От инженеров для инженеров

FPGA-Systems.ru

Разработка IP-блока с помощью инструментов высокоуровнего синтеза: HLS

Часть 1

Автор: **PointPas**

Рецензенты: intekus

KeisN13

2019 г.

Оглавление

Vivado HLS: Разработка простого IP-блока

Эта статья – небольшое руководство для тех, кто хочет сделать собственный IP-блок для FPGA (фирмы Xilinx) с помощью HLS (High Level Synthesis [1], синтез с языков высокого уровня). Ниже по порядку будут описаны основные шаги такой разработки. В качестве примера будет разработан простейший ШИМ с управлением по шине AXI4-Lite, который будет изменять яркость светодиода на отладочной плате MiniZed [2].

Предполагается, что у вас уже установлена Vivado. Я использовал Vivado HL WebPACK Edition 2018.2.

Высокоуровневые средства разработки: что это и зачем оно нужно?

Для разработки IP-блоков в FPGA обычно используют языки описания аппаратуры (HDL) такие как VHDL, Verilog, System Verilog. Чтобы писать код на этих языках, необходимо иметь определённое представление о цифровой схемотехнике.

Vivado HLS транслирует в описания на HDL (которые можно использовать для дальнейшего синтеза и имплементации) код на языках C/C++ или SystemC. Использование HLS [3, 4] снижает порог входа в разработку на FPGA, т. к. позволяет писать код разработчику, не знакомому с HDL: для создания своего работающего модуля (или даже проекта) уже не обязательно досконально знать, что такое «тактовая частота» и некоторые другие низкоуровневые вещи. Естественно, редко люди с таким уровнем подготовки создают проект на FPGA целиком, «от и до». Но реализовать на FPGA хотя бы на уровне прототипа отдельный модуль, исполняющий знакомый специалисту (программисту для ПК, инженеру по обработке сигналов, математику и т. д.) алгоритм становится для него вполне «подъёмной» задачей и без постоянного и плотного привлечения ПЛИС-разработчика. Также использование HLS удобно, когда важно добиться работы на FPGA поведенческой модели, не беспокоясь об объёме занятых ресурсов кристалла.

Итак, Vivado HLS позволяет:

- Вести разработку на C-уровне
- Верифицировать на C-уровне
- Контролировать процесс синтеза с помощью директив компилятора “pragma HLS”
- Создавать код, работающий на любом семействе FPGA

Шаг 1: Создание нового проекта

Предлагаю два варианта: первый – просто «протыкать» GUI и заполнить пошагово все описанные поля. Второй способ – для совсем ленивых: запустить tcl-скрипт, который сам создаст проект и добавит файлы (кстати, аналогичным способом можно сразу запустить тест, синтез, ко-симуляцию и экспорт IP – совсем без вызова GUI, что иногда может быть очень удобно).

Способ 1

1. Запускаем Vivado HLS. В Windows – двойным щелчком по соответствующей пиктограмме на рабочем столе, в Linux – набрав в командной строке «vivado_hls». Далее все шаги одинаковы для обеих ОС.

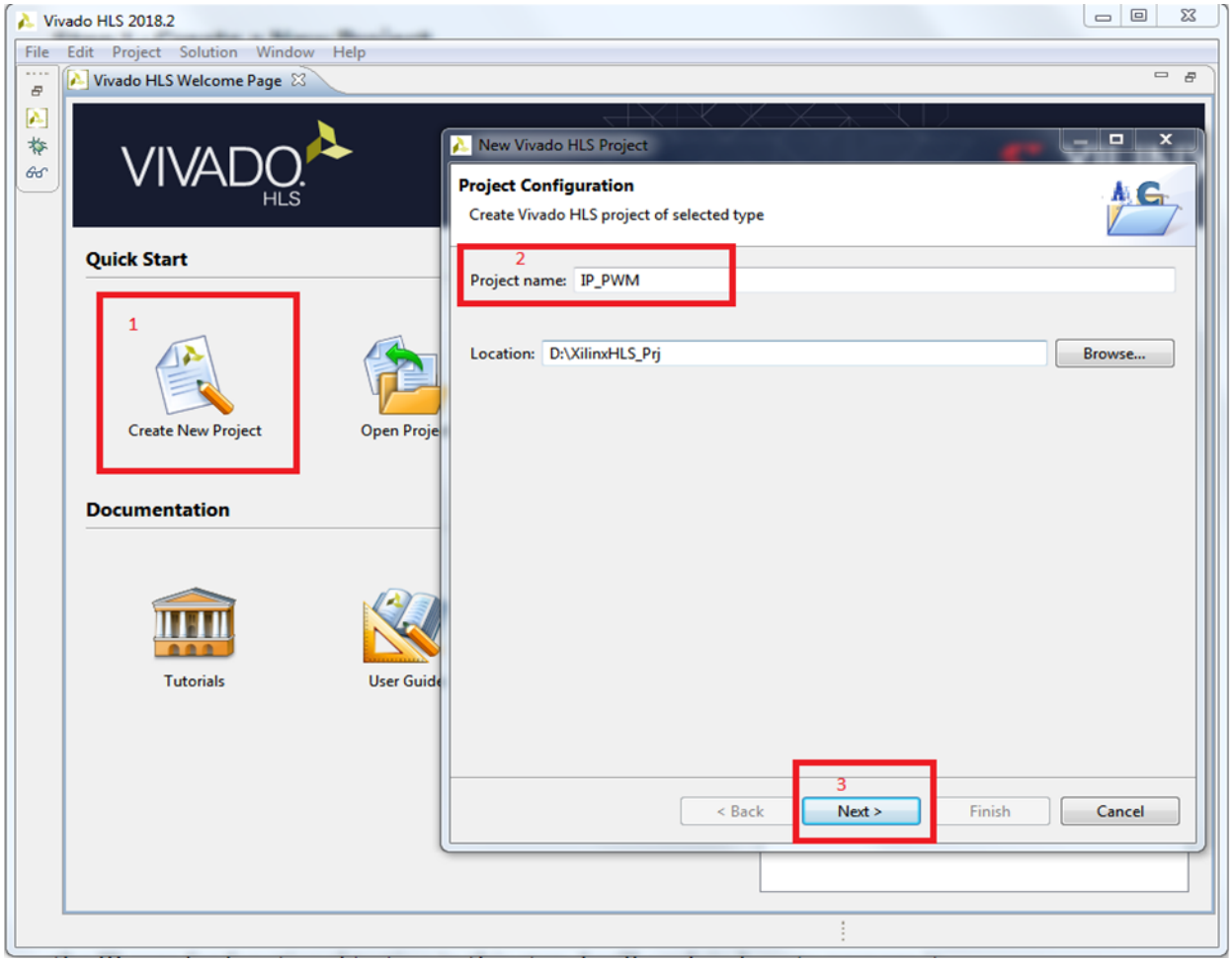

Рисунок 1 – Создание проекта

2. Рисунок 1:
   2.1. Жмем «Создать новый проект».
   2.2. Называем его «`IP_PWM`». Директория, где будет храниться проект, нам не важна.
   2.3. Жмем «Далее».

3. Рисунок 2:
   3.1. Жмем «Добавить файлы» (в этом окне – файлы исходных текстов наших модулей). Если вы создаете свой проект, и пока не знаете, как захотите назвать файлы, то можно просто нажать «Далее».
   3.2. «Top Function» для HLS – это аналог модуля верхнего уровня для проекта на HDL. Набираем тут «`PWM`».
3.3. Жмем «Далее».

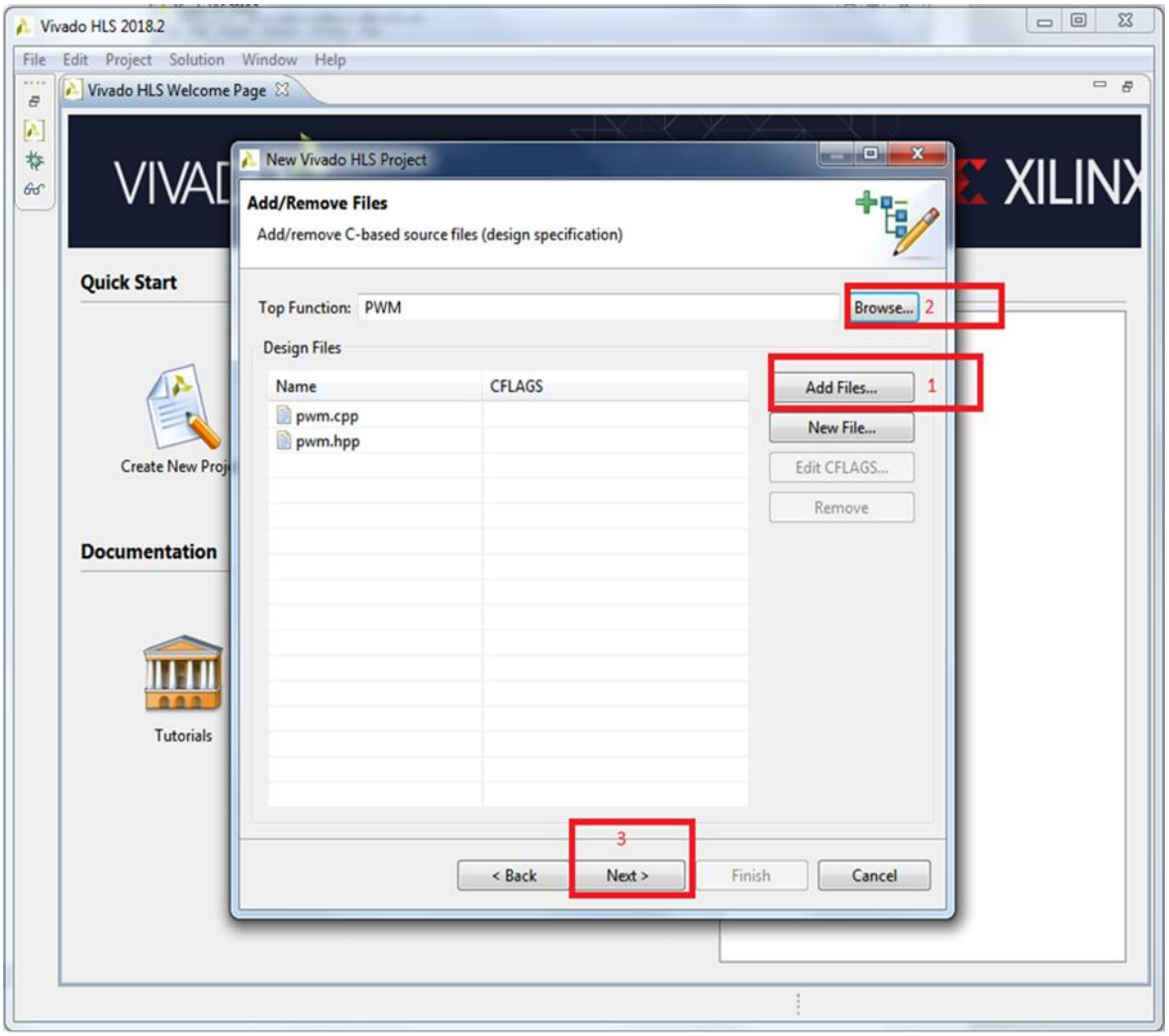

Рисунок 2 – Добавляем файлы исходных текстов модулей

4. Рисунок 3:
   4.1. Жмем «Добавить файлы» (на сей раз – файлы тестов). Если Вы создаете свой проект, и пока не знаете, как захотите назвать файлы, то можно просто нажать «Далее».
   4.2. Жмем «Далее».

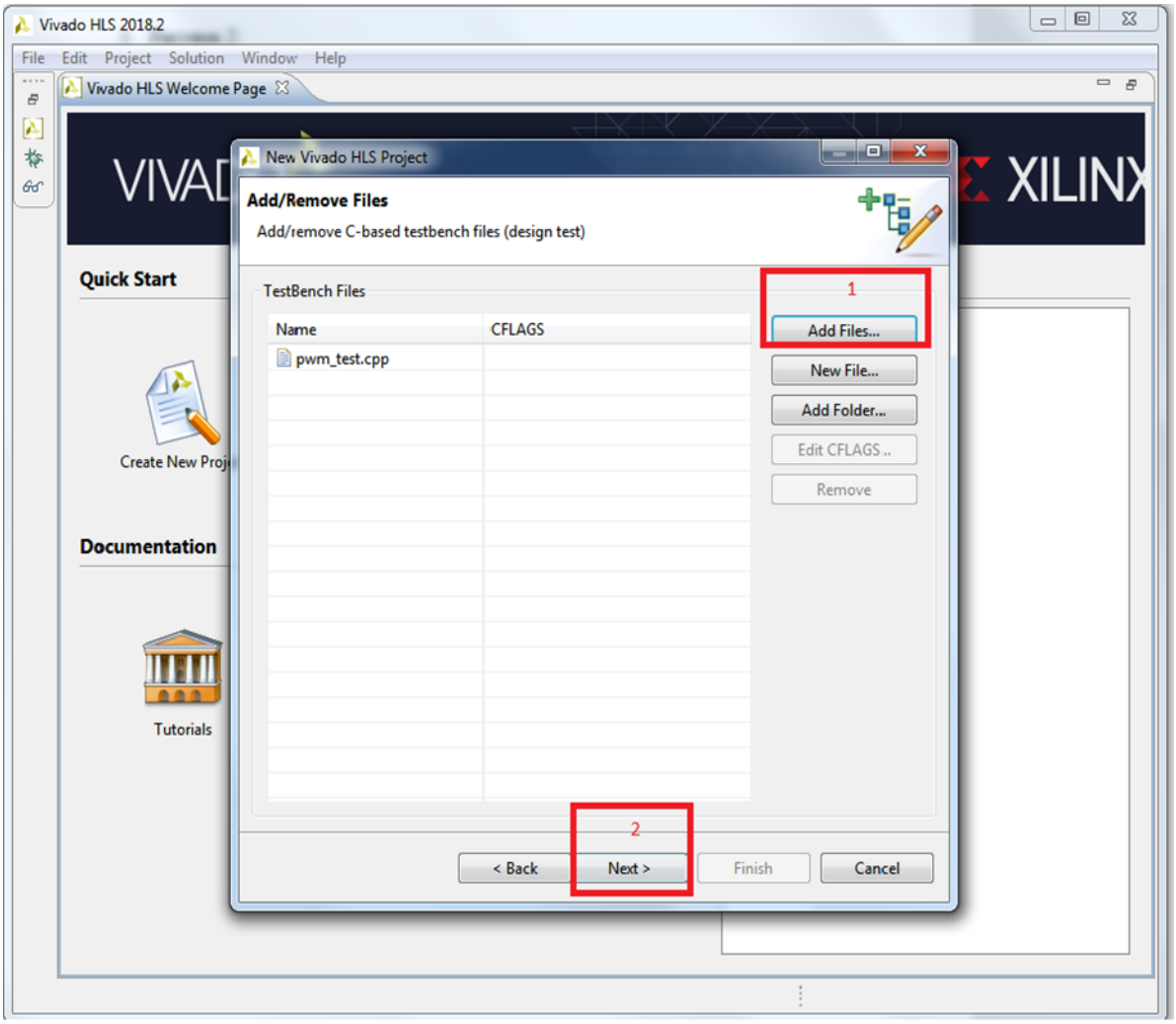

Рисунок 3 – Добавляем файл теста

5. Рисунок 4:
   5.1. Называем решение «PWM».

   Выбираем свою ПЛИС. Если у вас есть какая-нибудь отладочная плата, то можно прописать в файл `VivadoHls_boards.xml` строку по аналогии с теми, что уже там написаны. Для MiniZed она выглядит так:

```
<board name="Xilinx_Minized" display_name="Minized" family="zynq"
    part="xc7z007sclg225-1"  device="xc7z007s" package="clg225"
            speedgrade="-1" vendor="em.avnet.com" />
```

Файл можно найти в

`путь_к_директории_установки\Vivado\2018.2\common\config`.

5.2. Период оставляем равным 10 нс.

5.3. Жмем «Завершить»

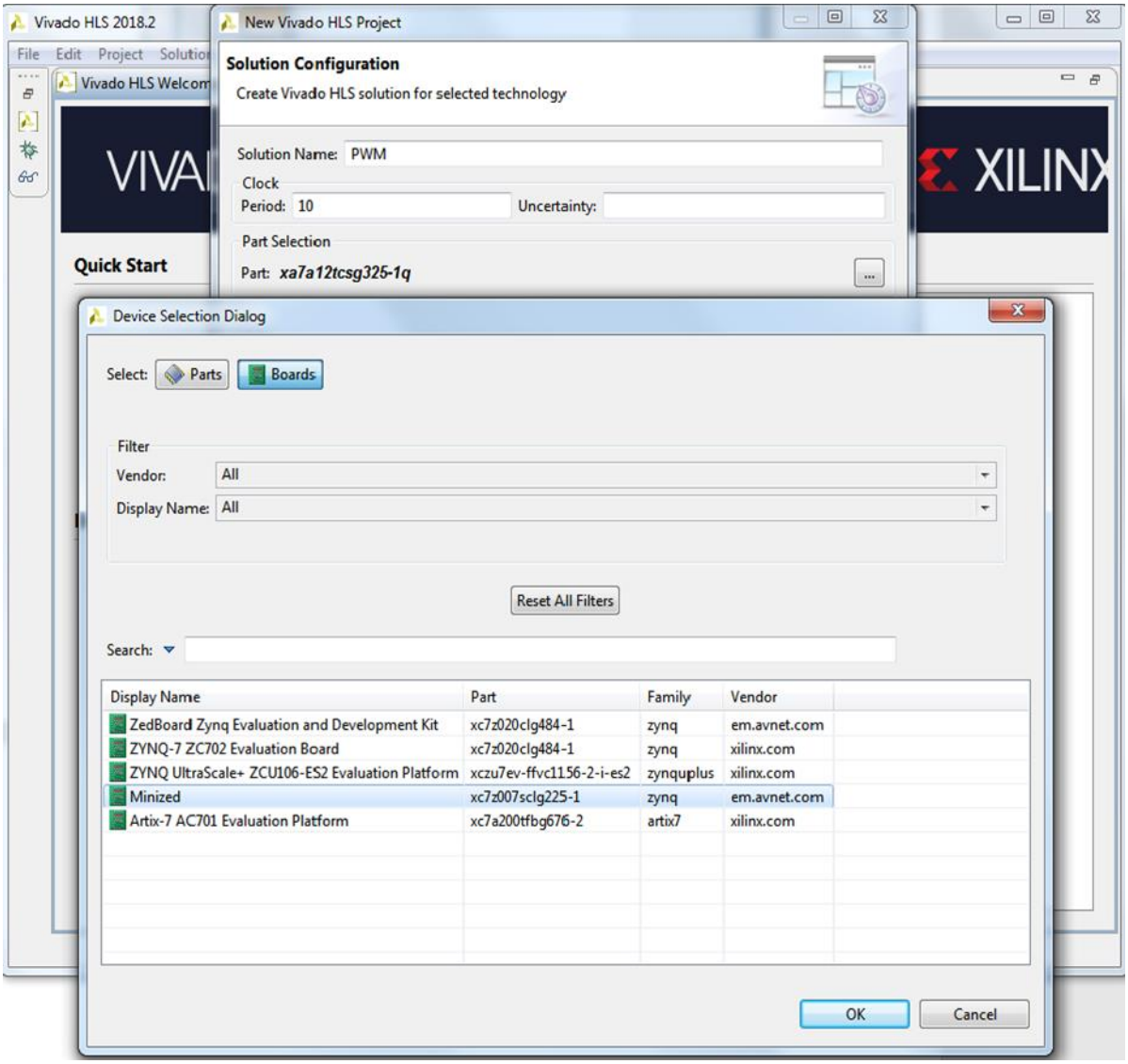

Рисунок 4 – Выбираем ПЛИС/плату

Способ 2

1. В windows запускаем командную строку Vivado HLS (рисунок 5), в Linux запускаем новый терминал.

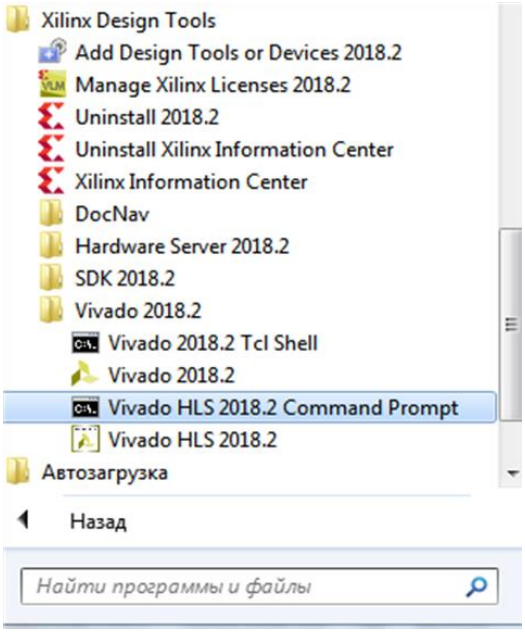

Рисунок 5 – Запуск командной строки Vivado HLS

2. В командной строке пишем `cd путь_к_директории_где_лежат_исходники` (Рисунок 6).

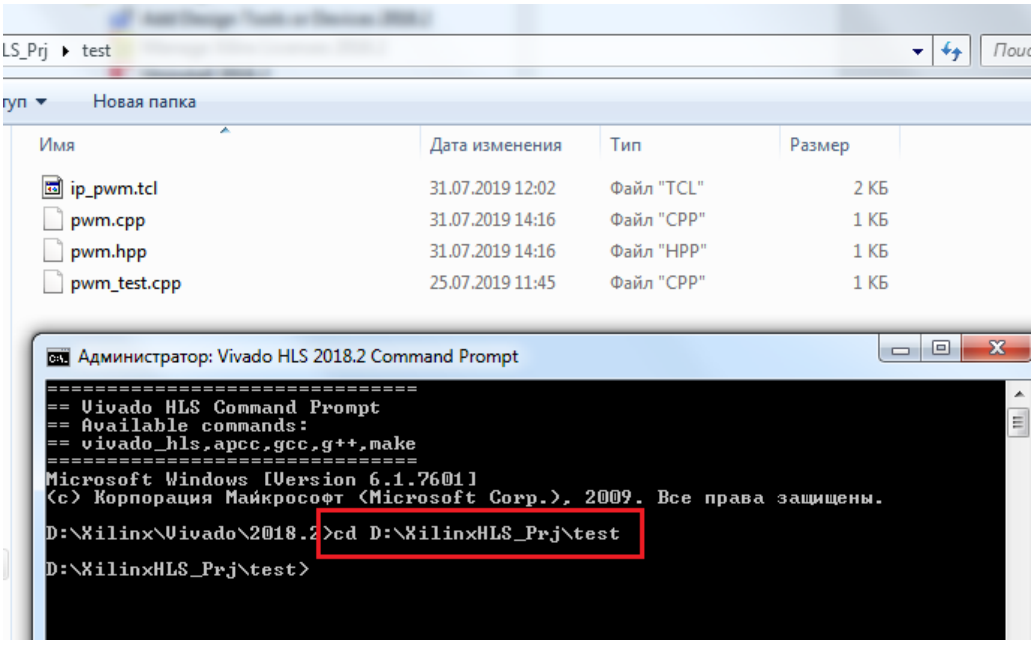

Рисунок 6 – Заходим в папку с файлами проекта

3. Пишем в консоль `vivado_hls -f ip_pwm.tcl` (Рисунок 7).

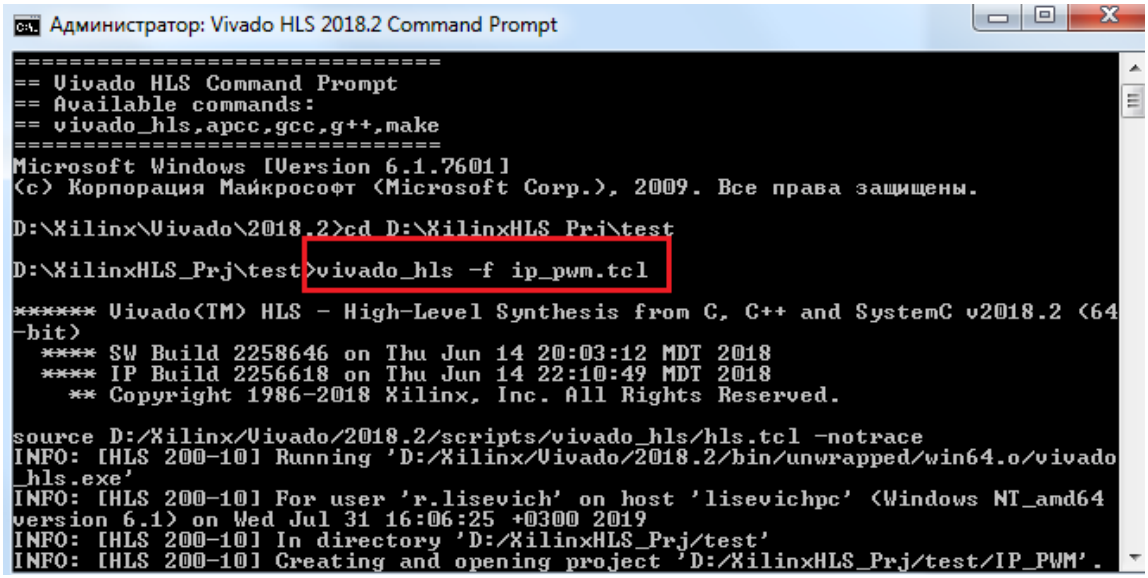

Рисунок 7 – Запускаем tcl скрипт

4. Пишем в консоль `vivado_hls -p IP_PWM` (Рисунок 8).

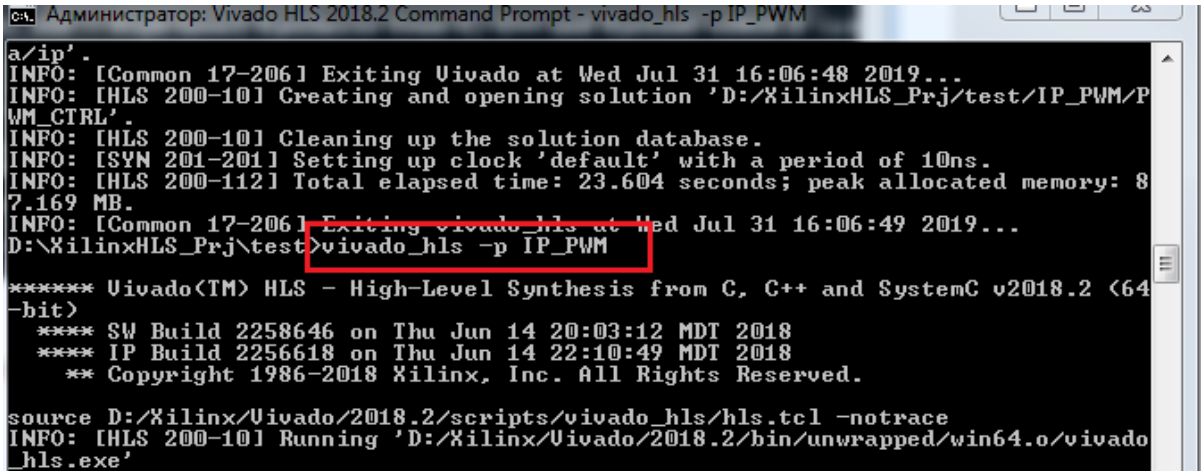

Рисунок 8 – Запуск проекта из консоли

Шаг 2: Разработка IP-блока

Разработку нашего IP начнем с рисования структурной схемы системы. Т. к. в нашем распоряжении – система на кристалле (на плате MiniZed стоит SoC Xilinx Zynq [5]), то мы являемся счастливыми обладателями ARM-ядра, и управлять нашим IP станем с его помощью. Если же у Вас просто ПЛИС без процессорной части, то Вы можете поднять софт-процессор Microblaze [6] и управлять нашим IP им.

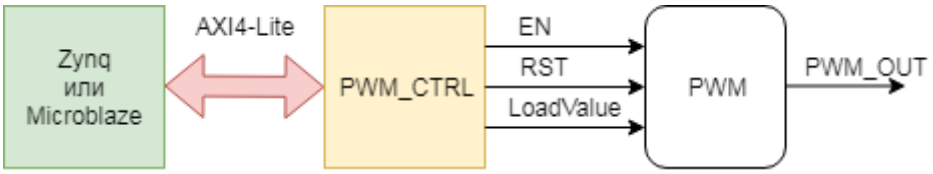

Рисунок 9 – Структурная схема

Наш ШИМ будет состоять из двух функций. Первая будет реализовывать интерфейс AXI4-Lite, через который мы будем задавать значения регистров. Вторая функция будет реализовывать счетчик (ШИМ представляет собой счетчик, который, досчитав до заданного ему значения, меняет состояние выхода регистра на противоположное). Почему это реализовано в виде двух разных функций, станет понятно, когда мы дойдем до написания теста к нашему модулю.

Если в прошлой части Вы сделали все правильно, то в левой части рабочего окна у Вас должен появиться Проводник (Рисунок 10). При этом если Вы создавали проект первым способом, то решения `PWM_CTRL` у Вас не будет, и мы добавим его позже.

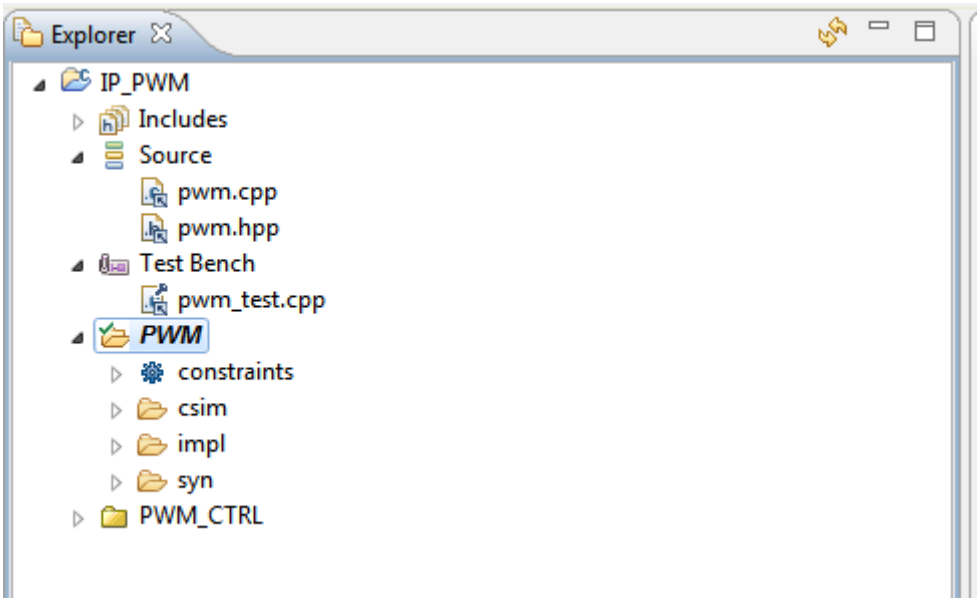

Рисунок 10 – Проводник в Vivado HLS

Приступим к реализации. Для начала подключим заголовочные файлы, которые нам понадобятся. В проекте все заголовочные файлы, определенные новые имена существующих типов данных и прототипы функций вынесены в отдельный файл `pwm.hpp` (Рисунок 11).

Для C++ заголовочный файл `<ap_int.h>` определяет целочисленные типы данных произвольной точности: `ap_int<N>` и `ap_uint<N>` (беззнаковый), где N может принимать значение от 1 до 1024 (можно переопределить и до значения в 32768). Рекомендуется использовать этот тип данных, даже если у Вас разрядность совпадает со стандартными типами данных, такими как char, int и т.д., потому что у переменных такого типа есть методы для работы с операциями на битовом уровне, т. е. мы можем получать или устанавливать значения определенных бит внутри переменной, объединять переменные и многое другое (все подробности смотрите в UG902 [4]).

Сразу определим новые имена типов данных для удобства.

Заголовочный файл `<stdio.h>` – стандартный заголовочный файл с функциями ввода-вывода. Он нам понадобится, когда мы будем писать тест для нашего IP.

```
#include <ap_int.h>
#include <stdio.h>

typedef ap_uint<1>  wire;
typedef ap_uint<16> data16;

void PWM(data16 LoadValCnt, wire EN, wire Rst, wire* OutPWM);

void PWM_CTRL(data16 LoadValCnt, data16 *LoadValCnt_r, wire EN, wire *EN_r, wire Rst, wire *Rst_r);

int PWM_test();
```

Рисунок 11 – Файл `pwm.hpp`

Теперь следует написать функцию, которая бы непосредственно выполняла необходимую нам задачу. Во время этого шага мы пока не обращаем внимания на то, во что она будет синтезирована как потом подготовить функцию к синтезу, будет рассказано позже.

Начнем с функции `PWM_CTRL` (Рисунок 12). Эта функция должна просто хранить и выдавать значения, которые будут управлять счетчиком. Она не должна ничего возвращать, поэтому её тип возврата – `void`. Т. к. счетчик будет шестнадцатиразрядный, то загружаемое значение должно быть той же разрядности. Сигналы включения и программного сброса – однобитные.

```
#include "pwm.hpp"

void PWM_CTRL(data16 LoadValCnt,
              data16 *LoadValCnt_r,
              wire EN,
              wire *EN_r,
              wire Rst,
              wire *Rst_r
              ){
    *LoadValCnt_r = LoadValCnt;
    *EN_r = EN;
    *Rst_r = Rst;
};
```

Рисунок 12 – Функция PWM_CTRL

Теперь рассмотрим функцию `PWM` (Рисунок 13). Эта функция должна увеличивать значение переменной `counter` каждый раз, когда мы вызываем эту функцию, при условии, что сброс не выставлен (т. е. `Rst == 0`) и активно разрешение счёта (`EN == 1`). Поэтому мы используем ключевое слово `static` (это значит, что переменная `counter` создается и инициализируется только один раз, и каждый следующий вызов этой функции будет использовать её предыдущее значение). Когда значение в счетчике перестанет быть меньше значения, с которым мы его сравниваем, выход изменит значение с логической «1» на логический «0».

```
void PWM(data16 LoadValCnt,
         wire EN,
         wire Rst,
         wire* OutPWM
         ){

static data16 counter = 0;

*OutPWM = (counter < LoadValCnt) ? 1 : 0;

if(!Rst & EN){
        counter++;
    }
    else{
        counter = 0;
    }
};
```

Рисунок 13 – Функция PWM

Шаг 3: Пишем тест

Теперь для проверки корректности работы созданного модуля нам необходимо написать функцию, которая будет вызывать функции, написанные нами на прошлом этапе, и проверять правильность полученного от них результата. Эта функция будет вызываться в функции `main` нашего теста и должна возвращать 0, если тест пройден успешно. Можно и просто визуально смотреть, что получится, вручную «натыкав» в код операторов `printf`, либо воспользоваться отладчиком.

Нам необходимо проверить, что при достижении счетчиком значения, равного заданному (`LoadValCnt`), выход изменит значение с «1» на «0». Для этого зададим значения с помощью `PWM_CTRL` и вызовем наш счетчик (`LoadValCnt + 1`) раз. Переменная `res` увеличивается на 1, если выход ШИМ равен «1», и значение счетчика на следующей итерации должно стать равным значению `LoadValCnt`. При достижении значения `LoadValCnt` выход должен быть равен «0», а переменная `res` уменьшится на 1. Если что-то будет не так, то `res` не будет равна нулю. После написания это выглядит так: Рисунок 14.

```
#include "pwm.hpp"

int PWM_test(){

    wire OutPWM;
    wire Rst = 0;
    wire Rst_r;
    wire EN = 1;
    wire EN_r;
    data16 LoadValCnt = 100;
    data16 LoadValCnt_r;
    int res = 0;

    PWM_CTRL(LoadValCnt, &LoadValCnt_r, EN, &EN_r, Rst, &Rst_r);

    for (int i = 0; i <= LoadValCnt; i++){

        PWM(LoadValCnt_r, EN_r, Rst_r, &OutPWM);

        if((i == (LoadValCnt - 1)) & (OutPWM == 1)) res++;
        if((i == LoadValCnt) & (OutPWM == 0)) res--;
        }

    if(res == 0){
        std::cout << "**********Test PASS**********"<< std::endl;
        return 0;
    }
    else{
        std::cout << "**********Test FAIL**********"<< std::endl;
        return 1;
    }
};


int main(){

    return PWM_test();

}
```

Рисунок 14 –Тест

Шаг 4: Запускаем тест

Просто нажимаем кнопку «Run C Simulation» на панели инструментов вверху (Рисунок 15). Если у Вас свой проект, то понадобится указать свой файл с тестом в настройках проекта во вкладке «Моделирование»:

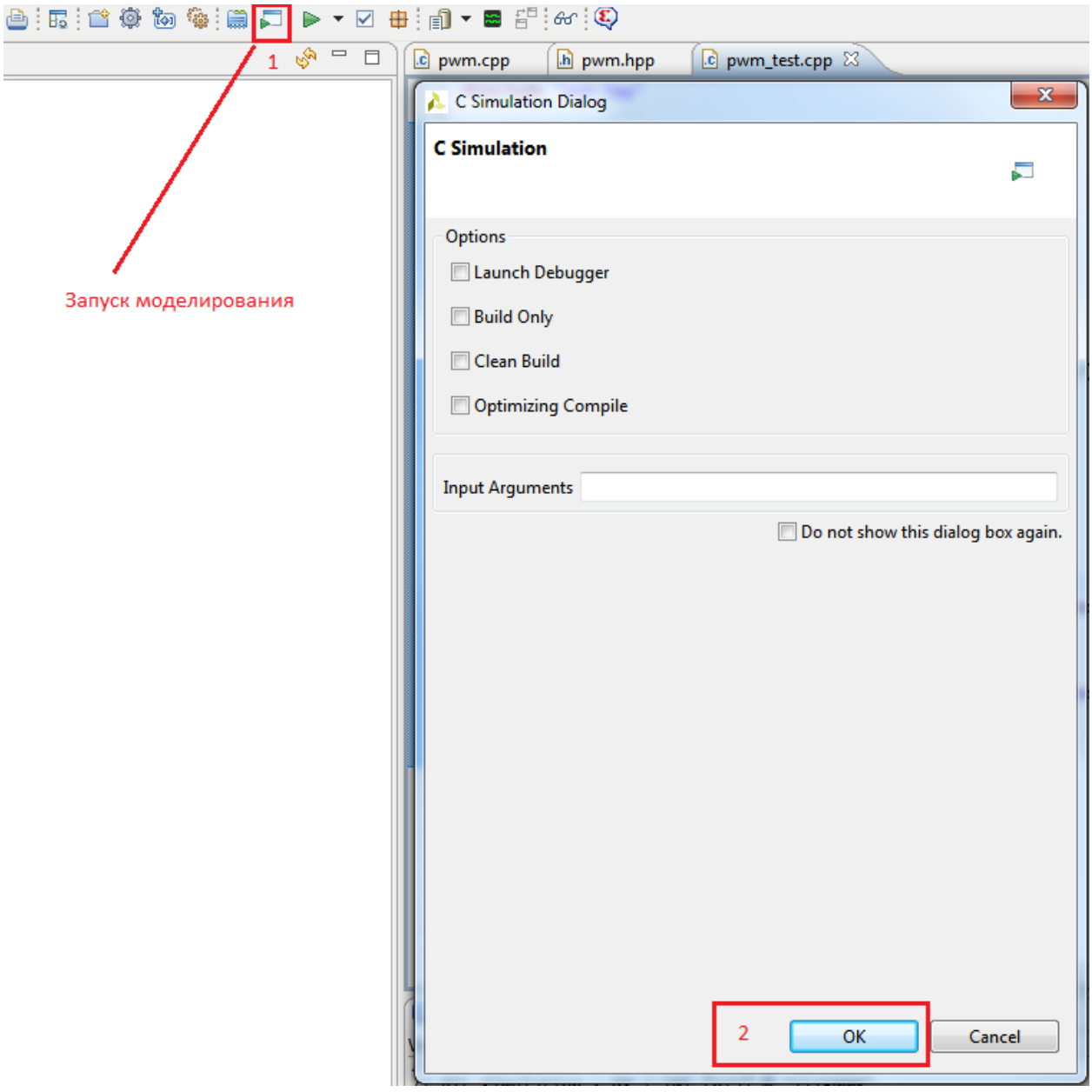

Рисунок 15 – Запускаем моделирование

Если все прошло нормально, то в консоли будет написано, что тест прошел успешно, без ошибок (Рисунок 16).

```
**********Test PASS**********
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

Рисунок 16 – Результат теста

Теперь перейдем к объяснению, почему было решено разделить наш IP-блок на две функции. Если бы весь функционал размещался в одной функции, то при каждом её вызове это отражалось бы в ко-симуляции, как транзакция записи по шине AXI4-Lite. Но такое поведение не соответствует заложенным нами в систему идеям, т. к. при использовании IP-блоков их регистры обычно только конфигурируются при инициализации и реконфигурируются по каким-то событиям, а в ходе обычной работы доступа к ним не осуществляется.

Шаг 5: Подготовка к синтезу

Контролировать процесс синтеза можно с помощью директив компилятора `pragma HLS`. Если этого не сделать, то в ходе синтеза будут применены параметры по умолчанию. Примененные директивы можно посмотреть в правой части рабочего экрана во вкладке директив.

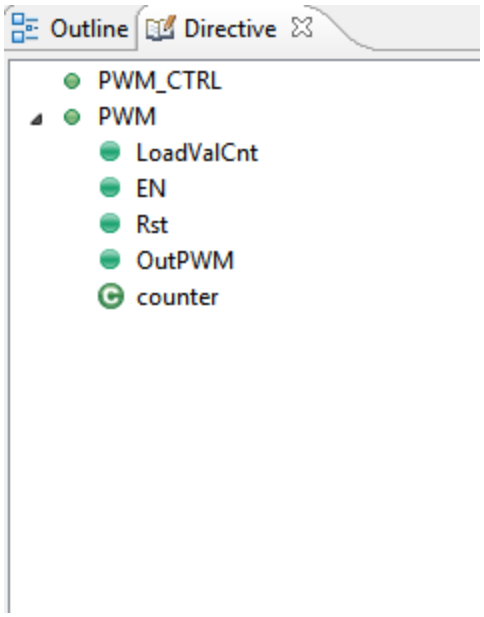

Рисунок 17 – Вкладка с директивами

Дважды щёлкнем по «`PWM`», откроется редактор директив, выберем в нём директиву и опции, как показано на рисунке ниже (Рисунок 18). Выбранная опция указывает компилятору, что на уровне функции нам не нужны сигналы управления (т. к. управлять мы будем сами). Директивы могут находиться непосредственно в описании функции, а могут находиться в файле с директивами, в нашем случае будем использовать первый вариант. Теперь сделаем тоже самое для

`OutPWM` (Рисунок 19). Если нажать кнопку «Help», то можно подробно ознакомиться с каждой опцией, которая относится к этой директиве. Также там можно прочитать, какие опции используются по умолчанию. На этом подготовка функции `PWM` к синтезу окончена.

Замечание по поводу ключевого слова `static`. В выходном HDL-коде соответствующая переменная будет описана в виде регистра. Но это не означает, что нужно использовать `static` в описании всех переменных, которые должны хранить свое значение в течение нескольких циклов, HLS способен распознать данную ситуацию сам. Но конкретно в нашем случае ключевое слово `static` указать необходимо, т. к. без него каждый новый вызов функции приводил бы к обнулению переменной `counter`.

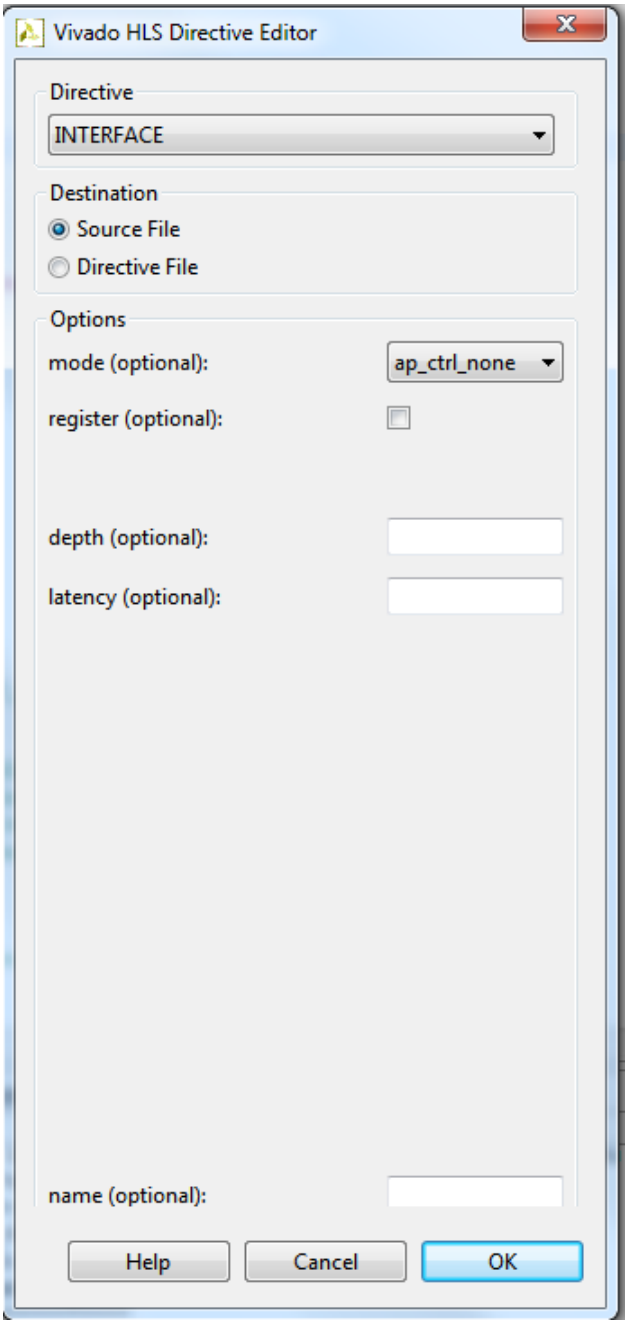

Рисунок 18 – Выбор директивы для PWM

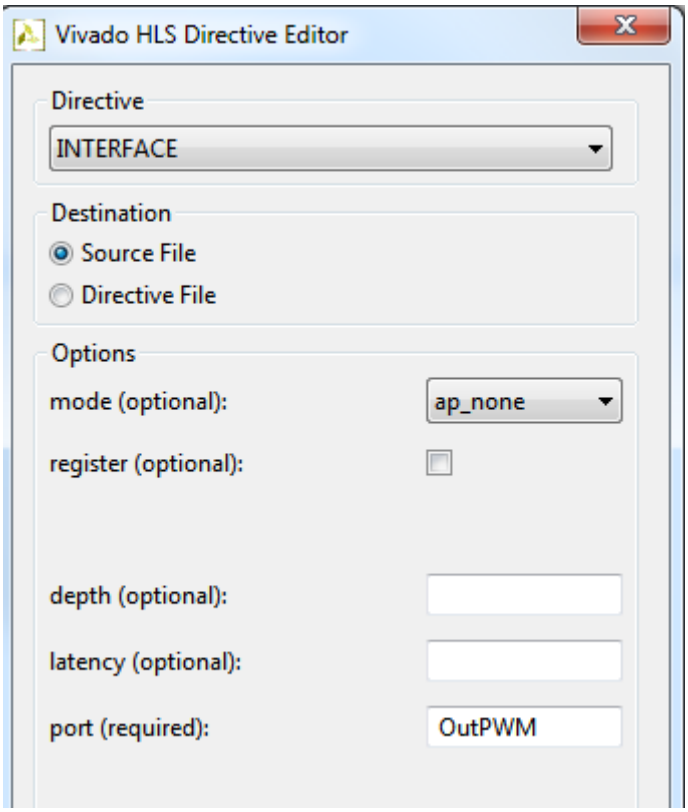

Рисунок 19 – Выбор директивы для OutPWM

Теперь подготовим к синтезу функцию `PWM_CTRL`. Для начала нужно указать в настройках проекта функцию, которая будет синтезирована (Рисунок 20). Тут, по аналогии с проектом на HDL, синтезируется все, что включается в модуль (в HLS – функцию), указанную «топовой» (верхнего уровня).

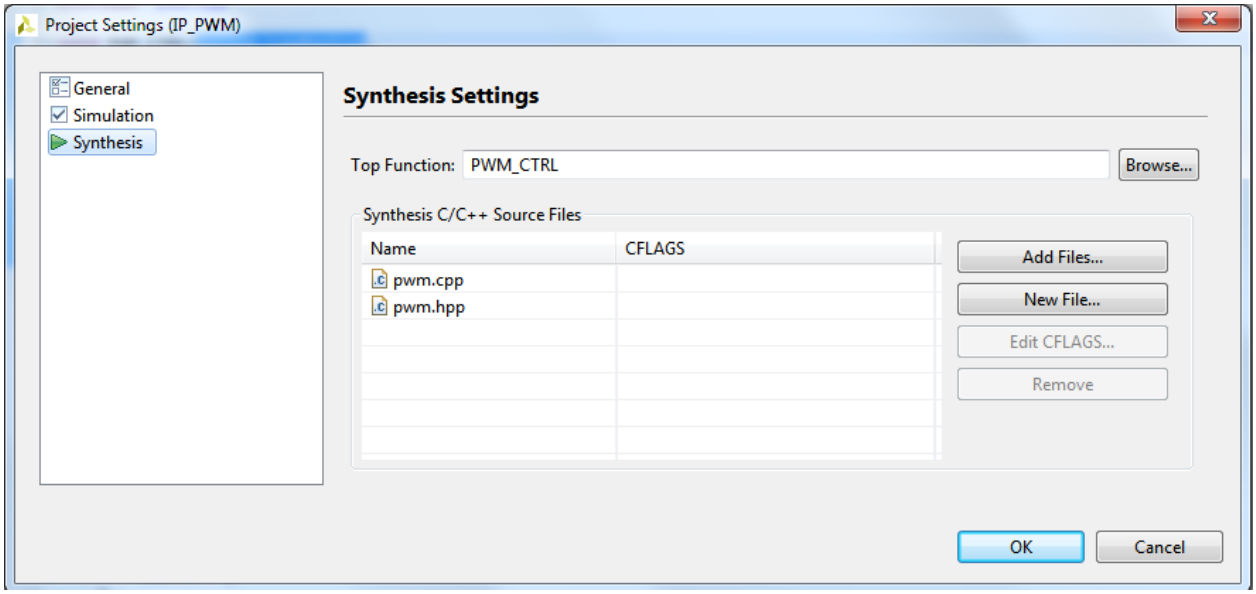

Рисунок 20 – Выбор функции для синтеза

Дважды щёлкнем по «PWM_CTRL», откроется редактор директив, в нём выберем директиву и опции как на рисунке ниже (Рисунок 21). Выбранная опция добавит к нашему IP порт AXI4-Lite с управляющими сигналами, с помощью которого будут задаваться значения наших переменных Rst, EN и LoadValCnt.

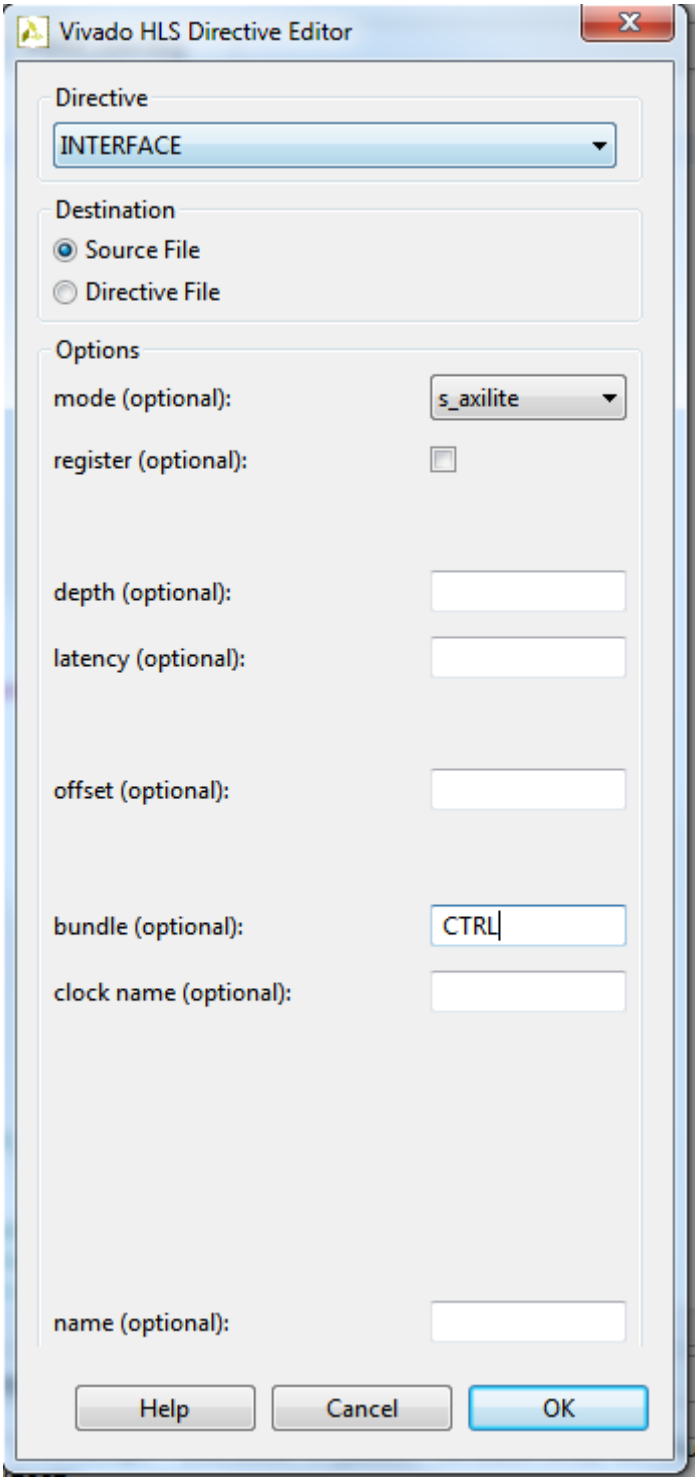

Рисунок 21 – Выбор директивы для PWM_CTRL

Для того, чтобы наши переменные были доступны через AXI, необходимо указать это (Рисунок 22). Важным моментом является опция названия объединения (bundle) т. к. иначе для каждой переменной будет создан свой отдельный порт.

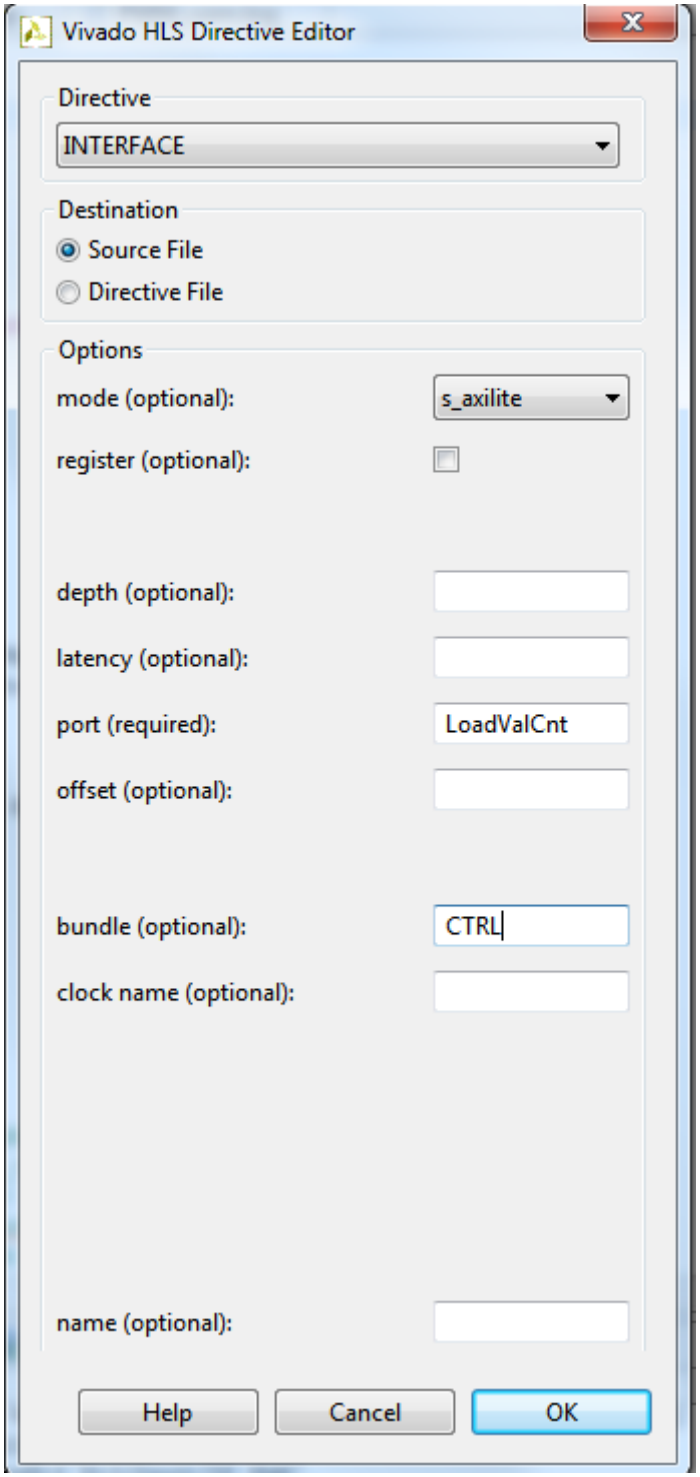

Рисунок 22 – Выбор директивы для переменной LoadValCnt

Повторим последнее действие для переменных `Rst` и `EN`. Для выходов применим директиву такую же, как ранее для вывода `OutPWM` (Рисунок 19). Должно получится, как на рисунке ниже (Рисунок 23):

```
#include "pwm.hpp"

void PWM_CTRL(data16 LoadValCnt,
              data16 *LoadValCnt_r,
              wire EN,
              wire *EN_r,
              wire Rst,
              wire *Rst_r
              ){
#pragma HLS INTERFACE ap_none port=LoadValCnt_r
#pragma HLS INTERFACE ap_none port=Rst_r
#pragma HLS INTERFACE ap_none port=EN_r
#pragma HLS INTERFACE s_axilite port=Rst bundle=CTRL
#pragma HLS INTERFACE s_axilite port=EN bundle=CTRL
#pragma HLS INTERFACE s_axilite port=LoadValCnt bundle=CTRL
#pragma HLS INTERFACE s_axilite port=return bundle=CTRL

    *LoadValCnt_r = LoadValCnt;
    *EN_r = EN;
    *Rst_r = Rst;
};

void PWM(data16 LoadValCnt,
         wire EN,
         wire Rst,
         wire* OutPWM
         ){
#pragma HLS INTERFACE ap_none port=OutPWM
#pragma HLS INTERFACE ap_ctrl_none port=return

static data16 counter = 0;

*OutPWM = (counter < LoadValCnt) ? 1 : 0;

if(!Rst & EN){
        counter++;
    }
    else{
        counter = 0;
    }
};
```

Рисунок 23 – Функции, подготовленные к синтезу

Шаг 6: Синтез функций, ко-симуляция, экспорт IP

Тем, кто создавал проект первым способом, сейчас необходимо будет добавить новое решение (Solution). Для этого вверху на панели инструментов нажимаем кнопку «New Solution» (Рисунок 24). Называем его `PWM_CTRL`.

Рисунок 24 – Создаем новое решение

Если вы делали все по пунктам, то у вас должна быть указана функция для синтеза `PWM_CTRL`; если делали не по пунктам, указываем её сейчас. Если у вас свой проект или вы создавали проект первым способ, то нужно в настройках решения указать, что нам нужен сброс с низким активным уровнем (Рисунок 25). Затем запускаем синтез (Рисунок 26).

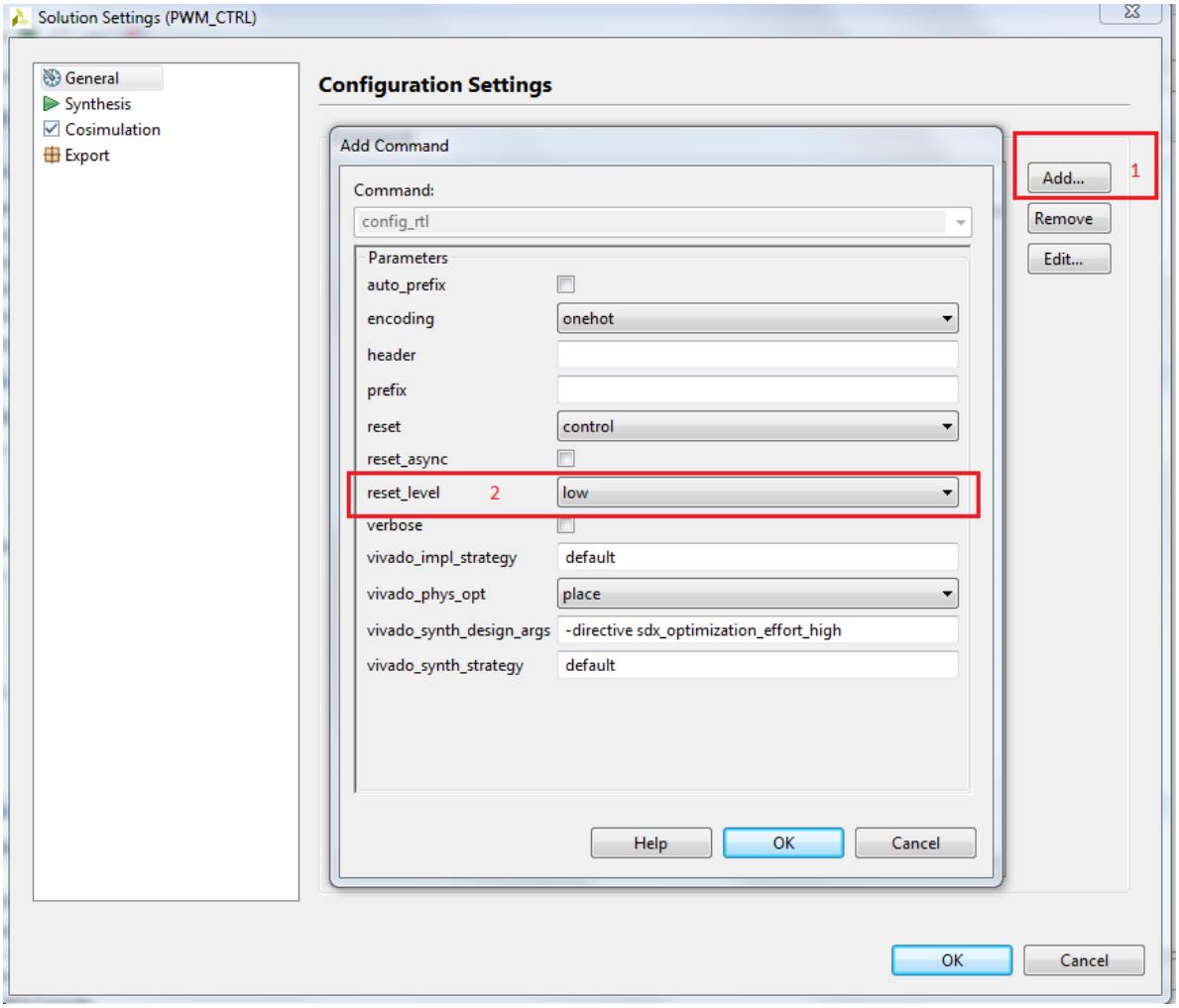

Рисунок 25 – Выбор активного уровня сброса

Рисунок 26 – Запуск синтеза

После того, как синтез закончится, автоматически откроется отчет, в котором можно посмотреть получившуюся долю использования ресурсов кристалла (утилизацию) и то, какие интерфейсы будут у нашего IP-блока (Рисунок 27). Как видим, добавились порты тактирования, сброса и прерываний.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	0	-	72	76
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	72	76
Available	100	66	28800	14400
Utilization (%)	0	0	~0	~0

Detail

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_CTRL_AWVALID	in	1	s_axi	CTRL	scalar
s_axi_CTRL_AWREADY	out	1	s_axi	CTRL	scalar
s_axi_CTRL_AWADDR	in	6	s_axi	CTRL	scalar
s_axi_CTRL_WVALID	in	1	s_axi	CTRL	scalar
s_axi_CTRL_WREADY	out	1	s_axi	CTRL	scalar
s_axi_CTRL_WDATA	in	32	s_axi	CTRL	scalar
s_axi_CTRL_WSTRB	in	4	s_axi	CTRL	scalar
s_axi_CTRL_ARVALID	in	1	s_axi	CTRL	scalar
s_axi_CTRL_ARREADY	out	1	s_axi	CTRL	scalar
s_axi_CTRL_ARADDR	in	6	s_axi	CTRL	scalar
s_axi_CTRL_RVALID	out	1	s_axi	CTRL	scalar
s_axi_CTRL_RREADY	in	1	s_axi	CTRL	scalar
s_axi_CTRL_RDATA	out	32	s_axi	CTRL	scalar
s_axi_CTRL_RRESP	out	2	s_axi	CTRL	scalar
s_axi_CTRL_BVALID	out	1	s_axi	CTRL	scalar
s_axi_CTRL_BREADY	in	1	s_axi	CTRL	scalar
s_axi_CTRL_BRESP	out	2	s_axi	CTRL	scalar
LoadValCnt_r_V	out	16	ap_none	LoadValCnt_r_V	pointer
EN_r_V	out	1	ap_none	EN_r_V	pointer
Rst_r_V	out	1	ap_none	Rst_r_V	pointer
ap_clk	in	1	ap_ctrl_hs	PWM_CTRL	return value
ap_rst_n	in	1	ap_ctrl_hs	PWM_CTRL	return value
interrupt	out	1	ap_ctrl_hs	PWM_CTRL	return value

Рисунок 27 – Отчет после синтеза

После синтеза запускаем ко-симуляцию. Нажимаем соответствующую кнопку на панели инструментов, в открывшемся окне оставляем все как есть (Рисунок 28). Если в «Dump Trace» выбрать «all», то можно будет посмотреть временные диаграммы в Vivado.

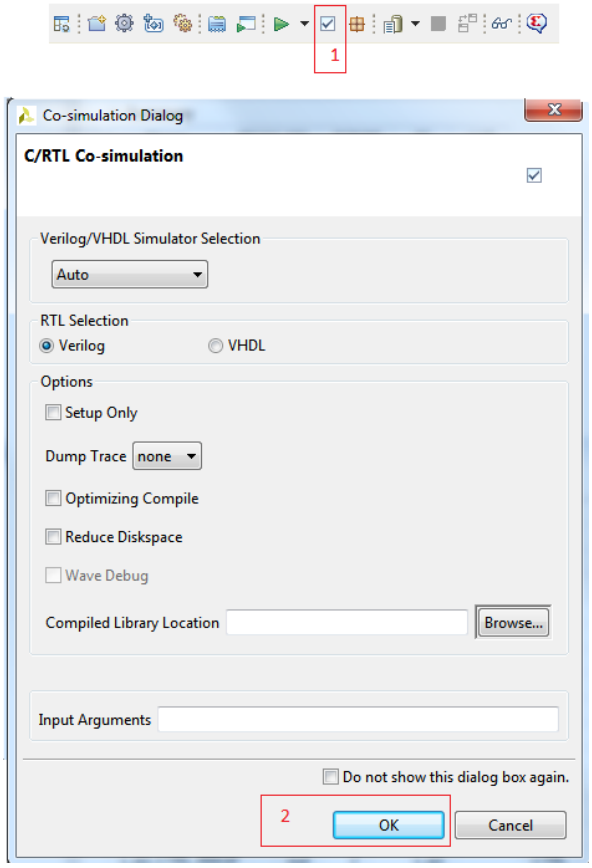

Рисунок 28 – Запуск ко-симуляции

После того, как ко-симуляция закончится, автоматически откроется отчет. Не пугаемся того, что у нас в нём везде нули. Раздел «Latency» показывает, какое количество периодов тактовой частоты необходимо функции для вычисления всех выходных значений – но у нас ничего не вычисляется, а просто значение регистра, задаваемого через AXI, присваивается выходному сигналу. Раздел «Interval» показывает, какое количество периодов тактовой частоты необходимо функции, чтобы она могла принять новые данные на вход.

Cosimulation Report for 'PWM_CTRL'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	0	0	0	0	0	0

Export the report(.html) using the Export Wizard

Рисунок 29 – Результат ко-симуляции функции PWM_CTRL

Осталось экспортировать IP. Для этого нажимаем на панели инструментов «Export RTL». В появившемся окне можно указать настройки экспорта и выполнить синтез средствами Vivado, что даст более точную оценку параметров IP, но и займет больше времени (Рисунок 32).

Сделаем активным решение `PWM`, дважды щёлкнув по нему в Проводнике. Выберем функцию `PWM` для синтеза (как мы это делали, можно посмотреть тут: Рисунок 20). Если у вас свой проект или вы создавали проект первым способ, то нужно в настройках решения указать, что нам нужен сброс с низким активным уровнем (Рисунок 25). Запускаем синтез, а затем и ко-симуляцию с опцией «Dump Trace all».

Cosimulation Report for 'PWM'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	0	0	0	1	1	1

Export the report(.html) using the Export Wizard

Рисунок 30 – Результат ко-симуляции функции PWM

Давайте теперь посмотрим временные диаграммы. Для этого жмем на панели инструментов кнопку «Open Wave Viewer».

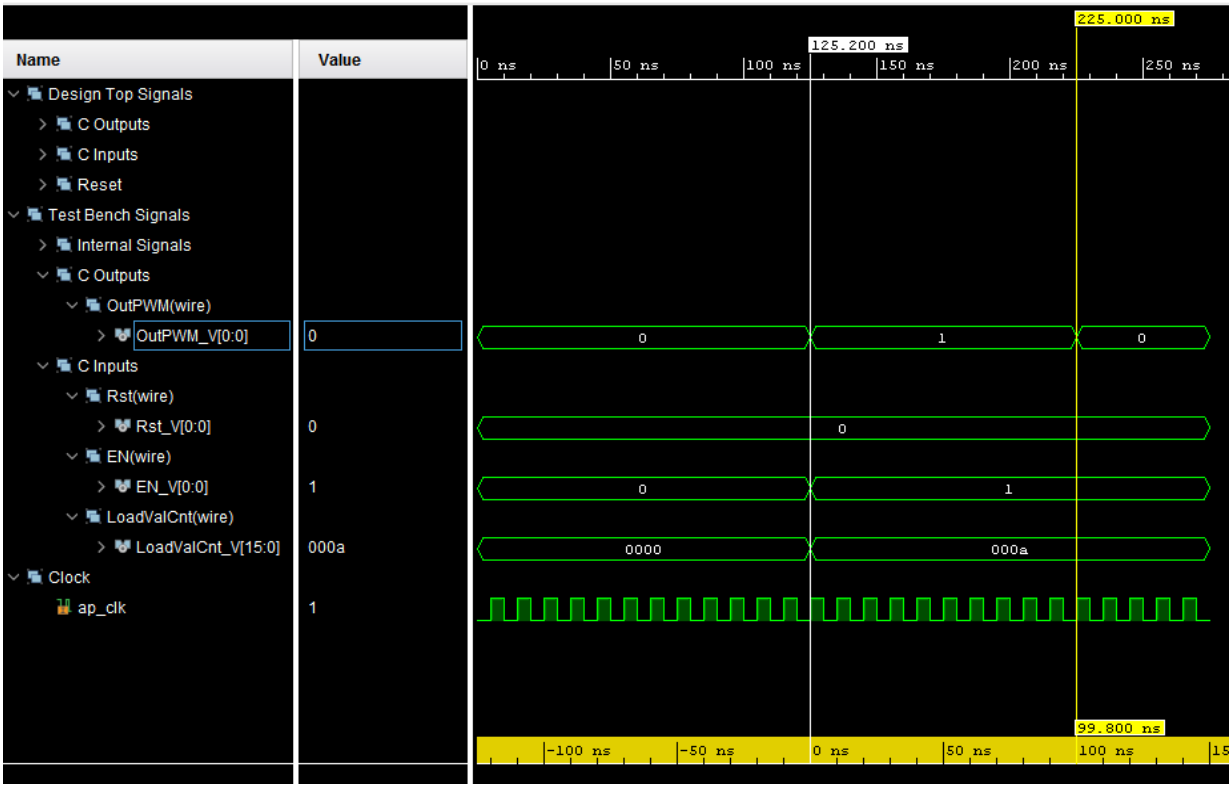

Рисунок 31 – Временные диаграммы для IP PWM

Экспортируем IP `PWM`; тут все точно так же, как с прошлым IP-блоком (Рисунок 32).

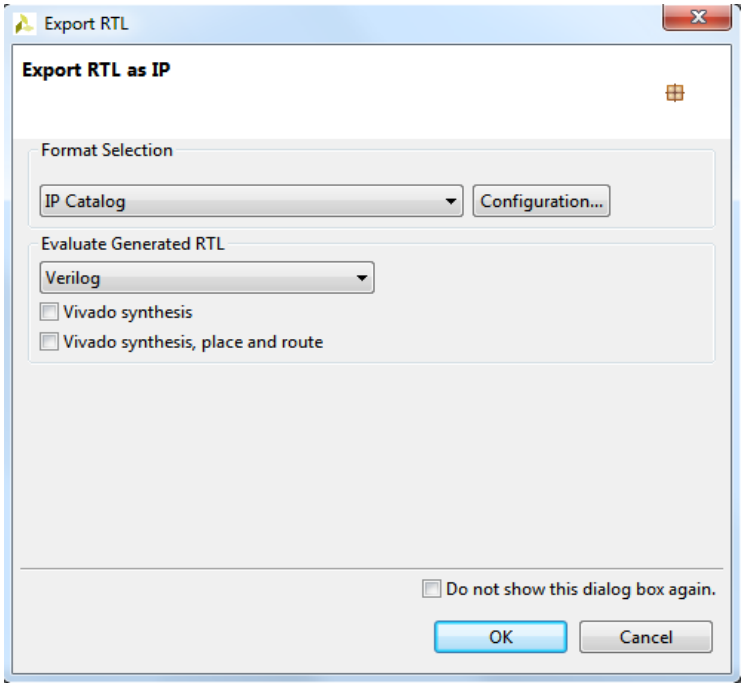

Рисунок 32 – Экспорт IP

Следующим шагом является интеграция этих IP в блок-дизайн в Vivado и их тестирование с помощью SDK в standalone-режиме.

Продолжение следует.

Список литературы

1. Vivado High-Level Synthesis
2. Осваиваем Zynq-7000s с бесплатной отладкой
3. UG871. Vivado Design Suite Tutorial: High-Level Synthesis
4. UG902. Vivado Design Suite UserGuide: High-Level Synthesis
5. Xilinx Zynq-7000 SoC
6. Разработка софт-процессорной системы на базе MicroBlaze (цикл статей)