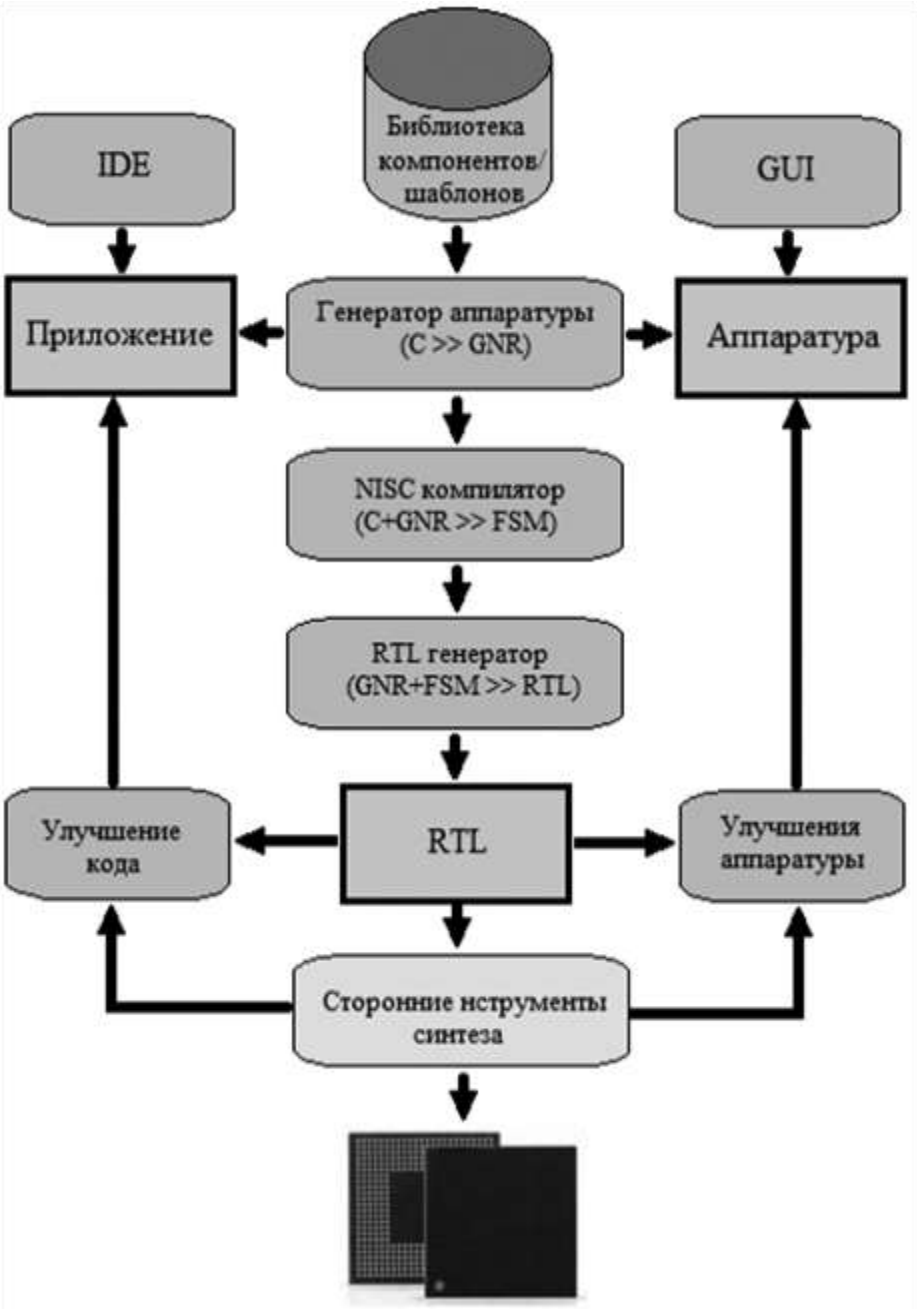


Рис. 26. Разработка специализированного NISC процессора для заданной задачи.

Контроллер генерируется после компиляции программы для конкретного выбранного тракта данных. На вход компилятора подаются и программа, и описание тракта данных. Тракт данных может быть создан с применением различных технологий (заимствование готовых IP блоков, автоматическая генерация из алгоритма программы). Для его описания используется GNR, который представляет тракт данных в виде списка связей компонентов и набора атрибутов для них.

NISC компилятор отображает программу на заданный тракт данных и генерирует набор управляющих сигналов для каждого цикла. Генератор

RTL синтезирует контроллер из информации, полученной от компилятора, и использует сведения о тракте данных для окончательно синтезируемого RTL описания (на языке Verilog), используемого для синтеза и симуляции. После получения информации о «таймингах», энергопотреблении и объёме полученной реализации, тракт данных может улучшаться и дополняться. При этом ввиду того, что нет надобности в разработке наборов инструкций (так как компилятор автоматически анализирует тракт данных и извлекает возможные операции), улучшение может осуществляться достаточно быстро.

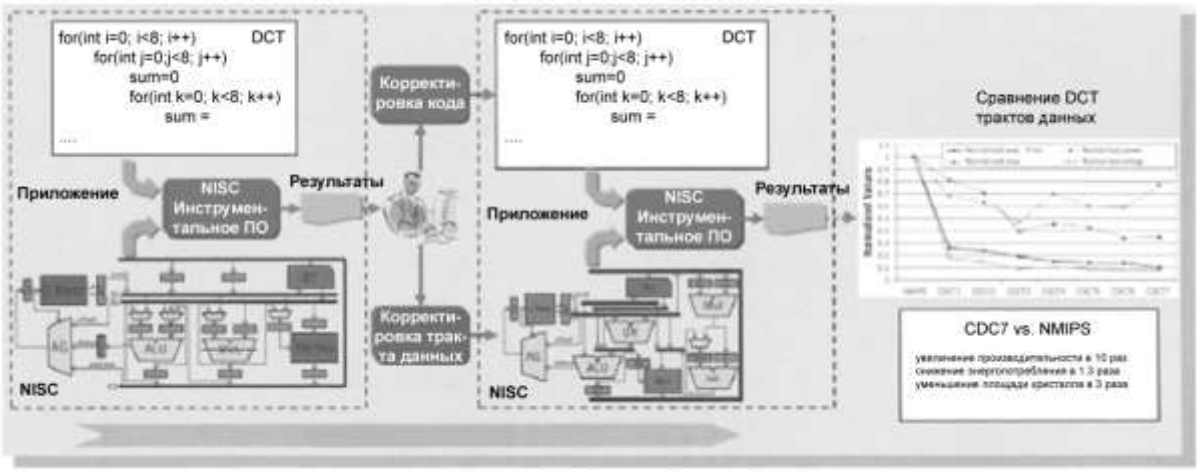


Рис. 27. Процесс создания NISC [25].

С технологией NISC возможно генерировать произвольные архитектуры для решения конкретных задач, подобно инструментам высокоуровневого синтеза, с итеративным улучшением результатов. В каждой итерации можно фокусироваться только на одном из показателей качества (например, сначала добавить параллелизм, затем увеличить временные характеристики тракта данных, после этого применить оптимизацию по объёму). Последовательное применение оптимизаций позволяет им не влиять друг на друга, тем самым усложняя. В конце разработчик может выбрать наиболее подходящий вариант из всех сгенерированных, который удовлетворяет лучше всего заданным требованиям.

Производительность и энергопотребление тракта данных в NISC архитектуре могут быть улучшены несколькими способами:

- регулированием числа и типа функциональных модулей, числа портов регистровых файлов, конвейерных регистров, путей перенаправления для соответствия характеристикам задачи;
- добавлением произвольных функциональных модулей;
- добавлением внешних NISC акселераторов.

Современные системы состоят из множества вычислительных ядер, связанных друг с другом шинами данных, разделяемой памятью и сетями. Успешная парадигма разработки должна учитывать эти особенности и не ограничиваться описанием отдельных ядер, а предусматривать упрощение их последующей интеграции. Язык GNR предназначен для описания подобных систем.

В NISC поведение системы описывается на языке высокого уровня (например, C), и компилируется для заданного тракта данных. Структурное описание используется не только для генерации аппаратуры, но и для окончательной доводки тракта данных (заключающейся в добавлении пропущенных компонентов и связей), валидации (предотвращении часто встречающихся ошибок, например, неверного соединения портов), оптимизации (изменения структуры связей для улучшения качества разрабатываемой архитектуры), и компиляции (размещения переменных и операций заданной программы в функциональных блоках и хранилищах, так, чтобы высокоуровневое поведение могло исполняться на заданном тракте данных). Все эти действия требуют извлечения информации о тракте данных и его компонентах. Этого можно достигнуть, надстроив обычный HDL так, чтобы из описания аппаратуры можно было извлечь необходимую информацию, что и реализовано в GNR: он включает в себя структурные детали — типы элементов (предоставляют инструментам информацию о том, что конкретно каждый структурный элемент собой представляет, без необходимости анализировать его поведение), и информацию для конкретных инструментов — аспекты. Для иллюстрации этой концепции на Рис. 28, Рис. 29 сначала представлено описание NISC архитектуры на структурном уровне, далее, представление той же архитектуры с добавлением информации о типах (появляется форма и цветовое выделение стандартных элементов), и в конце в упрощённом виде, когда используется автоматическое завершение частично описанных трактов обработки данных («парсер» GNR сам генерирует не заданные явно соединения, связывая порты, назначение которых известно).

Типичная мультиядерная система состоит из вычислительных элементов, поведение которых описывается на C. В GNR такие элементы представлены как *behavioralPE* — поведенческие вычислительные элементы. Они могут представлять собой специальную аппаратуру, процессоры общего назначения или некоторую специализированную NISC, для представления которой используется элемент *NiscArchitecture*. Список связей внутри *NiscArchitecture* либо генерируется автоматически, либо создаётся вручную. Полученные после компиляции управляющие слова хранятся в управляющей памяти (Cmem), которая иногда сама

достаточно сложна и требует синтеза. На Рис. 30 простой пример системы с двумя элементами обработки, шиной и арбитром.

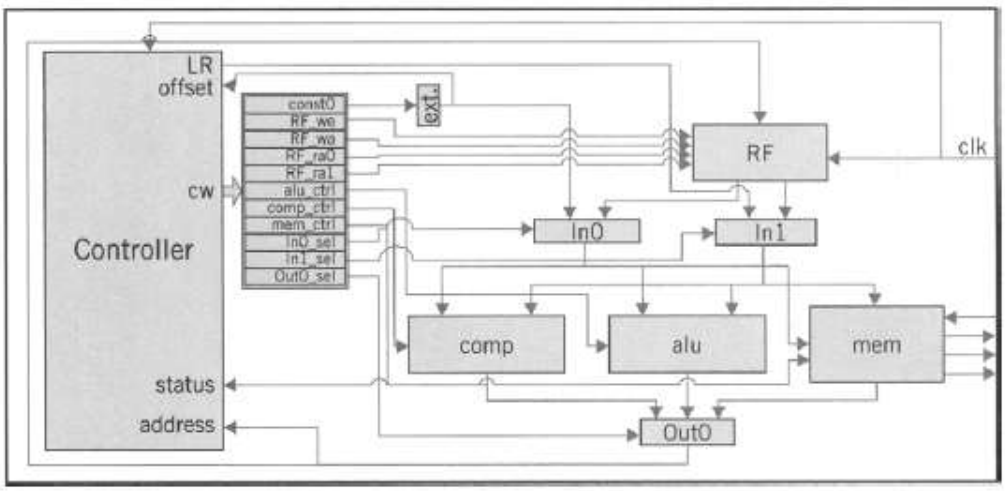


Рис. 28. Блок-диаграмма NISC на структурном уровне [25].

Ключевой особенностью GNR являются типы и аспекты, которые он добавляет всем элементам и портам. Элемент *x* представляется как (*Tx*, *Px*, *Cx*, *Lx*, *Ax*), где *Tx* — тип компонента, *Px* — набор управляющих портов, *Cx* — набор компонентов внутри *x*, *Lx* — набор внутренних связей, *Ax* — набор аспектов, описывающих поведение *x* для различных инструментов.

Тип компонента *Tx* может быть: регистр, регистровый файл, шина, мультиплексор, буфер с тремя состояниями, функциональный модуль, память, контроллер, *NiscArchitecture*, *behavioralPE*, модуль). Последние четыре типа компонента — иерархические, включают в себя внутренние списки соединений, остальные — базовые RTL компоненты.

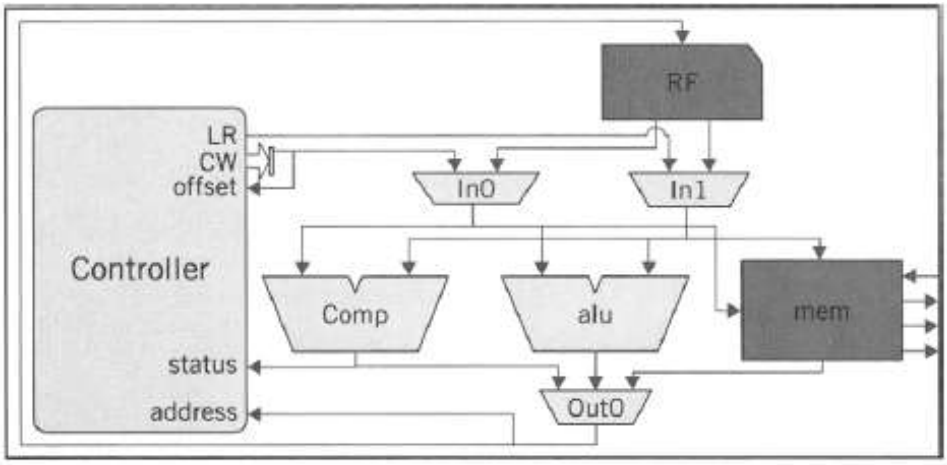


Рис. 29. Упрощённая блок-диаграмма [25].

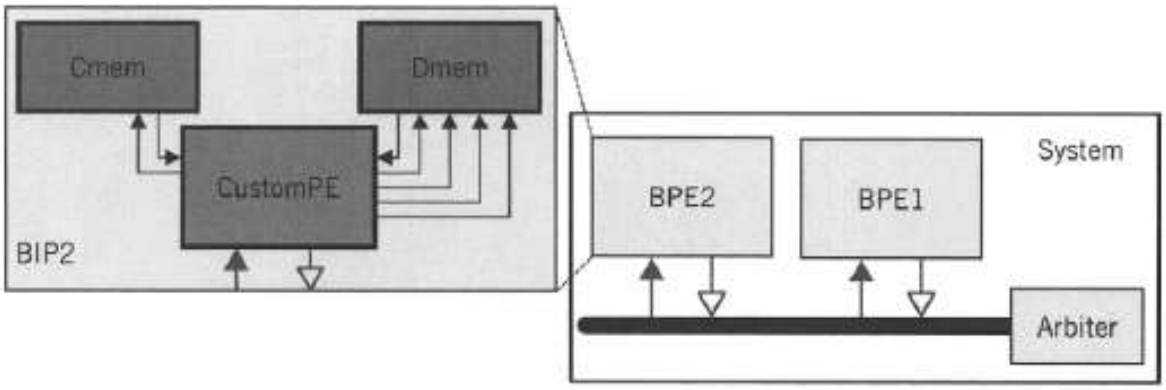


Рис. 30. Система с двумя элементами [28].

Каждый порт *p* из набора *Px* имеет разрядность *Bp* и тип *Op*. Тип может быть: порт тактирования, управляющий, вход, выход, *cwPort* (использующийся для управления компонентами в *NiscArchitecture*).

Аспекты компонента зависят от его типа. Аспекты могут быть: аспект генерации (высокоуровневого синтеза), аспект компиляции, аспект симуляции, аспект синтеза RTL. Аспект генерации определяет, каким способом генерируется список внутренних соединений. Аспект компиляции обычно описывает связь между поведением компонента и операциями на C или функциями высокоуровневой программы. Аспекты симуляции и синтеза обычно содержат описание компонента на HDL или информацию, необходимую для генерации hardwired core (память, делитель и т.д.), ориентированного на конкретную платформу. Для некоторых компонентов, если аспекты не заданы явно, они генерируются.

У *NiscArchitecture* есть следующие дополнительные параметры: *Y* представляет собой управляющий элемент NISC, который подаёт на входы управления тракта обработки данных скомпилированные управляющие слова. Аспект компиляции *NiscArchitecture* моделируется как *CAY* = (*freq, CNST, ORDER, sPt, fPt*). *Freq* — частота тактирования *NiscArchitecture*, используется компилятором для генерации управляющих слов, при этом учитываются также задержки компонентов. *CNST* — поля с неизменным значением, используются для переходов и других операций с постоянным операндом. *ORDER* определяет порядок следования постоянных и управляющих полей в управляющем слове. *sPt, fPt* — указатель стека и указатель на кадр (frame pointer).

BehavioralPE — компонент, чьё поведение описано на C, и может быть исполнено специализированным процессором или процессором общего назначения. Аспект компиляции *behavioralPE* определяет набор заголовков и исполняемых файлов, которые должны на нём выполняться. Если такой компонент реализуется как NISC, список его связей включает *NiscArchitecture* и, если необходимо, подсистему памяти. Компилятор

NISC компилирует код C приложения для тракта данных *NiscArchitecture*. *BehavioralPE* также может представлять собой обычный процессор с инструкциями или специализированный вычислитель, в таком случае аспект синтеза — это обычно описание ядра, полученное от сторонних разработчиков и использующее соответствующие инструменты.

Формальное и типизированное описание GNR позволяет указывать правила для проверки корректности списков связей, что позволяет облегчить разработку, исключив некоторые ошибки ещё до симуляции. В зависимости от типа элемента, правила ограничивают количество и тип портов, входящие в него элементы, а также их подключение. Правила делятся на общие и специфичные для NISC.

Общие правила:

- порты тактирования подключаются только к портам тактирования;
- выходные порты подключаются только к входным;
- только одна связь дозволена каждому биту каждого порта, кроме компонентов шинного типа.

Правила специфичные для NISC:

- каждый элемент *NiscArchitecture* имеет только один элемент типа контроллер;
- только элемент типа контроллер может иметь один и только один порт типа *cwPort;*
- чтобы обеспечить компилируемость, каждый элемент *NiscArchitecture* имеет, по крайней мере, один компонент типа «регистровый файл», размер которого может быть нулевым, если ни один из регистров не используется;
- разрядность управляющего порта *cwPort* должна равняться сумме разрядностей всех управляющих входов архитектуры и всех постоянных полей *CNST*;
- управляющие связи в *NiscArchitecture* заданы между *cwPort* и управляющими входами компонентов архитектуры.

Синтаксис GNR использует XML для описания моделей вычислительных элементов. На рисунке представлена XML Схема, иерархически описывающая свойства для *NiscArchitecture*.

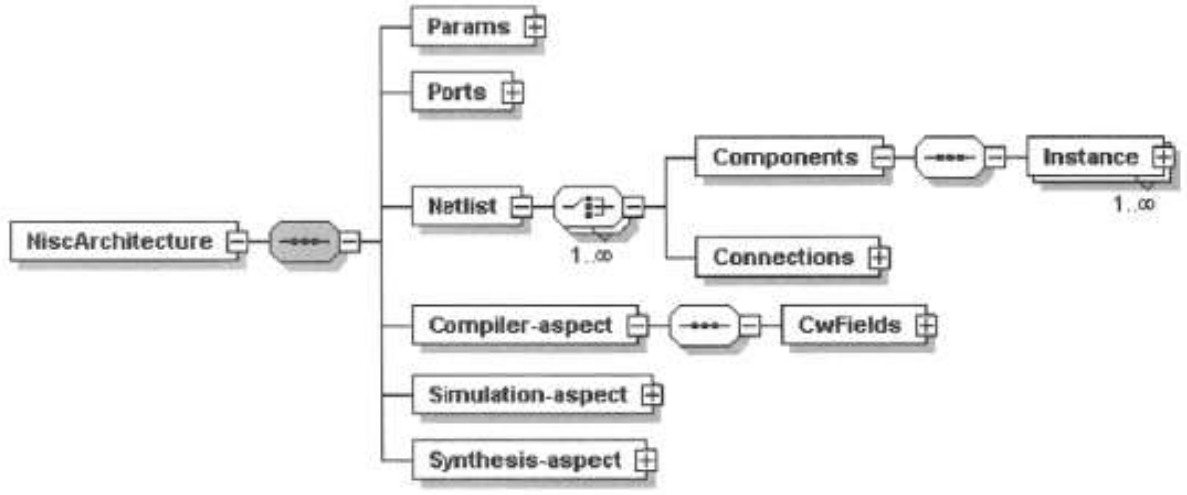


Рис. 31. Схема XML [28].

Аспект компиляции каждого базового компонента может определять одно или больше машинных действий (machine actions), низкоуровневой функциональности. Они бывают четырёх типов: чтение, запись, перемещение и выполнение. Чтение и запись описывают доступ к регистровым хранилищам, тогда как перемещение описывает перемещение данных от одного порта к другому. Выполнение открывает доступ к операциям, которые компонент может совершать. Машинные действия могут быть мультитактовыми, а также конвейеризированными. Базовые компоненты делятся на 4 группы: регистры и регистровые файлы позволяют только чтение и запись; мультиплексоры, буферы с тремя состояниями и шины — только перемещение; функциональные элементы и память — только выполнение или перемещение; контроллер — только набор заданных выполняемых операций.

Иерархические компоненты используются для упрощения конструирования новых компонентов из уже доступных, для чего используется специальный тип компонентов — модули. Иногда для часто используемых модулей с целью ускорения компиляции можно задавать аспект компиляции, описывающий внутреннее поведение модуля — тогда компилятор не будет каждый раз спускаться вниз по иерархии для генерации управляющих битов.

Компонент *NiscArchitecture* — верхний модуль в иерархии, предоставляющий всю информацию о NISC архитектуре. Порты этого компонента используются для подключения к остальной системе.

В дополнение к информации о связях компилятору может поставляться ещё информация, полученная из аспекта компиляции. Не вся такая информация обязательна, и в случае если такой необязательной информации нет, некоторые дополнительные возможности компилятора

просто не доступны. Аспект компиляции объявляет один компонент памяти в тракте данных как основную память. Можно использовать и другие модули памяти, но в таком случае пользователь должен будет напрямую обращаться к ним, используя предварительную привязку (prebinding). Все нормальные операции доступа к памяти (чтения/записи) и операции работы со стеком (push/pop) привязаны к основной памяти. Если аспект компиляции не описывает такой памяти, эти операции будут недоступны, и компилятор сгенерирует ошибку, если они встретятся в программе. Для вызова функций обязательно должны быть указаны указатель стека (указывает на конец стека) и указатель кадра, frame pointer (указывает на начало стека функции).

Аспект компиляции также определяет период тактирования компонента в базовых периодах тактирования, которое вместе с данными о времени, требуемым компонентом на обработку информации, позволяет компилятору определить, выполнится операция за один или за несколько тактов.

Структура управляющих слов задаётся в аспекте компиляции, определяя разрядность постоянных полей и порядок управляющих битов каждого компонента в управляющем слове. Так как управляющие порты типизированы, эти поля могут добавляться автоматически, без участия пользователя.

GNR может использоваться для сбора информации на битовом уровне. На системном уровне модели, обрабатывающие элементы, создаются и связываются друг с другом при помощи различных парадигм связи: очередей, разделяемых блоков памяти, шин, сетей. Каждый элемент должен иметь минимум один интерфейс коммуникации, который должен соответствовать парадигме связи, к которой он относится. Исполняемый код каждого элемента обработки должен иметь драйверы, которые соответствуют выбранному интерфейсу. Модель системы на уровне «пинов» и приложения могут быть написаны вручную, или могут генерироваться из модели уровня транзакций, где поведения компонентов передают информацию через абстрактные каналы.

Обычно, для процессоров драйвера пишутся на ассемблере. Специализированные же модули обработки информации вообще, и NISC архитектуры в частности, не имеют инструкций и, следовательно, ассемблера. В качестве альтернативы набор инструментов NISC предоставляет привязанные к аппаратуре (pre-bound) функции и переменные, предоставляющие детальный низкоуровневый контроль на аппаратном уровне для разработчиков программного обеспечения. Они имеют обычный синтаксис языка C, однако компилятор назначает им специально заданные аппаратные ресурсы. Вместо обычного вызова при

помощи jump операций, pre-bound функции исполняются как просто операции; переменные же привязываются к заданным регистрам, а не к памяти или стеку.

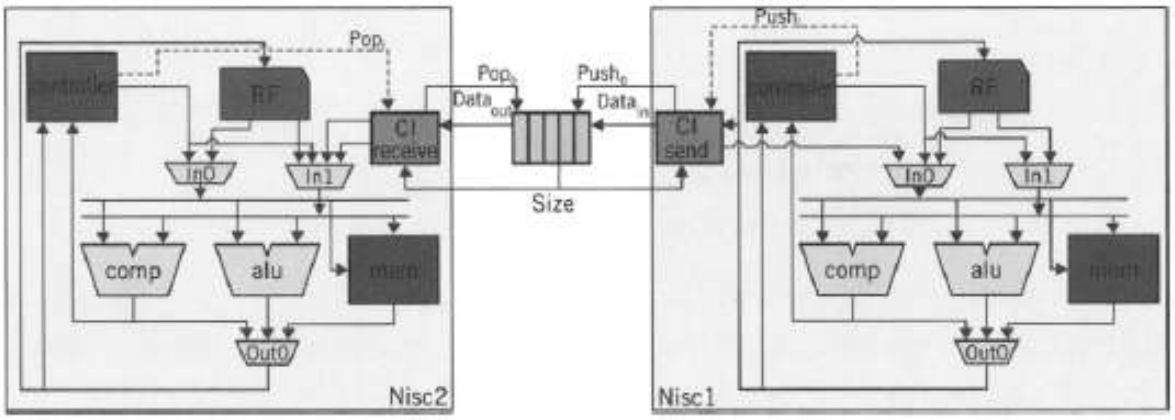


Рис. 32. Межпроцессорное взаимодействие [28].

Сигнатуры таких функций определяет и управляющие значения, и порты, использующиеся как входы и выходы. Сигнатуры объявляются в GNR и используются компилятором для отображения высокоуровневых вызовов этих функций из C на соответствующие RTL операции.

В примере (Рис. 32) показывается, как GNR используется для описания коммуникации между двумя модулями NISC архитектурами через очередь. Один модуль добавляет данные в очередь, другой забирает. Первый поддерживает функции *Push* и *Size*, второй — *Pop* и *Size*.

```
1 void send(int N, int* data)
2 {
3   while(__$ci_Size()!=0);
4   for(i=0;i<N; i++)
5     __$ci_Push(data[i]);
6 }
```

(a)

```
1 void receive(int N, int* data)
2 {
3   for(i=0;i<N; i++){
4     while(__$ci_Size()==0);
5     data[i++] = __$ci_Pop();}
6 }
```

(b)

Рис. 33. Пример описания драйверов: отправляющего и получающего данные [28].

Имена навязанных функций, доступные программисту, генерируются комбинированием имени экземпляра интерфейса коммуникации в тракте данных с названиями функций, указанных в GNR модели интерфейса.

RTL генератор, входящий в набор инструментов NISC, генерирует аппаратное описание системы из модели на битовом уровне, полученной из GNR.

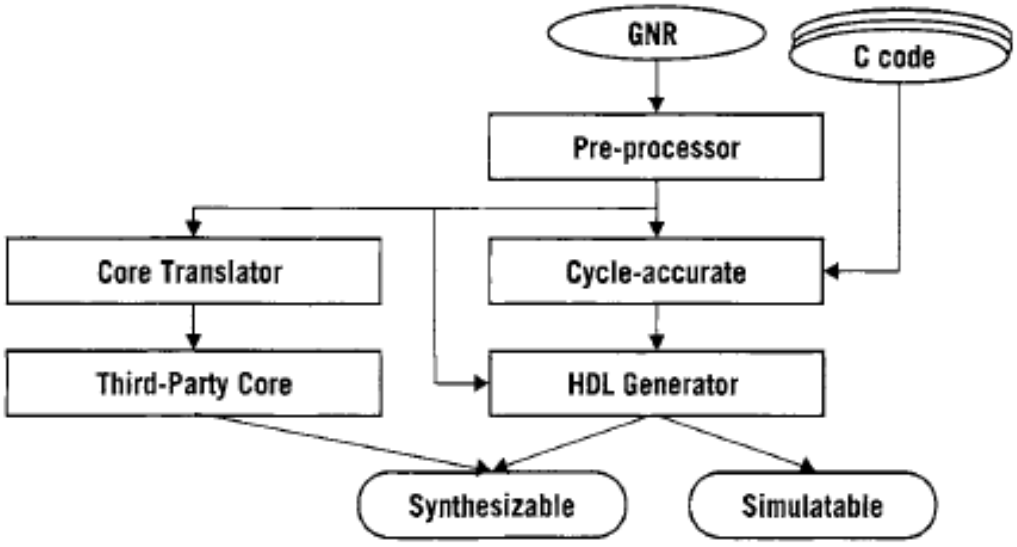


Рис. 34. Логический путь генерации HDL из описания GNR и кода приложения на C для его процессорных элементов [28].

Препроцессор проверяет синтаксис заданного GNR описания, используя XML схему для GNR. Далее, он составляет список соединений из параметров компонентов, не указанных явно сигналов тактирования и управления, и добавляя управляющие поля. Затем, семантическая корректность списка связей проверяется через правила GNR. На выходе получаются синтезируемые и симулируемые RTL-коды.

После компиляции компилятор генерирует содержание блоков памяти данных и памяти программ. Генератор HDL использует GNR и результаты компиляции, чтобы сгенерировать окончательные симулируемые и синтезируемые HDL коды.

Для улучшения производительности, размера и энергопотребления генерируемых IP ядер разработчики могут использовать hardwired ядра. Это требует соответствующей поддержки в ADL и RTL генераторах. Аспект синтеза GNR компонентов позволяет вызов программ-трансляторов для генерации подходящих данных для сторонних генераторов ядер. Например, для Xilinx FPGA генератор ядер (*LogicCore*) требует XCO файлы, которые описывают свойства ядра. Для других ядер дополнительная информация должна предоставляться в соответствующих форматах. Информация о ядрах извлекается из GNR описания и предоставляется программам-трансляторам для генерации файлов сторонних ядер для заданной платформы. Трансляторы обычно имеют разную реализацию для разных платформ и позволяют минимизировать платформозависимую работу, а тем самым увеличить производительность. Полученные ядра комбинируются со сгенерированным HDL кодом для последующего синтеза.

В NISC технологии контроллер генерируется автоматически после компиляции приложения под заданный тракт обработки данных, т.е. после определения очерёдности выполнения операций — планирования (scheduling), и назначения их к конкретным аппаратным ресурсам — связывания (binding).

На вход компилятору подаётся зафиксированный тракт обработки данных, алгоритм решаемой задачи и данные о тактировании. Результатом является набор элементарных операций — передач данных, выполнения функций функциональными блоками, исходя из которых затем генерируются управляющие биты.

Основная идея алгоритма компиляции — выполнение команд параллельно. При анализе компилируемого приложения оно делится на базовые блоки. Базовые блоки разбиваются на поддеревья — последовательности операций с произвольным числом входов и одним выходом. Сначала берутся последние операции каждого такого поддерева и анализируется, какие другие операции должны им предшествовать. Планирование и связывание определяют, когда и где необходим результат каждой из операций последовательности. В результате естественно («native») реализуются конвейеризация, перенаправление данных между операциями, и организация очередей. Определяется, откуда берутся исходные данные: переменные и константы привязываются к различным хранилищам.

Без каких-либо оптимизаций и модификации кода тестов для увеличения параллелизма были получены результаты, сравнимые с результатами, полученными при использовании gcc.

Также особого внимания заслуживает обработка прерываний в NISC технологии [28]. Ввиду того, что в NISC архитектуре промежуточные данные могут храниться во внутренних регистрах тракта обработки данных, а также возможно выполнение операций продолжительнее одного такта, стандартный подход к обработке прерываний не годится. Это решается разбиением на базовые блоки, каждый из которых должен читать исходные данные из памяти или регистрового файла и записывать обратно результаты. NISC контроллер проверяет прерывания только когда выставлены биты соответствующие jump-операции, т.е. в конце базовых блоков. После выполнения jump-операции поток выполнения программы переходит либо к указанному в jump'е адресу, либо к процедуре обработки прерывания. В последнем случае адрес из jump'а передаётся в прерывание и используется им как возвращаемый адрес. Данный подход может увеличить задержку на обработку прерываний, но статистически большие базовые блоки встречаются редко, и, если задержка критична, компилятор

может разбивать их на меньшие блоки. Кроме того, можно оптимизировать архитектуру для ускорения обработки именно прерываний.

Оптимизация объёма кода в NISC осуществляется следующим способом [29]: создаются массивы уникальных значений и управляющие слова заменяются на адреса двоичных последовательностей, из которых они состоят.

NISC компилятор может генерировать значения: «0», «1» или «X» (не имеет значения) для каждого бита управляющего слова. Для повышения эффективности компрессии можно просто заменить все «X» на «0», или подобрать значения так, чтобы минимизировать количество уникальных слов в программе. Благодаря подобному подходу, код сжимается примерно в 3 раза.

Для уменьшения энергопотребления «X» можно задать так, чтобы минимизировать количество переключений, и тем самым уменьшить энергопотребление (взяв значения из предыдущих управляющих слов). Для некоторой универсальной архитектуры это позволяет уменьшить энергопотребление на 26%.

Чтобы одновременно понизить и энергопотребление, и объём кода, используется то, что большинство программ 90% времени выполнения проводят в 5-10% базовых блоков. Поэтому, можно применить оптимизацию по энергопотреблению (замену «X» на значения из предыдущих слов) к этим 5-10% блокам, а для оставшиеся — задать «X» такими, чтобы уменьшить объём кода.

Таким образом, технология NISC является мощнейшим и всесторонне продуманным инструментом. GNR позволяет описывать процессоры общего назначения, специализированные, а также мультиядерные NISC системы. В NISC технологии целевая архитектура является статически планируемой нанокодируемой архитектурой без заданного набора инструкций, благодаря этому GNR очень краток. Язык типизирован и объявляет дополнительные аспекты для компонентов, чтобы различные инструменты могли обрабатывать их корректно. Кроме того, GNR может использоваться для описания мультиядерных систем на уровне пинов. Язык может использоваться для исследований как на уровне ядра, так и на системном уровне, также как промежуточный шаг между моделью на уровне транзакций и RTL описанием. Технология позволяет генерировать машинный код и некоторую базовую архитектуру из обычного C-кода, есть возможность вручную улучшать сгенерированную архитектуру, оптимизируя её и отлаживая при помощи автоматически симулятора. В подобных процессорах можно реализовывать поддержку любых низкоуровневых функций и операций, а также прерываний. Возможно создание соединений с другими вычислительными блоками, добавление

готовых IP-ядер, использование сторонних инструментов для генерации реализаций аппаратуры. Однако, при этом возможности по автоматическому распараллеливания последовательных алгоритмов и генерации сложных компонентов тракта обработки данных весьма ограничены.

3 ВЫСОКОУРОВНЕВОЕ ПРОЕКТИРОВАНИЕ ВСТРАИВАЕМЫХ СИСТЕМ

3.1 Понятие высокоуровневого проектирования

Высокоуровневое, системное, архитектурное проектирование ВсС и СнК — проектирование, при котором производится рассмотрение объекта проектирования на системном уровне, особое внимание уделяется формированию цельного взгляда на организацию всех фаз вычислительного процесса. При высокоуровневом проектировании рассматривается как сам объект проектирования, так и собственно маршрут проектирования. Поставленные задачи решаются комплексно, с учётом взаимодействия подсистем между собой и с внешней средой и прочих влияющих на качество изделия факторов.

В англоязычной литературе понятия SLD (System Level Design) и HLD (High Level Design) в контексте ВсС и СнК часто упоминаются как синонимы понятий сопряжённого проектирования аппаратуры и программного обеспечения (Hardware/Softeware Co-Design) и ESL (Electronic System Level), и означают примерно одно: нисходящее проектирование от некоторых абстрактных представлений проектируемой системы к её реализации, без разделения на аппаратуру и ПО на верхнем уровне. Последнее вызвано спецификой проектирования специализированных ЭВМ и оправданностью именно таких подходов при их рассмотрении на системном уровне, хотя высокоуровневое проектирование вообще является более общим понятием, может включать несколько другие подходы и применяться к другим областям.

Целями высокоуровневого проектирования (далее, для краткости, будем применять аббревиатуру «SLD») ВсС и СнК являются:

- борьба со сложностью современных электронных систем;
- ускорение процесса проектирования;
- снижение рисков проектирования;
- ответ на вероятное в будущем замедление технологического прогресса (обеспечение дальнейшего увеличения производительности ВсС);
- повышение качества получаемых продуктов;
- повышение степени повторного использования решений;
- предоставление разработчикам возможности сосредоточения на решении конкретных прикладных задач, а не технологических проблем.

Задачи SLD включают:

- получение концепции решения целевой задачи, исходных спецификаций;
- организация вычислительного процесса;
- генерация архитектуры и микроархитектуры;
- оценка и выбор архитектурных решений;
- верификация архитектурных решений;
- создание спецификаций для этапа реализации.

Суть SLD кратко и доступно можно охарактеризовать известной поговоркой: «семь раз отмерь, один раз отрежь». При системном подходе предполагаются некоторые дополнительные усилия перед началом самой «работы» (точнее, перед тем, что обычно понимают под этим словом: перед созданием аппаратуры, написанием программ, и т.д.), направленные на то, чтобы впоследствии не пришлось всё переделывать из-за явных ошибок или недостаточной эффективности конечного изделия. Конечно, при достаточно простых задачах возможно опустить этот этап и делать сразу, «отмеряя на глаз». Но как только появляется необходимость создания чего-то действительно сложного и состоящего из многих составных частей, без хоть каких-то минимальных предварительных выкладок решение становится невозможным.

Функции и предназначение SLD иллюстрируются простым и наглядным примером. Предположим, имеется необходимость нарисовать рисунок (например, человека). Логичным и оправданным является сначала набросать примерные контуры изображения: фигуру, положение различных частей тела, черты лица — и только потом приступать к прорисовке деталей. В противном случае — например, если начать рисовать с ноги — может выясниться, что рисунок вылезает за пределы листа, могут обнаружиться диспропорции после завершения рисунка и т.д.

В случае, когда присутствует несколько исполнителей, подход «снизу-вверх» в рамках описанного примера становится серьёзной проблемой и источником ошибок. Если кто-то отдельно будет рисовать лицо, кто-то — фигуру, кто-то — фон или некие другие детали, на конечном этапе может выясниться, что различные фрагменты просто не стыкуются между собой, нарисованы различными стилями и в разной цветовой гамме и т.д. В случае же если сначала был сделан предварительный набросок, дающий представление об общей композиции, и были оговорены детали реализации, в идеальном случае проблем при интеграции отдельных составляющих частей вообще не должно возникнуть (на практике, конечно, предугадать всё очень сложно — но стремиться к этому, безусловно, надо). Аналогично описанному примеру, при проектировании электронных систем SLD позволяет значительно

уменьшить риски, количество концептуальных ошибок, и повысить качество конечного продукта.

Рис. 35. Y-модель [30].

Перед тем как говорить о проектировании на системном/архитектурном уровне, следует сначала договориться — что же понимается под *системным/архитектурным уровнем*. Так, в отношении проектирования интегральных микросхем, системным уровнем принято считать любой уровень, чей уровень абстракции выше RTL (это могут быть модели на уровне транзакций, а также поведенческие, алгоритмические или функциональные модели). «International Technology Roadmap for Semiconductors» (ITRS) в 2004 определила SLD как «уровень выше RTL, включающий проектирование аппаратуры и ПО». SLD был определен как состоящий из двух уровней: поведенческого (до разделения на аппаратуру и ПО) и архитектурного (после разделения). В отношении ВсС, в том числе и РВсС, в системный уровень должны входить спецификации всех составляющих систему компонентов, их связей и внутренней организации — всё, необходимое для описания устройства и поведения системы.

Y-модель [30] позволяет наглядно представить уровни абстракции для аппаратуры ВС (Рис. 35). Две верхние оси отображают поведенческий и структурный аспекты представления системы, нижняя – её физическую реализацию. С увеличением расстояния от начала оси увеличивается уровень абстракции. Фактически, крайние точки, изображённые на рисунке, и представляют собой системный уровень представления ВС.

Процесс проектирования, отображённый на Y-модель при помощи перемещений между осями, представлен на Рис. 36. Стрелки слева направо

соответствуют задачам синтеза на различных уровнях абстракции, стрелки справа налево – представляют собой анализ полученных решений. Далее, стрелки сверху вниз – генерация из описаний аппаратуры конкретных реализаций, стрелки снизу-вверх – извлечение описаний из существующих реализаций. Стрелки вдоль осей показывают увеличение уровня абстракции или детализацию, закруглённая стрелка – символизирует оптимизацию полученного структурного представления без изменения уровня абстракции.

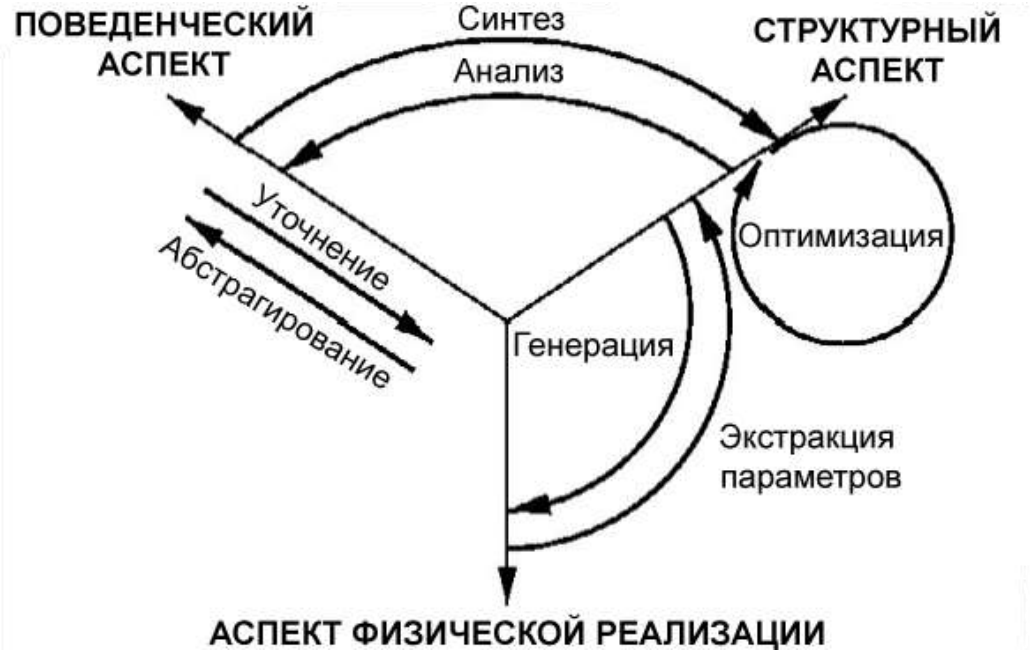


Рис. 36. Y-модель [30].

В Y-модели не представлен временной аспект, аспект коммуникаций между объектами и абстракции данных. Кроме того, хотя такое представление можно расширить и на представление ПО, но в явном виде оно в модели не представлено. Более точной, пусть и несколько громоздкой, является Rugby-модель. Она изображает идеализированный процесс проектирование системы от некоторой абстрактной идеи до реализации через повышение абстракций ортогональных аспектов: вычислений, коммуникаций, данных и временного аспекта.

SLD — это работа с абстрактными описаниями некоторой ВС, предполагающая их создание, анализ, модификацию и дальнейшее использование в процессе проектирования. Т.е., недостаточно просто создать некоторую модель будущей системы: это должна быть достаточно детальная и одновременно не очень громоздкая модель, отражающая именно те свойства, которые важны на системном уровне. Кроме того, это должна быть модель оптимального для данной задачи решения, действительно эффективного и адекватного имеющимся ресурсам. И,

наконец, при проектировании должны присутствовать понимание и инфраструктура, позволяющие связать высокоуровневое описание с процессом реализации и с оценкой промежуточных результатов, т.е., процесс проектирования должен завершиться получением именно предсказанного результата — что, собственно, является также задачей высокоуровневого проектирования.

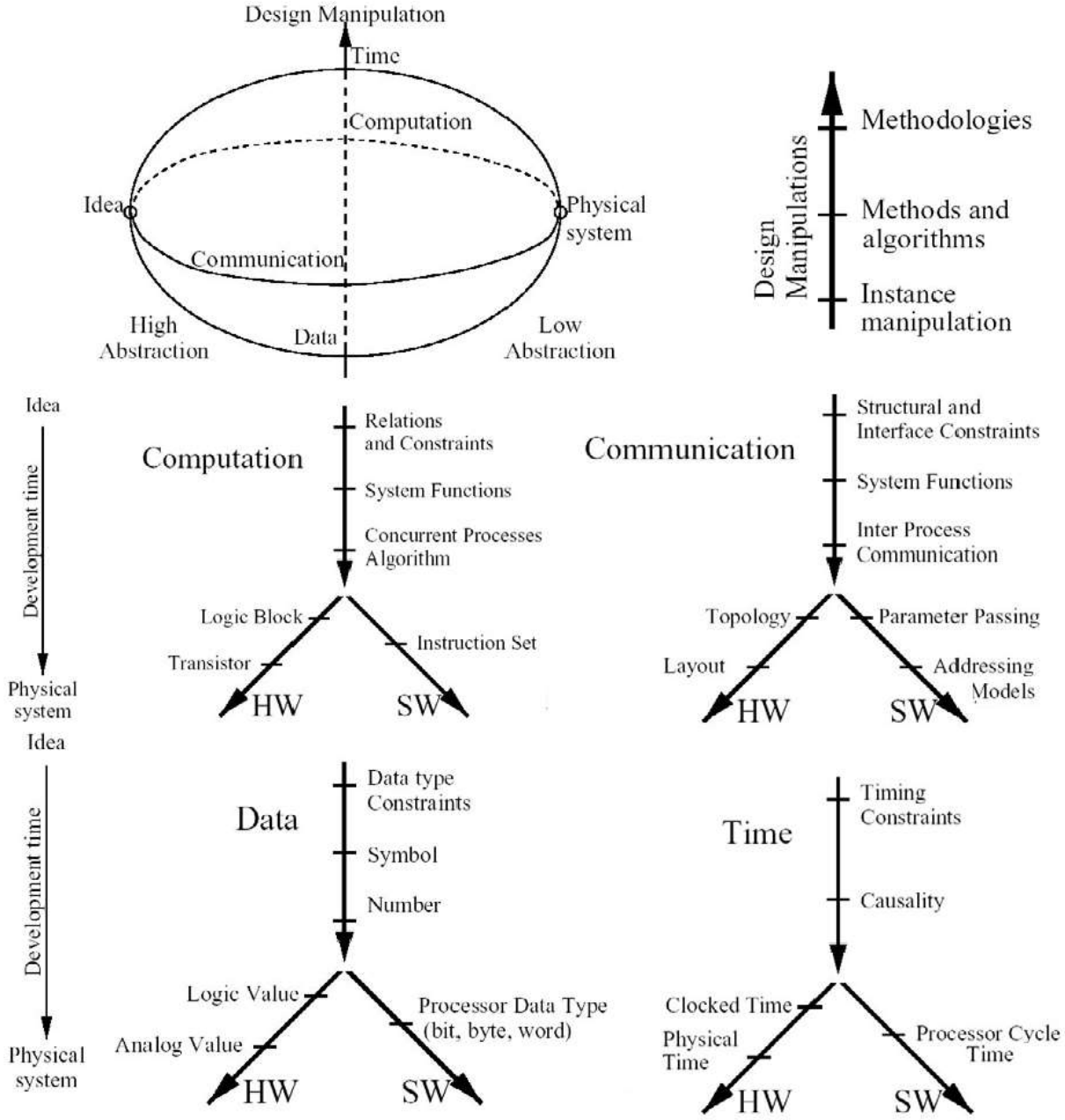


Рис. 37. Rugby-модель [31].

Ввиду большой сложности и комплексности решений, SLD невозможен без использования в процессе проектирования абстракций.

Архитектурные абстракции [1] позволяют описывать организацию и устройство объекта проектирования, модели вычислений — характеризуют поведенческие аспекты без привязки к конкретным ресурсам. Качественное проектирование должно также обязательно подразумевать учёт влияющих лишь косвенно на вычислительный процесс требований и ограничений — нефункциональных аспектов (надёжности, энергопотребления и т.д.).

Также важно понимать, что помимо выделения некоторой абстрактной сути из решаемой задачи и рассмотрения проектируемого технического изделия на архитектурном уровне, имеет смысл рассматривать на абстрактном уровне также и сам процесс проектирования, производить его формализацию. Формализация процесса проектирования, чёткая фиксация решений и последствий, которые они за собой повлекли, при всей своей сложности позволяет повысить уровень повторного использования — причём даже не столько технических решений, сколько методов их получения. Иначе говоря, при фиксации решений на системном уровне появляется возможность передавать опыт отдельных экспертов и знания об архитектуре системы.

В качестве примеров эффективности и необходимости такого подхода можно привести программы на языках C и ассемблера: программы на C, являющиеся более высокоуровневыми по отношению к программам на ассемблере, можно компилировать для различных целевых платформ, они являются кроссплатформенными. Ассемблерные коды, ввиду отличий в системах команд различных процессоров, можно переносить обычно лишь на ограниченное число вычислительных ядер. Также и системный подход позволяет оперировать достаточно высокоуровневыми понятиями при проектировании, имеющими потенциал для переноса решений на больший спектр проблем и областей. Кроме того, создание четкой системы понятий архитектурного уровня позволяет разработчику ВсС эффективно работать на уровне принципов организации ВС/вычислительного процесса, а не на уровне примеров реализаций.

Вообще выделение шаблонов, некоторых элементарных действий и шагов позволяет значительно облегчить труд человека (в идеале — за счёт создания средств автоматизации проектирования (САПР), полностью снимающих с проектировщика отдельные задачи). Появление САПР является прямым и наиболее очевидным следствием формализации процесса проектирования. САПР призваны снизить долю рутинной работы и уменьшить количество ошибок, допускаемых из-за присутствия в процессе проектирования человеческого фактора. Все это помогает сделать соответствующую технологию доступной для использования в индустрии, так как при этом увеличивается вероятность получения продукта заданного качества. В области проектирования встраиваемых систем

поддержку со стороны САПР можно увидеть только в отдельных областях, в основном – для создания СнК и печатных плат.

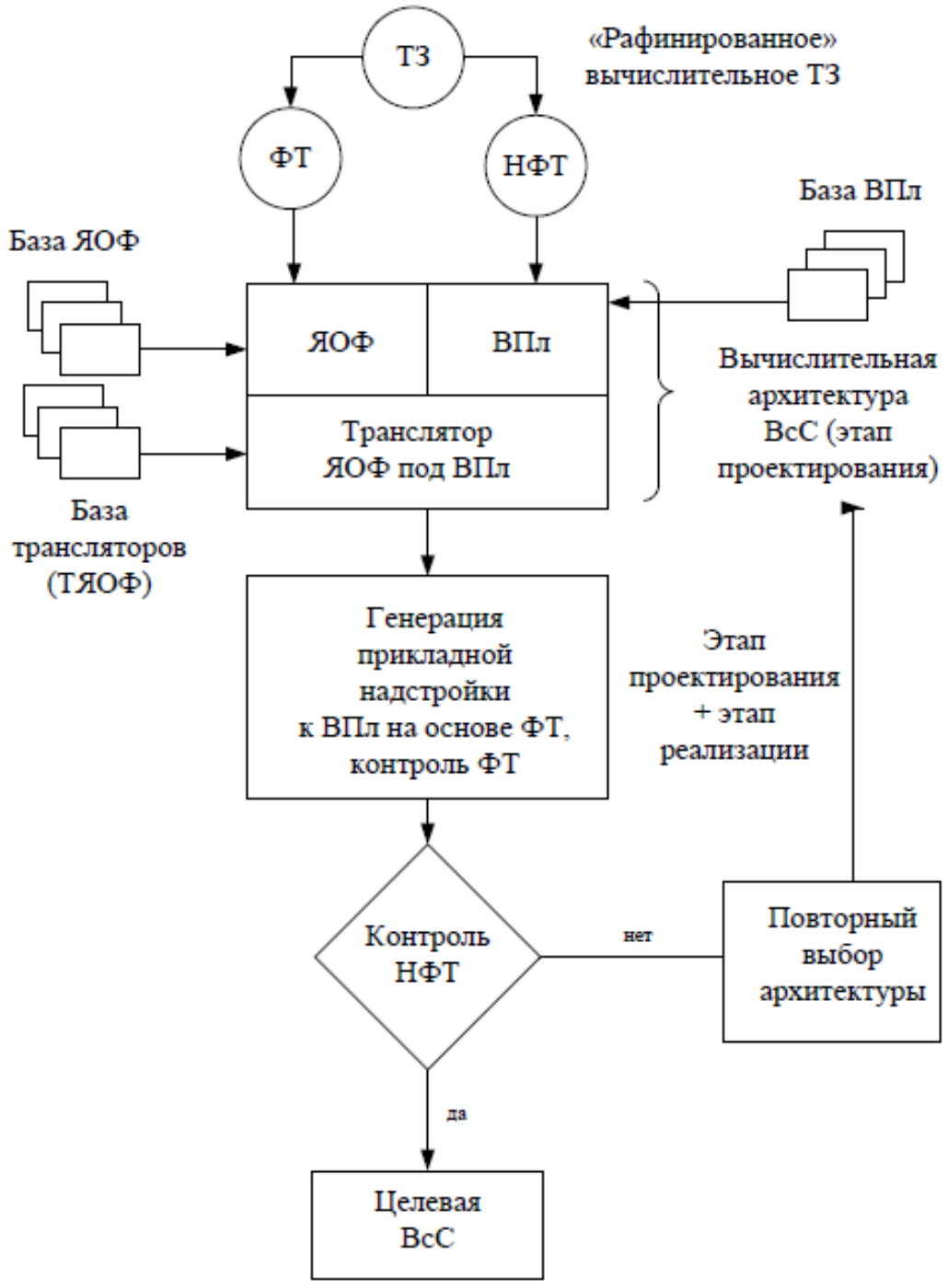


Рис. 38. Шаблоны проектирования ВсС на основе традиционной схемы проектирования [61].

Примеры формализации маршрута проектирования и выделения из него шаблонов приведены на Рис. 38, Рис. 39. Аббревиатуры ФТ, НФТ обозначают функциональные и нефункциональные требования, ЯОФ /

ТЯОФ – языки описания функциональности / трансляторы ЯОФ, ВПл – вычислительная платформа.

Очевидно, что SLD влечёт за собой большие затраты на этапе анализа и предварительного проектирования. Но они окупаются ввиду роста качества конечного продукта, снижения рисков. В первую очередь, конечно, это справедливо для сложных комплексных систем — однако практика показывает, что подход оправдан также и для простых, казалось бы, задач.

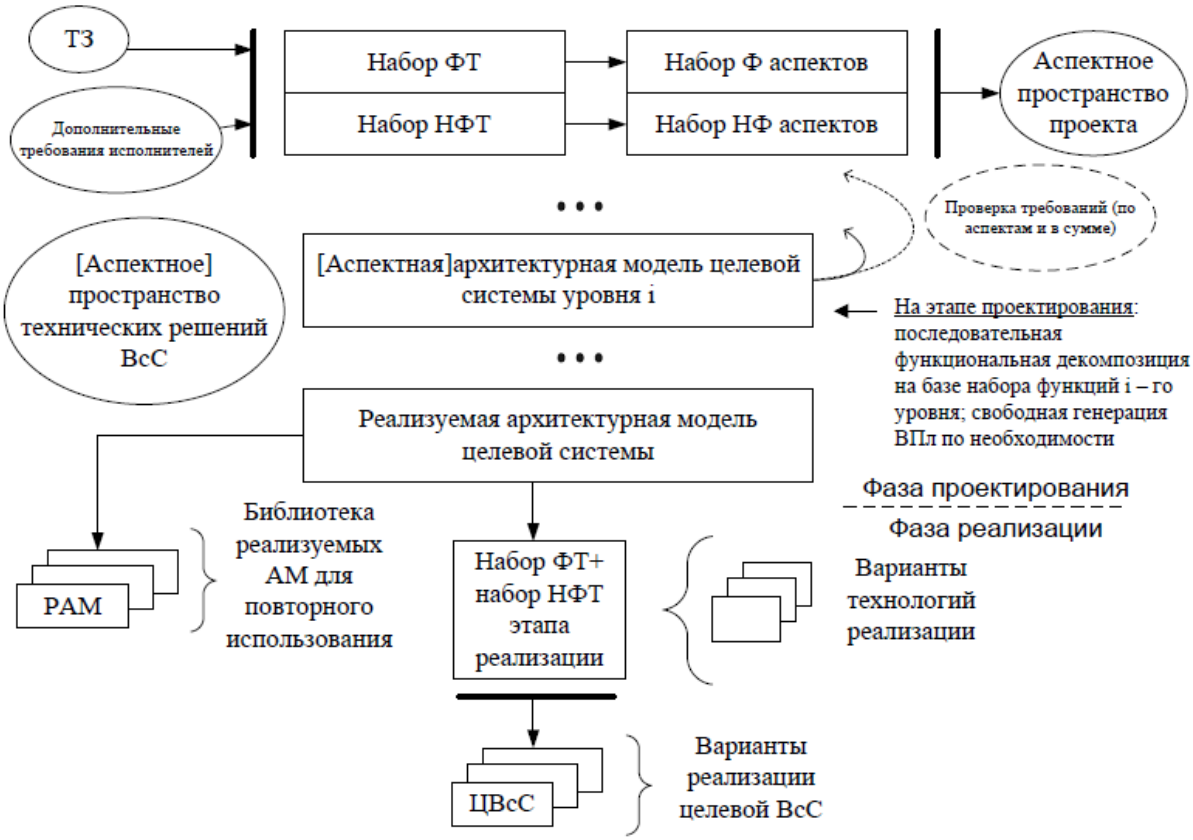


Рис. 39. Шаблоны проектирования ВсС на базе аспектной модели и композиции ВПл [61].

Как уже было сказано выше, помимо создания более качественного, оптимизированного под задачу продукта, проектирование «сверху вниз» позволяет снизить возникающие в процессе проектирования риски, в первую очередь интеграционные. Иначе говоря, уменьшается вероятность того, что на конечном этапе проектирования обнаруживается невозможность или сложность взаимодействия некоторых составляющих компонентов системы, а также вероятность появления критичных ошибок и «узких мест» (пример – недостаточная производительность канала передачи данных).

Следует признать, что в подавляющем большинстве коллективов проектировщиков ВсС сегодня недостаточно высоко оценивается роль и

трудоемкость этапов высокоуровневого проектирования, отсутствует адекватный технический язык для общения на этом уровне. Разработчики оперируют лишь конкретными реализациями вычислительных механизмов (то есть «ассемблерными кодами»), в которых трудно или невозможно проследить концептуальные моменты и решения. Мери Шоу в статье «Мы можем обучать информатике лучше» пишет: «Давайте организуем наши курсы вокруг идей, а не вокруг артефактов. Это сделает наши цели более ясными как для студентов, так и для преподавателей. Машиностроительные институты не преподают проектирование бойлера, они преподают термодинамику. В то же время как минимум два из основных курсов по информатике «Создание компиляторов» и «Операционные системы» являются артефактами динозаврами программирования» [32].

3.1.1 Архитектурные абстракции

Использование абстракций является одним из основных методов работы со сложными, комплексными объектами. Это обусловлено особенностью нашего мышления — ограничением на количество объектов/понятий/сущностей, с которыми мозг может одновременно работать (по разным оценкам, это число в среднем равно четырём-семи). Поэтому способность быстро вычленять суть, значимую для решения конкретной задачи, из общего массива информации – является одной из важнейших составляющих успешного решения любой проблемы.

Преимущество экспертов во всех областях как раз и заключается в том, что они, основываясь на собственном опыте, могут сразу извлекать выводы о рассматриваемых явлениях и получать из них все необходимые абстрактные представления, минуя долгий процесс последовательного, итерационного размышления и осмысления всего массива информации, а иногда — могут предугадывать долгосрочные последствия.

В нашем случае, взгляд на вычислительную технику и инфокоммуникационные технологии (ВТ&ИКТ) с «абстрактных» высот позволяет [1]:

- определять (ограничивать снизу и сверху) область деятельности – зону ответственности ВТ&ИКТ-специалиста;
- делать «прозрачным» представление того, что называют «технологиями» во всех областях ВТ&ИКТ;
- взвешенно распределять влияние реализации на принцип решения задачи в ВТ&ИКТ – областях;

- позволять аккумулировать и свободно (осознанно и без «шор») развивать концептуальные технические решения, менять/расширять уровень технологий повторного использования;
- выявлять (проявлять) «болевые точки» в организации и проектировании ВТ&ИКТ – систем, следовательно, обеспечивать грамотную постановку задач.

Также, в контексте проектирования ВсС и СнК множество базовых архитектурных абстракций можно свести в следующие четыре группы:

1) базовые элементы ВС (вычислительный механизм, вычислительная платформа, архитектурный агрегат);
2) абстракции представления ВС на системном уровне (архитектура, архитектурная платформа, архитектурная модель, аспект);
3) абстракции процесса проектирования (проектирование ВсС, инфраструктура проекта, проектное пространство, аспектное пространство);
4) понятия для анализа и оценки существующих архитектурных решений (вычислительный процесс, виртуальная машина, модель вычислений).

Часть из перечисленных абстракций имеют ярко выраженную «вычислительную» специфику, их будем относить к категории вычислительных. Следующие абстракции будем относить к категории невычислительных: архитектурный агрегат, аспект, проектирование ВсС, инфраструктура проекта, проектное пространство, аспектное пространство.

Важным является понимание и принятие тезиса о возможности представления вычислительного процесса в терминах вычислительных абстракций, не привязанных непосредственно к конкретным реализациям. Будем считать, что такие виртуальные вычислительные процессы должны строиться на основе вычислительных механизмов различного назначения. Следует отметить, что на практике виртуальные вычислительные процессы активно реализуются в современных ВС, при этом разработчики такие процессы связывают с понятием виртуальной машины, трактуя ее как программно-реализованную модель реальной вычислительной машины. Данный технический прием является удобной архитектурной абстракцией, направленной на структуризацию вычислительного процесса и, тем самым, на локальное понижение степени его сложности.

Существует также известный тезис о "семантическом разрыве" между элементами (операциями, структурами данных) представления вычислительных процессов в языках программирования высокого уровня и возможностями физических (аппаратных, микропрограммных) средств реализации на уровне микроопераций, микрокоманд и команд с соответствующими форматами данных, уровнем и степенью параллелизма, ограничениями памяти и т.д. Развитие вычислительной техники в области методов и технологий проектирования идет по пути развития вычислительных парадигм (таких, как модели вычислений, парадигмы программирования), предлагая качественно новые решения. Это движение стремительно увеличивает семантический разрыв с уровнем физической реализации.

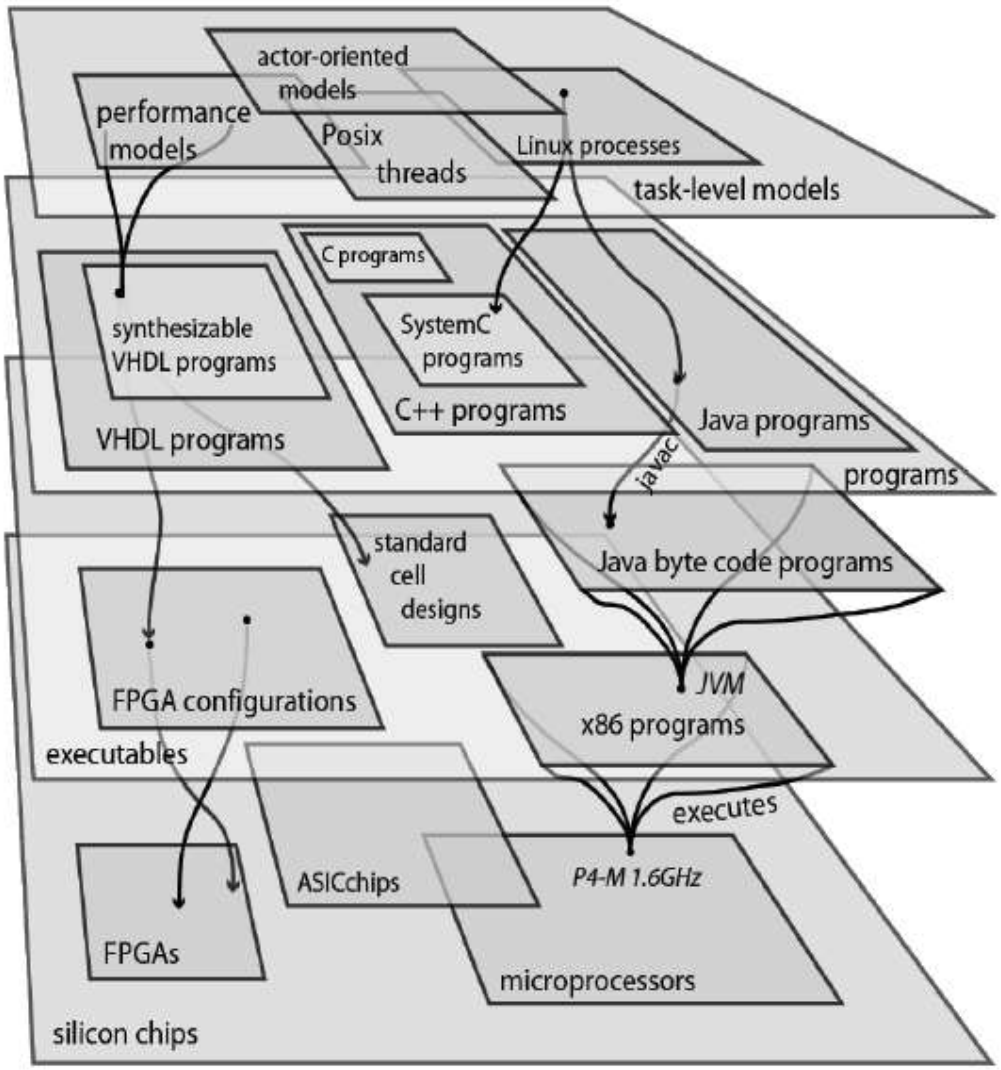


Рис. 40. Многоуровневая организация BcC [34].

Рассматривая архитектуру BcC через призму целевой функциональности, можно говорить о различных уровнях детализации

представления, о различной направленности представления (для категорий специалистов), о различной степени оптимальности реализации в соотношении с технологической сложностью. Более высокоуровневым, доступным для восприятия (прозрачным) и простым технологически представляется способ подачи архитектуры ВсС как простой иерархии виртуальных машин. При этом считается, что *виртуальная машина* – вычислитель (computer), полученный в результате виртуализации базовой вычислительной платформы. Такое представление с последующей прямой реализацией удобно и понятно в контексте ряда параллельно работающих команд исполнителей и при условии действия проектной модели "неограниченных вычислительных ресурсов". Но требование минимизации ресурсов вступает в противоречие с подобной моделью, заставляя сокращать число уровней иерархии и переходить к так называемым "плоским" моделям реализации. Хотя, некоторые виртуальные машины используются повсеместно (компиляция из С в ассемблер для микропроцессоров, описание аппаратуры на уровне RTL).

Иерархическое представление ВсС в терминах виртуальных машин является очень важным и мощным инструментом проектирования. Важнейшее свойство такого представления состоит в возможности достигать сокращения трудоемкости проектирования и повышения степени повторного использования при условии выполнения других ресурсных ограничений проекта. Кроме того, такое представление полезно при обучении и формировании специалистов в области ВсС, так как оно позволяет демонстрировать воспринимаемый человеком образ системы, заставляет оперировать в явном виде различными вычислительными моделями, согласовывать их друг с другом.

Переход от высокоуровневого абстрактного представления вычислительного процесса к уровню физической реализации так называемого "не вычислительного базиса" (границей можно считать вентильный или транзисторный уровень) в силу сложности задачи требует большого числа промежуточных уровней и представлений. Эти уровни необходимы, прежде всего, разработчику для борьбы со сложностью задачи. На практике чаще всего выделение и реализация таких уровней в вычислительной системе выполняется различными коллективами разработчиков и на разных этапах проектирования. Причем в большинстве проектов значительная (основная) часть работы присутствует в повторно используемых компонентах, таких как процессоры, операционные системы, различные API, коммуникационные протоколы, трансляторы и т.д.

Любая задача в рамках ВсС может быть подготовлена к исполнению (решению) с использованием той или иной степени вложенности уровней вычислительной иерархии. Это в первую очередь определяется

размерностью задачи, квалификацией, личными способностями и пристрастиями разработчиков, допустимыми сроками проектирования. В дальнейшем такое решение может быть реализовано с различной степенью оптимизации, от использования "как есть", до глобально оптимизированного "плоского" одноуровневого представления нижнего уровня вычислительной иерархии. Естественно, ресурсоёмкость целевого варианта системы в зависимости от числа оставшихся в результате оптимизации уровней будет меняться в широких пределах.

Само по себе проектное пространство ВсС представляется посредством совокупности базовых абстракций процесса проектирования. Сразу оговоримся, что проектное пространство ВсС не является чисто вычислительной абстракцией.

Прежде всего, в пространство входят объекты или элементы, участвующие в процессе проектирования системы, которые позволяют описать, структурировать, зафиксировать функциональность ВсС. Примерами таких абстракций выступают понятия вычислительной архитектуры, вычислителя, интерфейса, платформы, процесса, вычислительного механизма, виртуальной машины, программируемого интерпретатора, модели вычислений и другие.

Как было отмечено выше, необходима система вычислительных абстракций, позволяющих разработчику оперировать в рамках целевого и проектного пространств при создании ВсС. Традиционно перечисленные выше элементы рассматриваются в качестве составляющих целевой системы. Частью проектного пространства, которую в этом случае целесообразно представлять в подобных понятиях, будет выступать внутренняя организация инструментария. Однако полезно и рассмотрение целевой системы с позиций организации вычислительного процесса с его off- и online фазами. В этом случае принадлежность и область действия вычислительных абстракций расширяется, захватывая оба пространства проектирования. Примером такого перехода, в значительной мере привычного и понятного большинству разработчиков ВсС, является трактовка термина "вычислительная платформа" как совокупности целевых аппаратно-программных средств (например, аппаратура и ОС ПК) и транслятора ЯВУ.

Существующие сегодня вычислительные абстракции в терминологическом и смысловом плане нечетко определены, часто имеют несколько различных трактовок, иногда пересекаются по области действия или противоречат друг другу. Кроме того, существует потребность в расширении области действия, унификации трактовки существующих понятий и введении ряда новых абстракций при проектировании ВсС.

Переход в область абстрактного проектирования и расширение этой области при создании ВсС возможны только при серьезном изменении способа представления проектируемой системы и создании новых проектных методов и средств. Важнейшим тезисом, который обеспечивает такой переход, является принципиальная свобода разработчика в выборе реализации построенной архитектуры ВсС (и ее частей). При этом элементная база (в традиционном понимании) должна выступать не более чем частным ограничением, даже в том случае, если она фиксируется на уровне технического задания.

На свободу проектировщика от ограничений элементной базы в широком смысле, включающей в себя абстрактные механизмы, на этапе архитектурного проектирования в значительной мере влияет вопрос о проведении границы в проекте между фазами проектирования и реализации. Тезис такой свободы при определении архитектуры и, возможно, других значимых проектных решений, требует выбора глубины архитектурной проработки.

3.1.2 Модели вычислений

Ключевой задачей в процессе проектирования является обоснованный выбор той или иной модели целевой системы [33]. Рассмотрение системы с точки зрения той или иной модели автоматически привнесет в систему свойства, присущие выбранной модели. При рассмотрении моделей ВсС, обычно говорят о так называемых “моделях вычислений” (МоС). Некоторые направления исследований в области технологий проектирования встраиваемых систем обращают особое внимание на использование в проектировании этого понятия. При представлении вычислительной системы как иерархии виртуальных машин каждому уровню такой модели соответствует своя МоС.

МоС можно интуитивно представить, как набор правил, необходимых для построения вычислительного процесса системы. Это парадигма, описывающая протекание вычислительного процесса, способы обмена данными, взаимодействия между отдельными функциональными элементами. Кроме того, МоС предлагает терминологию и примитивы, в базисе которых требуется выражать и описывать целевую систему. МоС описывает природу потоков данных, элементов синхронизации, роль времени в процессе выполнения системой целевой функции. Различные МоС по-разному описывают одни и те же процессы, протекающие в целевой системе. Для больших и сложных систем совершенно нормальное положение дел, когда различные части системы представляются различными МоС.

Примерами МоС являются: модель с дискретными событиями, сеть обработки потоков данных, взаимодействующие конечные автоматы,

синхронная модель вычислений, объектно-событийная модель вычислений и денотативно-объектная модель вычислений.

Ограниченное количество распространенных и полезных моделей вычислений и хорошая изученность их свойств позволяют видеть в моделях вычислений компонент повторного использования. Модель вычислений может служить «точкой пересечения» понимания работы системы разными специалистами. Использование терминологии, предоставляемой моделью вычислений, также помогает коллективной разработке.

Проектирование ВсС с использованием понятия моделей вычислений представляется как проектирование вычислительного процесса. Использование разных моделей вычислений позволяет математически точно описать определенные аспекты встроенной вычислительной системы.

Важной и неотъемлемой частью модели вычислений является язык МоС. Как и любой другой язык, язык МоС определяется алфавитом, синтаксисом и семантикой. Алфавит представляет собой множество допустимых символов языка, которые могут быть скомбинированы различными способами. Правила комбинирования и допустимые комбинации определяются синтаксисом языка. Смысл и интерпретация тех или иных допустимых комбинаций символов алфавита определяется семантикой языка. С точки зрения языка его МоС представляет собой поведение некоторой абстрактной вычислительной машины (или просто абстрактного вычислителя) в рамках семантики языка. С этой точки зрения можно вести разговор о МоС традиционных языков программирования, таких как C, C++, Java, Pascal и т.д. В этом случае, оценивая применимость того или иного языка, а на самом деле МоС, на которой каждый конкретный язык базируется, для решения той или иной задачи, стоящей перед разработчиком, можно обсуждать и оценивать характеристики системы, которые навязывает МоС.

Если рассмотреть целевую систему на некотором уровне абстракции как набор взаимодействующих изолированных блоков (вычислительных компонентов системы), то МоС системы на данном уровне абстракции, используя выразительные средства языка, описывает следующие аспекты системы:

1) Поведение вычислительных компонентов;
2) Взаимодействие вычислительных компонентов;
3) Способы передачи данных и синхронизацию вычислений;
4) Способы декомпозиции и агрегации вычислительных компонентов.

МоС должна содержать характеристики системы, важные на данном уровне абстракции. Элементы модели (примитивы, языковые средства, требования и др.), а, следовательно, и вся модель в целом, не должны быть слишком абстрактными или слишком конкретными. Т.е. МоС должна быть в состоянии описать целевую систему. Описание в данном контексте понимается как определенная степень спецификации системы, возможно не до конца формальной. Это требование к МоС можно сформулировать как адекватность описания целевой системы.

Кроме собственно описания целевой системы на заданном уровне абстракции, МоС должна обеспечивать разработчику средства работы с этим описанием. Разработчик должен иметь возможность доказывать истинность или ложность определенных утверждений, относительно целевой системы, проверять соответствие определенным требованиям и ограничениям, накладываемым на целевую систему. Инструменты, предоставляемые моделью, должны позволять проводить оценку тех или иных характеристик целевой системы, проводить оптимизацию по выбранным параметрам. Все эти действия разработчика можно назвать моделированием целевой системы в терминах выбранной МоС.

3.1.3 Аспектное представление проекта

Комплексный характер проектов ВсС в сочетании с ростом их сложности требует создания методов и технологий проектирования, которые позволят эффективно учитывать, анализировать, синтезировать, отслеживать качество всех признанных существенными сторон организации ВсС и существующей вокруг нее инфраструктуры на протяжении всего жизненного цикла, особенно на этапах создания и модификации. Выделение таких относительно самостоятельных сторон является процессом нетривиальным. Мы будем называть такие локализованные стороны проекта или целевой системы аспектами.

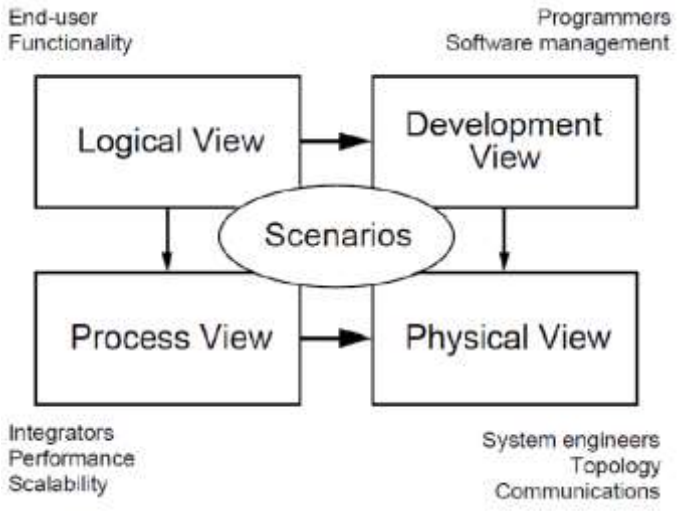


Рис. 41. Аспектное представление проекта [34].

Другими словами, аспект – это некоторая частная проблема проектирования в рамках задачи создания ВсС. Подчеркнем еще раз, что аспекты существуют не в рамках какого-либо этапа или шага развития проекта или целевой системы, а на протяжении всего процесса проектирования или всего жизненного цикла системы («вес» аспекта в проекте меняется во времени и может вырождаться до нуля). Множество, включающее все аспекты проектирования, будем называть аспектным пространством процесса проектирования ВсС. Множество, непосредственно принадлежащее проектируемой целевой системе, будем называть аспектным пространством целевой системы.

Такие стороны, или аспекты, могут обладать различной степенью пересечения. Кроме того, полезным является рассмотрение проектного пространства целевой системы, как части пространства всего проекта в целом, что повышает эффективность проектирования ВсС. В этом случае в поле зрения разработчиков попадают дополнительные аспекты. Вообще, проектируемое изделие обязательно следует рассматривать с различных точек зрения, и в первую очередь – с точки зрения конечного пользователя. Также, очевидным является стремление выделять ортогональные аспекты, что позволяет выполнять их условно независимую и параллельную разработку в рамках проекта. Список аспектов в проекте ВсС всегда конечный, но их общий перечень является открытым.

Типовыми и наиболее важными аспектами процесса проектирования ВсС можно считать:

- структурно-функциональный,
- конструктивно-технологический,
- энергетический,
- инструментальный,
- повторного использования,
- организационно-экономический,
- документный,
- надежностный,
- точностной.

Проработка аспекта в рамках проекта должна идти последовательно на всех стадиях и выражается в его специфицировании, проектировании, верификации, реализации и т.д. Другими словами, работа в рамках аспекта представляет собой мини-проект, который направлен на реализацию одного из свойств создаваемой системы. Причем это свойство может как непосредственно обеспечивать требуемую функциональность целевой

системы, так и быть направленным на достижение иных целей, например, на поддержание заданного процента повторного использования объектов некоторой категории в проекте.

Важнейшим фактором успеха проектирования ВсС является учет требований и ограничений, которые накладываются в явном и неявном виде через ТЗ, а также вносятся самим коллективом проектировщиков, отражая его возможности, предпочтения, вторичные задачи и т.д. Традиционное на сегодня выделение только "чисто вычислительных" сторон (аспектов) ведения проекта в распространенных методиках и инструментальных средствах проектирования ВсС существенно ограничивает возможности совершенствования всех основных показателей качества проектов.

Распространенное мнение о том, что внутренние факторы проектного коллектива не должны оказывать влияние на результаты разработки, если они не отражены явным образом в ТЗ, не является конструктивным, так как на практике такие факторы всегда работают. Примером такого фактора может служить вектор коллектива в области освоения определенных семейств аппаратных или программных продуктов (микроконтроллеров, языковых средств и т.д.), что будет проявляться в создании целевых систем. Аспектная модель проектной инфраструктуры с вычислительными и невычислительными элементами позволяет легализовать многие важные процессы, сделать проект прозрачным и управляемым.

3.2 Платформно-ориентированное проектирование

Платформно-ориентированное проектирование (platform-based design, PBD) – интенсивное повторное использование для ВсС, когда значительные части проекта основываются на предварительно разработанных и верифицированных компонентах, которое в жёстких рыночных условиях позволяет значительно повысить конкурентоспособность разработки. PBD – «ориентированный на интеграцию подход к проектированию сложных систем, при котором придаётся особое значение систематическому повторному использованию, и который основывается на заданных инструментальных платформах и виртуальных компонентах совместимых аппаратных средств и ПО с целью снижения рисков разработки, стоимости и времени выхода на рынок» [35]. Принципы PBD состоят из старта на самом высоком уровне абстракции, когда скрыты ненужные подробности реализации, суммирования важных параметров реализации в абстрактной модели, ограничения исследования пространства проектирования до набора доступных компонентов, и выполнения проектирования как последовательности детализирующих начальную абстракцию шагов, которые идут от начальной спецификации к конечной реализации, используя платформы на различных уровнях абстракции [4].

В концепции платформно-ориентированного системного проектирования основным понятием является платформа – некоторый компонент разрабатываемых систем, пригодный для повторного использования. Это могут быть платы или их части, драйверы, интерфейсы, операционные системы, API и т.п.). Аппаратная и программная платформы объединяются в так называемую системную платформу. Функциональность – то, что требуется от разрабатываемого изделия – рассматривается отдельно от структурного представления системы. Процесс проектирования представляет собой поиск отображения для заданной функциональности на какой-то из существующих компонентов (может сравниваться несколько платформ и выбираться оптимальная).

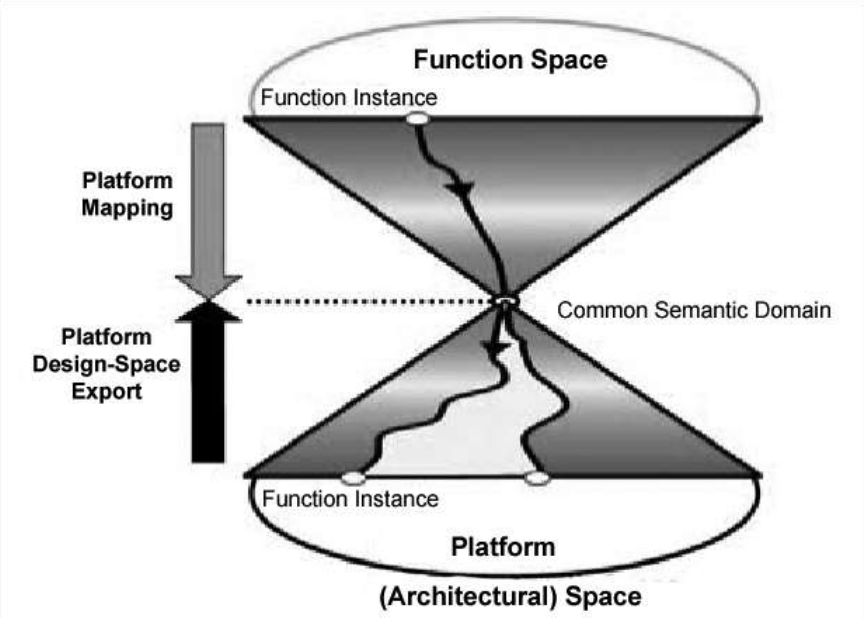


Рис. 42. Платформно-ориентированное проектирование [4].

Аппаратная платформа – это множество архитектур вычислительных машин, позволяющее решать поставленную задачу. При проектировании ограничения архитектуры обычно определяются в терминах производительности и размеров. Обычно в аппаратной платформе больше возможностей, чем требуется от проектируемой системы. Нет смысла использовать аппаратуру, пригодную для решения единственной задачи.

Программная платформа – абстрактный уровень взаимодействия программы с аппаратурой. Основная идея – это разработка платформно-независимого API. Т.е. между программой и аппаратурой вставляется программный слой, унифицирующий работу программы с некоторым набором аппаратуры. К программной платформе относят:

- операционную систему (обычно ОСРВ), распределяющую аппаратный ресурс;
- драйверы устройств, обеспечивающие подсистему ввода/вывода;
- сетевую подсистему, обеспечивающую взаимодействие компонентов вычислительной системы.

Системная платформа, как уже говорилось, объединяет аппаратную и программную платформы. В начале проектирования обе платформы являются абстракциями. Причем, чем выше эта абстракция, тем лучше – больше свободы в выборе решений по конкретизации системы. Нежелательно преждевременное разделение функций между аппаратурой и ПО. Возможно и желательно рассмотрение нескольких платформ, пригодных для реализации одной и той же функциональности с целью выбора оптимальной. В таком случае количество платформ для рассмотрения и глубина их анализа определяются выделенными на проектирование ресурсами. Проектируемая система с добавлением ограничений (быстродействие, габариты, надежность, потребление энергии, доступное API, наиболее подходящая ОСРВ и т.п.) уточняется (актуализируется) и вариантов ее реализации остается все меньше и меньше. В итоге получается единственное решение: однозначный выбор аппаратуры и определенная модель ПО.

Для обеспечения возможности повторного использования платформы должны быть хорошо верифицированы и документированы. В идеале, желательны стандарты, не определяющие их внутреннее устройство, но фиксирующие функциональность и интерфейсы для интеграции (шины, API и т.д.). Повсеместное использование таких стандартов может значительно видоизменить всю индустрию электроники, позволив быструю и эффективную интеграцию существующих сторонних разработок. В качестве примера подобной стандартизации можно привести сферу персональных компьютеров, где стандартизованы внутренние соединения, устройства ввода вывода (мыши, клавиатуры), архитектура системы команд. Здесь, под абстракциями «процессор», «память», «клавиатура», «дисплей» могут скрываться произвольно исполненные устройства, реализующие соответствующую функциональность с определёнными параметрами качества и имеющие определённую стоимость. В общем случае сборка ПК представляет собой компоновку готовых изделий без необходимости вникать в их внутреннее устройство. Применение подобного подхода в области проектирования электроники должно принести пользу как изготовителям платформ, так и тем, кто их использует.

Примером такой платформы из области ВсС является платформа DaVinci производства компании Texas Instruments, представляющая собой линейку высокопроизводительных мультимедийных процессоров, программное обеспечение в виде готовых библиотек, позволяющих использовать их возможности, документацию и развитую сеть для поддержки быстрого и эффективного проектирования на основе данных процессоров.

3.3 Модельно-ориентированное проектирование

В настоящее время во многих областях явно прослеживается тенденция к более абстрактному и декларативному проектированию через автоматическое или автоматизированное управление последовательно детализируемыми моделями. Данный процесс не новый, в прошлом подобное уже наблюдалось и в программировании, и в разработке аппаратуры, где произошёл переход от языка ассемблера и проектирования на уровне логических ячеек к высокоуровневым языкам – C, VHDL, Bluespec. Одной из основных характеристик тенденции является отделение доменной модели (модели предметной области) и архитектурных моделей платформы, что приводит к более платформно-независимому проектированию и повышению степени повторного использования решений. Кроме того, отделение доменной модели системы от реализации позволяет узкоспециализированным инженерам решать проблемы, связанные с их предметными областями, абстрагируясь от реализации вычислений.

Под модельно-ориентированным проектированием (model-based design, MBD) обычно понимается парадигма, в соответствии с которой проектирование ПО ВС (и всей системы вообще) фокусируется на высокоуровневых исполняемых моделях проектируемой системы [36]. Наиболее популярным и широко распространённым примером реализации MBD является среда проектирования MATLAB/Simulink, которая повсеместно используется для проектирования систем автоматического управления (САУ), систем цифровой обработки сигналов (ЦОС, DSP), механических систем и т.д. Учитывая одновременное моделирование физических непрерывных процессов и дискретных вычислений, MATLAB/Simulink позволяет вполне реализовать киберфизический подход.

Кроме того, в области создания ПО известно и широко используется управляемое моделями проектирование (model-driven development, model-driven engineering, model-driven architecture). В основе этого подхода лежит генерация программного кода из моделей (абстрактных описаний ПО), которые являются основными артефактами разработки. Фактически, это соответствует визуализации структуры программы. В основном данный метод используется для коммерческого прикладного ПО, но известны

подходы к адаптации этого метода для создания ПО микроконтроллерных встраиваемых систем и конфигураций для ПЛИС.

Понятия MBD и MDD во многом пересекаются и фактически отражают один и тот же подход к проектированию, на основе синтеза реализаций из абстрактных моделей. При таком подходе каждый последовательный шаг детализации моделей приближает разрабатываемый проект к реализации на заданной платформе, благодаря чему разработчик может концентрироваться больше на крупномасштабных аспектах системы, а не на деталях её конкретного исполнения. При более абстрактном видении проблемы имеется больше свободы для выбора более подходящего решения, и больше возможностей конечных реализаций, в противовес проектированию для одной и только одной платформы.

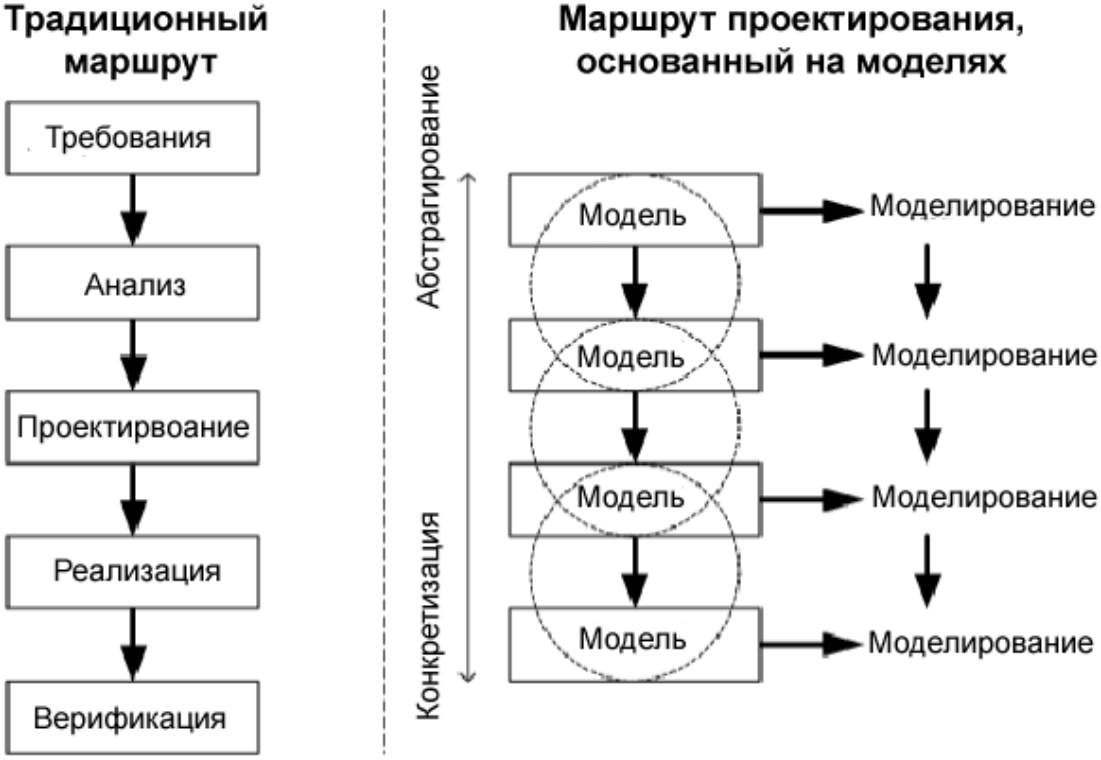


Рис. 43. Традиционный и основанный на моделях маршруты проектирования [35].

Традиционные методы проектирования (аппаратуры, ПО, систем в целом) представляют собой стадии разработки требований, анализа, проектирования, реализации и верификации. Проектирование же на основе моделей вращается вокруг моделей и трансформаций между этими моделями. В процессе проектирования работа производится над некоторым набором моделей в каждый момент времени, что предоставляет обратную связь за счёт верификации. Верификация осуществляется сравнением поведения высокоуровневых, более абстрактных, и низкоуровневых, детализированных моделей.

Важно понимать, что проектирование и разработка при использовании моделей не метод или процесс в себе, это структурирование вместе набора моделей, каждая из которых имеет чётко определённые цели и дополняет остальные. Таким образом, это скорее описание взаимоотношений между конкретными моделями, чем описании их самих.

Проектирование на основе моделей обычно представляется как ряд моделей, связанных между собой нисходящими трансформациями. Однако, в реальном мире ни такое, строго последовательное проектирование сверху вниз, ни проектирование снизу-вверх не работают, кроме очень специальных случаев. Разработка должна производиться одновременно на разных уровнях абстракций и с учётом многих аспектов в одно и то же время. Критично понимать и категоризировать все эти аспекты системы с самых ранних этапов процесса, с учётом их отношений между собой.

Симуляции должны включать в себя обширные наборы технологий верификации и валидации, их результаты должны сравниваться с результатами симуляций более высокоуровневых моделей, проверяться на соответствие ограничениям. В течение процесса проектирования модели трансформируются автоматически или полуавтоматически в более платформно-ориентированные. Выделяются следующие модели:

1) Не зависящие от организации вычислительного процесса (Computation Independent Model). Это – модели использования системы, модели предметных областей, модели данных.
2) Не зависящие от платформы (Platform Independent Model). Это модели, выполненные в рамках основных моделей вычислений (конечный автомат, сети процессов, и др.).
3) Платформно-зависимые (Platform Specific Model).
4) Реализации спецификаций.

Использование при проектировании высокоуровневых моделей системы позволяет исследовать альтернативы реализации разрабатываемого продукта, упрощает коммуникации с заказчиком и исполнителями, удобнее для документирования. При этом следует помнить о необходимости тщательного тестирования, осознавать, что для проектирования может требоваться несколько моделей и инструментов, и не использовать модели вне их области применения (например, для рекурсивных вычислений).

3.3.1 Пример: Ptolemy

Ptolemy II представляет собой комплекс программ, библиотек Java-классов и механизмов. Комплекс предназначен для построения и исследования моделей различного рода систем, в основном для

встраиваемых систем, а также для академических исследований в области моделирования.

На основе идей и подходов, реализованных в Ptolemy II, активно развиваются средства моделирования как во многих исследовательских HLD САПР, так и в продуктах индустриального назначения [37].

Кратко возможности и особенности системы Ptolemy II представлены ниже.

Ptolemy II создан и постоянно развивается группой исследователей Ptolemy Project из университета Калифорнии в Беркли, возглавляемой профессором Edward A. Lee [38,39]. Комплекс разрабатывается как open-source ресурс и распространяется свободно. Ptolemy II является третьим по счету инструментом, который разрабатывается группой Ptolemy Project, после Gabriel (1986-1991) и Ptolemy Classic (1990-1997). Большое количество идей для Ptolemy II было взято из Ptolemy Classic, который представлял собой средство моделирования и проектирования систем обработки данных и управляющих систем. Ptolemy Classic позволял создавать в графической среде гетерогенные модели систем с использованием разных моделей вычислений, исполнять эти модели на симуляторах различных аппаратных и программных платформ, а также генерировать из моделей код на ассемблере (для DSP), C и VHDL.

Ptolemy II начал разрабатываться в 1996 году и был призван сместить акцент на моделирование с использованием технологий, предоставляемых платформой Java™. В новой среде были введены понятия доменного полиморфизма, «режимных» (modal) моделей, а также типового полиморфизма. Был также разработан механизм разрешения типов данных в модели, множество моделей вычислений, технология распределенного моделирования, а также мощный язык выражений, который может быть использован, в том числе, и в исполняющейся модели.

Ptolemy II обеспечивает моделирование вычислительных систем различной природы. На основе комплекса созданы специализированные подсистемы для моделирования:

- беспроводных сетей и распределенных систем, объединенных радио-, оптическими или акустическими каналами связи (пакет VisualSense);
- гибридных систем, объединяющих в себе дискретную логику управления и модели с непрерывным временем. Пакет HyVisual включает в себя модели вычислений FSM (finite state machines) и CT (continuous-time), библиотеку акторов, поддерживающих эти модели вычислений и средства визуализации моделей.

Каждая подсистема представляет собой подмножество классов, механизмов и акторов, объединенное одной управляющей программой.

Ptolemy II создан для моделирования гетерогенных вычислительных систем, то есть систем, объединяющих компоненты, описываемые в разных моделях вычислений: дискретную и аналоговую технику, сети и телекоммуникационные протоколы, стохастические и криптографические системы, радиокомпоненты, компоненты, реализующие обработку сигналов и изображений и т.д. Для этого комплекс поддерживает моделирование в рамках различных моделей вычислений и объединение отдельных моделей в гетерогенную иерархию с последующим комплексным моделированием. В настоящее время комплекс поддерживает следующие модели вычислений:

- CT (continuous time) – модели с непрерывным временем;
- DE (discrete-event) – модели с дискретными событиями;
- DDE (distributed discrete events) – системы с частично упорядоченным множеством дискретных событий;
- FSM (finite-state machines) – конечные автоматы, в модели отсутствует понятие времени;
- PN (process networks) – сети процессов Кана (Kahn), в модели отсутствует понятие времени;
- SDF (synchronous dataflow) – модели для представления систем обработки сигналов, в моделях отсутствует понятие времени;
- DDF (dynamic dataflow) – расширение SDF, позволяющее компонентам изменять количество потребляемых и генерируемых элементов данных за одну итерацию в процессе исполнения модели;
- HDF (heterogeneous dataflow) – расширение SDF, позволяющее сохранить статическое планирование и другие свойства моделей в ее рамках и избегать ограничений на неизменность количества потребляемых и генерируемых компонентами элементов данных;
- PSDF (parameterized synchronous dataflow) – расширение SDF с параметризируемой дисциплиной планирования вычислений;
- DT (discrete time) – периодически запускаемые статически планируемые процессы с дискретным временем. Аналог SDF, в котором присутствует понятие времени;
- SR (synchronous-reactive) – синхронно-реактивная модель вычислений;

- CI (component interaction) – модель вычислений со стилем взаимодействия компонентов «push/pull»;
- CSP (communicating sequential processes) – взаимодействующие последовательно выполняемые процессы;
- Giotto – поддержка одноименного языка и его семантики. Один из вариантов синхронной модели, в котором компоненты могут активироваться с разной частотой;
- TM (timed multitasking) – вариант взаимодействия задач ОСРВ;
- Wireless – модель вычислений для исследования систем, объединенных беспроводными каналами связи;
- GR – поддержка 3-D графики.

Комплекс Ptolemy II является open-source-проектом, что способствует распространению и широкому использованию подходов и механизмов, реализованных в нем. Комплекс является масштабируемым практически в любом аспекте, включая поддержку новых моделей вычислений. Исходный текст комплекса оформлен аккуратно и в соответствии с принятым в проекте стилем кодирования, что позволяет легко разобраться в исходном коде любого Java-класса. Кроме того, весь исходный код документирован с помощью утилиты Javadoc, что позволяет, избегая тщательного анализа исходных текстов, быстро перемещаться по систематизированной документации в поиске требуемой информации о том или ином классе. Комплекс снабжен тремя томами подробного описания всех возможностей, реализованных механизмов и моделей вычислений, а также техники моделирования и рекомендациями по применению и расширению возможностей комплекса Ptolemy II. Лицензия Ptolemy II предоставляет право свободного применения всего или частей комплекса в любых разработках, включая коммерческие продукты.

Система Ptolemy II включает в себя графическую среду разработки Vergil, позволяющую схематично представлять модели, аннотировать и конфигурировать их, проверять их корректность и запускать на исполнение (Рис. 44).

Vergil включает редактор моделей в акторном представлении (Graph editor), редактор конечных автоматов (Рис. 45), текстовый редактор, редактор иконок акторов, позволяет просматривать HTML-страницы и редактировать текст файлов.

Компонентами моделей Ptolemy II по большей части являются акторы, атомарные функциональные преобразователи сигналов, поведение которых описано на Java. Каждый такой актор представляет собой Java-класс, который может быть в процессе моделирования расширен и дополнен различными механизмами Ptolemy II. Акторы могут быть

составными, то есть являться контейнерами других моделей, обеспечивая, таким образом, их иерархичность. Комплекс включает обширную библиотеку акторов, систематизированную по группам выполняемых ими преобразований (Рис. 46).

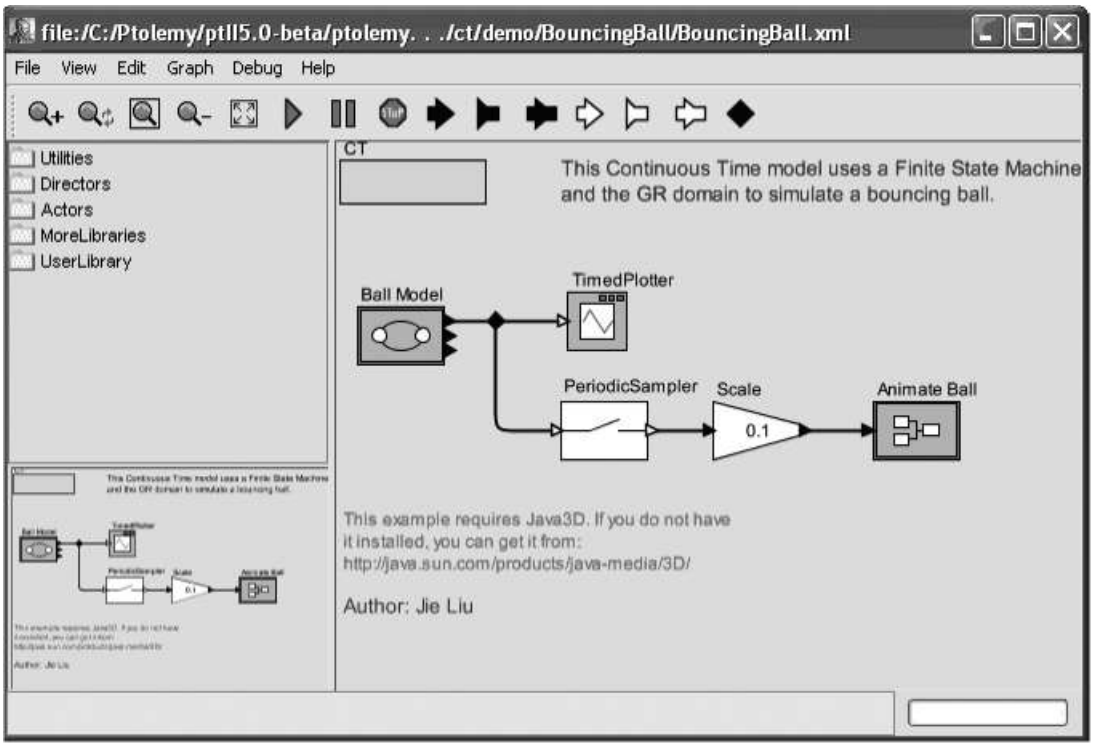


Рис. 44. Редактор моделей Vergil.

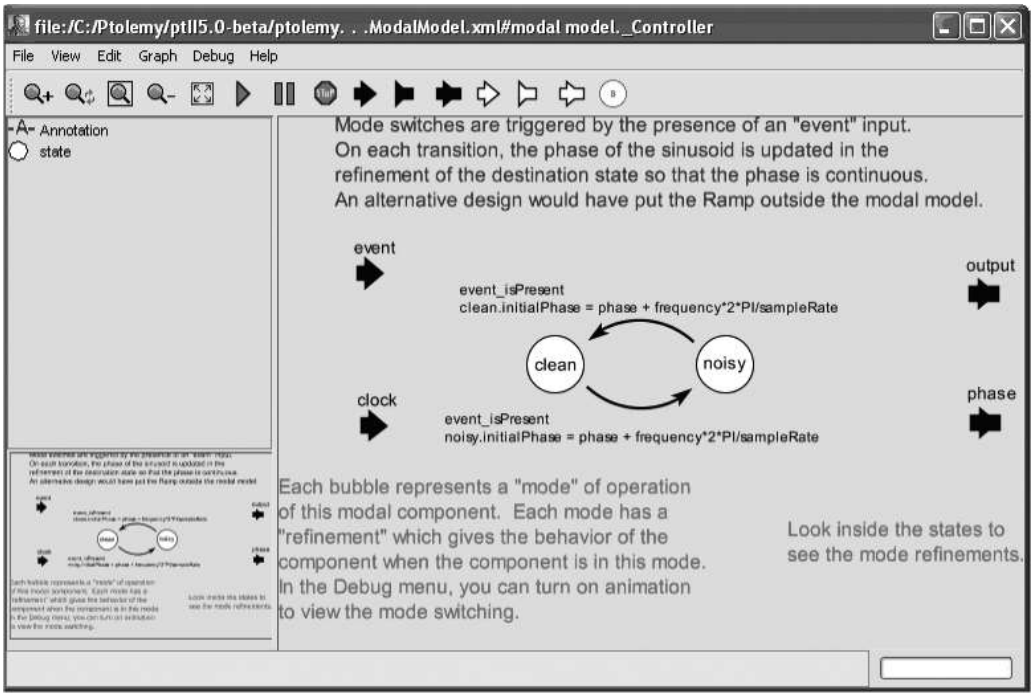


Рис. 45. Редактор конечных автоматов Vergil.

Большинство акторов стандартной библиотеки Ptolemy II обладает полиморфизмом функциональности относительно модели вычислений, в которой они используются. Этот механизм является одним из базовых механизмов Ptolemy II и обеспечивает широкие возможности поведенческого описания в рамках почти любой модели вычислений пользователем в графическом редакторе Vergil, не прибегая к кодированию на Java или существенной модификации стандартных акторов.

Еще одним базовым механизмом является полиморфизм функциональности акторов относительно типа преобразуемых данных. Большинство акторов могут реализовывать свое поведение над сигналами различных типов. Этот механизм в значительной степени опирается на существующую в Ptolemy II систему типов и механизм их разрешения при инициализации модели. Он реализован за счет построения иерархии классов, которые представляют элементы данных различных типов, и реализации множества операций, смысл которых инвариантен к типу данных.

Типы данных, с которыми может работать модель в Ptolemy II, организованы в систему, которая сравнима по сложности с системой типов C++ или Java (Рис. 47). Помимо простых типов данных (int, double, long, unsignedByte, complex, fixedpoint, boolean, string, scalar, matrix и т.д.), существуют составные типы (массивы, матрицы, записи и объекты). Компоненты модели, то есть акторы, обмениваются друг с другом в процессе ее исполнения элементами данных, имеющими тот или иной тип. С помощью встроенного в Ptolemy II языка выражений и механизма разрешения типов данные, имеющие один тип, могут быть явно или неявно преобразованы к другому типу.

Массивы в Ptolemy II определяются как упорядоченные наборы данных одного типа, имеющие одно измерение. Матрицы представляют собой двумерные упорядоченные наборы данных одного типа. Ptolemy II поддерживает матрицы, состоящие из элементов следующих типов: boolean, complex, double, fixedpoint, int и long. Другие типы элементов в матрицах не разрешены. Для матриц предусмотрено множество операций векторно-матричной алгебры и функций, практически все из которых присутствуют в MATLAB (MathWorks Ltd.). Записи в Ptolemy II состоят из именованных полей, которые могут иметь разные типы. Доступ к отдельным полям осуществляется с помощью операции ‘.’ (как в C++ или Java). Интересно отметить, что для данных, имеющих тип «запись», предусмотрены бинарные операции пересечения и объединения по именам полей.

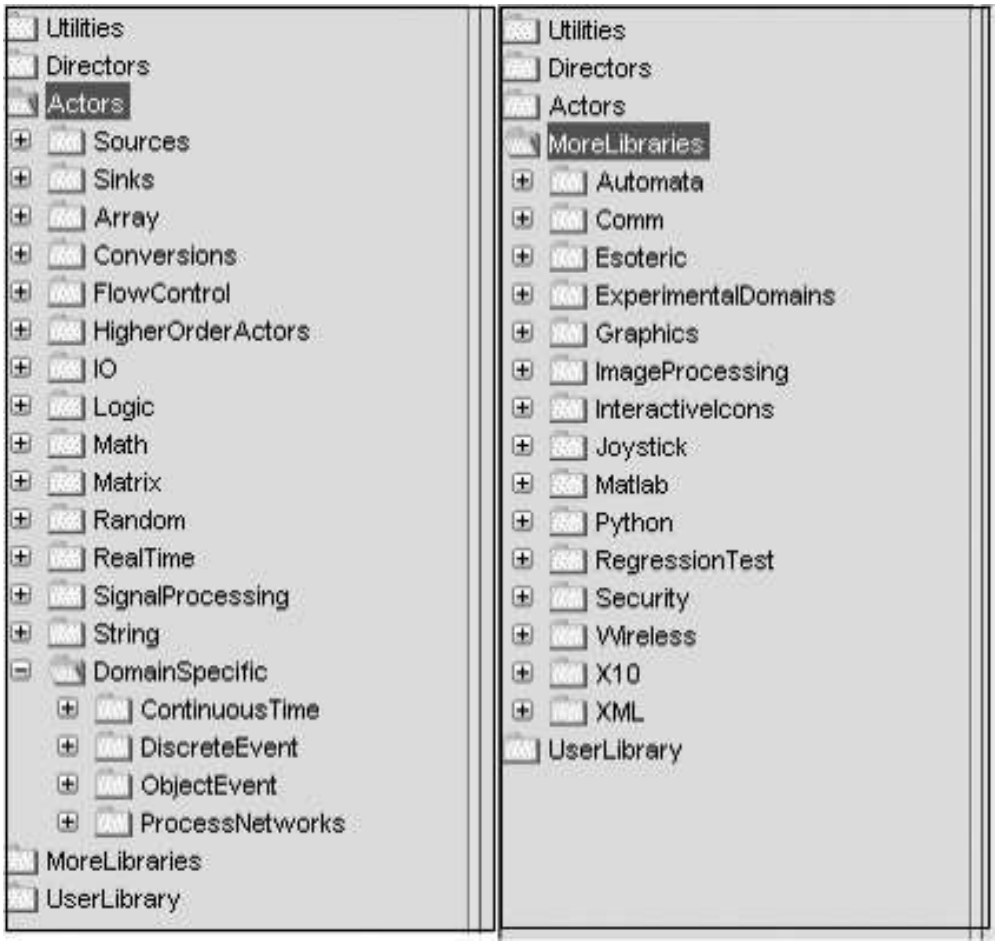


Рис. 46. Библиотека акторов Ptolemy II.

Механизм разрешения типов Ptolemy II позволяет автоматически проверять корректность преобразований данных в модели в целом, не указывая вручную конкретные типы или ограничения на порты отдельных акторов. Разрешение типов происходит в процессе инициализации модели и устанавливает для каждого порта актора конкретный тип данных, которые он воспринимает, исходя из информации о зависимостях между типами данных портов, декларируемой акторами. Ограничения, накладываемые на порт, могут быть менее строгими, чем указание его конкретного типа, благодаря возможности преобразования данных одних типов к другим (Рис. 47).

Например, ограничение на порт может заключаться в том, что тип данных, которые он воспринимает, может быть больше или равен конкретному типу. Под «больше» понимается то, что данный тип соединен нисходящим путем в графе на рис. 1.6. В случае несоответствия ограничениям, то есть невозможности разрешения типов, инициализация модели завершается с ошибкой. Ошибка может быть устранена вручную.

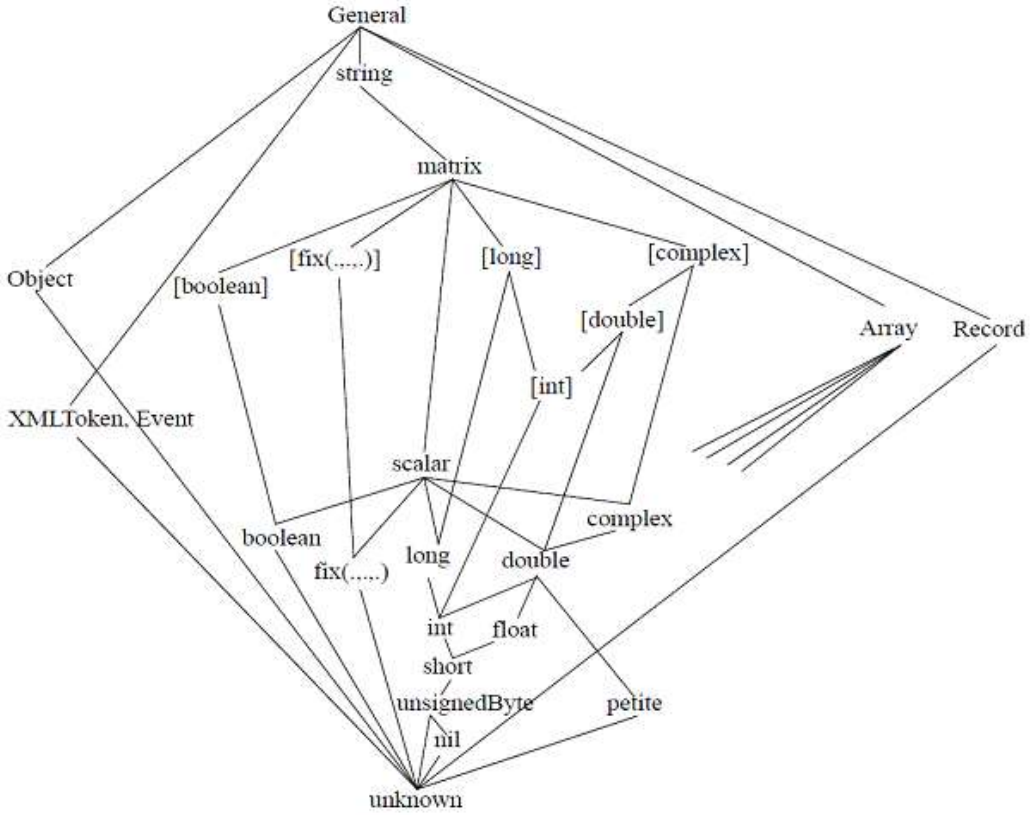


Рис. 47. Граф типов данных Ptolemy II. Типы данных, которые находятся ниже на рисунке, могут быть преобразованы к связанным с ними линиями типам, которые находятся выше них

В Ptolemy II разработан довольно мощный язык выражений, позволяющий не только выполнять математические операции над данными, получая значение результата, но и создавать переменные различных типов, в том числе и произвольных составных, выполнять преобразования типов, выполнять сложные встроенные процедуры обработки данных и создавать собственные, производить ввод/вывод и управлять вычислительным процессом. В языке выражений предусмотрен набор стандартных именованных констант, которые могут быть использованы в выражениях. Это числовые константы: PI, pi, E, e, true, false, i, j, NaN, Infinity, PositiveInfinity, NegativeInfinity, MaxUnsignedByte, MinUnsignedByte, MaxInt, MinInt, MaxLong, MinLong, MaxDouble, MinDouble; и строковые константы PTII, HOME и CWD. Эти константы имеют заранее определенное значение. Константы с другими значениями могут быть определены с помощью литералов, задающих как значение, так и (либо явно, либо неявно) тип константы. Например, 2.0 даст константу типа double, 2i (или 2j) даст константу, имеющую тип complex. Константы составных типов также задаются с помощью литералов:

- массивы с использованием фигурных скобок с элементами, разделяемыми запятыми: {2, 3, 4, 5};

- матрицы задаются с помощью квадратных скобок и разделения столбцов запятыми, а строк знаками ‘;’: [1,2,3; 4,5,6] – матрица из двух строк и трех столбцов. В языке выражений также поддержан способ задания матриц в стиле MATLAB ([p:q:r]);
- записи задаются с помощью фигурных скобок, в которых перечислены имена полей и их значения: {first_name = ”Edward”, data = {0, 1, 2}, associated_record = {a = 1, b = 2 + PI * j}}.

В языке выражений предусмотрено задание переменных и функций с произвольными именами. Операции, предусмотренные в языке выражений, включают как алгебраические (над числами, строками, массивами, записями и матрицами), так и логические: сравнения, побитовые (&,|,~,#,>>,<<), преобразования типов и условного вычисления (‘ ? : ’). Также в языке предусмотрены комментарии в стиле C++ (/* */). Достаточно объемная библиотека стандартных функций, включающая также и статические методы стандартных классов Java (java.lang.Math, java.lang.Double и т.п.), позволяет реализовать достаточно сложные вычисления.

Выражения могут быть использованы в редакторе выражений Vergil при задании значений параметров в отдельных акторах, а также в качестве элементов данных (Рис. 48).

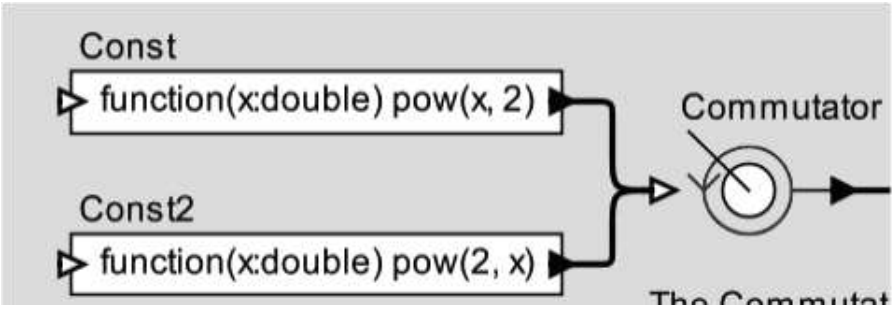


Рис. 48. Использование языка выражений в акторах.

В Ptolemy II существует группа акторов, называемая «компонентами высшего порядка» (Higher-Order Components, HOC). Эти акторы выполняют действия над структурой и функциональной организацией модели в процессе ее исполнения. Примеры акторов этого типа:

- ModelReference – актор, представляющий собой ссылку на модель, заданную с помощью URL. Активация этого актора представляет собой команду на исполнение модели, на которую он ссылается;
- MultiInstanceComposite – составной актор, создающий заданное количество копий самого себя в процессе инициализации модели;

- ModalModel – актор, реализующий модель, которая может работать в нескольких «режимах». Каждый такой «режим» представляет собой произвольную модель с теми же портами, что у ModalModel, а переход между ними осуществляется с помощью конечного автомата. Актор может быть использован для моделирования гибридных систем (дискретная логика в сочетании с непрерывной динамикой).

В Ptolemy II реализован механизм объявления и наследования классов, реализующий механизмы из объектно-ориентированного подхода для составных акторов (Рис. 49). Составной актор может реализовываться в графическом редакторе в рамках какой-либо модели вычислений, а затем может быть преобразован в описание «класса». Экземпляры этого «класса» затем могут быть использованы в модели как обычные составные акторы. Любые изменения в описании класса ведут к изменениям в его экземплярах. Этот подход удобен, когда в модели верхнего уровня иерархии используется несколько однотипных компонентов, преобразующих различные потоки данных.

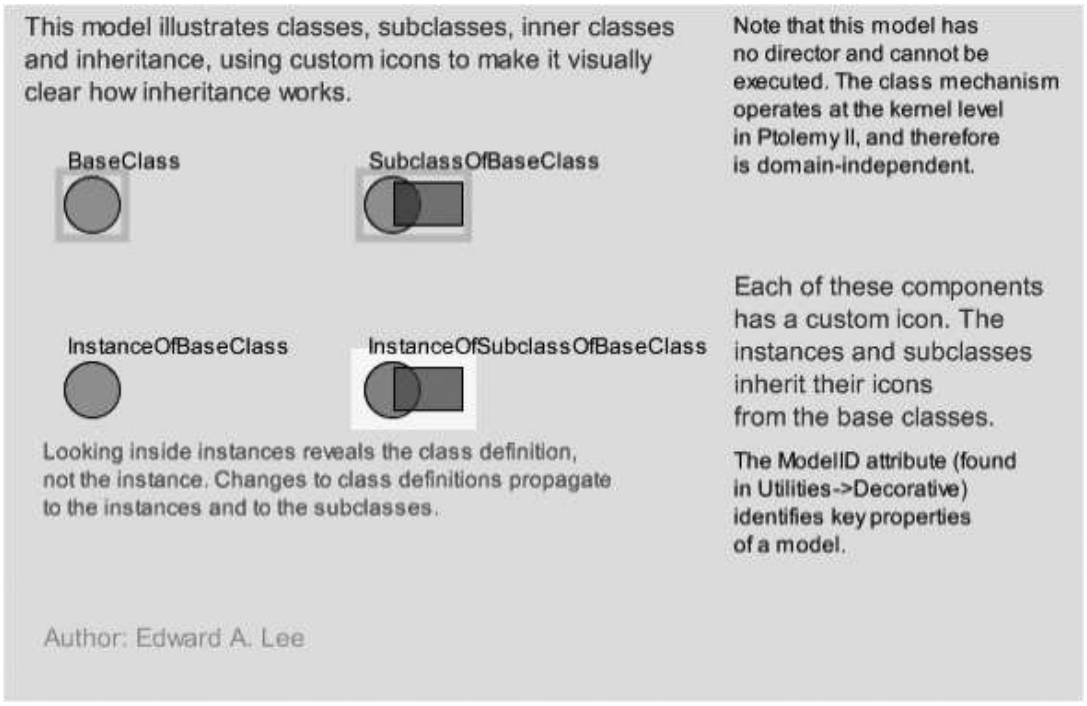


Рис. 49. "Классы" составных акторов.

Кроме того, механизм позволяет строить иерархию «классов», «наследуя» от базовых «классов» все элементы модели нижнего уровня, добавляя в них новые компоненты и «перегружая» часть наследуемых.

Реализация Ptolemy II на базе Java™ позволяет пользоваться всеми возможностями, которые предоставляет эта платформа. В частности, стандартная библиотека Ptolemy II включает акторы, позволяющие:

- пользовательский ввод и интерактивное управление моделированием и структурой модели в процессе ее исполнения;
- ввод, вывод и обработку сигналов и изображений, включая преобразование видеосигналов в реальном масштабе времени;
- файловый ввод-вывод и доступ к устройствам инструментального ПК;
- доступ к ресурсам сети Internet и распределенное моделирование;
- интеграцию с другими инструментами и языками.
- Модели в Ptolemy II представляют собой Java-приложения, выполнением которых управляет один из специально созданных классов. В зависимости от используемых моделей вычислений, приложения могут быть как однопоточными, так и многопоточными (например, при использовании модели вычислений PN). Такой подход позволяет гибко использовать как сами модели, так и предоставляемые ими результаты. Например, можно получить готовую отлаженную модель, включающую средства отображения результатов моделирования, и оформить ее в виде самостоятельной программы (отдельного исполняемого модуля). Или можно создать Java-апплет и опубликовать в Internet, обеспечив возможность удаленной загрузки готовой модели, ввода исходных данных и получения результатов моделирования на любом ПК, в котором реализована поддержка Java™ (Рис. 50).

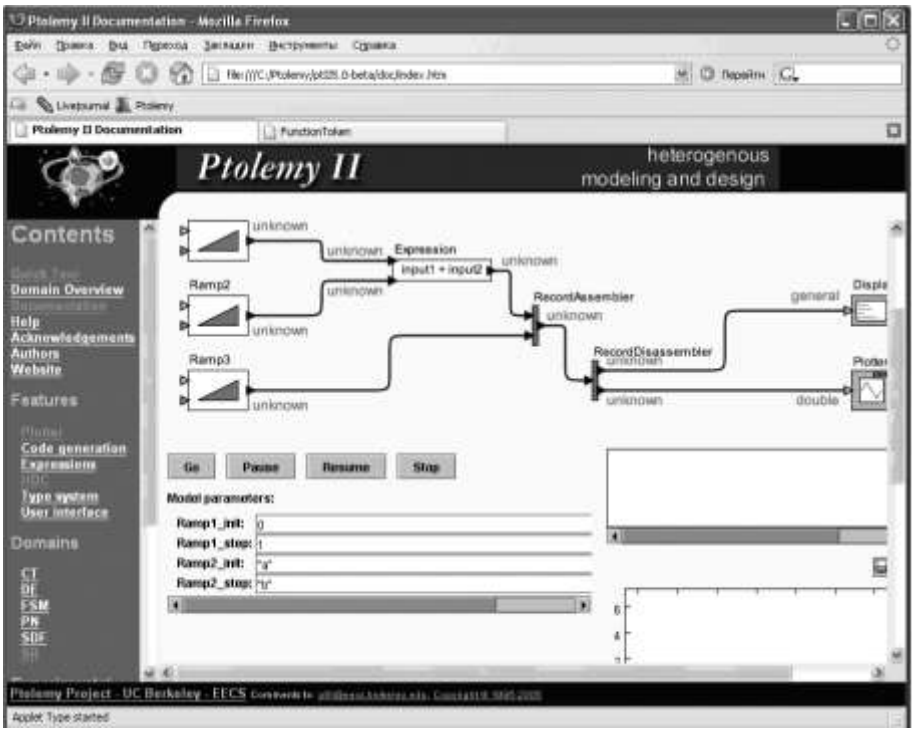


Рис. 50. Оформление модели Ptolemy II в виде апплета.

Наконец, можно исключить весь графический ввод-вывод модели, существенно снизив вычислительную нагрузку на операционную систему, и запустить модель в виде консольной программы Java, обеспечив вывод результатов моделирования в файл. Этот подход удобен, если объем вычислений и время моделирования существенны, в отличие от анимации и диалога с пользователем.

ЛИТЕРАТУРА

1. Платунов А.Е., Постников Н.П. Высокоуровневое проектирование встраиваемых систем. Санкт-Петербург: СПбГУ ИТМО, 2011. P. 125.
2. Платунов А.Е. Встраиваемые системы управления // Control Eng. Россия. 2013. Vol. 1. P. 9.
3. Lee E.A., Seshia S.A. Introduction to Embedded Systems - A Cyber-Physical System Approach. UC Berkeley, 2011. 491 p.
4. Sangiovanni-Vincentelli A. Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design // Proc. IEEE. IEEE, 2007. Vol. 95, № 3. P. 467–506.
5. Teich J. Hardware/Software Codesign: The Past, the Present, and Predicting the Future // Proc. IEEE. 2012. Vol. 100, № Special Centennial Issue. P. 1411–1430.
6. Lee E.A. Cyber Physical Systems: Design Challenges // 2008 11th IEEE Int. Symp. Object Component-Oriented Real-Time Distrib. Comput. IEEE, 2008. P. 363–369.
7. Flynn M.J., Luk W. Computer System Design: System-on-Chip // Comput. Syst. Des. Syst. 2011.
8. Smith G. DAC Panel Presentation. Anaheim: 40th. Design Automation Conference (DAC), 2003.
9. Rutgers J.H. Programming models for many-core architectures: a co-design approach. 2014.
10. Platunov A., Kluchev A., Penskoi A. HLD Methodology: The Role of Architectural Abstractions in Embedded Systems Design // 14th GeoConference Informatics, Geoinformatics Remote Sens. Albena, BULGARIA, 2014. P. 209–218.
11. Platunov A., Kluchev A., Penskoi A. Expanding Design Space for Complex Embedded Systems with HLD-methodology // 2014 6th Int. Congr. Ultra Mod. Telecommun. Control Syst. Work. - Telecommun. 2015. P. 157–164.
12. Kiczales G., Hilsdale E. Aspect-oriented programming // ACM SIGSOFT Softw. Eng. Notes. ACM, 2001. Vol. 26, № 5. P. 313.
13. Clements P.C. A survey of architecture description languages // Proc. 8th Int. Work. Softw. Specif. Des. 1996. P. 16–25.
14. Dijkstra E.E.W. EWD 447: On the role of scientific thought // Sel. Writings Comput. A Pers. Perspect. Springer-Verlag, 1982. P. 60–66.
15. Boehm B., Valerdi R., Honour E. The ROI of systems engineering: Some quantitative results for software-intensive systems // Syst. Eng. 2008. Vol. 11, № 3. P. 221–234.
16. Tassey, Gregory. The Economic Impacts of Inadequate Infrastructure for Software Testing. 2002.
17. Broman D. et al. Viewpoints, formalisms, languages, and tools for cyber-physical systems // Proc. 6th Int. Work. Multi-Paradigm Model. - MPM

’12. 2012. Vol. 0931843. P. 49–54.
18. Schaumont P.R. A Practical Introduction to Hardware/Software Codesign // Media. 2010. 403 p.
19. Vajda A. Programming Many-Core Chips // Springer Sci. 2011. 9-43 p.
20. Николаев А. Поддержка многоядерных процессоров во встраиваемых системах // Открытые системы. 2006. Vol. 7.
21. Santambrogio M.D., Sciuto D. Reconfigurable Computing and Hardware/Software Codesign // EURASIP J. Embed. Syst. 2007.
22. Luppold A. et al. A new concept for system-level design of runtime reconfigurable real-time systems // ACM SIGBED Rev. - Spec. Issue 5th Work. Adapt. Reconfigurable Embed. Syst. 2013. Vol. 10, № 4. P. 57–60.
23. Vahid F., Stitt G., Lysecky R. Warp processing: Dynamic translation of binaries to FPGA circuits // Computer (Long. Beach. Calif). 2008. Vol. 41, № 7. P. 40–46.
24. Atak O. Design of application specific instruction set processors for the FFT and FHT alogorithms. Bilkent University, 2006.
25. Önder S. Processor Description Languages // Process. Descr. Lang. 2008. 247-273 p.
26. Gorjiara B., Reshadi M., Gajski D. Designing a custom architecture for DCT using NISC technology // Asia South Pacific Conf. Des. Autom. 2006. 2006. Vol. 1. P. 1–2.
27. Gorjiara B. et al. Generic netlist representation for system and PE level design exploration // Proc. 4th Int. Conf. Hardware/Software Codesign Syst. Synth. (CODES+ISSS ’06). 2006. P. 282–287.
28. Reshadi M., Gajski D. A cycle-accurate compilation algorithm for custom pipelined datapaths // Proc. 3rd IEEE/ACM/IFIP Int. Conf. Hardware/software codesign Syst. Synth. - CODES+ISSS ’05. 2005. P. 21.
29. Gorjiara B., Gajski D. A novel profile-driven technique for simultaneous power and code-size optimization of microcoded IPs. Lake Tahoe, CA: Computer Design, 2007. ICCD 2007. 25th International Conference on, 2007. P. 609–614.
30. Gajski D.D., Kuhn R.H. Guest Editors’ Introduction: New VLSI Tools // Computer (Long. Beach. Calif). 1983. Vol. 16, № 12. P. 11–14.
31. Jantsch A., Kumar S., Hemani A. A Metamodel for Studying Concepts in Electronic System Design // IEEE Des. Test. 2000. Vol. 17, № 3. P. 78–85.
32. Shaw M. We Can Teach Software Better // Comput. Res. New. Carnegie Mellon University, 1992. Vol. 4. P. 2–4, 12.
33. Платунов А.Е. Теоретические и методологические основы высокоуровневого проектирования встраиваемых вычислительных систем. Санкт-Петербург: Университет ИТМО, 2010.
34. Kruchten P. Architectural Blueprints - The 4+1 View Model of Software Architecture // IEEE Softw. 1995. Vol. 12, № 6. P. 42–50.

35. Bailey B., Martin G., Piziali A. ESL Design and Verification // ESL Des. Verif. 2007. 113-138 p.
36. Poole J.D. Model-Driven Architecture : Vision , Standards And Emerging Technologies // Work. Metamodeling Adapt. Object Model. ECOOP01. 2001. № April. P. 1–15.
37. Densmore D., Passerone R., Sangiovanni-Vincentelli A. A platform-based taxonomy for ESL design // IEEE Des. Test Comput. 2006. Vol. 23, № 5. P. 359–373.
38. Lee E.A., Neuendorffer S., Wirthlin M.J. Actor-Oriented Design of Embedded Hardware and Software Systems // J. Circuits, Syst. Comput. 2003. Vol. 12. P. 231–260.
39. Bhattacharyya S. et al. Vol. 1 - Ptolemy 0.7 User's Manual. California: College of engineering department of electrical engineering and computer sciences Berkeley, 1997. 532 p.

УНИВЕРСИТЕТ ИТМО

Миссия университета – генерация передовых знаний, внедрение инновационных разработок и подготовка элитных кадров, способных действовать в условиях быстро меняющегося мира и обеспечивать опережающее развитие науки, технологий и других областей для содействия решению актуальных задач.

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра ВТ Университета ИТМО создана в 1937 году и является одной из старейших и авторитетнейших научно-педагогических школ России.

Традиционно основной упор в подготовке специалистов на кафедре делается на фундаментальную базовую подготовку в рамках общепрофессиональных и специальных дисциплин, охватывающих наиболее важные разделы вычислительной техники.

Кафедра является одной из крупнейших в университете. Учебными курсами и научно-исследовательскими работами руководят 8 профессоров и 16 доцентов. На кафедре обучаются более 500 студентов и 30 аспирантов.

Кафедра имеет собственные компьютерные классы и специализированные исследовательские лаборатории, оснащенные современной вычислительной и оргтехникой, уникальным инструментальным и технологическим оборудованием, измерительными приборами и программным обеспечением.

В 2007-2008 гг. коллективом кафедры была успешно реализована инновационная образовательная программа СПбНИУ ИТМО по научно-образовательному направлению «Встроенные вычислительные системы».

Начиная с 2009 года кафедра вычислительной техники является активным участником реализации программы развития национального исследовательского университета НИУ ИТМО, вошла в состав крупнейшего в НИУ ИТМО научно-исследовательского центра «Интеллектуальные системы управления и обработки информации».

Начиная с 2013 года кафедра ВТ принимает активное участие в реализации мероприятий программы повышения международной конкурентоспособности Университета ИТМО «5 в 100».

Быковский Сергей Вячеславович
Горбачев Ярослав Георгиевич
Ключев Аркадий Олегович
Пенской Александр Владимирович
Платунов Алексей Евгеньевич

Сопряжённое проектирование встраиваемых систем (Hardware/Software Co-Design)

Часть 1

Учебное пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел

Университета ИТМО

197101, Санкт-Петербург, Кронверкский пр., 49