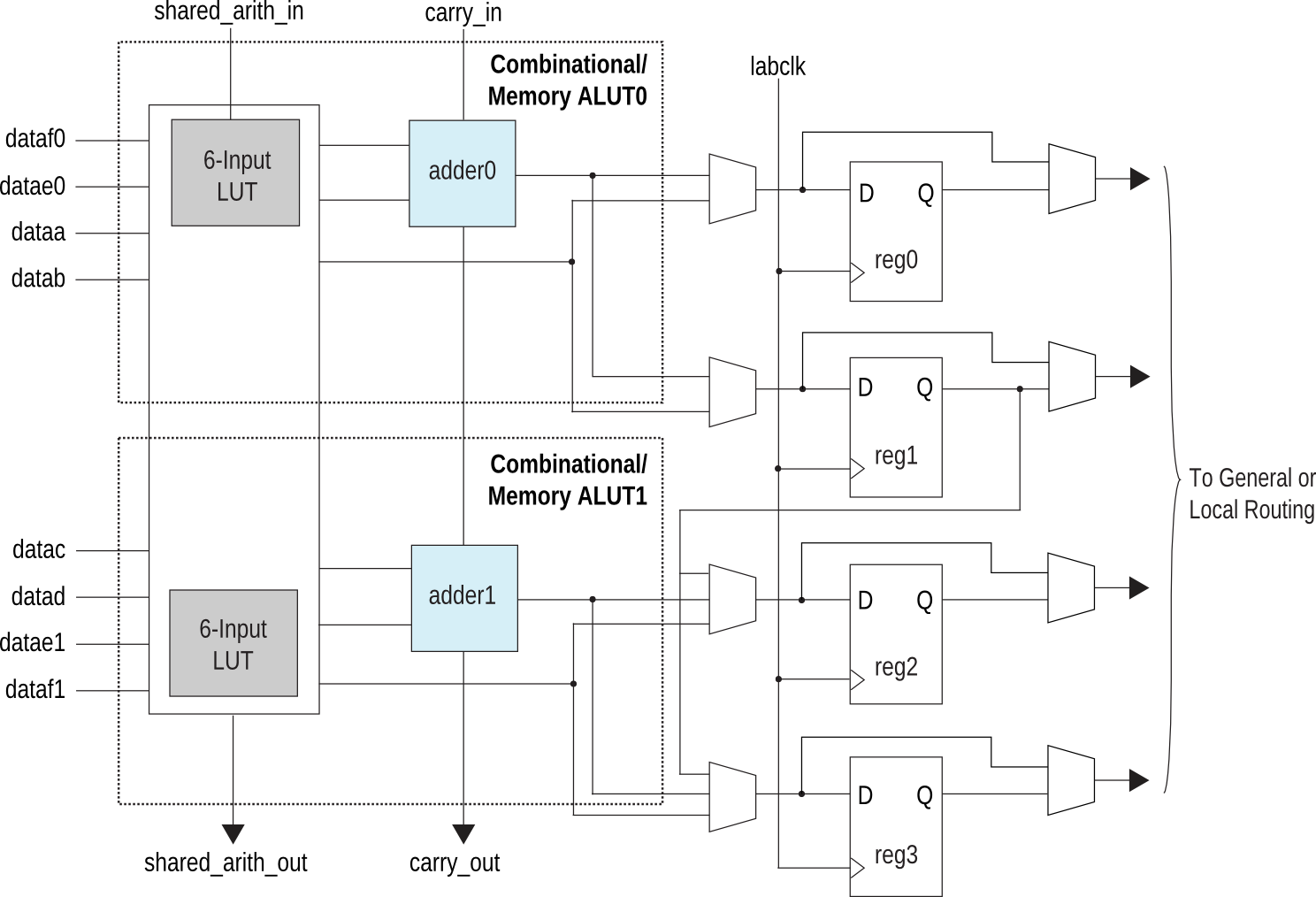
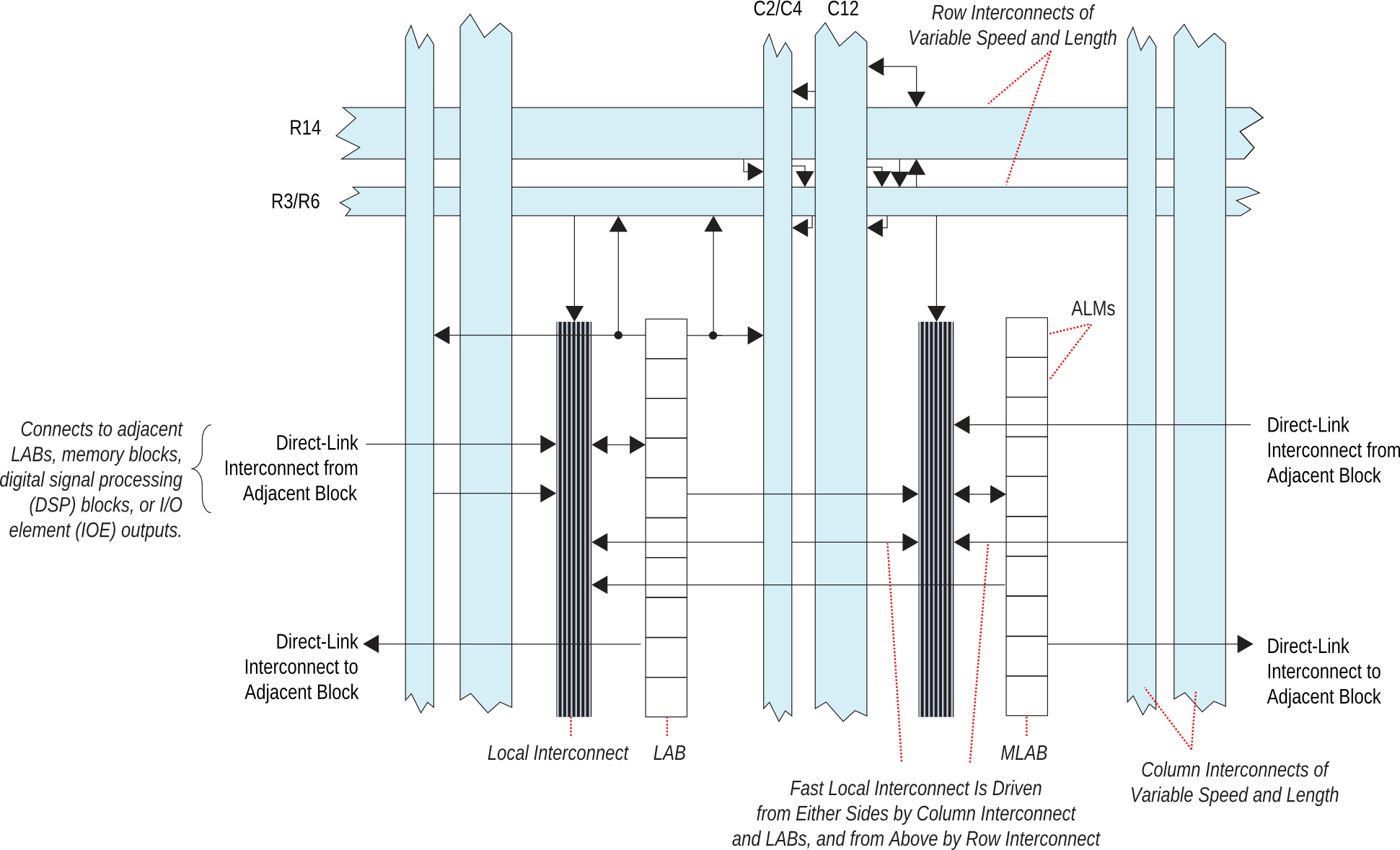
# Композиционный подход для проектирования аппаратного обеспечения

Любое аппаратное обеспечение, которое предназначено для вычислений может быть проектировано композиционным подходом. Ведь с нашей точки зрения программное обеспечение отличается от от аппаратного только исключенными абстракциями. Можно рассматривать программное решение как память с программой плюс процессор. Таким образом программное обеспечение – вариация аппаратного обеспечения, просто более специфическая. Мы же предлагаем избавиться от такого разделения на программу и процессор. Ведь сегодняшнее состояние индустрии позволяет уйти от разбиение на программу и процессор, получив специфический вычислитель для решения конкретной задачи.

Рассмотрим ближе аппаратную платформу, на которой будет вестись разработка. Такой платформой будет ПЛИС(программируемая логическая интегральная схема). На базе ПЛИС путем коммутации логических ячеек, можно строить любые вычислительные устройства, в том числе микропроцессоры, вычислители хэш-функций, перекодировщики сигнала и т. п.

Рис. 1. Структура логической ячейки ПЛИС.

ПЛИС, или FPGA, имеют матричную структуру из логических ячеек с добавлением других специализированных элементов: например, памяти и у множителей и разновидностей вышеупомянутых элементов. Типичная логическая ячейка (рис.1) состоит look-up таблиц, сумматоров, мультиплексоров, триггеров. Такие элементы позволяют реализовать как комбинационную, так и последовательностную (триггеры в логических ячейках) логику высокой сложности. Они коммутируются между собой специальными межсоеидинтельными шинами (рис. 2).

Рис. 2. Межсоединения в ПЛИС.

Систематического подхода к проектированию аппаратного обеспечения не существует, есть только разрозненные приемы конструирования, которые не охватывают весь объем конструкторских задач от задумки до решения. Для применения предложенного способа проектирования аппаратного обеспечения, важным качеством платформы, на которой выполняется проектирование, есть возможность менять свою структуру, хоть это и не есть необходимым условием, так как ресурсы ПЛИС крайне ограниченные и на ней нельзя решить сложную задачу целиком. Потому есть смысл по мере вычислений перепрограммировать ПЛИС несколько раз под самые затратные вычислительные процессы, таким образом ускорив вычисления в целом. На сегодняшний день разработчики ПЛИС не предоставляют доступ к спецификациям файлов, которые непосредственно программируются на ПЛИС. Стало быть и нет возможности явным образом коммутировать логические элементы на ней. Вместо этого предоставляется доступ к логическим синтезаторам HDL-языков, компоновщикам и трассировщикам синтезированной логики. Сейчас существует два популярных и приблизительно эквивалентных по функциональности HDL-языка: Verilog и VHDL. Именно на их уровне есть возможность разрабатывать решения на ПЛИС, именно они в дальнейшем будут рассматриваться.

## Этапы проектирования от HDL-языка и ниже

Не детализируя синтаксис HDL-языков давайте вкратце рассмотрим стадии проектирования вычислителей от HDL-языков и ниже, иными словами именно те этапы, которые в большинстве своем автоматизированны и на которые разработчик полноценно влиять не может.

HDL-языки позволяют проектировать на уровне регистровых передач (RTL — register transfer level). При разработке аппаратного обеспечения уровень регистровых передач (RTL) – уровень абстракции, который моделирует синхронные цифровые схемы в понятиях потоков цифровых сигналов (данных) между регистрами, а так же логические операции над этими сигналами. Синхронные схемы, которые моделируются на уровне регистровых передач состоят из регистров и комбинационной логики. Регистры (обычно это D-триггеры) синхронизируют работу схемы по фронтам тактового сигнала и они единственные элементы в схеме, обладающие памятью. Комбинационная логика позволяет вычислить все логические функции в схеме и обычно состоит из логических элементов.

Далее происходит крупноблочное планирование, когда размещаются основные функциональные блоки. В современном процессе проектирования, размещения создаются после выработки RTL. В зависимости от методологии разработки, определение термина может варьироваться. В некоторых подходах крупноблочное планирование может разбивать всю площадь кристалла на выровненные по осям прямоугольники, в которых размещаются блоки аппаратного решения. При таком разбиении необходимо, что бы выполнялось много условий и оптимизационных требований: площадь, занимаемая блоков, соотношение сторон, оценка количества будущих межсоединений и т.д. Поиск удачного крупноблочного размещения -- задача NP-сложная, занимающая большое количество вычислительных ресурсов. Таким образом наиболее распространенный подход ее решения -- использование разнообразных эвристик. Иным решением этой задаче есть использование ограниченное количество способов разбиения, что позволяет уменьшить вычислительные затраты.

Логический синтез – процесс, который из желаемого поведения схемы, описвнного на уровне регистровых передач, получает реализацию устройства на логических вентилях. Обычно синтез выполняется специализированной программой для синтеза, которая может выполнять синтез как для ПЛИС, так и для заказных ИС (ASIC). В подавляющем большинстве случае процесс синтеза автоматизирован.

Одним из этапов синтеза есть минимизация логики, которая, ранее выполнялась картами Карно, а сейчас методами подобными методу Квайна Маккласки, которые легче запрограммировать на компьютере или эвристическими минимизаторами (например, Эспрессо). Это всё касается комбинационной логики. Но кроме этого решается и задача минимизации состояний и синтеза конечных автоматов. Кроме этого проводятся технологически зависимые оптимизации, на которые влияют тип логических ячеек, специализированные ячейки на кристалле, необходимые задержки, необходимая потребляемая мощность, необходимая занимаемая площадь на кристалле и т. д.

Компоновка – важный шаг при разработке аппаратного обеспечения, во время которого определяется точное расположение каждого компонента схемы на кристалле. Плохая компоновка не только плохо влияет на производительность чипа, но так же может сделать невозможным его производство из-за слишком большой длины проводников. Таким образом компоновщик решает следующие задачи:

* Минимизация суммарной длины проводников, что приводит к минимизации размера кристалла и минимизация затрат
* Минимизация задержек. Период тактового сигнала на чипе определяется задержкой по самому длинному пути от регистра к регистру в схеме (critical path). Компоновщик должен гарантировать, что ни один из путей не превысит максимальный, предварительно заданный, critical path.
* Балансировка нагрузки на ресурсы межсоединений. Если в одном месте скапливается слишком много компонентов с большим количеством соединений, то есть вероятность, что локальных ресурсов межсоединений может не хватить для всех компонентов.
* Потребляемая мощность. Очень важно равномерно распределить по чипу источники наибольшего тепловыделения во избежание горячих точек и для более плавного градиента температур на чипе.

После компоновки выполняется разводка, во время которой все размещенные элементы соединяются между собой. Задача, которая решается ПО для разводки, одна. Дано именованное множество точек на чипе, разбитое на группы. В каждой группе их нужно соединить между собой проводником, такие группы называют узлами. Все точки одного узла имеют один и тот же электрический потенциал. При разводке, должен выполнятся набор правил разводки, который определяется характеристиками ПЛИС, а так же правила, заданные инженерами, например про максимальную задержку сигнала в проводнике.

## HDL языки

Языки описания аппаратного обеспечения позволяют формально и точно описать электрическую схему, что позволяет в дальнейшем автоматически их анализировать и симулировать, а также компилировать HDL-код в более низкоуровневую спецификацию схем, например, набор фотошаблонов для ИС или схему на логических ячейках ПЛИС.

HDL много чем подобен языкам программирования таким, как Си. Это текстовое описание схемы, состоящее из выражения, утверждений и управляющих структур. Одно важное различие между большинством языков программирования и HDL-языками в том, что HDL-языки явно оперируют временем. HDL-языки стали составной частью САПРов, в которых проектируются сложные схемы, такие как микропроцессоры. Из-за взрывоподобного роста сложности электронных схем с 1970х годов (закон Мура), разработчикам схем было необходимо выполнять описание логики на высоком уровне без привязки к технологии (КМОП, схемы на биполярных транзисторах).

HDL-языки были созданы именно для реализации уровня абстракции регистровых передач (RTL), что бы моделировать поток данных и поведение схемы во времени. HDL-языки -- средства текстового выражения структуры электронных схем и их поведения во времени. Так же как и языки программирования с поддержкой параллелизма, HDL - синтаксис и семантика включают возможность явного выражения параллелизма. Хотя в отличии от других языков, они так же поддерживают явную работу со временем, что есть характерным для аппаратного обеспечения. К слову, языки,которые предназначены только для выражения соединений между иерархическими блоками подпадают под классификацию языков описания соединений (netlist languages), а не HDL. HDL может использоваться для описания конструкций структурным, поведенческим и RTL способами. В последних двух случаях синтезатор определяет архитектуру и соединение логических ячеек.

Описания аппаратного обеспечения на HDL-языках могут быть так же смоделированы на компьютере. Программа, которая разработана для реализации семантики выражений языка и симуляции поведения во времени, позволяет разработчику аппаратного обеспечения моделировать аппаратное обеспечение перед тем, как оно будет создано физически. Именно эта способность моделирования дает иллюзию того, что HDL -- языки программирования, в то время когда они определены как языки описания или языки моделирования. Симуляторы способны поддерживать как дискретное (цифровое), так и непрерывное (аналоговое) моделирование. Для обоих видов моделирования существуют HDL-языки, но в контексте нашей работы нас интересует именно цифровое моделирование.

Определенно возможно представить семантику аппаратного обеспечения с использованием традиционных языков программирования, таких как С++, которые работают с управляющей логикой, которая противоположна логике потоков данных, которая необходима для реализации структур аппаратного обеспечения. Для поддержки логики работы с потоками данных, программы должны быть дополнены громоздкими библиотеками. В основном языки программирования не позволяют явно выражать ход времени и таким образом функционировать в полном объеме как HDL-языки. Перед появлением SystemVerilog, интеграция С++ с логическим симулятором была одним из немногих путей к использованию ООП для верификации аппаратного обеспечения. SystemVerilog -- первый популярный HDL, который поддерживает объектную ориентацию и сборщик мусора.

Используя правильное подмножество HDL-языка, синтезатор или инструмент для синтеза логики может выделить логические операции из выражений языка и произвести эквивалентный список соединений (netlist) аппаратных примитивов (например, логических ячеек) лоя реализации заданного поведения. Сами синтезаторы в основном игнорируют выражения со временем в тексте. Синтезаторы цифровой логики, например, используют фронты тактовых импульсов, как исредство работы со временем. Но наличие синтезируемого подмножества языка само по себе не делает его языком описания аппаратного обеспечения.

Cуществует два популярных HDL-языка для работы с цифровой логикой: Verilog и VHDL, которые способны описать как комбинационную, так и последовательностную логики. В дальнейшем будет вестись дело с Verilog.

## Язык Verilog

Для начала рассмотрим синтаксис языка Verilog. Полная спецификация языка Verilog оформлена в стандарт IEEE 1364. Тут же будут выражены только основные положения языка.

Схема на языке Verilog – иерархическая структура модулей, на входы которой попадают сигналы, изменяющиеся во времени. Модуль описывается синтаксической конструкцией

module modulename(*список портов разделенный комой*);

endmodule

Эти модули могут быть иерархически подчиненными

module newmodule(*список портов разделенный комой*);

modulename m1(*список портов разделенный комой*);

endmodule

Порты в таких модулях могут быть трех типов:

* input – вход;
* output – выход;
* inout – двунаправленный;

Типы портов устанавливают при определении модуля:

**module** modulename(a, b, c);

**input** [31:0] a;

**output** [15:0] b;

**inout** c;

**endmodule**

При этом в квадратных скобках задается количество бит порта. В примере параметр *a* 32 битный, параметр *b* 16 битный, а так у параметра *c* не указана ширина порта, он автоматически становится однобитным.

Существует два основных представления данных в Verilog: узел (net) и регистр (reg). Узел сам по себе данные не хранит, он их только именует. Каждый узел должен иметь как минимум один источник (driver) данных.. Таким источником может выступать регистр либо другой узел.. Источник узла не может динамически сменятся, он всегда один и тот же. Самым популярным и простым типом узла есть *wire*. Узлы можно определять только внутри модуля. Пря назначения узлу источника используется присвоение через ключевое слово *assign*.

**module** modulename(a, b, c);

**input** [31:0] a;

**input** [31:0] b;

**output** [31:0] c;

**wire** [31:0] a1;

**wire** [31:0] b1;

**wire** [31:0] c1;

**assign** a1 = {a[31:16], b[15:0]};

**assign** b1 = {b[31:16], a[15:0]};

**assign** c1 = a1 + b1;

**assign** c = c1;

**endmodule**

Представленный пример избыточен, но иллюстрирует работу с определением источника для узла. Источником для узла может быть не только другой проводник либо регистр, но и комбинационное выражение. *a1* и *b1* назначены конкатенации половинок входных портов *a* и *b*, а выходному узлу *c* (если для портов не обозначается явно их тип, то по умолчанию он wire) – сумма a1 и b1 из узла c1. Такой модуль, очевидно полностью может быть определен в терминах комбинационной логики. Для логики последовательностной необходимо место где хранить значения. Таким местом есть регистры. Регистр, как и net, есть упорядоченным набором бит, к каждому зи которых разработчик может получить доступ, но кроме всего другого, он может хранить информацию, и, таким образом, вовлекаться в построение последовательностных схем. Переменную регистр можно определить следующим образом:

**reg** [31:0] a;

Это регистр с именем *а* и размером 32 бита. Регистр не должен иметь постоянного источника. Он может изменять свое значения по событиям в схеме.

Можно также объявлять массивы таких регистров, таким образом моделируя память.

**reg** [31:0] a [0:7];

Таким образом был задан массив 32-битных регистров из 8 элементов.

Для того, что бы описать реакцию схемы на событие, используют структуру *always*.

**always** @(событие) **begin**

последовательность выражений;

**end**

В действительности структура *always* имеет более богатый синтаксис, но это наиболее популярное ее употребление. Событием в *always* может быть, в общем случае, перечисление сигналов и способов их изменения, на которые следует реагировать.. Изменение сигнала можно записать следующим образом:

* posedge a – положительный фронт сигнала a;
* negedge a – отрицательный фронт сигнала a;
* a – произвольный фронт сигнала a;

**always** @(**posedge** CLK **or** **negedge** a) **begin**

последовательность выражений;

**end**

Блок *always*, изображенный выше выполняется только в случае если фронт тактового импульса *CLK* нарастает, либо фронт сигнала *a* спадает. Кроме комбинационных выражений, между *begin* и *end* могут выполнятся и специфические последовательностные. В первую очередь это касается операции присвоения. Операций присвоения есть два типа: «=» и «<=». Первый оператор это тот, который мы подразумеваем, когда говорим присвоение в языках программирования (блокирующее присвоение), второй оператор – описывает параллельное (неблокирующее) присвоение . Для того, что бы понять в чем разница между ними, рассмотрим примеры.

**always** @(**posedge** CLK) **begin**

a = b;

c = a;

**end**

**always** @(**posedge** CLK) **begin**

a <= b;

c <= a;

**end**

Есть также параллельные блоки, которые заключаются в *fork* и *join*. Все операции там выполняются одновременно без соблюдения порядка. В данном кратком описании языка Verilog эти блоки можно использовать в тех же местах, где используются блоки *begin-end*, но они не синтезируемы.

В первом случае в *c* окажется текущее значение *b*, а во втором – старое значение *a*. Неблокирующее присвоение более экономично для ресурсов кристалла, чем блокирующее, так как отпадает необходимость запоминать промежуточные значения переменных.

К управляющим структурам языка Verilog относят:

* оператор ветвления *if*;
* оператор множественного выбора *case*;
* оператор циклирования *repeat*;
* операторы циклирования *while*, *for*, *forever*.

Эти операторы похожи на свои аналоги в программировании. Вкратце рассмотрим их синтаксис.

Для оператора if.

**if** (условие1) **begin**

Последовательность1

**end**

**else** **if** (условие2) **begin**

Последовательность2

**end**

**else** **begin**

Последовательность3

**end**

Если выполняется первое условие, то выполняется *последовательность1*, иначе происходит проверка второго условия. Если в второе условие истинно, то выполняется *последовательность2*. Если же оба условия не истинные, то выполняется *последовательность3*.

Оператор *case*

**case**(величина)

значение1: **begin**

Последовательность1

**end**

значение2: **begin**

Последовательность2

**end**

значение3: **begin**

Последовательность3

**end**

**default**: **begin**

Последовательность по умолчанию

**end**

**endcase**

Анализируется *величина*, если она равна *значению1*, то выполняется *последовательность1*, если она равна *значению2*, то выполняется *последовательность2*, если она равна *значению3*, то выполняется *последовательность3*. Если же она не равна ни одному с вышеупомянутых значений, то выполняется *последовательность по умолчанию*.

Оператор циклирования *repeat*. Он выделен в отдельно от других операторов циклирования, так как его аналоги не сильно распространены в других языках.

**repeat** (выражение) **begin**

Последовательность

**end**

*Последовательность* выполняется такое количество раз, какое получилось в результате *выражения*.

Остальные операции циклирования выделены в отдельную группу, так как хоть они и принадлежат множеству языка Verilog, но не есть синтезируемыми, иными словами из них нельзя воплотить в аппаратном обеспечении, или они воплощаются, но с существенными ограничениями, следовательно бесполезны в контексте данного рассмотрения языка Verilog.

Что бы получить исчерпывающую информацию про синтаксис языка Verilog, можно рассмотреть его БНФ, описанную в приложении A стандарта IEEE 1364.

Язык Verilog можно разбить на три множества:

* синтезируемое;
* игнорируемое при синтезе;
* не синтезируемое.

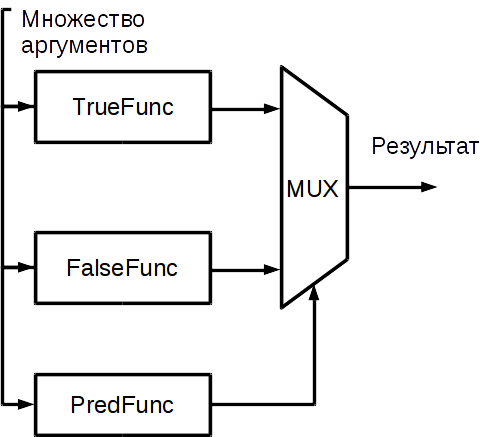
Код написанный исключительно с синтезируемым и игнорируемым множествами гарантировано будет отображен на ПЛИС, на ASIC, не фотошаблоны. Необходимо вкратце упомянуть хотя бы основные ограничения на языка в отношении синтезируемости, в первую очередь, управляющих структур. Для каждого синтезатора эти данные могут быть разными, но в основном они совпадают. Циклы *forever* и *while* не синтезируемы вообще. Цикл *repeat* синтезируем только в том случае, если выражение, которое вычисляет количество итераций цикла, постоянно. Цикл *for* синтезируем только в том случае, если все присвоения для итератора (индекса) цикла константны. Следует помнить, что синтезированная схема не имеет ни времени начала, ни времени конца работы, она выполняется постоянно. Поэтому игнорируются операции, которые пользователь задает для выполнения в начале симуляции. Начальных значений регистров не существует. Так же игнорируются временные задержки выполнения, которые пользователь может задавать для выражений. Решение этих «неудобств» не происходит в рамках языка. Для таких случаев есть сигнал сброса, о подачи которого и о реакции на который должен позаботится разработчик. Именно сигнал сброса может быть условным началом работы схемы. Подавая тактовый сигнал, разработчик задает равномерно течение дискретного времени. Большинство цифровых электронных устройств работают именно в дискретном времени, задаваемом тактовым сигнал, хоть это и не есть их особенностью

## Композиции в языке Verilog

В синтаксисе языка Verilog скрыт набор композиций. Вскроем семантику синтаксических конструкций, которые представляют олицетворяют композиции. Структура *if* в общем виде может быть записана так.

Но вместе с этим может быть упрощена частичным применением к пустой функции.

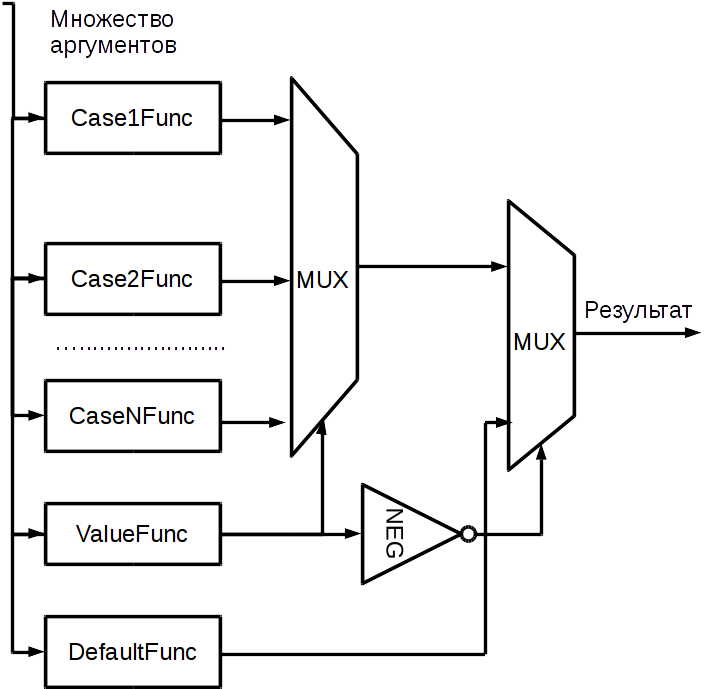
С точки зрения схемотехники, она подобна мультиплексору.

Рис. 3. Схематическое представление композиции if.

Одновременно вычисляются предикат, случай для истинного предиката и случай для ложного предиката. Результат предиката (истина или ложь, 1 или 0) передается на адресный вход мультиплексора и, в зависимости от его истинности, он подает на выход данные из первого или второго его входов. Операция ветвления очень важна для обеспечения формальной системы полнотой по Тьюрингу.

Следующей конструкцией есть конструкция *case*. Она подобна операции ветвления за исключением того, что, ветвеление происходит на больше чем два исхода. Записать его можно так:

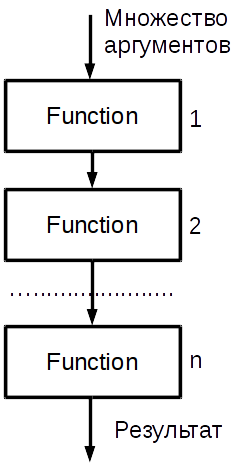
Схемотехнически эта конструкция может быть изображена следующим образом.

Рис. 4. Схематическое представление композиции case.

Эта композиция по своей семантике есть обобщенным вариантом композиции ветвления. Вычисляются все возможные исходы конструкции case, в том числе исход по умолчанию, а так же величина, которая определяет исход. Все исходы подаются на мультиплексор и выбирается тот, который, соответствует посчитанной величине в функции ValueFunc. На случай, если исхода, соответствующего величине нет, результатом становится результат работы DefaultFunc. Устройство NEG как раз и проводит анализ того, задан ли исход для данной величины или нет, и если не задан, то коммутирует на выход выходного мультиплексора значение DefaultFunc. Такое устройство очень легко организовать на комбинационной логике, но это не предмет рассмотрения сейчас.

Конструкция repeat позволяет применить одну функцию конечное количество раз к аргументам.

Количество раз должно быть константным и известным уже на этапе синтеза.

Рис. 5. Схематическое представление композиции repeat.

Эта композиция особенна тем, что принимает на вход лишь один аргумент-функцию и константу. Т.е. функция композиционируется сама с собой. Следует отметить, что такого ограничения при моделировании нет.

Следующая конструкция – конструкция *for*. Она подобна другим конструкциям for, которые встречается в языках программирования, но синтезируемость тоже накладывает свои ограничения. В общем случае для всех операций циклирования на языке Verilog должно выполнятся одно условие: синтезатор должен знать когда они закончатся.

С учетом ограничений, накладываемых на for, его ее можно схематически изобразить точно так же как и *repeat*, в силу того, что инициализирующая функция *InitFunc*, функция условия выхода из цикла *CondFunc*, функция, выполняющаяся после итерации цикла *PostFunc*, функция тела цикла *BodyFunc* после синтеза не должны никаким образом влиять на количество итераций, выполняемых циклом.

Конструкция while имеет те же недостатки, что и конструкция for.

Ни само тело цикла *BodyFunc*, ни условие *CondFunc*, не должны влиять на количество итераций цикла while, таким образом конструкция также вырождается в *repeat*.

Хоть в Verilog управляющих структур немало, из них удалось выделить только две основных композиции: ветвления и повторения применения функции n раз, где n – константа, получаемая на этапе синтеза.

Также неявно языке Verilog присутствует суперпозиция в двух представлениях. Первое – суперпозиция с использованием понятия «модуль». Например:

**module** topmodule(IN0, IN1, OUT);

**input wire** [7:0] IN0;

**input wire** [7:0] IN1;

**output wire** [7:0] OUT;

**wire** [7:0] w1, w2,

module1 m1(IN0, IN1, w1);

module2 m2 (IN0, w1, w2);

module3 m3 (IN0, w2, OUT);

**endmodule**

Модуль в языке Verilog можно представить как функцию. В коде выше в в составе основного модуля *topmodule*, существует три модуля *m1*, *m2*, *m3*. Последний порт в каждом из модулей – выходной, а предыдущие – входные. Проводники *w1* и *w2*, позволяют передать выход одного модуля передать на вход другого. Таким образом это явление можно рассмотреть как суперпозицию.

Суперпозицией можно назвать и последовательную запись команд в блоках *begin-end*, например:

**module** (a, b, c);

**input wire** [7:0] a;

**output reg** [7:0] b;

**input wire** [7:0] c;

**reg** [7:0] d;

**always** @(**posedge** CLK) **begin**

d = a + 5;

b = d \* c;

**end**

**endmodule**

Последовательные операции сложения и умножения могут быть рассмотрены как как функции. Результат операции сложения «+» дальше используется в операции умножения «\*». Таким образом это можно рассматривать как неявное представление композиции суперпозиции.

Вместе с этим группа названных композиций не полна по Тьюрингу, так как на них можно построить только комбинационные схемы. Последовательностные схемы таким образом строить невозможно, так как для них необходимо вводить понятие времени и состояний, чего не замечено в вышеназванных представителях композиций синтезируемого подмножества языка Verilog.

В самом деле задача получения полной по Тьюрингу системы лежит частично на плечах разработчика, который должен использовать еще одно неявное проявления композиций. Для полноты по Тьюрингу недостаточно еще механизма циклирования, в котором условие выхода определялось бы динамически, во время компиляции, а не синтеза. Такую схему можно получить на языке Verilog, с оговоркой, что разработчик гарантирует поступление тактового сигнала на вход схемы для обеспечения течения дискретного времени. Вот, например, реализация «полнофункциональной» композиции «while».

**module** topmodule(CLK, ST, IN, OUT);

**input wire** [7\*3:0] IN;

**input wire** CLK, ST;

**output reg** [7\*3:0] OUT;

**wire** [7:0] mb\_result;

**wire** mc\_result;

**reg** [7\*3:0] MI, MCI;

**wire** [7\*3:0] MR;

module\_condition mc(MCI, mc\_result);

module\_body mb(MI, MR);

**always** @(**posedge** CLK) **begin**

**if** (ST == 1) **begin**

MI <= IN;

MCI <= IN;

**end** **else** **begin**

**if** (mc\_result == 1) **begin**

MI <= MR;

MCI <= MR;

**end** **else** **begin**

OUT <= MI;

**end**

**end**

**end**

**endmodule**

Это композиция двух функций, выраженных в модулями mc (функция условия) и mb (модуль тела цикла). Для начала вычислений используется сигнал ST, по которому на следующий тактовый импульс CLK, входные аргументы передаются модулю условия (регистр MCI) и модулю тела цикла (регистр MI). Пока выход модуля условия, mc\_result, равен 1, тело цикла принимает свой выход MR на вход MI. После того, как условие прекращает выполнятся, результат их регистра MI выдается на вход главного модуля OUT.

Использование композиционного подхода к конструированию с таким образом скрытыми композициями затруднительно, хоть и возможно.

## Использование композиционного подхода при конструировании вычислительных систем на языке Verilog

Освещенные выше проблемы можно решить создав прослойку некоторых базовых композиций, созданных средствами самого языка. Добавляется новый композиционный уровень абстракции. В ходе исследований был создан примитивный базовый набор композиций, состоящий из композиций, предложенных Алонзо Чёрчем, называемый алгеброй Чёрча и состоящий из: примитивной рекурсии, минимизации и суперпозиции, а также небольшого набора вычислимых операций: генерация нуля, выбор необходимого аргумента из n аргументов, операции следования (инкремент аргумента на единицу). Все операции производятся над множеством натуральных чисел (включая ноль), а все функции имеют одну арность. Естественно такой подход не оптимальный. Функция, которая проводит операцию над одним аргументом, должна принимать все остальные даже тогда, когда она их не использует.

Кроме того, был разработан простейший синтаксис верхнего уровня, позволяющий абстрагироваться от понятий, специфических для языка Verilog: узлы, регистры, события, а также его компилятор в Verilog. Результатом работы такого компилятора будет Verilog-модуль, который производит необходимые вычисления.

Как раньше упоминалось, программа – суть функция, а суть процесса разработки – построить необходимую функцию, посредством порождения и применения композиций. В Verilog функции можно представить в виде модулей, соответственно, комбинируя ограниченный набор модулей, можно получить решения задачи таким же образом, как и комбинируя базовый набор функций.

Если комбинирование комбинационных функций и композиций проблем не вызывает, то комбинирование последовательностных функций и композиций – задача на порядок сложнее, так само понятие последовательности чуждо функциональному подходу к разработке. Соответственно возникает задача незаметного перехода от функциональной семантики к последовательностному синтаксису.

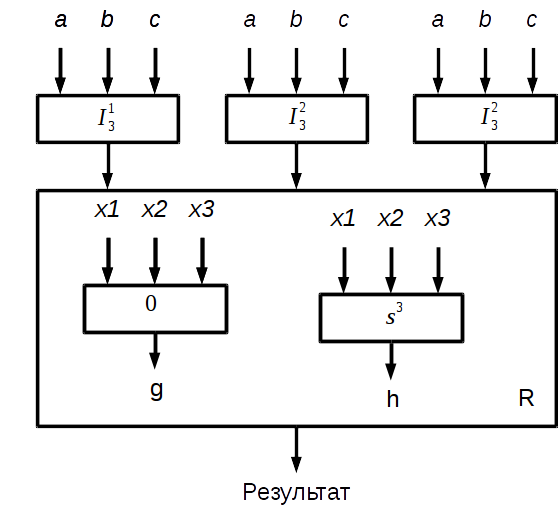
Решение в терминах языка Verilog можно изобразить в виде иерархического дерева модулей. В качестве примера можно рассмотреть дерево, соответствующее операции сложения. Конечно, операцию сложения на языке Verilog реализовать крайне просто, но задача стоит в использовании ограниченного набора вычислимых операций и композиций, где операции сложения нет. Естественно, он не претендует на полезность в реальных применениях. Само выражение с использованием композиционного подхода с n-арными функциями можно записать следующим образом:

R – композиция примитивной рекурсии. В квадратных скобках изложены операции, к которым применяется данная композиция: 0 – генератор нуля, и операция следования. Обычно для функции генерации нуля не нужно каких либо аргументов, но так как в алгебре Чёрча все без исключения функции n‑арные, необходимо было добавить три фиктивных аргумента, которые не влияют на ход вычислений, именно такое определение фиктивного аргумента будет подразумеваться далее при использовании этого термина. Такая же ситуация с оператором следования. В верхнем индексе показано, какой из аргументов он будет увеличивать на единицу, остальные аргументы – фиктивные. В круглых скобках указаны аргументы операций и результирующих функций, полученных в результате применения композиций. Операция – операция выбора, возвращает m-й аргумент из n ей переданных. *a* и *b* – слагаемые, а *c* – фиктивный аргумент .

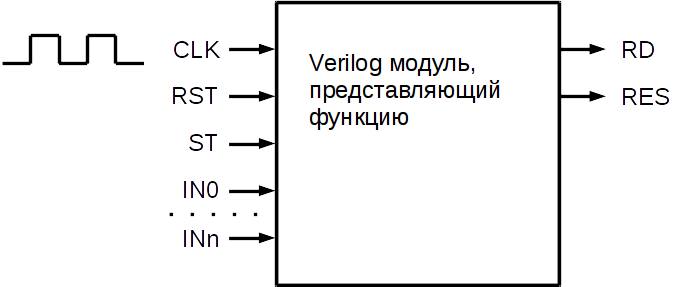
Следует детальней рассмотреть композицию примитивной рекурсии. Примитивная рекурсия сочетает две функции: функцию-инициализатор *g* и функцию, которая вычисляется на каждой итерации, *h*. Вычисления просходят следующим образом:

,

где – результат применения примитивной рекурсии за m+1 шагов.

Рис. 6. Схематическое представление иерархии модулей при операции сложения.

На рис. 6 схематически изображена иерархия функций, которые можно представит, как иерархию модулей, которая позволяет вычислить сумму a и b. Каждый из таких модулей вычисляется разное количество времени и перед разработчиком стоит задача синхронизировать работу модулей, и, где это возможно, выполнить их параллельно для экономии времени. Для этого каждый модуль, кроме входов и выходов должен иметь универсальный интерфейс синхронизации для сочетания с другими модулями. Не важно проходят вычисление мгновенно, как в комбинационной схеме, либо для них необходимо некоторое количество времени. Вот таким образом должен выглядеть модуль с интерфейсом синхронизации (рис. 7).

Рис. 7. Verilog модуль функции с интерфейсом синхронизации.

А так он выглядит в коде Verilog:

**module** node\_*modname*(RST, ST, CLK, RD, RES, *IN0, ... INn*);

**input** **wire** RST;

**input** **wire** ST;

**input** **wire** CLK;

**output** **reg** RD;

**output** **reg** [*buswidth*:0] RES;

***input******wire*** *IN1;*

*...............*

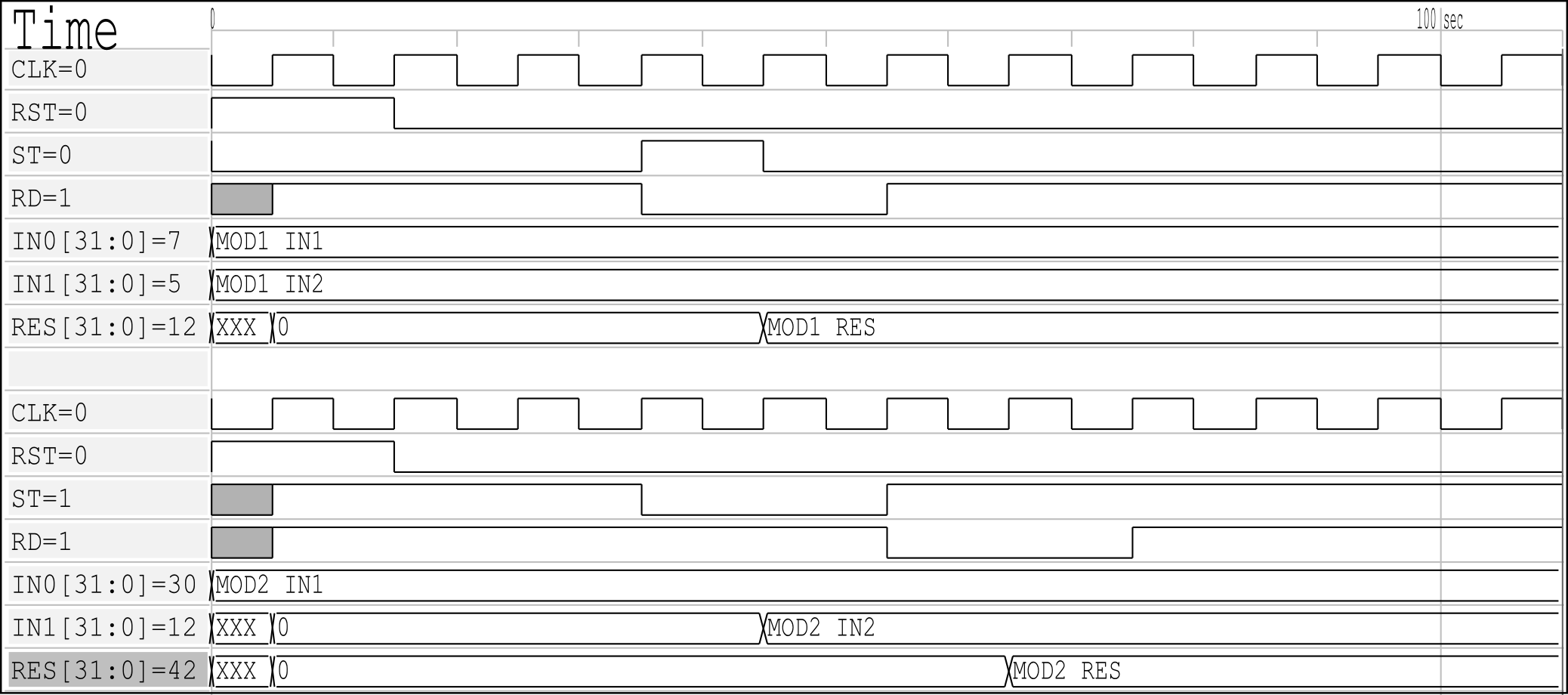
***input******wire*** *INn;*

*module body*

**endmodule**

RST – сигнал сброса. Подается однократно при запуске либо сбросе всей схемы. ST– сигнал начала вычислений, генерируется при поступлении новых входных данных. CLK – тактовый сигнал, который подается всё время работы схемы. RD – сигнал результата, устанавливается когда на выходе появляется корректный результат. RES – собственно результат. *IN1* … *INn* – входные аргументы. *Buswidth* – битность работы схемы, сколько бит будет занимать каждый из аргументов и результат. *Modname* – имя модуля, его идентификатор.

Теперь рассмотрим работу модулей в динамике.

Рис. 8. Временные диаграммы работы двух модулей (разделены пустой строкой).

На временной диаграмме (рис. 8) изображен фрагмент вычислительного процесса, где один модуль получает данные от другого модуля по завершению вычислений оного. В начале работы схемы происходит одновременный сброс всех модулей схемы, на следующем шаге выставляются входах INi действительные значения входных аргументов, после этого для первого модуля подается сигнал ST, который запускает вычисления в нем, после происходят вычисления, по завершению вычислений, результат попадает на порт RES, а один период тактового сигнала спустя, выставляется сигнал RD (от ready), который оповещает, об успешном окончании вычислений в модуле и, одновременно с этим, есть сигналом ST для второго модуля.

qЕще одной задачей есть построение дерева из выражения, описывающего семантику необходимой функции (программы). Сначала рассмотрим синтаксис таких выражений. Композиция представляется следующим образом:

*Compname* – название композиции, принимающей аргументами m функций, *F1* ... *Fm* – функции-аргументы композиции. *Arg1* … *argn* – аргументы результирующей функции, полученной в результате применения композиции. Функции-аргументы и аргументы результирующей функции разделены между собой точкой с запятой. А между собой они разделяются запятыми.

Кроме композиций есть обычные функции, которые принимают на вход числа, а не функции. Записываются они следующим образом:

Они отличаются только тем, что функций-аргументов там нет.

Из таких выражений необходимо построить дерево. Необходимо понять, какие параметры должны быть в каждом узле дерева. Узел дерева в компиляторе представляется в формате JSON (JavaScript Object Notation).

{

"name":function\_name,

"id":identificator,

"arguments":[],

"static":[]

}

*name* – название узла, *id* – числовой идентификатор из 6 цифер, *arguments* – упорядоченный массив аргументов, *static* – в случае, если узел – композиция, в нем содержатся функции-аргументы, в противном случае массив пустой.

Вот так, например, представляется выражение f = x + y\*z.

{

'id': '199180',

'static': [],

'name': 'add',

'arguments': [

'x',

{

'id': '357004',

'static': [],

'name': 'mul',

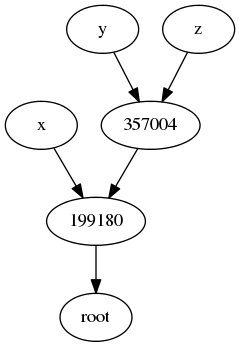
'arguments': ['y', 'z']

}

]

}

Таким образом выглядит графическое представление этого же выражения (рис. 9).

Рис. 9. Графическое представления дерева выражения f = x + y\*z.

После этого выражение раскладывается на примитивные составляющие из базового набора функций и композиций, которые имеют реализацию на языке Verilog. Он выглядит как набор шаблонов, в которые можно подставлять ширину шин данных, модули, которые комбинируют композиции композиции, необходимое количество входов и т. д. Кроме того существуют так называемые  
 «корневые модули». Они предназначены для того, что бы обернуть конечную иерархию модулей, в один модуль для удобства дальнейшего использования. Таким образом в конструкции обязательно должен присутствовать главный корневой модуль, в котором заключена вся иерархия модулей. Так же корневые модули присутствуют в качестве функций-аргументов композиций. Таким образом каждое наименование в поле «static» структуры – корневой модуль.

Вот таким образом выглядит шаблон корневого модуля:

**module** root*%ROOT\_NODE\_ID%*(RST, ST, CLK, RD, RES, *%IN%*);

**input** **wire** RST;

**input** **wire** ST;

**input** **wire** CLK;

*%INPUT\_DEFINITIONS%*

**output** **wire** RD;

**output** **wire** [*%BUS\_WIDTH%*-1:0] RES;

*%WIRES%*

*%ASSIGNMENTS%*

*%MODULES%*

**endmodule**

В *ROOT\_NODE\_ID* содержится идентификатор модуля, в поле *IN* перечислены через запятую входные сигналы модуля, а в *INPUT\_DEFINITIONS* – их определения. В *WIRES* определены проводники, которые используются для соединений между модулями. В *ASSIGN* назначаются сигналы синхронизации корневого модуля с внешним миром, порт результата, а также стартовые сигналы для разных модулей. Стоит отметить, что если несколько разных модулей используются в качестве источников выходных данных для другого модуля, то модуль начинает свою работу только после того, как все источники данных закончат свои вычисления. В поле *MODULES* происходит включения всех модулей дерева. В *BUS\_WIDTH* указывается ширина шины данных, т. е. Разрядность данных, которыми оперируем.

Теперь рассмотрим шаблон модуля композиции примитивной рекурсии, как наиболее сложного из набора примитивов:

**module** composition\_r(RST, ST, CLK, RD, RES, %IN\_LIST%);

**input** **wire** RST;

**input** **wire** ST;

**input** **wire** CLK;

**output** **reg** RD;

**output** **reg** [%BUS\_WIDTH%-1:0] RES;

**reg** [%BUS\_WIDTH%-1:0] BUF; //Вх сгн для мод тела рек.

**wire** [%BUS\_WIDTH%-1:0] res\_g; //Результ. модуля иниц.

**wire** [%BUS\_WIDTH%-1:0] res\_h; //Рез. мод. тела рек.

**reg** [%BUS\_WIDTH%-1:0] CNT; //Счетчие итераций

**reg** [%BUS\_WIDTH%-1:0] ITERATIONS; //Рег. кол-ва итер

**wire** rd\_g; //Сигнал готовности модуля инициализации

**wire** rd\_h; //Сигнал готовности модуля тела рекурсии

**reg** st\_h;

**reg** st\_h\_f;

**reg** STold;

**reg** rd\_g\_old; //Сигн готов. Мод. Иниц. на пред такте

**reg** rd\_h\_old; //Сигн готов. тела рек. на пред такте

%IN\_DEF%//Определения входных аргкментов

**wire** [%BUS\_WIDTH%-1:0] %ADDITIONAL\_IN%;

//Включение аргументов-функций примитивной рекурсии.

%G\_CLASS% g(RST, ST, CLK, rd\_g, res\_g, %IN\_LIST%);

%H\_CLASS% h(RST, st\_h, CLK, rd\_h, res\_h, %IN\_LIST%, %ADDITIONAL\_IN%);

//Тут определен последний вход модуля тела рекурсии

//который есть либо выходом модуля рек либо иниц.

**assign** %ADDITIONAL\_IN%[%BUS\_WIDTH%-1:0] = (CNT[%BUS\_WIDTH%-1:0] > 1)?res\_h[%BUS\_WIDTH%-1:0]:res\_g[%BUS\_WIDTH%-1:0];

**always** @(**posedge** CLK) **begin**

**if**(RST == 1) **begin** //Обнуление регистров

RD = 1; //когда происходит сброс

CNT = 0;

STold = 0;

rd\_g\_old = 0;

rd\_h\_old = 0;

st\_h\_f = 0;

st\_h = 0;

**end** **else** **begin**

**if**(ITERATIONS == CNT && rd\_g == 1) **begin**

RD = 1; //Когда выполнились все

CNT = 0; //итерации прим. рекурсии

st\_h = 0; //Либо закончил работу модуль

st\_h\_f = 0; //инициализации

**if**(ITERATIONS == 0) **begin**

RES = res\_g; //Закончил раб. Мод. иниц

**end** **else** **begin**

RES = BUF; //Выполнились все итерации

**end**

**end**

**if**(RD == 0) **begin**

**if**(st\_h\_f == 0) **begin**

st\_h = 0;

**end**

**if**(st\_h == 1) **begin**

st\_h\_f = 0;

**end**

**if**(rd\_g == 1 && rd\_g\_old == 0 && CNT == 0) **begin** //Пришел фронт готовности модуля инициализации

BUF = res\_g;//Сохранить результат.

st\_h\_f = 1;

st\_h = 1;

**end**

**if**(rd\_h == 1 && rd\_h\_old == 0) **begin**

BUF = res\_h; //Пришел фронт сиг. гот.

CNT = CNT + 1;//модуля тела рекурсии

st\_h\_f = 1; //Сохранить результат

st\_h = 1; //в регистр BUF

**end**

**end**

**if**(ST == 1 && STold == 0) **begin**

RD = 0;//Пришел фронт сигнала начала выч.

**if**(%MAX\_IN% > 0) **begin** //MAX\_IN = послед.

ITERATIONS = %MAX\_IN% + 1;//вход мод.

**end** **else** **begin** //где хранится кол-во итер.

ITERATIONS = 0;//рекурсии

**end** //Которое записывается в

**end** //Iterations с приходом

**end** //Стартового фронта

**if**(ST == 1) **begin** //Сохранениестарого значения

STold = 1; //сигнала начала вычислений

**end** **else** **begin** //Что бы обеспечить срабат.

STold = 0; //по фронту

**end**

rd\_g\_old = rd\_g; //Сохранение старых значений

rd\_h\_old = rd\_h; //сигналов rd\_g и rd\_h

**end** //(для срабатывания по фронту)

**endmodule**

## Возможные улучшения подходах

После некоторых исследований решено перейти на более гибкий синтаксис, который бы позволил легко расширять язык по мере необходимости, хоть у него есть недостаток – чрезмерная простота. В связи с эти был выбран синтаксис языка LISP. Выбрана следующая форма для записи композиций:

(COMP\_NAME (FUNC1 FUNC2...FUNCM) (ARG1 ARG2...ARGN))

где *COMP\_NAME* – название композиции, FUNC1, FUNC2...FUNCM – функции-аргументы композиции, ARG1, ARG2...ARGN – аргументы результирующей функции, полученной в результате применения композиции.

Для записи функций используется следующий синтаксис:

(FUNC\_NAME (ARG1, ARG2, … ARGN))

где FUNC\_NAME – название функции, ARG1, ARG2, … ARGN – аргументы функции. Это со стороны синтаксиса.

Есть смысл отказаться от подхода с n-арными функциями, хоть реализация такого подхода существенно проста, но на плечи разработчика ложится задача работы с функциями фиксированной арности, что вынуждает создавать функции с фиктивными входами, переупорядочивать аргументы собственными средствами. Например, дана функция

(F1 (X, Y, Z, A, B)) = (MUL ((ADD (X Y Z)) (DIV (A B)))),

которая эквивалентна (X+Y+Z) \* (A/B). И дан кортеж обратно упорядоченный кортеж аргументов к ней <B, A, Z, Y, X>. Для n-арных функций тут есть много работы по упорядочиванию аргументов в правильный порядок для применения функции. Но если бы каждый аргумент был бы именованным, такого бы не происходило. Также такой подход удобен при применении композиций. Например дана композиция:

(COMP\_NAME ((FUNC1 (A B)) (FUNC2 (C D))) (D, A, B, C))

С именованными аргументами не важен порядок их передачи в результирующую функцию после применения композиции COMP\_NAME к функциям FUNC1 и FUNC2. Хоть это осложняет математические основы работы с функциями и композициями, но упрощает их порождение и применение.

В связи с этим немного изменилось JSON представление узлов дерева, представляющего решение задачи:

{"name":"название функции/композиции",

"id":шестизнаковый\_идентификатор,

"static":[массив функций-арг композиции (если комп)],

"arguments":[

{"value":аргумент0,

"no":0},

{"value":аргумент1,

"no":1},

........................

{"valueN":аргументN,

"no":N},

]}

Также подобраны более эффективный базовый набор композиций и операций, хоть уже не минимальный, определенный на множестве натуральныъ чисел включительно с нулем:

* if – композиция ветвления. Принимает на вход три функции: предикат, функция, которая вычисляется при истинном предикате и функция, которая вычисляется при ложном предикате;
* for – композиция циклирования, подобна циклам for в языка программирования, с некоторым отличием. Для каждого аргумента композиции выполняется своя функция-аргумент для того, что бы получить его новое значение на следующей итерации. Существует также предикат, по которому цикл останавливается, результатом работы цикла есть значение первого аргумента;
* eq – оператор равенства. При равных его операндах, результат применения оператора – единица, а при их неравенстве – ноль.
* add – арифметическое сложение;
* zero – генерация нуля;
* one – генерация единицы;
* nop – операция-пустышка. Принимает единственный аргумент и его же возвращает;
* mul – произведение двух чисел;
* div – целочисленное частное двух числ в случае если делимое меньше делителя, то результат операции ноль.
* sub – разница двух чисел. Если число из которого вычитают, меньше вычитаемого, то результат операции – ноль.
* const – генерация численной константы.

В принципе достаточно для полноты по Тьюрингу операций eq, add, zero, one, if, for. Все остальные можно вывести из них. Для создания новой функции используют ключевое слово def. Тогда, что бы задать функцию, необходимо сделать следующую запись:

(def FUNC\_NAME FUNC\_DEF (ARG\_LIST))

где FUNC\_NAME – название функции, FUNC\_DEF – оперделение функции в терминах известных компилятору, ARG\_LIST – список аргументов функции разделенных пробелами. Вот несколько примеров выведенных фукций их минимального базиса ( eq, add, zero, one, if, for ).

*Инкремент A+1*

(def INC (ADD ((ONE) A)) (A))

*Вычитание A-B*

(def SUB

(FOR

((IF ((ZERO) (INC (IN2)) (EQ (IN0 (INC (IN1))))) (IN2 IN3 IN0))

(NOP (IN1))

(NOP (IN2))

(INC (IN3))

(INC (IN4))

(ADD ((EQ (IN2 IN3)) (EQ (IN1 IN4)))))

((ZERO) A B A B)

)

(A B)

)

*Декремент B-1*

(def DEC (SUB (B (ONE))) (B))

*Произведение D\*E*

(def MUL

(FOR

((ADD (IN0 IN1)) (NOP (IN1)) (DEC (IN2)) (EQ (IN2 (ZERO))))

((ZERO) D E))

(D E)

)

*Строго больше A > B*

(def GT

(IF

((ONE) (ZERO) (SUB (IN0 IN1)))

(A B)

)

(A B)

)

*Строго меньше H < I*

(def LT

(IF

((ZERO) (ONE) (SUB (IN0 IN1)))

(H I)

)

(H I))

*Частное F/G*

(def DIV

(FOR (

(IF (

(NOP (IN0))

(INC (IN0))

(GT (IN0 (DEC (IN1))))

)

(IN0 IN1)

)

(SUB (IN1 IN2))

(NOP (IN2))

(EQ (IN1 (ZERO)))

)

((ZERO) F G)

)

(F G)

)

*Вычисление остатка от деления F%G*

(def MOD

(FOR ((SUB (IN0 IN1)) (NOP (IN1)) (LT (IN0 IN1)))

(F G)

)

(F G)

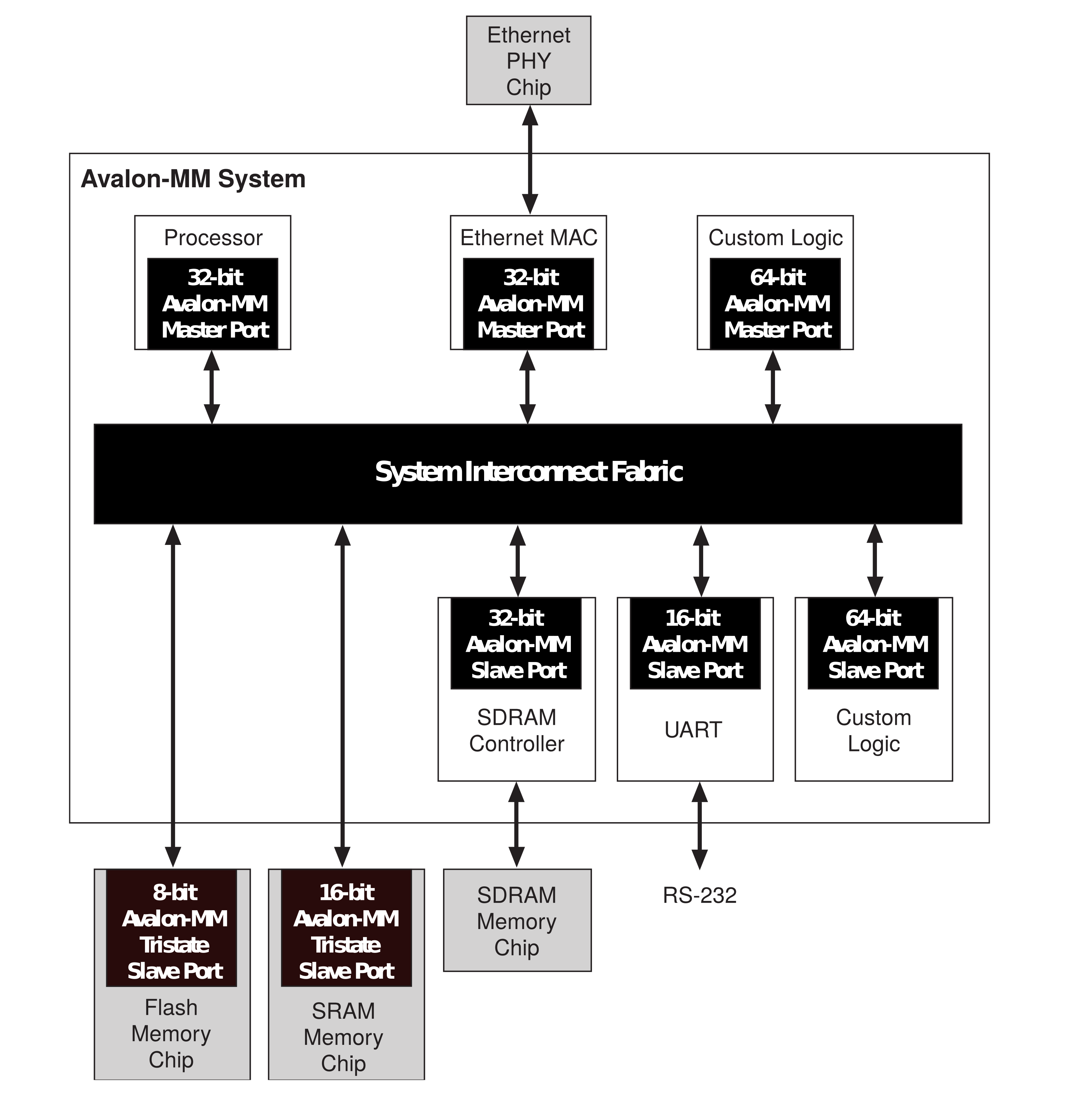
)

При проектировании для реальных применений, следует обращать внимание на возможности реальных платформ. Таким образом реализовывать операцию вычитания категорически не оптимально, так как она реализовывется намного лучшим образом, например, синтезатором языка Verilog. Иногда на чипах даже существуют специальные блоки для вычисления часто используемых математических операций: умножение, взятие хеш-функции, шифрование AES-256.

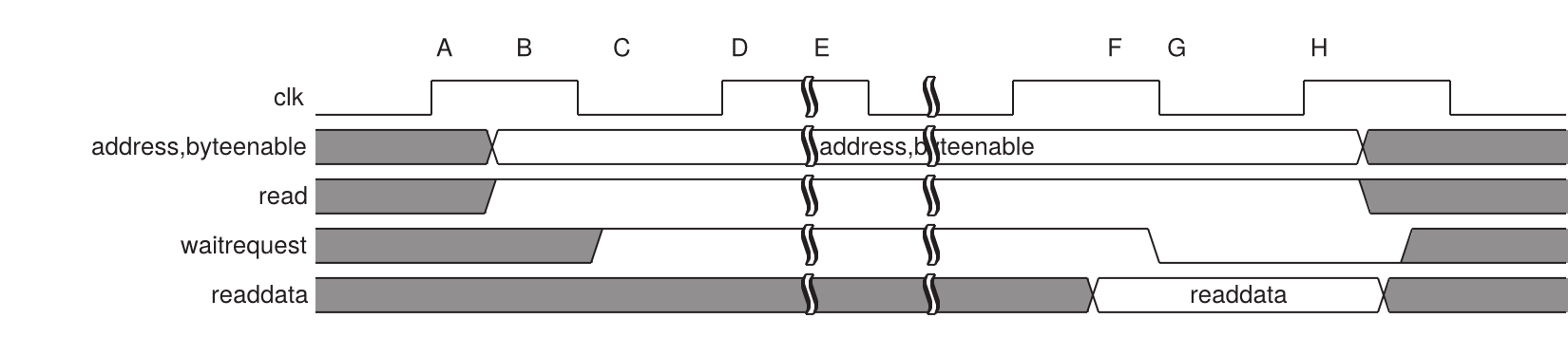
## Улучшение взаимодействия устройств, разработанных композиционным методом проектирования с другими устройствами

Композиционный метод проектирования позволяет решить немало задач, но в реальном мире намного больше задач там где его применение только осложняет задачу и может не дать нужного выигрыша в производительности и борьбе со сложностью. В связи с этим необходимо осознавать то, что разработкам с применением композиционного подхода к проектированию придется взаимодействовать с традиционными вычислительными средствами, в том числе с центральными процессорами. Самым оптимальным способом взаимодействия с ЦП есть шина, которая позволяет всем устройствам, размещенным на ней, взаимодействовать между собой через единственный интерфейс для каждого устройства. В памяти процессора такие устройства отображаются как области памяти.

Популярные шины (Wishbone, Avalon, Amba и др.) приблизительно эквивалентны по своему устройству. Рассмотрим устройства шины, для которой производились опытные работы: Avalon (Memory Mapped устройства) (прим. На рис. 10). Ядром шины есть система межсоединений (bus fabric), которая включает в себя средства коммутации и логику коммутации устройств на шине между собой, в том числе так называемые «арбитры», которые позволяют определять очередность доступа устройств к шине, так как одновременно на шине может выставлять свою информацию только одно устройство. Кроме этого на шине находится два вида интерфейсов и, соответственно им, два вида устройств: master и slave. Master отличает от slave тем, что он может инициировать транзакции на шине и обращаться к slave'ам. Slave может лишь генерировать ответ для master'а и инициировать транзакции может лишь опосредованно: через прерывания, которые сами по себе к шине не относятся.

Рис. 10. Пример шины Avalon с memory mapped устройствами.

Рассмотрим каким образом происходят транзакции на шине Avalon. Допустим master отправляет запрос на чтение регистра устройства (рис. 11). Для этого после прихода нарастающего фронта тактового импульса (A), master устанавливает действительные сигналы адреса (address), чтения (read), действительных байт (byteenable) (B). Byteenable определяет то, какие байты master хочет прочесть. Дело в том, что иногда для ускорения транзакций не читается все слово (если ширина шины кратна степени двойки), а лишь его часть. Шина устанавливает запрос на ожидание (waitrequest) (С ). Пока он выставлен, master не снимает данные шины, которые он установил, таким образом можно обрабатывать случаи, когда на шине находится много master устройств, которые одновременно присылают запросы на транзакции. По нарастающему фронту тактового сигнала (D) master принимает запрос waitrequest и ожидает (E). На шине выставляются прочитанные данные readdata (F) и со спадающим фронтом тактового сигнала (G) сигнал waiterequest переводится в ноль. По следующему фронту тактового сигнала (H) master читает данные readdata и цикл чтения для master завершается.

Рис. 11. Процесс считывания данных master'ом

Для slave'а процесс чтения выглядит другим образом (рис. 12). После прихода тактового импульса (A) шина устанавливает корректный адрес и сигнал read равным единице (B), далее она декодирует адрес определяя какому устройству он отвечает и устанавливает сигнал выбора устройства chipselect для нужного slave (C ). Перед следующим нарастающим тактовым импульсом (D) шина устанавливает сигнал waitrequest в единицу и slave ничего не выставляет на шину до его окончания. Пока установлен waitrequest, slave в произвольный момент времени выставляет в readdata корректные данные (G) и продолжает ожидать пока waitrequest не будет снят со спадающим фронтом тактового импульса (H). Со следующим тактовым сигналом завершается цикл чтения из slave.

Рассмотрим как происходит запись данных master'ом. (рис. 13) После наростающего фронта тактового сигнала clk (A), master выставляет на шине корректные значения адреса, даных для записи (writedata) и сигнал write в единицу. С нарастающим фронтом тактового сигнала, шина выставляет сигнал waitrequest в единицу и до его снятия (E) master держит выставленные данные неизменными. На следующем нарастающем фронте тактового сигнала (F) master обнаруживает, что waitrequest снят и он может убрать данные с шины. Цикл записи завершился.

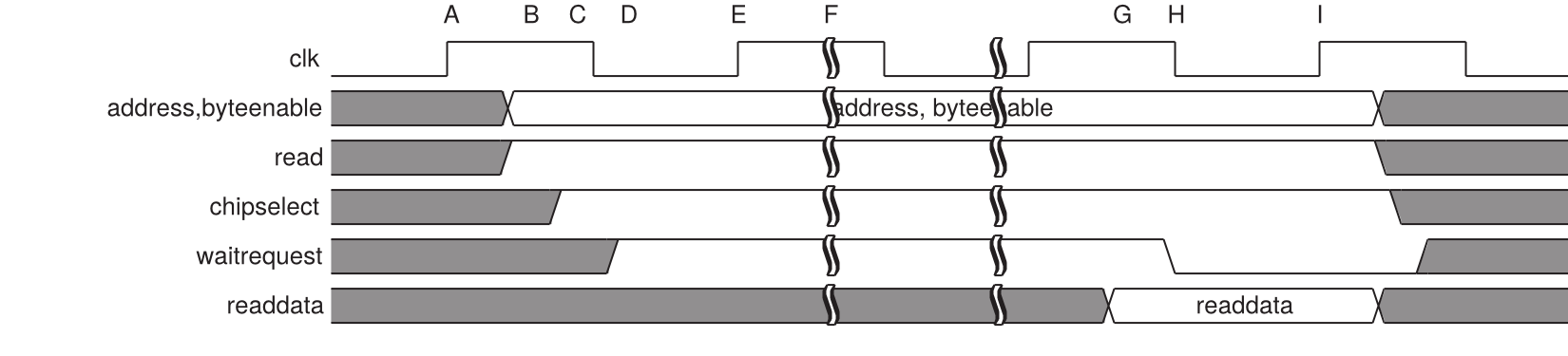
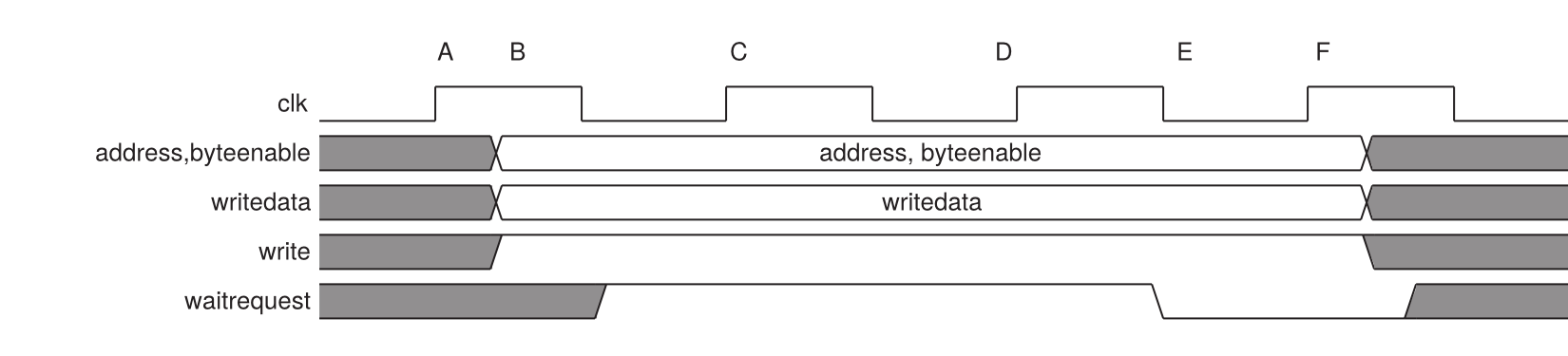
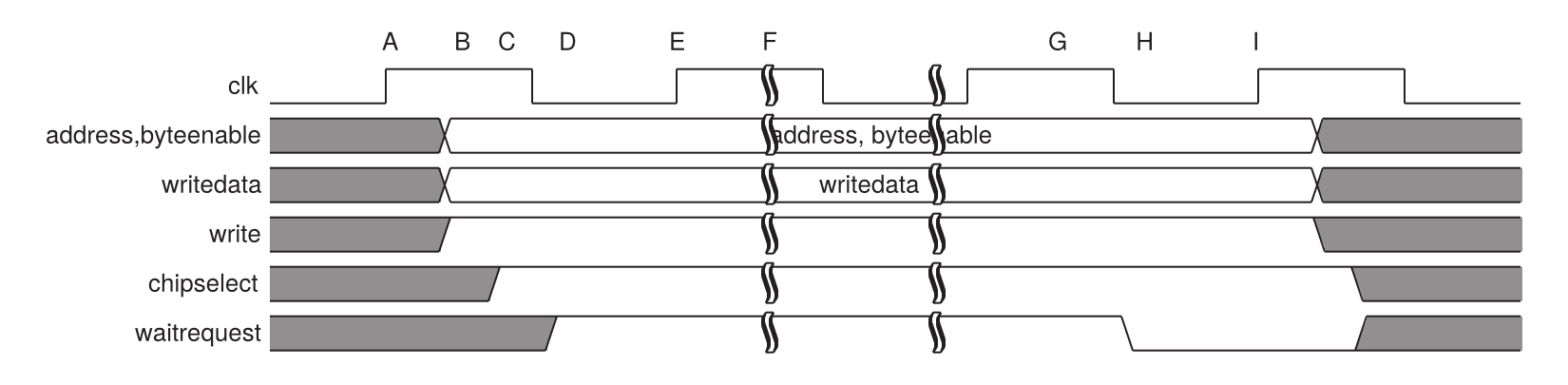
Рис. 12. Считывание данных из slave на шине Avalon.

Рис. 13. Запись данных master'ом на шине Avalon.

Запись же в slave происходит следующим образом (рис. 14). После нарастающего фронта тактового сигнала (A), шина устанавливает корректный адреса (address), данные для записи (writedata), байты слова, которые будут писаться (byteenable), а также сигнал записи write равным единице (B).

Рис. 14. Процесс записи в slave-устройство на шине Avalon.

Далее шин дешифрует адрес устройства и для необходимого устройства выставляет сигнал chipselect (C ). Перед следующим нарастающим фронтом тактового сигнала (E), шина выставляет сигнал waitrequest в единиц (D), после которого на протяжении неопределенного времени (F) сигналы на шине остаются неизменными. В какой-то момент (G) на протяжении ожидания, slave записывает в свой регистр данные writedata. После этого сигнал waitrequest снимается (H), и на следующем положительном фронте (I) тактового сигнала это замечает slave-устройство, после этого цикл записи завершен.

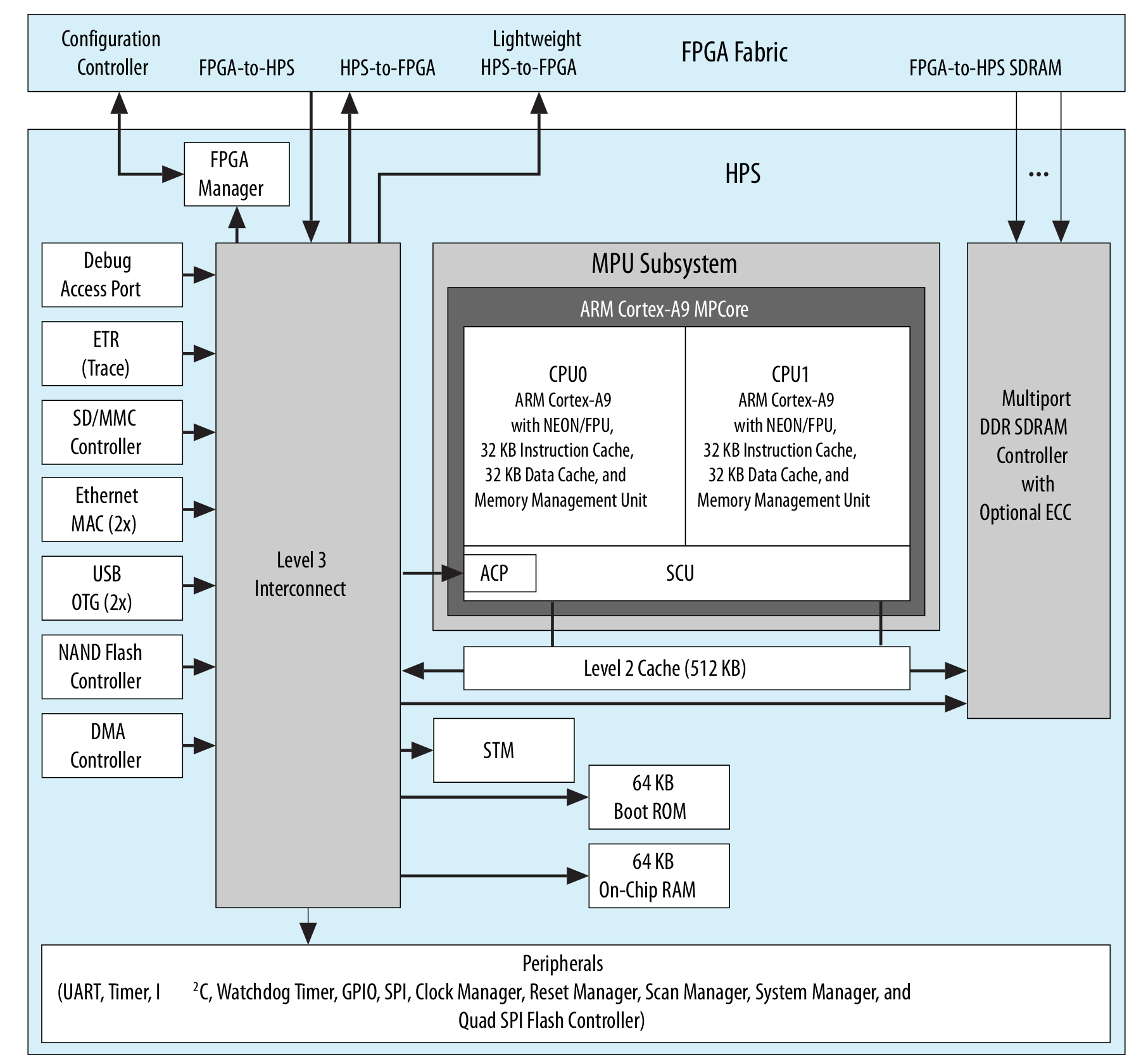
Самой подходящей ролью для устройства, спроектированного композиционным методом конструирования, является роль slave-устройства. Вместе с этим стала задача автоматизации создания из такого проекта. Для этого разработана программа, которая генерирует slave-устройство для САПР Quartus II, которое можно применять в программе Qsys из программного комплекса Quartus II. Устройство имеет следующие порты:

* RST\_I – сигнал сброса;
* CLK\_I – тактовый сигнал;
* WE\_I – сигнал записи;
* STB\_I – строб, сигнал выбора микросхемы или chipselect;
* ADR\_I – порт адреса;
* DAT\_I – входные данные (writedata);
* DAT\_O – выходные данные (readdata);
* IRQ\_O – сигнал прерывания.

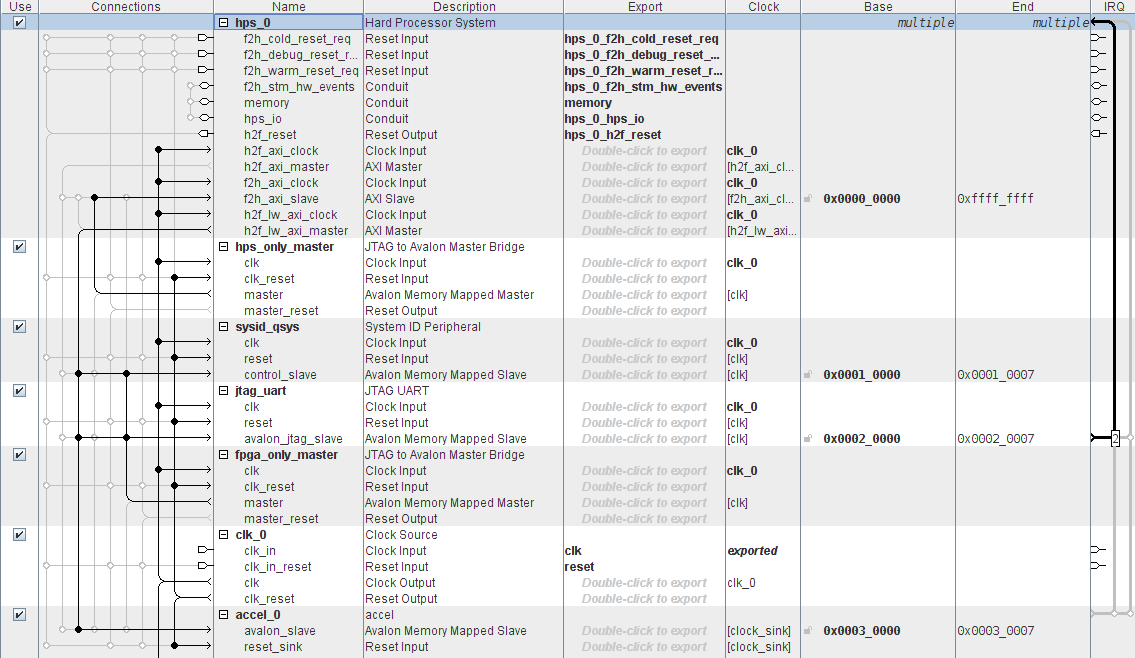
Следует заметить некоторые особенности в этом наборе портов. В первую очередь это отсутствие сигнала чтения, но без него можно обойтись, так как считываться с устройства будут только одни данные – результат. Следующей особенностью есть отсутствие сигнала waitrequest, но это приемлемо, так как master-устройство на этой шине будет одно – центральный процессор. Отсутствует и сигнал byteenable, его нет, так как объем всех транзакций на шине кратны одному слову. Добавлен сигнал прерывания IRQ\_O, с помощью которого устройство может сигнализировать процессору о завершении вычислений.

Естественно, само по себе устройство бесполезно. Следует рассмотреть каким же образом его можно интегрировать, например, в систему на кристалле. Рассмотрим типичную систему на кристалле с интегрированным ускорителем вычислений. Рассмотрим это на примере серии Altera Cyclone V со

встроенным ARM ядром, так называемой Hardware Processor System (HPS)(рис. 15).

Рис 15.Структурная схема микросхемы Cyclone V.

HPS с процессорными ядрами ARM Cortex-A9 связана с FPGA посредством AXI мостов FPGA-to-HPS и HPS-to-FPGA, а также их легковесных (lightweight) версий. Таким образом master и slave устройства могут размещаться как на ПЛИС, так и на процессорной системе. Кроме самого ускорителя вычислений в системе на кристалле (рис. 16) располагаются другие необходимые функциональный блоки.

Рис. 16. Пример системы на кристалле с ускорителем.

* hps\_0 – сама аппаратная процессорная система;
* hps\_only\_master – мост из отладочного интерфейса JTAG на шину Avalon для отладки работы процессора;
* sysid\_qsys – модуль идентификации системы на кристалле;
* fpga\_only\_master – мост из отладочного интерфейса на шину Avalon для отладки устройств на FPGA;
* jtag\_uart – преобразователь интерфейсов JTAG<->UART для консольного доступа к HPS по последовательному интерфейсу;
* clk\_0 – источник тактовых импульсов и сигнала сброса;
* accel\_0 – само спроектированное устройство-ускоритель вычислений.

Осталось рассмотреть как такое устройство будет выглядеть со стороны программиста его использующего, какова будет его программистская модель.

Устройство представляется как набор регистров в памяти. Каждый регистр на запись – один из аргументов функции, воплощенной в проекте. При чтении любого из регистров, считывается результат вычислений.

Рассмотрим пример программы для ОС GNU/Linux, которая осуществляет работу с таким модулем ускорения:

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/mman.h>

#define REGISTER\_SPAN 0x04000000

#define ADDRESS\_STM 0xFC000000

#define OFFSET\_LWFPGASLAVES 0x03200000

#define ACCELERATOR\_OFFSET 0x00030000

void \*accelerting\_device;

int main(int argc, char \*argv[]) {

void \*virtual\_base;

int fd;

if( ( fd = open( "/dev/mem", ( O\_RDWR | O\_SYNC ) ) ) == -1 ) {

printf( "ERROR: could not open \"/dev/mem\"...\n" );

return( 1 );

}

virtual\_base = mmap( NULL, REGISTER\_SPAN, ( PROT\_READ | PROT\_WRITE ),

MAP\_SHARED, fd, ADDRESS\_STM);

if( virtual\_base == MAP\_FAILED ) {

printf( "ERROR: mmap() failed...\n" );

close( fd );

return( 1 );

}

printf("Starting...\n");

accelerting\_device=virtual\_base +

(unsigned int)(OFFSET\_LWFPGASLAVES + ACCELERATOR\_OFFSET);

\*((unsigned int \*)(accelerting\_device) + 0x00) = 0x11111111;

\*((unsigned int \*)(accelerting\_device) + 0x01) = 0x22222224;

sleep(1);

printf("Result1 %08X\n", \*((unsigned int \*)(accelerting\_device) + 0x00));

if( munmap( virtual\_base, REGISTER\_SPAN) != 0 ) {

printf( "ERROR: munmap() failed...\n" );

close( fd );

return( 1 );

}

close( fd );

return( 0 );

}

Первое следует отметить то, что ОС GNU/Linux в подавляющем большинстве своих вариантов работает в защищенном режиме, что значит то, что реальные (физические) адреса не доступны пользователю, так как каждое приложение операционной системы выполняется в пользовательском пространстве адресов (user-space). Для того, что бы получить доступ к конкретному диапазону реальных адресов (например, где находится ускоритель), необходимо использовать устройство /dev/mem, работа с которым напоминает работу с файлом.

Сперва его надо открыть функцией open получить на него дескриптор (fh).

fd = open( "/dev/mem", ( O\_RDWR | O\_SYNC ) )

Флаг O\_RDWR значит, что устройство открыто как для чтения, так и для записи, O\_SYNC значит, что функции чтения и записи на устройство не должны быть отложенными.

Дальше вызывается функция mmap, которая способна отразить диапазон реальных адресов в пространство виртуальных адресов пользовательского приложения.

virtual\_base = mmap( NULL, REGISTER\_SPAN, ( PROT\_READ | PROT\_WRITE ), MAP\_SHARED, fd, ADDRESS\_STM);

Первый аргумент NULL означает, что не имеет значения в какой диапазон отражать диапазон реальных адресов. REGISTER\_SPAN – продолжительность диапазона, который нужно отразить. PROT\_READ и PROT\_WRITE – флаги, описывающие допустимые операции с памятью, флаг MAP\_SHARED значит, что изменения в виртуальной памяти без задержки синхронизируются с памятью реальной, fd – дескриптор открытого устройства /dev/mem, ADDRESS\_STM начало диапазона адресов, который будет отражаться в виртуальную память совпадает с началом диапазона адресов, который выделен под разного рода шины и устройства.

Дальше вычисляется смещение для конкретного ускорителя, который сконструирован:

accelerting\_device=virtual\_base + (unsigned int)(OFFSET\_LWFPGASLAVES + ACCELERATOR\_OFFSET);

Согласно рис. 16. устройство находмтся на шине lightweight FPGA slaves, которая имеет смещение OFFSET\_LWFPGASLAVES по смещению ACCELERATOR\_OFFSET.

Теперь с регистрами устройства можно работать как с участком памяти.

\*((unsigned int \*)(accelerting\_device) + 0x00) = 1;

\*((unsigned int \*)(accelerting\_device) + 0x01) = 2;

result = \*((unsigned int \*)(accelerting\_device) + 0x00));