

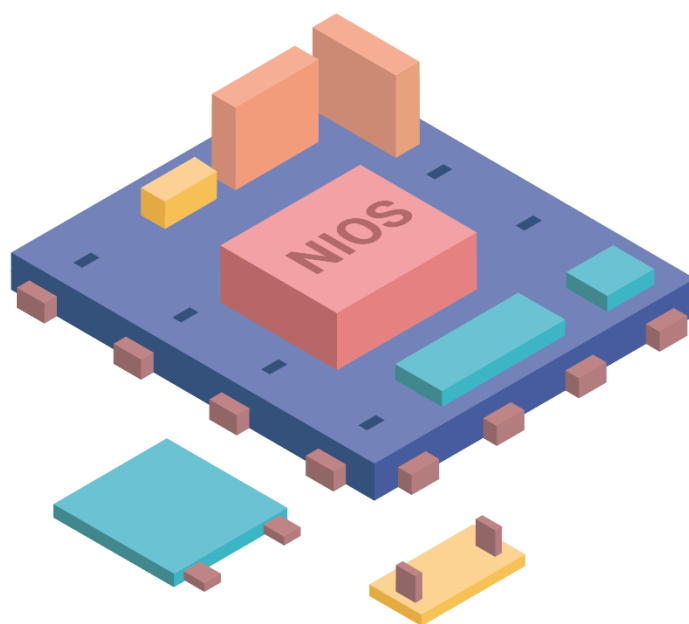


УНИВЕРСИТЕТ ИТМО

Д.С. Смирнов, И.Г. Дейнека, А.С. Алейник, И.А. Шарков

**ОСНОВЫ РАЗРАБОТКИ ВСТРАИВАЕМЫХ СИСТЕМ НА ПЛИС  
С ИСПОЛЬЗОВАНИЕМ ПРОЦЕССОРА NIOS II®**

Учебное пособие



Санкт-Петербург

2019

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ  
УНИВЕРСИТЕТ ИТМО**

**Д.С. Смирнов, И.Г. Дейнека, А.С. Алейник, И.А. Шарков**

**ОСНОВЫ РАЗРАБОТКИ ВСТРАИВАЕМЫХ СИСТЕМ НА ПЛИС  
С ИСПОЛЬЗОВАНИЕМ ПРОЦЕССОРА NIOS II®**

Учебное пособие

РЕКОМЕНДОВАНО К ИСПОЛЬЗОВАНИЮ В УНИВЕРСИТЕТЕ ИТМО  
по направлению подготовки (специальностям) 16.04.01 Техническая физика в  
качестве учебного пособия для реализации основных профессиональных  
образовательных программ высшего образования магистратуры



**УНИВЕРСИТЕТ ИТМО**

**Санкт-Петербург**

**2019**

Д.С. Смирнов, И.Г. Дейнека, А.С. Алейник, И.А. Шарков. Основы разработки встраиваемых систем на ПЛИС с использованием процессора NIOS® II. Учебное пособие. – СПб: Университет ИТМО, 2019. – 95 с.

**Рецензент:**

Китаев Юрий Васильевич, кандидат технических наук, Университет ИТМО, тьютор

Учебное пособие представляет собой практическое руководство по созданию встраиваемых приложений на базе программируемых логических интегральных схем – ПЛИС. За основу взяты микросхемы программируемой логики фирмы Intel® FPGA, для описания выбран микропроцессор с программным ядром NIOS II. Подробно рассмотрен процесс реализации процессорной системы, объяснено функциональное назначение блоков архитектуры микропроцессора и файлов, генерируемых на разных этапах компиляции. Значительное внимание уделено способу подачи материала, повествование ведется последовательно от начала проекта до практической реализации на отладочной плате.

Исходный код и рисунки в высоком разрешении можно найти по ссылке: [http://sf.ifmo.ru/ru/niosii\\_textbook](http://sf.ifmo.ru/ru/niosii_textbook).

Учебное пособие рекомендовано для магистров, обучающихся по направлениям 16.04.01 «Техническая физика» (образовательная программа «Световодная фотоника») и изучающим курс «Программируемая электроника на ПЛИС». Кроме того, оно может быть полезно для студентов, инженеров и научных работников, занимающихся в области проектирования встраиваемых систем.

Intel, Altera, Nios, Quartus и Cyclone являются торговыми марками компании Intel Corporation или ее дочерними компаниями в США и/или других странах.

Arm и Cortex являются торговыми марками компании Arm Limited или ее дочерними компаниями в США и/или других странах.



УНИВЕРСИТЕТ ИТМО

© Университет ИТМО 2019

© Д.С. Смирнов, И.Г. Дейнека, А.С. Алейник, И.А. Шарков 2019

## Содержание

<b>ВВЕДЕНИЕ</b>	<b>5</b>
<b>1. ПРОЦЕССОР С ПРОГРАММНЫМ ЯДРОМ В ПЛИС. ПРОЦЕССОРНАЯ СИСТЕМА НА БАЗЕ NIOS II.</b>	<b>7</b>
1.1. Программируемая логика и процессорные ядра. Общие тенденции	7
1.2. Особенности процессора с программным ядром. Процессор Nios II	10
<b>2. ЗНАКОМСТВО С ПРОГРАММНЫМ ОБЕСПЕЧЕНИЕМ И РЕАЛИЗАЦИЯ ПЕРВОГО ПРОЕКТА С ИСПОЛЬЗОВАНИЕМ NIOS II GEN2®</b>	<b>13</b>
2.1. Структура процесса разработки	13
2.1.1. Разработка аппаратной части	14
2.1.2. Разработка программной части	14
2.1.3. Постановка задачи для первого проекта	15
2.2. Разработка аппаратной части проекта <code>nios_load1</code>	15
2.2.1. Создание проекта в Quartus	16
2.2.2. Сборка процессорной системы <code>nios_load1</code> в Platform Designer	18
2.2.3. Реализация файла верхнего уровня	26
2.2.4. Настройка периферии в Pin Planner	29
2.2.5. Компиляция проекта и прошивка платы	31
2.3. Разработка программной части	33
2.3.1. Генерация BSP	33
2.3.2. Реализация приложения	34
2.3.3. Компиляция и запуск программы на плате	35
2.4. Обзор реализации программы для системы на основе Nios II	37
2.5. Листинг	43
<b>3. ОБЗОР ПРОЦЕССОРА NIOS II GEN2</b>	<b>44</b>
3.1. Введение	44
3.1.1. Концепция настраиваемого ядра софт-процессора	45
3.2. Архитектура процессора	46
3.2.1. Файл регистров	46

3.2.2. АЛУ	47
3.2.3. Контроллеры исключений и прерываний	48
3.2.4. Память и организация ввода/вывода	48
3.2.5. Блок отладки JTAG	52
<b>4. ГОТОВЫЕ РЕШЕНИЯ ПЕРИФЕРИИ ДЛЯ NIOS II GEN2</b>	<b>54</b>
4.1. Введение	54
4.2. On-Chip Memory	55
4.3. PIO	56
4.4. Interval Timer	58
4.5. JTAG UART	60
4.6. SDRAM Controller	63
<b>5. МОДИФИКАЦИЯ ПРОЕКТА nios_load С ИСПОЛЬЗОВАНИЕМ IP-ЯДЕР</b>	<b>66</b>
5.1. Разработка аппаратной части	66
5.2. Разработка программной части	70
5.2.1. Процедуры драйверов	70
5.2.2. HAL	71
5.2.3. BSP	72
5.2.4. Приложение nios_load_hal	75
<b>6. ПРЕРЫВАНИЯ и ISR</b>	<b>85</b>
6.1. Введение	85
6.2. Прерывания в Nios II	85
6.3. Обработка прерываний в рамках HAL	87
6.4. Реализация прерываний в проекте nios_load2	88
<b>Заключение</b>	<b>92</b>
<b>Ссылки</b>	<b>93</b>
<b>Приложение</b>	<b>94</b>

## ВВЕДЕНИЕ

В настоящее время продолжается ускоренное развитие цифровых систем. Микроконтроллеры, процессоры, микросхемы программируемой логики уже используются повсеместно, а увеличение интереса к созданию и применению интеллектуальных систем в огромном количестве направлений и областей науки и техники обуславливает рост рынка цифровых устройств. Однако растут также требования к количеству специалистов, способных осуществлять проектирование печатных плат и создавать программное обеспечение для цифровых микросхем.

В современном мире имеет место тенденция к диверсификации специальностей. Это, в том числе, справедливо и для области программирования, в частности, встроенных систем. Например, процесс обучения проектированию систем с FPGA после изучения фундаментальных вопросов разделяется на набор специальностей – Hardware Development, Embedded Hardware Development, Software Development, Digital Signal Processing, Interfaces Development. Освоение столь широкого спектра задач требует огромного количества специализированных знаний, но понимание основ, базовых понятий и терминов, относящихся к области цифровых систем и архитектуры компьютера, является общим и необходимым фактором для того, чтобы стать профессионалом.

Данное учебное пособие подходит как для тех, кто делает свои первые шаги в работе с программируемой электроникой, так и для тех, кто имеет опыт работы с ПЛИС и хочет дополнить их знаниями и навыками работы с микропроцессором с программным ядром. Описание основных элементов составлено таким образом, чтобы дать ответ на все «почему», а практические примеры позволяют закрепить полученные знания. Авторы предлагают последовательность изучения материала, в которой основы закладываются при работе с программируемой логикой, а уже затем из блоков, описанных на языке описания аппаратуры (*Hardware Description Language – HDL*), собираются микропроцессорные ядра и периферия. Поэтому в рамках учебного пособия в качестве аппаратной платформы были выбраны программируемые логические интегральные схемы, ПЛИС.

На сегодняшний день на рынке цифровых микросхем представлены различные варианты процессорных ядер, в том числе микропроцессоров с программным ядром. В данном учебном пособии рассматривается реализация процессорной системы на базе софт-процессора Nios II<sup>®</sup> с использованием средств разработки Intel FPGA. Настоящее пособие состоит из семи разделов.

**В первом разделе** мы обозначим некоторые тенденции развития процессорных систем и микросхем программируемой логики, опишем особенности выбранной аппаратной платформы и процессорной системы.

**Во втором разделе** пособия мы познакомимся со средствами разработки встраиваемых систем на базе ПЛИС фирмы Intel FPGA и соберем первый базовый проект.

**В третьем разделе** мы рассмотрим ядро процессора Nios II: его преимущества, конфигурации и возможности настройки.

**В четвертом разделе** мы познакомимся с базовыми блоками периферии – их функционалом, возможностями.

**В пятом разделе** мы воспользуемся блоками, описанными в разделе 4, и соберем улучшенную версию базового проекта.

**В шестом разделе** мы познакомимся с принципами прерываний и воспользуемся встроенными возможностями ядра Nios II для оптимизации программы с использованием прерываний.

Принципы построения в ПЛИС систем на основе soft-процессоров остаются неизменными, поэтому знания и навыки, полученные в результате изучения данного пособия, могут быть применены на практике для разных микросхем и при использовании других САПР, не рассмотренных в тексте издания. Авторы надеются данным учебным пособием и его возможными продолжениями решить задачу создания качественного и понятного учебного материала по современным ПЛИС с практическими примерами.

Авторы постарались прямо в тексте ответить на большое количество потенциальных вопросов читателей, включив в текст дополнительные сноски и выделив важные особенности. Часто в тексте в скобках указывается альтернативный перевод того или иного термина. Авторы решились использовать сленг, слова, которые приняты у инженеров, чтобы приблизить читателя к специфике программирования ПЛИС. Мы считаем, что иногда такая терминология очень удачна для описания понятий и адаптации к русскому языку англоязычных слов.

Для лучшего освоения материала методического пособия в качестве предварительных требования к читателям желательно обладать базовыми знаниями в программировании (например, знания языка Си) и микропроцессорной технике.

Методические рекомендации к использованию. Учебное пособие является частью дисциплины «Программируемая электроника на ПЛИС». При написании курсового проекта в рамках данной дисциплины студенту рекомендуется ориентироваться на последовательность, изложенную в данном пособии. Выполнение всех шагов разработки согласно структуре пособия позволит читателю развить компетенции в разработке как аппаратной, так и программной реализации встраиваемых систем на ПЛИС.

Авторы выражают благодарность Китаеву Юрию Васильевичу за экспертную оценку и поддержку, Шуклину Филиппу Александровичу за помощь в проверке и корректировке описанного материала и Гужвиевой Марии Викторовне за создание логотипа для обложки.

# 1. ПРОЦЕССОР С ПРОГРАММНЫМ ЯДРОМ В ПЛИС. ПРОЦЕССОРНАЯ СИСТЕМА НА БАЗЕ NIOS II.

## 1.1. Программируемая логика и процессорные ядра. Общие тенденции

Программируемая логическая интегральная схема (ПЛИС), выбранная в качестве аппаратной платформы для реализации процессорного ядра, представляет собой своего рода конструктор для микроэлектроники. Конфигурируя программируемую логику ПЛИС, можно создавать самые разные электронные схемы – от простых комбинаторных схем, например, сумматора или мультиплексора, до сложных систем управления роботизированными комплексами, нейросетевых ускорителей вычислений для поисковых интернет систем и блоков управления автономными транспортными средствами.

Часто можно встретить понятие «отладочная плата» (англ. *Development board*) – платформа, имеющая в своём составе одну или несколько ключевых микросхем (ПЛИС, микроконтроллер и другие), используемая для того, чтобы итеративным путём отработать схемотехническое и программные решения. Например, с помощью платы STM32F429 Discovery (рисунок 1.1(а)) можно познакомиться с архитектурой контроллеров STM32 и отладить работу с LCD экраном.

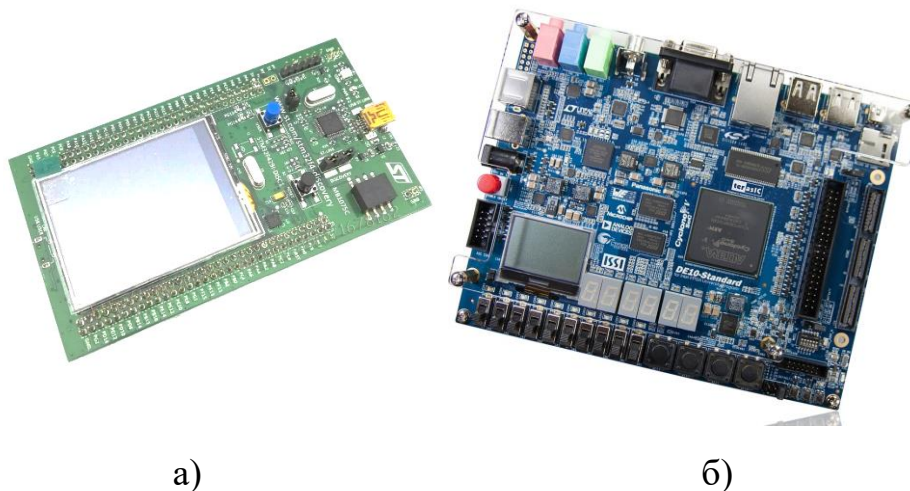


Рисунок 1.1 Отладочные платы: а - STM32F429 Discovery; б - DE10-Standard

На таких платах с FPGA (рисунок 1.1(б)) часто отрабатывается, или «прототипируется», архитектура процессорных ядер перед его запуском в серийное производство. Мы поступим похожим образом – мы будем использовать программируемую логику ПЛИС для того, чтобы собрать из её элементов процессорную систему.



Что же такое процессор? Дадим несколько определений. Сначала определим процессор как логическую схему, состоящую из набора функциональных блоков, совокупность которых позволяет в конечном итоге выполнять команды управления компьютерной системой. Другое определение возьмём из Википедии: «Процессор – это электронная схема, выполняющая инструкции компьютерной программы, производя арифметические, логические, управляющие операции и операции ввода/вывода, определяемые инструкциями». Процессор обычно принято считать главной микросхемой, часто можно встретить название Центральный Процессорный Элемент (англ. *Central Processing Unit*), но в современных реалиях это не всегда верно, так как процессор может быть лишь малой частью сложной системы и использоваться, например, лишь для реализации ТСП/IP стека.

Процессор обладает рядом ключевых преимуществ, обуславливающих его повсеместное применение. Это прежде всего возможность, всего лишь модифицировав содержимое памяти процессора (поменяв последовательность и набор инструкций), полностью изменить выполняемый функционал – только что процессор декодировал архивный файл, и вот он уже играет против вас в шахматы. То есть, архитектура процессора не меняется при изменении задачи – меняется лишь содержимое памяти. Второе преимущество, происходящее из первого, – возможность использования высокоуровневых языков программирования. Ведь несмотря на особенности архитектуры процессоров и их разнообразие, ключевые блоки – регистровый файл, блоки памяти, счётчик и декодер инструкций, арифметико-логическое устройство – присутствуют во всех процессорах. А это значит, что, изменяя и дополняя компилятор, можно создавать возможность написания кода, который будет работать на разных процессорах, при этом нет необходимости глубоко понимать архитектуру процессора.

Вышеописанные преимущества позволяют понять, почему процессоры так широко распространены и почему так много программистов, пишущих программы на языках высокого уровня. Однако на сегодняшний день имеет место набор тенденций, заключающихся в том, что:

- во-первых, для увеличения производительности всё чаще требуется применять специализированное (англ. *custom*), а не общее (англ. *generic*) решение. Данная тенденция просматривается во многих сферах. В наше время все чаще можно слышать об «индивидуальном подходе» и «учете чьих бы то ни было особенностей». В сфере программирования и цифровой схемотехники можно говорить о создании и использовании «специализированного аппаратного обеспечения» (англ. *dedicated hardware*) для эффективного решения конкретной задачи. Наиболее

показательными примерами могут стать специализированные инструкции (англ. *custom instructions*), призванные увеличивать производительность процессоров за счёт создаваемых под конкретную задачу ускорителей (англ. *accelerators*). На момент написания данного пособия корпорация Intel вела работы по созданию среды, упрощающей работу с микросхемой, объединяющей процессоры Xeon® и ПЛИС – *Intel Acceleration Stack*. Данная среда призвана как раз обеспечить возможность создания аппаратных ускорителей для функций, выполняющихся на процессоре;

- во-вторых, наблюдается активный рост рынка встроенных систем и приложений для них, и всё важнее становится вес такого фактора, как время выхода продукта на рынок (англ. *time to market*). В быстро изменяющемся мире все важнее бывает первым заявить о себе, первым продемонстрировать решение, даже в недоделанном, предварительном виде. Для задач, требующих создания специализированных микросхем (англ. *application-specific standard product, ASIC*), время, потраченное на разработку архитектуры и топологии, изготовление чипов, может стать критическим. Возможность быстрого прототипирования схемы делает программируемую логику всё более востребованной;
- в-третьих, имеет место взаимопроникновение аппаратных и программных решений. Процессорные ядра с фиксированной архитектурой объединяются с программируемой логикой, как это сделал Intel. Аппаратные, еще недавно недоступные для программируемой логики процессоры Arm® Cortex® получили свою HDL (англ. *hardware description language*) версию, то есть теперь доступны для имплементации в ПЛИС. То же самое справедливо и для процессоров MIPS. Программисты ПЛИС всё чаще сталкиваются с необходимостью работы с процессорными системами и, как следствие, важностью навыков высокоуровневого программирования. В свою очередь, программисты, пишущие на языках высокого уровня, желающие увеличить производительность своих алгоритмов, даже при наличии таких программных пакетов, как высокоуровневый синтез (англ. *high level synthesis, HLS*), должны понимать архитектуру и возможности программируемой логики, разбираться в языках описания аппаратуры.

Таким образом, развитие цифровых технологий и программирования потребует от программистов и инженеров-схемотехников стать более универсальными и в определенном смысле размоет границу между ними. Можно сделать вывод о том, что понимание архитектуры микросхемы – это важный и полезный навык, позволяющий оптимальнее решить поставленную задачу. А знание языков высокого уровня – теперь необходимость и для инженеров-схемотехников.

## 1.2. Особенности процессора с программным ядром. Процессор Nios II

Итак, мы будем реализовывать процессор на ПЛИС. Поскольку это будет процессор с программным ядром, имеет смысл предварительно определить его особенности, сравнив с ASIC-процессором, то есть с процессором с аппаратным ядром. В таблице 1.1 представлены особенности обоих решений.

Таблица 1.1 Сравнительная таблица, описывающая преимущества и недостатки двух видов реализации процессорного ядра

Параметр	Процессор с аппаратным ядром (Hard-core)	Процессор с программным ядром (Soft-core)
Максимальная тактовая частота работы	Единицы ГГц	Сотни МГц
Место, занимаемое на кристалле	Меньше, чем в случае программного решения, из-за использования оптимального количества и типа логических элементов	Больше, чем в случае аппаратного решения, из-за избыточности программируемой логики для решения специфической задачи (в частности имплементации процессорного ядра)
Гибкость, возможность настройки	Ввиду того, что архитектура и расположение процессорного ядра на кристалле фиксирована, возможность настройки ограничена заложенной заранее избыточностью.	Широкие возможности настройки параметров под конкретную задачу. Кроме того, появляется возможность имплементации нескольких процессорных ядер, и их расположение в том месте кристалла, где это необходимо
Возможность обновления и повторное использование	В устройствах, где используются процессоры с аппаратными ядрами, обновление процессора невозможно. По этой же причине повторное	В устройствах, где используются процессоры с программными ядрами, существует возможность полностью обновить систему, заменив

	использование устройства для других целей затруднено.	процессор на другой, или отказаться от процессора в пользу системы на программируемой логике (HDL). Так же подобные устройства могут быть повторно использованы для альтернативных задач посредством полной переработки системы.
--	---	---

Из таблицы становится понятным, что оба варианта реализации процессоров имеют свои преимущества и недостатки, а выбор в пользу того или иного решения должен делаться исходя из специфики поставленной задачи.

На программируемой логике существует возможность как собрать написанное самостоятельно на языке HDL программное ядро, так и воспользоваться готовыми программными ядрами. Производитель микросхем программируемой логики Altera, некоторое время назад перешедшая под всемирно известный бренд Intel, в течение продолжительного времени поддерживает свой 32-разрядный процессор Nios II, считающийся наиболее широко используемым процессором с программным ядром в индустрии ПЛИС<sup>1</sup>.

Процессор Nios II имеет 32-разрядную RISC (англ. *reduced instruction set computer*) микроархитектуру, то есть обладает архитектурой с ограниченным набором команд<sup>2</sup>. К его ключевым особенностям следует отнести две полноценные значительно отличающиеся друг от друга версии («сборки») – Nios II Fast и Nios II Ecosystem, возможность работы в широком диапазоне тактовых частот (более 400 МГц), аппаратная (и программная со стороны интегрированной среды разработки) поддержка специализированных инструкций для аппаратного ускорения (до 256), поддержка векторного контроллера прерываний (до 32), возможность подключения быстрой памяти из ресурсов программируемой логики и многое другое.

Для программирования под Nios II предоставляется встроенная среда разработки (англ. *Embedded Design Suite*<sup>®</sup>, *IDE*). Она представляет собой пользовательский интерфейс для работы с кодом и отладки программы и, кроме

<sup>1</sup> Источник: <https://www.intel.co.uk/content/www/uk/en/products/programmable/processor/nios-ii.html>

<sup>2</sup> Подробнее: <https://ru.wikipedia.org/wiki/RISC>

всего прочего, включает в себя поддержку операционной системы реального времени.

В следующей главе мы перейдём к практической реализации процессора с программным ядром на ПЛИС. Мы построим аппаратную часть системы, в которой центральным элементом будет процессор Nios II.

## 2. ЗНАКОМСТВО С ПРОГРАММНЫМ ОБЕСПЕЧЕНИЕМ И РЕАЛИЗАЦИЯ ПЕРВОГО ПРОЕКТА С ИСПОЛЬЗОВАНИЕМ NIOS II GEN2®

Система на основе процессора Nios II состоит из двух отдельных частей - аппаратной и программной. Встроенный в Quartus инструмент *Platform Designer*® используется для реализации аппаратной части: настройка конфигурации процессора и периферии. Платформа *Nios II EDS* используется для работы с программной частью проекта. В этой главе мы соберем базовую систему для того, чтобы продемонстрировать процесс разработки аппаратной части и особенности работы с программной частью проекта: рассмотрим последовательность и специфику разработки приложений для собираемой аппаратной системы.

### 2.1. Структура процесса разработки

Диаграмма процесса разработки системы на основе Nios II показана на рисунке 2.1.

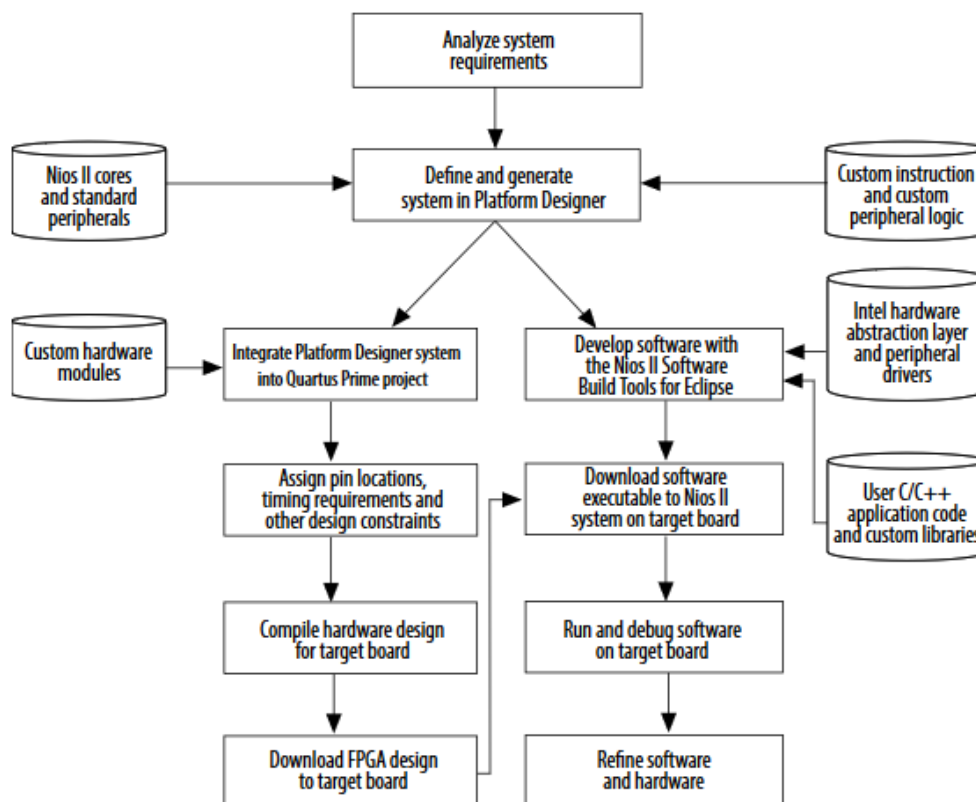


Рисунок 2.1 Схема процесса разработки на Nios II<sup>3</sup>

<sup>3</sup>Взято из: <https://www.intel.com/content/www/us/en/programmable/documentation/iga1446487888057.html#sss1409047110026>

### 2.1.1. Разработка аппаратной части

При возрастании сложности разрабатываемых систем значительно увеличивается время, необходимое для реализации необходимого функционала (левая часть схемы). Одним из решений такой проблемы является использование готовых модулей в проекте. Подобные модули могут быть разработаны как самим разработчиком, так и сторонними программистами, и часто оформляются в виде интеллектуального продукта, выполненного по определённым стандартам, поэтому их принято называть *IP-ядрами (Intellectual Property cores)*, т.е. объектами интеллектуальной собственности. Такие ядра могут быть бесплатными, могут быть специализированными под определённого производителя, а могут стоить десятки тысяч долларов США.

Intel FPGA предоставляет обширный выбор таких модулей, часть из которых являются бесплатными и доступными всем пользователям. Для разработки систем на основе готовых модулей в Quartus доступен дополнительный инструмент – *Platform Designer*. Данный инструмент позволяет значительно сэкономить время, автоматизируя большую часть генерации логики межсоединений. По результатам сборки системы Platform Designer проверяет сборку на ошибки и генерирует HDL-коды для дальнейшей компиляции и файлы с расширением \*.qsys и \*.sopcinfo, хранящие в себе информацию о конфигурации системы. Эти файлы в дальнейшем используются для компиляции и генерации файлов для программной части проекта.

### 2.1.2. Разработка программной части

Intel FPGA предоставляет набор инструментов для разработки и реализации программной части – *Nios II Embedded Design Suite (EDS)*. Данный пакет содержит в себе все необходимое:

- *Nios II Software Build Tools for Eclipse* – набор инструментов для разработки программной части для Nios II с использованием графического интерфейса;
- *Инструменты GNU* (GCC компилятор, GDB отладчик) – поддержка инструментов GNU toolchain;
- Примеры и шаблоны с применением готовых решений для IP-ядер;
- Бесплатный доступ к использованию *Nichestack TCP/IP Network Stack*;
- Бесплатная ознакомительная версия распространённой операционной системы реального времени (RTOS) от Micrium – *MicroC/OS-II*

Данный набор позволяет быстро и эффективно разрабатывать приложения для процессоров Nios II.

### 2.1.3. Постановка задачи для первого проекта

Процесс разработки системы на основе процессора Nios II состоит из нескольких этапов. Для иллюстрации процесса разработки системы мы соберем небольшую систему управления светодиодами. Полоса из десяти светодиодов будет имитировать визуализацию процесса загрузки – светодиоды последовательно будут загораться один за другим, пока все они не начнут светиться, после чего они все погаснут, и так далее. Десять переключателей на плате будут регулировать скорость, с которой светодиоды будут зажигаться. Для подобной простой системы мы воспользуемся минимальной конфигурацией процессора Nios II - Economy.

Главной задачей этого проекта является ознакомление с программным обеспечением, необходимым для построения подобных и более сложных систем на базе Nios II. Основными шагами в разработке проекта будут:

#### 1. Создание аппаратного проекта в Quartus;

1.1. Сборка системы на основе Nios II и генерация HDL-кода с помощью Platform Designer;

1.2. Добавление системы в файл верхнего уровня проекта, подключение системы к другим модулям проекта и портам ПЛИС;

1.3. Компиляция проекта и конфигурация ПЛИС, ознаменовывающая собой завершение создания аппаратной части проекта;

#### 2. Генерация библиотеки BSP;

#### 3. Написание кода приложения на языке Си;

#### 4. Сборка программного проекта и запуск его на плате.

### 2.2. Разработка аппаратной части проекта `nios_load1`

Для реализации аппаратной части проекта необходимо выполнить следующие действия:

- Создать новый проект в среде разработки *Quartus*;
- Собрать простую систему на базе процессора Nios II с использованием инструмента *Platform Designer*:

- Добавить и сконфигурировать модули периферии, памяти, идентификатора системы и процессора;

- Выбрать корректные подключения в матрице межсоединений и экспортировать из системы необходимые порты;

- Определить вектора сброса и исключений;

- Определить базовые адреса;

- Сгенерировать HDL-файлы и bsf-файл;

- Создать файл верхнего уровня, в котором будет присутствовать собранная система, и подключить порты системы к портам ПЛИС;



- Настроить периферию с помощью инструмента *Pin Planner*;
- Скомпилировать проект, сконфигурировать (на сленге – «прошить») ПЛИС.

### 2.2.1. Создание проекта в Quartus

Чтобы начать сборку системы на базе Nios II, необходимо создать проект, в состав которого будет включен процессорный блок. Для этого запустите среду Quartus и создайте пустой проект выбором в верхней левой панели **File > New Project Wizard**. В открывшемся окне **Introduction** появится описание последовательности создания проекта (рисунок 2.2). Нажмите **Next >**.

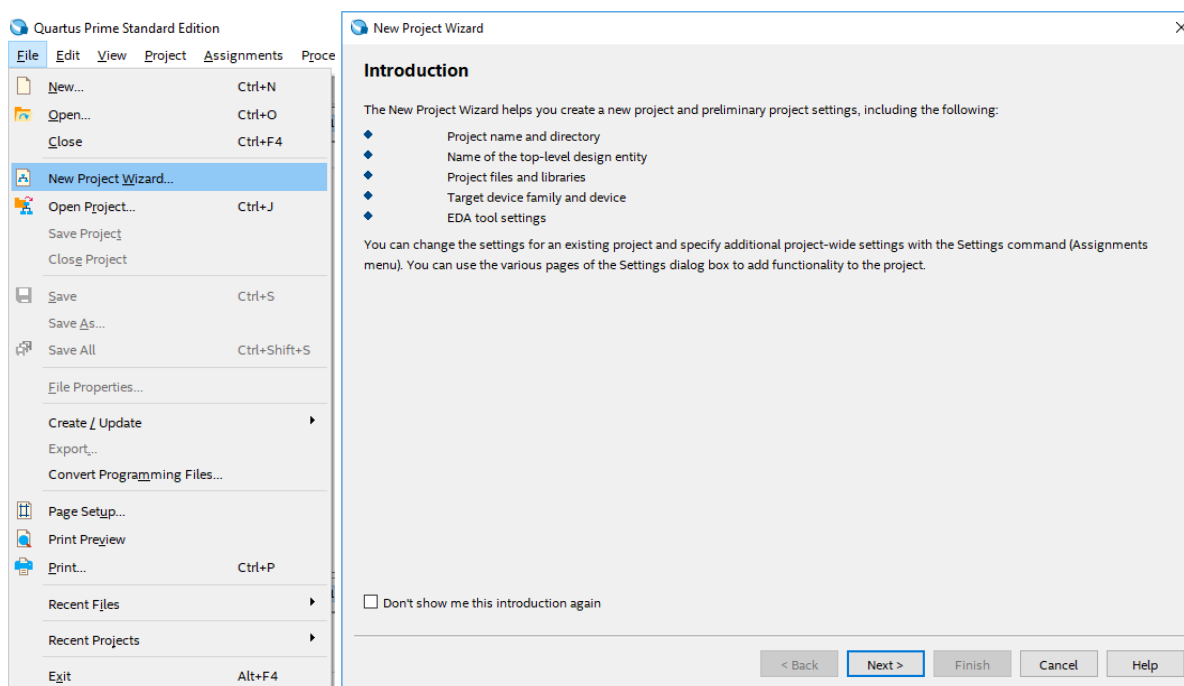


Рисунок 2.2 Первые шаги по созданию проекта в среде Quartus

Следующим шагом **Directory, Name, Top-level Entity** необходимо указать директорию, в которой будет храниться проект и все его файлы, и определить название проекта (рисунок 2.3). По умолчанию считается, что файл верхнего уровня имеет то же название, что и сам проект.

---

**!** Путь к директории проекта не должен содержать пробелы, буквы русского алфавита и различные спецсимволы. По умолчанию предлагается адрес, по которому расположена директория самой среды разработки – этот адрес **●** идеально подходит по всем требованиям. Также обратите внимание, что название файла верхнего уровня чувствительно к регистру.

---

Данный проект реализуется с нуля, поэтому в следующем шаге **Project type** выбирается пункт **Empty project**. По этой же причине шаг **Add files** пропускается. В следующем шаге, **Family, Device & Board Settings**, необходимо определить, для

какого именно семейства ПЛИС разрабатывается проект (рисунок 2.4). В этом проекте используется плата **DE10-Standard** от тайваньского производителя **Terasic**, на которой установлен кристалл с артикулом (**part number**) **5CSXFC6D6F31C6N**. Данный чип относится к семейству **Cyclone V**, это необходимо указать в выпадающем списке пункта **Family**. В списке доступных устройств необходимо выбрать указанный кристалл.

---

! Если у Вас отсутствует возможность выбора семейства, Вам необходимо  
• скачать их с сайта **Intel** и добавить через меню **Tools > Install Devices**.

---

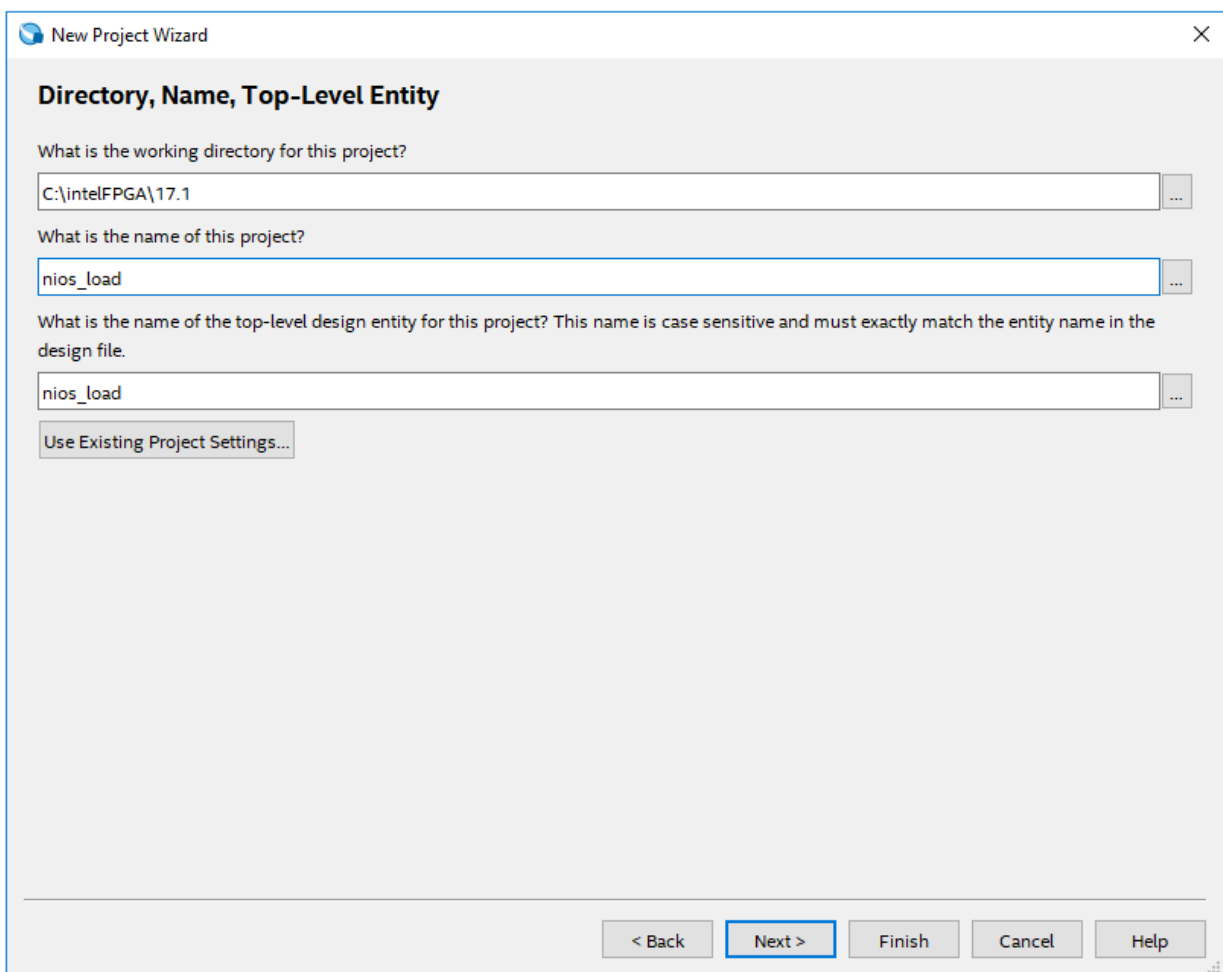


Рисунок 2.3 Окно выбора директории и названия для проекта и файла верхнего уровня

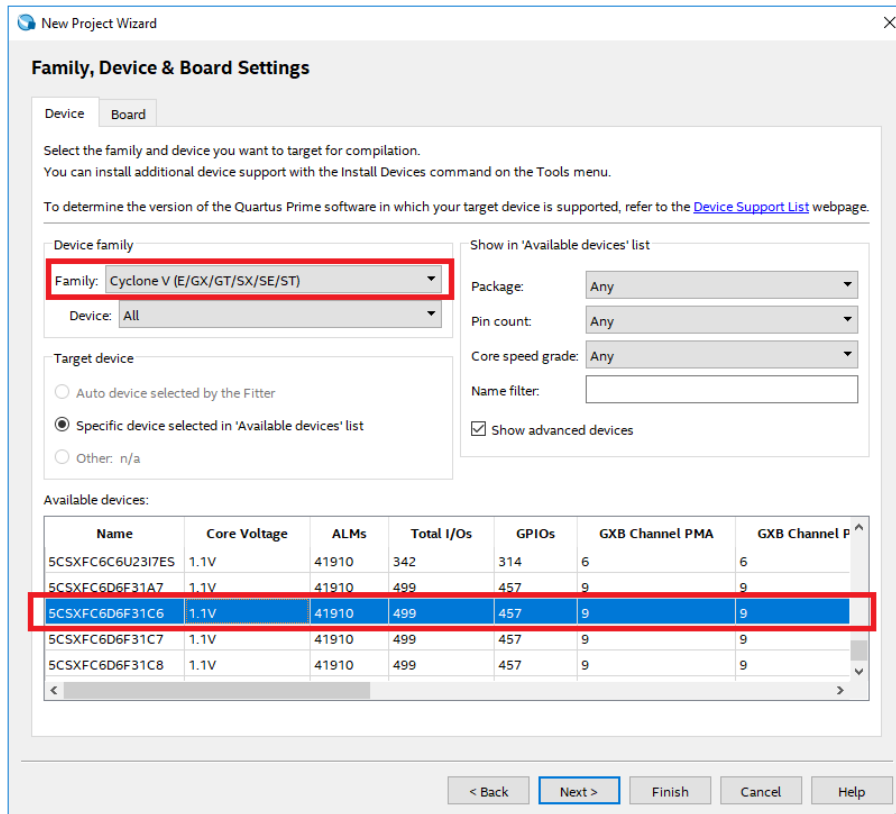


Рисунок 2.4 Выбор семейства (*family*) и устройства (*device*), для которого создается проект

Этих настроек достаточно для текущего проекта, поэтому можно нажать кнопку **Finish**. Теперь можно приступить к сборке системы на основе Nios II в среде Platform Designer.

### 2.2.2. Сборка процессорной системы `nios_load1` в Platform Designer

Запустить инструмент Platform Designer можно как нажав иконку  на панели инструментов, так и через меню **Tools > Platform Designer**. После инициализации откроется пустая система, в которой есть только источник тактового сигнала (рисунок 2.5). В левой части экрана находится набор тематически подобранных библиотек, **IP Catalog**, элементы которого мы будем добавлять в систему. На вкладке System Contents отображаются все добавленные в собираемую систему блоки, их межсоединения и различная информация, включая базовый адрес, приоритеты прерываний, экспортированные порты и пр. Приступим к сборке системы.

#### *Добавление процессора Nios II Gen2*

В окне IP Catalog выбираем категории **Processors and Peripherals > Embedded Processors** (или находим с помощью поиска) и выбираем компонент **Nios II**

**Processor**, нажимаем **Add**, чтобы добавить его в систему, появляется окно настройки процессора (рисунок 2.6).

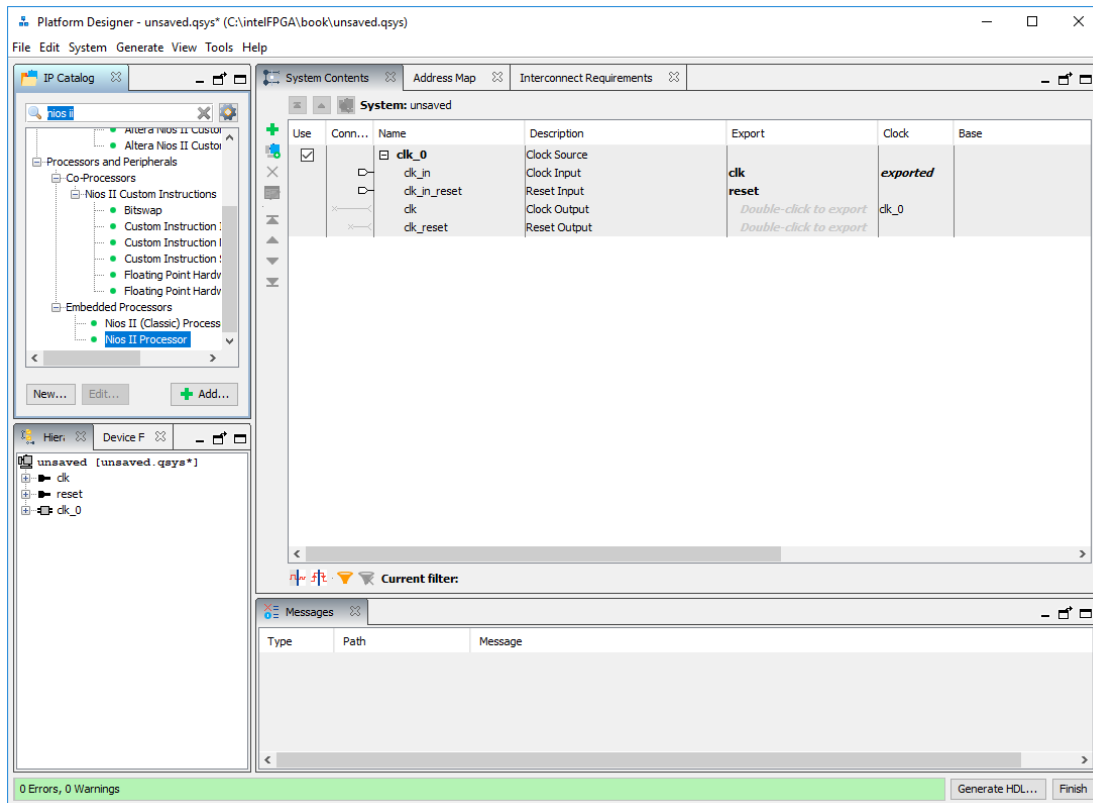


Рисунок 2.5 Окно инструмента Platform Designer

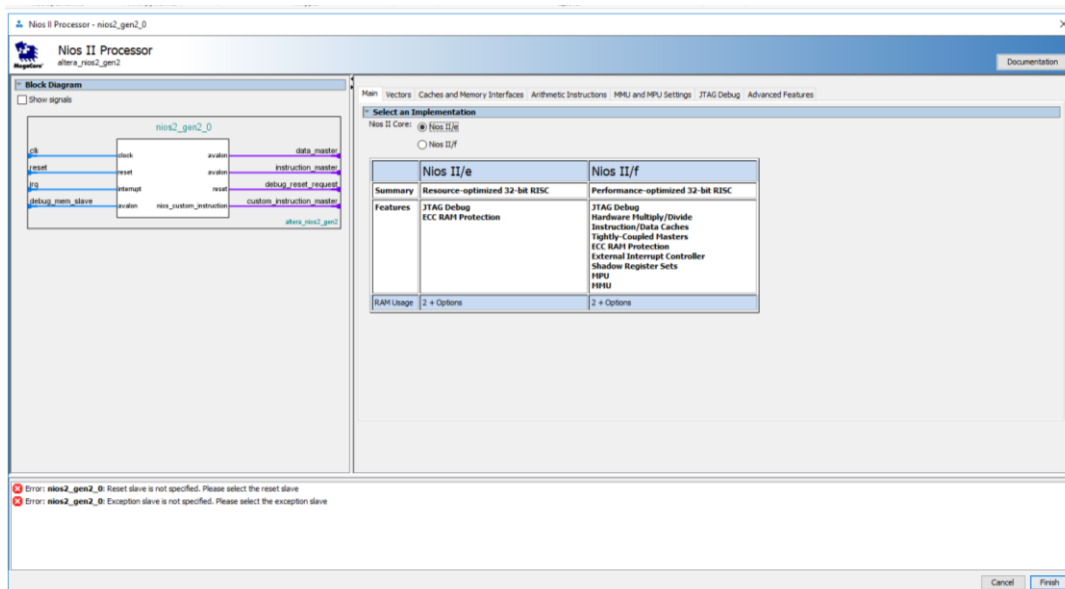


Рисунок 2.6 Начальное окно настройки процессора Nios II

Для текущего проекта необходима выбрать на вкладке **Main** конфигурацию ядра Nios II/e (подробнее о конфигурациях Nios – в разделе 3). На вкладке **Vectors** определяются вектора сброса и исключений, однако мы еще не добавили память в систему, поэтому заполним это поле позже. На вкладке **Cache and Memory Interfaces** настраиваются параметры кэша, на вкладке **Arithmetic Instructions** расположены настройки способов реализации арифметических операций в процессоре; функции на этих двух вкладках доступны только при использовании конфигурации Nios II/f. Аналогичная ситуация с настройками блоков управления и защиты памяти на вкладке **MMU and MPU Settings**. На вкладке **JTAG Debug** по умолчанию добавлен модуль отладки через интерфейс JTAG. Более продвинутые настройки доступны на вкладке **Advanced Featured** и не рассматриваются в рамках данного методического пособия. Нажимаем клавишу **Finish**, чтобы добавить блок в систему. Нажмите правой кнопкой мышки по имени блока и переименуйте его в сру.

### *Добавление модуля памяти*

К процессору Nios II можно подключить различные модули памяти, в данном проекте мы воспользуемся встроенной памятью (*on-chip memory*). В каталоге IP-ядер в категории **Basic Functions > On Chip Memory** выберем **On-Chip Memory (RAM or ROM)**. Наше приложение будет занимать немного места в памяти, в открывшемся окне настроек блока (рисунок 2.7) укажем размер с запасом – 20480 Байт. Остальные настройки сохраним по умолчанию. Нажмите правой кнопкой мышки по имени блока и переименуйте его в `onchip_mem`.

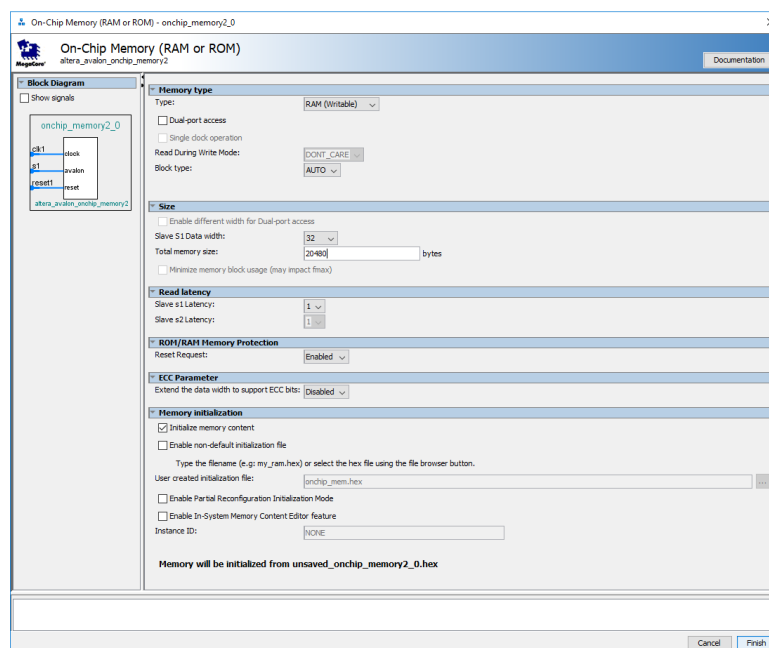


Рисунок 2.7 Окно настроек блока On-Chip Memory (RAM or ROM)

---

! В нижней части окна настроек находится поле «*Memory Initialization*», в котором можно выбрать и добавить hex-файл в качестве файла инициализации. В этом поле можно указать скомпилированное приложение, чтобы при запуске процессора программа мгновенно начала свое выполнение.

---

#### *Добавление модулей периферии*

В текущей системе в качестве периферии используются светодиоды, т.е. средства вывода информации, и переключатели, т.е. средства ввода информации. Для подключения данных элементов необходимо воспользоваться IP-ядром **PIO** (*Parallel Input/Output*), которое можно найти в категории **Processors and Peripherals > Peripherals**. В настройках в обоих случаях необходимо указать битность подключаемой шины и направление движения данных.

При настройке IP-ядра для переключателей (рисунок 2.8 слева) необходимо выбрать направление (**Direction**) **Input**. Другие настройки в текущей сборке останутся по умолчанию и будут рассмотрены позже. Нажмите правой кнопкой мышки по имени блока и переименуйте его в `switch`.

В случае настройки для светодиодов (рисунок 2.8 справа) выбирается направление **Output**. Остальные настройки остаются по умолчанию. В поле **Output Port Reset Value** определяется значение, которое будет выставлено на эту шину при инициализации. Нажмите правой кнопкой мышки по имени блока и переименуйте его в `ledr`.

---

! Некоторые разработчики записывают уникальные значения в **Output Port Reset Value**, чтобы при отсутствии прямого подключения к системе в режиме отладки определять, например, проинициализирована ли система и какая версия именно.

---

#### *Добавление модуля системного идентификатора*

При работе со процессорными системами с программным ядром полезно иметь возможность проверки соответствия аппаратной и программной частей, т.е., при наличии определённого идентификатора, присутствующего в системе, программист имеет возможность удостовериться, что данное приложение написано именно для данной системы. Для задания уникального идентификатора конфигурации системы в таких случаях используют IP-ядро **System ID**. Его можно найти в каталоге IP-ядер в категории **Basic Functions > Simulation; Debug and Verification > Debug and Performance**. Данный модуль содержит уникальный 32-битный идентификатор, который можно определить самостоятельно (рисунок 2.9),

а также временную метку, генерируемую автоматически. После изменения значения и добавления модуля переименуйте его в `sysid`.

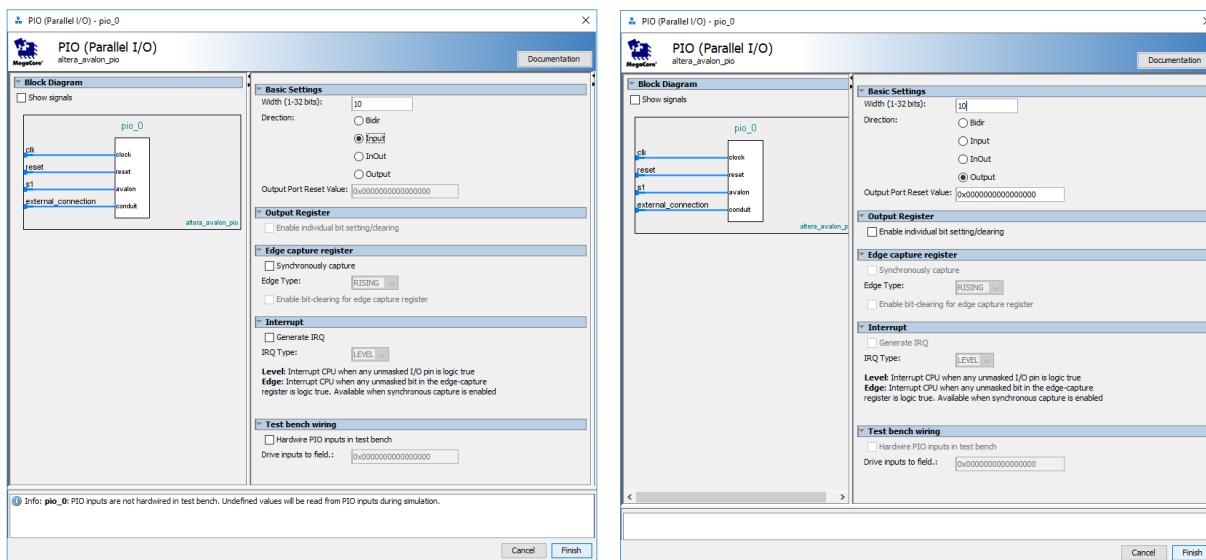


Рисунок 2.8 Параметры блоков PIO для переключателей (слева) и светодиодов (справа)

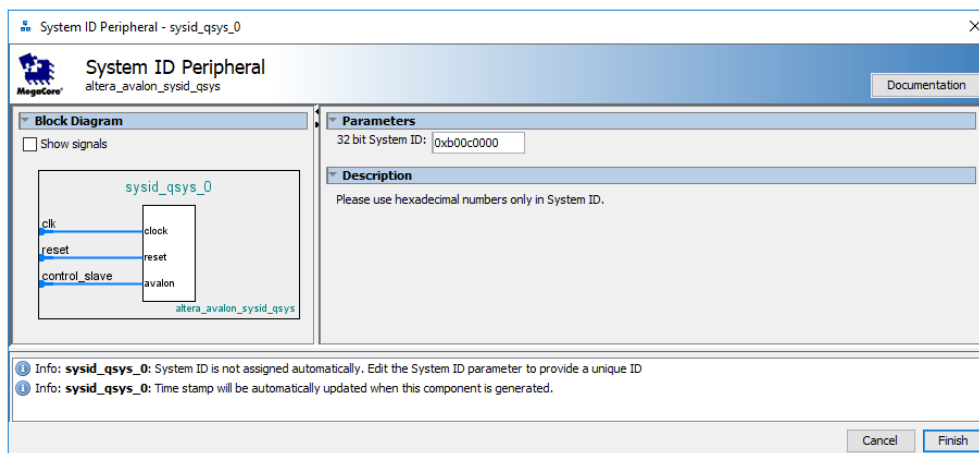


Рисунок 2.9 Окно настройки модуля System ID

### Определение межсоединений и экспорт портов

После добавления и настройки всех необходимых для данного проекта модулей мы можем заметить, что во вкладке **System Contents** блоки не соединены между собой, а в нижней вкладке **Messages** находится большое количество сообщений об ошибке. Для формирования связей между блоками необходимо нажать на пустую точку в месте пересечений потенциальных линий связи между блоками. Такие

точки могут быть только между шинами одного типа, то есть, редактор не позволит соединить между собой, например, порты тактового сигнала и сброса.

Каждому блоку необходим тактовый сигнал, поэтому для начала соединим источник тактового сигнала – порт `clk` блока `clk_0` – с портами для тактового сигнала у всех блоков.

Аналогично с сигналом сброса – порт `clk_reset` блока `clk_0`.

Периферия взаимодействует с процессором посредством интерфейса *Avalon MM*. Порты ввода/вывода не воспринимают инструкции и принимают и/или возвращают только данные, поэтому их необходимо соединить с портом процессора `data_master`. В то же время блок памяти взаимодействует с процессором через инструкции и передает и принимает данные, поэтому необходимо соединить память с портом процессора `instruction_master`.

Относительно проекта собираемая система является черным ящиком с набором входов и выходов. Порты для взаимодействия с внешними элементами формируются экспортированием. Примером такого экспорта являются порты `clk` и `reset` блока `clk_0`, экспортированные наружу по умолчанию. В собранной системе внешне также подключаются светодиоды и переключатели, поэтому у данных блоков необходимо экспортировать порты. Для этого напротив порта `external_connection` необходимо дважды щелкнуть мышью в столбце **Export**, после чего дать имя данному порту. Назначим имена аналогично названию блоков. Результат выполненных операций представлен на рисунке 2.10.

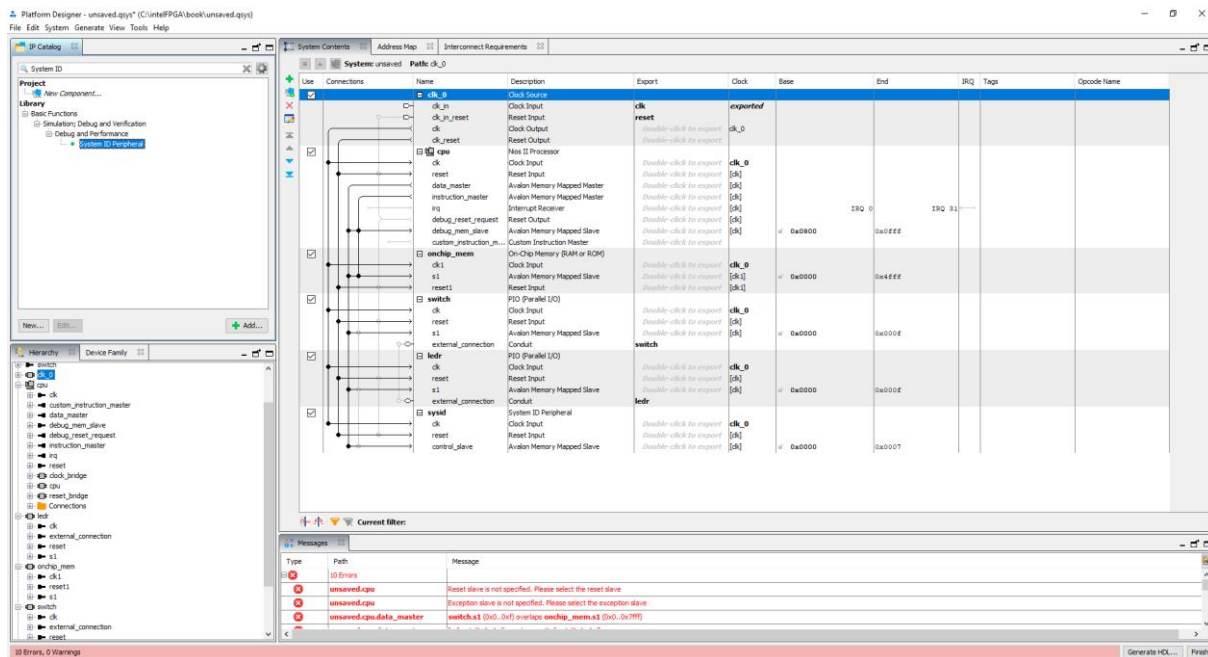


Рисунок 2.10 схема межсоединений в системе на базе Nios II



### Назначение базовых адресов и определение векторов сброса и исключений

Для доступа к периферии процессор использует уникальный базовый адрес устройства. Просмотреть и изменить базовый адрес каждого устройства можно в Platform Designer во вкладке **Address Map**. По умолчанию при добавлении каждого устройства используется стандартная область адресов, однако пересечение адресных пространств между собой не допускается, о чем свидетельствуют ошибки в нижней части экрана (рисунок 2.10). Platform Designer позволяет как отредактировать базовые адреса вручную, так и автоматически распределить адресное пространство между элементами системы. Для этого необходимо в меню **System** выбрать **Assign Base Addresses**. Результат автоматического распределения представлен на рисунке 2.11.

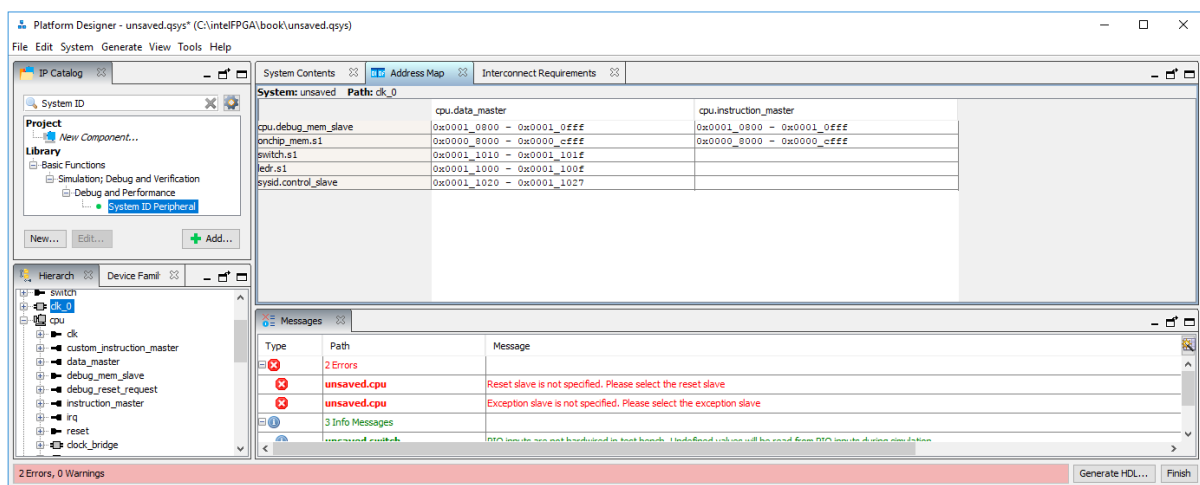


Рисунок 2.11 Адресное пространство системы

Теперь, когда мы добавили в нашу систему блок памяти и распределили адресное пространство, необходимо назначить вектора сброса и исключения в процессоре. В настройках процессора Nios II на вкладке **Vectors** в полях **Reset Vector Memory** и **Exception vector memory** в выпадающем списке выбрать `onchip_mem.sl`.

Далее необходимо сохранить собранную систему нажатием сочетания клавиш **Ctrl+S** или через меню **File > Save**, назвать систему `nios_load1.qsys` и сохранить ее в каталоге проекта.

Процесс создания системы почти завершён. Нам осталось лишь сгенерировать файлы для синтеза системы в среде Quartus, т.е. сформировать HDL-код и bsf-файл. Для этого нажмем кнопку **Generate HDL...** В появившемся окне необходимо выбрать язык синтеза, в нашем случае это **VHDL**, поставить галочку **Create block symbol file (.bsf)** и нажать клавишу **Generate** (рисунок 2.12). После завершения

генерации и появления сообщения об успешной генерации всех файлов можно закрыть инструмент Platform Designer.

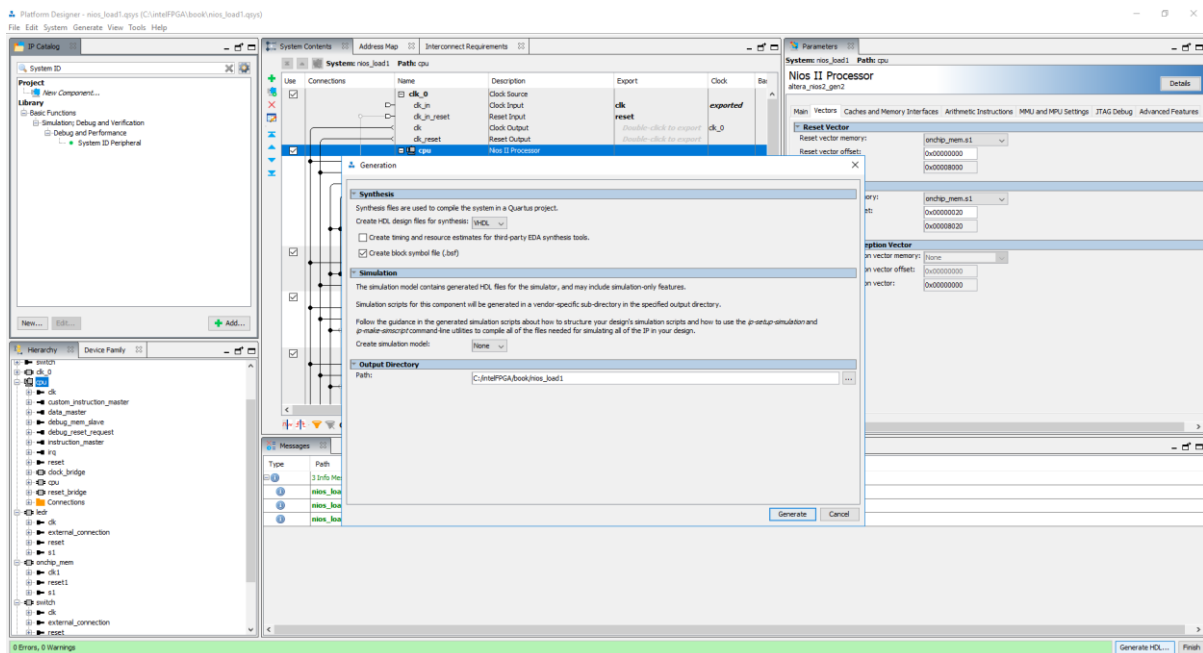



Рисунок 2.12 Окно настройки параметров генерации выходных файлов HDL в Platform Designer

Platform Designer генерирует следующие файлы:

- `nios_load1.qsys`: файл дизайна, содержащий конфигурацию системы. Его можно рассматривать как «исходный файл», который можно использовать с помощью Platform Designer для восстановления других файлов.
- `nios_load1.sopcinfo`: содержит необходимую информацию о конфигурации, используется Nios II EDS для генерации BSP.
- `nios_load1.vhd`: Это файл VHDL верхнего уровня для созданной системы Nios II.
- Другие файлы VHDL: это файлы VHDL для модулей ввода/вывода и подсистем процессора Nios II. Файлы `onchip_mem.vhd`, `switch.vhd` и `ledr.vhd` предназначены для модулей памяти и ввода/вывода системы Nios II, их содержимое можно просмотреть в текстовом редакторе. Однако код, описывающий структуру ядра самого процессора Nios II, обфусцирован.

После закрытия проверьте, что система `nios_load1.qsys` добавлена в проект. Если в списке файлов она отсутствует, добавьте её вручную через меню **Project > Add/Remove Files in Project...**, в появившемся окне в категории **Files** необходимо нажать клавишу  и в появившемся окне выделить добавляемый файл, после чего нажать Открыть. Далее необходимо нажать клавишу **Add** и



Теперь необходимо добавить информацию о периферии. Для этого добавим на блок-диаграмму выводы (на сленге «пины» от англ. «pins») с помощью инструмента **Pin Tool** (рисунок 2.14).

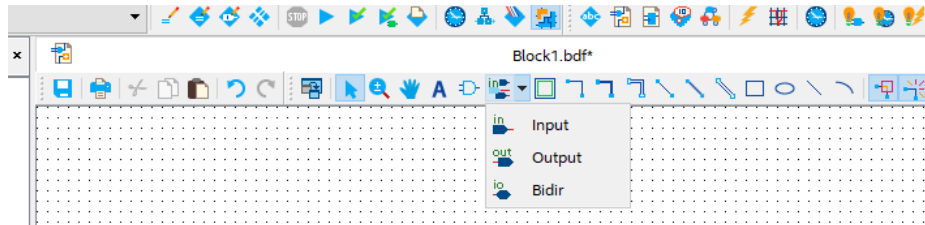


Рисунок 2.14 Панель инструментов Pin Tool

Входные (*Input*) пины для Nios II системы необходимо установить на порты `clk`, `reset_n` и `switch_export`, выходной пин – на порт `ledr_export`. Добавим пины на поле редактора. Каждый из них следует переименовать, а в случае `ledr` можно горизонтально отразить (*flip horizontal*) для удобства подключения. Кроме того, в случае переключателей и светодиодов, в имени пина необходимо указать, что это 10-битная шина, добавив к названию ширину шины в квадратных скобках [`9..0`]. После этого необходимо соединить их линиями с соответствующими им портами на системе Nios II, как это показано на рисунке 2.15.

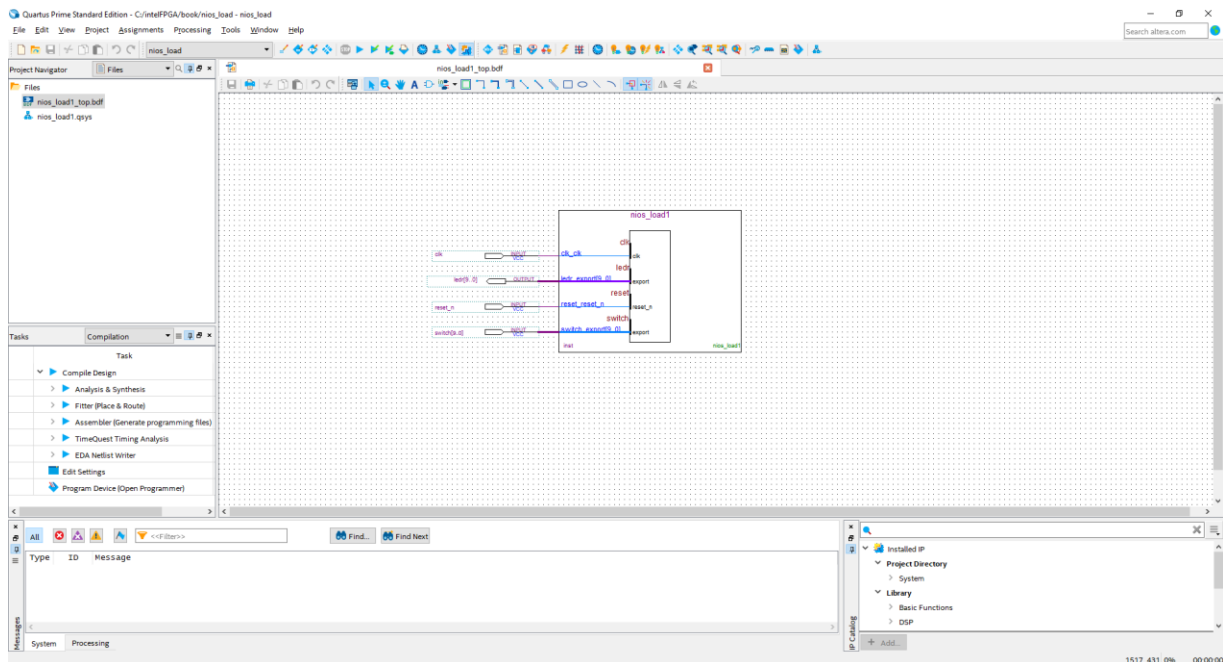


Рисунок 2.15 Блок-схема проекта со светодиодами

Далее необходимо сохранить этот файл с названием `nios_load1_top` и указать его в качестве верхнеуровневого, для этого в списке файлов нажать по нему правой кнопкой мыши и выбрать пункт **Set as Top-Level Entity**.

Во втором способе создания файла верхнего уровня необходимо выполнить аналогичную операцию, описав ее языком VHDL. Объявление объекта верхнего уровня можно найти в файле верхнего уровня HDL `nios_load1.vhd`, а имя объекта – `nios_load1`. Этот файл содержит несколько проектных единиц и является достаточно объёмным. Мы можем открыть файл и использовать ключевое слово `nios_load1` для поиска объявления объекта.

В нашем случае код для объявления объекта выглядит следующим образом:

```
entity nios_load1 is
  port(
    clk_clk      : in std_logic := '0';
    ledr_export  : out std_logic_vector(1 downto 0);
    reset_reset_n : in std_logic := '0';
    switch_export : in std_logic_vector(9 downto 0) :=
(others => '0')
  );
end entity nios_load1;
```

Он показывает, что система, в дополнение к сигналам тактового сигнала и сброса, содержит 10-битный входной порт и двухбитный выходной порт. Имена этих портов определяются именами модулей, созданных в Platform Designer.

Система Nios II может быть представлена в виде обычного компонента HDL и соответственно легко интегрироваться с другими блоками в проекте. Поскольку наша демонстрационная система не содержит дополнительной логики, нам просто нужно создать блок верхнего уровня для того, чтобы «упаковать», «обернуть» (англ. «*wrapper*») в него нашу систему с Nios II. Еще раз заострим внимание: сущность верхнего уровня (**Top-Level Entity**) – такой блок, чьи выводы (входы и выходы) непосредственно присоединяются к выводам самой микросхемы, т.е. к элементам ввода-вывода, «ножкам» или «пинам» (сленг) ПЛИС. Код HDL показан в листинге 2.1.

После определения файла верхнего уровня необходимо проверить корректность конфигураций и подключений, запустив этап компиляции под названием **Analysis & Synthesis**, дважды щелкнув по нему в списке **Compile Design**. При успешном завершении этого этапа компиляции необходимо установить связь (программно) между созданными пинами и физическими ножками, к которым подключена соответствующая периферия.

## Листинг 2.1 Система верхнего уровня

---

```
library ieee;
use ieee.std_logic_1164.all;
entity nios_loadl_top is
  port(
    clk      : in  std_logic;
    switch   : in  std_logic_vector(9 downto 0);
    ledr     : out std_logic_vector(9 downto 0);
    reset    : out  std_logic
  );
end nios_loadl_top;
architecture arch of nios_loadl_top is
  component nios_loadl
    port(
      clk           : in  std_logic;
      reset_n       : in  std_logic;
      switch_export : in  std_logic_vector(9 downto 0);
      ledr_export   : out std_logic_vector(9 downto 0)
    );
  end component;
begin
  nios_unit : nios_loadl port map(
    clk           => clk,
    reset_n       => reset,
    switch_export => switch,
    ledr_export   => ledr
  );
end arch;
```

---

### 2.2.4. Настройка периферии в Pin Planner


Одним из преимуществ использования ПЛИС перед, скажем, микроконтроллерами, является возможность конфигурирования, в том числе в реальном времени, разнообразных параметров элементов ввода-вывода. Для данных задач в качестве инструмента, встроенного в Quartus, мы воспользуемся утилитой **Pin Planner**. В данном учебном пособии мы воспользуемся только одной из множества его функций – мы зададим соответствие физическому контакту на чипе тому или иному пину в проекте (определим свойство пина «*Location*»). Запустить Pin Planner можно нажатием по значку  на панели меню, либо из главного меню **Assignments > Pin Planner**. В появившемся окне в центре изображен вид сверху на расположение входов и выходов кристалла, снизу –

таблица пинов проекта, по краям – инструменты настройки отображения информации на экране (рисунок 2.16).

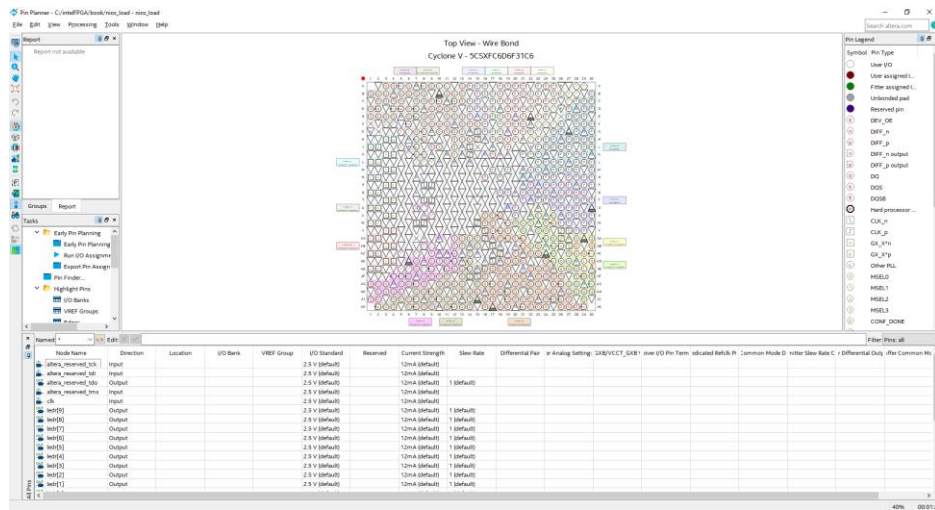


Рисунок 2.16 Окно утилиты Pin Planner


В таблице пинов проекта в начале расположены четыре с приставкой altera\_ – это резервные пины, генерирующиеся автоматически при создании Nios II-системы и не требуют настройки. Остальным пинам необходимо указать их расположение в столбце **Location**, вписав туда координаты соответствующего вывода на кристалле. Эта информация указана в документации к плате (таблица 2.1). Для сигнала сброса мы воспользуемся кнопкой KEY0. После записи данных в таблицу можно закрыть утилиту.

Таблица 2.1 Соответствие периферии выводам ПЛИС

Название вывода	Вывод на ПЛИС	Название вывода	Вывод на ПЛИС
clk	PIN_AF14	reset_n	PIN_AJ4
ledr[0]	PIN_AA24	switch[0]	PIN_AB30
ledr[1]	PIN_AB23	switch[1]	PIN_Y27
ledr[2]	PIN_AC23	switch[2]	PIN_AB28
ledr[3]	PIN_AD24	switch[3]	PIN_AC30
ledr[4]	PIN_AG25	switch[4]	PIN_W25
ledr[5]	PIN_AF25	switch[5]	PIN_V25
ledr[6]	PIN_AE24	switch[6]	PIN_AC28
ledr[7]	PIN_AF24	switch[7]	PIN_AD30
ledr[8]	PIN_AB22	switch[8]	PIN_AC29
ledr[9]	PIN_AC22	switch[9]	PIN_AA30

### 2.2.5. Компиляция проекта и прошивка платы

Для генерации прошивки для ПЛИС необходимо выполнить пункт **Assembler** из списка компиляции (**Compile Design**), для этого дважды щелкнем мышью по этому пункту. Ввиду того, что анализ и синтез мы уже запускали и с тех пор дизайн проекта не меняли, компиляция уже начнется со следующего пункта – **Fitter**, а затем уже запустит **Assembler**. Этапы компиляции проекта для ПЛИС в данном учебном пособии подробно не рассматриваются, подробнее это описано в [1].

По результатам компиляции сгенерируется файл прошивки для ПЛИС в формате \*.sof, его название будет совпадать с названием файла верхнего уровня, найти его можно в подкаталоге проекта **/output\_files**. Для прошивки ПЛИС подключите плату к питанию и к компьютеру, запустите её и запустите утилиту **Programmer** нажатием на значок  или из меню **Tools > Programmer**. В появившемся окне расположены настройки конфигурирования ПЛИС (рисунок 2.17).

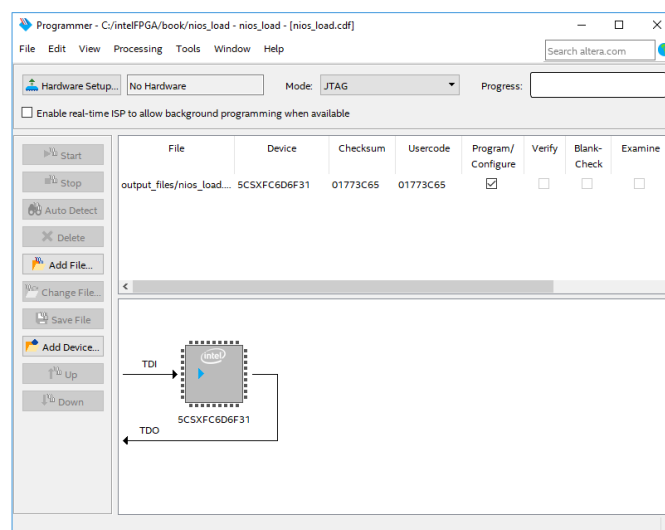
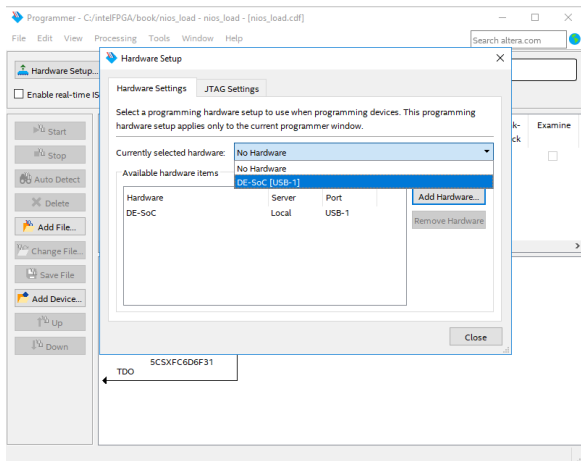


Рисунок 2.17 Окно утилиты Programmer

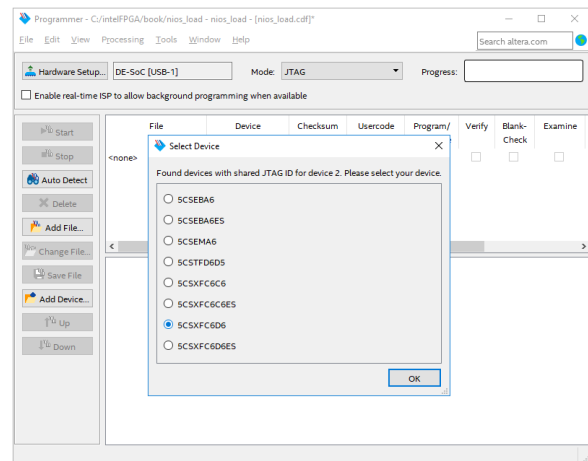
Устройство, для которого мы разработали дизайн, было выбрано в самом начале при создании проекта, эту информацию и сам файл для прошивки утилита автоматически выбрала из каталога проекта. Однако часть площади нашей микросхемы занимает аппаратная система на кристалле с ядром **ARM Cortex-A9**, которая называется **HPS (hardware processing system)**. Данная система также может быть сконфигурирована по **JTAG** интерфейсу, но наш проект относится только к части программируемой логики микросхемы, работа с HPS требует отдельного пособия. Для корректной настройки параметров прошивки платы необходимо выполнить следующие шаги:



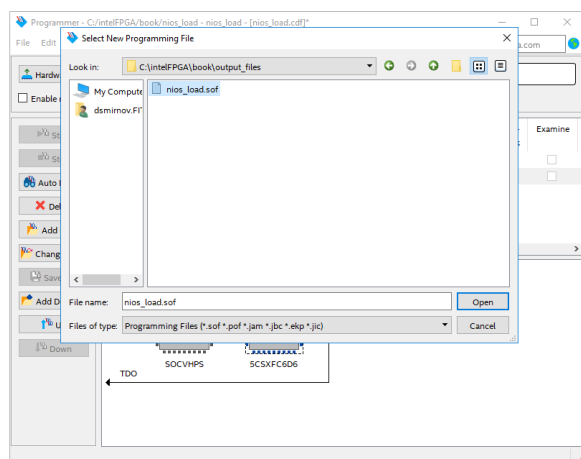
1. Нажмите на кнопку **Hardware Setup...**, в появившемся окне выберите подключенное к ПК устройство, нажмите кнопку **Close** (рисунок 2.18, а);
2. Удалите из списка файл прошивки и нажмите кнопку **Auto Detect**;
3. В появившемся окне (рисунок 2.18, б) необходимо выбрать наше устройство: **5CSXFC6D6**, нажать клавишу **OK**; в списке подключенных устройств добавится **SOCVHPS** – это и есть ядро ARM;
4. Выделите в списке устройство ПЛИС (5CSXFC6D6), нажмите клавишу **Change File...**, в открывшемся окне (рисунок 2.18, в) выберите файл прошивки и нажмите **Open**;
5. В столбце **Program/Configure** поставьте галочку в строке устройства ПЛИС и нажмите кнопку **Start** для запуска процесса прошивки (конфигурации ПЛИС); процесс передачи данных отображается в верхнем правом углу окна (рисунок 2.18, г), надпись **100% Successful** означает успешное завершение процесса.



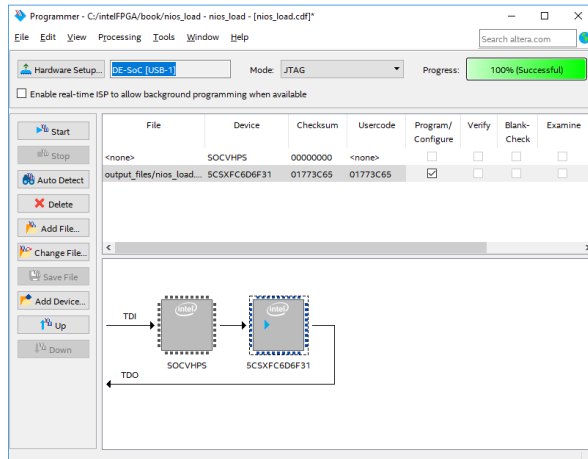
а



б



в



г

Рисунок 2.18 Последовательность прошивки ПЛИС с помощью утилиты

Programmer

Данным этапом заканчивается реализация аппаратной части проекта, процессорная система на базе Nios II загружена на плату, теперь необходимо создать приложение и загрузить его в память процессора.

### 2.3. Разработка программной части

Для разработки программной части проекта используется комплект утилит, в общем случае называемый **Nios II SBT** (*Software Build Tools*) *GUI*. Для реализации необходимо выполнить шаги 2, 3 и 4 из пункта 2.1.3. Для разработки программного обеспечения есть два варианта: разработка с помощью команд в командной строке или через графический интерфейс. В данном учебном пособии используется второй метод, и в качестве графического интерфейса используется *Nios II Eclipse*.

#### 2.3.1. Генерация BSP

Программное обеспечение для системы на базе Nios II состоит из приложения пользователя и **BSP библиотеки**. **BSP** (*Board Support Package*) – это библиотека, основанная на конфигурации собранной системы на базе Nios II; она содержит необходимую информацию для взаимодействия пользовательского приложения с элементами системы. Для генерации библиотеки BSP и создания заготовки под приложение пользователя необходимо выполнить следующие действия:

1. Запустите Eclipse: его можно запустить как из меню **Пуск**, так и из среды **Quartus: Tools > Nios II Software Build Tools for Eclipse**; Точно так же Eclipse можно запустить из Platform Designer.
2. При первом запуске появится окно запроса адреса для рабочего пространства по умолчанию;
3. После инициализации среды выберите в меню **File > New > Nios II Board Support Package**;
4. В появившемся окне необходимо дать название проекту и добавить \*.sopcinfo файл, который был сгенерирован нами ранее (рисунок 2.19);
5. После обработки файла системой убедитесь, что в списке **CPU** указан тот процессор, который вы создали в **Platform Designer**, остальные настройки оставьте по умолчанию;
6. Нажмите **Finish** для генерации проекта с библиотеками BSP.

---

! При создании проекта типа BSP принято к названию добавлять постфикс bsp, чтобы отличать от каталогов приложений.

---

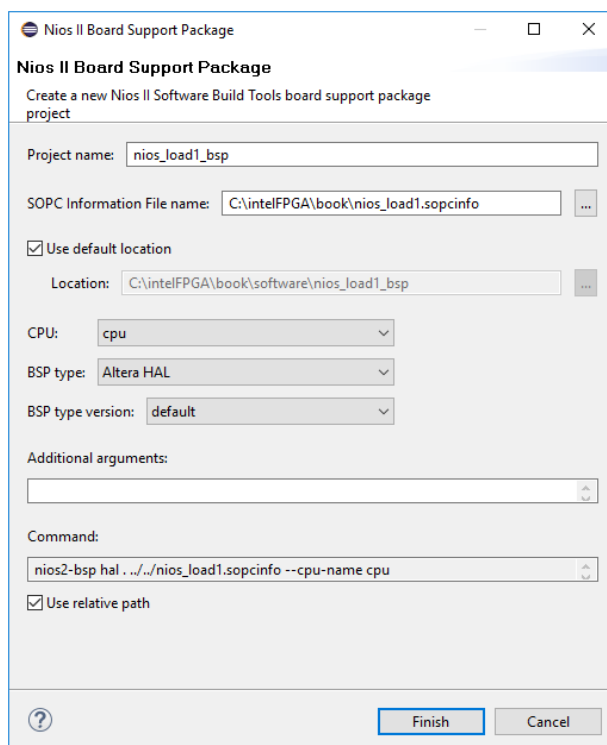


Рисунок 2.19 Окно создания BSP

Для данного проекта не требуется дополнительных настроек BSP, возможности конфигурирования этих библиотек будут рассмотрены в главе 5.

### 2.3.2. Реализация приложения

Для создания проекта приложения необходимо из меню **File > New** выбрать **Nios II Application**. В открывшемся окне назовем проект `nios_test1` и укажем привязку к BSP, созданному ранее (рисунок 2.20). Остальные поля оставим по умолчанию и нажмем кнопку **Finish**.

Подготовка к созданию завершена, теперь необходимо написать код нашего приложения. Для этого необходимо создать файл `main.c` в проекте `nios_test1`, нажав правой кнопкой мыши на каталог проекта пользовательского приложения и выбрав **New > Source File**. В открывшемся окне необходимо задать имя файла с расширением `*.c` и выбрать шаблон (**Template**) `<None>` для создания абсолютно пустого файла. Нажмите клавишу **Finish**.

Готовый код приложения указан в разделе 2.5 ниже, вставьте его в файл и сохраните; его функционал разберем подробнее в разделе 2.4.

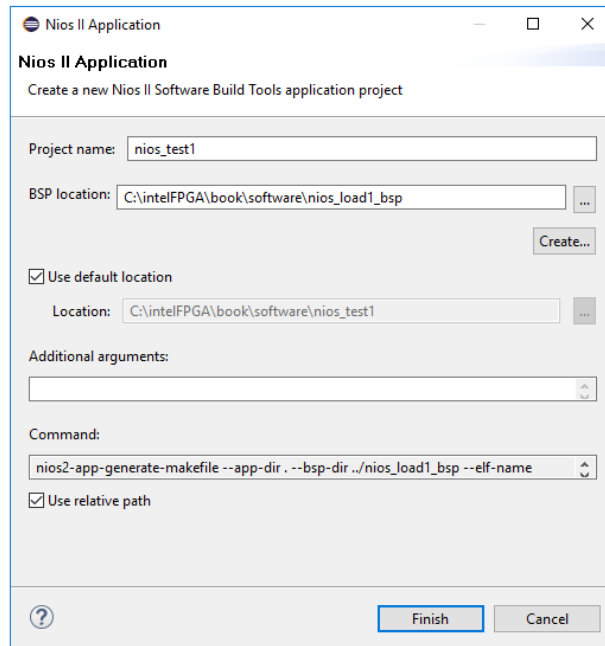


Рисунок 2.20 Окно создания проекта приложения `nios_test1`

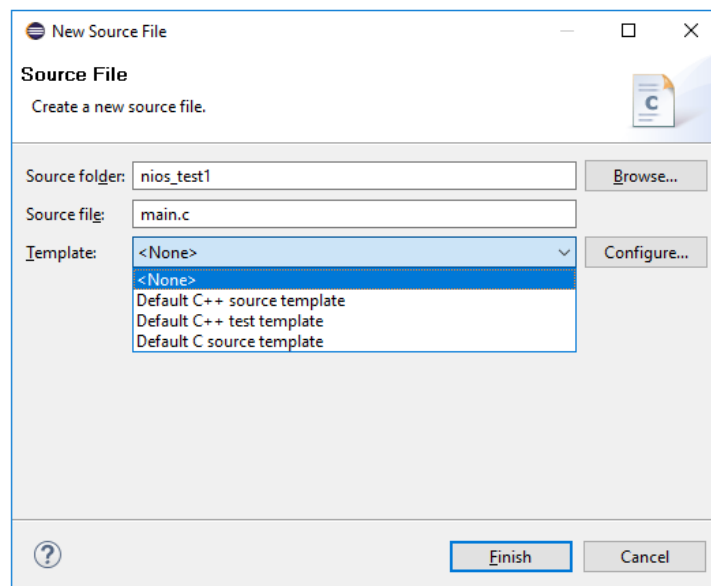


Рисунок 2.21 Окно создания файла приложения `main.c`

### 2.3.3. Компиляция и запуск программы на плате

После сохранения нажмите правой кнопкой мыши по каталогу проекта и выберите **Build Project**. Прогресс сборки отображается в консоли, при удачном завершении компиляции в последних сообщениях указывается размер программы и размер оставшейся свободной памяти, а в последней строке говорится об успешном завершении сборки (рисунок 2.22). Результатом компиляции является

набор файлов, ключевой из них для нас на текущий момент – это файл с расширением \*.elf – это файл образа приложения.

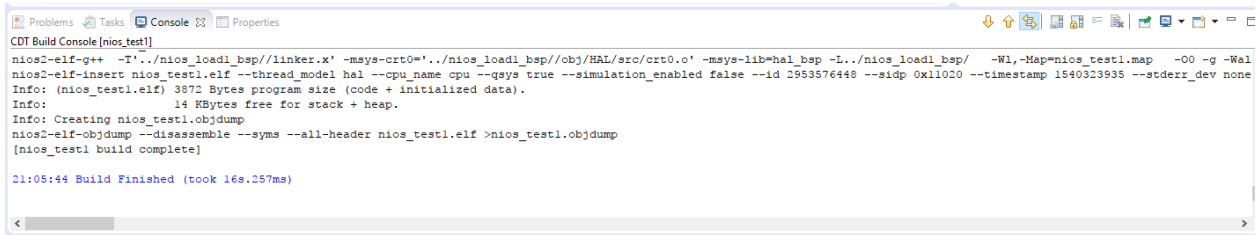


Рисунок 2.22 Результат сборки приложения

Файл образа необходимо загрузить в ПЛИС, непосредственно в память процессора. Для этого необходимо нажать правой кнопкой мышки по каталогу приложения и выбрать **Run as > Nios Hardware**. При первом обращении к данной команде откроется окно с настройкой конфигурации запуска приложения. В появившемся окне на вкладке **Project** отображается информация о проекте, с которым идет работа и указывается имя файла образа (рисунок 2.23).

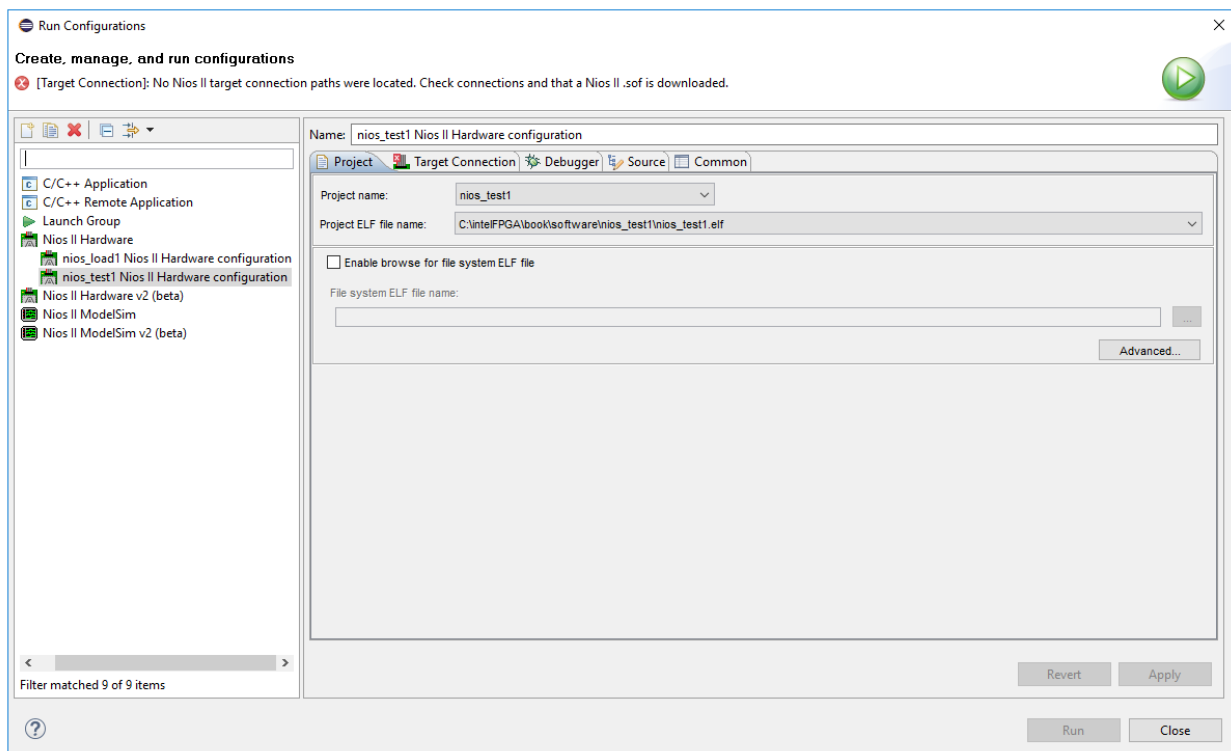


Рисунок 2.23 Окно настройки запуска приложения

На вкладке **Target Connection** отображаются доступные подключенные платформы, на которых можно запустить скомпилированное приложение. Нажмем **Refresh Connections**, чтобы подключенная плата с системой на Nios II появились

в списке. При появлении устройства в списке можно проверить совпадение **System ID** и временных меток с тем, чтобы удостовериться, что созданное приложение сделано именно для подключенной платформы (рисунок 2.24).

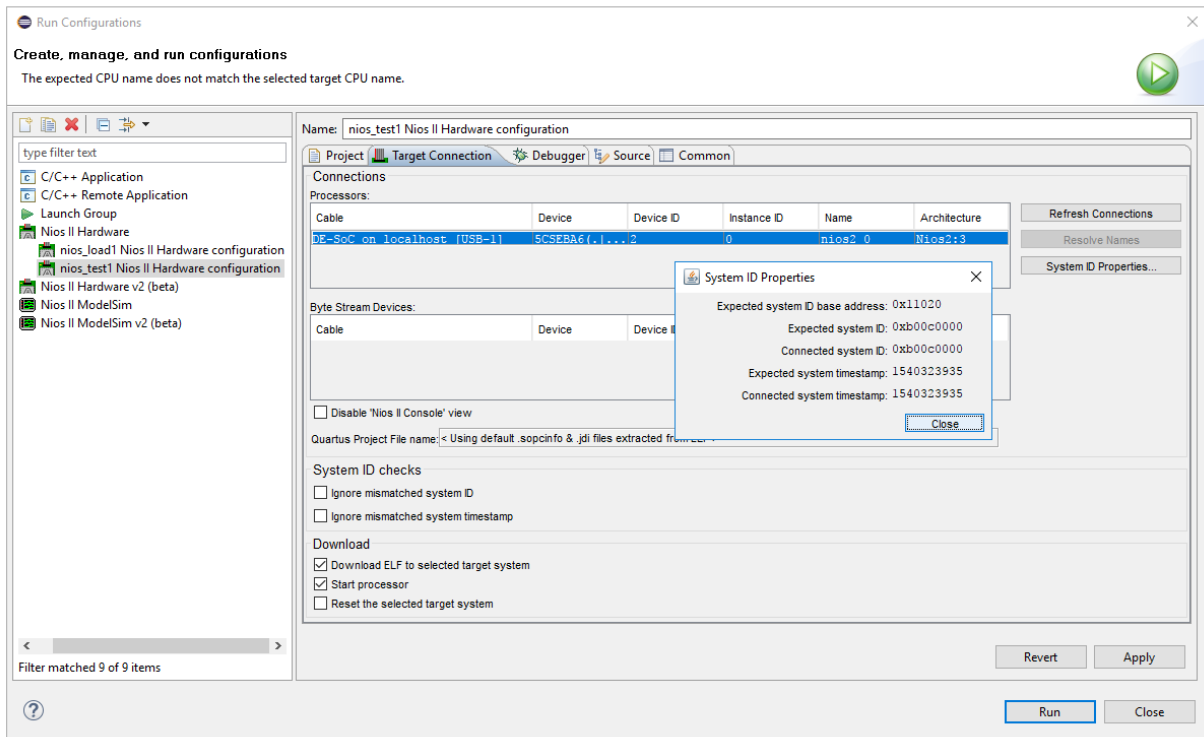


Рисунок 2.24 Выбор подключаемого устройства и проверка его System ID

Оставим остальные настройки по умолчанию, далее необходимо нажать клавишу **Apply** и запустить приложение нажатием кнопки **Run**. В командной строке отобразится процесс загрузки данных на ПЛИС, после чего приложение запустится на плате (рисунок 2.25).

#### 2.4. Обзор реализации программы для системы на основе Nios II

В этом разделе мы подробнее рассмотрим, какие файлы отображаются в списке при создании программной части проекта. Содержимое каталогов отображается во вкладке Project Explorer в Eclipse (рисунок 2.26). В каталоге приложения отображены два вида файлов:

- **Includes** – это библиотеки, которые используются компилятором в процессе сборки, они не копируются в директорию самого проекта, в настройках генерации прописаны пути к этим библиотекам;
- **Makefile** – главный файл, описывающий правила сборки программы для компиляции. Его содержимое не рекомендуется изменять без необходимости.

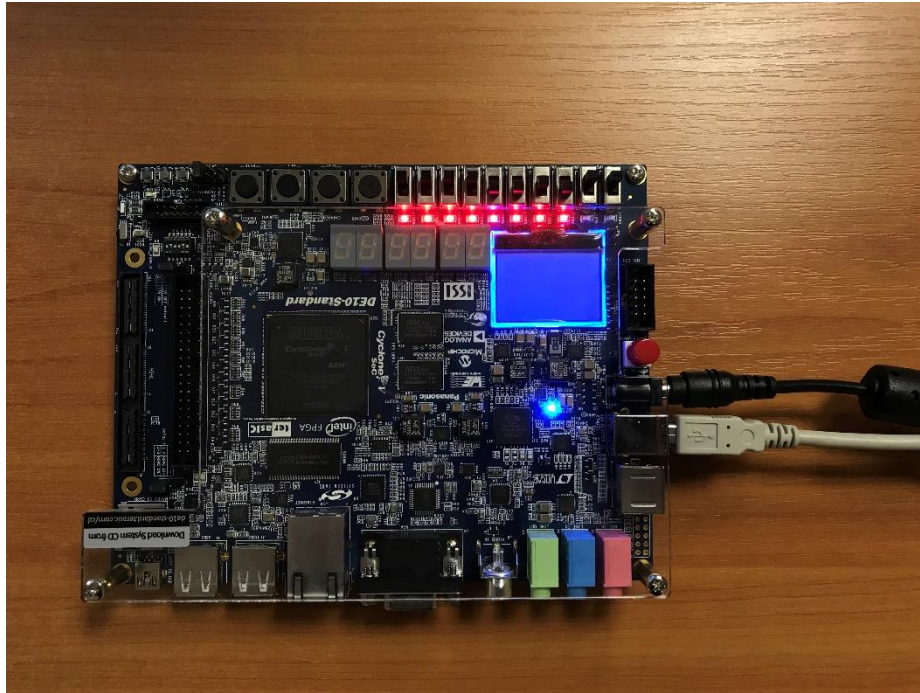
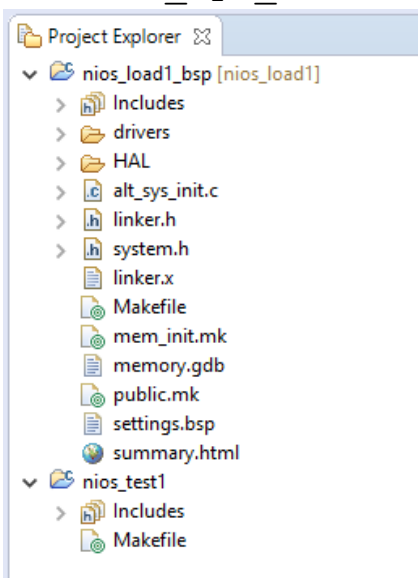


Рисунок 2.25 Плата с запущенным приложением, фрагмент работы приложения

В каталоге BSP список больше:

- Includes – то же что и в каталоге проекта;
- drivers – папка со всеми драйверами устройств;
- HAL – содержит файлы HAL, подробнее будет рассмотрено в разделе 5;
- alt\_sys\_init.c – код инициализации драйверов устройств;



- linker.h – заголовочный файл с информацией об элементах памяти;
- system.h – заголовочный файл с объявлением переменных, описывающих адресное пространство системы;
- linker.x – скрипт для GNU линковщика;
- Makefile – описывает правила сборки библиотеки;
- mem\_init.mk – фрагмент Makefile, определяющий, в какой файл и по каким правилам приложение будет генерироваться в файл инициализации памяти;
- memory.gdb – файл с объявлением областей памяти для GNU отладчика;

Рисунок 2.26 Список файлов программной части проекта

- `public.mk` – публичный фрагмент Makefile, используется также и при сборке проекта;
- `settings.bsp` – файл настроек BSP, редактируется в *BSP Editor*, подробнее это рассматривается в разделе 5.2.3;
- `summary.html` – содержит общую информацию о библиотеке.

Для реализации работы приложения мы воспользовались тремя базовыми файлами из библиотек:

- `system.h` – содержит список автоматически сгенерированных базовых адресов;
- `alt_types.h` – файл с явным определением типов данных;
- `io.h` – набор макросов для чтения и записи регистров ввода/вывода.

Рассмотрим их подробнее.

`system.h`

Базовые адреса периферии определяются в Platform Designer, их можно посмотреть в таблице адресов и определить постоянными величинами в приложении вручную. Однако при небольших изменениях аппаратной реализации системы ввиду автоматического распределения адресного пространства адреса могут изменяться, что порождает потенциальную возможность ошибок при разработке.

Чтобы избежать описанных проблем, утилиты разработки автоматизируют этот процесс. В момент сборки библиотеки BSP происходит анализ файла `*.sopcinfo`, информация о каждом модуле извлекается и генерируется файл `system.h`, в который эта информация записывается.

Откроем файл, чтобы ознакомиться с его содержимым. Пролистав весь файл, мы увидим большой набор директив, которые разбиты на группы с помощью комментариев, в которых написано название модуля, к которому данные директивы относятся. Например, директивы, относящиеся к модулю PIO для светодиодов, выглядят следующим образом:

```
/*
 * ledr configuration
 *
 */

#define ALT_MODULE_CLASS_ledr altera_avalon_pio
#define LEDR_BASE 0x11000
#define LEDR_BIT_CLEARING_EDGE_REGISTER 0
```



```
#define LEDR_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LEDR_CAPTURE 0
#define LEDR_DATA_WIDTH 10
...
```

Добавив этот файл в код программы, мы можем подставлять обозначение `LEDR_BASE` для обращения к базовому адресу модуля PIO светодиодов.

`alt_types.h`

В языке Си есть различные типы данных, однако битность получаемого результата определяется компилятором при реализации программы. Важно отметить, что при низкоуровневом взаимодействии с периферией необходимо иметь строгое определение битности и формата данных, что не может быть гарантировано при использовании стандартных типов данных в языке Си. Для этого используется небольшая библиотека, `alt_types.h`, в которой определены конкретные типы данных:

- `alt_8` – знаковый, целочисленный, 8-битный;
- `alt_u8` – беззнаковый, целочисленный, 8-битный;
- `alt_16` – знаковый, целочисленный, 16-битный;
- `alt_u16` – беззнаковый, целочисленный, 16-битный;
- `alt_32` – знаковый, целочисленный, 32-битный;
- `alt_u32` – беззнаковый, целочисленный, 32-битный;
- `alt_64` – знаковый, целочисленный, 64-битный;
- `alt_u64` – беззнаковый, целочисленный, 64-битный.

`io.h`

Взаимодействие процессора с периферией производится через регистры периферии, каждый регистр имеет свой адрес, который мы определили при сборке программы (см. выше). Для реализации программного обмена данными генерируется библиотека `io.h`, в которой описаны макросы доступа к периферии для чтения и записи данных по базовому адресу. Используемые нами макросы имеют следующий вид:

- `IORD(base, offset)` – макрос считывания данных по базовому адресу `base` со смещением `offset`;
- `IOWR(base, offset, data)` – макрос записи данных `data` по базовому адресу `base` со смещением `offset`.

### Описание программы main.c

Ознакомившись с доступными инструментами и библиотеками, рассмотрим теперь реализацию приложения. Простейшим способом реализации программы для встраиваемых систем является так называемый циклический алгоритм (англ. «*super loop*» или *round-robin*): в такой реализации программа состоит из набора задач, выполняемых последовательно в бесконечном цикле. В псевдокоде это выглядит следующим образом:

```
main() {
    функция_инициализации();
    бесконечный_цикл() {
        задача_1();
        задача_2();
        ...
        задача_n();
    }
}
```

В такой программе один раз запускается функция для инициализации необходимых элементов системы, а затем запускается бесконечный цикл последовательного выполнения задач. Такая реализация считается достаточной, если время выполнения всего цикла удовлетворяет запрашиваемым требованиям. Этот подход подразумевает, что ни одна из задач не имеет приоритета перед остальными, что редко встречается в жизни. Решение проблемы приоритетов будет рассмотрено в разделе 6.

Рассмотрим по частям код программы (листинг 2.4).

В главной части программы производится инициализация переменных, затем программа переходит в бесконечный цикл, где выполняются следующие операции: функция, созданная для считывания данных с переключателей, пустой цикл для паузы и условный оператор для записи последовательности на светодиоды.

Функция `sw_get_command()` принимает в качестве аргументов адрес переключателей и целочисленную (листинг 2.3).

Считывание данных в переменную производится с использованием макроса `IORD()`, а для того, чтобы избежать потенциальных проблем, значение маскируется путем выполнения логической операции И с маской – `0x000003ff`.

## Листинг 2.2 Главная часть приложения

---

```
int main(){
    int prd;
    int i = 0x1;
    unsigned long j, itr;

    . . .

    // Round-robin
    while(1){
        sw_get_command(SWITCH_BASE , &prd); // считывание
значений переключателей
        itr = prd * 2500;
        for (j=0; j<itr; j ++){} // пустой цикл в
качестве задержки
        if (i >= 0x800) { // если все включены
            IOWR(LED_BASE, 0, 0x0); // выключить все
            i = 0x1;
        } else {
            IOWR(LED_BASE, 0, i-1); // включить некоторые
            i = (i<<1); // плюс 1 включенный
        }
    }
}
```

---

## Листинг 2.3 Функция считывания данных с переключателей

---

```
void sw_get_command(alt_u32 sw_base, int *prd)
{
    *prd = IORD(sw_base, 0) & 0x000003ff;
}
```

---

Пустой цикл создается для того, чтобы искусственно создать небольшую паузу по времени. Предполагая, что одна итерация цикла занимает приблизительно 400 нс, 2500 итераций дадут задержку примерно в 1 мс. Это очень грубая реализация задержки, более оптимальный вариант реализации мы рассмотрим в следующих разделах.

## 2.5.ЛИСТИНГ

### ЛИСТИНГ 2.4 main.c

---

```
//Добавляем стандартные библиотеки
#include "io.h"
#include "alt_types.h"
#include "system.h"

//Описание функции считывания значений переключателей
void sw_get_command(alt_u32 sw_base, int *prd)
{
    *prd = IORD(sw_base, 0) & 0x000003ff;
}

int main(){
    int prd;
    int i = 0x1;
    unsigned long j, itr;

    // Round-robin
    while(1){
        sw_get_command(SWITCH_BASE ,&prd);
        itr = prd * 2500;
        for (j=0; j<itr; j ++){}
        if (i >= 0x800) {
            IOWR(LED_BASE, 0, 0x0);
            i = 0x1;
        } else {
            IOWR(LED_BASE, 0, i-1);
            i = (i<<1);
        }
    }
}
```

---

### 3. ОБЗОР ПРОЦЕССОРА NIOS II GEN2

В этой главе мы рассмотрим процессор Nios II Gen2, его ключевые особенности и его отличие от предыдущей версии процессора от Altera.

#### 3.1. Введение

Nios II – это процессор, ядро которого реализовано программно (*soft-processor*) для устройств ПЛИС фирмы Intel FPGA (ранее Altera). В отличие от процессоров, структура которых реализована на кристалле (*hard processors*), данный процессор описывается языками описания аппаратуры – *HDL-языками* – после чего реализуется на логических ячейках ПЛИС. Nios II – это **RISC-процессор**, то есть это процессор, использующий сокращённый набор инструкций. Компания Intel FPGA предлагает пользователям на выбор классический Nios II процессор и его продвинутую реализацию второго поколения **Nios II Gen2**. Основными преимуществами последнего являются:

- возможность полного использования 32-битного адресного пространства;
- возможность добавления пользователем адресов периферии для обхода кэша данных;
- улучшенный интерфейс;
- полная поддержка **ЕСС** (*Error Correction Code*), включая кэш данных и **ТСМ** (*Tightly-coupled memory*);
- возможность использования статического предиктора ветвления;
- высокопроизводительный умножитель;
- поддержка 64-битного умножения для всех устройств;
- улучшенный блок операции сдвига до четырех бит за такт;
- возможность выключения кэша инструкций даже при использовании отладки по JTAG<sup>4</sup>.

Существует две версии ядра Nios II gen2:

- *Nios II/f* – ядро для высокой производительности. Обладает большими возможностями настройки для более точной конфигурации.
- *Nios II/e* – ядро для максимальной экономии на размере. Ядро с ограниченными возможностями, многие настройки отсутствуют.

Система на основе процессора Nios II состоит из ядра процессора, набора периферии на кристалле, памяти на кристалле и интерфейса к памяти вне кристалла; при этом все перечисленное реализуется на выбранном устройстве.

---

<sup>4</sup> Подробнее: <https://www.intel.com/content/www/us/en/programmable/documentation/iga1432837083642.html>

Подобно микроконтроллеру, все процессоры Nios II используют соответствующий набор инструкций и модель программирования.

### 3.1.1. Концепция настраиваемого ядра софт-процессора

#### *Конфигурируемость процессора*

Процессор Nios II является настраиваемым IP-ядром. Вы можете добавлять или убирать различные функции для того, чтобы удовлетворить необходимым требованиям по производительности и занимаемому на ПЛИС количеству логических элементов. Приставка «софт» означает, что ядро процессора не зафиксировано в кремнии и может быть реализовано на любой ПЛИС фирмы Altera при наличии достаточного количества ресурсов.

#### *Набор гибкой периферии и таблица адресов*

Одним из наиболее заметных отличий между процессором Nios II и аппаратным микроконтроллером является набор гибкой периферии. Поскольку процессор Nios II реализован на программируемой логике, мы можем легко создавать индивидуальные (*made-to-order*) системы на основе Nios II именно с использованием только необходимого для проекта набора периферии. Подробнее о периферии мы поговорим в главе 4.

Altera предоставляет конструкциям программного обеспечения доступ к памяти и периферии в целом, независимо от адреса. Поэтому набор гибкой периферии и таблица адресов не влияют на разработчиков приложений.

#### *Доступность Nios II.*

Ядро Nios II fast может использоваться без лицензии в следующих случаях:

- для поведенческой симуляции процессора Nios II с вашей системой;
- для функциональной верификации вашего дизайна, а также для простой и быстрой оценки его скорости;
- для генерации ограниченных по времени файлов прошивки с включенным в них процессором Nios II;
- для программирования вашего устройства – такая прошивка будет работать в течение 30 минут и только пока подключена к компьютеру.

Для доступа к полноценному функционалу необходимо запросить лицензию, подробнее с этим можно ознакомиться в документе AN320.

## 3.2. Архитектура процессора

В этом разделе мы кратко ознакомимся с аппаратной структурой процессора Nios II.

Архитектура Nios II описывает **ISA** – архитектура набора инструкций (англ. *instruction set architecture*). ISA, в свою очередь, делает необходимым набор функциональных блоков для выполнения инструкций. Ядро процессора Nios II представляет собой аппаратный блок, который реализует набор инструкций для Nios II и поддерживает функциональные блоки, подробно описанные в [2]. Ядро процессора не включает в себя периферию и другую подключенную логику. Оно включает в себя только схемы, необходимые для реализации архитектуры Nios II.

Архитектура процессора Nios II (рисунок 3.1) определяет следующие функциональные блоки:

- файл регистров;
- Арифметико-Логическое Устройство (АЛУ);
- интерфейс для логики пользовательских инструкций;
- контроллер исключений;
- внутренний или внешний контроллер прерываний;
- шину инструкций;
- шину данных;
- блок управления памятью (MMU);
- блок защиты памяти (MPU);
- блоки памяти для кэшей инструкций и данных;
- интерфейс тесно-связанной памяти для инструкций и данных;
- модуль отладки JTAG.

### 3.2.1. Файл регистров

Архитектура процессора Nios II поддерживает плоский файл регистров, состоящий из 32 целочисленных регистров битностью 32, а также стольких же 32-битных контрольных регистров. Архитектура поддерживает режимы «supervisor» и «user», позволяющие коду системы защитить контрольные регистры от «блуждающих» приложений.

Процессор Nios II может дополнительно иметь один и более набор теневых (*shadow*) регистров. При использовании наборов теневых регистров поле CRS статусного регистра показывает, какой набор регистров используется в данный момент. Инструкция обращается к регистрам общего назначения, какой бы набор ни был активен.

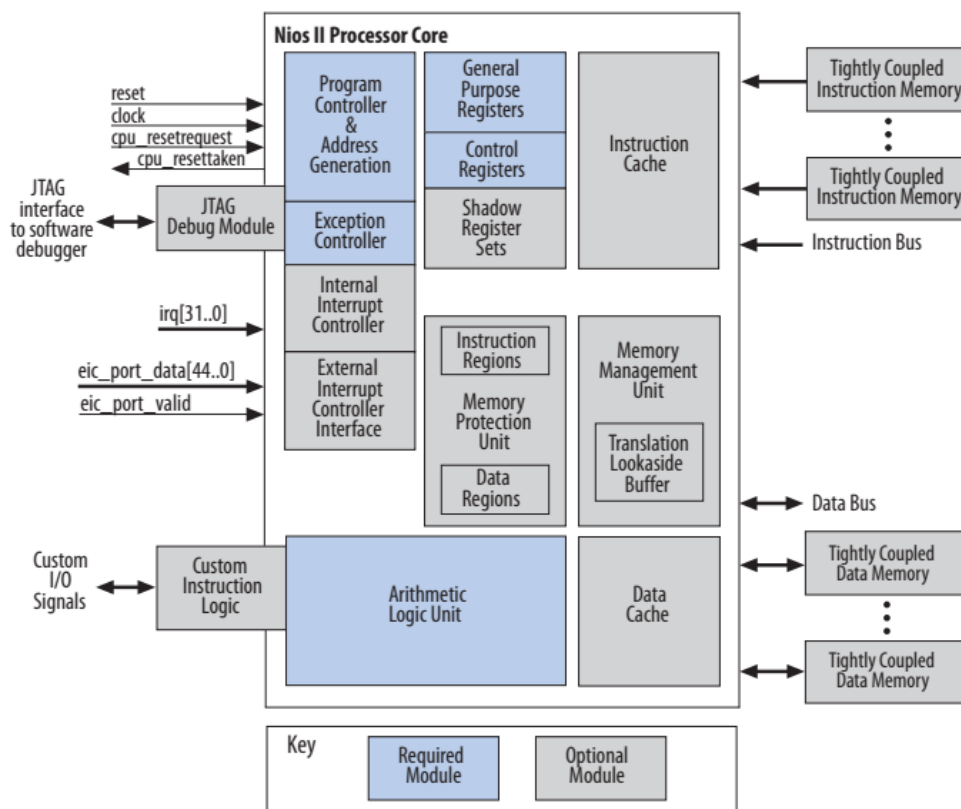


Рисунок 3.1 Схема внутренних и внешних межсоединений процессора Nios II

Классическим применением наборов теневого регистров является переключение процессов (*context switching*). При применении наборов теневого регистров процессор Nios II использует специальные инструкции, *rdprs* и *wrprs*, для перемещения данных между наборами регистров. Наборы теневого регистров, как правило, управляются ядром операционной системы и прозрачны для кода приложения. Процессор Nios II может иметь до 63 наборов теневого регистров [3].

### 3.2.2. АЛУ

**Арифметико-логическое устройство (АЛУ)** процессора Nios II работает с данными, которые хранятся в регистрах общего назначения. Операции АЛУ используют одно или два значения из регистров и сохраняют результат в другом регистре. АЛУ поддерживает следующие операции с данными:

- Арифметические операции: сумма, разность, произведение и деление как со знаковыми, так и беззнаковыми операндами;
- Операции сравнения: равенство ( $==$ ), неравенство ( $!=$ ), больше-или-равно ( $>=$ ) и меньше ( $<$ ) как со знаковыми так и беззнаковыми операндами;
- Логические операции: И (AND), ИЛИ (OR), НЕ-ИЛИ (NOR), ИСКЛ (XOR)



- Операции сдвига и циклического сдвига: АЛУ поддерживает возможность сдвига и циклического сдвига данных от 0 до 31 позиции бита за инструкцию. АЛУ поддерживает арифметический сдвиг вправо и логический сдвиг вправо/влево. Также поддерживается циклический сдвиг вправо/влево.

В некоторых конфигурациях процессора Nios II отсутствует аппаратная поддержка некоторых инструкций. В таких реализациях инструкции называются нереализованными (*unimplemented*). Для выполнения такой инструкции процессор выдает исключение (*exception*) и вызывает процедуру, которая эмулирует необходимую операцию программно. Такие инструкции никак не влияют на то, как программист видит процессор.

### 3.2.3. Контроллеры исключений и прерываний

Процессор Nios II содержит аппаратный модуль для обработки исключений, включая аппаратные прерывания. Процессор также имеет опциональный интерфейс для внешнего контроллера прерываний **ЕІС** (*External Interrupt Controller*). Интерфейс ЕІС позволяет нам ускорить обработку прерываний в сложной системе с помощью пользовательского контроллера прерываний.

#### *Контроллер исключений*

Архитектура Nios II предоставляет простой скалярный контроллер исключений для обработки всех типов исключений. Каждое исключение, включая внутренние аппаратные прерывания, заставляет процессор во время выполнения программы перейти к адресу исключения. Обработчик исключений по этому адресу определяет причину исключения и отправляет на соответствующую процедуру исключения. Адреса исключений определяются в Platform Designer.

#### *Встроенный контроллер прерываний*

Архитектура Nios II поддерживает 32 внутренних аппаратных прерывания. Ядро процессора содержит 32 чувствительных к уровню входа для запроса прерываний (*IRQ*), *irq0-irq31*, предоставляя индивидуальный вход для каждого источника прерывания. Приоритет между *IRQ* определяется программно. Архитектура поддерживает вложенные прерывания.

### 3.2.4. Память и организация ввода/вывода

Гибкая природа памяти и организации системы ввода/вывода в процессоре Nios II – это одни из самых заметных отличий процессора Nios II от классических микроконтроллеров (рисунок 3.2). Процессор Nios II имеет следующие возможности взаимодействия с памятью и другой периферией:

- Мастер-порт инструкций – мастер-порт типа Avalon Memory-Mapped (АММ), соединяющий процессор с памятью инструкций;

- Кэш инструкций – быстрая кэш-память внутри ядра Nios II;
- Мастер-порт данных – АММ мастер-порт для соединения с памятью данных и периферией;
- Кэш данных – быстрая кэш-память внутри ядра Nios II;
- Порт для тесно-связанной памяти инструкций или данных – интерфейс процессора Nios II для доступа к внешней быстрой памяти.

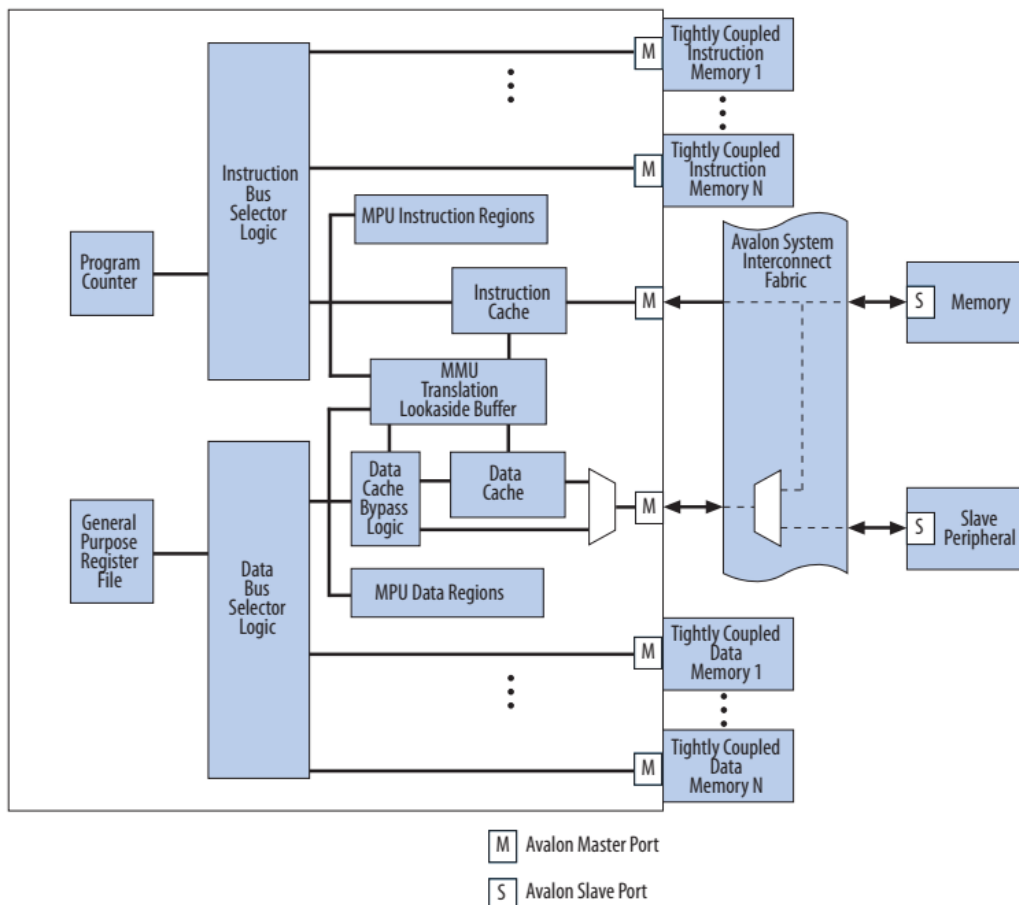


Рисунок 3.2 Схема подключения элементов памяти и периферии

### Шины инструкций и данных

Архитектура процессора Nios II предоставляет отдельные шины инструкций и данных, определяя их как Гарвардскую архитектуру. И шина инструкций, и шина данных реализованы как мастер-порты типа Avalon-MM, которые придерживаются спецификации интерфейса Avalon-MM. Мастер-порт данных подключается и к элементам памяти, и к элементам периферии, в то время как порт инструкций подключается только к компонентам памяти.

---

! Шины инструкций и данных имеют совместную таблицу адресов. Модель памяти организована так, что инструкции и данные находятся в одном  
● адресном пространстве.

---

Единственной задачей мастер-порта инструкций является выборка инструкций для их выполнения процессором. Данный порт не выполняет какие-либо операции записи.

### *Доступ к памяти и данным*

Архитектура Nios II предоставляет доступ к периферии через отображение в память. И данные из памяти, и периферия отображены в адресном пространстве мастер-порта данных. Таким образом, архитектурно нигде не указано что-либо о существовании памяти и периферии; количество, тип и размер памяти и периферии зависит от системы к системе: по одному и тому же адресу в одном случае может лежать доступ к периферии, а в другом и вовсе отсутствовать какой-либо элемент.

Мастер-порт данных выполняет две функции: чтение из памяти или периферии при выполнении инструкции загрузки и запись данных в память или периферию при выполнении инструкции хранения.

### *Кэш-память*

Архитектура Nios II поддерживает кэш-память как для инструкций, так и для данных. Использование такой памяти позволяет улучшить среднее время доступа к памяти в случае использования в системе медленной памяти, такой как SDRAM, для хранения программы и данных.

Кэши инструкций и данных доступны постоянно в течение работы процессора, однако методы, используемые программами для обхода кэша данных для доступа к периферии, не возвращают кэшированные данные. Управление кэшем и когерентность кэша управляются программно.

Блоки кэша являются дополнительной опцией, их наличие и размер определяются при конфигурировании ядра Nios II в Platform Designer.

---

! На этапе аппаратной сборки системы на основе Nios II вы могли заметить, что адресное пространство элементов памяти и периферии объединено  
● (рисунок 2.11), то есть, когда процессор взаимодействует с данными по определенному адресу, он не знает элемент памяти это или периферия (подробнее организация памяти рассматривается в следующем разделе).

---

Конструкции языка Си позволяют взаимодействовать с данными в памяти посредством *указателей*:

```
int *ptr= SWITCH_BASE;           //0x00011010
int val;

val = *ptr;
```

В приведенном примере создается указатель на базовый адрес переключателей и передается значение переключателей в переменную, которая может использоваться в дальнейшем в приложении; аналогичная операция может производиться и для записи данных. Однако на практике такой подход может вызвать ошибки в работе системы при использовании в качестве платформы более продвинутые процессоры, например, Nios II/f, в конфигурации которого присутствует *кэш*. Ошибка может быть вызвана тем, что процессор по умолчанию взаимодействует с внешней памятью через кэш. Поскольку при использовании указателей мы воспринимаем периферию как внешнюю память, то данные, передаваемые по указателю, на деле могут остаться в кэше для ускорения работы программы и так и не дойти до элемента периферии. Данная проблема вызвана спецификой аппаратной реализации подключения периферии. Для решения данной проблемы в архитектуре Nios II реализованы различные механизмы обхода кэша, которые скрыты непосредственно от разработчика программного обеспечения. (Подробнее описано в [2] раздел 2-16 *Cache Bypass Methods*)

### *Сильно-связанная память*

Сильно-связанная память, или память с сильной связью (*Tightly-Coupled Memory, TCM*), предоставляет возможность быстрого доступа к памяти для приложений, критичным по отношению ко времени выполнения. В сравнении с кэшем TCM предоставляет следующие возможности:

- Производительность на уровне кэша;
- Программа может гарантировать, что критичный по времени код или данные расположены в TCM;

- Отсутствие таких «накладок» в стиле кэша, как загрузка или очистка памяти;

Физически TCM порт – это отдельный опциональный мастер-порт ядра процессора Nios II, как, например, мастер-порт инструкций или данных. Ядро Nios II поддерживает TCM как для инструкций, так и для данных. К процессору Nios II можно подключить несколько TCM, для каждой из которых есть отдельный порт, что гарантирует низкую и фиксированную задержку доступа. По сути эта память является внешней по отношению к Nios II, тем не менее она находится на чипе.

TSM располагается в адресном пространстве, как если бы к процессору подключали любую другую память через Avalon, поэтому, с точки зрения разработчика приложений, способ доступа к TSM ничем не отличается от других.

#### *Блок управления памятью (MMU)*

При конфигурировании системы на базе Nios II блок управления памятью (*Memory Management Unit, MMU*) является опциональным и не обязателен в использовании. Многие системы на Nios II не требуют сложного набора периферии и большого объема памяти, а программы занимают незначительные объемы памяти. Однако с усложнением реализуемых задач и с использованием операционных систем необходимы решения задач виртуализации памяти, именно для таких задач и необходим блок MMU.

#### *Блок защиты памяти (MPU)*

Процессор Nios II предоставляет блок защиты памяти (*Memory Protection Unit, MPU*) для операционных систем и сред выполнения, требующих защиту памяти, но не требующих виртуализацию памяти.

- 
- ! В процессоре Nios II блоки MMU и MPU взаимоисключающие: при настройке включить можно только один из них.
- 

Когда MPU включен, он следит за всеми запросами инструкций процессора Nios II и доступом к памяти данных, чтобы защитить от некорректного выполнения программ. Блок MPU – это аппаратная возможность, использующаяся для программного определения разделов памяти и определения возможного к ним доступа. MPU запускает исключение, если программа пытается получить доступ к памяти, к которой ей запрещено обращаться, позволяя пользователю самостоятельно решать вопрос данного исключения. Подробнее можно узнать в документации к процессору [2] в разделах 3-1 и 3-33.

#### 3.2.5. Блок отладки JTAG

Для реализации задач эмуляции и управления процессором с компьютера архитектура процессора Nios II поддерживает отладочный модуль *JTAG (JTAG Debug Module)*. Программные отладочные средства на компьютере взаимодействуют с модулем JTAG и предоставляют такие возможности, как:

- загрузка программы в память;
- запуск и остановка выполнения программ;
- установка точек прерывания (*breakpoint*) и точек наблюдения, или контрольных точек данных (*watchpoint*);

- анализ регистров и памяти;
- сбор данных для отслеживания их в реальном времени.

Модуль отладки одной стороной соединяется с JTAG-частью самой ПЛИС для возможности внешнего доступа к процессору и другой стороной соединяется с самим процессором. Все системные ресурсы, доступные процессору в режиме суперпользователя (привилегированный пользователь, `superuser`), доступны и блоку отладки. Для отслеживания и записи наборов данных модуль отладки может сохранять данные как в памяти на кристалле, так и в памяти внешнего отладчика.

Конфигурируемость процессора Nios II предоставляет уникальную возможность: по завершению разработки системы отладчик может быть исключен из состава ядра для экономии пространства на кристалле.

## 4. ГОТОВЫЕ РЕШЕНИЯ ПЕРИФЕРИИ ДЛЯ NIOS II GEN2

### 4.1. Введение

Ядро процессора Nios II взаимодействует с периферией посредством интерфейса Avalon MM. Этот интерфейс имеет свои стандарты, подробно описанные в [4]. В общем случае данный документ описывает то, по каким правилам должно происходить взаимодействие модулей между собой для корректного обмена информацией. С другой стороны, переключатели и светодиоды не имеют как такового интерфейса подключения – каждому переключателю или светодиоду соответствует просто однонаправленный сигнал с двумя устойчивыми состояниями “ноль” или “единица”. Для корректного взаимодействия периферии с Nios II необходима некая оболочка, которая будет принимать необходимые команды по интерфейсу Avalon MM, с одной стороны, и устанавливать или считывать значения переключателей/светодиодов, с другой. В качестве такой оболочки используются специальные *IP-ядра*. Они абстрагируют разработчика от рутины разбора внутренних сигналов интерфейса, позволяя подключать необходимую периферию напрямую и управлять ею по заданным стандартом [4] правилам, в частности, с помощью процессора Nios II. В предыдущем разделе мы уже воспользовались некоторыми из них, в этом разделе мы рассмотрим наиболее часто используемые ядра, с помощью которых затем соберем улучшенную версию аппаратной реализации программы для управления светодиодами.

При рассмотрении каждого IP-ядра мы обратим внимание на следующее:

- *Функциональное описание*: крайне важно ознакомиться с сопутствующей информацией к каждому IP-ядру, чтобы понять, насколько оно подходит для использования в конкретной задаче;
- *Возможности настройки конфигурации*: например, количество подключенных к ПЛИС переключателей может быть разным, и в настройках интерфейса работы с переключателями должна быть возможность выбора ширины шины; в целом универсальность IP-ядра зависит от гибкости настройки его конфигурации;
- *Схема регистров*: подключаемые IP-ядра находятся в адресном пространстве, и со стороны процессора каждое из них видится как набор регистров, через которые производится взаимодействие с подключаемой периферией, а доступ к нему осуществляется через обращение к необходимому регистру по соответствующему адресу – это и входит в понятие схемы регистров.

В рамках данного учебного пособия мы рассмотрим следующие IP-ядра: *On-Chip Memory, PIO, JTAG UART, Interval Timer, SDRAM Controller* и *PLL*.

## 4.2. On-Chip Memory

IP-ядро On-Chip Memory предоставляет автоматическое создание области памяти типа RAM или ROM, к которой возможен доступ по интерфейсу Avalon MM. Данный блок позволяет использовать ячейки памяти прямо на том же кристалле, на котором реализуется процессор Nios II, а значит доступ к данным будет значительно быстрее по сравнению с памятью, подключаемой снаружи.

Окно настройки конфигурации представлено на рисунке 2.7. Обязательными полями являются:

- *Type* – в выпадающем списке можно выбрать тип подключаемой памяти: ROM (только для чтения) или RAM (для чтения и записи);
- *Slave S1 Data width* – в выпадающем списке можно выбрать битность данных, с которыми будут оперировать подключаемые элементы, для процессора Nios II это значение 32;
- *Block type* – эта настройка позволяет выбрать, какие внутренние ресурсы выбранной ПЛИС будут использованы для реализации памяти – специализированные блоки памяти (в нашем случае M10K) или регистры в составе базовых логических элементов ПЛИС (в нашем случае MLAB);
- *Total memory size* – в этом поле выставляется необходимый размер памяти в байтах;
- *Read Latency > Slave s1 Latency* – данный параметр позволяет выбирать абсолютную задержку в тактах на доступ к памяти. Большая задержка может повысить максимальную частоту работы интерфейса и может быть нужна для корректного чтения из памяти при превышения определённого порога её размера;
- *ROM/RAM Memory Protection Reset Request* – эта настройка создает дополнительный порт запроса на сброс для защиты памяти при сбросе;
- *Extend the data width to support ECC bits* – при необходимости можно расширить битность данных и добавить поддержку битов ECC (*Error Correction Code*), при этом без реализации логики их кодирования или декодирования.

Кроме того, есть дополнительные настройки:

- *Dual-port access* – включение данной настройки позволяет создать два входа для того, чтобы два устройства могли получить информацию из памяти одновременно;
  - *Single clock operation* – позволяет использовать два различных тактовых сигнала для двух разных портов;
- *Read During Write Mode* – настройка определения выходных данных (в случае двухпортовой памяти) при одновременных операциях чтения и записи по одинаковому адресу;



- *Enable different width for Dual-port access* – позволяет установить разную битность шин при включении настройки Dual port access;

- *Minimize memory block usage* – при выборе этой опции синтезатор будет использовать минимальное количество встроенных блоков памяти, что может негативно сказаться на временных параметрах схемы.

Наконец, память необходимо инициализировать, для чего есть отдельные настройки:

- *Initialize memory content* – включение возможности инициализации памяти пользователем;

- *Enable non-default initialization file* – возможность установки собственного файла для инициализации памяти;

- *Enable Partial Reconfiguration Initialization Mode* – поддержка возможности частичной реконфигурации (Partial Reconfiguration) – описание этой продвинутой опции в современных ПЛИС выходит за рамки данного пособия;

- *Enable In-System Memory Content Editor feature* – включение возможности использовать специальную утилиту для просмотра содержимого памяти.

### 4.3. PIO

IP-ядро для параллельного ввода/вывода – PIO (*Parallel input/output*) предоставляет интерфейс взаимодействия с вводом/выводом общего назначения (*GPIO*) через шину Avalon-MM. Помимо базового функционала, которым мы воспользовались в разделе 2.1.2, данное IP-ядро обладает рядом возможностей, необходимых при работе с периферией.

Окно настройки конфигурации представлено на 2.8 и содержит следующие настройки:

- Базовые:

- *Width* – это поле определяет битность шины данных, максимальный размер – 32;

- *Direction* – в зависимости от выбранного параметра блок может использоваться как для считывания входных данных, как для выдачи данных на выход, работать в двунаправленном режиме или иметь одновременно две шины: на вход и на выход;

- *Output Port Reset Value* – в случае настройки работы блока в любом режиме, кроме **Output**, имеется возможность установить значение, которое будет выводиться на шине данных по умолчанию;

- Настройка детектирования фронтов (доступно во всех режимах, кроме **Output**):

- *Synchronously capture* – поле включения возможности детектирования перепада входящего сигнала;
  - *Edge Type* – можно выбрать, какой тип перепада необходимо детектировать: по положительному фронту, по отрицательному фронту или по любому из них;
  - *Enable bit-clearing for edge capture register* – поле включения возможности сброса конкретных битов детектирования перепада.
  - Возможность генерации сигналов прерывания<sup>5</sup> (доступно во всех режимах, кроме **Output**):
    - *Generate IRQ*: поле включения возможности генерации запроса на прерывание;
    - *IRQ Type*: данная настройка позволяет выбрать причину прерывания, Level, если достаточна информация о высоком уровне сигнала на конкретном входе, или Edge, если необходимо выполнить запрос на прерывание по фронту.
- Управление работой IP- ядра осуществляется процессором посредством записи необходимых значений в соответствующие регистры, описание этих регистров приведено в таблице 4.1: здесь n – битность порта данных блока PIO.

Таблица 4.1 Схема регистров IP-ядра PIO

	Смещение	(n-1)	...	1	0
Данные	0	Входные/выходные данные			
Направление	1	Направление данных (для двунаправленного режима работы)			
Маска прерываний	2	Установка определенного бита в единицу в данном регистре означает, что прерывание будет вызываться по изменению сигнала в этом бите			
Детектирование фронта	3	При возникновении фронта в каком-либо из битов сигнала в соответствующем ему бит этого регистра устанавливается единица			
Установка в единицу	4	Регистр, запись данных в который установит в единицу значения соответствующих битов			
Сброс в ноль	5	Регистр, запись данных в который сбросит значения соответствующих битов			

<sup>5</sup> Использование прерываний будет подробнее рассмотрено в разделе 6.

#### 4.4.Interval Timer

При реализации первой программы для создания задержки во времени мы воспользовались пустым циклом, однако точность выставления такой задержки ограничена; кроме того, в период этой паузы процессор буквально «простаивал», так как в цикле не содержалось никакого полезного действия. Для работы с временными метками, а также чтобы отмерять довольно точно временные интервалы, во встраиваемых системах используются *счетчики*. Простейшей версией такого счетчика в каталоге IP-ядер является *Interval Timer*. Данный счетчик позволяет задавать величину, начиная с которой он начинает считать вниз и по достижению нуля выдавать сигнал об истечении установленного интервала времени; может работать в однократном или непрерывном режимах: останавливаться по истечению заданного периода или постоянно начинать отсчет заново; счетчик можно досрочно сбросить, поставить на паузу или запустить заново посредством регистров управления, а также выход счетчика может использоваться как сигнал для прерывания.

При настройке конфигурации Interval Timer содержит следующие поля (рисунок 4.1):

- Начальное значение периода:
  - *Period* – целочисленная величина, определяющая первоначальное значение счетчика;
  - *Units* – период счетчика можно определить как в микросекундах, так в миллисекундах, секундах или тактах системного генератора;
- *Timer Counter Size* – данная настройка определяет битность счетчика – 32 или 64 бита; в системе с тактовой частотой 100 МГц 32-битный счетчик может досчитать примерно до 43 секунд ( $2^{32} \times 10$  нс), а 64-битный счетчик может насчитать более 5 тыс. лет ( $2^{64} \times 10$  нс);
- Опциональные регистры:
  - *No Start/Stop control bits* – возможность отключения регистров управления запуском счетчика;
  - *Fixed period* – при выборе данного параметра задаваемый период счетчика не может быть изменен программно;
  - *Readable snapshot* – при включении этой настройки процессор может получать мгновенные «снимки» счетчика, при отключении процессор сможет определять, в каком состоянии находится счетчик, только с помощью регистра статуса или через прерывания;
- Опциональные порты:

- *Timeout pulse (1 clock wide)* – при включении данного порта в конфигурацию на нем появляется импульс в один такт каждый раз, когда счетчик доходит до нуля;
- *System reset on timeout (Watchdog)* – похожий по функционалу на предыдущий, используется для сброса всей системы по истечению определенного времени.

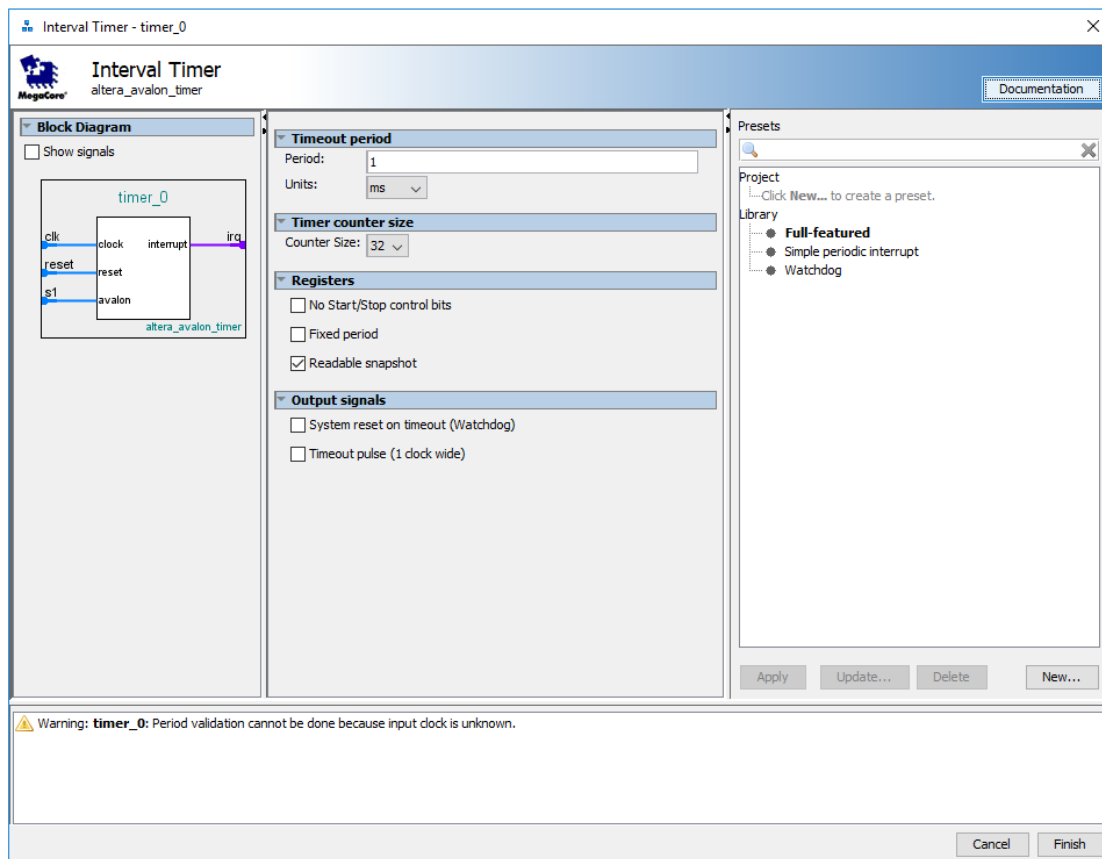


Рисунок 4.1 Окно настройки конфигурации счетчика

---

! В окне настройки конфигурации также есть поле справа, называемое *Presets*. В нем сохранены базовые предустановленные версии счетчиков, а также

- имеется возможность сохранить свою конфигурацию. Из доступных конфигураций есть *Full-featured* (выбран по умолчанию), которая содержит все необходимые регистры, *Simple periodic interrupt* – шаблон, в котором отключены все возможности управления счетчиком, и *Watchdog* – настройка счетчика в режиме сторожевого таймера.

---

Схема регистров блока Interval Timer представлена в таблице 4.2.

Таблица 4.2 Схема регистров блока Interval Timer

Смещение	15	...	4	3	2	1	0
Статус счетчика						run	to
Регистры управления				stop	start	cont	ito
Период, мл.	timeout period [15:0]						
Период, ст.	timeout period [31:16]						
Снимок, мл.	counter snapshot [15:0]						
Снимок, ст.	counter snapshot [31:16]						

Регистр статуса счетчика содержит два поля:

- run – в данном бите регистра записана единица, пока счетчик не достигает нуля – в таком случае бит обнуляется; данный бит регистра доступен только для чтения;
- to – как только счетчик достигает нуля, в этот бит записывается единица, сброс его значения достигается путем записи в него же нуля или единицы.

Регистр управления счетчиком содержит четыре поля:

- ito – в данном бите регистра устанавливается единица, если включена функция прерываний по счетчику, в противном случае бит равен нулю;
- cont – данный бит определяет режим работы счетчика– при записи в него единицы счетчик работает в непрерывном режиме, в противном случае счетчик работает в однократном режиме отсчета;
- start – запись единицы в этот бит запускает работу счетчика;
- stop – запись единицы в этот бит останавливает работу счетчика.

Регистры периода хранят в себе младшие и старшие 16 бит значения периода счетчика, а регистры снимка – старшие и младшие 16 бит текущего значения счетчика. В случае, когда при настройках выбирается 64-битный счетчик, битность регистров не меняется, но добавляются по 2 дополнительных регистра.

#### 4.5.JTAG UART

Простейшим способом обмена информацией для встраиваемых систем является *UART*<sup>6</sup> – буквально, *универсальный асинхронный приемопередатчик*. В

<sup>6</sup> UART – англ. *Universal Asynchronous Receiver and Transmitter*, является периферийным устройством ввода/вывода без пересылки тактового сигнала. Вместо этого системы должны заранее договориться о скорости

простейшей реализации – это два провода между устройствами: один на прием, другой на передачу данных. Однако на используемых нами платах отсутствует разъем для взаимодействия через UART, но присутствует другой вариант подключения между платой и компьютером – JTAG. Данный интерфейс позволяет взаимодействовать ПЛИС с компьютером благодаря контроллеру на стороне первого и специального сервера на стороне второго (рисунок 4.2).

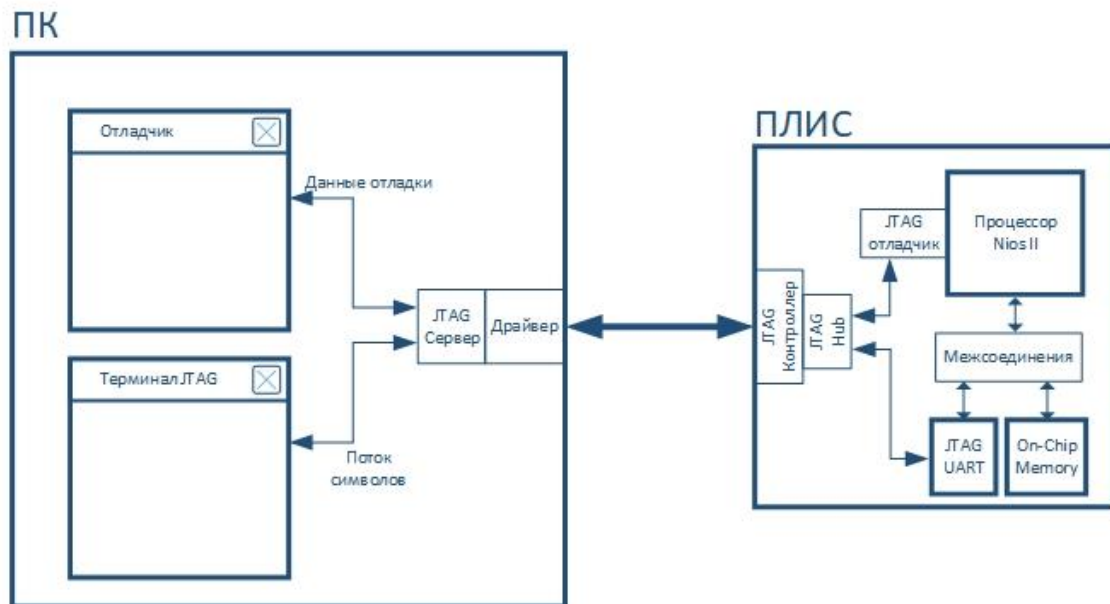


Рисунок 4.2 Схема взаимодействия между ПК и ПЛИС

По сравнению с UART интерфейс JTAG является достаточно сложным, и даже опытный разработчик может потратить значительное время, чтобы необходимый режим заработал надлежащим образом. В то же время для простейшего обмена информацией достаточно иметь низкоскоростной интерфейс взаимодействия с процессором. IP-ядро *JTAG UART* сделано для того, чтобы скрыть сложность JTAG-интерфейса от пользователя под видом символьного последовательного интерфейса, схожего с UART.

В настройках данного IP-ядра устанавливается глубина буферов FIFO чтения и записи данных (рисунок 4.3), а также порог срабатывания прерывания: в данном поле указывается количество символов в буфере FIFO, по достижению которого IP-ядро генерирует запрос на прерывание.

Для взаимодействия с IP-ядром процессор имеет доступ к двум 32-битным регистрам: регистру данных и регистру управления (таблица 4.3).

---

передачи данных. Хотя этот способ передачи данных обладает рядом недостатков, UART обеспечивает надежную асинхронную связь [6].

Таблица 4.3 Таблица регистров IP-ядра JTAG UART

Смещение	31...16	15	14...11	10	9	8	7	...	1	0	
Данные	0	ravail	Rvalid					data			
Управление	1	wspace					ac	wi	ri	we	re

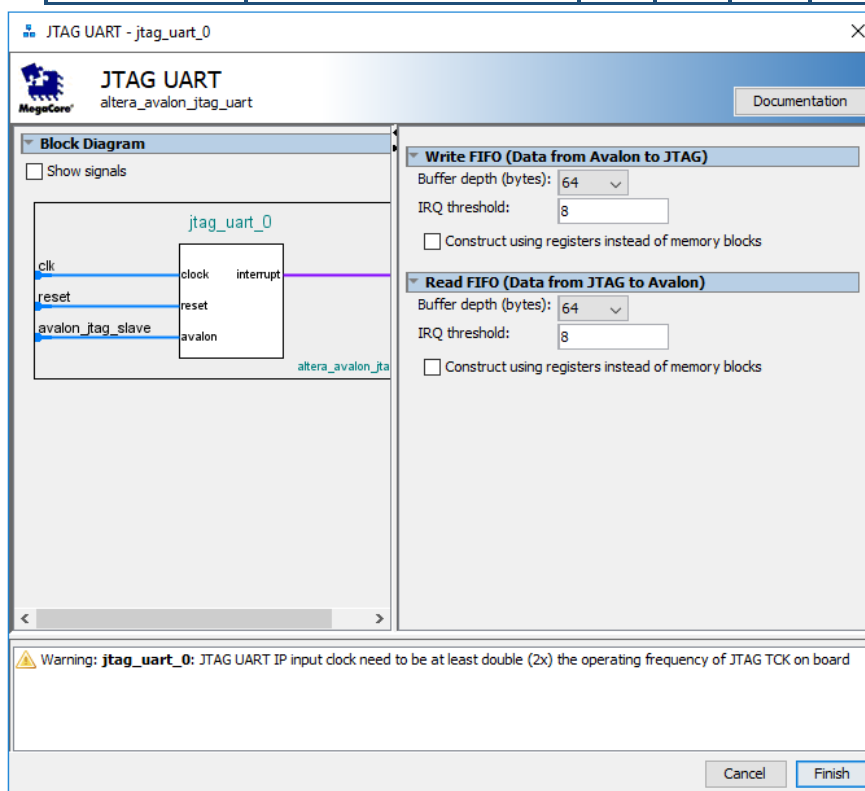


Рисунок 4.3 Окно настройки IP-ядра JTAG UART

Регистр данных содержит следующие поля:

- *data* – это поле содержит байт для передачи или приема данных. В процессе операции записи оно содержит символ, записываемый в буфер *write FIFO*. В процессе чтения оно содержит символ, считанный из буфера *read FIFO*;
- *rvalid* – этот бит равен единице, если поле *data* корректно, в противном случае значение в поле *data* не определено.
- *ravail*: – это поле содержит количество символов, оставшихся в буфере *read FIFO* (после текущего считывания).

Регистр управления блоком содержит следующие поля:

- *re* – в этот бит необходимо записать единицу, чтобы включить запрос на прерывания по чтению;
- *we* – этот бит необходимо установить в единицу, чтобы включить запрос на прерывания по записи;
- *ri* – данный бит показывает ожидание запроса на прерывание по чтению;
- *wi* – данный бит показывает ожидание запроса на прерывание по записи;

- `asc` – данный бит показывает была ли какая-либо активность по JTAG после того, как этот бит сбросили; для сброса данного бита необходимо в него записать единицу;
- `wspace` – это поле содержит объем свободного пространства в буфере *write FIFO*.

#### 4.6.SDRAM Controller

Внутри ПЛИС существует небольшое количество блоков памяти, которые мы ранее использовали с помощью IP-ядра *On-Chip Memory*, чтобы хранить программу и данные. Однако ПЛИС ограничена в своих возможностях, и более сложные и объемные программы уже нельзя будет уместить в рамках ПЛИС. Решением этой проблемы является подключение чипов внешней памяти. Этот чип имеет входы/выходы управления для корректного выполнения чтения и записи данных. В библиотеке IP-ядер имеется ядро *SDRAM Controller*, которое реализует низкоуровневое общение с чипом и производит корректную интерпретацию получаемых данных на шину Avalon-MM. Взаимодействие с SDRAM-памятью определяется настройками физических параметров (рисунок 4.4). На вкладке параметров подключенной памяти (*Memory profile*) указывается битность данных, архитектура памяти – количество чипов внутри SDRAM и количество банков – и параметры адресации: количество строк и столбцов.

На второй вкладке – *Timing* – указываются временные характеристики, которые необходимо соблюдать относительно различных управляющих сигналов для корректного взаимодействия с памятью. Все эти параметры указаны в документации (*datasheet*), поставляемой вместе с чипом. Тем не менее, некоторые величины указаны нестрого, поскольку на время распространения сигналов также влияет и разводка самой платы.

Некоторые уже готовые настройки для наиболее распространенных чипов SDRAM есть в наборе предустановок (*Presets*), аналогично со счетчиком. Можно также сохранять свои конфигурации в эти наборы для дальнейших проектов.



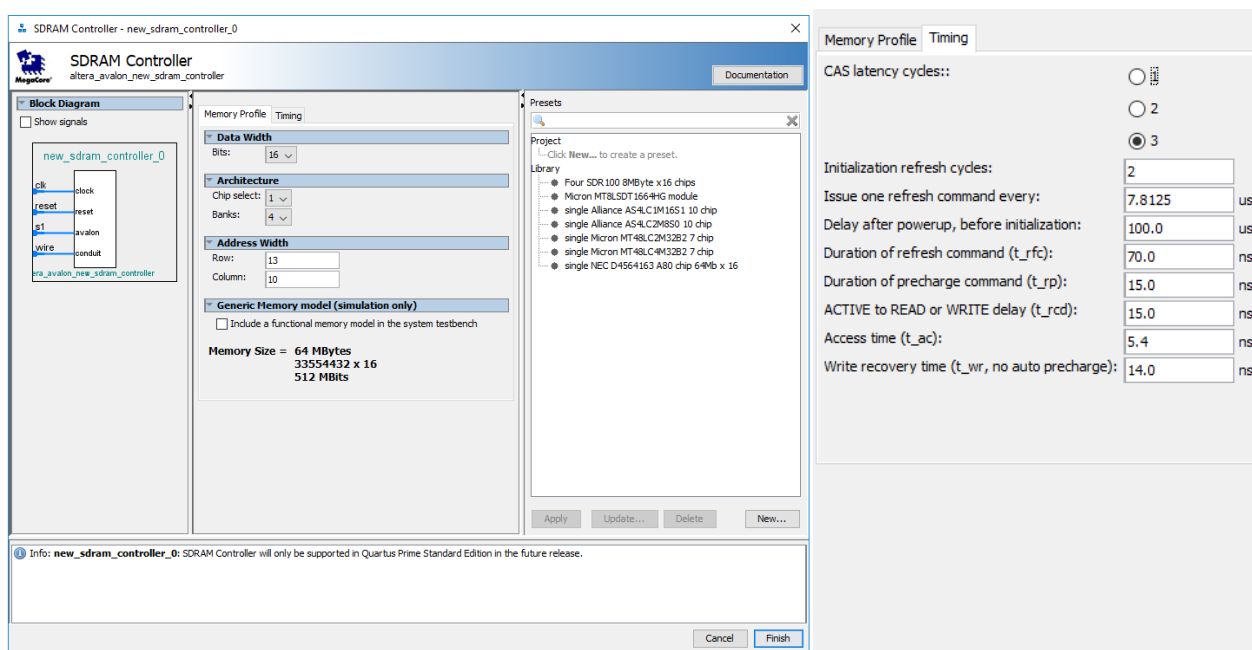


Рисунок 4.4 Настройка параметров памяти и временных ограничений в блоке SDRAM Controller

## PLL

В подавляющем большинстве случаев при построении систем на ПЛИС используются синхронные схемотехнические решения. В таких системах изменение состояний триггеров, переход от одного состояния конечного автомата к другому определяется сигналом тактирования. Тактовый сигнал – это сигнал, по положительному/отрицательному фронту которого синхронизируются события внутри цифровых электронных схем, производится ввод и вывод управляющих сигналов и данных. В упрощённом случае внутри одного устройства для синхронизации используется один единый тактовый сигнал. Это позволяет в большинстве случаев не тратить время на настройку фазы тактового сигнала.

Для создания сигналов тактовой частоты с необходимыми параметрами при проектировании на ПЛИС используют IP-ядра для управления внутренними блоками, именуемыми *PLL* (англ. *phase-locked loop*), в русской литературе ФАПЧ – блок *фазовой автоподстройки частоты*. IP-ядро позволяет настраивать PLL для генерации нескольких источников тактового сигнала с различными частотами и сдвигами фаз в различных режимах и даже имеет возможность регулировать параметры в процессе работы, т.е., например, в реальном времени изменять частоту или фазу тактового сигнала.

В нашей системе при реализации обмена данными между внешней микросхемой памяти и ПЛИС задача корректной синхронной передачи данных несколько усложняется. Мы имеем дело уже с передачей данных между двумя микросхемами, и в работе уже два тактовых сигнала – тактовый сигнал,

синхронизирующий работу системы с NIOS (NIOS II SYSTEM CLK), и тактовый сигнал, синхронизирующий работу SDRAM (SDRAM CLK), своим положительным фронтом (перепадом из 0 – в – 1) порождающий изменение данных на шине DATA (рисунок 4.5). Для того, чтобы данные, идущие от ПЛИС к памяти и от памяти к ПЛИС, корректно принимались (сленг “защёлкивались”), необходимо смещать тактовые частоты друг относительно друга. В нашей системе применяется смещение тактового сигнала SDRAM относительно тактового сигнала системы с NIOS на 3 нс.

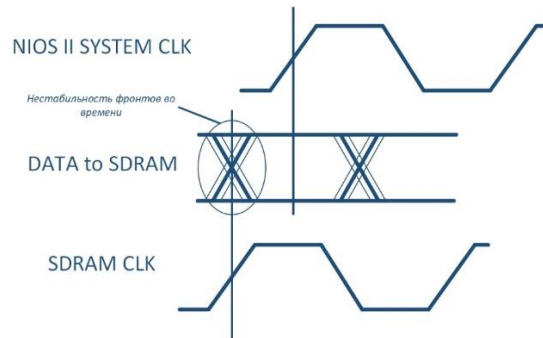


Рисунок 4.5 Временная диаграмма сигналов ПЛИС и SDRAM

В нашем случае такое решение является достаточным, чтобы гарантировать корректный обмен данными между ПЛИС и SDRAM. В более сложных случаях смещение подбирают исходя из характеристик каждого отдельного элемента ввода/вывода, с учётом длины соединительных дорожек на печатной плате, также осуществляется постоянный контроль импеданса линии, однако это выходит за рамки данного учебного пособия.

## 5. МОДИФИКАЦИЯ ПРОЕКТА `nios_load` С ИСПОЛЬЗОВАНИЕМ IP-ЯДЕР

В качестве демонстрации использования описанных ранее IP-ядер в данном разделе предлагается усовершенствовать собранную ранее систему со светодиодами: подключить SDRAM-память, добавить JTAG UART для вывода сообщений, воспользоваться PIO, чтобы подключить кнопки, которыми можно управлять программой, а также подключить семисегментные светодиоды-индикаторы для вывода отображения величины паузы между включениями светодиодов, воспользоваться IP-ядром счетчика для более точного отсчета времени и настроить межсоединения и распределение тактовых сигналов блока PLL.

### 5.1. Разработка аппаратной части

Процедура разработки аппаратной части нового проекта аналогична описанной в разделе 2.2, за исключением того, что необходимо добавить дополнительные блоки IP-ядер. Для упрощения предлагается в уже существующем проекте создать новую систему на базе процессора Nios II с использованием утилиты Platform Designer.

#### *Сборка основных элементов*

- Добавление процессора в новую систему аналогично этому же шагу в разделе 2.2.2.
- Добавление модуля системного идентификатора аналогично этому же шагу в разделе 2.2.2.
- Добавление модулей периферии схоже с этим же шагом, описанным в разделе 2.2.2. Добавление светодиодов и переключателей полностью совпадает с описанным ранее.
  - Добавим модуль PIO для подключения двух кнопок: в настройках при добавлении укажем битность (два бита), а также добавим возможность детектирования положительного фронта и возможность побитового сброса для регистра детектирования фронта и добавим возможность генерировать прерывания по перепаду сигналов данного блока. Переименуем его в `btn`.
  - Добавим модуль PIO для подключения массивов семисегментных индикаторов как 32-битную шину: на каждый сегмент в этой шине мы будем использовать байт данных (8 бит), что удобно при написании программы; переименуем блок в `sevs`.
- Для более точного измерения времени добавим блок счетчика, описанный в разделе 4.4. Все настройки предлагается оставить по умолчанию, кроме единиц времени: выберем микросекунды (`us`); переименуем блок в `timer`.

- Для взаимодействия с ПК через консоль JTAG добавим блок JTAG UART, оставим настройки по умолчанию; переименуем блок в `j_uart`.
- Добавление модуля памяти в новую систему производится с использованием блока SDRAM Controller, описанного в разделе 4.6. На плате DE10-Standard установлена и подключена к FPGA-части память типа SDRAM от фирмы ISSI модели IS42S16320D. Параметры конфигурации IP-ядра для подключения данной памяти приведены на 4.4. Для удобства требуется переименовать блок в `sdr`.
- Наконец, добавим источник тактовых сигналов и настроим их согласно описанию, данному в разделе 4.6: в качестве опорного источника тактового сигнала воспользуемся генератором на плате, его тактовая частота равна 50 МГц; в спецификации к модулю памяти указаны оптимальные частоты тактового сигнала, выберем из них частоту 143 МГц, с помощью блока PLL создадим два опорных источника со смещением по фазе одного из них относительно другого (рисунок 5.1). Ввиду ограничений внутренней архитектуры предупреждение показывает, что реальные значения будут немного отличаться от заданных. Переименуем блок в `pll`.

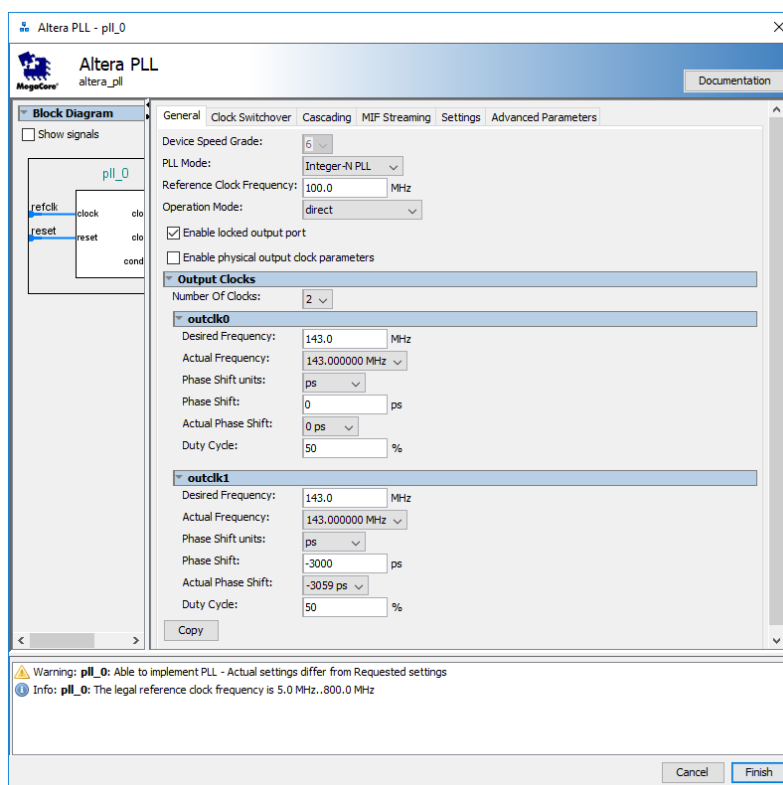


Рисунок 5.1 Основные настройки блока PLL для проекта `nios_load2`

Мы добавили все необходимые блоки в систему, следующим шагом нужно определить межсоединения:

- Входящий в систему сигнал тактовой частоты (`clk_0`) подключим от блока `clk_0` к блоку `pll`. Сигнал сброса (`reset`) из блока `clk_0` подключим к портам сброса всех модулей собираемой системы (за исключением блока `clk_0`: его порт сброса экспортирован).
- Порт тактового сигнала `outclk1` блока `pll` и порты внешнего подключения (*conduit*) модулей ввода/вывода (**PIO**) необходимо экспортировать, как это было сделано в разделе 2.2.2.
- Порт тактового сигнала `outclk0` блока `pll`, относительно которого `outclk1` «опережает», используем как основной тактовый сигнал и подключим к портам входа тактового сигнала у каждого блока (за исключением блока `clk_0`: его порт тактового сигнала экспортирован).
- Все модули необходимо подключить к процессору по интерфейсу Avalon-MM в порт `data_master`, а модуль памяти подключить также и к порту `instruction_master`, как это было сделано в разделе 2.2.2.
- Собираемую аппаратную схему мы будем использовать и в следующей главе для реализации задач прерываний, поэтому теперь необходимо порты `irq` модулей `btn`, `timer` и `j_uart` подключить к соответствующему порту `irq` процессора Nios II.
- В колонке IRQ таблицы с модулями расставим приоритеты прерываний в соответствии с рисунком 5.2.
- Для автоматического распределения адресного пространства в меню **System** выберем **Assign Base Addresses**.
- После определения базовых адресов всех блоков в модуле процессора Nios II необходимо переопределить вектора сброса и исключений, указав в качестве памяти блок `sdram`.

После выполнения всех перечисленных операций схема будет выглядеть так, как это представлено на рисунке 5.2.

Сохраним систему под именем `nios_load2.qsys`, после чего сгенерируем необходимые файлы, как это было в разделе 2.2.2, добавим `qsys`-файл в проект и создадим `bdf`-файл верхнего уровня. Для удобства подключения семисегментных индикаторов шину можно разбить на набор 7-битных шин и соединить между собой, обращаясь к имени шины, как показано в левом нижнем углу на рисунке 5.3.

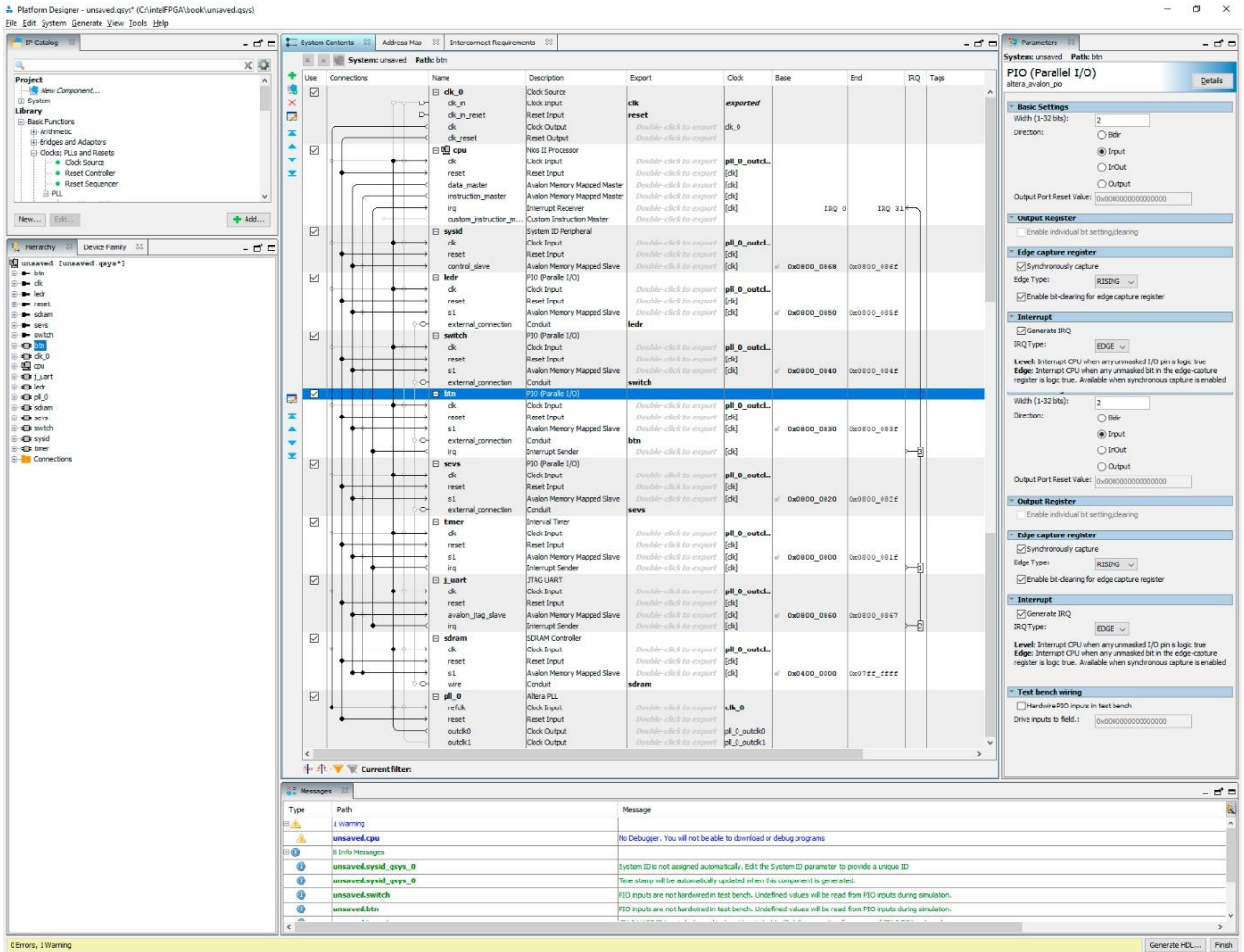


Рисунок 5.2 Содержимое системы nios\_load2 в Platform Designer

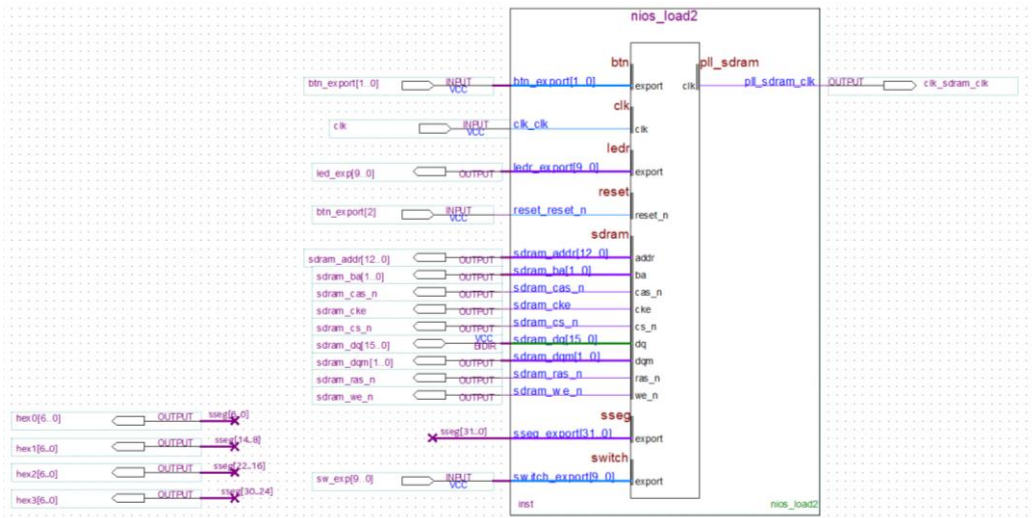


Рисунок 5.3 Графический файл верхнего уровня

Задать имя шины можно либо двойным нажатием левой кнопки мыши по изображению порта, либо, наведя указатель на порт, нажать правую клавишу и выбрать пункт **Properties**. Порты ввода/вывода можно создать также автоматически, для этого необходимо выбрать блок, для которого требуется сгенерировать порты (в нашем случае это `nios_load2`), нажать правой кнопкой мышки и выбрать пункт **Generate Pins for Symbol Ports**. Программа автоматически определяет, какой тип порта необходимо сгенерировать и какой битности будет порт, а также задает ему имя, аналогичное имени порта блока. После всех этих операций необходимо провести этап компиляции «синтез» и затем установить физическое соответствие созданным портам системы, как это было сделано в разделе 2.2.3. Эти данные указаны в приложении 1.

После этого можно производить полную компиляцию проекта, конфигурировать плату и переходить к программной части проекта.

## 5.2. Разработка программной части

Наличие новых элементов позволяет усовершенствовать ранее собранную систему включения светодиодов, сделать ее более наглядной и точной. Псевдокод новой программы выглядит следующим образом:

```
Инициализация системы;
Бесконечный цикл () {
    Считывание информации с кнопок и переключателей;
    Отображение информации на семисегментных индикаторах;
    Переключение светодиодов;
    Отображение информации в консоли;
}
```

Система стала гораздо более сложной, в зависимости от решаемой задачи код программы можно разделить на три уровня иерархии:

- Верхний уровень – главная программа;
- Средний уровень – процедуры решения для указанных задач;
- Нижний уровень – процедуры взаимодействия с периферией.

### 5.2.1. Процедуры драйверов

При разработке первого проекта мы изначально использовали базовые элементы взаимодействия с периферией. Несмотря на робастность этих решений, они достаточно громоздки и трудны для понимания, что негативно влияет на читаемость кода. Чтобы решить эти проблемы, принято использовать небольшие процедуры, которые скрывают от конечного разработчика громоздкую часть низкоуровневого функционала взаимодействия с периферией. Такие процедуры называются *драйверы*. Для удобного взаимодействия с периферией компания

Intel FPGA предоставляет набор библиотек и драйверов, называемых *Hardware Abstraction Layer*, или *HAL*.

### 5.2.2. HAL

Разработка систем с использованием HAL является неким средним, промежуточным звеном между двумя классическими парадигмами разработки приложений – приложений для ПК (англ. *desktop-like*) и так называемая *barebone system*, ближайшим русским переводом будет являться базовая система. В первом случае приложение разрабатывается с учетом еще одной «прослойки» – операционной системы – через которую осуществляется доступ к драйверам устройств. Критическим аспектом в такой системе является совместимость драйверов с операционной системой.

Barebone-система, как правило, представляет собой простую систему на базе микроконтроллера. Ввиду простоты в ней нет дополнительных слоев между приложением пользователя и аппаратным обеспечением системы, поэтому управление периферией осуществляется напрямую из программы. Разработчик может самостоятельно разработать драйвера устройств или воспользоваться готовыми.

В такой классификации HAL является неким промежуточным звеном: с одной стороны, он позволяет избежать прямого взаимодействия с аппаратной частью проекта, с другой стороны, использование HAL не является обязательным, и пользователь имеет полный доступ к периферии без дополнительных прослоек (рисунок 5.4).



Рисунок 5.4 Условное расположение HAL при разработке встраиваемых систем для Nios II



Набор библиотек HAL предоставляет следующие возможности:

- Объединение со стандартными библиотеками **newlib** в ANSI C, что позволяет использовать знакомые разработчикам встраиваемых систем конструкции языка для решения задач;
- Готовые драйверы стандартных устройств;
- HAL API (application programming interface) – набор процедур для таких задач как доступ к устройству, обработка прерывания и пр.
- Инициализация системы – выполнение различных задач процессора и периферии перед выполнением основной части программы.

Также HAL предоставляет набор готовых моделей взаимодействия с периферией, такими устройствами являются:

- Устройства, работающие в символьном режиме – периферия, которая принимает и/или передает информацию последовательно и посимвольно, например, UART;
- Счетчики – устройства, которые считают количество тактов (единиц системного времени процессора) и могут генерировать периодические запросы на прерывание;
- Файловые подсистемы – механизм доступа к данным, которые хранятся в физическом устройстве; в зависимости от внутренней реализации устройства возможен доступ напрямую или с использованием отдельного драйвера устройства;
- Ethernet-устройства – устройства для подключения через Ethernet;
- Устройства прямого доступа к памяти (DMA – англ. Direct Memory Access) – периферия, которая выполняет работу по переносу больших объемов данных, при этом источник и приемник могут быть как элементом памяти, так и любым другим устройством, например, устройством Ethernet;
- Устройства энергонезависимой памяти (flash) – память, использующая специальный протокол, чтобы хранить данные.

Подробнее об этом и об остальных аспектах использования HAL при разработке встраиваемых систем можно прочитать в [5].

### 5.2.3. BSP

Инструментарий HAL включает в себя постоянный комплект библиотек и сам по себе входит в набор утилит, который называется *BSP* (англ. *Board support package*). В разделе 2.3.1 этот набор был сгенерирован автоматически с настройками по умолчанию. Однако в зависимости от решаемых задач различные настройки BSP могут быть изменены. Для этих целей воспользуемся графическим

интерфейсом инструмента **BSP Editor**: в среде разработки **Nios II SBT for Eclipse** выберем: **Nios II > BSP Editor**. В появившемся окне выбрать **File > New Nios II BSP**, указать **sopcinfo**-файл системы, для которой будет генерироваться набор библиотек (рисунок 5.5).

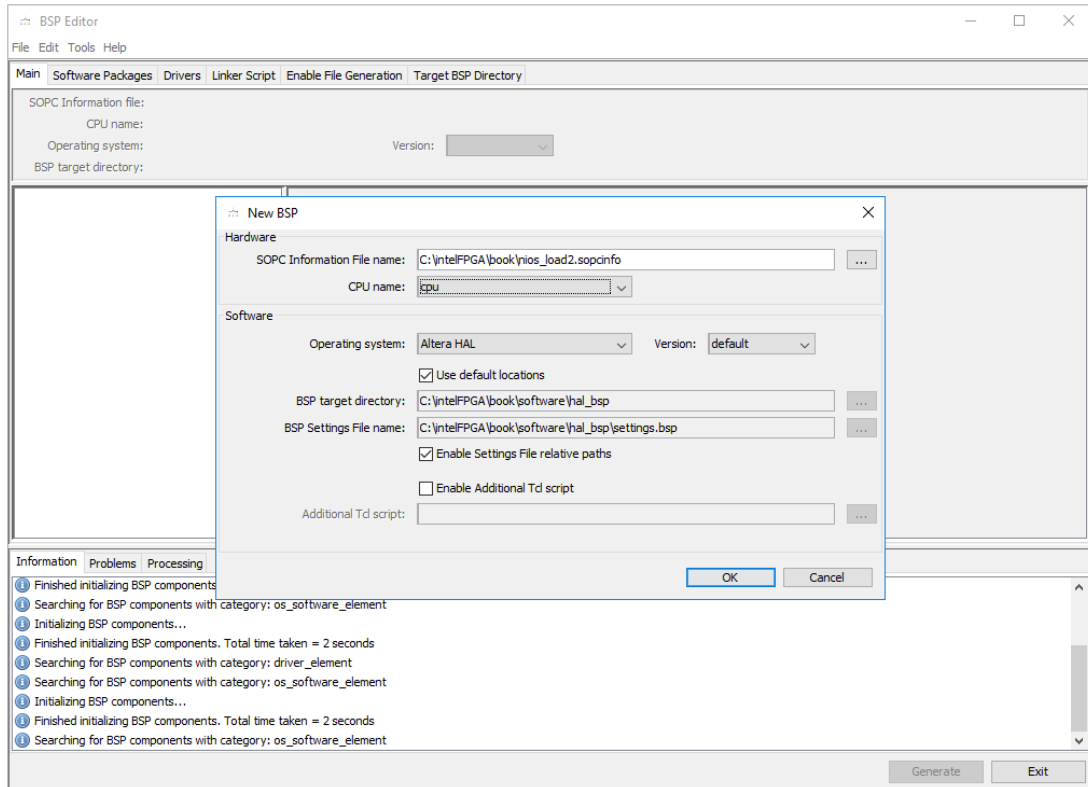


Рисунок 5.5 Создание нового BSP

При необходимости можно снять галочку **Use default locations** и изменить расположение директории для файлов BSP. После нажатия кнопки **Ok** программа автоматически сгенерирует пакет BSP с настройками по умолчанию. В окне станут активны все вкладки и появится доступ к настройкам. В рамках данного пособия мы рассмотрим базовые настройки, более подробную информацию можно найти в [5].

На главной вкладке – **Main** – расположена информация о BSP, а также основные настройки, они представлены в дереве настроек в левой части экрана и разделены на основные (**Common**) и дополнительные (**Advanced**). В основной, или базовой, группе настройки разделены следующим образом (рисунок 5.6):

- **hal** – базовые настройки параметров HAL – определение системного счетчика и счетчика для временных отметок, выбор устройства для работы в символьном режиме и оптимизация библиотек по размеру (использование уменьшенных библиотек Си), добавление профилирования с использованием

gprof, использование «урезанных» драйверов устройств и оптимизация компиляции BSP для ускорения проведения симуляций hdl-кода;

- `hal.linker` – раздел настройки linker-скрипта, который определяет разделы памяти и их размеры;
- `hal.make` – настройки сборки приложения: определение настройки отладчика, выбор уровня оптимизации и настройка доступа глобального указателя.

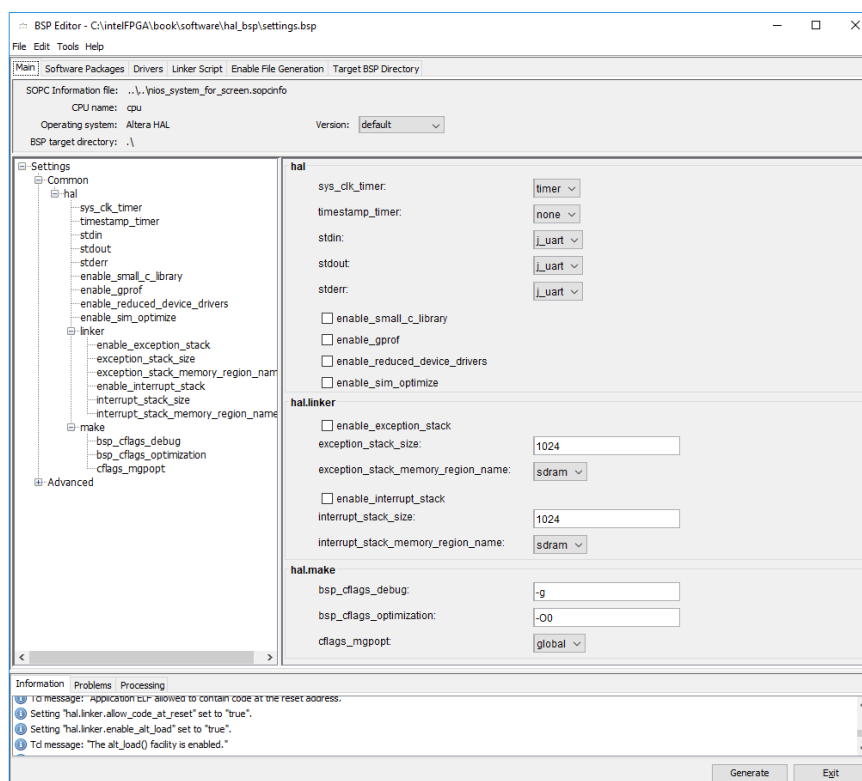


Рисунок 5.6 Базовые настройки BSP

В рамках данного пособия мы не будем рассматривать дополнительные настройки, о них можно узнать подробнее в [5].

На вкладке **Software Packages** расположены опциональные наборы программ для более удобного взаимодействия с периферией; такие наборы, как правило, поставляются производителями периферии.

Вкладка **Drivers** позволяет выбирать, какие драйверы устройств необходимо добавить в систему, а какие можно убрать. Кроме этого, у каждого элемента периферии может быть набор различных драйверов и их версий, что можно также определить в выпадающем списке в таблице у каждого из элементов.

На вкладке **Linker Script** представлена более подробная информация о разметке памяти, которая производится с использованием linker-скрипта. Данный инструмент позволяет с использованием графического интерфейса переназначить

области адресации, добавить их или удалить, а также редактировать информацию о доступных элементах памяти.

Вкладка **Enable File Generation** предоставляет доступ к ручному выбору файлов, которые необходимо сгенерировать для дальнейшей разработки приложений.

На последней вкладке **Target BSP Directory** расположена ознакомительная информация о том, какие файлы по итогу всех внесенных изменений будут сгенерированы в выбранной директории.

В приложении `nios_load_hal` будут использоваться различные возможности блока **Interval Timer**, поэтому необходимо убедиться, что он выбран в качестве системного счетчика в поле `sys_clk_timer`, а модуль `j_uart` выбран как устройство стандартного ввода/вывода в полях `stdin`, `stdout` и `stderr`.

После настройки необходимо сгенерировать файлы, нажав клавишу **Generate**. Теперь можно закрыть BSP Editor и приступить к разработке основного приложения.

#### 5.2.4. Приложение `nios_load_hal`

Вернувшись обратно в интерфейс Nios II SBT for Eclipse, необходимо создать новое приложение: **File > New > Nios II Application**. В поле BSP location нужно указать путь до BSP (рисунок 5.7), который был сгенерирован в выше разделе. Затем нужно нажать кнопку **Finish**.

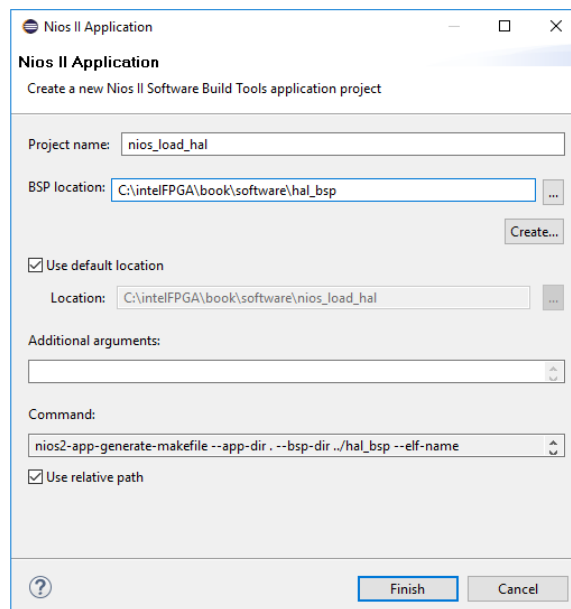


Рисунок 5.7 Параметры для создания приложения `nios_load_hal`

В начале этого раздела мы описали базовые блоки системы, каждый из них можно либо реализовать в форме функции, которая будет вызываться в теле `main()`, либо можно воспользоваться функциями HAL для решения задачи.

Для удобства передачи информации между функциями создадим структурный тип данных с двумя полями: статус установки паузы (от англ. *hold* – удержание) и время включения между светодиодами (от англ. *duration* - длительность):

```
typedef struct cmnds{
    int hold;
    int dur;
} cmd_type;
```

### *Инициализация системы*

Одной из возможностей HAL является инициализация, другими словами, предварительная настройка окружения перед выполнением главной программы. Для инициализации в первую очередь выполняется файл `crt0.S`. Он содержит базовые операции ассемблера, которые выполняют сброс кэшей инструкций и данных, настройку регистров указателей, сброс `bss` после чего выполняет файл `alt_main.c`. В нем инициализируется контроллер прерываний, вызывается функция `alt_sys_init()` для инициализации драйверов периферии по умолчанию, переопределяется символьный ввод/вывод (`stdin`, `stdout` и `stderr`) согласно ранее выбранному в BSP Editor, вызывается конструктор C++ и затем вызывается главная функция `main()`.

В нашу систему добавим дополнительные условия инициализации: обнулим с самого начала регистры детектирования фронта, установим счетчику период работы и зададим необходимый режим:

---

```
. . .
#include "io.h"
#include "alt_types.h"
. . .
void init_sys(alt_u32 btn_base, alt_u32 timer_base)
{
    int period = 1; //период в миллисекундах
    //Инициализация счетчика: период и режим работы
    IOWR(timer_base, 3, (period>>16)); //старшие 16 бит
    IOWR(timer_base, 2, (period & 0x0000ffff)); //младшие 16
бит
    IOWR(timer_base, 1, 0x0006);
    IOWR(btn_base, 3, 0b11); //Сброс регистров перепада
}
```

---

### *Считывание информации с кнопок и переключателей*

Для работы с PIO автоматически сгенерированы файлы с макросами для взаимодействия с периферией. Таким образом, можно воспользоваться готовыми решениями для того, чтобы считать регистры фронтов сигналов нажатия кнопок и по их наличию в зависимости от того, какая кнопка нажата, поставить переключение светодиодов на паузу или обновить значение периода между включениями светодиодов, считав значение со светодиодов.

---

```
. . .
#include "altera_avalon_pio_regs.h"
. . .
void pio_info(alt_u32 btn_base, alt_u32 sw_base, cmd_type
*cmd)
{
    alt_u8 btn;

    btn = (alt_u8) IORD_ALTERA_AVALON_PIO_EDGE_CAP(btn_base)
& 0b11;    // считывание наличия произошедших перепадов на
кнопках
    if (btn!=0){
        if (btn & 0b01)                // нажата первая кнопка
            cmd->hold = cmd->hold ^ 1; // изменить состояние паузы
        if (btn & 0b10)                // нажата вторая кнопка
            cmd->dur=IORD_ALTERA_AVALON_PIO_DATA(sw_base)&0x03ff;
// изменить паузу между включениями
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(btn_base, 0b11);
        // обнулить регистры детектирования фронта
    }
}
```

---

### *Отображение информации на семисегментных индикаторах*

Семисегментный индикатор – это устройство отображения информации, визуализирующее десятичные цифры, т.е. цифры от 0 до 9. По сути, это семь светодиодов (иногда 8 светодиодов, восьмой – точка-разделитель для отображения дробной части), расположенных так, что включение определенного набора определяет визуальное отображение арабских цифр. На рисунке 5.8 показан расширенный набор, отображающий набор шестнадцатеричных цифр.

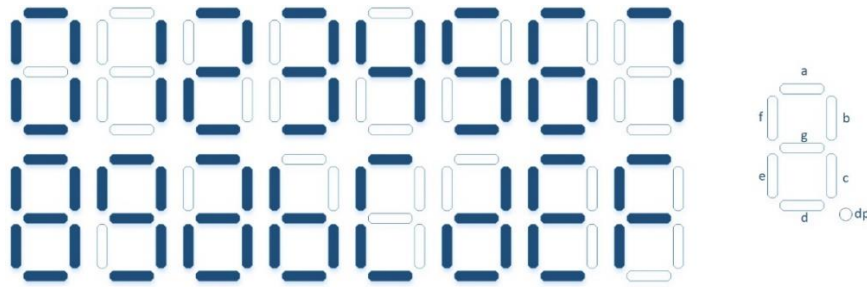


Рисунок 5.8 Паттерны для шестнадцатеричных значений

Существуют несколько вариантов подключения наборов семисегментных индикаторов к устройству управления, например, когда одни и те же сегменты подключены к одному выводу и плюс каждый индикатор подключен к отдельному выводу управления питанием на сегментах (рисунок 5.9). Такой вариант подключения используют для экономии количества используемых портов.

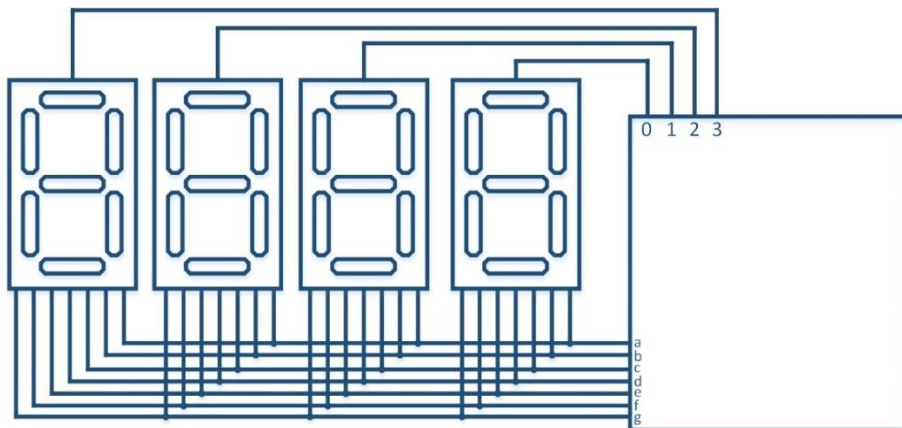


Рисунок 5.9 Альтернативный вариант подключения семисегментных индикаторов

На плате DE10-Standard каждый сегмент каждого индикатора подключен отдельно, поэтому достаточно выставить на выход контроллера SEVS необходимую последовательность для включения соответствующих светодиодов. Последовательность определяется двоичным кодом, который задается переключателями на плате. На каждом из трех индикаторов необходимо отобразить единицы, десятки и сотни соответственно, а на четвертом отображать символ паузы в случае, когда нажата кнопка паузы. При значении на переключателях больше 999 условимся продолжать отображать 999.

Основной задачей в данной функции является отделить друг от друга единицы, десятки и сотни и выставить эти значения на соответствующие индикаторы. В рамках данной главы рассмотрим алгоритм **Double-Dabble**. Этот алгоритм относится к группе «сдвинуть-сложить» алгоритмов и предназначен для преобразования двоичного числа в двоично-десятичный вид при помощи операций

сдвига и сложения. Несмотря на то, что есть более простые способы – например, используя остаток от деления – алгоритм Double-Dabble является более интересным для рассмотрения. Алгоритм выполняется итерационно, каждая итерация начинается с операции сдвига влево. Преобразуемое число «вдвигается» по битам в буфер, каждые 4 бита которого впоследствии будут обозначать цифру преобразуемого числа. По завершении операции сдвига содержимое каждой цифры проверяется: если оно больше 4, то к значению цифры прибавляют 3. Количество итераций зависит от того, сколько цифр предполагается отображать. Пример для двузначного числа 73 представлен в таблице 5.1.

Таблица 5.1 Пример работы алгоритма Double-Dabble для десятичного числа 73

Десятки	Единицы	Число	Операция
0000	0000	0100 1001	Начало
0000	0000	1001 0010	Сдвиг 1
0000	0001	0010 0100	Сдвиг 2
0000	0010	0100 1000	Сдвиг 3
0000	0100	1001 0000	Сдвиг 4
0000	1001	0010 0000	Сдвиг 5
0000	1100	0010 0000	Прибавить 3 в единицы, т.к. больше 4
0001	1000	0100 0000	Сдвиг 6
0001	1011	0100 0000	Прибавить 3 в единицы, т.к. больше 4
0011	0110	1000 0000	Сдвиг 7
0011	1001	1000 0000	Прибавить 3 в единицы, т.к. больше 4
0111	0011	0000 0000	Сдвиг 8
7 <sub>10</sub>	3 <sub>10</sub>		

Паттерны зажигания индикаторов неизменны, поэтому их можно хранить в массиве, в котором порядковый номер каждого элемента соответствует отображаемой цифре. После определения цифр необходимо все эти значения записать в регистр с учетом нажатой кнопки паузы: если она нажата, то на старшем индикаторе необходимо отобразить букву «П».

---

```
void sevs_ind(alt_u32 sseg_base, cmd_type cmd)
{
    static const alt_u8 SEVS_VALUES[16] = {
        0x40, 0x79, 0x24, 0x30, 0x19, 0x92, 0x02, 0x78, 0x00,
        0x10, // 0-9
        0x88, 0x03, 0x46, 0x21, 0x06, 0x0E}; // a-f
    int pd,msg;
    int i=0;
```



```

int bcd=0b0000000000000;
int mask=0b1000000000000;
int a=0,b=0,c=0,buf1,buf2;

if (cmd.dur > 999) // 999 - максимум отображения
    pd = 999;
else
    pd = cmd.dur;

//алгоритм double-dabble
for (i=0;i<12;i++){
    if(a>4){a = a+3;}
    if(b>4){b = b+3;}
    if(c>4){c = c+3;}
    bcd = a | (b<<4) | (c<<8);
    buf1 = (bcd&0x7ff);
    buf1 = buf1 << 1;
    buf2 = (pd&mask);
    buf2 = buf2 >> (11-i);
    bcd = buf1 | buf2;
    mask = mask >> 1;
    a = bcd & 0x00f;
    b = (bcd & 0x0f0)>>4;
    c = (bcd & 0xf00)>>8;
}

msg = (SEVS_VALUES[a]) | (SEVS_VALUES[b]) << 8 |
(SEVS_VALUES[c]<<16);
if (cmd.hold)
    msg = 0x48<<24 | msg; // знак паузы
else
    msg = 0xff<<24 | msg; // пустой
IOWR_ALTERA_AVALON_PIO_DATA(sseg_base, msg);
}

```

---

### *Переключение светодиодов*

Условие переключения светодиодов и паузы между переключениями вынесем в отдельную функцию. Алгоритм переключения аналогичен описанному в разделе 2.4, а для реализации паузы можно воспользоваться различными функциями HAL:

- `usleep(unsigned int t)`: останавливает выполнение кода на `t` микросекунд;

- `alt_nticks()`: возвращает количество тактов, которое прошло с момента последнего сброса;
- `alt_ticks_per_second()`: возвращает количество тактов, которое происходит за секунду.

В текущей реализации предлагается применить первый вариант:

---

```
void led_line(alt_u32 led_base, cmd_type cmd)
{
    alt_u16 led_pattern;
    static alt_u32 i = 0x1;

    if (cmd.hold)
        return;
    if (i >= 0x800) {
        led_pattern = 0x00;
        i = 0x01;
    } else {
        led_pattern = i-1;
        i = (i<<1);
    }
    IOWR_ALTERA_AVALON_PIO_DATA(led_base, led_pattern);
    usleep(1000*cmd.dur);
}
```

---

### *Отображение информации в консоли*

Для работы с JTAG UART автоматически генерируются файлы драйверов, в которых прописаны регистры и базовые функции. Тем не менее, при настройке BSP в качестве устройства символьного вывода в предыдущем разделе был указан модуль `j_uart`. Для ввода/вывода воспользуемся стандартными конструкциями языка Си. В итоге файл `main.c` выглядит следующим образом:

---

```
#include <stdio.h>
#include <unistd.h>
#include "io.h"
#include "alt_types.h"
#include "system.h"
#include "altera_avalon_pio_regs.h"

typedef struct cmnds{
    int hold;
    int dur;
} cmd_type;
```

```

void init_sys(alt_u32 btn_base, alt_u32 timer_base){
    int period = 1; //период в миллисекундах
    //Инициализация счетчика: период и режим работы
    IOWR(timer_base, 3, (period>>16));
    //старшие 16 бит
    IOWR(timer_base, 2, (period & 0x0000ffff));
    //младшие 16 бит
    IOWR(timer_base, 1, 0x0006);
    //Сброс регистров детектирования перепада
    IOWR(btn_base, 3, 0b11);}

void pio_info(alt_u32 btn_base, alt_u32 sw_base, cmd_type
*cmd){
    alt_u8 btn;

    btn = (alt_u8) IORD_ALTERA_AVALON_PIO_EDGE_CAP(btn_base)
& 0b11; // считывание наличия произошедших перепадов на
кнопках

    if (btn!=0){
        if (btn & 0b01) // нажата первая кнопка
            cmd->hold = cmd->hold ^ 1; // изменить состояние паузы
        if (btn & 0b10) // нажата вторая кнопка
            cmd->dur = IORD_ALTERA_AVALON_PIO_DATA(sw_base) &
0x03ff; // изменить паузу между включениями
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(btn_base, 0b11);}} //
обнулить регистры детектирования фронта

void sevs_ind(alt_u32 sseg_base, cmd_type cmd){
    static const alt_u8 SEVS_VALUES[16] = {
        0x40, 0x79, 0x24, 0x30, 0x19, 0x92, 0x02, 0x78, 0x00,
0x10, // 0-9
        0x88, 0x03, 0x46, 0x21, 0x06, 0x0E}; // a-f
    int pd,msg;
    int i=0;
    int bcd=0b00000000000000;
    int mask=0b10000000000000;
    int a=0,b=0,c=0,buf1,buf2;

    if (cmd.dur > 999) // 999 - максимум отображения
        pd = 999;
    else
        pd = cmd.dur;

```

```

//алгоритм double-dabble
for (i=0;i<12;i++){
    if(a>4){a = a+3;}
    if(b>4){b = b+3;}
    if(c>4){c = c+3;}
    bcd = a | (b<<4) | (c<<8);
    buf1 = (bcd&0x7ff);
    buf1 = buf1 << 1;
    buf2 = (pd&mask);
    buf2 = buf2 >> (11-i);
    bcd = buf1 | buf2;
    mask = mask >> 1;
    a = bcd & 0x00f;
    b = (bcd & 0x0f0)>>4;
    c = (bcd & 0xf00)>>8;}

msg = (SEVS_VALUES[a]) | (SEVS_VALUES[b]) << 8 |
(SEVS_VALUES[c]<<16);

if (cmd.hold)
    msg = 0x48<<24 | msg;           // знак паузы
else
    msg = 0xff<<24 | msg;         // пустой

IOWR_ALTERA_AVALON_PIO_DATA(sseg_base, msg);}

void led_line(alt_u32 led_base, cmd_type cmd){
    alt_u16 led_pattern;
    static alt_u32 i = 0x1;

    if (cmd.hold)
        return;
    if (i >= 0x800) {
        led_pattern = 0x00;}
        i = 0x01;
    else {
        led_pattern = i-1;
        i = (i<<1);}

    IOWR_ALTERA_AVALON_PIO_DATA(led_base, led_pattern);
    usleep(1000*cmd.dur);}

```

```

int main(){
    cmd_type sw_cmd={0,100}; // первоначальные значения паузы
и интервала
    int current;

    init_sys(BTN_BASE, TIMER_BASE);
    while(1){
        pio_info(BTN_BASE, SWITCH_BASE ,&sw_cmd);
        if (sw_cmd.dur!=current){ // детектирование изменения
            printf("Interval: %03u ms \n", sw_cmd.dur);
            current = sw_cmd.dur; // обновление значения
        }
        sevs_ind(SEVS_BASE, sw_cmd);
        led_line(LED_BASE, sw_cmd);
    }
}

```

По завершению написания программы необходимо повторить шаги, описанные в разделе 2.3.3. После загрузки программы на плату на семисегментных индикаторах отобразится значение по умолчанию (100). При нажатии на кнопку 0 переключение светодиодов остановится, появится буква «П», при повторном нажатии работа возобновится. По нажатию на кнопку 1 время переключения светодиодов изменится, и это отобразится на семисегментных индикаторах. Фрагмент работы кода на плате представлен на рисунке 5.10.

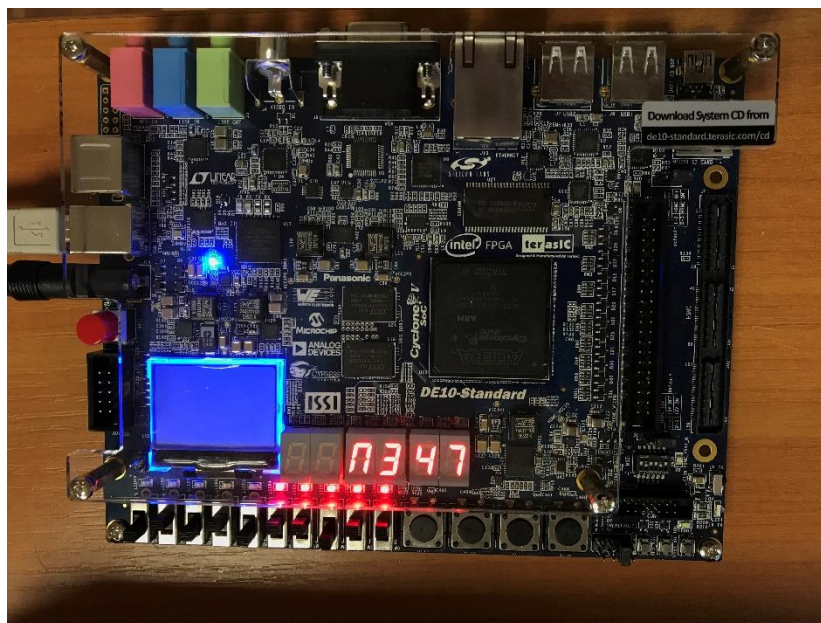


Рисунок 5.10 Фрагмент работы программы `nios_load2` на отладочной плате

## 6. ПЕРЫВАНИЯ и ISR

Архитектура одноядерных микроконтроллеров и микропроцессоров предполагает последовательное выполнение операций: код, записанный в память, выполняется шаг за шагом, инструкция за инструкцией. Однако зачастую в программе могут появляться задачи, которые являются наиболее важными к исполнению, т.е. у них более высокий *приоритет* выполнения. В системе типа round-robin, которую мы реализовывали в предыдущих разделах, все задачи имеют одинаковый приоритет и выполняются строго друг после друга. В данной главе мы оптимизируем программу за счёт использования *прерываний*.

### 6.1. Введение

*Исключение (exception)* – это ситуация, при которой выполняется остановка выполнения главного кода и переход к выполнению другой задачи, что вызвано событием – внутренним или внешним, требующим немедленной обработки. По умолчанию все исключения в процессоре Nios II делятся на следующие категории:

- Исключение по сбросу (*reset*): возникает при сбросе процессора. Выполнение автоматически переходит к адресу сброса, который указывается при настройке процессора;
- Исключение по остановке (*break*): может возникнуть при использовании точек останова в модуле отладки **JTAG Debug**;
- Исключение по прерыванию: возникает вследствие сигналов, полученных от периферии, которые необходимо обработать вне очереди;
- Программное исключение: могут возникать вследствие использования неопределенных команд или каких-либо некорректных событий, таких как деление на ноль,

В рамках данного учебного пособия мы подробнее рассмотрим реализацию прерываний с использованием процессора Nios II.

### 6.2. Прерывания в Nios II

Архитектура процессора Nios II имеет *встроенный контроллер прерываний* (англ. *Internal Interrupt Controller*), который мы рассмотрим далее и будем использовать в данной главе, однако, помимо этого, имеется поддержка использования внешнего контроллера прерываний.

При использовании встроенного контроллера прерываний устройства периферии могут подключаться к одному из входов `irq[31..0]` для выполнения запроса на прерывания. Прерывание будет обработано при выполнении трех условий:

- Бит `PIE` (*Peripheral Interrupts Enable*) регистра `status` установлен в единицу;
- На один из 32 входов порта `irq` принят запрос на прерывание;
- Соответствующий входу `irq` бит регистра `ienable` установлен в единицу.

Установка бита `PIE` в ноль полностью отключает возможность обработки прерываний, а значение каждого бита в регистре `ienable` означает, что прерывание с соответствующего ему входа разрешено. Схема аппаратного прерывания представлена на рисунке 6.1. Регистр `ipending` записывает информацию о том, какой из входов является источником прерывания.

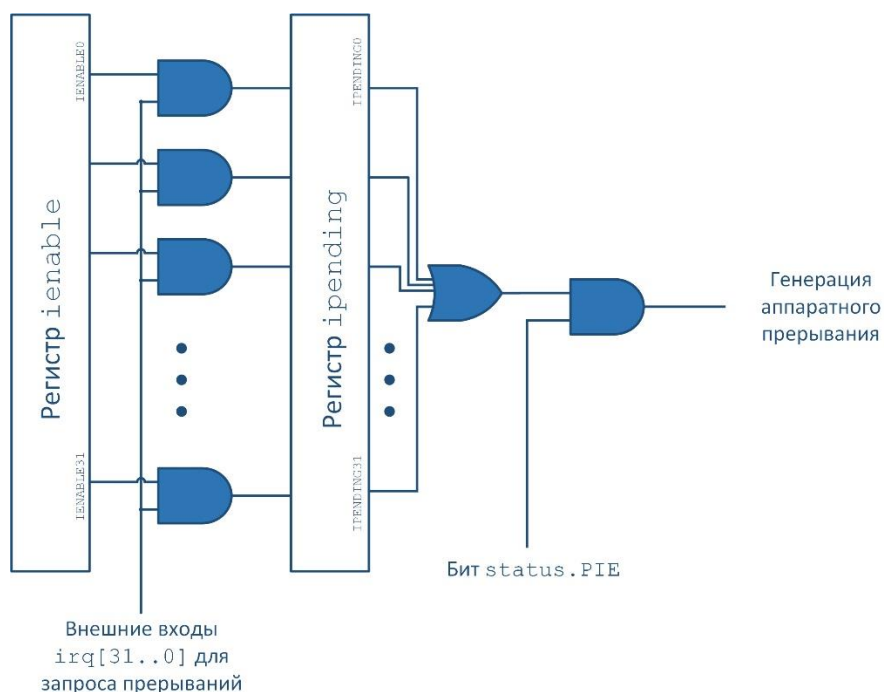


Рисунок 6.1 Схема аппаратного прерывания в Nios II

Ранее в разделе 3.2.3 мы кратко определили функционал и работу обработчика исключений. Рассмотрим подробнее что происходит при возникновении прерываний<sup>7</sup>:

1. Завершается выполнение текущей инструкции;
2. Производится отключение дальнейших прерываний (бит `PIE` обнуляется), сохранение содержимого **регистров статуса** и программного счетчика (в нем находится адрес следующей инструкции прерываемой программы)
3. В программный счетчик загружается адрес, по которому располагаются процедуры обработки исключений;

<sup>7</sup> Порядок обработки для версии Nios II/f имеет некие отличия, подробнее можно прочитать в [2]

4. Обработчик сохраняет содержимое регистров **процессора**;
5. Обработчик определяет источник прерывания;
6. В соответствии с источником прерываний вызывается процедура обработки прерывания (англ. ISR, interrupt service routine);
7. ISR сбрасывает соответствующие условия возникновения прерываний;
8. Производится выполнение всех необходимых функций, связанных с этим прерыванием, по окончании выполнения возвращается в обработчик исключений;
9. Обработчик восстанавливает изначальное содержимое всех регистров;
10. Работа процессора восстанавливается путем выполнения команды ассемблера `eret` (*exception return*), которая восстанавливает содержимое регистра `status` и значение программного счетчика; процессор вновь работает в нормальном режиме.

Обработчик исключений для процессора Nios II является программной частью, скрытой от пользователя, и входит в HAL.

### 6.3. Обработка прерываний в рамках HAL

В HAL содержится процедура, которая наблюдает за прерываниями и выполняет необходимые для них задачи. Данная процедура называется *обработчиком исключений* (*exception handler*). Эта процедура располагается в памяти по адресу, который был указан при сборке системы в Platform Designer в качестве вектора исключений (раздел 2.2.2). Обработчик прерываний выполняет три основные операции:

- Сохраняет содержимое регистров (`alt_irq_entry.S`, `alt_exception_entry.S`);
- Определяет источник прерывания и в зависимости от приоритета перенаправляет программу на выполнение к соответствующей процедуре (`alt_irq_handler.c`);
- По окончании обработки прерывания подгружает ранее сохраненные значения в регистры (`alt_exception_entry.S`).

Для работы с процедурами прерываний HAL содержит набор функций:

- `alt_irq_register()` – регистрация процедуры прерывания;
- `alt_irq_disable()` – отключение прерывания от конкретного источника;
- `alt_irq_enable()` – включение прерывания от конкретного источника;
- `alt_irq_disable_all()` – отключение прерываний от всех источников;



- `alt_irq_enable_all()` – включение прерываний от всех источников;
- `alt_irq_interruptible()` и `alt_irq_non_interruptible()` – настройка срабатывания прерываний в процессе выполнения процедуры другого прерывания
- `alt_irq_enabled()` – возвращает информацию о разрешённых (англ. *enabled*) прерываниях.

Рассмотрим подробнее процедуру регистрации прерывания:

```
int alt_irq_register (alt_u32 id,
                    void* context,
                    alt_isr_func handler)
```

Аргумент `id` соответствует номеру модуля, для которого регистрируется прерывание, `context` – это указатель на данные, которые необходимы для конкретной процедуры обработки прерывания, а `handler` – это название самой функции, которую необходимо выполнить по прерыванию. Регистрация прерывания необходима для того, чтобы при получении сигнала прерывания процессор знал, какую процедуру для данного прерывания вызвать.

Функция, указанная аргументом `handler`, имеет следующий вид:

```
void isr (void* context, alt_u32 id)
```

При этом аргументы `context` и `id` соответствуют этим же аргументам, передаваемым в функцию `int alt_irq_register()`.

#### 6.4. Реализация прерываний в проекте `nios_load2`

В системе `nios_load2` прерывание может использоваться счетчиком, чтобы производить переключение светодиодов. Ранее при сборке системы в среде `Platform Designer` прерывания уже были настроены, поэтому для новой реализации системы можно использовать готовую аппаратную реализацию из предыдущего раздела. Новая версия программы в псевдокоде будет выглядеть следующим образом:

```
Инициализация системы;
Регистрация обработки прерываний от счетчика;
Бесконечный цикл () {
    Считывание информации с кнопок и переключателей;
    Отображение информации на семисегментных индикаторах;
    Отображение информации в консоли;
}
```

Обработчик прерываний в рамках HAL может взаимодействовать с другими функциями как через глобальные переменные, так и с помощью *контекста* (англ. *context*) – такой переменной, которая содержит все необходимые данные для обработки прерывания. В данной реализации будет продемонстрирован второй способ обмена данными. При обработке прерывания от счетчика необходима информация как о базовых адресах периферии, так и информация о состоянии системы (период, пауза). Для объединения всех этих данных в одну переменную воспользуемся структурой, которую определим следующим образом:

```
typedef struct cntxt{
    alt_u32 timer_base;
    alt_u32 led_base;
    cmd_type *cmd_p;
} cntxt_type;
```

Системный счетчик осуществляет запрос на прерывание каждую микросекунду, в обработчике прерывания необходимо провести подсчет микросекунд и по достижению необходимого значения переключить светодиоды:

---

```
. . .
#include "sys/alt_irq.h"
. . .
static void led_line_isr(void* context, alt_u32 id)
{
    alt_u16 led_pattern;
    static alt_u32 i = 0x1;
    cntxt_type *ctxt;
    cmd_type *cmd;
    static int ntick = 0;

    ctxt = (cntxt_type *) context;
    cmd = ctxt->cmd_p;
    //Сброс регистра to:
    IOWR(ctxt->timer_base, 0, 0);

    if (cmd->hold)
        return;
    if (ntick < cmd->dur)
        ntick++;
    else {
        if (i >= 0x800) {
```

```

        led_pattern = 0x00;
        i = 0x01;
    } else {
        led_pattern = i-1;
        i = (i<<1);
    }

    IOWR_ALTERA_AVALON_PIO_DATA(ctxt->led_base,
led_pattern);
    ntick = 0;
}

```

---

Каждый раз при вызове данной функции проверяется достижение количества «тиков» указанного переключателями значения, и по его достижению производится переключение значения, выводимого на светодиоды. Остальные функции (инициализации, обработки данных кнопок и переключателей и вывод на семисегментные индикаторы) остаются такими же, как и в предыдущей главе. Код главной программы выглядит следующим образом:

---

```

. . .
#include "sys/alt_irq.h"
. . .
typedef struct cmnds{
    int hold;
    int dur;
} cmd_type;

typedef struct cntxt{
    alt_u32 timer_base;
    alt_u32 led_base;
    cmd_type *cmd_p;
} cntxt_type;

static void led_line_isr(void* context, alt_u32 id){

    alt_u16 led_pattern;
    static alt_u32 i = 0x1;
    cntxt_type *ctxt;
    cmd_type *cmd;
    static int ntick = 0;
    ctxt = (cntxt_type *) context;
    cmd = ctxt->cmd_p;

```

```

IOWR(ctxt->timer_base, 0, 0);
if (cmd->hold)
    return;
if (ntick < cmd->dur)
    ntick++;
else {
if (i >= 0x800) {
    led_pattern = 0x00;
    i = 0x01;
} else {
    led_pattern = i-1;
    i = (i<<1);}

    IOWR_ALTERA_AVALON_PIO_DATA(ctxt->led_base,
led_pattern);
    ntick = 0;}
// for (j=0;j<255*cmd.dur;j++){;}

. . .

int main(){
    cmd_type sw_cmd={0,100}; // инициализация: паузы нет,
интервал 100 мс
    int current;
    cntxt_type my_cntxt;
    init_sys(BTN_BASE);
    my_cntxt.timer_base = SYS_TMR_BASE;
    my_cntxt.led_base = LEDR_BASE;
    my_cntxt.cmd_p = &sw_cmd;
    alt_irq_register(SYS_TMR_IRQ, (void *) &my_cntxt,
led_line_isr);
    while(1){
        pio_info(BTN_BASE, SWITCH_BASE ,&sw_cmd);
        if (sw_cmd.dur!=current){//изменение интервала
            printf("Interval: %03u ms \n", sw_cmd.dur);
            current = sw_cmd.dur; // запоминание текущего
интервала
        }
        sevs_ind(SEVS_BASE, sw_cmd);}}

```

## Заключение

В рамках данного учебного пособия были представлены материалы, описывающие последовательность создания системы на базе процессора с программным ядром Nios II. Мы познакомились с концепцией настраиваемого ядра софт-процессора, разобрали его архитектуру и научились подключать различную периферию к процессорной системе. Также мы научились создавать приложения для собранной системы и поработали с прерываниями.

Параллельно с изучением процессора Nios II был разобран интерфейс САПР Quartus Prime – инструмента, который позволяет проектировать системы на базе ПЛИС фирмы Intel. Целью данного пособия было дать «быстрый старт» и ответить на основные вопросы, которые чаще всего возникают у читателя. Полученные в ходе прохождения пособия навыки должны послужить основой для дальнейшего изучения процессорных систем. Для последующего изучения авторы рекомендуют ознакомиться с материалами, указанными в ссылках, а также посетить следующие ресурсы:

- <https://marsohod.org> – российский ресурс, посвященный тематике ПЛИС;
- <https://habr.com/ru/hub/fpga/> – раздел сайта Хабр, где многие пользователи делятся своим опытом различных проектов на ПЛИС;
- <https://hackaday.com/tag/fpga/> – зарубежный ресурс, так же регулярно публикующий различные проекты на ПЛИС.

Авторы надеются, что материал учебного пособия окажется полезным и поможет читателям сделать свои первые шаги в освоении процессорных систем.

## Ссылки

- [1] Intel FPGA, «Intel Quartus Prime Standard Edition User Guide: Design Compilation,» 2019.
- [2] Intel FPGA, Nios II Gen2 Processor Reference Guide, 2016.
- [3] Intel FPGA, «Nios II Processor Reference Handbook,» 2016.
- [4] Intel FPGA, «Avalon Interface Specifications,» 2018.
- [5] Intel FPGA, «Nios II Gen2 Software Developer's Handbook,» 2018.
- [6] Д. Харрис и С. Харрис, Цифровая схемотехника и архитектура компьютера, 2017.

## Приложение

Таблица подключения периферии к ПЛИС на плате DE10-Standard

SDRAM_ADDR[0]	PIN_AK14	KEY[0]	PIN_AJ4
SDRAM_ADDR[1]	PIN_AH14	KEY[1]	PIN_AK4
SDRAM_ADDR[2]	PIN_AG15	KEY[2]	PIN_AA14
SDRAM_ADDR[3]	PIN_AE14	HEX0[0]	PIN_W17
SDRAM_ADDR[4]	PIN_AB15	HEX0[1]	PIN_V18
SDRAM_ADDR[5]	PIN_AC14	HEX0[2]	PIN_AG17
SDRAM_ADDR[6]	PIN_AD14	HEX0[3]	PIN_AG16
SDRAM_ADDR[7]	PIN_AF15	HEX0[4]	PIN_AH17
SDRAM_ADDR[8]	PIN_AH15	HEX0[5]	PIN_AG18
SDRAM_ADDR[9]	PIN_AG13	HEX0[6]	PIN_AH18
SDRAM_ADDR[10]	PIN_AG12	HEX1[0]	PIN_AF16
SDRAM_ADDR[11]	PIN_AH13	HEX1[1]	PIN_V16
SDRAM_ADDR[12]	PIN_AJ14	HEX1[2]	PIN_AE16
SDRAM_DQ[0]	PIN_AK6	HEX1[3]	PIN_AD17
SDRAM_DQ[1]	PIN_AJ7	HEX1[4]	PIN_AE18
SDRAM_DQ[2]	PIN_AK7	HEX1[5]	PIN_AE17
SDRAM_DQ[3]	PIN_AK8	HEX1[6]	PIN_V17
SDRAM_DQ[4]	PIN_AK9	HEX2[0]	PIN_AA21
SDRAM_DQ[5]	PIN_AG10	HEX2[1]	PIN_AB17
SDRAM_DQ[6]	PIN_AK11	HEX2[2]	PIN_AA18
SDRAM_DQ[7]	PIN_AJ11	HEX2[3]	PIN_Y17
SDRAM_DQ[8]	PIN_AH10	HEX2[4]	PIN_Y18
SDRAM_DQ[9]	PIN_AJ10	HEX2[5]	PIN_AF18
SDRAM_DQ[10]	PIN_AJ9	HEX2[6]	PIN_W16
SDRAM_DQ[11]	PIN_AH9	HEX3[0]	PIN_Y19
SDRAM_DQ[12]	PIN_AH8	HEX3[1]	PIN_W19
SDRAM_DQ[13]	PIN_AH7	HEX3[2]	PIN_AD19
SDRAM_DQ[14]	PIN_AJ6	HEX3[3]	PIN_AA20
SDRAM_DQ[15]	PIN_AJ5	HEX3[4]	PIN_AC20
SDRAM_BA[0]	PIN_AF13	HEX3[5]	PIN_AA19
SDRAM_BA[1]	PIN_AJ12	HEX3[6]	PIN_AD20
SDRAM_LDQM	PIN_AB13	HEX4[0]	PIN_AD21
SDRAM_UDQM	PIN_AK12	HEX4[1]	PIN_AG22
SDRAM_RAS_N	PIN_AE13	HEX4[2]	PIN_AE22
SDRAM_CAS_N	PIN_AF11	HEX4[3]	PIN_AE23
SDRAM_CKE	PIN_AK13	HEX4[4]	PIN_AG23
SDRAM_CLK	PIN_AH12	HEX4[5]	PIN_AF23
SDRAM_WE_N	PIN_AA13	HEX4[6]	PIN_AH22
SDRAM_CS_N	PIN_AG11		

Смирнов Даниил Сергеевич  
Дейнека Иван Геннадьевич  
Алейник Артем Сергеевич  
Шарков Илья Александрович

**ОСНОВЫ РАЗРАБОТКИ ВСТРАИВАЕМЫХ СИСТЕМ НА ПЛИС  
С ИСПОЛЬЗОВАНИЕМ ПРОЦЕССОРА NIOS II®**

Учебное пособие

В авторской редакции

Редакционно-издательский отдел Университета ИТМО

Зав. РИО

Н.Ф. Гусарова

Подписано к печати

Заказ №

Тираж

Отпечатано на ризографе



**Редакционно-издательский отдел**  
**Университета ИТМО**  
197101, Санкт-Петербург, Кронверский пр., 49