МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ

РАБОТА БАКАЛАВРА

ИСПОЛЬЗОВАНИЕ ВЫСОКОУРОВНЕВЫХ СРЕДСТВ ПРОЕКТИРОВАНИЯ ДЛЯ РАЗРАБОТКИ АППАРАТНЫХ ФИЛЬТРОВ НА ПЛИС LATTICE

Студент гр. 3530901/60101 Н.С. Макаревич

Санкт-Петербург
2020г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Директор высшей школы _____

_____,

_____ В.М. Ицыксон

«_____»_____ 20__г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

ИСПОЛЬЗОВАНИЕ ВЫСОКОУРОВНЕВЫХ СРЕДСТВ ПРОЕКТИРОВАНИЯ ДЛЯ РАЗРАБОТКИ АППАРАТНЫХ ФИЛЬТРОВ НА ПЛИС LATTICE

по направлению подготовки **09.03.01** «Информатика и вычислительная техника» по образовательной программе **09.03.01_01** «Вычислительные машины, комплексы, системы и сети»

Выполнил студент гр. 3530901/60101 ——————— Н.С. Макаревич

Научный руководитель
старший преподаватель ВШИСиСТ ИКНТ ——————— А.А. Федотов

Консультант по нормоконтролю
старший преподаватель ВШИСиСТ ИКНТ ——————— С.А. Нестеров

Санкт-Петербург
2020г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

УТВЕРЖДАЮ

Директор высшей школы _____

_____,

_____ В.М. Ицыксон

«____»_____ 20__г.

ЗАДАНИЕ

на выполнение выпускной квалификационной работы бакалавра

студенту Макаревичу Никите Сергеевичу, группа 3530901/60101

1. Тема выпускной квалификационной работы: Использование высокоуровневых средств проектирования для разработки аппаратных фильтров на ПЛИС Lattice

2. Срок сдачи студентом законченной выпускной квалификационной работы: 15.06.2020.

3. Исходные данные к выпускной квалификационной работе: отечественная и зарубежная литература и периодика.

4. Содержание расчётно-пояснительной записки (перечень подлежащих разработке вопросов):

а) Изучение доступных средств высокоуровневого синтеза

б) Изучение среды разработки Lattice и возможностей целевой платформы

в) Создание описания цифровых фильтров при помощи средств высокоуровневого синтеза

г) Создание описания адаптивного фильтра на основе неадаптивного фильтра

5. Дата выдачи задания «____»_____ 20__г.

Научный руководитель *старший преподаватель ВШИСиСТ ИКНТ*	_____(_____) *подпись расшифровка*	Федотов А.А.
Задание принял к исполнению	_____(_____) *подпись расшифровка*	Макаревич Н.С.

РЕФЕРАТ

Работа содержит 93 страницы, 29 рисунков, 5 таблиц, 4 приложения

Ключевые слова: ПЛИС, LATTICE SEMICONDUCTOR, ФИЛЬТР, ВЫСОКОУРОВНЕВЫЙ СИНТЕЗ, CHISEL, SCALA

В работе рассмотрены различные варианты цифровых фильтров, синтезированные на ПЛИС при помощи средств языка Chisel, Firrtl, и САПР Lattice Diamond. Рассмотрены методы синтеза адаптивного цифрового фильтра при помощи средств Chisel.

В процессе выполнения работы я рассмотрел особенности конструирования фильтров на ПЛИС, существующие средства высокоуровневого синтеза, сравнил их возможности, полезные для решения поставленной задачи.

Проведён сравнительный анализ синтеза фильтров под разные аппаратные платформы двумя разными средствами синтеза: Lattice LSE и Synplify Pro.

В симуляторе Aldec ActiveHDL я проверил работу синтезированных фильтров с адаптацией и без неё. Весовые коэффициенты для тестов были получены при помощи Matlab.

ABSTRACT

This work contains 93 pages, 29 figures, 5 tables, 4 appendices

Keywords: FPGA,LATTICE SEMICONDUCTOR, FILTER, HIGH-LEVEL SYNTHESIS, CHISEL, SCALA

This work is devoted to solving the problem of developing different types of digital filters on FPGA. Synthesis is performed with Chisel, Firrtl and Lattice Diamond IDE tools. Filter with static weight coefficients and adaptive filter designs compared in this work.

In the course of this work I examined the features of designing filters for hardware platform (FPGAs). I reviewed existing high-level synthesis tools and compared their capabilities useful for solving the problem.

I compared filter synthesis results for different hardware platforms with two different synthesis tools: Lattice LSE and Synplify Pro.

The testing of synthesized modules was performed with Aldec ActiveHDL simulation tool, adaptive and static weights used in testing design. For the testing purpose I generated weight coefficients for filter using Matlab.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В этой работе мы рассматриваем возможности синтеза цифровых фильтров при помощи высокоуровневых средств проектирования, при этом в качестве целевой платформы рассматривается ПЛИС компании Lattice Semiconductor. Рассматриваются разные виды и конструкции цифровых фильтров.

Цели работы: изучить доступные средства высокоуровневого синтеза, совместимые с ПЛИС Lattice Semiconductor и пригодные для синтеза цифровых фильтров. Создать описание разных видов цифровых фильтров и проверить их работоспособность. Так как целевой платфрормой является ПЛИС, рассматриваются только варианты фильтров, пригодные для аппаратной реализации.

Задачи: Провести анализ разных типов цифровых фильтров, возможности их реализации на ПЛИС и особенности аппаратной реализации фильтров. Рассмотреть принципы работы адаптивных цифровых фильтров. Рассмотреть инструменты для работы с ПЛИС Lattice Semiconductor – среду разработки Lattice Diamond и её составные части – некоторые из инструментов от компании Lattice. Провести сравнительный анализ различных средств высокоуровневого синтеза и их пригодность для работы с цифровыми фильтрами. Создать описание разных типов цифровых фильтров, учитывая особенности целевой платформы. Рассматриваются простые (параллельные) и алгоритмические фильтры, адаптивные и неадаптивные.

В этой работе для синтеза цифровых фильтров используется язык описания аппаратных средств Chisel. Для того, чтобы скомпилировать описание устройства на ПЛИС Lattice из кода на Chisel синтезируется описание на Verilog, а Verilog синтезируется средствами САПР Lattice Diamond. Сравнивая созданные устройства, мы используем отчёты компилятора в среде Lattice Diamond, а также отчёты симулятора кода на Verilog – ActiveHDL.

Актуальность темы дипломной работы объясняется тем, что цифровые фильтры – один из наиболее частоиспользуемых аппаратных блоков в задачах ЦОС. Появляются новые инструменты для описания аппаратных устройств, многие из них распространяются как свободное ПО и поддерживаются сообществом.

Новые возможности языков описания аппаратных средств позволяют создавать более гибкие и параметризуемые шаблоны для описания цифровых фильтров, что в перспективе должно экономить время разработчиков описаний на ПЛИС. Цифровые фильтры часто реализуют на процессоре, но иногда производить фильтрацию сигнала на процессоре нецелесообразно. Это может быть слишком трудоёмко или слишком долго. В некоторых устройствах вообще отсутствует процессор, и тогда легче создать аппаратный фильтр на ПЛИС. В этой работе рассмотрены разные конструкции фильтров: для ситуаций, когда важнее создать быстрый фильтр, либо когда важнее затратить меньше аппаратных ресурсов. Предполагается, что современные средства высокоуровневого синтеза помогут создать более гибкое и параметризуемое описание устройства, чем традиционные сейчас языки разработки: Verilog и VHDL.

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1. Виды цифровых фильтров

Цифровой фильтр – это любой фильтр, работающий с цифровым сигналом. В нашем случае цифровой фильтр синтезируется на ПЛИС, но он также может быть реализован и другими способами, например на интегральных схемах или программно.

Любой цифровой фильтр может быть описан математически. Если аналоговый фильтр – непрерывная система и может быть описан дифференциальным уравнением n-го порядка, то цифровой фильтр – дискретная система и он описывается разностным уравнением. Существуют нерекурсивные и рекурсивные цифровые фильтры. Нерекурсивные фильтры также называют КИХ-фильтрами (фильтрами с конечной импульсной характеристикой), а рекурсивные называют БИХ-фильтрами (фильтрами с бесконечной импульсной характеристикой).

Это наиболее распространённые типы фильтров в цифровой обработке сигналов. Существуют и другие типы, но в данной работе рассматриваются только эти два типа: КИХ и БИХ фильтры, так как они лучше всего подходят для аппаратной реализации на ПЛИС или интегральных схемах.

Передаточная функция этих фильтров имеет следующий вид:

$$K(z) = a_0 + a_1 z^{-1} + \cdots + a_n z^{-n} \tag{1}$$

$$K(z) = \frac{a_0 + a_1 z^{-1} + \cdots + a_n z^{-n}}{1 - b_1 z^{-1} - \cdots - b_n z^{-n}} \tag{2}$$

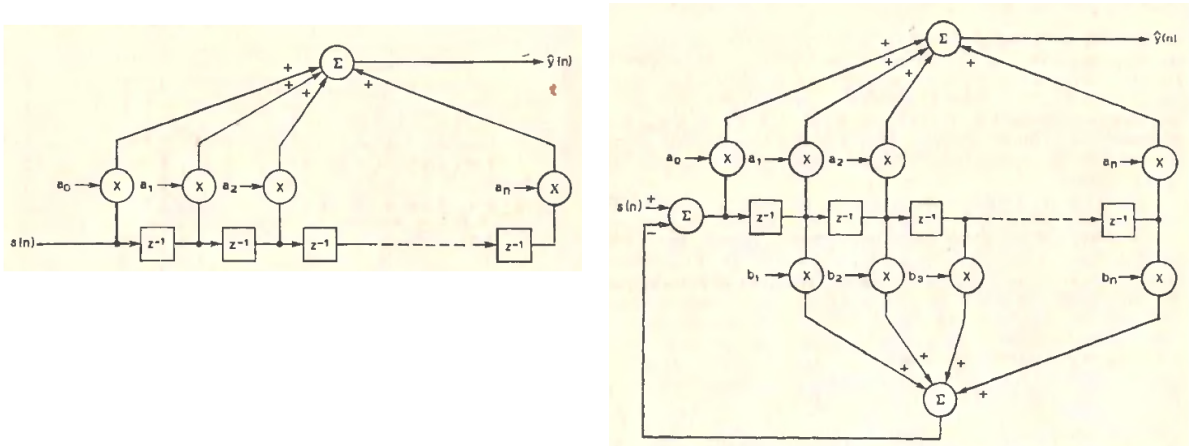
Нерекурсивный фильтр – уравнение 1, рекурсивный фильтр – уравнение 2.

Эти фильтры соответственно описываются разностными уравнениями 3 и 4:

$$f_2[n] = a_0 f_1[n] + a_1 f_1[n-1] + \cdots + a_n f_1[n-n] \tag{3}$$

$$f_2[n] = a_0 f_1[n] + a_1 f_1[n-1] + \cdots + a_n f_1[n-n] + b_1 f_2[n-1] + \cdots + b_n f_2[n-n] \tag{4}$$

Можно заметить, что поведение такой системы полностью определяется весовыми коэффициентами фитьтра a_i b_i. Нерекурсивный фильтр можно рассматривать как частный случай рекурсивного фильтра, у которого весовые коэффициенты $b_i = 0$. a_i – это коэффициенты умножителей с прямой связью, а b_i – коэффициенты умножителей с обратной связью. В результате построения структуры получается фильтр с характеристикой полюсно-нулевого типа, где размещение полюсов определяется коэффициентами b_i, а размещение нулей – коэффициентами a_i. [3] С.17.



(а) Нерекурсивный фильтр (КИХ) (б) Рекурсивный фильтр (БИХ)

Рис. 1.1. Структурная схема нерекурсивного 1.1а и рекурсивного 1.1б фильтра. [3] С.18-19.

Можно заметить, что у БИХ-фильтра коэффициентов b_i на 1 меньше, чем a_i. Для удобства обращения с БИХ-фильтрами принимают коэффициент $b_0 = 1$. Это позволяет работать с A и B как с двумя массивами коэффициентов одинаковой длины.

Использование каждого из этих двух типов фильтров имеет свои преимущества и недостатки. При использовании КИХ-фильтра мы можем быть уверены, что система всегда устойчива. [1] С.115. Фильтр не имеет обратных связей, а следовательно его выходное значение зависит только от фиксированного количества отсчётов времени (фильтр имеет ограниченную память).

При использовании БИХ-фильтра мы имеем дело со структурой, теоретически имеющей бесконечную память (бесконечную импульсную характеристику). Её сложнее моделировать и описывать математически, но при этом система более

гибкая для настройки. Для получения частотной характеристики с крутым срезом мы можем использовать БИХ-фильтр меньшего порядка, чем КИХ-фильтр с похожей характеристикой. [3]. Также БИХ-фильтр может быть неустойчив, если неправильно подобраны коэффициенты b_i.

БИХ-фильтр может быть представлен в виде двух разных типов структур, работающих одинаково [13] С.344.

Представленная ниже на рис. 1.2 форма БИХ-фильтра – наиболее часто используемая структура, в которой присутствует только одна линия задержки сигнала (на схеме обозначено как z^{-1}) и два вектора весовых коэффициентов (см. рис. 1.2 также рис. 1.1б).

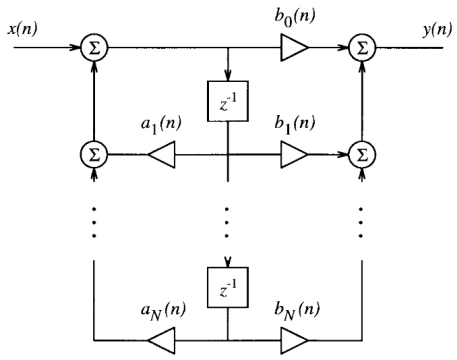


Рис. 1.2. БИХ-фильтр в форме 1

Бих-фильтр в этой форме (в тексте будем называть её форма 1) описывается уравнением:

$$\nu(n) = x(n) + \sum_{i=1}^{N} a_i(n) \cdot \nu(n-i); \qquad y(n) = \sum_{j=0}^{N} b_j(n) \cdot \nu(n-j) \qquad (5)$$

БИХ-фильтр в форме 2 включает уже две линии задержки сигнала и два вектора весовых коэффициентов. Фактически это два КИХ-фильтра, соединённых

между собой [2].

$$y(n) = -\sum_{i=1}^{N} \frac{a_i(n)}{a_0(n)} y(n-i) + \sum_{j=0}^{N} \frac{b_j(n)}{a_0(n)} x(n-j)$$

или (6)

$$y(n) = -\sum_{i=1}^{N} a_i(n) y(n-i) + \sum_{j=0}^{N} b_j(n) x(n-j); \qquad a_0 = 1$$

Для удобства вычисления можем записать уравнение 6 в векторном виде:

$$y(n) = \frac{W^T(n) U(n)}{a_0} \tag{7}$$

, где

$$\begin{aligned} W(n) &= [a_1(n), a_2(n) \cdots a_N(n), b_0(n), b_1(n), b_2(n) \cdots b_N(n)]^T \\ U(n) &= [y(n-1), y(n-2) \cdots y(n-N), x(n), x(n-1) \cdots x(n-N)]^T \end{aligned} \tag{8}$$

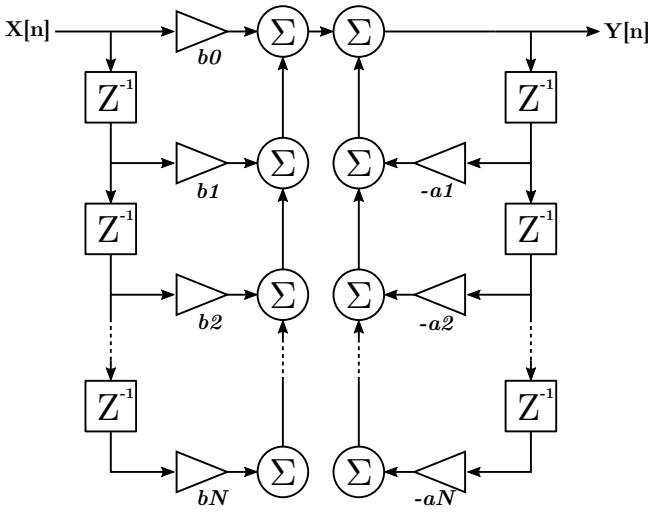


Рис. 1.3. БИХ-фильтр в форме 2 [2]

При выборе типа фильтра мы в первую очередь обращаем внимание на сложность вычисления выходных значений. От этого параметра зависит количество занимаемых логических ячеек ПЛИС и время вычисления одного выходного значения $y(n)$ [13] С.430.

Эта структура хорошо отражает уравнение 6, она редко используется в программных реализациях фильтров, так как требует больше памяти, однако при аппаратной реализации фильтров форма 2 имеет ряд преимуществ, о которых сказано в следующем разделе.

1.2. Особенности реализации БИХ-фильтров на ПЛИС

Программная и аппаратная реализация цифровых фильтров сильно отличаются. С КИХ-фильтрами ситуация обстоит проще, так как у них отсутствуют обратные связи, достаточно произвести ряд умножений и сложений.
У БИХ-фильтров ситуация осложняется тем, что из-за присутствующих в структуре обратных связей мы должны разобраться с возрастающей при каждом умножении разрядностью. В программной реализации КИХ и БИХ фильтров чаще всего задействуют аппаратную поддержку процессором операций с плавающей точкой и используют числа с плавающей точкой для хранения входных значений и весов. При реализации фильтров на ПЛИС чисел с плавающей точкой не используют, так как это сильно усложняет схему. Используют один из двух следующих подходов:

а) Использовать арифметику с фиксированной точкой. В этом случае мы берём числа с фиксированной точкой, достаточно хорошо приближенные к оригинальным весовым коэффициентам и производим все операции с учётом фиксированной точки. Для вычислений с фиксированной точкой в Chisel (раздел 5) существует специальная абстракция FixedPoint(), позволяющая автоматизировать простые арифметические операции с ними. Пока этот инструмент является экспериментальным.

б) Использовать только целые числа, выбрав значение a_0. Этот вариант более предпочтителен, так как позволяет нам отказаться от арифметики с фиксированной точкой. Как показано в [2], если увеличить a_0, можно также увеличить и другие весовые коэффициенты, избавившись от дробной части. В таком случае выход нашего фильтра будет описываться уравнением

7, а его структурную схему можно изобразить так:

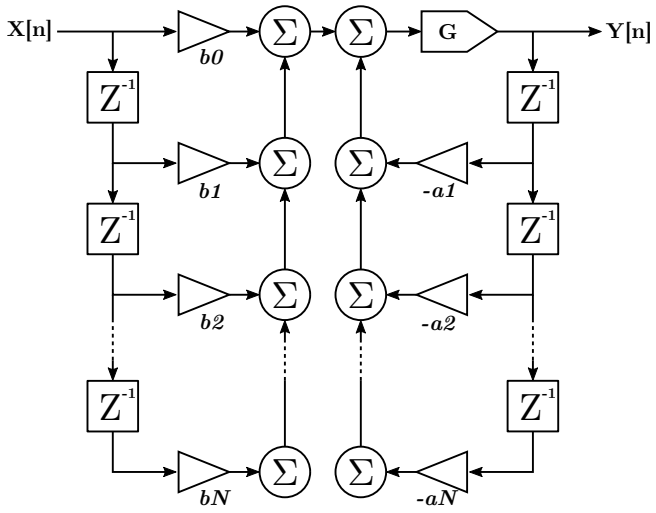


Рис. 1.4. Структурная схема целочисленного БИХ-фильтра

На этом рисунке буквой G обозначен коэффициент, равный $\frac{1}{a_0}$. В аппаратной реализации мы можем заменить деление сдвигом вправо на определённое количество разрядов, а следовательно более предпочтительно выбирать a_0 равным степени числа 2. Это позволит обойтись без полноценного аппаратного делителя. Мы можем выбрать определённое значение a_0, а уже относительно него рассчитать остальные целочисленные весовые коэффициенты.

Для реализации фильтра на ПЛИС мы выберем структуру типа 2, так как она позволяет легко контролировать разрядность данных в линиях задержки и использовать целочисленную арифметику.

1.3. Адаптивность цифровых фильтров

Адаптивность в общих чертах можно описать так: фильтр автоматически подстраивает свои коэффициенты с целью достичь желаемого результата. Обобщённую схему адаптивного цифрового фильтра можно представить в следующем виде: рис. 1.5

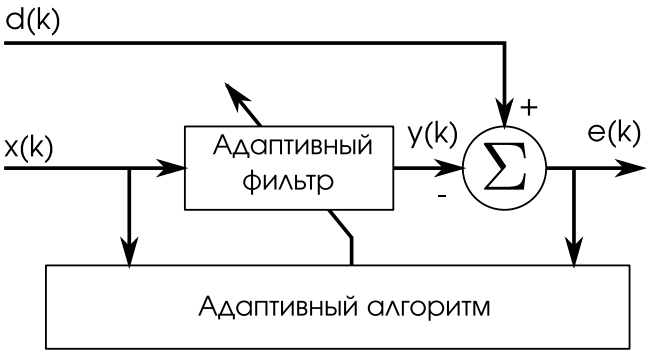


Рис. 1.5. Структурная схема адаптивного фильтра

а) $x(k)$ – это входной сигнал

б) $y(k)$ – выходной сигнал

в) $d(k)$ – требуемый сигнал, который так же можно назвать обучающей последовательностью (обозначается $d(k)$ от слова “desired”)

г) $e(k)$ – сигнал ошибки, разность желаемого и действительного сигналов

По своей структуре адаптивный фильтр – это обыкновенный неадаптивный фильтр, к которому добавлен модуль адаптивного алгоритма [3] С.30. Адаптивный алгоритм не всегда использует все три источника ($(x(k), e(k), d(k))$, иногда достаточно даже одного. Помимо входного и выходного сигналов может быть использован требуемый сигнал – это произвольная последовательность. Она может быть заранее сгенерирована, или же может генерироваться прямо на ПЛИС/на процессоре в реальном времени в зависимости от задачи.

Адаптивный КИХ-фильтр, в отличие от обыкновенного КИХ-фильтра, будет уже являться нелинейной системой, так как его весовые коэффициенты меняются в процессе работы. Однако, в любой фиксированный момент времени он будет являться линейной системой. Если отключить адаптивный алгоритм, то он превращается в обыкновенный КИХ-фильтр.

Содержание требуемой последовательности $d(k)$, а также адаптивный алгоритм рис. 1.5 полностью зависит от области применения фильтра. Существуют различные алгоритмы подстройки весовых коэффициентов, некоторые из которых

мы рассмотрим в этой работе.

основную зависимость выхода от входа адаптивного фильтра можно записать так [13] С.428 :

$$\begin{aligned} y(n) = f(&W(n), y(n-1), y(n-2), \cdots, y(n-N), \\ &x(n), x(n-1), \cdots, x(n-M+1)) \end{aligned} \tag{9}$$

Выше мы рассмотрели (рис. 1.5), что метрика ошибки для адаптивного фильтра получается с участием сигнала $d(n)$ – желаемого сигнала. Но при этом мы по ряду причин не можем использовать $d(n)$ и вынуждены создавать адаптивный фильтр.

Адаптивный алгоритм не всегда опирается на $d(n)$, иногда это $x(n)$, $y(n)$, или что-то другое. Есть ситуации, в которых $d(n)$ не доступно всё время. В таких ситуациях адаптивный алгоритм работает только пока $d(n)$ нам доступно. В различных стандартах связи на физическом уровне распространённая практика – в начале работы создавать в линии связи специальный сигнал, по которому могут адаптироваться фильтры на принимающей стороне (эхоподавители и эквалайзеры) [8] С.3. Затем адаптивный алгоритм отключают и в процессе работы устройства весовые коэффициенты уже статичны [13] С.430.

Использование адаптивных фильтров для анализа систем

Адаптивные фильтры изучаются много лет и существует много примеров их применения на практике

На рисунке 1.6 пунктирной линией обведена система, параметров которой мы не знаем (black box) и внутренние сигналы нам недоступны. $\eta(n)$ на схеме обозначает воздействие внешних помех на выходной сигнал системы [13] С.431.

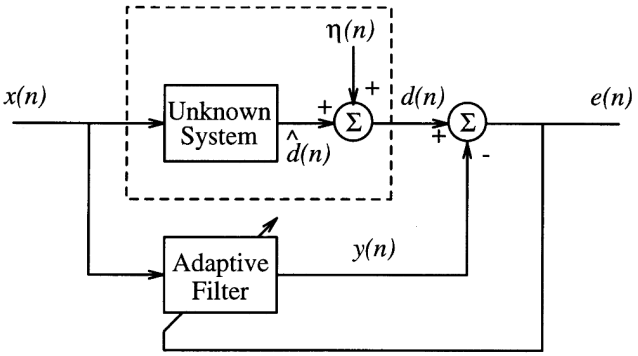


Рис. 1.6. Схема использования адаптивного фильтра для анализа систем

При помощи адаптивного фильтра мы можем “обучить” весовые коэффициенты и в итоге получить приближённую функцию зависимости выхода неизвестной системы от входа. Желаемый сигнал $d(n) = \widehat{d}(n) + \eta(n)$ Главная задача адаптивного фильтра в этом случае – как можно точнее получить приближение сигнала $\widehat{d}(n)$.

Если мы используем для анализа системы линейный фильтр, то наилучшие результаты он будет показывать с линейными системами, близкими по структуре с фильтром.

$$d(n) = W_{opt}^{T}(n)X(n) + \eta(n) \tag{10}$$

Здесь $W_{opt}(n)$ – это оптимальный набор весовых коэффициентов – результат адаптации.

При помощи этого способа чаще всего подавляют эхо, возникающее в канале передачи данных и в приёмной части устройств. Сначала происходит обучение фильтра эхоподавления, мы добиваемся, чтобы его поведение было максимально приближено к поведению канала передачи данных. Затем при помощи этого фильтра мы можем компенсировать реальное эхо, возникающее в канале [10].

Обратное преобразование сигнала

Принцип работы адаптивного фильтра в этом режиме хорошо иллюстрирует рисунок 1.7. На нём изображена неизвестная нам система, выход которой поступает на вход адаптивного фильтра. В качестве желаемого сигнала $d(n)$ выступает входной сигнал системы, проходящий через лилию задержки (задержка равна

времени прохождения сигнала через неизвестную систему).

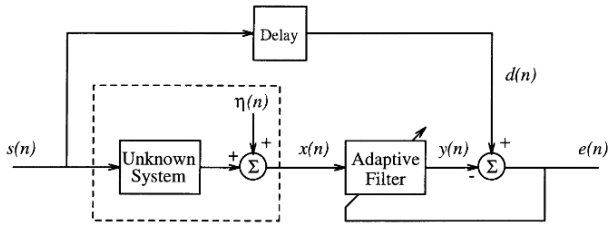


Рис. 1.7. Схема использования адаптивного фильтра для получения обратного преобразования сигнала

Задача адаптивного фильтра в этом случае – воздействие на сигнал таким образом, чтобы он был как можно больше приближен к исходному входному сигналу.

Типичный случай применения такого подхода – подстраивание приёмной части устройств под особенности канала передачи данных. В системах связи информация передаётся между устройствами через каналы разлчного качества и разной протяжённости. В канале возникают межсимвольные помехи – *inter-symbol interference (ISI)*. [8] С.4. В таком случае адаптивные фильтры применяются чтобы подстроиться под определённый канал связи, минимизировав межсимвольные помехи. Фильтры для борьбы с межсимвольными помехами применяются тогда, когда более простые средства распознавания сигнала не могут быть использованы или же когда мы имеем компромисс между скоростью передачи и протяжённостью линии [13] С.433.

На рисунке ниже показаны три временные диаграммы принятого сигнала [8] С.13.

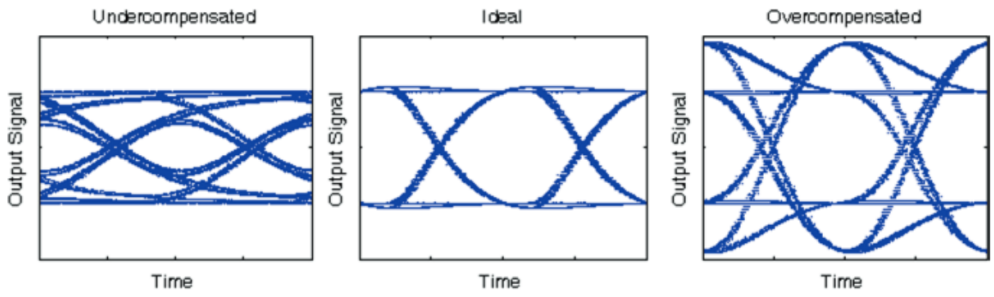


Рис. 1.8. Пример различных уровней компенсации эквалайзера

На первой изображён сигнал с недостаточным уровнем компенсации эквалайзером, на второй – сигнал, оптимально компенсированный эквалайзером, а на третьей – слишком сильное воздействие эквалайзера на сигнал. Для того, чтобы приблизить воздействие эквалайзеров к оптимальному, используются адаптивные фильтры и различные критерии оценивания ошибки.
Также пример работы цифрового адаптивного эквалайзера показан в [8] С.15.

Линейное предсказание
Этот вариант позволяет частично предсказывать поведение систем, особенно если это линейные системы. Схема использования представлена ниже:

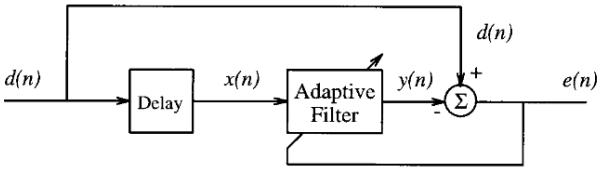


Рис. 1.9. Схема использования адаптивного фильтра для линейного предсказания

В данном случае прии обучении на вход адаптивного фильтра подаётся сигнал с задержкой, а в качестве желаемого сигнала используется тот же сигнал, только без задержки. Тогда адаптивный фильтр будет стремиться предсказать входной сигнал в будущем, используя набор предыдущих сэмплов.
Например, если во входном сигнале присутствуют полезные данные и другой сигнал, имеющий периодическую природу, то мы можем обучить адаптивный фильтр предсказывать его поведение (форму) и таким образом отделить полезные данные от шума.

1.4. Адаптивные алгоритмы

Адаптивный алгоритм – это процедура подстройки параметров фильтра с целью минимизации оценочной функции. Оценочная функция выбирается разработчиком в зависимости от поставленной задачи [3] С.30. Здесь мы рассмотрим адаптивные алгоритмы применительно к КИХ-фильтрам, так как КИХ-фильтры

используются чаще, чем БИХ. КИХ-фильтры гарантированно устойчивы и алгоритм подстройки весовых коэффициентов для них реализуется проще.
Адаптивный алгоритм в обобщённой форме может быть записан так: [15] С.2-3, [13] С.437.

$$W(n+1) = W(n) + \mu(n)G(e(n), X(n), \Phi(n)) \tag{11}$$

G – это определённая векторная функция, в которой $\Phi(n)$ – это вектор состояний, в которых хранится информация о характеристиках входных сигналов/сигналов ошибки/весовых коэффициентов в предыдущие моменты времени. $\mu(n)$ – это размер шага адаптации. В простейших алгоритмах $\Phi(n)$ не используется и единственная информация, необходимая для регулировки коэффициентов, - это сигнал ошибки, вектор входного сигнала и размер шага.
Размер шага $\mu(n)$ определяет магнитуду изменения весовых коэффициентов. Выбор размера шага зависит от назначения адаптивного фильтра.
В этой работе мы будем рассматривать фильтр с адаптивным LMS алгоритмом (least mean square) [3] С.46, [13] С.437, [15] С.10-18. Его оценочная функция – функция наименьших квадратов. Она не использует $\Phi(n)$ и записывается следующим образом:

$$J_{LS}(n) = \sum_{k=0}^{n} \alpha(k) \left(d(k) - W^T(n)X(k)\right)^2 \tag{12}$$

Здесь α – это подобранная взвешивающая последовательность, регулирующая влияние каждого значения на конечный результат. Такая оценочная функция сложна, она требует многократных вычислений для получения результата. В упрощённом виде оценочную функцию можно записать так:

$$J_{LMS}(n) = \frac{1}{2}e^2(n) \tag{13}$$

Используя функцию, оценивающую отклонение желаемого сигнала от действительного, мы можем выразить новые весовые коэффициенты через старые: [13] С.550

$$W(n+1) = W(n) + \mu(n)e(n)X(n) \tag{14}$$

Этот алгоритм хорош тем, что задействует только операции умножения и сложения, а значит его будет проще реализовать на ПЛИС.

1.5. Вывод

Проанализировав разные виды цифровых фильтров было решено, что для реализации на ПЛИС лучше всего подходят фильтры, использующие целочисленную арифметику. Это могут быть как простые фильтры, так и алгоритмические. Фильтры формы 1 лучше подходят для программной реализации с использованием арифметики с плавающей точкой, а фильтры формы 2 – лучше для реализации на ПЛИС с использованием целочисленной арифметики, где необходимо контролировать разрядность данных.

Также были рассмотрены варианты реализации адаптивного фильтра и для экспериментов был выбран LMS-алгоритм, так как он прост и хорошо реализуется аппаартно.

2. ОБЗОР СРЕДЫ РАЗРАБОТКИ LATTICE DIAMOND

Микросхемы FPGA и CPLD компании Lattice Semiconductor, как правило, применяются в устройствах, требующих низкой себестоимости и экономного энергопотребления. [4] Компания Lattice разрабатывает много линеек ПЛИС и для разработки проектов под эти ПЛИС существует несколько САПР (систем автоматического проектирования) от Lattice. Это Lattice ispLever, Lattice Diamond, Lattice ICE Cube и lattice Radiant.

САПР Lattice Diamond пришла на смену системе предыдущего поколения ispLever. Так как мы будем синтезировать модули под ПЛИС семейств ECP3 и MachXO2, то следовательно необходимо использовать среду Lattice Diamond.

В данной среде разработки имеются:

а) Каталог готовых конфигурируемых IP-ядер от Lattice IPexpress

б) Возможность разработки на VHDL, Verilog, SystemVerilog

в) Интеграция с симулятором ActiveHDL от Aldec

г) Два инструмента синтеза кода: LSE (lattice synthesis engine) и Synplify Pro

д) Средства просмотра результатов синтеза

е) Средства оценки соответствия синтезированной схемы временным требованиям

ж) Логический анализатор Reveal Analyzer

з) Средства генерации файлов прошивки и прошивки flash ПЛИС

В САПР Lattice Diamond исользуется концепция имплементаций (Implimentations) и стратегий (Strategies). Пользователю доступна полная информация о проекте: все файлы проекта, иерархия и результаты выполнения всех этапов проектирования. [4]

В данной работе использовалась версия САПР 3.11.

Для запуска САПР необходимо подключить к нему лицензию. Лицензия – это

файл, который можно подключить к проекту при помощи утилиты FlexLM или напрямую через настройки. Лицензия привязана к физическому адресу компьютера (MAC). На Diamond существует лицензия по подписке (Subscription License) и свободная лицензия (Free License), которая предоставляет меньше возможностей, но их для данной работы этих возможностей достаточно.
Возможность работы с ПЛИС ECP3 не входит в стандартную свободную лицензию, но есть возможность запросить у Lattice дополнительную лицензию для этих ПЛИС.
Один проект может содержать несколько имплементаций и стратегий для выбора оптимальной структуры разрабатываемого устройства. Имплементации представляют собой один из возможных вариантов реализации проекта при заданном наборе настроек [4] С.56. Таким образом в одном проекте мы можем переключаться между несколькими реализациями, не меняя при этом код и настройки. Из всех имплементаций в проекте может быть активной только одна.
Каждая имплементация имеет связанную с ней активную стратегию. Стратегия – это совокупность всех параметров утилит САПР, связанных с имплементацией. В САПР Diamond изначально имеется четыре заданные стратегии, но также можно создавать и пользовательские стратегии.

а) Area – стратегия, минимизирующая используемые ресурсы ПЛИС (общее количество логических вентилей).

б) I/O Assistant – Стратегия, оптимизирующая размещение сигналов ввода/-вывода. Помогает распределить выводы плис в начале разработки проекта в соответствии с требованиями к компоновке печатной платы

в) Quick – Быстрая стратегия, позволяющая в начале работы быстро синтезировать проект, удовлетворяя минимальным требованиям, что может экономить рабочее время разработчиков.

г) Timing – Стратегия достижения максимальной тактовой частоты работы устройства. Требует больших вычислительных ресурсов на этапе размещения и разводки.

2.1. Средства синтеза

2.1.1. Lattice LSE

Это средство синтеза, разработанное самой компанией Lattice и ориентированное только на их архитектуру ПЛИС. В некоторых ситуациях оценки результатов синтеза у LSE будут лучше, чем у Synplify pro.

Фронт-энд этого синтезатора поддерживает все современные стандарты VHDL, Verilog и SDC-файлов с требованиями. Первая версия этой программы вышла в 2010 году в составе среды Lattice Diamond.

У этого инструмента есть параметр, задающий приоритеты оптимизации дизайна. Доступны три режима оптимизации: “Area”, “Balanced” и “Timing”. “Area” – акцент делается на уменьшении количества занимаемых логических ячеек. “Timing” – акцент делается на достижении максимальной скорости работы. “Balanced” – это некий компромисс между двумя первыми режимами, используется по-умолчанию.

Как заявляет производитель, LSE показывает наилучшие результаты при работе с чипами серии MachXO/MaxhXO2. В среде разработки Diamond также есть графическая среда для задания требований к дизайну LDC Editor, а также средство просмотра и анализа результатов синтеза – Netlist Analyzer.

Распространяется этот синтезатор вместе с САПР компании Lattice и входит в состав лицензии.

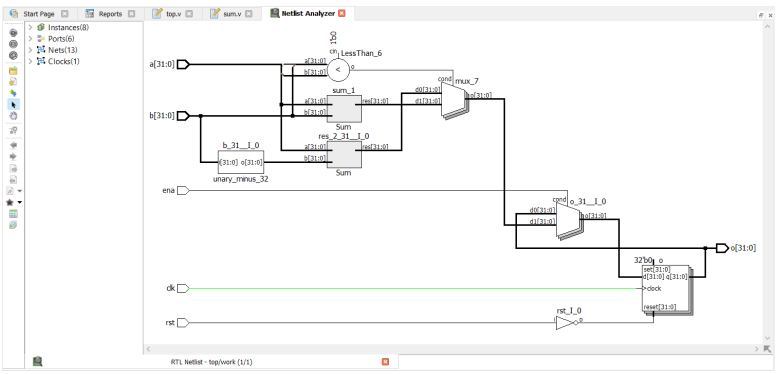


Рис. 2.1. Netlist Analyzer

2.1.2. Synplify pro

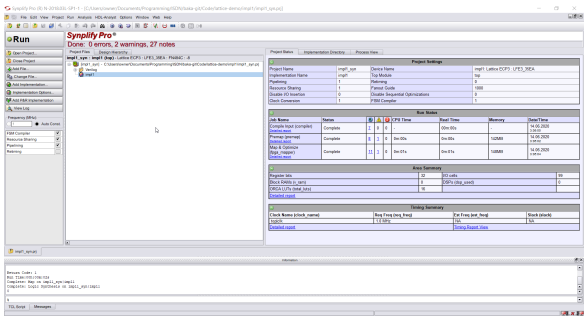
Synplify Pro – это известное средство синтеза от стороннего производителя Synopsys. Этот синтезатор используется не только для ПЛИС компании Lattice, но и для других ПЛИС, в составе других САПР. Поддерживаются плис производства Intel, Achronix, Lattice, Microsemi, Xilinx и другие менее известные производители. Этот синтезатор также поддерживает все современные стандарты Verilog и VHDL. Это позволяет синтезировать один и тот же код с одинаковыми настройками синтезатора на разные платформы.

Synplify Pro поддерживает инкрементальную сборку проектов, к тому же скорость синтеза у него выше, чем у Intel и Xilinx.

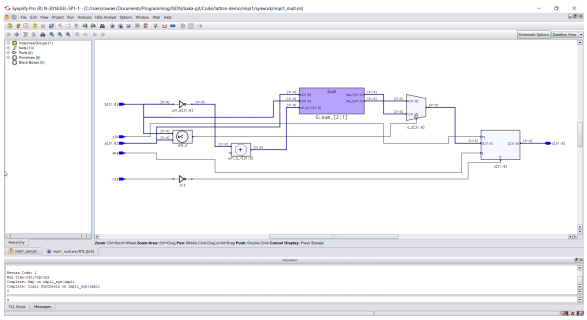
У Synplify Pro есть свой графический интерфейс, который к тому же интегрирован в Lattice Diamond. Он позволяет просматривать результаты синтеза в виде отчётов и схем. У Lattice LSE аналогичные функции выполняет Netlist Analyzer. В окне Synplify Pro на панели слева есть список всех модулей проекта в виде дерева объектов для более удобной навигации. Модули можно просматривать иерархически вплоть до регистров и логических функций, из которых модуль состоит.

Существует также Synplify Premier, который предоставляет пользователю ещё больше возможностей, но мы пользуемся Synplify Pro.

Этот синтезатор (Synplify Pro lattice edition) входит в состав лицензии на САПР Lattice.



(а) Главное окно Synplify Pro



(б) RTL view – графическое отображение схем

Рис. 2.2. Synplify Pro

2.2. Библиотека IPexpress

IPexpress – это каталог IP-ядер LatticeCORE, которые могут быть сконфигурированы при помощи графического интерфейса и включены в проект. Также в каталоге есть IP-ядра-интерфейсы для аппаратных блоков на кристалле ПЛИС. Они находятся в разделе “Architecture_Modules”. Например, у некоторых ПЛИС Lattice присутствует EFB – это аппаратный блок, размещённый на ПЛИС по соседству с логическими ячейками, в нём реализованы стандартные интерфейсы SPI и I2C, а также многофункциональные таймеры-счётчики. Конфигурирование EFB и подключение его к системной шине происходит через интерфейс IP-ядра EFB.

Каждый модуль из этой бибилиотеки сначала конфигурируется при помощи графического интерфейса, а уже после этого САПР генерирует исходный код на Verilog.

Некоторые ядра в этом каталоге лицензируются отдельно и за их использование необходимо платить.

Основные разделы каталога IPexpress:

- Architecture_Modules – интерфейсы архитектурных блоков ПЛИС
- Arithmetic_Modules – модули для выполенния арифметических преобразований, работы с комплексными числами. Также есть модуль для выполнения быстрого преобразования Фурье (FFT) и синусно-косинусная таблица.
- DSP_Modules – модули, предназначенные для цифровой обработки сигналов. В основном различные арифметические операции, также здесь присутствует модуль “1D_FILTER” – КИХ-фильтр .
- DSP_Modules – интерфейсы с различными типами модулей внешней памяти, а также FIFO и память на регистрах.

2.3. Логический анализатор Reveal Analyzer

Это средство отладки на кристалле, входящее в состав Lattice Diamond. У Intel похожий логический анализатор называется “SignalTap Logic Analyzer”.

Для работы со встроенным логическим анализатором в Diamond есть две программы: Reveal Inserter и Reveal Analyzer. Reveal Inserter – это программа, которая при помощи графического интерфейса позволяет настроить логический анализатор. Задать объём памяти, тактирующие сигналы, а также настроить условия (triggers), по которым будет происходить начало работы и считывание данных в память.

После того, как логический анализатор сконфигурирован, мы должны синтезировать проект. Логический анализатор становится частью RTL-описания. На кристалле ПЛИС рядом с нашим устройством синтезируется логика анализатора и ячейки памяти. Чем больше логический объём ПЛИС, тем больше объём памяти возможно синтезировать.

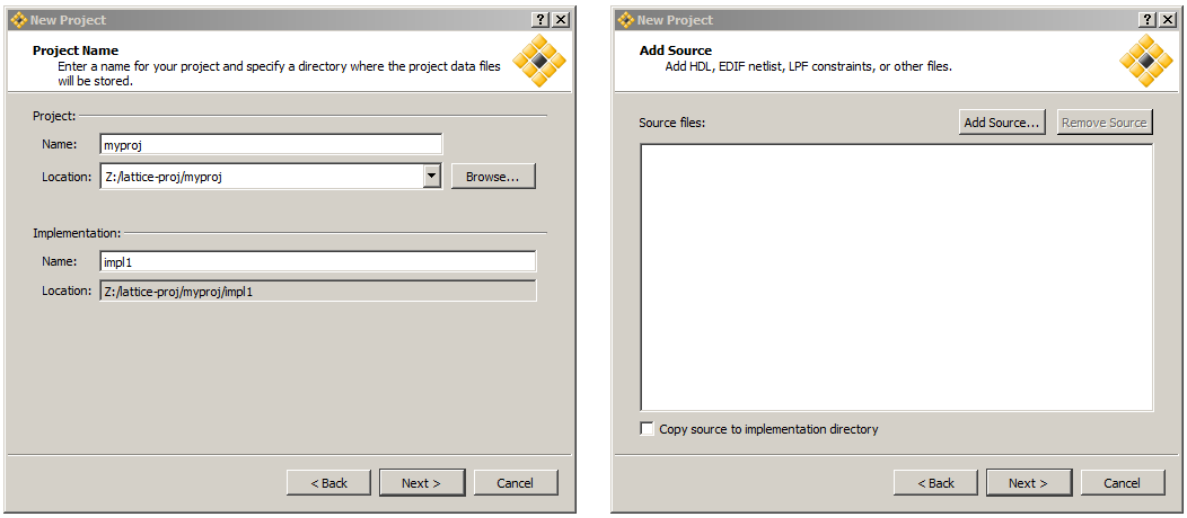
После прошивки ПЛИС мы можем воспользоваться Reveal Analyzer – это отдельная программа, которая позволяет нам следть за работой анализатора на ПЛИС. Через её интерфейс мы можем просмотреть то, что было записано в память логического анализатора, а также принудительно запустить его, если триггер не сработал. Считанные данные можно сохранить в один из распространённых форматов: csv, wvf, итд.

Достоинства такого способа отладки в том, что он не требует изменений платы устройства и не требует дополнительного оборудования. Всё синтезируется на уже имеющейся ПЛИС. При этом мы можем отслеживать состояние любого регистра или сигнала в нашем дизайне. Недостатки такого подхода в том, что при размещении логического анализатора на ПЛИС мы теряем часть логических элементов, на которых могло бы быть размещено само устройство. К тому же, объём памяти анализатора ограничен логической ёмкостью ПЛИС, поэтому анализатор способен захватывать только короткие промежутки времени (обычно несколько тысяч значений).

2.4. Процесс создания проекта

Запустив Lattice Diamond, мы видим стартовый экран среды разработки. Здесь мы можем открыть один из проектов, которые были открыты последними или создать новый проект.

Новый проект мы создаём при помощи мастера новых проектов (New Project Wizard). Задаётся имя проекта и каталог, в котором он будет расположен рис.2.3а. В следующем окне можно выбрать файлы исходного кода, которые будут дорбавлены в проект при его создании рис.2.3б. Можно добавить файлы в проект и позже.

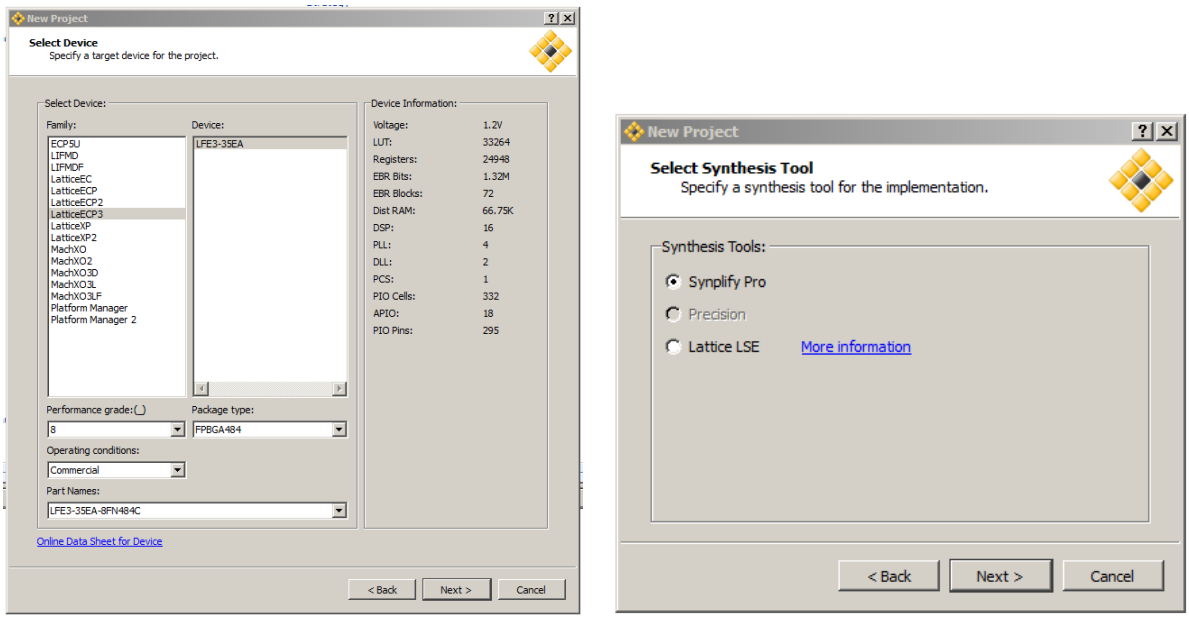


(а) Имя и расположение проекта

(б) Добавление в проект файлов с исходным кодом

Рис. 2.3

В следующем окне мы выбираем аппаратную платформу, для которой будет собираться этот проект. Выберем чип Lattice ECP3 (LFE3-35EA-8FN484C) Рис.2.4а. Наконец, нам необходимо выбрать средство синтеза кода на VHDL/Verilog: Lattice LSE или Synplify Pro. В данном случае пока выберем Synplify Pro Рис.2.4б.



(а) Выбор модели чипа ПЛИС (б) Выбор средства синтеза кода

Рис. 2.4

У главного окна Lattice Diamond можно выделить следующие поля:

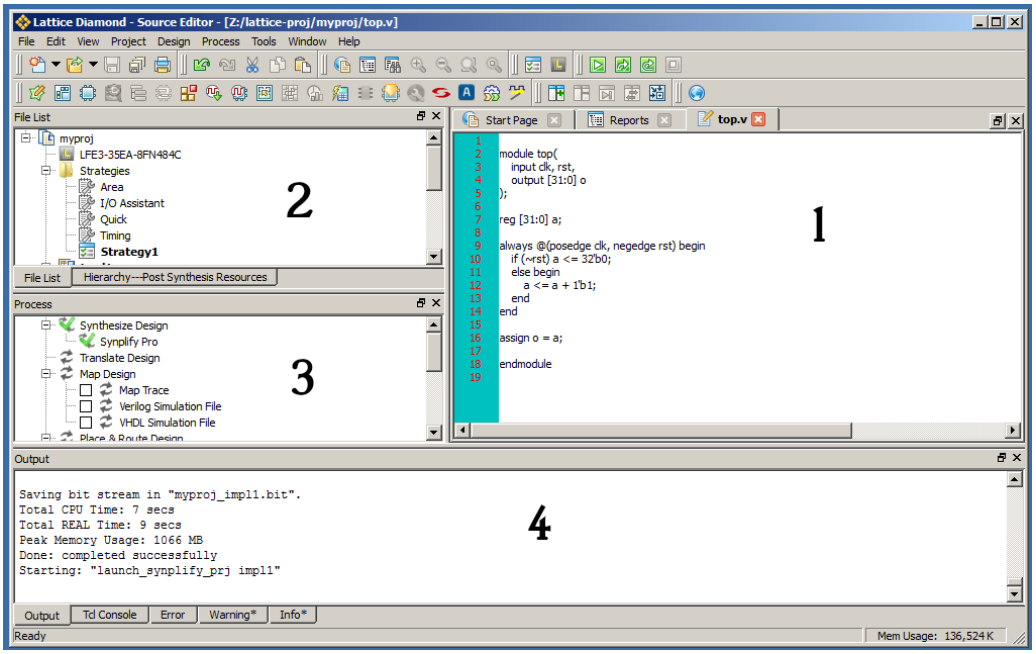


Рис. 2.5. Главное окно Lattice Diamond

1 Рабочая область – здесь виден редактор кода, отчёты компиляции и открываемые инструменты (например логический анализатор)

2 Здесь видны файлы проекта и структура синтезированного устройства

3 Управление маршрутом проектирования устройства. Здесь отмечаются этапы, которые необходимо выполнить при сборке проекта

4 Область вывода информации САПР и TCL-консоль, предназначенная для управления проектом посредством команд и скриптов на языке TCL (Tool Command Language)

Ниже приведён пример списка файлов в области №2 окна САПР:

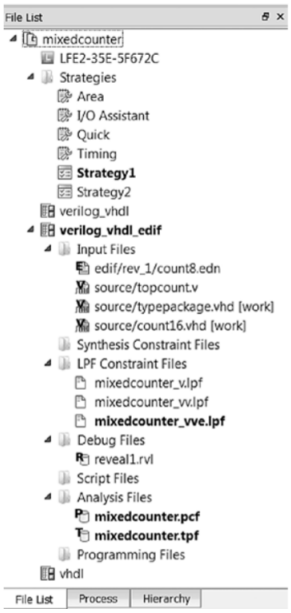


Рис. 2.6. Пример списка файлов Diamond [4]

Среди них можно различить следующие типы файлов:

– Исходные HDL-файлы (VHDL, Verilog)

– Файлы списка цепей (Netlist) в формате EDIF

– Файлы настроек синтеза проекта

– Файлы настроек размещения и разводки проекта (LPF)

– Файлы средства отладки Reveal Analyzer (RVI, RVL)

– Скриптовые файлы для моделирования

– Файлы для анализа энергопотребления

– Файлы для временного анализа

– Конфигурационные файлы программатора

Окно Process View отображает этапы маршрута проектирования на системном уровне и статус их выполнения. Map Design – это упаковка логических функций на элементы логической библиотеки ПЛИС.
Place & Route Design – это размещение и разводка проекта на кристалле.
В разделе Export Files можно указать файлы, которые должны быть получены в результате работы. Например Bitstream и JEDEC – это два различных формата файлов прошивки ПЛИС.

2.5. Сводная таблица входов и выходов ПЛИС Spreadsheet View

Окно Spreadsheet View – это мощное средство задания параметров проекта. В окне Spreadsheet View имеется несколько вкладок:

– Port Assignments и Pin Assignments – назначение соответствия сигналов ввода/вывода конкретным выводам ПЛИС и работа с атрибутами этих сигналов.

– Clock Resource – управление источниками тактирования в проекте. Этот инструмент позволяет назначить тактовый домен первичного и вторичных тактовых сигналов, назначить конкретные пути и регионы кристалла, по которым тактовый сигнал может идти.

– Route Priority – Назначение приоритета разводки определённых цепей.

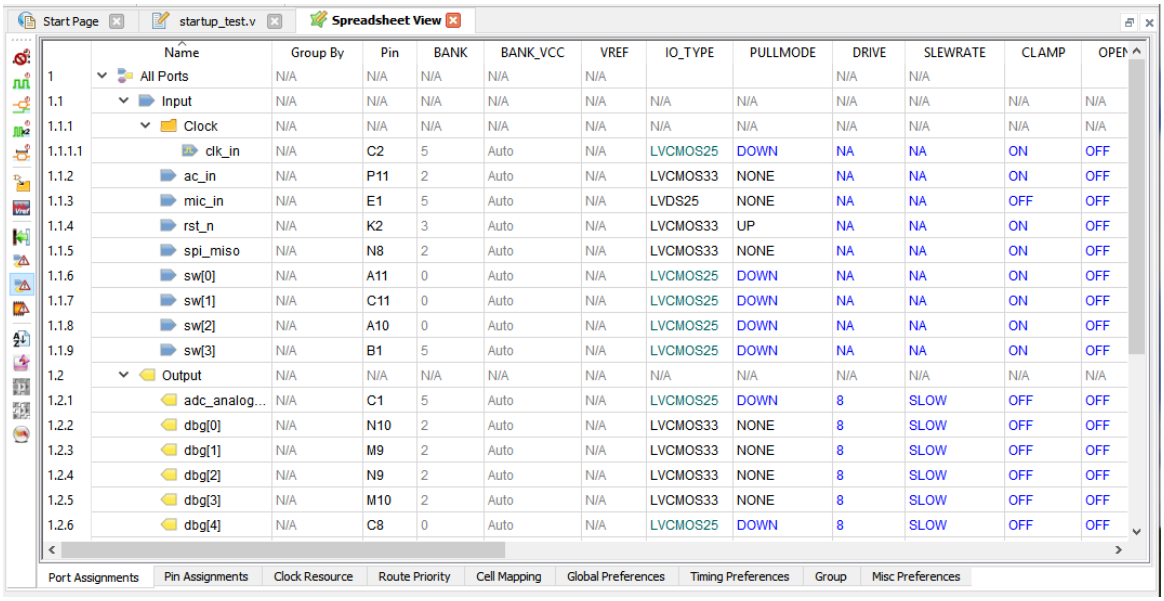
– Cell Mapping – управление размещением триггеров в ячейках ввода/вывода.

– Global Preferences – глобальные настройки, относящиеся ко всему проекту. Например, напряжение питания ядра.

– Timing Preferences – Временные настройки проекта

– Group – Управление группами ячеек, портов, а также пользовательскими группами

– Misc Preferences – позволяет определить конкретные области, в которых будет размещена логика или источники напряжения



	Name	Group By	Pin	BANK	BANK_VCC	VREF	IO_TYPE	PULLMODE	DRIVE	SLEWRATE	CLAMP	OPEN
1	All Ports	N/A	N/A	N/A	N/A	N/A			N/A	N/A		
1.1	Input	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1.1.1	Clock	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1.1.1.1	clk_in	N/A	C2	5	Auto	N/A	LVCMOS25	DOWN	NA	NA	ON	OFF
1.1.2	ac_in	N/A	P11	2	Auto	N/A	LVCMOS33	NONE	NA	NA	ON	OFF
1.1.3	mic_in	N/A	E1	5	Auto	N/A	LVDS25	NONE	NA	NA	OFF	OFF
1.1.4	rst_n	N/A	K2	3	Auto	N/A	LVCMOS33	UP	NA	NA	ON	OFF
1.1.5	spi_miso	N/A	N8	2	Auto	N/A	LVCMOS33	NONE	NA	NA	ON	OFF
1.1.6	sw[0]	N/A	A11	0	Auto	N/A	LVCMOS25	DOWN	NA	NA	ON	OFF
1.1.7	sw[1]	N/A	C11	0	Auto	N/A	LVCMOS25	DOWN	NA	NA	ON	OFF
1.1.8	sw[2]	N/A	A10	0	Auto	N/A	LVCMOS25	DOWN	NA	NA	ON	OFF
1.1.9	sw[3]	N/A	B1	5	Auto	N/A	LVCMOS25	DOWN	NA	NA	ON	OFF
1.2	Output	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1.2.1	adc_analog...	N/A	C1	5	Auto	N/A	LVCMOS25	DOWN	8	SLOW	OFF	OFF
1.2.2	dbg[0]	N/A	N10	2	Auto	N/A	LVCMOS33	NONE	8	SLOW	OFF	OFF
1.2.3	dbg[1]	N/A	M9	2	Auto	N/A	LVCMOS33	NONE	8	SLOW	OFF	OFF
1.2.4	dbg[2]	N/A	N9	2	Auto	N/A	LVCMOS33	NONE	8	SLOW	OFF	OFF
1.2.5	dbg[3]	N/A	M10	2	Auto	N/A	LVCMOS33	NONE	8	SLOW	OFF	OFF
1.2.6	dbg[4]	N/A	C8	0	Auto	N/A	LVCMOS25	DOWN	8	SLOW	OFF	OFF

Рис. 2.7. Окно Spreadsheet View – назначение выводов ПЛИС

2.6. Симулятор Active HDL

Active HDL от Aldec – это среда для симуляции и отладки проектов на ПЛИС. В этой среде можно создать проект для симуляции одного из тестовых файлов (testbench) в проекте. Тестовый файл – это файл на verilog, в котором объявлен экземпляр тестируемого модуля, а также в нём описаны тестовые воздействия на входы этого модуля.

Ниже показан вид главного окна симулятора:

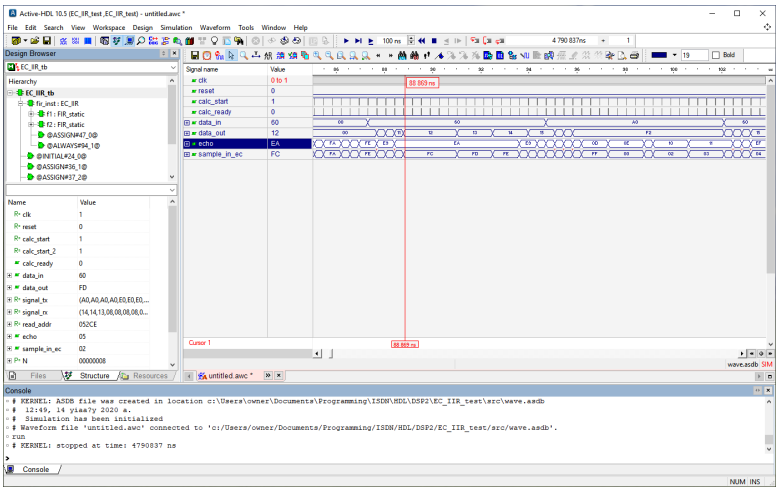
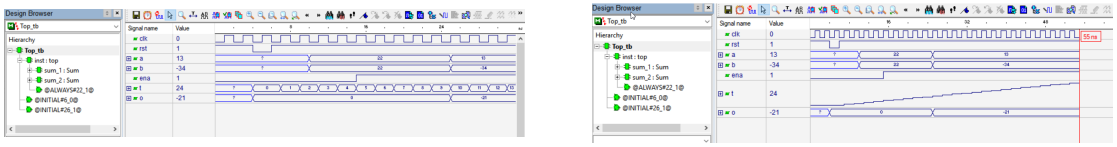


Рис. 2.8. Главное окно Active HDL

После трансляции кода на Verilog в окне Structure появляется структура проекта в виде дерева объектов. Любой сигнал из любого тестируемого модуля можно добавить в окно временных диаграм (выбрать ”add to waveform“ в контекстном меню).

После повторного запуска симуляции эти временные диаграммы сигналов будут отображены. В настройках есть возможность менять способ отображения чисел. Можно изменить систему счисления, знак и способ кодирования знака, также есть поддержка чисел с фиксированной точкой и с плавающей точкой. Эта функция очень полезна при проектировании модулей, выполняющих арифметические операции.

Помимо стандартного числового отображения сигналов есть возможность выбрать аналоговое представление и настроить диапазон отображения. В этом случае сигнал выводится в виде графика.



(а) (б)

Рис. 2.9. Отображение сигналов в цифровой (a) и в аналоговой (b) форме в Active HDL

2.7. Вывод

Так как в качестве целевой платформы были выбраны ПЛИС компании Lattice Semiconductor, то выбирать пришлось между САПР компании Lattice Semiconductor. Diamond – это проприетарный САПР, предназначенный для работы с ПЛИС линеек ECP3, MachXO2, и другими линейками. Этот САПР поддерживается Lattice Semiconductor уже больше двадцати лет. В состав САПР входит множество фирменных утилит, полезных для анализа результатов синтеза и управления проектом, а также у Diamond есть интеграция с синтезатором Synplify Pro и с симулятором ActiveHDL – программами сторонних производителей.

Lattice Diamond позволит нам синтезировать описания на Verilog под конкретные платформы, а также проанализировать результаты синтеза.

3. АППАРАТНАЯ ПЛАТФОРМА

В качестве целевой платформы были выбраны ПЛИС Lattice ECP3 (LFE3-35EA-8FN484C) и ПЛИС MachXO2 (LCMXO2-4000HC-csBGA132). Эти ПЛИС были доступны мне во время обучения. Это одни из самых популярных чипов от Lattice. Каждая из платформ обладает своими особенностями, которые влияют на подход к синтезу конечного устройства.

Lattice ECP3 [11]

- Чип – LFE3-35EA-8FN484C
- Напряжение питания ядра – 1.2v
- 33000 логических ячеек
- 1296 Kibit EBR RAM
- 68 Kibit distributed RAM
- Только внешний SPI flash
- 4 PLL, 2 DLL
- 64 умножителя 18x18 разрядов

Lattice MachXO2 [12]

- Чип – LCMXO2-4000HC-csBGA132
- Напряжение питания ядра – 3.3v
- 4320 логических ячеек
- 92 Kibit EBR RAM
- 34 Kibit distributed RAM
- 96 Kibit пользовательской Flash-памяти

– 2 PLL

– Нет аппаратных умножителей

ECP3 – это более дорогая и быстрая ПЛИС. У ECP3 больше логическая ёмкость, больше памяти и есть аппаратные умножители. MachXO2 же наоборот не имеет умножителей и обладает более скромными характеристиками.
Если мы будем синтезировать цифровые фильтры на этих ПЛИС, то на MachXO2 нам будет доступно меньше логических элементов, следовательно лучше синтезировать фильтры меньшего порядка, БИХ-фильтры использовать лучше, чем КИХ, так как они могут иметь меньший порядок при схожих характеристиках выходного сигнала.
Также в MachXO2 отсутствуют аппаратные умножители, поэтому необходимо использовать минимальное количество блоков умножения в дизайне. Мы можем либо синтезировать умножитель на логических ячейках в виде комбинаторной схемы, либо использовать отдельный блок, реализующий умножение алгоритмически за несколько тактов. Алгоритмический умножитель может занимать намного меньше логических ячеек, особенно при большой разрядности множителей. Также на обеих ПЛИС есть блоки RAM-памяти, которую можно задействовать в дизайне. Следовательно, мы можем рассмотреть два варианта: синтезировать память на регистрах, либо использовать блоки RAM.
Если сравнивать быстродействие этих двух ПЛИС, то логические устройства чипа ECP3 могут работать на частотах в среднем в 2 раза больше, чем аналогичные устройства у MachXO2. Таблица производительности различных логических устройств приведена ниже: (данные из datasheet machXO2 [12] и datasheet ECP3 [11])

Таблица 3.1
Сводная таблица производительности логических функций Lattice ECP3 и MachXO2

	Lattice ECP3	Lattice MaxhXO2
16-битный декодер	4.7 ns	8.9 ns
Мультиплексор 4:1	4.1 ns	7.5 ns
Мультиплексор 16:1	4.7 ns	8.3 ns

Таблица 3.2
Сводная таблица производительности устройств на регистрах Lattice ECP3 и MachXO2

	Lattice ECP3	Lattice MaxhXO2
Мультиплексор 16:1	500 MHz	412 MHz
16-битный сумматор	500 MHz	297 MHz
16-битный счётчик	500 MHz	324 MHz
64-битный счётчик	320 MHz	161 MHz

4. ОБЗОР СУЩЕСТВУЮЩИХ ИНСТРУМЕНТОВ ОПИСАНИЯ АППАРАТНЫХ СРЕДСТВ

Сейчас в сфере разработки описаний на ПЛИС большинство разработчиков использует два языка – это VHDL и Verilog. Оба этих языка были созданы в восьмидесятых годах двадцатого века, зарекомендовали себя как надёжные средства разработки дизайнов, которые поддерживаются всеми САПР для разработки устройств на ПЛИС и в представлении не нуждаются.
Но разработчики жалуются на недостаточную гибкость этих языков, которая мешает масштабировать проекты и создавать сложные описания, состоящие из сотен модулей. В связи с этим возникли новые средства разработки RTL-дизайнов.

– *Chisel*

  Chisel (Чизел) [14] – это свободный язык описания аппаратных средств, основанный на Scala и разработаный в университете Беркли (UC Berkeley). Scala – это язык, объединяющий объектно-ориентированную и функциональную концепции программирования, в языке Scala каждое значение – объект, а каждая операция – вызов метода. Chisel является набором абстракций в языке Scala, которые позволяют описывать поведение аппаратных модулей. Этот язык нацелен на масштабируемость и повторное использование кода. В процессе работы из Chisel способен генерировать код на Verilog, который уже поддерживается САПР от производителей ПЛИС. Также Chisel позволяет использовать в дизайне готовые модули на Verilog/FirRTL.

  Фактически Chisel - это просто библиотека в языке Scala, содержащая специализированные классы, и когда мы пишем на Chisel, то на самом деле мы пишем на Scala программу, которая строит граф аппаратного устройства.

Разработчики Chisel выбрали Scala в качестве основы, так как в Scala имеется много возможностей, полезных для кодогенерации, а также потому что этот язык был задуман как основа для создания узкоориентированных языков программирования.
Для работы с Chisel можно использовать два бэк-енда: Firrtl и Treadle.

а) *Firrtl*

Firrtl – это название языка промежуточного представления (IR) электронных схем, а также название интерпретатора этого языка, позволяющего симулировать работу схем. В первую очередь Chisel генерирует именно описание устройства в формате Firrtl, а затем уже он может быть сконвертирован в Verilog или другие описания.
По-умолчанию для синтеза в Chisel используют бэк-енд Firrtl

б) *Treadle*

Treadle – второй альтернативный бэк-енд, работающий с промежуточным представлением Firrtl. Это пока экспериментальный инструмент, позволяющий симулировать работу модулей. В отличие от интерпретатора Firrtl, Treadle не используют для генерации кода на других языках описания аппаратуры, а только в качестве симулятора.

– *Clash*

Clash (Клэш) – это свободный язык описания аппаратных средств, основанный на Haskell и разработанный QBayLogic. Haskell, в отличие от Scala, полностью функциональный язык, и в Clash для описания аппаратных блоков используются только функциональные средства. Clash способен генерировать файлы на SystemVerilog, Verilog и VHDL.
Каждый модуль в Clash описывается функцией из входящего сигнала в исходящий (сигналы могут быть составными). Модуль верхнего уровня должен иметь имя topEntity. Идея использовать функциональный язык

программирования для описания аппаратных устройств основывается на том, что комбинаторные схемы могут быть легко представлены в виде логических функций, а функциональные языки созданы для работы со всевозможными функциями. Каждая функция в Clash описывает какой-то компонент, а каждое использование функции – экземпляр этого компонента в схеме. Обратные связи при таком подходе могут описываться в виде рекурсивных функций.

– *SystemC и HLS*

  Это библиотека на C++, позволяющая моделировать аппаратные устройства на различном уровне абстракции и использующая средства метапрограммирования на C++. SystemC поддерживает как цифровые (дискретные) системы, так и аналоговые. Целью разработчиков SystemC было создание альтертативы SystemVerilog, которая к тому же позволяла бы разрабатывать алгоритмы и писать встроенное ПО. C++ – это очень популярный, гибкий и универсальный язык, на котором уже существует множество готовых библиотек и алгоритмов.

  Для отладки простых описаний можно воспользоваться отладчиком GDB, который поддерживается многими IDE C++, а если его возможностей недостаточно, то есть ряд коммерческих отладочных средств, ориентированных именно на SystemC.

  Для синтеза кода на SystemC используются пакеты HLS (High-level synthesis). HLS способен синтезировать не только описание на SystemC, но и алгоритмы на C++ без привязки к аппаратуре. Лучшими средствами синтеза SystemC на рынке являются “Stratus” от Cadence и “Catapult C” от Calypto/Mentor Graphics. Современные HLS для оптимизации алгоритмов используют средства llvm.

  В целом SystemC позволяет синтезировать аппаратную реализацию алгоритмов на C++, не перекладывая их на специализированные языки описания аппаратуры, что может значительно ускорить разработку и помочь

продукту быстрее выйти на рынок. Однако эффективность синтезаторов HLS подвергается критике, так как в этой среде разработчик создаёт не описание на уровне регистровых передач, как в рассмотренных до этого языках, а работает на более высоком уровне алгоритмов C++.

– *BlueSpec*

  BlueSpec – это язык описания аппаратуры а также его компилятор, симулятор и набор дополнительных инструментов. Этот язык, как и Clash, сделан на базе функционального языка Haskell. При этом у него существует два различных синтаксиса и два компилятора для них соответственно. BH – классический синтаксис, основанный на Haskell, BSV – синтаксис, подобный SystemVerilog.

  BlueSpec 20 лет был закрытым коммерческим стандартом и только совсем недавно, 1 февраля 2020 года, его исходный код стал открытым. Теперь он распространяется под лицензией BSD.

4.1. Обоснование выбора языка Chisel для разработки цифровых фильтров

При выборе инструмента разработки я всерьёз рассматривал три варианта: Chisel, Clash и SystemC. SystemC было решено не использовать, так как многие инструменты для работы с SystemC являются проприетарными, к тому же неизвестно, насколько эффективно будут синтезированы устройства на SystemC, а цифровые фильтры желательно сделать как можно компактнее.

Мой выбор остановился на языке Chisel, так как этот инструмент является полностью открытым, поддерживается институтом Беркли и сообществом разработчиков. Сейчас Chisel активно развивается и совершенствуется, многие элементы языка были добавлены недавно и всё ещё являются экспериментальными. Популярности Chisel также поспособствовало то, что на нём была разработана реализация процессорного ядра Risc-V от университета Беркли – Risc-V Rocket Chip.

В языке Clash используется подход к проектированию, похожий на подход в языке Chisel, только вместо объектов в нём используются функции. Clash менее популярен, чем Chisel, к тому же, по моей субъективной оценке, полностью функциональный подход к проектированию является более сложным, так что от использования Clash было решено отказаться.

Chisel позволяет использовать возможности языка Scala (объектно-ориентированные и функциональные средства), но при этом разработчик на Chisel может описывать устройство на уровне регистровых передач.

5. ЯЗЫК ОПИСАНИЯ АППАРАТНЫХ СРЕДСТВ CHISEL

5.1. Представление данных

Все типы данных в Chisel, как и в Scala, являются объектами.

Самыми частоиспользуемыми типами данных в Chisel являются SInt (signed int), UInt (unsigned int) и Bool (1 бит – “истина” или “ложь”). Типы SInt и UInt унаследованы от универсального типа Bits – простой коллекции битов, они могут иметь любую длину, заданную явно или вычисленную автоматически при синтезе описания.

Помимо цифровых типов данных Chisel также поддерживает тип Analog – аналог типа inout в Verilog. Однако, в этой работе он нам не потребуется.

Целые числа в Chisel устроены таким образом, что могут быть получены из обыкновенных Int языка Scala. При этом разрядность результата можно либо указать явно, либо не указывать, и тогда будет автоматически использована минимально возможная разрядность [14] С.9.

```
22.U  // Преобразование в UInt
-22.S // Преобразование в Sint
22.U(6.W)  // Преобразование в UInt, разрядность задана явно
-22.S(6.W) // Преобразование в Sint, разрядность задана явно
true.B  // Преобразование в Bool
false.B //
```

Листинг 5.1: Объявление целых чисел

Также в Chisel существует возможность передавать в конструктор целочисленных типов строки в двоичном, восьмеричном и шестнадцатеричном форматах. Из строки может быть получено только беззнаковое число:

```
"h3AB".U  // 16-ричный формат
"o572".U  // 8-ричный формат
"b1100".U // двоичный формат
```

Листинг 5.2: Объявление целых чисел при помощи строк

При помощи строк мы можем объявлять беззнаковые числа любой разрядности, не ограничиваясь 32-битным форматом Int. Проблемы возникают, когда

необходимо объявлять целые знаковые числа большой длины, так как в виде строк можно задать только беззнаковые числа. Но эту проблему можно решить, используя встроенный тип Scala BigInt, конструкторы числовых типов в Chisel принимают BigInt, поэтому с его помощью мы можем объявлять целые числа теоретически бесконечной длины.

```
BigInt("-1024").S  // Преобразование BigInt в знаковое целое
BigInt("100000").U // Преобразование BigInt в беззнаковое целое
```

Листинг 5.3: Объявление целых чисел при помощи BigInt

Для обозначения разрядности данных используется специальный тип данных Width, поэтому мы преобразовываем количество разрядов в Width перед тем, как передавать его в конструктор объекта данных. делается это в формате <число>.W . Пример в листинге 5.1.

В Chisel мы можем оперировать не только одиночными значениями SInt, UInt, итд., но и коллекциями значений, такими как Seq и Vec [14] С.15. Коллекции, как и любой другой объект можно передать в качестве параметра модуля. Пример использования коллекций в листинге 6.1. Класс Vec может иметь аппаратное представление и используется для в описании модулей, а остальные коллекции используются для передачи параметров и предварительной обработки информации.

Существует также специальный класс Bundle – его роль схожа с ролью struct в языке Си. Bundle может объединять несколько именованных значений разной длины и разных типов. Классы Vec и Bundle, как и многие другие в Chisel, унаследованы от класса Data, а следовательно их можно представить в виде массива (вектора) битов, имея доступ к каждому биту отдельно.

Bundle чаще всего используется для описания интерфейса ввода/вывода модуля (подробнее в разделе 5.4)

```
val io = IO( new Bundle {
  val ena  = Input(Bool())
  val din  = Input(UInt(8.W))
  val dout = Output(UInt(16.W))
})
```

Листинг 5.4: Пример использования Bundle в качестве интерфейса модуля

По-умолчанию объекты в Chisel являются Wire (асинхронными сигналами). Чтобы синтезировать регистры, необходимо явно это указать при помощи Reg(), RegInit(), RegNext(). В качестве Reg можно объявить любой сигнал Data, в том числе и Vec (вектор объектов). Пример использования регистров в листингах 5.5 и 6.1. При работе с регистровыми передачами необходимо помнить, что передача данных по-умолчанию будет происходить только по фронту тактового сигнала. [14] С.14. В Verilog и SystemVerilog для регистровых передач используется оператор => , здесь же регистровые передачи организуются автоматически для любых объектов типа Reg.

Существует два оператора присоединения в Chisel: := и <>. Оператор := – это одностороннее подключение выхода ко входу, а <> – это двухстороннее соединение. Оба оператора побитово подключают друг к другу два объекта в дизайне.

```
// Регистровая задержка сигнала a
val b = RegNext(a)
// Объявление 32-битного регистра и его значения по сбросу 0
val x = RegInit(0.U(32.W))
// Регистровая передача - запись результата в x
x := a ^ b
```

Листинг 5.5: Пример использования регистров в Chisel

5.2. Комбинаторные схемы

Как и в Verilog, в Chisel есть возможность оперировать регистрами (Reg) и сигналами (Wire). Chisel поддерживает логические и побитовые операции [14] С.11 приблизительно в том же виде, что и в Verilog или языке Си. Например:

```
val c = a + b  // Сложение
val d = a & b  // Побитовое "и"
val e = a && b // Логическое "и"
```

Листинг 5.6: Примеры операторов Chisel

Но существуют некоторые отличия, в том числе и из-за того, что операторы Chisel должны вписываться в экосистему Scala. Например, операторы "равно"и "не равно":

```
val sw = Wire(UInt(8.W))

when (a === 50) {
  sw := 12.U
} .otherwise {
  sw := 88.U
}

when (a =/= 50) {
  sw := 12.U
} .otherwise {
  sw := 88.U
}
```

Листинг 5.7: Примеры операторов Chisel

Такая длинная форма записи “равно” и “не равно” была выбрана специально для сохранения совместимости с операторами Scala “==” и “!=”.

В листинге 5.11 представлена конструкция “when”, которая часто используется для условного ветвления в Chisel. Но “when” удобнее использовать, когда мы синхронно записываем значение в регистр. Когда же нам требуется создать комбинаторную схему с двухвходовым мультиплексором (условное ветвление), то чаще используется конструкция “Mux(c, x, y)”. Это аналог тернарного оператора в Verilog, где “c” – это условие.

Следует отметить, что в Chisel присутствуют особые операторы сложения и вычитания, которые позволяют автоматически избежать переполнения разрядной сетки.

```
val z = x +  y // Сложение БЕЗ увеличения разрядности
val z = x +% y // Сложение БЕЗ увеличения разрядности
val z = x +& y // Сложение С увеличением разрядности
// С вычитанием аналогично
```

Листинг 5.8: Примеры операторов Chisel

“+” “-” или “+%” “-%” – это равноценные операторы, при использовании которых увеличения количества разрядов не происходит. В этом случае количество разрядов результата равно: $w(z) = max(w(x), w(y))$

“+&” и “-&” – это операторы, при использовании которых количество разрядов

результата равно: $w(z) = max(w(x), w(y)) + 1$

5.3. Использование функций Scala в Chisel

Так как Chisel является надстройкой над Scala, то мы можем свободно объявлять и использовать функции/методы для достижения требуемого результата. Это могут быть как обыкновенные функции, так и лямбда-функции (анонимные) [6] С.40, [14] С.73

Функции можно использовать для описания повторяющихся логических конструкций в Chisel, а также для улучшения параметризуемости модулей.

```
// Функция, не возвращающая значений
def function_name(parameter: type_of_parameter, ...) {
  // Code
}
// Функция, возвращающая значение
def function_name(parameter: type_of_parameter, ...) : return_type = {
  // Code
}
// Функция, возвращающая значение
// Тип возвращаемого значения не указывается явно
def function_name(parameter: type_of_parameter, ...)= {
  // Code
}
// Лямбда-функция
(parameter: type_of_parameter, ...) => {Code}
```

Листинг 5.9: Синтаксис объявления функций Scala

Приведём несколько примеров их использования:

```
// Пример функции, вычисляющей разрядность наибольшего значения в списке
def clog2(d: Seq[Int]): Int = { (log(max(a.map(_.abs))) / log(2)).floor.
    ↪ asInt }
// Пример функции, вычисляющей логическое выражение
def example(a: Bool, b: Bool, c: Bool): Bool = (a & b) ^ c
```

Листинг 5.10: Примеры использования функций в Chisel

Так как в Scala функция – это объект, то это позволяет нам хранить функцию как переменную, передавать функцию в качестве аргумента и возвращать функцию

из другой функции. Такой подход можно использовать при параметризации объектов Chisel, мы можем передавать в качестве параметров не просто значения, а целые алгоритмы и логические конструкции.

5.4. Работа с интерфейсами ввода/вывода

Интерфейс модуля в Chisel описывается при помощи метода IO, принимающего объект типа Data. В нём должны быть обозначены направления всех сигналов (Input(), Output() или Analog() – двунаправленный сигнал). Сейчас в Chisel 3 есть экспериментальная возможность создавать несколько IO у одного модуля. Традиционно интерфейс модуля описывается при помощи пользовательского Bundle, содержащего необходимые нам сигналы. Можно объявлять в качестве IO не только Bundle, но Bundle позволяет объединить все сигналы интерфейса в один объект, не занимая при этом пространство имён внутри самого модуля. Объектно-ориентированный подход позволяет нам объявить набор стандартных интерфейсов, а потом использовать их в своих модулях, или же наследовать от них другие интерфейсы [6] С.9.
В качестве интерфейсов в Chisel запрещено использовать регистры. Это может показаться неудобным, так как разработчику приходится объявлять регистры отдельно от интерфейса в то время, как в Verilog мы можем сразу объёвить выход модуля как регистр. Но это было сделано разработчиками Chisel специально, чтобы сохранить “чистоту” интерфейсных объектов.
Особого внимания заслуживает пакет “chisel3.util”. В нём содержатся в основном примитивы для организации интерфейсов между модулями, рассмотрим некоторые из них:

- DecoupledIO – это стандартный FIFO-интерфейс, оперирующий сигналами valid, ready и data [6] С.31, [9].

- ValidIO – Интерфейс, работающий аналогично DecoupledIO, только без сигнала ready.

- Queue – Модуль, позволяющий организовать очередь, имеющую входной

и выходной FIFO-интерфейсы DecoupledIO

– Pipe – Модуль, создающий линию задержки входного сигнала на заданное количество периодов clk.

– Counter – Модуль параметризуемого аппаратного счётчика

– Arbiter – арбитр интерфейсов, позволяющий из нескольких источников мультиплексировать сигналы в один принимающий интерфейс.

Такие сущности можно быстро описать и вручную, однако наличие готовых примитивов снижает вероятность допустить ошибку в коде.
Также для удобства HDL-разработчиков в CHisel есть функция Flipped, которая преобразует все входы интерфейса в выходы, а выходы наоборот во входы. Так мы можем один раз описать интерфейс, а потом создать из этого класса master и slave интерфейсы, применив к одному из них функцию Flipped.
Также Chisel имеет экспериментальную возможность использовать внешние модули, написанные на verilog. Для этого используются специальные абстракции ExtModule и BlackBox, при помощи которых мы описываем интерфейс внешних модулей, сигналы тактирования и сброса. Можно заметить, что в описании на Chisel, в отличие от Verilog, отсутствуют в явном виде сигналы Clock и Reset. Это особенность языка, которая по-умолчанию добавляет сигналы Clock и Reset ко всем созданным модулям, тем самым упрощая программный код. Чаще всего для модуля достаточно тактирования общим Clock сигналом и обыкновенного синхронного сброса.
Если разработчику требуется использовать несколько тактовых сигналов, то это возможно сделать при помощи специальной конструкции withClock().
Также иногда может потребоваться использование асинхронного сброса или других сигналов сброса Reset. В ранних версиях Chisel не было возможности создать регистры с асинхронным сбросом средствами языка и приходилось редактировать код на Verilog. Сейчас асинхронный сброс стало возможно сделать при помощи специального типа модуля RawModule – это модуль, в который по-умолчанию не заведены сигналы Clock и Reset, поэтому мы управляем ими

вручную.

5.5. Сравнение Chisel и Verilog

Chisel удобно сравнивать с Verilog, так как код на Verilog может генерироваться из кода на Chisel, а также потому что Verilog очень популярен среди разработчиков аппаратуры на ПЛИС.

Verilog – это полноценный язык описания аппаратуры, который был создан ещё в 1983-1984 годах. Он поддерживается множеством программ: программами синтеза от проивводителей ПЛИС: Intel, Xilinx, Lattice, итд., средствами синтеза от сторонних производителей, например Synplify Pro, а также симуляторами, например ModelSim или Icarus Verilog. То, что Verilog – самостоятельный язык – и его достоинство, и недостаток, если сравнивать Verilog с Chisel.

Chisel нельзя назвать полноценным языком описания аппаратуры, так как это скорее набор абстракций в языке Scala, которые описывают аппаратные блоки устройства. Это приводит к тому, что в Chisel есть спорные языковые конструкции, которые нельзя упростить, так как Chisel должен соответствовать синтаксису Scala.

Например, условное выражение в Chisel описывается не так коротко, как в Verilog, так как является объектом в среде языка Scala. К тому же языковые конструкции Chisel не должны конфликтовать с сущностями Scala.

```
when (условие) {
  // здесь код
}

when (условие) {
  // здесь код
} .otherwise {
  // здесь код
}

when (условие) {
  // здесь код
} .elsewhen (условие) {
  // здесь код
} .otherwise {
```

```
16    // здесь код
}
```

Листинг 5.11: Условное выражение в Chisel

```
if (условие) begin
  // здесь код
end

if (условие) begin
  // здесь код
end else begin
  // здесь код
end

if (условие) begin
  // здесь код
end else if () begin
  // здесь код
end else begin
  // здесь код
end
```

Листинг 5.12: Условное выражение в Verilog

За счёт того, что Verilog – самостоятельный язык программирования, его разработчики вольны использовать любые синтаксические конструкции, делая его ориентированным именно на описание аппаратных блоков. Но с другой стороны, разрабатывая описание на Chisel, мы можем использовать все возможности языка Scala, который является гораздо более “мощным” и современным, чем Verilog.

Например, рассмотим процесс объявления регистра в этих двух языках:

```
  val bar = RegInit(0.U(32.W))
```

Листинг 5.13: Объявление регистра в Chisel

Здесь мы объявили 32-битный регистр bar, который сбрасывается в значение 0. Аналогичный регистр на Verilog объявляется так:

```
  reg [31:0] bar;
  always @(posedge clk) begin
    if (rst) bar <= 32'b0;
```

```
end
```

Листинг 5.14: Объявление регистра в Verilog

В описании модулей на Chisel по-умолчанию не используются сигналы Clock и Reset в отличие от Verilog, где взаимодействие с ними всегда описывается явно. Однако, Chisel позволяет задавать их при необходимости. Язык Chisel за счёт использования средств Scala позволяет разработчику параметризировать модули так, как в Verilog это было бы невозможно сделать. Благодаря объектно-ориентированному подходу можно передать модулю в качестве параметра любой объект Scala, в том числе и коллекцию объектов, и объекты, тип которых унаследован от типа аргумента.

Помимо Verilog также существует SystemVerilog – логическое продолжение Verilog с добавлением новых возможностей, упрощающих процесс разработки, например, многомерные входы/выхода модулей или новые типы, отражающие суть разных видов сигналов, или расширение Generate-макросов. К тому же в SystemVerilog появился объектно-ориентированный подход к описанию тестовых окружений, а также специальные конструкции для упрощения тестирования и улучшения качества тестирования. Но стандарты SystemVerilog должны быть обратно-совместимы с кодом Verilog, что накладывает на разработчиков этого языка ряд ограничений.

Несмотря на новые возможности SysyemVerilog, разработчикам кода на Verilog и SystemVerilog часто приходится прибегать к использованию сторонних генераторов (например скриптов на Python или Perl), чтобы впоследствии сгенерировать необходимый код на Verilog.

В итоге можно сказать, что Chisel даёт разработчику новые возможности, но не является универсальным средством проектирования. Для описаний небольших размеров лучше подходит Verilog, а для более сложных устройств – Chisel.

5.6. Преимущества Chisel для разработки цифровых фильтров

Цифровые фильтры – широко распространённое устройство как на ПЛИС, так и в программных реализациях. Зачастую в составе одного устройства может быть

много разных фильтров, поэтому целесообразно создавать одно параметризуемое описание, которое в дальнейшем позволит на его основе создать фильтры разного порядка и с разными весовыми коэффициентами. В языке Chisel есть ряд особенностей, делающий разработку фильтров более удобной, чем на Verilog. Специфика разработки описаний на Chisel такова, что на конечный результат не влияет сложность самой программы на Scala, а только то, сколько объектов Chisel в процессе было создано.

Как было упомянуто выше, Chisel позволяет параметризовать модули лучше, чем Verilog, SystemVerilog или VHDL. Для классов языка Chisel, как и для всего кода на языке Scala, справедливы концепции ООП: *наследование, инкапсуляция, полиморфизм*. Благодаря этому мы можем создавать одни модули и интерфейсы на основе других модулей и интерфейсов, таким образом избавляясь от повторяющегося кода.

Для адаптивных фильтров легче всего использовать одну структуру фильтра с различными адаптивными функциями. В Chisel можно легко менять адаптивную функцию внутри модуля средствами функционального программирования.

Цифровые фильтры на ПЛИС – это во многом повторяющаяся структура, которая может быть сгенерирована автоматически на основе набора параметров (веса, разрядность данных, порядок фильтра, и т.д.).

Цифровые фильтры могут использоваться как в устройствах с общей шиной, так и без неё. Широкое распространение имеет стандарт шины Wishbone, используемый во многих открытых аппаратных устройствах. Если фильтр работает не на шине, то чаще всего используются DecoupledIO [9] или ValidIO интерфейсы. При помощи средств Chisel удобно будет создать базовый модуль самого фильтра и несколько интерфейсов для него, как отдельные классы.

6. РАЗРАБОТКА ЦИФРОВЫХ ФИЛЬТРОВ НА ЯЗЫКЕ CHISEL

6.1. Разработка простого КИХ-фильтра

Опишем простой КИХ-фильтр на Chisel. В качестве входного и выходного интерфейсов используем DecoupledIO. Интерфейс модуля описывается в виде Bundle из двух DecoupledIO [9]. Для того, чтобы определить входной и выходной интерфейс используется конструкция Flipped(). Данный фильтр принимает в качестве параметров разрядность данных и последовательность весовых коэффициентов. Из этой последовательности автоматически вычисляется порядок фильтра.

```
package filter

import chisel3._
import chisel3.util._

class FIR_filter (
  XW: Int = 8, // Разрядность данных
  weights: Seq[BigInt] = Seq(1) // Весовые коэффициенты
) extends Module {
  // Интерфейс модуля включает два
  // FIFO-интерфейса: входной и выходной
  val io = IO( new Bundle {
    val din  = Flipped(Decoupled(SInt(XW.W)))
    val dout = Decoupled(SInt())
  })

  // Регистры линии задержки входных данных
  val delays = RegInit(VecInit(Seq.fill(weights.length-1)(0.S(XW.W))))

  // Объявление соединений между регистрами линии задержки
  when (io.din.valid & io.din.ready) {
    // В первый регистр поступают входные данные
    delays(0) := io.din.deq()
    delays.indices.tail.foreach { i => delays(i) := delays(i-1) }
  }

  // Регистровый выход
```

```
28    val dout = RegNext(
        io.din.bits * weights(0).S + (delays, weights).zipped.map(_*_.S).reduce(
      ↪ _+_)
30    )
      io.dout.bits <> dout
32
      // Сигналы FIFO
34    io.din.ready := io.dout.ready
      io.dout.valid := io.din.valid
36  }
```

Листинг 6.1: Описание простого КИХ-фильтра на Chisel

В коде мы создаём вектор из регистров (строка 18). Это регистры линии задержки для входных данных. Длина этого вектора на 1 меньше порядка фильтра (см. рис.1.1а).

Затем, начиная со строки 21, описывается логика линии задержки. При чтении входных данных они попадают в первый регистр линии задержки и происходит “продвижение” данных по линии задержки.

В конце мы умножаем данные из линии задержки на весовые коэффициенты и суммируем все полученные произведения. Делается это при помощи инструментов функционального программирования map() и reduce(). Результат мы пропускаем через регистр, чтобы избежать “гонки” сигналов на выходе.

В целом работа этого модуля, как и работа КИХ-фильтра, описывается следующим уравнением:

$$y[n] = \sum_{k=0}^{N-1} x[N-k]w[k] \tag{15}$$

Есть возможность сделать описание ещё короче, используя функциональные средства Scala, как это показано [6] на С.46 и в примере ниже:

```
def delays[T <: Data](x: T, n: Int): List[T] =
  if (n <= 1) List(x) else x :: delays(Reg(next = x), n-1)

def FIR[T <: Num](hs: Seq[T], x: T): T =
  (hs, delays(x, hs.length)).zipped.map( _ * _ ).reduce( _ + _ )

class TstFIR extends Module {
  val io = new Bundle{ val x  = SInt(INPUT, 8); val y = SInt(OUTPUT, 8) }
  val h  = Array(SInt(1), SInt(2), SInt(4))
  io.y  := FIR(h, io.x)
}
```

Рис. 6.1. Описание простого КИХ-фильтра на Chisel

Но быстродействия конечного устройства это не увеличит. Читаемость кода же, напротив, становится хуже.

На веб-сайте проекта Chisel тоже есть пример КИХ-фильтра, однако не параметризуемого и неадаптивного: [7].

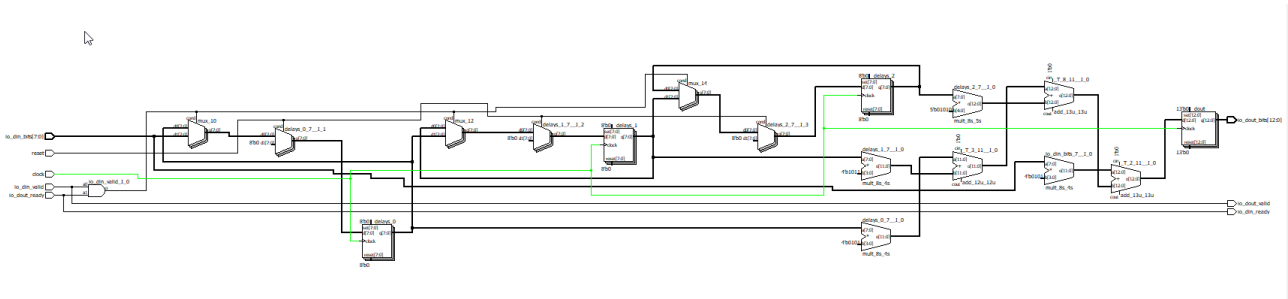


Рис. 6.2. Структура, синтезированная при помощи Lattice LSE

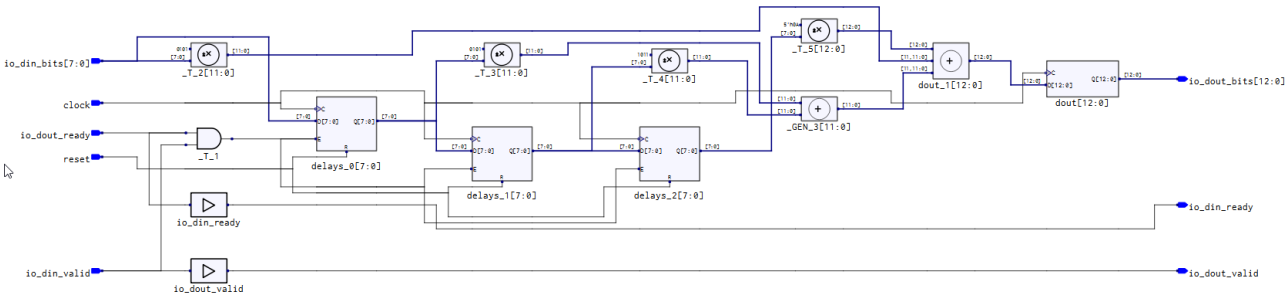


Рис. 6.3. Структура, синтезированная при помощи Synplify Pro

Эти две структуры отображены в двух разных средствах просмотра (Lattice Netlist Viewer и Synplify Pro), так как использовались разные синтезаторы. В обоих случаях синтезировалось три 8-битных регистра в линии задержки, один выходной регистр и четыре умножителя.

Следует отметить, что Synplify Pro в данном случае синтезирует регистры с

сигналом Enable и асинхронным сбросом, а LSE синтезирует регистры с синхронным сбросом и без enable.

Такой вариант фильтра вычисляет каждое новое значение за один такт, он прост в реализации, однако при этом занимает много аппаратных ресурсов ПЛИС. Такой вариант имеет смысл использовать только в задачах, где требуется максимальное быстродействие.

Если быстродействие фильтра не очень критично, то легче прибегнуть к использованию алгоритмической раелизации фильтров.

6.2. Разработка алгоритмического умножителя для фильтров

При алгоритмической реализации цифровых фильтров в дизайне обычно присутствуют один или два умножителя. Эти умножители могут задействовать аппаратные умножители на ПЛИС, если они доступны. Если аппаратных умножителей на ПЛИС нет, например как у MachXO2 (LCMXO2-4000HC-csBGA132), то можно рассматривать вариант реализации алгоритмического умножителя, так как умножители, синтезированные в виде комбинаторной схемы, занимают много аппаратных ресурсов.

Исходный код реализованного на Chisel алгоритмического умножителя представлен в приложении 1.

Он реализует алгоритм умножения “старшими разрядами вперёд” [5] С.57, при этом поддерживает числа со знаком. Два входных множителя обозначаются как “a” и “b”. Время работы умножителя зависит от разрядности “b”.

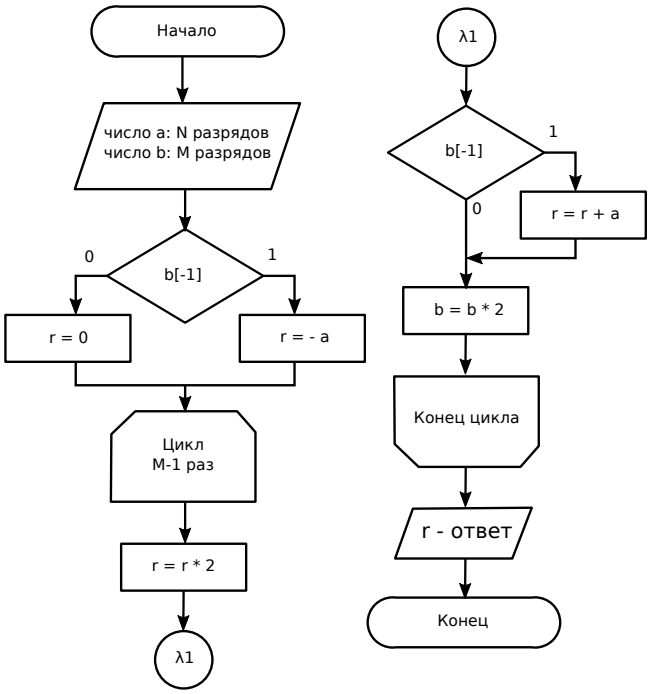


Рис. 6.4. Схема алгоритма работы умножителя

На схеме “b[-1]” обозначает старший разряд числа b.

Модуль имеет 3 FIFO-интерфейса. 2 входных для двух операндов и 1 выходной для результата. При этом модуль имеет независимую логику управления входными интерфейсами и выходным интерфейсом, так что умножитель может производить умножение с новыми операндами в то время, как старый результат ещё не считан из выходного интерфейса.

Синтезированный модуль на Verilog был протестирован в стиуляторе ActiveHDL на наборе тестовых значений разной величины и разного знака. Ниже показана временная диаграмма в симуляторе ActiveHDL, демонстрирующая процесс вычисления одного произведения (показаны только наиболее важные сигналы внутри модуля).

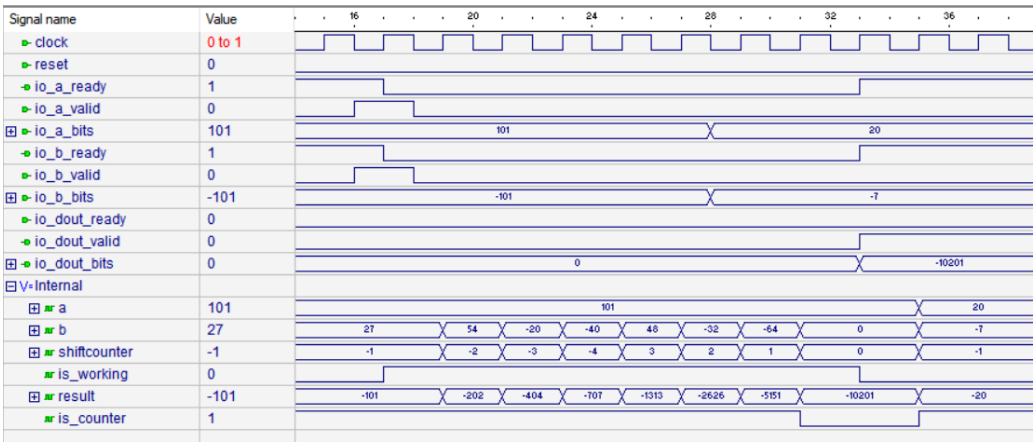


Рис. 6.5. Пример работы умножителя

В этом примере перемножаются числа a: 101 и b: -101. В результате получается -10201 (сигнал io_dout_bits)

На диаграмме хорошо видно счётчик разрядов (shiftcounter), сдвиговый регистр (b), аккумулирующий регистр (result), а также сигналы FIFO-интерфейса.

Ниже представлены результаты синтеза на ПЛИС:

Таблица 6.1

Результаты синтеза на ПЛИС Lattice ECP3

	Последовательный	Параллельный
Lattice LSE	178 LUT4, 199 регистров	0 регистров, 1 LUT4, 4 MULT18x18
Synplify Pro	216 LUT4, 199 регистров	0 регистров, 1 LUT4, 4 MULT18x18

Таблица 6.2

Результаты синтеза на ПЛИС Lattice MachXO2

	Последовательный	Параллельный
Lattice LSE	181 LUT4, 199 регистров	1089 LUT4, 0 регистров
Synplify Pro	217 LUT4, 292 регистра	1163 LUT4, 0 регистров

Последовательным умножителем назван алгоритмический, а параллельным – умножитель, синтезированный автоматически на ПЛИС (оператор умножения *).

По результатам видно, что на ПЛИС Lattice ECP3, на которой есть аппаратные умножители, параллельный умножитель синтезировался с меньшими аппаратными затратами, задействовав 4 блока умножителей 18x18. На ПЛИС MachXO2

результат противоположный, потому что параллельный умножитель там синтезируется на логических ячейках LUT4. Последовательный (алгоритмический) умножитель занимает в 5 раз меньше ресурсов ПЛИС.

6.3. Разработка алгоритмического БИХ-фильтра

Было решено реализовать БИХ-фильтр, так как его мы можем использовать и как КИХ, и как БИХ-фильтр. При помощи параметров модуля мы сможем задать весовые коэффициенты обратных связей равными нулю, тем самым превратив его в КИХ-фильтр. При этом логика обработки нулевых весов будет исключена из дизайна при упрощении кода firrtl.

Этот модуль реализует схему целочисленного БИХ-фильтра типа 2 (см. рис.1.4). Каждая ступень фильтра обрабатывается отдельно, следовательно мы используем одновременно два умножителя. Умножители в данном дизайне алгоритмические, но их можно легко заменить на аппаратные, тем самым ускорив работу фильтра.

```
  XW: Int = 8, // Разрядность данных
2 WW: Int = 8, // Разрядность весов
  w_numer: Seq[BigInt] = Seq(1), // Стартовые весовые коэффициенты
4 w_denom: Seq[BigInt] = Seq(1)  //
```

Листинг 6.2: Параметры модуля

При создании объекта фильтра мы должны указать разрядность выходных данных и весов. w_numer и w_denom – это два массива значений, которыми инициализируются весовые коэффициенты фильтра. Если фильтр неадаптивный, то они останутся такими. Если же фильтр адаптивный, то весовые коэффициенты будут постоянно подстраиваться, но и стартовые значения тоже нужны. Не всегда фильтр способен адаптироваться правильно, если при старте его веса равны нулям.

Веса задаются в формате BigInt, так что, несмотря на использование целочисленных операций, теоретически можно синтезировать фильтр любой точности. Массивы стартовых весовых коэффициентов должны быть одинаковой длины,

иначе во время синтеза программа вернёт исключение. По длине массивов автоматически определяется порядок фильтра.

```
val io = IO( new Bundle {
  val din  = Flipped(Decoupled(SInt(XW.W)))
  val dout = Decoupled(SInt(outW.W))
  val init = Input(Bool())
})
```

Листинг 6.3: Интерфейс модуля

Интерфейс фильтра состоит из двух FIFO-интерфейсов: входного и выходного [9], а также флага init, который запускает механизм сброса фильтра к стартовым значениям.

В фильтре, выполняющем вычисления параллельно, линия задержки представляет собой последовательность регистров, соединённых в цепочку. В алгоритмической реализации фильтра можно пойти иным путём и хранить значения из линии задержки в ячейках RAM-памяти. Для работы с памятью в Chisel есть специальная абстракция Mem или SyncReadMem. Если память будет синтезирована на банках регистров ПЛИС, то это позволит нам задействовать меньше ячеек ПЛИС (не всегда возможно синтезировать память на блоках RAM). Если же нам потребуется освободить ячейки RAM, то всегда есть возможность включить синтез памяти на регистрах в настройках синтеза.

Сдвиг значений в линии задержки в алгоритмического фильтра производится последовательно. Для этого создан дополнительный регистр “x_swap”, в котором временно хранится сдвигаемое значение.

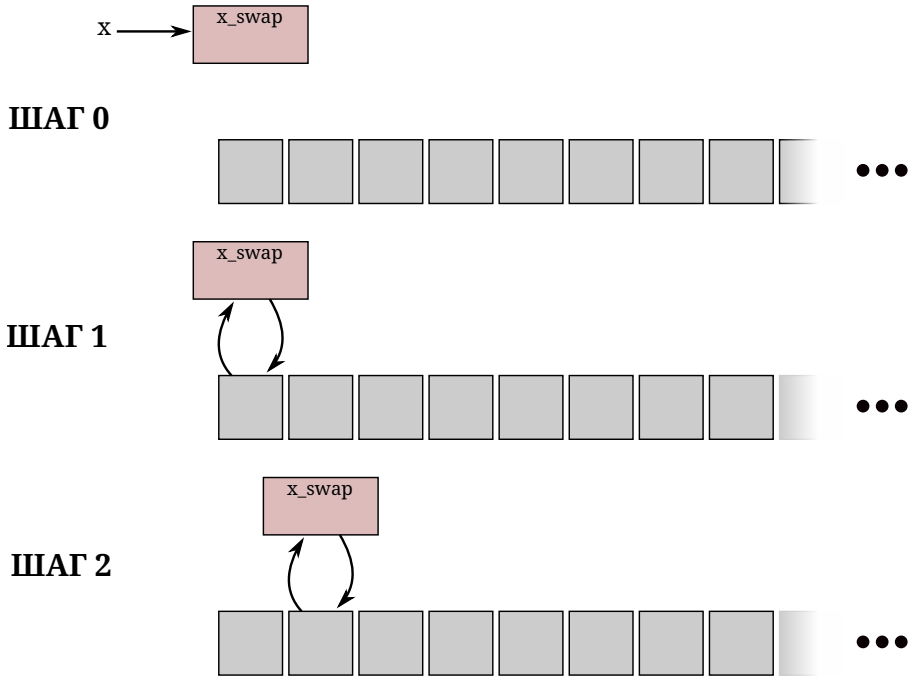


Рис. 6.6. Процесс сдвига значений в алгоритмическом фильтре

На рисунке приблизительно показан процесс сдвига значений. Перед началом сдвига новое входное значение помещается в регистр “x_swap”, а после n шагов происходит сдвиг. На каждом шаге значение в регистре “x_swap” меняется местами с текущим значением в RAM памяти. цепочка блоков – это RAM, к каждой ячейке необходимо обращаться по адресу.
Исходный код этого фильтра представлен в приложении 2.

6.4. Разработка алгоритмического адаптивного БИХ-фильтра

Адаптивный фильтр отличается от неадаптивного только наличием адаптивной функции. Так как Scala поддерживает функциональное программирование, то любая функция – это объект с определённым типом и функцию можно передать в модуль в качестве аргумента. Это нам и нужно.

```
class IIR_adaptive(
  XW: Int = 8, // Разрядность данных
  WW: Int = 8, // Разрядность весов
  w_numer: Seq[BigInt] = Seq(1), // Стартовые весовые коэффициенты
  w_denom: Seq[BigInt] = Seq(1), //
  calc_w: ( // Функция рассчёта новых весовых коэффициентов
    Int, Int, Int, // xw, ww, order
```

```
    Bool, // din valid signal
    Bool, // dout ready signal
    UInt, // pos
    SInt, // din bits
    SInt, // dout bits
    Mem[SInt], // delay_1
    Mem[SInt], // delay_2
    SInt, // x_swap_1
    SInt, // x_swap_2
    Mem[SInt], // w_1
    Mem[SInt], // w_2
    Algomult, // mult_1
    Algomult, // mult_2
    SInt // External signal d
  ) => (SInt, SInt, Bool, Bool) // new_w_1, new_w_2, din_ready, dout_valid
) extends Module {
```

Листинг 6.4: Параметры модуля

Набор параметров модуля остаётся прежним за исключением последнего, это новый параметр – функция рассчёта новых весовых коэффициентов. Аргументами этой функции являются те объекты внутри фильтра, по которым можно рассчитать новые веса. Мы заранее не знаем, какой алгоритм будет использовать эта функция, так что передаём в функцию все объекты, которые могут быть полезны. Лишние параметры функции никак не скажутся на объёме итогового устройства.

Так как в Chisel память описывается объектом, то мы можем передать в качестве параметра память целиком (ссылку на объект памяти), а не каждое значение по отдельности. Аналогично мы поступаем с умножителями – в функцию передаётся ссылка на объект вместе с его входными и выходными состояниями.

Важным для адаптации является последний аргумент – сигнал “d” или желаемый сигнал, который был обозначен на рисунке 1.5. Мы заводим в модуль сигнал “d” извне и на его основе можем вычислять величину ошибки. Затем нам остаётся лишь вызывать эту функцию внутри модуля, сохраняя результат на каждом шаге. Поддержка функционального программирования позволяет создать на основе одного модуля несколько фильтров с различными адаптивными

функциями.

Рассчёт каждого весового коэффициента может потребовать нескольких тактов, особенно если используются алгоритмические умножители, так что функция также как и модуль имеет входной и выходной FIFO-интерфейсы. Пример реализации адаптивной функции представлен ниже:

```
def lms(
    XW: Int,
    WW: Int,
    order: Int,
    din_valid: Bool,
    dout_ready: Bool,
    pos: UInt,
    din: SInt,
    dout: SInt,
    delay_1: Mem[SInt],
    delay_2: Mem[SInt],
    x_swap_1: SInt,
    x_swap_2: SInt,
    w_1: Mem[SInt],
    w_2: Mem[SInt],
    mult_1: Algomult,
    mult_2: Algomult,
    d: SInt
  ): (SInt, SInt, Bool, Bool) = {
    val e = d - dout // error
    // Два умножителя для рассчёта двух весов за раз
    val mw_1 = Module(new Algomult(e.getWidth, din.getWidth))
    val mw_2 = Module(new Algomult(e.getWidth, dout.getWidth))
    // Подключение умножителей к сигналам FIFO и линиям данных
    mw_1.io.a.valid := din_valid
    mw_1.io.b.valid := din_valid
    mw_2.io.a.valid := din_valid
    mw_2.io.b.valid := din_valid
    mw_1.io.a.bits := e
    mw_1.io.b.bits := x_swap_1
    mw_2.io.a.bits := e
    mw_2.io.b.bits := x_swap_2

    val din_ready = mw_1.io.a.ready && mw_1.io.b.ready && mw_2.io.a.ready &&
    ↪  mw_2.io.b.ready
    val dout_valid = mw_1.io.dout.valid && mw_2.io.dout.valid
```

```
    mw_1.io.dout.ready := dout_ready
    mw_2.io.dout.ready := dout_ready

    (
      w_1.read(pos) + (mw_1.io.dout.bits.asSInt << 16-7 ).head(WW).asSInt,
      w_2.read(pos) + (mw_2.io.dout.bits.asSInt << 27-7 ).head(WW).asSInt,
      din_ready, dout_valid
    ) // Return tuple
  }
```

Листинг 6.5: Пример адаптивной функции (LMS алгоритм)

Эта функция использует метод адаптации по алгоритму наименьших квадратов (LMS) [3] С.46. Так как она использует алгоритмические умножители, на вычисление результата требуется несколько тактов и дополнительно используются сигналы интерфейса FIFO valid/ready.

Так как алгоритму LMS (уравнение 14), как и многим другим адаптивным алгоритмам, требуется значение из линии задержки и выходное значение с одного и того же шага, то появляется необходимость в создании дополнительной задержки значений x (из линии задержки). В нашем случае дополнительные регистры создавать не пришлось, так как уже существует регистр “x_swap”, в котором оказываются значения “x” с предыдущего шага.

Ниже показана временная диаграмма в симуляторе ActiveHDL, демонстрирующая процесс обработки одного входного значения.

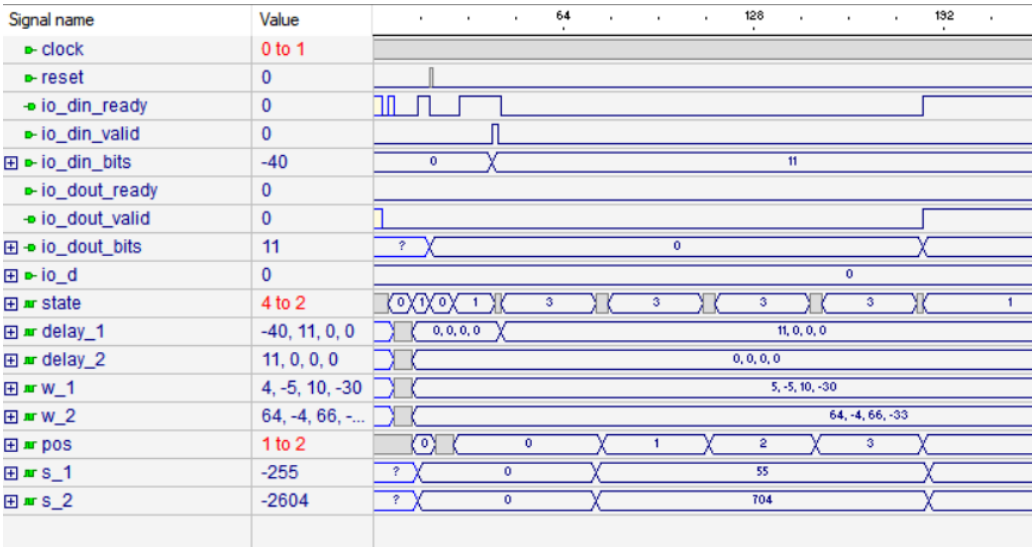


Рис. 6.7. Пример работы фильтра

На диаграмме хорошо видны: счётчик ступеней фильтра (pos), массивы весовых коэффициентов (w_1, w_2), линии задержки (delay_1, delay_2), аккумулирующие регистры (s_1, s_2) и сигналы FIFO-интерфейса. На диаграмме сложно продемонстрировать все сигналы, так как работа алгоритмического фильтра занимает много тактов и одни сигналы изменяются слишком быстро относительно других, чтобы изобразить их на одной диаграмме. Состояния конечного автомата (state) изменяются быстро, однако можно увидеть, что наибольшее время фильтр затрачивает на ожидание окончания работы умножителей (умножителей внутри фильтра и умножителей адаптивной функции). Работу фильтра можно ускорить, используя аппаратные умножители, однако это возможно не на всех ПЛИС. В таком режиме на обработку одного входного значения требуется около 100 тактов.

Правильность работы фильтра была проверена на наборе входных значений в симуляторе.

Ниже представлены результаты синтеза адаптивного фильтра разными средствами синтеза и на разных ПЛИС. Это фильтр 6 порядка с 8-битными входными данными, 16-битными весовыми коэффициентами, адаптируется он по алгоритму LMS из листинга 6.5:

Таблица 6.3

Результаты синтеза адаптивного фильтра

	MachXO2	ECP3
Lattice LSE	913 LUT4, 576 регистров	830 LUT4, 624 регистра
Synplify Pro	1345 LUT4, 1206 регистров	1340 LUT4, 1089 регистров

По результатам синтеза видно, что адаптивные фильтры на Lattice ECP3 и MachXO2 синтезируются приблизительно одинаково эффективно, так как конструкция логических ячеек LUT4 у этих ПЛИС одинаковая. При этом Lattice LSE синтезирует описание на меньшем количестве логических ячеек, чем Synplify Pro.

Также мы можем уменьшить аппаратные затраты и увеличить скорость адаптивного фильтра, синтезируя параллельные умножителя вместо последовательных.

Это показано в таблицах 6.1 и 6.1.

Полный исходный код модуля представлен в приложении 3.

Для проверки правильности работы фильтра создадим модель фильтра верхних частот 6 порядка в Matlab:

```
hpFilt = designfilt('highpassiir','FilterOrder', 6, ...
         'PassbandFrequency',700,'PassbandRipple',0.2, ...
         'SampleRate',1/ts);
[b,a] = tf(hpFilt);
```

Листинг 6.6: Создание модели БИХ-фильтра в Matlab

Здесь b и a – массивы полученных весовых коэффициентов фильтра в формате float. Эти же веса используются при синтезе фильтра.

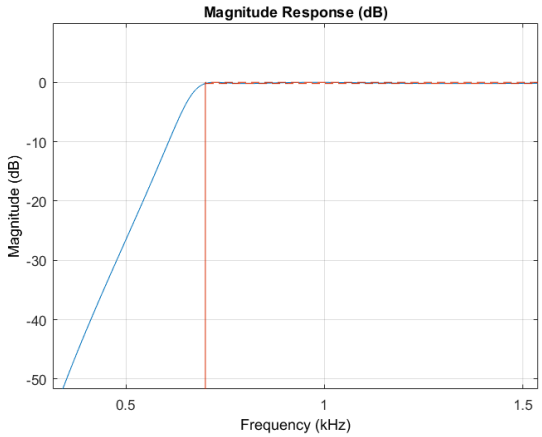
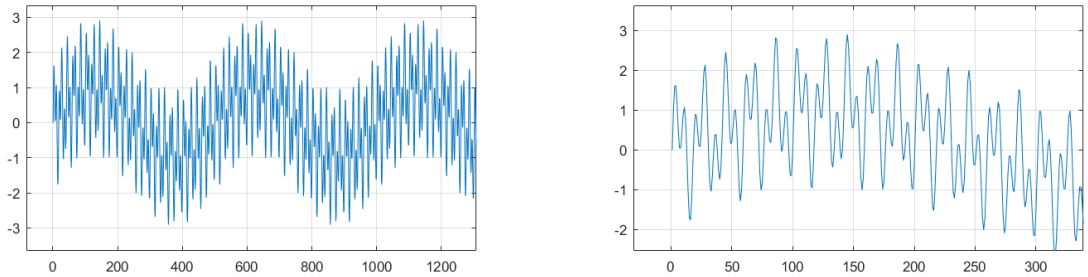


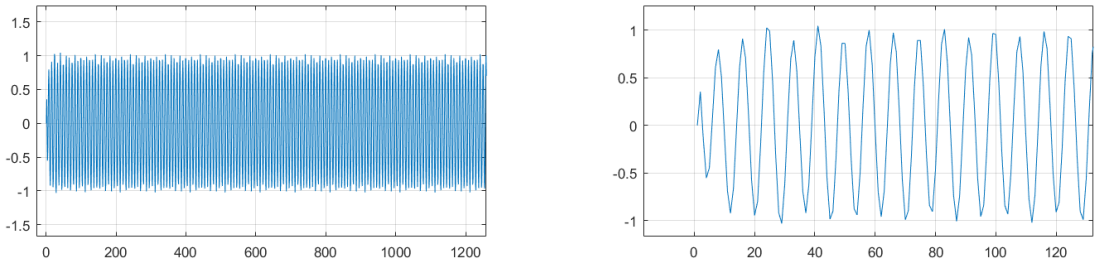
Рис. 6.8. АЧХ БИХ-фильтра в Matlab

В качестве входных значений фильтра используется последовательность, состоящая из суммы синусоид разной частоты.



(а) (б)

Рис. 6.9. Входные данные фильтра в Matlab



(а) (б)

Рис. 6.10. Выходные данные фильтра в Matlab

В Chisel были использованы те же веса, только для аппаратного фильтра с целочисленными весовыми коэффициентами все веса домножаются на 2^{20}.

```
val b = Seq(
    +0.353494866,
    -2.120969196,
    +5.302422989,
    -7.069897319,
    +5.302422989,
    -2.120969196,
    +0.353494866,
  ).map(e => BigInt((e*1048576).toLong)) // Перевод в целые числа. 1048576 =
   ↪  2^20
  val a = Seq(
    +1.000000000,
    -3.973726290,
    +6.860878999,
    -6.505073811,
    +3.575180738,
    -1.086720466,
    +0.149064122
  ).map(e => BigInt((e*1048576).toLong)) // Перевод в целые числа. 1048576 =
   ↪  2^20
chisel3.Driver.execute(args, () => new IIR_adaptive(12, 24, b, a, lms2))
```

Листинг 6.7: Конфигурирование неадаптивнгого фильтра верхних частот на Chisel

В этом листинге представлены весовые коэффициенты a и b, а также создание объекта IIR_adaptive с определёнными параметрами. Это модуль адаптивного фильтра, но адаптивный алгоритм мы отключили, использовав пустую адаптивную функцию.

Используя средства Scala, мы можем записать весовые коэффициенты в привычном формате с плавающей точкой, а затем преобразовать их в BigInt перед передачей в конструктор фильтра.

После этого входные данные из Matlab были выгружены в файл .dat для обработки в симуляторе. Для данных были использованы 12-битные значения, при этом у самих данных сохранена точность 8 знаков после запятой. Разрядность весов 24 бит.

В итоге в симуляторе можно наблюдать характеристику, схожую с характеристикой, полученной в Matlab.

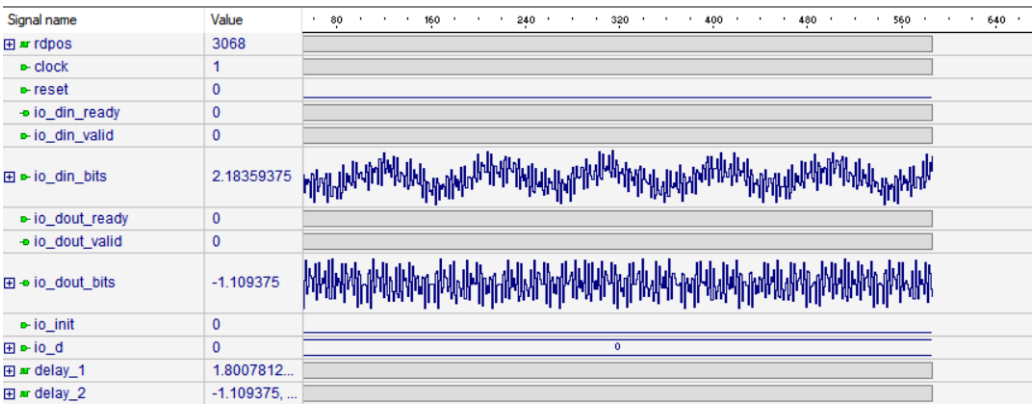


Рис. 6.11. Входной и выходной сигналы БИХ-фильтра в симуляторе ActiveHDL

Так как в фильтрах Matlab используются коэффициенты и данные с плавающей точкой, а в аппаратном фильтре – с фиксированной точкой, то появляется расхождение выходных значений, однако в начале работы фильтра выходные значения близки к полученным в Matlab (расхождение появляется в третьем десятичном знаке после запятой). Это позволяет судить о том, что алгоритм обработки данных в фильтре реализован правильно, но из-за погрешностей во входных данных и весах результаты получаются другими, а так как это БИХ-фильтр, погрешность со временем накапливается.

Тем не менее, со временем фильтр не сбивается и убирает низкочастотную составляющую сигнала.

Адаптивный алгоритм можно проверить, вычисляя ошибку как разность какого-то фиксированного числа и выхода фильтра.

```
val e = Mux(dout < 0.S, -1105392.S, 1105392.S) - dout
```

Листинг 6.8: Пример вычисления ошибки в адаптивной функции

В данном случае функция такова, что выходные значения фильтра должны стремиться к двум уровням: -1105392 и 1105392. Полный исходный код функции представлен в приложении к отчёту в файле Launcher.scala.

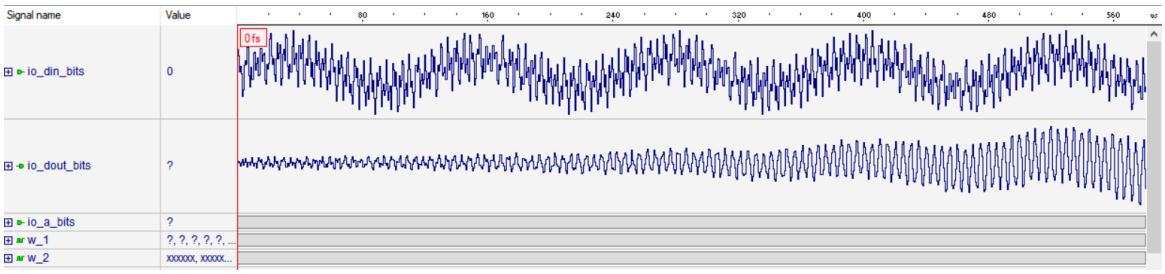


Рис. 6.12. Входной и выходной сигналы адаптивного БИХ-фильтра

В данном случае фильтр не адаптируется правильно, так как стартовые весовые коэффициенты остались прежними (это коэффициенты для ФВЧ). Можно наблюдать, как постепенно увеличивается амплитуда выходного сигнала, при этом весовые коэффициенты постепенно подстраиваются адаптивной функцией. По алгоритму LMS мы вычисляем небольшие приращения к текущему значению веса.

Ниже показано изменение трёх весовых коэффициентов во времени. Видно, что значения изменяются нелинейно, их изменение зависит от входных данных и от других весовых коэффициентов.

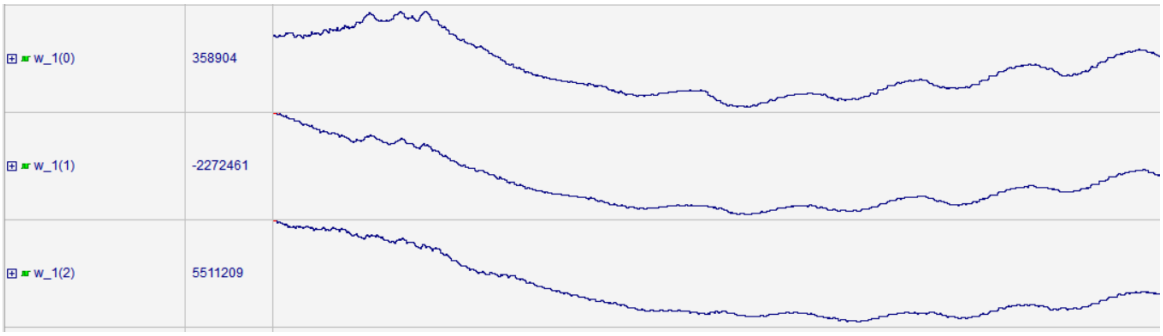


Рис. 6.13. Пример изменения весовых коэффициентов во времени

Для правильной адаптации фильтра необходимо соблюсти подходящие стартовые значения весов, величину приращений и функцию оценивания ошиб-

ки.

По результатам симуляции также видно, что адаптивный алгоритм использует значение из линии задержки с предыдущего шага (x_swap_1, x_swap_2), работа адаптивного алгоритма соответствует теории (уравнение 14).

ЗАКЛЮЧЕНИЕ

В этой работе я рассмотрел возможности синтеза цифровых фильтров при помощи высокоуровневых средств проектирования.

Был проведён анализ разных типов цифровых фильтров и особенности их аппаратной реализации. Проведён сравнительный анализ средств высокоуровневого синтеза устройств на ПЛИС, их сильные и слабые стороны. Для разработки описаний для этой работы я выбрал язык Chisel, так как этот язык сейчас набирает популярность, он позволяет использовать возможности Scala для параметризации модулей и автоматической генерации устройства, но при этом оставляет разработчику возможность контролировать работу устройства на уровне регистровых передач.

Рассмотрен процесс разработки проектов в среде Lattice Diamond и составные части этой среды. Для того, чтобы получить прошивку для ПЛИС, из кода на Chisel сначала был я генерировал код Verilog, который поддерживается САПР Lattice Diamond. Отчёты САПР были использованы для сравнения результатов синтеза фильтров. Также синтезированные модули были проанализированы в симуляторе Aldec ActiveHDL.

В результате я получил параметризуемые модули на Chisel: простой нерекурсивный фильтр, вычисляющий результат параллельно, алгоритмический БИХ-фильтр и алгоритмический адаптивный БИХ-фильтр.

В ходе работы обнаружено, что на ПЛИС, имеющих аппаратные умножители, использование параллельных умножителей (встроенных в язык операторов умножения *) позволяет задействовать меньше логических ячеек. С другой стороны, на ПЛИС без поддержки аппаратных умножителей наоборот, использование алгоритмических умножителей позволяет задействовать меньше логических ячеек.

Результаты работы синтезированных цифровых фильтров проверены в симуляторе, они совпадают с теоретическими значениями и результатами, полученными в Matlab.

В будущем я планирую усовершенствовать созданные описания: добавить воз-

можность переключаться между типами умножителей, параллельным и последовательным, а также завести больше сигналов извне в адаптивную функцию. Также можно создать набор готовых адаптивных функций, реализующих разные адаптивные алгоритмы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Бесекерский В.А., Попов Е.П.* Теория систем автоматического управления. — СПб : Профессия, 2003. — 752 с. — ISBN 5-93913-035-6.
2. *Бугров Владимир, Пройдаков Вадим, Артемьев Владимир*. Поисковые технологии проектирования целочисленных цифровых фильтров // Компоненты и технологии. — 2014. — № 6. — с. 124—130.
3. *Грант П.М., Коуэн К.Ф.Н.* Адаптивные фильтры. — Москва : Мир, 1988. — 392 с. — ISBN 5-03-000004-6.
4. *Д.А.Комолов, Р.Н.Золотухо, А.А.Антонов*. Руководство по использованию САПР Diamond компании Lattice / пер. с англ. ООО "ЭФО". — Санкт-Петербург : Медиа-группа Файнстрит, 2018. — 176 с. — ISBN 978-5-6040819-0-7.
5. *Зыков А.Г., Поляков В.И.* Арифметические основы ЭВМ. — Санкт-Петербург : Университет ИТМО, 2016. — 141 с.
6. *Bachrach J.*, *Izraelevitz A.*, *Koenig J.* Advanced Chisel Topics. — EECS UC Berkeley : EECS UC Berkeley, 2017. — 82 с.
7. Chisel/FIRRTL Hardware Compiler Framework. — URL: `https://www.chisel-lang.org/`.
8. CMOS Continuous-Time Adaptive Equalizers for High-Speed Serial Links / Chagoyen, Concepción Aldea and Gasca, Cecilia Gimeno and Pueyo, Santiago Celma and Celma, Santiago. — Zaragoza, Spain : Springer International Publishing, 22.09.2016. — 172 с. — ISBN 978-3-319-38485-6. — URL: `https://www.ebook.de/de/product/26482200/concepcion_aldea_chagoyen_cecilia_gimeno_gasca_santiago_celma_pueyo_santiago_celma_cmos_continuous_time_adaptive_equalizers_for_high_speed_serial_links.html`.
9. EECS150: Interfaces: “FIFO” (a.k.a. Ready/Valid). — UC Berkeley College of Engineering. — 5 с.

10. *Ka Fai Cedric Yiu,YaoLu,ChunHokHo, Wayne Luk, Jiaquan Huo, Sven Nordholm*. Reconfigurable FPGA-based switching path frequency-domain echo canceller withapplications to voice control device // Digital Signal Processing. — 2012. — c. 376—390.

11. *Lattice Semiconductor, inc*. LatticeECP3 Family Data Sheet DS1021. — вер. 02.8EA. 2015. — URL: `http://www.latticesemi.com/-/media/LatticeSemi/Documents/DataSheets/Lattice/LatticeECP3EAFamilyDataSheet.ashx`.

12. *Lattice Semiconductor, inc*. MachXO2 Family Data Sheet FPGA-DS-02056. — вер. 3.4. — 2019. — URL: `http://www.latticesemi.com/-/media/LatticeSemi/Documents/DataSheets/MachXO23/FPGA-DS-02056-3-4-MachXO2-Family-Data-Sheet.ashx`.

13. *Madisetti V. K.*, *Williams D. B.* The Digital Signal Processing Handbook. — Boca Raton, Florida : CRC Press, 1999. — 1776 c. — ISBN 0-8493-8572-5.

14. *Schoeberl M.* Digital Design with Chisel. — Second Edition. — USA : Kindle Direct Publishing, 2019. — 164 c. — ISBN 9781689336031.

15. *Stewart B.* Adaptive IIR Filtering. — Scotland, UK, 2006. — 68 p.

ПРИЛОЖЕНИЕ 1.

ИСХОДНЫЙ КОД АЛГОРИТМИЧЕСКОГО УМНОЖИТЕЛЯ НА CHISEL

```
package algomath

import chisel3._
import chisel3.util._
import connect_util.util.reg_connect

// Алгоритмический умножитель знаковых целых чисел старшими разрядами вперёд
class Algomult (
  aW: Int = 8,
  bW: Int = 8
) extends Module {
  // Интерфейс модуля
  val io = IO( new Bundle {
    val a = Flipped(Decoupled(SInt(aW.W)))
    val b = Flipped(Decoupled(SInt(bW.W)))
    val dout = Decoupled(SInt())
  })

  val result_w = aW + bW
  // Объект с выходными регистрами модуля
  object ioreg {
    val din_ready  = RegInit(true.B)
    val dout_valid = RegInit(false.B)
    val dout_bits  = RegInit(0.S(result_w.W))
  }
  // Подключение выходов модуля к регистрам
  io.a.ready    := ioreg.din_ready
  io.b.ready    := ioreg.din_ready
  io.dout.valid := ioreg.dout_valid
  io.dout.bits  := ioreg.dout_bits

  val a = RegInit(0.S(aW.W))
  val b = RegInit(0.S((bW-1).W))
  val shiftcounter = RegInit((bW-1).U)
  val is_working   = RegInit(false.B)
  val result       = RegInit(0.S(result_w.W))
```

```
37
   val dout_writeable: Bool = ~io.dout.valid | io.dout.ready
39 val reading_input:  Bool = io.a.valid & io.b.valid & ioreg.din_ready
   val is_counter = shiftcounter.orR;
41
   when (is_working) {
43   when (is_counter) {
       // Суммирование в общий аккумулятор result
45     result := (result << 1) + Mux(b.head(1).asBool, a, 0.S)
       b := b << 1
47     shiftcounter := shiftcounter - 1.U;
       ioreg.din_ready := false.B
49   } .otherwise {
       // Ожидание, пока освободится выходной регистр
51     // и запись результата в него
       ioreg.din_ready  := dout_writeable
53     is_working := !dout_writeable
       when (dout_writeable) { ioreg.dout_bits := result }
55   }
   } .otherwise {
57   // Ожидание начала работы
     is_working := reading_input
59   ioreg.din_ready := !reading_input
     a := io.a.bits
61   b := io.b.bits.tail(1).asSInt
     shiftcounter := b.getWidth.asUInt
63   result := Mux(io.b.bits.head(1).asBool, -io.a.bits, 0.S)
   }
65
   ioreg.dout_valid := (ioreg.dout_valid & ~io.dout.ready) | (is_working & !
    ↪ is_counter & dout_writeable)
67 }
```

Листинг П.1: Исходный код алгоритмического умножителя на Chisel

ПРИЛОЖЕНИЕ 2.

ИСХОДНЫЙ КОД АЛГОРИТМИЧЕСКОГО БИХ-ФИЛЬТРА НА CHISEL

```
package filter

import chisel3._
import chisel3.util._
import chisel3.experimental.ChiselEnum
import math.log

import algomath.Algomult

class IIR_static(
  XW: Int = 8, // Разрядность данных
  WW: Int = 8, // Разрядность весов
  w_numer: Seq[BigInt] = Seq(1), // Стартовые весовые коэффициенты
  w_denom: Seq[BigInt] = Seq(1)  //
) extends Module {

  def log2(x: BigInt): Int = (log(x.toDouble) / log(2)).toInt
  def clog2(x: BigInt): Int = (log(x.toDouble) / log(2)).ceil.toInt

  if (w_numer.length != w_denom.length) throw new IllegalArgumentException("
   ↪ Numerator and denominator should be the same length")

  val order = w_numer.length
  val log_a0 = log2(w_denom(0))
  val outW = XW + WW + order // Разрядность выходных значений

  val io = IO( new Bundle {
    val din  = Flipped(Decoupled(SInt(XW.W)))
    val dout = Decoupled(SInt(outW.W))
    val init = Input(Bool())
  })
  object ioreg {
    val din_ready  = RegInit(false.B)
    val dout_valid = RegInit(false.B)
    val dout_bits  = RegInit(0.S(outW.W))
    val step_start_strobe = RegInit(false.B)
```

```
    val read_vals_ready = RegInit(false.B)
  }
  // Подключение выходов модуля к регистрам
  io.din.ready  := ioreg.din_ready
  io.dout.valid := ioreg.dout_valid
  io.dout.bits  := ioreg.dout_bits

  val dout_writeable = ~io.dout.valid | io.dout.ready
  //val reading_input = io.din.valid & io.din.ready

  val delay_1 = Mem(order, SInt(XW.W))
  val delay_2 = Mem(order, SInt(outW.W))

  val pos      = RegInit(0.U(clog2(order).W))
  val y1       = RegInit(0.S(outW.W))
  val y2       = RegInit(0.S(outW.W))
  val x_swap_1 = RegInit(0.S(XW.W))
  val x_swap_2 = RegInit(0.S(outW.W))

  val step_is_last: Bool = pos === (order-1).U

  val mult1 = Module(new Algomult(WW, XW))
  val mult2 = Module(new Algomult(WW, outW))
  val w_numer_vec = VecInit(w_numer.map{ e => e.asSInt })
  val w_denom_vec = VecInit(w_denom.map{ e => e.asSInt })
  mult1.io.a.valid := ioreg.step_start_strobe
  mult1.io.b.valid := ioreg.step_start_strobe
  mult2.io.a.valid := ioreg.step_start_strobe
  mult2.io.b.valid := ioreg.step_start_strobe
  mult1.io.dout.ready := ioreg.read_vals_ready
  mult2.io.dout.ready := ioreg.read_vals_ready
  mult1.io.a.bits := w_numer_vec(pos)
  mult1.io.b.bits := delay_1.read(pos)
  mult2.io.a.bits := w_denom_vec(pos)
  mult2.io.b.bits := delay_2.read(pos)

  //==================[ STATE MACHINE ]==================//
  object states extends ChiselEnum {
    val INIT,
      WAIT,
        STEP_START,
      READ_VALS,
        STEP_END
```

```
79        = Value
  }
81  val state = RegInit(states.INIT)
  val state_next = Wire(chiselTypeOf(states.INIT))
83  when (io.init) {
    state := states.INIT
85  } .otherwise {
    state := state_next
87  }

89  { // CASE
    when (state === states.INIT) {
91    when (step_is_last) { state_next := states.WAIT }
      .otherwise          { state_next := states.INIT }
93  }

95  .elsewhen (state === states.WAIT) {
      when (io.din.valid & io.din.ready) { state_next := states.STEP_START }
97    .otherwise                         { state_next := states.WAIT }
    }
99
    .elsewhen (state === states.STEP_START) {
101   state_next := states.READ_VALS
    }
103
    .elsewhen (state === states.READ_VALS) {
105   when (
        mult1.io.dout.valid &&
107     mult2.io.dout.valid
      ) { state_next := states.STEP_END }
109   .otherwise { state_next := states.READ_VALS }
    }
111
    .elsewhen (state === states.STEP_END) {
113   when (step_is_last) {
        state_next := Mux(dout_writeable, states.WAIT, states.STEP_END)
115   } .otherwise {
        state_next := states.STEP_START
117   }
    }
119
    .otherwise { state_next := states.INIT } // DEFAULT CASE
121 }
```

```
123  //==================[ FILTER LOGIC ]==================//

125  ioreg.dout_valid := ((state =/= states.INIT) && ioreg.dout_valid && ~io.
      ↪ dout.ready) ||
       ((state === states.STEP_END) && step_is_last && dout_writeable)
127
     ioreg.din_ready := (state === states.WAIT) ||
129    ((state === states.STEP_END) && step_is_last && dout_writeable)

131  switch (state) {
       is (states.INIT) {
133      delay_1.write(pos, 0.S)
         delay_2.write(pos, 0.S)
135      when (!step_is_last) { pos := pos + 1.U }
         .otherwise           { pos := 0.U }
137    }
       is (states.WAIT) {
139      pos := 0.U
         y1 := 0.S
141      y2 := 0.S
         x_swap_1 := io.din.bits
143      x_swap_2 := io.dout.bits
       }
145    is (states.STEP_START) {
         delay_1.write(pos, x_swap_1)
147      x_swap_1 := mult1.io.b.bits

149      delay_2.write(pos, x_swap_2)
         x_swap_2 := mult2.io.b.bits
151
         ioreg.step_start_strobe := true.B
153      ioreg.read_vals_ready := false.B
       }
155    is (states.READ_VALS) {
         when (
157        mult1.io.dout.valid &&
           mult2.io.dout.valid
159      ) {
           ioreg.read_vals_ready := true.B
161        y1 := y1 + mult1.io.dout.bits
           y2 := y2 + mult2.io.dout.bits
163      }
```

```
      .otherwise { ioreg.read_vals_ready := false.B }
      ioreg.step_start_strobe := false.B
    }
    is (states.STEP_END) {
      ioreg.read_vals_ready := false.B
      when (step_is_last) {
        when (dout_writeable) {
          ioreg.dout_bits := (y1 + y2) >> log2(w_denom(0))
          pos := 0.U
        }
      } .otherwise { pos := pos + 1.U }
    }
  }
}
```

Листинг П.2: Исходный код алгоритмического БИХ-фильтра на Chisel

ПРИЛОЖЕНИЕ 3.

ИСХОДНЫЙ КОД АЛГОРИТМИЧЕСКОГО АДАПТИВНОГО БИХ-ФИЛЬТРА НА CHISEL

```
package filter

import chisel3._
import chisel3.util._
import chisel3.experimental.ChiselEnum
import math.log

import algomath.Algomult


class IIR_adaptive(
  XW: Int = 8, // Разрядность данных
  WW: Int = 8, // Разрядность весов
  w_numer: Seq[BigInt] = Seq(1), // Стартовые весовые коэффициенты
  w_denom: Seq[BigInt] = Seq(1), //
  calc_w: ( // Функция рассчёта новых весовых коэффициентов
    Int, Int, Int, // xw, ww, order
```

```
    Bool, // din valid signal
    Bool, // dout ready signal
    UInt, // pos
    SInt, // din bits
    SInt, // dout bits
    Mem[SInt], // delay_1
    Mem[SInt], // delay_2
    SInt, // x_swap_1
    SInt, // x_swap_2
    Mem[SInt], // w_1
    Mem[SInt], // w_2
    Algomult, // mult_1
    Algomult, // mult_2
    SInt // External signal d
  ) => (SInt, SInt, Bool, Bool) // new_w_1, new_w_2, din_ready, dout_valid
) extends Module {

  def log2(x: BigInt): Int = (log(x.toDouble) / log(2)).toInt
  def clog2(x: BigInt): Int = (log(x.toDouble) / log(2)).ceil.toInt

  if (w_numer.length != w_denom.length) throw new IllegalArgumentException("
   ↪ Numerator and denominator should be the same length")

  val order = w_numer.length
  val log_a0 = log2(w_denom(0))
  val YW = XW + WW + order - log2(w_denom(0)) // Разрядность выходных значен
   ↪ ий

  val io = IO( new Bundle {
    val din  = Flipped(Decoupled(SInt(XW.W)))
    val dout = Decoupled(SInt(YW.W))
    val init = Input(Bool())
    val d = Input(SInt(YW.W))
  })
  object ioreg {
    val din_ready  = RegInit(false.B)
    val dout_valid = RegInit(false.B)
    val dout_bits  = RegInit(0.S(YW.W))
    /*object mult {
      val w_1_bits = RegInit(0.S(WW.W))
      val w_2_bits = RegInit(0.S(WW.W))
    }*/
    val step_start_strobe = RegInit(false.B)
```

```
59      val read_vals_ready = RegInit(false.B)
    }
61  // Подключение выходов модуля к регистрам
    io.din.ready  := ioreg.din_ready
63  io.dout.valid := ioreg.dout_valid
    io.dout.bits  := ioreg.dout_bits
65
    val dout_writeable = ~io.dout.valid | io.dout.ready
67  //val reading_input = io.din.valid & io.din.ready

69  val delay_1 = Mem(w_numer.length,   SInt(XW.W))
    val delay_2 = Mem(w_denom.length-1, SInt(YW.W))
71  val w_1     = Mem(order, SInt(WW.W))
    val w_2     = Mem(order, SInt(WW.W))
73
    val pos      = RegInit(0.U(clog2(order).W))
75  val s_1      = RegInit(0.S((XW + WW + order).W))
    val s_2      = RegInit(0.S((YW + WW + order).W))
77  val x_swap_1 = RegInit(0.S(XW.W))
    val x_swap_2 = RegInit(0.S(YW.W))
79
    val step_is_last: Bool = pos === (order-1).U
81
    val mult_1 = Module(new Algomult(WW, XW))
83  val mult_2 = Module(new Algomult(WW, YW))
    mult_1.io.a.valid := ioreg.step_start_strobe
85  mult_1.io.b.valid := ioreg.step_start_strobe
    mult_2.io.a.valid := ioreg.step_start_strobe
87  mult_2.io.b.valid := ioreg.step_start_strobe
    mult_1.io.dout.ready := ioreg.read_vals_ready
89  mult_2.io.dout.ready := ioreg.read_vals_ready
    mult_1.io.a.bits := w_1.read(pos) //ioreg.mult.w_1_bits
91  mult_1.io.b.bits := delay_1.read(pos)
    mult_2.io.a.bits := w_2.read(pos) //ioreg.mult.w_2_bits
93  mult_2.io.b.bits := delay_2.read(pos)


95
    // din ready signal ignored
97  val (new_w_1, new_w_2, _, calc_w_dout_valid) = calc_w(
      XW, WW, order,
99    ioreg.step_start_strobe,
      ioreg.read_vals_ready,
101   pos, io.din.bits, io.dout.bits,
```

```
    delay_1, delay_2, x_swap_1, x_swap_2,
103   w_1, w_2, mult_1, mult_2,
    io.d
105 )

107 //==================[ STATE MACHINE ]==================//
  object states extends ChiselEnum {
109   val INIT,
      WAIT,
111     STEP_START,
      READ_VALS,
113     STEP_END
      = Value
115 }
  val state = RegInit(states.INIT)
117 val state_next = Wire(chiselTypeOf(states.INIT))
  when (io.init) {
119   state := states.INIT
  } .otherwise {
121   state := state_next
  }
123
  { // CASE
125   when (state === states.INIT) {
      when (step_is_last) { state_next := states.WAIT }
127   .otherwise          { state_next := states.INIT }
    }
129
    .elsewhen (state === states.WAIT) {
131   when (io.din.valid & io.din.ready) { state_next := states.STEP_START }
      .otherwise                         { state_next := states.WAIT }
133 }

135 .elsewhen (state === states.STEP_START) {
      state_next := states.READ_VALS
137 }

139 .elsewhen (state === states.READ_VALS) {
      when (
141     mult_1.io.dout.valid &&
        mult_2.io.dout.valid &&
143     calc_w_dout_valid
      ) { state_next := states.STEP_END }
```

```
145       .otherwise { state_next := states.READ_VALS }
    }
147
    .elsewhen (state === states.STEP_END) {
149       when (step_is_last) {
            state_next := Mux(dout_writeable, states.WAIT, states.STEP_END)
151       } .otherwise {
            state_next := states.STEP_START
153       }
    }
155
    .otherwise { state_next := states.INIT } // DEFAULT CASE
157 }

159 //==================[ FILTER LOGIC ]==================//

161 ioreg.dout_valid := ((state =/= states.INIT) && ioreg.dout_valid && ~io.
     ↪ dout.ready) ||
      ((state === states.STEP_END) && step_is_last && dout_writeable)
163
    ioreg.din_ready := (state === states.WAIT) ||
165   ((state === states.STEP_END) && step_is_last && dout_writeable)

167 switch (state) {
      is (states.INIT) {
169     delay_1.write(pos, 0.S)
        delay_2.write(pos, 0.S)
171     val w_numer_vec = VecInit(w_numer.map{ e => e.asSInt })
        val w_denom_vec = VecInit(w_denom.tail.map{ e => e.asSInt })
173     w_1.write(pos, w_numer_vec(pos))
        w_2.write(pos, w_denom_vec(pos))
175     when (!step_is_last) { pos := pos + 1.U }
        .otherwise           { pos := 0.U }
177   }
      is (states.WAIT) {
179     pos := 0.U
        s_1 := 0.S
181     s_2 := 0.S
        x_swap_1 := io.din.bits
183     x_swap_2 := io.dout.bits
      }
185   is (states.STEP_START) {
        delay_1.write(pos, x_swap_1)
```

```
187       x_swap_1 := delay_1.read(pos)

189       delay_2.write(pos, x_swap_2)
          x_swap_2 := delay_2.read(pos)
191
          //ioreg.mult.w_1_bits := w_1.read(pos)
193       //ioreg.mult.w_2_bits := w_2.read(pos)
          ioreg.step_start_strobe := true.B
195       ioreg.read_vals_ready := false.B
        }
197     is (states.READ_VALS) {
          when (
199         mult_1.io.dout.valid &&
            mult_2.io.dout.valid &&
201         calc_w_dout_valid
          ) {
203         ioreg.read_vals_ready := true.B
            s_1 := s_1 + mult_1.io.dout.bits
205         when (!step_is_last) { s_2 := s_2 + mult_2.io.dout.bits }
            w_1.write(pos, new_w_1)
207         when (pos =/= 0.U) { w_2.write(pos, new_w_2) }
          }
209       .otherwise { ioreg.read_vals_ready := false.B }
          ioreg.step_start_strobe := false.B
211     }
        is (states.STEP_END) {
213       ioreg.read_vals_ready := false.B
          when (step_is_last) {
215         pos := 0.U
            when (dout_writeable) {
217           ioreg.dout_bits := (s_1 - s_2) >> log2(w_denom(0))
            }
219       } .otherwise { pos := pos + 1.U }
        }
221   }
}
```

Листинг П.3: Исходный код алгоритмического адаптивного БИХ-фильтра на Chisel

ПРИЛОЖЕНИЕ 4.

ПРИМЕРЫ АДАПТИВНЫХ ФУНКЦИЙ И СОЗДАНИЕ ОБЪЕКТА ФИЛЬТРА

```
package filter

import chisel3._
import algomath.Algomult

object FilterDriver extends App {
  def lms(
    XW: Int,
    WW: Int,
    order: Int,
    din_valid: Bool,
    dout_ready: Bool,
    pos: UInt,
    din: SInt,
    dout: SInt,
    delay_1: Mem[SInt],
    delay_2: Mem[SInt],
    x_swap_1: SInt,
    x_swap_2: SInt,
    w_1: Mem[SInt],
    w_2: Mem[SInt],
    mult_1: Algomult,
    mult_2: Algomult,
    d: SInt
  ): (SInt, SInt, Bool, Bool) = {
    val e = d - dout // error
    // Два умножителя для рассчёта двух весов за раз
    val mw_1 = Module(new Algomult(e.getWidth, din.getWidth))
    val mw_2 = Module(new Algomult(e.getWidth, dout.getWidth))
    // Подключение умножителей к сигналам FIFO и линиям данных
    mw_1.io.a.valid := din_valid
    mw_1.io.b.valid := din_valid
    mw_2.io.a.valid := din_valid
    mw_2.io.b.valid := din_valid
    mw_1.io.a.bits := e
    mw_1.io.b.bits := x_swap_1
```

```
    mw_2.io.a.bits := e
    mw_2.io.b.bits := x_swap_2

    val din_ready = mw_1.io.a.ready && mw_1.io.b.ready && mw_2.io.a.ready &&
    ↪  mw_2.io.b.ready
    val dout_valid = mw_1.io.dout.valid && mw_2.io.dout.valid
    mw_1.io.dout.ready := dout_ready
    mw_2.io.dout.ready := dout_ready

    ( // Деление на 128 сдвигом на 7 вправо
      w_1.read(pos) + (mw_1.io.dout.bits.head(WW).asSInt >> 7),
      w_2.read(pos) + (mw_2.io.dout.bits.head(WW).asSInt >> 7),
      din_ready, dout_valid
    ) // Return tuple
  }

  // No adaptivity
  def lms2(
    XW: Int, WW: Int, order: Int,
    din_valid: Bool, dout_ready: Bool,
    pos: UInt, din: SInt, dout: SInt,
    delay_1: Mem[SInt], delay_2: Mem[SInt],
    x_swap_1: SInt, x_swap_2: SInt,
    w_1: Mem[SInt], w_2: Mem[SInt],
    mult_1: Algomult, mult_2: Algomult, d: SInt
  ): (SInt, SInt, Bool, Bool) = {
    (w_1(pos), w_2(pos), true.B, true.B)
  }

  def lms3(
    XW: Int, WW: Int, order: Int,
    din_valid: Bool, dout_ready: Bool,
    pos: UInt, din: SInt, dout: SInt,
    delay_1: Mem[SInt], delay_2: Mem[SInt],
    x_swap_1: SInt, x_swap_2: SInt,
    w_1: Mem[SInt], w_2: Mem[SInt],
    mult_1: Algomult, mult_2: Algomult, d: SInt
  ): (SInt, SInt, Bool, Bool) = {
    val e = Mux(dout < 0.S, -1105392.S, 1105392.S) - dout // error
    // Два умножителя для рассчёта двух весов за раз
    val mw_1 = Module(new Algomult(e.getWidth, din.getWidth))
    val mw_2 = Module(new Algomult(e.getWidth, dout.getWidth))
    // Подключение умножителей к сигналам FIFO и линиям данных
```

```
    mw_1.io.a.valid := din_valid
    mw_1.io.b.valid := din_valid
    mw_2.io.a.valid := din_valid
    mw_2.io.b.valid := din_valid
    mw_1.io.a.bits := e
    mw_1.io.b.bits := x_swap_1
    mw_2.io.a.bits := e
    mw_2.io.b.bits := x_swap_2

    val din_ready = mw_1.io.a.ready && mw_1.io.b.ready && mw_2.io.a.ready &&
     ↪  mw_2.io.b.ready
    val dout_valid = mw_1.io.dout.valid && mw_2.io.dout.valid
    mw_1.io.dout.ready := dout_ready
    mw_2.io.dout.ready := dout_ready

    (
      w_1.read(pos) + (mw_1.io.dout.bits.asSInt << 16-7 ).head(WW).asSInt,
      w_2.read(pos) + (mw_2.io.dout.bits.asSInt << 27-7 ).head(WW).asSInt,
      din_ready, dout_valid
    ) // Return tuple
  }

  //chisel3.Driver.execute(args, () => new TestReg())
  //chisel3.Driver.execute(args, () => new Algomult(32, 32))
  //chisel3.Driver.execute(args, () => new FIR_filter(8, Seq(5, -5, 10, -30)
   ↪ ))
  //chisel3.Driver.execute(args, () => new IIR_static(8, 8, Seq(5, -5, 10,
   ↪ BigInt("-30")), Seq(1, -4, 66, -33)))
  //chisel3.Driver.execute(args, () => new IIR_adaptive(8, 16, Seq(5, -5,
   ↪ 10, BigInt("-30"), 22, 3), Seq(64, -4, 66, -33, 0, 0), lms))
  val b = Seq(
    +0.353494866,
    -2.120969196,
    +5.302422989,
    -7.069897319,
    +5.302422989,
    -2.120969196,
    +0.353494866,
  ).map(e => BigInt((e*1048576).toLong)) // Перевод в целые числа. 1048576 =
   ↪  2^20
  val a = Seq(
    +1.000000000,
    -3.973726290,
```

```
    +6.860878999,
    -6.505073811,
    +3.575180738,
    -1.086720466,
    +0.149064122
  ).map(e => BigInt((e*1048576).toLong)) // Перевод в целые числа. 1048576 =
   ↪  2^20
  println("b:", b)
  println("a:", a)
  //chisel3.Driver.execute(args, () => new IIR_adaptive(20, 24, b, a, lms2))
  chisel3.Driver.execute(args, () => new IIR_adaptive(20, 24, b, a, lms3))
}
```

Листинг П.4: Примеры адаптивных функций и создание объекта фильтра