Оглавление

Введение

Начиная с момента изобретения первой электронно-вычислительной машины и вплоть до наших дней идет постоянная миниатюризация средств вычислительной техники, вызванная в свою очередь прогрессом в области производства интегральных схем. Как результат, мы можем делать все более сложные и сложные системы не получая при этом увеличения их размеров или стоимости. Для разработчика, использующего стандартные методы проектирования на уровне RTL-передач, реализация подобной системы будет настоящей пыткой. И стоит отметить, что подобный разрыв между способностями разработчика и сложностью систем в будущем будет только увеличиваться.

Чтобы разработчик мог адекватно работать в условиях все увеличивающейся сложности систем, необходимо переходить на все более высокие уровни абстракции при проектировании вычислительных систем.

Создаваемые сегодня вычислительные системы концентрируют в рамках одной интегральной схемы множество относительно сложных и самодостаточных подсистем. Подобные системы принято называть системами на кристалле. В строгом смысле, данное понятие не является термином и означает общую тенденцию к интеграции на системном уровне. Как отмечается в работе [1], сложность создания подобных систем очень высока и при использовании стандартных подходов к проектированию (например, RTL-уровень), в условиях прогресса технологий производства, время на их создание будет увеличиваться логарифмически.

1991	1994	1006	1998	2000	2002	2004	2006	2008	2010
0.7	0.5	0.35	0.25	0.18	0.13	90	65	45	32
1k	5k	15k	30k	45k	80k	150k	300k	600k	1.2M
#Gates / Die (50mm2) conservative numbers									
50K	250K	750k	1.5M	2.2M	4M	7.5M	15M	30M	60M
#Gates per Designer per year									
4k	6k	9k	40k	56k	91k	125k	200k	200k	200k
Men / Years per 50 mm2 Die									
~10	~40	~80	~40	~40	~43	~60	~75	~150	~300

→ It is urgent to win some productivity

Рис 1. Тенденция к увеличению времени разработки вследствие технологического прогресса [1]

Для повышения продуктивности разработчиков, в промышленности и научной сфере ведутся работы по созданию новых технологий, объединяемых в рамках системного уровня проектирования электроники (ESLD, Electronics System Level Design). Сюда относятся такие концепции как совместное проектирование (HW/SW co-design) и высокоуровневый синтез (High-Level Synthesis, HLS).

Высокоуровневый синтез — это концепция, в рамках которой предполагается создание средств перевода некоторых алгоритмических описаний в их аппаратную реализацию [2]. В принципе, не существует единого языка, предписываемого к использованию в HLS, для описания алгоритмической составляющей [1]. На взгляд автора, такие языки должны учитывать свойства решаемой задачи или класса задач и предоставлять достаточный уровень абстракции. В большинстве своём, в качестве такого языка в промышленных средствах высокоуровневого синтеза выступают языки Си и Си++. Что касается языков представления спецификации аппаратной реализации, то в качестве таковых выступают стандартные для индустрии языки — Verilog и VHDL.

Цели и задачи дипломного проекта

В рамках данной работы перед автором стояла следующая цель: создание системы автоматической генерации HDL кода.

В рамках проекта стоит задача преобразования некоторого высокоуровневого описания вычислительной системы в ее синтезируемое описание, которое по сути является уже готовым устройством – достаточно провести процесс синтеза и прошить получившуюся прошивку в ПЛИС. В соответствии с утвержденной структурой проекта, данная задача должна решаться системой автоматической генерации HDL кода, которая и рассматривается в данной работе. В ходе работы над исследовательским проектом было принято решение ввести данную систему в проект как дополнительный уровень абстракции, состоящий из специализированного языка описания аппаратуры и транслятора. Причины, побудившие автора построить свою систему так, а не иначе, будут рассмотрены далее. Пока же предлагаю рассмотреть цель проекта и задачи, поставленные при разработке системы автоматической генерации HDL-кода:

Цель проекта:

- создание системы автоматической генерации HDL-кода;

Задачи проекта:

- анализ сферы задач, решаемых проектируемой системой, составление проектных требований к ней;
- выбор HDL, пригодного для составления синтезируемого описания аппаратуры в рамках проекта “Arrow Compution Group”;
- выбор синтезируемого подмножества, необходимого для успешного решения задачи генерации описания аппаратуры;

- анализ синтаксических конструкций выбранного HDL и разработка синтаксиса специализированного языка описания аппаратуры, определение семантики специализированного языка описания аппаратуры;
- определение этапов трансляции системы автоматической генерации HDL-кода;

Система генерации кода на языке описания аппаратуры

Общая концепция

Проект, развиваемый академической группой “Arrow Computing Group”, можно отнести к классу систем высокоуровневого синтеза. Основной особенностью является нацеленность на ограниченный класс систем, предназначенных для критических применений (работа в режиме жёсткого реального времени, малое время отклика). Соответственно, в качестве базового языка описания алгоритмической компоненты выбран язык, обеспечивающий возможности формальных оценок времени исполнения. Данный язык также является оригинальной разработкой в рамках проекта и обладает рядом положительных особенностей, как то, возможность выражения естественного параллелизма задачи, функциональная чистота, широкие возможности формального анализа, вследствие наличия системы типов данных и др [3].

В дальнейшем, созданное описание транслируется в аппаратную реализацию в виде непрограммируемого процессора [3]. Трансляция осуществляется посредством средств инструментальной цепочки [4].

Имеет смысл коротко рассмотреть инструментальный аспект (см рис. 2) разрабатываемой технологии, и определить зоны ответственности автора.

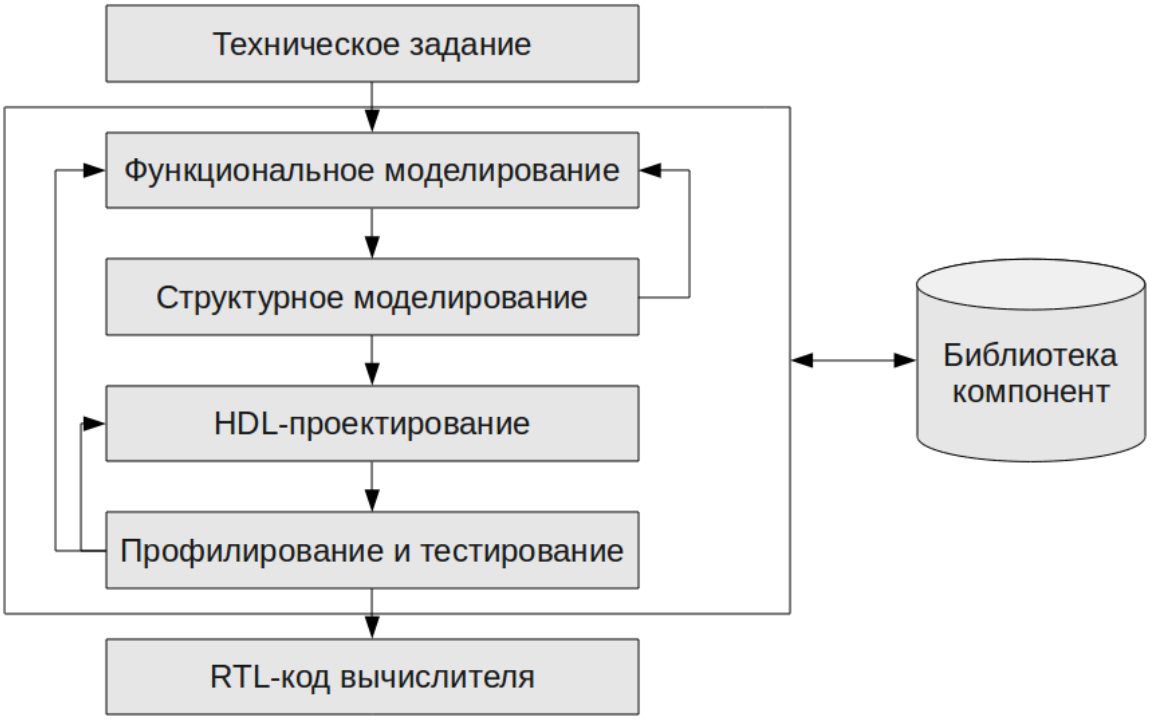

Рис 2. Технология получения RTL-описания специализированного вычислителя

Инструментальный аспект описан на уровне концепции — т.е. не является описанием реализации инструментальной цепочки. Реализованные в рамках данной работы элементы инструментальной цепочки описываются в [4] и созданы с запасом на использование в разрабатываемом проектном процессе.

Задачей инструментального ПО является генерация на основе технического задания HDL-описания вычислителя. Требования, заложенные в ТЗ, могут быть разделены на функциональные и нефункциональные. Под функциональными требованиями подразумевается то, как система должна работать при тех или иных вариантах ее использования (use cases). Под нефункциональными требованиями имеются ввиду такие параметры как время отклика, габариты, энергозатраты, стоимость.

В целях систематизации инструментальные средства можно разделить на 4 области (рис. 3):

- домен функционального моделирования;
- домен HDL-описания стрелки;
- домен синтеза и оптимизации расписания;
- домен генерации HDL-описания системы.

Область ответственности автора лежит в домене генерации HDL-описания системы, а именно части подсистемы синтеза. К *домену генерации HDL-описания системы* относятся средства синтеза HDL-описания системы на основе расписания, HDL-кода примитивных стрелок и информации об описании инфраструктуры стрелочного вычислителя. Под описанием инфраструктуры стрелочного вычислителя понимается описание взаимосвязей между аппаратными блоками данного вычислителя.

Здесь же находятся средства верификации HDL-описания вычислителя, инструментарий, автоматизирующий выбор аппаратной платформы, инструментарий верификации вычислителя в контексте выбранной аппаратной платформы (проверка на соответствие функциональным и нефункциональным требованиям к системе).

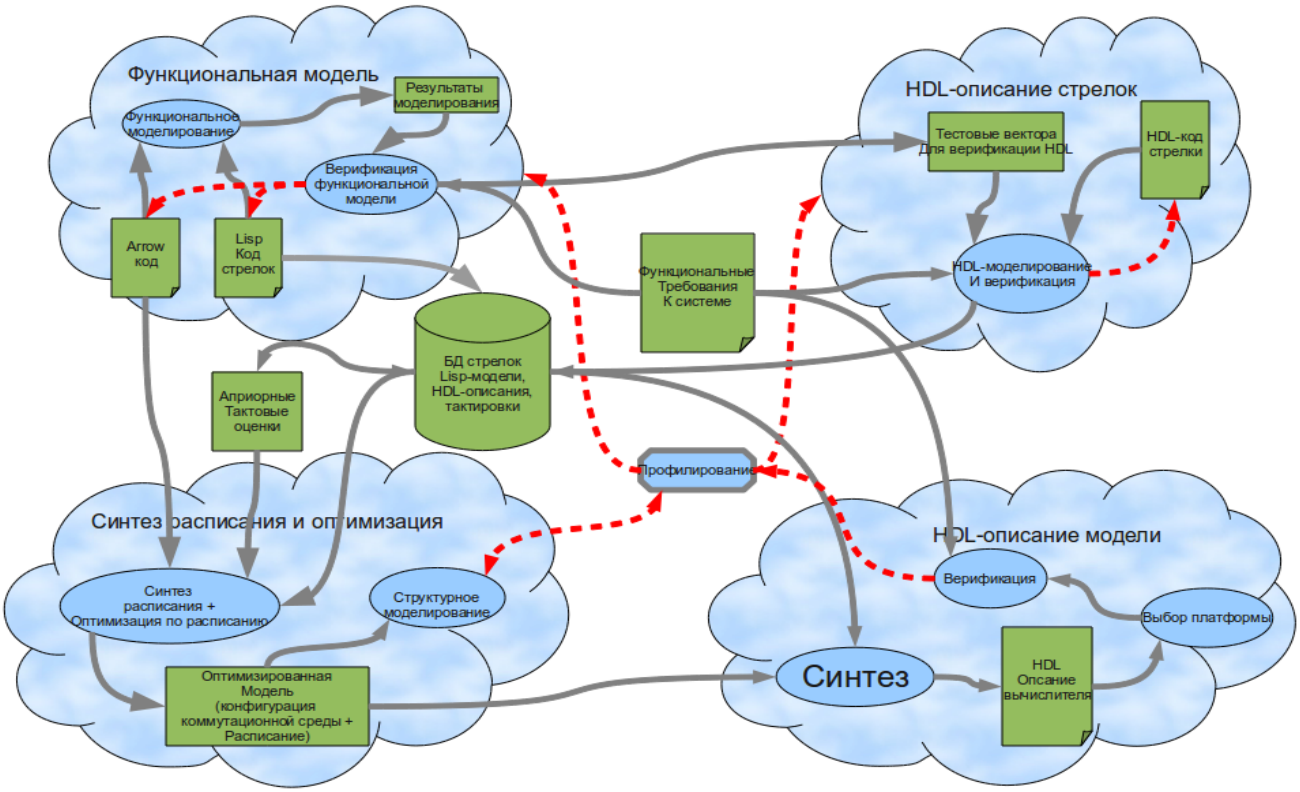

Рис 3. Инструментальный аспект технологии [4]

Цели проекта

Основной целью проекта является создание технологии для разработки специализированных вычислителей жёсткого реального времени на языке высокого уровня. Разрабатываемая технология, в том числе, нацелена на решение таких основополагающих проблем в области проектирования, как:

- высокие проектные риски;
- высокая стоимость разработки;
- низкие показатели переиспользования;

Попытка решения данных проблем состоит в предложенном проектном процессе, в котором присутствуют все основные этапы проектирования, учитывается аспект

реального времени и предлагаются варианты организации библиотеки компонент для повторного использования наработок[4].

Проблематика

Прежде чем рассматривать систему автоматической генерации HDL кода, стоит рассмотреть весь проект “Arrow Compution Group” как единое целое. Это позволит получить представление о некоторой предметной области, в рамках которой создавалась вышеупомянутая система.

Проект “Arrow Compution Group” был задуман сравнительно недавно. Но несмотря на свой малый возраст он уже дал некоторый полезный, с академической точки зрения, результат в виде двух магистерских и одной работы по специализированной модели вычислений. Проект позиционируется как средство для создания специализированных вычислителей жесткого реального времени.

В проекте, данные вычислители создаются из описания их поведения в рамках оригинального вычислительного процесса, названного стрелочной моделью вычислений (СМВ). К сожалению, рассмотрение данной модели вычислений выходит за рамки данной дипломной работы – интересующиеся могут обратиться к работе магистра кафедры Вычислительной техники Александра Пенского, посвященной как раз стрелочной модели вычислений (приведена в приложении 2).

Помимо описания поведения специализированного вычислителя, в его синтезе участвует набор стандартных программных блоков, поставляемых вместе со средой разработки. Кроме того, пользователю дана возможность описывать свои блоки, специализированные для решения каких-нибудь особых задач, что позволяет производить тонкую настройку синтезируемого целевого вычислителя под свои нужды.

После прохождения цепочки синтеза разработчик получает описание взаимосвязей между стандартными (и самостоятельно написанными, если они использовались) блоками, названное “описанием инфраструктуры стрелочного вычислителя”

в терминах проекта. Стоит отметить, что в стандартные блоки входят те блоки, что были в стандартной поставке инструментальной цепочки. Любые, написанные самостоятельно и помещенные в библиотеку компонентов блоки, уже являются специализированными.

Полученное описание инфраструктуры стрелочного вычислителя не пригодно для последующего синтеза аппаратуры, хотя бы потому что отсутствуют средства для синтеза. Конечно, можно создать свой особый синтезатор, но в результате общего проектного совещания подобная ветка развития проекта была признана тупиковой из-за ее сложности, сопоставимой по сложности с самим проектом. Было принято решение пойти другим путем и транслировать описание инфраструктуры стрелочного вычислителя в один из синтезируемых HDL. Так и появилась идея создать для проекта систему автоматической генерации HDL кода.

Пути решения проблемы

В самом общем приближении, данная система должна иметь возможность принять на входе описание инфраструктуры стрелочного вычислителя и выдать разработчику на выходе описание проектируемого специализированного вычислителя на каком-либо HDL. При этом, для семантически верной генерации HDL-кода используются еще и библиотеки описаний программных блоков на выбранном HDL.

При решении задачи трансляции описания инфраструктуры стрелочного вычислителя, которое было выполнено на S-выражениях[1], в описание проектируемой аппаратуры, выполненное на HDL, возникла вполне ожидаемая проблема – в свете того, что эти два описания не совпадают не только по синтаксису, но еще и по семантике – трансляция из одного описания в другое выглядит, мягко говоря, нетривиальной задачей.

[1]Один из способов записи символических выражений в доступной для понимания человеком форме. Представляет собой набор слов заключенных в скобочки, при этом часто первое слово имеет какое-то особое значение (имя функции, макроса). S-выражения применяются в Лиспе.

Можно не выполнять трансляцию напрямую, а применить платформу, использование которой облегчит задачу трансляции из описания инфраструктуры стрелочного вычислителя в описание аппаратуры на HDL. Стоит отметить, что данной платформой является разрабатываемый в рамках дипломной работы специализированный HDL.

Если как следует рассмотреть оба предложенных варианта решения проблемы упрощения генерации HDL-кода, то можно заметить следующее. Если принять время разработки транслятора для преобразования из изменчивого описания инфраструктуры стрелочного вычислителя непосредственно в HDL-код за N и допустить, что время разработки транслятора, способного выполнить преобразование из описания инфраструктуры стрелочного вычислителя в специализированный HDL будет проще (M < N), то получится, что использование специализированного HDL даст нам выигрыш во времени разработки.

Проще всего пояснить данный довод следующей схемой:

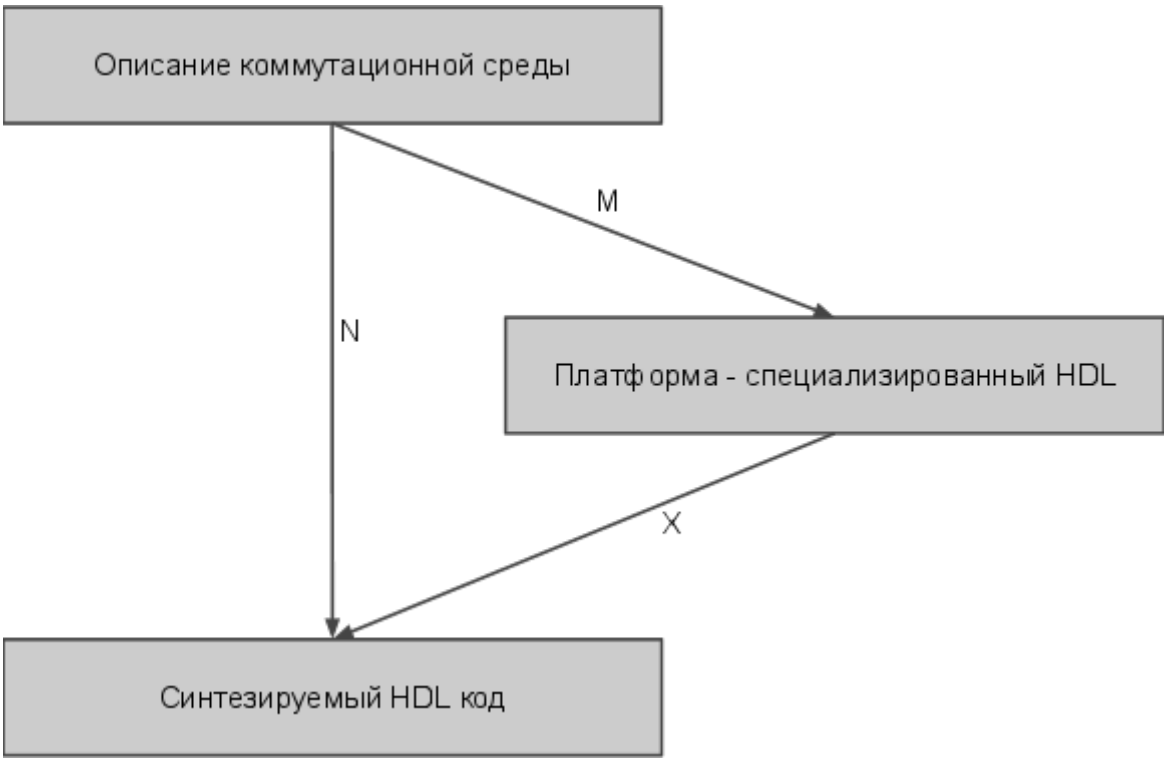

Рис 4. Специализированный HDL как платформа

Допустим, что время разработки транслятора для прямого преобразования из описания инфраструктуры стрелочного вычислителя в синтезируемый HDL-код равна N условным временным единицам. При том условии, что будет использоваться несколько различных реализаций инфраструктуры стрелочного вычислителя, время разработки транслятора складывается ($S_t = \sum_{i=1}^{k} N_i$, где k – количество различных реализаций трансляторов, а N_i – время разработки транслятора для прямого преобразования) и возрастает линейно с прибавлением каждой новой инфраструктуры. В данной и последующих формулах k – это количество разных реализаций инфраструктуры стрелочного вычислителя.

Тогда как в случае использования специализированного HDL время разработки транслятора для преобразования описания инфраструктуры стрелочного вычислителя в этот HDL будет равна M условным единицам, причем M < N. В случае большого количества разнообразных инфраструктур стрелочного вычислителя сложность точно также складывается и также возрастает линейно. Но при этом к ней один раз прибавляется время разработки транслятора из специализированного HDL в обычный синтезируемый HDL ($S_p = X + \sum_{i=1}^{k} M_i$, где X – время разработки транслятора для преобразования с использованием специализированного HDL, а M_i – время разработки транслятора для преобразования с использованием специализированного HDL). Единичное прибавление происходит за счет того, что условия преобразования из специализированного HDL в обычный зафиксированы и статичны. Иначе говоря, наш специализированный HDL является платформой, которая окупается при повторном использовании.

Между двумя рассмотренными вариантами мы можем наблюдать следующие соотношения, выраженные на графике:

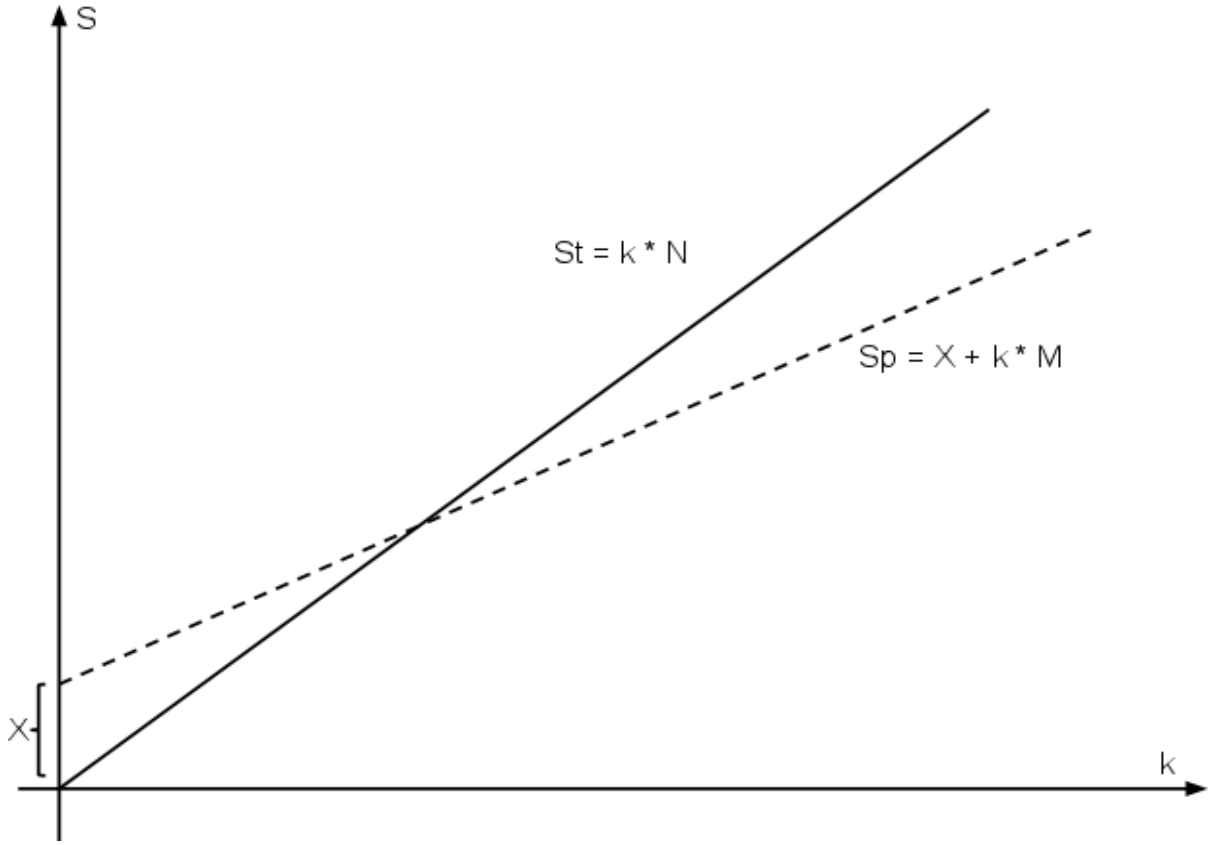

Рис 5. Соотношения между различными подходами к решению проблемы

Видно, что при большом количестве разнообразных изменчивых описаний инфраструктуры стрелочного вычислителя выигрывает вариант с применением специализированного HDL.

Теперь, осталось определиться как будет выглядеть наш специализированный HDL. При этом обязательно стоит учесть такое общепроектное требование, глася-щее что создаваемая инструментальная цепочка должна быть гомогенной. Это об-легчает разработку инструментальной цепочки и ее последующую поддержку – ведь необходимо знание лишь одного языка, языка на котором пишется программ-ный код разрабатываемых инструментов. В рамках системы автоматической гене-рации HDL-кода это означает, что финальное преобразование в HDL-код должно

откладываться как можно дольше. Иначе говоря, мы должны использовать диалект Лиспа, на котором пишется инструментальная цепочка, до последнего.

Данное требование определило вид, используемого в системе автоматического синтеза HDL-кода, специализированного HDL. Данный HDL построен на базе S-выражений, таким образом одна из препон при трансляции из описания инфраструктуры стрелочного вычислителя, а именно большая разница в синтаксисе, исчезает и именно поэтому можно говорить о том, что подобная трансляция проще, чем трансляция из описания инфраструктуры стрелочного вычислителя непосредственно в синтезируемый HDL-код.

Стоит отметить, что идея реализации специализированного HDL на S-выражениях не является чем-то принципиально новым, что ранее ни разу не приходило в голову людям, связанным с вычислительной техникой. Так например, существует язык описания аппаратуры Verilisp [5], в котором операторы языка Verilog “завернуты” в S-выражения. Данный язык является DSL построенным на таком диалекте Лиспа, как Common Lisp. К сожалению, данный язык прекратил свое развитие в 2009 году.

Еще можно упомянуть интерпретируемый лиспоподобный язык Cadence SKILL от компании Cadence. Данный язык вообще-то используется для расширения функциональности EDA сред разработки, выпущенных вышеупомянутой компанией. Но, существуют примеры использования данного языка для генерации HDL-кода, причем это происходит в форме, весьма похожей на используемую в специализированном HDL, рассматриваемом в данной работе.

Следует отметить малочисленность HDL на S-выражениях и их некоторую “стихийность” в развитии, которая иногда приводит даже к смерти разрабатываемого языка. По вышеизложенным причинам, командой “Arrow Compution Group” было принято решение разрабатывать свой специализированный HDL на S-выражениях, вместе с системой трансляции с этого языка на один из синтезируемых HDL.

Разработка языка

Очевидно, что в процессе разработки нового языка необходимо определить во что будет транслироваться описание на нашем новом языке, а также определить синтаксис и семантику нового языка. Процесс решения данной, несомненно важной для разработки, задачи описывается далее в данной главе.

Стоит отметить, что для разработки использовался язык Clojure. Необходимость использования данного языка программирования была поставлена в техническом задании на проект.

Анализ целевых языков

Человечеством было придумано огромное множество языков описания аппаратуры. Но не все из них подходят для нашего проекта — у многих из языков описания аппаратуры есть существенные (относительно проекта «Arrow Compution Group») недостатки:

- отсутствие или крайне низкое развитие свободных средств синтеза для данного языка описания аппаратуры. Стоит учитывать, что высокая стоимость промышленных синтезаторов не позволяет использовать их в развивающемся академическом проекте;
- слабая развитость сообществ и форумов разработчиков, использующих данный язык, как следствие — затрудненность получения помощи, когда она необходима, а официальная документация либо бедна, либо не дает ответа;
- ориентированность языка на описание аппаратуры на уровне описания поведения системы, а не на более низком уровне, например уровне регистровых передач. В рамках проекта «Arrow Compution Group» все высокоуровневое описание происходит на его верхних уровнях. От уровня же синтеза HDL-кода требуется лишь получить корректное,

синтезируемое описание системы на одном из HDL. Какая-либо дополнительная функциональность здесь излишняя.

В рамках проекта рассматривались следующие синтезируемые языки описания аппаратуры: Verilog, VHDL и XDL.

Verilog и VHDL являются общепризнанными средствами для промышленного синтеза аппаратуры. XDL – специализированное средство, позволяющее работать на уровне базовых элементов FPGA фирмы Xilinx (этап размещения, Floorplanning).

Данные HDL были проанализированы с применением следующих критериев:

- наличие открытых синтезаторов;
- количество экспертов в команде, имеющих опыт работы с данным HDL;
- простота изучения выбранного HDL;
- распространенность документации для заданного HDL;
- существование поддержки производителя или сообщества разработчиков для HDL.

Первый критерий выбора является критичным для разрабатываемого проекта ибо использование платных синтезаторов в академическом проекте является весьма трудноисполнимой задачей. А использование проприетарных синтезаторов чревато тем, что их производитель может внезапно изменить в нежелательную сторону либо некоторые особенности синтеза, либо изменить применяемые для работы с ним API. По данному критерию нашему проекту удовлетворяют следующие языки описания аппаратуры – Verilog и VHDL. К сожалению, для XDL есть лишь проприетарные средства для синтеза, предоставляемые компанией Xilinx.

Второй критерий – количество экспертов в команде, имеющих опыт работы с данным HDL важен тем, что без наличия людей, являющихся высококвалифицированными разработчиками на выбранном HDL, будет затруднительно написать транслятор, адекватно производящий преобразование в описание аппаратуры на

HDL из некоторого, используемого в проекте промежуточного описания. В группе “Arrow Compution Group”, есть эксперты в области таких языков описания аппаратуры как Verilog и VHDL.

Критерий простоты изучения был выбран с расчетом на то, что первый язык для реализуемой системы автоматического синтеза HDL-код должен быть достаточно простым, чтобы автор мог как можно быстрее начать разработку системы не тратя большого количества времени на изучение конечного HDL. По этому критерию первым идет Verilog, затем VHDL и лишь в конце XDL.

В принципе, подобное отставание XDL вполне связано с критерием распространенности документации на язык описания аппаратуры. Данный критерий можно трактовать как составной критерий, состоящий из критерия наличия документации на язык в открытом доступе и критерия объема документации на язык. XDL здесь проигрывает – документация на него закрыта от разработчика и предоставляется лишь платно и по запросу. Два первых языка – Verilog и VHDL здесь снова выигрывают.

Последний критерий – существование поддержки производителя или сообщества разработчиков показывает нам, насколько быстро и легко можно получить ответ на вопрос по языку, если вдруг документация по языку не способна предоставить такой ответ. Здесь, Verilog и VHDL опять таки выигрывают – у Verilog’а есть большое сообщество разработчиков на Западе, у нас же более популярен VHDL, в основном по историческим причинам.

После анализа критериев наметился аутсайдер среди рассмотренных языков – XDL. Он не удовлетворяет практически ни по одному из выбранных нами критериев. Из двух лидеров – Verilog и VHDL выигрывает Verilog, в основном, за счет наличия в студенческой научной группе “Arrow Compution Group”, большого количества экспертов по данному языку – три эксперта по Verilog'у против одного по VHDL, а также из-за простоты его изучения. Последний из доводов был особо важен для автора, ибо он уже имел небольшой опыт написания прошивок для

ПЛИС на Verilog’е и ему не нужно было начинать разбираться в конструкциях выбранного HDL с нуля.

Синтезируемое подмножество

Стоит отметить, что реализация транслятора, который использовал бы при трансляции все множество синтаксических конструкций языка Verilog отнюдь не тривиальная задача. Поэтому было решено взять некоторое используемое подмножество языка, выделенное из его синтезируемого подмножества. При этом, в качестве критерия выбора синтаксических конструкций языка было их частое использование в разнообразных проектах, но также учитывались и рекомендации по написанию описаний на Verilog’е. Так например, из всех операторов выбора в наше подмножество были отобраны лишь case и casez – оператор casex не рекомендовался для использования в коде на Verilog’е [6].

В процессе решения данной задачи возникла проблема – отсутствие единого и открытого стандарта на синтезируемое подмножество языка Verilog. Поэтому, пришлось предварительно выделить из языка его синтезируемое подмножество. Это было достигнуто при помощи анализа:

- репозитория кафедральных проектов (проект “Ancile”);
- репозитория IP-компонент opencores.org;
- документации промышленных средств синтеза (Xilinx ISE, Synopsys Synplify).

В итоге, было выделено следующее подмножество синтаксических конструкций Verilog’a:

- module/endmodule;
- типы данных – reg/wire/integer;
- типы выводов модуля – input/output/inout;
- always и @;
- posedge/negedge/edge;

- assign;
- = и <=;
- begin/end;
- if/then/else;
- операторы выбора – case/casez;
- for;
- localparam/param;
- побитовые операции – ? | & ^ ~^;
- логические операции – = && || !;
- описания констант вида 32’hdeadf00d;
- использование следующей конструкции для частичного извлечения значений из шин – instr[31:26];
- конкатенация – {val1, val2};
- бинарные арифметические операторы – + - * / %;
- унарные арифметические операторы – + и -;
- операторы сдвига – << >> <<< >>>;
- операторы отношения – != == === !== < > <= =>;
- синтаксические конструкции для объявления массивов и обращения к ним.

В дальнейшем, разработка Vericlo велась в рамках данных синтаксических конструкций. Были определены прямые преобразования между синтаксическими конструкциями обоих языков. Но, стоит отметить, что Vericlo при этом не ограничивается лишь повторением синтаксиса и семантики Verilog’a и отличается от него лишь использованием S-выражений.

Синтаксис и семантика

Специализированный язык описания аппаратуры, названный Vericlo, наследует органичным образом как синтаксис и семантику Verilog'a, так и синтаксис и семантику Clojure. В этом и кроется его необычность и отличие от "Verilog'a на S-выражениях".

Для выделения участков кода на Clojure используются функции вида on-clo и on-clo-macro. Действия, выполняемые данными функциями будут рассмотрены далее, в третьей главе. Сейчас же стоит отметить лишь то, что первая функция используется для вставки простого Clojure-кода, который перейдет на следующий этап трансляции "как есть". А вторая функция применяется, когда необходимо написать макрос на Clojure, который будет разворачиваться непосредственно в код на Vericlo, что может быть полезным для решения различных задач кодогенерации, что могут встать в будущем при использовании нашего языка описания аппаратуры.

Необходимо обратить внимание, что код на Clojure, может быть вставлен лишь внутри вышеприведенных функций – он не может использоваться ни в каких других местах. В то время как код, написанный на Vericlo может вставлять либо непосредственно после начала модуля (module …), либо внутри функции on-veri. В первом случае, явное использование кода на Vericlo подразумевается по умолчанию. Второй же случай предназначен для использования вставок кода на Vericlo внутри вставок кода на Clojure, что опять таки может быть полезным для решения задач кодогенерации.

Теперь, рассмотрим синтаксические конструкции Vericlo, взятые непосредственно из Verilog'a. Описание модуля на нашем языке описания аппаратуры выглядит так:

```
(module имя-модуля интерфейс-модуля тело-модуля)
```

Стоит отметить, что в Vericlo интерфейс модуля может быть как встроен в сам модуль, так и быть отдельным от него. Эта особенность пришла в наш язык из про-

мышленного языка описания аппаратуры, известного как VHDL. Данная возможность позволяет разработчику использовать один интерфейс для нескольких модулей, что позволяет, например, описать несколько различных по внутреннему устройству контроллеров, но имеющих одинаковый интерфейс. Это имеет особую важность в свете использования одного и того же интерфейса устройства, для подключения его, например, к общей шине. При наличии одного описания интерфейса нам достаточно лишь переиспользовать его в своих модулях. И кроме того, в случае изменения спецификации интерфейса подключения устройства к шине, нам достаточно поменять его описание лишь в одном месте, а не менять множество описаний в каждом из модулей.

Отдельное описание интерфейса модуля выглядит так:

```
(def-struct имя-интерфейса пары-вида-имя-порта-тип-порта)
```

При этом в описании модуля на место поля “интерфейс-модуля” достаточно подставить лишь соответствующее имя. В случае же использования встроенного в модуль интерфейса вместо имени интерфейса подставляется конструкция вида:

```
{пары-вида-имя-порта-тип-порта}
```

Отдельно стоит рассмотреть синтаксис определения переменных в Vericlo, который используется в том числе и при описании интерфейсов. Простая переменная типа reg описывается следующим образом:

```
:simple-var (Reg)
```

Стоит отметить, что имя переменной всегда должно начинаться с двоеточия – эта небольшая особенность языка, обусловлена большими преимуществами при разработке транслятора.

Шина, в рамках Vericlo, описывается следующим образом:

```
:bus (Wire [15 0])
```

Как видно, единственное различие – после спецификатора типа (Wire) идет указание разрядности шины в квадратных скобках. Обратиться же к отдельным проводам в данной шине можно следующим образом:

```
(part :bus [3 0])
```

что эквивалентно используемому в Verilog’e:

```
(bus[3:0]
```

Также можно использовать массивы из шин. Приведу их описание в Vericlo и соответствующее ему описание на Verilog’e, а также примеры обращения к отдельной шине в массиве:

```
:mem (array [0 1023] Reg [15 0])
reg [15:0] mem [0:1023]
```

Пример обращения:

```
(part (nth :mem 512) [7 0])
mem[512][7:0]
```

В случае определения внутренних для модуля переменных, их определения “заворачиваются” в следующее: S-выражение:

```
(!let определения-переменных)
```

Если же определения используются в интерфейсе модуля, то нужно дополнительно указать назначение отдельного элемента интерфейса – используется ли он только на вход, на выход или туда и обратно. Для этого перед типом интерфейса (который может быть только Reg или Wire) ставится спецификатор input, output или inout. Выглядит же это примерно так:

```
:simple-var (input Reg)
:bus (output Wire [15 0])
```

Теперь рассмотрим, что может входить в тело модуля. Во-первых, именно внутри него определяются внутренние для модуля переменные. Также, именно там могут располагаться различные синтаксические единицы, являющиеся операторами, перекочевавшими из Verilog’a. Необходимо отметить, что вставки кода на Clojure, не обернутые в on-clo могут быть вставлены лишь в тело модуля, но не куда-либо еще, в отличие от вставок on-clo-macro.

Рассмотрим, как представлены в Vericlo операторы, пришедшие из Verilog’a. Большинство из них являются просто завернутыми в S-выражения полными аналогами, но есть и исключения. Синтаксические конструкции, являющиеся подобными исключениями, приведены в таблице ниже:

Verilog	Vericlo
always @(negedge clk)	(always (sensivity-list negedge :clk))
out = data;	(sync= :out :data);
out <= data;	(async= :out :data);
data[counter]	(part :data :counter)
data_2[3][4:3]	(part (nth :data-2 3) [4 3])
{c, sum}	[:c :sum]
begin … end	(do …)
a % b	(rem :a :b)
a == b	(= :a :b)
a === b	(== :a :b)
a != b	(not= :a :b)
a !== b	(not== :a :b)
!a	(not :a)
x && y	(and :x :y)
x || y	(or :x :y)
a >> 2	(bit-shift-right :a 2)
a << 3	(bit-shift-left :a 3)
a >>> 2	(bit-shift-right-arithm :a 2)
a <<< 3	(bit-shift-left-arithm :a 3)
~x	(bit-not :x)
x & y	(bit-and :x :y)
x | y	(bit-or :x :y)
x ^ y	(bit-xor :x :y)

if (counter == 0) begin … end else begin … end	(if (= :counter 0) (do …) (do …))
for (i = 0; i < 16; i = i + 1) begin … end	(for i form 0 to 15 by 1 (do …))

Таблица 1. Синтаксические конструкции Vericlo синтаксические отличные от аналогичных в Verilog'e

Отдельно стоит отметить как реализован синтаксис определения числовых констант с заданной разрядностью и основанием системы счисления. Как уже было показано выше, в Verilog’e подобные константы выглядят так:

```
32'hdeadf00d
```

В Vericlo принято такое же описание подобных констант. Оно лишь оборачивается в S-выражение вида:

```
(% константа)
```

чтобы избежать появления синтаксических ошибок в коде, в случае прямого использования синтаксиса Verilog’a. Таким образом, вышеприведенную константу можно записать в Vericlo как:

```
(% "32'hdeadf00d")
```

Примеры

Теперь пришло время привести несколько примеров описания аппаратуры на Vericlo и соответствующих описаний на Verilog’e (использовались описания аппаратуры, используемые в одном из онлайн-руководств по языку Verilog [7]), в качестве иллюстрации того, что было изложено в предыдущей части работы. Вначале будет приводиться описание на Vericlo, а затем соответствующее ему на Verilog’e.

В нижеследующем описании простейшего мультиплексора демонстрируются базовые синтаксические структуры языка. Пример:

```
(module mux-using-assign
         {:din-0 (input)
          :din-1 (input)
          :sel (input)
          :mux-out (output Reg)}
```

```
     (always (sensivity-list (or :sel :din-0 :din-1))
                  (if (= :sel (% ''1'b0''))
                      (sync= mux-out din-0)
                      (sync= mux-out din-1))))

module mux_using_assign(
din_0, // Mux first input
din_1, // Mux Second input
sel, // Select input
mux_out // Mux output
);
input din_0, din_1, sel ;
output mux_out;
reg mux_out;

always @ (sel or din_0 or din_1)
begin : MUX
    if (sel == 1'b0) begin
          mux_out = din_0;
    end else begin
          mux_out = din_1 ;
    end

endmodule
```

В нижеследующем примере демонстрируется синтаксис оператора ветвления, который не был взят с Verilog'a, а является калькой на подобную синтаксическую конструкцию, предоставляемую одной из библиотек Common Lisp'a.

Дешифратор:

```
(def-struct decoder-interface
  :binary-in (input [3 0])
  :enable (input)
  :decoder-out (output (Reg [15 0])))

(module decoder-using-case decoder-interface
        (always (sensivity-list (or :enable :binary-in))
                (sync= :decoder-out 0)
                (if :enable
                  ;;match - case
                  ;;matchz - casez
                  (match :binary-in
                   (% "4'h0") (sync= :decoder-out (% "16'h0001"))
                   (% "4'h1") (sync= :decoder-out (% "16'h0002"))
                   (% "4'h2") (sync= :decoder-out (% "16'h0004"))
```

```
                    (% "4'h3") (sync= :decoder-out (% "16'h0008"))
                    (% "4'h4") (sync= :decoder-out (% "16'h0010"))
                    (% "4'h5") (sync= :decoder-out (% "16'h0020"))
                    (% "4'h6") (sync= :decoder-out (% "16'h0040"))
                    (% "4'h7") (sync= :decoder-out (% "16'h0080"))
                    (% "4'h8") (sync= :decoder-out (% "16'h0100"))
                    (% "4'h9") (sync= :decoder-out (% "16'h0200"))
                    (% "4'hA") (sync= :decoder-out (% "16'h0400"))
                    (% "4'hB") (sync= :decoder-out (% "16'h0800"))
                    (% "4'hC") (sync= :decoder-out (% "16'h1000"))
                    (% "4'hD") (sync= :decoder-out (% "16'h2000"))
                    (% "4'hE") (sync= :decoder-out (% "16'h4000"))
                    (% "4'hF") (sync= :decoder-out (% "16'h8000"))))))

module decoder_using_case (
   binary_in,       //  4 bit binary input
   decoder_out, //  16-bit  out
   enable            //  Enable for the decoder
 );
 input [3:0] binary_in  ;
 input  enable ;
 output [15:0] decoder_out ;

 reg [15:0] decoder_out ;

 always @ (enable or binary_in)
  begin
    decoder_out = 0;
    if (enable) begin
      case (binary_in)
        4'h0 : decoder_out = 16'h0001;
        4'h1 : decoder_out = 16'h0002;
        4'h2 : decoder_out = 16'h0004;
        4'h3 : decoder_out = 16'h0008;
        4'h4 : decoder_out = 16'h0010;
        4'h5 : decoder_out = 16'h0020;
        4'h6 : decoder_out = 16'h0040;
        4'h7 : decoder_out = 16'h0080;
        4'h8 : decoder_out = 16'h0100;
        4'h9 : decoder_out = 16'h0200;
        4'hA : decoder_out = 16'h0400;
        4'hB : decoder_out = 16'h0800;
        4'hC : decoder_out = 16'h1000;
        4'hD : decoder_out = 16'h2000;
        4'hE : decoder_out = 16'h4000;
```

```
        4'hF : decoder_out = 16'h8000;
      endcase
    end
  end
endmodule
```

В последнем примере демонстрируется синтаксис объявления внутренних переменных, использование параметров и работа с массивом регистров.

RAM с двунаправленной шиной для чтения/записи:

```
(module ram_sp_sr_sw
        {:ADDR-WIDTH (parameter 8)
         :DATA-WIDTH (parameter 8)
         :RAM-DEPTH (parameter (bit-shift-right 1 ADDR-WIDTH))
         :clk (input Wire)
         :address (input (Wire [(- :ADDR-WIDTH 1) 0]))
         :data (inout [(- :DATA-WIDTH 1) 0])
         :cs (input Wire)
         :we (input Wire)
         :oe (input Wire)}

        (let! :data-out (Reg [(- :DATA-WIDTH 1) 0])
              :mem (array [0 (- :RAM-DEPTH 1)] Reg [(- :DATA-WIDTH 1) 0])
              :oe-r Reg)

        (assign data (if (and :cs :oe (not :we)) :data-out z))

        (always (sensivity-list posedge :clk)
                (if (and :cs :we)
                  (sync= (nth :mem :address) :data)))

        (always (sensivity-list posedge :clk)
                (if (and :cs (not :we) :oe)
                  (do (sync= :data-out (nth :mem :address))
                      (sync= :oe-r 1))
                  (sync= :oe-r 0))))


module ram_sp_sr_sw (
  clk         , // Clock Input
  address     , // Address Input
  data        , // Data bi-directional
  cs          , // Chip Select
  we          , // Write Enable/Read Enable
  oe            // Output Enable
```

```
  );

 parameter DATA_WIDTH = 8 ;
 parameter ADDR_WIDTH = 8 ;
 parameter RAM_DEPTH = 1 << ADDR_WIDTH;

 input                   clk         ;
 input [ADDR_WIDTH-1:0] address     ;
 input                   cs          ;
 input                   we          ;
 input                   oe          ;

 inout [DATA_WIDTH-1:0]  data        ;

 reg [DATA_WIDTH-1:0] data_out ;
 reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];
 reg                  oe_r;

 assign data = (cs && oe &&  ! we) ? data_out : 8'bz;

always @ (posedge clk)
 begin : MEM_WRITE
    if ( cs && we ) begin
        mem[address] = data;
    end
 end
always @ (posedge clk)
 begin : MEM_READ
   if (cs &&  ! we && oe) begin
     data_out = mem[address];
     oe_r = 1;
   end else begin
     oe_r = 0;
   end
 end
 endmodule
```

Предложенный подход к трансляции

Для трансляции из нашего специализированного HDL, названного Vericlo, в HDL код был придуман и предложен следующий подход, изложенный в нижеприведенной нотации [8]:

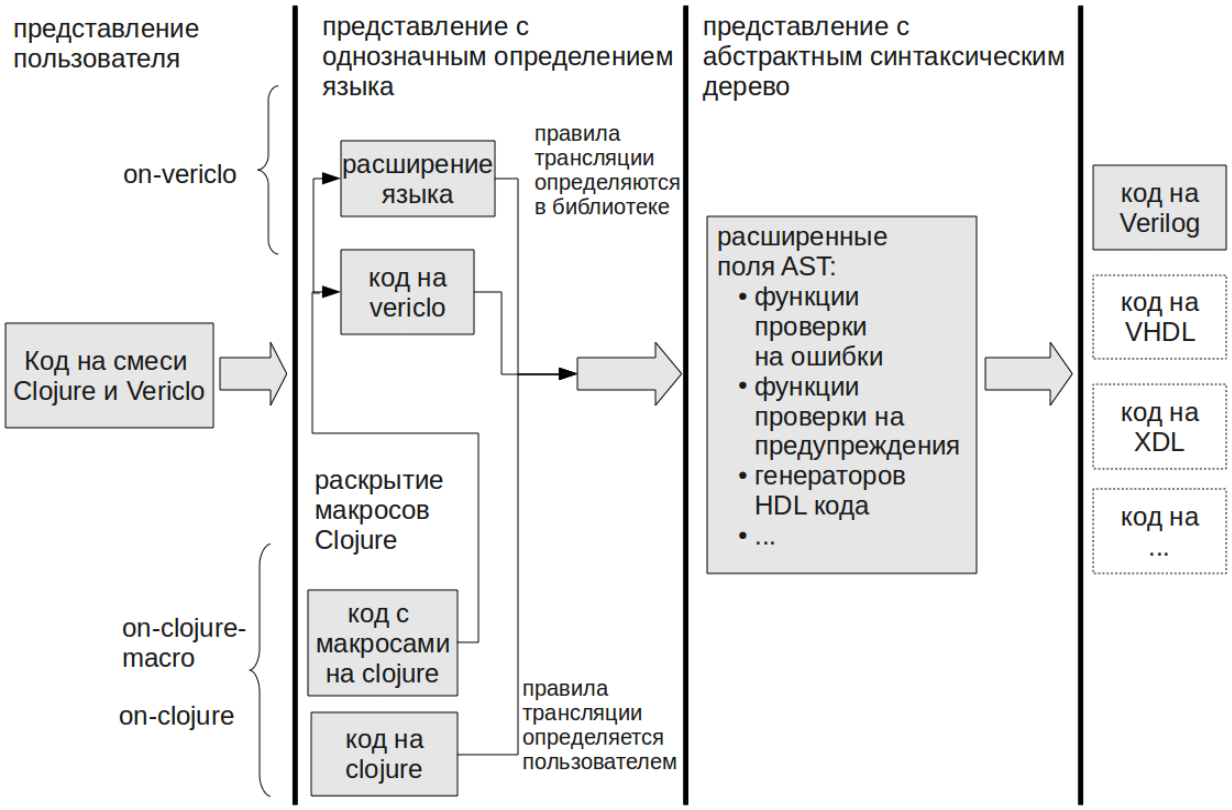

Рис 6. Подход к трансляции с Vericlo на Verilog

Стоит отметить, что в данном подходе не используется транслятор, в его традиционном понимании – упрощенно говоря, код на Vericlo способен разворачивать сам себя в код на Verilog'e. Впрочем, рассмотрим все поподробнее, начиная с первого уровня.

Вначале рассмотрим, что происходит на первом уровне, который носит название уровня представления пользователя. Описание аппаратуры, на котором мы работаем на этом уровне, вполне может содержать в себе вставки кода на Clojure помимо

кода на Vericlo. В итоге, описание аппаратуры на нашем специализированном HDL может сочетать в себе несколько различных по семантике языков. На следующих уровнях данное описание разделяется на два вышеупомянутых языка. Для этого применяются специальные on-функции, которые на схеме стоят слева от фигурных скобок.

Впрочем, это уже относится ко второму уровню. На втором уровне происходит еще кое-что, помимо разделения между собой кода на Clojure и кода на Vericlo. Происходит однозначное определение используемого языка для синтаксических единиц. Как пример, синтаксическая единица (nth …) работает совершенно по разному, в Clojure она выдает пользователю определенный элемент списка, а в Vericlo она же выдает значения определенных шин/регистров в массиве из них. Какую из версий nth нам надлежит использовать - сходу непонятно, понять это можно лишь в контексте языка.

Для однозначного разделения подобных синтаксических единиц между собой используются namespace. В итоге – в коде на Vericlo вызовется именно та версия nth, которая относится к Vericlo, и то же самое и для Clojure. Определение того, какой namespace для какой синтаксической единицы надлежит использовать, происходит как раз при помощи вышеупомянутых on-функций.

Как было замечено ранее, у нас есть два вида вставок кода на Лиспе. Первая содержит в себе просто чистый код на Лиспе, а вторая содержит в себе макрос, который будет разворачиваться в код на Vericlo. Вставки второго вида разворачиваются также как раз на рассматриваемом нами в данный момент уровне. Результат раскрытия макроса подставляется “как есть” в код на Vericlo.

Стоит отметить, что на этом, рассматриваемом нами уровне, введена возможность для безболезненного расширения Vericlo. Если мы захотим, например, добавить в язык возможность введения ограничений на размещение блоков на кристалле и транслировать затем все это в netlist или в XDL, то придется соответствующим образом расширять язык. А в будущем неизбежно случится такая ситуация, что опи-

сание аппаратуры, выполненное с применением вышеупомянутых ограничений на размещение блоков не получится транслировать в Verilog или иной HDL, не поддерживающий ограничений на размещение блоков.

Чтобы избежать этого, были введены расширения языков. Используя расширение пользователь может использовать в своей работе какие-либо особенности выбранного HDL, но делает он это исключительно на свой страх и риск.

Итогом всех этих действий, выполненных на втором уровне, является смесь кода на Vericlo и чистого кода на Clojure, которая передается дальше на следующий этап трансляции. Отмечу, что под чистым кодом на Clojure подразумевается код, который не содержит в себе макросов, напрямую разворачивающихся в Vericlo,

На третьем этапе наш код на обоих языках наконец-то объединяется. В функциях, которые использовались изначально (например +, -, nth) сокрыты генераторы структур из которых строится синтаксическое дерево нашего кода. В самом общем виде, данные структуры содержат в себе следующие поля: оператор, его операнды и имена специальных функций. Эти функции могут быть генераторами кода на выбранном конечном HDL или предназначаться для проверки исходного кода на ошибки. Стоит отметить, что некоторые из специальных функций вызываются на этом же этапе, а именно, функции для разнообразных проверок. Если код не прошел проверку, то процесс трансляции останавливается и пользователю выводится сообщение об ошибке. В обратном случае, мы проходим на следующий этап и, кроме того, при этом мы уже можем гарантировать синтаксическую верность генерируемого нами кода – ведь возможные ошибки уже были отловлены.

На последнем, четвертом этапе мы просто пробегаемся по узлам созданного нами синтаксического дерева, вызывая функции-генераторы, которые строят из структур в узлах соответствующие им представления на HDL.

Заключение

В рамках данной работы был предложен подход к решению задачи синтеза описания аппаратуры на синтезируемом HDL из описания инфраструктуры стрелочного вычислителя. Данный подход заключается в использовании платформы в виде HDL Vericlo при трансляции из описания инфраструктуры в HDL код.

В процессе работы были решены следующие задачи:

- был предложен подход к решению задачи генерации HDL описания аппаратуры на основе описания инфраструктуры стрелочного вычислителя;
- были проведен анализ несколько существующих HDL-языков и по совокупности критериев был выбран Verilog;
- было выделено синтезируемое подмножество выбранного языка, необходимое для успешной генерации синтезируемого описания аппаратуры;
- были разработаны основные синтаксические конструкции для специализированного языка описания аппаратуры Vericlo, была определена семантика данного языка, наследуемая сразу от Verilog'а и Clojure;
- были разработаны и предложены этапы трансляции из специализированного языка описания аппаратуры в синтезируемый HDL.

Язык, используемый в разработанной платформе, планируется использовать в том числе и для составления HDL-описаний специализированных функциональных блоков, что позволит сделать разрабатываемую в рамках проекта "Arrow Compution Group" систему, как можно более гомогенной и в итоге ее будет легче поддерживать и видоизменять при необходимости.

В будущем, в систему автоматической генерации HDL-кода будет добавлена возможность генерировать целевое описание аппаратуры на некоторых других HDL,

что обеспечит возможность получить описание стрелочного вычислителя на том языке описания аппаратуры, который необходим заказчику.

Также, возможности языка, связанные с кодогенерацией и построением на нем различных DSL получат свое развитие в рамках проекта, что позволит описывать инфраструктуру стрелочного вычислителя непосредственно на DSL, построенном над Vericlo.

Глоссарий

Стрелка — носитель вычислительного процесса .

Примитивы или примитивные стрелки — стрелки, находящиеся на нижнем уровне иерархии в стрелочной программе, и функциональность которых не определяется в рамках стрелочной модели вычислений.

Составная (иерархическая) стрелка — стрелка, полученная путём композиции других стрелок .

Стрелочная программа — составная стрелка, описывающая целевой алгоритм работы системы.

Функциональный блок — аппаратная реализация примитивной стрелки .

Расписание — стрелочная программа, приведённая к виду, пригодному для аппаратной интерпретации. Состоит из отдельных микрокоманд.

Стрелочный вычислитель — непрограммируемый процессор, выполняющий расписание.

Инфраструктура стрелочного вычислителя — описание взаимосвязей между функциональными блоками стрелочного вычислителя.

Модель вычислений — набор сущностей и правил их взаимодействия между собой.

Список литературы

1) Philippe Coussy. High-Level Synthesis: from Algorithm to Digital Circuit / Philippe Coussy and Adam Morawiec – Springer Publishers. – 2010.
2) Wang Laung-Terng. Electronic Design Automation: Synthesis, Verification, and Test / Wang Laung-Terng, Chang Yao-Wen, Cheng Kwang-Ting. – Morgan Kaufmann Publishers, 2009.
3) Басов М.А. Проектирование архитектуры модифицированного потокового вычислителя: магистер. дис. / СпбГУИТМО – Спб., 2011 – 148 с.
4) Ряховский А.С. Инструментальная цепочка модифицированного потокового вычислителя: магистер. дис. / СпбГУИТМО – Спб., 2011 – 69 с.
5) Verilisp [Электронный ресурс] / Электрон. дан. – Режим доступа: http://verilisp.sourceforge.net/, свободный.
6) Don Mills. RTL Coding Styles That Yield Simulation and Synthesis Mismatches / Don Mills – Rev 1.1 - SNUG 1999.
7) Verilog Examples [Электронный ресурс] / Электрон. текстовые дан. – Режим доступа: http://asic-world.com/examples/verilog/index.html, свободный.
8) Debasish Ghosh. DSL's in Action / Debasish Ghosh – Manning Publications, 2010.

Приложение 1. Примеры описаний аппаратуры на языке Vericlo

Описание RAM с двунаправленной шиной чтения/записи. Интерфейс описывается отдельно от модуля.

```
;; Original verilog code you can find here: http://www.asic-world.com/exam-
ples/verilog/ram_sp_sr_sw.html#Single_Port_RAM_Synchronous_Read/Write

;;Интерфейс модуля можно описать отдельно от модуля, как структуру.
;;Имя этой структуры будет идти третьим элементом в списке - описании моду-
ля
;;
;;Причем переменные (выводы, регистры и т.п.) внутри стуктуры можно
;;описывать двумя способами.
;;Первый способ представлен ниже. В нем можно переиспользовать описание
;;типа если есть несколько переменных с таким типом при помощи ключевого
;;слова with-type.
(def-struct one-interface
  :with-type (:ADDR-WIDTH :DATA-WIDTH parameter 8)
  :RAM-DEPTH (parameter (bit-shift-right 1 ADDR-WIDTH))
  :clk (input (Wire))
  :address (input (Wire [(- :ADDR-WIDTH 1) 0]))
  :data (inout [(- :DATA-WIDTH 1) 0])
  :with-type (:cs :we :oe input Wire))
)

;;Без скобочек
(def-struct one-interface
  :with-type (:ADDR-WIDTH :DATA-WIDTH parameter 8)
  :RAM-DEPTH (parameter (bit-shift-right 1 ADDR-WIDTH))
  :clk (input Wire)
  :address (input Wire [(- :ADDR-WIDTH 1) 0])
  :data (inout [(- :DATA-WIDTH 1) 0])
  :with-type (:cs :we :oe input Wire))

;;Второй способ заключается в том, что мы описываем каждую переменную
;;по отдельности, даже если есть несколько переменных с одним и тем же ти-
пом:
(def-struct one-interface
  :ADDR-WIDTH (parameter 8)
  :DATA-WIDTH (parameter 8)
  :RAM-DEPTH (parameter (bit-shift-right 1 ADDR-WIDTH))
  :clk (input Wire)
  :address (input (Wire [(- :ADDR-WIDTH 1) 0]))
  :data (inout [(- :DATA-WIDTH 1) 0])
  :cs (input Wire)
  :we (input Wire)
  :oe (input Wire))
```

```
(module ram-sp-sr-sw one-interface
        (let! :data-out (Reg [(- :DATA-WIDTH 1) 0])
              :mem (array [0 (- :RAM-DEPTH 1)] Reg [(- :DATA-WIDTH 1) 0])
              :oe-r Reg)

        ;;Можно передавать константы просто в виде чисел (помимо них
        ;;допустимо z-состояние -- z). Тогда наш транслятор сам определит
        ;;разрядность константы по переменной, которой она присваивается.
        ;;Если же нужно указать разрядность явно, то следует передать
транслятору
        ;;нашу константу как (% "8'hDE"), тогда она будет вставлена "как
есть" в
        ;;результирующий код на верилоге. Этот подход также можно использо-
вать и в
        ;;операторе case для его веток.
        ;;Вставлять "как есть" готовый код на верилоге в наш верикловский
код
        ;;считается недопустимым. Такое можно делать только с константами!
        (assign data (if (and :cs :oe (not :we)) :data-out z))

        (always (sensivity-list posedge :clk)
                (if (and :cs :we)
                  (sync= (nth :mem :address) :data)))

        (always (sensivity-list posedge :clk)
                (if (and :cs (not :we) :oe)
                  (do (sync= :data-out (nth :mem :address))
                      (sync= :oe-r 1))
                  (sync= :oe-r 0))))
```