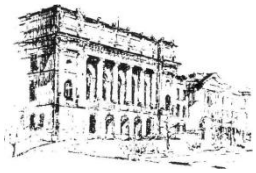


МИНИСТЕРСТВА НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Курсовой проект

Тема проекта: Фильтры с конечной импульсной характеристикой

по курсу: «Проектирование реконфигурируемых гибридных вычислительных систем»

Выполнил студент гр. 3540901/81501 Селиверстов Я.А.

Выполнил студент гр. 3540901/81501 Селиверстов С.А.

Руководитель, к.т.н. Антонов А. П.

Санкт-Петербург
2019

ОГЛАВЛЕНИЕ

1. Фильтры с конечной импульсной характеристикой .

1.1. Введение

Фильтры с конечной импульсной характеристикой (FIR) являются обычным явлением в приложениях цифровой обработки сигналов (DSP) - они, возможно, являются наиболее широко используемой операцией в этой области. Они хорошо подходят для реализации аппаратного обеспечения, поскольку они могут быть реализованы в виде высокооптимизированной архитектуры. Ключевым свойством является то, что они являются линейным преобразованием на смежных элементах сигнала. Это хорошо отображает структуры данных (например, FIFO или линии задержки ответвления), которые могут быть эффективно реализованы на аппаратном уровне. В общем, потоковые приложения обычно хорошо отображаются на FPGA, например, большинство примеров, которые мы представляем в книге, имеют своего рода потоковое поведение.

Два основных применения фильтра - восстановление сигнала и разделение сигнала. Разделение сигналов, возможно, является более распространенным вариантом использования: здесь мы пытаемся изолировать входной сигнал на разные части. Как правило, мы думаем о них как о разных частотных диапазонах, например, мы можем захотеть выполнить фильтр низких частот, чтобы удалить высокие частоты, которые не интерес. Или мы можем захотеть выполнить полосовой фильтр для определения присутствия конкретной частоты, чтобы демодулировать ее, например, для изолирующих тонов во время демодуляции с частотной манипуляцией. Восстановление сигнала относится к удалению шумов и других общих артефактов искажения, которые могли быть введены в сигнал, например, когда данные передаются по беспроводному каналу. Это включает в себя сглаживание сигнала и удаление компонента постоянного тока.

Цифровые КИХ-фильтры часто имеют дело с дискретным сигналом, генерируемым путем дискретизации непрерывного сигнала. Наиболее знакомая выборка выполняется во времени, то есть значения из сигнала принимаются в дискретных случаях. Они чаще всего отбираются через равные промежутки времени. Например, мы можем сэмплировать напряжение на антенне через равные промежутки времени с помощью аналого-цифрового преобразователя. В качестве альтернативы можно определить ток, создаваемый фотодиодом, для определения интенсивности света. В качестве альтернативы, образцы могут быть взяты в космосе. Например, мы можем сэмплировать значения различных местоположений в датчике изображения, состоящего из матрицы фотодиодов, для создания цифрового изображения. Более подробное описание сигналов и выборки можно найти в [1].

Формат данных в выборке изменяется в зависимости от приложения. Цифровые коммуникации часто используют комплексные числа (синфазные и квадратурные или значения I / Q) для представления выборки. Позже в этой главе мы опишем, как разработать сложный FIR-фильтр для обработки таких данных. Обработка изображений Мы часто думаем о пикселе как об образце. Пиксель может иметь несколько полей, например красный, зеленый и синий (RGB) цветовые каналы. Мы можем захотеть снова отфильтровать каждый из этих каналов по-разному в зависимости от приложения.

Цель этой главы - дать общее представление о процессе принятия алгоритма, создающего хороший аппаратный дизайн с использованием синтеза высокого уровня.

Первым шагом в этом процессе всегда является глубокое понимание самого алгоритма. Это позволяет нам упростить оптимизацию дизайна, например, реструктуризацию кода. В следующем разделе дается понимание теории и расчетов КИХ-

фильтра. В оставшейся части главы представлены различные методы HLSoptimizations для фильтра FIR. Они предназначены для обеспечения обзора этих оптимизаций. Каждый из них будет описан более подробно в последующих главах.

1.2. Предпосылки

Выходной сигнал фильтра с заданным импульсным входным сигналом является его импульсной характеристикой. Импульсная характеристика линейного, не зависящего от времени фильтра содержит полную информацию о фильтре. Как следует из названия, импульсный отклик КИХ-фильтра (линейный ограниченный по времени тип фильтра с ограничениями) конечен, то есть он всегда равен нулю вдали от нуля. Учитывая импульсную характеристику КИХ-фильтра, мы можем вычислить выходной сигнал для любого входного сигнала в процессе свертки. Этот процесс объединяет выборки импульсного отклика (также называемые коэффициентами или ответвлениями) с выборками входного сигнала для вычисления выборок выходного сигнала. Выходные данные фильтра могут быть рассчитаны другими способами (например, в частотной области), но для целей этой главы мы сосредоточимся на вычислениях во временной области. Свертка КИХ-фильтра с N-отводом с коэффициентами h [] с входным сигналом x [] описывается общим разностным уравнением:

$$y[i] = \sum_{j=0}^{N-1} h[j] * x[j-i] \tag{1}$$

Обратите внимание, что для вычисления одного значения выходного сигнала N-отводного фильтра требуется N умножений и N-1 дополнений.

Фильтры скользящего среднего значения представляют собой простую форму низкочастотного КИХ-фильтра, где все коэффициенты идентичны и суммируются в единицу. Например, в случае трехточечного движущегося фильтра, коэффициенты h = [13,13,13]. Из-за формы его сверточного ядра его также называют коробчатым автомобильным фильтром. В качестве альтернативы, вы можете думать о фильтре скользящего среднего как о получении среднего значения нескольких соседних выборок входного сигнала и их усреднении вместе. Мы можем видеть эту эквивалентность, подставляя 1 / N вместо h [j] в приведенном выше уравнении свертки и переставляя, чтобы получить семейное выражение для среднего из N элементов:

$$y[i] = \frac{1}{N} \sum_{j=0}^{N-1} x[j-i] \tag{2}$$

Каждая выборка в выходном сигнале может быть вычислена по вышеприведенному уравнению с использованием N-1 сложений и одного окончательного умножения на 1 / N. Даже окончательное умножение часто можно перегруппировать и объединить с другими операциями. В результате фильтры скользящего среднего проще вычислить, чем обычный FIR-фильтр. В частности, когда N = 3, мы выполняем эту операцию для вычисления y [2]:

$$y[12] = \frac{1}{3} \left(x[12] + x[11] + x[10]\right) \tag{3}$$

Этот фильтр является казуальным, что означает, что выходные данные являются функцией отсутствия будущих значений входных данных. Например, возможно и обычно изменить это так, чтобы среднее значение центрировалось на текущей выборке, то есть

y[12] = 13 · (x [11] + x [12] + x [13]). Хотя фундаментальная причинность является важным свойством для системного анализа, она менее важна для аппаратной реализации, поскольку конечный не причинный фильтр может быть сделан казуальным с буферизацией и / или переиндексированием данных.

Фильтры скользящего среднего можно использовать для сглаживания сигнала, например, для удаления случайного (в основном высокочастотного) шума. По мере того как количество отводов N увеличивается, мы делаем усреднение по большему количеству выборок и, соответственно, должны выполнять больше вычислений. Для фильтра скользящего среднего большие значения N соответствуют уменьшению полосы пропускания выходного сигнала. По сути, он действует как фильтр нижних частот (хотя и не очень оптимальный). Интуитивно, это должно иметь смысл. Поскольку мы усредняем все большее и большее количество выборок, мы исключаем более высокие вариации частоты во входном сигнале. То есть «сглаживание» эквивалентно уменьшению более высоких частот. Фильтр скользящего среднего является оптимальным для уменьшения белого шума при сохранении самого резкого шага, т. Е. Он создает наименьший шум для заданной резкости края.

Обратите внимание, что в общем случае коэффициенты фильтра могут быть созданы для создания множества различных типов фильтров: низкие частоты , высокочастотный, полосовой и т. д. Как правило, большее значение числа отводов обеспечивает большую степень свободы при проектировании фильтра, что обычно приводит к фильтрам с лучшими характеристиками. Существует значительное количество литературы, посвященной созданию коэффициентов фильтра с конкретные характеристики для данного приложения. При реализации фильтра фактические значения этих коэффициентов в значительной степени не имеют значения, и мы можем игнорировать то, как были получены сами коэффициенты. Однако, как мы видели в фильтре скользящих средних, структура фильтра или конкретные коэффициенты могут оказывать большое влияние на количество операций, которые необходимо выполнить. Например, симметричные фильтры имеют несколько отводов с одинаковым значением, которые можно сгруппировать, чтобы уменьшить количество умножений. В других случаях можно преобразовать умножение на известный постоянный коэффициент фильтра в операции сдвига и сложения [3]. В этом случае значения коэффициентов могут кардинально изменить производительность и область применения фильтра [4]. Но мы пока проигнорируем это и сосредоточимся на создании архитектур, которые имеют постоянные коэффициенты, но не пользуются преимуществами значений констант.

2.1. Базовая архитектура фильтра с конечным импульсным откликом

Рассмотрим код для 11-отводного КИХ-фильтра на рисунке 1. Функция принимает два аргумента: входную выборку x и выходную выборку y. Эта функция должна вызываться несколько раз для вычисления всего выходного сигнала, поскольку каждый раз, когда мы выполняем функцию, мы предоставляем одну входную выборку и получаем одну выходную выборку. Этот код удобен для моделирования потоковой архитектуры, поскольку мы можем вызывать его столько раз, сколько потребуется, когда становится доступным больше данных [5].

Коэффициенты для фильтра хранятся в массиве c [], объявленном внутри функции. Это статически определенные константы. Обратите внимание, что коэффициенты симметричны. то есть они отражаются вокруг значения центра c [5] = 500. Многие КИХ-фильтры имеют этот тип симметрии. Мы могли бы воспользоваться этим, чтобы уменьшить объем памяти, необходимый для массива c [].

Код использует typedeff или разные переменные. Хотя в этом нет необходимости, это удобно для изменения типов данных. Как мы обсудим позже, оптимизация ширины в битах - в частности, установка количества целых и дробных бит для каждой переменной - может обеспечить значительные преимущества с точки зрения производительности и площади.

Задание 1а. Рассмотрите реализацию FIR-фильтра на рисуноке1.

Задание 1б. Перепишите код так, чтобы он использовал преимущества симметрии, найденной в коэффициентах. То есть измените с [] так, чтобы в нем было шесть элементов (от с [0] до с [5]). Какие изменения необходимы в остальной части кода? Как это влияет на количество ресурсов? Как это меняет производительность?

```
#define N 11
#include "ap_int.h"

typedef int coef_t;
typedef int data_t;
typedef int acc_t;
void fir(data_t *y, data_t x) {
    coef_t c[N] = {
        53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53
    };
    static
     data_t shift_reg[N];
    acc_t acc;
    int i;
    acc = 0;
    Shift_Accum_Loop:
    for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```

Рисунок 1.Функционально правильная, но крайне неоптимизированная реализация 11-отводного КИХ-фильтра.

```
#define N 6
#include "ap_int.h"

typedef int coef_t;
typedef int data_t;
typedef int acc_t;
void fir(data_t *y, data_t x) {
    coef_t c[N] = {
        53, 0, -91, 0, 313, 500
    };
    static
     data_t shift_reg[N];
    acc_t acc;
    int i;
    acc = 0;
    Shift_Accum_Loop:
    for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```

Рисунок 1.2.Функционально правильная, но крайне неоптимизированная реализация 6-отводного КИХ-фильтра

Код написан как функция потоковой передачи. Он получает один образец за раз, и поэтому он должен хранить предыдущие образцы. Так как это фильтр с 11 taps (число регистров в цепочке линии задержки), мы должны сохранить предыдущие 10 taps.Это цель массива shift_reg []. Этот массив объявлен статическим, поскольку данные должны быть постоянными при нескольких вызовах функции.

Цикл for выполняет две фундаментальные задачи в каждой итерации. Сначала он выполняет операцию умножения и накопления на входных выборках (текущая входная выборка x и предыдущие входные выборки, сохраненные в shift_reg []). Каждая итерация цикла выполняет умножение одной из констант на одну из выборок и сохраняет текущую сумму в переменной acc. Цикл также сдвигает значения через массив смещения, который работает как FIFO. Он сохраняет входной образец x в shift_array [0] и перемещает предыдущие элементы «вверх» через shift_array:

```
shift array[10] = shift array[9]
shift array[9] = shift array[8]
shift array[8] = shift array[7]
……
shift array[2] = shift array[1]
shift array[1] = shift array[0]
shift array[0] = x
```

После завершения цикла for переменная acc получает полный результат свертки входных выборок с массивом коэффициентов КИХ. Окончательный результат записывается в аргумент функции y, который действует как выходной порт этой функции КИХ. Это завершает процесс потоковой передачи для вычисления одного выходного значения КИХ фильтра. Эта функция не обеспечивает коэффициентную реализацию КИХ - фильтра. Он в значительной степени последовательный и использует значительное количество ненужной логики управления. В следующих разделах описан ряд различных оптимизаций, которые улучшают его производительность.

2.2. Программная реализация базовой архитектуры фильтра с конечным импульсным откликом в Vivado HLS

Задание 1а. Рассмотрите реализацию FIR-фильтра на рисуноке1.

Решение 1а. Рассмотрим реализацию FIR-фильтра на рисуноке1.1.
Программный код имеет вид Листинг П.1.

```
#define N 11
//#include "ap_int.h"
typedef int coef_t;
typedef int data_t;
typedef int acc_t;
void fir(data_t *y, data_t x) {
	coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
	static
	data_t shift_reg[N];
	acc_t acc;
	int i;
	acc = 0;
Shift_Accum_Loop:
	for (i = N - 1; i >= 0; i--) {
		if (i == 0) {
			acc += x * c[0];
			shift_reg[0] = x;
		} else {
			shift_reg[i] = shift_reg[i - 1];
			acc += shift_reg[i] * c[i];
		}
	}
	*y = acc;
}
```

Листинг.П.1.

Выполним C synthesis. Результаты синтеза представлены на рис. П.1.

```
INFO: [HLS 200-10] -- Generating RTL for module 'fir'
INFO: [HLS 200-10] ------------------------------------------------------------
INFO: [RTGEN 206-500] Setting interface mode on port 'fir/y' to 'ap_vld'.
INFO: [RTGEN 206-500] Setting interface mode on port 'fir/x' to 'ap_none'.
INFO: [RTGEN 206-500] Setting interface mode on function 'fir' to 'ap_ctrl_hs'.
INFO: [RTGEN 206-100] Finished creating RTL model for 'fir'.
INFO: [HLS 200-111]  Elapsed time: 0.105 seconds; current allocated memory: 74.703 MB.
INFO: [RTMG 210-278] Implementing memory 'fir_shift_reg_ram (RAM)' using distributed RAMs with power-on initialization.
INFO: [RTMG 210-279] Implementing memory 'fir_c1_rom' using distributed ROMs.
INFO: [HLS 200-111] Finished generating all RTL models Time (s): cpu = 00:00:01 ; elapsed = 00:00:07 . Memory (MB): peak
INFO: [SYSC 207-301] Generating SystemC RTL for fir.
INFO: [VHDL 208-304] Generating VHDL RTL for fir.
INFO: [VLOG 209-307] Generating Verilog RTL for fir.
INFO: [HLS 200-112] Total elapsed time: 7.1 seconds; peak allocated memory: 74.703 MB.
Finished C synthesis.
```

Рисунок. П.1.

Оценка производительности представлена на рис. П.2

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
23	45	23	45	none

Рисунок. П.2.

Оценка использованных ресурсов представлена на рисунке П.3.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	6	0	107
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	74	8
Multiplexer	-	-	-	120
Register	-	-	213	-
Total	0	6	287	235
Available	40	40	16000	8000
Utilization (%)	0	15	1	2

Рисунок П.3.

Задание 1б. Перепишите код так, чтобы он использовал преимущества симметрии, найденной в коэффициентах. То есть измените с [] так, чтобы в нем было шесть элементов (от с [0] до с [5]).

Решение 1б. Перепишем код согласно рисунку 1.2. так, чтобы он использовал преимущества симметрии, найденной в коэффициентах. То есть измените с [] так, чтобы в нем было шесть элементов (от с [0] до с [5]). Оценим изменения, которые необходимы в остальной части кода и посмотрим как это влияет на количество ресурсов и как это меняет производительность (Задание 1 на стр.6)

Программный код имеет вид Листинг.П.2.

```
#define N 11
//#include "ap_int.h"

typedef int coef_t;
```

```
typedef int data_t;
typedef int acc_t;

void fir2(data_t *y, data_t x) {
   coef_t cc[N/2] = {53, 0, -91, 0, 313 };
        coef_t c[N] = {cc[0], cc[1], cc[2], cc[3], cc[4], 500, cc[4], cc[3], cc[2], cc[1], cc[0]};
        static
        data_t shift_reg[N];
        acc_t acc;
        int i;
        acc = 0;
Shift_Accum_Loop:
        for (i = N - 1; i >= 0; i--) {
                if (i == 0) {
                        acc += x * c[0];
                        shift_reg[0] = x;
                } else {
                        shift_reg[i] = shift_reg[i - 1];
                        acc += shift_reg[i] * c[i];
                }
        }
        *y = acc;
```

Листинг П.2.

Выполним C synthesis. Результаты синтеза представлены на рис. П.4.

```
INFO: [HLS 200-10] -- Generating RTL for module 'fir'
INFO: [HLS 200-10] ------------------------------------------------------------
INFO: [RTGEN 206-500] Setting interface mode on port 'fir/y' to 'ap_vld'.
INFO: [RTGEN 206-500] Setting interface mode on port 'fir/x' to 'ap_none'.
INFO: [RTGEN 206-500] Setting interface mode on function 'fir' to 'ap_ctrl_hs'.
INFO: [RTGEN 206-100] Finished creating RTL model for 'fir'.
INFO: [HLS 200-111]  Elapsed time: 0.105 seconds; current allocated memory: 74.703 MB.
INFO: [RTMG 210-278] Implementing memory 'fir_shift_reg_ram (RAM)' using distributed RAMs with power-on initialization.
INFO: [RTMG 210-279] Implementing memory 'fir_c1_rom' using distributed ROMs.
INFO: [HLS 200-111] Finished generating all RTL models Time (s): cpu = 00:00:01 ; elapsed = 00:00:07 . Memory (MB): peak
INFO: [SYSC 207-301] Generating SystemC RTL for fir.
INFO: [VHDL 208-304] Generating VHDL RTL for fir.
INFO: [VLOG 209-307] Generating Verilog RTL for fir.
INFO: [HLS 200-112] Total elapsed time: 7.1 seconds; peak allocated memory: 74.703 MB.
Finished C synthesis.
```

Рисунок. П.4.

Оценка производительности представлена на рис. П.5.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
23	45	23	45	none

Рисунок. П.5.

Оценка использованных ресурсов представлена на рисунке П.6.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	6	0	107
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	74	8
Multiplexer	-	-	-	120
Register	-	-	213	-
Total	0	6	287	235
Available	40	40	16000	8000
Utilization (%)	0	15	1	2

Рисунок. П.6.

В результате программной реализации выяснено, что данный массив коэффициентов является константным. В результате синтеза как в первом, так и во втором случае получается одинаковый результат. Это обусловлено тем, что Vivaldo понимает, что данные переменные также являются константами и не имеет смысла использовать лишнее ресурсы на них. Таким образом разработчик частично защищен от подобных странных решений(зачем заводить две константы, когда можно завести одну).

3. Расчет производительности

Прежде чем мы перейдем к оптимизации, необходимо определить точные показатели. При определении производительности устройства важно тщательно указать метрику. Например, есть много разных способов указать, как быстро работает ваш фильтр. Например, вы можете сказать, что он работает со скоростью X (бит/сек). Или что он может выполнять Y (операций/сек). Другие общие показатели производительности, специально предназначенные для КИХ-фильтров, говорят о количестве операций фильтра в секунду. Еще одной метрикой является операция умножение-сложение: MAC / сек. В вычислительной технике, особенно в цифровой обработке сигналов совмещённое умножение-сложение (multiply-accumulate) — распространённая операция, при которой умножаются два числа и складываются с аккумулятором. Каждая из этих операций в некоторой степени связана друг с другом, но при сравнении разных реализаций важно сравнивать «яблоки с яблоками». Например, прямое сравнение одного устройства с производительностью в (битов/сек) с другим с производительностью (операций фильтрации/сек) может вводить в заблуждение; Для полного понимания относительных преимуществ проектов необходимо сравнить их, используя один и тот же показатель. И для этого может потребоваться дополнительная информация, например, для перехода от (операций фильтрации/сек) к (битам/сек) требуется информация о размере входных и выходных данных.

Все вышеупомянутые метрики используют секунды. Инструменты синтеза высокого уровня рассматривают производительность устройств с точки зрения количества циклов и частоты часов. Частота обратно пропорциональна времени, которое требуется для завершения одного тактового цикла. Использование их обоих дает нам количество времени в секундах для выполнения какой-либо операции. Количество циклов и тактовая частота являются важными: проект, который занимает один цикл, но с очень низкой частотой, не является необходимым лучше, чем другой дизайн, который занимает 10 тактов, но работает на гораздо более высокой частоте.

Тактовая частота - это сложная функция, которую инструмент Vivado HLS пытается оптимизировать наряду с количеством циклов. Обратите внимание, что для

инструмента Vivado HLS можно указать целевую частоту. Это делается с помощью команды create_clock tcl. Например, команда create_clock-period 5 предписывает инструменту выбрать тактовый период 5 нс и эквивалентную тактовую частоту 200 МГц.

Обратите внимание, что это только целевая тактовая частота, и в первую очередь она влияет на то, насколько много операций выполняет инструмент. После генерации RTL инструмент Vivado HLS предоставляет начальную временную оценку относительно этой тактовой частоты. Тем не менее, остается некоторая неопределенность в работе схемы, которая разрешается только после того, как проект полностью размещен и размечен.

Хотя достижение более высоких частот часто имеет решающее значение для достижения более высокой производительности, увеличение целевой тактовой частоты не обязательно является оптимальным с точки зрения всей системы. Более низкие частоты дают инструменту больше возможностей для объединения нескольких зависимых операций в одном цикле, процесс, называемый цепочкой операций. Иногда это может обеспечить более высокую производительность за счет оптимизации оптимизации логического синтеза и увеличения объема кода, который может поместиться на устройстве. Улучшенная цепочка операций также может улучшить (т.е. уменьшить) интервал инициализации конвейеров с повторениями. В общем, хорошим вариантом является предоставление ограниченной, но не слишком ограниченной целевой задержки. Что-то в диапазоне от 5 до 10 нс обычно является хорошим начальным вариантом. Как только вы оптимизируете свой дизайн, вы можете изменять период времени и наблюдать за результатами. Мы опишем операции цепочки более подробно в следующем разделе.

Поскольку Vivado HLS имеет дело с оценками тактовой частоты, он включает некоторый запас для учета того факта, что в оценке есть некоторая ошибка. Цель этого запаса состоит в том, чтобы обеспечить достаточную временную задержку в проекте, чтобы сгенерированные RTL могли быть успешно размещены и маршрутизированы. Этим запасом можно напрямую управлять с помощью команды TCL set_clock_unterminty. Обратите внимание, что эта команда влияет только на RTL, сгенерированный HLS, и отличается от концепции неопределенности часов в ограничениях времени уровня RTL. Временные ограничения, сгенерированные Vivado HLS для потока реализации RTL, основаны исключительно на целевом тактовом периоде.

Также необходимо сопоставить выполняемую задачу с метрикой производительности, которую вы рассчитываете. В нашем примере каждое выполнение функции КИХ приводит к одной выходной выборке. Но мы выполняем N = 11 операций умножения-сложения для каждого выполнения КИХ. Следовательно, если интересующий вас показатель - это (MACs / сек), вы должны рассчитать задержку задачи для КИХ в секундах, а затем разделить ее на 11, чтобы получить время, необходимое для выполнения эквивалента одной операции MAC.

Расчет производительности становится еще сложнее, поскольку мы выполняем конвейерную обработку и другие оптимизации. В этом случае важно понимать разницу между интервалом задачи и задержкой задачи. Это хорошее время, чтобы освежить ваше понимание этих двух показателей производительности. Это обсуждалось в разделе 1.4. И мы продолжим обсуждение того, как разные оптимизации влияют на разные показатели производительности.

4.1. Цепь операций

Цепочка операций является важной оптимизацией, которую выполняет Vivado HLS для оптимизации окончательного проекта. Это не то, над чем инженер имеет большой контроль, но важно, чтобы инженер понимал, как это работает, особенно в отношении производительности. Рассмотрим операцию умножение-сложение, которая выполняется в

отводе (регистрах) КИХ-фильтра. Предположим, что операция сложения занимает 2 нс, а операция умножения занимает 3 нс. Если мы установим период синхронизации на 1 нс (или эквивалентно тактовой частоте 1 ГГц), то для завершения операции MAC потребуется 5 циклов. Это изображено на рисунке 2 а). Операция умножения выполняется в течение 3 циклов, а операция сложения выполняется в течение 2 циклов.

Общее время для операции MAC составляет 5 циклов x 1 нс за цикл = 5 нс. Таким образом, мы можем выполнить 1/5 нс = 200 миллионов MAC / сек.

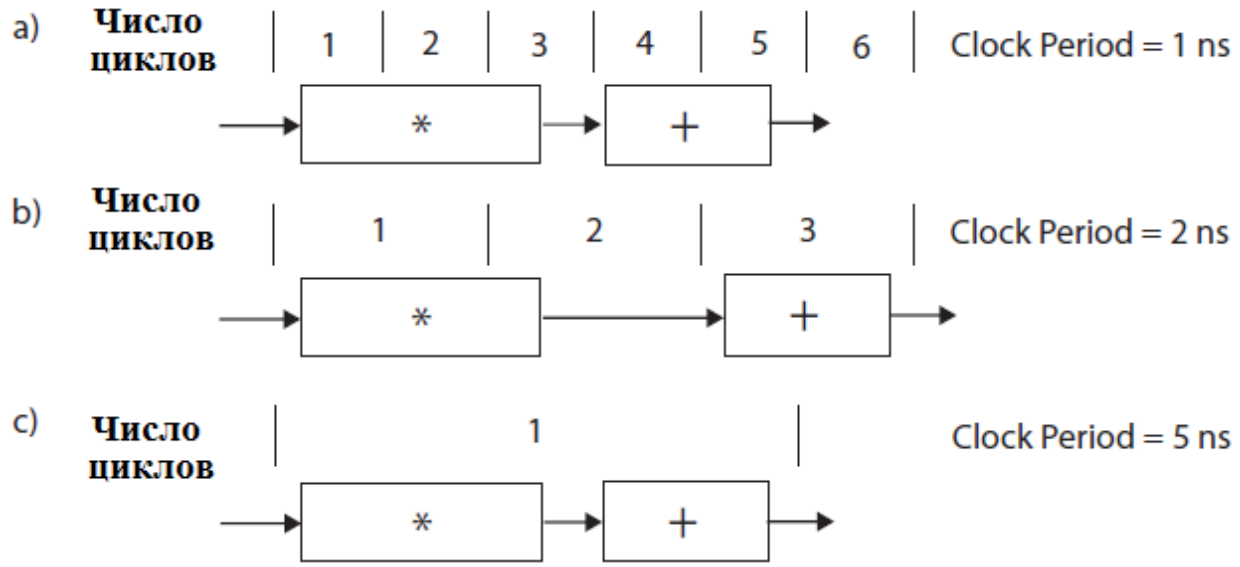


Рисунок 2. Производительность операции умножения – сложения изменяется в зависимости от целевого тактового периода. Предположим, что операция умножения занимает 3 нс, а операция сложения - 2 нс.
Часть a) имеет тактовый период 1 нс, и одна операция MAC занимает 5 циклов. Таким образом, производительность составляет 200 миллионов MAC / сек.
Часть b) имеет тактовый период 2 нс, и MAC занимает 3 цикла, что приводит к приблизительно 167 миллионам MAC / сек.
Часть c) имеет тактовый период 5 нс. Используя цепочку операций, операция MAC занимает 1 цикл в течение тактового периода 200 миллионов MAC / сек.

Если мы увеличим тактовый период до 2 нс, операция умножения теперь охватывает два цикла, и операция сложения должна ждать, пока не начнется цикл 3. Этот процесс может завершиться за один цикл. Таким образом, операция MAC требует 3 циклов, так что всего 6 нс для завершения. Это позволяет нам выполнять примерно 167 миллионов MAC/ сек. Этот результат ниже, чем предыдущий результат с тактовым периодом 1 нс. Это можно объяснить тем, что существует "dead time " в цикле 2, где никакие операции не выполняются.

Однако не всегда верно, что увеличение тактового периода приводит к ухудшению производительности. Например, если мы устанавливаем тактовый период равным 5 нс, мы можем выполнить операции умножения и сложения в одном и том же цикле, используя цепочку операций. Это показано на рисунке 2c). Таким образом, операция MAC занимает 1 цикл, где каждый цикл равен 5 нс, поэтому мы можем выполнить 200 миллионов MAC / секунду. Это та же производительность, что и на рисунке 2a), где тактовый период быстрее (1 нс).

До сих пор мы выполняли цепочку только из двух операций в одном цикле. Можно объединить несколько операций в одном цикле. Например, если период синхронизации

равен 10 нс, мы могли бы выполнить 5 операций добавления последовательно, или мы могли бы выполнить две последовательные операции MAC.

Становиться очевидным, что период синхронизации играет важную роль в том, как программа Vivado HLS оптимизирует дизайн. Тем не менее, важно иметь хорошее представление о том, как эта программа может работать. Это позволит вам лучше понимать результаты и даже писать более оптимизированный код.

Хотя Vivado HLS может генерировать различное оборудование для разных целевых тактовых периодов, общая оптимизация производительности и определение оптимального целевого тактового периода все еще требуют некоторой креативности со стороны пользователя. По большей части, мы рекомендуем придерживаться небольшого подмножества тактов. Например, в проектах мы предлагаем установить период синхронизации на 10 нс и сосредоточиться на понимании того, как другие оптимизации, такие как конвейерная обработка, могут использоваться для создания различных архитектур. Эту тактовую частоту 100 МГц относительно легко достичь, но она обеспечивает хороший результат первого порядка. Конечно, можно создавать проекты, которые работают с более высокой тактовой частотой. Возможны 200 МГц и более быстрые конструкции, но часто они требуют более тщательного баланса между тактовыми частотами и другими целями оптимизации Вы можете изменить целевой период времени и наблюдать разницу в производительности. К сожалению, нет хорошего правила выбора оптимальной частоты.

Задание 2. Измените период синхронизации для базовой архитектуры КИХ-фильтра (Листинг 1) с 10 нс до 1 нс с шагом 1 нс. Какой такт обеспечивает наилучшую производительность? Что дает лучший район? Как вы думаете, почему это так? Видите ли вы какие-либо тенденции?

4.2. Программная реализация цепи операций:изменение периода синхронизации.

Задание 2. Измените период синхронизации для базовой архитектуры КИХ-фильтра (Листинг 1) с 10 нс до 1 нс с шагом 1 нс. Какой такт обеспечивает наилучшую производительность? Что дает лучший район? Как вы думаете, почему это так? Видите ли вы какие-либо тенденции?

Решение 2. Изменим период синхронизации для базовой архитектуры КИХ-фильтра (Листинг П.1) с 10 нс до 1 нс с шагом 1 нс. Определим какой такт обеспечивает наилучшую производительность?

```
#define N 11
//#include "ap_int.h"

typedef int coef_t;
typedef int data_t;
typedef int acc_t;
void fir(data_t *y, data_t x) {
        coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
        static
        data_t shift_reg[N];
        acc_t acc;
        int i;
        acc = 0;
```

```
Shift_Accum_Loop:
        for (i = N - 1; i >= 0; i--) {
                if (i == 0) {
                        acc += x * c[0];
                        shift_reg[0] = x;
                } else {
                        shift_reg[i] = shift_reg[i - 1];
                        acc += shift_reg[i] * c[i];
                }
        }
        *y = acc;
}
```

Листинг П.1. (Дублирование кода)

Установим clock_period 10.

Отчет синтеза представлен на рисунке П.7.

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
23	45	23	45	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	22	44	2 ~ 4	-	-	11	no

Рисунок. П.7.

Оценка использованных ресурсов представлена на рис. П.8.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	6	0	107
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	74	8
Multiplexer	-	-	-	120
Register	-	-	213	-
Total	0	6	287	235
Available	40	40	16000	8000
Utilization (%)	0	15	1	2

Рисунок. П.8.

Диаграмма расписания просмотра (Schedule viewer) представлена на рисунке П.9.

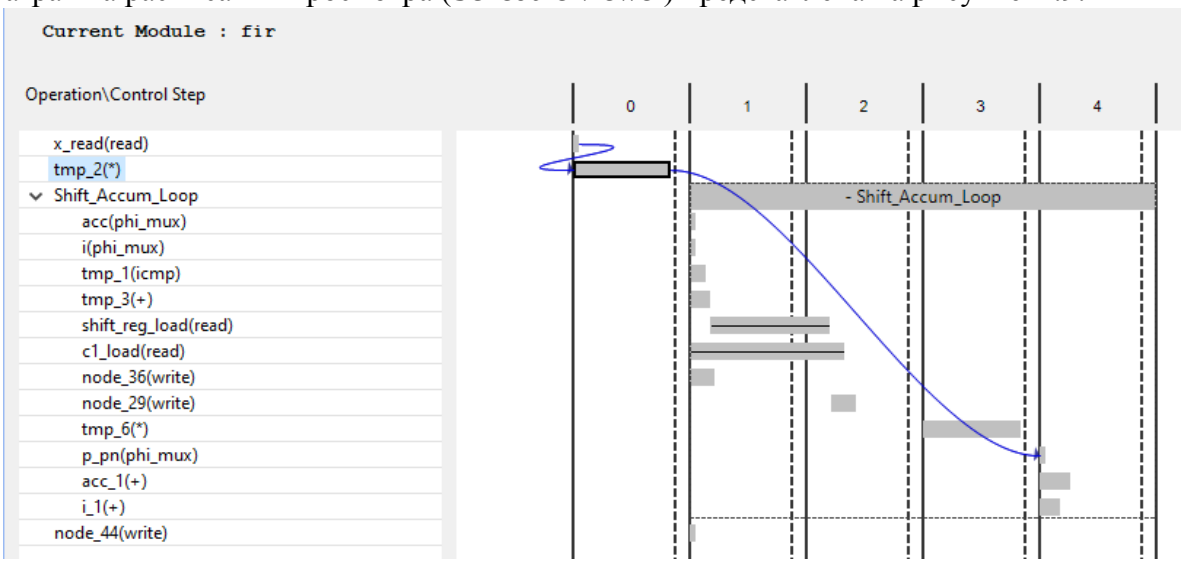


Рисунок. П.9.

Установим clock_period 9.

Отчет синтеза представлен на рисунке П.10.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	9.00	6.860	1.13

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
24	57	24	57	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	22	55	2 ~ 5	-	-	11	no

Рисунок П.10.

Оценка использованных ресурсов представлена на рис. П.11

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	65
FIFO	-	-	-	-
Instance	-	8	330	100
Memory	0	-	74	8
Multiplexer	-	-	-	128
Register	-	-	215	-
Total	0	8	619	301
Available	40	40	16000	8000
Utilization (%)	0	20	3	3

Рисунок. П.11.

Установим clock_period 8.

Отчет синтеза представлен на рисунке П.12.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	8.00	6.860	1.00

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
24	57	24	57	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	22	55	2 ~ 5	-	-	11	no

Рисунок. П.12.

Оценка использованных ресурсов представлена на рис. П.13

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	65
FIFO	-	-	-	-
Instance	-	8	330	100
Memory	0	-	74	8
Multiplexer	-	-	-	128
Register	-	-	215	-
Total	0	8	619	301
Available	40	40	16000	8000
Utilization (%)	0	20	3	3

Рисунок. П.13.

Установим clock_period 7.

Отчет синтеза представлен на рисунке П.14.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	7.00	5.690	0.88

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
25	69	25	69	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	22	66	2 ~ 6	-	-	11	no

Рисунок. П.14.

Оценка использованных ресурсов представлена на рис. П.15

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	65
FIFO	-	-	-	-
Instance	-	8	332	98
Memory	0	-	74	8
Multiplexer	-	-	-	134
Register	-	-	217	-
Total	0	8	623	305
Available	40	40	16000	8000
Utilization (%)	0	20	3	3

Рисунок. П.15.

Диаграмма расписания просмотра (Schedule viewer) представлена на рисунке П.16.

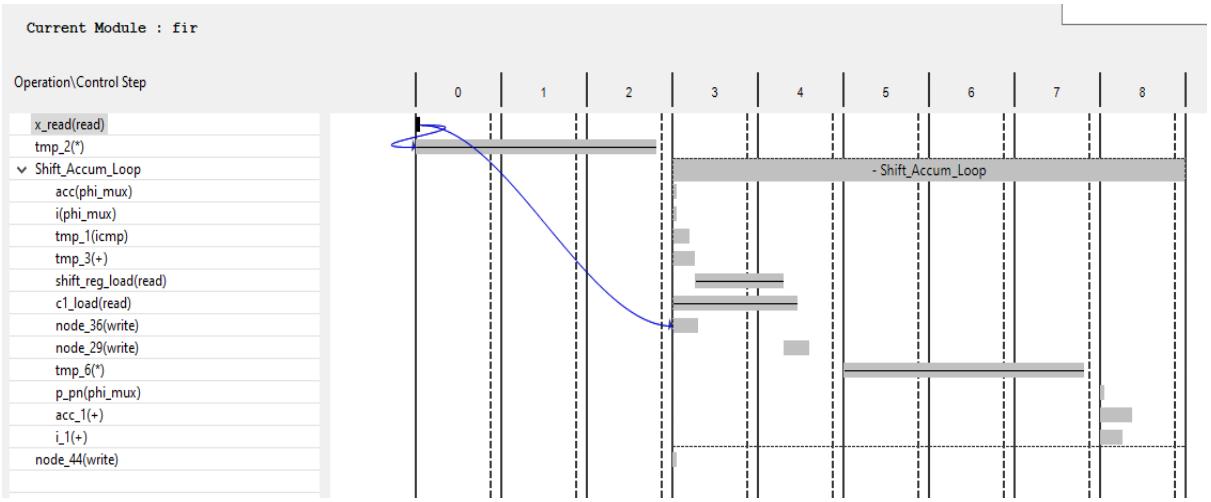


Рисунок. П.16.

Установим clock_period 6.

Отчет синтеза представлен на рисунке П.17.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	6.00	4.366	0.75

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
27	93	27	93	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	22	88	2 ~ 8	-	-	11	no

Рисунок. П.17.

Оценка использованных ресурсов представлена на рис. П.18.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	65
FIFO	-	-	-	-
Instance	-	8	430	2
Memory	0	-	74	8
Multiplexer	-	-	-	146
Register	-	-	221	-
Total	0	8	725	221
Available	40	40	16000	8000
Utilization (%)	0	20	4	2

Рисунок. П.18.

Установим clock_period 5.

Отчет синтеза представлен на рисунке П.19.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	4.366	0.63

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
27	93	27	93	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	22	88	2 ~ 8	-	-	11	no

Рисунок. П.19.

Оценка использованных ресурсов представлена на рис. П.20.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	65
FIFO	-	-	-	-
Instance	-	8	430	2
Memory	0	-	74	8
Multiplexer	-	-	-	146
Register	-	-	221	-
Total	0	8	725	221
Available	40	40	16000	8000
Utilization (%)	0	20	4	2

Рисунок. П.20.

Установим clock_period 4.

Отчет синтеза представлен на рисунке П.21.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.257	0.50

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
40	139	40	139	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	33	132	3 ~ 12	-	-	11	no

Рисунок. П.21.

Оценка использованных ресурсов представлена на рис. П.22.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	65
FIFO	-	-	-	-
Instance	-	8	494	38
Memory	0	-	74	8
Multiplexer	-	-	-	175
Register	-	-	264	-
Total	0	8	832	286
Available	40	40	16000	8000
Utilization (%)	0	20	5	3

Рисунок. П.22.

Установим clock_period 3.

Отчет синтеза представлен на рисунке П.23.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	3.00	2.152	0.38

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
51	150	51	150	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	44	143	4 ~ 13	-	-	11	no

Рисунок. П.23.

Оценка использованных ресурсов представлена на рис. П.24.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	26
FIFO	-	-	-	-
Instance	-	8	647	75
Memory	0	-	74	8
Multiplexer	-	-	-	179
Register	-	-	265	-
Total	0	8	986	288
Available	40	40	16000	8000
Utilization (%)	0	20	6	3

Рисунок. П.24.

Установим clock_period 2.

Отчет синтеза представлен на рисунке П.25.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	2.00	2.298	0.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
63	294	63	294	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	44	275	4 ~ 25	-	-	11	no

Рисунок. П.25.

Оценка использованных ресурсов представлена на рис. П.26.

Utilization Estimates

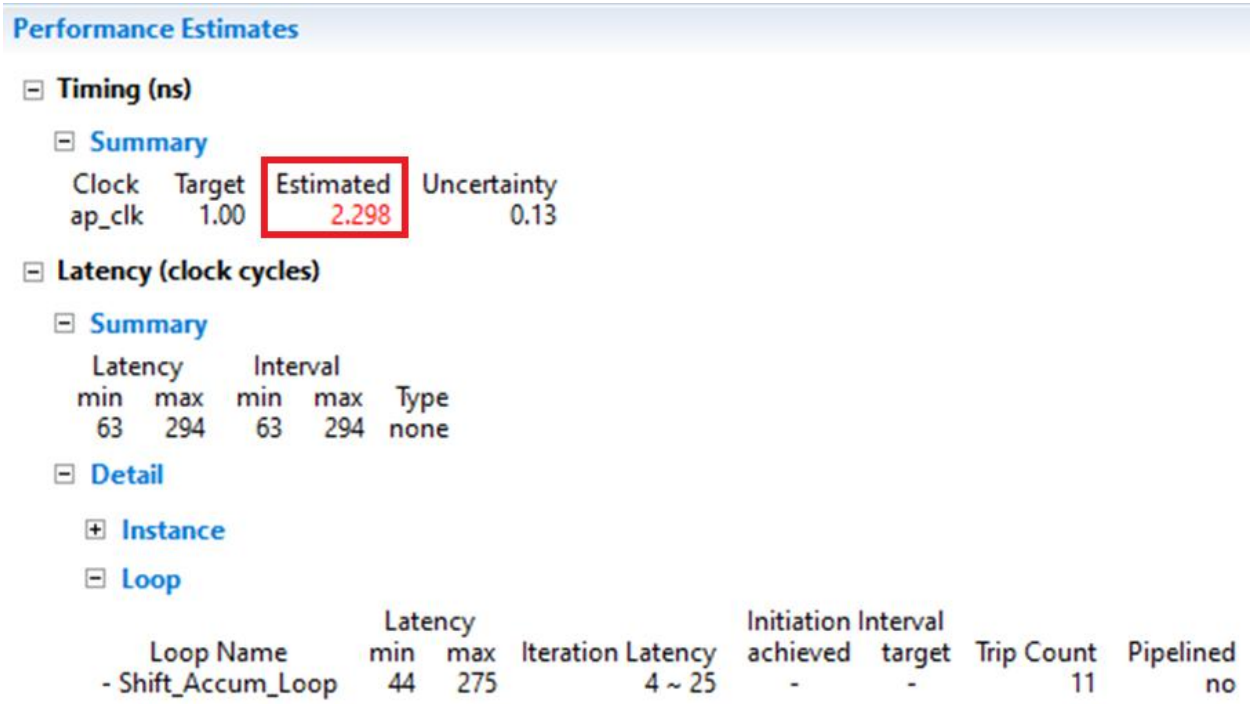
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	26
FIFO	-	-	-	-
Instance	-	8	1107	171
Memory	0	-	74	8
Multiplexer	-	-	-	275
Register	-	-	289	-
Total	0	8	1470	480
Available	40	40	16000	8000
Utilization (%)	0	20	9	6

Рисунок. П.26.

Установим clock_period 1.

Отчет синтеза представлен на рисунке П.27.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	1.00	2.298	0.13

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
63	294	63	294	none

Detail

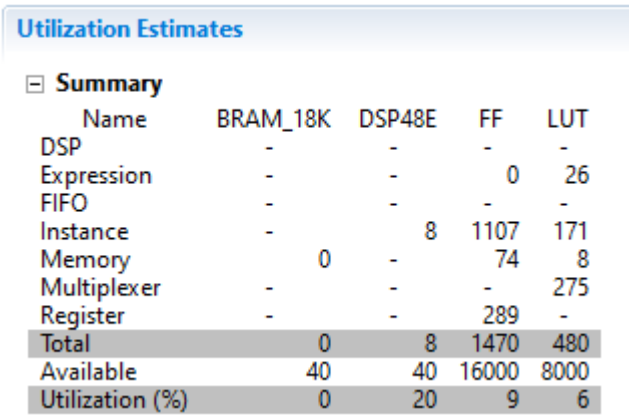
Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Shift_Accum_Loop	44	275	4 ~ 25	-	-	11	no

Рисунок. П.27.

Оценка использованных ресурсов представлена на рис. П.28.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	26
FIFO	-	-	-	-
Instance	-	8	1107	171
Memory	0	-	74	8
Multiplexer	-	-	-	275
Register	-	-	289	-
Total	0	8	1470	480
Available	40	40	16000	8000
Utilization (%)	0	20	9	6

Рисунок. П.28.

Вывод по заданию 2. Последовательное уменьшение **clock_period с 10 до 1 н.с.** показало, что с уменьшением времени выполнения одного такта увеличивается количество используемых для синтеза компонентов. Количество DSP блоков увеличилось в 6 до 8, количество флип-флопов возросло с 287 FF до 1470 FF, количество таблиц входов-выходов (Look-Up Table) с 235 до 480. Из всего интервала изменения времени одного такта (10..1) самым успешным оказалось значение 8,47 нс. При этом время задержки интервала составило 381,15 нс. Также был опробован синтез при значении времени такта - 16 нс, в этом случае итерация цикла помещалась в два такта и задержка интервала составила 370,139 нс.

5.1. Оптимизация кода путем вынесения условных операторов за пределы цикла

Оператор if / else внутри цикла for неэффективен. Для каждой управляющей структуры в коде инструмент Vivado HLS создает логическое аппаратное средство, которое проверяет, выполняется ли условие, которое выполняется на каждой итерации цикла. Кроме того, эта условная структура ограничивает выполнение операторов в ветвях if или else; эти операторы могут быть выполнены только после разрешения оператора if. Оператор if проверяет, когда x == 0, что происходит только на последней итерации.

Следовательно, операторы if/else могут быть удалены из цикла. Наконец, мы должны изменить границы цикла от выполнения "0-й" итерации. Это преобразование показано на рисунке 3. Оно показывает только те изменения, которые необходимы для цикла for. Конечным результатом является гораздо более компактная реализация, которая готова для дальнейшей оптимизации цикла, например, развертывания и конвейеризации. Мы обсудим эти оптимизации позже.

```
Shift_Accum_Loop:
for (i = N - 1; i > 0; i--) {
    shift_reg[i] = shift_reg[i - 1];
    acc += shift_reg[i] * c[i];
}

acc += x * c[0];
shift_reg[0] = x;
```

Рисунок 3. Удаление условного оператора из цикла for создает более эффективную аппаратную реализацию.

Задание 3. Сравните реализации до и после удаления условия if / else, выполненного с помощью подъема цикла. Какая разница в производительности? Как меняется количество ресурсов?

5.1. Программная реализация оптимизации кода путем вынесения условных операторов за пределы цикла

Задание 3. Сравните реализации до и после удаления условия if / else, выполненного с помощью подъема цикла. Какая разница в производительности? Как меняется количество ресурсов?

Решение 3. Рассмотрим код Листинга 1 и 2 в решении 1,2, заметим. что ветвление if / else на каждой итерации цикла потребляет дополнительные ресурсы. А по факту используется один раз. Поэтому логично его вынести после цикла.

Код программы с вынесенным из цикла условных операторов if / else представлен на Листинге П.3.

```
#define N 11
typedef int coef_t;
typedef int data_t;
```

```
typedef int acc_t;

void fir3(data_t *y, data_t x) {

   coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
   static
          data_t shift_reg[N];
          acc_t acc;
          int i;
          acc = 0;
Shift_Accum_Loop:
   for (i = N - 1; i > 0; i--) {
      shift_reg[i] = shift_reg[i - 1];
      acc += shift_reg[i] * c[i];
   }
   acc += x * c[0];
   shift_reg[0] = x;
   *y = acc;
```

Листинг П.3.

Отчет синтеза представлен на рисунке П.29.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
42	42	42	42	none

Рисунок. П.29.

Оценка использованных ресурсов представлена на рис. П.30.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	6	0	142
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	74	8
Multiplexer	-	-	-	92
Register	-	-	156	-
Total	0	6	230	242
Available	40	40	16000	8000
Utilization (%)	0	15	1	3

Рисунок. П.30.

Вывод по заданию 3. Вынос условных операторов за пределы цикла привел к следующим результатам. В сравнении с решением 1 минимальный Latency увеличился с 23 до 42, а максимальный сократился с 45 до 42, все остальные показатели производительности

остались такими же. Что касается использованных ресурсов, то количество регистров сократилось с 213 до 156, количество мультиплексеров снизилось со 120 до 92, Expression увеличилось со 107 до 142, таким образом общая величина количество таблиц входов-выходов (Look-Up Table) возросла с 235FF до 242 FF.

6.1. Разложение циклов

Мы делаем две фундаментальные операции в цикле for. Первая часть перемещает данные через массив смещения регистра. Вторая часть выполняет операции умножения и накопления для вычисления выходной выборки. Деление цикла берет эти две операции и реализует каждую из них в своем собственном цикле. Хотя это может показаться не очень хорошей идеей, это позволяет нам выполнять оптимизацию отдельно для каждого цикла. Это может быть выгодно, особенно в тех случаях, когда результирующие оптимизации в циклах разделения отличаются. Код на рисунке 4 показывает результат ручной оптимизации разложения циклов. Фрагмент кода разбивает цикл из рисунка 1 на два цикла. Обратите внимание на названия меток для двух петель. Первый - это TDL, а второй - MAC. Линия задержки с ответвлением (TDL- Tapped delay line) является общим термином DSP для операции FIFO; MAC - сокращение от «умножить-сложить».

```
TDL:
for (i = N − 1; i > 0; i−−) {
    shift_reg[i] = shift_reg[i − 1];
}
shift_reg[0] = x;

acc = 0;
MAC:
for (i = N − 1; i >= 0; i−−) {
    acc += shift_reg[i] ∗ c[i];
}
```

Рисунок 4: Фрагмент кода, соответствующий разбиению цикла for на два отдельных цикла.

Задание 4. Сравните реализации до и после деления цикла. Какая разница в производительности? Как меняется количество ресурсов?

Само деление цикла часто не обеспечивает более эффективную аппаратную реализацию. Тем не менее, он позволяет оптимизировать каждый цикл независимо, друг от друга, что может привести к лучшим результатам, чем оптимизация одного исходного цикла. Обратное также верно; объединение двух (или более) циклов for в один цикл for может дать наилучшие результаты. Это сильно зависит от приложения, что верно для большинства оптимизаций. В общем, нет единого практического правила по оптимизации вашего кода. Таким образом, важно иметь в своем распоряжении много приемов и, что еще лучше, иметь глубокое понимание того, как работает оптимизация. Только тогда вы сможете создать лучшую аппаратную реализацию.

6.2. Программная реализация разложение циклов

Задание 4. Сравните реализации до и после разложения цикла на два цикла на примере программы Листинга П.1. Какая разница в производительности? Как меняется количество ресурсов?

Решение 4. Разобьем цикл for в Листинге П.1 на два цикла, в итоге получим следующий код программы, представленный на Листинге П.4.

```
#define N 11
typedef int coef_t;
typedef int data_t;
typedef int acc_t;
void fir(data_t *y, data_t x) {
	coef_t c[N] = {
		53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53
	};
	static data_t shift_reg[N];
	acc_t acc;
	int i;

	acc = 0;
TDL:
	for(i = N - 1; i > 0; i--) {
		shift_reg[i] = shift_reg[i - 1];
	}
	shift_reg[0] = x;
	acc = 0;
MAC:
	for ( i = N - 1; i >= 0; i--) {
		acc += shift_reg[i] * c[i];
	}
	*y = acc;
}
```

Листинг П.4.

Отчет синтеза представлен на рисунке П.31.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
66	66	66	66	none

Detail

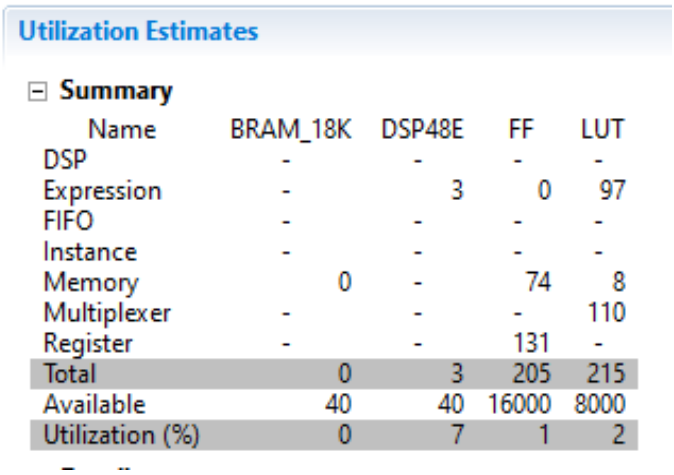
Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- TDL	20	20	2	-	-	10	no
- MAC	44	44	4	-	-	11	no

Рисунок. П.31.

Оценка использованных ресурсов представлена на рис. П.32.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	97
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	74	8
Multiplexer	-	-	-	110
Register	-	-	131	-
Total	0	3	205	215
Available	40	40	16000	8000
Utilization (%)	0	7	1	2

Рисунок. П.32.

Вывод по заданию 4. Разложение одного цикла на два цикла TDL и MAC привел к следующим результатам: можно наблюдать рост минимальной задержки с 23 до 66 и максимальной с 45 до 66, снижение количества используемых ресурсов: количество блоков DSP сократилось с 6 до 3, регистров с 213 до 131, общее количество флип-флопов сократилось с 287FF до 205 FF, количество Expression сократилось со 107 до 97, а количество мультиплексоров со 120 до 110, таким образом общая величина количество таблиц входов-выходов (Look-Up Table) уменьшилась с 235LUT до 215LUT. Данный результат можно объяснить что сложение\умножение теперь происходит только в одном месте в теле цикла MAC.

7.1. Развёртывание цикла

По умолчанию инструмент Vivado HLS синтезирует циклы for последовательно. Инструмент создает путь данных, который реализует одно выполнение операторов в теле цикла. Путь к данным выполняется последовательно для каждой итерации цикла.

Это создает эффективную архитектуру области; однако, это ограничивает возможность использовать параллелизм, который может присутствовать в итерациях цикла. Развертывание цикла повторяет тело цикла несколько раз (это называется фактором). И это уменьшает количество итераций цикла на тот же коэффициент.

В лучшем случае, когда ни один из операторов в цикле не зависит от каких-либо данных, сгенерированных на предыдущих итерациях, это может существенно увеличить доступный параллелизм и, таким образом, обеспечить архитектуру, которая работает намного быстрее.

```
TDL:
for (i = N − 1; i > 0; i−−) {
    shift_reg[i] = shift_reg[i − 1];
}
shift_reg[0] = x;

acc = 0;
MAC:
for (i = N − 1; i >= 0; i−−) {
    acc += shift_reg[i] ∗ c[i];
}
```

Рисунок 4: Фрагмент кода, соответствующий разбиению цикла for на два отдельных цикла (дублирование рисунка 4)

Первый цикл for (с меткой TDL) на рисунке 4 сдвигает значения вверх через массив shift_reg. Цикл повторяется от наибольшего значения (N-1) до наименьшего значения (i = 1). Развернув этот цикл, мы можем создать путь данных, который выполняет ряд этих операций сдвига параллельно.

На рисунке 5 показан результат развертывания цикла в два раза. Этот код дублирует тело цикла дважды. Каждая итерация цикла теперь выполняет две операции сдвига. Соответственно, мы должны выполнить половину количества итераций. Обратите внимание, что есть дополнительное условие if после цикла for. Это необходимо в случае, когда цикл не имеет четного числа итераций. В этом случае мы должны выполнить последнюю \ половинную итерацию самостоятельно. Код в операторе if выполняет эту последнюю «половинную» итерацию, то есть перемещает данные из shift_reg [0] в shift_reg [1].

Также обратите внимание на эффект развертывания цикла в заголовке цикла for. Операция декремента меняется с i - на i = i-2. Это связано с тем, что мы выполняем в два раза больше «работы» в каждой итерации, поэтому мы должны уменьшить значение на 2 вместо 1.

Задание 5. Минимизируйте количество итераций цикла

Наконец, условие завершения цикла for изменяется с i> 0 на i> 1. Это связано с тем, что мы должны убедиться, что «последняя» итерация может полностью завершиться, не вызывая ошибки. Если последняя итерация цикла for выполняется, когда i = 1, то второй оператор попытается прочитать из shift_reg [-1]. Вместо того, чтобы выполнять эту недопустимую операцию, мы делаем последний сдвиг в операторе if после цикла for.

Задание 6. Напишите код, соответствующий ручному развертыванию этого цикла TDL for с коэффициентом три. Как это меняет тело цикла? Какие изменения необходимы для заголовка цикла? Необходим ли дополнительный код в операторе if после цикла for? Если да, то чем он отличается?

Развертывание цикла может увеличить общую производительность при условии, что у нас есть возможность выполнять некоторые (или все) операторы параллельно. В развернутом коде каждая итерация требует, чтобы мы считали два значения из массива shift_reg; и мы записываем два значения в один и тот же массив. Таким образом, если мы хотим выполнить оба оператора параллельно, мы должны иметь возможность выполнить две операции чтения и две операции записи из массива shift_reg в одном и том же цикле.

Предположим, что мы храним массив shift_reg в одном BRAM, а BRAM имеет два порта чтения и один порт записи. Таким образом, мы можем выполнить две операции чтения за один цикл. Но мы должны упорядочить операции записи по двум последовательным циклам.

Есть способы выполнить эти два оператора в одном цикле. Например, мы могли бы хранить все значения массива shift_reg в отдельных регистрах. Можно читать и записывать в каждый отдельный регистр в каждом цикле. В этом случае мы можем выполнить оба оператора в этом развернутом цикле for за один цикл. Вы можете указать инструменту Vivado HLS поместить все значения в массиве shift_reg в регистры, используя директиву:

#pragma HLS array partition variable=shift reg complete.

Это важная оптимизация, поэтому мы обсудим директиву **array_partition** более подробно позже. Пользователь может указать инструменту Vivado HLS на автоматическое развертывание цикла, используя директиву unroll. Чтобы автоматически выполнить развертывание, выполненное вручную на рисунке 5, мы должны поместить директиву #pragma HLS unroll factor=2 в тело кода, сразу после заголовка цикла for. Хотя мы всегда можем выполнить развертывание цикла вручную, гораздо проще позволить инструменту сделать это за нас. Это облегчает чтение кода; и это приведет к меньшему количеству ошибок кодирования.

```
TDL:
for (i = N − 1; i > 1; i = i − 2) {
    shift_reg[i] = shift_reg[i − 1];
    shift_reg[i − 1] = shift_reg[i − 2];
}
if (i == 1) {
    shift_reg[1] = shift_reg[0];
}
shift_reg[0] = x;
```

Рисунок 5: Ручное развертывание цикла TDL в функции r11.

Теперь рассмотрим второй цикл for (с меткой MAC) на рисунке 4. Этот цикл умножает значение из массива c[ ] на значение из массива shift_array[ ]. На каждой итерации он получает доступ к i-му значению из обоих массивов. И затем он добавляет результат этого умножения в переменную acc.

Каждая итерация этого цикла выполняет одну операцию умножения и одну операцию добавления. Каждая итерация выполняет одну операцию чтения из массива shift_reg [ ] и массива c[]. Результат умножения этих двух значений накапливается в переменную acc.

Операции загрузки и умножения не зависят от всех итераций для цикла. Операция сложения, в зависимости от того, как она реализована, может зависеть от значений предыдущих итераций. Однако можно развернуть этот цикл и удалить эту зависимость.

На рисунке 6 показан код, соответствующий развертыванию цикла MAC for в четыре раза. Первый цикл for - это развернутый цикл. Заголовок цикла for изменяется аналогично тому, как мы развернули цикл TDL. Граница изменяется на i> = 3, и i уменьшается в 4 раза для каждой итерации развернутого цикла.

Несмотря на то, что в исходном виде была циклическая зависимость, развернутый for больше не присутствует в развернутом цикле. Зависимость, переносимая циклом, возникла из-за переменной acc; Так как результат умножения накапливается, записывается в эту переменную при каждой итерации, и мы читаем из этого регистра на каждой итерации (для выполнения промежуточной суммы), он создает зависимость

чтения после записи (RAW) между итерациями. Обратите внимание, что в развернутом цикле нет зависимости от переменной acc из-за того, как это написано. Таким образом, мы можем распараллелить четыре отдельные операции MAC в развернутом цикле for.

```
acc = 0;
MAC:
for (i = N − 1; i >= 3; i −= 4) {
    acc += shift_reg[i] ∗ c[i] + shift_reg[i − 1] ∗ c[i − 1] +
    shift_reg[i − 2] ∗ c[i − 2] + shift_reg[i − 3] ∗ c[i − 3];
}

for (; i >= 0; i−−) {
    acc += shift_reg[i] ∗ c[i];
}
```

Рисунок 6: Ручное развертывание цикла MAC в функции ***_r11*** *с коэффициентом четыре.*

После развернутого цикла for есть дополнительный цикл for. Это необходимо для выполнения любых частичных итераций. Так же, как нам требовался оператор if в TDL, он выполняет любые вычисления на потенциальной последней итерации. Это происходит, когда число итераций в исходном цикле, развернутом для цикла for, не является четным кратным 4.

Еще раз, мы можем сказать инструменту Vivado HLS автоматически развернуть цикл в 4 раза, вставив код #pragma HLS unroll factor=4 в теле цикла MAC.

Задание 7. Проведите эксперимент с помощью директивы #pragma HLS unroll factor=4 Как использование директивы меняет количество ресурсов (FFs, LUTs, BRAMs, DSP48)?

Указав необязательный аргумент skip_exit_check в этой директиве, инструмент Vivado HLS не добавит финальный цикл for для проверки частичных итераций.

Это полезно в случае, когда вы знаете, что цикл никогда не потребует этих конечных частичных итераций. Или, возможно, выполнение последних нескольких итераций не оказывает (существенного) влияния на результаты, и поэтому его можно пропустить. При использовании этой опции инструменту Vivado HLS не нужно создавать этот дополнительный цикл for. Таким образом, получаемое аппаратное обеспечение является более простым и более эффективным.

Цикл for полностью разворачивается, когда не указан аргумент фактора. Это эквивалентно развертыванию по максимальному количеству итераций; в этом случае полное развертывание и развертывание с коэффициентом 11 эквивалентны. В обоих случаях тело цикла повторяется 11 раз. И заголовок цикла не нужен; нет необходимости вести счетчик или проверять, выполняется ли условие выхода из цикла. Чтобы выполнить полное развертывание, границы цикла должны определяться статически, т. е. Инструмент Vivado HLS должен знать количество итераций для цикла for во время компиляции.

Развертывание полного цикла обеспечивает максимальную степень параллелизма за счет создания реализации, требующей значительного количества ресурсов. Таким образом, можно выполнить полное развертывание цикла для \ меньшего числа циклов. Но полное развертывание цикла с большим количеством итераций (например, цикла, который повторяется миллион раз), как правило, невозможно. Часто инструмент Vivado HLS будет работать в течение очень долгого времени (и много раз не сможет завершиться после

нескольких часов синтеза), если полученное в результате развертывание цикла создаст очень большой код.

7.2. Программная реализация развёртывания цикла

Задание 5. Требуется минимизировать количество итераций цикла.
Решение 5. Идея заключается в том, чтобы минимизировать количество итераций цикла и по максимуму выполнять вычисления в одной итерации. То есть вытянуть цикл по возможности. Преобразованный код примет вид представленный на листинге П.5.

```
#define N 11
typedef int coef_t;
typedef int data_t;
typedef int acc_t;
void fir5(data_t *y, data_t x) {
   coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
   static
         data_t shift_reg[N];
         acc_t acc;
         int i;
         acc = 0;
TDL:
   for (i = N - 1; i > 1; i = i - 2) {
         shift_reg[i] = shift_reg[i - 1];
         shift_reg[i - 1] = shift_reg[i - 2];
   }
   if(i==1){
         shift_reg[1] = shift_reg[0];
   }
   acc = 0;
MAC:
   for (i = N - 1; i >= 0; i--) {
      acc += shift_reg[i] * c[i];
   }
  *y = acc;
}
```

Листинг П.5.

Отчет синтеза представлен на рисунке П.33.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
56	56	56	56	none

Detail

Instance

Loop

Рисунок П.33.

Оценка использованных ресурсов представлена на рис. П.34.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	110
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	2	-	10	2
Multiplexer	-	-	-	125
Register	-	-	135	-
Total	2	3	145	237
Available	40	40	16000	8000
Utilization (%)	5	7	~0	2

Рисунок П.34.

Диаграмма расписания просмотра (Schedule viewer) представлена на рисунке П.35.

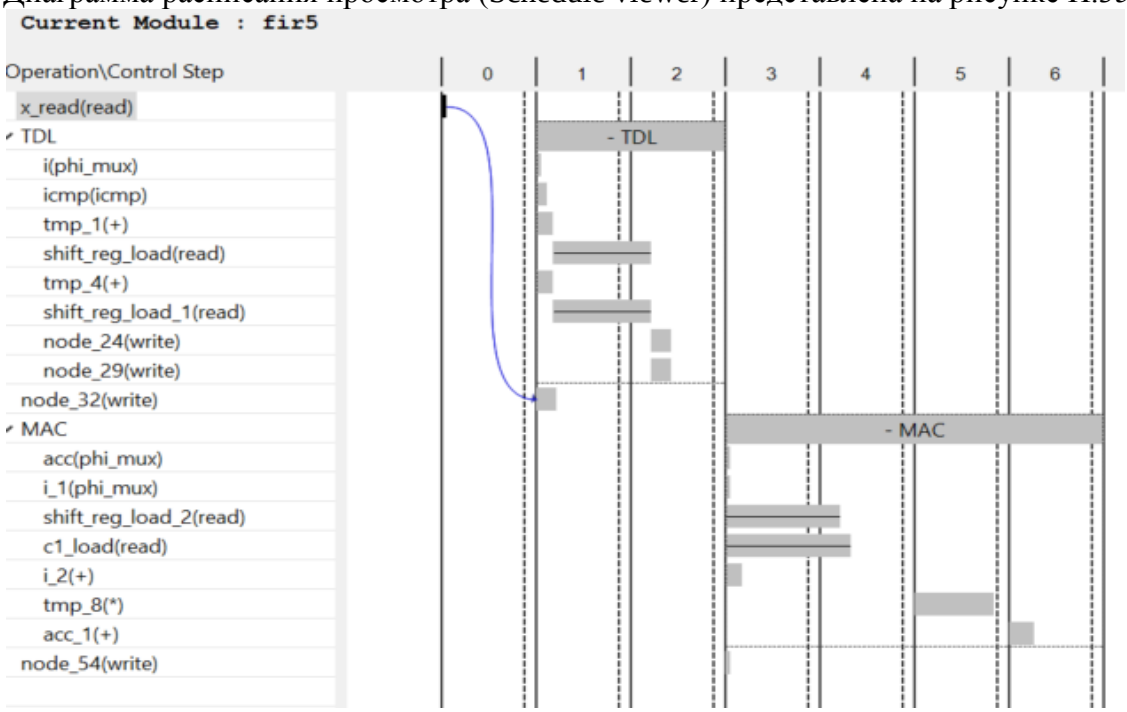


Рисунок П.35.

Вывод по заданию 5. В результате минимизации количества итераций цикла было достигнуто сокращение ресурсов: количество блоков DSP48E сократилось с 6 до 3, памяти с 74 до 10, регистров с 213 до 135, общее количество флип-флопов сократилось с 287FF до 145FF, вместе с тем увеличилось количество BRAM_18K с 0 до 2, возросло количество Expression (LUT) с 107 до 110 и мультиплексеров (LUT) с 120 до 125, таким образом общая величина количество таблиц входов-выходов (Look-Up Table) увеличилась с 235LUT до 237LUT. С точки зрения производительности latency увеличилась с 45 до 56.

Задание 6. Напишите код, соответствующий ручному развертыванию этого цикла TDL for с коэффициентом три. Как это меняет тело цикла? Какие изменения необходимы для заголовка цикла? Необходим ли дополнительный код в операторе if после цикла for? Если да, то чем он отличается?
Решение 6. Увеличим количество присваиваний в одной итерации до трех. Код программы представлен на листинге П.6.

```
#define N 11

typedef int coef_t;
typedef int data_t;
typedef int acc_t;

void fir6(data_t *y, data_t x) {

   coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
   static
         data_t shift_reg[N];
         acc_t acc;
         int i;
         acc = 0;
TDL:
   for (i = N - 1; i > 2; i = i - 3) {
         shift_reg[i] = shift_reg[i - 1];
         shift_reg[i - 1] = shift_reg[i - 2];
      shift_reg[i - 2] = shift_reg[i - 3];
   }

   if(i==1){
         shift_reg[2] = shift_reg[1];
         shift_reg[1] = shift_reg[0];
   }
   shift_reg[0] = x;
MAC:
   for (i = N - 1; i >= 0; i--) {
      acc += shift_reg[i] * c[i];
   }
  *y = acc;
}
```

Листинг П.6.

Отчет синтеза представлен на рисунке П.36.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
57	57	57	57	none

Рисунок П.36.

Оценка использованных ресурсов представлена на рис. П.37.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	123
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	2	-	10	2
Multiplexer	-	-	-	178
Register	-	-	142	-
Total	2	3	152	303
Available	40	40	16000	8000
Utilization (%)	5	7	~0	3

Рисунок П.37.

Диаграмма расписания просмотра (Schedule viewer) представлена на рисунке П.38.

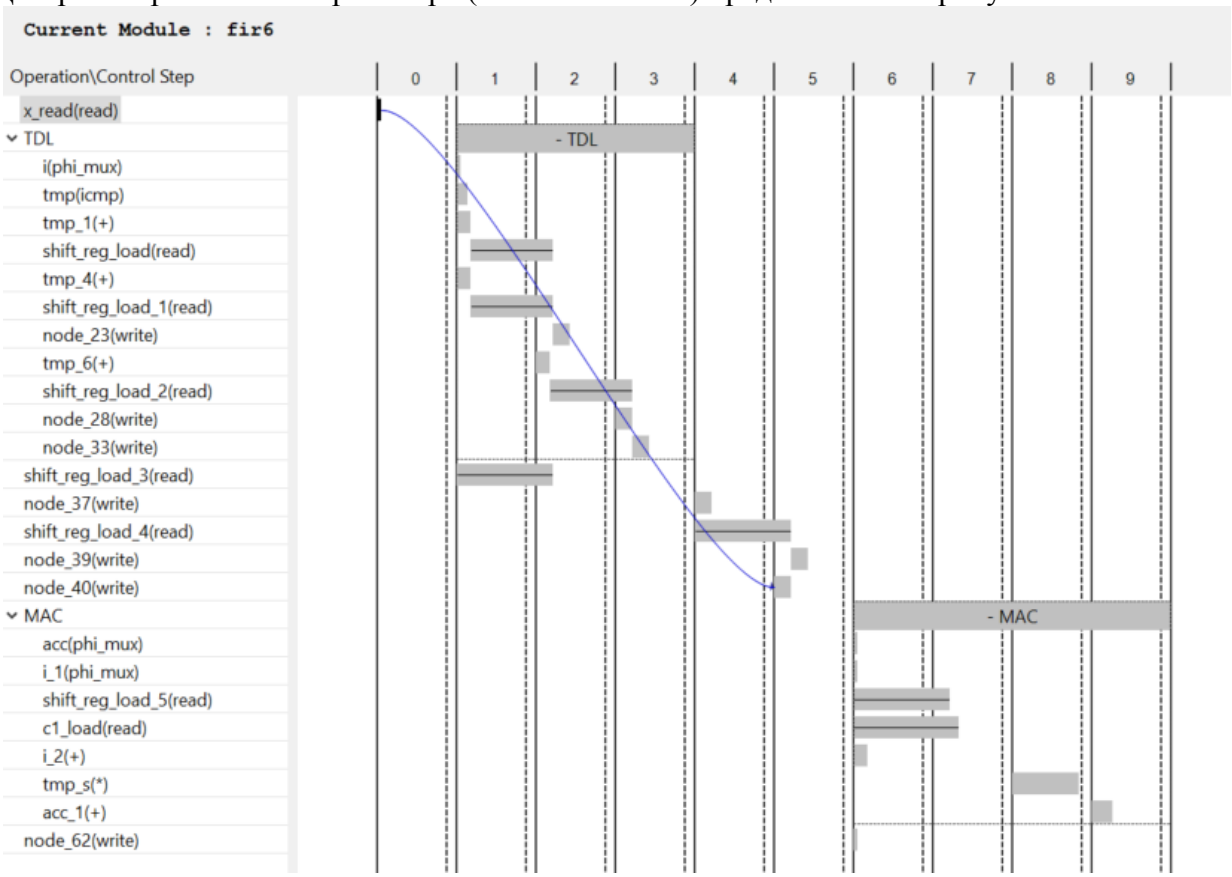


Рисунок П.38.

Вывод по заданию 6. В результате увеличения количества присваиваний в одной итерации до трех в сравнении с решением 5 возросло количество используемых ресурсов:

возросло количество регистров со 135 до 142, общее количество флип-флопов возросло со 145FF до 152FF, возросло количество Expression (LUT) с 110 до 123 и мультиплексеров (LUT) с 125 до 178, таким образом общая величина количество таблиц входов-выходов (Look-Up Table) увеличилась с 237LUT до 303LUT, величина latency увеличилась с 56 до 57, количество блоков памяти BRAM_18K осталось неизменным, но используется уже не так эффективно, поскольку третье чтение происходит во время уже следующего такта. Таким образом данное решение хуже чем первое и четвертое.

Задание 7. Проведите эксперимент с помощью директивы #pragma HLS unroll factor=4
Как использование директивы меняет количество ресурсов (FFs, LUTs, BRAMs, DSP48)?
Решение 7. Проведем эксперимент с помощью директив. Код программы к которому будут подключены директивы представлен на листинге П.7.

```
#define N 11
//#include "ap_int.h"
typedef int coef_t;
typedef int data_t;
typedef int acc_t;
void fir7(data_t *y, data_t x) {
   coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
   static
         data_t shift_reg[N];
         acc_t acc;
         int i;
         acc = 0;
Shift_Accum_Loop:
   for (i = N - 1; i > 0; i--) {
#pragma HLS unroll factor=4
      shift_reg[i] = shift_reg[i - 1];
      acc += shift_reg[i] * c[i];
   }
   acc += x * c[0];
   shift_reg[0] = x;
   *y = acc;
}
```

Листинг П.7.

Отчет синтеза представлен на рисунке П.38.

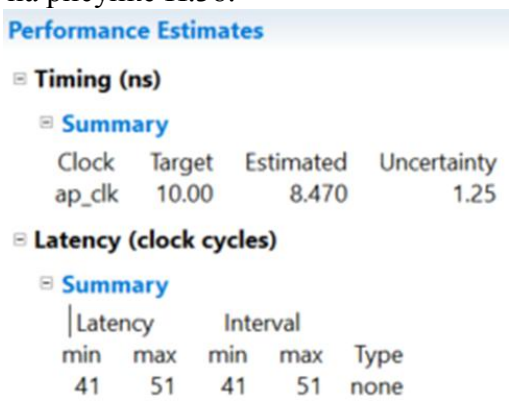


Рисунок П.38.

Оценка использованных ресурсов представлена на рис. П.39.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	15	0	415
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	74	8
Multiplexer	-	-	-	193
Register	-	-	503	-
Total	0	15	577	616
Available	40	40	16000	8000
Utilization (%)	0	37	3	7

Рисунок П.39.

Вывод по заданию 7. В результате использования директивы **#pragma HLS unroll factor=4** в сравнении с решением 1 возросло количество используемых ресурсов: возросло количество регистров со 213FF до 503FF, общее количество флип-флопов возросло со 287FF до 577FF, возросло количество Expression (LUT) с 107 LUT до 415 LUT и мультиплексеров (LUT) с 120 до 193, таким образом общая величина количество таблиц входов-выходов (Look-Up Table) увеличилась с 235LUT до 616LUT, минимальная величина latency увеличилась с 23 до 41, а максимальная с 45 до 51, количество блоков DSP48E увеличилось в 6 до 15. Таким образом данное решение хуже чем все предыдущие, кроме этого следует обратить внимание, что в данном случае у нас нет разбиения на циклы TDL и MAC в результате чего мы получаем совсем не оптимальное решение.

8.1. Контурная конвейерная обработка

По умолчанию инструмент Vivado_HLS синтезирует циклы последовательно. Например, цикл for на рисунке 1 будет выполнять каждую итерацию цикла один за другим. То есть все операторы во второй итерации выполняются только тогда, когда все операторы из первой итерации завершены; то же самое верно для последующих итераций.

Это происходит даже в тех случаях, когда возможно выполнение операторов из итераций параллельно. В других случаях некоторые из операторов можно запустить на более поздней итерации, прежде чем все операторы на предыдущей итерации будут завершены.

Этого не произойдет, если дизайнер специально не заявит, что должен. Это мотивирует идею конвейерной обработки цикла, которая позволяет одновременно выполнять несколько итераций цикла.

Рассмотрим MAC для цикла из рисунка 4. Это выполняет одну операцию умножения с накоплением (MAC) за итерацию. Этот цикл MAC for имеет четыре операции в теле цикла:

- Прочесть c []: загрузить указанные данные из массива C.
- Прочесть shift_reg []: загрузить указанные данные из массива shift_reg.
- *: Умножьте значения из массивов c [] и shift_reg [].
- +: Накопить этот умноженный результат в переменную acc.

Расписание, соответствующее одной итерации цикла MAC for, показано на рисунке 7 а). Каждая операция чтения требует 2 цикла. Это связано с тем, что первый цикл предоставляет адрес в память, а данные из памяти доставляются во время второго цикла. Эти две операции чтения могут выполняться параллельно, поскольку между ними нет никаких зависимостей. Операция * может начаться в Цикле 2; предположим, что для его завершения требуется три цикла, то есть завершен цикл 4. Операция «+» прикована к

цепи для запуска и завершения в течение цикла 4. Все тело цикла MAC for занимает 4 цикла. Существует ряд показателей производительности, связанных с циклом for.

Задержка итерации - это количество циклов, которое требуется для выполнения одной итерации тела цикла. Задержка итерации для этого цикла MAC for составляет 4 цикла.

Задержка цикла for - это количество циклов, необходимое для завершения всего цикла.

Это включает время для вычисления оператора инициализации (например, i = 0), оператора условия (например, i> = 0) и оператора приращения (например, i--).

Предполагая, что эти три оператора заголовка могут быть выполнены параллельно с выполнением тела цикла, инструмент Vivado HLS сообщает о задержке этого цикла MAC for как для 44 циклов. Это количество итераций (11), умноженное на задержку итерации (4 цикла) плюс один дополнительный цикл, чтобы определить, что цикл должен прекратить итерацию. А потом вы вычитаете один.

Возможно, единственная странная вещь здесь - это "вычитание 1" ". Мы вернемся к этому через секунду. Но сначала, есть один дополнительный цикл, который требуется в начале следующей итерации, который проверяет, удовлетворен ли оператор условия (это не так), а затем выходит из цикла. Теперь "вычитание 1": Vivado HLS определяет задержку как цикл, в котором выходные данные готовы. В этом случае окончательные данные готовы в течение цикла 43. Они будут записаны в регистр в конце цикла 43 и, соответственно, в начале цикла 44.

Другой способ думать об этом состоит в том, что задержка равна максимальному числу регистров между входными данными и выходными данными.

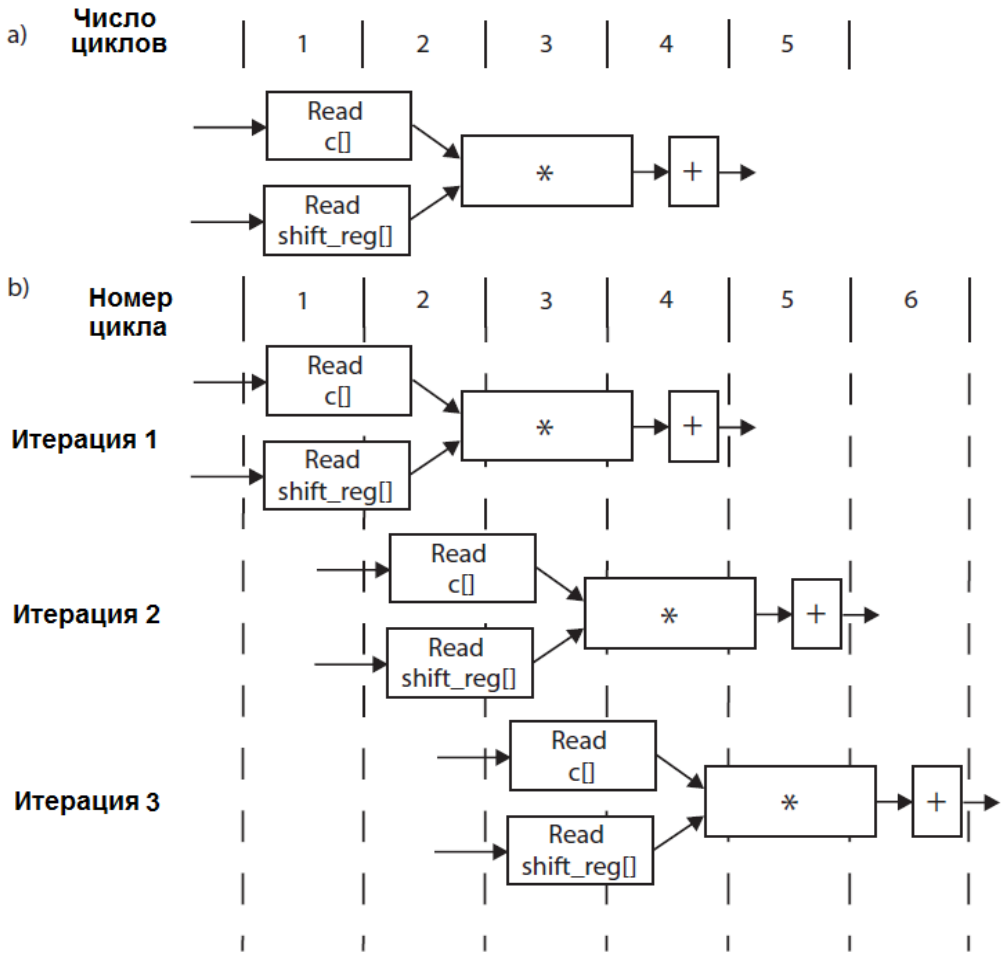


Рисунок 7: Часть a) показывает расписание для тела цикла MAC for. В части b) показано расписание трех итераций конвейерной версии цикла MAC for.

Циклическая конвейеризация - это оптимизация, которая перекрывает несколько итераций цикла for. На рисунке 7 б) приведен пример конвейерной передачи для цикла

MAC for На рисунке 7 показаны три итерации for, которые выполняются одновременно. Первая итерация эквивалентна неконвейерной версии, как показано на рисунке 7 а). Разница - это время начала последующих итераций. В не конвейерной версии вторая итерация начинается после завершения первой итерации, то есть в цикле 5.

Однако конвейерная версия может начать последующую итерацию до завершения предыдущих итераций. На рисунке итерация 2 начинается с цикла 2, а итерация 3 начинается с цикла 3. Остальные итерации начинаются каждый последующий цикл.

Таким образом, последняя итерация, Итерация 11, начнется в Цикле 11 и завершится в Цикле 14. Таким образом, задержка цикла равна 14. Интервал инициирования цикла (II) является еще одним важным показателем производительности. Он определяется как количество тактов до начала следующей итерации цикла.

В нашем примере цикл II равен 1, что означает, что мы начинаем новую итерацию цикла каждый цикл. Это графически изображено на рисунке 7 б). II может быть явно установлен с помощью директивы. Например, директива #pragma HLS pipeline II=2 информирует инструмент Vivado HLS о попытке установить II = 2.

Обратите внимание, что это не всегда возможно из-за ограничений ресурсов и / или зависимостей в коде. В выходных отчетах будет точно указано, чего удалось достичь инструменту Vivado R HLS.

Задание 8. Явно установите интервал инициирования цикла, начиная с 1. Как увеличение II влияет на задержку цикла? Каковы тенденции? В какой-то момент установка значения II на большее значение не имеет смысла. Что это за значение в этом примере? Как вы описываете это значение для общего цикла for?

Любой цикл for может быть конвейерным, поэтому давайте теперь рассмотрим цикл TDL for. Этот цикл for аналогичен заголовку цикла MAC for. Тело цикла выполняет поэлементное смещение данных через массив, как описано в разделе 3. Есть две операции: одна Чтение и одна Запись в массив shift_reg. Задержка итерации этого цикла составляет 2 цикла. Операция чтения занимает два цикла, а операция записи выполняется в конце цикла 2. Задержка цикла for для этого неконвейерного цикла составляет 20 циклов. Мы можем конвейеризовать этот цикл, вставив директиву #pragma HLS pipeline II=1 после заголовка цикла.

Результатом синтеза является цикл инициализации цикла, равный 1 циклу. Это означает, что мы можем начать итерацию цикла каждый цикл. Слегка изменив пример, мы можем продемонстрировать сценарий, когда ограничения ресурсов не позволяют инструменту Vivado HLS достичь значения II = 1. Для этого мы явно устанавливаем тип памяти для массива shift_reg. Не указав ресурс, мы оставляем его на усмотрение инструмента Vivado_HLS. Но мы можем указать память, используя директиву, например, директива #pragma HLS resource variable=shift reg core=RAM 1P заставляет инструмент Vivado HLS использовать ОЗУ с одним портом.

При использовании этой директивы в сочетании с целью конвейерной обработки цикла инструменту Vivado HLS не удастся выполнить конвейерную петлю с II = 1. Это связано с тем, что конвейерная версия этого кода требует выполнения операций чтения и записи в одном и том же цикле. Это невозможно при использовании ОЗУ с одним портом. Это видно на рисунке 8 б). Рассматривая цикл 2, мы требуем операцию записи в массив shift_reg в итерации 1 и операцию чтения в тот же массив в итерации 2. Мы можем изменить директиву, чтобы предоставить HLS большую свободу планирования, удалив явный запрос для II = 1, например #pragma HLS pipeline. В этом случае HLS автоматически увеличит интервал инициации, пока не найдет выполнимое расписание.

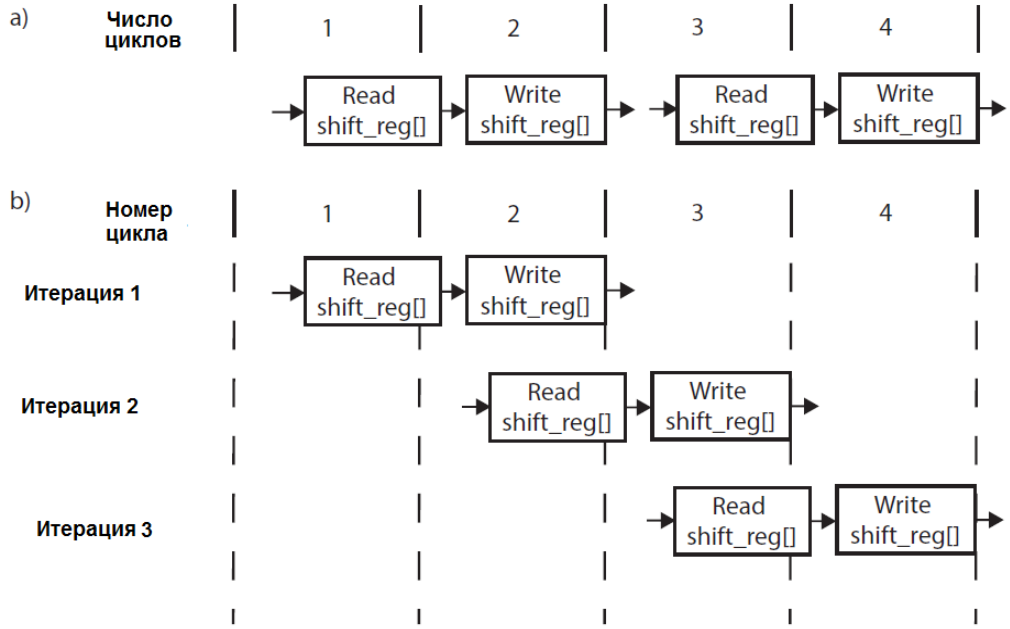


Рисунок 8: Часть a) показывает расписание для двух итераций тела цикла TDL for. В части b) показано расписание трех итераций конвейерной версии цикла TDL for с II = 1.

8.2. Программная реализация конвейерной обработки

Задание 8. Явно установите интервал инициирования цикла, начиная с 1. Как увеличение II влияет на задержку цикла? Каковы тенденции? В какой-то момент установка значения II на большее значение не имеет смысла. Что это за значение в этом примере? Как вы описываете это значение для общего цикла for?

Решение 8. Рассмотрим оптимизацию через распараллеливание независимых операций с помощью директивы pipeline для программы, представленной выше на Листинге П.4.

1) Воспользуемся этой директивой для цикла MAC со значением iteration interval = 1. Результаты синтеза представлены на рисунке П.40.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
36	36	36	36	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- TDL	20	20	2	-	-	10	no
- MAC	13	13	4	1	1	11	yes

Рисунок П.40.

Отчет об использовании ресурсов представлен на рисунке П.41.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	113
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	74	8
Multiplexer	-	-	-	120
Register	0	-	193	32
Total	0	3	267	273
Available	40	40	16000	8000
Utilization (%)	0	7	1	3

Рисунок П.41.

В результате получаем, что задержка выполнения цикла MAC снизилась с 44 до 13.

2) Воспользуемся этой директивой для цикла TDL со значением iteration interval = 1.

Результаты синтеза представлены на рисунке П.42.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
27	27	27	27	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- TDL	10	10	2	1	1	10	yes
- MAC	13	13	4	1	1	11	yes

Рисунок П.42.

Отчет об использовании ресурсов представлен на рисунке П.43.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	3	0	129
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	2	-	10	2
Multiplexer	-	-	-	132
Register	0	-	196	32
Total	2	3	206	295
Available	40	40	16000	8000
Utilization (%)	5	7	1	3

Рисунок П.43.

Вывод по заданию 8. В результате использования директивы pipeline для обоих циклов (MAC и TDL) в сравнении с решением 4 удалось сократить время выполнения циклов, и тем самым уменьшить интервальную задержку всей функции с 66 до 27 тактов. Величину latency TDL удалось сократить с 20 до 10, а latency MAC с 44 до 13. Вместе с тем возросло количество используемых ресурсов: увеличилось количество регистров с 131FF до 196FF, но количество памяти сократилось с 74 FF до 10FF, общее количество флип-флопов возросло совсем незначительно со 205FF до 206FF, количество мультиплексеров возросло с 110LUT до 132 LUT, регистров с 0LUT до 32 LUT, количество памяти незначительно сократилось с 8 LUT до 2 LUT, таким образом общая величина количество таблиц входов-выходов (Look-Up Table) увеличилась с 215LUT до 295LUT.

9.1. Оптимизация битовой полосы

Язык C предоставляет множество различных типов данных для описания различных типов поведения. До этого момента мы фокусировались на типе int, который Vivado HLS рассматривает как 32-разрядное целое число со знаком. Язык C также предоставляет типы данных с плавающей точкой, такие как float и double, и целочисленные типы данных, такие как char, short, long и long long. Целочисленные типы данных могут быть без знака. Все эти типы данных имеют размер, который является степенью 2.

Фактическое количество битов для этих типов данных C может варьироваться в зависимости от архитектуры процессора. Например, int может быть 16 бит на микроконтроллере и 32 бита на процессоре общего назначения. Стандарт C диктует минимальную битовую ширину (например, int составляет по меньшей мере 16 бит) и отношения между типами (например, long не меньше int, который не меньше short).

Стандарт языка C99 устранил эту неоднозначность с такими типами, как int8_t, int16_t, int32_t и int64_t. Основные преимущества использования этих различных типов данных в программном обеспечении зависят от объема хранилища, необходимого для этого типа данных. Для больших массивов использование 8-битных значений вместо 16-битных может сократить использование памяти в два раза.

Недостатком является то, что диапазон значений, которые вы можете представить, уменьшается. 8-битное значение со знаком допускает числа в диапазоне [-128,127], в то время как 16-битный тип данных со знаком имеет диапазон [-32,768, 32,767]. Меньшие операции могут также потребовать меньше тактов для выполнения или могут позволить параллельному выполнению большего количества команд.

Те же преимущества видны в реализации FPGA, но они еще более выражены. Поскольку инструмент Vivado HLS создает собственный путь к данным, он создает реализацию, соответствующую указанным типам данных. Например, оператор a = b*c будет иметь различную задержку и использование ресурсов в зависимости от типа данных. Если все переменные имеют ширину 32 бита, то необходимо выполнить больше примитивных логических операций, чем если бы переменные имели ширину всего 8 бит.

В результате необходимо использовать больше ресурсов ПЛИС (или более сложных ресурсов). Кроме того, более сложная логика обычно требует более конвейерной передачи для достижения той же частоты. Для 32-разрядного умножения может потребоваться 5 внутренних регистров для соответствия той же частоте, которую 8-разрядное умножение может достичь только с одним внутренним регистром. В результате задержка операции будет больше (5 циклов вместо 1 цикла), и HLS должен учитывать это.

Задание 9. Создайте простой дизайн, который реализует код a = b * c. Измените тип данных переменных на char, short, int, long и long long. Сколько циклов занимает операция

умножения в каждом случае? Сколько ресурсов используется для разных типов данных?

Во многих случаях для реализации оптимизированного оборудования необходимо обрабатывать данные, где битовая пропускная способность не является степенью двойки. Например, аналого-цифровые преобразователи часто выводят результаты в 10 битах, 12 битах или 14 битах. Мы могли бы сопоставить их с 16-битными значениями, но это, вероятно, снизит производительность и увеличит использование ресурсов. Для более точного описания таких значений Vivado HLS предоставляет типы данных произвольной точности, которые допускают типы данных со знаком и без знака любой битовой ширины.

Существует два отдельных класса для неподписанных и подписанных типов данных:

Без подписи: ap_uint <width>

С Подписью: ap_int <width>

где переменная width - это целое число от 1 до 1024 (1024 - максимальное значение по умолчанию, и оно может быть изменено при необходимости. См. Руководства пользователя Vivado HLS для получения дополнительной информации о том, как это сделать). Например, ap_int<8> является 8-битным значением со знаком (аналогично char), а ap_uint<32> является 32-битным значением без знака (так же, как unsigned int). Это обеспечивает более мощный тип данных, поскольку он может использовать любую битовую пропускную способность, например, ap_uint<4> или ap_int<537>. Чтобы использовать эти типы данных, вы должны использовать C ++ и включить файл ap_int.h, то есть добавить код #include "ap_int.h" в ваш проект и использовать имя файла, оканчивающееся на `.cpp ' (Аналогичные возможности также доступны в C, используя файл ap_cint.h).

Рассмотрим массив коэффициентов c [] из кода fir filter (ФИЛЬТРЫ КОНЕЧНОГО ИМПУЛЬСНОГО ОТВЕТА) на рисунке 1.

```
#define N 11
#include "ap_int.h"

typedef int coef_t;
typedef int data_t;
typedef int acc_t;

void fir(data_t *y, data_t x) {
    coef_t c[N] = {
        53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53
    };
    static
     data_t shift_reg[N];
    acc_t acc;
    int i;

    acc = 0;
    Shift_Accum_Loop:
    for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```

Рисунок 1.Функционально правильная, но крайне неоптимизированная реализация 11-отводного КИХ-фильтра (дублирование).

Дублирование рисунка 1. Функционально правильная, но крайне неоптимизированная реализация фильтра с конечной импульсной характеристикой в 11 отводов

Перепечатано здесь для вашего удобства: coef_t c [N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};. Тип данных coef_t определяется как int, что означает, что у нас есть 32 бита точности. Это не нужно для этих констант, так как они варьируются от -91 до 500. Таким образом, мы могли бы использовать для этого меньший тип данных. Нам понадобится подписанный тип данных, поскольку у нас есть положительные и отрицательные значения. И максимальное абсолютное значение для любой из этих 11 записей составляет 500, что требует $\lceil \log_2 500 \rceil = 9 \text{bits}$

Поскольку нам нужны отрицательные числа, мы добавляем дополнительный бит. Таким образом, coef_t может быть объявлен как ap_int <10>.

Задание 10. Какой тип данных подходит для переменной i в функции fir (см. рисунок 1)?

Мы также можем более точно определить типы данных для других переменных в функции fir, например, acc и shift_reg. Сначала рассмотрим массив shift_reg. Здесь хранятся последние 11 значений входной переменной x. Итак, мы знаем, что значения shift_reg могут безопасно иметь тот же тип данных, что и x. Под «безопасным» мы подразумеваем, что не будет потери точности, т. е. Если бы shift_reg имел тип данных с

меньшей битовой шириной, то некоторые значимые биты x нужно было бы исключить, чтобы они соответствовали значению в shift_reg. Например, если x был определен как 16 бит (ap uint <16>), a shift_reg был определен как 12 бит (ap_uint <12>), то мы бы обрезали 4 наиболее значимых бита x, когда мы сохранили его в shift_reg. Определение подходящего типа данных для acc является более сложной задачей. Переменная acc хранит сумму умножения и накопления по shift_reg и массиву коэффициентов c [], то есть выходному значению фильтра. Если мы хотим быть в безопасности, мы вычисляем максимально возможное значение, которое может быть сохранено в acc, и устанавливаем битовую пропускную способность как таковую.

Чтобы достичь этого, мы должны понимать, как увеличивается пропускная способность при выполнении арифметических операций. Рассмотрим операцию a = b + c, где ap_uint <10> b и ap_uint <10> c. Какой тип данных для переменной a? Здесь мы можем выполнить анализ наихудшего случая и предположить, что a и b являются наибольшим возможным значением 210 = 1024. Добавление их вместе приводит к a = 2024, который может быть представлен как 11-битовое число без знака, то есть ap_uint <11>. В общем случае нам понадобится на один бит больше, чем наибольшая битовая ширина добавляемых двух чисел. То есть, когда ap_uint <x> b и ap_uint <y> c, тип данных для a равен ap_uint <z>, где z = max (x; y) + 1. Это также верно при добавлении целых чисел со знаком.

Это решает одну часть вопроса о назначении соответствующего типа данных для acc, но мы также должны иметь дело с операцией умножения. Используя ту же терминологию, мы хотим определить значение битовой ширины z с учетом битовой ширины x и y (то есть ap_int <z> a, ap_int <x> b, ap int <y> c) для операции a = b * c. , Пока мы не будем вдаваться в подробности, формула имеет вид z = x + y.

Задание 11. Учитывая эти две формулы, определите битовую ширину acc так, чтобы это было безопасно.

В конечном итоге мы сохраняем acc в переменной y, которая является выходным портом функции. Следовательно, если битовая ширина acc больше, чем битовая ширина c, значение в acc будет усечено для сохранения в y. Таким образом, важно ли гарантировать, что битовая ширина acc достаточно велика, чтобы он мог обрабатывать полную точность операций умножения и накопления?

Ответ на вопрос заключается в толерантности заявки. Во многих приложениях для обработки цифровых сигналов сами данные зашумлены, а это означает, что младшие разряды могут не иметь никакого значения. Кроме того, в приложениях обработки сигналов мы часто выполняем численные аппроксимации при обработке данных, которые могут вносить дополнительную ошибку.

Таким образом, может быть не важно, чтобы мы обеспечивали достаточную точность переменной acc, чтобы получить полностью точный результат. С другой стороны, может быть желательно сохранить больше битов в накоплении и затем снова округлить окончательный ответ, чтобы уменьшить общую ошибку округления при вычислении. В других приложениях, таких как научные вычисления, часто требуется больший динамический диапазон, что может привести к использованию чисел с плавающей точкой вместо арифметики с целыми или с фиксированными точками. Так каков правильный ответ? В конечном итоге это зависит от терпимости дизайнера приложений.

9.2. Программная реализация оптимизации битовой полосы

Задание 9. Создайте простой дизайн, который реализует код a = b * c. Измените тип данных переменных на char, short, int, long и long long. Сколько циклов занимает операция умножения в каждом случае? Сколько ресурсов используется для разных типов данных?

Решение. Рассмотрим в данном пункте разные типы данных. Проведем эксперименты с умножением, заменяя в typedef-ах тип: int, char, short, long, long long.

Код программы представлен на Листинге П.8.

```
typedef int data_t;
void type9(data_t *y, data_t x) {
   data_t c = 10;
   *y = x * c;
```

Листинг П.8.

1) Тип INT.

Отчет синтеза по производительности и использованию ресурсов для операции над типом INT представлены на рисунке П.44.

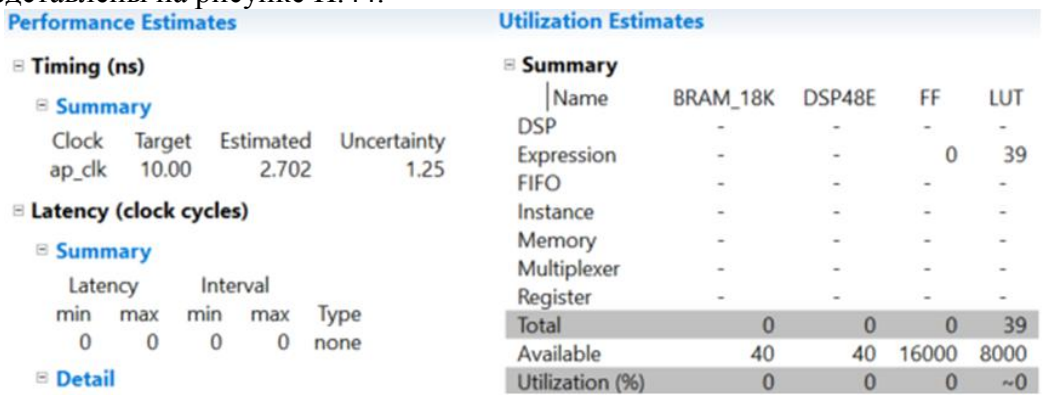


Рисунок П.44.

Диаграмма расписания просмотра (Schedule viewer) представлена на рисунке П.45.

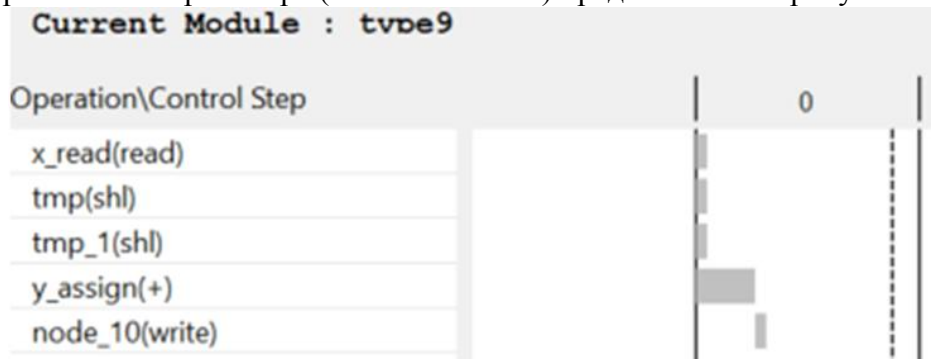


Рисунок П.45.

2) Тип SHORT.

Отчет синтеза по производительности и использованию ресурсов для операции над типом SHORT представлены на рисунке П.46.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	2.146	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
0	0	0	0	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	23
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	23
Available	40	40	16000	8000
Utilization (%)	0	0	0	~0

Рисунок П.46.

Диаграмма расписания просмотра (Schedule viewer) представлена на рисунке П.47.

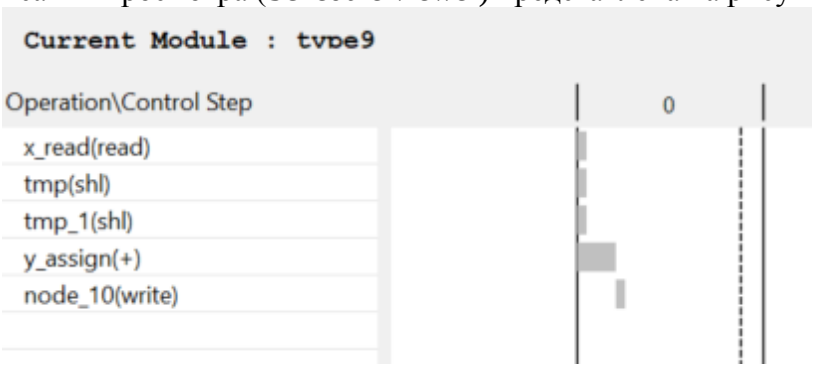


Рисунок П.47.

3) Тип CHAR.

Отчет синтеза по производительности и использованию ресурсов для операции над типом CHAR представлены на рисунке П.48.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	2.115	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
0	0	0	0	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	15
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	15
Available	40	40	16000	8000
Utilization (%)	0	0	0	~0

Рисунок П.48.

Диаграмма расписания просмотра (Schedule viewer) представлена на рисунке П.49.

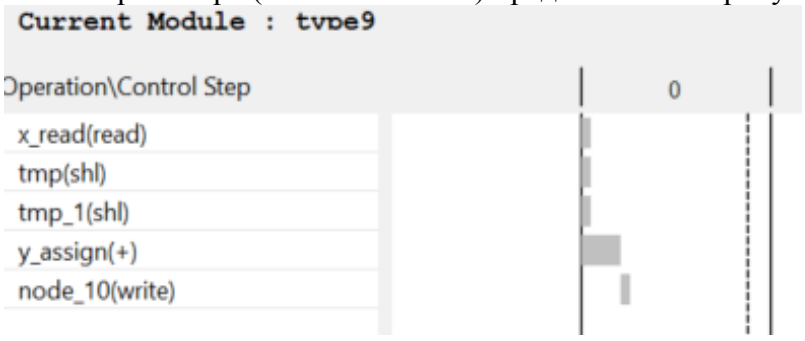
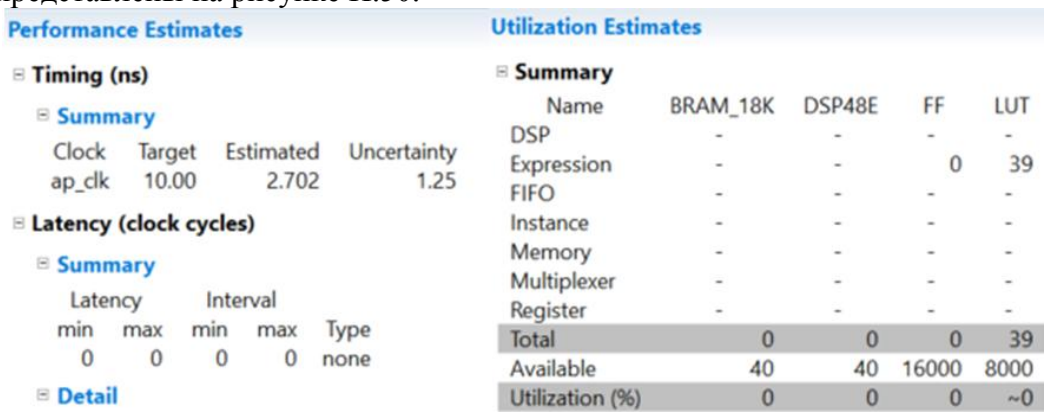


Рисунок П.49.

3) Тип LONG.

Отчет синтеза по производительности и использованию ресурсов для операции над типом LONG представлены на рисунке П.50.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	2.702	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
0	0	0	0	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	39
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	39
Available	40	40	16000	8000
Utilization (%)	0	0	0	~0

Рисунок П.50.

Диаграмма расписания просмотра (Schedule viewer) представлена на рисунке П.51.

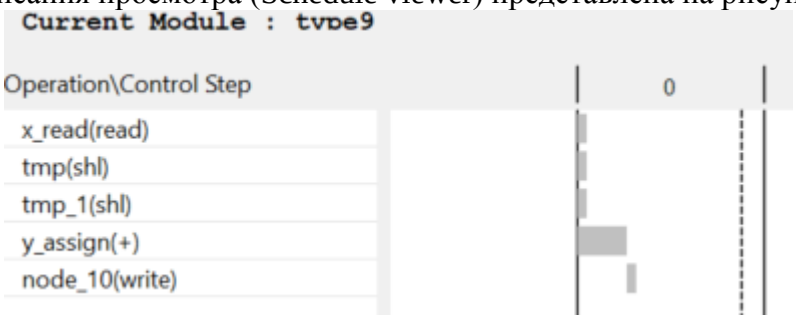
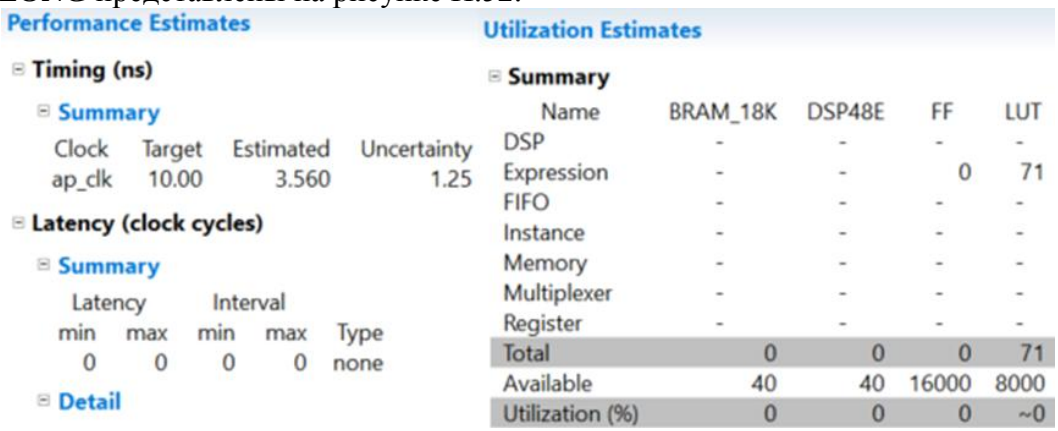


Рисунок П.51.

3) Тип LONG LONG.

Отчет синтеза по производительности и использованию ресурсов для операции над типом LONG LONG представлены на рисунке П.52.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	3.560	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
0	0	0	0	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	71
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	0	71
Available	40	40	16000	8000
Utilization (%)	0	0	0	~0

Рисунок П.52.

Диаграмма расписания просмотра (Schedule viewer) представлена на рисунке П.53.

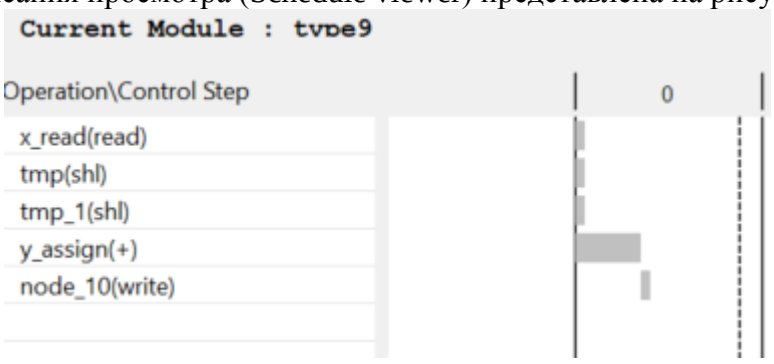


Рисунок П.53.

Вывод по заданию 9. В результате анализа операций с разными типами данных и количества бит в каждом из типов были получены следующие значения используемых для этого ресурсов. Поученные значения отличных друг от друга используемых для рассматриваемых типов сведены в Таблицу П.1.

Таблицу П.1. Анализ используемых ресурсов

№	Характеристики	INT	SHORT	CHAR	LONG	LONG LONG
1	Estimated	2.702	2.146	2.115	2.702	3.560
2	LUT	39	23	15	39	71

Таким образом, можно заключить, что количество бит в каждом из рассмотренных типов данных пропорционально количеству логических элементов и чем больше байт занимает тип данных, тем дольше latency.

Задание 10. Какой тип данных подходит для переменной i в функции fir (см. рисунок 1)?
Решение 10. Поиграем с разрядностью используемых типов, для этого поменяем расширение файла на .cpp и подключим библиотеку "ap int.h".
Для базовой имплементации фильтра найдем подходящее значение переменной i.
Правильный тип для этой переменной - int.

Задание 11. Определите битовую ширину acc так, чтобы это было безопасно.
Решение 11. Пусть data_t будет int10, тогда согласно формуле разрядность acc = 10 + 10 + 1. Результаты синтеза представлены на рисунке П.54.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.987	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
23	34	23	34	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop	22	33	2 ~ 3	-	-	11	no

Рисунок П.54.

Отчет об использовании ресурсов представлен на рисунке П.55.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	0	193
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	30	4
Multiplexer	-	-	-	99
Register	-	-	93	-
Total	0	0	123	296
Available	40	40	16000	8000
Utilization (%)	0	0	~0	3

Рисунок П.55.

Вывод по заданию 11. Таким образом, благодаря сужению разрядности типов данных удалось снизить задержку с 45 до 33.

10.1. Комплексный FIR-фильтр

К этому моменту мы рассмотрели исключительно фильтрацию действительных чисел. Многие цифровые системы беспроводной связи работают с комплексными числами, используя синфазные (I) и квадратурные (Q) компоненты. К счастью, можно создать сложный FIR-фильтр, используя реальный FIR-фильтр, как мы опишем ниже.

Чтобы понять, как построить сложный FIR-фильтр из реальных FIR-фильтров, рассмотрим уравнение (4). Предположим, что (I_{in}; Q_{in}) является одной выборкой входных данных, которые мы хотим отфильтровать. И один из комплексных коэффициентов FIR-фильтра обозначается как (I_{fir}; Q_{fir}). Будет более одной входной выборки и сложного коэффициента, но пока не будем об этом беспокоиться.

$$\left(I_{in} + jQ_{in}\right)\left(I_{fir} + jQ_{fir}\right) = \left(I_{in}I_{fir} - Q_{in}Q_{fir}\right) + j\left(Q_{in}I_{fir} + I_{in}Q_{fir}\right) \quad (4)$$

Уравнение (4) показывает умножение входного комплексного числа на один коэффициент комплексного фильтра FIR. Правая часть уравнения показывает, что действительная часть вывода комплексного ввода, отфильтрованного комплексным фильтром FIR , - это $I_{in}I_{fir} - Q_{in}Q_{fir}$, а мнимый вывод - $Q_{in}I_{fir} + I_{in}Q_{fir}$. Это подразумевает, что мы можем разделить сложную операцию FIR-фильтра на четыре реальных фильтра, как показано на рисунке 9.

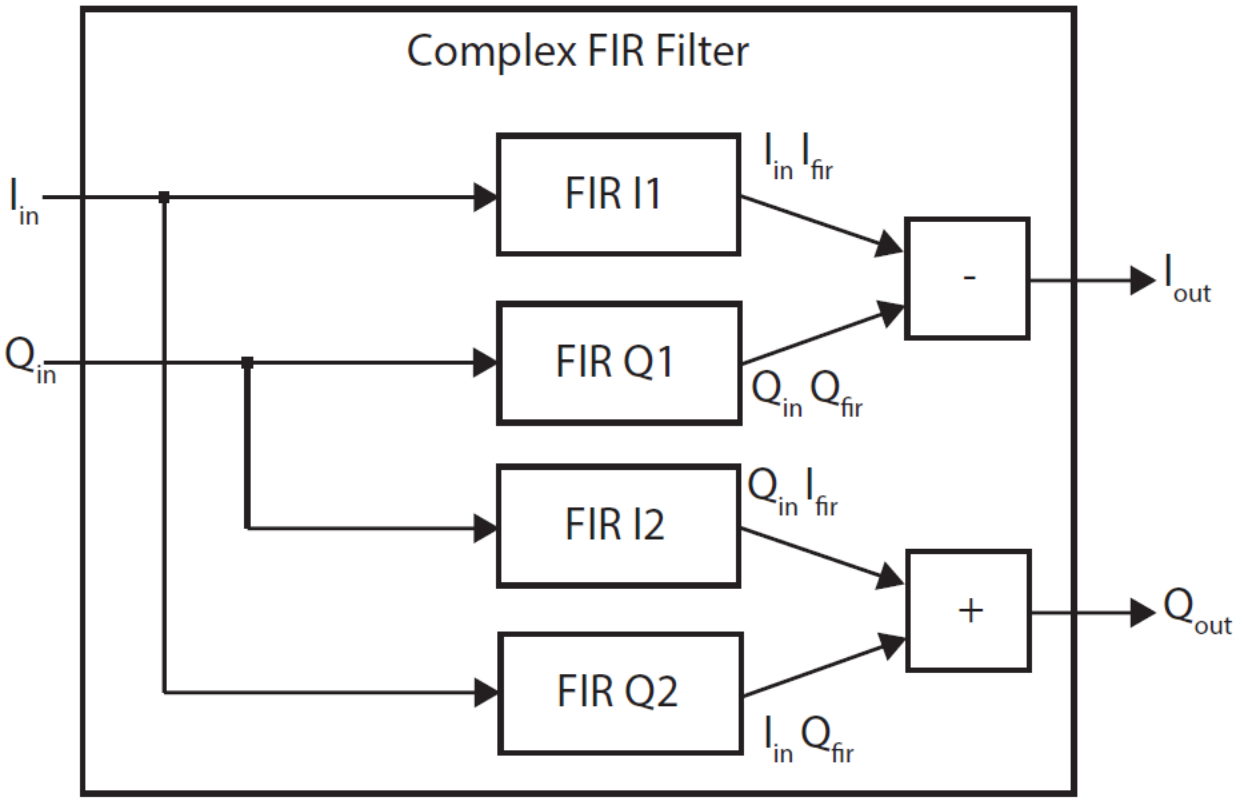


Рисунок 9: Сложный FIR-фильтр, построенный из четырех реальных FIR-фильтров. Входные сэмплы I и Q подаются в четыре разных реальных фильтра FIR. Фильтры содержат синфазные (FIR I) и квадратурные (FIR Q) комплексные коэффициенты.

```
typedef int data_t;
void firI1(data_t *y, data_t x);
void firQ1(data_t *y, data_t x);
void firI2(data_t *y, data_t x);
void firQ2(data_t *y, data_t x);

void complexFIR(data_t Iin, data_t Qin, data_t *Iout, data_t *Qout) {

    data_t IinIfir, QinQfir, QinIfir, IinQfir;

    firI1(&IinIfir, Iin);
    firQ1(&QinQfir, Qin);
    firI2(&QinIfir, Qin);
    firQ2(&IinQfir, Iin);

    *Iout = IinIfir + QinQfir;
    *Qout = QinIfir - IinQfir;
}
```

Рисунок 10. Код Vivado R HLS для иерархической реализации сложного КИХ-фильтра с использованием четырех реальных КИХ-фильтров.

Комплексный FIR-фильтр принимает в качестве входных данных комплексное число (I_{in}; Q_{in}) и выводит комплексное отфильтрованное значение (I_{out}; Q_{out}). На рисунке 9 представлена блок-схема этого комплексного фильтра с использованием четырех реальных фильтров FIR (FIR I1, FIR Q1, FIR I2, FIR Q2). Фильтры FIR I1 и FIR I2 эквивалентны, то есть имеют одинаковые коэффициенты. FIR Q1 и FIR Q2 также эквивалентны. Выход каждого из этих фильтров соответствует члену из уравнения 2.4.

Задание 12. Адаптируйте базовую имплементацию фильтра fir для работы с комплексными числами.

Эти выходные данные затем добавляются или вычитаются для получения окончательного отфильтрованного комплексного выхода (I_{out}; Q_{out}).

Мы использовали иерархическую структуру для определения этого сложного КИХ-фильтра. Vivado HLS реализует иерархию с использованием функций. Взяв предыдущую реальную функцию FIR void fir (data_t * y, data_t x), мы можем создать еще одну функцию, которая инкапсулирует четыре версии этой функции fir для создания комплексного фильтра FIR. Этот код показан на рисунке 10. Код определяет четыре функции firI1, firQ1, firI2 и firQ2. Каждая из этих функций имеет точно такой же код, например, код функции fir на рисунке 1. Как правило, нам не нужно дублировать функцию; мы могли бы просто вызвать одну и ту же функцию четыре раза.

Однако это невозможно в этом случае из-за статического ключевого слова, используемого в функции fir для shift_reg.

Вызовы функций действуют как интерфейсы. Инструмент Vivado HLS не оптимизирует границы функций. То есть каждая функция FIR синтезируется независимо и более или менее рассматривается как черный ящик в функции complexFIR. Вы можете использовать встроенную директиву, если хотите, чтобы инструмент Vivado HLS совместно оптимизировал определенную функцию в своей родительской функции. Это добавит код из этой функции в родительскую функцию и исключит иерархическую структуру. Хотя это может увеличить потенциальные преимущества в

производительности и области, оно также создает большой объем кода, который инструмент должен синтезировать. Это может занять много времени, даже не синтезировать, или может привести к неоптимальному дизайну. Поэтому осторожно используйте встроенную директиву. Также обратите внимание, что инструмент Vivado HLS может выбрать встроенные функции самостоятельно. Обычно это функции с небольшим количеством кода.

10.2. Программная реализация комплексного FIR-фильтра

Задание 12. Адаптируйте базовую имплементацию фильтра fir для работы с комплексными числами.

Решение 12. Код базовой имплементации фильтра для работы с комплексными числами представлена на Листинге.

```
typedef int data_t;
void firI1(data_t *y, data_t x);
void firQ1(data_t *y, data_t x);
void firI2(data_t *y, data_t x);
void firQ2(data_t *y, data_t x);
void complexFIR(data_t Iin, data_t Qin, data_t *Iout, data_t *Qout) {
        data_t IinIfir, QinQfir, QinIfir, IinQfir;
        firI1(&IinIfir, Iin);
        firQ1(&QinQfir, Qin);
        firI2(&QinIfir, Qin);
        firQ2(&IinQfir, Iin);
        *Iout = IinIfir + QinIfir;
        *Qout = QinIfir - IinQfir;
}
```

Листинг П.9.

Результаты синтеза представлены на рисунке П.56.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.470	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
114	180	114	180	none

Detail

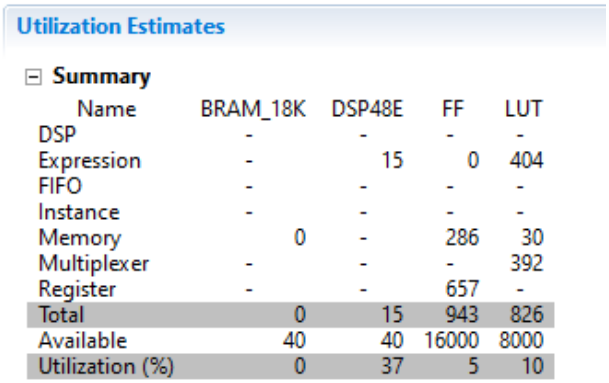
Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop_I1	33	55	3 ~ 5	-	-	11	no
- Loop_Q1	22	22	2	-	-	11	no
- Loop_I2	33	55	3 ~ 5	-	-	11	no
- Loop_Q2	22	44	2 ~ 4	-	-	11	no

Рисунок П.56.

Отчет об использовании ресурсов представлен на рисунке П.57.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	15	0	404
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	286	30
Multiplexer	-	-	-	392
Register	-	-	657	-
Total	0	15	943	826
Available	40	40	16000	8000
Utilization (%)	0	37	5	10

Рисунок П.57.

Вывод по заданию 12. КИХ-Фильтр для работы с комплексными числами потребует приблизительно в 4 раза больше ресурсов, чем обычный фильтр, а именно: 15 блоков DSP48E, 286 FF памяти, 657FF регистров, общее количество фип-флопов составит 943FF, количество expression составит 404LUT, памяти 30 LUT, мультиплексеров 392 LUT, таким образом количество таблиц входов-выходов (Look-Up Table) составит с 826LUT.

ВЫВОД

В работе были рассмотрены основные программные методы оптимизации КИХ-фильтра с использованием инструмента Vivado HLS на языке C/C ++. КИХ фильтры — линейные нерекурсивные фильтры, достаточно просты в реализации на современных FPGA и процессорах.

Подробно были рассмотрены оптимизации по вынесению условных операторов за границы цикла, разворачиванию циклов, конвейерной обработки и распараллеливанию циклов, по структуре и типам данных. Создание оптимальной архитектуры КИХ-фильтра требует хороших навыков программирования на языке C/C ++, базового понимания того, как инструмент HLS выполняет процесс синтеза и оптимизации в коде RTL. Часто для оптимизации требуется проанализировать код и учитывать как работает RTL синтез и его оптимизации.

Хочется отметить следующие моменты, которые необходимо учитывать:

- Порядок фильтра = длина импульсной характеристики = количество коэффициентов = количество линий задержки.
- КИХ фильтры могут иметь точную линейную фазовую характеристику, постоянные групповую и фазовую задержки.
- Для линейной фазовой характеристики требуются симметричные коэффициенты фильтра.
- Старайтесь упростить код, используя метод разбиение циклов
- Существует множество методов разработки КИХ фильтров на ПЛИС. Выбирайте тот, который покажется проще.

Список источников

[1] Edward A. Lee and Pravin Varaiya. Structure and Interpretation of Signals and Systems, Second Edition. 2011. ISBN 0578077191. URL LeeVaraiya.org.

[2] Stephen Brown and Jonathan Rose. FPGA and CPLD architectures: A tutorial. IEEE Design and Test of Computers, 13(2):42{57, 1996.

[3] Ryan Kastner, Anup Hosangadi, and Farzan Fallah. Arithmetic optimization techniques for hardware and software design. Cambridge University Press, 2010.

[4] Shahnam Mirzaei, Anup Hosangadi, and Ryan Kastner. FPGA implementation of high speed fir filters using add and shift method. In Computer Design, 2006. ICCD 2006. International Conference on, pages 308{313. IEEE, 2007.

[5] Ryan Kastner, Janarbek Matai, and Stephen Neuendorer. Parallel Programming for FPGAs. 2018-12-11. 246p. 2018-12-11