**Министерства науки и высшего образования**

**Российской Федерации**

Федеральное государственное автономное образовательное учреждение

высшего профессионального образования

**Санкт-Петербургский политехнический университет Петра Великого**

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Курсовой проект**

Тема проекта: Быстрое преобразование Фурье

**по курсу: «Проектирование реконфигурируемых гибридных вычислительных систем»**

Выполнил студент гр. 3540901/81501 Ерниязов Т.Е.

Руководитель, к.т.н. Антонов А. П.

Санкт-Петербург

2019

ОГЛАВЛЕНИЕ

[Глава 1. Быстрое преобразование Фурье. Теория. 3](#_Toc30300837)

[Введение 3](#_Toc30300838)

[Суть алгоритма 3](#_Toc30300839)

[Глава 2. Реализация FFT в среде Vivado HLS 8](#_Toc30300840)

[Исходные данные 8](#_Toc30300841)

[Результаты синтеза и симуляции 10](#_Toc30300842)

[Оптимизация bit\_reverse 13](#_Toc30300843)

[Конвейеризация 18](#_Toc30300844)

[Выводы 23](#_Toc30300845)

# Глава 1. Быстрое преобразование Фурье. Теория.

## Введение

Использование дискретного преобразования Фурье с помощью перемножения матриц и векторов имеет сложность . Для этого алгоритма возможно уменьшить его сложность, используя структуру матрицы коэффициентов. Каждая строка в матрице отображает количество оборотов по кругу, поэтому эти значения могут значительно сокращены.

Алгоритм быстрого преобразования Фурье (FFT) использует метод разделяй и властвуй, основанный на симметрии матрицы коэффициентов. Алгоритм Кули-Таки способен получить тот же результат, что и дискретное преобразование Фурье, но со сложностью .

## Суть алгоритма

FFT использует симметрию матрицы коэффициентов DFT, рассмотрим это на примере 2 точечного алгоритма. DFT использует в себе перемножение матрицы и вектора:

Где g[] – входные данные, G[] – частотная характеристика (выходные данные), а S[][] это коэффициента DFT. Для 2 точечного алгоритма она будет равна:

Здесь

Графическое представление перемножения этой матрицы на входной вектор выглядит в виде бабочки:

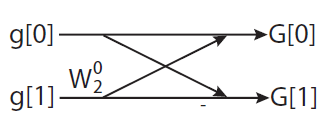


Рис. 1 Бабочка

Перекрещение линий значит операцию сложения. Любая подпись на линии означает перемножение значения на линии с этой переменной.

В дальнейшем увеличении количества точек FFT можно увидеть, что, например,

, а это означает, что можно значительно сократить подсчет тригонометрических функций. Например, для 4 точечного алгоритма матрица коэффициентов будет выглядеть следующим образом:

Тогда

После перестановки битов можно увидеть, что

Чтобы воспользоваться симметрией для начала разделим входной массив на четные и нечетные элементы. Также можно увидеть что в G[0] и G[2] четные и нечетные элементы суммируются друг с другом, а в других вычитаются. В конце концов нечетные элементы перемножаются с коэффициентом, а четные нет. При этом для четной пары можно применить алгоритм 2 точечного FFT. Приведем еще упростим еще больше полученные выражения:

Теперь можно увидеть тенденцию как перейти от сложности к . 4 точечный алгоритм использует 2 точечный, а 8 точечный использует, в свою очередь, 4 точечный и т. д. Таким образом, 32 точечный алгоритм использует под собой 16 2-точечных алгоритмов FFT.

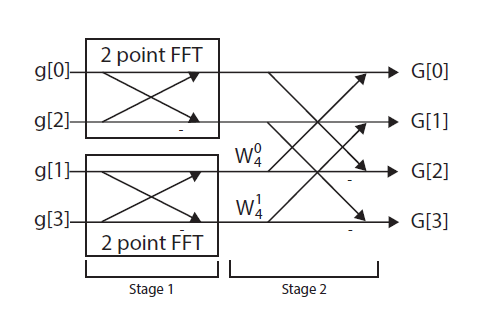


Рис. 2 Представление 4 точечного алгоритма

Следовательно, с помощью рекурсии можно посчитать алгоритмы с более большим количеством точек, которые будут базироваться на результатах алгоритмов с более меньшим количеством точек.

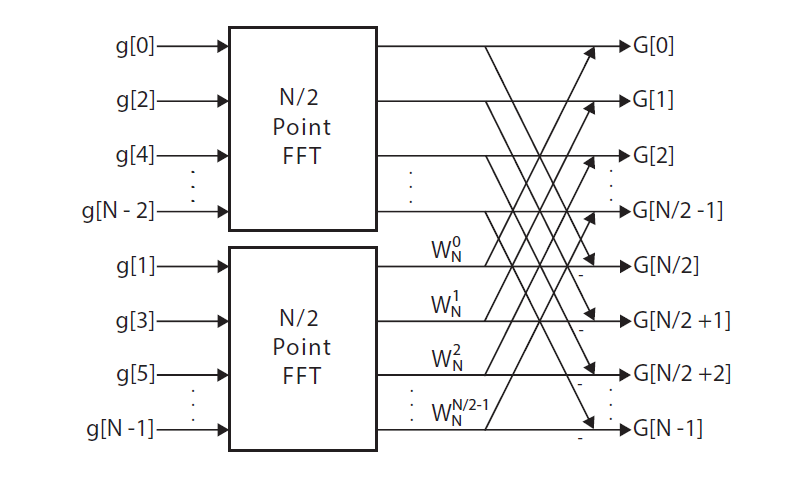


Рис. 3 Взаимосвязь N точечного алгоритма с N/2 точечным

В данной работе мы будем использовать 8 точечный алгоритм, поэтому приведем его полную структуру.

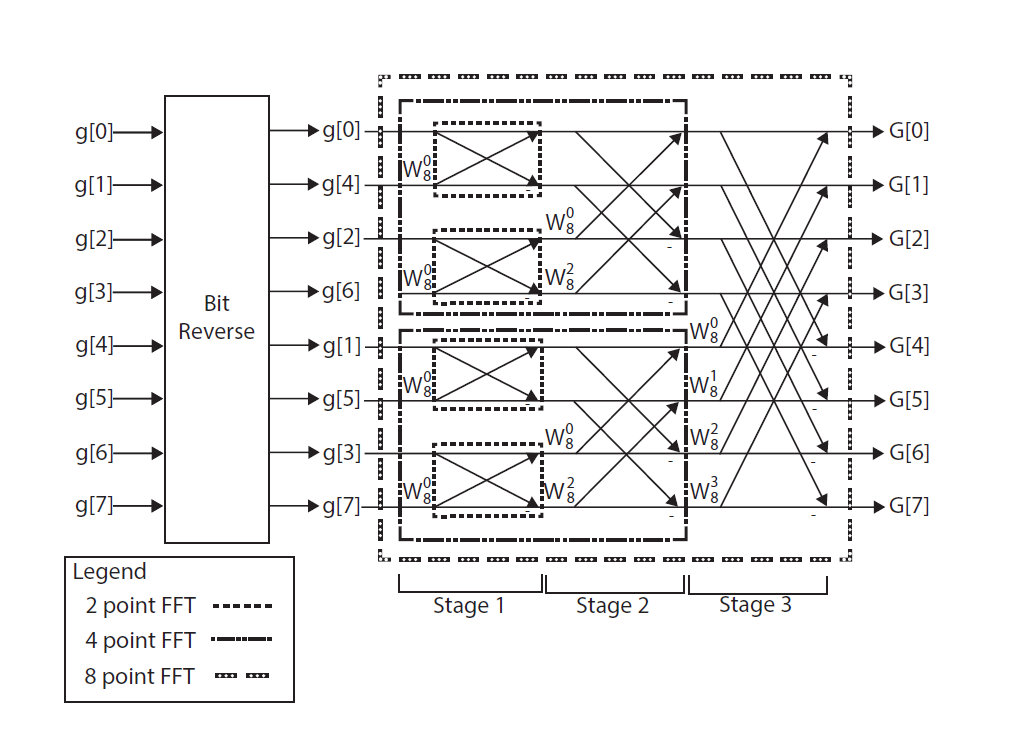


Рис. 4 8 точечный алгоритм быстрого преобразования Фурье

Также следует отметить перестановку элементов в входном массиве, так как N/2 алгоритмы, на которых основаны вычисления текущего, берут четные и нечетные элементы. Верхний берет четные, а нижний нечетные (в соответствии с рисунком 3). Однако, после того как мы их разделили, нам следует опять отобрать четные и нечетные элементы, чтобы разделить их между алгоритмами более низких уровней.

Допустим, для 8 точечного алгоритма мы сначала делим входной массив на 2 части:

{g[0]; g[2]; g[4]; g[6]} и {g[1]; g[3]; g[5]; g[7]}. Теперь каждую часть надо поделить еще раз. Например, {g[0]; g[4]} и {g[4]; g[6]} и их позиции в изначальном массиве становятся {g[0]; g[4]; g[2]; g[6]}.

Следующая таблица показывает перестановку элементов для 8 точечного алгоритма

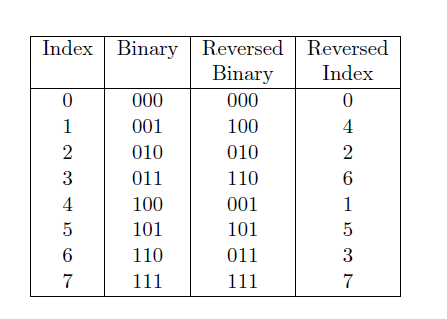


Рис. 5 Перестановка элементов по их индексам

Техника перестановки работает только если количество точек в алгоритме равно степеням 2, обычно это так, иначе нельзя было бы использовать рекурсию для убыстрения счета.

# Глава 2. Реализация FFT в среде Vivado HLS

## Исходные данные

Сначала приведем прямолинейную реализацию без оптимизаций:

|  |
| --- |
| **#include** "math.h"  **#define** DTYPE **float**  **#define** SIZE 8  **#define** SIZE2 4  **#define** M 3  **unsigned** **int** **reverse\_bits**(**unsigned** **int** input) {  **int** i, rev = 0;  **for** (i = 0; i < M; i++) {  rev = (rev << 1) | (input & 1);  input = input >> 1;  }  **return** rev;  }  **void** **bit\_reverse**(DTYPE X\_R[SIZE], DTYPE X\_I[SIZE]) {  **unsigned** **int** reversed;  **unsigned** **int** i;  DTYPE temp;  bit\_reverse\_label0:  **for** (i = 0; i < SIZE; i++) {  reversed = reverse\_bits(i); // Find the bit reversed index  **if** (i <= reversed) {  // Swap the real values  temp = X\_R[i];  X\_R[i] = X\_R[reversed];  X\_R[reversed] = temp;  // Swap the imaginary values  temp = X\_I[i];  X\_I[i] = X\_I[reversed];  X\_I[reversed] = temp;  }  }  }  **void** **fft**(DTYPE X\_R[SIZE], DTYPE X\_I[SIZE]) {  DTYPE temp\_R; // temporary storage complex variable  DTYPE temp\_I; // temporary storage complex variable  **int** i, j, k; // loop indexes  **int** i\_lower; // Index of lower point in butterfly  **int** step, stage, DFTpts;  **int** numBF; // Butterfly Width  **int** N2 = SIZE2; // N2=N>>1  bit\_reverse(X\_R, X\_I);  step = N2;  DTYPE a, e, c, s;  stage\_loop:  **for** (stage = 1; stage <= M; stage++) { // Do M stages of butterflies  DFTpts = 1 << stage; // DFT = stage = points in sub DFT  numBF = DFTpts / 2; // Butterfly WIDTHS in DFT  k = 0;  e = -6.283185307178 / DFTpts;  a = 0.0;  // Perform butterflies for j-th stage  butterfly\_loop:  **for** (j = 0; j < numBF; j++) {  c = **cos**(a);  s = **sin**(a);  a = a + e;  // Compute butterflies that use same W\*\*k  dft\_loop:  **for** (i = j; i < SIZE; i += DFTpts) {  i\_lower = i + numBF; // index of lower point in butterfly  temp\_R = X\_R[i\_lower] \* c - X\_I[i\_lower] \* s;  temp\_I = X\_I[i\_lower] \* c + X\_R[i\_lower] \* s;  X\_R[i\_lower] = X\_R[i] - temp\_R;  X\_I[i\_lower] = X\_I[i] - temp\_I;  X\_R[i] = X\_R[i] + temp\_R;  X\_I[i] = X\_I[i] + temp\_I;  }  k += step;  }  step = step / 2;  }  } |

Листинг 1. Простая реализация алгоритма FFT

Далее приведем тестбенч для проверки реализации. Будем проводить тестирование для алгоритма FFT на 8 точек и типом данных float. Для двух массивов исходных данных (реальные и мнимые части комплексных чисел) посчитаем алгоритм FFT отдельно с помощью сторонних средств и составим 2 массива для проверки корректности алгоритма.

|  |
| --- |
| **#include** <stdio.h>  **int** **main**()  {  **float** r[8] = {2.0, 1.0, 3.0, 6.0, 9.0, 12.0, 5.0, 7.0};  **float** i[8] = {6.0, 7.0, 1.0, 4.0, 2.0, 1.0, 4.0, 9.0};  **float** r\_res[8] = {45.000000, -16.363960, -2.000000, 3.778174, -7.000000, -3.636039, 8.000000, -11.778174};  **float** i\_res[8] = {34.000000, 22.263456, 3.000000, 2.707106, -8.000000, -10.263456, 3.000000, 1.292893};  fft(r, i);  **int** pass = 1;  **for** (**int** j=0; j<8; j++)  {  **fprintf**(stdout, "%d Real part = %f Imaginary part = %f \n", j, r[j], i[j]);  **if** (r[j] != r\_res[j] | i[j] != i\_res[j]) {  pass = 1;  }  }  **if** (pass) {  **fprintf**(stdout, "----------Pass!------------\n");  **return** 0;  } **else** {  **fprintf**(stderr, "----------Fail!------------\n");  **return** 1;  }  } |

Листинг 2. Тестбэнч

## Результаты синтеза и симуляции

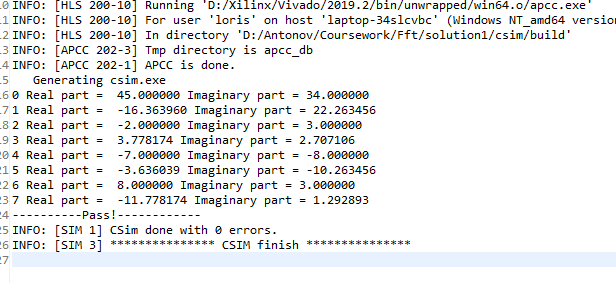
Результаты симуляции представлены на Рис 12, как можно увидеть тесты пройдены успешно.

Рис. 6. Результаты симуляции

Результат синтеза приведен ниже. Однако, средств статического анализа в Vivado HLS не хватило для того, чтобы рассчитать задержку и интервал инициализации в циклах

алгоритма.

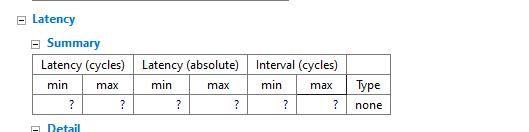


Рис. 7 Результаты синтеза

Для того, чтобы решить эту проблемы добавим директиву tripcount для цикла dft loop. В этой директиве мы можем указать значения max и min. Для FFT алгоритма на 8 точек максимальным значением будет 4, а минимальным 1. Это легко увидеть на рисунке с “бабочками”: на первом этапе мы имеем 1 бабочку, на втором - 2, а на 3 этапе уже делаем 4 бабочки.

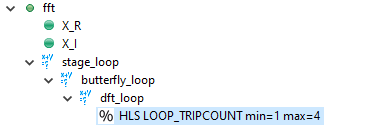


Рис. 8 Директива tripcount

Теперь после синтеза можно увидеть результаты:

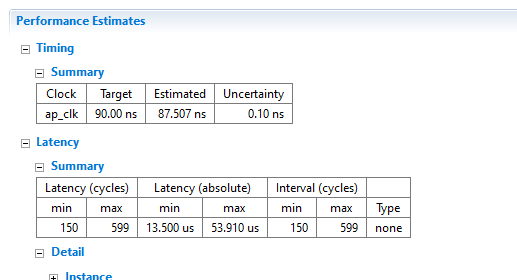


Рис.9 Performance Estimates

Получилось уложиться в заданный интервал. Однако, можно увидеть, что приходиться делать очень много итераций.

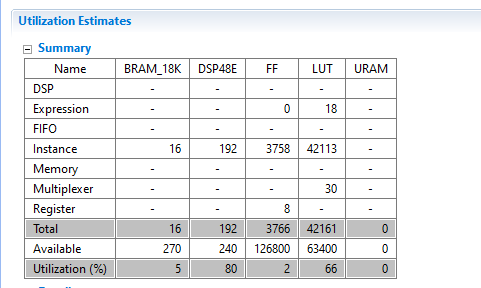


Рис.10 Utilization Estimates

Наша реализация требует очень много ресурсов, поэтому изначальную пришлось заменить на xc7a100tcsg324-2

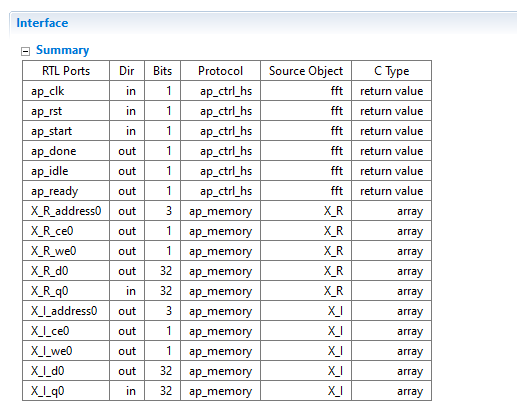
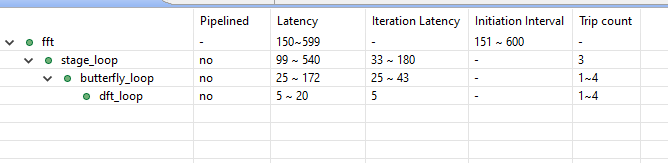
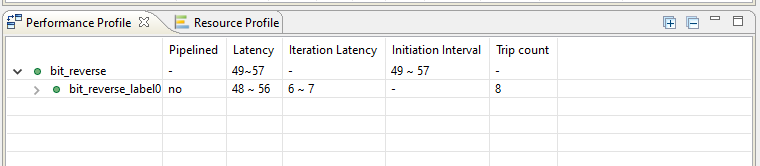


Рис.11 Interface

Для работы с массивами Vivado HLS сама выбрала интерфейс ap\_memory, так как он является стандартным.





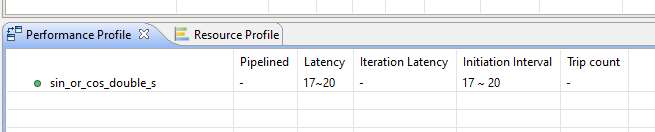
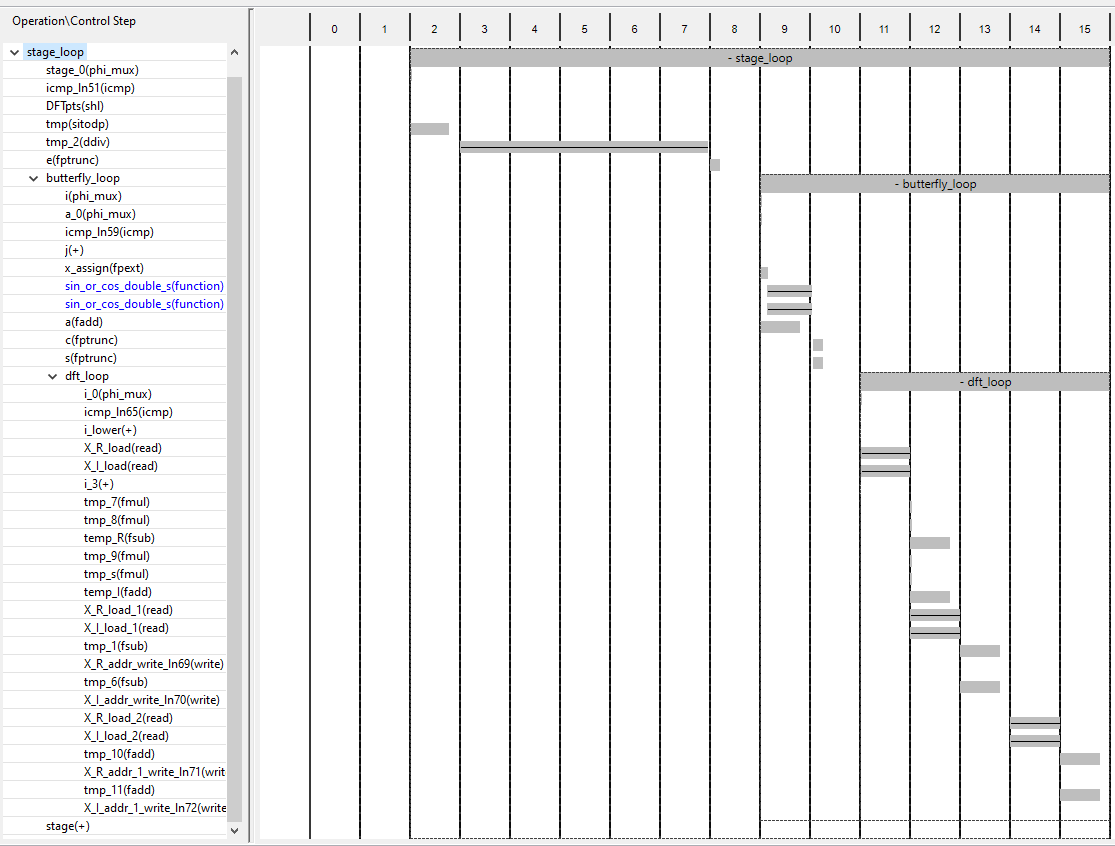


Рис.12 Performance profile

Так как задержка между итерациями очень большая, то не представляется возможности показать Schedule Viewer для всего алгоритма. Поэтому для начала посмотрим конкретно на функцию bit\_reverse, которая перемешивает входные массивы, чтобы сразу разделить четные и нечетные позиции для FFT, потом уже на основные этапы алгоритма.



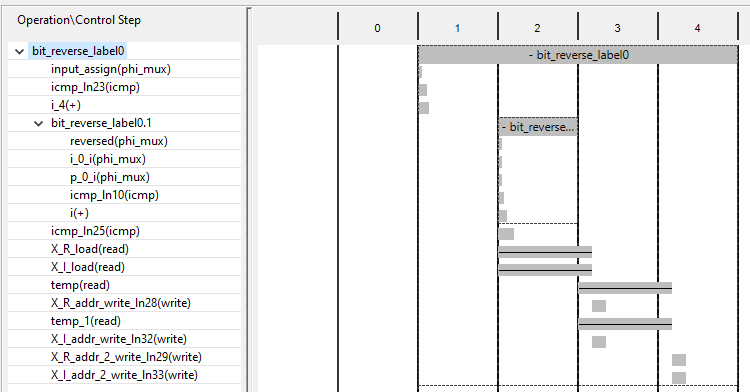


Рис.13 Schedule Viewer

На этапах основного алгоритма можно увидеть, что арифметические операции в dft-loop выполняются параллельно.

## Оптимизация bit\_reverse

Для начала попробуем оптимизировать функцию bit\_reverse, которая запускается в начале работы алгоритма. Попробуем запустить цикл как конвейер через директиву pipeline:

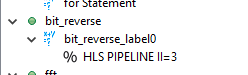


Рис.14 Директивы для bit\_reverse

По умолчанию II =1, и некоторые операции не успевают уложиться в 1 цикл, поэтому был выставлен II=3, чтобы этого избежать.

Теперь приведем анализ синтеза после оптимизации:

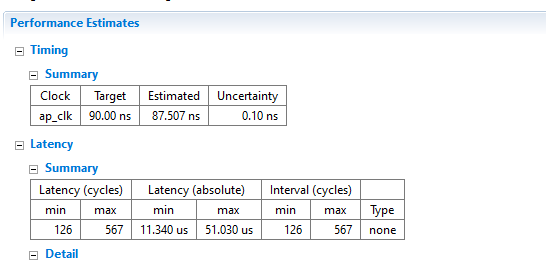


Рис.15 Задержка после оптимизации цикла в bit\_reverse

Уже можно увидеть, что задержка немного снизилась.

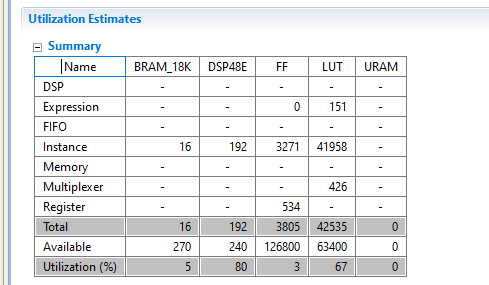


Рис.16 Потребление ресурсов после оптимизации цикла в bit\_reverse

Также несколько снизилось потребление ресурсов.

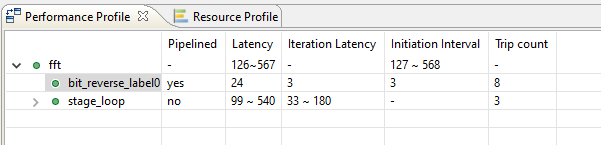


Рис.17 Performance profile

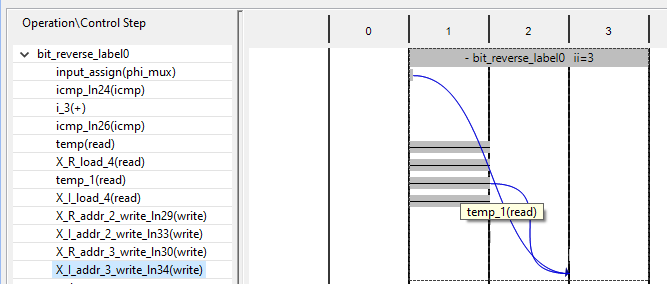


Рис.18 Schedule Viewer для bit\_reverse

Запись результирующих значений происходит в несколько тактов, именно поэтому не удалось выставить II < 3. Происходит это из за того, что не хватает memory портов

Попробуем поставить директиву pipeline на всю функцию bit\_reverse.

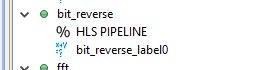


Рис.19 Директивы для bit\_reverse

Запустим еще раз синтез и проанализируем результаты:

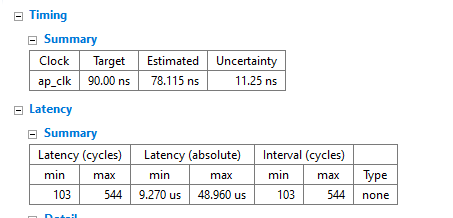


Рис.20 Задержка после оптимизации функции bit\_reverse

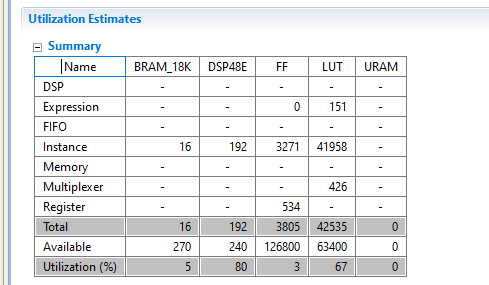


Рис.21 Потребление ресурсов после оптимизации цикла в bit\_reverse

Задержка еще более снизилась после вставки директивы на всю функцию, однако потребление ресурсов немного выросло.

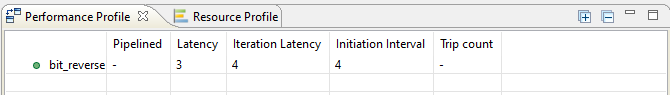


Рис.22 Performance profile

Теперь bit\_reverse вообще не содержит в себе цикла и работает всего за 4 такта.

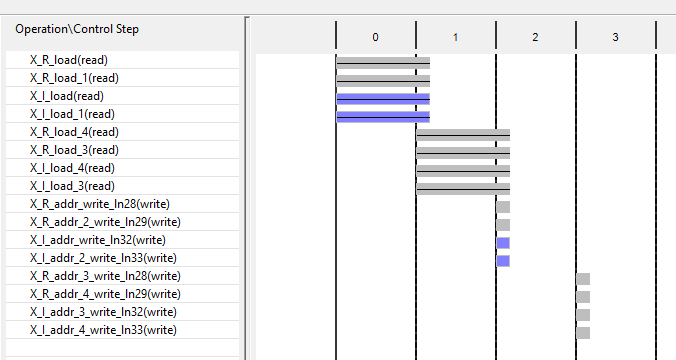


Рис.23 Schedule Viewer для bit\_reverse

Теперь можно посмотреть на Scheduler Viewer основных этапов, больше всего нас интересует dft-loop, так как основные операции проходят именно в нем.

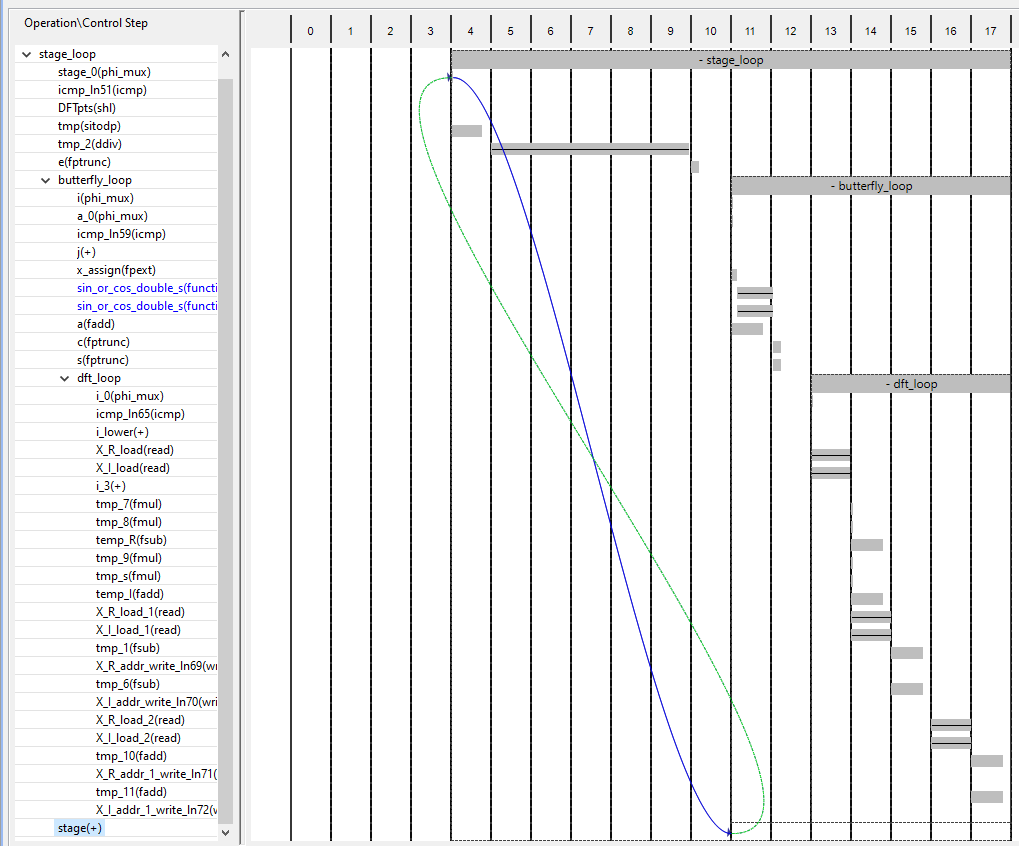


Рис.24 Schedule Viewer для dft-loop

Очень долго выполняется операция tmp\_2(ddiv), которая скорее всего отвечает за деление.

## Конвейеризация

Теперь попробуем, чтобы этапы работали одновременно с разными данными. Это можно сделать с помощью директивы dataflow. Реализация все также будет сделана для 8 точечного алгоритма FFT, поэтому мы имеем 4 этапа – 1 этап на переворачивание битов в адресах входного массива и еще 3, собственно, на сам FFT. Для конвейеризации нам понадобится переписать исходный код.

|  |
| --- |
| **#include** "math.h"  **#define** DTYPE **float**  **#define** SIZE 8  **#define** M 3  **#define** SIZE2 4  **unsigned** **int** **reverse\_bits**(**unsigned** **int** input) {  **int** i, rev = 0;  **for** (i = 0; i < M; i++) {  rev = (rev << 1) | (input & 1);  input = input >> 1;  }  **return** rev;  }  **void** **bit\_reverse**(DTYPE X\_R[SIZE], DTYPE X\_I[SIZE], DTYPE Stage\_R[SIZE], DTYPE Stage\_I[SIZE]) {  **#pragma** HLS PIPELINE  **unsigned** **int** reversed;  **unsigned** **int** i;  DTYPE temp;  bit\_reverse\_label0:  **for** (i = 0; i < SIZE; i++) {  reversed = reverse\_bits(i); // Find the bit reversed index  **if** (i <= reversed) {  // Swap the real values  Stage\_R[i] = X\_R[reversed];  Stage\_R[reversed] = X\_R[i];  // Swap the imaginary values  Stage\_I[i] = X\_I[reversed];  Stage\_I[reversed] = X\_I[i];  }  }  }  **void** **fft\_stage**(**int** stage, DTYPE X\_R[SIZE], DTYPE X\_I[SIZE],  DTYPE Out\_R[SIZE], DTYPE Out\_I[SIZE]) {  **int** DFTpts = 1 << stage; // DFT = 2^stage = points in sub DFT  **int** numBF = DFTpts / 2; // Butterfly WIDTHS in sub-DFT  **int** step = SIZE >> stage;  DTYPE k = 0;  DTYPE e = -6.283185307178 / DFTpts;  DTYPE a = 0.0;  // Perform butterflies for j-th stage  butterly\_loop:  **for** (**int** j = 0; j < numBF; j++) {  DTYPE c = **cos**(a);  DTYPE s = **sin**(a);  a = a + e;  // Compute butterflies that use same W\*\*k  dft\_loop:  **for** (**int** i = j; i < SIZE; i += DFTpts) {  **#pragma** HLS LOOP\_TRIPCOUNT min=1 max=4  **int** i\_lower = i + numBF; // index of lower point in butterfly  DTYPE temp\_R = X\_R[i\_lower] \* c - X\_I[i\_lower] \* s;  DTYPE temp\_I = X\_I[i\_lower] \* c + X\_R[i\_lower] \* s;  Out\_R[i\_lower] = X\_R[i] - temp\_R;  Out\_I[i\_lower] = X\_I[i] - temp\_I;  Out\_R[i] = X\_R[i] + temp\_R;  Out\_I[i] = X\_I[i] + temp\_I;  }  k += step;  }  }  **void** **fft\_streaming**(DTYPE X\_R[SIZE], DTYPE X\_I[SIZE], DTYPE OUT\_R[SIZE], DTYPE OUT\_I[SIZE]) {  **#pragma** HLS DATAFLOW  DTYPE Stage\_R[M][SIZE], Stage\_I[M][SIZE];  **#pragma** HLS array partition variable=Stage\_R dim=1 complete  **#pragma** HLS array partition variable=Stage\_I dim=1 complete  bit\_reverse(X\_R, X\_I, Stage\_R[0], Stage\_I[0]);  stage\_loop:  **for** (**int** stage = 1; stage < M; stage++) { // Do M-1 stages of butterflies  **#pragma** HLS unroll  fft\_stage(stage, Stage\_R[stage-1], Stage\_I[stage-1], Stage\_R[stage], Stage\_I[stage]);  }  fft\_stage(M, Stage\_R[M-1], Stage\_I[M-1], OUT\_R, OUT\_I);  } |

Листинг 3. Оптимизированный алгоритм

Посмотрим на результаты синтеза:

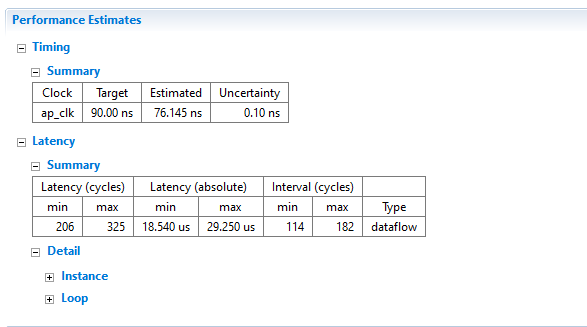


Рис.25 Performance estimates

Максимальное количество тактов почти в 2 сократилось, однако, минимальное количество возросло также возросло в 2 раза. А затраченное время уменьшилось на 2 ns.

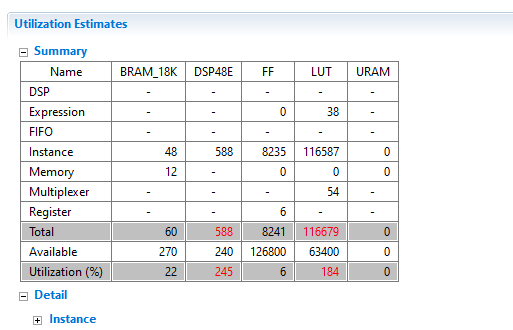


Рис.26 Utilization estimates

Очень сильно возросло потребление ресурсов (более чем в 2 раза для DSP48E), так что ресурсов выбранной платы опять стало не хватать.

Теперь проанализируем изменения в Schedule Viewer.

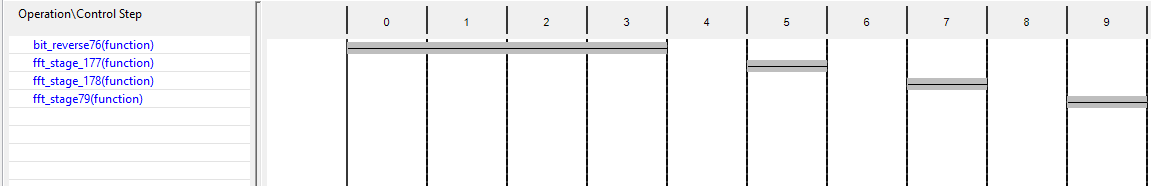


Рис.27 Schedule Viewer

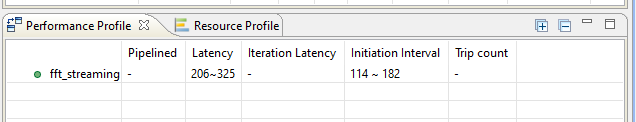
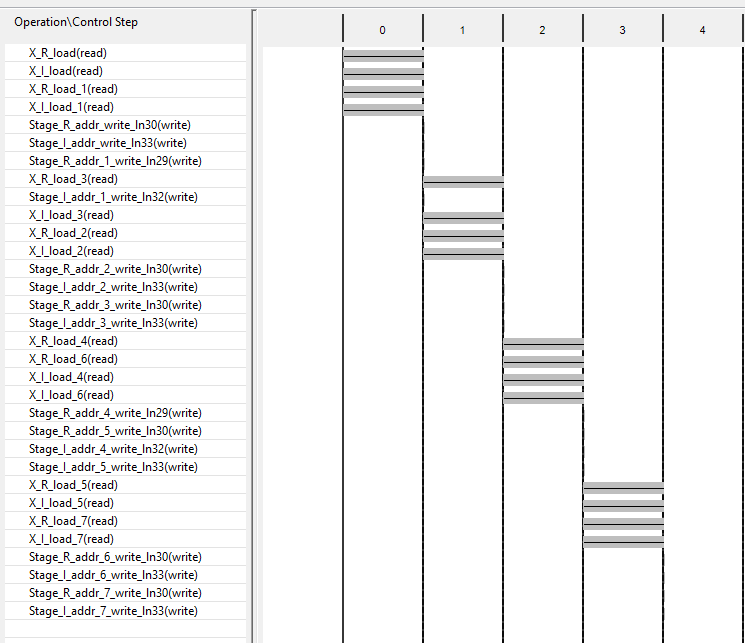
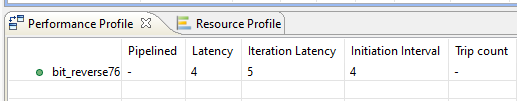


Рис.28 Performance profile

Для начала взглянем на bit\_reverse. Добавление дополнительных массивов для хранения результатов добавило новые операции и поэтому задержка увеличилась на 1 такт.





Цикл все так же раскладывается в 1 итерацию.

Теперь посмотрим на основные этапы алгоритма FFT.

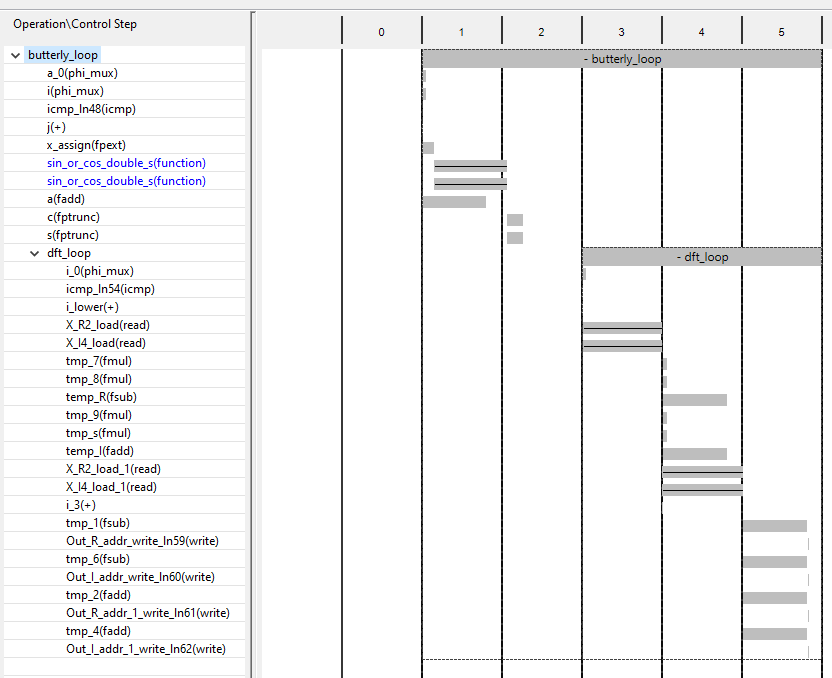
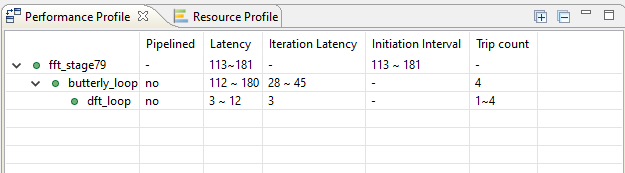
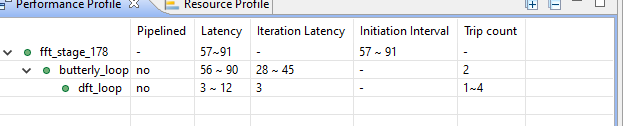


Рис.29 Scheduler Viewer для dft-loop

На Scheduler Viewer видно, что внутри butterfly-loop и dft-loop не различаются с предыдущей реализацией, а значит выигрыш был достигнут именно за счет конвейеризации самих этапов.





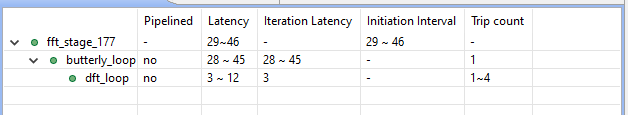


Рис.30 Performance profile для разных этапов FFT

Оптимизация с помощью конвейеризации позволила сократить задержку почти в 2 раза, однако, также и повысила количество задействуемых ресурсов.

Попробуем C/RTL симуляцию, подавая на вход несколько массивов в цикле.

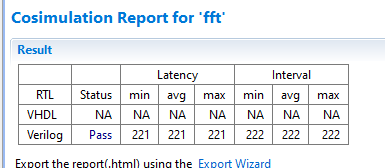


Рис.31 Результат неоптимизированной версии

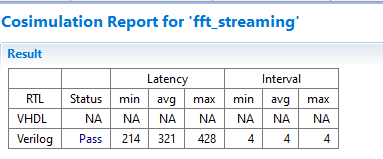


Рис.32 Результат оптимизированной версии

Из отчета видно, что оптимизированная версия считает конвейером сразу несколько задач, имея интервал между запусками всего 4 такта.

## Выводы

В ходе данной работы были разобраны разные методы для оптимизации, которые были разобраны на примере быстрого преобразования Фурье. Получилось получить сложность алгоритма log2(N + 1), где N это количество точек в алгоритме (размер входных данных).

Можно с уверенностью сказать, что различные методы оптимизации работают лучше всего совместно. Например, директива dataflow применяется на верхних уровнях функций, когда работа еще идет с целыми блоками данных, однако, она не поможет сильно увеличить производительность, если какой-то из этапов конвейера выполняется сильно дольше остальных, так как задержка между итерациями равна именно времени максимально долгого этапа. Поэтому для начала стоит оптимизировать маленькие этапы, допустим с помощью директивы pipeline, которая уже работает с конкретными элементами.

Также необходимо глубокое понимание алгоритма перед тем, как заниматься его оптимизацией, в данной работе вообще нельзя было бы получить какого анализа, если не посчитать сколько «бабочек» будет считаться на каждом этапе. Неверно выбранные значения для максимального и минимального значения испортили бы результаты всего анализа.