# Применение векторного сопроцессора для ускорения операции быстрого преобразования Фурье

к.ф.м.н. А.А. Бурцев
ФНЦ НИИСИ РАН
burtsev@niisi.msk.ru

**Аннотация**

В ФГУ ФНЦ НИИСИ РАН в качестве расширения универсальных микропроцессоров семейства КОМДИВ создан специализированный 128-разрядный сопроцессор, позволяющий ускорять вычисления над векторами комплексных и вещественных чисел одинарной и двойной точности. В докладе представлены результаты применения этого сопроцессора, направленные на повышение скорости исполнения операции быстрого преобразования Фурье (БПФ), являющейся одной из основных в задачах цифровой обработки сигналов.

## Введение

Для поддержки информационной безопасности и технологической конкурентноспособности страны необходимо создавать современные отечественные микропроцессоры, способные обеспечивать в реальном времени высокую производительность научно-технических и инженерных расчётов даже в экстремально жёстких условиях их эксплуатации.

Для решения этой проблемы в ФГУ ФНЦ НИИСИ РАН разрабатывается семейство высокопроизводительных микропроцессоров КОМДИВ [1], в которых в качестве расширения базовой архитектуры предлагаются специализированные сопроцессоры, ориентированные на ускоренное исполнение заданного набора математических функций, наиболее часто употребляемых при решении задач определённой области применения. Такой подход к созданию суперкомпьютеров в ФГУ ФНЦ НИИСИ РАН получил название “встречной оптимизации” [2].

Так микропроцессор К64РИО (1890ВМ6) помимо обычного 64-разрядного сопроцессора (СР1) плавающей вещественной арифметики был дополнен 64-разрядным сопроцессором (СР2) комплексной арифметики, который при дальнейшем развитии был расширен до 128-разрядов и дополнен рядом полезных команд, позволяющих ускорять исполнение ряда типичных операций обработки векторов и матриц. Получившийся в результате 128-разрядный сопроцессор, условно названный векторным (CPV), как предполагается, будет функционировать в составе нового высокопроизводительного микропроцессора (1890ВМ8) семейства КОМДИВ.

На ряде тестовых задач было практически подтверждено [3], что во многих случаях применение CPV позволяет значительно ускорить типичные функции обработки векторов и матриц, характерные для задач линейной алгебры.

В данной докладе объясняется, какой выигрыш (по отношению к СР1) даёт применение этого векторного сопроцессора (CPV) для повышения скорости исполнения операции быстрого преобразования Фурье (БПФ), являющейся ключевой в задачах цифровой обработки сигналов.

## 1 Краткая характеристика векторного сопроцессора

Векторный сопроцессор (CPV) содержит 64 128-битных регистра, в каждом из которых можно хранить:

- 1 комплексное число двойной точности (DC);
- 2 комплексных числа одинарной точности (SC);

- 2 вещественных числа двойной точности (DR);
- 4 вещественных числа одинарной точности (SR).

Для каждого из этих 4-х форматов значений, помещённых в его регистры, CPV обеспечивает соответствующие ему вычислительные команды. Например, каждая из команд умножения с накоплением (см. таблицу 1) выполняет свойственную ей операцию для одной тройки (z,y,x) величин типа DC, либо двух троек типа SC или DR, либо сразу для четырёх троек типа SR.

Таблица 1. Вычислительные команды группы умножения

| 1 DC (9)* | 2 SC (7)* | 2 DR (5)* | 4 SR (4)* | cmd z,y,x |
|---|---|---|---|---|
| **cmul.d** | **cmul.s** | **vmul.d** | **vmul.s** | z=y·x |
| **cmadd.d** | **cmadd.s** | **vmadd.d** | **vmadd.s** | z=z+y·x |
| **cmsub.d** | **cmsub.s** | **vmsub.d** | **vmsub.s** | z=z-y·x |
| **cmaddsub.d** | **cmaddsub.s** | **vmaddsub.d** | **vmaddsub.s** | z=z+y·x<br>y=z-y·x |
| (t)* - *длительность в тактах (если команда уже в кэше)* | | | | |

Длительность вычислительных команд зависит от типа обрабатываемых величин (см. в скобках: 4 такта для SR, 5 для DR, 7 для SR и 9 для DC). Но поскольку CPV разрешает на каждом такте начинать исполнение новой команды вычислительного потока, то в итоге можно добиться такой максимальной производительности CPV, при которой n его вычислительных команд смогут исполниться всего за n+8 тактов.

Команды CPV для работы с памятью (см. таблицу 2) позволяют загрузить 128-битное значение целиком в регистр или заполнить его старшую и младшую половину 64-битными значениями по отдельности. Можно одной командой (vldq) загрузить два соседних регистра двумя 128-битными смежными словами из памяти. Аналогичные команды (vsd, vsdm, vsdq) предусмотрены и для сохранения значений регистров в памяти.

Таблица 2. Команды работы с памятью

| | команды загрузки | команды сохранения |
|---|---|---|
| 256 /32 L2(5)* | **vldq**, **vldqx** | **vsdq**, **vsdqx** |
| 64 /8 L1(3)* | **vld**, **vldh**,<br>**vldx**, **vldhx**, **vldlx** | **vsd**,**vsdh**,<br>**vsdx**, **vsdhx** |
| 128 /16 L1(3)*<br>**если данные уже в L1(L2)-кэше* | **vldm**, **vldd**, **vlddh**,<br>**vldmx**, **vlddx**,<br>**vlddhx**, **vlddlx** | **vsdm**, **vsdd**, **vsddh**,<br>**vsdmx**, **vsddx**,<br>**vsddhx** |

Для указания виртуального адреса в этих командах можно использовать базовую адресацию с относительным смещением (vld) или базовую индексную (vldx). Получаемый адрес должен быть выровнен на границу обрабатываемого слова, т.е. кратен 8, 16 или 32 (см./d).

Архитектурой КОМДИВ предусмотрена возможность в каждом такте взять на исполнение две очередные команды, если эти команды разных потоков. Это позволяет совместить во времени вычислительные команды CPV над одной группой данных с командами CPV для загрузки/сохранения в/из памяти другой группы данных.

Самая продуктивная команда (**cmaddsub**), предусмотренная в CPV над комплексными числами, осуществляет 10 арифметических операций (4 умножения и 6 сложений) над вещественными числами двойной точности или 20 арифметических операций над вещественными числами одинарной точности. Именно такой командой и реализуется базовая операция цифровой обработки сигналов, называемая бабочкой Фурье (БФ).

Таким образом, векторный сопроцессор позволяет ускорить в 10 раз исполнение основной команды (**cmaddsub.d** — бабочки Фурье), многократно используемой для вычисления операции БПФ с комплексными числами двойной точности, и в 20 раз ускорить основную команду (**cmaddsub.s)**, используемую в операции БПФ с комплексными числами одинарной точности, достигая в таком случае потолка своей пиковой производительности (20 ГФлопс на частоте 1 Мгц).

Проанализируем, даёт ли это нам возможность реально добиться на практике такого же ускорения (в 10 и 20 раз соответственно) для операции БПФ?

## 2 Алгоритм БПФ

Пусть задан вектор комплексных чисел $X_N$ длиной N. Его одномерным дискретным преобразованием Фурье (ДПФ) называется комплексный вектор $Y_N$, элементы которого $Y_{m\ (m=0,1,..,N-1)}$ вычисляются по формуле:

$Y_m = \sum\{X_k \cdot W(m \cdot k,N)\}_{k=0,1,..,N-1}$

где $W(q,N) = \exp(-i \cdot 2\pi \cdot q/N)$ ( i – мнимая единица).

Вычисление ДПФ непосредственно по этой формуле требует порядка $O(N^2)$ арифметических операций с вещественными числами. Существует, однако, целый класс алгоритмов, объединённых общим названием «быстрое преобразование Фурье» (БПФ), позволяющих вычислить ДПФ для любых значений N всего за $O(N \cdot \log_2 N)$ арифметических операций.

Для выполнения БПФ будем применять вариант известного алгоритма Кули-Тьюки с прореживанием по времени для длины $N=2^P$ (степени двойки). Подробное описание и математическое обоснование такого алгоритма можно найти, например, в [4] или [5, гл.8-12, 31] или в Интернете [6,7].

В этом алгоритме вектор $Y_N$ получается на месте исходно заданного вектора $X_N$ путём многократного поэтапного выполнения над различными его парами элементов одной и той же базовой операции, так называемой «бабочки Фурье».

Бабочкой Фурье BF(A,B,V) с коэффициентом V над комплексными величинами A и B называется такая единая операция, в результате которой новые значения A′ и B′ для этих величин вычисляются на основе их предыдущих значений по формулам (см. рис. 1):

$A' = A + B \times V$, $B' = A - B \times V$

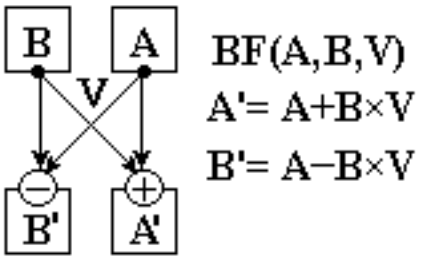


Рис. 1. Операция «бабочка Фурье»

Объясним сначала словесно применяемый алгоритм БПФ. Схематично его можно представить состоящим из двух частей (см. рис. 2).

В первой части выполняется перестановка (сортировка) элементов вектора $X_N$ в так называемом бит-реверсном порядке. Для такой бит-реверсной перестановки (БРП) каждому элементу вектора с индексом k, двоичное значение которого задаётся набором битов $k=(b_P,b_{P-1},\ldots,_2,b_1)$ длиной P ($P=\log_2 N$), сопоставляется элемент вектора с индексом m, двоичная запись которого представляется набором тех же битов, но записанных в обратном порядке $m=(b_1,b_2,\ldots,b_{P-1},b_P)$. И все такие пары элементов вектора (для которых $k \neq m$) в процессе БРП меняются местами ($X_k \leftrightarrow X_m$). Например, для вектора X длиной N=16 (P=4) будут переставлены элементы с индексами: $1(0001_2)$ и $8(1000_2)$, $2(0010_2)$ и $4(0100_2)$, $3(0011_2)$ и $12(1100_2)$, $5(0101_2)$ и $10(1010_2)$, $7(0111_2)$ и $14(1110_2)$, $11(1011_2)$ и $13(1101_2)$, а элементы с индексами: $0(0000_2)$, $6(0110_2)$, $9(1001_2)$ и $15(1111_2)$ не изменятся, т.к. у этих индексов битовые коды симметричны и потому совпадают с их бит-реверсными значениями.

Вторая часть алгоритма предполагает цикл из P ($P=\log_2 N$) этапов. На каждом t-ом этапе (t=1,..,P) вектор X длиной N рассматривается разбитым по определённому правилу на N/2 пар элементов.

Пары образуются из элементов, расположенных в массиве на расстоянии s элементов друг от друга (на t-ом этапе $s=2^{t-1}$). Сначала на 1-ом этапе (t=1) пары образуются из соседних элементов (s=1), затем на 2-ом этапе (t=2) пары образуются из элементов, отстоящих друг от

друга на 2 элемента (s=2), потом (при t=3) на 4 элемента (s=4), на 8 элементов и т.д., на t-ом этапе расстояние между парами $s=2^{t-1}$, а на последнем этапе (t=P) пары составляются из элементов, стоящих в массиве на расстоянии половины его длины ($s=2^{P-1}=N/2$).

Далее на t-ом этапе над каждой парой вида ($X_m$,$X_{m+s}$), составленной по описанному выше правилу, выполняется операция бабочки Фурье $BF(X_m,X_{m+s},V_m)$ с коэффициентом $V_m=W(m,h)=\exp(-i\cdot 2\pi\cdot m/h)$, где $h=2^t$.

Для оптимизации вычислений бабочек Фурье целесообразно распределить все пары на группы, собрав в одну k-ую группу (k=0,1,..,G-1) пары элементов вида:

$(X_{k+j\cdot h},X_{k+j\cdot h+s})$ j=0,1,..,R-1.

Заметим, что для всех пар в одной группе при вычислении бабочки Фурье будет использован одинаковый коэффициент $V_k$, т.к. $V_{k+j\cdot h}=V_k$ для любого j, ибо:

$W(k+j\cdot h,h)=\exp(-i\cdot 2\pi\cdot(k+j\cdot h)/h)=\exp(-i\cdot 2\pi\cdot k)/h)$.

На 1-ом этапе (t=1) будет всего одна группа (G=1), а в ней N/2 пар элементов (R=N/2), и один коэффициент $V_0=W(0,2)$ для всех бабочек. На 2-ом этапе (t=2) будет две группы (G=2), в каждой по N/4 пар элементов (R=N/4), и соответственно два коэффициента: $V_0=W(0,4)$ для всех пар 0-й группы и $V_1=W(1,4)$ для всех пар 1-ой группы.

На каждом следующем этапе число групп уменьшается в 2 раза, а число пар в группах удваивается, так что на t-ом этапе количество групп становится равным $G=2^{t-1}$, а количество пар в каждой группе $R=2^{P-t}$. И на последнем этапе число групп будет $G=2^{P-1}=N/2$, но в каждой группе будет всего одна пара ($R=2^{P-P}=2^0=1$).

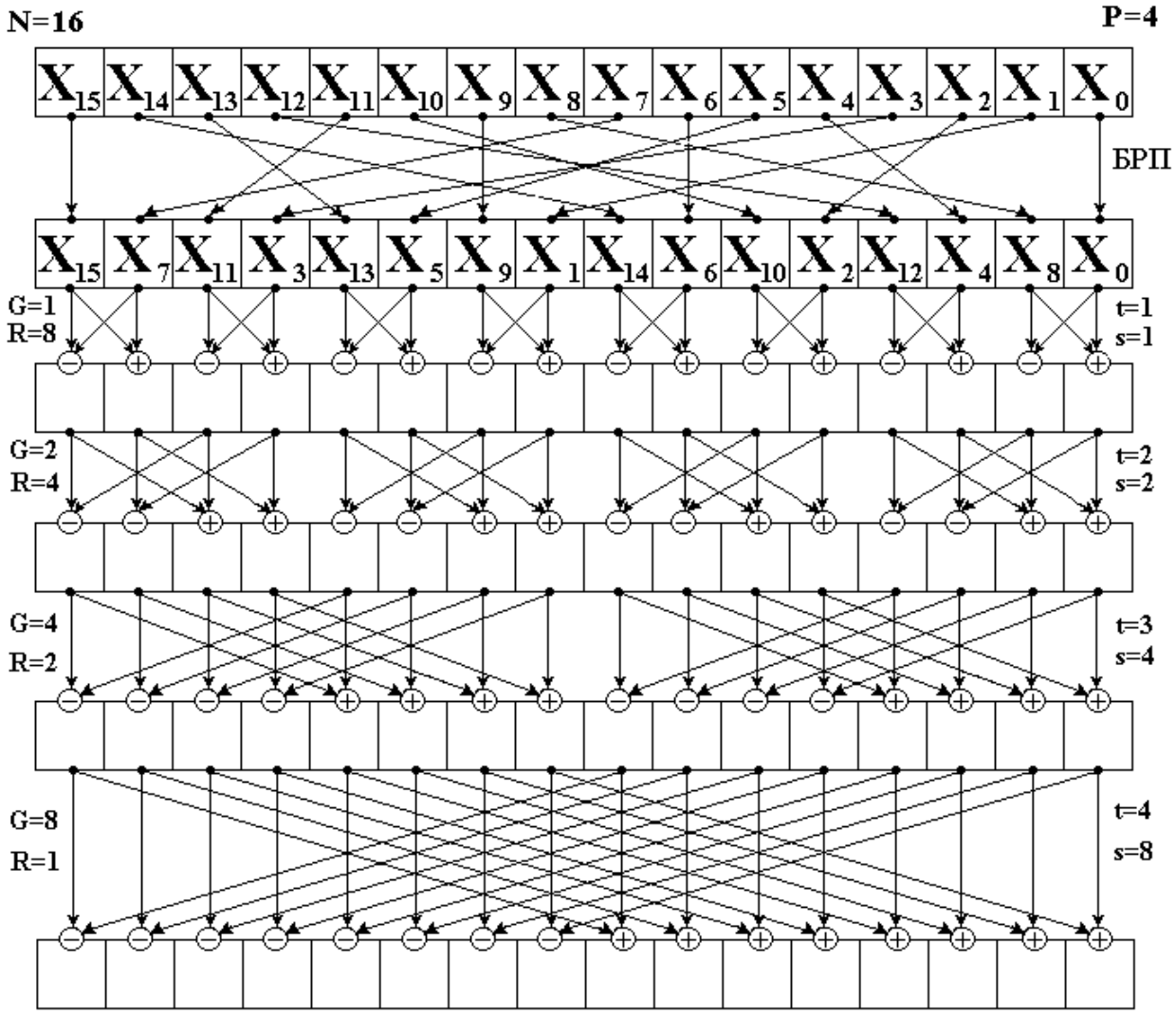


Рис. 2. Схема алгоритма БПФ для N=16

Заметим, что используемые коэффициенты можно не вычислять в процессе БПФ, а приготовить заранее в форме массива комплексных констант Cf[0:N], вычислив Cf[q]=W(q,N), полагая, что:

$W(q,N)= \exp(-i\cdot 2\pi\cdot q/N)= \cos(2\pi\cdot q/N) - i\cdot\sin(2\pi\cdot q/N)$.

А поскольку для $u= k\cdot 2^{P-t} = k\cdot N/2^t$ справедливо:

$W(u,N)= \exp(-i\cdot 2\pi\cdot k\ (N/2^t)/N))= \exp(-i\cdot 2\pi\cdot k/2^t)= W(k,2^t)$,

то используя формулу приведения: $W(k,2^t)=W(u,N)$, можно брать требуемые на t-ом этапе коэффициенты $V_k$ из таблицы Cf по индексам $u=k\cdot 2^{P-t}$ соответственно, причём величину $d=2^{P-t}$ не вычислять заново, а формировать постепенно: d:=N/2 на 1-ом этапе и далее d:=d/2 при переходе на следующий этап.

В итоге вторую часть описанного алгоритма БПФ можно представить в следующем виде (на языке Си):

```
s=1; h=2; G=1; R=N/2; d=N/2;
for(t=1, t<=P; t++ ) {
//s=2^(t-1); h=2^t; G=2^(t-1); R=2^(P-t)
  for(k=0,u=0; k<G; k++, u=u+d ) {
    V= Cf[u]; //V=exp(-I*(2*Pi*k/2^t));
    for(j=0,m=k; j<R; j++, m=m+h ) {
      BF(X[m],X[m+s],V); //{*}
    }//for j
  }//for k
  h=h*2; s=s*2; G=G/2; R=R*2; d=d/2;
}//for t
```

Теперь остаётся уточнить, какими действиями (командами, операторами) реализовать в этом алгоритме выделенную (см.**{*})** операцию бабочки Фурье.

## 3 Реализация бабочки Фурье на CP1

Чтобы вычислить бабочку Фурье BF(A,B,V), требуется получить новые значения A′ и B′ для комплексных величин на основе их предыдущих значений:

$A' = A + B\times V;\ B' = A - B\times V$

Если для такой операции не предусмотрена единая команда, то для вычисления бабочки Фурье потребуется три операции над комплексными значениями:

$T = B\times V;\ A = A + T;\ B = A - T;$

и дополнительная память для сохранения вспомогательной комплексной величины T.

Если вообще не предусмотрено арифметических команд (сложения, умножения) над комплексными числами, то выполнение нужных операций умножения и сложения комплексных значений придётся осуществлять серией команд над вещественными числами, представляя каждое комплексное значение парой вещественных. (По сути именно таким способом компилятор с языка Си и реализует работу с комплексными числами, используя обычный сопроцессор плавающей арифметики).

Обозначая каждую комплексную величину парой вещественных: A=(Ar,Ai), B=(Br,Bi), V=(Vr,Vi), представим операцию бабочки Фурье как единое преобразование над парами вещественных чисел:

$(Ar',Ai') = (Ar+(Br\times Vr - Bi\times Vi),Ai+(Br\times Vi + Bi\times Vr)$

$(Br',Bi') = (Ar-(Br\times Vr - Bi\times Vi),Ai-(Br\times Vi + Bi\times Vr)$

Тогда для реализации операции бабочки Фурье BF(A,B,V) можно применить следующий алгоритм:

```
Ti=(Br×Vi+Bi×Vr);Tr=(Br×Vr-Bi×Vi);
Ai= Ai+Ti; Ar= Ar+Tr;
Bi= Bi-Ti; Br= Br-Tr;
```

Рассмотрим далее, как этот алгоритм может быть реализован для вычисления комплексных значений двойной точности на сопроцессоре СР1 плавающей вещественной арифметики микропроцессора семейства КОМДИВ. В соответствии с этим алгоритмом Си-компилятор сгенерирует ассемблерный (или машинный код) примерно следующего вида:

| вычислительные команды | команды работы с памятью |
|---|---|
| | `Ldc1 FVi,8(rV)` |
| | `Ldc1 FVr,0(rV)` |
| | `Ldc1 FBi,8(rB)` |
| `Mul.d FTi,FBi,FVr;` | `Ldc1 FBr,0(rB)` |
| `Mul.d FTr,FBi,FVi;` | `Ldc1 FAi,8(rA)` |
| `Madd.d FTi,FTi,FBr,FVi;` | `Ldc1 FAr,0(rA)` |
| `Msub.d FTr,FTr,FBr,FVr;` | |
| `Sub.d FBi,FAi,FTi;` | |
| `Sub.d FBr,FAr,FTr;` | |
| `Add.d FAi,FAi,FTi;` | `Sdc1 FBi,8(rB)` |
| `Add.d FAr,FAr,FTr;` | `Sdc1 FBr,0(rB)` |
| | `Sdc1 FAi,8(rA)` |
| | `Sdc1 FAr,0(rA)` |
| Назначение вещественных 64-разрядных регистров СР1:<br>FAr=Ar, FAi=Ai; FBr=Br, Fbi=Bi; FVr=Br, FVi=Bi;<br>FTr=(Br×Vr−Bi×Vi), FTi=(Br×Vr−Bi×Vi);<br>А целочисленные регистры rV, rA и rB указывают на расположенные в памяти комплексные значения V, A и B: rV→(Vr,Vi), rA→(Ar,Ai), rB→(Br,Bi) | |

В нём 8 вычислительных команд (8M) и 10 команд работы с памятью: 6 команд загрузки (6L) и 4 команды выгрузки (4S). Если и программа, и обрабатываемые данные уже находятся в кэш-памяти, то такой блок из 18 команд выполняется за **28** тактов (установлено экспериментально).

А если этот блок команд будет исполняться многократно в цикле с одинаковыми значениями V, то команды загрузки в регистры FVi и FVr можно поместить перед циклом, и тогда оставшиеся в цикле 16 команд будут исполняться уже за **26** тактов.

Почему же не за 16? Дело в том, что многие команды исполняются более, чем за 1 такт: команды загрузки/вызрузки (Ldc1, Sdc1) – за 2 такта, а вычислительные (Mul.d, Madd.d, Msub.d, Sub.d, Add.d) – от 5 до 8 тактов.

Заметим, что СР1 не позволяет на следующем такте начать исполнять над другими регистрами ту же самую вычислительную команду, которую он только что начал, поэтому дальнейшей существенной оптимизации кода путём совмещения вычислений сразу над несколькими группами данных на СР1 добиться не получается. Но такая оптимизация становится возможной при применении векторного сопроцессора CPV.

# 4 Оптимизация вычислений операций бабочки Фурье на CPV

Каким бы совершенным не был имеющийся аппаратный исполнитель, эффективность реализации требуемой программы во многом определяется совершенством её программного кода. Поэтому анализ возможного выигрыша в производительности, который может обеспечить CPV для вычислений операций бабочек Фурье, будем проводить с учётом известных приёмов оптимизации кода программы для микропроцессоров семейства КОМДИВ.

## 4.1 Одиночное вычисление бабочки

Для вычисления на CPV одиночной операции бабочки Фурье BF(A,B,V) над комплексными величинами двойной точности можно предложить следующий вариант (1) кода:

| ## вариант 1 (18 тактов) | такты | пояснения: |
|---|---|---|
| v**L**dm VV,0(rV) | 3 | VV ← (Vr,Vi) |
| v**L**dm VB,0(rB) | 3 | VB ← (Br,Bi) |
| v**L**dm VA,0(rA) | 3 | VA ← (Ar,Ai) |
| c**M**addsub.d VA,VB,VV | 9 | BF(VA,VB,VV); |
| v**S**dm VB,0(rB) | 3 | VB → (Br,Bi) |
| v**S**dm VA,0(rA) | 3 | VA → (Ar,Ai); |
| Назначение 128-разрядных регистров CP2:<br>VV = V= (Vr,Vi); VB = B = (Br,Bi); VA = A = (Ar,Ai);<br>rV → V= (Vr,Vi); rB→ B = (Br,Bi); rA→ A = (Ar,Ai); | | |

Но эти 6 команд потребуют аж **18** тактов (5+9+4=18). Так что применение CPV для вычислений бабочек Фурье в таком варианте «по одной за раз» никакой особой выгоды не приносит.

Однако, можно существенно оптимизировать код, чтобы ускорить на CPV вычисления, когда требуется исполнить сразу несколько операций бабочек Фурье.

## 4.2 Групповое вычисление бабочек

Допустим, нужно вычислить группу из n операций бабочек Фурье: BF(Ak,Bk,Vk) k=1,2,..,n над различными не зависимыми друг от друга величинами. (Пусть n<22, чтобы хватило 64-х регистров CPV для размещения в них величин Ak, Bk, Vk для k=1,2,..,n,).

Тогда для вычисления всей этой группы можно предложить следующий вариант (2) кода:

```
## BFG (rA, rB, rV, n)
##  вариант 2 ( 6n+2 тактов на n бабочек при n>8 )
##  rVk → Vk,  rBk → Bk,  rAk → Ak   k=1,2,..,n
 vLdm  VV1,0(rV1); ... vLdm  VVn,0(rVn);
 vLdm  VB1,0(rB1);     vLdm  VA1,0(rA1);
       ...                 ...
 vLdm  VBn,0(rBn);     vLdm  VAn,0(rAn);
 cMaddsub.d VA1,VB1,VV1;
  ...
 cMaddsub.d VAn,VBn,VVn;
 vSdm  VB1,0(rB1);     vSdm  VA1,0(rA1)
  ...                   ...
 vSdm  VBn,0(rBn);     vSdm  VAn,0(rAn);
```

Полагая, что команды v**L**dm и v**S**dm могут исполняться так за тактом без задержек (для загрузки и выгрузки разных регистров CPV), и CPV позволяет на каждом такте брать на исполнение очередную команду c**M**addsub.d (над другими регистрами), то весь этот блок из 6·n команд при n>8 можно исполнить без задержек за 6·n+2 тактов (+2 такта потребутся для завершения последней команды v**S**dm). В случае, когда n<9, понадобится 5·n+11 тактов (плюс 9–n тактов задержки перед исполнением первой командой v**S**dm).

В результате такой оптимизации получим скорость вычислений примерно **6** тактов на одну бабочку.

## 4.3 Вычисление бабочек для векторов

Более значительной оптимизации (при чётном n>9) для группового вычисления бабочек Фурье можно добиться, когда элементы пар Aj и Bj для j-ой бабочки являются j-ми элементами векторов A и B, непрерывно лежащих в памяти, а все коэффициенты Vj предварительно уже загружены в регистры CPV.

В таком случае можно использовать одну команду v**L**dq (вместо двух v**L**dm) для загрузки двух элементов вектора в два соседних регистра и одну команду v**S**dq (вместо двух v**S**dm) для выгрузки двух элементов из двух соседних регистров соответственно.

Продемонстрируем вариант (3) кода для n=16:

```
## вариант 3 для n=16 ( 52 такта на 16бабочек )
## BFV16 (rA, rB)
##  rB →B[0:n-1],  rA → A[0:n-1],  p=n/2=8
 vLdq VB0,0(rB);       vLdq VA0,0(rA)
 vLdq VB2,2*16(rB);    vLdq VA2,2*16(rA)
  ...                     ...
 vLdq VB14,14*16(rB); vLdq VA14,14*16(rA)
  cMaddsub.d VA0,VB0,VV0;
  cMaddsub.d VA1,VB1,VV1; ...
  cMaddsub.d VA14,VB14,VV14;
  cMaddsub.d VA15,VB15,VV15;
 vSdq VB0,0(rB);       vSdq VA0,0(rA)
 vSdq VB2,2*16(rB);    vSdq VA2,2*16(rA)
  ...                     ...
 vSdq VB14,14*16(rB); vSdq  VA14,14*16(rA)
```

Такой блок из 48-ми команд (16L+16M+16S) сможет выполниться за 52 такта (+4 такта потребутся для завершения последней команды v**S**dq). В итоге получим скорость вычислений примерно **3.25** такта (52/16=3+4/16= 3.25) на одну бабочку.

## 4.4 Совмещённое вычисление бабочек

Для дальнейшей оптимизации воспользуемся тем, что архитектурой КОМДИВ предусмотрена возможность на каждом такте брать на исполнение сразу две команды разных потоков. Поэтому в одном такте можно совмещать одну вычислительную команду CPV с одной командой загрузки или выгрузки одного или двух регистров. Значит, блок из 48-ми команд, рассмотренных выше в варианте 3, можно исполнить за меньшее число тактов, если каждую команду c**M**addsub.d удастся совместить в одном такте с какой-нибудь командой загрузки v**L**dq или выгрузки v**S**dq для других регистров.

Для этого можно предложить, например, следующий вариант (4) кода:

```
## вариант 4 для n=16 ( 36 тактов на 16 бабочек )
## BFV16 (rA, rB)
##   rB →B[0:n-1],  rA → A[0:n-1],  p=n/2=8
 vLdq VB0,0*16(rB);    vLdq VA0,0*16(rA)
 vLdq VB2,2*16(rB);    vLdq VA2,2*16(rA)
 vLdq VB4,4*16(rB);    vLdq VA4,4*16(rA)
cMaddsub.d VA0,VB0,VV0; vLdq VB6,6*16(rB)
cMaddsub.d VA1,VB1,VV1; vLdq VA6,6*16(rA)
  ...                     ...
cMaddsub.d VA8,VB6,VV8; vLdq VB14,14*16(rB)
cMaddsub.d VA9,VB7,VV9; vLdq VA14,14*16(rA)
cMaddsub.d VA10,VB10,VV10; vSdq VB0,0*16(rB)
cMaddsub.d VA11,VB11,VV11; vSdq VA0,0*16(rA)
cMaddsub.d VA12,VB12,VV12; vSdq VB2,2*16(rB)
cMaddsub.d VA13,VB13,VV13; vSdq VA2,2*16(rA)
cMaddsub.d VA14,VB14,VV14; vSdq VB4,4*16(rB)
cMaddsub.d VA15,VB15,VV15; vSdq VA4,4*16(rA)
 vSdq VB6,6*16(rB);     vSdq VA6,6*16(rA)
  ...                     ...
 vSdq VB14,14*16(rB);   vSdq VA14,14*16(rA)
```

В таком случае 16 бабочек Фурье удастся вычислить за: 6L+10M||10L+6M||6S+10S+4= (32+4)= 36 тактов. А если группу из n бабочек (n=16·p) вычислять циклом по 16 бабочек за один шаг, то на это потребуется: (32+1)·p+4 тактов (+1 на команду перехода с приращением счётчика цикла), что в итоге даст оптимальную скорость вычислений (33·p+4)/(16·p)≈2, т.е. примерно **2** такта на 1 бабочку.

Чтобы добиться ещё большего ускорения, необходимо уменьшать количество команд работы с памятью. Ведь сейчас (см. варианты 3 и 4) на каждую пару команд c**M**addsub.d, в которых задействованы 4 регистра ($VA_i$,$VA_{i+1}$,$VB_i$,$VB_{i+1}$), требуется две команды загрузки v**L**dq и две команды выгрузки v**S**dq для этих регистров. Получается на 2 бабочки такой баланс команд: 2L+2M+2S, который никак не удастся исполнить менее, чем за 4 такта.

В идеальном случае, когда все нужные для вычислений бабочек величины уже хранятся в регистрах и остаются там после исполнения каждой операции, можно достигать предельной скорости вычислений, исполняя одну бабочку за каждый такт. Но для этого необходимо вначале загрузить из памяти в регистры CPV все величины, участвующие в дальнейших вычислениях бабочек Фурье, затем выполнить вычисления всех бабочек с участием загруженных регистров, и лишь после этого полученные в регистрах значения отправлять в память как новые значения этих величин.

## 5 Реализация БПФ на CPV

Для исполнения операции БПФ над вектором длиной N элементов необходимо вычислить $M=N\cdot\log_2 N/2$ бабочек Фурье и при этом хотя бы по одному разу прочитать из памяти в регистр CPV каждый элемент вектора, а затем записать в него обновлённое значение. Значит, в лучшем случае для вычисления $N\cdot\log_2 N/2$ бабочек потребуется загрузить N элементов вектора (да ещё N/2 коэффициентов) в регистры, а после выгрузить их обратно. Но такой благоприятный вариант возможен только если у CPV хватит регистров для размещения всех N+N/2 величин (для элементов и коэффициентов).

Поскольку количество регистров CPV ограничено (их всего 64), то такой вариант можно применить лишь при выполнении БПФ для векторов малой длины.

### 5.1 БПФ длины 32

Содержать все элементы вектора и все коэффициенты в регистрах CPV на протяжении вычислений всех бабочек Фурье возможно лишь при выполнении БПФ длины не более 32 (если рассматривать в качестве длины только степени двойки $N=2^P$).

Чтобы выполнить БПФ для вектора длиной N=32 (БПФ-32), потребуется вычислить 80 бабочек: по 16 бабочек на каждом из 5 этапов ($P=\log_2 32=5$). Элементы вектора можно прочитать из памяти в 32 регистра лишь один раз перед 1-ым этапом, а далее оставлять их в регистрах, и только при завершении последнего 5-ого этапа отправлять полученные значения в память. Ещё 16 регистров потребуется загрузить коэффициентами, необходимыми для вычисления этих 80 бабочек. Итого будет занято 48 регистров CPV.

Заметим, что первоначальную загрузку значений элементов вектора в регистры можно совместить с их бит-реверсной перестановкой, т.е. в регистр VXk (k=0,1..,31) загружать из памяти такой элемент X[m] вектора, индекс m которого соответствует бит-реверсному двоичному значению индекса k.

В результате для выполнения БПФ-32 потребуется:

1) 8 команд v**L**dq для загрузки 16 коэффициентов;
2) 32 команды v**L**dm для чтения в регистры 32-х элементов вектора в бит-реверсном порядке;
3) 80 команд c**M**addsub.d для вычислений 80 бабочек Фурье (по 16 на каждом этапе);
4) 16 команд v**S**dq для записи из регистров в память новых значений для 32-х элементов вектора.

Используя определения следующих макросов:

```
##    BF(i,j,k) => cmaddsub.d VXi,VXj,VVk;
#define BF(i,j,k) cmaddsub.d VX##i##,VX##j##,VV##k##;
##    LoadX(i,j) => vldm VXi,j*16(rX) ## VXi ← X[j]
#define LoadX(i,j) vldm VX##i##,##j##*16(rX);
##   LoadXX(i) => vldq VXi,i*16(rX) (VXi,VXi+1)←X[i:i+1]
#define LoadXX(i) vldq VX##i##,##i##*16(rX);
## StoreXX(i) => vsdq VXi,i*16(rX) (VXi,VXi+1)➔X[i:i+1]
#define StoreXX(i) vsdq VX##i##,##i##*16(rX);
##   LoadVV(i) => vldq VVi,i*16(rV) (VVi,VVi+1)←V[i:i+1]
#define LoadVV(i) vldq VV##i##,##i##*16(rV);
```

приведём возможный вариант кода, осуществляющий такое БПФ-32:

```
## cp3_VFFT32 (rX,rV)  ## rX → X[0:31], rV → V[0:15]
## load VX00..VX18 || load VV0,VV1:
 LoadVV(0);   LoadX(0,0);   LoadX(1,16);  LoadX(2,8);
 LoadX(3,24);  LoadX(4,4);   LoadX(5,20);  LoadX(6,12);
 LoadX(7,28);  LoadX(8,2);   LoadX(9,18);  LoadX(10,10);
 LoadX(11,26); LoadX(12,6);  LoadX(13,22); LoadX(14,14);
 LoadX(15,30); LoadX(16,1);  LoadX(17,17); LoadX(18,9);
## t=1;(k=0,V0)|| load VX19..VX31 || load VVi, i=8,9,4,5,12,13:
BF(0,1,0);    LoadX(19,25);     BF(2,3,0);    LoadX(20,5);
BF(4,5,0);    LoadX(21,21);     BF(6,7,0);    LoadX(22,13);
BF(8,9,0);    LoadX(23,29);     BF(10,11,0); LoadX(24,3);
BF(12,13,0); LoadX(25,19);     BF(14,15,0); LoadX(26,11);
BF(16,17,0); LoadX(27,27);     BF(18,19,0); LoadX(28,7);
BF(20,21,0); LoadX(29,23);     BF(22,23,0); LoadX(30,15);
BF(24,25,0); LoadX(31,31);     BF(26,27,0); LoadVV(8);
BF(28,29,0); LoadVV(4);        BF(30,31,0); LoadVV(12);
##--- t=2; (k=0,V0), (k=1,V8); || load VVi, i=2,3,6,7,10,11,14,15:
BF(0,2,0);    LoadVV(2);       BF(4,6,0);    LoadVV(6);
BF(1,3,8);    LoadVV(10);      BF(5,7,8);    LoadVV(14);
BF(8,10,0);   BF(12,14,0);     BF(9,11,8);   BF(13,15,8);
BF(16,18,0); BF(20,22,0);      BF(17,19,8); BF(21,23,8);
BF(24,26,0); BF(28,30,0);      BF(25,27,8); BF(29,31,8);
##--- t=3; (k=0,V0), (k=1,V4), (k=2,V8), (k=3,V12);
BF(0,4,0);    BF(8,12,0);      BF(1,5,4);    BF(9,13,4);
  BF(2,6,8);    BF(10,14,8);     BF(3,7,12);   BF(11,15,12);
BF(16,20,0); BF(24,28,0);      BF(17,21,4); BF(25,29,4);
  BF(18,22,8); BF(26,30,8);      BF(19,23,12); BF(27,31,12);
##--- t=4; (для k,Vk·2), (k=0,1,..,7);
BF(0,8,0); BF(1,9,2); BF(16,24,0); BF(2,10,4);
BF(17,25,2); BF(3,11,6); BF(18,26,4); BF(4,12,8);
BF(19,27,6); BF(5,13,10); BF(20,28,8); BF(6,14,12);
 BF(21,29,10); BF(7,15,14); BF(22,30,12); BF(23,31,14);
##--- t=5; (для k,Vk), (k=0,1,8,9,2,3,10,11,4,5);
BF(0,16,0);  BF(1,17,1);  BF(8,24,8);  BF(9,25,9);
BF(2,18,2);  BF(3,19,3);  BF(10,26,10); BF(11,27,11);
BF(4,20,4);  BF(5,21,5);
##--- t=5; (для k,Vk), (k=12,13,6,7,14,15); || VX ➔ X[0:31]
BF(12,28,12); StoreXX(0);      BF(13,29,13); StoreXX(16);
BF(6,22,6);   StoreXX(8);      BF(7,23,7);   StoreXX(24);
BF(14,30,14); StoreXX(2);      BF(15,31,15); StoreXX(18);
## ----- store VX0..VX31 ➔ X[0:31] :
StoreXX(10); StoreXX(26); StoreXX(4); StoreXX(20);
StoreXX(12); StoreXX(28); StoreXX(6); StoreXX(22);
StoreXX(14); StoreXX(30);
```

В нём 136 команд (8L+32L+80M+16S=136) размещены так, чтобы они могли исполниться за 114 тактов (20L+20M||20L+54M+6M||6S+10S=110, плюс 4 такта на завершение последней операции v**S**dq 110+4=114). Итого получается 114/80=**1.425** тактов на 1 бабочку.

## 5.2 БПФ длины $N=2^P>32$

Вторая часть алгоритма БПФ для вектора длиной $N=2^P>32$ , которая начинает выполняться после бит-реверсной перестановки (БРП) элементов вектора, на первых 5-ти этапах по сути совершает над каждой группой из 32-х элементов операцию БПФ длиной 32, но только уже без БРП.

Поэтому в алгоритме БПФ произвольной длины будем применять вариант БПФ-32 (из п.5.1, но без БРП) для исполнения 5-ти первых этапов (часть 2А). А на последующих этапах (часть 2Б) будем вычислять бабочки Фурье группами (по 16 штук), как было рассмотрено в п.4.3-4.4. В итоге получим алгоритм БПФ, который можно выразить схематично в следующем виде (на языке Си):

```
// часть 1 Бит-Реверсная Перестановка (БРП)
cp3_VFFT_BitReverseExchange(X,BRT,BRN);//{*1}
// часть 2А t=1..5 БПФ-32 (без БРП) над группами по 32
cp3_VWLoad32(Cf32); {*2}
for(k=0;k<N>>5;k++)
 cp3_DoVFFT32(&X[k*32]);//{*3}
// часть 2Б t=6..P
s=32; h=2; G=32; R=N/64; d=N/64;
for(t=6, t<=P; t++ ) {
 for(k=0,u=0; k<G; k=k+16, u=u+16*d ) {
  cp3_VFFT_Load16W(&Cf[u],d); //{*4}
  for(j=0,m=k; j<R; j++, m=m+h ) {
   cp3_VFFT_Do16BF(&X[m],&X[m+s]);//{*5}
  }//for j
 }//for k
 h=h*2; s=s*2; G=G/2; R=R*2; d=d/2;
}//for t
```

Характеристика ассемблерных функций:

**cp3_VFFT_BitReverseExchange** – выполняет бит-реверсную перестановку (БРП) элементов массива

**cp3_VWLoad32** – загружает в регистры VV0,..,VV15 CPV все 16 коэффициентов бабочек Фурье для БПФ-32

**cp3_DoVFFT32** – выполняет БПФ для вектора длиной 32 как описано в п.6.1, но без БРП и без загрузки коэффициентов

**cp3_VFFT_Load16W** – загружает в регистры VV0,..,VV15 CPV очередные 16 коэффициентов бабочек Фурье

**cp3_VFFT_Do16BF** – вычисляет 16 бабочек Фурье BFV16 как описано в п.5.3 (или 5.4)

Приведём вариант первоначальной части кода для **cp3_DoVFFT32**, которой он отличается от кода рассмотренной выше процедуры **cp3_VFFT32** для БПФ-32 в предположении, что все 16 коэффициентов для БПФ-32 уже загружены (перед циклом) в CPV-регистры VV0..VV15 процедурой **cp3_VWLoad32**:

```
##    cp3_DoVFFT32 (rX)      ## rX → X[0:31]
## ---- load X[0:9] ➔ VX0..VX9 (без БРП):
 LoadXX(0); LoadXX(2); LoadXX(4); LoadXX(6); LoadXX(8);
##--- t=1; (k=0,V0) || load X[10:31] => VX10..VX31:
BF(0,1,0);    LoadXX(10); BF(2,3,0);    LoadXX(12);
BF(4,5,0);    LoadXX(14); BF(6,7,0);    LoadXX(16);
```

```
BF(8,9,0);     LoadXX(18); BF(10,11,0); LoadXX(20);
BF(12,13,0); LoadXX(22); BF(14,15,0); LoadXX(24);
BF(16,17,0); LoadXX(26); BF(18,19,0); LoadXX(28);
BF(20,21,0); LoadXX(30); BF(22,23,0);
BF(24,25,0); BF(26,27,0); BF(28,29,0); BF(30,31,0);
##--- t=2; (k=0,V0), (k=1,V8);
BF(0,2,0);     BF(4,6,0) BF(1,3,8);     BF(5,7,8);
BF(8,10,0);   BF(12,14,0);        BF(9,11,8);   BF(13,15,8);
## . . . и далее как в процедуре cp3_VFFT32 . . .
```

В этом варианте кода (16L+80M+16S=112) 112 команд размещены так, чтобы они могли исполняться за (5L+11M||11L+63M+6M||6S+10S=95) 95+4=99 тактов. А при исполнении их в цикле m раз (m=N/32) потратится (95+1)·m+4 тактов, т.е. получится примерно (96·m+4)/(80·m)≈96/80= **1.2** тактов на 1 бабочку.

Заметим, что, вообще говоря, выполнить серию БПФ-32 можно быстрее, чем одиночную БПФ-32. Когда предстоят вычисления БПФ-32 для нескольких векторов, то цикл, на одном шаге которого вычисляется БПФ для очередного вектора, можно усовершенствовать.

Во-первых, коэффициенты для бабочек Фурье можно загружать в регистры CPV один раз перед циклом, что и предусмотрено в рассмотренном варианте.

Во-вторых, на каждом шаге цикла вычисления бабочек для текущего вектора можно совмещать с записью в память вычисленных значений для предыдущего вектора и чтением из памяти в регистры значений следующего вектора.

Такое усовершенствование, казалось бы, позволило бы нам добиться скорости вычислений, близкой к идеальной (1 бабочка за такт). Но для его осуществления необходимо выделить регистры CPV для хранения в них значений элементов не одного, а сразу двух векторов. Так что для реализации этой возможности для БПФ длины 32 потребуется 32+32+16=80 регистров, но таким количеством регистров CPV не располагает. Значит, усовершенствовать таким способом рассмотренный вариант кода для БПФ-32 не получится.

Можно, конечно, было бы проанализировать возможный вариант такого усовершенствования для БПФ длиной 16 (БПФ-16). Но БПФ-16 не может быть эффективно реализован на CPV по другой причине: вычисляемые в ходе БПФ-16 32 бабочки Фурье ($16\cdot\log_2 16/2=16\cdot 4/2=32$) не могут уложиться в идеальные 32 такта, т.к. команда cmaddsub.d исполняется аж 9 тактов, а возникающие зависимости между следующими и предыдущими бабочками вынуждено порождают значительные задержки, так что на чистое вычисление 32-х бабочек (без операций чтения/записи) потребуется как минимум 42 такта, что уже уступает по производительности рассмотренному выше оптимальному варианту БПФ-32 (42/32=1.3125 > 1.2).

Таким образом, с помощью векторного сопроцессора удаётся значительно ускорить операцию БПФ для вектора комплексных величин двойной точности. В частности, выполнить БПФ-32 на CPV путём вызова описанной выше функции **cp3_VFFT32** можно почти в **22.5** раза быстрее, чем на CP1 путём вызова Си-функции, алгоритм которой представлен в п.2. На внутренний цикл этой функции, где вычисляется одна бабочка BF(X[m],X[m+s],V) уходит **32** такта (установлено экспериментально). Значит, даже без учёта БРП на вычисление 80 бабочек на CP1 потратится как минимум 32·80 тактов, а на CPV — всего 114 тактов (32·80/114≈22.456).

Для вектора произвольной длины $N=2^P>32$ можно также существенно ускорить (примерно в **16** раз) вторую часть алгоритма БПФ, где производятся многочисленные вычисления бабочек Фурье, т.к. вместо **32** тактов можно затрачивать на выполнение одной бабочки в среднем 1.2 такта (в части 2А) и примерно 2 такта (в части 2Б).

Остаётся выяснить, как векторный сопроцессор способствует ускорению процедуры битреверсной перестановки, которую требуется выполнять в первой части алгоритма БПФ.

## 5.3 Оптимизация БРП

Чтобы оптимизировать исполнение бит-реверсной перестановки (БРП) вектора длиной N, можно заранее подготовить таблицу BRT пар различных индексов ($k_j \neq m_j$), задающих, какие элементы вектора должны будут переставляться. Индексы ($k_j$,$m_j$) в каждой такой паре должны быть друг для друга бит-реверсными перестановками их двоичных кодов: $k_j=(b_P,\ldots,b_2,b_1)$, $m_j=(b_1,b_2,\ldots,b_P)$. Тогда алгоритм БРП будет просто переставлять всевозможные пары $X[k_j] \leftrightarrow X[m_j]$ элементов вектора:

```
for(j=0; j<BRN; j++ )
{ k= BRT[2*j]; m= BRT[2*j+1];
  T= X[m]; X[m]=X[k]; X[k]=T; }//for j
```

Заметим, что для доступа к элементу массива X[k] по индексу (k) потребуется вычислять смещение элемента относительно начала массива путём домножения его индекса на размер элементов массива (k*16). Поэтому для оптимизации БРП целесообразно приготовить в таблице BRT не просто индексы переставляемых элементов, а вычисленные для них смещения, что позволит сократить внутренний цикл БРП на 2 команды. Но для этого и алгоритм БРП (на языке Си) придётся переписать немного иначе:

```
for(j=0; j<BRN; j++ )
{ Xk=(char*)&X+BRT[2*j];
  Xm=(char*)&X+BRT[2*j+1];
  T=*Xm; *Xm=*Xk; *Xk=T; }//for j
```

Для выполнения такой же БРП на CPV можно предложить следующий ассемблерный код:

```
cp3_VFFT_BitReverseExchange: ##
move rX, a0; move rBRT,a1; move rK,a2
BRELoop: ## for(rK=BRN;rK>0;rK=rK-1)
         ## Exchange X[rI]<=>X[rJ] :
 lwu rI,0(rBRT) ## rJ=BRT[2*rK]
 lwu rJ,4(rBRT) ## rJ=BRT[2*rK+1]
    addiu rBRT,rBRT,8
 vldmx VXI,rI(rX) ## VXI:= X[rI]
    addi rK,rK,-1 ## rK:= rK-1
 vldmx VXJ,rJ(rX) ## VXJ:= X[rJ]
 vsdmx VXI,rJ(rX) ## X[rJ]:= VXI;
 bgtz rK,BRELoop
 vsdmx VXJ,rI(rX) ## X[rI]:= VXJ;
 jr ra ; ssnop
```

Но он позволит ускорить процедуру БРП с помощью CPV всего лишь примерно в **1.75** раза (теперь 1 шаг цикла перестановки будет исполняться за 8 тактов вместо 14).

Чтобы уменьшить затраты на БРП можно предложить реализовать (при дальнейшем развитии CPV) специальные режимы так называемой бит-реверсной автоинкрементной адресации, которыми обычно наделяют сопроцессоры, ориентированные на ускоренное исполнение БПФ.

Такой режим адресации, например, был предусмотрен в процессорах обработки сигналов DSP96002 фирмы Motorolla (см.[8], п.3.5, с.235-250), а также осуществлён в сопроцессоре CP2 микропроцессора К128РИО (1890ВМ7) семейства КОМДИВ (см.[4], с.130). Использование такого бит-реверсного режима адресации позволяет выборку очередных элементов вектора из памяти в регистры осуществлять в соответствии с требуемым бит-реверсным порядком, исключая тем самым всякую необходимость в предварительной перестановке элементов вектора.

Поскольку первую часть алгоритма БПФ, в которой выполняется БРП, не удаётся ускорить на CPV в той же степени, что и её вторую часть, в которой производятся вычисления

бабочек Фурье, общий выигрыш в ускорении операции БПФ на CPV получится меньше. Насколько меньше? Чтобы оценить итоговый получаемый выигрыш, решим следующую задачу.

Задача Операция P состоит из двух частей $P_1$ и $P_2$: $P=P_1+P_2$, временные доли исполнения которых $d_1$ и $d_2$: $T=T_1+T_2$, $d_1=T_1/T$, $d_2=T_2/T$. Вопрос: если исполнять часть $P_1$ в $k_1$ раз, а часть $P_2$ — в $k_2$ раз быстрее, то во сколько раз (k) ускорится исполнение всей операции ?

Решение Выразим новые времена исполнения для $P_1$, $P_2$, P:

$T_1'=T_1/k_1=T\cdot d_1/k_1$, $T_2'=T_2/k_2=T\cdot d_2/k_2$, $T'=T_1'+T_2'$,

$T'= T\cdot d_1/k_1 + T\cdot d_2/k_2 = T\cdot(d_1\cdot k_2+ d_2\cdot k_1)/(k_1\cdot k_2)$.

Тогда коэффициент k ускорения операции P, равный отношению $k=T/T'$, должен вычисляться по формуле:

$k= (k_1\cdot k_2) / (d_1\cdot k_2+ d_2\cdot k_1)$ { Ф1 }.

А в том случае, когда часть $P_1$ в новом исполнении исчезает совсем ($k_1\to\infty$), вычисление коэффициента k можно упростить:

$T_1'=0$, $T'=T\cdot d_2/k_2$, значит, $k=k_2/d_2$. { Ф2 } .

Используя эти формулы, оценим, какой можно ожидать итоговый выигрыш в ускорении операции БПФ на CPV. При исполнении на CP1 на долю БРП (1-ой части БПФ) затрачивается примерно 5% времени всей операции БПФ (это установлено экспериментально). Значит, для операции БПФ доли её частей будут соответственно: $d_1=0.05$, $d_2=0.95$. Коэффициент ускорения на CPV для 1-ой части $k_1=1.75$, для 2-ой части БПФ-32 — $k_2=22.5$, а для БПФ длиной $N>32$ — $k_2=16$.

Для операции БПФ длиной $N>32$ вычисляем итоговый коэффициент ускорения на CPV по формуле Ф1:

$K_N=(1.75\cdot16)/(0.05\cdot16+0.95\cdot1.75)=$

$28/(0.8+1.6625)= 28/2.4625 \approx$ **11.37**.

Для БПФ-32 — по формуле Ф2: $K_{32}=22.5/0.95 \approx$ **23.68**.

## 6 Результаты ускорения на CPVоперации БПФ

Коэффициенты возможного ускорения операций БПФ на CPV, выявленные теоретически в ходе проведённого анализа, требовалось проверить на практике. Для этого были составлены специальные программы — тесты, в которых каждую операцию БПФ над вектором заданной длины $N=2^P$ предусматривалось выполнить дважды. Один раз с помощью применения обычного процессора плавающей арифметики CP1 путём вызова функции, составленной на языке Си (по алгоритмам, описанным в п.2 и п.5.3). А другой раз с помощью вызовов одной (как в п.5.1 для БПФ-32) или нескольких ассемблерных функций (как в п.5.2 для БПФ длины $N>32$), в которых применяются команды векторного сопроцессора.

Тесты прогонялись многократно для векторов различной длины N=32,64,..,4096 (N фиксировалось заранее перед компиляцией программы). Прогоны тестов выполнялись на доступной в настоящее время версии RTL-модели (от 02.09.2015) создаваемого микропроцессора 1890ВМ8 семейства КОМДИВ. В каждом тесте непосредственно перед вызовом самой операции БПФ её программный код и обрабатываемые данные помещались в кэш-память. Время выполнения операции БПФ в каждом случае оценивалось по числу тактов, затраченных на её исполнение.

### 6.1 Результаты ускорения БПФ для DC

Результаты прогонов тестовых программ и полученные в итоге коэффициенты ускорения операции БПФ для вектора комплексных величин двойной точности (DC – Double Complex) представлены в таблице 3. Её столбец **KV** характеризует коэффициент выигрыша, который

показывает, во сколько раз операция БПФ для DC исполняется быстрее при её реализации с применением векторного сопроцессора CPV.

Таблица 3. Ускорение на CPV операции БПФ для DC

| N | BFN | %L2(L1) | CP1 | CPV | KV | KE% |
|---|---|---|---|---|---|---|
| 32 | 80 | 0.12 (3.71) | 3385 | 168 | **20.15** | **47.62** |
| 64 | 192 | 0.24 (7.62) | 7675 | 829 | **9.26** | **23.16** |
| 128 | 448 | 0.48 (15.23) | 17161 | 1899 | **9.04** | **23.59** |
| 256 | 1024 | 0.96 (30.86) | 38252 | 4397 | **8.7** | **23.29** |
| 512 | 2304 | 1.93 (61.72) | 84454 | 10013 | **8.43** | **23.01** |
| 1024 | 5120 | 3.88 (124.22) | 190977 | 23716 | **8.05** | **21.59** |
| 2048 | 11264 | 7.76 (248.44) | 466638 | 55620 | **8.39** | **20.25** |
| 4096 | 24576 | 15.58 (498.44) | 1012344 | 125089 | **8.09** | **19.65** |

**BFN** – количество вычисляемых бабочек Фурье
**%L2(L1)** – доля заполненности L2(L1)-кэша (в процентах %)
**CP1** – число тактов, затраченных на операцию БПФ на CP1
**CPV** – число тактов, затраченных на операцию БПФ на CPV
**KV** – коэффициент выигрыша или ускорения, который CPV обеспечивает для БПФ по сравнению с CP1 (= такты **CP1**/ такты **CPV**)
**KE** – коэффициент эффективности использования CPV (в %), равный отношению полученной производительности к пиковой

Полученные экспериментальным путём результаты показывают, что с применением CPV удаётся реально в значительной степени ускорить операцию БПФ для DC: почти в **20** раз операцию БПФ-32 и в **8-9** раз операцию БПФ длины N>32.

### 6.2 Результаты ускорения БПФ для SC

До сих пор анализировалась возможность ускорения на CPV процедуры БПФ и операции бабочки Фурье для комплексных величин двойной точности (DC). При обработке комплексных величин одинарной точности (SC – Single Complex) одна команда cmaddsub.s исполняет сразу две бабочки Фурье (одну для величин из левых половин, а другую для величин из правых половин регистров). Даёт ли это возможность ускорить БПФ для SC в **2** раза более, чем для DC?

Таблица 7. Ускорение на CPV операции БПФ для SC

| N | BFN | %L2(L1) | CP1 | CPV | KV | KE% |
|---|---|---|---|---|---|---|
| 32 | 80 | 0.07(2.15) | 3296 | 152 | **21.68** | **26.32** |
| 64 | 192 | 0.14 (4.49) | 7524 | 715 | **10.52** | **13.43** |
| 128 | 448 | 0.28 (8.98) | 16771 | 1546 | **10.85** | **14.49** |
| 256 | 1024 | 0.57 (18.36) | 37289 | 3408 | **10.94** | **15.02** |
| 512 | 2304 | 1.15 (36.72) | 82183 | 7378 | **11.14** | **15.61** |
| 1024 | 5120 | 2.32 (74.22) | 180176 | 16215 | **11.11** | **15.79** |
| 2048 | 11264 | 4.64 (148.44) | 400610 | 38417 | **10.43** | **14.66** |
| 4096 | 24576 | 9.33 (298.44) | 954779 | 90118 | **10.59** | **13.64** |

Коэффициенты ускорения операции БПФ для SC, полученные экспериментальным путём, демонстрируются в таблице 7. Как видно из этой таблицы, выигрыш в ускорении оказывается не столь уж большим (Он всё же выше, чем для аналогичных операций БПФ для DC, но не в 2 раза, как спрашивалось в поставленном выше вопросе).

Это объясняется тем, что для использования команды cmaddsub.s приходится тратить дополнительные команды для упаковки в один регистр двух величин, прочитанных из разных элементов вектора (при их выборке из памяти в бит-реверсном порядке) или из разных регистров (после 1-го этапа вычислений бабочек Фурье).

## Заключение

Проведённый анализ и полученный практический опыт разработки ряда БПФ-процедур для имеющегося варианта векторного сопроцессора, реализованного в составе микропроцессора 1890ВМ8 семейства КОМДИВ, позволяют сделать вывод о возможности его применения для ускорения операций быстрого преобразования Фурье над векторами комплексных чисел, непрерывно расположенных в памяти.

Данный векторный сопроцессор CPV реально обеспечивает ускорение (по отношению к CP1) в **8-9** раз для операций БПФ с непрерывными векторами длиной $N=2^P>32$ и в **20** раз для БПФ с векторами длиной 32 (если они уже в кэш-памяти).

## Литература


1. В.Б. Бетелин. Отечественные суперкомпьютерные технологии экзафлопсного класса – необходимое условие обеспечения технологической конкурентноспособности России в XXI веке. // Программные продукты и системы. 2013. №4 (104). С.4-9.

2. Методы встречной оптимизации в задачах обработки сигналов. / Сб. под ред. академика В.Б. Бетелина М.: Изд-во НИИСИ РАН, 2005. 130 с.

3. А.А. Бурцев. О возможности оптимизации некоторых функций библиотеки линейной алгебры с помощью векторного сопроцессора // Труды НИИСИ РАН, т.4 №2. М.: Изд-во НИИСИ РАН, 2014. с.5-15.

4. О.Ю. Сударева. Эффективная реализация алгоритмов быстрого преобразования Фурье и свёртки на микропроцессоре КОМДИВ128-РИО. // под ред. академика В.Б. Бетелина М.: НИИСИ РАН, 2014. 266 с.

5. Стивен Смит. Цифровая обработка сигналов. Практическое руководство для инженеров и научных работников / Стивен Смит; пер. с англ. А.Ю. Линовича, С.В. Витязева, И.С. Гусинского. — 720 с.

6. Быстрое преобразование Фурье по основанию 2 с прореживанием по времени. URL: http://www.dsplib.ru/content/thintime/thintime.html (дата обращения: 02.09.2015).

7. Быстрое преобразование Фурье. URL: http://psi-logic.shadanakar.org/fft/fft.htm (дата обращения: 02.09.2015).

8. В.В. Корнеев, А.В. Киселёв. Современные микропроцессоры. М.: НОЛИДЖ, 2000. 320с.