**Содержание**

**Введение**

В данной книге описана архитектура многоядерных матричных процессоров SEAforth, представляющих одну из интересных разработок в области процессоров для встраиваемых систем. Помимо большого количества ядер – сорок, процессор обладает высокой производительностью и очень высокой энергоэффективностью

Описаны команды языка Ассемблер с примерами реализации. Ключевым фактором для данного процессора является то, что Ассемблер процессора является одновременно и языком среднего уровня – представляет собой вариацию языка Форт.

Рассмотрены примеры программ – от простых арифметических операций и работы с периферийными устройствами, до базовых алгоритмов цифровой обработки сигналов.

# Глава 1. Архитектура процессоров SEAforth

Процессоры SEAforth [**1**, **2**] позиционируются как многоядерные процессоры для встраиваемых систем реального времени. Построены по технологии встраиваемого масштабируемого массива — Scalable Embedded Array.

Одно из основных возможных применений процессов SEAforh — производственные контролирующие и сенсорные системы, контроллеры сенсорных сетей, мультимедийные системы [**3, 4**].

Данные процессоры отвечают практически всем современным тенденциям — параллелизм, низкое энергопотребление, высокая скорость работы ядра. Есть также и моменты, которые ставят их особняком: кроме стековой архитектура ядра это отсутствие модуля вещественных вычислений; отсутствие кэша, если сравнивать с процессорами; малый объем памяти, небольшое количество периферийных устройств, если проводить сравнение с микроконтроллерами.

Имея в своем распоряжении несколько десятков ядер, программист может выделять некоторые группы из них для решения различных специализированных задач. Для типовых задач возможно создание т. н. параметризированных макроблоков из нескольких ядер. В результате получается гибкая мультипроцессорная система, предназначенная для решения широкого круга прикладных задач.

Поскольку объем памяти для размещения программ мал, типичной практикой в программировании процессора является замена исполнимого кода во время исполнения программы. Как правило, каждое ядро исполняет относительно небольшую функцию или блок приложения, что характерно для большого количества задач управления или обработки потоков данных. В качестве примера можно привести реализацию QPSK-приемника и передатчика на SEAforth24 [**3**].

Одно из основных возможных применений процессов SEAforth —

производственные контролирующие и сенсорные системы, контроллеры сенсорных сетей [**4**], мультимедийные системы. Причины, обусловливающие это:

— возможность параллельной обработки данных в реальном времени;

— возможность распределенного управления промышленной сетью.

К процессору в системе сбора данных могут быть подключены датчики практически с любым интерфейсом (при условии согласования уровней напряжения) и различными протоколами обмена. Это достигается за счет программного управления линиями ввода-вывода, таким образом, различные ядра процессора имеют возможность взаимодействовать с датчиками по различным протоколам. Упрощается также замена датчиков — при необходимости вносятся коррективы только в программное обеспечение.

Кроме непосредственно сбора данных параллельно процессор может вести их обработку в реальном времени, включая такие вычислительно емкие задачи, как цифровая фильтрация, быстрые преобразования (Фурье, Уолша, Хартли и др.), анализ аудио-, видео- или радиосигналов, PID-регулирование, ФАПЧ.

Процессор состоит из множества независимых вычислителей, соединенных по топологии двумерной прямоугольной решетки, имеет ряд последовательных портов, параллельные порты, выводы общего назначения, АЦП, ЦАП (Рисунок 1). Каждый из вычислителей является фон-неймановским стековым процессором с сокращенным набором команд, имеет собственную оперативную память, порты ввода-вывода и постоянную память с рядом предустановленных фирменных функций. Взаимодействие вычислителей (далее — ядер) осуществляется посредством механизма параллельных взаимодействующих процессов (парадигма **ОККАМ**). В некотором приближении процессор можно

рассматривать как сеть из миниатюрных транспьютероподобных узлов. Глобальная синхронизация ядер отсутствует.

Линии ввода-вывода можно применять для программной реализации различного рода интерфейсов, помимо ограниченной поддержки основных последовательных интерфейсов в ПЗУ некоторых ядер. Последовательные порты могут быть использованы для соединения нескольких процессоров.

Программное управление линиями ввода-вывода позволяет поддерживать несколько форматов цифровых сигналов и видов модуляций аналоговых сигналов, в отличие от специализированных СБИС, поддерживающих 1-2 стандарта.

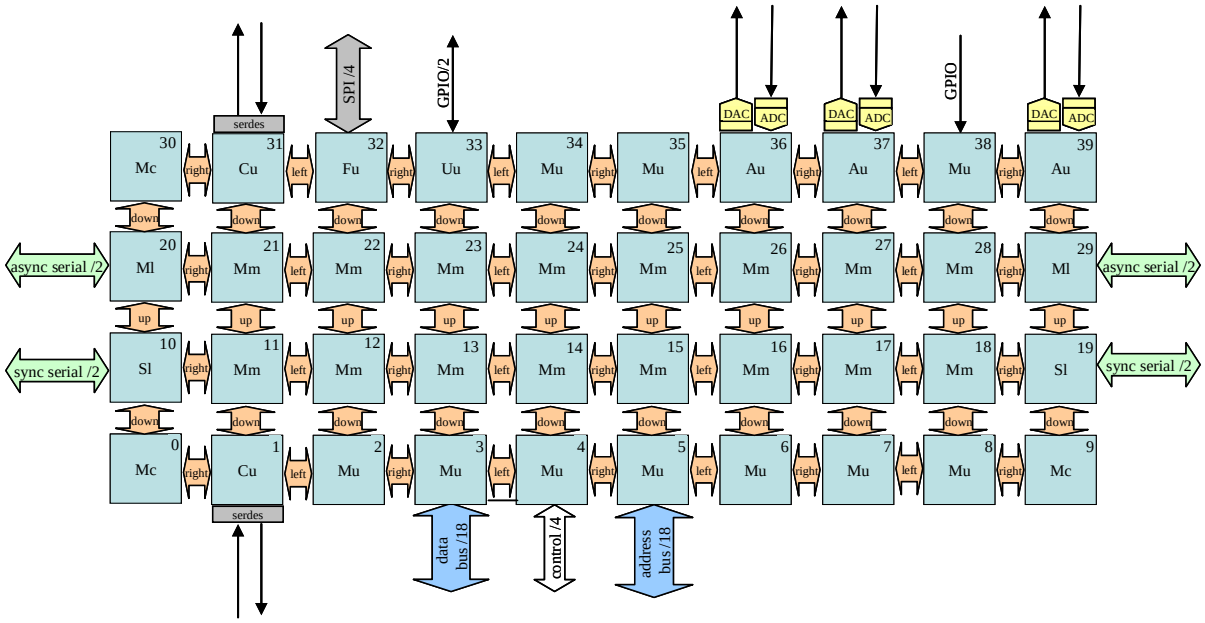


**Рисунок 1. Структурная схема процессора SEAforth40.**

Интерфейсы и протоколы, реализуемые программно, могут варьироваться от простых протоколов, аналогичных RS-232, SPI, до достаточно сложных алгоритмов кодирования типа QPSK или QAM. При этом максимальные скорости обмена данными лежат в диапазоне 10-20 Мбит/с.

В процессорах применен ряд технологий, позволяющих добиться значительной экономии энергии. Наиболее заметная из них: ядро, первым

подошедшее к точке передачи данных, автоматически переходит в спящий (пассивный) режим до тех пор, пока взаимодействующее с ним ядро не подойдет к этой же точке. При этом время выхода из пассивного режима составляет примерно 10 нс. В спящем режиме ядро рассеивает порядка 1 мкВт, плюс обеспечивается прозрачная для прикладного программиста синхронизация процессов, выполняющихся на различных ядрах. Как правило, во время работы приложения активны одновременно активны 8-16 ядер. Вторая особенность — внутреннее представление данных. На физическом уровне в ядре в соседних битах одинаковые логические уровни представлены противоположными уровнями напряжений. Например, если в регистре находится число 100010b, то физические уровни напряжений на его выходах будут следующие: HHHBBB. И третий момент — отсутствие глобальной синхронизации ядер и асинхронное исполнение самого ядра (нет тактовых генераторов). Низкое энергопотребление и высокая энергетическая эффективность являются одними из ключевых свойств процессоров SEAforth.

В процессорах SEAforth не используется механизм прерываний. В случае необходимости ядро переходит в пассивный режим, ожидая записи или чтения в коммуникационный порт. При этом время перехода из пассивного в активный режим и наоборот крайне мало — <10нс. Благодаря наличию нескольких периферийных ядер процессор может одновременно обслуживать несколько источников событий с минимальной задержкой, что идеально для систем, работающих в режиме жесткого реального времени. Поскольку ядра асинхронные, исполнение программы происходит с максимально возможной скоростью. Процессор не нуждается в режимах пониженной частоты, характерных для обычных микроконтроллеров с целью понижения энергопотребления.

Процессор SEAforth40 (40C18) содержит 40 ядер (рис.2), объединенных в решетку 4x10.

Ввод-вывод в SEAforth программно доступен по трем путям:

— цифровые выводы, доступные через специальные регистры или регистры IOCS;

— межпроцессорные коммуникации, выполняемые при помощи направленных портов;

— аналоговый ввод-вывод, доступный через специальные регистры или регистры IOCS.

Каждое ядро C18 разделяет до четырех портов ввода-вывода со своими соседями. В общем случае межпроцессорные коммуникации являются блокирующими и самосинхронизирующимися — это значит, что процессор уходит в спящее состояние, пока операция не будет завершена.

Каждый межпроцессорный коммуникационный порт соединен напрямую со своими соседями — соседние узлы разделяют один порт. Общий порт имеет для соседних процессоров один и тот же адрес. Нет регистров или FIFO-буфера — одни линии порта напрямую соединены с соседними линиями записи. Значение, записанное в порт, может иметь различные интерпретации — узел, проводящий чтение, может исполнить считанный код. Когда процессор проводит операцию чтения, он блокирует запись соседнего процессора; когда процессор пишет, он блокирует операцию записи соседа. Подобный метод синхронизирует соседние процессоры. Блокировка — ключевой элемент в уменьшении потребляемой процессором мощности и средство синхронизации ядер (процессов). В блокированном состоянии ядро практически не потребляет мощности. Только один из процессоров блокируется во время транзакции — тот, кто входит в нее первым. Блокировки можно избежать, считывая бит статуса в регистре IOCS перед операцией чтения или записи. Возможны операции множественного чтения/записи в порт. При наличии готового к транзакции соседнего узла операции чтения/записи занимают порядка 4,2 нс.

Каждое ядро, входящее в состав процессоров SEAforth, содержит 18-

разрядный микропроцессор C18, имеющий 64 18-разрядных слова ОЗУ, 64 18-разрядных слова ПЗУ с заранее прошитыми функциями (т. н. Intellasys-bios), 4 порта ввода-вывода (рис. 2). Некоторые узлы, находящиеся по краям решетки, имеют дополнительно последовательный или параллельный порт, АЦП или ЦАП. Функции, прошитые в ПЗУ отдельных ядер, также несколько отличаются.

По сравнению с предыдущей моделью в его ядра C18 внесены изменения, существенно упрощающие организацию вычислительного процесса. Был добавлен т. н. режим расширенной арифметики, при котором перенос сохраняется в 10-м бите регистра P (данный бит в адресации памяти не участвует); таким образом, появляется возможность производить операции над многоразрядными числами. Вторая особенность АЛУ — операция +* производится между вторым элементом стека и регистром A. Старшее слово результата помещается на вершину стека, младшее остается в регистре A, второй элемент стека — регистр S — остается без изменений. Два ядра процессора имеют высокоскоростные интерфейсы SERDES, отсутствовавшие в SEAforth24.

Производительность ядра — примерно 700 MIPS (время выполнения инструкции примерно 1,4 нс) при уровне потребления мощности ~12 мВт. Соответственно — максимальная мощность, рассеиваемая процессором при полной загрузке всех узлов, не более 250-400 мВт, что сравнимо со многими сигнальными процессорами.

Помимо ОЗУ и ПЗУ C18 содержит два стека — данных и возвратов, с выделенными регистрами вершин стеков и логикой управления, регистр слова-инструкции, программный счетчик, АЛУ, декодер команд и логику выборки команд, а также два индексных регистра.

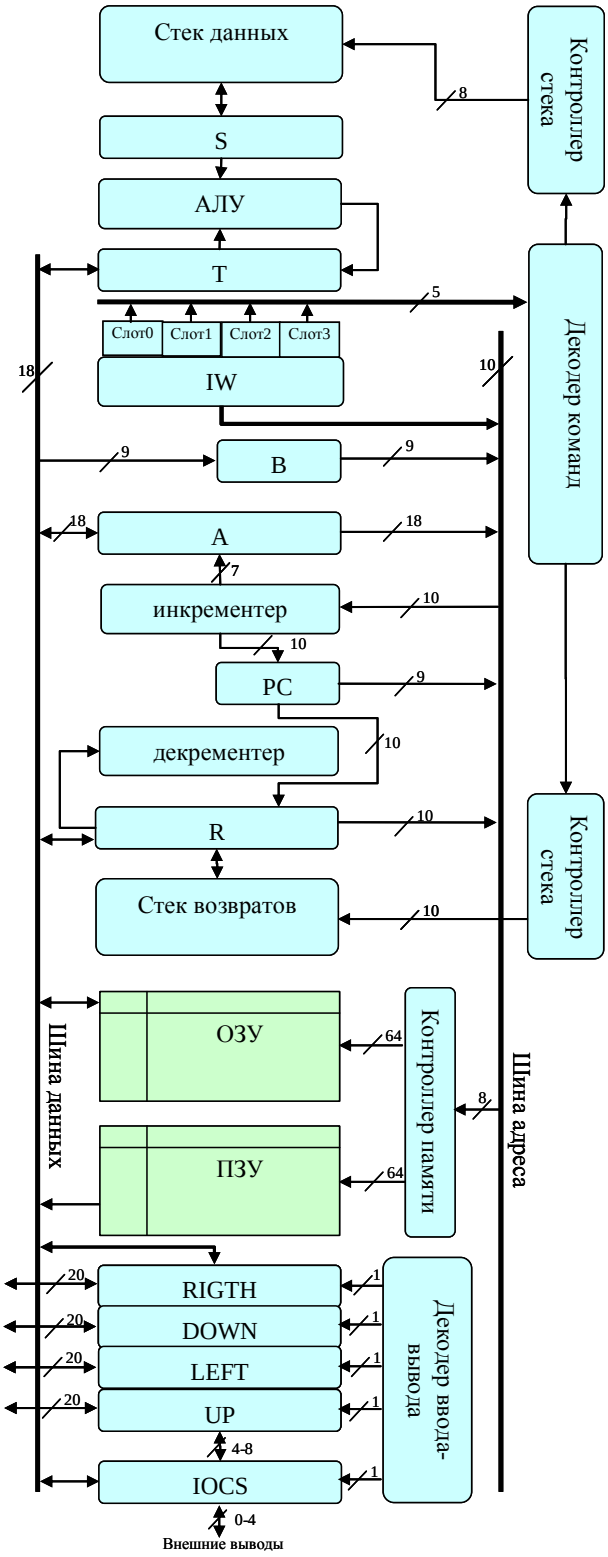


**Рисунок 2. Структурная схема процессорного ядра C18.**

Инструкции имеют длину всего лишь 5 бит, что позволяет упаковывать три или четыре инструкции в одно 18-битное слово. 18-битное

слово содержит до 4 опкодов, выбираемых слот-селектором, и передается на декодер и логику контроля, управляющую функциями ядра. Восемь из 5-битных инструкций могут быть помещены в 3-битный слот, как последний код операции в слове. Таким образом, максимально в оперативной памяти C18 можно разместить 256 команд, что с учетом высокой реентерабельности форт-кода достаточно для реализации многих алгоритмов и прикладных программ.

Стеки в C18 — массивы регистров. Стек данных используется для выполнения арифметических операций. Два наиболее часто используемых регистра могут быть доступны непосредственно — T и S. Оба они подключены к АЛУ, результат операций помещается в регистр T. Ниже находится циклический массив из 8 регистров. Один из этих регистров в каждый момент времени выбирается как регистр, следующий за S. АЛУ, использующее регистры T и S, вычисляет все возможные арифметические операции параллельно, используя комбинаторную логику. Текущей командой выбирается необходимый результат из АЛУ и помещается в T.

C18 имеет девять 18-битных регистров стека возвратов. Верхняя позиция стека возвратов находится в регистре R. Под R также находится циклический массив из 8 регистров. Инструкции вызова оставляют значение программного счетчика на стеке возвратов. Команды возврата снимают только 9 младших разрядов. Стек возвратов также может быть использован для временного хранения данных и в качестве счетчика циклов. Аппаратного контроля за переполнением и исчерпанием стеков не предусмотрено. Поскольку регистры в стеке связаны в кольцо, они не могут переполниться или исчерпаться, они просто «прокручиваются». Так как глубина стека ограничена, добавление элемента на стек означает затирание самого нижнего элемента. Когда идет извлечение из стека, нижние 8 элементов будут повторяться. После двух чтений T и S будут содержать копии двух элементов массива стековых регистров.

**Таблица 1**

**Регистры C18**

| | |
|---|---|
| PC | 9-ти битный программный счетчик (10-битный для процессора 40c18) |
| T,S | Первый и второй элемент стека данных |
| R | Верхний регистр стека возвратов. Один из 9-ти регистров стека возвратов, доступен через push/pop,call/return |
| A | 18-битный регистр общего назначения, адресный, автоинкрементный |
| B | 9-битный адресный регистр |
| IW | 18- битное слово инструкции |
| RIGTH, DOWN, LEFT, UP | Коммуникационные регистры (являются общими с соседними ядрами) |
| IOCS | Регистр режим выводов и статус ввода-вывода |
| DATA | Внешняя шина данных |
| ADDRESS | Адресный регистр внешней памяти |

Выходы программного счетчика PC управляют адресной шиной (подаются на шину адреса) программный счетчик по выполнении инструкции инкрементируется. Данные, адресуемые PC, загружаются в регистр инструкций IW.

Регистры IOCS - предназначены как для ввода-вывода, так и для мониторинга межпроцессорного обмена. Они являются комбинацией элементов хранения бит статуса, показывающих, произошли ли операции чтения или записи (с той или другой стороны). Также они содержат значение, выводимое или считанное с линии ввода-вывода. Режим и выходное состоя-

ние линии устанавливается записью в IOCS. Типовое время доступа к регистрам IOCS – 2,8нс.

Структура адресного пространства. Карта распределения адресного пространства микропроцессора C18 выглядит следующим образом - рис.3. Оперативная, постоянная память, специальные регистры и порты ввода-вывода находятся  в одном адресном пространстве и для работы с ними применяются одни и те же команды.

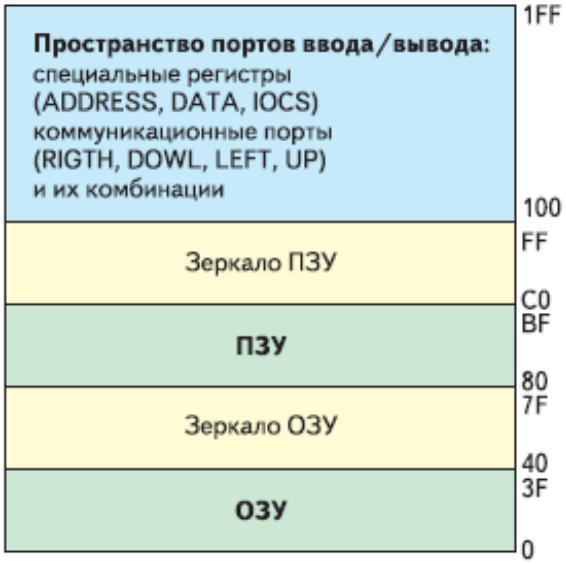


**Рисунок 3. Карта памяти C18.**

При выполнении программы логика берет значение с адресной шины и вычисляет новый адрес параллельно со временем доступа шины. Результат становится доступным для PC или для регистра A. Использование инкрементированного адреса определяется текущей инструкцией. При нормальном потоке выполнения результат записывается в PC. В пределах границы 128 слов, инкрементированный адрес может прокрутиться к началу страницы, следовательно, RAM и ROM появляются в двух различных диапазонах.

Когда бит 8 адреса равен 1, идет адресация в пространстве ввода-вывода и увеличение адреса запрещено, т.е. когда регистр указывает на порт, указатель не сдвигается. Это значит, что выборка инструкций, лите-

ралов, запись литералов выполняются непосредственно с порта. Вызов с порта возвращает на порт.

Процессор имеет достаточно большой объем математических библиотек и библиотек ввода-вывода (таблица 2). Математическая библиотека содержит наборы функции умножения: mlt, fmult, для операций с целыми числами и с числами в формате с фиксированной точкой, деления с остатком um/mod (имеет несколько точек входа), линейной интерполяции — interp, работы с векторами — rotation, вычисление полиномов — poly, вычисления триангулярной функции — triangle, taps — слова для организации расчета цифровых фильтров.

**Таблица 2**

**Слова и наборы слов ПЗУ процессоров SEAforth**

| **Обозначение** | **Слова и наборы слов ПЗУ SEAforth40** |
|---|---|
| **Mc** | relay, cornerwarm, poly, mlt, fmlt, taps, interp, triangle, -u/mod |
| **Mm** | relay, centerwarm, poly, mlt, fmlt, taps, interp, triangle, -u/mod |
| **Mu** | relay, upwarm, poly, mlt, fmlt, taps, interp, triangle, -u/mod |
| **Ml** | relay, leftwarm, poly, mlt, fmlt, taps, interp, triangle, -u/mod |
| **Au** | relay, upwarm, poly, mlt, fmlt, -dac, interp, triangle, -u/mod |
| **Ac** | relay, cornerwarm, poly, mlt, fmlt, -dac, interp, triangle, -u/mod |
| **Ul** | relay, leftwarm, poly, mlt, fmlt, taps, interp, triangle, bget |
| **Fu** | relay, upwarm, spi, u2/ |
| **Uu** | relay, upwarm, serial, lshift, rshift |
| **Sl** | relay, leftwarm, sget, mlt, taps, triangle |
| **Cu** | relay, upwarm, upserdes, mlt, fmlt, taps, interp, triangle, -u/mod |

Функции ввода-вывода представлены набором слов для последовательных интерфейсов (синхронного и асинхронного — представлены в основном функциями чтения данных с порта). Spi — слова

начальной загрузки с флэш-памяти, чтения последовательной памяти. Serial — чтение данных с последовательного порта. Sget — работа с синхронным последовательным портом. Bget — чтение данных с асинхронного последовательного порта. Upserdes — поддержка работы с высокоскоростными последовательными портами SERDES (загрузка и исполнение кода).

***Система команд.*** Система команд C18 состоит из 32 базовых инструкций, составляющих язык VentureForth. VentureForth имеет все достоинства языка Форт: экономичность, простота и расширяемость. IntellaSys расширяет возможности VentuerForth добавлением поддержки Forthlet – объектов, которые могут распространяться между ядрами. Список команд с их кратким описанием представлен в таблице 3.

Инструкции имеют длину всего лишь 5 бит, что позволяет упаковывать три или четыре инструкции в одно 18-битное слово. 18-ти битное слово содержит до 4х опкодов (рисунок 4), выбираемых слот-селектором и передается на декодер и логику контроля, управляющей функциями ядра. Восемь из 5-ти битовых инструкций могут быть помещены в 3-х битный слот, как последний код операции в слове (3 полных 5-ти битных слота, плюс 3-битный остаток). Таким образом, максимально в оперативной памяти C18 можно разместить 256 команд, что с учётом высокой реентерабельности форт-кода достаточно для реализации многих алгоритмов и прикладных программ.

| | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Opcode Slot | Slot 0 | | | | | Slot 1 | | | | | Slot 2 | | | | | Slot 3 | | |

**Рисунок 4. Слоты в регистре инструкций IW.**

Типичная последовательность выполнения кода начинается со значения загруженного в программный счетчик и выставленного на шину ад-

реса. Это значение используется двумя способами - выбирает адрес в памяти (в некоторых случаях - порты ввода-вывода) и управляет «инкрементером», который записывает увеличенное значение обратно в счетчик. Выбранное значение подается на шину данных, значение шины фиксируется в регистре IW. Значение в первом слоте подается на блок логики контроля и декодирования, который управляет работой элементов ядра. Также блок имеет логику «предсказания» - если обнаруживается, что ни один из опкодов не использует шину адреса, разрешается выдача значения PC на шину адреса для предварительной выборки следующей команды.

Загрузка литералов (констант/чисел), вызовы, переходы, обращения к памяти и портам требуют операнды. Команда перехода или вызова может иметь 3, 8 или 9-ти битный аргумент. Инструкции-литералы используют 5-битный опкод и 18-битное слово - литерал, который будет помещен на стек.

Некоторые ядра, находящиеся по краям решетки имеют дополнительные устройства в виде параллельных или последовательных портов, аналого-цифрового преобразователя (АЦП), цифроаналогового преобразователя (ЦАП), и, соответственно имеют определенное количество внешних выводов.

Параллельные порты доступны для чтения/записи либо как коммуникационные порты ядра (как правило UP порт), при этом операции с ними являются блокирующими или при помощи специальных регистров DATA, ADDRESS, в этом случае блокировок не происходит. Параллельные порты могут быть использованы, как для подключения внешней памяти, так и как порты ввода-вывода общего назначения. Подключение внешней памяти будет иметь свою специфику для каждого из процессоров семейства - в SEAforth24 параллельные порты принадлежат одному ядру, в SEAforth40 - двум, и третий управляет контролирующими линиями. Время доступа к параллельным портам менее

5нс.

**Таблица 3**

**Список команд процессорного ядра C18**

| **имя** | **нотация** | **описание** |
|---|---|---|
| **call** | R: -- a | вызов подпрограммы |
| **;** | R: a -- | возврат из подпрограммы |
| **jump** | | безусловный переход |
| **;:** | R: a1 -- a2<br>p: a2 -- a1 | передача управления сопрограмме (примитив для реализации многозадачности) |
| **if** | D: x -- x | переход, если T=0 |
| **-if** | D: x -- x | переход, если число в T отрицательное (старший бит = 1) |
| **next** | R: n -- n-1<br>R: 0 -- | если n<>0, переход по адресу, указанному в поле перехода |
| **unext** | R: n -- n-1<br>R: 0 -- | аналогично next, но в пределах одного слова инструкции |
| **@a** | D: -- x | T=(A). помещает на стек значение, по адресу в регистре A |
| **!a** | D: x -- | (A)=T. заносит в память по адресу, указанному в A значение с вершины стека |
| **@a+** | D: --x<br>A=A+1 | помещает на стек значение, по адресу в регистре A, содержимое регистра инкрементируется |
| **!a+** | D: x --<br>A=A+1 | заносит в память по адресу, указанному в A значение с вершины стека, содержимое регистра инкрементируется |
| **@b** | D: -- x | помещает на стек значение, по адресу в регистре B |

| | | |
|---|---|---|
| **!b** | D: x --<br> | заносит в память по адресу, указанному в B значение с вершины стека, содержимое регистра |
| **!p+** | D: x --<br> | значение с вершины стека записывается по адресу, указанному в p |
| **@p+** | D: -- x | помещает на стек число с адреса, указанного в p |
| **.** | | Пустая операция |
| **push** | D: x --<br>R: -- x | переносит число со стека данных на стек возвратов |
| **pop** | R: x --<br>D: -- x | переносит число со стека возвратов на стек данных |
| **dup** | D: x -- x x | дублирует вершину стека |
| **drop** | D: x -- | удаляет значение с вершины стека |
| **over** | D: x1 x2 -- x1 x2 x1 | дублирует второй элемент стека на вершину |
| **a!** | D: x --<br>A = x | переносит значение с вершины стека в регистр a |
| **a@** | a=x<br>D: -- x | копирует значение регистра a на вершину стека |
| **b!** | D: x --<br>b=x | переносит значение с вершины стека в регистр b |
| **not** | D: x -- not(x) | инвертирует вершину стека |
| **and** | D: x1 x2 -- (x1 and x2) | на вершину стека помещается логическое И регистров T и S |
| **xor** | D: x1 x2 -- (x1 xor x2) | на вершину стека помещается побитовое исключающее или регистров T и S |
| **2/** | D: x -- x/2 | деление вершины стека на 2 |

| | | |
|---|---|---|
| **2*** | D: x -- x*2 | умножение вершины стека на 2 |
| + | D: x1 x2 -- x1+x2 | суммирует вершину и второй элемент стека |
| **+*** | | шаг умножения |

По умолчанию, ядра имеющие два вывода, считаются ядрами с последовательными портами, поскольку имеется возможность организовать полнодуплексный режим работы. Ядра с одним выводом относят к портам общего назначения. Ядра с синхронными и асинхронными последовательными портами отличаются только поддерживающими кодами программ в ПЗУ, никаких реальных отличий в устройстве ядер нет.

Асинхронные порты функционируют как универсальные асинхронные приемо-передатчики (UART), и служат для подключения внешних устройств или других процессоров SEAforth. Код в ПЗУ позволяет загружать устройство через асинхронный порт, и позволяет ядру выходить из режима покоя при поступлении стартового бита. Отличия ядер с синхронными интерфейсами в том, что они используют отдельную линию как сигнал синхронизации.

SPI порты реализуются четырьмя линиями ввода-вывода и также имеют программную поддержку в ПЗУ. Код в ПЗУ предусматривает возможность загрузки исполнимого кода из последовательной флэш памяти - высокий уровень на линии spi-in начинает процесс загрузки. В процессе загрузки SPI работает на скорости 150 кбит/с, позволяя использовать относительно дешевые устройства памяти. По окончании загрузки интерфейс может функционировать со скоростью порядка 10 Мбит/с.

Ядра с одиночным выводом могут использовать его как входной либо как выходной однобитовый порт. Данный порт доступен и конфигурируется через регистр IOCS. Время перехода вывода из состояния с высоким импедансом в активное составляет порядка 110-115нс.

АЦП, имеющиеся на некоторых ядрах представляют собой генераторы, управляемые напряжением (ГУН), связанные с 18 разрядными счетчиками. Аналого-цифровое преобразование осуществляется посредством двух последовательных процедур чтения DATA регистра и вычислении скорости работы генератора. Время полного счета для входного напряжения 400мВ - 40 мкс, для 1500мВ - 75 мкс. Выделенными битами в регистре IOCS ядра счётчик может быть запущен или остановлен, на ГУН подано напряжение с внешнего вывода, или напряжения ноля или питания для калибровки. Характеристика напряжение-код *($V_{in}$--ADCcount)* АЦП является нелинейной. Зависимость *$V_{in}$(ADCcount)* можно с некоторой точностью рассматривать как кубическую вида:

$$V_{in}(ADCcount) = a_0 \cdot (ADCcount - a_1)^3 + a_2.$$

При работе АЦП потребляет не более 4,5 мВт. Время, затрачиваемое на чтение данных из АЦП примерно 5-5,1нс.

ЦАП реализован, как набор двоично-взвешенных источников тока и рассчитан на номинальную нагрузку 75Ом при токе 17 мА. Программно доступен через регистр IOCS.

SERDES - специализированные параллельно-последовательные регистры, предназначенные для осуществления коммуникаций между процессорами. Могут передавать 18-разрядные слова (которые в свою очередь могут быть, как данными, так и инструкциями), могут служить генераторами функций. Имеют две двунаправленные линии - данных и тактовую. Направление устанавливается в регистре IOCS ядра, данные записываются по адресу DATA, для контроля прихода/отправки слова используется адрес UP. Тактирование осуществляется от специального ос-

циллятора. Передача идет на скорости порядка 400Мбит/с, на передачу одного слова требуется 19 тактов.

Проведем сравнение процессоров SEAforth с наиболее распространенными целочисленными контроллерами различных архитектур [7-13] (Таблица 4).

**Таблица 4**

**Сравнительные характеристики целочисленных контроллеров**

| **Характеристика** | **AVR (picoPower)** | **AVR32** | **MSP430** | **ARM*** | **SEAforth-ядро C18 (суммарно по 24/40 ядрам)** |
|---|---|---|---|---|---|
| Разрядность | 8 | 32 | 16 | 32 | 18 |
| Производительность, MIPS | 20 | 72-210 | 8 | 50-150 | 700 (18000/ 26000) |
| Потребляемая мощность (максимальная), мВт | 13,3 | 7,6 | 4,9 | 5,8-48,75 | 12,6 (302/504) |
| Потребление в пассивном режиме (энергосберегающем), мкВт | 0,06/0,9 | 9,9 | 0,22/ 1,76/ 70 | 20-1000 | 1 (24/40) |
| Время перехода в активное состояние, мкс | | | 6 | 0,16 -60 | <0,01 |

| Затраты энергии на выполнение операций (средние значения), нДж: | | | | | |
|---|---|---|---|---|---|
| логические | 0,67 | 0,127 | 1,8 | 0,2-3,5 | 0,018 (0,42/0,7) |
| арифметические | 0,67 | 0,127 | 2 | 0,2-3,5 | 0,036 (0,84/1,4) |
| умножение | 1,33 | 0,127 | 1,25 | 0,2-3,5 | 1,15** (27,5/46) |
| операции с памятью | 1,33 | 0,19 | 2,4 | 0,2-3,5 | 0,064 (1,54/2,56) |

* средние по семейству

** программная реализация умножения

Как видно из таблицы 4, процессоры SEAforth выигрывают по показателям энергоэффективности в активном режиме, производительности по отдельным ядрам и суммарной производительности.

Большим преимуществом является очень малое время реакции на событие, высокая скорость выдачи данных на внешние выводы — до 90 МГц, относительно низкое пиковое энергопотребление, максимально возможная скорость работы в активном состоянии. Наиболее близки по этим показателям процессорные ядра AVR32 и ARM Cortex-M3 [**7**-**13**].

Процессор проигрывает в операциях типа умножение, умножение с накоплением, деление, поскольку они в нем реализованы программными средствами. Также может несколько снизить производительность ограниченный набор команд и необходимость динамической замены кода во время работы приложений.

К недостаткам также можно отнести  малый размер памяти, небольшой набор периферийных устройств, необходимость подключения внешней памяти для хранения пользовательских программ.

# Глава 2. Среда разработки - IDE VentureForth

Программное обеспечение включает в себя компилятор-симулятор языка VentureForth, имеющий версии под операционные системы Windows и Linux. Симулятор позволяет производить отладку программ, отслеживать состояние регистров и памяти любого из ядер процессора [**5-6**]. При отладке программ одновременно отображается состояние всех ядер процессора (отображаются основные регистры ядра), выделяются ядра, находящиеся в активном состоянии.

Среда VentureForth имеет достаточно простую структуру директорий:

- docs - содержит документацию на процессор и среду разработки;
- projects – директории проектов приложений;
- vf – рабочая директория среды, содержит:
  - поддиректории с настройками среды и компилятора, специфичными для конкретных версий процессора;
  - файлы с исходными текстами симулятора и вспомогательные библиотеки компилятора.

Как и в большинстве IDE, в VentureForth присутствует понятие проекта. Директория проекта содержит исходный код проекта и несколько вспомогательных файлов для запуска приложения.

Желательно для каждого нового приложения создавать новую Директорию проекта. Проще всего это сделать при помощи копирования существующей директории проекта и использования её как шаблон. Рассмотрим пример приложения s40blink, демонстрирующий работу с внешними выводами процессора и с ЦАП.

Директория приложения содержит следующие файлы:

blinkLED.vf - вывод импульсов с меняющейся длительностью на

внешние выводы

sawtooth.vf - генерация пилообразного сигнала на выводах ЦАП;

toggle17-1.vf - переключение состояния пары выводов;

blinktest.vf - центральный файл проекта, собирающий отдельные его части воедино;

project.vfp - исполняемый средой файл проекта.

**Компиляция, Симуляция и загрузка кода**

В этом примере, файл blinktest.vf определяет, что код будет запущен в каждом узле (т.е. в каждом чипе C18) и что порядок кода может доставляться узлу (рисунок 5).

**Компиляция**

Программа для каждого узла компилируется в 5 шагов.

1. Выбирается узел, к примеру: 02 {node
2. Определяется адрес, с которого начинается компиляция: 0 org
3. Определяется адрес точки входа для кода: here =p (это делается каждый раз перед выходом из узла)
4. Далее следует исходный код узла. Источник может быть включен в файл, компилирован как макро или подключен.
5. Вызывается node} в конце кода каждого узла.

В этом примере директивы {node org =p и node} используются в основном в главном файле приложения, но оны могут быть использованы и в макросах или подключаемых файлах, как представлено в BlinkLed.

**Подключаемые файлы и определение путей файлов**

Файлы загружаются относительно директории проекта, директории установки среды или относительно указанных директорий (полный путь). Слово +include” ( a n) на входе берет строку пути и выделяет имя файла из входного потока.

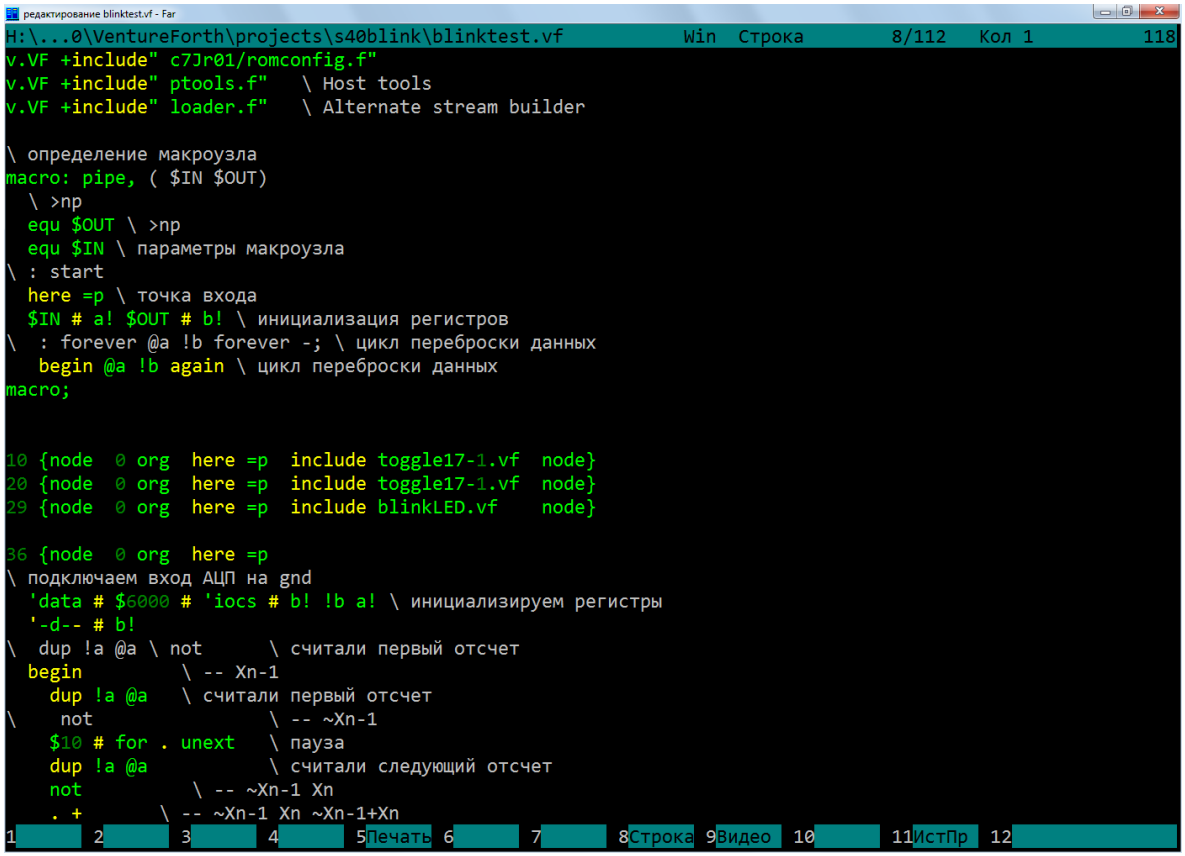


**Рисунок 5. Вид файла *Blinketest.vf*.**

Для загрузки файла, находящегося в директории проекта, используется include <filename>.

В заголовке файле подключается файл с библиотеками, специфичными для текущей версии процессора, надстройки симулятора, библиотеки для формирования загрузочного бинарного кода:

**v.VF +include" c7Jr01/romconfig.f"**

**v.VF +include" ptools.f" \ Host tools**

**v.VF +include" loader.f" \ Alternate stream builder**

**Симуляция**

Когда исходный код скомпилирован, объектный код для каждого узла располагается в виртуальном пространстве, представляющим собой содержимое ROM и RAM каждого узла. Оттуда мы можем скомпилированный объект переслать в реальный процессор, используя. Как правило, во время симуляции пропускается процесс загрузки и выполнение программы начинается так, как будто весь код уже загружен в процессор SEAforth.

Команда reset запускает симулятор так, как будто код уже в RAM SEAforth. Альтернативно, мы можем подключить тестовую плату, которая симулирует внешнее подключение. Командой power мы говорим симулятору симулировать процесс загрузки перед запуском кода приложения.

Команда simulate (короткий вариант sym) запускает процесс симуляции. После выполнения - появляется двумерный массив, отображающий статус всех ядер. Нажатие пробела выполняет один шаг симуляции работы процессора (по умолчанию, скорость – один шаг на нажатие).

**Основные команды симулятора**

- **simulate** or **sim** – запускает или останавливает процесс симуляции;
- <space-bar> - шаг симуляции;
- <any-key> - останавливает симуляцию;
- ( n) **setmax –** устанавливает максимально возможное число шагов симуляции – n. Симуляция останавливается по достижении заданного числа шагов или после нажатия на клавишу. -1 – запускает бесконечную симуляцию. Значение по умолчанию – ;.
- ( n) **setstep -** Симуляция выполняется без обновления экрана в течении n шагов по умолчанию =1;

- ( n) **upto** - Симулируется n шагов без отображения, после чего симуляция завершается. Для продолжения требуется повторный запуск simulate . В это время возможно обновление значений setmax, setsteps.
- **watch4** – вывод детальной информации по четырем выбранным ядрам, например - **03 04 05 -1 watch4** отобразит состояние 19, 20 и 5 ядер.

Следующие два рисунка отражают, как выглядит процесс симуляции в SwiftForth. Первая – перед процессом симуляции. Вторая после выполнения 273 шагов (рисунки 6, 7). На обоих рисунках, красным обозначены узлы, которые активны в данный момент. Черные узлы - спят (в состоянии покоя/низкого энергопотребления), ожидая инициализации/запуска через порт, или завершения процесса обмена данными с соседними узлами.

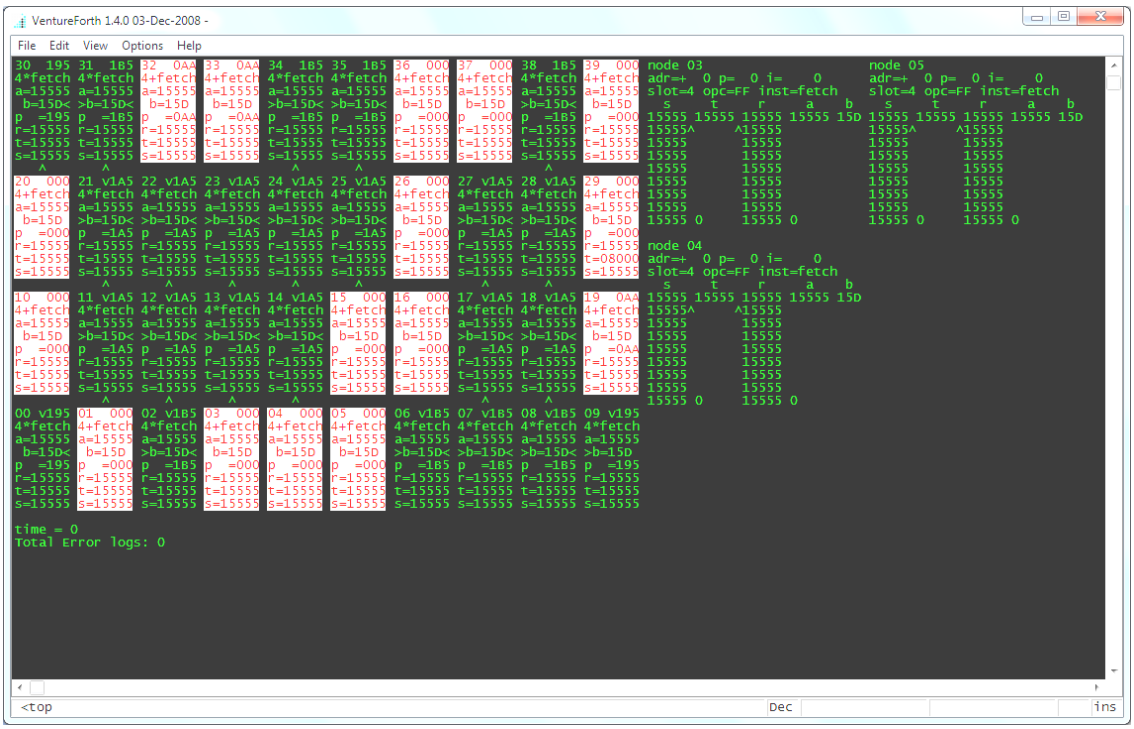


**Рисунок 6. Дисплей после запуска симуляции приложения.**

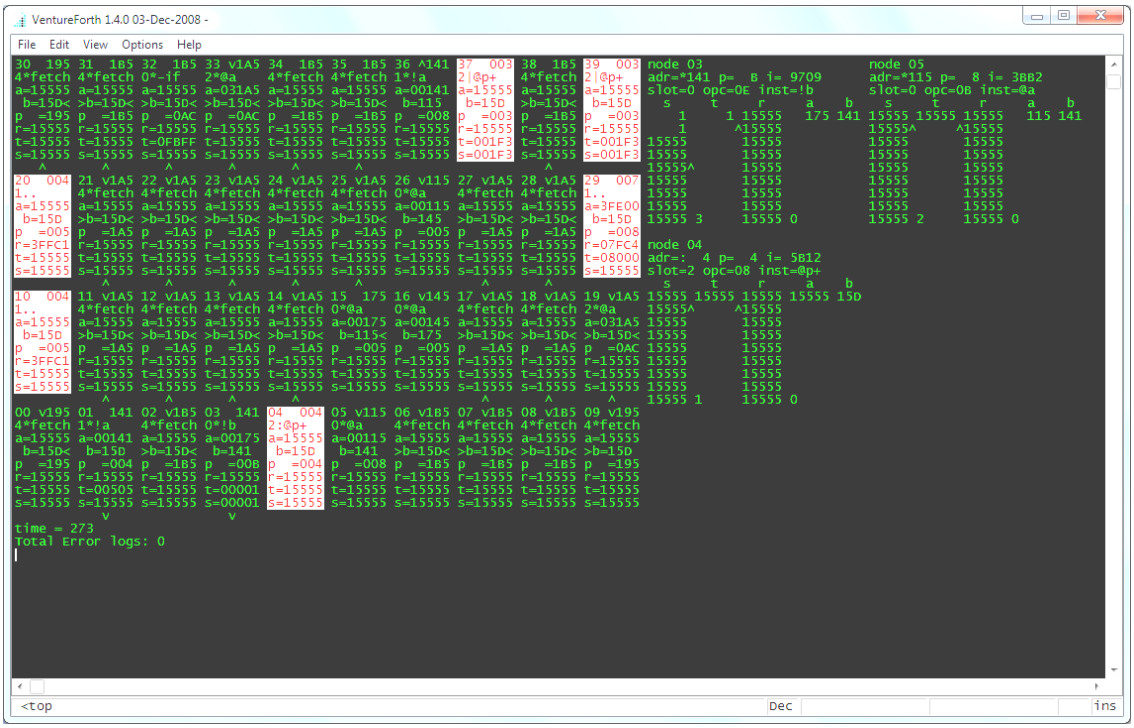


**Рисунок 7. Дисплей после 273 шагов.**

### Загрузка кода в процессор

После того, как приложение скомпилировано и отлажено в симуляторе, следующий шаг – загрузка кода в чип SEAforth. Единственный путь поместить приложение в SEAforth – послать его на один из выводов, способных загрузит приложение через внешний вывод. Каждый из узлов SEAforth, который имеет загрузочный драйвер понимает протокол, созданный для загрузки приложения. Обычно это spi интерфейс или асинхронный последовательный порт (точнее ядра, имеющие загрузчик с асинхронного порта), реже синхронные линии или интерфейсы SERDES.

Наиболее простой путь синтезировать загрузочный код для процессора - используя слова из библиотеки ptools.f , модифицированной П. Советовым [**20**].

Например, следующий пример иллюстрирует создание загрузочного образа для spi-флеш памяти для всех 40 ядер процессора:

**macro: s40>32**

**32 31 30 20 10 00 01 11**

**21 22 12 02 03 13 23 33**

**34 24 14 04 05 15 25 35**

**36 26 16 06 07 17 27 37**

**38 28 18 08 09 19 29 39**

**40 >stream**

**macro;**

**0 :xnode s40>32 StreamFlash bsave blink.bin**

Для загрузки с serial используется:

**0 :xnode s40>33 xserial serial.bin**

Аналогичное макро определение s40>33:

**macro: s40>33**

**33 32 31 30 20 21 22 23 24 25**

**26 27 28 18 17 16 15 14 13 12**

**11 10 00 01 02 03 04 05 06 07**

**08 09 19 29 39 38 37 36 35 34**

**40 >stream**

**macro;**

# Глава 3. Первые шаги программирования мультикомпьютеров SEAforth40

## Введение

Подход к программированию мультикомпьютеров SEAforth во многом существенно отличается от программирования многоядерных процессоров общего назначения. Это выражается в специфике среды разработки, структуре процессора (массовый параллелизм, распределенная память), сокращенный набор команд, небольшой объем памяти для оперативного хранения программ и данных.

## Простая арифметика

В данном разделе рассмотрим простые примеры, иллюстрирующие работу с константами, переменными и математическими операциями:

- сложение/вычитание переменной с константой;

- различные варианты умножения – сокращенный, подпрограмма, вызов слов ПЗУ.

Задача первая – написать программу, для двух соседних ядер, одно из которых складывает два числа, другое вычитает, после чего они обмениваются результатами операций.

Для начала, каждое из ядер должно настроить один из индексных регистров на порт соседнего ядра, положить на стек два числа, произвести назначенную операцию, записать результат в коммуникационный порт и получить данные от соседа (рисунок 8).

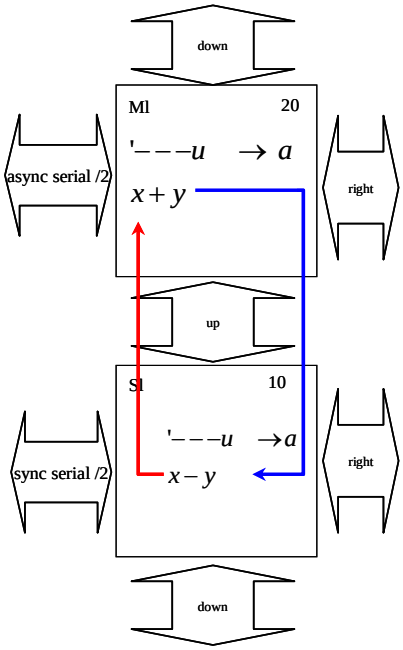


**Рисунок 8. Схема передачи данных между ядрами при обмене результатами операций.**

Возможный источник взаимной блокировки ядер – момент передачи данных. Для исключения блокировки, одно из ядер должно начать процедуру обмена с чтения порта, другое с записи. Графически процесс взаимодействия ядер достаточно удобно изображать в виде диаграммы последовательностей действий [**16**] (рисунок 9).

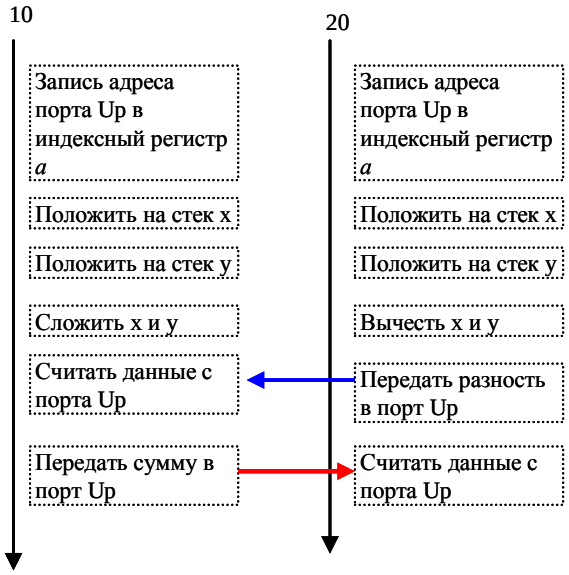


**Рисунок 9. Диаграмма взаимодействия двух ядер при обмене результатами.**

Т.к. в системе команд процессора отсутствует команда вычитания, её придется реализовать, как сложение с числом в дополнительном коде.

**\ макросы, определяющие операции сложения и вычитания чисел на стеке**

```
macro: add
 . +    \ сложение
macro;

macro: sub \ вычитание
 not 1 # . + \ перевод числа на стеке в доп. код
 add
macro;



\ зададим пару констант
10 constant x
5 constant y
\ пусть ядро 10 выполняет сложение, ядро 20 - вычитание
\ т.к. сложение завершится быстрее, 10е ядро первым выполнит чтение с порта
\ дополнительно это позволит на некоторое время снизить энергопотребление процессора :

10 {node  0 org  here =p  '---u # a!  \
 x # y # add \ x+y
 @a          \ x+y x-y
 over !a     \ x+y x-y
.
```

```
node}

20 {node  0 org  here =p  '---u # a! \
  x # y # sub \ x-y
  !a @a        \ x+y
.
node}
```

При необходимости (например, многократное использование в коде программы), вычитание можно оформить в виде слова.

```
: -
  not 1 # . + . +
;
```

Следующий интересный момент связан с выполнением операции умножения на данном процессоре. Аппаратного умножителя в процессорных ядрах нет, и умножение реализуется программно на базе команды +*.

**+ *** используется в качестве строительных блоков для перемножения двух чисел, находящихся в регистрах S и A. Работает по принципу вычисления частичных произведений (сложение-сдвиг). Результат размещается в регистрах T и A, которые работают при этом как 36-битный сдвиговый регистр. В T (вершина стека) – старшая часть, в A – младшая.

Если бит 0 (LSB) в A равен 0, то 37-бит (бит расширения знака) регистра T.A просто сдвигается вправо на один бит. Если он равен 1, содержимое S прибавляется к T, перед тем, как 37-битный регистр T.A будет сдвинут. Младший бит T сдвигается в старший бит регистра A.

Код для ядра 30 иллюстрирует беззнаковое перемножение 18-битных чисел. В A помещается один множитель, вершина стека обнуляется, второй элемент стека содержит второй множитель.

```
30 {node  0 org  here =p  \ "ручная" реализация беззнакового
умножения
```

```
'--l- # b!
\ t=0 s=x-любого знака a=y-положительный
155 # a! 2200 # \ t=4 a=5
dup dup      \ -- 4 s=4 t=4 a=5
xor   \ -- s=4 t=0 a=5
17 # for . +* unext \ -- s=x t=x*y_h a=x*y_l
@b
node}
```

В том случае, если оба или один из сомножителей имеет небольшую размерность, можно использовать следующий прием – в регистр А помещается число с наименьшим количеством разрядов, число на вершине стека сдвигается вправо на это же число разрядов и выполнить команду +* столько раз, сколько значащих разрядов в регистре А.

Пример ниже демонстрирует умножение 25 на 3.

```
00 {node  0 org  here =p  \ умножение малоразрядных чисел
 '--l- # b!
  25 # 2* 2* 3 # dup a! . +* . +*
 @b
 node}
```

Для экономии пространства ОЗУ можно использовать вызов слова * из ПЗУ ядер (есть практически во всех ядрах). Кроме всего прочего, это слово поддерживает знаковое умножение.

```
01 {node  0 org  here =p  \ работа подпрограммы умножения
 '---u # b!
 155 # -2200 # * 2/
 @b
 node}


02 {node  0 org  here =p  \ работа подпрограммы умножения
```

**'---u # b!**

**155 # 2200 # * 2/**

**@b**

**node}**

## Обмен данных между ядрами

Рассмотренная ниже задача носит учебный характер и скорее всего не будет иметь смысла вне рамок запуска её на симуляторе. Состоит она в следующем – надо организовать передачу слова между всеми ядрами процессора по цепочке – чем-то напоминает старую игру «Питон» - только роль клеток будут исполнять ядра процессоров.

Для начала зададим путь обхода процессора – рисунок 10.

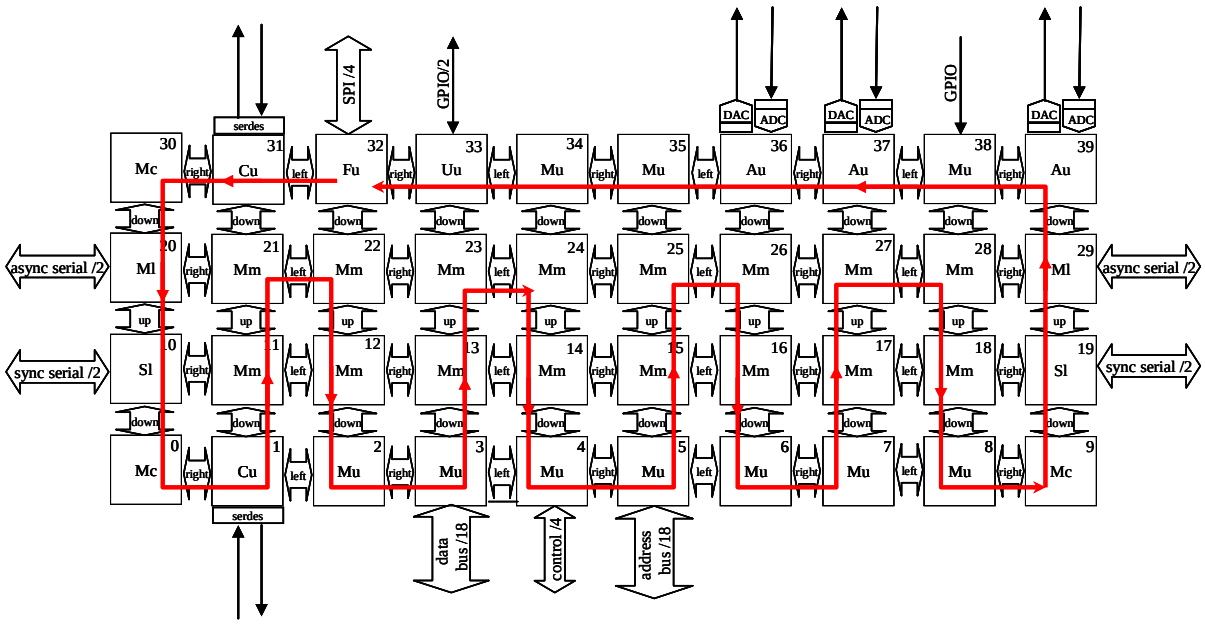


**Рисунок 10. Путь передачи слова по ядрам процессора.**

Алгоритм работы каждого из ядер предельно прост – считать слово данных от соседнего ядра, согласно месту в схеме обхода и передать следующему по обходу ядру. Исключение составит первое ядро в цепочке – оно только инициирует передачу слова.

Программный код ядер будет одинаковым - различаются только ад-

реса портов. Снизить объем работы по написанию исходного текста можно, используя макроопределения (они же макросы).

Одно из возможных решений приводится ниже.

Макрос, реализующий описанный выше алгоритм будет иметь следующий вид:

```
macro: pipel, ( $in $out -- )
: start
 here =p
  # b! # a!
  @a dup !b
macro;
```

Он задает точку старта процессорного ядра, берет со стека компиляции пару чисел – адресов портов источника и приемника слова и компилирует код, отвечающий за прием-передачу слова.

Согласно структурной схеме процессора выделим те пары портов, между которыми происходят передачи – источник-приемник. Для каждой из пар определим макрос с нужными для данной пары адресами портов:

```
macro: lr, ( $in $out )
 '--l- 'r--- pipel,
macro;
macro: rd, ( $in $out )
 'r--- '-d-- pipel,
macro;
macro: du, ( $in $out )
 '-d-- '---u pipel,
macro;
macro:       ud, ( $in $out )
 '---u '-d-- pipel,
macro;
```

**macro: ul, ( $in $out )**

**'---u '--l- pipel,**

**macro;**

**macro: dr, ( $in $out )**

**'-d-- 'r--- pipel,**

**macro;**

**macro: rl, ( $in $out )**

**'r--- '--l- pipel,**

**macro;**

**macro: lu, ( $in $out )**

**'--l- '---u pipel,**

**macro;**

Как видно из исходного кода, компилятором допускается использование вложенных макросов. Последним шагом зададим код для каждого из ядер.

**\ начальное ядро**

**32 {node 0 org here =p '--l- # b! 0 # !b 'r--- # a! @a node}**

**\ ядра с направлением передачи от левого порта правому**

**31 {node lr, node} 35 {node lr, node} 37 {node lr, node} 33 {node lr, node}**

**\ от правого - нижнему**

**30 {node rd, node} 1 {node rd, node} 3 {node rd, node} 5 {node rd, node}**

**7 {node rd, node} 9 {node rd, node}**

**\ от нижнего - верхнему**

**20 {node du, node} 11 {node du, node} 13 {node du, node} 15 {node**

**du, node}**

**17 {node du, node} 19 {node du, node}**

**\ от верхнего - нижнему**

**10 {node ud, node} 12 {node ud, node} 14 {node ud, node} 16 {node ud, node}**

**18 {node ud, node} 29 {node ud, node}**

**\ от верхнего - левому**

**21 {node ul, node} 23 {node ul, node} 25 {node ul, node} 27 {node ul, node}**

**\ от нижнего - правому**

**0 {node dr, node} 2 {node dr, node} 4 {node dr, node} 6 {node dr, node}**

**8 {node dr, node} 39 {node dr, node}**

**\ от правого - левому**

**34 {node rl, node} 36 {node rl, node} 38 {node rl, node}**

**\ от левого - верхнему**

**22 {node lu, node} 24 {node lu, node} 26 {node lu, node} 28 {node lu, node}**

Запуская на симуляторе, можно видеть примерно следующее – область с активными ядрами перемещается согласно заданному нами пути (рисунок 11).

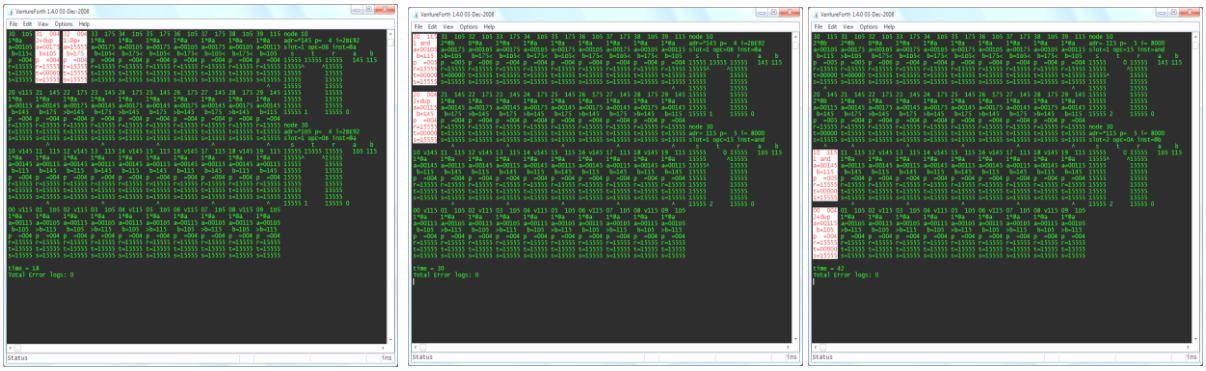

**Рисунок 11. Перемещение слова по ядрам процессора.**

**Работа с периферийными устройствами**

**АЦП**

Запустим АЦП на ядре 36 и данные, генерируемые на нем будем выдавать в параллельном виде на линиях ядра №5. В этом случае в работе приложения будут задействованы несколько ядер (а именно – 36, 26, 16, 15 и 5-е), выполняющих каждое свою задачу. Ядро 36 считывает данные с АЦП, ядра 26, 16, 15 служат для передачи значений АЦП 5-му ядру, и ядро 5 по мере поступления данных выставляет их в параллельный порт. Путь данных показан на рисунке 12.

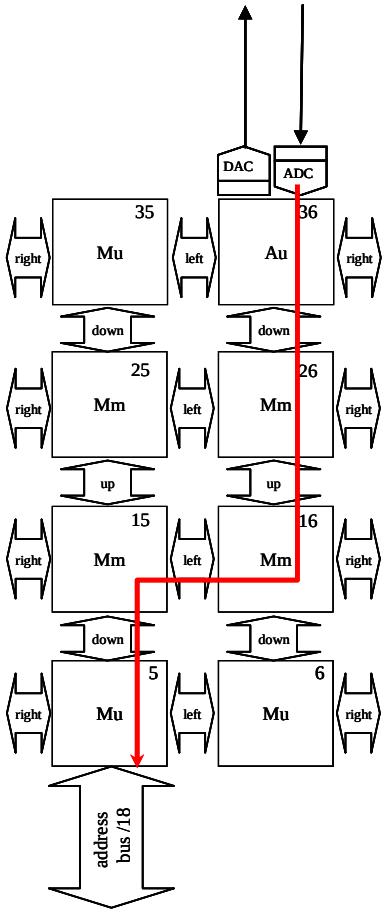


**Рисунок 12. Вывод данных с АЦП на параллельный порт.**

Диаграмма последовательностей выглядит следующим образом – рисунок 13.

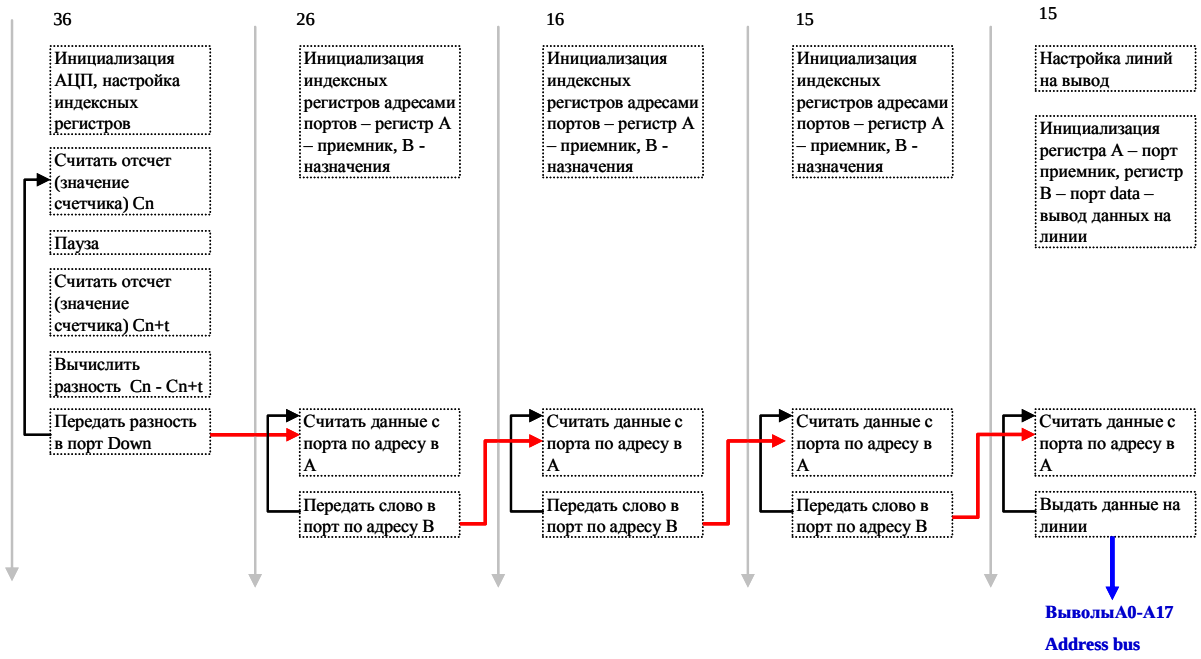


**Рисунок 13. Диаграмма последовательностей работы ядер.**

**SERDES**

Процессор обладает двумя высокоскоростными последовательными интерфейсами –SERDES, расположенными в двух ядрах по разные стороны процессора – в 1м и в 31м. Средняя скорость передачи данных по SERDES около 400 Мбит/с, режим передачи полудуплексный.

Для выдачи данных через SERDES последовательность действий следующая – надо записать передаваемое слово по адресу DATA, включить кольцевой тактовый генератор приемопередатчика (запись $20000 по адресу IOR), записать любое слово по адресу UP (ожидание окончания передачи предыдущего слова), выключить тактовый генератор после задержки в 18 тактов.

Пример бесконечного цикла выдачи значения через SERDES

```
01 {node 0 org here =p
 'data # a! \ инициализировать регистр a значением DATA
 Begin
   $0505 # !a        \ записать число по адресу в регистре a (data)
```

```
  $20000 # 'iocs # b! !b \ записать в b адрес ior
                \ и записать по этому адресу код включения тактового
                генеротора
  0 # dup '---u # b! !b   \ пустая запись по адресу UP
  | 18 # for . unext       \ цикл ожидания окончания передачи
  'iocs # b! !b            \ выключение генератора
  | 10000 # for . unext | \ пауза между передачами (около 15 мкс)
again node}
```

# Глава 4. Базовые алгоритмы цифровой обработки сигналов

## Цифровая фильтрация - КИХ-фильтр

Одно из часто встречающихся применений данных процессоров (собственно оно является одним из целевых для процессоров этой серии) - обработка сигналов в реальном времени.

Рассмотрим один из наиболее распространенных алгоритмов цифровой обработки сигналов – цифровую фильтрацию [**17**]. В ряде областей применения популярно использовать КИХ или БИХ фильтры довольно высоких порядков – с числом коэффициентов от десяток, до нескольких сотен.

Может показаться, что ядра процессора SEAforth40 не смогут поддерживать вычисления такого рода фильтров ввиду небольшого объема памяти. Но в данном случае первое впечатление обманчиво. В большинстве случаев экономить объем ОЗУ позволяет использование функций, прошитых в ПЗУ ядер. Это не только простые арифметические операции, но и целые процедуры, наподобие аппроксимации полиномами или табличной интерполяции – в этом случае расход памяти идет только на хранение ключевых точек. Аналогичная функция разработана и для поддержки цифровой фильтрации [**17**].

Слово taps позволяет внутри определений задавать таблицу коэффициентов фильтра и его начальные значения [**5**]. Например, КИХ фильтр на одном ядре:

```
: fir-kernel 4 #
taps a0 , 0 , a1 , 0 , a2 , 0 , a3 , 0 , a4 , 0 ,
: fir ( B:in -- out ) dup dup xor @b fir-kernel drop ;
```

КИХ фильтр, задействующий несколько ядер:

**: long_fir start dup dup xor @b fir-kernel !b !b ;**

**: long_fir_mid @b push @b pop fir-kernel !b !b ;**

**: long_fir_end @b push @b pop fir-kernel drop ;**

БИХ фильтр Чебышева:

**: lp.15.2p 4 # taps $4038 , 0 , $8070 , 0 , $4038 , here 0 ,**
**$19D39 , 0 , $361E7 , 0 , ( here ) ,**
**: iir ( n -- n') push dup dup xor pop lp.15.2p drop dup !a ;**

Если ядро будет использовано приложением только для вычисления фильтра, то для хранения отсчетов сигнала и коэффициентов фильтра можно отвести порядка 50-58 слов ОЗУ. При этом для программной части останется от 6 до 14 слов, что с учетом высокой плотности бинарного кода даст 24 – 56 команд языка.

Таким образом, в одном ядре может храниться и обрабатываться от 25 до 29 отсчетов входного сигнала и столько же коэффициентов. Если использовать все ядра для вычисления фильтра, получим значение 1000 - 1150 коэффициентов. В реальной ситуации, конечно показатели могут быть меньше из-за возникающей потере точности – обработка данных ведется в целочисленном формате с 18-битной точностью. Как правило, удовлетворительные показатели достигаются при фильтрах порядка нескольких десятков.

В некоторых случаях для вычисления не изменяющихся в процессе выполнения программы значений функций можно использовать ресурсы компилятора, предварительно вычисляя нужные значения.

Для примера возьмем КИХ-фильтр нижних частот с 46 коэффициентами. В реализации фильтра задействуем три ядра процессора – первые два ядра вычисляют по 15 произведений коэффициентов с отсчетами входного

сигнала, третье остальные 16. Для проверки на отладочной плате используется ещё пара ядер – первое используется, как источник сигнала (сигнал формируется программно или формируются отсчеты АЦП), второе выдает сигнал на ЦАП.

Структурная схема вычислений с привязкой к ярдам процессора выглядит следующим образом – рис.1.

На рисунке 14 представлены осциллограммы реакции фильтра на одиночный импульс. Входной сигнал обрабатывается со скоростью, не превышающей скорости вычисления наибольшей частичной суммы *y11* или *y22*. В данном случае реализуется конвеейрный вариант параллельных вычислений – ядро 27 выдав промежуточную сумму фильтра способно принимать новый отсчет сигнала. Ядро 37 также может задавать меньшую скорость вычисления отсчетов в соответствии с требованиями приложения. Синхронизация обеспечивается за счет передачи между ядрами промежуточных сумм и отсчетов входного сигнала.

На рисунке 15 также хорошо заметно запаздывание выходного сигнала относительно входного, составляющее величину, равную произведению количества коэффициентов фильтра на время между двумя последовательными отсчетами.

Для фильтра в примере максимальная частота следования отсчетов входного сигнала составляет порядка 83 КГц. Максимальная загрузка оперативной памяти ядер 65% (занято 42 слова из 64). Потеря точности обработки сигнала при использовании 18-битной арифметики примерно 4-5% (из-за ошибки округления коэффициентов). Скорость работы фильтра можно повысить, уменьшив количество коэффициентов фильтра, рассчитываемых одним ядром, увеличив при этом количество задействованных ядер.

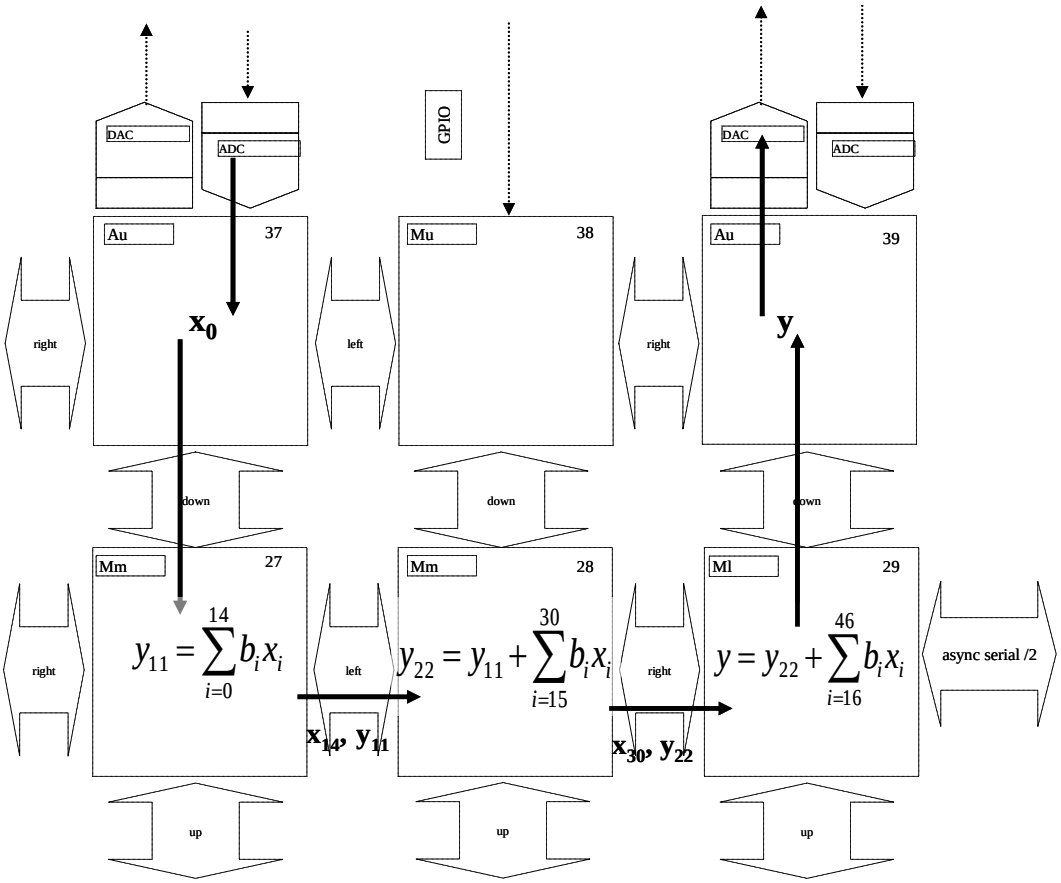


**Рисунок 14. Структурная схема вычисления КИХ фильтра на нескольких ядрах процессора SEAforth.**

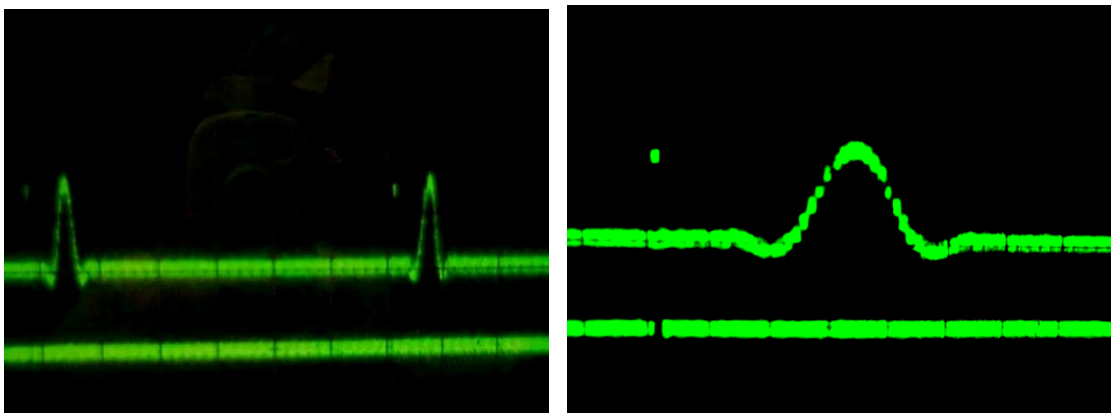

**Рисунок 15. Осциллограмма одиночного импульса исходного сигнала и импульсная характеристика фильтра.**

Исходный код ниже.

```
\ первый блок фильтра
38 {node  0 org
: fir-kernel 15 # taps a0 , 0 , a1 , 0 , a2 , 0 , a3 , 0 , a4 , 0 ,
                a5 , 0 , a6 , 0 , a7 , 0 , a8 , 0 , a9 , 0 ,
                a10 , 0 , a11 , 0 , a12 , 0 , a13 , 0 , a14 , 0 , a15 , 0 ,
: long-fir-start '--l- # b!   dup dup xor @b fir-kernel '-d-- # b! !b !b ;
  here =p
   begin
    long-fir-start
   again
node}

\ середина фильтра
28 {node  0 org
: fir-kernel 15 # taps a16 , 0 , a17 , 0 , a18 , 0 , a19 , 0 , a20 , 0 ,
                a21 , 0 , a22 , 0 , a23 , 0 , a24 , 0 , a25 , 0 ,
                a26 , 0 , a27 , 0 , a28 , 0 , a29 , 0 , a30 , 0 , a31 , 0 ,
: long-fir-mid '-d-- # b! @b push @b pop fir-kernel 'r--- # b! !b !b ;
  here =p
   begin
     long-fir-mid
   again
node}

\ выходной блок фильтра
29 {node  0 org  \ here =p
: fir-kernel 13 # taps a32 , 0 , a33 , 0 , a34 , 0 , a35 , 0 , a36 , 0 ,
                a37 , 0 , a38 , 0 , a39 , 0 , a40 , 0 , a41 , 0 ,
                a42 , 0 , a43 , 0 , a44 , 0 , a45 , 0 ,
```

```
: long-fir-end 'r--- # b!  @b push @b pop fir-kernel drop ;
  here =p
   begin
     long-fir-end '-d-- # b! !b
   again
node}

\ выдача выходных отсчетов на ЦАП
 39 {node 0 org here =p
    begin
      '-d-- # b!  @b $100 # . + $155 # xor 'iocs # a! !a
    again
 node}
```

**Вычисление преобразования Фурье через преобразование Хартли**

Введение

Рассмотрим задачу вычисления дискретного преобразования Фурье [**17**, **18**] на процессоре SEAforth40. Ограничимся вычислениями в формате фиксированной точкой - т.е. только целочисленные операции. Для вычисления преобразования Фурье требуется введение комплексной арифметики. В данном случае ограничимся только действительными числами при помощи дополнительного преобразования - преобразования Хартли, которое является чисто действительным. Имея вычисленные отсчёты преобразованиям Хартли можно получить как действительные и мнимые коэффициенты Фурье преобразования, так и спектр мощности и фазовый спектр.

Дискретное преобразование Хартли вычисляется следующим образом:

$$H(\nu) = \frac{1}{N}\sum_{0}^{N-1} f(t)\cdot cas(\frac{2\pi t\nu}{N})\,; \qquad (1)$$

где

$$cas(y) = \cos(y) + \sin(y)\,; \qquad (2)$$

*f(t)* – входной сигнал**.**

Одним из замечательных свойств этого преобразования является то, что обратное преобразование вычисляется по аналогичной формуле, за исключением масштабирующего коэффициента 1/N.

С преобразованием Фурье оно связано следующим образом.

гармоники спектра мощности:

$$\Phi_p(\nu) = \frac{H(\nu)^2 + H(N-\nu)^2}{2}; \qquad (3)$$

фазовый спектр:

$$\Phi\phi(\nu) = arctg\left(\frac{H(\nu) - H(N-\nu)}{H(\nu) + H(N-\nu)}\right). \qquad (4)$$

Таким образом, задача вычисления преобразования Фурье разбивается на два больших этапа - вычисление преобразование Хартли и уже на его основе преобразования Фурье.

Трудоемкость задачи может быть оценена в $N^2$ операций сложения и умножения на вычисление *H(v)*, плюс *3N* умножений и сложений на получение отсчётов преобразования Фурье. Прямое вычисление Фурье преобразования даст порядка $(2N)^2$ операций сложения и умножения. Для быстрого преобразования Фурье имеем *2NlogN* умножений и *3Nlog(N)* сложений. Быстрое преобразование Хартли имеет порядка *3/2Nlog(N)* сложений и *Nlog(N)* умножений.

**Разбиение задачи**

Учитывая скромные ресурсы процессора по встроенной оперативной памяти, ограничимся преобразованиями с небольшим числом *N*. Для ускорения вычислений ДПХ распределим вычисления коэффициентов преобра-

зования на различные ядра процессора - пусть каждое ядро вычисляет один или несколько коэффициентов преобразования.

Рассмотрим случай $N$=16. Для него возможен вариант, при котором одно ядро процессора вычисляет один коэффициент (аналогичная ситуация возможна и при количестве отсчётов сигнала порядка 32-х). Т.о количество операций выполняемых ядром можно оценить как $N+3$ операций сложения и умножения. Вычисления тригонометрических функций целесообразно выполнить табличным методом - каждое ядро будет иметь свой фиксированный набор значений функции cas() и arctg().

Для удобства в вычислениях будут задействованы центральные 16 ядер. Предполагается, что отсчёты сигнала поступают в процессор посредством одного из периферийных ядер. В рассмотренном ниже примере источником сигнала является ядро №10 (можно предполагать, что оно получает сигнал с внешнего АЦП), ядро №20 принимает сигнал после подачи его на преобразователь и может передать его на дальнейшую обработку другим ядрам. Ядро, вычисляющее коэффициент преобразования ждёт отсчёт входного сигнала, копирует его себе, передает следующему ядру, вычисляет произведение с накоплением. Результаты преобразования остаются в ядрах.

**Диаграммы потоков данных**

В итоге имеем следующую диаграмму потоков данных при вычислении преобразования Хартли (рисунок 16).

11е ядро вычисляет H(0), 12е - H(1), ... 28е H(8) и т.д.

На втором этапе - вычислении Фурье спектра диаграмма потоков данных для рассматриваемого N выглядит следующим образом (рисунок 17).

В силу симметрии ДПФ относительно $N/2$ для вычисления спектра мощности достаточно произвести вычисление только на половине ядер.

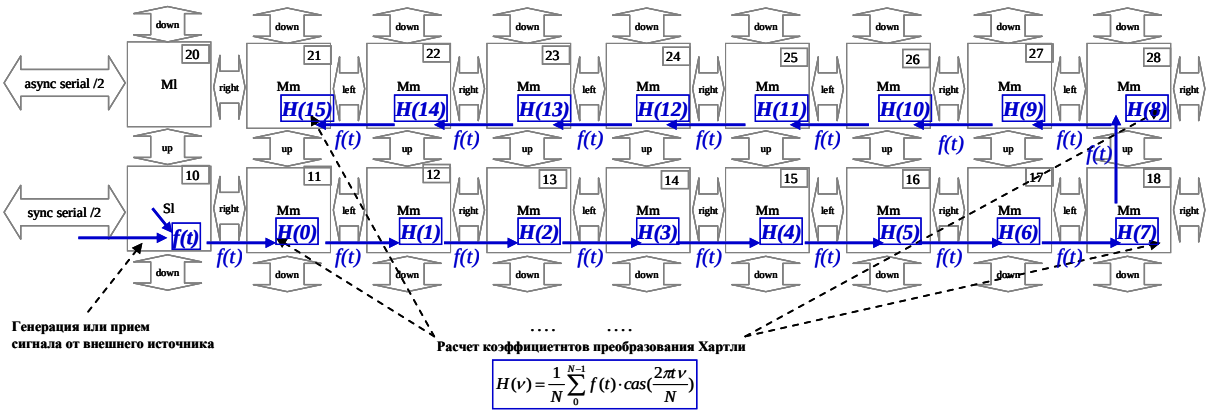


**Рисунок 16. Схема распараллеливания вычисления дискретного преобразования Хартли.**

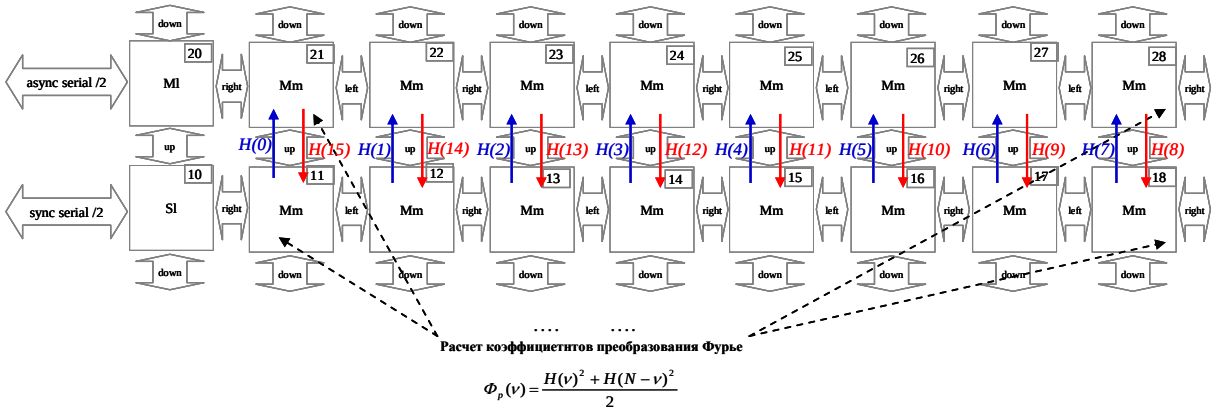


**Рисунок 17. Передачи данных между ядрами при вычислении преобразования Фурье по коэффициентам преобразования Хартли.**

### Алгоритмы работы ядер

Алгоритм работы ядер, вычисляющих коэффициенты преобразований, практически одинаков, и может быть представлен в виде последовательности нескольких шагов.

Этап 1

1. начальная инициализация, промежуточная сумма *s=0*;
2. цикл от 0 до 15 из следующих ниже шагов;
3. прием входного отсчёта сигнала - *f(t)*;
4. копирование сигнала себе;

5. передача отсчёта сигнала соседним ядрам;

6. выборка из таблицы значения *cas(vt)*;

7. вычисление произведения *p=p*cas(vt)*;

8. вычисление промежуточной суммы *s=s+p*;

9. если отсчёт не *N*-ый (16-й), возврат на начало цикла - п.3.

10. *H(v)=s*;

Этап 2

11. прием *H(N-v)*;

12. вычисление $\Phi_p(v) = \frac{H(v)^2 + H(N-v)^2}{2}$;

Ниже приводится один из вариантов реализации описанного алгоритма.

Основная программа

hart2.vf реализует этапы с 1 по 10й.
fourier2.fv этапы 11, 12.

**файл hartley_test.vf**

**Код:**

**v.VF +include" c7Gr01/romconfig.f" \ подключение функций ROM нужной версии процессора**

**include <path>\Fpmath.f \ для вычисления таблиц функции cas() подключаем библиотеку плавающей точки компилятора SwiftForth**

**16 VALUE num \ =N количество отсчетов**
**0 VALUE v \ номер отсчет Хартли спектра**

**include coef0.vf  \ определяем слова, вычисляющие cas в формате с фиксированной точкой**

**10 {node  \ ядро выдает 16 последовательных отсчётов сигнала**
**0 org**
**here =p**
**  'r--- # b!**
**1 # !b**
**1 # !b**
**1 # !b**
**1 # !b**

**0 # !b**
**0 # !b**
**0 # !b**
**0 # !b**

**1 # !b**
**1 # !b**
**1 # !b**
**1 # !b**

**0 # !b**
**0 # !b**
**0 # !b**
**0 # !b**
**node}**

```
( код для  ядер практически одинаков, за исключением портов приема
и передачи данных, определяем несколько переменных, позволяющих
настроить порты в коде для каждого из ядер
)
0 VALUE in_
0 VALUE out_

0 TO v \ - номер отсчёта X
11 {node  0 org \ установка начального адреса компиляции
\ задание таблицы коэффициентов cas(vt)
   coef , , , , , , , , , , , , , , , ,
    here *cy =p \ включение режима расширенной арифметики
   'r--- TO in_ '--l- TO out_ include hart2.vf  \ настройка переменных и
переход к вычислению коэффициента H(v)
\  -- hh hl
   '---u TO in_ '---u TO out_ include fourier2.vf  \ вычисление отсчёта
мощностного спектра Фурье - Фр(v)
\ -- fh fl
node}

1 TO v
12 {node  0 org
   coef , , , , , , , , , , , , , , , ,
    here *cy =p \ включение режима расширенной арифметики
    '--l- TO in_ 'r--- TO out_ include hart2.vf
  '---u TO in_ '---u TO out_ include fourier2.vf
node}

2 TO v
```

```
13 {node  0 org
   coef , , , , , , , , , , , , , , , ,
    here *cy =p \ включение режима расширенной арифметики
    'r--- TO in_ '--l- TO out_ include hart2.vf
  '---u TO in_ '---u TO out_ include fourier2.vf
node}

3 TO v
14 {node  0 org
   coef , , , , , , , , , , , , , , , ,
    here *cy =p \ включение режима расширенной арифметики
    '--l- TO in_ 'r--- TO out_ include hart2.vf
  '---u TO in_ '---u TO out_ include fourier2.vf
node}

4 TO v
15 {node  0 org
   coef , , , , , , , , , , , , , , , ,
    here *cy =p \ включение режима расширенной арифметики
    'r--- TO in_ '--l- TO out_ include hart2.vf
  '---u TO in_ '---u TO out_ include fourier2.vf
node}

5 TO v
16 {node  0 org
   coef , , , , , , , , , , , , , , , ,
    here *cy =p \ включение режима расширенной арифметики
  '--l- TO in_ 'r--- TO out_ include hart2.vf
  '---u TO in_ '---u TO out_ include fourier2.vf
```

```
node}

6 TO v
17 {node  0 org
   coef , , , , , , , , , , , , , , , ,
    here *cy =p \ включение режима расширенной арифметики
   'r--- TO in_ '--l- TO out_ include hart2.vf
  '---u TO in_ '---u TO out_ include fourier2.vf
node}

7 TO v
18 {node  0 org
   coef , , , , , , , , , , , , , , , , ,
    here *cy =p \ включение режима расширенной арифметики
  '--l- TO in_ '---u TO out_ include hart2.vf
  '---u TO in_ '---u TO out_ include fourier2.vf
node}

8 TO v
28 {node  0 org
   coef , , , , , , , , , , , , , , , , ,
    here *cy =p \ включение режима расширенной арифметики
  '---u TO in_ '--l- TO out_ include hart2.vf
  $3fff0 # ~u/mod \ масштабирование коэффициентов
  '---u TO out_ out_  # b! dup !b  \ передача результата ядрам, вычисля-
ющим отсчеты Фурье спектра
node}

9 TO v
```

```
27 {node  0 org
   coef , , , , , , , , , , , , , , , ,
    here *cy =p \ включение режима расширенной арифметики
  '--l- TO in_ 'r--- TO out_ include hart2.vf
  $3fff0 # ~u/mod \ масштабирование коэффициентов
  '---u TO out_ out_  # b! dup !b
node}

10 TO v
26 {node  0 org
   coef , , , , , , , , , , , , , , , ,
    here *cy =p
  'r--- TO in_ '--l- TO out_ include hart2.vf
  $3fff0 # ~u/mod \ масштабирование коэффициентов
  '---u TO out_ out_  # b! dup !b
node}

11 TO v
25 {node  0 org
   coef , , , , , , , , , , , , , , , ,
    here *cy \ включение режима расширенной арифметики
  '--l- TO in_ 'r--- TO out_ include hart2.vf
  $3fff0 # ~u/mod \ масштабирование коэффициентов
  '---u TO out_ out_  # b! dup !b
node}

12 TO v
24 {node  0 org
   coef , , , , , , , , , , , , , , , ,
```

```
   here *cy =p \ включение режима расширенной арифметики
  'r--- TO in_ '--l- TO out_ include hart2.vf
  $3fff0 # ~u/mod \ масштабирование коэффициентов
  '---u TO out_ out_  # b! dup !b
node}

13 TO v
23 {node  0 org
   coef , , , , , , , , , , , , , , , ,
   here *cy =p \ включение режима расширенной арифметики
  '--l- TO in_ 'r--- TO out_ include hart2.vf
  $3fff0 # ~u/mod \ масштабирование коэффициентов
  '---u TO out_ out_  # b! dup !b
node}

14 TO v
22 {node  0 org
   coef , , , , , , , , , , , , , , , ,
   here *cy =p \ включение режима расширенной арифметики
  'r--- TO in_ '--l- TO out_ include hart2.vf
  $3fff0 # ~u/mod \ масштабирование коэффициентов
  '---u TO out_ out_  # b! dup !b
node}

15 TO v
21 {node  0 org
   coef , , , , , , , , , , , , , , , ,
   here *cy =p \ включение режима расширенной арифметики
  '--l- TO in_ 'r--- TO out_ include hart2.vf
```

**$3fff0 # ~u/mod \ масштабирование коэффициентов**

**'---u TO out_ out_ # b! dup !b**

**node}**

**reset \ сброс процессора**

**11 watch1 10 setstep \ настройка детального просмотра состояния 11-го ядра, задали шаг симулята в 10 «тактов»**

**sim \ запуск симуляции**

**Вычисление коэффициентов Хартли**

Рассмотрим вычисление отсчетов Хартли

Ядро вычисляет сумму произведений отсчетов сигнала на функцию *cas(vt)*:

$$H(\nu) = \frac{1}{N}\sum_{0}^{N-1} f(t)\cdot cas(\frac{2\pi t\nu}{N}).$$

При необходимости масштабирование коэффициента выполняется отдельно.

Выполняются следующие действия:

1. Начальная инициализация, промежуточная сумма s=0;
2. Цикл от 0 до 15 из следующих ниже шагов;
3. прием входного отсчёта сигнала - *f(t)*;
4. копирование сигнала себе;
5. передача отсчёта сигнала соседним ядрам;
6. выборка из таблицы значения *cas(vt)*;
7. вычисление произведения *p=p*cas(vt)*;
8. вычисление промежуточной суммы *s=s+p*;
9. если отсчёт не 16й, возврат на начало цикла - п.3.
10. *H(v)=s*.

Накопление суммы ведется с 36-ти битной точностью, входной сигнал и коэффициенты cas – 18-разрядные.

**Файл hart2.vf**

**Код:**

```
0 # 0 # 0 #
\ на стеке начальная сумма в формате двойной точности и номер отсчета входного сигнала --  sh sl t
15 #  for \ начинаем цикл вычисления H(v)
  \ --  sh sl t
  dup dup xor dup . + drop \ очищаем бит переноса
  in_ # b! @b \    -- sh sl t f(t) принимаем входной отсчёт
  dup out_ # b! !b \     -- sh sl t f(t) скопировали на стек и передали следующему ядру

  \ выбираем из таблицы коэффициентов cas(vt)
  push \   сохраняем отсчет сигнала  -- sh sl i   r: -- f(t)
  a! @a+ \ -- sh sl cas; a=t+1; r: -- f(t)
  pop a@ push \ временно сохраняем номер отсчета на стеке возвратов -- sh sl cas f(t) ; r: -- t+1

   ( умножаем f(t) на cas(vt)
учитываем особенности реализации слова * - умноженная на 2 старшая часть на стеке;  - сохранение множимого во втором элементе стека;  в регистре a – младшая часть с инвертированным старшим битом
   )
  * 2/ \ умножаем и приводим к нормальному виду старшую часть произведения
```

```
\ -- sh sl cas cas*f(t) ; r: -- t+1
push drop pop \ избавляемся от множителя на второй позиции стека
a@ $20000 # xor \ помещаем на стек младшую часть
\ -- sh sl yh yl; r: -- t+1; где y=cas*f(t)

\ производим сложение чисел двойной точности
\ складываем младшие слова
push a! \ -- sh sl; r: -- yl t+1; a=yh
pop \ -- sh sl yl; r: -- t+1; a=yh
. + \ -- sh sl+yl; r: -- t+1; a=yh
\ складываем старшие слова
push  \ -- sh ; r: -- sl+yl t+1; a=yh
a@ | . + pop \ -- sh+yh+c sl+yl t+1; r: -- t+1; a=yh

pop \ возвращаем на стек сохраненный номер отсчета -- sh+yh+c sl+yl t+1; a=yh
next \ -- hvh hvl t+1
drop \  сбрасываем ненужный теперь номер отсчета t   -- hh hl - отсчет Хартли спектра
```

**Вычисление преобразования Фурье**

Вычисление коэффициентов Фурье спектра ведется на основе вычисленных ранее коэффициентов Хартли спектра. Последовательность действий следующая:

1. получаем *H(v)*;
2. принимаем от соответствующего ядра отсчет *H(N-v)*;

3. вычисляем коэффициент Фурье спектра мощности

$$\Phi_p(\nu) = \frac{H(\nu)^2 + H(N-\nu)^2}{2};$$

**Файл fourier2.vf**

**Код:**

```
\ на стеке вычисленный в hart2.vf отсчет Хартли (старшая и младшая часть) -- hh hl
  $3fff0 # ~u/mod \  масштабируем коэффициент – делим на N (N=16) -- r q ( -- r hv )
( получившийся остаток можно не учитывать, а поскольку переполнения стека в данном процессоре не бывает, можно считать ячейки стека ниже hv пустыми)
  in_ # b! @b  \   принимаем масштабированный отсчет h(N-v)   -- hv h(N-v)

\ вычисляем H(N-v)^2
  dup \ дублируем
  *     \  умножаем
\ приводим произведение к нормальному виду
  2/ push drop pop \  -- hv f2h ; a=f2l
  a@ $20000 # xor \  -- hv f2h f2l ; a=f2l
\ временно сохраняем его на стеке возвратов
  push push \ -- hv ; r: -- f2h f2l;

\ вычисляем H(v)^2
  dup  \ дублируем
  *     \  умножаем
\ приводим произведение к нормальному виду
```

```
  2/ push drop pop \
 a@ $20000 # xor \ -- f1h f1l ; r: -- f2h f2l;

\ складываем два числа с двойной точностью, одно из них находится на стеке возвратов
 pop a! pop  \ взяли младшую часть слагаемого со стека возвратов -- f1h f1l f2l; a=f2h
 . + push \ сложили, результат запомнили на стеке возвратов  -- f1h ; a=f2h ; r: -- f1l+f2l
 a@ . + pop \ сложили старшие части, младшую часть суммы вернули на стек -- Фph Фpl
\ на стеке удвоенный отсчет Фурье спектра мощности -- Фph Фpl
……..
\ далее может идти код, нацеленный на последующую обработку или передачу результатов
 '-d-- # b! @b \ останов ядра – исключительно для просмотра результата в симуляторе
```

Вспомогательный файл coef0.vf содержит слова, помогающие автоматизировать заполнение таблиц функции cas(vt).

**Файл coef0.vf**

**Код:**

```
: cas ( f: x -- ; -- cas ) \ вычисляет целочисленный вариант cas(x)
 fdup fcos fswap fsin f+ $01000 s>f f* f>s
;
: 2pivt/N ( N v t --   ; f: -- 2pivt/N ) \ формирует аргумент для cas
```

```
  s>f s>f f* s>f f/ pi 2.e f* f*
;

: coef  ( -- cas[N] cas[N-1] … cas[0] )
\ оставляет на стеке N отсчетов в обратном порядке для заполнения
таблицы cas(vt)
  0 num ?do
    num v i 2pivt/N cas
  -1 +loop
;
```

**Результаты**

Оперативная память ядер вычисляющих и $H(v)$ и $\Phi_p(v)$ загружена на 92% из которых 27% занято таблицей cas(vt).

Дамп памяти одного из ядер приведен ниже.

**Код:**

```
RAM Node 12

addr data  mnemonics/code
000  01000          --
001  014E8           ¦
002  016A1           ¦
003  014E8           ¦
004  01000           ¦
005  008A9           ¦
006  00000           ¦
007  3F757           ¦
008  3F000           } таблица коэффициентов cas(vt)
009  3EB18           ¦
```

```
00A  3E95F           ¦
00B  3EB18           ¦
00C  3F000           ¦
00D  3F757           ¦
00E  00000           ¦
00F  008A9           --
010  05D17 @p+ @p+ @p+ @p+ --------------------------------
011  00000
012  00000
013  00000
014  0000F
015  2E9B2 push . . .
016  24DE3 dup dup xor dup
017  2C1EF . + drop @p+
018  00175
019  29F97 b! @b dup @p+
01A  001D5
01B  29BBA b! !b push .
01C  2BC9A a! @a+ pop .          вычисление H(v)
01D  228B2 a@ push . .
01E  134CA call CA  *
01F  308EA 2/ push drop .
020  26E12 pop a@ @p+ .
021  20000
022  388AA xor push a! .
023  269F2 pop . + .
024  2EEB2 push a@ . .
025  2C19A . + pop .
026  27016 pop next 16
```

**027  3BDB2 drop @p+ . .  ----------**

**028  3FFF0**

**029  136AE call 2AE  ~u/mod**

**02A  04B03 @p+ b! @b dup**

**02B  00145**

**02C  134CA call CA  ***

**02D  308EA 2/ push drop .**

**02E  26E12 pop a@ @p+ .**

**02F  20000**

**030  388BB xor push push dup  вычисление Фр(v)**

**031  134CA call CA  ***

**032  308EA 2/ push drop .**

**033  26E12 pop a@ @p+ .**

**034  20000**

**035  38CAA xor pop a! .**

**036  269F2 pop . + .**

**037  2EEB0 push a@ . +**

**038  27DA2 pop @p+ b! .  ----------**

**039  00115**

**03A  00000 @b and @b +**

**03B**

**03C**

**03D**

**03E**

**03F**

Временные параметры вычислений следующие:

- вычисление коэффициента Хартли спектра ~ 2940 тактов;
- вычисление Фурье ~ 350 тактов;

- общее время выполнения ~ 3320 тактов.

С учетом того, что такт примерно соответствует 1.4нс, возможно выполнение примерно 215000 преобразований в секунду.

**Оптимизация**

Приведенный выше код для вычисления H(v) и Фр(v) не вполне соответствует форт стилю программирования и более напоминает код на ассемблере, что впрочем, для данного процессора не далеко от истины. Более оптимально, с точки зрения структурного программирования и экономии места было бы определение слов M*, D+ с вызовом их в нужных местах.

**Файл math.vf**

**Код:**

```
: M* ( x y -- ph pl )
  * 2/ \  -- x y ;
  push drop pop
  a@ $20000 # xor \ -- ph pl
;

: D+ ( ah al bh bl -- sh sl )
  push a! pop  \ -- ah al bl; a=bh
  . + push \ -- ah ; a=bh ; r: -- sl
  a@ . + pop \ -- sh sl
;
```

Код для ядер изменится следующим образом. К примеру, код для 11-го ядра:

**Код:**

```
11 {node  0 org
   coef , , , , , , , , , , , , , , , ,
   include math.vf
    here *cy =p
   'r--- TO in_ '--l- TO out_ include hart2_.vf  \  -- hh hl
   '---u TO in_ '---u TO out_ include fourier2_.vf \ -- fh fl
node}
```

Аналогично, для остальных задействованных ядер после формирования таблицы, подключаются слова M* , D+.

Код для вычисления коэффициентов Хартли и Фурье примет вид

**Файл hart2_.vf**

**Код:**

```
0 # dup dup
15 #  for
  dup dup xor dup . + drop \ очищаем бит переноса
\ --  sh sl i
  in_ # b! @b \    -- sh sl i x приняли входной отсчёт
  dup out_ # b! !b \     -- sh sl i x передали следующему
  push \     -- sh sl i   r: -- x
  a! @a+ \ -- sh sl cas; a=i+1; r: -- x
  pop a@ push \ -- sh sl cas x ; r: -- i+1
  M* \ -- sh sl yh yl; r: -- i+1; где y=cas*x
  D+
pop
next \ -- hvh hvl i+1 - отсчет Хартли спектра
```

```
  drop \  -- hh hl
\  '-d-- # b! @b \ останов ядра
```

**Файл fourier2_.vf**

**Код:**

```
\ -- hh hl \ - вычисляем Фурье спектр на основе Хартли отсчетов
  $3fff0 # ~u/mod \ -- r q ( -- r hv )
  in_ # b! @b  \   -- hv h(N-v)
  dup
  M*
  push push \ -- hv ; r: -- f2h f2l;
  dup
  M*
pop pop
D+

  '-d-- # b! @b \ останов ядра
```

При таком подходе оперативная память ядер вычисляющих и H(v) и Фр(v) загружена на 80%.

Дамп памяти одного из ядер приведен ниже.

**Код:**

```
RAM Node 12

000  01000    ---
001  014E8
002  016A1
```

```
003  014E8
004  01000
005  008A9
006  00000
007  3F757
008  3F000        }коэффициенты
009  3EB18
00A  3E95F
00B  3EB18
00C  3F000
00D  3F757
00E  00000
00F  008A9    ---
: M*
010  134CA call CA  *
011  308EA 2/ push drop .
012  26E12 pop a@ @p+ .
013  20000
014  39555 xor ;
: D+
015  2EA9A push a! pop .
016  3C88A + push a@ .
017  3CC55 + pop ;
018  04D97 @p+ dup dup @p+  ----------------------+
019  00000
01A  0000F
01B  2E9B2 push . . .
01C  24DE3 dup dup xor dup
01D  2C1EF . + drop @p+
```

```
01E  00175
01F  29F97 b! @b dup @p+
020  001D5                          вычисление H(v)
021  29BBA b! !b push .
022  2BC9A a! @a+ pop .
023  228B2 a@ push . .
024  13410 call 10  M*
025  13415 call 15  D+
026  2701C pop next 1C
027  3BDB2 drop @p+ . .        ----------------------------+
028  3FFF0
029  136AE call 2AE  ~u/mod
02A  04B03 @p+ b! @b dup
02B  00145
02C  13410 call 10  M*
02D  2E892 push push dup .          вычисление Фр(v)
02E  13410 call 10  M*
02F  26CB2 pop pop . .
030  13415 call 15  D+
031  04B00 @p+ b! @b +
032  00115                     ----------------------------+
033
034
035
036
037
038
039
03A
```

**03B**

**03C**

**03D**

**03E**

**03F**

Временные параметры вычислений следующие:

- вычисление коэффициента Хартли спектра ~ 3110 тактов;
- вычисление Фурье ~ 380 тактов;
- общее время выполнения ~ 3490 тактов.

Скорость преобразования упала до 200000 преобразований в секунду.

## Вычисление быстрого преобразования Хартли

Алгоритмы быстрого преобразования Хартли [**18**] строятся приблизительно на тех же принципах, что и преобразования Фурье. Многоточечные преобразования также строятся на основе 2-, 3- или 4-точечных преобразований, т.н. <бабочек>.

### 16-ти точечное БПХ

Для примера рассмотрим 16-точечное быстрое преобразование Хартли.

Преобразование входной последовательности f(t) начинается с её перестановки в двоично-инверсном порядке. f(t)>F(0,t). Далее элементы последовательности подвергаются трем этапам преобразований.

**1й этап**

**F(0,0)+F(0,1)->F(1,0)**

**F(0,0)-F(0,1)->F(1,1)**

**F(0,2)+F(0,3)->F(1,2)**

**F(0,2)-F(0,3)->F(1,3)**

**F(0,4)+F(0,5)->F(1,4)**

**F(0,4)-F(0,5)->F(1,5)**

**F(0,6)+F(0,7)->F(1,6)**

**F(0,6)-F(0,7)->F(1,7)**

**F(0,8)+F(0,9)->F(1,8)**

**F(0,8)-F(0,9)->F(1,9)**

**F(0,10)+F(0,11)->F(1,10)**

**F(0,10)-F(0,11)->F(1,11)**

**F(0,12)+F(0,13)->F(1,12)**

**F(0,12)-F(0,13)->F(1,13)**

**F(0,14)+F(0,15)->F(1,14)**

**F(0,14)-F(0,15)->F(1,15)**

**2й этап**

**F(1,0)+F(1,1)->F(2,0)**

**F(1,2)+F(1,3)->F(2,1)**

**F(1,0)-F(1,1)->F(2,2)**

**F(1,2)-F(1,3)->F(2,3)**

**F(1,4)+F(1,5)->F(2,4)**

**F(1,6)+F(1,7)->F(2,5)**

**F(1,4)-F(1,5)->F(2,6)**

**F(1,6)-F(1,7)->F(2,7)**

**F(1,8)+F(1,9)->F(2,8)**

**F(1,10)+F(1,11)->F(2,9)**

**F(1,8)-F(1,9)->F(2,10)**

**F(1,10)-F(1,11)->F(2,11)**

**F(1,12)+F(1,13)->F(2,12)**

**F(1,14)+F(1,15)->F(2,13)**

**F(1,12)-F(1,13)->F(2,14)**

**F(1,14)-F(1,15)->F(2,15)**

**3й этап**

**F(2,0)+F(2,4)->F(3,0)**

**F(2,1)+rF(2,5)+rF(2,7)->F(3,1)**

**F(2,2)+F(2,6)->F(3,2)**

**F(2,3)-rF(2,7)+rF(2,5)->F(3,3)**

**F(2,0)-F(2,4)->F(3,4)**

**F(2,1)-rF(2,5)-rF(2,7)->F(3,5)**

**F(2,2)-F(2,6)->F(3,6)**

**F(2,3)+rF(2,7)-rF(2,5)->F(3,7)**

**F(2,8)+F(2,12)->F(3,8)**

**F(2,9)+rF(2,13)+rF(2,15)->F(3,9)**

**F(2,10)+F(2,14)->F(3,10)**

**F(2,11)-rF(2,15)+rF(2,13)->F(3,11)**

**F(2,8)-F(2,12)->F(3,12)**

**F(2,9)-rF(2,13)-rF(2,15)->F(3,13)**

**F(2,10)-F(2,14)->F(3,14)**

**F(2,11)+rF(2,15)-rF(2,13)->F(3,15)**

**4й этап**

**F(3,0)+F(3,8)->> F(4,0)=H(0)**

**F(3,1)+F(3,9)c1+F(3,15)c3->>F(4,1)=H(1)**

**F(3,2)+F(3,10)c2+F(3,14)c2->>F(4,2)=H(2)**

**F(3,1)+F(3,11)c3+F(3,13)c1->>F(4,3)=H(3)**

**F(3,4)+F(3,12)->> F(4,4)=H(4)**

**F(3,5)-F(3,13)c3+F(3,12)c1->>F(4,5)=H(5)**

**F(3,6)-F(3,14)c2+F(3,11)c2->>F(4,6)=H(6)**

**F(3,7)-F(3,15)c1+F(3,10)c3->>F(4,7)=H(7)**

**F(3,0)-F(3,8)->> F(4,8)=H(8)**

**F(3,1)-F(3,9)c1-F(3,15)c3->>F(4,9)=H(9)**

**F(3,2)-F(3,10)c2-F(3,14)c2->>F(4,10)=H(10)**

**F(3,1)-F(3,11)c3-F(3,13)c1->>F(4,11)=H(11)**

**F(3,4)-F(3,12)->> F(4,12)=H(12)**

**F(3,5)+F(3,13)c3-F(3,12)c1->>F(4,5)=H(5)**

**F(3,6)+F(3,14)c2-F(3,11)c2->>F(4,6)=H(6)**

**F(3,7)+F(3,15)c1-F(3,10)c3->>F(4,7)=H(7)**

**где**

**r=1/sqrt(2);**

**c1= cos(2pi/16)=0,924;**

**c2= cos(2pi*2/16)=0,707;**

**c3= cos(2pi*3/16)=0,383.**

Как видно из приведенных выше соотношений, основная вычислительная нагрузка идёт на операции типа сложения/вычитания и выборку

значений из памяти, особенно на первых трех этапах.

Будем предполагать, что отсчёты входного сигнала последовательно принимаются одним из ядер процессора (из внешнего или внутреннего АЦП, или, например, из памяти).

Применим конвейерную схему распараллеливания - каждое ядро или группа ядер заняты вычислением результатов отдельного этапа преобразования, с распараллеливанием операций по этапам. Выделим для проведения вычислений группу средних ядер процессора, тем самым, снижая нагрузку на периферийные ядра.

Диаграмма потоков данных двоично-инверсной перестановки для 16-ти ядер выглядит следующим образом:

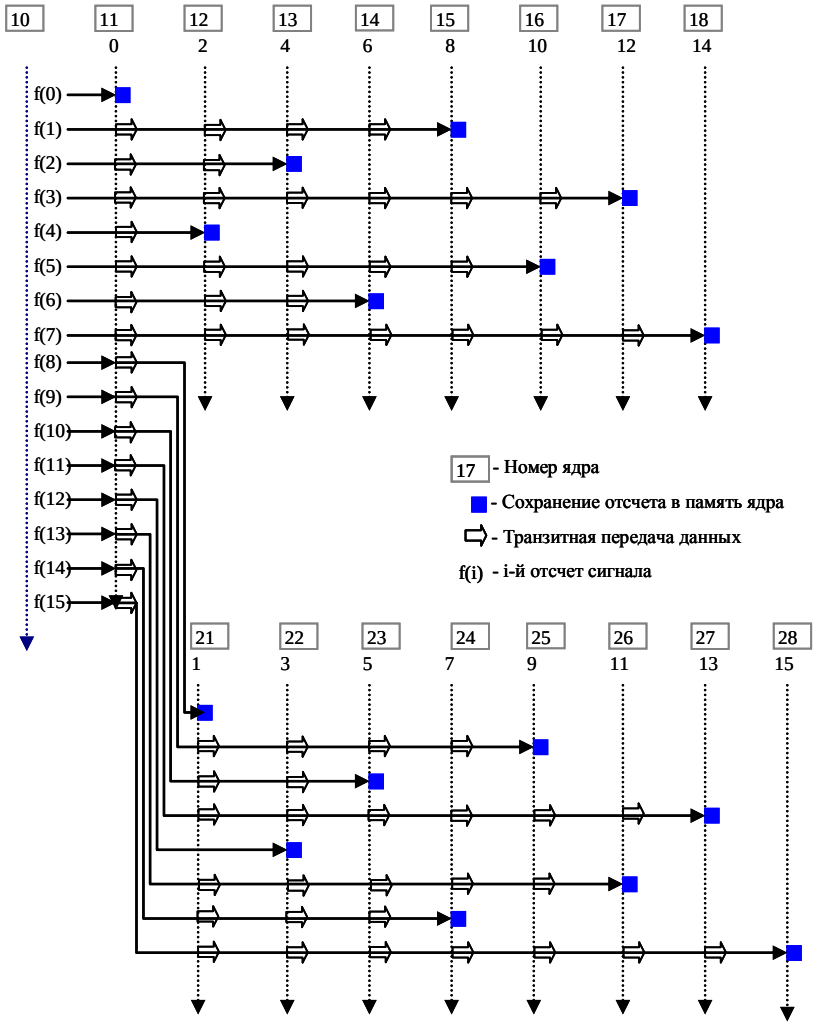


**Рисунок 18. Диаграмма последовательностей для двоично-инверсной перестановки.**

Для первого этапа имеем следующее (рис 19) – ядра с четным и с нечетными отсчетами обмениваются ими, после этого в одном случае осуществляется

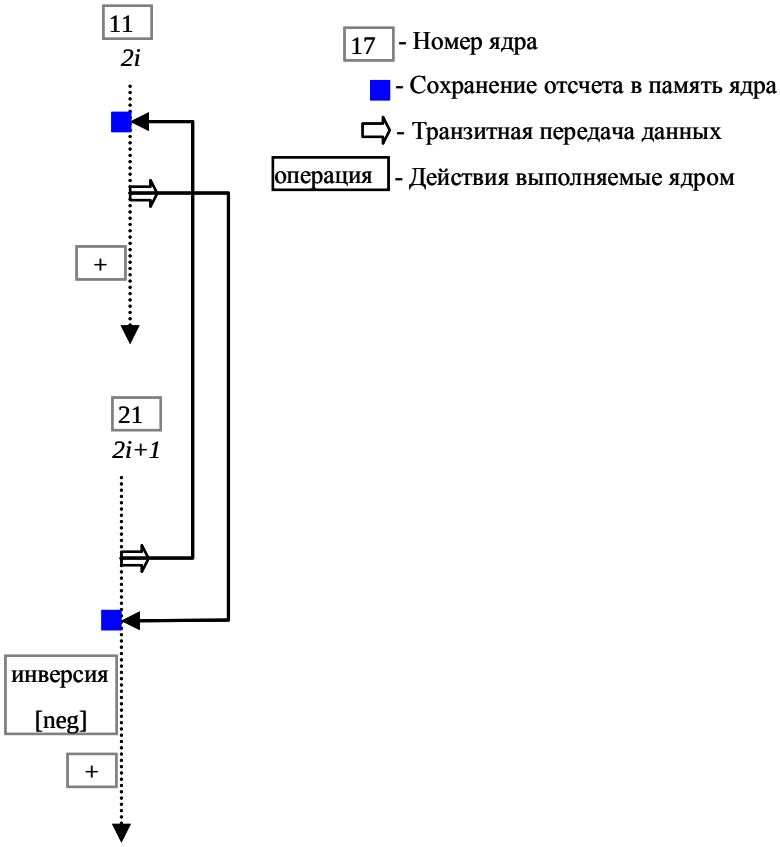


**Рисунок 19. Диаграмма последовательностей вычислений для первого этапа (действия однотипные для пар ядер с номерами 1x-2x) .**

Для второго этапа действия представлены на рисунке 20. Действия будут однотипными для пар ядер 10-11, 12-13, 14-15, 16-17 и 20-21, 22-23, 24-25, 26-27.

Для третьего этапа (рис 21) – в вычислениях будет задействовано 4 группы ядер, выполняющих 2 разные процедуры-бабочки. Это группы 11-14, 15-18 ядра и 21-24, 25-28

И заключительный четвертый этап охватит группы по 8 ядер с четными и с нечетными коэффициентами преобразования (рис 22).

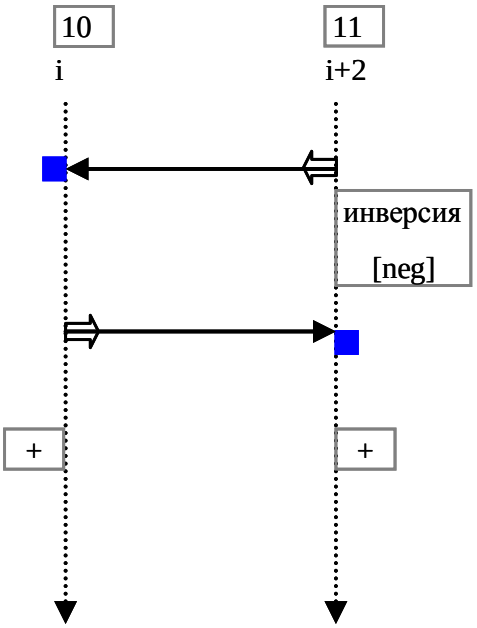


**Рисунок 20. Диаграмма последовательностей для второго этапа преобразования.**

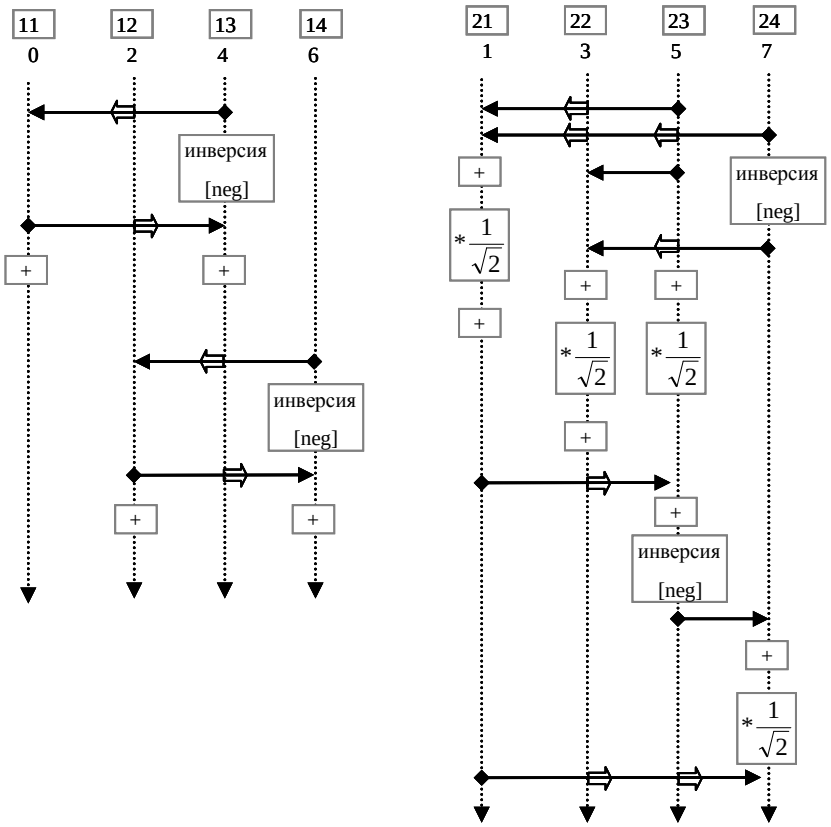


**Рисунок 21. Структура вычислений на третьем этапе преобразования.**

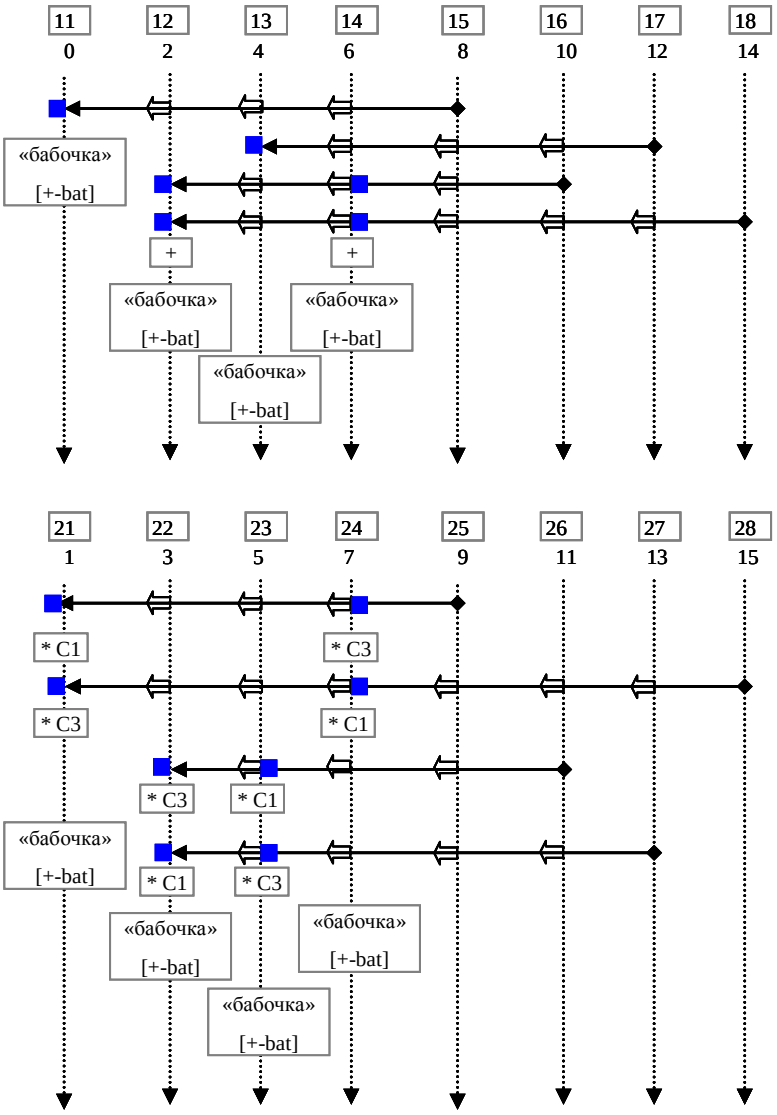


**Рисунок 22. Диаграмма последовательностей действий для финального этапа БПХ.**

В результате в ядре с номером 0 хранятся коэффициенты Хартли 0 и 8; в 1-м - 1 и 9; во 2-м - 2 и 10; в 3-м - 3 и 11; в 4-м - 4 и 12; в 5-м - 5 и 13; в 6-м - 6 и 14; в 7-м - 7 и 15.

Для примера рассмотрена реализация БПХ на средних 16-ти ядрах SEAforth40, при этом ядра 28-21 соответствуют четным индексам (0==28, 2==27, и т.д.), а ядра 18-11 нечетным индексам (1==18, 3==17, и т.д.). Ядро

38 работает генератором сигнала - выдает шестнадцать последовательных отсчетов.

При создании кода учитывались следующие ограничения - работаем в формате с фиксированной точкой одинарной точности. Масштабирование - $100 ==1.

**Файл coef0.vf**

**Код:**

```
: 2pi_k/N ( N k --   ; f: -- 2pi_k/N )
  s>f s>f f/ pi 2.e f* f*
;
: icos ( N k -- icos )
2pi_k/N cos
; IMMEDIATE
```

**Файл math.vf**

**Код:**

```
: d*-shift ( x y -- pl ) \ учитываем только младшую часть
  * drop drop
  a@ $20000 # xor \ -- ph pl
  2/ 2/ 2/ 2/ 2/ 2/ 2/ 2/ ;

: negate not 1 # . + ;
```

**Файл Hartley_test_.vf**

**Код:**

```
{
считаем, что обрабатываем данные с ацп
```

```
28-0 27-2.....21-14
18-1 27-3.....21-15
}

v.VF +include" c7Gr01/romconfig.f"


0 VALUE in_
0 VALUE in_1
0 VALUE in_2
0 VALUE out_
0 VALUE out_1
0 VALUE out_2
$b5 VALUE rr

16 VALUE num \ количество отсчетов
0 VALUE v \ текущий отсчет Хартли спектра
include Fpmath.f
include coef0.vf

  num  0  icos VALUE c0
  num  1  icos VALUE c1
  num  2  icos VALUE c2
  num  3  icos VALUE c3
  num  4  icos VALUE c4


38 {node  \ ядро - "генератор" сигнала
0 org  here =p
```

**'-d-- # b!**

**$100 # !b**

**$100 # !b**

**$0 # !b**

**$0 # !b**

**$0 # !b**

**$0 # !b**

**$0 # !b**

**$0 # !b**

**$0 # !b**

**$0 # !b**

**$0 # !b**

**$0 # !b**

**$0 # !b**

**$100 # !b**

**$100 # !b**

**node}**

**28 {node 0 org**

**\ входной порт в регистре -b ; выходной -a;**

**: 8transit @b !a : 7transit @b !a @b !a**

**@b !a**

**@b !a @b !a**

```
@b !a  @b !a ;

: a!b! ( a# b# -- ; a= b= ) b! a! ;

: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
: negate not 1 # . + ;

  here =p

\ ----перестановка----------------
    '--l- # '-d-- # a!b!
    @b \ -- f0
    7transit
    '---u # a!
    8transit

\ ----перый этап-----------------
  '---u # b! +bat

\ ----второй этап----------------
  '--l- # dup a!b! +bat

\ ----третий этап----------------
  +bat \ --f0

\ ----четвертый этап-------------
\ работаем с числами одинарной точности
\ т.е. просто отбрасываем старшую часть
\ --f0 a=l; b=l;
```

```
  @b \ -- f0 f8

  include bat.vf
  . '-d-- # b! @b \ останов ядра - только для режима симуляции

node}

27 {node

: 3transit @b !a
: 2transit @b !a : transit> @b !a ;
: transit< @a !b ;
: a!b! ( a# b# -- ; a= b= ) b! a! ;
  include math.vf

: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;
\ входной порт в регистре -b ; выходной -a;


  here =p
\ ----перестановка----------
  'r--- # '--l- # a!b!
  3transit
  @b \ -- f2
  3transit
\ ----перый этап-----------------
  '---u # dup a!b! +bat
\ ----второй этап----------------
```

```
  '--l- # dup a!b! -bat
\ ----третий этап----------------
'r--- # b! \ a=l ; b=r
  transit>
  transit<
  'r--- # a! +bat \ a=r b=r
\ ----четвертый этап--------------
\ -- f2 ; a=r b=r
  '--l- # a! \ -- f2 ; a=l b=r
  transit>      \ -- f2 ; a=l b=r
  @b @b \ -- f2 f10 f14
  . + \ -- f2 f10+f14
  c2 # d*-shift \ -- f2 [f10+f14]*c2
  include bat.vf
\ -- H2 H10

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции

node}

26 {node
: 4transit @b !a @b !a
: 2transit @b !a : transit> @b !a ;
: transit< @a !b ;
: a!b! ( a# b# -- ; a= b= ) b! a! ;

  include math.vf
: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;
```

**\ входной порт в регистре -b ; выходной -a;**

**here =p**

**\ ----перестановка----------**

**'--l- # 'r--- # a!b!**

**transit>**

**@b \ -- f4**

**4transit**

**\ ----перый этап----------**

**'---u # dup a!b! +bat**

**\ ----второй этап----------**

**'--l- # dup a!b! +bat**

**\ ----третий этап----------**

**'r--- # dup a!b! -bat \ a=r b=r**

**'--l- # b! transit> \ a=r b=l ; transit @b>>!a ;**

**transit<**

**\ ----четвертый этап----------**

**\ -- f4 ; a=r b=l ;**

**transit>**

**@b \ -- f4 f12 ; a=r b=l ;**

**2transit \ -- f4 f12 ; a=r b=l ;**

**include bat.vf**

**\ -- H4 H12**

```
  . '-d-- # b! @b       \ останов ядра - только для режима симуляции

node}

25 {node

: 3transit @b !a
: 2transit @b !a : transit> @b !a ;
: transit< @a !b ;
: a!b! ( a# b# -- ; a= b= ) b! a! ;
  include math.vf
: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;
\ входной порт в регистре -b ; выходной -a;

  here =p

\ ----перестановка----------------
  'r--- # '--l- # a!b!
  3transit
  @b \ -- f6
  transit>

\ ----перый этап-----------------
  '---u # dup a!b! +bat

\ ----второй этап----------------
  '--l- # dup a!b! -bat
```

```
\ ----третий этап----------------
  -bat

\ ----четвертый этап--------------
\ -- f6 ; a=l ; b=l
  'r--- # b! \ -- f6 ; a=l ; b=r
  2transit
  @b dup !a @b dup !a \ -- f6 f10 f14; a=l ; b=r
  negate + c2 # d*-shift     \ -- f6 [f10-f14]*c2; a=l ; b=r
  include bat.vf  \ -- H6 H14

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции

node}

24 {node

: 3transit @b !a @b !a @b !a ;
: a!b! ( a# b# -- ; a= b= ) b! a! ;

  include math.vf
: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
\ входной порт в регистре -b ; выходной -a;

  here =p

\ ----перестановка----------------
  '--l- # 'r--- # a!b!
  @b \ -- f8
```

```
  3transit

\ ----перый этап-----------------
  '---u # dup a!b! +bat

\ ----второй этап----------------
  '--l- # dup a!b! +bat

\ ----третий этап----------------
  +bat

\ ----четвертый этап-------------
\ -- f8; a=l; b=l
  'r--- # a! \ -- f8; a=r; b=l;
  dup !a \ -- f8; a=r; b=l;
  3transit

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции

node}

23 {node

: 2transit @b !a : transit> @b !a ;
: transit< @a !b ;
\ : a!b! ( a# b# -- ; a= b= ) b! a! ;
: negate not 1 # . + ;

  include math.vf
```

```
: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;

\ входной порт в регистре -b ; выходной -a;

  here *cy =p
\ ----перестановка----------------
  'r--- # '--l- # b! a!
  transit>
  @b \ -- f10
  transit>
\ ----перый этап------------------
  '---u # dup a! b! +bat
\ ----второй этап-----------------
  '--l- # dup a! b! -bat
\ ----третий этап-----------------
 'r--- # b! \ a=l ; b=r
  transit>
  transit<
  'r--- # a! +bat \ a=r b=r
\ ----четвертый этап--------------
\ -- f10 ; a=r; b=r;
  '--l- # a! \ -- f10 ; a=l; b=r;
  transit>
  dup !a \  -- f10 ; a=l; b=r;
  transit>

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции
```

```
node}

22 {node
: transit> @b !a ;
: transit< @a !b ;
: a!b! ( a# b# -- ; a= b= ) b! a! ;

  include math.vf
: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;

\ входной порт врегистре -b ; выходной -a;

  here *cy =p
\ ----перестановка----------------
  '--l- # 'r--- # a!b!
  @b \ -- f10
  transit>
\ ----перый этап------------------
  '---u # dup a!b! +bat
\ ----второй этап-----------------
  '--l- # dup a!b! +bat
\ ----третий этап-----------------
  'r--- # dup a!b! -bat \ a=r b=r
  '--l- # b! transit> \ a=r b=l ; transit @b>>!a ;
  transit<
\ ----четвертый этап--------------
\ -- f12 ; a=r ; b=l
  dup !a
```

```
  transit>

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции

node}

21 {node
: a!b! ( a# b# -- ; a= b= ) b! a! ;
  include math.vf
: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;

\ входной порт в регистре -b ; выходной -a;

  here *cy =p
\ ----перестановка----------------
  'r--- # '--l- # a!b!
  @b \ -- f10
\ ----перый этап------------------
  '---u # dup a!b! +bat
\ ----второй этап-----------------
  '--l- # dup a!b! -bat
\ ----третий этап-----------------
  -bat
\ ----четвертый этап--------------
\ -- f14; a=l; b=l;
  dup !a
  . '-d-- # b! @b       \ останов ядра - только для режима симуляции
```

```
node}
\ ------------------------------------------------------------------
\ ---- нечетные номера-----------------------------------------------

18 {node  0 org
: 7transit
@b !a @b !a
@b !a  @b !a @b !a
@b !a  @b !a ;

: a!b! ( a# b# -- ; a= b= ) b! a! ;

include math.vf
: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;

\ входной порт в регистре -b ; выходной -a;

  here *cy =p
\ ----перестановка----------
    '--l- # '---u # a!b!
    @b \ -- f1
    7transit
\ ----первый этап-----------------
  '---u # dup a!b! -bat
\ ----второй этап----------------
  '--l- # dup a!b! +bat \ -- f(2,1)
\ ----третий этап----------------
\ -- f; a=l;b=l;
```

```
  dup push \ -- f ; r: -- f
  @b @b . + rr # d*-shift \ -- f [5+7]*r ; r: -- f
  dup dup xor dup . +  drop .
  + \ f3
  pop  \ f3 f
  dup !b !b
\  . '-d-- # b! @b
\ ----червертый этап----------
\ -- f1 ; a=** ; b=l
  @b c1 # d*-shift \ -- f1 f9*c1  ; a=** ; b=l\
  @b c3 # d*-shift      \ -- f1 f9*c1 f15*c3 ; a=** ; b=l
  +                     \ -- f1 f9*c1+f15*c3 ; a=** ; b=l
  include bat.vf

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции
node}

17 {node

: 3transit @b !a : 2transit @b !a : transit> @b !a ;

include math.vf
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;
\ входной порт в регистре -b ; выходной -a;

  here *cy =p
\ ----перестановка----------------
  'r--- # '--l- # b! a!
  3transit
```

```
  @b \ -- f2
  3transit
\ ----первый этап-----------------
  '---u # dup a! b! -bat

\ ----второй этап----------------
  '--l- # dup a! b! -bat

\ ----третий этап----------------
  'r--- # b! 2transit          \ -- f(2,3)
  @b @b . + rr # d*-shift \ -- f [5+7]d*r ; r: -- f
  dup dup xor dup . +  drop .  \ очищаем бит переноса
  + \ f3
  'r--- # '--l- # b! a! 2transit \ a=r; b=l;
\  . '-d-- # b! @b

\ ---четвертый этап-------------
  '--l- #  'r--- # b! a!
\ -- f3 ; a=l ; b=r
  2transit
  @b c3 # d*-shift \ -- f1 f11*c1  ; a=** ; b=l\
  @b c1 # d*-shift      \ -- f1 f11*c1 f13*c3 ; a=** ; b=l
  +                     \ -- f1 f11*c1+f13*c3 ; a=** ; b=l
  include bat.vf
\ -- H3 H11

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции
node}
```

```
16 {node

  include math.vf
: negate not 1 # . + ;
: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;
\ входной порт в регистре -b ; выходной -a;



  here *cy =p
\ ----перестановка----------------
  '--l- # 'r--- # b! a!
  @b !a \ transit>
  @b \ -- f4
  @b !a @b !a @b !a @b !a \ 4transit

\ ----первый этап------------------
  '---u # dup a! b! -bat

\ ----второй этап-----------------
  '--l- # dup a! b! +bat \ --f(2,5)

\ ----третий этап-----------------
  dup \ -- f f ; a=l ; b=l
  'r--- # a! !a   \ -- f a=r; b=l
  @b !a \ transit>
  dup !a
  @b !a \ transit> \ -- f a=r; b=l
  dup @b \ -- f5 f5 f7
```

```
  a@ push
  . + rr # d*-shift  negate  \ -- f5 -r*[f5+f7]
  dup dup xor dup . +  drop . \ очищаем бит переноса
  pop a!
  @a . + \ -- f5 -r*[f5+f7]+f3 ; a=r;b=l
  over !b
  @a !b \ transit<
\  . '-d-- # b! @b

\ ---четвертый этап----------
  'r--- # '--l- # b! a!
\ -- f5 ; a=r; b=l
  @b !a @b !a \ 2transit
  a@ push
  @b dup !a c1 # d*-shift
  pop a!
  @b dup !a c3 # d*-shift
  negate + \ -- f5 c1*f11-c3*f13 ; a=**; b=l
  include bat.vf
\ -- H5 H13

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции
node}

15 {node
: 3transit @b !a @b !a @b !a ;

  include math.vf
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;
```

```
\ входной порт в регистре -b ; выходной -a;

  here *cy =p

\ ----перестановка----------
  'r--- # '--l- # b! a!
  3transit
  @b \ -- f6
  @b !a \ transit>

\ ----первый этап----------
  '---u # dup a! b! -bat

\ ----второй этап----------
  '--l- # dup a! b! -bat

\ ----третий этап----------
  dup negate over \  -- f -f f a=l; b=l
  !b !b  \ -- f
  dup !b
  @b negate + rr # d*-shift
  @b . + \
\  . '-d-- # b! @b

\ ---четвертый этап----------
  '--l- # 'r--- # b! a!
\ -- f7; a=l; b=r
  a@ dup push push
  @b dup !a c3 # d*-shift \ -- f7 f9*c3; a=**; b=r
```

```
  pop a!
  @b dup !a c1 # d*-shift \ -- f7 f9*c3 f15*c1; a=**; b=r
  negate + \ -- f7 f9*c3-f15*c1; a=**; b=r
  pop a!
  @b !a @b !a \ 2transit
  include bat.vf
\ -- H7 H15
  . '-d-- # b! @b       \ останов ядра - только для режима симуляции
node}

14 {node

: 3transit @b !a
@b !a  @b !a ;
: a!b! ( a# b# -- ; a= b= ) b! a! ;

  include math.vf
: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;

\ входной порт в регистре -b ; выходной -a;

  here *cy =p
\ ----перестановка----------
  '--l- # 'r--- # a!b!
  @b \ -- f8
  3transit

\ ----первый этап-----------------
```

```
  '---u # dup a!b! -bat

\ ----второй этап----------------
  '--l- # dup a!b! +bat

\ ----третий этап----------------
\ -- f; a=l;b=l;
  dup push \ -- f ; r: -- f
  @b @b . + rr # d*-shift \ -- f [5+7]*r ; r: -- f
  dup dup xor dup . + drop .
  + \ f3
  pop  \ f3 f
  dup !b !b
\  . '-d-- # b! @b

\ ---четвертый этап--------------
  'r--- # '--l- # a!b!
\ -- f9 ; a=r; b=l;
  dup !a
  3transit

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции
node}

13 {node

: 2transit @b !a : transit> @b !a ;
: a!b! ( a# b# -- ; a= b= ) b! a! ;
```

```
  include math.vf
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;

\ входной порт в регистре -b ; выходной -a;

  here *cy =p
\ ----перестановка----------
  'r--- # '--l- # a!b!
  transit>
  @b \ -- f10
  transit>

\ ----первый этап----------
  '---u # dup a!b! -bat

\ ----второй этап----------
  '--l- # dup a!b! -bat

\ ----третий этап----------
  'r--- # b! 2transit              \ -- f(2,3)
  @b @b . + rr # d*-shift \ -- f [5+7]d*r ; r: -- f
  dup dup xor dup . +  drop . \ очищаем бит переноса
  + \ f3
  'r--- # '--l- # a!b! 2transit
\  . '-d-- # b! @b

\ ---четвертый этап----------
  '--l- # 'r--- # a!b!
\ -- f11 ; a=l; b=r
```

```
  transit>
  dup !a
  transit>

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции
node}

12 {node

: transit> @b !a ;
: transit< @a !b ;
: a!b! ( a# b# -- ; a= b= ) b! a! ;

  include math.vf
: +bat ( f1 -- f1+f2; @b=f2 -- ; -- !a=f1  ) @b over !a . + ;
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;
\ входной порт в регистре -b ; выходной -a;

  here *cy =p

\ ----перестановка----------------
  '--l- # 'r--- # a!b!
  @b \ -- f10
  transit>

\ ----первый этап-----------------
  '---u # dup a!b! -bat

\ ----второй этап-----------------
```

```
  '--l- # dup a!b! +bat

\ ----третий этап----------------
  dup \ -- f f ; a=l ; b=l
  'r--- # a! !a   \ -- f13 a=r; b=l
  transit> dup !a transit> \ -- f13 a=r; b=l
  dup negate @b \ -- f13 -f13 f15
  a@ push
  . + rr # d*-shift  \ -- f13 r*[-f13+f15]
  dup dup xor dup . +  drop . \ очищаем бит переноса
  pop a!
  @a . + \ -- f13 r*[-f13+f15]+f9 ; a=r;b=l
  over !b
  transit<
\  . '-d-- # b! @b

\ ---четвертый этап--------------
\ -- f12 ; a=r; b=l;
  transit>
  dup !a

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции
node}

11 {node
: a!b! ( a# b# -- ; a= b= ) b! a! ;
  include math.vf
: -bat ( f1 -- f1-f2; @b=f2 -- ; -- !a=f1  ) dup !a negate @b . + ;
\ входной порт в регистре -b ; выходной -a;
```

```
  here *cy =p
\ ----перестановка----------
  'r--- # '--l- # a!b!
  @b \ -- f10
\ ----первый этап----------
  '---u # dup a!b! -bat
\ ----второй этап----------
  '--l- # dup a!b! -bat
\ ----третий этап----------
  dup negate over \  -- f -f f a=l; b=l
  !b !b  \ -- f
  dup !b
  @b negate + rr # d*-shift
  @b . + \
\  . '-d-- # b! @b
\ ---четвертый этап----------
  '--l- # dup a!b!
  dup !a
\ -- f15

  . '-d-- # b! @b       \ останов ядра - только для режима симуляции
node}

   reset     \ reset prepares the system to run the code in the Simulator
            \ enter  simulate  or  sim   to start the simulator
   cr
\   14 13 12 11 watch4    1 setstep
```

**28 27 26 25 watch4  1 setstep**

**sim**

**Результаты**

Временные параметры вычислений следующие:

- вычисление коэффициента Хартли спектра ~ 1150 тактов или около 620000 преобразований в секунду.

Распараллелив вычисления на 4-м этапе скорость преобразования можно еще несколько поднять.

Наиболее загруженное по коду ядро - ядро номер 16:

**Код:**

**RAM Node 16**

**: d*-shift**

**000 31JS 134CA call CA  ***

**001 NNR8 3A28F drop drop a@ @p+**

**002 QLAK 20000**

**003 MIIS 387C2 xor 2/ 2/ .**

**004 IIIS 307C2 2/ 2/ 2/ .**

**005 III0 307C5 2/ 2/ 2/ ;**

**006 J8SK 33DB0 not @p+ . +**

**007 ALAG 00001**

**008 0000 15555 ;**

**: negate**

**009 J8SK 33DB0 not @p+ . +**

**00A ALAG 00001**

**00B 0000 15555 ;**

**: +bat**

**00C AQFS 00F2A @b over !a .**

**00D K000 3D555 + ;**

**: -bat**

**00E OF3G 25A49 dup !a call 9  negate**

**00F ASK0 009F5 @b . + ;**

**010 88US 05DA2 @p+ @p+ b! .**

**011 AK40 00175**

**012 AKG0 001D5**

**013 VAFS 2BF2A a! @b !a .**

**014 AAFS 01F2A @b @b !a .**

**015 AFAS 01A02 @b !a @b .**

**016 FAF8 0BF2F !a @b !a @p+**

**017 AK20 00145**

**018 OVUS 24AA2 dup a! b! .**

**019 31BC 1340E call E  -bat**

**01A 8OVS 04DAA @p+ dup a! .**

**01B AK40 00175**

**01C U3B4 2960C b! call C  +bat**

**01D O8VS 25DAA dup @p+ a! .**

**01E AKG0 001D5**

**01F FAFO 0BF2B !a @b !a dup**

**020 FAFO 0BF2B !a @b !a dup**

**021 ARTS 00EBA @b a@ push .**

**022 K8SS 3DDB2 + @p+ . .**

**023 ALS0 000B5**

**024 31AK 13400 call 0  d*-shift**

**025 31BG 13409 call 9  negate**

**026 OOMO 24DE3 dup dup xor dup**

**027 SKNS 2C1EA . + drop .**

**028 PVBS 26A0A pop a! @a .**

**029 KQES 3CF22 + over !b .**

**02A BE88 03B17 @a !b @p+ @p+**

**02B AKG0 001D5**

**02C AK40 00175**

**02D UVAS 28A02 b! a! @b .**

**02E FAFS 0BF2A !a @b !a .**

**02F RTAO 22803 a@ push @b dup**

**030 F8SS 0BDB2 !a @p+ . .**

**031 ALN0 000ED**

**032 31AK 13400 call 0  d*-shift**

**033 PVAO 26A03 pop a! @b dup**

**034 F8SS 0BDB2 !a @p+ . .**

**035 AL6S 00062**

**036 31AK 13400 call 0  d*-shift**

**037 31BG 13409 call 9  negate**

**038 KQQS 3CF82 + over over .**

**039 31BG 13409 call 9  negate**

**03A KTSK 3C8B0 + push . +**

**03B PS8S 26912 pop . @p+ .**

**03C AK80 00115**

**03D UAAK 29F00 b! @b @b +**

**03E 31VG**

**03F 31VG**

В среднем, загрузка RAM задействованных ядер порядка 70-80%.

# Глава 5. Некоторые интересные программные трюки

В данной главе приводятся нестандартные программные решения, иллюстрирующие неочевидные аспекты программирования процессоров SEAforth **[19]**.

Счетчик единичных бит в слове:

```
: bc0 ( n -- c )
 dup dup xor .
: bc1 ( n c' -- c )
 not push . .
: bc2 ( n r:c' -- c )
 begin
 dup push zif \ 'zif' is just forward next' to 'then' below
 drop pop not ;
 then
 pop and next
```

Вычисление среднего вектора
C = (A + B) / 2, A,B,C вектора

```
2 base ! 011111011111011111 constant mask \ 6:6:6 mask,
                            \ use 011111110111101111 for 8:5:5, etc
hex

: average ( a b -- c )
  over over and  .
  push xor  2/   mask #
```

```
 and  pop  . +    ;                    \ <6 ripple
```

Табличная интерполяция

B = F(A) : A = Ah.Al -> B = Fh[Ah] + Al*Fl[Ah]

```
: interpolate ( a - a' b )
 dup 2/ 2/ .
 2/ 2/ a! $0F
 and 2* 2* .
 2* 2* @a +*
 +* +* +* ;
```

Ротация бит в слове:

```
: bit-rotate ( a n -- b ) \ b is a's n left cycle rotation
 push . . .
: Loop
 not -if
 not 2* [ ' Loop ] next ; then
 2* not [ ' Loop ] next ;
```

Генератор псевдослучайных чисел

```
: rnd ( r -- r' )
 -if
 2* $2cd81 xor ;
 then
 2* ;
```

**: poll ( - )**

**@b $200 # and if \ Is write request set?**

**' - - - u # b! \ “Up” neighbor port to B**

**@b push ;: \ call in B to R; execute it.**

**'iocs # b! then \ Restore IOCS address to B**

**drop ; \ Discard 'if's' argument & return**

КИХ фильтр на одном ядре:

**: fir-kernel 4 # taps: a0 , 0 , a1 , 0 , a2 , 0 , a3 , 0 , a4 , 0 ,**

**: fir ( B:in - - out ) dup dup xor @b fir-kernel drop ;**

КИХ фильтр, задействующий несколько ядер:

**: long - fir - start dup dup xor @b fir - kernel !b !b ;**

**: long - fir - mid @b push @b pop fir - kernel !b !b ;**

**: long - fir - end @b push @b pop fir - kernel drop ;**

БИХ фильтр

**: lp.15.2p 4 # taps: $4038 , 0 , $8070 , 0 , $4038 , here 0 ,**

**$19D39 , 0 , $361E7 , 0 , ( here ) ,**

**: iir ( n - n') push dup dup xor pop lp.15.2p drop dup !a ;**

Вычисляемый переход (некоторый аналог CASE):

**: switch ( case -- ) pop + push ;**

Пример: **@b switch handle0 -; handle1 -; handle2 -; handle3 -; \ table of**

**jump opcodes**

**: switch2 ( case -- ) pop + a! @a push ;**

Пример: **@b switch2 handle0 handle1 handle2 handle3 \ table of addresses (call opcode ignored)**

**Заключение**

Процессоры SEAforth являются достаточно мощным решением для встраиваемых систем реального времени за счет высокой скорости исполнения команд и высокой степени параллелизма, недоступной многим другим процессорам. Неоспоримым плюсом является низкое энергопотребление, особенно в пересчете на количество операций в секунду. Система команд доведена до определенного минимума, по функциональности и удобству ее можно сравнивать с системой RISC-контроллеров, таких как PIC16xx, z86, z89. Набор периферийных устройств, по сравнению с современными контроллерами, несомненно, мал и сводится в основном к устройствам передачи данных и преобразователям сигнала. Упор сделан на программную эмуляцию необходимой периферии.

Несмотря на свои недостатки, матричные мульти-компьютеры являются прекрасным средством для развития навыков параллельного программирования. Особенно это касается продумывания алгоритмов, распределения вычислительной нагрузки, обеспечения когерентности данных и их передач между ядрами и процессами.

**Библиография**

1. IntellaSys — SEAforth-24.
http://www.intellasys.net/index.php?option=com_content&task=view&id=35&Itemid=63

2. IntellaSys — SEAforth 40C18.
http://www.intellasys.net/index.php?option=com_content&task=view&id=60&Itemid=75

3. Leslie O. Snively. RF Processing Using SEAforth.
http://www.intellasys.net/index.php?option=com_content&task=view&id=24&Itemid=43

4. Leslie O. Snively. SEAforth In Industrial Control and Sensing Applications.
http://www.intellasys.net/index.php?option=com_content&task=view&id=24&Itemid=43

5. New VentureForth Programmers Guide.
http://www.intellasys.net/index.php?option=com_content&task=view&id=57&Itemid=68

6. SEK 40C18 DataSheet 1.1
http://www.intellasys.net/index.php?option=com_content&task=view&id=57&Itemid=68

7. Технология picoPower для восьмиразрядных RISC-микроконтроллеров AVR. http://www.gaw.ru/html.cgi/txt/doc/micros/avr/pico_power/start.htm

8. 32-разрядные микроконтроллеры / ЦПОС семейства AVR32.
http://www.gaw.ru/html.cgi/txt/ic/Atmel/micros/avr_32/start.htm

9. Семейство микроконтроллеров MSP430 Texas Instruments.
http://www.gaw.ru/html.cgi/txt/ic/Texas_Instruments/micros/msp430/start.htm

10. 32-разрядные высокопроизводительные RISC-процессоры семейства

ARM. http://www.gaw.ru/html.cgi/txt/doc/micros/arm/arh/index.htm

11. Пономарев В. «Новые микроконтроллеры фирмы STMicroelectronics на базе ядра ARM Cortex-M3». Электроника: Наука, Технология, Бизнес. № 6, 2007.

12. Сидоренко Б. «AVR32- микроконтроллеры для применений XXI столетия». Chip News Украина. № 8, 2008. с. 2-7.

13. Сравнительный анализ микроконтроллеров с ядром ARM. http://www.gaw.ru/html.cgi/txt/pub/micros/arm.htm

14. SEAforth (Russian) Группы Google. https://groups.google.com/group/seaforth-russian?hl=ru.

15. А. Калачев. Процессоры семейства SEAforth. // Журн. Компоненты и технологии. - 2009. - №4. - С. 66-73.

16. Технологии разработки программного обеспечения: Учебник для вузов. 3-е изд. / С.А. Орлов. – СПб.: Питер, 2004. – 527 с.

17. Калабеков Б.А. Микропроцессоры и их применение в системах передачи и обработки сигналов: Учеб. пособие для ВУЗов. – М.:Радио и связь, 1988. – 368 с.

18. Брейсуэлл Р. Преобразование Хартли: пер. с англ. - М.: Мир, 1990.- 175с.

19. IntellaSys Blog -View forum - Programming Tips. **http://www.intellasys.net/phpBB/viewforum.php?f=8&sid=7df065eef3df132e6763e94f9e27a3b8**

20. П. Советов Программирование мультикомпьютеров на кристалле семейства SEAforth. tp://peter.sovietov.com/txt/seaforth/seaforth.pdf.