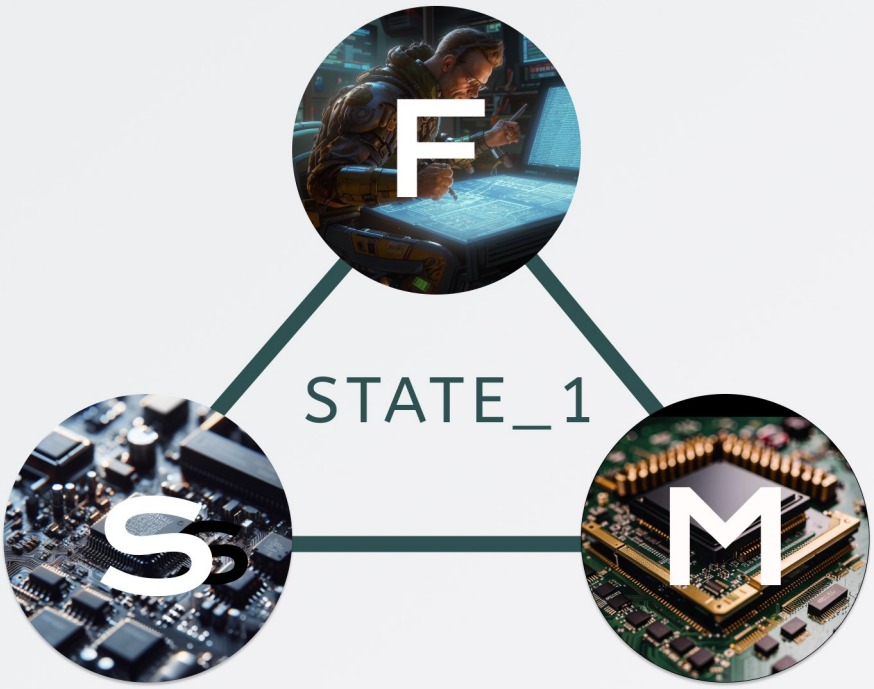
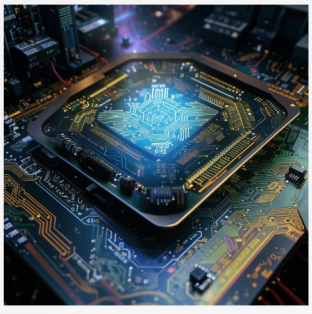


FPGA SYSTEMS

FPGA-Systems
magazine

FSM :: № BETA



Первый журнал о программируемой логике

V1.1

Страничка редакции

ПЛИС-культ привет, FPGA комьюнити!

Приветствую вас на страницах народного творчества FPGA / RTL / Verification разработчиков. Я уже и не надеялся на выход второго номера журнала, но вы приложили не мало усилий, чтобы он состоялся. С чем всех нас я и поздравляю.

За сим разрешите больше вас не задерживать и пожелать приятного прочтения. Надеюсь, что на страницах журнала читатель найдёт для себя много полезного и нового.

До встречи на страницах третьего номера журнала FPGA-Systems Magazine (FSM), информацию о выходе которого вы всегда сможете найти на веб странице нашего издания.

PS: если вы хотите написать о чем-то, но не можете определиться с темой, посмотрите этот список, где приведены 100+ тем, которые могли бы быть интересны читателям.

===

С уважением, вождь FPGA комьюнити

Коробков Михаил

Контакты

По всем вопросам обращайтесь в телеграм @KeisN13 или по электронной почте admin@fpga-systems.ru

Попов М. А., Романов А. Ю.

Versal... Как много в этом слове!

click обзор

Харабадзе Д.Э.

“Бегущие огни” на ATF22V10

click начинающим

Малышев Никита

Развитие отечественных САПР проектирования микроэлектроники на платформе Delta Design

click обзор

Коробков Михаил

Сдвиговый регистр или то, о чем не расскажут в статьях для начинающих :: атрибуты синтеза

click начинающим

Гнитеев Николай

Интерактивный HDL

click обзор

Сонин А.П., Хромцев А.В., Свирин Д.М.

DRFM на основе ПЛИС Virtex-7 для тестирования радиолокаторов с синтезированной апертурой антенны

click реализация

Заостровных Андрей

Обзор отладочной платы ALINX AXU15EGB

click обзор

Кузнецов Данила

Реализация IQ-модулятора для ПЛИС

click реализация

Графов М.В.

Отладочная плата ПИР СЦХ-254 «Карно»

click обзор

Погодаев А.А., Абрамов А.Е.

Загрузка драйвера UEFI, с помощью ПЛИС

click реализация

Романова И.И., Зунин В.В., Маршутина Е.Н., Американов А.А., Романов А.Ю.

DESim: как изучать проектирование на ПЛИС без отладочной платы

click начинающим

Кудинов Максим

Игра в Pong на SystemVerilog

click реализация

Хлуденьков А.Н.

Давайте создадим процессор! Step by Step.

click начинающим

Мальчуков А.Н.

if if'y рознь. QUARTUS vs VIVADO. SystemVerilog vs VHDL

click исследования

Аверченко А.П.

Шинная организация сигналов в DEEDS

click начинающим

Пузанов Николай

Быстрое вычисление медианы в целых числах

click исследования

Сергей Б.

Испытательный стенд с использованием YosysHQ MCY

click

туториал

Гуров В.В.

Опенсорс для ПЛИС... и наоборот

click

туториал

Белоусов Олег

Buildroot это просто

click

туториал

Мангушев Александр Вячеславович

Подключение физического устройства, размещенного на ПЛИС в симулятор QEMU при помощи Ethernet

click

туториал

Балакший Сергей

Шаблон проекта испытательного стенда с использованием Yosys, Verilator, Icarus Verilog

click

туториал

Кудинов Максим

Verilator как linter в Neovim

click

tips & tricks

Туровский Дмитрий Николаевич

Заметки ПЛИСовода

click

tips & tricks

VERSAL... КАК МНОГО В ЭТОМ СЛОВЕ!

Попов М. А., *гл. спец. по программному обеспечению ООО «Бигпринтер Цифровые Инновации»*, e-mail: *maks@bigprinter.ru*

Романов А. Ю., *канд. техн. наук, доц., Национальный исследовательский университет «Высшая школа экономики»*

e-mail: *a.romanov@hse.ru*

Обсуждение и комментарии: link

С момента анонса в 2019 еще компанией *Xilinx*, носившей тогда это имя без гордого префикса из трех букв, чипы *Versal ACAP* (*Adaptive Compute Acceleration Platform*) были малодоступны российским разработчикам — первые отладочные платы стоили десятки тысяч их американских долларов, а сложность разработки собственной платы под этот чип не отпугнула бы разве что Тони Старка.

Уже много воды и слез разработчиков утекло с тех пор, а *Versal* ~~силиконовый~~ кремниевый так же (не)доступен россиянам, как и Версаль королевский: самый дешевый кит от *AMD VPK120* стоит $12K без учета доставки и растаможки. Классический аргумент в стиле «если у вас нет денег на железную дверь, она вам и не нужна» в сфере НИОКР работает далеко не всегда — редкий разработчик откажется поизучать топовый чип ~~за счет своего работодателя~~.

Самые наблюдательные коллеги, конечно, вспомнят анонсированный пару лет назад вариант *SOM Kria* с *Versal* на борту, очевидно, призванный решить ту же задачу, что и уже существующий *Kria K26*: сделать цену приятной и разгрузить схемотехников и конструкторов печатных плат от разработки очередной двадцатислойки.

Остается признать, что тот анонс *AMD-Xilinx* пока так и остался анонсом, а вот китайцы из *Alinx* времени не теряли. Эта компания уже известна своими недорогими отладками (по сути — *SOM*) с *Zynq-7000* и *Ultrascale+* на борту, но теперь она не только замахнулась, но и — звучит барабанная дробь — таки серийно выпустила *SOM V100* (рис. 1) с чипом *XCVE2302-SFVA784-1LP-E-S* (семейство *Versal AI Edge*) [1]. Основные характеристики *SOM* приведены в табл. 1.

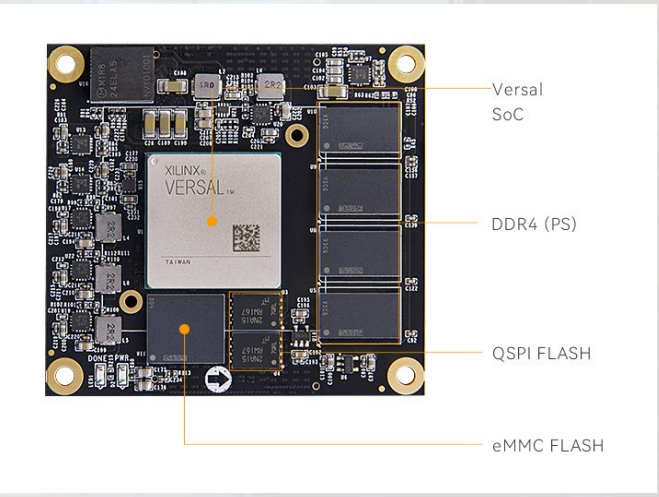


Рис. 1 – Внешний вид модуля *V100* (фото производителя)

Таблица 1 – Основные характеристики SOM V100

Устройство, интерфейс, величина	Характеристика, значение
ОЗУ	4 Гбайт (64 бита) DDR4
ПЗУ	64 Мбайт QSPI, 8 Гбайт eMMC
PCI-Express	Gen4 ×8
Трансиверы	8 x GTY, до 12.5 Гбит/с
Линии ввода/вывода	53 (PS) + 106 (PL)
Коннекторы	Samtec ADF6-40-03.5-L-4-2-A-TR два по 160 контактов
Напряжение питания	12 В
Потребляемый ток	не заявлен
Диапазон рабочих температур	0..100 градусов Цельсия
Габариты	65 x 60 x 25 мм
Масса	не заявлена

Обращает на себя внимание цена, которая в 20 (!!!) раз ниже, чем у «тру ориджинал девелопд ин зе юнайтед стейтс», так что теперь едва ли не только лишь все, но и каждый сможет приобщиться к прекрасному во всех отношениях *Versal*.

Видимо, *Alinx* настолько вдохновлялись *Kria*, что применили и столь нежно любимые всеми конструкторами и монтажниками разъемы *Samtec* (правда, с другим количеством контактов), ставшие уже легендой. Кто из нас не сверлил их сбоку тончайшим сверлом, забыв развести тот самый необходимый пин как раз посередине во внутреннем ряду? Впрочем, нужно признать — для реализации требуемых скоростей передачи при компактном размере разъемам *Samtec* едва ли найдется альтернатива.

К слову, у самых нетерпеливых есть прекрасная возможность не ждать разработки и изготовления собственной несущей платы — комплект *VD100* включает в себя и ее, поэтому готов к работе прямо из коробки (рис. 2).

Рис. 2 – Кит *VD100* в руках юного плисовода

На несущей плате реализованы:

- *PCIE* 4.0 x4;
- 2 x *SFP+* (12.5 Гбит/с);
- 2 x *Gigabit Ethernet* (1 x *PS* + 1 x *PL*);
- *UART*;

- 2 x *4-lane MIPI* (*PL*);
- 2 x *CAN* (*PS*);
- *USB 2.0*;
- *I2C EEPROM* (4 кбит);
- термодатчик (*LM75*);
- *RTC*.
- Размеры несущей платы: 182 x 107 мм.

Сова принесла наш комплект совсем недавно, поэтому изучение возможностей *VD100* еще впереди, но уже сейчас можно отметить полноценную документацию на нашем, самом что ни на есть, языке Шекспира, и репозиторий с большим количеством примеров [2]. Остается надеяться, что качество и платы, и кода в примерах будет по-версальски блестящим, а уж мы обязательно проверим это и доложим результат на конференциях и расскажем в будущих статьях.

Дисклеймер: материальная заинтересованность авторов в продвижении продукции компании *Alinx* отсутствует.

Список источников

1. AMD Versal AI Edge Series Product Selection Guide. XMP464 (V1.10). AMD, 2024. 7 p.
2. Alinx VD100 Development Board Repository on Github [Электронный ресурс]. – Электронные данные. – Режим доступа: https://github.com/alinxalinx/VD100_2023.2, свободный.

Развитие отечественных САПР проектирования микроэлектроники на платформе Delta Design

Малышев Никита

malyshev.n@eremex.ru

Обсуждение и комментарии: link

Исход с российского рынка западных вендоров и разработчиков систем автоматизированного проектирования микроэлектроники явно продемонстрировало зависимость и отсутствие в России собственных решений в данном направлении. Нужно признать, что некоторые попытки по импортозамещению в стране принимались, но, к сожалению, они оказались недостаточными. Это связано с тем, что поддерживалась разработка отдельных частей сквозного цикла проектирования, при этом, глобально, оставалась зависимость от западных решений. Так или иначе подобная ситуация сохранялась, пока не были начаты работы над системой сквозного проектирования конфигурации ПЛИС и проектирования СБИС Delta Design Simtera IC.

Введение

В отличие от микроэлектроники, уход с российского рынка крупных международных игроков в области проектирования радиоэлектроники стало большим толчком к развитию и узнаваемости отечественных систем. Звездный час настал для САПР Delta Design российской компании ЭРЕМЕКС, который уже больше 10 лет присутствовал на рынке и давал возможности «бесшовного» перехода с продуктов компании Altium и Mentor Graphics в разработке печатных плат без потери функциональности. Кроме того, в пакете предусмотрены собственные уникальные конкурентные решения моделирования и трассировки.

Огромным заделом было и то, что на платформе системы проектирования радиоэлектроники развивались и другие инструменты, на основе которых появились продукты DeltaCAM[1], SimOne[2], Delta Design Simtera[3], Delta Design Simtera IC. Статья посвящена развитию последнего из списка модуля.

Задачи и требования, предъявляемые к Delta Design Simtera IC

Развитие программного пакета Delta Design Simtera началось в начале 2010-ых годов и на тот момент он представлял из себя отдельный программный пакет по моделированию VHDL-кода. Основным

функционалом, дополнительно к моделированию была генерация моделируемого нетлиста по электрической схеме. При разработке схемотехнического описания использовались инструменты ведения библиотек, создания компонентов – их условно-графического описания, создания модели высокоуровневого описания и генерации его шаблона на основе УГО. На рисунке 1 представлено окно создания условно-графического обозначения D-триггера и его наполнение моделью высокоуровневого описания на языке Verilog с помощью генератора кода. Далее, на основе базы компонентов разработчику предлагается перейти к схемотехническому проектированию, генерации нетлиста на выбранном языке описания аппаратуры и его моделированию (рисунок 2). Стоит отметить, что генерация нетлиста и моделирование производится в несколько нажатий кнопок мыши – система автоматически производит все необходимые операции и выдает результаты моделирования в виде временных меандров и сообщений (ассертов), расставленных разработчиком.

В виду спроса и внешних обстоятельств позиционирование продукта изменилось в сторону расширения функциональности. Так появилась поддержка языков Verilog и SystemVerilog, вплоть до стандарта 2017 года (IEEE 1800-2017, последнего доступного на текущих момент). Стоит отметить, что продукт Delta Design Simtera постоянно развивается и в нем предполагается поддержка последнего стандарта 2023 года. IEEE, Институт инженеров электротехники и электроники, обновил стандарт SystemVerilog в декабре прошлого года.

Требования к Delta Design Simtera на сегодняшний день предъявляются как к системе симуляции и верификации HDL-проектов. По своей сути, данный продукт способен заменить в работе Questa Sim от Mentor Graphics (ныне Siemens). При прямом сравнении двух систем, на текущий момент в Simtera присутствует весь необходимый для этого функционал – интегрированное управление проектом, шаблоны и помощники по исходному коду, поддержка VHDL 1987, 1993, 2002, 2008 (кроме VHPI, PSL), Verilog 1995, 2001, 2005, 2009, платформо-

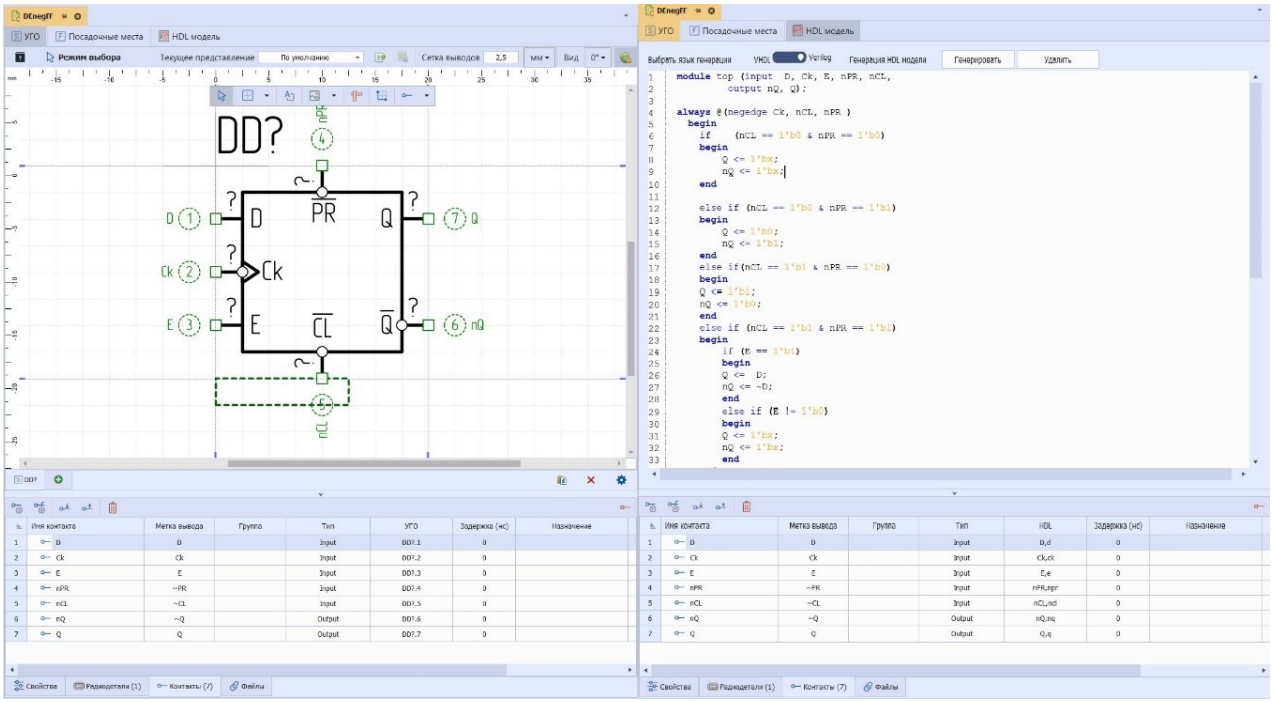


Рисунок 1. Окна создания библиотечного компонента. Слева – условно-графическое обозначение D-триггера, справа – Verilog-модель D-триггера

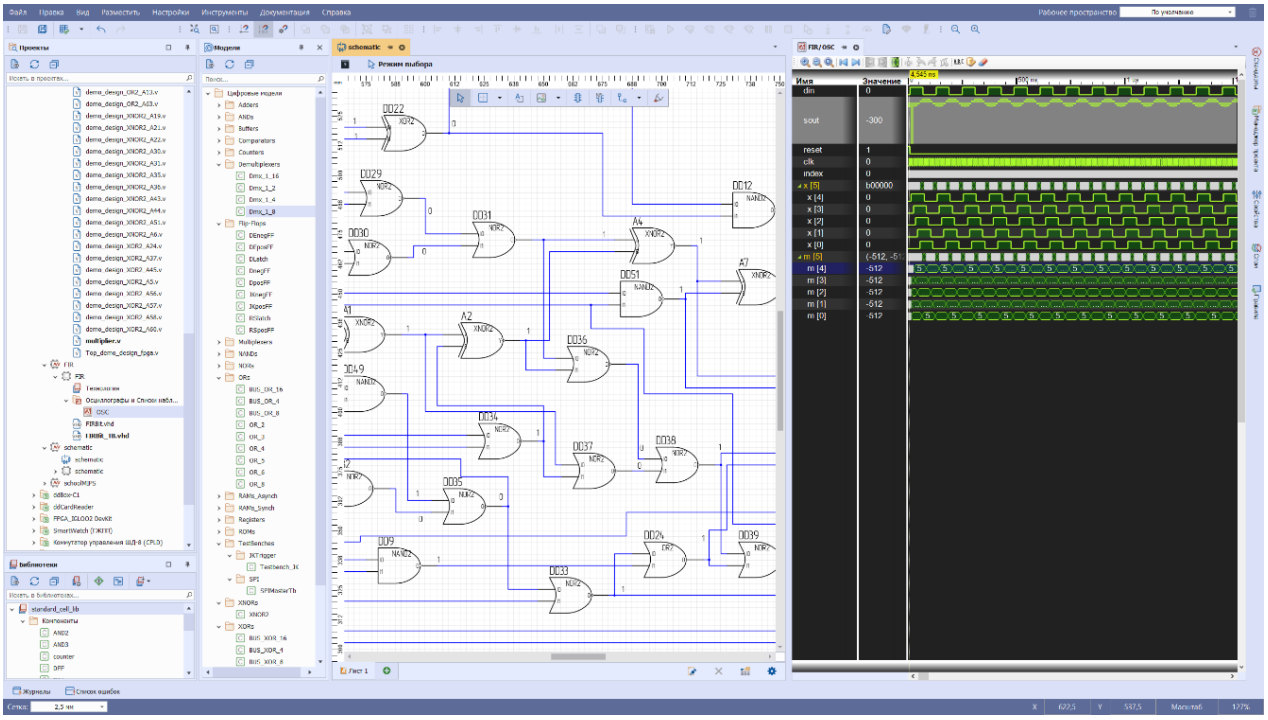


Рисунок 2. Рабочее пространство работы с цифровым проектом, представленным в схемотехническом виде. Слева направо – панель проектов, встроенная библиотека компонентов, схемотехнический редактор,

независимая скомпилированная база данных, интерактивная отладка и пр. В таблице 1 представлена сводная информация по функциональности систем для сравнения.

Поддержка расширенных инструментов верификации HDL-проектов, таких как OVM/ UVM, Verilog PLI/VPI в пакете моделирования Simtera ожидается в ближайшее время, в обновлениях, которые можно получить на сайте компании – eremex.ru и Telegram-канале проекта Simtera [4]. SystemC, VHDL FLI, VHDL VHPI, отладчик языка C не предусматриваются к поддержке в ближайшее время, но возможно, при интересе и запросе от клиентов.

Дополнительно к наличию функционала, система моделирования и верификации должна обладать сравнительно высокой скоростью работы. Сравнительный анализ показал, что скорость компиляции проектов в Questa Sim и Simtera практически одинаковы (последняя несколько выигрывает). По скорости моделирования Simtera отстает от своего конкурента, но это вопрос доработки и оптимизации системы, которая проходит при каждом новом обновлении. Разрыв сокращается, а отставание в скорости моделирования обусловлена юностью отечественной системы, тогда как продукт компании Siemens на рынке более 30 лет.

HDL-языки, и язык Verilog в частности, используются для описания поведения схемы во времени и запуска соответствующего моделирования. Однако воплощение реализующей такое поведение цифровой аппаратуры, то есть логических элементов и соединяющих их проводников, возможно не всегда. Логический синтез – перевод поведенческой модели в набор цифровых компонентов может быть выполнен при использовании лишь синтезируемого подмножества языка – Synthesizable Verilog/ VHDL. Вполне корректный с точки зрения моделирования код может содержать несинтезируемые конструкции. Система проектирования, а именно, логический синтезатор, должен предупредить пользователя о наличии нереализуемых для создания схемы соединения (нетлиста) языковых конструкций. Основная задача логического синтеза – создать схему по

Таблица 1. Основные параметры и их наличие в продуктах

Основные параметры	Questa Core	Questa Prime	Delta Design Simtera
Редактор HDL	Да	Да	Да
Интегрированное управление проектом	Да	Да	Да
Шаблоны и помощники по исходному коду	Да	Да	Да
VHDL 1987, 1993, 2002, 2008 (кроме VHPI, PSL)	Да	Да	Да
Verilog 1995, 2001, 2005, 2009	Да	Да	Да
SystemVerilog IEEE for Design 2005, 2009, 2012	Да	Да	Да
SystemVerilog IEEE for Verification 2005, 2009, 2012	Только assertions	Да	Assertions – да, полная поддержка в разработке
Платформо-независимая база данных	Да	Да	Да
Инкрементное компилирование	Да	Да	Да
Интерактивная отладка	Да	Да	Да
Отладка после моделирования	Да	Да	Да
Отслеживание причинно-следственных связей	Да	Да	Да
Несколько окон диаграмм сигналов	Да	Да	Да
Просмотр схемы	Да	Да	Да
Единое ядро моделирования	Да	Да	Да
VCD and Extended VCD Support	Да	Да	Да
Смешанный Verilog/VHDL	Опция (MixedHDL)	Да	В разработке, ожидается в ближайших обновлениях
Полная поддержка OVM/UVM	Нет	Да	В разработке, ожидается в ближайших обновлениях
Assertions (PSL для VHDL или Verilog, SVA для Verilog)	Да	Да	SVA – в разработке
Analog/Mixed Signal	Опция (Questa ADMS)	Опция (Questa ADMS)	Да
Verilog PLI/VPI	Да	Да	В разработке, ожидается в ближайших обновлениях
SystemVerilog Direct Programming Interface (DPI)	Да	Да	В разработке, ожидается в ближайших обновлениях
Оптимизация производительности моделирования и компиляции	Да	Да	Да
SystemC 2.2, 2.3	Опция	Да	Нет
VHDL FLI	Да	Да	Нет
VHDL VHPI	Нет	Нет	Нет
Отладчик C	Да	Да	Нет
Window 64-бит	Win XP, 7, 8.1, 10	Win XP, 7, 8.1, 10	Win 8.1, 10
Linux 64-бит	Да	Да	В разработке, ожидается в ближайших обновлениях

высокоуровневому описанию на HDL-языке с использованием технологических компонентов ПЛИС или СБИС, которые могут быть представлены, в том числе, в формате Liberty. На рисунке 3 представлен результат синтеза 4-битного арифметико-логического устройства 74181 по его Verilog-описанию[5], с использованием Liberty-библиотеки. В листинге 1 представлено сокращенное содержимое Liberty-библиотеки с единственным функциональным полем function, описывающим логическое поведение выходного пина компонента.

```
library(Nanocron) {
cell(AN2) {
    area: 5.488;
    pin(A) { direction: input; }
    pin(B) { direction: input; }
    pin(Z) { direction: output;
      function: "A&B"; }
  }
cell(OR2) {
    area: 5.488;
    pin(A) { direction: input; }
    pin(B) { direction: input; }
    pin(Z) { direction: output; function:
"A|B"; }
  }
…
}
```

Листинг 1. Liberty-библиотека

Логический синтезатор должен анализировать и оптимизировать нетлист проекта согласно различным характеристикам, заложенным в его HDL-описании с учетом требований, которые указываются в качестве атрибутов и ограничений. С другой стороны, синтезатор может использовать лишь те компоненты, которые разработчик указывает в качестве доступных. Так, в Liberty-библиотеке присутствуют дополнительные поля с указанием временных и мощностных характеристик компонентов, а к проекту прикладываются SDC-ограничения. Кроме того, задача логического синтезатора – подобрать оптимальную схему с наименьшей площадью, то есть сократить количество компонентов для удешевления стоимости производимого изделия. Если говорить о синтезаторах, то собственных решений на российском рынке нет.

В рамках работы по расширению функциональности и созданию системы полного цикла проектирования командой ЭРЕМЕКС, работающей над системой цифрового моделирования были проведены

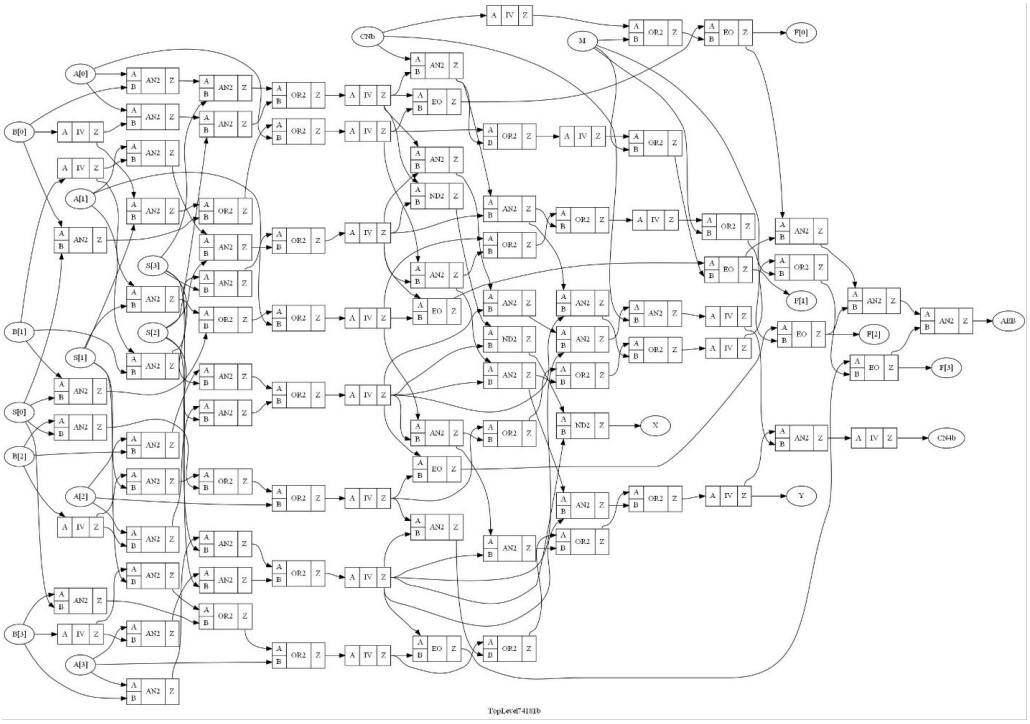


Рисунок 3. Схема, полученная при синтезе Verilog-проекта по Liberty-библиотеке в Delta Design Simtera IC

работы по созданию собственного логического синтезатора. На основе компилятора системы верификации были созданы новые пакеты, а именно: пакет проверки синтезируемости Verilog-проектов, высокоуровневый синтезатор, логический синтезатор, оптимизатор, системы распознавания атрибутов синтеза и файлов ограничения стандарта Synopsys (Synopsys Design Constraint), а также Liberty-анализатор. Были заложены инструменты статического временного анализа и проверки логической эквивалентности. О каждом из инструментов поговорим подробнее. Следует отметить, что приведенные результаты работ, о которых будет идти речь в статье, получены в рамках работы с пре-альфа версией системы и имеют больше исследовательский эффект, нежели технический, который уже сейчас можно применять разработчикам на практике в больших проектах. Под большими проектами следует понимать дизайны, рассчитанные на сотни тысяч, миллионы и более вентилей. Именно такие проекты создаются в рамках работ над созданием собственных уникальных по функциональности сверхбольших интегральных схем и конфигураций программируемых интегральных схем. Для небольших проектов, которые можно имплементировать в проекты до 2 тысяч вентилей, например в ПЛИС М3 от Миландр [6] или подобные, например, Altera Cyclone II (EP2C5) – можно использовать систему Simtera IC в качестве логического синтезатора.

Обнаружение несинтезируемых конструкций и предупреждение о них следует выполнять раньше, чем будет запущен трудоемкий процесс логического синтеза. Для этого был разработан модуль проверки синтезируемости Verilog-проектов. В листинге 2 представлен код, содержащий несинтезируемую конструкцию event, на рисунке 4 представлено сообщение (ошибка) системы при запуске логического синтеза.

```
module dff(clk,r_data,q);
  input clk;
  input r_data;
  output reg q;
  event e_data;

   always@(posedge(clk))
   begin
     if(clk)
       ->e_data;
   end

    always @(e_data)
      q = r_data;

endmodule
```

Листинг 2. Verilog-код, содержащий несинтезируемую конструкцию event

...	Имя файла	Строка	HDL проект	Описание
▸	file1.v	5	test5	Synthesizer Error 6903: Синтез событий (event) не поддерживается

Рисунок 4. Предупреждение системы проверки на синтезируемость

Классическим подходом к задаче реализации синтезатора является «послойное» построение нетлиста по исходному высокоуровневому описанию на HDL-языке. В первой итерации происходит построение схемы с использованием крупных функциональных блоков. Функциональные блоки представляются в виде компонентов вне зависимости от технологии и представляют схему, построенной на высокоуровневом описании. Уже на данном этапе происходит определение конечных автоматов, частей комбинаторной и последовательной логики, оптимизация повторяющихся узлов в единый блок. Следующий этап – анализ технологической библиотеки. В Delta Design Simtera встроена generic-библиотека, то есть универсальная библиотека логических компонентов, состоящая из 2-входовых примитивов. При наличии Liberty-библиотеки в проекте запускается ее анализ, расставляются весовые критерии, то есть коэффициенты приоритета использования в конечном нетлисте в зависимости от требований к

исходному проекту. По созданному на предыдущем шаге функциональной схеме составляется неоптимизированный технологический нетлист. Следующий этап синтеза – оптимизация технологического нетлиста. Теоретической основой для оптимизации логической схемы является задача минизации логических функций. Она заключается в привидении логических функций к такой форме, которой соответствует минимальное число операций над логическими переменными. За счет того, что логические (булевы) функции могут быть представлены в различных формах – в виде таблиц, бинарной диаграммы решений, графа и пр. выделяют алгоритмы, работающие на каждом из видов представлений. Каждый из видов представлений и алгоритмов минимизации в них обладают рядом преимуществ и недостатков. При работе с табличным представлением получение множества всех простых импликант булевой функции уже для числа независимых переменных $n > 10$ может потребовать значительной памяти и большого времени работы ЭВМ [7]. Одним из наиболее интересных с точки зрения функции минимизации логической схемы для большого количества независимых переменных является подход с использование ориентированного ациклического графа [8].

При разработке инструментов логического синтеза и оптимизации команда синтеза ЭРЕМЕКС ориентировалась и сравнивала результаты работы с результатами, полученными в инструментах с открытым исходным кодом – yosys, используемой в OpenLane [9]. OpenLane – САПР сквозного (RTL to GDSII) проектирования СБИС с открытым исходным кодом, который включает в себя инструменты yosys, magic, netgen, KLayout и др. На упомянутом выше проекте 4-битного АЛУ серии 74181 команде ЭРЕМЕКС удалось получить по отдельны блокам совпадающие результаты по площади и количеству компонентов. За счет экстракции вложенности и глобальной оптимизации удалось получить сокращение площади кристалла в сравнении с yosys. Выигрыш составил 5% (226 против 238).

Система логического синтеза не может правильно и эффективно работать без

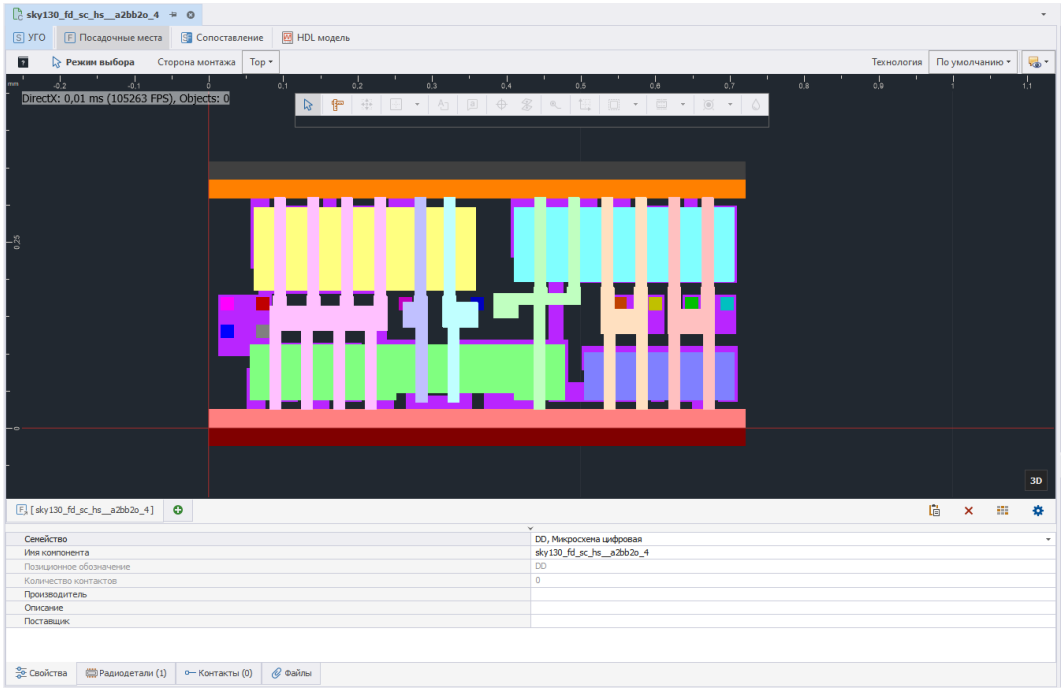


Рисунок 5. Отображение GDSII-компонента SkyWater PDK в Delta Design Simera IC

инструментов проверки логической эквивалентности и статического временного анализа. Первый – проверяет соответствует ли полученный на этапе синтеза нетлист исходному HDL-дизайну, второй – проводит расчет временных параметров схемы. Данные инструменты сейчас находятся в разработке. В разработке находится и часть, связанная с физическим проектированием СБИС. Уже реализованы инструменты импорта технологических библиотеки и их отображения. На рисунке 5 представлено отображение компонента библиотеки PDK SkyWater [10] в Simtera и его представление в OpenLane – рисунок 6. Видно, что они идентичны.

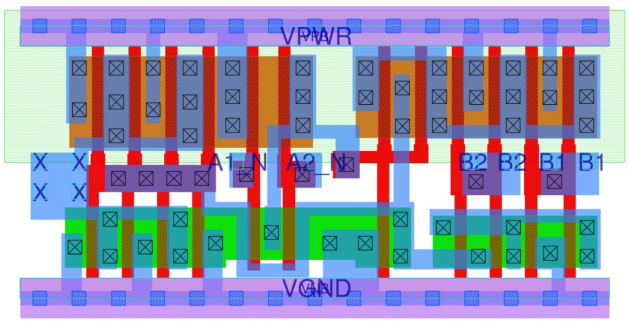


Рисунок 6. Отображение GDSII-компонента SkyWater PDK в OpenLane

Выводы

Развитие САПР микроэлектроники во многом сопровождается кооперацией производителей микросхем и разработчиков программного обеспечения, так как это единственно правильный вектор развития и выпуска конкурентноспособной продукции. Так, например, за последние несколько лет за счет спроса, развития собственных решений, а также за счет сотрудничества с вендорами Simtera и Simtera IC выросла в качестве моделирования и в количестве сопутствующих инструментов, началось развитие собственных инструментов «кремниевой компиляции», защищенной от встраивания «закладок». Уже сейчас можно сказать, что программный пакет от компании ЭРЕМЕКС может закрыть задачи в разработке конфигурации ПЛИС, особенно российских. На рисунке 7 представлен графический редактор подготовки файлов ограничений для ПЛИС М3 компании МИЛАНДР.

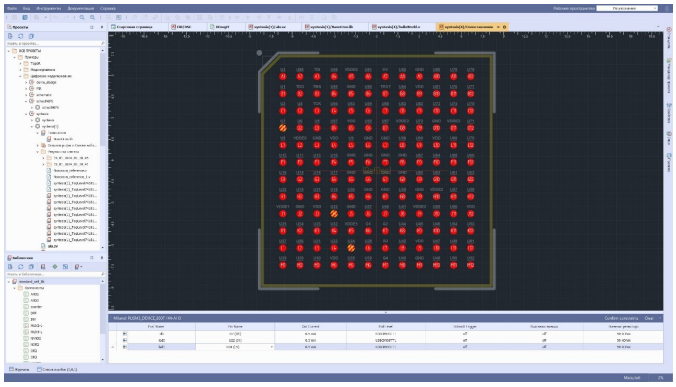


Рисунок 7. Графический редактор подготовки файлов ограничения в Delta Design Simtera IC

Список литературы:

1. Сергей Попов. DeltaCAM – инструмент подготовки производственных файлов.
2. Управление и производство, 2021, вып. 9., стр. 22-28 https://sapr.ru/article/26299
3. Андрей Смирнов, Алексей Гимеин, Схемотехническое моделирование в Delta Design SimOne. Часть 2, Современная Электроника, 2022, вып. 9
4. Никита Малышев, Аркадий Поляков, Библиотеки HDL-тестов для систем моделирования цифровой аппаратуры. Отечественная САПР проектирования микроэлектроники. Часть 1. Современная Электроника, 2023, вып. 9, стр.12-15
5. https://t.me/+ZnqV7hCNviAxYzMy
6. Mark C. Hansen, Verilog Bebaviral description of the TI 74181 Circuit, https://web.eecs.umich.edu/~jhayes/iscas.restore/74181b.v
7. Сергей Шумилин, ПЛИС М3, ОКР «Бриллиант». https://www.milandr.ru/upload/iblock/60c/60cdbff6fae9ad208ada6109a1f3cd9f.pdf

8. Лузин С.Ю. Математическое обеспечение синтеза минимальных форм представления переключательных функций для САПР БИС, г. Санкт-Петербург, 2001, ОАО «НИИ «ЗВЕЗДА», Диссертация на соискание ученой степени доктора наук, на правах рукописи
9. Gao M., Jiang J-H., Jiang Y., Mishchenko A., Sinha S., Villa T., Brayton R. Optimization of Multi-Valued Networjs, In Proc, 32nd IEEE International Symposium on Multiple-Valued Logic (ISMVL'02), Boston, MA, USA, May 2002, p.168-177. Доступно по ссылке: https://people.eecs.berkeley.edu/~alanmi/publications/2002/ismvl02.pdf
10. https://github.com/The-OpenROAD-Project/OpenLane
11. https://github.com/google/skywater-pdk

Интерактивный HDL

Гнитеев Николай

Телеграм @Godhart

Обсуждение и комментарии: link

Аннотация

В статье рассматривается подход к интерактивному вводу HDL кода и RTL симуляции.

Интерактивные оболочки позволяют в упрощённой форме вводить небольшие фрагменты кода и моментально исполнять его. Это очень удобно при изучении нового материала, проработке идей, да и просто при наборе кода, когда для небольшой кодовой конструкции писать тест может быть довольно накладно, а проверить, как она поведёт себя в определённых условиях, хочется сразу. На сегодняшний день автору статьи не удалось найти, как это можно было бы осуществлять для HDL кода, поэтому на основе уже распространённого решения был разработан свой подход, который рассмотрен в статье.

1. Знакомство со средой Jupyter

Среди интерактивных оболочек на сегодняшний день можно выделить среду Jupyter. Хотя в названии среды зашифровано сразу три языка программирования - Julia, Python и R, популярность она приобрела именно благодаря Python и распространению его в области машинного обучения. Именно эта среда будет использована в качестве основы в описанном подходе, и далее мы рассмотрим какие у неё есть возможности и особенности.

Работа с Jupyter осуществляется через web-браузер, и в простом случае не сильно отличается от редактирования документа Markdown. Кстати, Markdown - ещё одна составляющая этой среды. Документ Jupyter представляет из себя последовательность “ячеек”(cell). Каждая ячейка может содержать текст в формате Markdown, “сырой текст” или программный код (но только один из этих трёх вариантов).

В общем-то, на первый взгляд описание похоже на типовой документ Markdown. Однако в Jupyter кодовые ячейки **можно исполнять**. Это ключевое отличие документа Jupyter от обычного документа Markdown, которое всё определяет. В кодовых ячейках можно не только выполнять определённые расчёты, но и формировать текст для самого документа, а также представлять данные в

самых различных формах (как правило, для этого используются дополнительные пакеты Python / Julia / R). Результаты отображаются в документе после самой кодовой ячейки. Здесь же отображается и вывод потоков STDOUT/STDERR, если такой был при выполнении кодовой ячейки.

Код из ячеек выполняется с помощью “ядра” (kernel), по своей сути — некоторой программы с определённым интерфейсом, которое будет подключено к документу при его открытии. Каждое ядро предназначено для определённого языка программирования. Это м.б. ядро не только для кода Python / Julia / R, но возможно использование и стороннего ядра из великого множества. Вот только, к сожалению, ядер для HDL кода в этом множестве не нашлось.

Важно отметить, что при исполнении кода ячеек сохраняется контекст, и при выполнении очередной ячейки можно обращаться к результатам, полученным при выполнении предыдущих ячеек.

Контекст хранится только на время сессии работы с документом, но результаты выполнения ячеек хранятся в самом документе, пока их не сбросят, или не перезапустят кодовую ячейку

Важно заметить также, что оболочка Jupyter - интерактивная, и одна из её особенностей в том, что она не накладывает ограничений на то, в каком порядке и сколько раз выполняются кодовые ячейки. Это следует учитывать, и понимать, что при произвольном выполнении кодовых ячеек результат может оказаться, во-первых, не предсказуем, а во-вторых - навряд ли воспроизводимым.

2. Варианты адаптации Jupyter к HDL коду

Как было сказано выше, в среде Jupyter для выполнения кода подключается специальное “ядро”. И, поскольку среда позволяет дополнять её сторонними ядрами, это даёт возможность выполнять код на том языке, для которого изначально она не была рассчитана. Однако, в нашем случае таким путём идти не следует, т.к. при замене ядра мы потеряем возможность выполнять код Python вместе со множеством его пакетов, которые мы могли бы дополнительно использовать для своих целей — например, особую визуализацию результатов симуляции, сбор статистики и т.п.

У Jupyter есть другой механизм расширения его функционала - “магия” (magic) - некоторый префикс для строки кода или для целой кодовой ячейки. Строка или ячейка с таким префиксом обрабатывается особым образом.

Префикс магии строки начинается с одного символа `%`, за которым сразу следует имя магии, а префикс магии ячейки начинается с символов `%%`, за которым также следует имя. Префикс строковой магии должен быть в самом начале строки, а магия ячейки должна быть описана строго в первой строке ячейки. В той же строке указываются аргументы для вызываемой магии.

У Jupyter уже есть встроенная магия, которая позволяет устанавливать пакеты python, программные пакеты, листать каталоги файловой системы, и даже пользоваться командной оболочкой ОС, а также ещё многое другое.

Вот небольшие примеры магии, чтобы прояснить ситуацию:

```
# Установка пакета python
%pip install pandas
```

```
# Магия ниже покажет содержимое рабочего каталога
%ls -al .
```

```
%%bash
# Здесь мы можем выполнить целую пачку команд так же как и в обычной командной оболочке
# К примеру загрузить исходный код проекта, о котором далее пойдёт речь
git clone https://github.com/Godhart/vdf.git

# И также как и в примере выше показать содержимое рабочего каталога
ls -al .
```

```
# Показать всю доступную магию
%lsmagic
```

```
# Загрузить магию из внешнего источника
%load_ext vdf
```

Из приведённых примеров несложно догадаться, что магия может быть опасной. И не только магия, но и обычные кодовые ячейки, поэтому документы из недоверенных источников стоит выполнять в изолированном окружении - в виртуальных машинах или контейнерах.

Как и в случае с ядром, в Jupyter дополнительная магия может быть загружена из внешних источников. И уже существует предостаточно магии для интерактивной компиляции и выполнения кода различных языков программирования, но решения для HDL всё так же найти не удалось.

3. Постановка задачи

Поскольку такая задача выглядит интересной, а вопрос - наболевшим, было решено им заняться. Поставленная задача минимум:

- Уметь вводить несложный HDL код небольшими фрагментами без необходимости ввода файлов целиком. Желательно, чтобы при этом фрагменты кода были атомарными, т.е. в одном фрагменте можно было описать всё необходимое для той или иной функции без необходимости разбиения на несколько фрагментов.
- Уметь компилировать и/или выполнять код после ввода каждого фрагмента.
- В наглядной форме отображать результаты.
- Задержки должны быть в пределах разумного, чтобы сохранялась интерактивность процесса.

4. Подход к решению задачи в Jupyter

Среда Jupyter выбрана для решения этой задачи в силу уже отработанных в ней подходов; близости к распространённому и лаконичному формату Markdown; простоте запуска и работы с ней; огромным возможностям по её расширению; возможности развёртывания и работы со средой как локально, так и на удалённых компьютерах и в облаке через web интерфейс. К слову,у этой среды есть два варианта интерфейса: режим одного документа Notebook, и режим Lab, близкий к IDE, что также удобно для реализации разных сценариев работы. Кроме этого, уже существуют облачные сервисы с Jupyter, которыми можно воспользоваться для запуска кода без дополнительных усилий, если нет под рукой подходящего компьютера.

Для расширения Jupyter будет использован механизм магии. Рабочим ядром Jupyter будет ядро Python, поэтому и предпочтение в первую очередь будет отдаваться решениям на базе Python.

Для компиляции и симуляции HDL кода

будет использован пакет Cocotb, поскольку он позволяет легко интегрировать различные симуляторы и поддерживает основные языки описания аппаратуры - Verilog/SystemVerilog и VHDL, а также, в добавок ко всему, является пакетом Python.

Код HDL будет вводиться в кодовых ячейках документа Jupyter. Для этого потребуется написать специальную магию ячейки, чтобы ячейка целиком обрабатывалась особым образом, а не ядром.

Структура HDL файлов не подходит для ввода кода такими фрагментами, чтобы в них атомарно описывался некоторый функционал. Поэтому для решения поставленной задачи был выбран следующий подход:

- Файл логически разбивается на секции, которые и будут пополняться с помощью магии. Это позволит пополнять секции в желаемом для нас порядке, а не в том, в котором они должны следовать в файле.
- В кодовой ячейке в аргументах магии указывается, к какой секции документа вводится код. Введённый код каждой секции будет накапливаться в переменных контекста.
- Чтобы сократить число кодовых ячеек и обеспечить атомарность фрагментов кода, в одной ячейке может быть описание для нескольких связанных секций — например декларация сигнала и операция с присвоением ему значения.
- Перед компиляцией на диске будет создаваться файл, с использованием шаблона соответствующего типа и содержимого секций для заполнения этого шаблона.

Вот иллюстрация того, как происходит формирование файла из документа Jupyter (рисунок 1):

На примере простого файла VHDL это выглядит следующим образом:

1. Шаблон файла и используемые в нём секции в условных обозначениях:

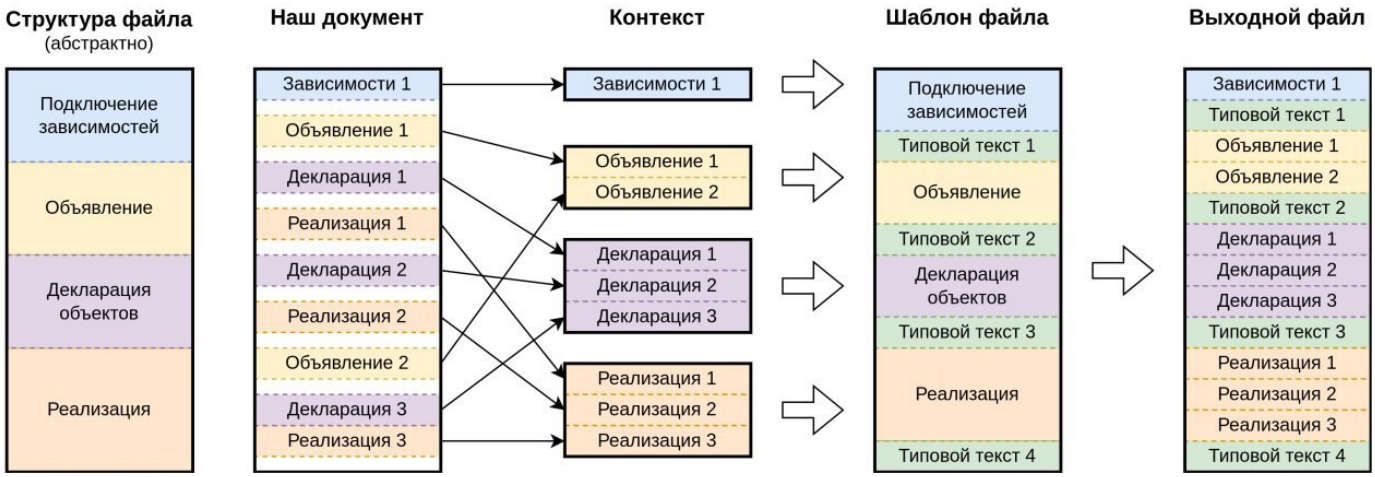


Рисунок 1. Формирование файла из документа Jupyter

```
<секция с библиотеками и пакетами, требуемыми
для объявления блока (entity)>

entity <имя файла> is
[ generic (
    <секция с параметрами блока>
);]
[ port (
    <секция с интерфейсом блока>
);]
end entity;


<секция с библиотеками и пакетами, требуемыми
для реализации блока (architecture)>

architecture behavioural of <имя файла> is

<секция с объявлениями сигналов, функций и
т.п.>

begin

<секция с непосредственным описанием
реализации>

end architecture;
```

2. Кодовые ячейки и команды:

- ввод кода секции с подключением библиотек и пакетов

```
%%vdf #header-body
library ieee;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all;
```

- комбо ввода кода секции с объявлением сигналов и секции реализации

```
%%vdf #code
signal clk : std_logic := '0';
---
clk <= not clk after 5 ns;
```

- ввод кода секции с объявлением сигналов и функций

```
%%vdf #code-declaration
signal v1  : unsigned(7 downto 0) := x"AB";
signal v2  : unsigned(7 downto 0) := x"97";
```

- комбо ввода кода секции с объявлением сигналов и секции реализации

```
%%vdf #code
signal s   : unsigned(7 downto 0) :=
                              (others => '0');
---
s <= v1 + v2 when rising_edge(clk);
```

3. Выходной файл:

```
entity main is
end entity;

library ieee;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all;

architecture behavioural of main is

signal clk : std_logic := '0';
signal v1  : unsigned(7 downto 0) := x"AB";
signal v2  : unsigned(7 downto 0) := x"97";
signal s   : unsigned(7 downto 0) :=
                              (others => '0');

begin

    clk <= not clk after 5 ns;
    s <= v1 + v2 when rising_edge(clk);

end architecture;
```

Как видно из примера, часть кода итогового файла вообще не вводится, а берётся из шаблона.

Принцип описания в магических ячейках следующий. В строке магии после префикса указывается команда и дополнительные аргументы к ней. Если используется магия ячейки, то всё, что находится в последующих строках (т.е. содержимое ячейки) является входными данными для команды - это м.б. исходный код, параметры компиляции, вывода и т.п. Формат содержимого ячейки (т.е. входных данных) зависит от команды, а также её аргументов.

Команды и аргументы начинаются с символа #, перед командой или аргументом должен быть хотя бы один пробел.

При использовании магии ячейки, аргументы могут быть указаны в нескольких строках, следующих за первой, если концы строк (в т.ч. и первой строки) экранируются символом \.

Команды позволяют вводить исходный текст, управлять компиляцией, симуляцией и выводом результатов, формировать материалы для документации.

Для всех команд используемое имя магии - `vdf`. Так была названа спецификация, в рамках которой описывается данный подход. Имя является аббревиатурой от **Versatile Description Format**, т.е. **Универсальный Формат Описания**, немного далее вы узнаете - почему.

Хотя подобное интерактивное выполнение кода и открывает новые возможности, оно сопряжено и с определёнными неудобствами, такими как:

- может оказаться сложным связать ошибку из сгенерированного файла с исходным документом,
- недостаток подсветки синтаксиса,
- отсутствие линтинга.

Однако, стоит отметить, что по мере развития эти неудобства могут быть устранены.

5. Возможные области применения данного подхода

Т.к. в итоге мы получили не просто возможность интерактивно вводить и выполнять код (производить симуляцию), а создавать целые документы - одной из полезных моделей использования этого подхода является создание интерактивных учебных материалов (для чего среда Jupyter нередко и используется). Документ в таком виде может позволить сразу запустить код, провести в нём собственные эксперименты, и даже провести тест на усвоение материала.

А также, поскольку среда Jupyter позволяет работать с ней удалённо, и так же удалённо хранить данные, то открывается ещё одна модель использования - при запуске на общедоступном сервере такие документы можно использовать для взаимодействия в сообществе для помощи с ответами на вопросы. Можно как задать вопрос, сопроводив его кодом, который можно запустить, так и получить к нему одно или несколько решений, и сразу же их проверить.

6. Дальнейшие перспективы

Кроме этого, по мере проработки этой идеи открылись и другие модели её использования и варианты подходов, которые в итоге переродились в целую спецификацию, черновик которой есть в репозитории проекта. Спецификация помимо прочего подразумевает:

- Выполнение кодовых ячеек без использования среды Jupyter.
- Возможность использования различных форматов разметки текста в качестве основы документа, а не только Jupyter Notebook. Уже можно посмотреть на рабочий пример в формате Markdown. Соответственно, для создания документов можно будет использовать разнообразные среды разработки с дополнительными вспомогательными утилитами.
- Возможность использовать документ как единый источник истины для формирования документации, соответствующего ей кода HDL и сопутствующего программного кода. Например, для описания интерфейсов, карт регистров, структуры и т.п.
- Возможность использовать в качестве основы документа сами файлы с исходным кодом. Концепт такого применения уже есть в репозитории.

Задумывается, что полученная система сможет расширяться как с точки зрения возможностей ввода кода и выполняемых операций, так и со стороны поддерживаемых языков и форматов, поэтому описанный подход и предлагаемая спецификация могут быть применены к коду на многих других языках программирования, а не только HDL. Проект открытый, и при желании вы можете принять в нём участие и внести свой вклад.

7. Доступность и способы применения подхода

Т.к. реализация этой идеи началась не так давно, на момент написания статьи она реализована минимальным образом и слабо оттестирована. На данный момент уже можно вводить код на языке VHDL для одного исходного файла, выполнять его и просматривать временные диаграммы. Также можно формировать временные диаграммы по описанию из текста ячейки, для этих целей используется wavedrom. Список доступных на текущий момент команд приведён в репозитории проекта. Этого, в принципе, уже достаточно для того, чтобы произвести оценку такого подхода.

Для того, чтобы оценить его самостоятельно на практике, вы можете:

- Запустить среду Jupyter локально. Для этого необходимо загрузить себе репозиторий git https://github.com/Godhart/vdf, выполнить установку Python (если он ещё не установлен), недостающих пакетов и симуляторов (ghdl), запустить среду Jupyter и начать работать. В репозитории уже есть пример документа, часть которого вы уже видели в этой статье. Более подробную инструкцию для развёртывания и быстрого старта можно найти в самом репозитории.
- Также в проработке вариант запуска в облаке google colab и в контейнере docker, и в ближайшем времени они уже будут доступны.

В ближайших планах также добавить поддержку языков Verilog / SystemVerilog и дополнительные варианты запуска симуляции и просмотра результатов.

Обзор отладочной платы ALINX AXU15EGB

Заостровных Андрей

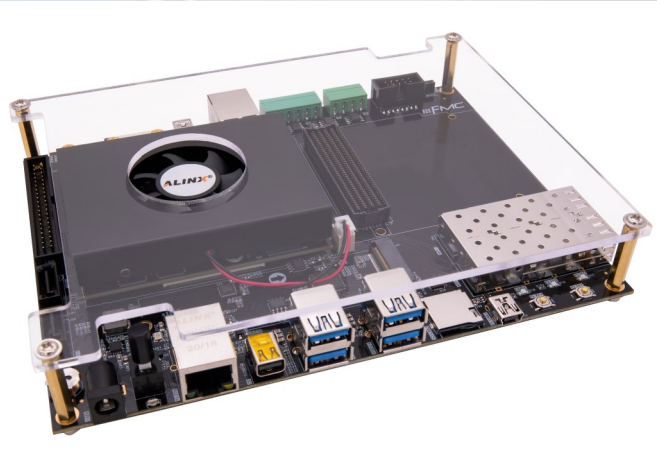
Телеграм @megalloid

Обсуждение и комментарии: link

Обзор отладочной платы ALINX AXU15EGB

Давненько ко мне в руки не попадалось ничего интересного, но ситуация поменялась. Попутным ветром принесло тут платку и я решил, мол, а почему бы мне не сделать на нее небольшой обзорчик. Отладочная плата с Zynq MPSoC от небезызвестной компании Alinx которая торгует всяким интересным на Aliexpress.

Давайте посмотрим вместе, что есть на этой плате и какие возможности предоставляет разработчикам данный отладочный набор.



Нетехническая сторона вопроса

Коснусь нетехнических особенностей данной платы. Данная отладка стоит просто космических денег - около 400 тысяч рублей. Такая стоимость явно не подъемная для простых энтузиастов и подходит лишь для профессиональной разработки и компаниям которые готовы выделить достаточно денег для ее приобретения.

Но мне показалось, что эта плата даёт возможностей соразмерно ее стоимости. Рассмотрим ее возможные применения:

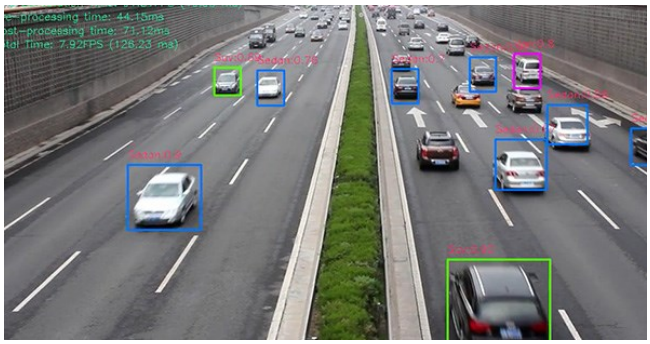
1. Automotive применения, для организации помощи водителю, бортовых систем с анализом данных с видеокамер;
2. Для организации систем беспроводной связи, SDR и т.п.;
3. Сетевые применения, где требуются мультигигабитные скорости, за счёт наличия на борту двух интерфейсов для подключения SFP/SFP+ и медного порта Ethernet. Количество портов также можно увеличить за счёт

установки дополнительных плат на FMC HPC интерфейс и сделать полноценный коммутатор или маршрутизатор.

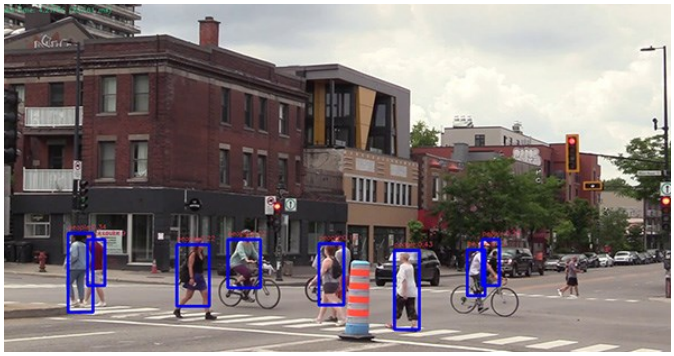
4. Высокоскоростной обмен и обработка данных с использованием параллельных вычислений в очень емкой ПЛИС и очень богатого на возможности высокоскоростного интерконнекта с ARM-ядрами.
5. Применения в области машинного обучения, искусственного интеллекта и потокового распознавания в видеонаблюдении и аналитики с потоковой обработкой видеопотока в качестве 4K.
6. Область аэрокосмических применений и многие другие.

Поковыряв документацию я нашел целую кучу изображений с применением данной платы. Среди них:

- обнаружение машин:

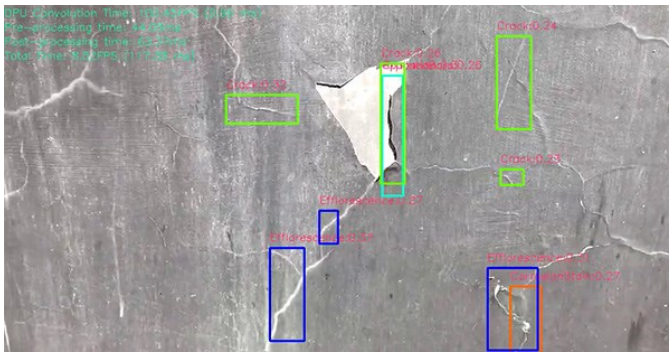


- обнаружение людей:

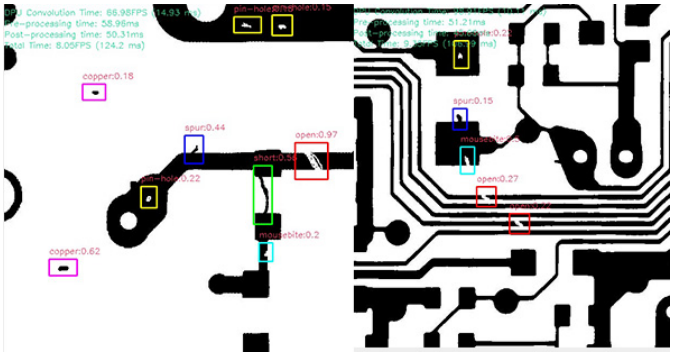


- детекция огня:

- обнаружение проблем и дефектов:



- обнаружение нарушения целостности проводников на PCB:



Вот бы иметь время все эти примеры раскурить и написать статьи на тему создания подобных проектов. Но идём дальше.

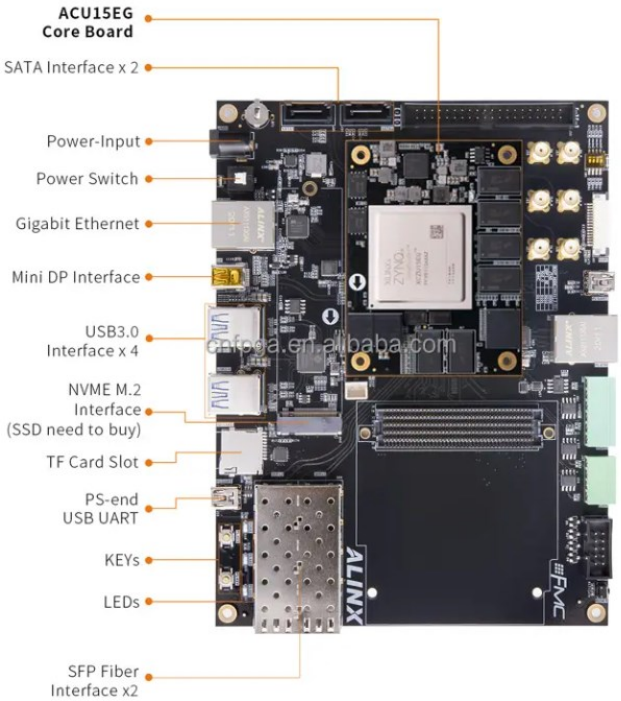
Общее описание комплекта поставки

Рассмотрим, используя изображения с сайта продавца, комплект поставки этой платы. Отладочный комплект ALINX AXU15EG состоит из двух основных компонент: SoM и материнская плата с интерфейсами.

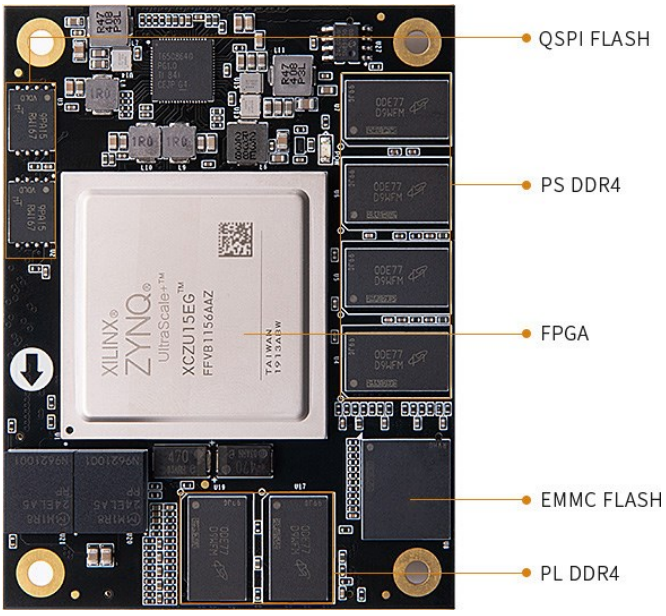
В качестве ключевого узла SoM выступает SoC Zynq UltraScale+ MPSoC,

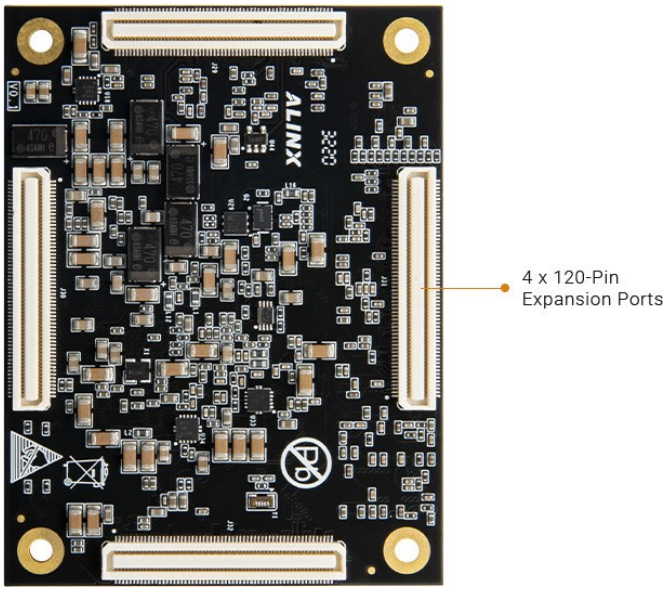
XCZU15EG-2FFVB1156I с классом скорости 2, который содержит в себе четырехъядерный процессор с ядрами Cortex-A53 и двухъядерный Cortex-R5F для реалтайм вычислений и ПЛИС UltraScale с 746550 System Logic Cells, 682560 Flip-Flops, 274080 CLB LUT и многим другим. В общем прям жЫр.

В сборе это хозяйство выглядит следующим образом:



Так выглядит SoM модуль с обеих сторон:





Помимо этого в комплекте имеется JTAG отладчик Alinx AL321 на чипе FTDI FT232H, который позволяет достаточно быстро осуществлять операции отладки в отличие от достаточно медленных чипов которые идут в Platform Cable USB, типа DLC9LP.

Выглядит он вот так и поставляется с microUSB-кабелем и кабелем подключения к отладочному разъему:

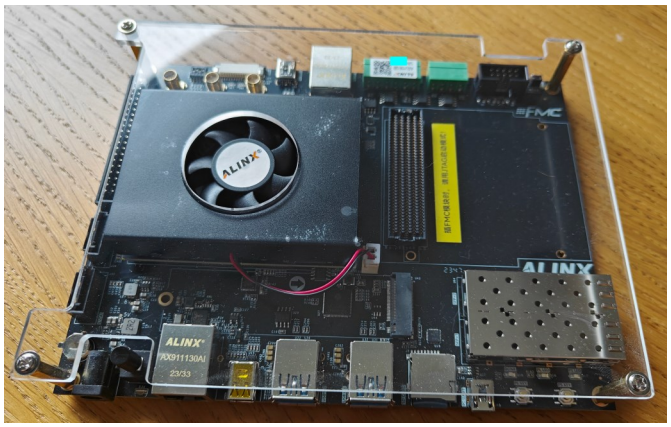
Для питания платы используется блок питания на +12V DC на 36W (3A):

Для загрузки ПО и битстрима идет в комплекте MicroSD Flash карта, может быть поставлена и на 16 Гб и на 32 Гб. Мне повезло и в этом комплекте была флешка на 32Гб. Хотя для работы на начальном этапе более чем достаточно и 4 Гб флешки:

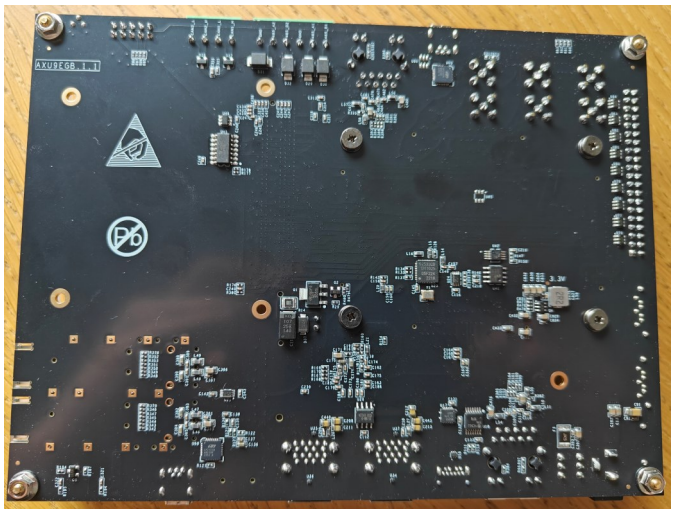
Плюсом еще докладывают Card Reader для microSD карт:

В качестве еще одного элемента поставки можно отдельно отметить систему охлаждения основного чипа в виде 30W Fan Cooler:

В качестве приятного дополнения и защиты от случайных механических воздействий имеется в комплекте плексигласовый защитный корпус:



Обратная сторона платы, к слову, выглядит следующим образом:

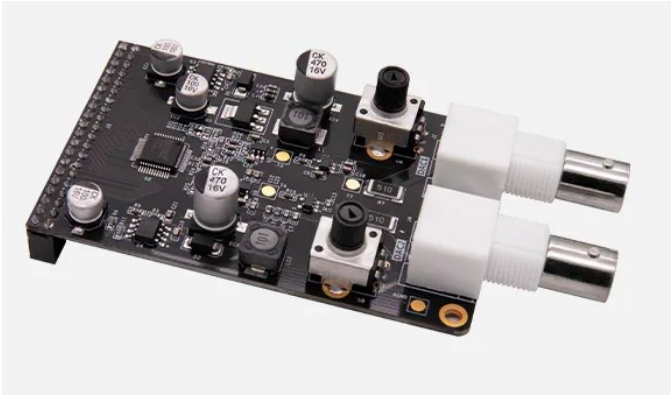


Внимательные читатели увидят на шелкографии надпись AXU9EGB 1.1. Но пусть не смущает эта надпись, несущая плата видимо используется во многих комплектах поставки.

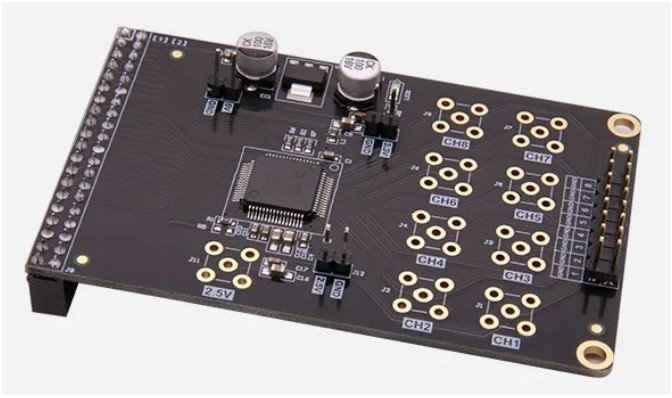
Дополнительные модули

Помимо этого у Alinx можно приобрести разные модули, которые устанавливаются в 40-пиновый разъем GPIO:

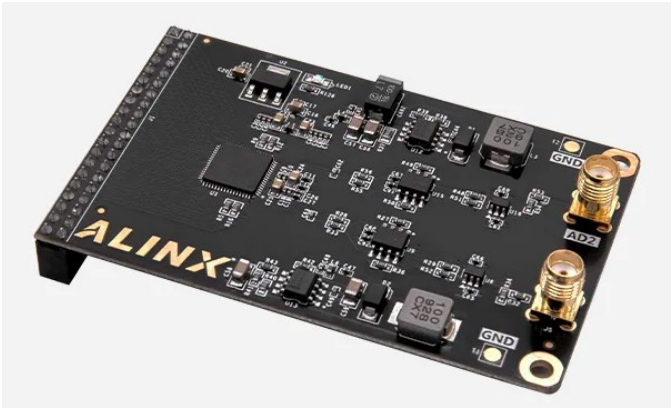
Dual Channel 14bit 125Msps DA BNC Analog Output Module AD9767;



Multi-channel 16 bit 200ksps Synchronous Sampling AD Module AD7606;



Dual-channel 65MSPS 12bit AD analog to digital signal conversion module AD9238;

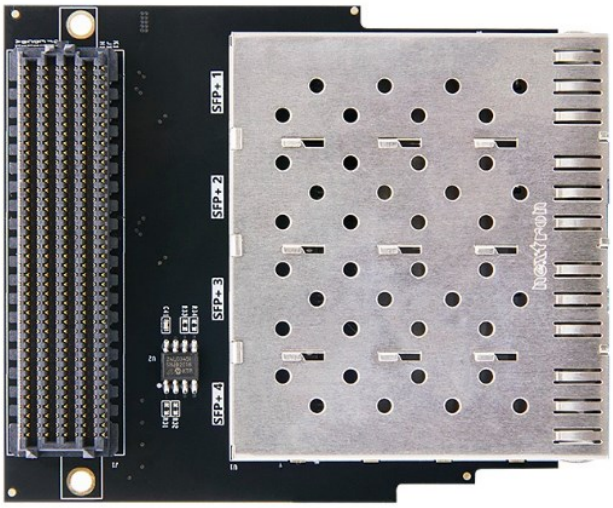


7-inch Touch Screen Module;

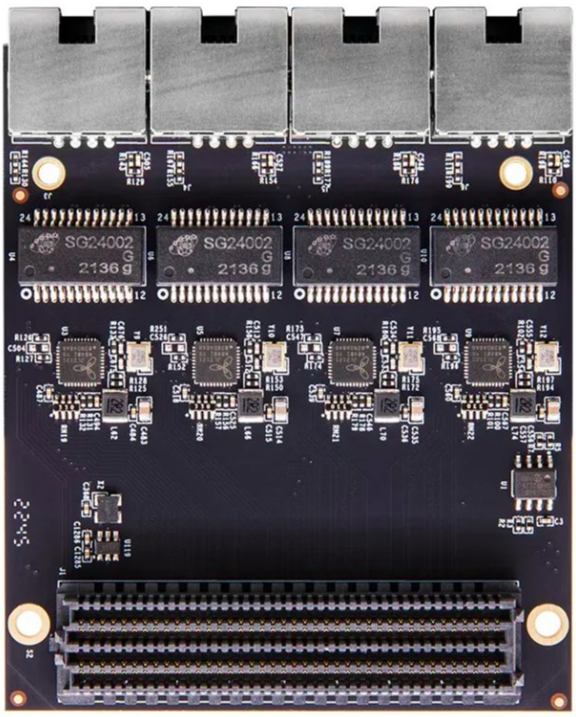
модуль для установки в MIPI-интерфейс в виде 5MP камеры на контроллере OV5640;

Также продаются разные модули для установки на FMC-разъем, среди которых:

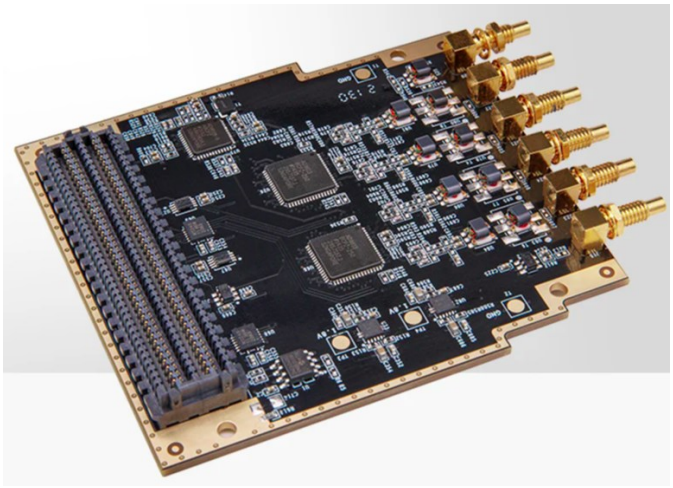
Alinx FH1223 FMC HPC to 4-Channel SFP Optical Fiber Interface Adapter Card;



Alinx FL2121 4-way 1000M Gigabit Ethernet LPC

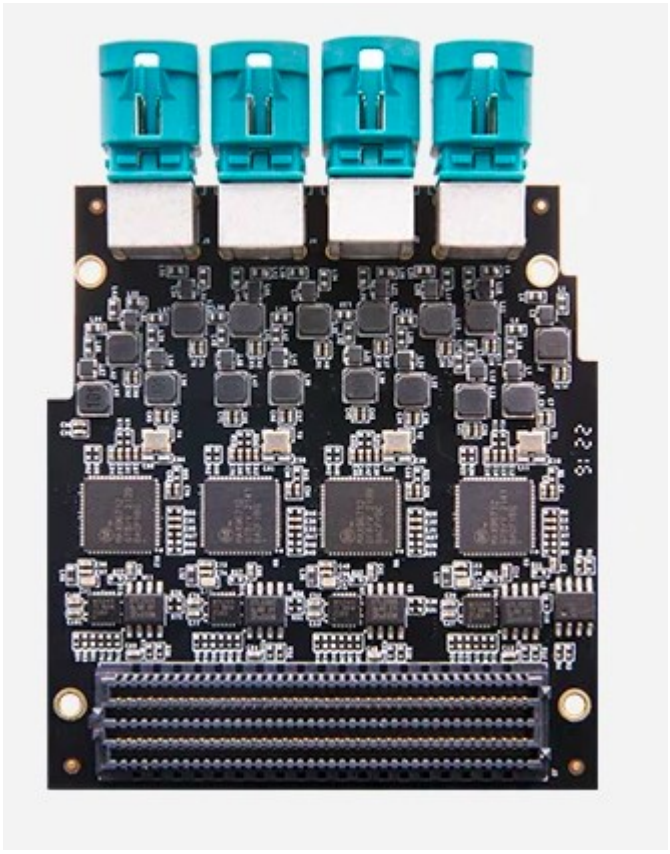


Alinx FMC HDMI CARD Daughter Board Input Output 1080p



Alinx FH9712 16 channel GMSL2/GMSL1 autopilot camera acquisition module MAX96712;

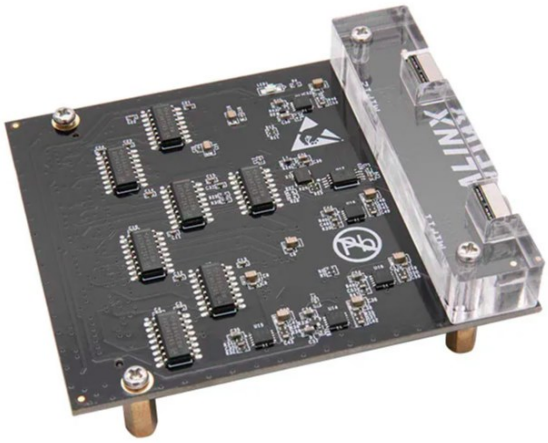
Alinx FL1404 FMC to 4-way MIPI module LPC interface



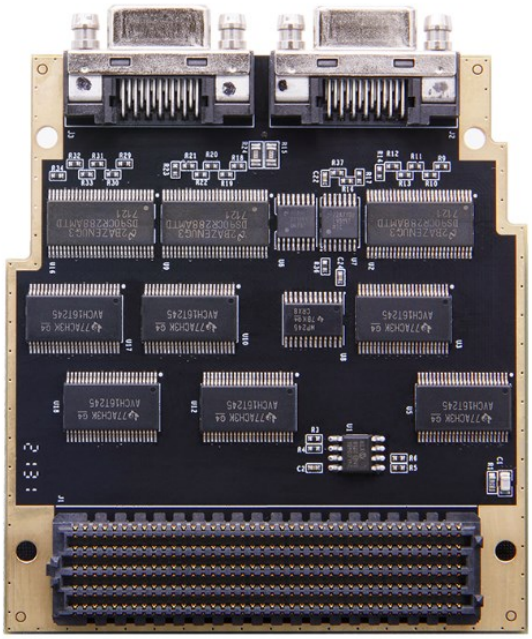
Alinx FL1010 FMC LPC Interface to 40-Pin Expansion Ports

Alinx FL9613 FMC LPC to 4-Channel AD 12-bit 250 MSPS;

Alinx FL0214 FMC LPC to Dual Lens MIPI 1.3 Megapixel IMX214 CMOS Camera



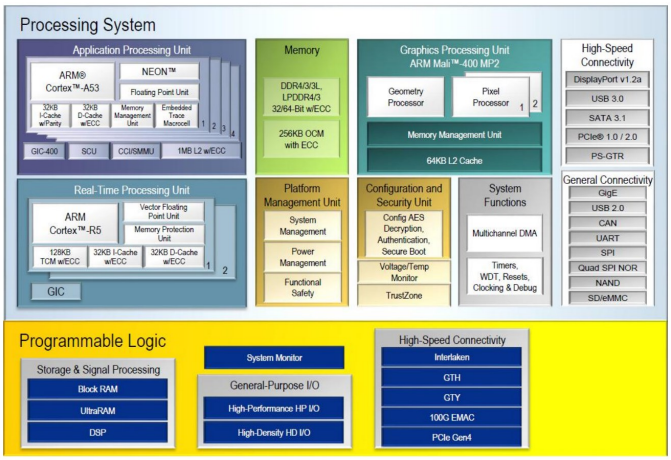
Alinx FH1226 FMC HPC to Cameralink Adapter



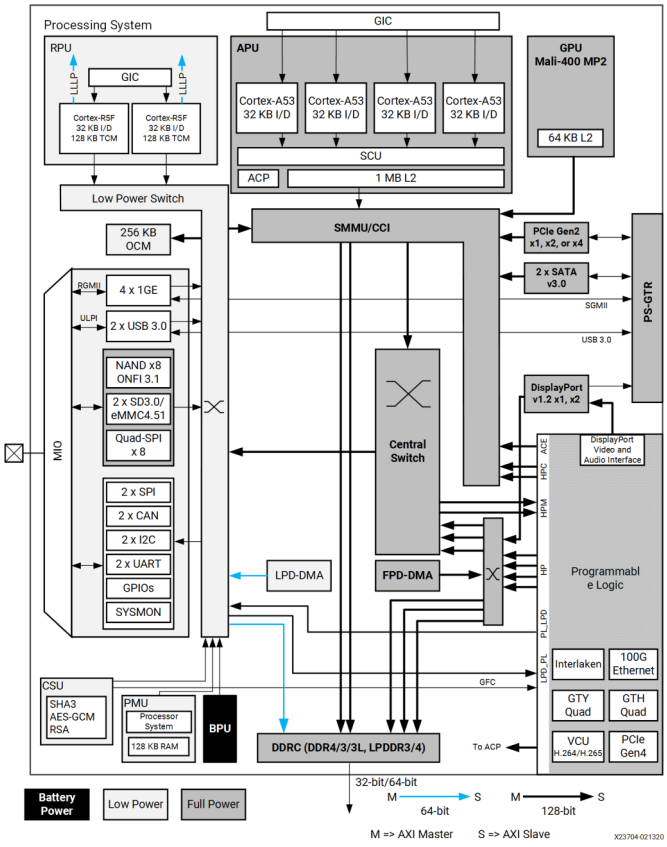
И многие другие, их можно найти на сайте Alinx или на Aliexpress. Как говорится: любой каприз за ваши деньги. Возможностей для реализации огромного спектра задач - целая уйма.

Zynq Ultrascale+ MPSoC

Широкими мазками опишу возможности центрального элемента данной отладки. Вот так возможности Zynq MPSoC отразили Xilinx в своих документах на данный чип:



Если рассматривать чип с точки зрения внутреннего технического устройства приводится следующая блок-схема:



Рассмотрим подробнее встроенный четырехъядерный ARM-процессор и я приведу несколько его отличительных особенностей:

- архитектура основного процессора ARMv8-A с рабочей частотой CPU до 1.5 GHz;
- медиапроцессор с поддержкой инструкций NEON Advanced SIMD;
- Floating Point Unit (FPU) с поддержкой

вычислений одинарной и двойной точности;

- CoreSight и Embedded Trace Macrocell (ETM) для расширенных функций отладки;
- Accelerator Coherency Port (ACP);
- AXI Coherency Extension (ACE);
- два кэша размером 32 KB L1 и кэш L2 1 MB;

Отдельной фишкой данного SoC является наличие двухядерного процессора Cortex-R5F с Real-Time Processing Unit (RPU):

- основан на архитектуре ARMv7-R с рабочей частотой до 600 MHz;
- Floating Point Unit (FPU) с поддержкой вычислений одинарной и двойной точности;
- CoreSight и Embedded Trace Macrocell (ETM) для расширенных функций отладки;
- кэш в 32KB L1 и кэш в 128 KB TCM с ECC;

Помимо этого присутствует On-Chip Memory:

- 256KB on-chip RAM (OCM) в PS с ECC;
- 36 Mb on-chip RAM (UltraRAM) с ECC в PL;
- 35 Mb on-chip RAM (block RAM) с ECC в PL;
- 11 Mb on-chip RAM (distributed RAM) в PL;

Для графики на чипе есть отдельный GPU ARM Mali-400 MP2 с частотой до 600 MHz:

- поддержка OpenGL ES 1.1 и 2.0;
- поддержка OpenVG 1.1;

Контроллеры памяти:

- мультипротокольный контроллер динамической памяти;
- контроллер памяти с шириной шины в 32 или 64 бита с поддержкой DDR4, DDR3, DDR3L, или LPDDR3 памяти и 32-битный интерфейс для LPDDR4;
- поддержка ECC для 32-битного и 64-битного режима работы;
- адресное пространство до 32GB;
- интерфейсы для статической памяти eMMC 4.51, NAND с поддержкой ONFI3.1 и SPI/QSPI интерфейс для подключения NOR Flash;
- два контроллера DMA, с 8 каналами каждый и поддержкой Memory-to-memory, memory-to-peripheral, peripheral-to-memory, и scatter-gather транзакций.

Также на борту имеется поддержка последовательных трансиверов:

- четыре PS-GTR трансивера с поддержкой скорости в 6.0 Gb/s;
- поддержка SGMII tri-speed Ethernet,

Очень богат данный чип на выделенные периферийные устройства ввода-вывода и интерфейсы:

- PCI Express Gen2 x1, x2, или x4 с Root complex и End Point конфигурации;
- Serial-ATA 3.1 (SATA) с поддержкой скорости передачи данных в 1.5, 3.0, и 6.0 Gb/s
- Display Port контроллер с поддержкой рейтов до 5.4 Gb/s и двумя Tx-линиями, без поддержки Rx поддерживающий режимы работы 4K@30Hz или 1080P@60Hz;
- Четыре 10/100/1000 Ethernet MAC и интерфейсами GMII, RGMII, SGMII;
- Два контроллера USB 3.0/2.0 для Device/Host/OTG с поддержкой до 12 эндпоинтов и Super-speed, high- speed, full-speed, и low-speed режимов работы;
- Два CAN 2.0B совместимых интерфейса;
- стандартные UART, SPI, I2C, RTC, WatchDog контроллеры;
- 78 MIO (PS) и 96 EMIO (PL) выводов GPIO;

Для реализации интерконнекта используется ARM AMBA AXI4-based

контроллер. Ну и многое другое.

Область программируемой логики PL

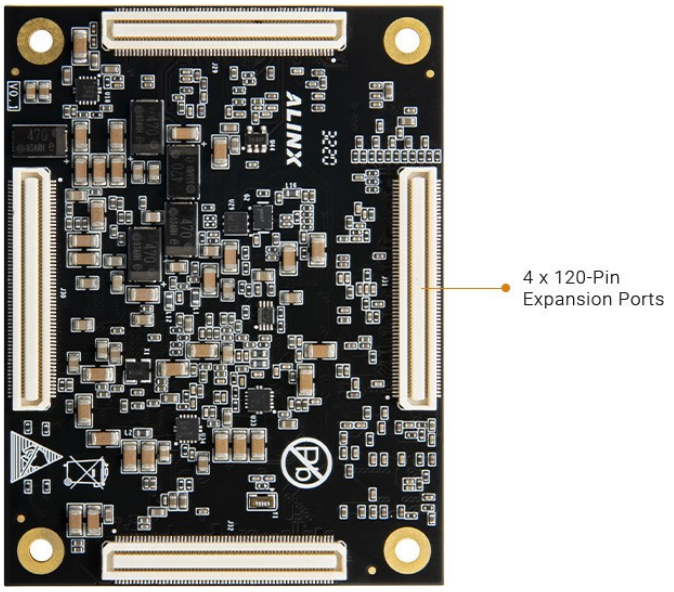
Тут тоже очень богато. Перечислю доступные ресурсы, предоставляемые ПЛИС:

- 746550 System Logic Cells;
- 682560 CLB Flip-Flops;
- 341280 CLB LUTs;
- 42660 Logic Array Block;
- 11.3 Mb Distributed RAM;
- 744 Block RAM Blocks;
- 26.2 Mb Block RAM;
- 112 UltraRAM Blocks;
- 31.5 Mb UltraRAM;
- 3528 DSP Slices;
- Clock Signal Management (CMT) 4 штуки;
- 24 GTH Transceiver 16.3Gb/s;
- Transceiver Fractional PLLs 12 штук;
- DSP блоки с 27 x 18 знаковыми вычислениями, 48-битным adder/ accumulator и 27-битным pre-adder;

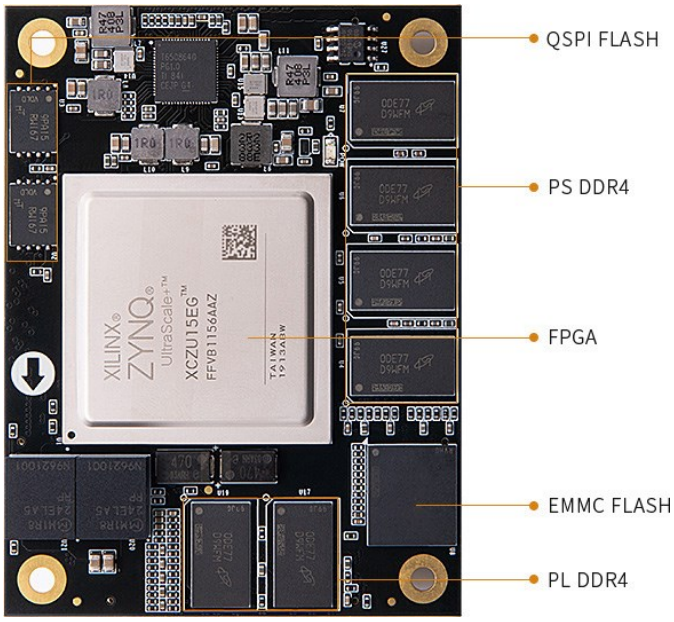
И многое другое, что можно очень долго перечислять...

SoM и его содержимое

Теперь настало время препарировать схематик SoM-а и подробнее посмотреть, что на нем есть и какие компоненты установлены. Начнем с того, что он состоит из 4-ёх 120 пиновых коннекторов, которые установлены так, чтобы SoM было невозможно воткнуть на плату в неправильном положении. Освежим в воспоминаниях картинку:

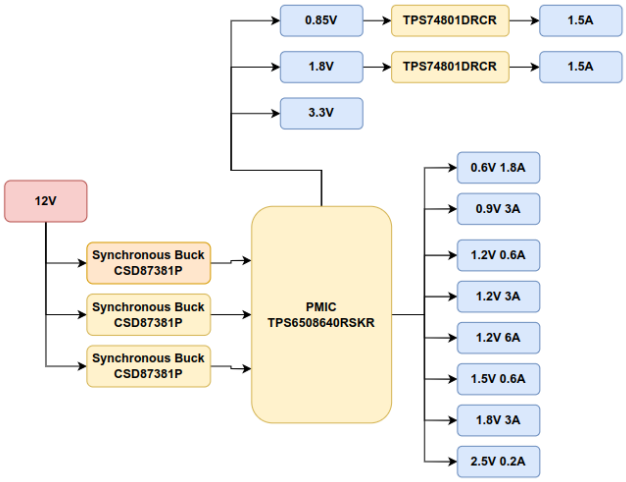


Теперь подробнее рассмотрим верхнюю часть SoM-а:



Питание на SoM

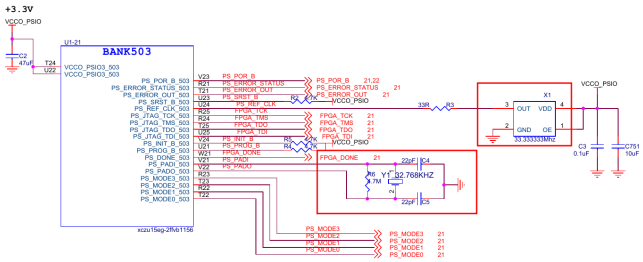
Для того чтобы сформировать набор разных напряжений - на SoMe установлен PMIC-котроллер TPS6508640RSKR в связке с большим количеством обвязки в виде линейных, синхронных, импульсных преобразователей и большого числа пассивных компонентов:



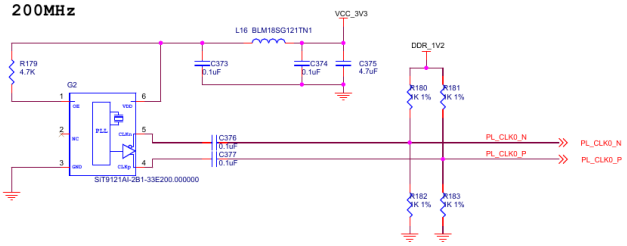
В целом всё достаточно просто, потом можете посмотреть самостоятельно схематик и сделать свои выводы.

Тактирование на SoM

На SoM установлен основной осциллятор 33.3333 MHz и низкочастотный генератор 32.768 kHz для RTC:



И дополнительно на SoM имеется генератор дифференциального тактового сигнала в 200MHz:



В целом достаточно стандартно и под стать всей периферии и сложности самой платы.

Оперативная память на SoM

На SoM-е установлено 6 чипов DDR4, часть из которых подключена к PS-части, а другая часть к PL. Если быть несколько конкретнее то:

- к PS-части подключено 4 чипа DDR4 MT40A512M16LY-062E общим объемом 4 Гбайта с 64-битной шириной полосы пропускания;
- к PL-части подключено 2 чипа DDR4 MT40A512M16LY-062E общим объемом 1 Гбайт с шириной пропускания в 32 бита;

Максимальная рабочая скорость DDR4 подключенной к PS может достигать 1200 МГц со скоростью передачи данных в 2400 Мбит/с. Подключается эта память к интерфейсу памяти BANK504 PS.

Максимальная скорость DDR4 SDRAM подключенная к PL так же может достигать 1200 МГц с такой же скоростью передачи данных. И подключается эта память к BANK64 в PL.

Постоянная память на SoM

Если рассматривать постоянную память, на SoM установлено следующее:

- eMMC MTFC8GAKAJCN-4M объемом 64 Гбайта;
- QSPI Flash MT25QL512ABB1EW9-0SIT объемом 64 Мбайта подключенный к QSPI0 PS-части;
- QSPI Flash MT25QL512ABB1EW9-0SIT объемом 64 Мбайта подключенный к QSPI1 PS-части;
- I2C EEPROM 24LC04B-I/SN объемом 4 Кбит;

Содержимое отладочной платы

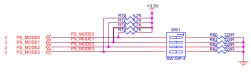
Как я уже упоминал выше, плата состоит из двух основных компонентов, SoM с Zynq и несущей платой, которая предоставляет возможность подключения периферийных

устройств и сетевых подключений.

Перечислю все основные возможности которые предоставляет несущая плата:

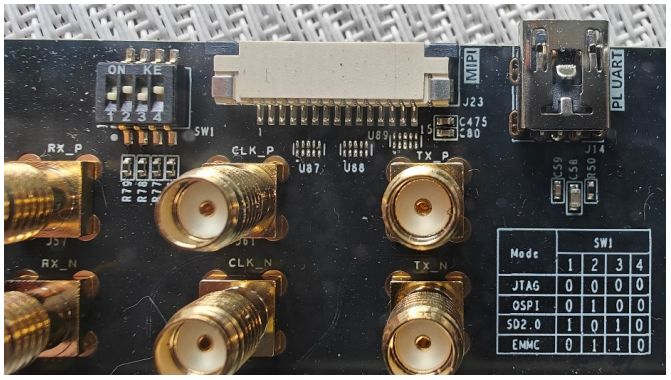
- PS High Speed интерфейс PCIe Gen2 x4 с интерфейсом M.2;
- 2 разъема USB3.0 и 2 разъема USB 2.0;
- 2 разъема SATA 3.1
- mini-DisplayPort с возможность передачи 4K@30Hz или 1080P@60Hz;
- PS Gigabit Ethernet;
- PL Gigabit Ethernet;
- SD/SDIO интерфейс для подключения microSD карт;
- 2 UART microUSB интерфейса для подключения к PS и PL
- 2 CAN 2.0B интерфейса;
- 2 RS-485 интерфейса;
- MIPI-интерфейс;
- FMC-интерфейс к которому можно подключить Alinx FMC модули;
- PL Ethernet 12.5Gb/s x16 идущие на порта для подключения SFP+;
- Температурный сенсор LM75;
- Real Time Clock с батарейным держателем;
- 40 пиновый GPIO интерфейс;
- 4 LED;
- 1 кнопка сброса и 2 пользовательских кнопки;

Помимо этого на плате имеется служебный переключатель для выбора источника загрузки:



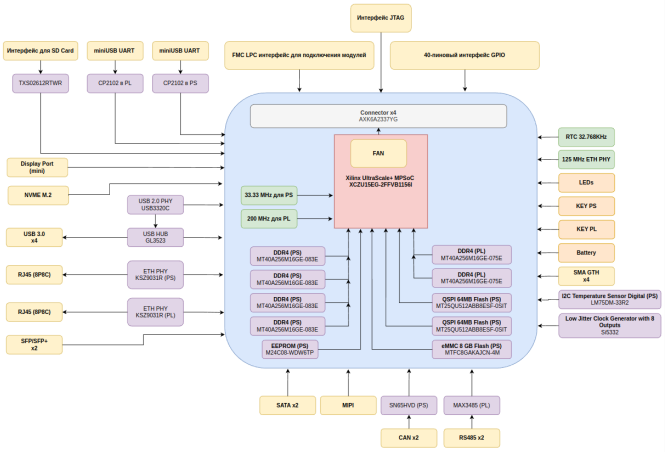
MODE[3:0]	BOOT MODE	Descritpion
0000	PS JTAG	PS JTAG Interface
0001	Quad_SPI(24b)	24-Bit addresssing(QSPI24)
0010	Quad_SPI(32b)	32-Bit addresssing(QSPI32)
0011	SD0(2.0)	SD2.0
0100	NAND	Requires 8-bit data bus width
0101	SD1(2.0)	SD2.0
0110	eMMC(1.8V)	eMMC version 4.5 at 1.8V
0111	USB0(2.0)	USB 2.0 only
1000	PJTAG(MIO #0)	PJTAG connection 0 option
1001	PJTAG(MIO #1)	PJTAG connection 1 option
1110	SD1 LS(3.0)	SD 3.0

Фото переключателя на плате:



А так же имеется 6 SMA интерфейсов для подключения GTH-трансивера:

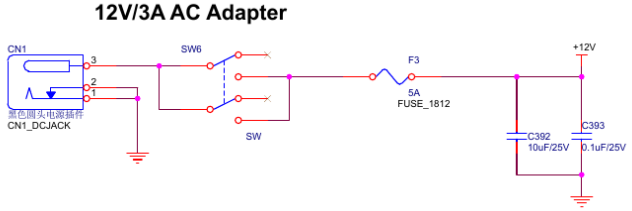
Проанализировав состав платы я накидал схему, отображающую все, что есть на плате:



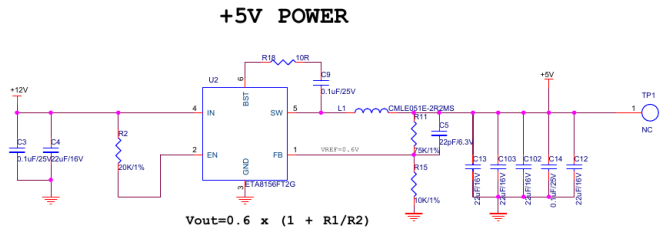
По порядку рассмотрим каждый из пунктов чуть подробнее.

Питание

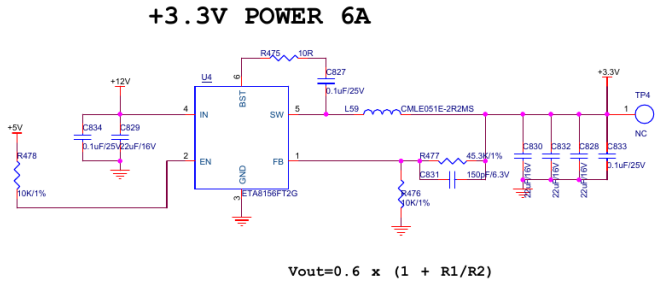
Плата запитывается от 12В через самый обычный штыревой разъем, подойдет любой качественный блок питания с выходным напряжением 12В и мощностью 36W.



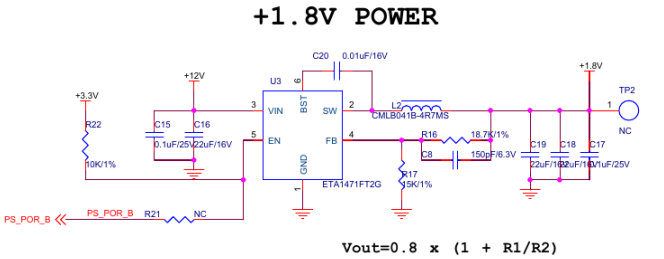
Помимо этого на плате имеется несколько преобразователей из 12V в целевые напряжения. Из 12V в 5V:



Из 12V в 3.3V:



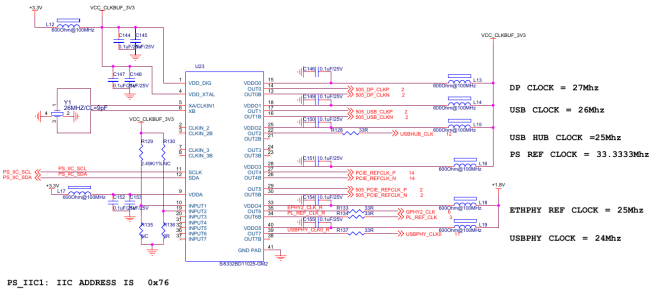
И из 12V в 1.8V:



Плюсом к этому на плате имеются и тест-поинты для контроля качества питания.

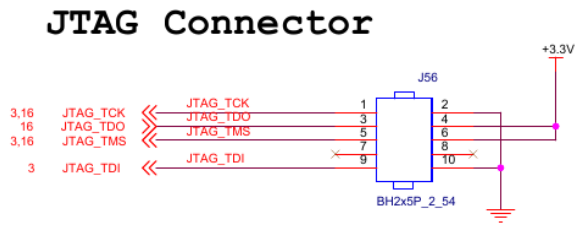
Источники тактирования

Очень интересным компонентом на плате является программируемый источник тактовой частоты Si5332BD11025-GM2, который позволяет генерировать весь необходимый спектр тактовых частот для всей необходимой периферии:



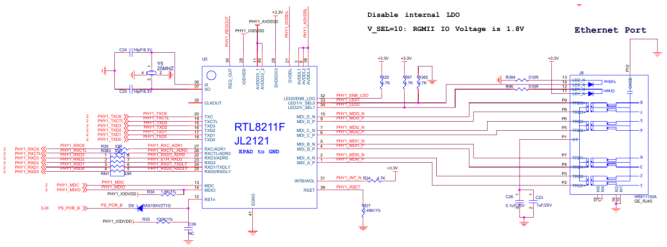
JTAG разъем

Для отладочных целей на плате имеется стандартный для всех отладок JTAG-интерфейс для подключения отладчика:

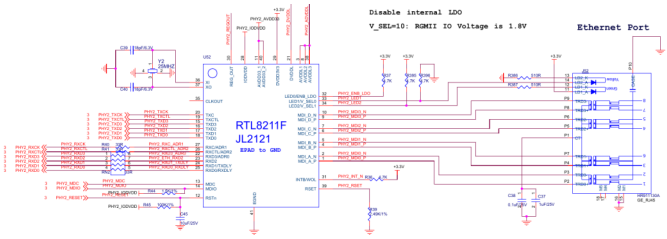


Сеть

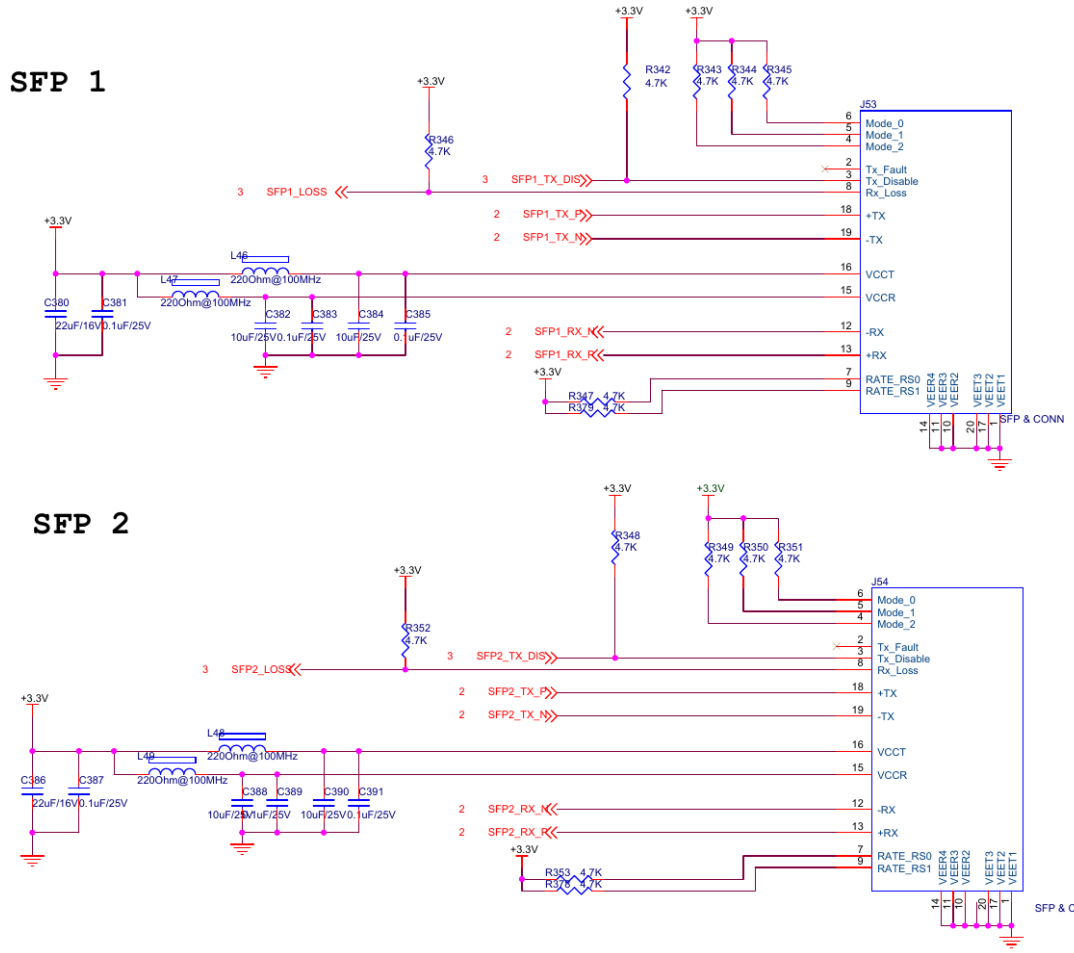
Данная плата также предоставляет обширные возможности для реализации сетевых сценариев. В первую очередь обратим внимание на два медных Ethernet-разъема. Первый - подключен к PS-части Zynq. Линии MDIO из раъема идут к PHY Ethernet от Realtek RTL8211F. От PHY к PS идёт подключение через интерфейс RGMII:



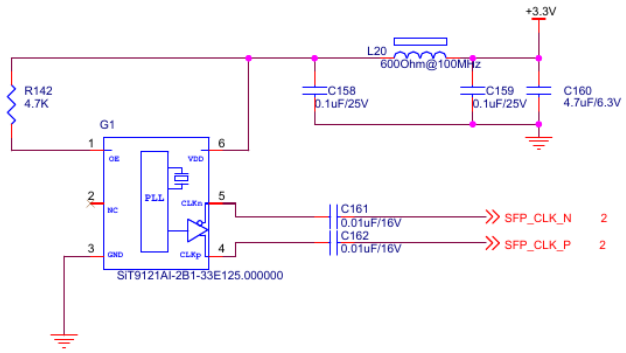
Второй подключен к программируемой логике по тому же принципу с тем же PHY Ethernet-контроллером:



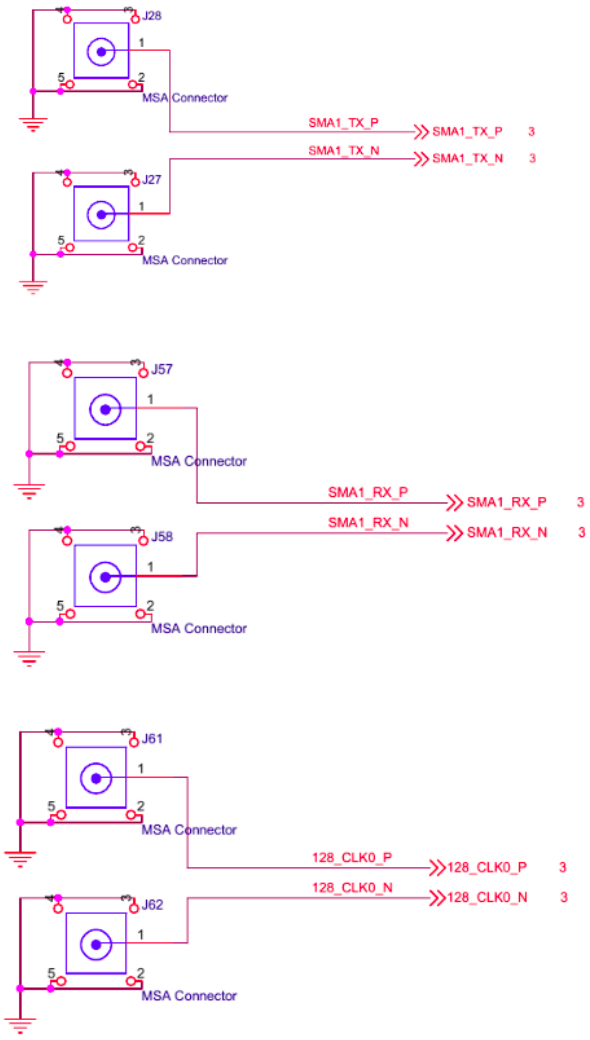
Помимо этого на плате установлен разъем для подключения 2-ух SFP/SFP+ модулей. Оба SFP-разъема подключаются к PL-логике устройства:



Помимо этого имеется еще генератор дифференциального тактового сигнала для работы трансиверов SiT9121AI-2B1-33E125.000000 с выходной тактовой частотой в 125МГц:



Помимо вполне стандартных сетевых интерфейсов на плате имеется подключение через SMA-разъемы к GTH-интерфейсам с возможностью передачи данных до 100 Гбит/с:

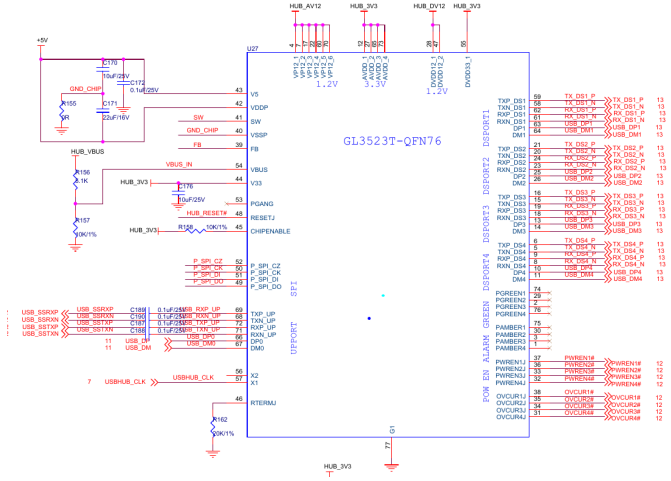


Интерфейс FMC

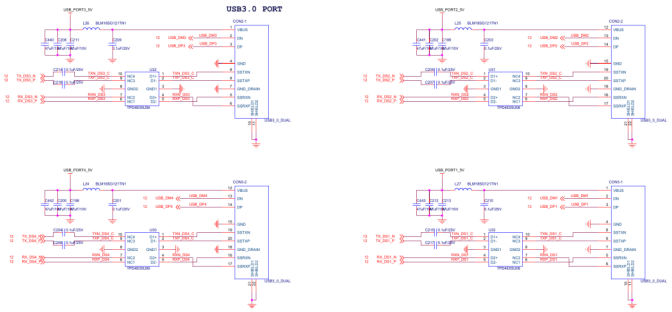
На плате имеется FMC-разъем для подключения большого количества различных модулей, которые я перечислил в начале статьи. Схематик в виду большого его размера - прикладывать тут не буду. Приведу лишь фото интерфейса:

Интефейсы USB

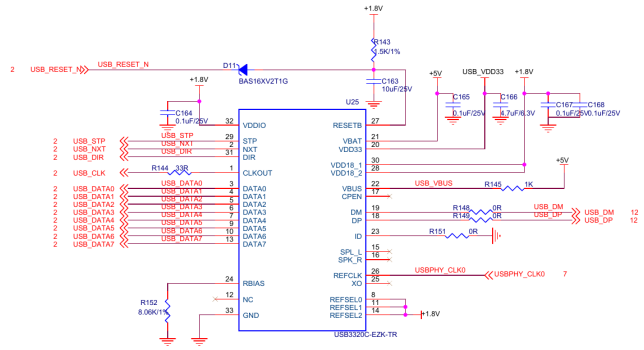
В части USB-подключений плата также предоставляет целую кучу возможностей. На плате установлен USB3.1 хаб GL3523T-QFN76 который подключен к соответствующей части PS-части Zynq MPSoC:



К хабу подключены 2 USB3.0 и 2 USB2.0 разъема. Правда на схеме все 4 почему-то обозначены как USB3.0-разъема:

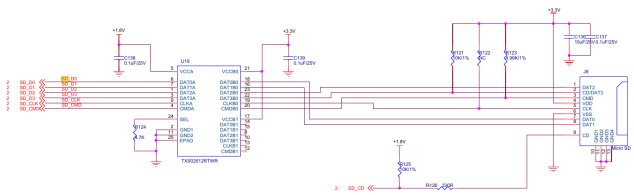


Помимо этого на плате установлен USB2.0 PHY контроллер USB3320C-EZK-TR, который подключен к USB-хабу:



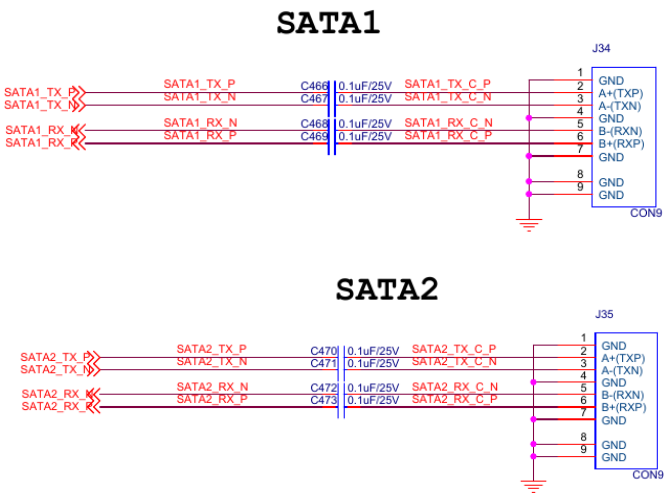
Интерфейс microSD

На плате имеется SD-разъем, для подключения внешнего накопителя данных или источника загрузки. Разъем на плате подключается к Zynq осуществляется через расширитель портов SDIO с преобразователем уровня логического сигнала 1TXS02612RTWR:



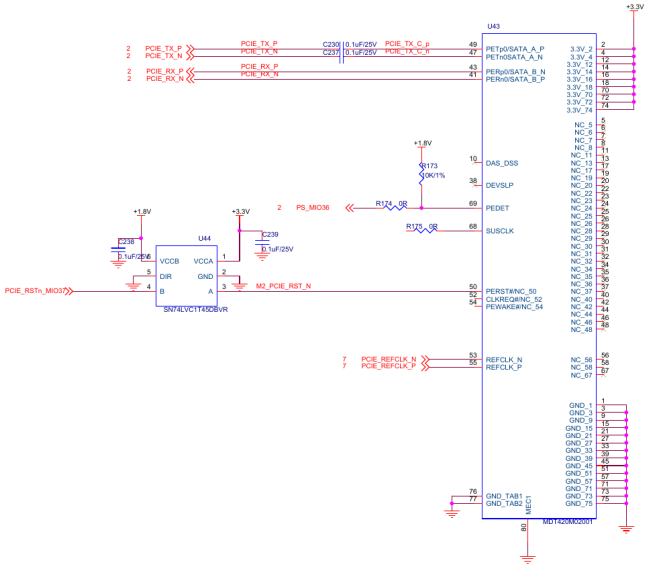
Интерфейс SATA

Еще одна интересная возможность, которая предоставляется платой - это возможность подключить до 2-х SATA устройств, будь то SSD или HDD:



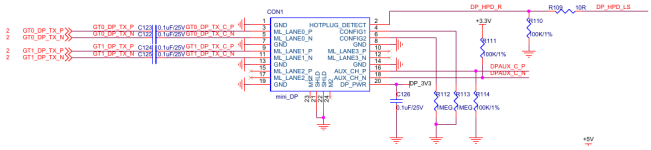
Интерфейс M.2

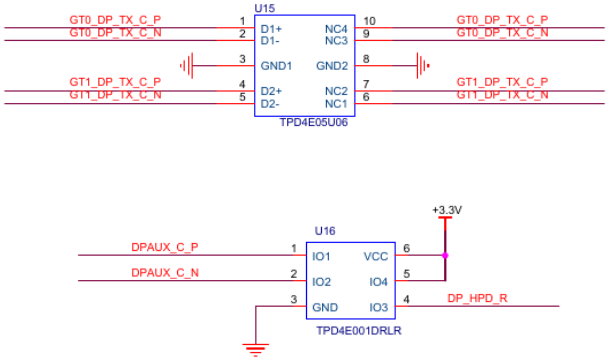
Приятная особенность платы - наличие M.2 разъема с ключом M. Сюда возможно подключить любое подходящее PCI-e устройство, например SSD-диск подходящего форм-фактора:

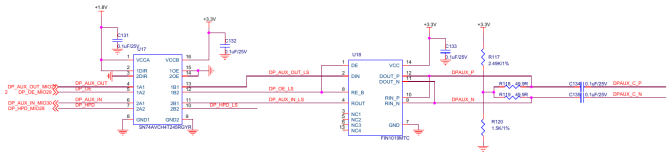


Интерфейс Display Port

Для подключения внешнего дисплея, вместо HDMI, на плате использован интерфейс Display Port:

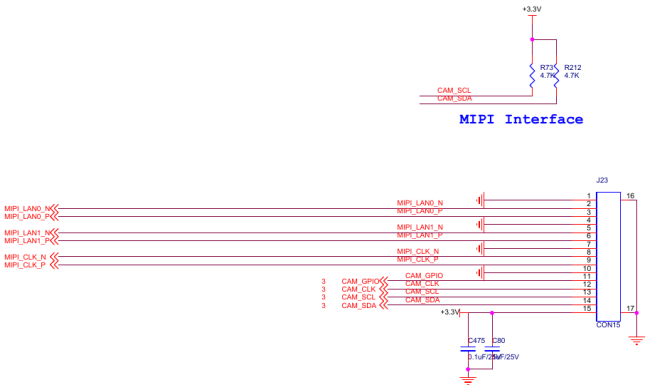






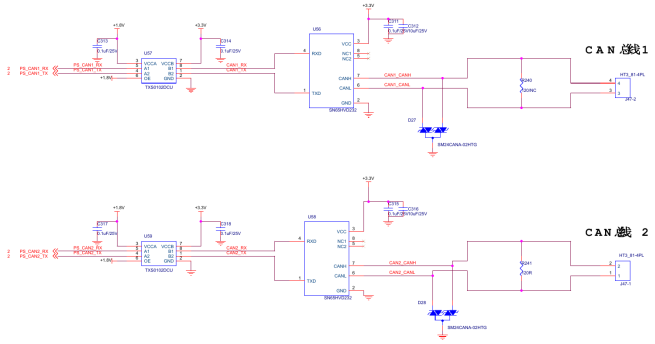
Интерфейс MIPI

Для подключения внешних мультимедиа-устройств, будь то дисплей или камера - на плате предусмотрен интерфейс MIPI:



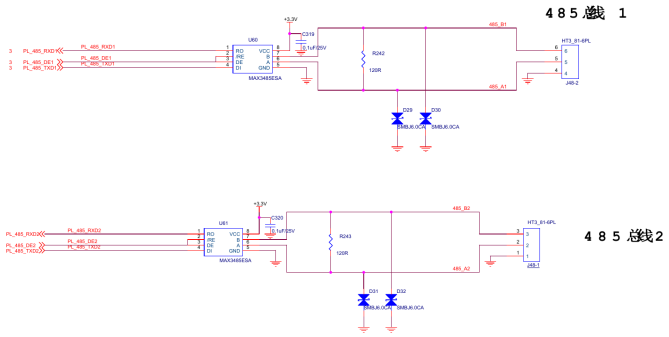
Интерфейс CAN

В случае, если плата разрабатывается под какие-либо automotive-применения - выведен интерфейс CAN с подключением в клеммный разъем HT3.96-4P к контроллеру SN65HVD232 через преобразователь логических уровней TXS0102DCU. Сигнальные линии подключаются к PS-части Zynq:



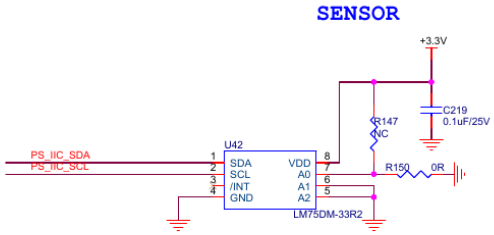
Интерфейс RS-485

Для промышленных применений на плате реализован вывод интерфейса RS-485 через приемопередатчик MAX3485ESA. Подключается к PL-части Zynq:



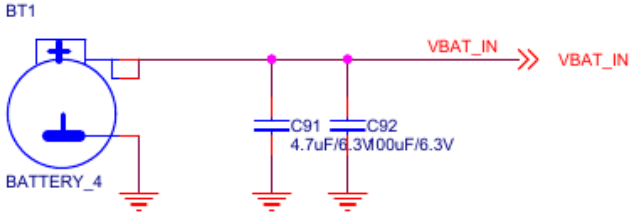
Температурный I2C сенсор

Для случаев, когда потребуется мониторинг внешней температуры - на плате имеется I2C температурный сенсор LM75DM-33R2, который подключается к PS-части Zynq:



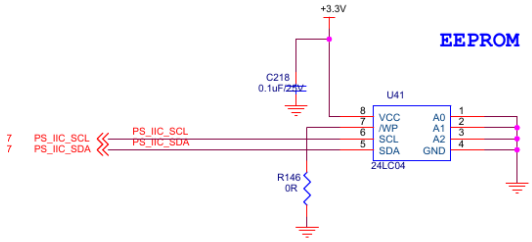
Батарейка

На плате имеется батарейный держатель, который снабжает соответствующий банк питания в PS-части:



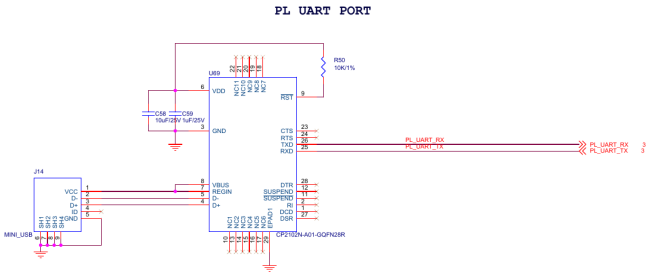
EEPROM

Помимо всего прочего на плате имеется I2C EEPROM 24LC04 объемом 4 Кбит, которая подключена к PS-части устройства:

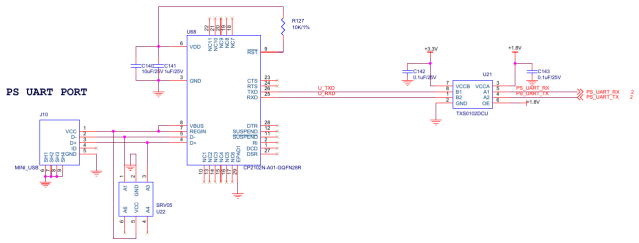


UART-интерфейсы

Для коммуникации с внешними устройствами и работы с консолью Zynq - на плате имеется два интерфейса UART, первый подключен к PS-части устройства, второй подключен к PL-части устройства. В обоих случаях подключение осуществляется с использованием USB-to-UART конвертера CP2102N-A01-GQFN28R по miniUSB-порту:

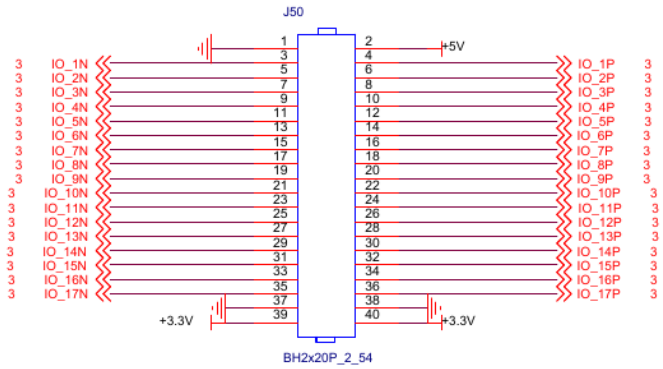


К PS-части UART-интерфейс подключается через преобразователь логических уровней 3.3V-to-1.8V TXS0102DCU:



Интерфейсы GPIO

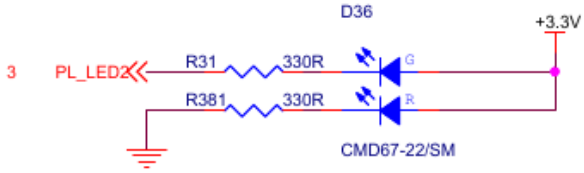
В дополнение к широчайшим возможностям отладки - на плате имеется стандартная 40-пиновая гребенка с GPIO-пинами:



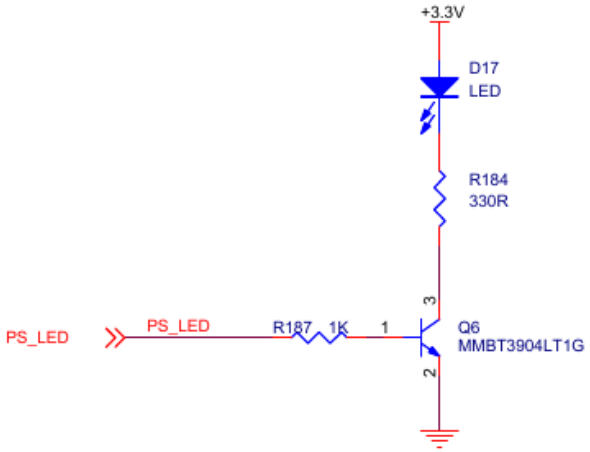
LED & Buttons

Для того, чтобы реализовать простую индикацию и базовое взаимодействие с пользователем - на плате имеется несколько LED-индикаторов и кнопок.

Первый - индикатор питания 3.3V и второй LED подключен к PS-части:

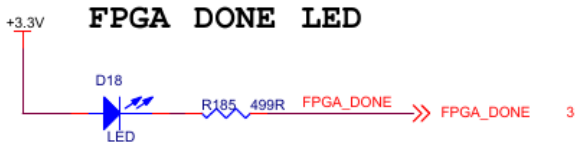


Имеется второй LED-индикатор, который подключен к PS-части:

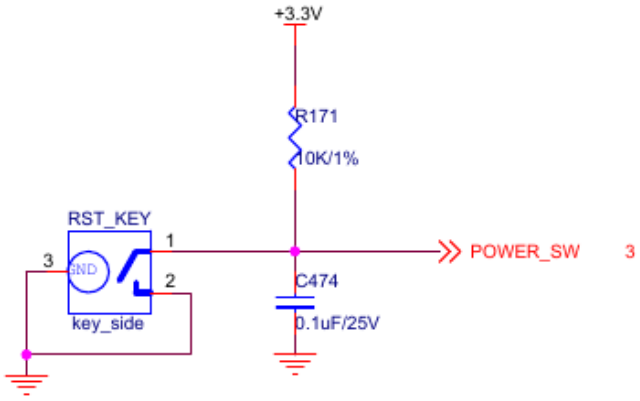


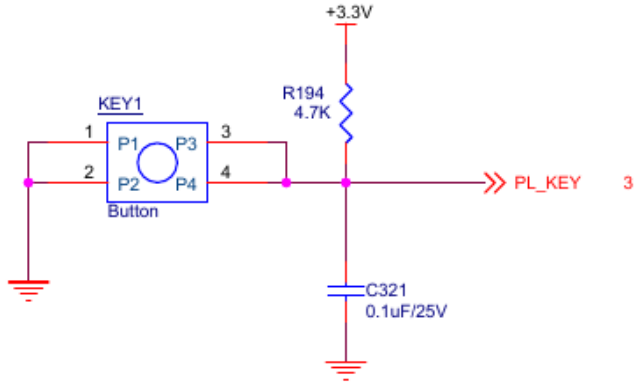
Третий LED-индикатор подключен к PL-части:

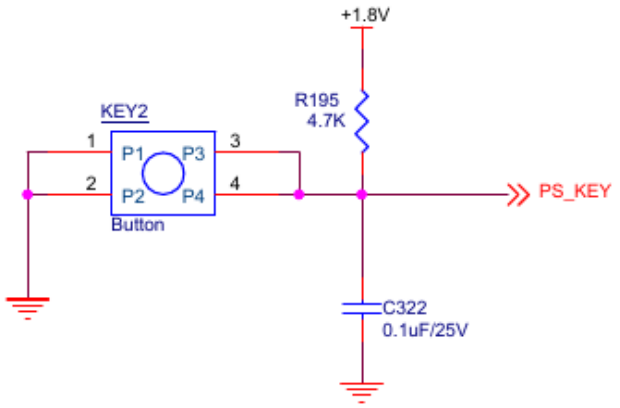
Четвертый, служебный, индикатор используется для того, чтобы обозначить, что загрузка bitstream в FPGA прошла успешно:



Плюсом к этому на плате расположено несколько кнопок. Первая - используется для включения питания платы, вторая - подключена к PL-части, и третья - подключена к PS-части:







В общем достаточно стандартный набор. Теперь перейдем к рассмотрению того, что поставляется с платой в части примеров и SDK для работы с платой.

BSP и примеры проектов идущие в комплекте

В комплекте с платой идет архив размером 7.2 Гбайт в котором идёт самое подробное описание отладки:

- механические размеры и dxf-файлы плат;
- схематики на каждый из компонент;
- большое количество примеров;

Если просто перечислить набор примеров, то получится примерно следующий список.

Примеры для работы с периферией из ПЛИС и baremetal-приложений в PS-части:

1. Подключение LED к PL-части с параллельным использованием логического анализатора ILA;
2. Использование PLL в PL-части для генерации различных частот;
3. PS RAM;
4. ROM;
5. FIFO;
6. Кнопки и GPIO;
7. PWM;
8. UART PL;
9. RS485 PL;
10. PL DDR4 RAM;
11. HDMI out (с использованием FMC-платы);
12. HDMI in (с использованием FMC-платы);
13. Примеры с семисегментным дисплеем;
14. Ethernet BER тест;
15. Пример использования GTH-интерфейса;
16. PS Hello World;
17. PS RTC;
18. PS MIO;
19. PS UART;
20. PS CAN;
21. PS I2C;
22. PS Display Port;
23. PS SD Card;
24. PS Network;
25. PS Network with remote applications;
26. PS Sysmon;
27. PS EMIO;
28. PS AXI GPIO;
29. PS RS485;
30. PL Ethernet;
31. Custom PWM IP;
32. Dualcore AMP;
33. FreeRTOS;
34. PL read write PS DDR;
35. BRAM test;
36. DMA Loopback;
37. AD7606 DMA;
38. AD9708 DMA;
39. AD9280 DMA with Display Port;

40. AD9238 DMA with Display Port;
41. AD7606 DMA with Display Port;
42. AD9767;
43. AN5642 + SD Card;
44. AN5642 + LwIP;
45. AN5641 + MIPI + Displya Port;
46. LCD + Touch;
47. AD9280 + LwIP;
48. AD9238 + LwIP;
49. AD7606 + LwIP;
50. LED + QSPI + SDCard;

В комплекте так же множество примеров работы с периферией из Linux:

1. Сборка petalinux и куча примеров для него;
2. Сборка драйверов для Linux;
3. Драйверы для Char-устройств;
4. Device Tree;
5. Pinctl GPIO подсистема в Linux;
6. Concurrent system calls;
7. Чтение GPIO;
8. Таймеры;
9. Прерывания;
10. Система ввода/вывода;
11. Неблокируемый ввод/вывод;
12. Асинхронный ввод/вывод;
13. Работа с платформой с Device Tree;
14. I2C;
15. USB;
16. SPI;
17. UART;
18. Работа с блочными устройствами;
19. Работа с сетевыми устройствами;
20. DMA Loop;
21. Работа с тачсенсорами;
22. Работа с LCD-дисплеями;
23. Работа с ADC AD9238 и AD7606 в Linux;
24. Работа с DAC AD9767 из Linux;

В теории по каждому из этих примером можно было бы написать целую книгу или цикл статей. Данный BSP я выложил в своем телеграм-канале по Zynq: https://t.me/zynq7000

Заключение

Долго думал, чем завершить данный обзор. Плата конечно стоит очень дорого и обычному энтузиасту недоступна, но объем возможностей и ресурсов которые она предоставляет в итоге - точно соответствует ее цене. Мне показалось интересным заглянуть за пределы доступного, и поковырять что-то недоступное и предоставить это общественности. Возможно чуть позже напишу статью по теме сборки Linux под Ultrascale+ потому что там своих граблей оказалось навалено невероятно много и все казалось не настолько тривиально как в Zynq-7000. И есть пара кейсов на рассмотрение, связанных с PL Ethernet и AXI ENET, но это уже предмет отдельного разговора.

Спасибо, что дочитали до конца. До встречи в следующих статьях!

Отладочная плата ПИР СЦХ-254 «Карно»

Графов М.В.
преподаватель НИУ «МЭИ»
ведущий Школы Синтеза Цифровых Схем в кластере «МЭИ»
E-mail: GrafovMV@mpei.ru

Обсуждение и комментарии: link

Аннотация

Статья посвящена краткому обзору аппаратных средств отладочной платы разработчика «Карно» [1] на базе FPGA серии ECP5 от Lattice Semiconductor, отечественного производителя ООО «Фабмикро» [2]. Рассмотрен состав платы, некоторые схемотехнические особенности, описан, обнаруженный в ходе работы над статьей, схемотехнический недостаток. В конце статьи приведены краткие выводы по возможностям платы, возможные области применения, выявленные достоинства и недостатки, а также полезные ссылки для дальнейшей самостоятельной работы специалистов и начинающих, заинтересованных в использовании данной платы.

Введение

Отладочные платы для инженеров, связанных с разработкой устройств на базе FPGA или микроконтроллеров, чаще всего применяются в двух случаях: для начального изучения, используемой на плате FPGA или микроконтроллера, либо для быстрой реализации конкретного проекта, при условии наличия на плате необходимых аппаратных средств. В любом из этих случаев, использование готовых, отладочных плат значительно экономит время и силы, позволяя сконцентрироваться на изучении, либо на разработке текущего проекта, нежели, чем на разработку и сборку самой отладочной платы!

От проприетарного ПО к open source toolchain и open source hardware

В случае выбора пути покупки готовой платы, встаёт вопрос предпочтения, той или иной плате из набора существующих на рынке плат, приобретения отладочной платы, и собственно, эксплуатация платы, чему, в свою очередь, может значительно помочь техническая поддержка производителем. Обычно, техническая поддержка заключается в предоставлении производителем примеров кода для периферийных устройств, установленных на плате, документов с описанием портов ввода - вывода GPIO, а также с принципиальной схемой устройства.

На сегодняшний день на рынке предлагается большое количество таких изделий различных производителей и для

большого количества микросхем программируемой логики, начиная от простых CPLD и заканчивая сложными FPGA большой ёмкости и SoC. Все они имеют очень разный набор периферийных средств, выпускаются разными производителями, но в ***подавляющем большинстве случаев*** их объединяет одно – необходимость наличия проприетарного программного обеспечения! Это значит, что формирующим bit-stream файл для целевой ПЛИС, должен быть софт от производителя самой ПЛИС! Да, бесспорным является тот факт, что существуют бесплатные версии IDE, поддерживающие ограниченный функционал и набор микросхем. Существуют сторонние редакторы HDL и различные вспомогательные программы, приложения, плагины и т.д., но... Как было сказано выше: «...конечным инструментом..» и т.д.! Хорошо это или плохо, вопрос не в этом, но это ***факт*** и его никак не оспорить! Ибо внутренняя структура микросхем ПЛИС известна только производителю этой микросхемы, а значит и софт должен быть только этого производителя! Однако, идеология open source не обошла и такую консервативную и закрытую тему, как ПЛИС и существует, как минимум один, производитель микросхем программируемой логики, с (условно) открытой архитектурой и также существует открытое ПО для таких микросхем. Речь идёт о производителе Lattice Semiconductor и открытом программном продукте (open source toolchain) Yosys. На этом месте необходимо сделать небольшой экскурс в историю, что бы понять смысл "условно" открытой архитектуры. Изделия фирмы Lattice, также, проприетарные и внутренняя их структура до недавнего времени была закрыта. Однако, в середине 2010-х небольшая группа исследователей из Германии смогла осуществить "реверс-инжениринг" некоторых из них - сначала модель iCE40, а потом и ECP5. Используя проприетарный тулчейн, они синтезировали большое количество определенных структур и изучали закономерности - как это выглядит в битстриме. Так появился первый *open-source* синтезатор Yosys и первый плейсей Arachne-pnr. Всю полученную информацию о структуре ПЛИС они сохраняли в базу данных специального формата - ChipDB. После того, как это дело было опубликовано на FOSDEM, большое количество энтузиастов подхватило её и занялось "реверсингом" ПЛИС других производителей. Сейчас в ChipDB содержится информация о структуре большого количества микросхем ПЛИС разных производителей, в том числе Altera, Xilinx и GOWIN. Вместо Arachne-pnr был написан другой, более универсальный и прогрессивный плейсей - NextPnR. Производители микросхем ПЛИС не стали мешать энтузиастам, а возможно, даже помогают, предоставляя закрытую документацию о структуре своих изделий. Это одна из причин, по которой в среде "опенсорс/опенхард" (OSHW) фирму Lattice любят. Учитывая не самые высокие показатели по месту на рынке, это может быть хорошим маркетинговым ходом для продвижения своей продукции.

В статье не будет обзора микросхем производителя, не будет разбора возможностей и характеристик микросхемы ПЛИС, на которой собрана отладочная плата, и тем более не будет ничего про Yosys. Это темы для других статей и обзоров гораздо большего объема, здесь же будет краткий обзор отладочной платы на базе FPGA от Lattice Semiconductor серии ECP5 и некоторые схемотехнические особенности платы. Желающим глубже погрузиться в эту тематику, можно порекомендовать к прочтению статьи разработчика платы Руслана Залата [3], [4]. Также, целью статьи не

является копирование информации о плате с сайта производителя, любой заинтересовавшийся читатель сможет самостоятельно пройти по ссылкам, приведённым ниже, и получить всю информацию от первоисточника, т.е. от разработчика и производителя платы.

Особенности платы ПИР СЦХ-254 «Карно»

Прежде чем начать содержательную часть статьи, необходимо отметить, что плата была предоставлена в кластер «МЭИ» школы синтеза, разработчиком, бесплатно, для ознакомления и дальнейшей работы заинтересованных студентов с Yosys, за что БОЛЬШОЕ СПАСИБО, а также за консультации, предоставленные при написании данной статьи! Плата поддерживается школой синтеза, т.е. на нее портированы *некоторые* лабораторные работы, поддерживается работа со скриптами школы и синтез в Yosys. Так что же такого особенного и замечательного в этой плате, что её отличает от большого количества других, таковых же, похожих плат?

1. Как уже было указано выше, целевая FPGA от Lattice Semiconductor, серии ECP5 поддерживается open source toolchain Yosys, что встречается крайне редко!

2. Отечественный производитель – ООО «Фабмикро». И это не просто контрактное производство плат, это разработчик схемы, ПО, сборка печатных плат на специализированных, промышленных автоматах, а не ручной монтаж. Т.е. это не домашняя разработка энтузиаста open source toolchain, а полноценное промышленное изделие, начиная от децимального номера на штампе конструкторской документации и заканчивая фирменной упаковкой и паспортом изделия с печатью организации и гарантийным сроком! Визуально, при осмотре платы, дефектов сборки не обнаружено. Единственное, что омрачает положительные эмоции от обладания данной платой, это то, что производитель лишил нас такой приятной процедуры, как отмывка флюса на плате… Вспомните, счастливые обладатели OMDAZZ – ов и RZRD и т.д. и тем более *no name of China*, много спирта извели на отмывку!?

Рис. 1 Упаковка комплекта

Рис. 2. Комплектность упаковки

Рис. 3. Упаковка самой платы

3. Полноценная техническая поддержка проекта на github [5]

4. Проект платы не только выполнен на ПЛИС с открытой архитектурой, поддерживается открытым ПО Yosys, но и разработка выполнена в открытом ПО для проектирования печатных плат – KiCad EDA. Все исходные файлы схем и плат доступны в репозитории поддержки. Любой желающий, при наличии соответствующей квалификации, может самостоятельно повторить этот проект.

5. Форм-фактор платы – Raspberry Pi 3. Не большая плата, «размером с кредитку».

6. Широкий диапазон питающих напряжений 5-24 В, что позволяет запитывать плату от постоянного напряжения 24 В, используемого в промышленной автоматике.

7. Возможность покупки физ. лицами за наличный расчёт через торговую площадку «OZON», а также безналичным путём, оплатой счёта юридического лица, что важно для возможности закупки коммерческим и бюджетным организациям!

Состав аппаратных средств платы

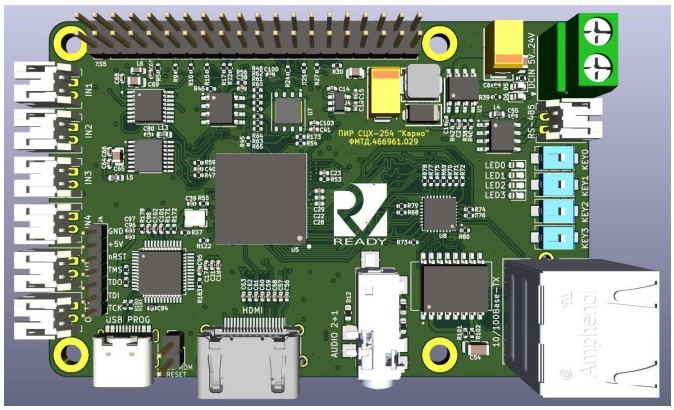


Рис. 4 Слой TOP (верхняя сторона установки элементов)

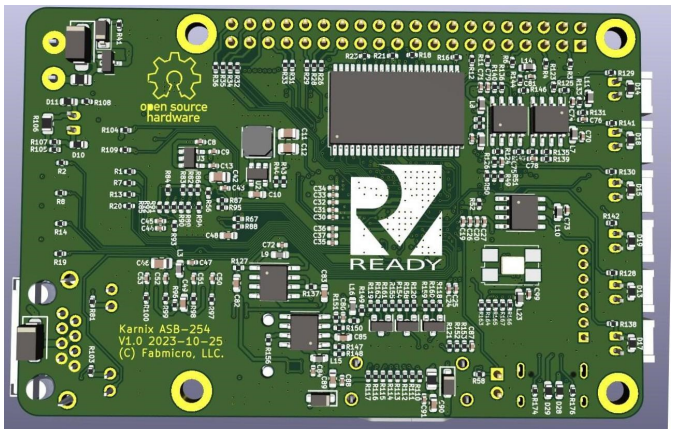


Рис. 5 Слой BOTTOM (нижняя сторона установки элементов)

На Рис. 4 и 5 приведен внешний вид платы начальной версии V1.0. На сегодняшний день в репозитории есть уже V1.1 и V1.2 и еще разработчик планирует новый релиз платы с устранением обнаруженных недостатков. Как видно по Рис. 1, на обзоре релиз платы V1.1

FPGA Lattice ECP5

Основа платы, целевая ПЛИС - ECP5 семейства, LFE5U-25 в корпусе BGA-256. Имеет "на борту" 24K LUTs, различные виды памяти - системная блочная, 56 блоков по 18 кБит, встроенная 1,008 кБит, распределенная 194 кБит. Имеются 28 умножителей 18x18, два канала PLL/DLL, поддержка большого числа стандартов ввода-вывода, DDR2/DDR3, синхронизация с высокоскоростными АЦП,

ЦАП. Сравнивать по производительности с другими типами ПЛИС других производителей нет смысла, так как это зависит от конкретного описания и теста.. Если попытаться грубо оценить относительно других семейств ПЛИС, например INTEL/ ALTERA, то можно сказать что это уровень младших чипов семейства CycloneIII - CycloneIV, т.е. EP3C5-EP3C25, или EP4CE6-EP4CE22. Хотя, технология изготовления ECP5 - 40 нм, что меньше чем у "циклонов", а значит быстрее и меньше потребляет мощности. Это FPGA младшей/ средней ценовой категории и производительности. Что касается тестов, то для оценки возможностей разработчиком был реализован проект SoC (СнК), на базе VexRiscV, работающим на частоте 65 МГц, с небольшим I$ и D$, и набором различной периферии (CGA через HDMI, Ethernet...) [6].

Программатор

Программирование целевой ПЛИС осуществляется JTAG интерфейсом через USB порт. Это реализовано при помощи специализированной микросхемы, осуществляющей функции моста USB-COM, FT2232D от фирмы FTDI. В ней содержится два аппаратных канала-моста USB-COM, один из которых выполняет программную эмуляцию JTAG интерфейса. Второй же канал последовательного порта доступен как пользовательский COM порт и может быть использован для обмена данными с ПК. Таким образом, одним кабелем, подключенным к USB порту ПК, осуществляется питание платы, загрузка битстрим файла и обмен данными по последовательному порту! Для работы с платой внешний программатор НЕ НУЖЕН!!! Однако, есть возможность подключения внешнего программатора, для чего на плате установлен разъём J4.

Микросхемы памяти

На плате присутствуют энергонезависимые устройства и статическая память. К энергонезависимым относятся: EEPROM AT24C01 объёмом 1 КБит (128 Байт), для хранения пользовательских данных, интерфейс – IIC (I^2C), а также NOR flash W25Q128 объёмом 128 Мбит для хранения конфигурации FPGA, интерфейс – SPI. Наибольший интерес тут представляет SRAM – K6R4016. Да, это fully static RAM, организацией 256Kx16 и временем доступа 10 нс! А это означает, что не нужен драйвер динамической памяти, это означает малое время доступа к памяти, отсутствие latency, а как следствие применение как память данных в быстрых вычислителях, как СОЗУ и ОЗУ в однотактных процессорах, как L0, L1 Cashe и т.д., в зависимости от решаемой задачи.

Интерфейсы

Несмотря на малые размеры платы, в ней есть несколько различных по сложности и типу взаимодействия с пользователем интерфейсов. Начать можно с самого простого - на плате присутствуют 5 тактовых кнопок без фиксации KEY[3:0], FPGA_RESET и 4 дискретных светодиода LED[3:0]. Все кнопки нормально разомкнутые, имеют "Pull-Down" подтяжку. Это значит, что если кнопка не нажата, то на входах KEY[3:0] и FPGA_RESET присутствует уровень лог. 0, при нажатии на кнопку формируется уровень лог. 1, так как кнопка подключена к питанию +3,3В. Светодиоды LED[3:0] включены по схеме с общим катодом, что означает то, что для засветки светодиода надо выставить уровень лог. 1 на выходы LED[3:0]. На плате нет семисегментных светодиодных индикаторов, так горячо любимых всеми производителями отладочных плат. Возможно, этот факт будет негативно воспринят начинающими, так как применение этих индикаторов, весьма наглядно на начальных этапах обучения

цифровому синтезу. Однако, этот недостаток легко устраним при помощи внешнего модуля на микросхеме TM1638, в различных ее применениях, с разными числом дополнительных кнопок и светодиодов. Модуль недорогой и легко приобретается. Будет хорошим учебным проектом, написание модуля на языке System Verilog для подключения к плате! Более того, в примерах школы синтеза цифровых микросхем это уже реализовано! Поэтому, малое количество кнопок и светодиодов и отсутствие семисегментых индикаторов, не являются негативным фактором при использовании данной платы! Это же утверждение справедливо и для других подобных модулей, например модули от Sipeed, для уже широко известных FPGA GOWIN. Модуль можно подключить к плате через еще один интерфейс платы - 40 пиновый разъем портов ввода-вывода общего назначения. Это пользовательские выводы FPGA, программируемые пользователем по своему усмотрению. На разъёме присутствуют 28 портов ввода-вывода GPIO[27:0], сигналы питания +5В, +3,3В и "земля". Например, для подключения модуля индикации и кнопок на TM1638, достаточно двух сигнальных выводов и питание. Модуль подключается по последовательному интерфейсу SPI. Аналогично, возможно подключение любых других внешних цифровых устройств, при наличии свободных портов GPIO, достаточной скорости обмена по этим портам и соответствии стандартов ввода-вывода устройств и выводов FPGA.

Как было уже сказано выше, модуль имеет возможность подключения к последовательному порту ПК, через имеющийся на плате двухканальный мост USB-COM. Реализация COM интерфейса программная, в данном случае это задача пользователя. Номер COM порта в ПК может быть разный, зависит от производителя "материнской платы" ПК, версии BIOS и драйверов на FT2232D. Обычно это COM3, COM4, но могут быть и другие из списка "виртуальных" портов (т.е. не COM1 - COM2). Последовательный порт очень широко применяется при передаче небольших объемов данных на малых скоростях (хотя современные чип-сеты "материнсикх плат" поддерживают скорости обмена до единиц МБит!) с использованием стандартных терминальных программ, что позволяет использовать этот механизм без написания своих сложных приложений!

На плате есть еще один последовательный интерфейс! Но, он не предназначен для подключения к COM портам ПК, так как имеет физический уровень RS485 стандарта! Это протокол последовательного обмена для промышленных применений, так как данные передаются дифференциальной парой сигналов, что значительно повышает помехоустойчивость и позволяет иметь длину линии связи в сотни метров! Таким образом, наличие RS485 на плате, позволяет использовать её как промышленный контроллер! Возможно и подключение к ПК через этот канал, но для этого понадобится переходник RS485-RS232, который также легко приобретается и доступен!

Для реализации видеоприложений и подключения к мониторам на плате предусмотрен HDMI разъём! Разъём подключен напрямую к выводам FPGA через разделительные конденсаторы. Используются дифференциальные пары при передаче сигналов, что необходимо будет указать в проекте в свойствах портов ввода - вывода. Существуют и *open-sourse* решения для HDMI, как например вот это [7]. Также, в группе по портированию примеров на платы школы синтеза, есть такое же рабочее решение

HDMI для этой платы и возможно, оно появится в лабораторных работах. Как было указано выше, разработанная СнК включает в себя драйвер CGA дисплея на HDMI.

И в заключение, последний интерфейс платы - Ethernet 10/100Base-TX. Он реализован установкой на плату трансивера физического уровня LAN8710A от Microcip. Также присутствует разделительный трансформатор и разъём RJ45 для подключения стандартными патч-кордами. Реализация протоколов, будь то TCP/IP или ModBus over IP - пользовательская программная.

Аналоговая часть

Весьма редкое явление - наличие полноценной аналоговой части, в виде предусилителей и отдельных дискретных АЦП - ЦАП. На данной плате присутствуют 8 канальные АЦП и ЦАП. АЦП - ADC128S052. Это 12 битный, 8 канальный, SAR ADC от Texas Instruments. Так как это SAR ADC, то больших скоростей преобразования ожидать не стоит, и для данной микросхемы это 200 - 500 kSPS, что вполне достаточно для аудиоприложений или оцифровки датчиков промышленной автоматики не высокой точности, так как разрешение 12 бит не подходит под определение "метрологического" АЦП, хотя вопрос тут уже в конкретной задаче и точности её решения. ЦАП - AD5328. Это 12 битный, 8 канальный, STRING-DAC (с цепочкой резистивных делителей). На плате присутствуют дискретные разъемы для 4 аналоговых входов и двух аналоговых выходов и комбинированный аудиоразъем типа jack 3,5 мм, на который выведены еще два аналоговых выхода и вход микрофона. Таким образом, на плате имеется 4 аналоговых входа, вход микрофона и 4 аналоговых выхода, подключенные к разного типа разъемам. Что касается входного динамического диапазона и коэффициента преобразования, то тут есть особенность.. Так как входные ОУ LM258 питаются однополярным питанием, то в районе низких входных напряжений может быть завышенная нелинейность коэффициента усиления. В описании на LM258 явно не указано, что тип входа и выхода поддерживается "RAIL to RAIL" хотя и указано что "*..Differential input voltage range equal to the power supply voltage* ". В процессе обсуждения статьи, разработчик выполнил симуляцию входного каскада, результаты которой приведены в репозитории [8]. Для входов IN1 - IN4 линейный участок находится в диапазоне 0,07 - 6,5 В, LSB - 1,6 мВ. Входной усилитель - инвертирующий! Это все необходимо учитывать, в случае применения АЦП для оцифровки каких либо датчиков. Микрофонный вход имеет коэффициент усиления порядка 840, что может быть не очень приемлемо для каких либо микрофонов, что разработчик подтверждает и в следующем релизе платы, возможно будет доработка для увеличения коэффициента передачи до 2000. На входе микрофона присутствует постоянное смещение равное половине питания, т.е. 1,65 В. Следует заметить, что наличие микрофонного аналогового входа на цифровой плате с FPGA или микронтроллером встречается весьма редко, так как сейчас возможно применение сразу цифровых микрофонов, с интерфейсом I^2S, что в школе синтеза сделано и работает на всех платах. Так что, если кому то понадобится качественный микрофонный вход, то лучше воспользоваться этим решением.

В процессе написания статьи и анализа принципиальной схемы включения ЦАП был обнаружен один недостаток - не подключенный вход ~LDAC ЦАП, т.е. первый вывод U11. Ошибка не является критичной, так как ЦАП имеет режим автозащёлкивания

данных по последнему тактовому сигналу и тесты в репозитории все рабочие и ЦАП там работает именно в этом режиме. Подключение ~LDAC необходимо, в случае если требуется одновременное, синхронное обновление данных на выходе ЦАП по какому либо внешнему сигналу, подаваемому на вход ~LDAC. Это необходимо при формировании каких либо синхронных многоканальных последовательностей или когерентных аналоговых сигналов, например при формировании квадратурных составляющих сигналов. Очевидно, что это очень узкоспециализированная задача, но если кому это понадобиться делать на версиях платы до V1.2, то придется самостоятельно соединить 1 вывод U11 с каким либо GPIO на выходном разъёме. Разработчик об этом недостатке знает и в следующем релизе платы этот недостаток будет устранён!

Заключение

Рассмотренная плата, в силу своих особенностей, может быть интересна специалистам разных уровней квалификации и применяться для разных задач. Плата может представлять интерес начинающим, делающим первые шаги в цифровом синтезе, так как поддерживается школой синтеза, примеры школы портированы на эту плату. Недостаток в виде малого числа кнопок и светодиодов и отсутствия семисегментных индикаторов легко устранятся приобретением модуля на TM1638. Плата просто незаменима для энтузиастов занимающихся изучением и продвижением open-source. Наличие промышленных интерфейсов и аналоговая часть, после её доработки, могут позволить использовать плату как промышленный контроллер, например для ввода стандартных аналоговых сигналов (токов 0-5, 0-20, 4-20 мА), а наличие портированого RISC-V ядра позволит делать обработку сигналов и подключать плату к другому хосту по Ethernet. Возможностей и приложений много, все зависит от конкретной задачи. Возможно, мы увидим разные релизы платы под применение в идеологии *configurable Input Output (cRio om National Instruments),* или это будет промышленный контроллер для оцифровки стандартных датчиков и сигналов, а может с развитием движения OSHW плата будет востребована в образовательных целях и учреждениях. Единственно, возможно, существенным недостатком платы, может быть её цена. На момент написания статьи, относительная стоимость платы была равна 2,5 - 3 OMDAZZ или одна DE10-Lite. Это может быть критично для начинающих, ибо они предпочтут более "раскрученную" ALTER-у и QUARTUS, чем малоизвестный Lattice и Yosys, тем более, что для начинающих дешёвый OMDAZZ выглядит предпочтительнее! Но по набору аппаратных средств, установленных на плате, она может быть весьма интересна и востребована! К тому же, следует учитывать реалии отечественного производства, а точнее необходимость закупки импортных комплектующих. В любом случае, появление таковой платы, это весьма знаковое и положительное событие!

Ссылки

1. Отладочная плата ПИР СЦХ-254 «Карно» https://www.fabmicro.ru/products/125.html
2. ООО «Фабмикро» https://www.fabmicro.ru/
3. Статья Руслана Залата, часть 1 https://habr.com/ru/articles/801191/
4. Статья Руслана Залата, часть 2 https://habr.com/ru/articles/802127/
5. https://github.com/Fabmicro-LLC/Karnix_ASB-254

6. https://github.com/Fabmicro-LLC/VexRiscvWithKarnix
7. https://github.com/hdl-util/hdmi
8. https://github.com/Fabmicro-LLC/Karnix_ASB-254/tree/master/Karnix_ASB-254-v1.2/simulation

DESim: как изучать проектирование на ПЛИС без отладочной платы

Романова И.И., Зунин В.В., Маршутина Е.Н., Американов А.А., Романов А.Ю.

Национальный исследовательский университет «Высшая школа экономики»

Обсуждение и комментарии: link

Введение

Самостоятельное обучение является сложным, но эффективным способом повышения собственной квалификации и получения дополнительного образования. Но в случае электроники и изучения ПЛИС – серьезным барьером является недостаток хороших обучающих материалов и необходимость покупки дорогостоящего оборудования.

Первый барьер постепенно устраняется за счет появления новых российских и переводных учебников по цифровому синтезу и проектированию на ПЛИС. Особенно хорошей для этого является переводная книга Харрис Д., Харрис С. «Цифровая схемотехника и архитектура компьютера: RISC-V» (https://dmkpress.com/catalog/electronics/circuit_design/978-5-97060-961-3/) [1], а на днях для нее вышел практикум: Романов А.Ю. и др. «Цифровой синтез: RISC-V» (https://dmkpress.com/catalog/electronics/circuit_design/978-5-93700-282-2/) [2], где на каждую тему приведено множество примеров кода, размещенных на Github.

Второй барьер состоит в том, что сейчас популярная плата для изучения ПЛИС DE1-SoC [3] у производителя стоит $377. Более простые китайские аналоги тоже не дешевы, но к тому же еще и лишены хорошей документации и не дают таких возможностей. Но решение есть: использование виртуальных лабораторий, таких как DESim, о которой будет рассказано в этой заметке. Эта заметка основана на тексте Главы 2 книги «Цифровой синтез: RISC-V» [2], поэтому больше подробностей и примеры использования DESim [4] для масштабных проектов, вроде аппаратной реализации систолического массива, можно найти там. Эта же заметка – краткая инструкция, как установить и настроить среду для того, чтобы попробовать ее в деле.

1. Виртуальная плата DE1-SoC

Приложение DESim предоставляет графический интерфейс для доступа к части функций платы DE1-SoC. Фактически приложение является графическим интерфейсом для симулятора ModelSim, в котором происходит моделирование работы

с платой. Благодаря графическому интерфейсу появляется возможность тестировать разработанные проекты по аналогии с тестированием на реальной отладочной плате DE1-SoC.

1.1. Установка DESim

Для установки приложения DESim необходимо скачать установщик из GitHub-репозитария проекта (https://github.com/fpgacademy/DESim) [4]. Данный репозитарий включает в себя исходный код приложения, с которым можно ознакомиться для более полного понимания принципов его работы. Для загрузки установочного файла DESim необходимо перейти на страницу «Releases» в репозитарии. На момент подготовки статьи, последней версией приложения является v2.1. После перехода на страницу необходимо скачать установочный файл для Windows (desim_setup.exe) или архив с приложением для Linux (desim.tar.gz).

На следующем шаге необходимо запустить скачанный установочный файл. После запуска отобразится окно с приветствием (Рис. 1). Необходимо нажать кнопку «Next» и перейти к следующему окну (Рис. 2), а затем принять лицензионное соглашение (кнопка «I Agree»).

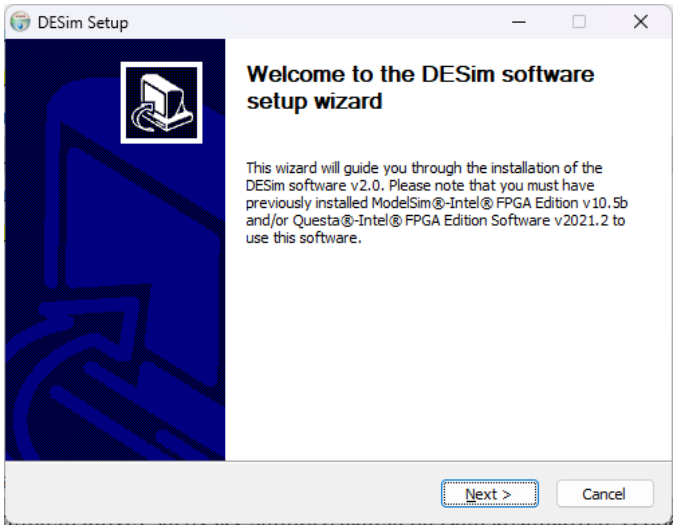


Рис. 1. Окно установки DESim после запуска

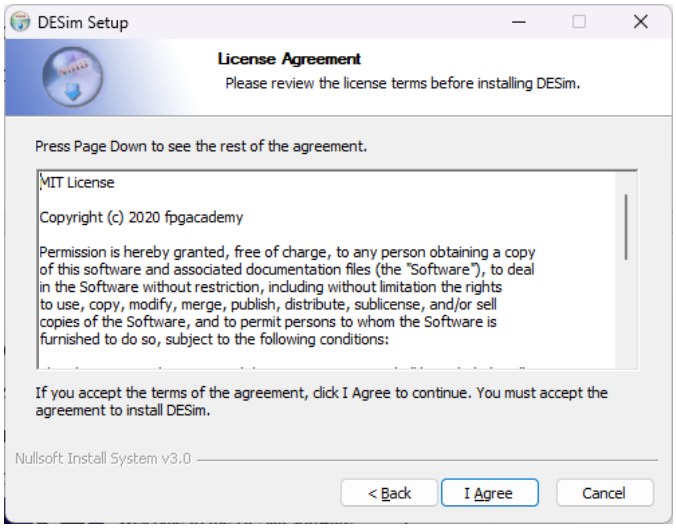


Рис. 2. Окно лицензионного соглашения

После принятия лицензионного соглашения появится окно выбора папки для установки приложения (Рис. 3). Предполагается, что установка приложения будет произведена в папку по умолчанию, но можно выбрать и другой путь установки. Остается нажать кнопку «Install» и дождаться завершения установки. В процессе установки есть возможность разместить иконку приложения DESim на Рабочем столе, что обеспечит быстрый запуск приложения.

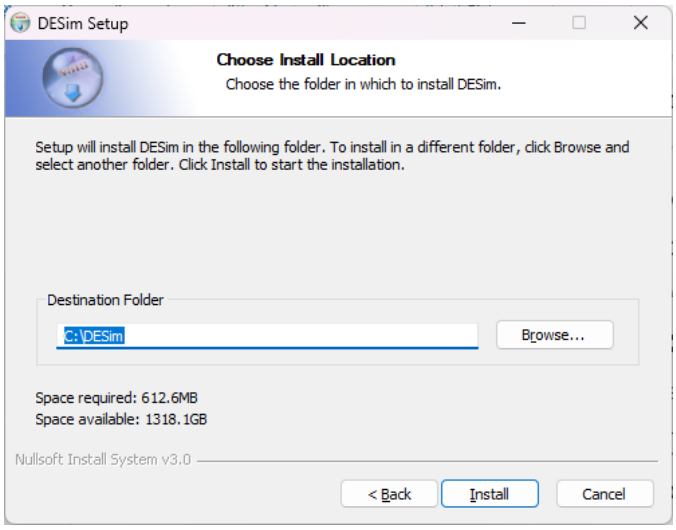


Рис. 3. Выбор директории установки

Для версии на Linux установка не требуется. Достаточно скачать архив и распаковать его.

Созданная при установке или распаковке папка содержит три папки (demos, docs и java) и два файла (скрипт для запуска и

scancode.csv). Скрипт для запуска DESim_run.bat для Windows или DESim.sh для Linux необходимы для корректного запуска приложения. Папка demos содержит набор базовых примеров, с помощью которых можно ознакомиться с функционалом приложения. В папке docs можно найти с англоязычными инструкциями по работе с проектами, реализованными на разных языках описания аппаратуры (VHDL, Verilog, SystemVerilog).

1.2. Возможности приложения DESim

Приложение DESim [4] позволяет моделировать взаимодействие с семью основными элементами периферии платы ПЛИС: светодиодами, переключателями, кнопками, семисегментными индикаторами, клавиатурой PS/2, параллельными портами GPIO и портом VGA.

Светодиоды представлены набором прямоугольников, каждый из которых отвечает за один светодиод. Белый цвет означает, что светодиод выключен, а красный – включен.

Рис. 4. Блок светодиодов DESim

Переключатели изображены в виде квадратных полей с возможностью их выбора. Галочка в квадрате означает, что сигнал переключателя равен логической единице, то есть включен. Если же он пуст, то переключатель выключен.

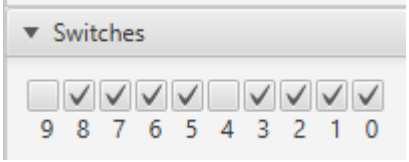


Рис. 5. Блок переключателей DESim

Кнопки изображены аналогично переключателям, но т.к. кнопки на плате DE1-SoC инвертированы, то пустой квадрат означает нажатую кнопку.

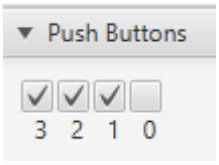


Рис.6. Блок кнопок DESim

Семисегментные индикаторы похожи на их физический аналог. Для включения сегмента индикатора необходимо подать на соответствующий вход логический ноль, а для выключения – логическую единицу.

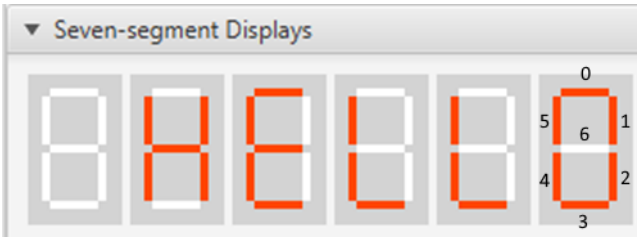


Рис. 7. Блок семисегментных индикаторов DESim

Блок клавиатуры PS/2 имеет текстовое поле, содержимое которого в формате байт добавляется в FIFO. ПЛИС может посылать команды на клавиатуру, и клавиатура будет возвращать соответствующий ответ. Данные команды можно использовать, например, для включения Scroll Lock, Num Lock и Caps Lock.

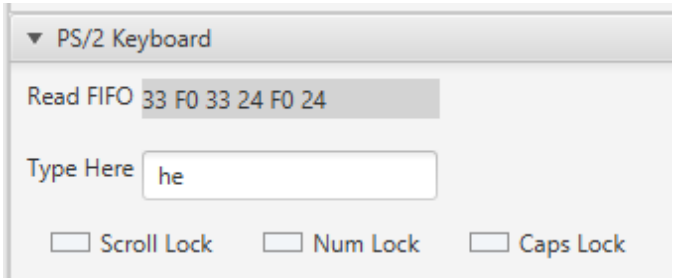


Рис. 8. Блок PS/2 клавиатуры DESim

Блок портов GPIO разделен на две части. Верхняя часть блока отображает текущее состояние портов GPIO и позволяет его изменять. Нижняя часть (*Direction Map*) нужна, чтобы задавать направление портов. Если поставить галочку в соответствующем поле, то данный порт будет восприниматься только

как выходной и на него нельзя будет подать какой-либо сигнал в верхней части блока. Также соответствующий бит порта GPIO вверху будет серым (неактивным), и в нем будет отображаться галочка, если будет подана логическая единица. Конфигурацию портов GPIO можно сохранять и загружать при необходимости.

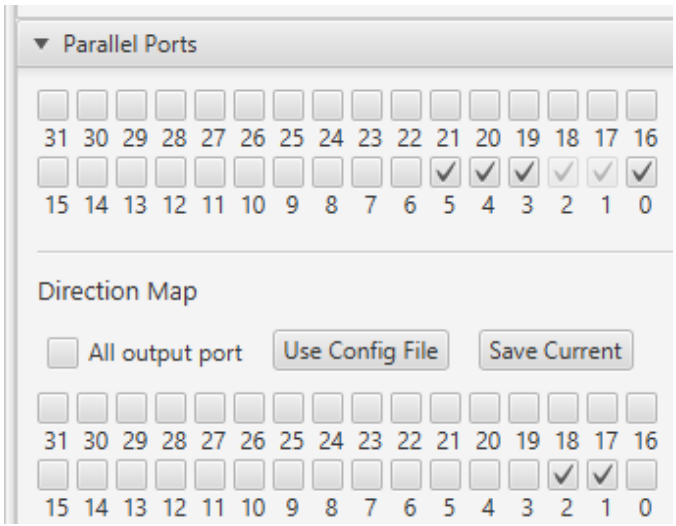


Рис. 9. Блок портов GPIO приложения DESim

Блок VGA Display представляет собой дисплей с заданным разрешением, на который можно выводить изображение с помощью сигналов с ПЛИС.

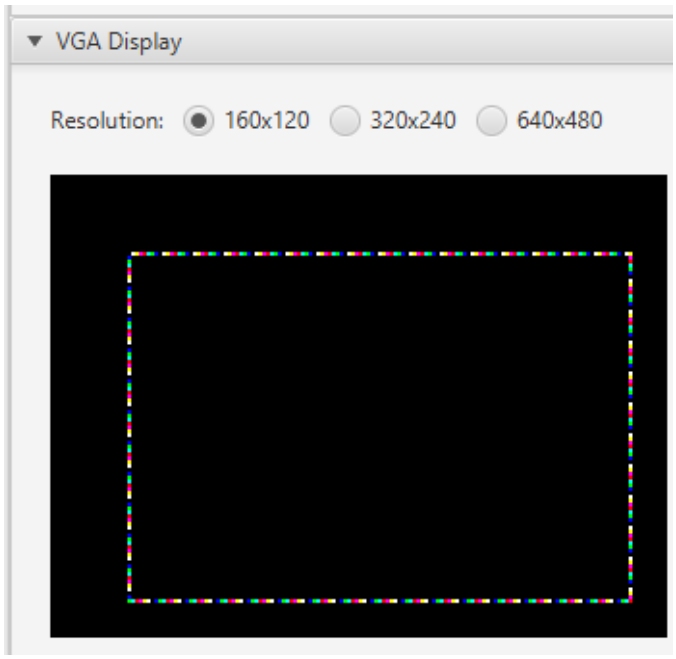


Рис. 10. Блок VGA приложения DESim

1.3. Основной интерфейс приложения DESim

Сразу после запуска приложения **DESim** должно появиться рабочее окно приложения (Рис. 11). В верхней левой части окна должно быть отображено сообщение «**The server is running...**», означающее корректный запуск приложения.

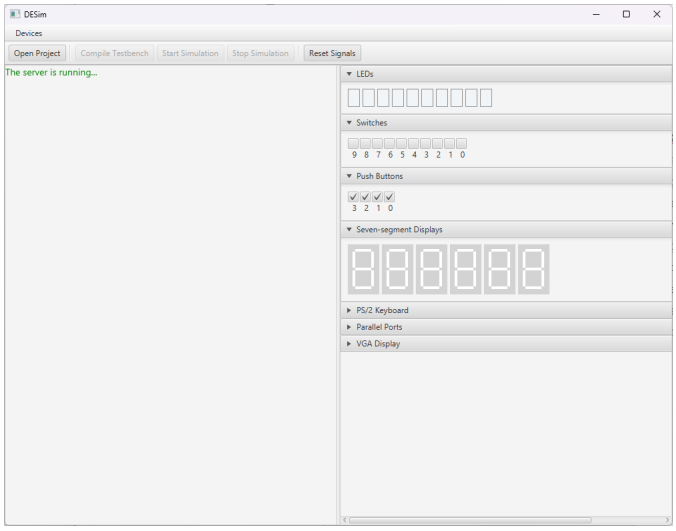


Рис.11. Рабочее окно DESim

Чтобы убедиться, что приложение DESim может взаимодействовать с симулятором ModelSim или QuestaSim, можно запустить один (или более) демонстрационных проектов (папка *demos*), которые идут вместе с DESim. Для этого нужно выбрать команду «*Open Project*» в графическом интерфейсе DESim, а затем перейти в папку demos. Далее нужно выбрать установленный симулятор, язык описания аппаратуры и один из представленных проектов (например, *counter*), а потом нажать на кнопку «Select Folder». После этого должно появиться сообщение «Project 'имя_проекта' opened.». После успешного открытия проекта его необходимо скомпилировать нажатием на кнопку «Compile Testbench». Если компиляция прошла успешно, то отобразится сообщение «Compilation successful» (Рис. 12).

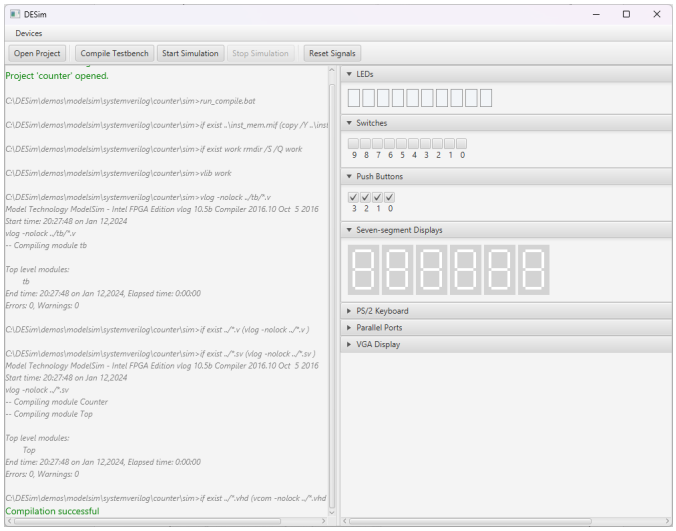


Рис. 12. Успешная компиляция проекта

После успешной компиляции проекта можно запустить симуляцию. В папке с выбранным проектом находится файл Read-me.txt, в котором описан сценарий работы с проектом. В данном примере (counter) используется только кнопка *KEY[0]*, отвечающая за сброс счетчика. Если кнопка нажата, то счетчик будет сброшен, иначе счетчик увеличивает значение переменной, отображающейся на светодиодах (Рис. 13).

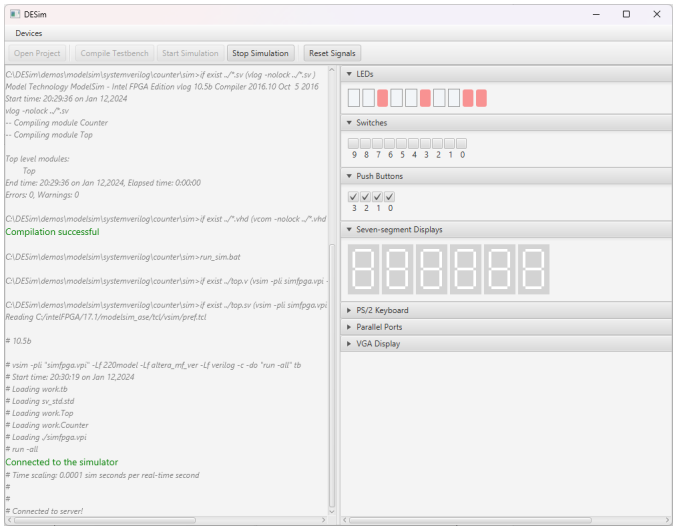


Рис. 13. Успешная симуляция демонстрационного проекта счетчика

```
module Counter (CLOCK, RESETn, LEDR);
    input  logic          CLOCK;
    input  logic          RESETn;
    output logic [ 9: 0] LEDR;

    parameter n = 24;

    logic         [n-1:0] count;

    // the counter
    always_ff @(posedge CLOCK)
        if (RESETn  == 1'b0)
            count <= 0;
        else
            count <= count + 1;

    assign LEDR = count[n-1:n-10];
endmodule
```

Листинг 1. Исходный код Counter.sv

Для того чтобы выполнить данный код, необходимо использовать модуль верхнего уровня (Листинг 1), в котором создается экземпляр тестируемого модуля *Counter* с соответствующим подключением необходимых сигналов.

```
`default_nettype none

module Top (CLOCK_50, KEY, LEDR);
    input  logic          CLOCK_50;
    input  logic [ 3: 0] KEY;
    output logic [ 9: 0] LEDR;

    Counter U1 (CLOCK_50, KEY[0], LEDR);

endmodule
```

Листинг 2. Исходный код файла верхнего уровня иерархии Top.sv

2. Использование шаблона проекта

Приложение DESim после установки содержит папку template, в которой содержится набор файлов, которые нужны, чтобы использовать полный функционал приложения. Эта папка может использоваться как шаблон для дальнейшей переделки под любой проект.

Рассмотрим входные и выходные сигналы модуля Top:

1. Входной сигнал CLOCK_50 является основным тактовым сигналом с частотой 50 МГц.
2. Входные сигналы KEY и SW являются входами, подключенными к кнопкам и переключателям.
3. Сигнал GPIO обозначен как *inout*, что позволяет использовать его в обоих направлениях без переопределения направления сигнала. Для того чтобы GPIO мог принимать сигнал, на него необходимо подать сигнал z.
4. Выходные сигналы HEX0...HEX5 соединены с семисегментными индикаторами.
5. Сигналы PS2_CLK и PS2_DAT отвечают за взаимодействие с PS/2 клавиатурой.
6. Сигналы VGA_X, VGA_Y, VGA_COLOR и plot отвечают за вывод изображения на VGA дисплее. VGA_X и VGA_Y обозначают координату пикселя, VGA_COLOR – цвет пикселя, а сигнал plot обозначает, нужно ли закрасить выбранный пиксель данным цветом.

Для использования данного шаблона под новый проект достаточно скопировать папку template в соответствующую директорию. После этого необходимо скопировать в нее все исходные файлы проекта и добавить вызов этого модуля в файле Top.

При появлении ошибки «Net type of 'port' must be explicitly declared.» при компиляции необходимо заменить logic на wire у соответствующего порта (port) при их объявлении.

Выводы

Таким образом, приложение DESim позволяет проводить тестирование разработанных модулей без платы, и в учебных целях может быть хорошей заменой учебной плате De1-SoC. Приведенное руководство является частью книги:

```
// Copyright (c) 2022 FPGAcademy
// Please see license at https://github.com/
fpgacademy/DESim

// Protect against undefined nets
`default_nettype none

module Top (
    CLOCK_50, KEY, SW, GPIO,
    HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, LEDR,
    PS2_CLK, PS2_DAT,
    VGA_X, VGA_Y, VGA_COLOR, plot);
    // DE-series 50 MHz clock signal
    input  logic         CLOCK_50;
    // DE-series pushbuttons
    input  logic [ 3: 0] KEY;
    // DE-series switches
    input  logic [ 9: 0] SW;
    // DE-series 40-pin header
    inout  logic [31: 0] GPIO;
    // DE-series HEX displays
    output logic [ 6: 0] HEX0;
    output logic [ 6: 0] HEX1;
    output logic [ 6: 0] HEX2;
    output logic [ 6: 0] HEX3;
    output logic [ 6: 0] HEX4;
    output logic [ 6: 0] HEX5;
    // DE-series LEDs
    output logic [ 9: 0] LEDR;
    // PS/2 Clock
    inout  logic         PS2_CLK;
    // PS/2 Data
    inout  logic         PS2_DAT;
    // "VGA" column
    output logic [ 7: 0] VGA_X;
    // "VGA" row
    output logic [ 6: 0] VGA_Y;
    // "VGA pixel" colour (0-7)
    output logic [ 2: 0] VGA_COLOR;
    // "Pixel" is drawn when this is pulsed
    output logic         plot;

    assign GPIO      = 32'hZZZZZZZZ;

    assign HEX0      = 7'h40;
    assign HEX1      = 7'h47;
    assign HEX2      = 7'h47;
    assign HEX3      = 7'h06;
    assign HEX4      = 7'h09;
    assign HEX5      = 7'h7F;

    assign LEDR      = 10'h155;

    assign PS2_CLK   = 1'bZ;
    assign PS2_DAT   = 1'bZ;

    assign VGA_X     = {4'h0, SW[3:0]};
    assign VGA_Y     = {3'h0, SW[7:4]};
    assign VGA_COLOR = KEY[3:1];
    assign plot      = KEY[0];

endmodule
```

Листинг 3. Код шаблонного модуля Top

«Цифровой синтез: RISC-V» [2]. Большая часть практикума из книги полностью выполнима в DESim.

Литература

[1] С. Харрис, Д. Харрис Цифровая схемотехника и архитектура компьютера: RISC-V / пер. с англ. В. С. Яценкова, А. Ю. Романова; под ред. А. Ю. Романова. – М.: ДМК Пресс, 2021. – 810 с.: ил.

[2] Цифровой синтез: RISC-V / под общ. ред. А. Ю. Романова. – М.: ДМК Пресс, 2024. – 636 с.: ил. – (Книжная полка Истового инженера).

[3] DE1-SoC, https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836

[4] DESim, https://github.com/fpgacademy/DESim/releases

Давайте создадим процессор! Step by Step

Хлуденьков А.Н.

ООО «Лассард»
model3d@mail.ru

Обсуждение и комментарии: link

Часть 1. АЛУ.

Схемотехника сложна для начинающих. Собрав пару устройств, большинство людей уходят в «ардуинщики». Ничего плохого в этом нет, но, на мой взгляд, это путь в никуда. Для того, чтобы расти, необходимо ставить более глобальные цели, например, создать свой ПРОЦЕССОР.

Мы будем создавать процессор вместе с обвязкой, периферией – блоками памяти, устройствами ввода-вывода. Мы будем делать то, что сейчас называется «система на кристалле» (SoC, System on Chip).

Процессор умеет вычислять. Для этого у него есть АЛУ – арифметико-логическое устройство, которое выполняет арифметические и логические операции. Данные операции процессор выполняет по программе, ход его работы зависит так же от данных. Процессор обрабатывает информацию и управляется информацией.

Информация представлена в двоичном виде – «0» и «1» (двоичная логика). С двоичной логикой работает булева алгебра. Будем считать, что у нас имеется одна логическая операция для одного операнда – инверсия (NOT), и три для двух операндов – дизъюнкция (OR), конъюнкция (AND), и «исключающее ИЛИ» (XOR). Построим для данных операций таблицы истинности.

Инверсия (NOT).

X	NOT (X)
0	1
1	0

Дизъюнкция (OR), конъюнкция (AND) и «исключающее или» (XOR).

X	Y	X «OR» Y	X «AND» Y	X «XOR» Y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Соберем схемы для данных операций. Паять транзисторы или низкоуровневые элементы плохая идея, поэтому данные операции мы будем моделировать. Именно моделировать, описывать. Про программирование на данном этапе речи нет. Для моделирования используют

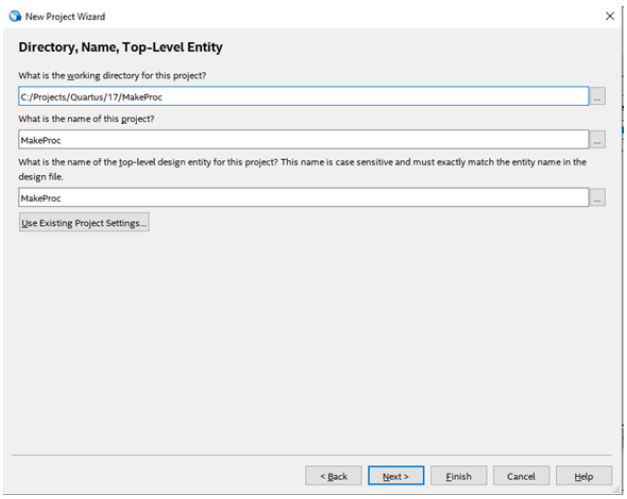
специальные программы автоматического моделирования – CAD (по-русски звучит «КАД»). Программ моделирования существует огромное количество. Так как наш журнал посвящен FPGA-ASIC, то и моделировать будет в соответствующих программах, к примеру в программе Quartus от Altera-Intel. Что такое FPGA и ASIC?

FPGA (по-русски звучит «Эф Пи Джи Эй»). Представьте, что у вас есть макетная плата с миллионами низкоуровневых блоков («OR», «AND», «XOR»), расположенных в прямоугольных ячейках. Вы можете ДО включения питания их расположить практически произвольным образом, переподключив провода между ними. ПОСЛЕ включения питания вы наслаждаетесь работой схемы или ищите ошибки в своем творении. Если что-то не устроило, можно выключить, ещё раз пересобрать и снова включить. И так практически до бесконечности.

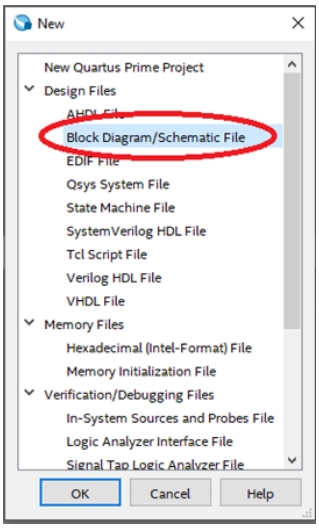
ASIC (по-русски звучит «асик»). Представим ту же плату, но только с ещё более низкоуровневой логикой (транзисторы, конденсаторы). Эту плату спаяли и залили компаундом. Скорость её работы намного выше, чем у FPGA, но только корректировать наше творение после «пайки» уже невозможно.

Моделировать будем в программе Quartus (по-русски «Квартус»). Скачиваем с любого дружественного файлообменника. Лучше скачать несколько версий, к примеру 13-ю, 17-ю и 21-ю. Мы планируем работать с настоящим железом, а более новые версии софта старое железо не поддерживают. Отдать приличную сумму за современную FPGA не каждому по силам.

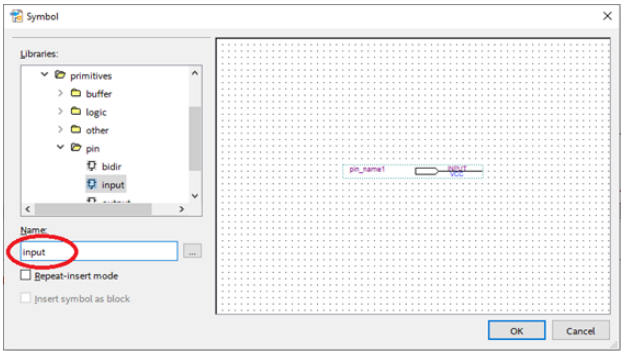
Устанавливаем Quartus 17, запускаем. Картинок будет по минимуму, иначе статья выйдет огромной. Создаем новый проект: «New Project». Создаем именно «ПРОЕКТ» (а не просто «файл»). Путь к проекту должен быть на латинице и без пробелов в названии. Желательно что-нибудь короткое наподобие «C:\Projects\Quartus\17». Назовём проект



Выбор конкретной микросхемы FPGA пока пропустим – «Next». Жмём OK. Теперь создаем новый файл. Выбираем сверху File à New à Block Diagram / Schematic File (файл с блок-схемами). Жмем «OK».

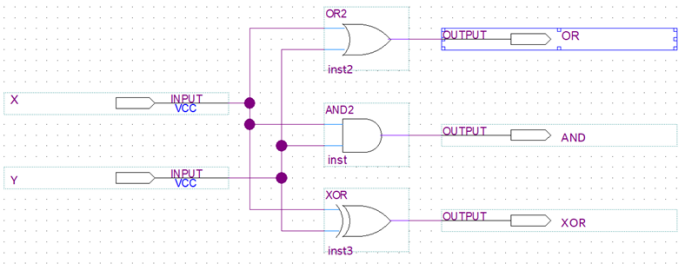


Щелкаем правой кнопкой мыши по середине поля, выбираем Insert → Symbol. В поле «Name» вводим «input». Выбираем, переносим на поле.



Опять правая кнопка, Insert → Symbol. Переименуем входные пины (Pin) в «X» и «Y».

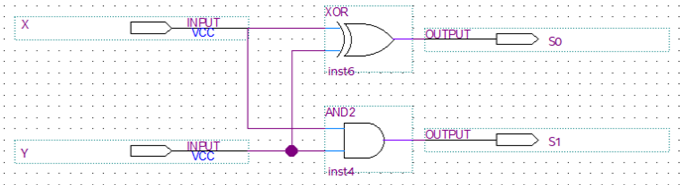
Добавим выходной сигнал. Поступаем так же: Insert → Symbol. В поле «Name» пишем: «Output». Жмем OK. Можно, зажав CTRL, потянуть левой кнопкой мышки за элемент блок-схемы и получить новые копии данного элемента, что мы и сделаем. Вставим элементы для логических операций «Не», «Или», «И». Получим схему.



Теперь сложим два бита, то есть сделаем арифметическую операцию. Построим таблицу истинности.

X	Y	S = X + Y	S1 = X «AND» Y	S0 = X «XOR» Y
0	0	00	0	0
0	1	01	0	1
1	0	01	0	1
1	1	10	1	0

Видно, что входные операнды являются однобитными, а выходной операнд двухбитным. Сложение увеличивает разрядность на 1 бит. Самый старший, «дополнительный», бит суммы часто называют «битом переноса» (Carry Out). Реализуем операцию в схеме.



Так как при сложении двух чисел производится операция переноса (carry out), то сумматор должен быть не двух, а трехвходным – входит еще бит переноса (carry in, Cin) от предыдущего сумматора. Таблица истинности для 3х битового сумматора (с переносом).

X	Y	Cin	S	S1	S0
0	0	0	00	0	0
0	0	1	01	0	1
0	1	0	01	0	1
0	1	1	10	1	0
1	0	0	01	0	1
1	0	1	10	1	0
1	1	0	10	1	0
1	1	1	11	1	1

Для создания схемы необходимо составить нормальную форму со всеми переменными. В интернете данная схема легко ищется (запрос «полный сумматор»).

Процессоры одиночные биты не складывают. Сложение происходит минимум 8 бит (1 байт), а в 64-х битных процессорах соответственно 64 бита (8 байт). Современные процессоры включают миллиарды простейших операций, поэтому таким способом, как мы начали, их уже никто не проектирует.

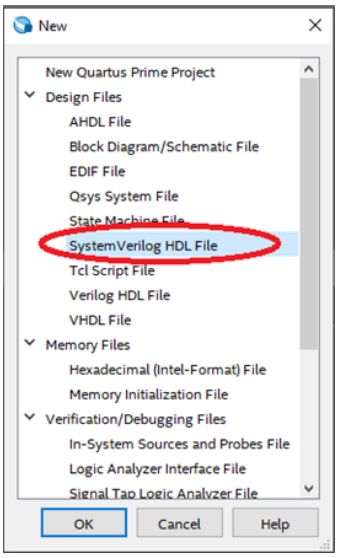
Мы проектировать мышкой тоже не будем. И так понятно, что это очень трудозатратный процесс. Мы еще вернемся к «проектированию блоками» посредством перетаскивания мышкой, но уже на уровне High-level synthesis (HLS) – синтезе высокого уровня.

Для описания проектируемых схем служат языки описания цифровой аппаратуры HDL (Hardware Description Language). Повторюсь, именно *описания*, так как про *программирование* пока речи не идет. Описание производится при помощи текстовой информации. На языках HDL именно описывают, как устроена микросхема, а не то, что в ней происходит. По большому счету, языков HDL всего два – это Verilog и VHDL. На взгляд автора, Verilog намного проще для изучения, поэтому начнем с него.

Языки HDL постоянно обновляются.

Сейчас на смену языку Verilog приходит язык System Verilog. Скорее всего, придя на работу, вы будете писать именно на System Verilog, поэтому его и будем изучать.

Сложим два байта с помощью языка SystemVerilog. Создадим новый проект (старый еще пригодится). Создаем «New Project». Назовем его «MakeProcSV». Далее создаем файл System Verilog HDL File.



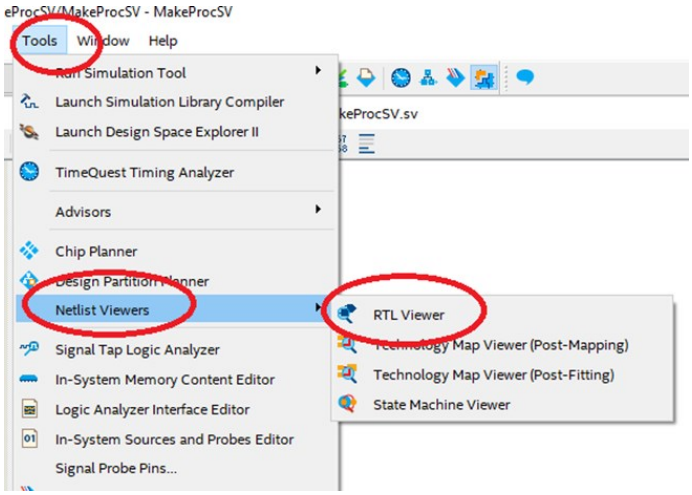
В появившееся поле введем код на языке System Verilog.

```
module MakeProcSV (X, Y, S); // объявление модуля
   input [7:0] X, Y; // входные сигналы
   output [8:0] S; // выходной сигнал

   assign S = X + Y; // собственно сложение
endmodule // завершение описания модуля
```

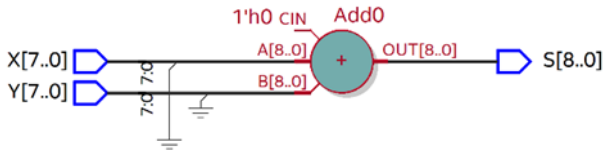
И это всё! Пять строчек кода!

Сохраняем, файл называем так же, как и модуль – MakeProcSV. Компилируем – Processing → Start Compilation (или «Ctrl + L»). Или нажимаем на пиктограмму - значок треугольника голубого цвета сверху на панели

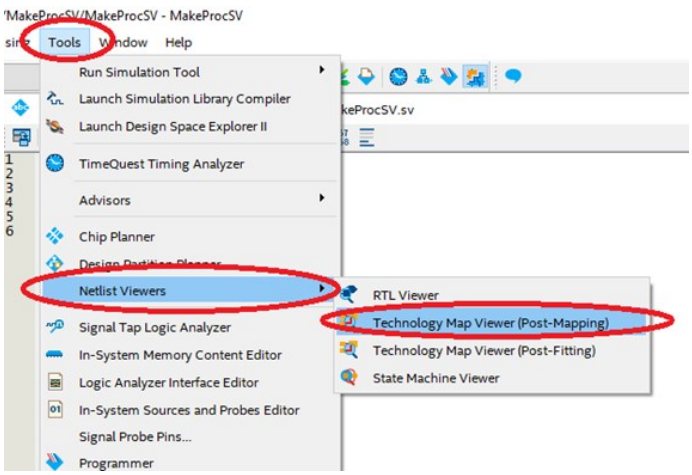
Компиляция занимает продолжительное время, за её ходом можно следить в левом нижнем углу программы Quartus. Посмотрим, что получилось. Используем средство просмотра Netlist Viewers (просмотрщик списка соединений). Его элемент RTL Viewer позволяет просматривать результаты компиляции проекта на уровне регистровых передач. Элемент Technology Map Viewer позволяет просматривать технологическую карту проекта на более



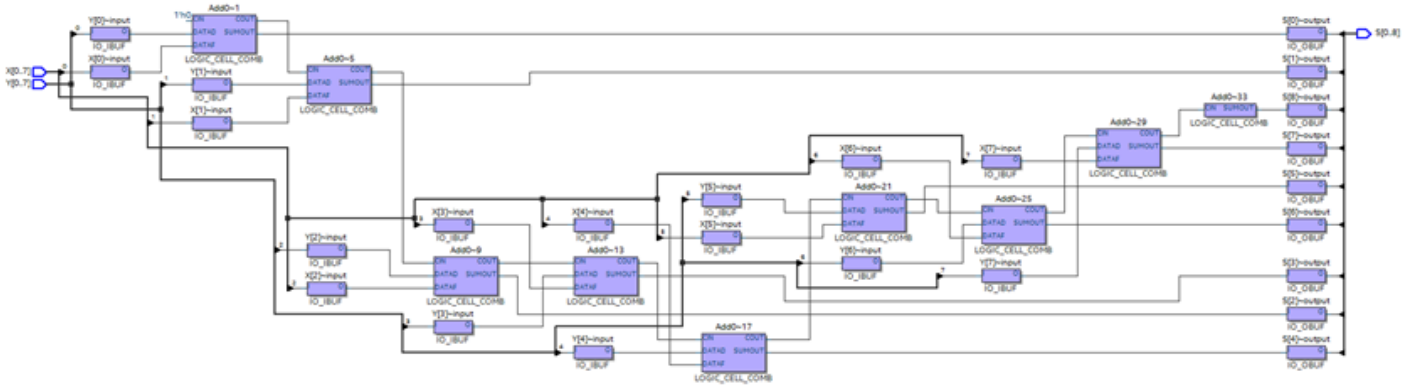
Посмотрим RTL схему.



Не особо впечатляюще. Теперь посмотрим Технологическую карту проекта (Technology Map Viewer).



Получим.

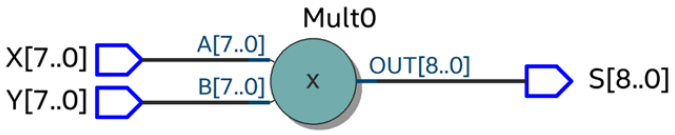


Пять строчек кода после компиляции превратились в схему с несколькими десятками элементов. Оцените всю мощь текстового описания!

Перемножим два байта.

```
module MakeProcSV (X, Y, S); // Объявление модуля
    input [7:0] X, Y; // Входные сигналы
    output [8:0] S; // Выходные сигналы

    assign S = X * Y; // Собственно умножение
endmodule // Завершение описания модуля
```

При просмотре получившейся схемы на уровне регистровых передач RTL (Tools → Netlist Viewers → RTL Viewer) мы увидим следующее.



При просмотре на уровне технологической карты проекта (Technology Map Viewer) мы увидим не такое большое количество элементов. Дело в том, что микросхема FPGA использует встроенные умножители и другие блоки.

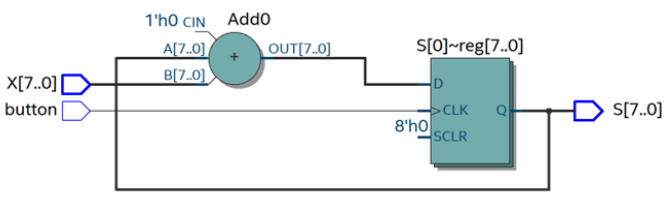
Мы пока используем КОМБИНАЦИОННЫЕ схемы – выходные значения зависят от комбинации входных. Но представьте, что необходимо сложить 20 чисел или более. При увеличении количества операндов до 20 схема возрастает неимоверно.

Сделаем по-другому. К примеру, произведём ввод одного операнда, потом нажмем кнопку, после введем другой операнд. И сделаем это столько раз, сколько нам необходимо. Результат сложения будем хранить отдельно, в так называемом АККУМУЛЯТОРе.

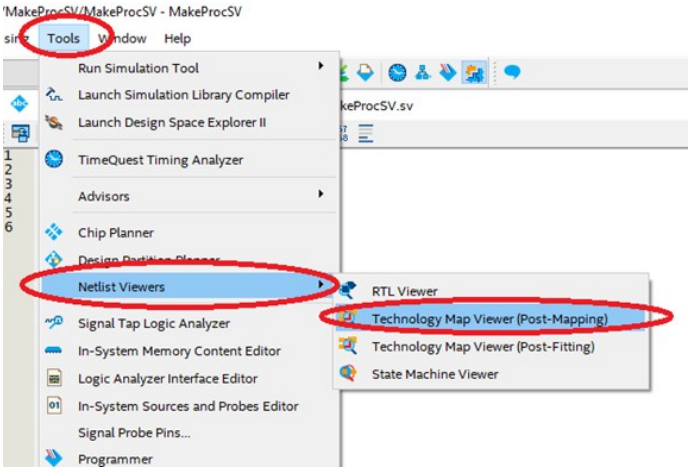
Реализуем ПОСЛЕДОВАТЕЛЬНОСТНУЮ схему, то есть такую схему, результат работы которой зависит в том числе от последовательности предыдущей ее работы.

```
module MakeProcSV (button, X, S); // Объявление модуля
   input button; // Входной сигнал синхронизации
   input [7:0] X; // Входное число
   output [7:0] S; // Сумма - выходное значение

    always @(posedge button) begin // Начало поведенческого блока
        S = S + X; // Собственно суммирование
    end // Окончание поведенческого блока
endmodule // Конец модуля
```

Новый блок «always» говорит схемосинтезатору о том, что при прохождении переднего фронта (posedge) сигнала button необходимо выполнить действия, описанные в данном блоке. Посмотрим схему на уровне RTL.



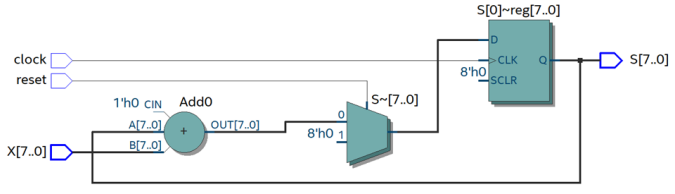
Видно, что в схеме появился набор регистров, в котором хранится результат суммирования. Обязательно посмотрите схему на технологическом уровне. Вы увидите много элементарных сумматоров и регистров, реализующих обратную связь.



Так как мы процессоростроители, то нажимать кнопку будем не мы, а тактовый генератор. Вместо ручного «нажатия на кнопку» (сигнал «button») доверим это дело генератору тактовых сигналов (сигнал «clock»). Для сброса в «0» результата суммирования введём команду «Reset» (сброс).

```
module MakeProcSV (reset, clock, X, S); //
Объявление модуля
   input reset; // Входной сигнал сброса
   input clock; // Входной сигнал
синхронизации
   input [7:0] X; // Входное число
   output [7:0] S; // Сумма - выходное
значение
   always @(posedge clock) begin // Начало
поведенческого блока
     if (reset) begin // Действуем в
зависимости от условия
       S = 0; // Обнуление суммы
     end else begin // Иначе суммируем
       S = S + X; // Собственно суммирование
    end // Конец else
   end // Конец if
endmodule // Конец модуля
```

Смотрим RTL-схему.



У нас появился новый элемент – мультиплексор. Это устройство, имеющее несколько сигнальных входов, один или более управляющих входов и один выход.

Однако процессор только с одной командой сложения на рынке микроэлектроники явно ажиотаж не вызовет.

Необходимо реализовать команды вычитания и умножения. Деление делать пока не будем, это очень трудозатратная для процессора операция.

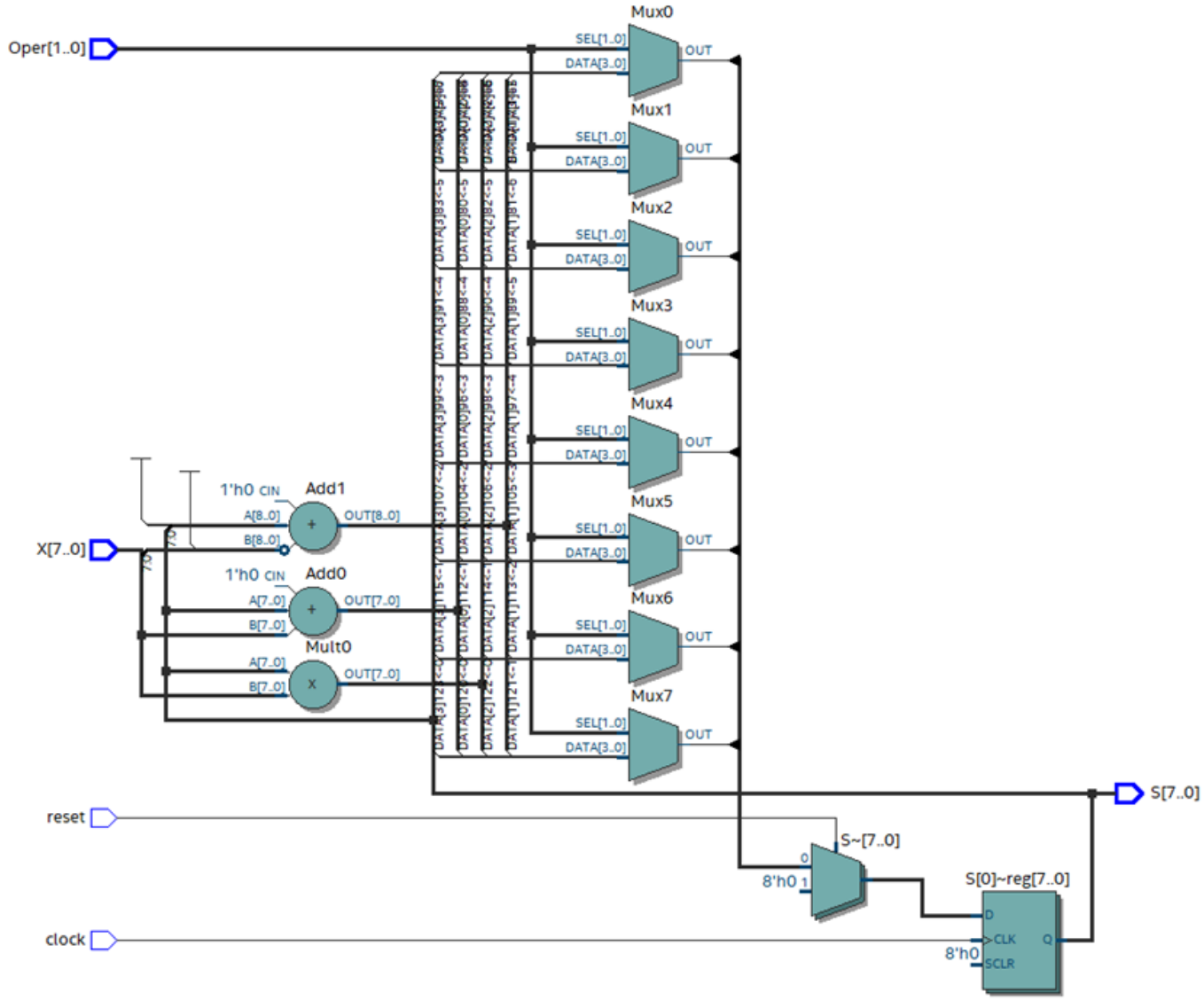
У нас будет одно АЛУ, в котором для выбора соответствующей арифметической операции будет применяться входной сигнал. Выбор будет производиться с помощью мультиплексора. Пока реализуем комбинационную схему с тремя операциями.

```
module MakeProcSV (X, oper, S); // Объявление
модуля
  input [7:0] X; // Входное число
  input [1:0] oper; // Выбор операции
  output [7:0] S; // Результат - выходное
значение

  always @(X) // При изменении X
    case(oper) // Выбор операции
      0: S = S + X; // Суммирование
      1: S = S - X; // Вычитание
      2: S = S * X; // Умножение
      default: S = S; // По умолчанию ничего
не делаем
    endcase // Конец выбора
endmodule // Конец модуля
```

Посмотрите получившуюся схему RTL.

Далее добавим аккумулятор и сигналы «clock» и «reset». Текст модуля.

```
module MakeProcSV (reset, clock, X, Oper,
S); // Объявление модуля
  input reset; // Входной сигнал сброса
  input clock; // Входной сигнал
синхронизации
  input [7:0] X; // Входное число
  input [1:0] Oper; // Выбор операции
  output [7:0] S; // Результат - выходное
значение

  always @(posedge clock) begin // При
изменении X
    if (reset) begin // Действуем в
```




```
зависимости от условия
     S = 0; // Обнуление суммы
    end // Конец if
    else begin // Иначе выполняем операцию
      case (Oper) // Выбор операции
      0: S = S + X; // Суммирование
      1: S = S - X; // Вычитание
      2: S = S * X; // Умножение
      default: S = S; // По умолчанию ничего
не делаем
      endcase // Конец выбора case
    end // Конец else
  end // Конец always
endmodule // Конец модуля
```

У нас получилось довольно хорошее АЛУ – ядро процессора!

Смотрим схему.

Слева на схеме видны два сумматора (один из них – на вычитание) и один блок умножения. В центре – восемь мультиплексоров (по разрядности нашего процессора). Правее – мультиплексор сброса reset и аккумулятор на 8 регистров, стробируемых по переднему фронту. Обязательно посмотрите схему на технологическом уровне (Tools → NetList Viwer → Technology Map Viewer).

Теперь, привязав входные и выходные пины схемы к выводам микросхемы FPGA, мы физически получим блок АЛУ, который при подаче соответствующих сигналов будет производить арифметические операции и выдавать получившийся результат.

Далее необходимо добавить регистры, программную память, блоки работы с периферией. И все – процессор готов! Но это все уже тема следующих частей проекта.

Шинная организация сигналов в DEEDS

Аверченко А.П.

ОмГТУ

apaverchenko@omgtu.ru

Обсуждение и комментарии: link

Продолжаем изучать программный пакет Deeds (Digital Electronics Education and Design Suite) - комплекс для обучения и разработки цифровой электроники. Это свободно распространяемое программное обеспечение, для преподавания цифровой схемотехники. Программу можно свободно скачать с официального сайта www.digitalelectronicsdeeds.com.

Проектирование цифровых схем ведётся при помощи соединения проводниками различных логических элементов. В случае не большого количества проводников (до 10-15) схема вполне нормально располагается на рабочей области проекта и вполне легко читается. Однако, когда количество элементов схемы и проводников значительно увеличивается, разобраться и проследить прокладку и ветвление проводников становится весьма затруднительно. Так же становится невозможно компактно расположить элементы на схеме и обеспечить необходимые соединения. Для решения возникающей проблемы в программной среде DEEDS существует возможность использовать такие компоненты как ШИНА  . Это группа проводников, находящихся в единой «трубе», которая и называется шиной. Каждый проводник имеет свой уникальный цифровой «идентификатор», по которому и определяется точка входа и выхода проводника в шину. На рисунке 1 представлены две абсолютно идентичные с точки зрения электрического соединения группы проводов. Однако явно видно, что вариант а) является более компактным. Более того, в варианте а) появляется свободное место, которое можно занять логическими элементами с соответствующими соединениями.

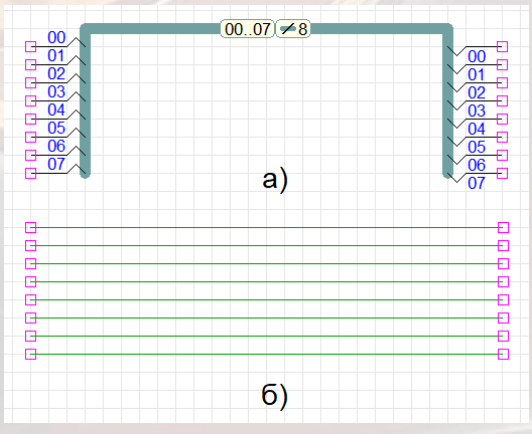


Рисунок 1 - два типа рисования проводников

Если же необходимо изменить порядок проводов, к примеру, «перевернуть» выходную шину, то в варианте а) достаточно будет просто изменить цифровые индексы отходящих соединений; а в варианте б) придётся полностью «перетасовывать» соединения, к примеру так, как показано на рисунке 2. При этом «разобраться на лету» становится весьма затруднительно.

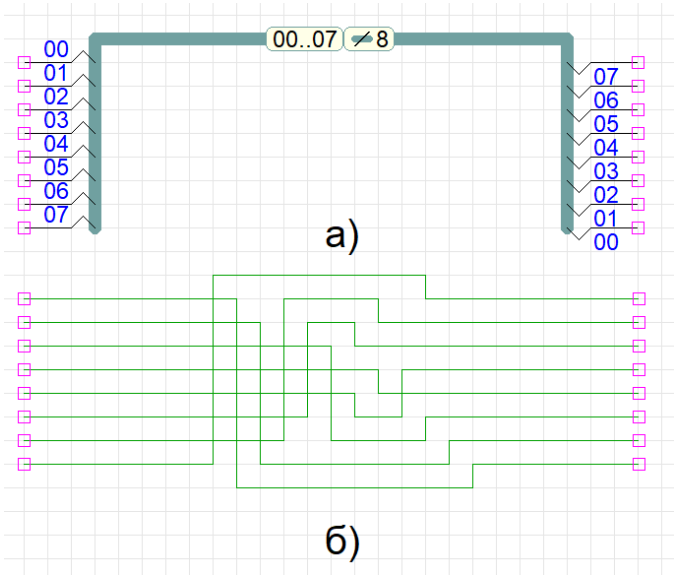


Рисунок 2 - «переворот» шины

Группа кнопок рисования шин представлена на рисунке 3.

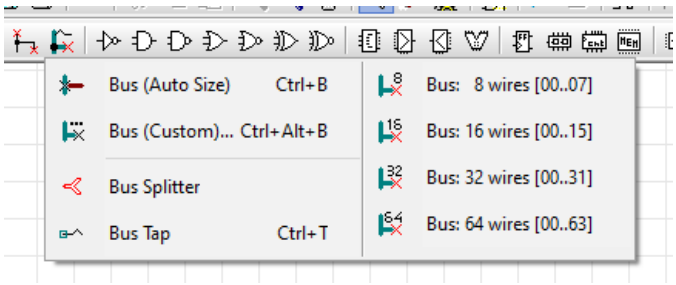


Рисунок 3 - кнопки рисования шин

С правой стороны рисунка 3 открываются варианты для рисования шин с наиболее часто встречающимися разрядностями (8, 16, 32, 64). В квадратных скобках указаны индексы проводников, которые будут присутствовать в данных шинах по умолчанию (как видно вся нумерация начинается с индекса «00», т.е. в шине присутствует проводник с таким индексом, что видно на рисунках 1 и 2). Рисование шин производится точно теми же принципами и способами, что и рисование проводников. Двойной клик на нарисованной шине открывает меню, позволяющее отобразить или скрыть подписи на шине, как это представлено на рисунке 4. При этом можно скрыть или отобразить диапазон индексов данной шины и/или разрядность шины.

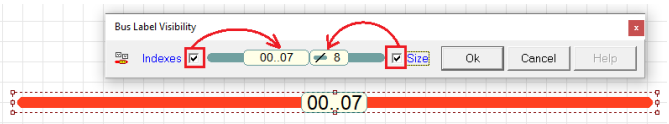


Рисунок 4 - подписи на шине

После того как шина будет вынесена на рабочую область, её разрядность и индексы проводников будут зафиксированы. Разрядность нарисованной шины поменять невозможно, и в случае необходимости шины с другой разрядностью, нужно рисовать новую шину. В нарисованной шине, в случае необходимости, можно изменить диапазон индексов: для этого следует воспользоваться пунктом контекстного меню (открывается по клику правой кнопки мыши на элементе шины) «Bus Indexes...».

Для создания шин с произвольной разрядностью и/или произвольными индексами используется кнопка  «Bus (Custom)...», представленная на рисунке 3. После её выбора открывается окно «Set New Bus Properties» и его первая вкладка «Bus Size», в которой необходимо выбрать разрядность создаваемой шины. Выпадающий список позволяет выбрать любое значение в диапазоне от 4 до 64. После чего необходимо нажать на кнопку «Next» и перейти во второе окно «Indexes», в котором также из выпадающего списка нужно выбрать необходимый диапазон индексов для создаваемой шины. При этом в выпадающем списке будут отображаться все возможные диапазоны (в пределах указанной на предыдущем шаге разрядности) для

проводников с индексами от «00» до «63». Если на этом шаге становится понятно, что разрядность выбрана неправильно, можно вернутся на предыдущую вкладку кнопкой «Previous». После выбора необходимого диапазона индексов становится активной кнопка «Next», по которой происходит переход на третью вкладку «Labels» которая настраивает отображение подписей на шине, так же как и меню, представленное на рисунке 4. Далее необходимо нажать «OK» и нарисовать созданную шину на рабочем пространстве проекта. Прямое соединение шин можно производить только при условии, что у них полностью соответствуют разрядность и индексы. Если необходимо соединить шины с разными разрядностями и/или индексами, необходимо использовать специальный компонент, о котором будет рассказано далее.

Часто необходимо продлить имеющуюся шину с уже определёнными разрядностью и индексами. Для этого не нужно заново создавать точно такую же шину, а можно воспользоваться кнопкой  «Bus (Auto Size)», которая так же представлена на рисунке 3. Данная кнопка при соединении с уже нарисованной шиной принимает все её параметры и рисует продолжение этой шины.

Подключение проводников к шинам производится компонентом «Bus Tap», который представлен на рисунке 3. При этом, как показано на рисунках 1 и 2, данный компонент должен подключаться к шине строго определенным образом, а именно концом с «уголком». Также необходимо, чтобы каждый компонент «Bus Tap» имел индекс, который присутствует в шине, как это показано на рисунке 5. Соответственно, для подключения «Bus Tap» с любой стороны шины его необходимо вращать перед установкой (как это делается описано в предыдущем журнале). После выноса одного компонента ему задаётся индекс из диапазона имеющихся в шине, к которой его подключили. Шинные отводы можно тиражировать до нужной разрядности. Это делается указанием соответствующего приращения к начальному индексу первого отвода «+1» или «-1» в меню, открывающемся при рисовании первого отвода, и тиражированием «фантомов» до нужной разрядности, после чего завершить ввод левой кнопкой мыши. С противоположной стороны «Bus Tap» подключаются обычные проводники, как это показано на рисунке 5. При этом «Bus Tap» сам не определяется как вход или выход шины. Это просто точка входа/выхода проводника в/из шины. Конкретное значение входа или выхода определяется исходя из подключённого к отводу электрическому сигналу (логическому компоненту). Общее правило точно такое же, как и для проводников: любой проводник может иметь только один источник информации и сколько угодно приёмников.

Шины, так же как и проводники, могут иметь межшинные соединения, которые отображаются жирной точкой в месте соединения шин, как это показано на рисунке 5. Если шины в месте пересечения не имею точки, значит, шины не соединены, а просто пересекаются.

Соединение шин разной разрядности, ответвление части шины или соединение шин с различными индексами в единую шину осуществляется при помощи компонента  «Bus Splitter», который представлен на рисунке 3. Данный компонент позволяет собирать шины разрядностью до 64 бит. Таблица индексов данного компонента открывается выбором пункта «Split Connections» в контекстном меню (кликом правой кнопкой миши на данном

компоненте). Таблица соединений для компонента «Bus Splitter», представленного на рисунке 5, отображена на рисунке 6.

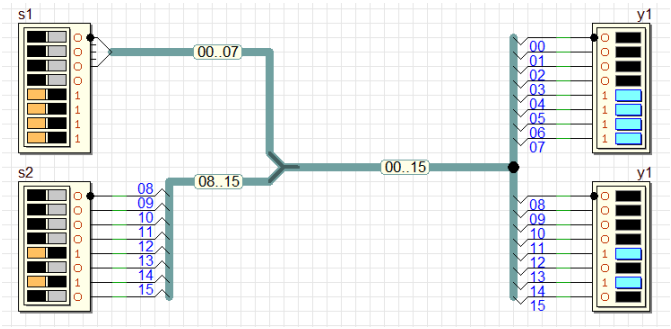


Рисунок 5 - Схема шинной организации

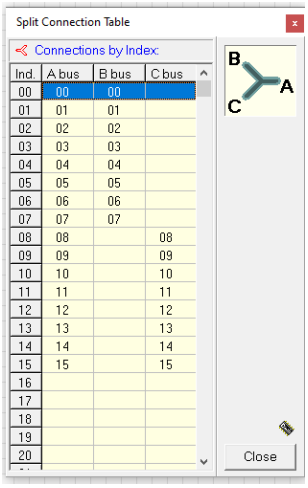


Рисунок 6 - таблица соединений компонента Bus Splitter

Компоненты в DEEDS могут иметь как шинные входы/выходы - компонент «s1» на рисунке 5, так и обычные проводниковые входы/выходы - «s2», «y1», «y2» на рисунке 5. Компоненты, имеющие шины в качестве точек контакта, позволяют подключать к себе только шины идентичной разрядности. При этом индексы могут быть любые.

У каждого из компонентов «s1», «s2», «y1», «y2», на рисунке 5, на одном из входов (верхних, согласно данной реализации) есть символ чёрной точки. Эта точка обозначает младший разряд при подключении к шине данных.

Пример, представленный на рисунке 5, можно симулировать и изучить, скачав по следующей ссылке:

https://drive.google.com/file/d/156Fr_C6Ck7GsTrpWhsOEfRQyfH5tHQDR

(в случае невозможности скачивания сообщите об этом автору документа)

Таким образом организуются шинные соединения в программной среды Deeds. В следующих статьях будем более детально рассматривать особенности использования программной среды Deeds.

“Бегущие огни” на ATF22V10

Харабадзе Д.Э.

141120@mail.ru

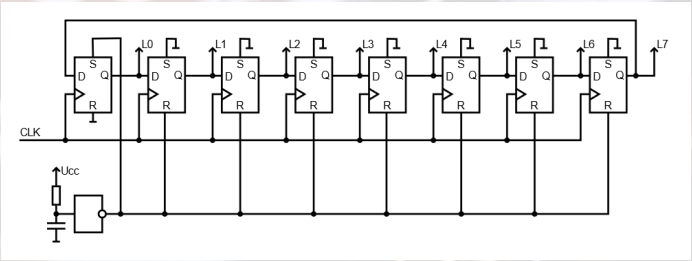
Обсуждение и комментарии: link

Микросхема ATF22V10 фирмы «Microchip» [1,2] представляет собой программируемую логическую микросхему, содержащую 10 конфигурируемых выходных блоков и программируемую матрицу соединений. Эта микросхема является аналогом снятой с производства микросхемы GAL22V10 фирмы «Lattice semiconductor»[3].

Главными достоинствами этой микросхемы является невысокая цена и доступность микросхемы в корпусе DIP-24, что позволяет использовать её в беспаечных макетных платах. Микросхема может использоваться для создания простых схем, не требующих большого количества вентилей.

Для демонстрации возможностей микросхемы было принято решение сделать устройство для создания эффекта «бегущих огней». В устройстве будет 8 светодиодов, которые будут загораться последовательно, создавая эффект «бегущего огонька». В качестве тактового генератора был использован RS-триггер с обратной связью, состоящей из двух резисторов и одного конденсатора. Причём RS-триггер был реализован на той же микросхеме.

Для того, чтобы обеспечить эффект «бегущих огней» часто используется сдвиговый регистр, например, образованный из 8-ми D-триггеров.



Этот сдвиговый регистр при каждом тактовом импульсе будет сдвигать своё содержимое. Для того, чтобы изначально в одном из D-триггеров появилась единица, необходимо обеспечить начальную установку D-триггеров. Схема начального сброса подключается входу «S» одного из D-триггеров, и ко входам «R» оставшихся триггеров, тем самым, обеспечивая нужное начальное состояние. К сожалению, эта схема не может быть реализована в AT-F22V10, так как входы установки начального состояния триггеров соединены вместе. То есть, можно одновременно установить их в «0» (асинхронно) или в «1» (синхронно), но

назначить разные сигналы им нельзя.

Выйти из положения можно, в начальный момент сбрасывая все D-триггеры в ноль, но инвертируя входы и выходы одного из D-триггеров:

Для описания этой схемы будем пользоваться языком описания CUPL [4]:

```
Name     ShiftReg ;
PartNo   00 ;
Date     17.05.2024 ;
Revision 01 ;
Designer David ;
Company  FPGA-Systems.ru ;
Assembly None ;
Location  ;
Device   G22V10 ;

// --------------- Input pins
PIN 1 = CLK;
PIN 2 = R;

// --------------- Output pins
PIN 21 = !L0;
PIN 20 =  L1;
PIN 19 =  L2;
PIN 18 =  L3;
PIN 17 =  L4;
PIN 16 =  L5;
PIN 15 =  L6;
PIN 14 =  L7;

L0.D=!L7;
L1.D=!L0;
L2.D=L1;
L3.D=L2;
L4.D=L3;
L5.D=L4;
L6.D=L5;
L7.D=L6;

L0.AR = !R;
L1.AR = !R;
L2.AR = !R;
L3.AR = !R;
L4.AR = !R;
L5.AR = !R;
L6.AR = !R;
L7.AR = !R;

L0.SP = 'B'0;
L1.SP = 'B'0;
L2.SP = 'B'0;
L3.SP = 'B'0;
L4.SP = 'B'0;
L5.SP = 'B'0;
L6.SP = 'B'0;
L7.SP = 'B'0;
```

Симуляция даёт следующий результат (рисунок 1).

Но эта схема требует схемы начальной установки, что потребует дополнительной внешней RC-цепочки для начальной установки. Тем не менее, можно сделать схему таким образом, чтобы начальная

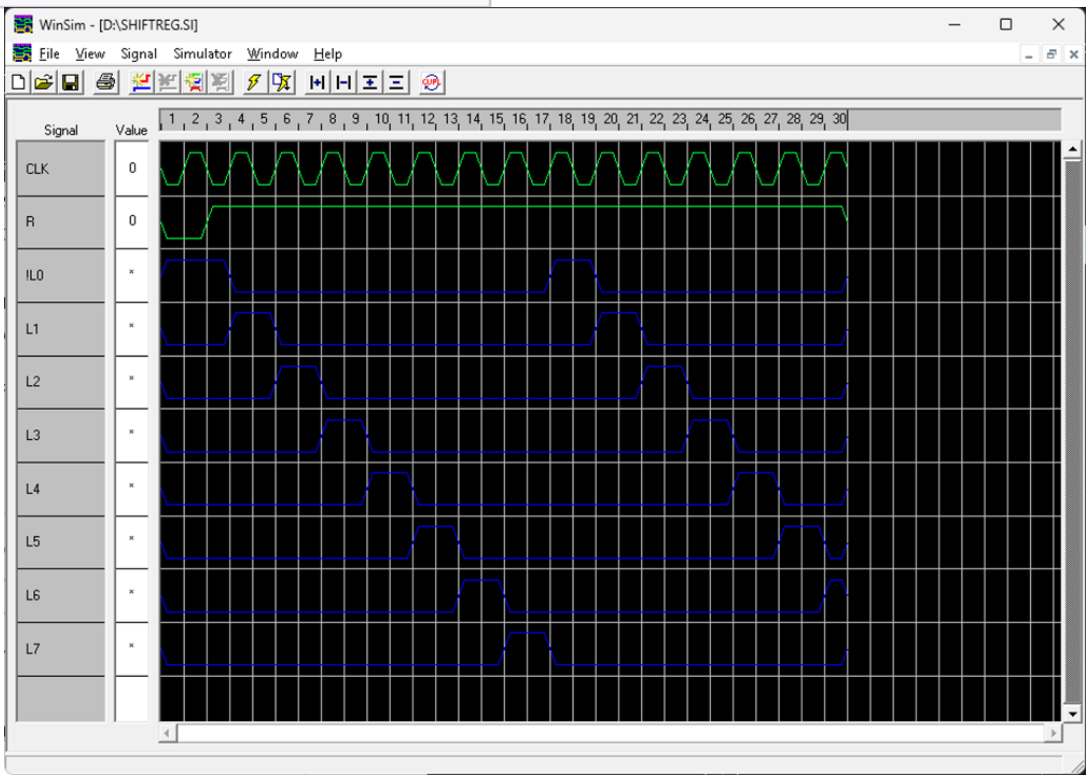


Рисунок 1

установка не требовалась. Для этого необходимо, чтобы любое состояние, в котором более, чем в одном D-триггере находилась бы единица, приводило бы к начальному состоянию. Заодно можно реализовать RS-триггер, чтобы использовать его в качестве генератора для тактирования схемы. Схема установки теперь не требуется.

Описание схемы будет:

```
Name     Lights ;
PartNo   ;
Date     16.05.2024 ;
Revision 01 ;
Designer David ;
Company  FPGA-Systems.ru ;
Assembly None ;
Location ;
Device   G22V10;

// --------------- Input pins
PIN 1 = S ;
PIN 2 = R ;

// --------------- Output pins
PIN 23 = NQ;
PIN 22 = Q;
PIN 21 = L0;
PIN 20 = L1;
PIN 19 = L2;
PIN 18 = L3;
PIN 17 = L4;
PIN 16 = L5;
PIN 15 = L6;
PIN 14 = L7;

// ---------------- Trigger
Q  = S & !R # !NQ;
NQ = !S & R # !Q;

//----------------- Lights
TEMP0 =  L0 & !L1 & !L2 & !L3 & !L4 & !L5 & !
L6 & !L7;
TEMP1 = !L0 &  L1 & !L2 & !L3 & !L4 & !L5 & !
L6 & !L7;
TEMP2 = !L0 & !L1 &  L2 & !L3 & !L4 & !L5 & !
L6 & !L7;
TEMP3 = !L0 & !L1 & !L2 &  L3 & !L4 & !L5 & !
L6 & !L7;
TEMP4 = !L0 & !L1 & !L2 & !L3 &  L4 & !L5 & !
L6 & !L7;
TEMP5 = !L0 & !L1 & !L2 & !L3 & !L4 &  L5 & !
L6 & !L7;
TEMP6 = !L0 & !L1 & !L2 & !L3 & !L4 & !L5
&  L6 & !L7;

L0.D = !(TEMP0 # TEMP1 # TEMP2 # TEMP3 #
TEMP4 # TEMP5 # TEMP6);
L1.D = TEMP0;
L2.D = TEMP1;
L3.D = TEMP2;
L4.D = TEMP3;
L5.D = TEMP4;
L6.D = TEMP5;
L7.D = TEMP6;

//----------------- Set and Reset
L0.AR = 'B'0;
L1.AR = 'B'0;
L2.AR = 'B'0;
L3.AR = 'B'0;
L4.AR = 'B'0;
L5.AR = 'B'0;
L6.AR = 'B'0;
L7.AR = 'B'0;

L0.SP = 'B'0;
L1.SP = 'B'0;
L2.SP = 'B'0;
L3.SP = 'B'0;
L4.SP = 'B'0;
L5.SP = 'B'0;
L6.SP = 'B'0;
L7.SP = 'B'0;
```

Симуляция даёт следующие результаты, показанные на рисунке 2.

Для того, чтобы RS-триггер стал работать в качестве генератора, следует подключить конденсатор ёмкостью 10 мкФ и два резистора по 3.3 кОм по приведённой схеме (рисунок 3).

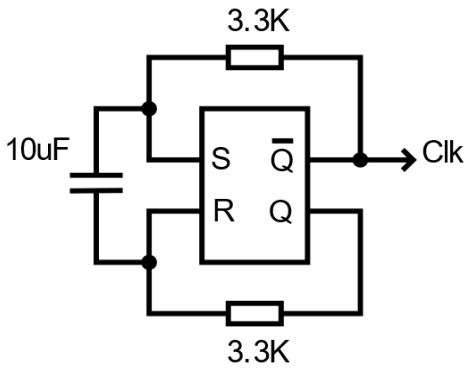


Рисунок 3

Общая схема подключения микросхемы будет следующей: (рисунок 4).

Предложенная схема может быть легко собрана на макетной плате и использоваться в качестве демонстрации возможностей микросхемы ATF22V10.

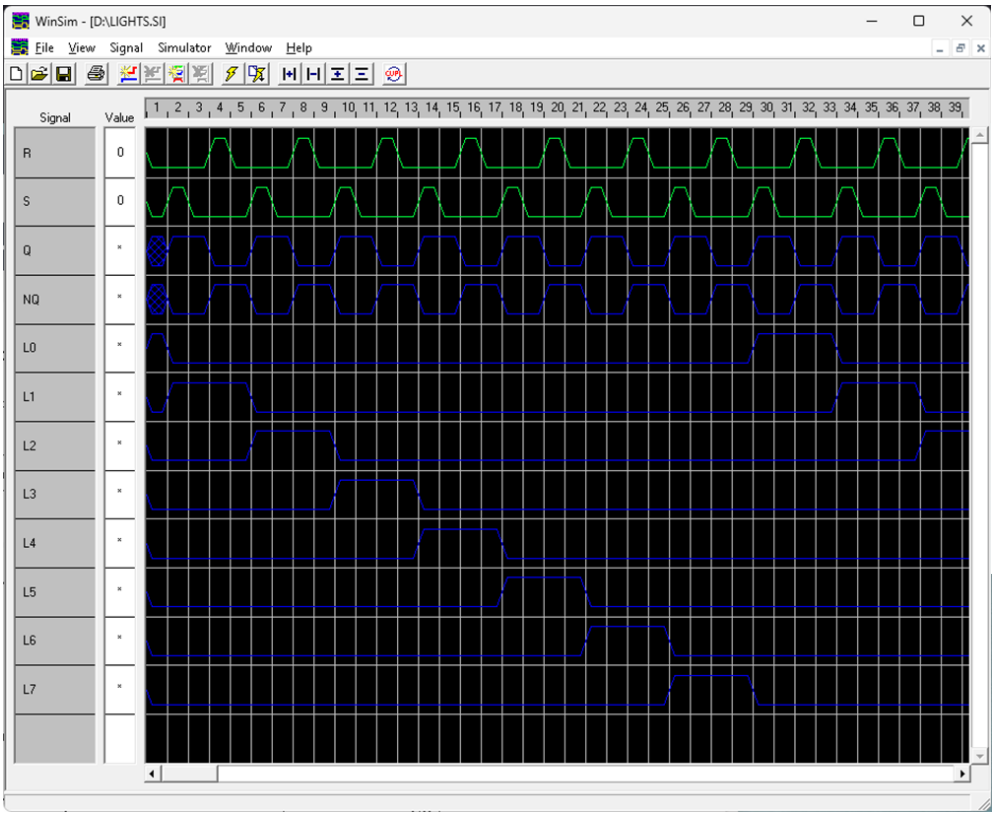


Рисунок 2

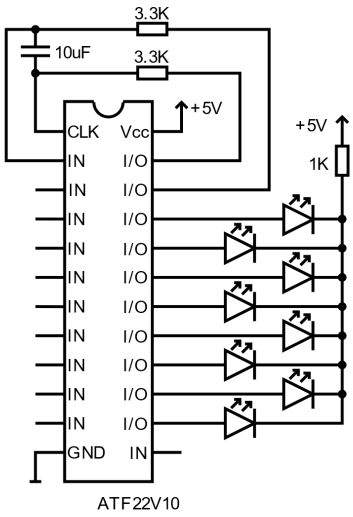


Рисунок 4

Источники

1. https://www.microchip.com/en-us/product/atf22v10c
2. https://ww1.microchip.com/downloads/en/DeviceDoc/doc0735.pdf
3. https://en.wikipedia.org/wiki/GAL22V10
4. https://ww1.microchip.com/downloads/en/DeviceDoc/doc0737.pdf

Сдвиговый регистр или то, о чем не расскажут в статьях для начинающих :: атрибуты синтеза

Коробков Михаил

Telegram: @KeisN13
e-mail: admin@fpga-systems.ru

Обсуждение и комментарии: link

Аннотация

Сдвиговый регистр – это одна из наиболее часто применяемых конструкций в проектах на ПЛИС. Сегодня мы уделим внимание тому, как можно и нужно писать VHDL и Verilog код для сдвиговых регистров, но при этом, чтобы синтезатор понял, что мы хотим реализовать сдвиговый регистр из нескольких триггеров или же задействовать для его реализации специальные ресурсы, такие SRL регистры.

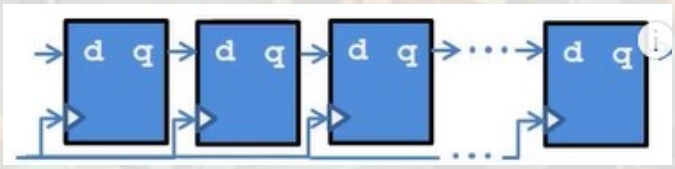
Мы разберем случаи, в которых надо и не надо использовать аппаратные сдвиговые регистры, разберем тонкие моменты их имплементации. Мы затронем пересечение тактовых доменов и увидим, что в некоторых случаях правильно описанный сдвиговый регистр может стать причиной некорректного поведения схемы.

Введение

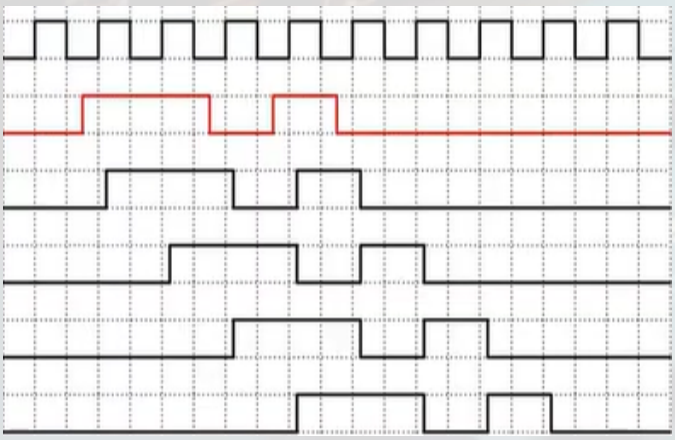
Перед тем как перейти к написанию кода мы очень кратко разберем, что же такое сдвиговый регистр, как и где он применяется.

Мы разберем сдвиговый регистр, применительно к FPGA, поскольку на Википедии и различных обучающих видео по дискретной цифровой электронике / схемотехнике вы можете увидеть несколько схем их построения, но сущность сдвиговых регистр от этого не меняется.

Итак, сдвиговый регистр представляет собой последовательно включенные триггеры, причем выход одно триггера подключен к входу другого триггера.

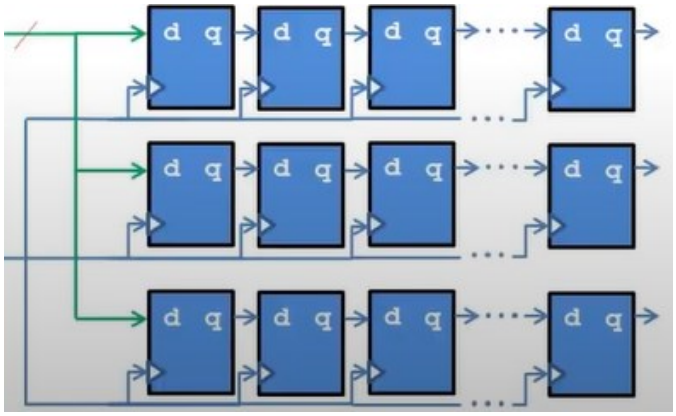


Как и следует из названия, сдвиговый регистр что-то сдвигает, а именно входящие данные.



На каждом такте поданные на вход сдвигового регистра данные сдвигаются вправо или влево, в зависимости от того, что вам требуется.

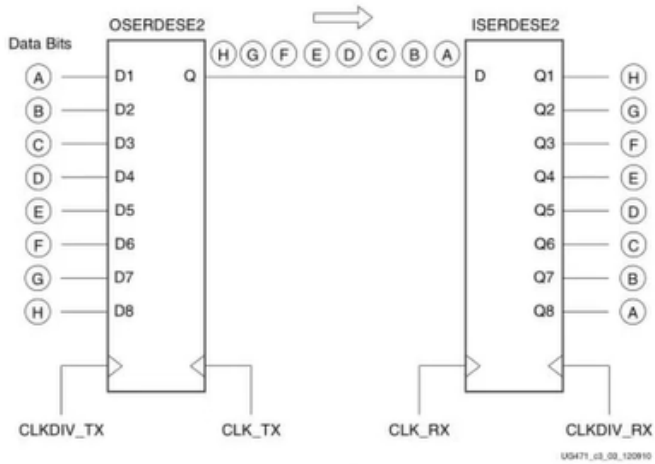
Сдвиговые регистры могут иметь не только один вход, но и работать с векторами или шиной. В таком случае количество цепочек триггеров просто многократно дублируется, а общее количество таких цепочек равно разрядности сдвигаемых данных.



Сдвиговые регистры могут иметь различный функционал, в зависимости от решаемой вами задачи. Они могут :

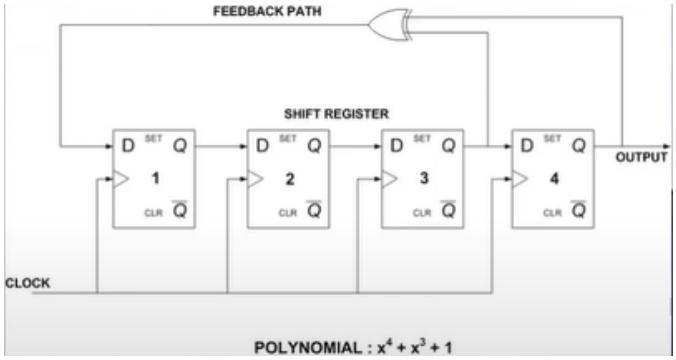
• Иметь управляющие сигналы, такие как сброс и сигнал разрешения тактовой частоты clock enable

• Иметь вход загрузки данных для сдвига – то есть выступать в качестве серилайзера



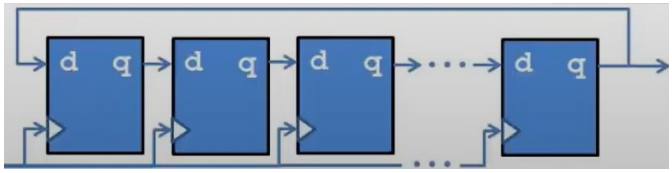
• Иметь разрешающий сигнал параллельной выгрузки данных – то есть выступать в качестве десерилайзера

• Иметь отводы от некоторых триггеров – таки образом строятся генераторы псевдослучайных последовательностей

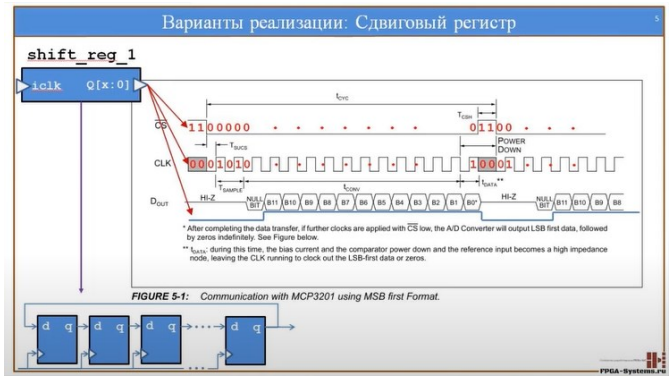


• Иметь отводы и управляющий мультиплексор для создания динамической линии задержки

• Выход сдвигового регистра может быть соединен с входом сдвигового регистра – таким образом можно циклически прокручивать заранее записанные данные по кругу.



Кстати, такой прием мы использовали на одном из наших стримов по реализации интерфейса SPI на ПЛИС для работы с АЦП, ссылка на этот стрим



Несколько слов о применении сдвиговых регистров. Наиболее часто их применяют:

• Для задержки сигналов на определенное количество тактов, причем величина задержки может быть как фиксированной, так и переменной величиной

• для построения генераторов псевдослучайных последовательностей

• для реализации серилайзеров и десерилайзеров

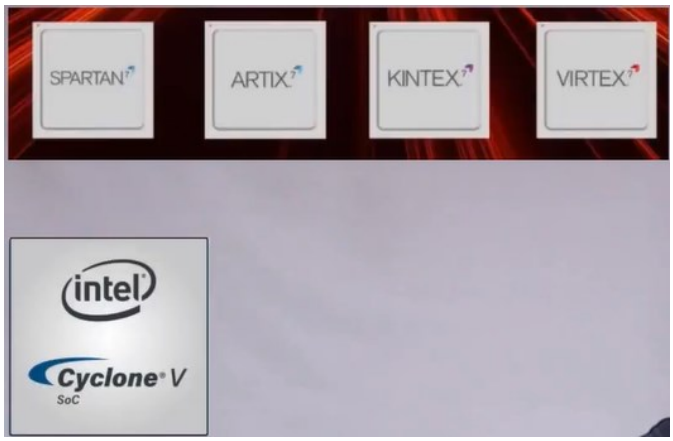
• для проигрывания зацикленных данных

• для реализации регистра барабанного сдвига – barrel shifter, применяемого, например, для сложения и вычитания чисел с плавающей точкой, где он производит нормализацию мантисс

• для реализации счетчиков и тд.

Вот небольшое количество применений сдвиговых регистров. Если вы уверены, что список применений можно дополнить, то обязательно напишите об этом в комментариях (ссылка для коментариев под заголовком статьи справа), указав область применения.

Перед тем как мы начнем, сделаю небольшое отступление. Надо помнить, что не все ПЛИС имеют сдвиговый регистр как аппаратный специально выделенный ресурс. Чтобы понять, есть ли он в вашей ПЛИС или нет нужно обратиться к документации на выбранную микросхему и посмотреть наличие SRL или Shift Register примитива в архитектуре. В наших примерах мы будем использовать микросхемы 7-ой серии для Xilinx и Cyclone V для Intel/Altera



Перейдем к практике.

Описать сдвиговый регистр на VHDL или Verilog достаточно просто.

В случае не параметризированного варианта код для сдвигового регистра, реализующего линию задержки на 5 тактов будет выглядеть следующим образом:

• для VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity srl_1_vhdl is
    port (
        id      : in std_logic;
        iclk    : in std_logic;
        oq      : out std_logic);
end srl_1_vhdl;

architecture rtl of srl_1_vhdl is

    signal dff0, dff1, dff2, dff3, dff4 :
std_logic := '0';

begin

    srl_1 : process (iclk) begin
        if rising_edge(iclk) then
            dff0 <= id;
            dff1 <= dff0;
            dff2 <= dff1;
            dff3 <= dff2;
            dff4 <= dff3;
        end if;
    end process srl_1;

    oq <= dff4;

end rtl;
```

• и для Verilog

```verilog
module srl_1_verilog(
    input id,
    input iclk,
    output oq
    );

    reg dff0, dff1, dff2, dff3, dff4 = 1'b0;

    always @(posedge iclk) begin
      dff0 <= id;
      dff1 <= dff0;
      dff2 <= dff1;
      dff3 <= dff2;
      dff4 <= dff3;
    end
    assign oq = dff4;
```

```
endmodule
```

Разумеется, стиль описания может быть и другим, например, мы можем записать это и вот так, используя более простую конструкцию для VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity srl_2_vhdl is
    port (
        id      : in std_logic;
        iclk    : in std_logic;
        oq      : out std_logic);
end srl_2_vhdl;

architecture rtl of srl_2_vhdl is

    signal dff : std_logic_vector( 4 downto
0 ) := (others => '0');

begin

    srl_2 : process (iclk) begin
        if rising_edge(iclk) then
            dff <= dff(3 downto 0) & id;
        end if;
    end process srl_2;

    oq <= dff(4);

end rtl;
```

и Verilog

```
 module srl_2_verilog(
    input id,
    input iclk,
    output oq
    );

   reg [4:0] dff = 5'b00000;
    always @(posedge iclk) begin
      dff <= {dff[3:0], id};
    end

    assign oq = dff[4];

endmodule
```

Или вот так, используя цикл for на языке VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity srl_3_vhdl is
    port (
        id      : in std_logic;
        iclk    : in std_logic;
        oq      : out std_logic);
end srl_3_vhdl;

architecture rtl of srl_3_vhdl is

    signal dff : std_logic_vector( 4 downto
0 ) := (others => '0');

begin

    srl_3 : process (iclk) begin
        if rising_edge(iclk) then
            dff(0) <= id;
            for i in 0 to 3 loop
                dff(i + 1) <= dff(i);
            end loop;
        end if;
    end process srl_3;

    oq <= dff(4);

end rtl;
```

и Verilog

```
module srl_3_verilog(
    input id,
    input iclk,
    output oq
  );

  reg [4:0] dff = 5'b00000;
  integer i;

  always @(posedge iclk)
  begin
    dff[0] <= id;
    for (i = 0; i < 4; i=i+1)
    begin
      dff[i + 1] <= dff[i]
       end
     end

     assign oq = dff[4];

endmodule
```

Здесь как говорится на вкус и цвет. Если вы хотите дополнить проект своими вариантами реализации и описания сдвиговых регистров, делайте пул реквесты в гит репозиторий, или же оставляйте ваш код в комментариях (ссылка для коментариев под заголовком статьи справа).

Я привел по три варианта реализации, но суть от этого не меняется.

Нам важно понять, каким образом мы можем управлять синтезатором, чтобы получить либо цепочку из отдельных триггеров, либо специально выделенный аппаратный ресурс. Нам нужно найти такие инструменты и средства либо языка описания аппаратуры VHDL или Verilog либо настройки синтеза в самих средах проектирования, таких как Quartus или Vivado, которые помогли бы нам решить поставленную задачу.

Аттрибуты в языках описания аппаратуры

Мы начнем со средств языков описания аппаратуры VHDL и Verilog. Влиять напрямую на синтез мы можем с помощью специальной языковой конструкции, которая называется атрибут. Атрибуты есть как в VHDL так и в Verilog. Синтаксис атрибутов следующий

Для VHDL он задается с помощью объявления атрибута, имени или идентификатора атрибута и типа атрибута. Затем записывается то, к чему атрибут присваивается: entity | architec.| procedure | function | package и тд и затем значение атрибута.

```
attribute identifier ':' ( type-name | subtype-name ) ';'
attribute attribute-name-identifier of entity-class-name-list ':' ( entity | architec.|
  configuration | procedure | function | package | type | subtype | constant | signal
  variable | component | label | literal | units | group | file ) is expression ';'
```

На практике это выглядит примерно вот так,

CASCADE_HEIGHT VHDL example

```
attribute cascade_height : integer;
attribute cascade_height of ram : signal is 4;
```

здесь атрибут CASCADE_HEGHT применяется к сигналу ram, а значение атрибута равно 4

Для языка Verilog атрибут задается непосредственно перед объявлением чего-либо, и записывается внутри специальной конструкции из скобочек и звездочек

Тот же атрибут CASCADE_HEGHT на языке Verilog задается следующим образом

CASCADE_HEIGHT Verilog example

```
(* cascade_height = 4 *) reg [31:0] ram [(2**15) - 1:0];
```

Следует обратить внимание на тот факт, что разные среды проектирования поддерживают разные атрибуты. Атрибуты могут выполнять одну и ту же задачу, но при этом иметь разные названия и значения. Например, приведенный ранее атрибут CASCADE_HEIGHT есть в Vivado, но отсутствует в Quartus.

Где же искать список поддерживаемых атрибутов?

О том, какие атрибуты поддерживает среда проектирования нужно смотреть в документации.

Если вы используете среду Xilinx Vivado, то ваш основной помощник это User Guide 901 или гайд по синтезу. В этом гайде собрано много интересных и полезных советов и правил. Обязательно с ним познакомьтесь, если еще не сделали этого.

Если же вы используете среду Intel Quartus, том пригодится Quartus II Handbook. В конце статьи вы найдете необходимые ссылки.

Ну что, давайте опробуем атрибуты на практике: для VHDL и Verilog, для Vivado и Quartus

Практика использования атрибутов

Для демонстрации мы будем использовать параметризированный вариант кода описания сдвигового регистра. Мы задами длину задержки или длину цепочки

параметром SRL_LENGTH.

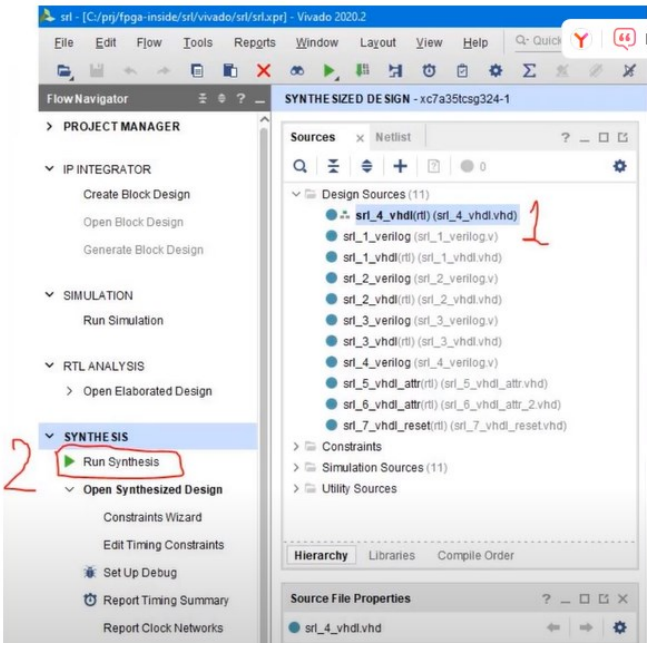
Мы рассмотрим две реализации: VHDL+Vivado и Verilog+Quartus. Однако исходники будут доступны и для связки VERILOG+Vivado и VHDL+Quatus. Каких-то принципиальных отличий вы не найдете, поскольку синтаксис атрибутов одинаковый, а отличаются они лишь в части названия атрибута и его значения. К тому же я оставил ссылки на дополнительную литературу в конце статьи, и вы всегда сможете самостоятельно пополнить свой багаж знаний и попрактиковаться.

Начнем с VHDL и Vivado.

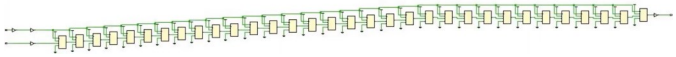
Код для параметризированного варианта для языка VHDL без атрибута будет выглядеть следующим образом.

```
library ieee;
use ieee.std_logic_1164.all;

entity srl_4_vhdl is
    generic (
        SRL_LENGTH : natural := 32
    );
    port (
        id      : in std_logic;
        iclk    : in std_logic;
        oq      : out std_logic);
end srl_4_vhdl;

architecture rtl of srl_4_vhdl is

    signal dff : std_logic_vector( SRL_LENGTH
- 1 downto 0 ) := (others => '0');

begin

    srl_4 : process (iclk) begin
        if rising_edge(iclk) then
            dff(0) <= id;
            for i in 0 to SRL_LENGTH - 2 loop
                dff(i + 1) <= dff(i);
            end loop;
        end if;
    end process srl_4;

    oq <= dff(SRL_LENGTH - 1);

end rtl;
```

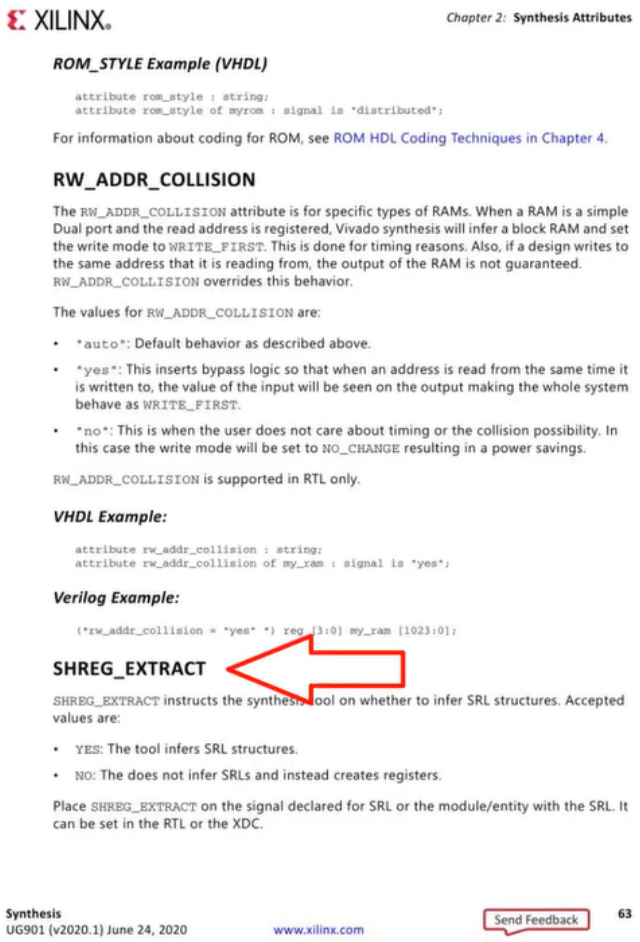
Давайте выполним синтез в вивадо и посмотрим на результат.



Как мы видим, получилась длинная цепочка, содержащая 32 триггера.



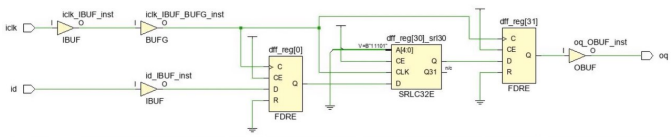
Теперь обратимся к UG901 и добавим атрибут SHREG_EXTRACT.



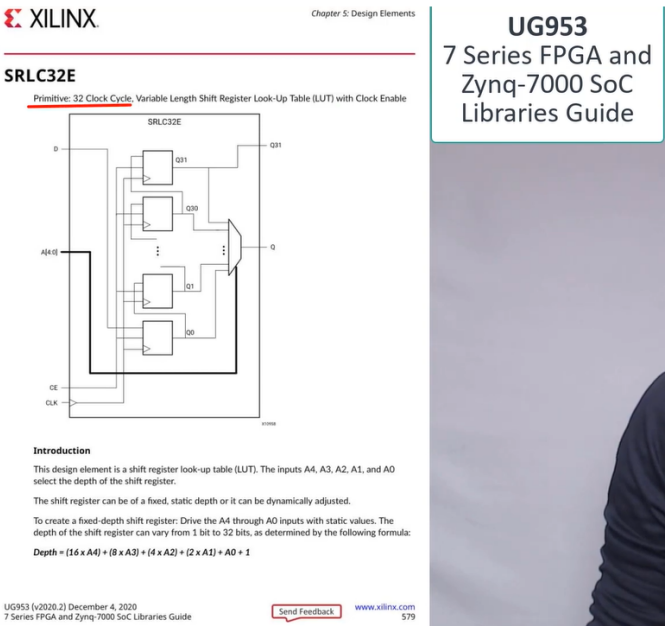
Данный атрибут позволит среде проектирования обнаружить цепочку регистров и реализовать ее как аппаратный ресурс. Добавим этот атрибут в наш код

```
library ieee;
use ieee.std_logic_1164.all;

entity srl_5_vhdl_attr is
    generic (
        SRL_LENGTH : natural := 32
    );
    port (
        id      : in std_logic;
        iclk    : in std_logic;
        oq      : out std_logic);
end srl_5_vhdl_attr;

architecture rtl of srl_5_vhdl_attr is

    signal dff : std_logic_vector( SRL_LENGTH
- 1 downto 0 ) := (others => '0');

    attribute shreg_extract : string;
    attribute shreg_extract of dff : signal
is "yes";

begin

    srl_5 : process (iclk) begin
        if rising_edge(iclk) then
            dff(0) <= id;
            for i in 0 to SRL_LENGTH - 2 loop
                dff(i + 1) <= dff(i);
            end loop;
        end if;
    end process srl_5;

    oq <= dff(SRL_LENGTH - 1);

end rtl;
```

После синтеза мы увидим, что регистры с 1 по 30 были объединены в один аппаратный блок. То есть вместо 32 триггеров у нас теперь всего 2 триггера с номерами 0 и 31 и один аппаратный сдвиговый регистр.



Вы наверняка зададите вопрос: «Почему остались регистры 0 и 31? Почему они не были объединены с остальными 30-ю триггерами? ведь ресурс SLR32 может обеспечить задержку на 32 такта».


XILINX
Chapter 5: Design Elements

SRLC32E

Primitive: 32 Clock Cycle, Variable Length Shift Register Look-Up Table (LUT) with Clock Enable

Introduction

This design element is a shift register look-up table (LUT). The inputs A4, A3, A2, A1, and A0 select the depth of the shift register.

The shift register can be of a fixed, static depth or it can be dynamically adjusted.

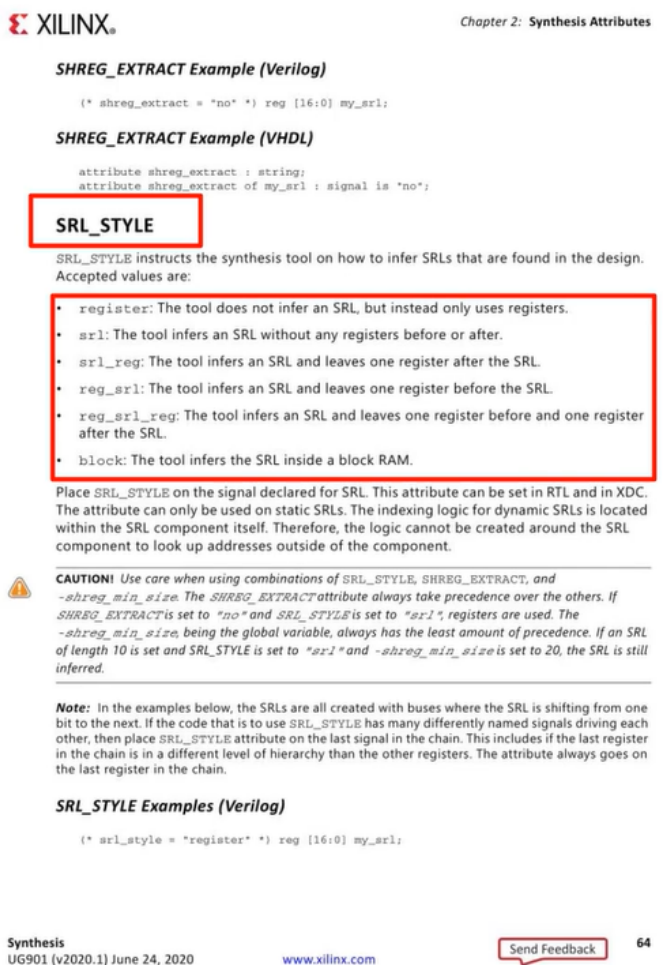
To create a fixed-depth shift register: Drive the A4 through A0 inputs with static values. The depth of the shift register can vary from 1 bit to 32 bits, as determined by the following formula:

Depth = (16 x A4) + (8 x A3) + (4 x A2) + (2 x A1) + A0 + 1

UG953 (v2020.2) December 4, 2020
7 Series FPGA and Zynq-7000 SoC Libraries Guide
Send Feedback
www.xilinx.com
579

UG953
7 Series FPGA and Zynq-7000 SoC Libraries Guide

Ответ тут кроется в настройках по умолчанию. Если вы проявите чуть больший интерес к UG901, то сможете найти атрибут SRL_STYLE, который отвечает за окончательный вид сдвиговых регистров. Данный атрибут имеет 6 доступных значений или 6 вариантов реализации.


XILINX
Chapter 2: Synthesis Attributes

SHREG_EXTRACT Example (Verilog)

```
(* shreg_extract = "no" *) reg [16:0] my_srl;
```

SHREG_EXTRACT Example (VHDL)

```
attribute shreg_extract : string;
attribute shreg_extract of my_srl : signal is "no";
```

SRL_STYLE

SRL_STYLE instructs the synthesis tool on how to infer SRLs that are found in the design. Accepted values are:

- register: The tool does not infer an SRL, but instead only uses registers.
- srl: The tool infers an SRL without any registers before or after.
- srl_reg: The tool infers an SRL and leaves one register after the SRL.
- reg_srl: The tool infers an SRL and leaves one register before the SRL.
- reg_srl_reg: The tool infers an SRL and leaves one register before and one register after the SRL.
- block: The tool infers the SRL inside a block RAM.

Place SRL_STYLE on the signal declared for SRL. This attribute can be set in RTL and in XDC. The attribute can only be used on static SRLs. The indexing logic for dynamic SRLs is located within the SRL component itself. Therefore, the logic cannot be created around the SRL component to look up addresses outside of the component.

CAUTION! *Use care when using combinations of SRL_STYLE, SHREG_EXTRACT, and -shreg_min_size. The SHREG_EXTRACT attribute always take precedence over the others. If SHREG_EXTRACT is set to "no" and SRL_STYLE is set to "srl", registers are used. The -shreg_min_size, being the global variable, always has the least amount of precedence. If an SRL of length 10 is set and SRL_STYLE is set to "srl" and -shreg_min_size is set to 20, the SRL is still inferred.*

Note: In the examples below, the SRLs are all created with buses where the SRL is shifting from one bit to the next. If the code that is to use SRL_STYLE has many differently named signals driving each other, then place SRL_STYLE attribute on the last signal in the chain. This includes if the last register in the chain is in a different level of hierarchy than the other registers. The attribute always goes on the last register in the chain.

SRL_STYLE Examples (Verilog)

```
(* srl_style = "register" *) reg [16:0] my_srl;
```

Synthesis
UG901 (v2020.1) June 24, 2020
www.xilinx.com
Send Feedback
64

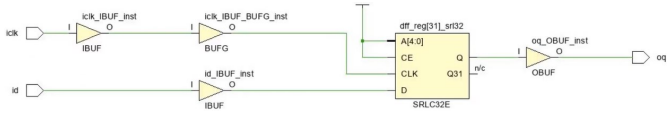
Давайте попробуем добавить еще 1

атрибут в наш код. Это будет атрибут SRL_STYLE со значением srl

```
library ieee;
use ieee.std_logic_1164.all;

entity srl_6_vhdl_attr_2 is
    generic (
        SRL_LENGTH : natural := 32
    );
    port (
        id      : in std_logic;
        iclk    : in std_logic;
        oq      : out std_logic);
end srl_6_vhdl_attr_2;

architecture rtl of srl_6_vhdl_attr_2 is

    signal dff : std_logic_vector( SRL_LENGTH
- 1 downto 0 ) := (others => '0');

    attribute shreg_extract : string;
    attribute shreg_extract of dff : signal
is "yes";

    attribute srl_style : string;
    attribute srl_style of dff : signal is
"srl";

begin

    srl_6 : process (iclk) begin
        if rising_edge(iclk) then
            dff(0) <= id;
            for i in 0 to SRL_LENGTH - 2 loop
                dff(i + 1) <= dff(i);
            end loop;
        end if;
    end process srl_6;

    oq <= dff(SRL_LENGTH - 1);

end rtl;
```

и посмотрим, что получится.

Как видите, все наши триггеры были упакованы в один сдвиговый регистр.



Чудеса, не так ли. Всего пара строк кода и мы сэкономили кучу ресурсов. Поиграйтесь с кодом, задайте разную длину сдвиговому регистру и поменяйте значения атрибутов. Посмотрите и сравните результаты. Посмотрите, что произойдет, если длина цепочки будет больше 32-х. Ну а мы движемся далее и на очереди у нас код на Verilog для пользователей Quartus

Для экспериментов с Intel взят кристалл серии Cyclone V, в котором реализация аппаратного сдвигового регистра сделана на блочной памяти. Однако, тут есть нюанс. Согласно документации преобразование из цепочки триггеров в аппаратный блок делается только при длине цепочки более 64,

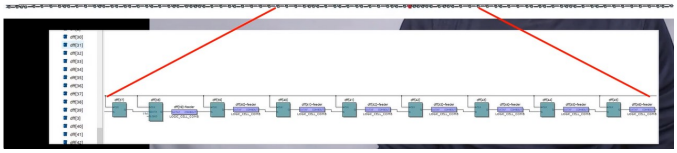
> MLAB memory. If the length of the register is less than 64 bits, the software implements the shift register in logic.

но поскольку у нас с вами имеется параметризированный верилог код для сдвигового регистра,

```
module srl_4_verilog(
    input id,
    input iclk,
    output oq
    );

    parameter SRL_LENGTH = 128;

    reg [SRL_LENGTH-1:0] dff;

    integer i;

    always @(posedge iclk) begin

      dff[0] <= id;

      for (i = 0; i < SRL_LENGTH-1; i = i+1)
begin
        dff[i + 1] <= dff[i];
      end

    end

    assign oq = dff[SRL_LENGTH-1];

endmodule
```

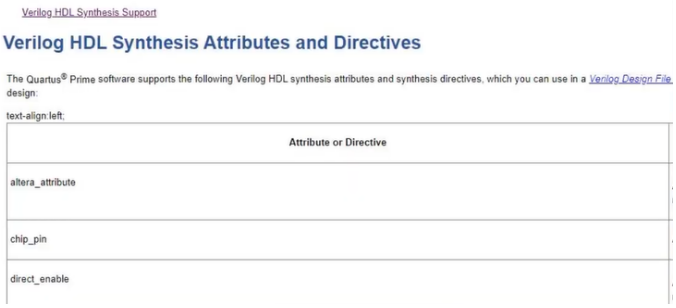
то мы просто изменим длину с 32 на 128 и запустим синтез.

Как мы видим, что по умолчанию имплементируется цепочка триггеров.



Правда, здесь пришлось пойти на небольшую хитрость, о которой я расскажу позднее.

Давайте теперь попробуем применить атрибут для упаковки нашей цепочки аппаратный примитив, который для Cyclone V делается на блочной памяти. Чтобы это осуществить мы воспользуемся атрибутом altera_attribute со значением «-name AUTO_SHIFT_REGISTER_RECOGNITION ON»


Verilog HDL Synthesis Support

Verilog HDL Synthesis Attributes and Directives

The Quartus® Prime software supports the following Verilog HDL synthesis attributes and synthesis directives, which you can use in a Verilog Design File design:

Attribute or Directive
altera_attribute
chip_pin
direct_enable

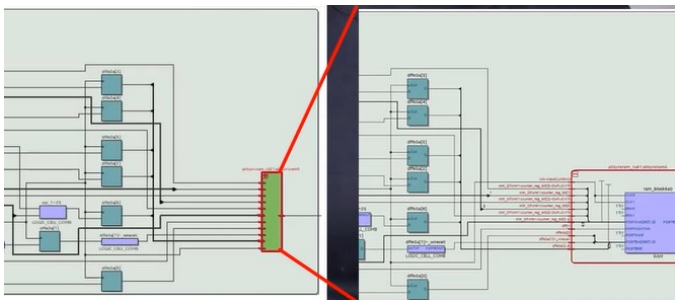
Данный атрибут мы пропишем перед объявлением нашего вектора reg_dff.

Код будет выглядеть следующим образом.

```
module srl_5_verilog_attr(
    input id,
    input iclk,
    output oq
    );
    parameter SRL_LENGTH = 128;

   (* altera_attribute = "-name AU-
TO_SHIFT_REGISTER_RECOGNITION ON" *)reg
[SRL_LENGTH-1:0] dff;

   integer i;

    always @(posedge iclk) begin
      dff[0] <= id;
      for (i = 0; i < SRL_LENGTH-1; i = i+1)
begin
        dff[i + 1] <= dff[i];
      end
    end

    assign oq = dff[SRL_LENGTH-1];

endmodule
```

Запустим синтез и посмотрим результат.

Как видим, часть нашей цепочки была упакована в блочную память.



Итак, первый эксперимент оказался вполне удачным, и мы смогли достичь поставленной цели по упаковке цепочки триггеров в аппаратный ресурс с помощью атрибутов в языке VHDL и Verilog. Ссылка на описания атрибутов для Vivado и Quartus вы найдете в конце статьи.

Ну а мы продолжаем. И теперь мы посмотрим, каким образом мы можем делать подобные процедуры и управлять синтезом с помощью настроек самой среды проектирования.

Управление настройками синтеза

И начнем мы с Xilinx Vivado.

Как я и говорил, што бы атрибуты корректно отработали, пришлось пойти на хитрость. А она в свою очередь заключалась в том, что я специально отключал глобальные настройки синтезатора в Vivado и Quartus, отвечающие за обнаружение и упаковку сдвиговых регистров. В Vivado за это отвечают несколько опций в настройках синтеза: это -no_srlextract и shreg_min_size.

Кстати, работа по оптимизации может также вестись и на этапе имплементации. Подробнее познакомиться с опциями по оптимизации проекта во время синтеза и имплементации вы можете в руководстве

UG835 Tcl Command Reference Guide. Опции по работе со сдвиговыми регистрами вы сможете найти по ключевому слову shift register.

Запретить обнаружение цепочки триггеров мы можем с помощью опции -no_srlextract, установив напротив нее галочку.

В этом случае мы получим обычную цепочку из 32-ти двух триггеров.

Сняв галочку с опции -no_srlextract мы скажем Vivado, што разрешаем обнаружение и упаковку цепочек триггеров в аппаратные ресурсы, и после синтеза получим вполне ожидаемый результат в виде двух триггеров и SRL32 ресурса.

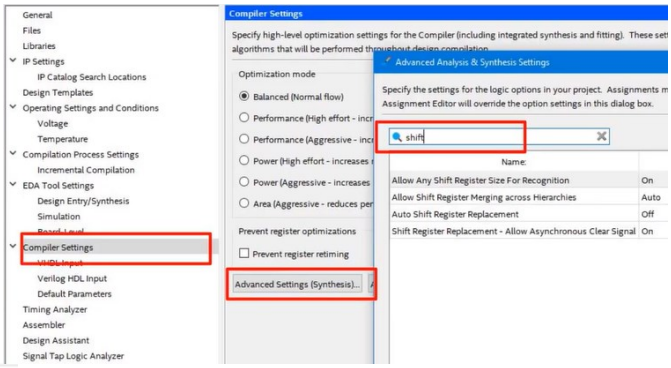
Втора опция shreg_min_size указывает при какой минимальной длине цепочки триггеров производить упаковку в аппаратный ресурс.

Если мы измени значение на 33, то увидим, как вивадо после синтеза вновь возвращает на цепочку из 32-х триггеров.

Следует обратить внимание на тот факт, что настройки синтеза являются глобальными и распространяются на все модули нашего проекта. Атрибуты же являются локальными и могут быть применены к конкретным сигналам, модулям, архитектурам и тд.

Переходим в Quartus

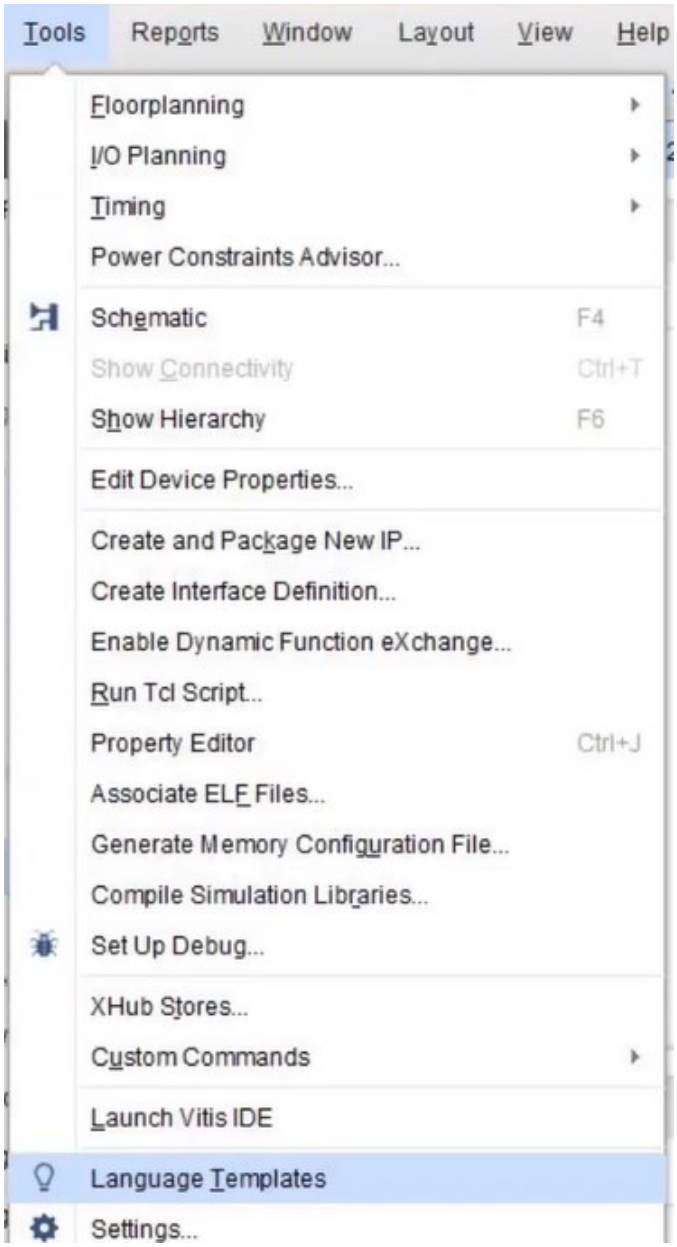
В квартус для управления упаковкой сдвиговых регистров имеется 3 атрибута. Управляя комбинацией этих настроек мы можем запретить или разрешить обнаружение и упаковку, комбинировать и объединять несколько сдвиговых регистров для экономии ресурсов.



Предлагаю вам самостоятельно поиграться с настройками и посмотреть, как их комбинации влияют на ваши проекты.

Сброс ломает всё

Поскольку мы работаем с аппаратными ресурсами, мы не всегда можем писать так как нам хочется и во многих случаях мы ограничены заготовленным функционалом аппаратного блока. Наиболее частая ситуация, которая не позволяет нам упаковать цепочки регистров в аппаратный ресурс возникает из-за банального сброса. Выполнение операции сброса не стоит путать с начальной инициализацией при объявлении переменной или сигнала. Инициализация в отличие от сброса не влияет на возможность упаковки цепочки триггеров в аппаратный ресурс. Поэтому при разработке поведенческих описаний, обязательно ознакомьтесь с шаблоном описания аппаратного ресурса. Такие шаблоны приводятся в документации – в UG901 для Vivado и Handbook для Quartus. Также в Vivado есть отдельная возможность вставки шаблонных описаний – вкладка называется language templates



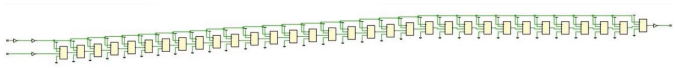
Давайте добавим сброс к нашему VHDL модулю с атрибутами и посмотрим, к каким последствиям это приведет.

Код перед вами, ни чего сложного, обычный сброс.

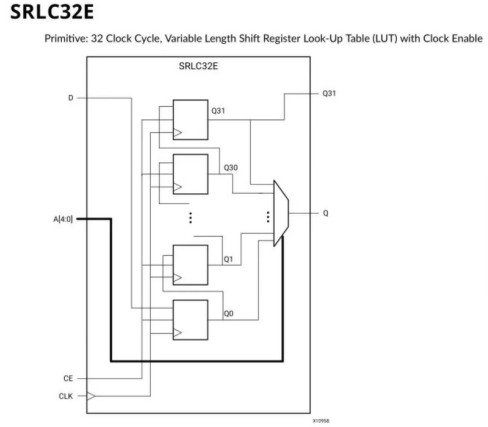
```
library ieee;
use ieee.std_logic_1164.all;

entity srl_7_vhdl_reset is
    generic (
        SRL_LENGTH : natural := 128
    );
    port (
        id      : in std_logic;
        iclk    : in std_logic;
        ireset  : in std_logic;
        oq      : out std_logic);
end srl_7_vhdl_reset;

architecture rtl of srl_7_vhdl_reset is
    signal dff : std_logic_vector( SRL_LENGTH - 1 downto 0 ) := (others => '0');

    attribute shreg_extract : string;
    attribute shreg_extract of dff : signal is "yes";
    attribute srl_style : string;
    attribute srl_style of dff : signal is "srl";

begin

    srl_7 : process (iclk) begin
        if rising_edge(iclk) then
            if ireset = '1' then
                dff <= (others => '0');
            else
                dff(0) <= id;
                for i in 0 to SRL_LENGTH - 2 loop
                    dff(i + 1) <= dff(i);
                end loop;
            end if;
        end if;
    end process srl_7;

    oq <= dff(SRL_LENGTH - 1);

end rtl;
```

Запустим синтез и проверим результат.

Как вы можете наблюдать, даже при установленных атрибутах и настройках синтеза мы получили не аппаратный блок, а цепочку регистров



и если вы обратитесь к руководству UG474 или UG953, то увидите, что примитив SRL не имеет входа сброса, поэтому Vivado и не может упаковать наш код в примитив SRL32.



Установив значение при сбросе не в 0 а в 1, мы получим такой же результат. Движемся далее

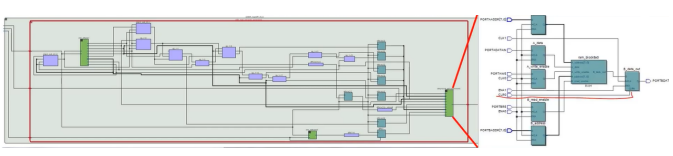
Давайте посмотрим как на ввод сигнала сброса отреагирует Quartus

Дополним наш код сигналом сброса.

```
module srl_7_verilog_reset(
    input id,
    input iclk,
   input ireset,
    output oq
    );
    parameter SRL_LENGTH = 128;

   (* altera_attribute = "-name AU-
TO_SHIFT_REGISTER_RECOGNITION ON" *)reg
[SRL_LENGTH-1:0] dff;

   integer i;

    always @(posedge iclk or posedge ireset)
begin
      if (ireset) begin
      dff <= {SRL_LENGTH{1'b1}};
    end else begin
      dff[0] <= id;
      for (i = 0; i < SRL_LENGTH-1; i = i+1)
begin
        dff[i + 1] <= dff[i];
      end
    end
    end

    assign oq = dff[SRL_LENGTH-1];

endmodule
```

Для начала посмотрим, что будет если мы будем по сигналу сброса обнулять наш сдвиговый регистр

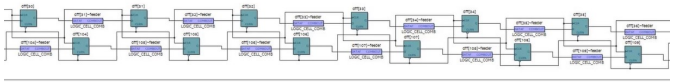
Как ни странно, но мы получили то же, что и в прошлый раз, но если мы развернем схему, то увидим, что в памяти сбрасывается только выходной триггер и это несколько странно, поскольку нам нужно было очистить весь сдвиговый регистр, а не последний элемент.



Давайте попробуем теперь заменить очистку регистра при сбросе на установку всех элементов в 1.

```
always @(posedge iclk or posedge ireset) begin
  if (ireset) begin
     dff <= {SRL_LENGTH{1'b1}};
  end else begin
     dff[0] <= id;
     for (i = 0; i < SRL_LENGTH-1; i = i+1) begin
        dff[i + 1] <= dff[i];
     end
  end
end
```

И тут мы получили опять цепочку триггеров, что несколько странно.



Если вы знаете, чем объясняется такое различие, напишите об этом комментариях.

Любопытно, что при реализации сдвигового регистра со сбросом на языке VHDL мы получим идентичные результаты – цепочку триггеров для обоих случаев.

Подытожим результаты: введение дополнительных управляющих сигналов или следование одному стилю кодирования не всегда оказывается правильным, поскольку в этом случае при разработке вы можете израсходовать значительно больше ресурсов, нежели при применении специально выделенных ресурсов. При их использовании старайтесь следовать шаблонным описаниям, которые обычно приводятся в документации на конкретное семейство ПЛИС, чтобы синтезатор правильно воспринимал ваш код и реализовывал описанные языковые конструкции в правильный набор аппаратных ресурсов.

Практические кейсы

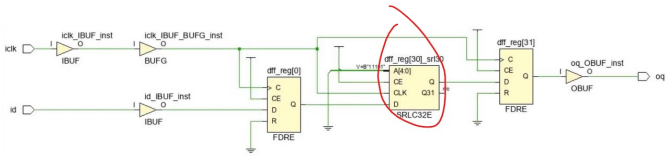
В заключении предлагаю разобрать несколько практических кейсов, с которыми на своей практике сталкивались или еще столкнутся большинство FPGA разработчиков

Случай 1. Когда вы пытаетесь реализовать что-то более сложное, чем моргание светодиодом на достаточно

высоких частотах, ну скажем 150-300 МГц, вы непременно столкнетесь с проблемой сведения таймингов. И как правило для того чтобы их сводить, вам придется оптимизировать ваш код и добавлять триггеры. Это вполне нормальная практика.

Но вот беда, вы начинаете добавлять триггеры, один , второй, десятый, и ни чего не меняется, слак по сетапу остается красным. У вас постепенно начинает подгорать и вы не можете понять в чем собственно дело.

А дело та оказывается в том, что ваша умная среда проектирования берет и ваши триггеры упаковывает в аппаратный сдвиговый регистр, что приводит лишь к тому, что у вас увеличивается латентность в тактах, но не меняется длина цепей.

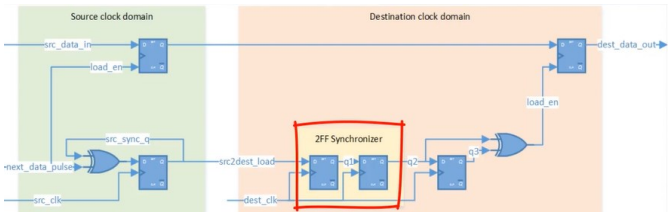


Ставь палец вверх, если у тебя такое было на практике.

Решить проблему можно простым введением атрибутов, с которыми мы сегодня познакомились и запретить упаковку цепочки триггеров в аппаратный ресурс

Случай 2. Часто в проектах мы сталкиваемся с пересечением тактовых доменов или необходимостью пересинхронизации асинхронных сигналов.

Для этого мы как правило используем широко известный прием, заключающийся в том, что мы ставим несколько триггеров подряд в приемном тактовом домене, обычно 2-3 триггера, что бы снизить вероятность появления метастабильных состояний.



Помимо введения триггеров, на цепи, их соединяющие надо дополнительно наложить специальные временные ограничения, типа false_path, max_delay и тд. Нередко и для самих триггеров надо прописать специальный атрибут, например ASYNC_REG в случае Xilinx

Я думаю вы понимаете, к чему я веду – у нас есть подряд несколько стоящих триггеров и умная среда проектирования может решить в благих целях сэкономить нам немножко ресурсов, упаковав их в аппаратный сдвиговый регистр. Такого конечно же желательно не допускать. Поэтому следует перестраховаться и ввести запрет какие либо манипуляции с нашей цепочкой триггеров в цепи пересинхронизации.

Подведем итоги

Разработка на ПЛИС не ограничивается написанием кода. Да, когда проект простой, можно следовать принципу «чик-чик и в продакшн», но на практике так лучше не поступать. Сложные проекты требуют от разработчика знаний не только языка проектирования, но и знаний архитектуры ПЛИС, настроек среды, в частности синтезатора, возможностей аппаратных ресурсов и много много другого. Сегодня, используя атрибуты для управления синтезатором, мы только приоткрыли дверь в мир проектирования на ПЛИС. Впереди еще много новых тем, про которые не принято говорить в статьях для начинающих.

Надеюсь сегодняшняя тема оказалась для вас действительно полезной и вы сможете использовать полученные знания на практике, по новому взгляните на ваши

проекты и уделите внимание тем нюансам проектирования, которые мы обсудили в рамках этой короткой.

Поддержите автора вашим царским лайком и оставьте в комментарии идеи для новых статей.

До встречи в следующей статье из серии «Практическое проектирование на ПЛИС или то о чем не принято говорить в статьях для начинающих»

Источники

1. Для пользователей Xilinx Vivado

1.1 UG901 Vivado Design Suite Synthesis Guide https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug901-vivado-synthesis.pdf

1.2 Vivado Design SuiteTcl Command Reference Guide https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug835-vivado-tcl-commands.pdf

1.3 Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug953-vivado-7series-libraries.pdf

2. Для пользователей Intel Quartus II

2.1 Intel Quartus Prime Software User Guides https://www.intel.com/content/www/us/en/programmable/products/design-software/fpga-design/quartus-prime/user-guides.html

2.2 Verilog HDL Synthesis Attributes and Directives https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/hdl/vlog/vlog_file_dir.htm

2.3 Quartus II Handbook Volume 1: Design and Synthesis https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/qts_qii5v1.pdf

3. Git репозиторий с исходниками видео https://github.com/FPGA-Systems/fpga_inside

4. Ссылка на стрим с 4 вариантами реализации SPI на сдвиговом регистре, конечном автомате, счетчике, памяти

4.1. ПЛИС и АЦП :: 1000 и 1 способ реализации SPI :: Часть 2 https://youtu.be/qVFO6D9Hj94

4.2 Интерфейсы :: SPI :: Часть 1:: Общие сведения https://youtu.be/s7kpZYsXM68?list=PLWMg96mLREOd-7U9LxkfNy1PgSiX1Moyj

4.3 Интерфейсы :: SPI :: Часть 2 :: Варианты реализации https://youtu.be/pGYkpPjy1Oc?list=PLWMg96mLREOd-7U9LxkfNy1PgSiX1Moyj

5. Статья в видеоформате

DRFM на основе ПЛИС Virtex-7 для тестирования радиолокаторов с синтезированной апертурой антенны

Сонин А.П. *dsplab@mail.ru*

Хромцев А.В.

Свирин Д.М. *chiffasvirin@yandex.ru*

Обсуждение и комментарии: link

Аннотация

В настоящей статье рассмотрена технология тестирования радиолокаторов с синтезированной апертурой антенны (РСА) при помощи цифровой радиочастотной памяти (Digital Radio-Frequency Memory - DRFM). Описан макет РСА, построенный на основе модульного оборудования компании Keysight Technologies, работающего под управлением программного пакета MATLAB. Рассмотрены структурная и функциональная схемы макета РСА, перечислены режимы его функционирования. Описано устройство DRFM, изготовленное на основе VPX модуля с ПЛИС Virtex-7, формирующее на радиолокационном изображении (РЛИ) несколько произвольно расположенных точечных имитационных отметок. Рассмотрена структура данной DRFM. Представлены результаты тестирования макета РСА при помощи разработанного устройства DRFM.

Введение

В настоящее время радиолокаторы с синтезированной апертурой антенны (РСА) авиационного и космического базирования широко используются для землеобзора в различных сферах человеческой деятельности [1...6]. При разработке, отладке и испытаниях таких радиолокаторов требуется многократное проведение радиолокационной съёмки различных типов участков местности с неподвижными и движущимися объектами в различных режимах. Проведение лётных экспериментов для отработки каждого типа режима съёмки является дорогостоящей, а иногда и нереализуемой задачей. Поэтому экономически более выгодными являются наземные методы исследования потенциальных характеристик и тестирования РСА в лабораторных, ангарных или полигонных условиях, без организации полётов и привлечения носителей РСА. Такими методами являются численное моделирование и макетирование [7...16].

При моделировании расчёт отсчётов зондирующего радиосигнала, цифровой радиоголограммы (ЦРГ) и синтез радиолокационного изображения (РЛИ) производятся на электронно-вычислительной машине (ЭВМ) в какой-либо программной

среде (например, в MATLAB) [7...16]. Для этого в данной программной среде создаются соответствующие модели:

- формирователя зондирующего радиосигнала;
- антенной системы;
- отражающей подстилающей поверхности и объектов;
- приёмника РСА;
- алгоритма обработки ЦРГ и формирования РЛИ,

и проводится вычисление процессов излучения, распространения, отражения, приёма и обработки радиолокационного сигнала во времени и пространстве.

При макетировании (или «прототипировании») формирование зондирующего радиосигнала и приём отражённой радиоголограммы производятся макетом РСА, собранным из какого- либо специального модульного оборудования (например, производимого компаниями Keysight, National Instruments и др.), обработка ЦРГ и синтез РЛИ – вычислительной системой, а синтез ЦРГ – при помощи цифровой радиочастотной памяти (Digital Radio-Frequency Memory - DRFM), запоминающей зондирующий радиосигнал радиолокатора и синтезирующей из его копий ответный радиосигнал, якобы «отражённый» от подстилающей поверхности и различных движущихся и неподвижных объектов, для приёмника радиолокатора.

Основные схемы тестирования РСА при помощи DRFM приведены на рисунках 1...4. Различают тестирование неподвижных РСА, размещаемых например в лаборатории, безэховой камере, в ангаре или на полигоне, как показано на рисунках 1...3 (в этом случае DRFM должна формировать пространственную модуляцию имитируемого отражённого радиосигнала), и тестирование движущихся РСА, как изображено на рисунке 4, при котором в DRFM формируется лишь дополнительная модуляция, обусловленная относительным пространственным удалением имитируемых точечных отражателей относительно места размещения приёмо-передатчика DRFM.

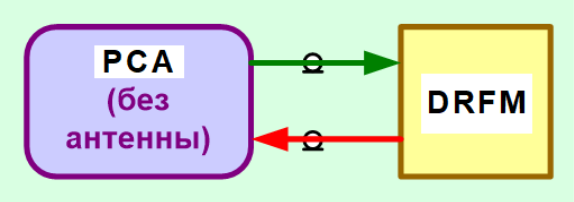


Рисунок 1 - Испытание РСА при помощи DRFM в лабораторных условиях без излучения радиосигналов

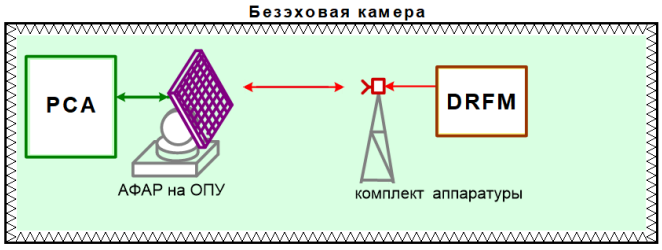


Рисунок 2 - Испытание РСА и его АФАР при помощи DRFM в безэховой камере

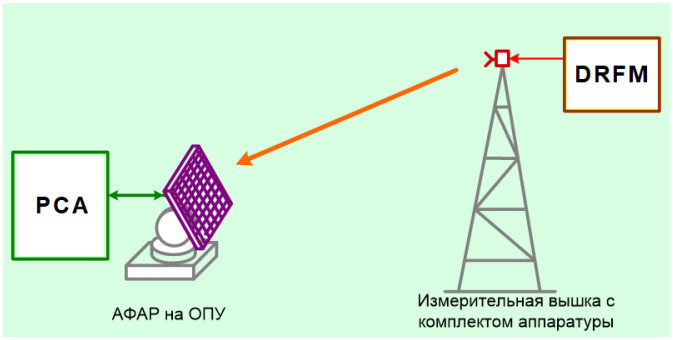


Рисунок 3 - Испытание РСА и АФАР при помощи DRFM на открытом полигоне

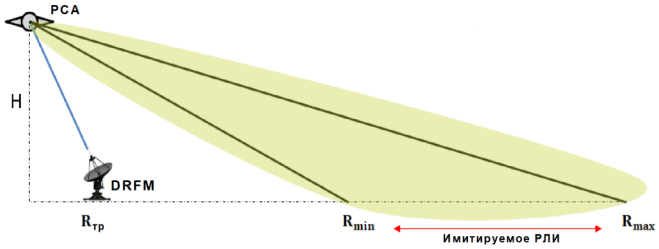


Рисунок 4 - Испытание РСА при помощи DRFM при проведении полёта

При испытаниях РСА при помощи DRFM в лабораторных условиях без излучения радиосигналов (рисунок 1) антенные системы РСА и приёмо-передатчика DRFM не используются, вход DRFM подключен к передатчику РСА, а выход DRFM – ко входу приёмника РСА.

В безэховой камере при помощи DRFM могут испытываться как весь РСА, так и отдельно его антенная система (АФАР), - рисунок 2. При этом антенны приёмо-передатчика DRFM могут находиться как в ближней, так и в дальней зонах антенны РСА, в зависимости от соотношений размеров антенны РСА, длины волны и размера безэховой камеры. В приёмо-передатчике DRFM в данном случае нет необходимости использовать узконаправленные антенны (типа АФАР или больших рупоров), и можно ограничиться применением широкоугольных антенн с малой площадью апертуры. При необходимости для обеспечения плоского фронта волны между антенной РСА и антеннами приёмо-передатчика DRFM может использоваться коллиматор. В альтернативном варианте (без использования коллиматора) АФАР РСА может испытываться по частям путём последовательного включения отдельных антенных секций или антенных блоков, составляющих общее антенное полотно.

При испытаниях РСА или АФАР при помощи DRFM на открытом полигоне (рисунок 3) может быть обеспечена дальняя зона для полной апертуры достаточно крупно габаритных АФАР РСА без использования коллиматора.

При проведении полётных испытаний РСА или его макета, размещённого на космическом или авиационном носителе, при помощи расположенного на Земле приёмо-передатчика DRFM (рисунок 4), в отличие от трёх предыдущих случаев, обеспечивается взаимное движение РСА и приёмо-передатчика, что должно учитываться при установке закона модуляции переизлучаемого устройством DRFM радиосигнала. В рассматриваемом режиме также должны быть обеспечены необходимый передающий энергопотенциал передатчика DRFM, и пространственная селекция зондирующего радиосигнала тестируемого радиолокатора (на фоне других мешающих источников радиоизлучений), для чего в качестве антенн в приёмо-передатчике DRFM целесообразно использовать АФАР, либо узконаправленные антенны.

В настоящей статье описывается DRFM, формирующая на РЛИ, выдаваемом неподвижным РСА, три точечных имитационных отметки в произвольно заданных точках кадра.

1. Тестируемый макет РСА

В ходе проведенных работ по отработке технологий макетирования и тестирования РСА была использована программно-аппаратная радиочастотная система, собранная из оборудования компании Keysight Technologies, работающего под управлением программного пакета MATLAB, в составе:

- 5-слотовое AXIe-шасси M9505A;
- генератор сигналов произвольной формы (Arbitrary Waveform Generator - AWG) M8190A;
- дигитайзер M9703A;
- PCIe-адаптер M9048A;
- вычислительный комплекс Forsite-6340 с четырьмя графическими ускорителями (GPU) NVIDIA Tesla K40c, использующими технологию «Compute Unified Device Architecture» (CUDA);
- осциллограф DSO90804A.

На базе данной системы был создан макет РСА, позволяющий:

- формировать сверхширокополосные (СШП) зондирующие радиосигналы различной, в том числе произвольной формы: фазо-кодо-модулированные (ФКМ), линейно-частотно-модулированные (ЛЧМ), нелинейно-частотно-модулированные (НлЧМ), частотно-кодо-модулированные (ЧКМ), шумоподобные, шумовые и другие;
- рассчитывать и формировать ЦРГ радиосигнала, отражённого от земной поверхности, с заданным рисунком радиолокационного контраста;
- принимать отражённые радиосигналы, несущие радиоголограмму изображения, сформированную при помощи DRFM, либо иного имитатора земной поверхности;
- производить первичную и вторичную цифровую обработку принятых сигналов в MATLAB, в том числе на CUDA GPU, аналогичную обработке сигнала, используемой в РСА, получая при этом РЛИ заданного рисунка радиолокационного контраста (либо РЛИ, формируемое DRFM).

Фото макета РСА, включая шасси M9505A с генератором M8190A и дигитайзером M9703A и вычислительный комплекс Forsite-6340, показано на рисунке 5. Структурная схема макета РСА изображена на рисунке 6.

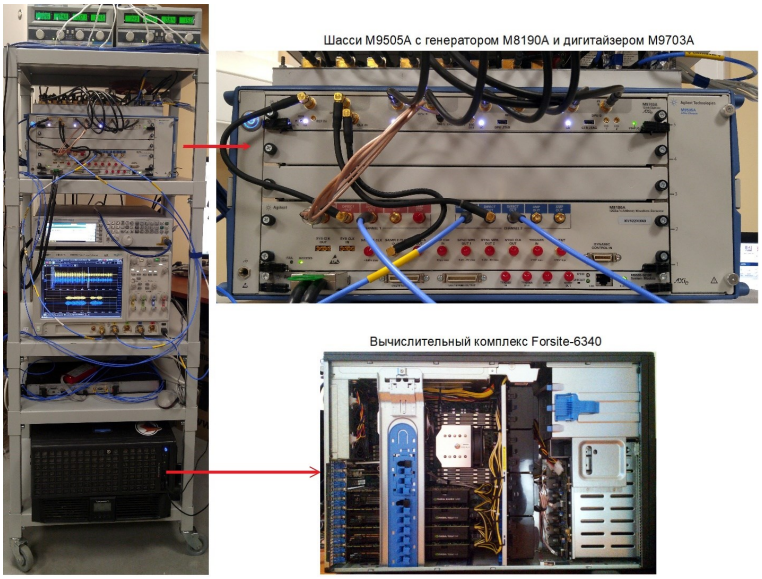


Рисунок 5 - Макет РСА, шасси M9505A с генератором M8190A и дигитайзером M9703A, вычислительный комплекс Forsite-6340

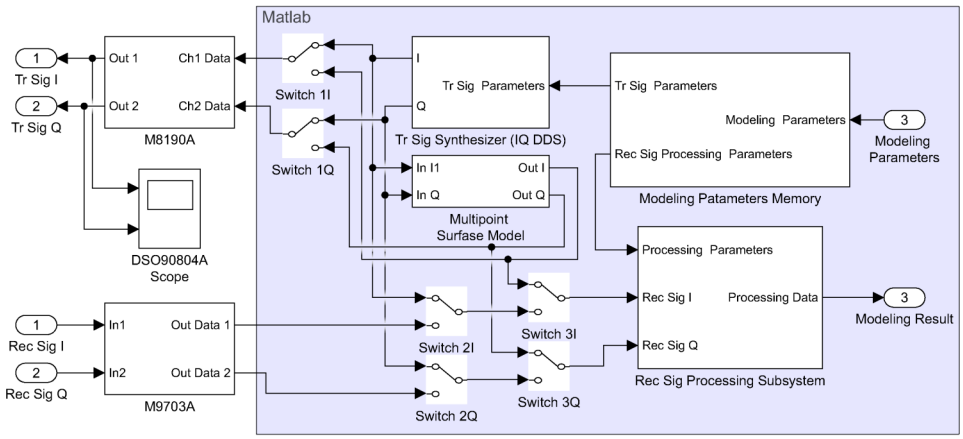


Рисунок 6 – Структурная схема макета РСА

Расчёт отсчётов зондирующего радиосигнала, обработка отсчётов принятого радиосигнала и управление параметрами работы макета осуществляются в программном пакете MATLAB, при помощи которого реализованы:

- память параметров моделирования (Modeling Patameters Memory);
- синтезатор зондирующего сигнала (Tr Sig Synthesizer), представляющий собой квадратурный (IQ) прямой цифровой синтезатор (DDS);
- многоточечная модель снимаемой поверхности (Multipoint Surface Model);
- подсистема обработки принятого сигнала (Rec Sig Processing Subsystem).

Сформированные в MATLAB отсчёты зондирующего радиосигнала РСА могут подаваться как на генератор сигналов произвольной формы M8190A и с его выходов в электрическом виде поступать на другое радиочастотное оборудование или излучаться в эфир, так и в математическом виде в многоточечную модель снимаемой поверхности, реализованную здесь же, в MATLAB, и представляющую собой по сути цифровой программный синтезатор многоточечной радиоголограммы. Принятый (отражённый от снимаемой поверхности) сигнал на вход подсистемы обработки принятого сигнала (Rec Sig Processing Subsystem) может поступать с трёх источников:

- с дигитайзера M9703A;
- с синтезатора зондирующего сигнала;
- с многоточечной модели снимаемой поверхности.

Выбор источника сигнала осуществляется программными переключателями Switch в MATLAB. Такая структура позволяет применять макет РСА в нескольких режимах работы, как с использованием приборов M8190A/M9703A и формированием/приёмом реальных электрических сигналов, так и без, оперируя с «виртуальными» сигналами только внутри MATLAB:

Режимы 1 и 2. Так, при включении в контур приборов M8190A/M9703A (ключи 1 в верхнем положении, 2 и 3 - в нижнем положении) макет РСА может быть использован непосредственно для радиолокационной съёмки (при наличии квадратурных преобразователей частоты, выходного усилителя мощности, приёмного малошумящего усилителя и приёмо-передающей антенны), либо совместно с имитатором отражённого от снимаемой поверхности радиосигнала, например, построенным на основе DRFM.

Режим 3. Если ключи 1 переключить в нижнее положение, то генератор M8190A будет воспроизводить отсчёты сигнала с выхода синтезатора радиоголограмм (Multipoint Surface Model). Дигитайзер так же, как и в предыдущих режимах, будет оцифровывать радиосигнал со входов Rec Sig, а подсистема обработки будет их обрабатывать. В данном режиме предполагается, что выходы Tr Sig соединены с соответствующими входами Rec Sig либо напрямую, либо через некоторую квадратурную цепь с небольшой задержкой. Данный режим может быть использован для тестирования исправности приборов M8190A/M9703A, либо некоторой квадратурной цепи, включенной между ними (например, DRFM или иного имитатора отраженного сигнала в режиме простой ретрансляции).

Режим 4. Приборы M8190A/M9703A не используются. На вход подсистемы обработки принятого сигнала поступают отсчёты зондирующего сигнала с выхода синтезатора зондирующего сигнала (ключи 2

– в верхнем положении, 3 – в нижнем). Макет РСА используется для контроля корректности расчёта отсчётов зондирующего сигнала и обработки принятого одноточечного сигнала. Подсистема обработки в данном режиме формирует одну размытую тестовую отметку на плоскости азимут-дальность.

Режим 5. Приборы M8190A/M9703A так же не используются. На вход подсистемы обработки принятого сигнала поступают отсчёты сигнала с выхода многоточечной модели поверхности (ключ 3 – в верхнем положении). Макет РСА применяется в режиме программного синтеза радиоголограмм заданных изображений и проверки алгоритмов подсистемы обработки принятого сигнала.

На рисунке 7 приведена функциональная схема формирования зондирующего и обработки принятого радиосигналов макета РСА для режима ScanSAR.

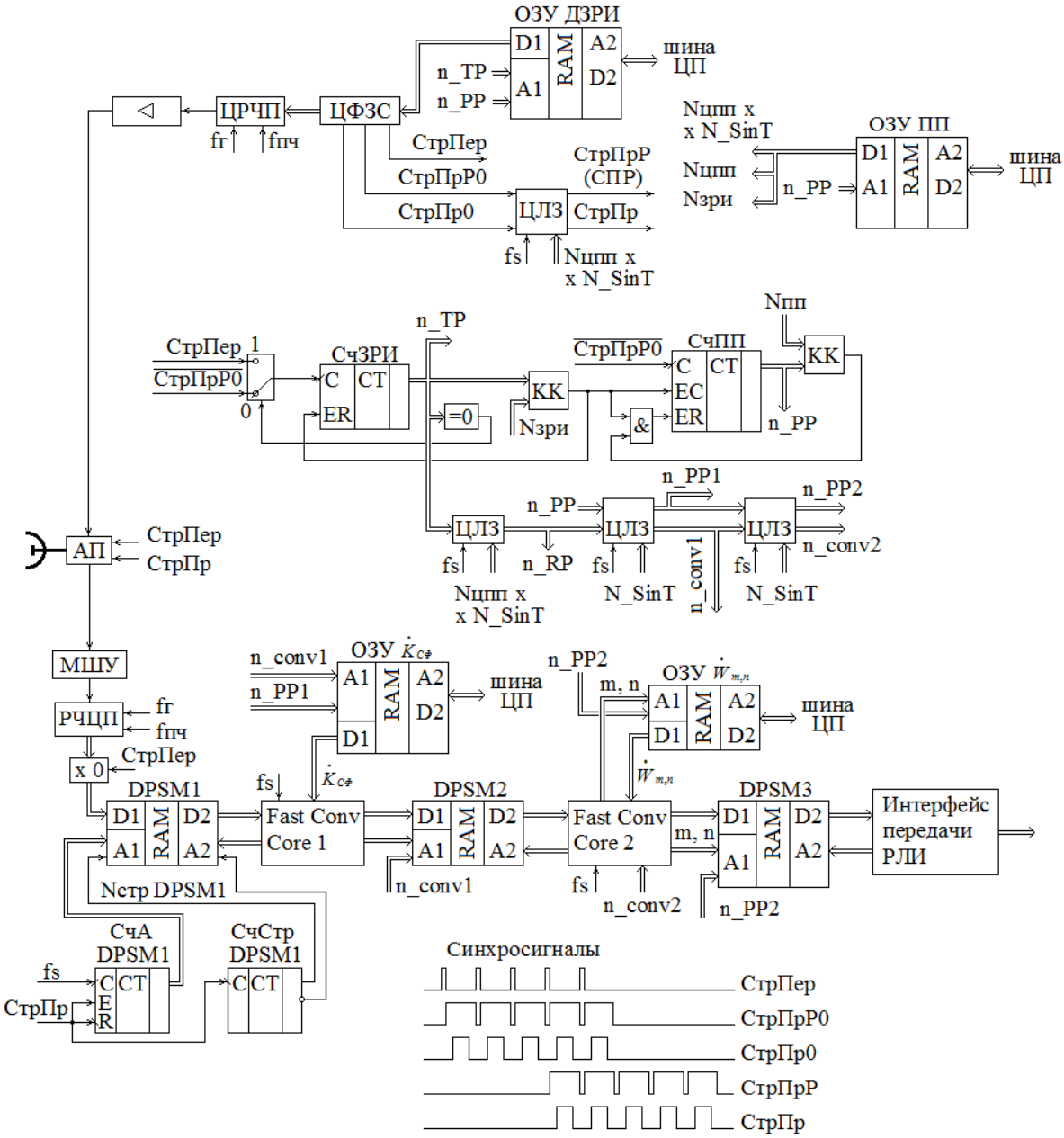


Рисунок 7 - Функциональная схема формирования зондирующего и обработки принятого радиосигналов макета РСА для режима ScanSAR

Зондирующий радиосигнал формируется в цифровом формирователе зондирующего сигнала (ЦФЗС). В приёмной части схема содержит два согласованных фильтра (СФ): дальностный СФ, осуществляющий временное сжатие принимаемых сложных радиоимпульсов, и азимутальный СФ, выполняющий синтез апертуры. Дальностный СФ реализован в IP-ядре быстрой свёртки Fast Conv Core 1, которое забирает отсчёты принятого радиосигнала из двухпортовой сигнальной памяти DPSM1 и записывает результат обработки в двухпортовую сигнальную память DPSM2, со второго порта которой отсчёты сжатого по дальности принятого сигнала подаются в азимутальный СФ, выполненный в IP-ядре Fast Conv Core 2. Данная схема, последовательно реализующая разделимые быстрые свёртки вдоль наклонной дальности и азимута представляет собой упрощённый Range-Doppler алгоритм без коррекции миграции по элементам дальности (Range Cell Migration Correction - RCMC), - ввиду грубого разрешения в режиме ScanSAR необходимость в RCMC отсутствует [5]. Результаты обработки азимутального СФ записываются в двухпортовую сигнальную память DPSM3, со второго порта которой через высокоскоростную радиолинию передаются на наземный пункт обработки. Параметры радиолокационной съёмки задаются центральным процессором РСА в оперативном запоминающем устройстве дескрипторов зондирующих радиоимпульсов (ОЗУ ДЗРИ) и ОЗУ парциальных полос (ПП).

Описанный макет РСА был использован для отработки методов и алгоритмов синтеза РЛИ. В ходе данной работы задавались различные точечные тестовые изображения, для которых синтезировались ЦРГ. Полученные радиоголограммы подавались на подсистему обработки, которая строила соответствующие радиолокационные изображения. Кроме этого, анализировались осциллограммы и спектры сигналов в различных точках схемы, приведенной на рисунке 7, что позволило оценить корректность работы РСА. Так, на рисунке 8 изображены зондирующий и отражённый сигналы для 3-точечного тестового кадра ScanSAR. На рисунке 9 показаны четыре следующих друг за другом зондирующих радиоимпульса (I- и Q- составляющие) с различными модулирующими М-последовательностями.

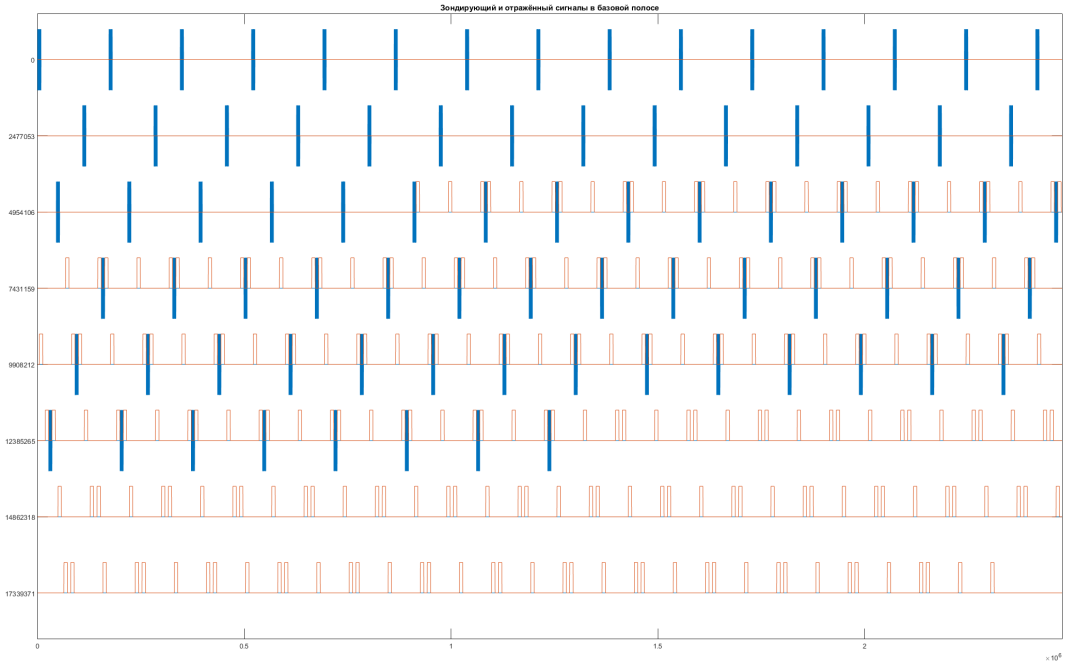


Рисунок 8 - Зондирующий и отражённый сигналы для 3-точечного тестового кадра

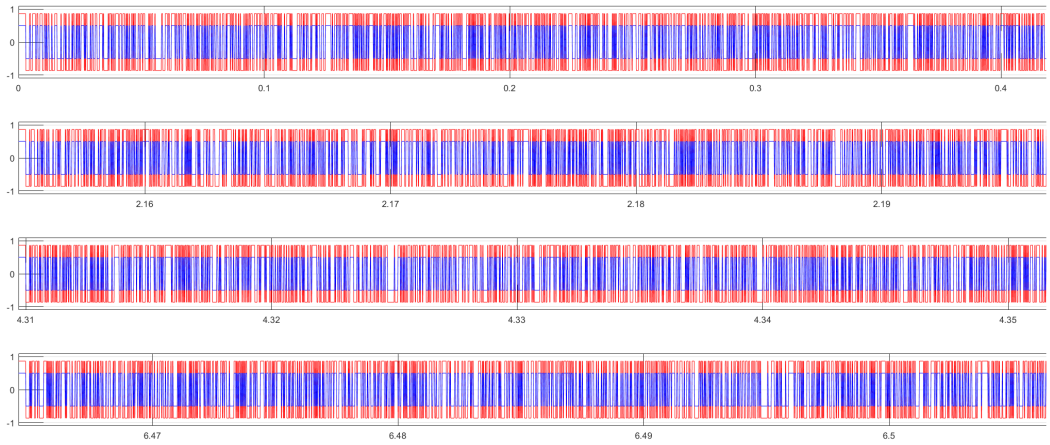


Рисунок 9 – Четыре следующих друг за другом зондирующих радиоимпульса (I- и Q- составляющие) с различными модулирующими М-последовательностями

Применение в зондирующем сигнале нескольких чередующихся модулирующих М-последовательностей позволило существенно снизить наложение на полезный отражённый радиосигнал паразитных радиосигналов, принятых по боковым и альтиметровому лепесткам диаграммы направленности (ДН) антенны РСА, тем самым повысив качество снимаемого РЛИ.

2. DRFM

Для имитации отражения зондирующего радиосигнала радиолокатора от подстилающей поверхности в классической радиолокации (без синтезирования апертуры) как правило используется DRFM, которая в общем случае применяется в широком спектре приложений [17-37]:

- в «Electronic Warfare» (EW) - и «Electronic Attack» (EA) - системах для формирования ответных радиосигналов;
- в «Electronic Support» (ES) - системах для приёма и запоминания радиосигналов с целью их последующего анализа;
- в AWG для генерации сигналов произвольной формы;
- в тренажёрах для обучения операторов радаров;
- в различного рода имитаторах фоно-целевой обстановки для испытания радаров;
- в самих радарах для их встроенного тестирования и проверки режимов работы;
- в составе радиолокационных целей;
- в различных системах для решения технических задач задержки сигнала по времени, сдвига по частоте, размножения и синтеза сложных сигналов.

DRFM преобразовывает вниз по частоте (в базовую полосу частот, «baseband») и оцифровывает поступающий на ее вход зондирующий радиосигнал РЛС, формируя цифровые квадратурные составляющие (I- и Q-), которые записываются через порт записи в двухпортовую сигнальную память. Через порт чтения данной памяти запомненный сигнал может быть многократно считан с варьируемой задержкой относительно записанного, наделен цифровыми способами частотной, фазовой и амплитудной модуляцией, восстановлен в аналоговую форму, преобразован по частоте в рабочий диапазон частот и переизлучён обратно в направлении РЛС. Поскольку переизлучённый DRFM радиосигнал представляет собой точную копию зондирующего радиосигнала, в приемнике

РЛС невозможно отличить переизлучённый радиосигнал от отраженного от реальных объектов (подстилающей поверхности, целей и т.д.). Таким образом, DRFM формирует на выходе приемника РЛС множество имитационных отметок, похожих на отметки от реальных точечных целей [17-37].

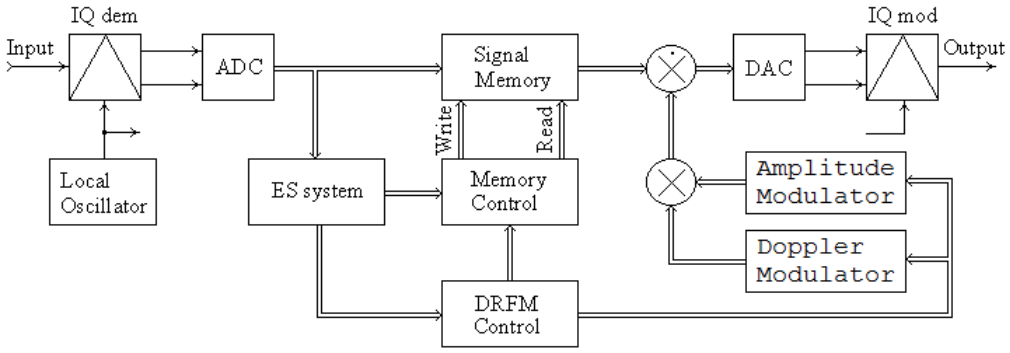
Упоминая о DRFM, нельзя не вспомнить об учителях, - первых отечественных её изобретателях, - Олеге Евгеньевиче Антонове [42 … 49] и Юрии Трофимовиче Карманове [20, 21, 25, 29, 31, 38…41], рисунок 10.

Рисунок 10 - д.т.н., директор ООО «Авиаконверсия» Олег Евгеньевич Антонов (слева), д.т.н., проф., директор НИИ ЦС ЧГТУ Юрий Трофимович Карманов (справа)

Д.т.н., О.Е. Антонов в 70-е годы прошлого века изобрёл первую отечественную DRFM, и, в поисках организации, способной её реализовать, обратился в НИИ Цифровых систем Челябинского Государственного Технического университета (НИИ ЦС ЧГТУ, в настоящее время – НИИ ЦС ЮУрГУ). Директор НИИ ЦС ЧГТУ д.т.н. Ю.Т.Карманов оценил перспективность предложенной темы, ухватился за её практическую реализацию и самостоятельно занялся дальнейшим её развитием. В результате в НИИ ЦС было развёрнуто направление разработки цифровых формирователей помех на основе принципа записи-воспроизведения сигналов, и, через несколько лет, были изготовлены цифровые блоки формирования помех для ряда типов бортовых авиационных станций помех, предназначенных для радиоэлектронного подавления (РЭП) радиолокаторов [29, 31, 38…41].

Олег Евгеньевич после развала СССР создал частную компанию, - ООО "Авиаконверсия", которая разработала и изготовила несколько типов станций активных помех для радиоэлектронного подавления не только радиолокаторов, но систем глобальной спутниковой навигации [42…49]. Изделия «Авиаконверсии» показали высокую эффективность в ходе их применения в Ираке [43, 44].

DRFM, изобретённая в 70-е годы XX века, использовалась для синтеза помеховых сигналов для обычных радаров без синтезирования апертуры антенны [17-37]. Структурная схема такой «классической» DRFM изображена на рисунке 11 [17-37].



Input – вход
IQ dem – квадратурный демодулятор
Local Oscillator – гетеродин
ADC – АЦП
ES System – система электронной поддержки
DSP – цифровой сигнальный процессор

Signal Memory – сигнальная память
Memory Control – управление памятью
DRFM Control – управление DRFM
DAC – ЦАП
IQ mod – квадратурный модулятор
Output – выход

Рисунок 11 – Структурная схема классической DRFM с постоянным в течение пачки зондирующих радиоимпульсов доплеровским сдвигом

Она содержит доплеровский модулятор («Doppler Modulator»), формирующий модулирующий сигнал с постоянным в течение пачки ретранслируемых радиоимпульсов доплеровским сдвигом. Однако, классическая DRFM для формирования ЦРГ для РСА в большинстве случаев непригодна по причине постоянства формируемых доплеровских сдвигов воспроизводимого радиосигнала и запоминания в сигнальной памяти лишь одного радиоимпульса.

При попытке использования классической DRFM (с постоянными в течение пачки доплеровскими сдвигами переизлучаемых радиоимпульсов) для формирования точечных отметок в РЛИ на выходе РСА, сформированные отметки расфокусируются по азимуту. Особенно ярко этот эффект проявляется при больших временах синтезирования апертуры. Так, на рисунке 12 показаны размещение имитируемых классической DRFM точечных отражателей (ТО) в тестовом кадре и соответствующее РЛИ, полученное на выходе неподвижного РСА, подключенного к выходу данной DRFM.

Время синтезирования в данном эксперименте составляло 172мс (800 зондирующих радиоимпульсов), разрешение одного пиксела РЛИ равно 20х20м, размер снимаемого парциального кадра (по углу места и азимуту) - 1497х113 пикселей, длина синтезированной апертуры – 1126м. Из рисунка видно, что при использовании классической DRFM сформированные имитационные отметки существенно размазаны по азимуту, - их азимутальная ширина много больше одного пикселя. Фактически они вырождаются в линии вдоль азимута. Аналогичная картина наблюдается и при попытке тестирования РСА путём простого подключения его выхода на вход (без DRFM), с той лишь разницей, что в этом случае формируется лишь одна размытая отметка.

Точечные имитационные отметки для движущегося РСА при помощи классической DRFM могут быть сформированы лишь в непосредственной близости за местом размещения DRFM. С увеличением относительной дальности имитируемых отметок они размазываются по азимуту.

Запоминание лишь одного радиоимпульса в классической DRFM приводит к невозможности формирования ЦРГ космическим РСА, в которых до приёма первого отражённого радиоимпульса пачки

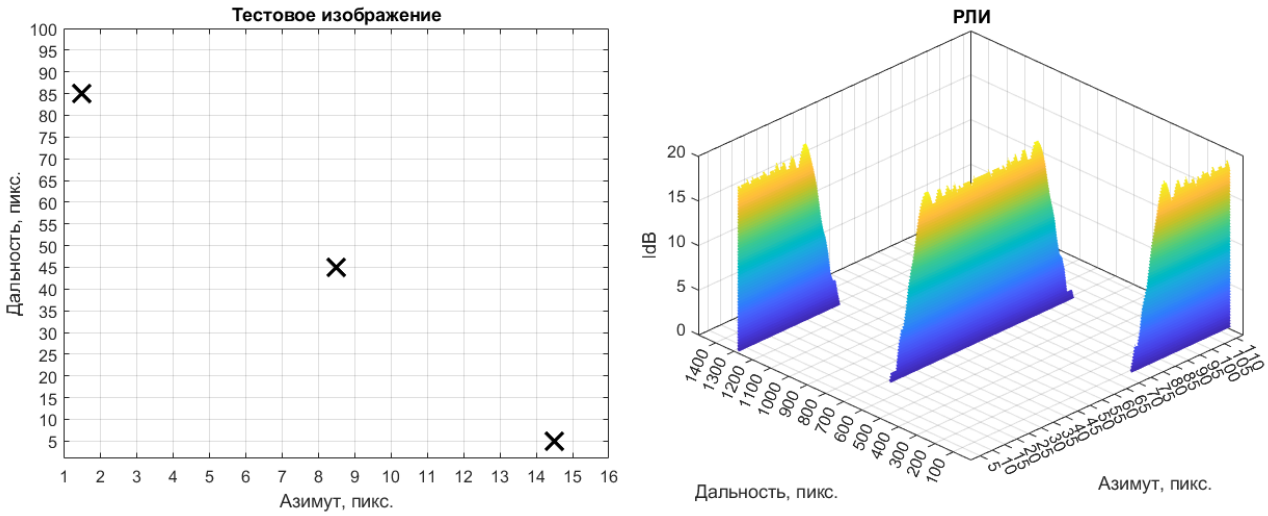


Рисунок 12 – Размещение имитируемых DRFM ТО (слева) и соответствующее РЛИ, синтезированное в РСА по ЦРГ, сформированной в DRFM (справа)

излучается несколько десятков зондирующих радиоимпульсов, которые должны быть запомнены для формирования ответной ЦРГ из их копий (см. рисунок 8).

По указанным причинам классическая DRFM не пригодна для тестирования РСА.

Однако, развитие техники не стоит на месте, и, начиная с 2000-х годов, были изобретены DRFM, специально предназначенные для формирования радиоголограмм отражённого радиосигнала для РСА [50...55]. Такие DRFM имеют существенные отличия относительно классической и могут быть разделены на два класса: формирующие на синтезируемом РЛИ небольшое количество точечных имитационных отметок путём записи-воспроизведения зондирующего радиосигнала и создающие заданные искусственные изображения (карты местности) путём свёртки на проходе, - рисунок 13.

Первый класс DRFM может быть использован при технологической оценке базовых характеристик РСА, таких, как:

- структура отклика двумерного согласованного фильтра (СФ) на одиночный ТО вдоль наклонной дальности и азимута при различных его положениях на РЛИ;
- уровень и положение боковых лепестков указанного отклика двумерного СФ на ТО;
- разрешающие способности РСА вдоль наклонной дальности и азимута.
- обнаружение ТО с малой эффективной площадью рассеяния (ЭПР) на фоне близкорасположенного яркого ТО (для оценки динамического диапазона РЛИ, формируемого при помощи РСА).

Для этих целей в DRFM достаточно формировать 1...3 точечных имитационных отметки.

Второй класс DRFM может формировать как искусственные изображения, так и большое количество точечных имитационных отметок (несколько десятков, сотен и более) и необходим для проверки как базовых, так и функциональных характеристик РСА, определяющих качество синтезируемого РЛИ при съёмке участков земной поверхности различных типов, а также применяется в EW-системах, предназначенных для борьбы с РСА, для формирования искусственных карт местности («Man-made Maps») [56...59].

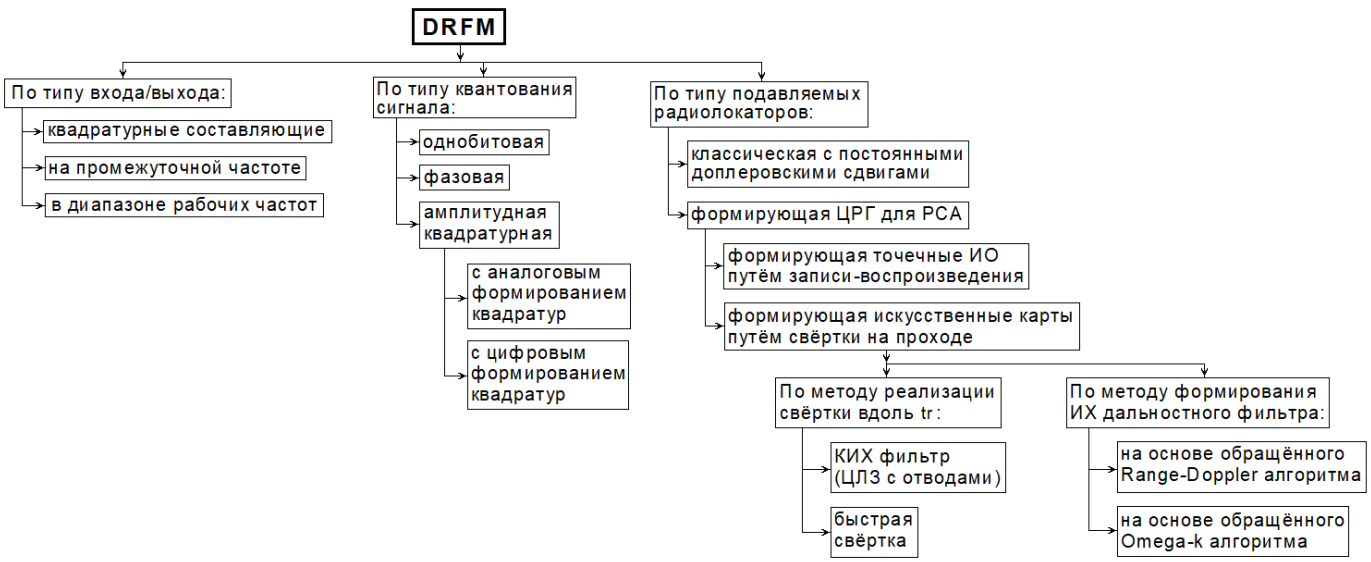


Рисунок 13 - Классификация DRFM

В настоящей статье рассматривается DRFM первого класса, предназначенная для тестирования РСА. Для решения данной задачи авторами настоящей статьи было создано устройство DRFM, формирующее точечные имитационные отметки для РСА. В качестве платформы цифровой обработки сигналов (ЦОС) были использованы VPX модуль SVP-721 с мезонинными субмодулями SFM-2A1000-2D1000 и SFM-4F10S, изготовленные компанией ЗАО «Скан Инжиниринг Телеком». Для преобразования входных радиосигналов из рабочего диапазона частот в базовую полосу и выходных сигналов из базовой полосы в рабочий диапазон частот был использован модуль преобразования частоты.

Фото VPX модуля SVP-721 приведено на рисунке 14, его структурная схема – на рисунке 15, характеристики указанного модуля – в таблице 1. Фото мезонинного FMC субмодуля SFM-2A1000-2D1000 представлено на рисунке 16, его структурная схема – на рисунке 17, характеристики данного субмодуля – в таблице 2. Фото VPX модуля SVP-721 в сборе с субмодулями SFM-2A1000-2D1000 и SFM-4F10S показано на рисунке 18. На рисунке 19 изображён модуль преобразователя частоты. Внешний вид устройства DRFM можно увидеть на рисунке 20.

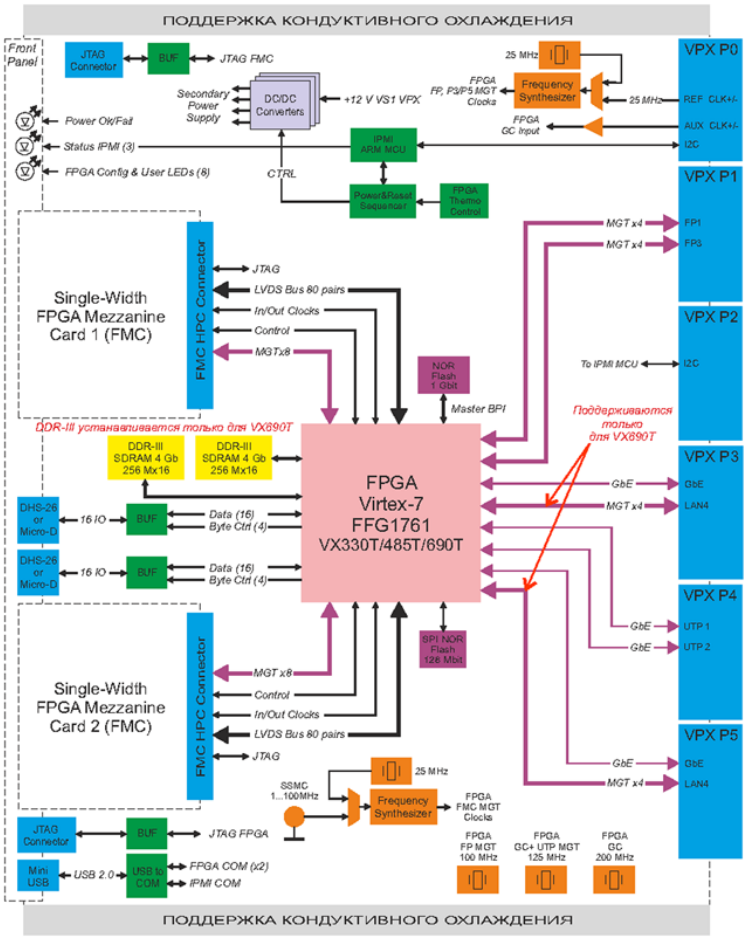


Рисунок 15 – Структурная схема VPX модуль SVP-721

Рисунок 14 - VPX модуль SVP-721

Таблица 1. Характеристики VPX модуля SVP-721

Форм-фактор	VPX 6U
Тип используемой FPGA	Xilinx Virtex-7 из ряда XC7VX330/485/690T
Тип используемой памяти	Два 16-ти разрядных банка динамического RAM DDR3 SDRAM
Поддержка системных интерфейсов	PCIe/SRIO/XAUI, Gigabit Ethernet (отдельно приобретаемые IP-ядра)

Рисунок 16 - Мезонинный FMC субмодуль SFM-2A1000-2D1000

Таблица 2. Характеристики мезонинного FMC субмодуля SFM-2A1000-2D1000

Число каналов АЦП	2
Частота дискретизации АЦП	1000МГц
Аналоговая полоса тракта АЦП	до 1400 МГц
Число разрядов АЦП	12
Число каналов ЦАП	2
Частота дискретизации ЦАП	1000МГц
Аналоговая полоса тракта ЦАП	до 500 МГц
Число разрядов ЦАП	16

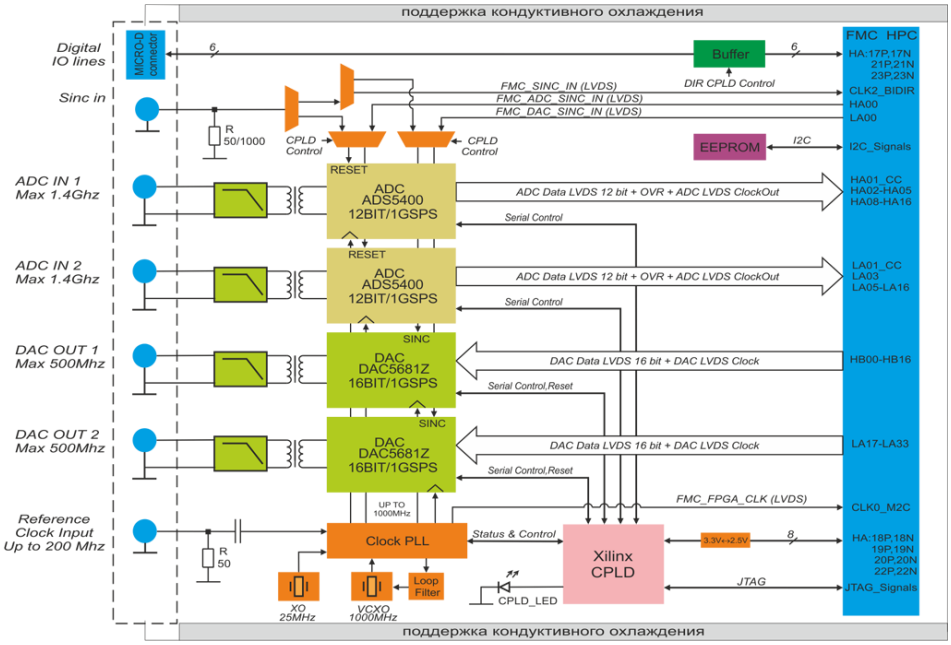


Рисунок 17 - Структурная схема мезонинного FMC субмодуля SFM-2A1000-2D1000

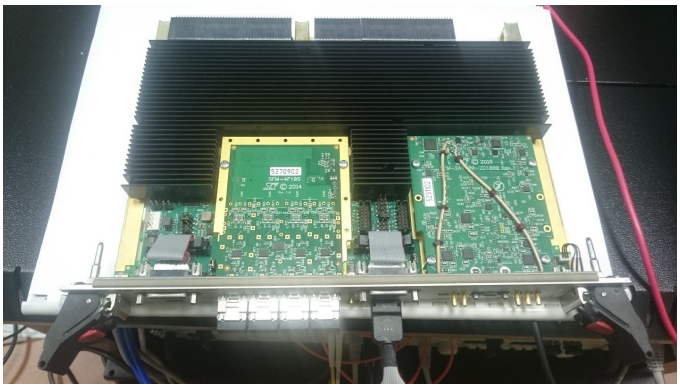


Рисунок 18 - VPX модуль SVP-721 в сборе с субмодулями SFM-2A1000-2D1000 и SFM-4F10S

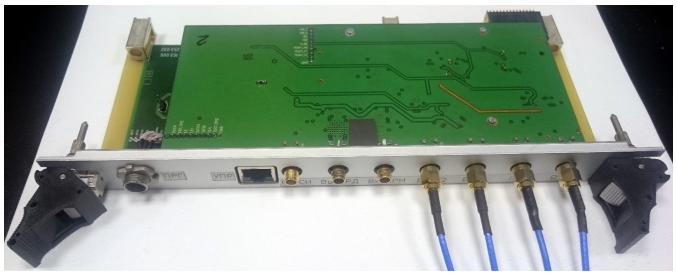


Рисунок 19 - Модуль преобразователя частоты

Рисунок 20 – Устройство DRFM

На рисунке 21 представлен вид программной оболочки Vivado, используемой для синтеза и отладки IP-ядра ПЛИС устройства DRFM. Окно программы управления устройством DRFM показано на рисунке 22. На рисунке 23 приведена структурная схема устройства DRFM, на рисунке 24 - схема его испытания совместно с макетом РСА, а на рисунке 25 - функциональная схема устройства. Основные характеристики устройства DRFM перечислены в таблице 3.

Таблица 3. Основные характеристики устройства DRFM

Диапазон рабочих частот (определяется модулем преобразования частоты)	X
Мгновенная полоса частот	до 1ГГц
Длительность запоминаемого радиосигнала	до 700мкс
Длительность обрабатываемой пачки зондирующих радиоимпульсов	17мс
Динамический диапазон ретранслируемых сигналов	не менее 60дБ
Уровень вносимых паразитных составляющих - с учётом модуля преобразования частоты - без учёта модуля преобразования частоты	 не более -35дБ не более -60дБ
Число имитируемых ТО	3
Размер кадра имитируемого тестового изображения	2250x25000м

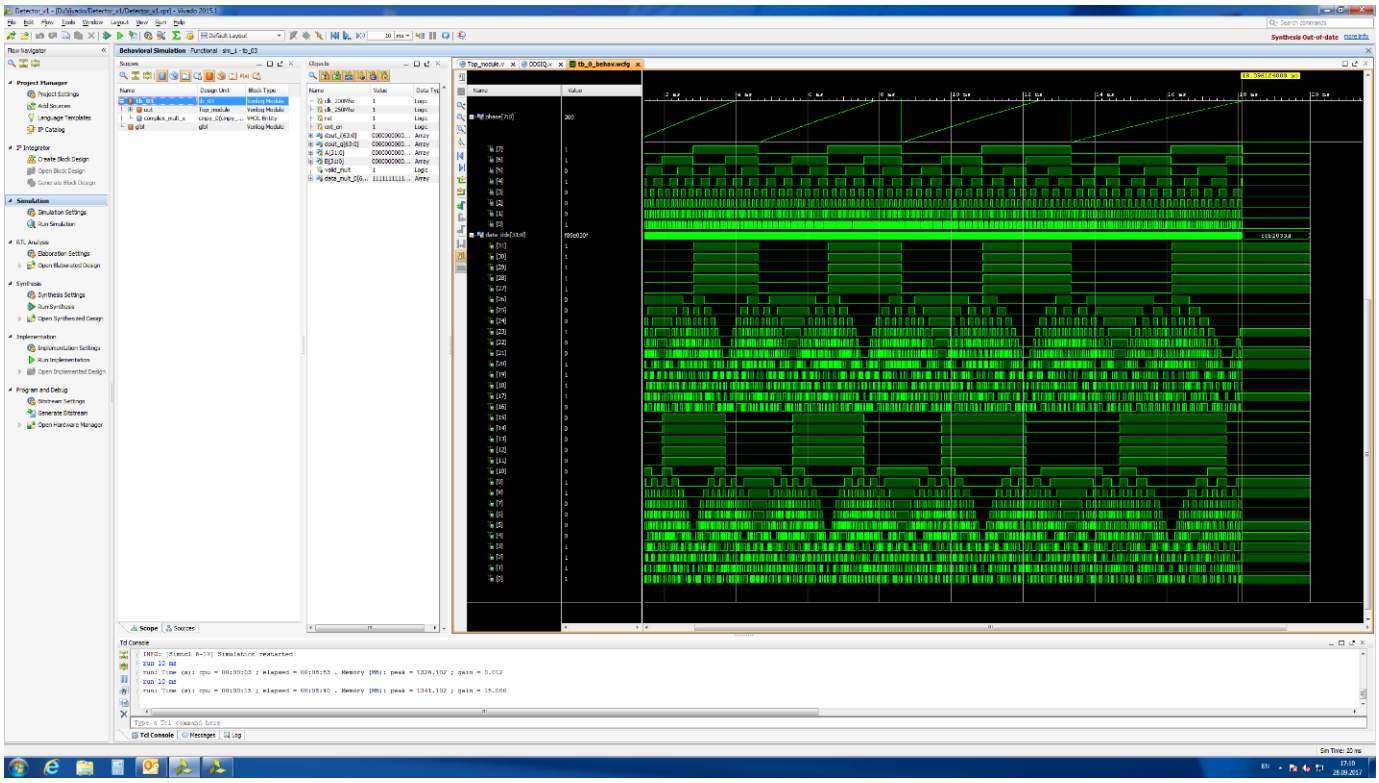


Рисунок 21 - Программная оболочка Vivado, используемая для синтеза и отладки IP-ядра ПЛИС устройства DRFM

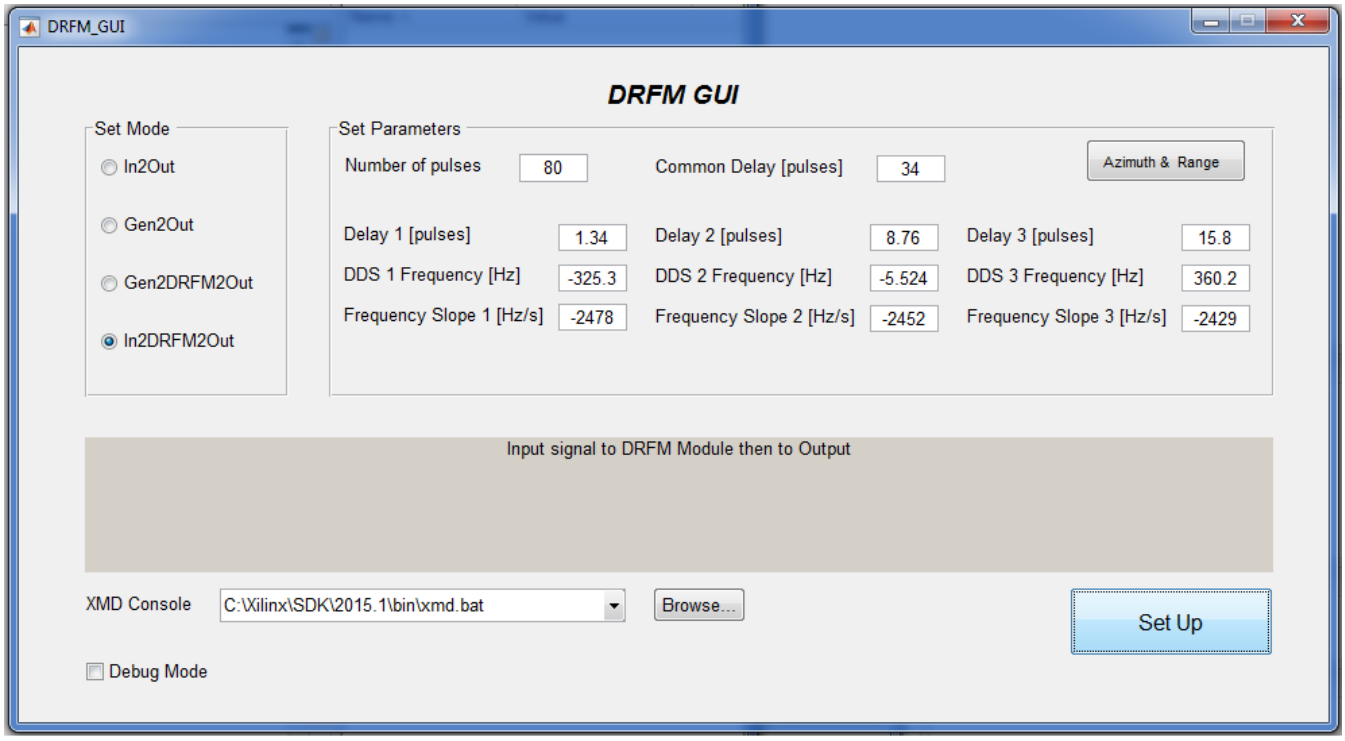


Рисунок 22 – Окно программы управления устройством DRFM

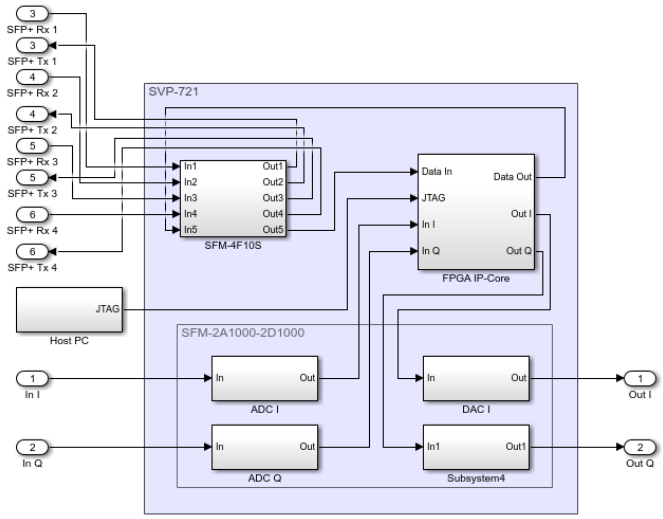


Рисунок 23 - Структурная схема устройства DRFM

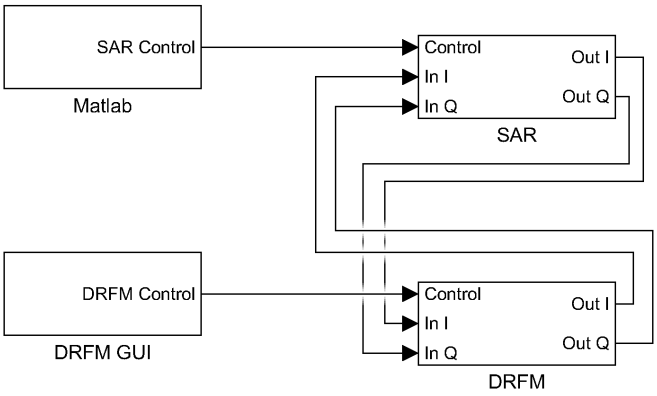


Рисунок 24 - Схема испытания устройства DRFM совместно с макетом РСА

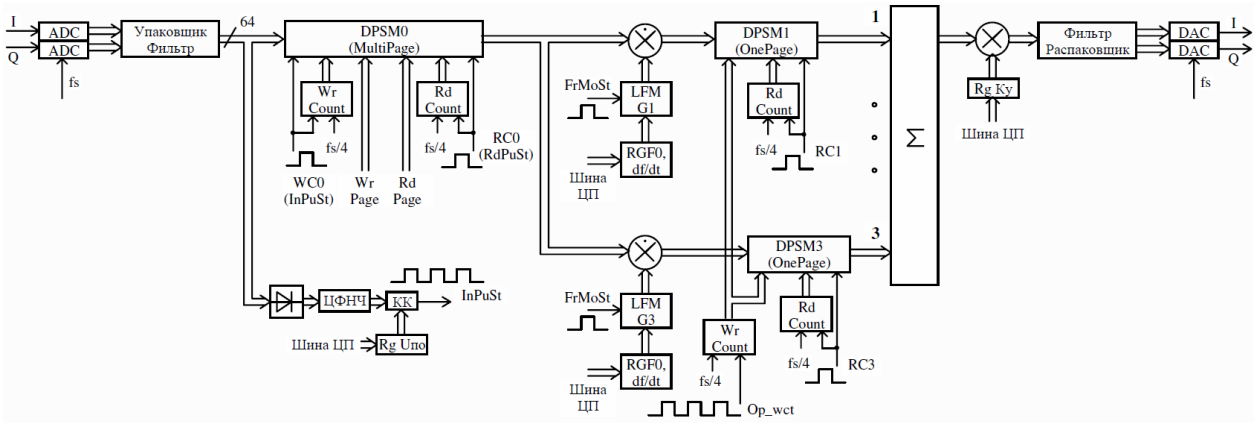


Рисунок 25 - Функциональная схема устройства DRFM

При синтезе функционального построения и IP-ядра разработанного устройства DRFM потребовалось решение ряда новых задач, обусловленных отличием принципов обработки принятых радиосигналов в РСА и в обычных радарах.

Во-первых, для РСА космического базирования, неоднозначно измеряющих дальность, ответный сигнал DRFM на временной оси должен формироваться исключительно в определённом приёмном стробе, соответствующем участку дальности, в котором расположен снимаемый кадр. Для этого приходится записывать в DRFM некоторое, достаточно большое количество радиоимпульсов, прежде чем на её выходе появится первый ответный радиоимпульс. Это приводит к необходимости хранения в сигнальной памяти указанного числа импульсов (в отличие от классической DRFM, запоминающей лишь один радиоимпульс). Данный фактор приводит к увеличению требуемого объёма сигнальной памяти DRFM и необходимости построения других, отличных от традиционных, схем синхронизации записи-считывания радиоимпульсов.

Во-вторых, потребовалось изобретение нового метода формирования имитационных отметок. Так, принцип формирования отражённого радиосигнала для РСА существенно отличается от соответствующего принципа для обычного радара. Если в обычной DRFM при формировании точечной имитационной отметки используется постоянный доплеровский сдвиг, неизменный в течение ответа на одно когерентное зондирование, то при формировании точечной отметки для РСА такой метод не пригоден: сформированная

отметка будет размазана по азимутальной координате кадра. Чтобы формируемые отметки были точечными, необходима реализация других, не постоянных, а изменяющихся в течение когерентного зондирования доплеровских сдвигов, которые бы обеспечивали на выходе азимутального СФ когерентное сложение всех участков принимаемого (сформированного DRFM) сигнала в течение всего интервала синтеза апертуры. Такие доплеровские сдвиги были найдены в ходе проведенных исследований, и для их формирования были разработаны соответствующие цифровые формирователи модулирующего сигнала, функциональная схема одного из которых представлена на рисунке 26.

В VPX модуле SVP-721 использовалась ПЛИС Xilinx Virtex-7 XC7VX690T. Проект ПЛИС разрабатывался в среде Vivado в редакторе проектов IP Integrator и IP Packager. Основной язык программирования - Verilog. Также применялись VHDL и Tcl.

Для реализации сигнальной памяти DRFM применялась блочная память ПЛИС BRAM. Деление всего объёма памяти на страницы, соответствующие отдельным радиоимпульсам, обеспечивалось путём запоминания моментов времени начал и концов записываемых радиоимпульсов в отдельный буфер памяти адресов типа FIFO.

Перед разработкой проекта ПЛИС сигнальный тракт DRFM был промоделирован в MATLAB с учётом целочисленной арифметики (рисунок 26), в результате чего были определены необходимые разрядности сигнальных шин во всех участках тракта.

Для управления всеми узлами схемы DRFM посредством программно-доступных регистров и прерываний использовался софт процессор MicroBlaze – рисунок 27. Данный процессор программировался на языке программирования C/C++ в среде разработки SDK из состава Vivado.

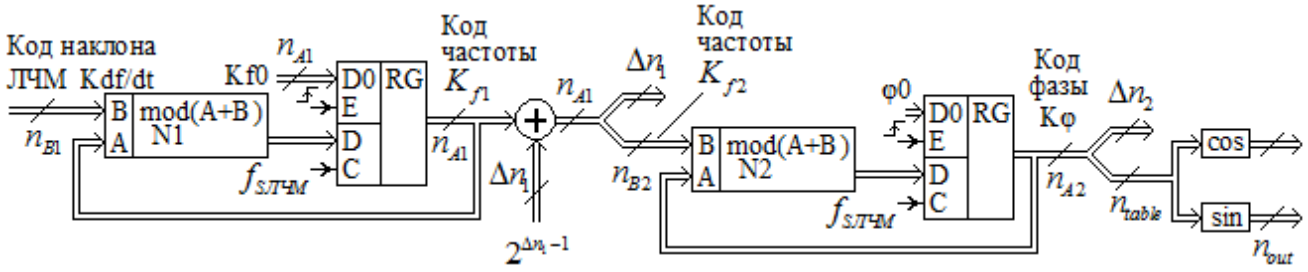


Рисунок 26 - Функциональная схема формирователя модулирующего сигнала

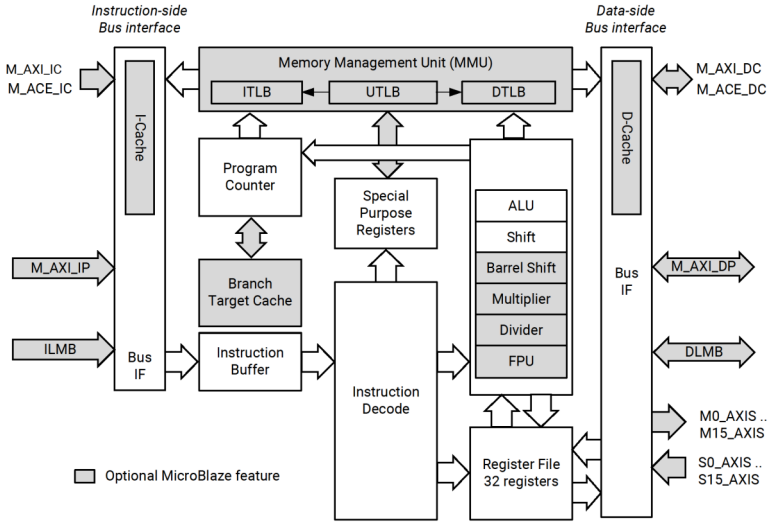


Рисунок 27 - Блок-схема ядра MicroBlaze

3. Тестирование макета РСА при помощи DRFM

Для радиолокационной съёмки поверхности Земли при помощи РСА используют различные разновидности трёх базовых режимов (рисунок 28): a – StripMap, b – ScanSAR, c – SpotLight [6]. Соответственно, при тестировании РСА, DRFM должна быть настроена на имитацию ЦРГ в соответствующих режимах. Кроме этого, тестированию может подвергаться неподвижный (рисунки 1...3) или движущийся (рисунок 4) РСА, что также должно учитываться в DRFM (см. Введение).

В ходе выполнения работ авторами настоящей статьи были проведены испытания разработанного устройства DRFM совместно с неподвижным макетом РСА в лабораторных условиях (по схеме, изображённой на рисунках 1 и 24). В процессе тестирования макетом РСА формировался один парциальный кадр РЛИ для обзорного режима ScanSAR (рисунок 28b) с характеристиками, перечисленными в таблице 4.

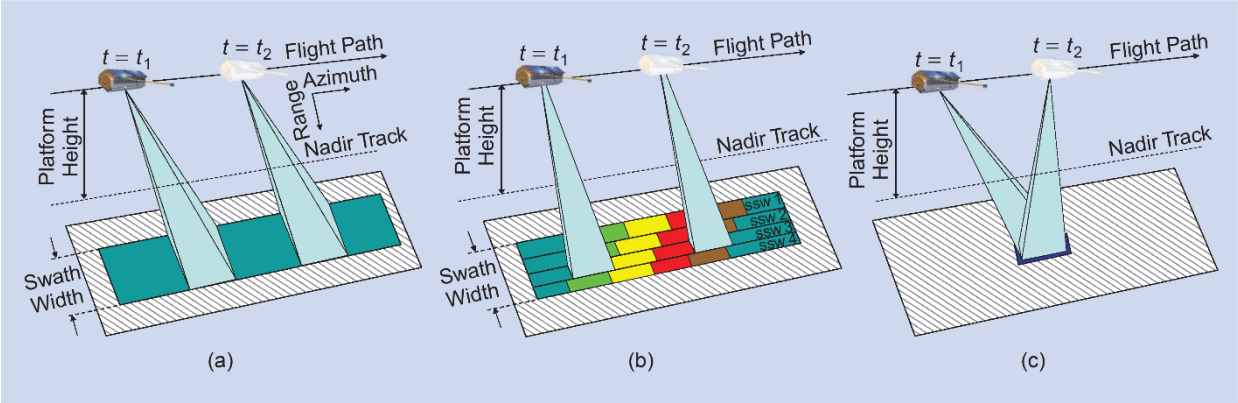


Рисунок 28 – Базовые режимы радиолокационной съёмки, реализуемые при помощи РСА:

a – StripMap, b – ScanSAR, c – SpotLight

Таблица 4. Основные характеристики режима работы тестируемого макета РСА

Диапазон частот	X
Диапазон снимаемых наклонных дальностей, км	1099...1129
Длительность пачки зондирующих радиоимпульсов (время синтеза апертуры), мс	17
Длина синтезируемой апертуры, м	111
Число зондирующих радиоимпульсов в пачке	80
Частота повторения зондирующих радиоимпульсов, Гц	4640.7
Длительность зондирующего радиоимпульса, мкс	8.19
Тип модуляции зондирующих радиоимпульсов	ФКМ
Длина М-последовательности	2047
Число различных чередующихся М-последовательностей в пачке	4
Ширина спектра зондирующего радиосигнала, МГц	250
Размер снимаемого парциального кадра (накл.дальн. х азимут), м	25000x2251
Размер снимаемого парциального кадра (накл.дальн. х азимут), пикселей	99x16
Разрешение по наклонной дальности и азимуту с учётом некогерентного накопления по наклонной дальности (размер одного пикселя РЛИ), м	300x141

В макете РСА использовался «Range-Doppler» алгоритм обработки ЦРГ (представленный на рисунке 7) с разделимыми быстрыми свёртками по наклонной дальности и азимуту без блока коррекции миграции по элементам дальности (Range Cell Migration Correction – RCMC), поскольку ввиду грубого разрешения тестируемого режима съёмки (размер пикселя РЛИ - 300х141м) такая коррекция не требовалась. Однако, в библиотеке макета РСА имеются и полные «Range-Doppler» и «Omega-k» алгоритмы, учитывающие RCMC. Их применение оправдано в более детальных режимах съёмки, таких, как StripMap и SpotLight. Отражения от подстилающей поверхности Земли в снимаемом кадре в целях обеспечения чистоты эксперимента не учитывались (отношение сигнал/шум достаточно велико).

Макетом РСА формировался зондирующий радиосигнал, подаваемый далее на вход устройства DRFM в виде аналоговых квадратурных составляющих (рисунок 24). В устройстве DRFM формировалась ЦРГ тестового 3-точечного изображения с размещением точек на плоскости кадра, задаваемым с хост-компьютера через программу управления устройством DRFM (рисунок 22). С выхода устройства DRFM сформированная ЦРГ принималась приёмной частью макета РСА, где производилась её цифровая обработка (в дальностном и азимутальном СФ) и строилось РЛИ имитируемого тестового кадра, которое отображалось на дисплее макета РСА.

На рисунке 29 можно видеть осциллограммы входного и выходного сигналов DRFM при формировании ЦРГ тестового кадра, содержащего 3 ТО.

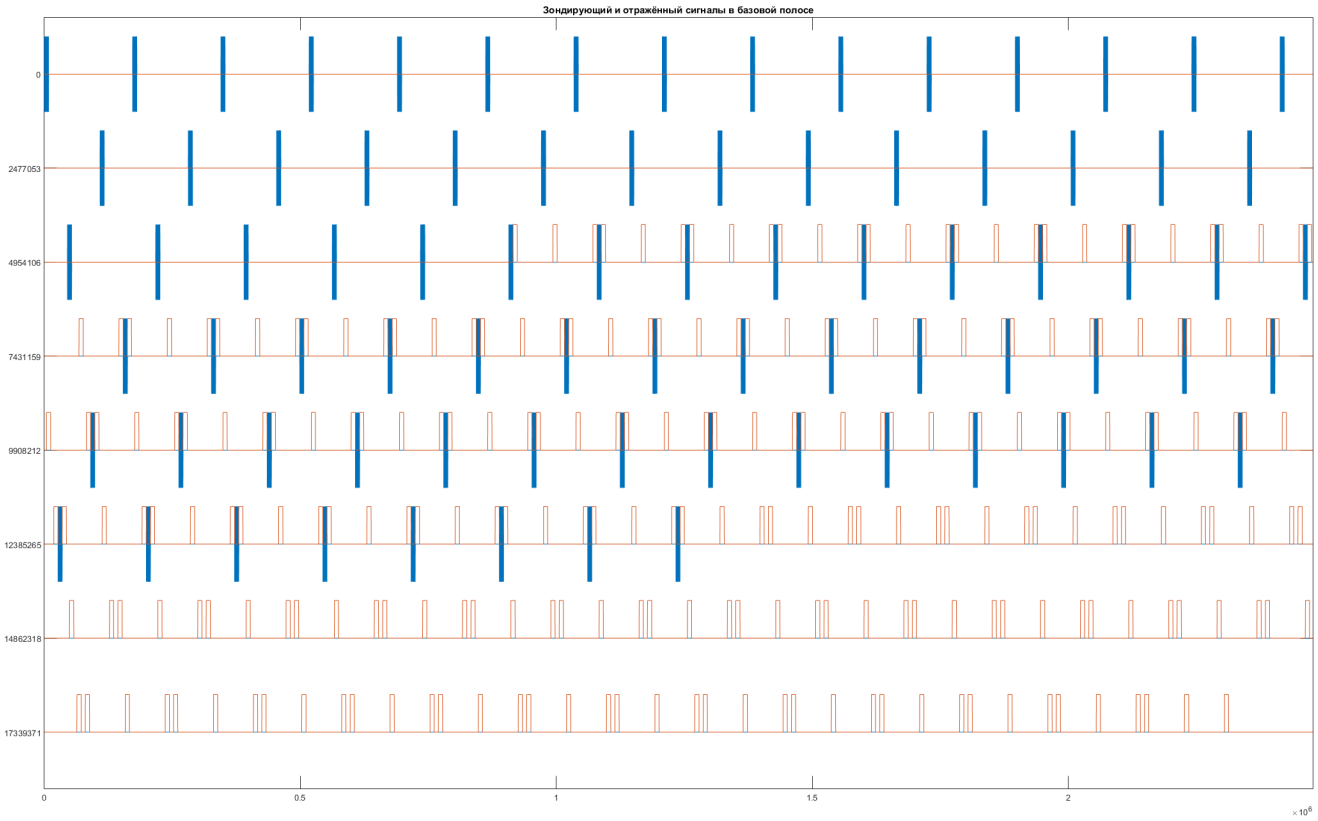


Рисунок 29 - Осциллограммы входного и выходного сигналов DRFM при формировании ЦРГ трёхточечного тестового кадра

Из рисунка видно, что полученные осциллограммы полностью соответствуют расчётным (рисунок 8), полученным в ходе численного расчёта ЦРГ трёхточечного тестового кадра. Из рисунка также наглядно видно, что DRFM начинает отвечать лишь по истечении некоторого времени (в данном примере – в ответ на 35й зондирующий радиоимпульс), что необходимо для имитации реального значения задержки отражённого от подстилающей поверхности радиосигнала.

На рисунке 30 приведены осциллограммы модулирующих сигналов (I- и Q- составляющие) упомянутых трёх ТО, расположенных (как показано далее) в разных местах тестового кадра.

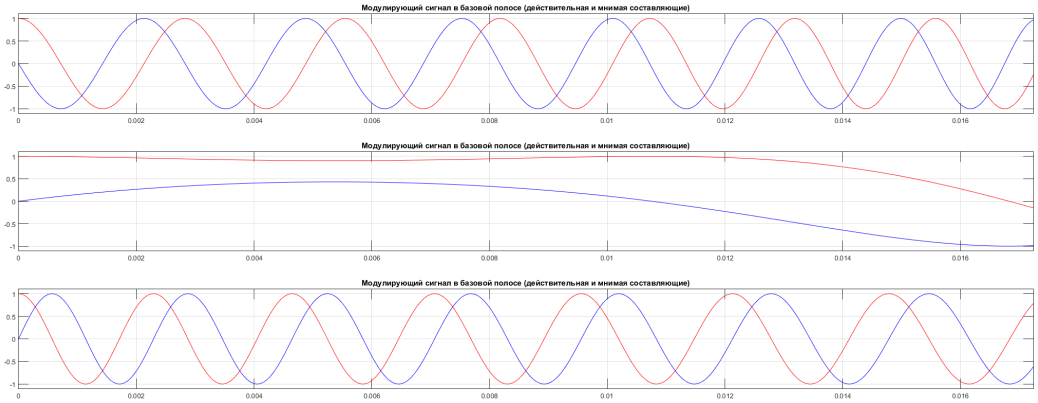


Рисунок 30 - Осциллограммы модулирующих сигналов (I- и Q- составляющие) трёх точечных отражателей, расположенных в разных местах тестового кадра

Из рисунка видно, что формы модулирующих сигналов существенно отличаются от постоянных доплеровских сдвигов, вносимых «классической» DRFM (отвечающей радарам без синтеза апертуры).

На рисунке 31 представлены трёхточечное тестовое изображение, задаваемое с хост-компьютера через программу управления устройством DRFM (слева), и соответствующее РЛИ (справа), полученное в макете РСА в результате обработки синтезированной в DRFM ЦРГ.

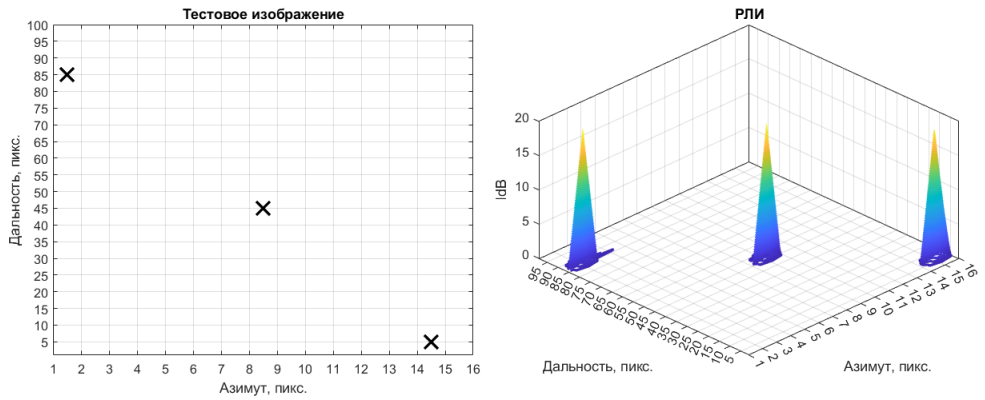


Рисунок 31 – Тестовое трёхточечное изображение (слева) и полученное РЛИ (справа)

На правом графике вдоль вертикальной оси отображается яркость РЛИ (по мощности, в дБ). Ширина откликов по уровню -3дБ составляет 1 пиксель как по дальности, так и по азимуту, а по уровню -20дБ – три пикселя по азимуту. Таким образом, из рисунка видно, что DRFM формирует достаточно чёткие точечные отметки в любом месте тестового кадра, как в его середине, так и на краях, что подтверждает корректность работы связки РСА – DRFM.

Использование разных гетеродинов в передающей и приёмной частях макета РСА

В ходе работ с макетом РСА и DRFM была продемонстрирована типичная ошибка, допускаемая некоторыми разработчиками РЛС и DRFM, - использование разных гетеродинов в передающей и приёмной частях. Так, первоначально в AWG генераторе M8190A и дигитайзере M9703A ошибочно использовались разные источники тактового сигнала: в AWG генераторе – свой встроенный, а дигитайзер тактировался от внешнего генератора радиочастотного сигнала N5171B. Использованные в макете генераторы компании Keysight M8190A и N5171B – если не прецизионные, то точно не ниже среднего класса, формируют достаточно качественные опорные сигналы с низким уровнем фазовых шумов. Однако, несмотря на это отметки трёх ТО на сформированном РЛИ оказались размытыми по азимуту, - рисунок 32, слева. Данный развал когерентного накопления по азимуту произошёл несмотря на его относительно короткое время (время синтезирования), – всего 17 мс (моделировалась съёмка одного кадра в режиме ScanSAR с грубым разрешением).

После безуспешного перебора различных вариантов причин данного развала радиоизображения, было высказано предположение, что причиной являются разные тактовые генераторы. Чтобы проверить данную гипотезу, на дигитайзер в качестве сигнала дискретизации был заведён тактовый сигнал AWG генератора M8190A (использующийся в последнем для тактирования ЦАП). После этого формируемые на РЛИ отметки ТО приняли форму острых пиков с шириной в 1 пиксель, как показано на рисунке 32, справа. Таким образом, экспериментально было подтверждено, что использование разных (на передачу и приём) тактовых генераторов или гетеродинов в радиолокационной системе недопустимо, поскольку приводит к развалу когерентной обработки. Как сигнал гетеродина, так и источник сигнала дискретизации ЦАП/АЦП для передающей и приёмной систем должны быть общими.

Заключение

Таким образом, в ходе проведенных работ был разработан и изготовлен стенд, включающий макет РСА и устройство DRFM, позволяющий моделировать, тестировать и отлаживать новые алгоритмы синтеза РЛИ и формирования ЦРГ. Новизна разработанного устройства DRFM заключается:

- в хранения в сигнальной памяти DRFM значительного числа зондирующих радиоимпульсов до момента ответа на первый радиоимпульс (в отличие от классической DRFM, запоминающей лишь один радиоимпульс), что приводит к увеличению требуемого объёма сигнальной

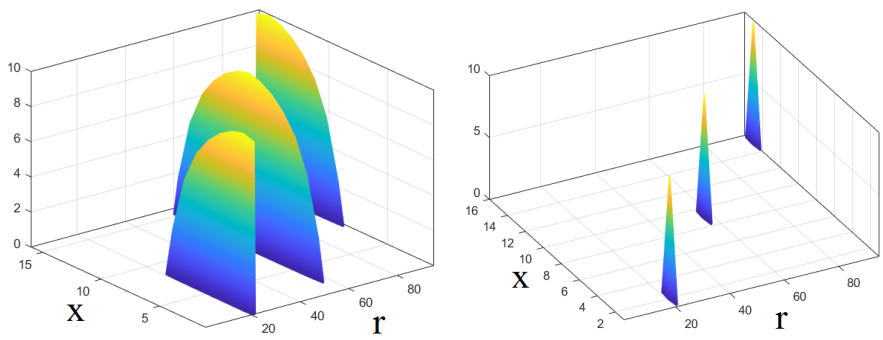


Рисунок 32 – РЛИ трёх ТО, сформированное в макете РСА при разных гетеродинах (слева) и при общем гетеродине (справа)

памяти DRFM и необходимости построения других, отличных от традиционных, схем синхронизации записи-считывания радиоимпульсов;

- в формировании новых (отличных от классической DRFM) сложных функций доплеровского сдвига переизлучаемого радиосигнала.

Разработанная DRFM пригодна для тестирования базовых характеристик РСА, таких, как разрешающая способность, уровень боковых лепестков отклика на точечный отражатель, динамический диапазон РЛИ. Однако, попытки сформировать при помощи данной DRFM изображение распределённых объектов, состоящих из множества точек, приводят к резкому увеличению требуемого количества аппаратных ресурсов ПЛИС. Так, описанная в [51] DRFM может формировать 32 точечные имитационные отметки. Таким количеством точек можно грубо сымитировать один простой объект, но формирование даже небольших искусственных карт снимаемой подстилающей поверхности не представляется возможным. Так, например, при среднем размере кадра в современных РСА 1000x1000 пикселей, требуется имитация 1 млн. точек. Поэтому для формирования ЦРГ, несущих искусственные карты, требуется «картографическая» DRFM, использующая совсем иные принципы обработки сигнала [56…59]. Такая DRFM была исследована и промоделирована в ходе последующих работ, на основе чего были предложены новые усовершенствованные схемы «картографической» DRFM, способной формировать чёткие искусственные карты для РСА, изменяющих период повторения зондирующих радиоимпульсов. Полученные результаты опубликованы в [60…62].

Список использованных источников

1. Верба В.С., Неронский Л.Б., Осипов И.Г., Турук В.Э. Радиолокационные системы землеобзора космического базирования / Под ред. В.С. Вербы. – М.: Радиотехника, 2010. – 680 с.
2. Авиационные системы радиовидения. Монография / Под ред. Г.С.Кондратенкова. – М.: «Радиотехника», 2015. 648 с.: ил.
3. Груздов В.В., Колковский Ю.В., Криштопов А.В., Кудря А.И. Новые технологии дистанционного зондирования Земли из космоса. Москва: ТЕХНОСФЕРА, 2018. – 482с.
4. Радиолокационные системы воздушной разведки, дешифрование радиолокационных изображений: учебник для курсантов ВВИА имени профессора Н.Е.Жуковского. Под ред. Л.А.Школьного. – М.: изд. ВВИА им. Проф. Н.Е.Жуковского, 2008
5. Ian G. Cumming, Frank H. Wong. Digital Processing of Synthetic Aperture Radar Data: Algorithms and Implementation, 1st ed. Artech House Publishers, Boston, London, 2005.
6. Alberto Moreira, Pau Prats-Iraola, Marwan Younis, Gerhard Krieger, Irena Hajnsek, and Konstantinos P. Papathanassiou. A Tutorial on Synthetic Aperture Radar. – IEEE Geoscience and remote sensing magazine, march 2013
7. Mehrdad Soumekh. Synthetic Aperture Radar Signal Processing with MATLAB Algorithms. – John Wiley & Sons, Inc., 1999
8. Mahafza B.R., Elsherbeni A.Z. MATLAB Simulations for Radar Systems Design. - CHAP-

MAN & HALL/CRC, 2004

9. Cumming I.G., Wong F.H. Digital Processing of Synthetic Aperture Radar Data: Algorithms and Implementation. - Boston: Artech House, 2005. - 436p.
10. Matthew Schlutz. Synthetic Aperture Radar Imaging Simulated in MATLAB. - San Luis Obispo, California Polytechnic State University, 2009
11. Bassem R. Mahafza. Radar Signal Analysis and Processing Using MATLAB. - CRC Press, 2009 - 500p.
12. Peter Knee. Sparse Representations for Radar with MATLAB Examples. Synthesis Lectures on Algorithms and Software in Engineering. - Arizona State University, 2012
13. Mahafza B.R. Radar Systems Analysis and Design Using MATLAB. Third Edition. - CRC Press, 2013 - 772p.
14. Марат Усс. Разработка радиолокационных систем в MATLAB и Simulink. - The MathWorks, Inc, 2016
15. Марат Усс. Проектирование радиолокационных систем. - ЦИТМ Экспонента, 2019
16. Сонин А.П. Моделирование цифровой обработки сигналов в радиолокаторах с синтезированной апертурой антенны на множестве CUDA GPU в MATLAB. - сборник трудов VII Всероссийской научно-практической конференции «Технологии разработки и отладки сложных технических систем: (Москва, 1-2 апреля 2020 г.). - Москва: Издательство МГТУ им. Н. Э. Баумана, 2021. - 457 с.
17. Sheldon C. Spector. A Coherent Microwave Memory Using Digital Storage: The Loopless Memory Loop. - Association of Old Crows, issue of Electronic Warfare, 1975, Jan/Feb
18. Lowenschuss O. «Когерентная ВЧ память – новое средство для обработки сигналов» - IEENEACON-80, Dayton. Proc.p. 1188-1194 пер. № 81/42813 в фонде ГПНТБ
19. Lowenschuss O., Bruce E. Gordon "Digital memory system" – US Patent №4280219, Jul.21.1981
20. Карманов Ю.Т., Рукавишников В.М., Шуняев М.И. Исследование параметров цифровых устройств запоминания и воспроизведения радиосигналов - Сборник научных трудов ЧПИ, 1980, с.70-76
21. Карманов Ю.Т., Шуняев М.И., Рукавишников В.М., Хабин В.А. Цифровой формирователь многократных ответных радиосигналов – Авторское свидетельство СССР № 187159, с приоритетом от 5.11.1980г.
22. Шуняев М.И., Рукавишников В.М., Мельников В.Н. Цифровые методы запоминания и воспроизведения радиосигналов. – Теория и техника радиосистем, Сб. научных трудов ЧПИ, №273 – Челябинск, ЧПИ, 1982, с.39-44
23. Крылов В.В., Никашов К.Ю. Перспективы развития техники и технологии систем радиоэлектронной борьбы. – Зарубежная радиоэлектроника 1988, №6
24. McMillian G. Digital RF Memory (DRFM), Austin, TX: Systems & Processes Engineering Corp., 1996
25. Карманов Ю.Т., Рукавишников В.М. Цифровые способы запоминания и воспроизведения радиосигналов. – научно-технический журнал «Цифровые радиотехнические системы», №1, Челябинск, 1997г., НИИ Цифровых Систем
26. Афинов В. Направления совершенствования средств РЭП индивидуальной защиты самолётов. - Зарубежное военное обозрение, №7 №9,

1998

27. Горбунов Ю.Н. Технология цифрового запоминания пространственных частот DRFM-S и перспективы ее внедрения в авионику нового поколения. – Радиотехника, 2003, №1. - с. 67-72
28. Schleher D. Curtis. Electronic Warfare in The Information Age. - Artech House, 1999. - 624p.
29. Карманов Ю.Т. Проблемы и перспективы развития цифровых устройств запоминания и воспроизведения радиосигналов. - Цифровые радиоэлектронные системы, 2002- 2004, №5
30. Сонин А.П. Основные тенденции в построении современных и перспективных цифровых устройств формирования помех на основе цифровой радиочастотной памяти (DRFM). По материалам открытой отечественной и зарубежной печати — Цифровые радиоэлектронные системы, 2002- 2004, №5
31. Карманов Ю.Т. 25-летний опыт применения цифровых технологий обработки радиосигналов в НИИ Цифровых Систем ЮУрГУ. – Вестник ЮУрГУ, №23, 2007
32. Непомнящий Г.А. О влиянии параметров цифрового устройства записи и воспроизведения на качество имитационного сигнала сложной структуры. – Цифровые радиоэлектронные системы №7, 2007-2008, с.89-95
33. Добыкин В.Д., Куприянов А.И., Пономарёв В.Г., Шустов Л.Н. Радиоэлектронная борьба. Цифровое запоминание и воспроизведение радиосигналов и электромагнитных волн. – М.: Вузовская книга, 2009
34. Бородин А.М., Лобанов Б.С., Сонин А.П. Пути построения интегрированных бортовых радиотехнических комплексов с цифровой обработкой радиосигналов. Учебное пособие. - М.: МИРЭА, 2011. - 57с.
35. Леньшин А.В. Бортовые системы и комплексы радиоэлектронного подавления. Воронеж: Издательско-полиграфический центр «Научная книга», 2014
36. Егоров Н., Кочемасов В. Технология цифровой радиочастотной памяти и её применение в системах РЭБ - Электроника: НТБ. - 2016. - №10. - С. 62-71
37. De Martino A. Introduction to Modern EW Systems, 2nd edition - Boston; London: Artech House, 2018. - 485p.
38. Карманов, Юрий Трофимович. Материал из Википедии — свободной энциклопедии https://ru.wikipedia.org/wiki/Карманов,_Юрий_Трофимович
39. Карманов, Юрий Трофимович https://ru.ruwiki.ru/wiki/Карманов,_Юрий_Трофимович
40. Карманов, Ю. Т. Жизнь как служба науке : [интервью] / провёл И. Загребин. – Текст: непосредственный. // Технополис. – 2015. – 27 янв. (№ 1). – С. 4. http://gate.lib.susu.ac.ru/ftd?base=SUSU_HISTORY&key=000529655&dtype=F&etype=.pdf
41. 75-летие Юрия Трофимовича Карманова. Карманов Юрий Трофимович (09.01.1945 – 29.06.2017). Календарь знаменательных дат и событий Южно-Уральского государственного университета. - Научная библиотека Южно-Уральского государственного университета, 9 января 2020
42. Aviaconversiya. From Wikipedia, the free encyclopedia https://en.wikipedia.org/wiki/

Aviaconversiya

43. Игорь Коротченко. Электронные друзья Саддама. - Независимая газета, интернет-версия 25.03.2003 https://www.ng.ru/politics/2003-03-25/1_friends.html
44. "Авиаконверсия": мы санкций не нарушали http://news.bbc.co.uk/hi/russian/special_report/bbcrussian/2002_07/newsid_2880000/2880765.stm
45. Олег Антонов. Чтобы не было конфуза... - Независимое военное обозрение, интернет-версия 25.07.2008 https://nvo.ng.ru/wars/2008-07-25/1_bpla.html
46. Олег Антонов. Почему коррупция разъедает ОПК. - Независимое военное обозрение, интернет-версия 22.08.2008 https://nvo.ng.ru/concepts/2008-08-22/10_opk.html
47. Способы защиты от "бесконтактного удара". - Независимое военное обозрение, интернет-версия 19.09.2008 https://nvo.ng.ru/wars/2008-09-19/6_iran.html
48. О.Антонов: РЭП - наиболее перспективная задача для БЛА https://www.aviaport.ru/news/165848/
49. Умер Олег Антонов http://www.lebedyan.net/novosti/umer-oleg-antonov/
50. Da-hai Dai, X.F. Wu, Xue-song Wang, Shunping Xiao. SAR Active-Decoys Jamming Based on DRFM. - 2007 IET International Conference on Radar Systems
51. Галашин М.Е., Лисовская Т.В., Мельников М.Ю., Кочеров А.Н., Полянский П.О. Формирование изображения ложной цели на экране радиолокаторов с синтезированной апертурой с помощью универсального устройства формирования сигнала на основе полузаказной СБИС 1879ВМ3 // Материалы конференции по беспилотным системам UVS Tech 2010. uvs-info.com; studydoc.ru
52. Shoalehvar A. Synthetic Aperture Radar (SAR) Raw Signal Simulation. - California Polytechnic State University, 2012
53. Белоруцкий Р.Ю. Цифровые методы имитации эхосигналов РЛС с синтезированием апертуры антенны. Диссертация на соискание ученой степени кандидата технических наук. – Томск, Новосибирский государственный технический университет, 2014
54. Xu Yinhui, Zeng Dazhi, Yan Tao, Xu Xiaoheng. A Real-time SAR Echo Simulator Based on FPGA and Parallel Computing. - TELKOMNIKA, Vol.13, No.3, September 2015, pp. 806~812
55. Ю.Н. Горбунов, А.П. Сонин, А.В. Хромцев, Д.М. Свирин. Тестирование радиолокаторов с синтезированной апертурой антенны при помощи DRFM. – Журнал радиоэлектроники, ISSN, 1684-1719, №1, 2022, http://jre.cplire.ru/jre/jan22/7/text.pdf, https://doi.org/10.30898/1684-1719.2022.1.7
56. Zhu Yan, Zhao Guoqing, Zhang Yu. Research on SAR Jamming Technique Based on Man-made Map. - 2006 CIE International Conference on Radar
57. Jamal Saeedi. A New Hybrid Method for Synthetic Aperture Radar Deceptive Jamming - International Journal of Microwave Engineering (JMICRO) Vol.4, No.1, January 2019
58. Qingyang Sun, Ting Shu, Kai-Bor Yu, Wenxian Yu. Efficient Deceptive Jamming Method of Static and Moving Targets Against SAR. - IEEE Sensors Journal (Volume: 18, Issue: 9, May1, 1 2018), DOI: https://doi.org/10.1109/JSEN.2018.2813521
59. Сонин А.П. Тестирование радиолокаторов

с синтезированной апертурой антенны при помощи цифровой радиочастотной памяти (DRFM), формирующей искусственные карты // Наукоемкие технологии. 2023. Т. 24. № 2. С. 67–78. DOI: https:// doi.org/10.18127/j19998465-202302-09

60. Сонин А.П. Тестирование радиолокаторов с синтезированной апертурой антенны при помощи цифровой радиочастотной памяти (DRFM), формирующей искусственные карты. - VIII Всероссийская научно-техническая конференция «Дальняя радиолокация на службе Отечеству» : труды конференции / Акционерное общество «Концерн «Радиотехнические и Информационные Системы», Акционерное общество «Научно-производственный комплекс «Научно-исследовательский институт дальней радиосвязи», Акционерное общество «Радиотехнический институт имени академика А.Л. Минца». — Москва : Издательство МГТУ им. Н.Э. Баумана, 2022. — 387, [1] с. : ил.
61. Сонин А.П. Параллельное моделирование в MATLAB цифровой радиочастотной памяти (DRFM), формирующей искусственные карты для тестирования радиолокатора с синтезированной апертурой антенны. - Сборник докладов Международной конференции «Моделирование в инженерном деле», Москва, МГТУ имени Н.Э.Баумана, 2023
62. Сонин А.П. Влияние перестройки периода повторения зондирующих радиоимпульсов на качество радиолокационного изображения, формируемого цифровой радиочастотной памятью на выходе тестируемого радиолокатора с синтезированной апертурой антенны. – сборник докладов X Всероссийской научно-технической конференции «Дальняя радиолокация на службе Отечеству». – Москва, 2023

Загрузка драйвера UEFI, с помощью ПЛИС

Погодаев А.А. *Telegram:* @altair700

Абрамов А.Е. *Telegram:* @alexandr_7722

Обсуждение и комментарии: link

Аннотация

В данной статье, я покажу свой опыт работы с платой KCU105, а именно как подгружать драйвер через EROM.

1 Постановка задачи

Задача следующая: Нужно сконфигурировать ПЛИС как Endpoint, записать во внутренню память драйвер, подключить к компьютеру, проверить что операционная система правильно считывает EROM и увидеть данные переданные по COM -порту. На рис.1 изображена плата, которую будем использовать в качестве примера.

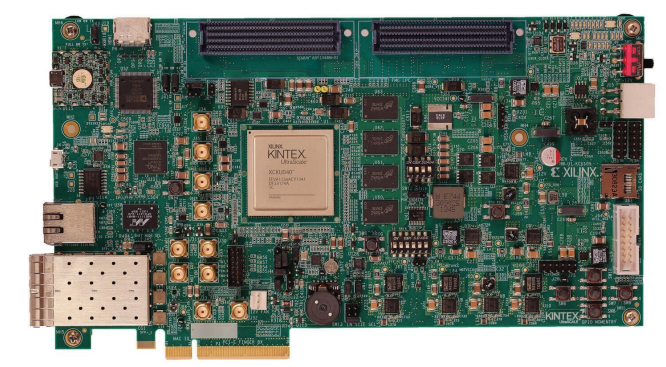


Рис. 1: AMD Kintex UltraScale FPGA KCU105 Evaluation Kit, можно приобрести на официальном сайте за сущие копейки (3,882.00$)

2 Полезные знания перед началом

2.1 EROM

Материнская плата имеет свою микросхему ROM BIOS, в которой записаны драйверы для поддержки устройств на плате. При необходимости дополнительные устройства, устанавливаемые в слоты шин расширения (ISA, PCI, PCMCIA), могут иметь микросхемы ПЗУ своей программной поддержки Если вы хотите добавить устройство через слоты шин расширение (в нашем случае это интерфейс PCI), то оно должно иметь своё ПЗУ внутри которой находится программная поддержка — Additional ROM BIOS (дополнительные модули ROM BIOS), они же Expansion ROM (далее EROM). Эта необходимость возникает, когда программная поддержка устройств требуется до загрузки ОС и прикладного ПО. Роль Expansion ROM может и не ограничиваться поддержкой данного устройства — в таком модуле может содержаться и вся программа функционирования специализированного бездискового контроллера на базе PC.

3 Приступаем к работе

3.1 Создание проекта

Для начала нужно создать проект с нашей платой.

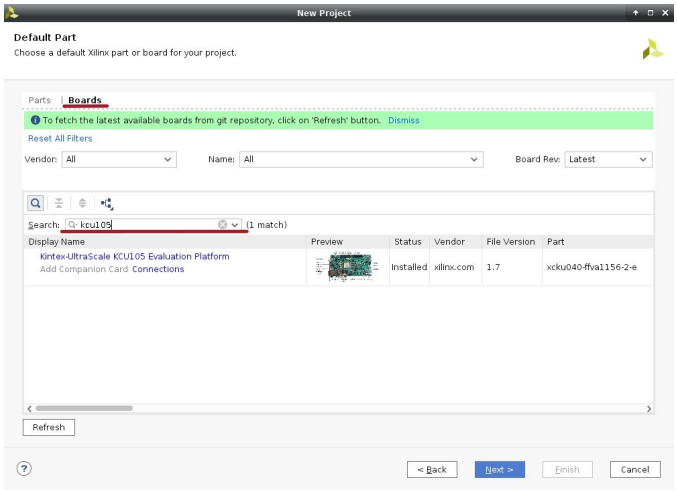


Рис. 2: Выбираем во вкладке "Boards" через поиск плату.

Далее нужно выбрать интересующее нас готовое IP.

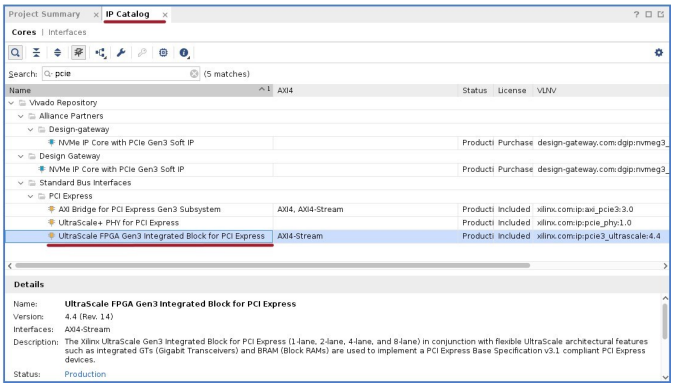


Рис. 3: Для появления вкладки "IP Catalog нужно нажать на "IP Catalog" под вкладкой PROJECT MANAGER

На данном этапе ничего в настройках IP менять не нужно, щелкаем по выбору и откроется новое окно в котором нужно нажать "OK". Пропускаем генерацию IP.

Рис. 4: В новом окне "Generate Output Products" жмём на "Skip".

Создаём пример проекта по IP. Во вкладке "Design Sources"нажимаем правой кнопкой мыши по модулю и выбираем "Open IP Example Design".

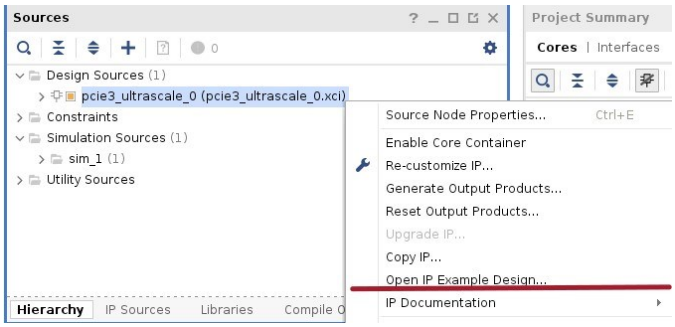


Рис. 5: Создание "примера" модуля готового проекта.

Разворачиваем проект в нужной нам директории и он автоматически откроется.

3.2 Сгенерированный пример

Далее мы будем работать с проектом, который предоставляет в качестве примера, Vivado. В этом примере есть обвзяка для нашего модуля, плюс автоматически добавлены входы и выходы интерфейса PCIE и подведены тактовые сигналы. Визуально это можно оценить открыв вкладку "RTL ANALYSIS -> Open Elaborated Design -> Schematic". Я же предоставлю схему в общем виде.

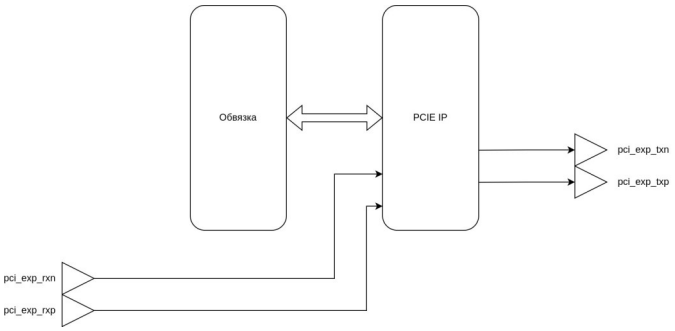


Рис. 6: Схема собранного проекта в общем виде.

Если опустить все ньюансы, на обвязку приходят пакеты по шине AXIS, по которой идут данные согласно спецификации. Подробнее о формировании пакетов PCIE отсылаю вас к статье "PCI Express в ПЛИС V-й серии Intel: основы интерфейса и особенности аппаратных ядер"именно эти

пакеты приходят на "Обвязку" в запакованном виде по AXIS. На рис.6 нас интересует "Обвязка" именно в ней находится память EROM и другие. Под "другими" имеется ввиду BAR's, это тоже память оконечного устройства, но используется не для наших целей.

3.3 Память в проекте

Давайте посмотрим, что сгенерировала Vivado и с чем нам предстоит работать. Интересующие нас элементы памяти находятся в модуле ep_mem.

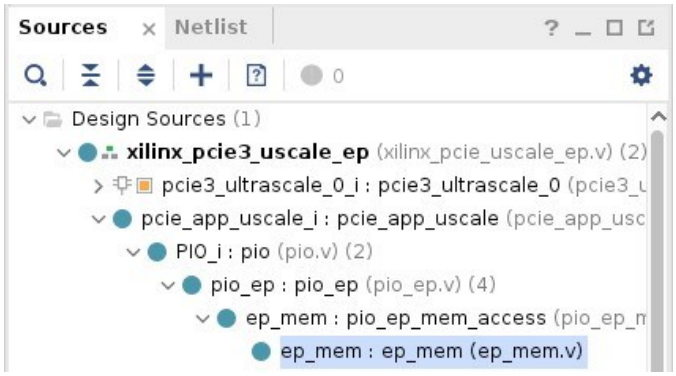


Рис. 7: Путь к нужному файлу.

Давайте рассмотрим файл подробнее. Первое на что стоит обратить внимание, это комментарий автора:

> *Description: Endpoint Memory: 8KB organized as 4 x (512 DW) BlockRAM banks.*
>
> *Block RAM Port A: Read Port.*
>
> *Block RAM Port B: Write Port.*

Давайте теперь разбираться в чем проблема. Если мы откроем конфигурацию нашего IP рис.8 то увидим следующее:

1. В настройках IP у нас подключена только одна память.
2. EROM отключен вовсе.
3. Модуль настроен на чтение памяти размером 2КБ.

Собственно можно сделать вывод - пример дизайна настроен на оганиченный функционал и в этой статье мы будем его модернизировать. Сразу отмечу, что если вы изначально при создании проекта в параграфе 3.1 зададите нужные вам параметры и создадите на его основе пример дизайна, то ничего не измениться.

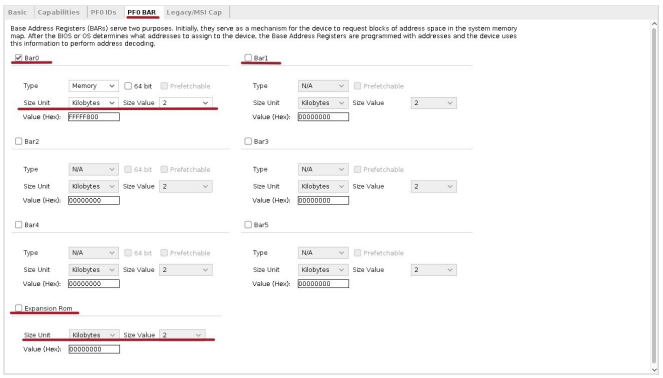


Рис. 8: Настройка блока.

Для некоторых драйверов нам может не хватить заявленых "по факту" 8КБ памяти, соответственно, наша задача, максимально увеличить память на оконечной точке в проекте. На данном этапе нам нужно поставить значение на 16КБ в плашках Bar0 и Expansion Rom (также поставить галку рядом).

4 Подробнее о проблеме

4.1 Описание памяти

В качестве памяти "Обвзяка"содержит примитив RAMB36E2 36Кб память, которая служит как и EROM так и BAR's, для нашего устройства. Видно, что на вход шина адреса имеет ширину 15 бит (в примере почемуто подводят 16 бит), в коде дела обстоят иначе:

```
.ADDRARDADDR({1 'b0 ,a_rd_a_i_3[8:0] ,6 'b0}) ,
.ADDRBWRADDR({1 'b0 ,b_wr_a_i_3[8:0] ,6 'b0}) ,
```

Не совсем понятно почему так сделано, возможно из-за параметров INIT_XX, с их помощью можно инициализировать память через параметры, но они занимают не всё пространство.

Примечание: В описании к памяти есть параметр INIT_FILE, но после экспериметнов не удалось выяснить как оно работает, после

чтения памяти не удалось понять как правильно инициализировать память через файл. Замечу, что в примере этот параметр отсутсвует, другие параметры работают.

Таблица 1: Описание примитива.

Attribute	Type	Allowed Values	Default	Description
Initialization File: File name of file used to specify initial RAM contents.				
INIT_FILE	STRING	String	"NONE"	File name of file used to specify initial RAM contents.

К решению этой проблемы мы приступим позднее.

5 Симуляция проекта

5.1 Запуск готового теста

Vivado помимо примера, предоставляет тестовое окружение для проекта, сразу отмечу что я не являюсь верификатором должного уровня, чтобы чётко разобраться во всех тонкостях теста. Тест использует множество тасков для проверки модели, но мы будем использовать один "большой" тест собранный из тасков. Для работы с простыми тестами нам нужно два файла: pci_exp_asrapp_tx.v и sample_tests.vh. Первый содержит инициализацию параметра test-name который отвечает за выбор теста, во втором его описание и состав. По умолчанию тест записывает и читает из памяти.

Запустим симуляцию и посмотрим тест на памяти. Советую заранее закинуть сигналы модуля ep_mem потому что симуляция очень долгая.

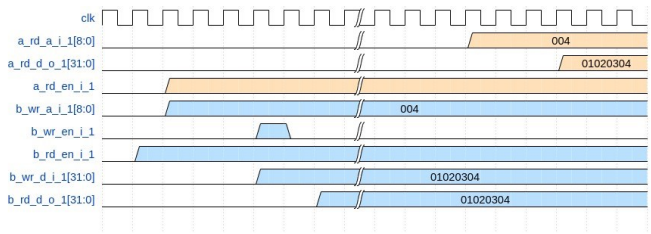


Рис. 9: Схемотичное изображение теста. Цветом разделены каналы a и b.

Видно, что запись и чтение проходит нормально, что также дублируется в терминале "TEST PASSED". Теперь у нас есть основные знания при работе с примером, можно теперь его улучшить.

6 Улучшение проекта

6.1 Замена памяти

Теперь нам нужно заменить память на свою. Напоминаю, что это нужно для того, чтобы инициализировать её через файл .coe, почему-то примитивы Xilinx не берут в оборот файлы через параметр (если вы знаете в чём может быть проблема, сообщите, с файлом .mif тоже были проблемы). Также имея свою память мы можем указывать её глубину сами. Заменять будем память с помощью IP Catalog, выбрать надо Block Memory Generator.

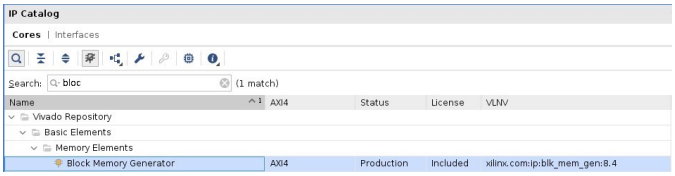


Рис. 10: Создаём свою блочную память.

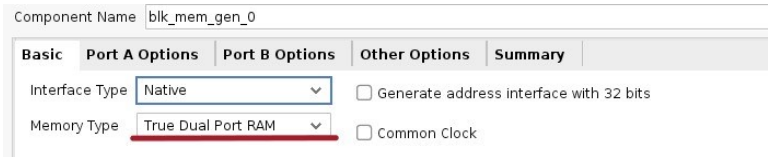


Рис. 11: Настройка страницы Basic.

Рис. 12: Настройка страницы Port A Options.

Далее необходимо вставить память вместо примитива RAMB36E2 в модуле ep_mem. Комментируем модуль ep_mem_erom и вставляем свой.

```
blk_mem_gen_0 erom_inst
    (
      .clka   ( clk_i     ) ,
      .ena    (a_rd_en_i_3) ,
      .wea    (1 'b0      ) ,
      .addra  (a_rd_a_i_3 ) ,
      .dina   (32 'b0     ) ,
      .douta  (a_rd_d_o_3 ) ,
      .clkb   ( clk_i     ) ,
      .enb    (b_rd_en_i_3) ,
      .web    (b_wr_en_i_3) ,
      .addrb  (b_wr_a_i_3 ) ,
      .dinb   (b_wr_d_i_3 ) ,
      .doutb  (b_rd_d_o_3 )
    );
```

Теперь можно проверить его расположение с помощью иерархии. Также это можно сделать с помощью "RTL ANALYSIS -> Open Elaborated Design -> Schematic".

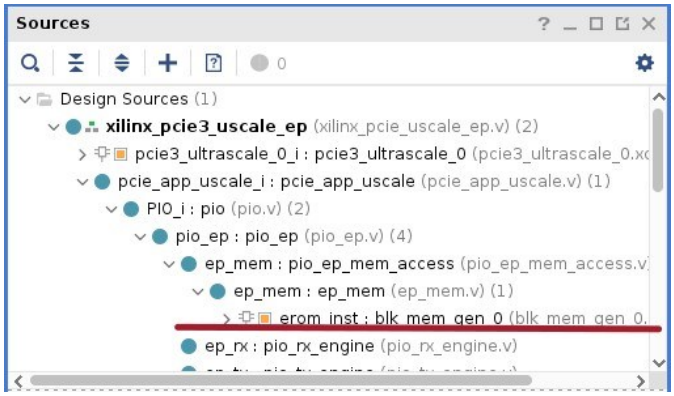


Рис. 13: Иерархия проекта.

6.2 Расширение шины адреса

Из прошлых папраграфов было ясно, что ширины адреса в 9 бит не хватит для установки драйвера, соответственно надо изменить обвязку, чтобы на модуль ep_mem приходил сигнал, нужной ширины. Для начала нужно установить пути сигналов, воспользуемся "RTL ANALYSIS -> Open Elaborated Design -> Schematic".

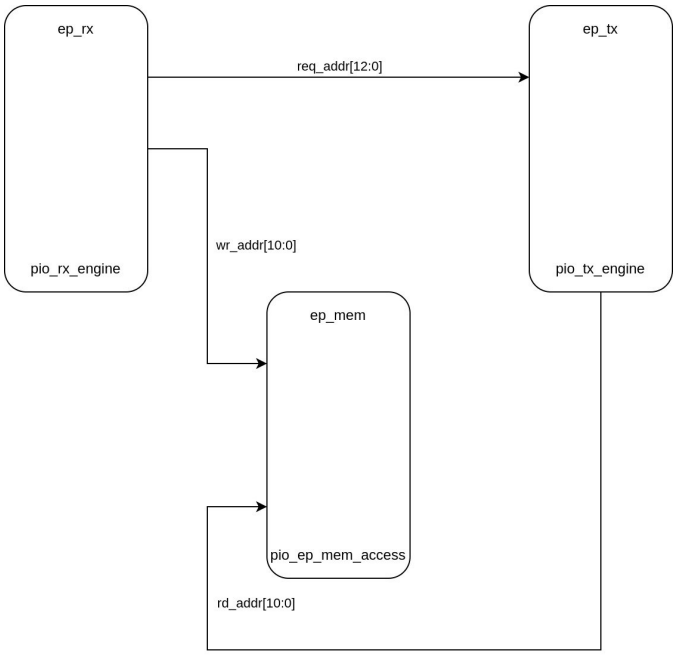


Рис. 14: Нужные сигналы и их соединения.

Из рисунка 14 видно, что в модуль pio_ep_mem_access заходят два сигнала rd_addr[10:0] и wr_addr[10:0], их формируют модули pio_tx_engine и pio_rx_engine с них и начнём.

6.3 Изменения pio_rx_engine

По рисунку 14 видно, что нам нужно изменить два сигнала wr_addr и req_addr. Далее я приведу список изменений в файле, идея в том, чтобы расширить ширину адреса до 16 бит, ищем выходной сигнал и прибавляем к его ширине недостающие биты, также расширяем регистры им присваемым.

```
0135. output reg    [19:0]  req_addr ,
0161. output reg    [17:0]  wr_addr ,
0275. req_addr  <= #TCQ 20 'b0 ;
0279. wr_addr   <= #TCQ 18 'b0 ;
0362. req_addr  <= #TCQ { . . . , desc_hdr_qw0 [17:2] , . . . } ;
0399. req_addr  <= #TCQ { . . . , desc_hdr_qw0 [17:2] , . . . } ;
0441. req_addr  <= #TCQ { . . . , desc_hdr_qw0 [17:2] , . . . } ;
0478. req_addr  <= #TCQ { . . . , desc_hdr_qw0 [17:2] , . . . } ;
0553. req_addr  <= #TCQ { . . . , desc_hdr_qw0 [17:2] , . . . } ;
0651. wr_addr   <= #TCQ req_addr [ 1 7 : 2 ] ;
0771. req_addr  <= #TCQ 20 'b0 ;
0775. wr_addr   <= #TCQ 18 'b0 ;
0840. req_addr  <= #TCQ {.. , m_axis_cq_tdata [ 1 7 : 2 ] , . . } ;
0877. req_addr  <= #TCQ {.. , m_axis_cq_tdata [ 1 7 : 2 ] , . . } ;
0920. req_addr  <= #TCQ {.. , m_axis_cq_tdata [ 1 7 : 2 ] , . . } ;
0957. req_addr  <= #TCQ {.. , m_axis_cq_tdata [ 1 7 : 2 ] , . . } ;
1033. req_addr  <= #TCQ {.. , m_axis_cq_tdata [ 1 7 : 2 ] , . . } ;
1135. wr_addr   <= #TCQ req_addr [ 1 7 : 2 ] ;
1273. wr_addr   <= #TCQ 18 'b0 ;
1330. req_addr  <= #TCQ {.. , m_axis_cq_tdata [ 1 7 : 2 ] , . . } ;
1381. wr_addr   <= #TCQ {.. , m_axis_cq_tdata [17:2]};
1386. req_addr  <= #TCQ {.. , m_axis_cq_tdata [ 1 7 : 2 ] , . . } ;
1407. req_addr  <= #TCQ {.. , m_axis_cq_tdata [ 1 7 : 2 ] , . . } ;
1439. req_addr  <= #TCQ {.. , m_axis_cq_tdata [ 1 7 : 2 ] , . . } ;
1460. wr_addr   <= #TCQ {.. , m_axis_cq_tdata [17:2]};
1525. req_addr  <= #TCQ {.. , m_axis_cq_tdata [ 1 7 : 2 ] , . . } ;
1630. wr_addr   <= #TCQ req_addr [ 1 7 : 2 ] ;
```

Проверяем изменения визуально в "RTL ANALYSIS -> Open Elaborated Design -> Schematic"и идём дальше.

6.4 Изменения pio_tx_engine

Делаем то же самое как и в разделе 6.3.

```
0149. input [19:0]  req_addr ,
0170. output reg    [17:0]  rd_addr ,
0253. rd_addr   <= #TCQ req_addr [ 1 9 : 2 ] ;
0258. rd_addr   <= #TCQ req_addr [19:2] + 11 'h001 ;
0272. rd_addr   <= #TCQ req_addr [ 1 9 : 2 ] ;
```

6.5 Изменения pio_ep

Модули соеденены на более высоком уровне, соответственно надо произвести изменения ширины проводов.

```
195.    wire    [17:0]  rd_addr ;
199.    wire    [17:0]  wr_addr ;
216.    wire    [19:0]  req_addr;
```

6.6 Изменения pio_ep_mem_access

Этот модуль обвязка для памяти, соответственно нужно рассортировать сигналы, которые идут на модули памяти.

```
083. input  [17:0]  rd_addr ,
090. input  [17:0]  wr_addr,
122. reg    [17:0]  wr_addr_ptr ;
311. case ({wr_addr [17:16]})
324. wire rd_data0_en = {rd_addr [17:16]  == 2'b00 };
325. wire rd_data1_en = {rd_addr [17:16]  == 2'b01 };
326. wire rd_data2_en = {rd_addr [17:16]  == 2'b10 };
327. wire rd_data3_en = {rd_addr [17:16]  == 2'b11 };
332. case ({rd_addr [17:16]})
354. rd_addr [15:0]
358. wr_addr_ptr [15:0]
360. wr_addr [17:16] == 2 'b00
362. wr_addr [17:16]
364. rd_addr [15:0]
368. wr_addr_ptr [15:0]
370. wr_addr [17:16]
372. wr_addr [17:16]
374. rd_addr [15:0]
378. wr_addr_ptr [15:0]
380. wr_addr [17:16]
382. wr_addr [17:16]
384. rd_addr [15:0]
388. wr_addr_ptr [15:0]
390. wr_addr [17:16]
392. wr_addr [17:16]
```

В данном модуле происходит выбор между памятью.

6.7 Изменения ep_mem

В данном модуле мы подводим к памяти расширенные сигналы.

```
0113. input [15:00] a_rd_a_i_0;
0117. input [15:00] b_wr_a_i_0;
0123. input [15:00] a_rd_a_i_1;
0127. input [15:00] b_wr_a_i_1;
0133. input [15:00] a_rd_a_i_2;
0137. input [15:00] b_wr_a_i_2;
0143. input [15:00] a_rd_a_i_3;
0147. input [15:00] b_wr_a_i_3;
0338. .ADDRARDADDR(a_rd_a_i_0)
0339. .ADDRBWRADDR (b_wr_a_i_0)
0547. .ADDRARDADDR(a_rd_a_i_1)
0548. .ADDRBWRADDR(b_wr_a_i_1)
0754. .ADDRARDADDR(a_rd_a_i_2)
0755. .ADDRBWRADDR(b_wr_a_i_2)
```

На этом модификация почти закончена, приступим к следующему этапу.

6.8 Проблемы с симуляцией

Если мы запустим симуляцию чтобы проверить увеличенное адресное пространство (это можно сделать в строках 368 и 395, файл sample_tests.vh), проблема возникнет в следующем:

На рисунке 15 указаны сигнала для нашего EROM, но и для BAR's они тоже актуальны. Дело в том, что почему-то IP PCIE формирует адрес на чтение иначе. Я пытался понять почему так происходит, возможно надо читать документацию на блок или более подробнее его изучать, но за неимением на это времени придётся пойти на хитрость. Если вы знаете в чем может быть особенность напишите.

Чтобы избежать этой проблемы, нужно пойти на хитрость, в 13м бите адреса возникаем непонятная единица, чтобы избежать этого сокращаем шину адреса до 12 бит. Меняем файл ep_mem.v.

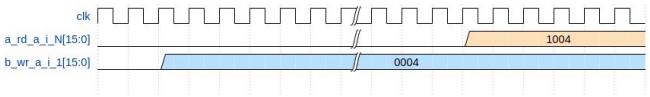


Рис. 15: Разница между адресом записи и чтения.

```
blk_mem_gen_0 erom_inst
(
    .clka   ( clk_i ) ,
    .ena    (a_rd_en_i_3) ,
    .wea    (1 'b0  ) ,
    .addra  (a_rd_a_i_3 [11:0]) ,
    .dina   (32 'b0 ) ,
    .douta  (a_rd_d_o_3 ) ,
    .clkb   ( clk_i ) ,
    .enb    (b_rd_en_i_3) ,
    .web    (b_wr_en_i_3) ,
    .addrb  (b_wr_a_i_3 [11:0]) ,
    .dinb   (b_wr_d_i_3 ) ,
    .doutb  (b_rd_d_o_3 )
);
```

Забегая вперёд, скажу, что это стало моментом, когда всё заработало.

7 Драйвер.

7.1 Инструкция

Сразу скажу что в этой части мне помог разработчик, предоставляю исходники как есть. В этом разделе я опишу какие шаги были предприняты для формирования окружения, компиляции и вставки своего драйвера в проект. Компиляция проекта производилась на Ubuntu. База знаний, которая легла в основу, расположена в "списке литературы" 9.1. Опишу по пунктам что надо сделать:

Установить необходимые зависимости:

```
sudo apt install build-essential uuid-dev
iasl git nasm
python-is-python3
```

Скачать/клонировать с Git репозитория:

https://github.com/tianocore/edk2

Инициализация модулей:

```
cd edk2
git submodule update –init
```

Скомпилировать build tools:

```
make -C BaseTools
. edksetup.sh
export EDK_TOOLS_PATH=$HOME/projects/edk2/
BaseTools
```

Редактировать файл /Conf/target.txt:

```
ACTIVE_PLATFORM = OvmfPkg/OvmfPkgX64.dsc
TARGET_ARCH = X64
TOOL_CHAIN_TAG = GCC5
MAX_CONCURRENT_THREAD_NUMBER = 5
```

Редактировать файл /OvmfPkg/OvmfPkgX64.dsc:

```
PLATFORM_GUID = https://www.guidgen.com/
(вставить из сайта)
DEFINE SOURCE_DEBUG_ENABLE = TRUE
```

Вставить в самый конец

```
MyDriver/MyDriver.inf{
 <LibraryClasses>
 DebugLib|MdePkg/Library/BaseDebugLibSerialPort/
 BaseDebugLibSerialPort.inf
}
```

Теперь нужно закинуть папку с драйвером в edk и скомпилировать проект следующей командой:

```
build -a X64 -p OvmfPkg/OvmfPkgX64.dsc -D
SOURCE_DEBUG_ENABLE=TRUE -b DEBUG -m
MyDriver/MyDriver.inf
```

Файл /Build/OvmfX64/DEBUG_GCC5/X64/MyDriver.rom нужно преобразовать в .coe для того, чтобы вставить в проект.

7.2 Преобразование .rom в .coe

Перед тем как преобразовывать файл, следует ознакомиться с самим форматом файла, это можно сделать на официальном сайте AMD COE File Syntax. На питоне был написан преобразователь из rom в coe, так как питоном владею через гугл, будьте аккуратны в его использовании, не предусмотрен момент если rom содержит нечётное количество элементов или не делится ровно на 8. Также хочу обратить внимание что байты из файла rom следует перевернуть, если открыть с помощью vim'a файл то можно заметить расположение значений в hex формате.

Например 55aa0a00(rom) -> 000aaa55 (coe).

7.3 Запихиваем файл .coe в проект

В данном разделе добавим наш coe файл в память которую мы сгенерировали.

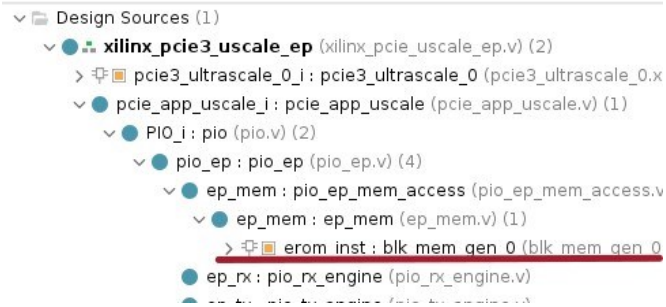


Рис. 16: Путь к нашей памяти.

Во вкладке Other Options добавляем память.

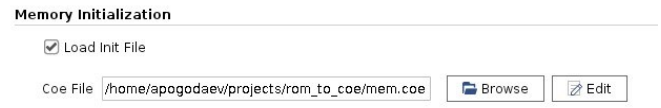


Рис. 17: Добавляем файл.

Далее можно сделать проверку нашего файла, для этого нажимаем Edit в открывшемся окне нажимаем Validate.

Рис. 18: Проверка пройдена.

Теперь можно собрать проект и обьяснить как это сделано на тестовом стенде.

8 Тестовый стенд

8.1 Два компьютера

Для проверки на железе нам понадобится два ПК, в одном из которых будет наша отладка. Один компьтер отвечает за принятие данных и прошивку ПЛИС, второй за обработку данных с платы. Также следует соединить COM порты на материнских платах и открыть терминал для вывода данных со второго компа.

8.2 Настройка UEFI на отладочном стенде

В настройках UEFI следует указать, чтоб весь вывод дублировался на COM порт. В моем случае это <Компоненты> -> Удаленный терминал [X] (да, у меня UEFI на русском).

8.3 Результат

Если всё сделано верно, то отладочный стенд передаст следующее:

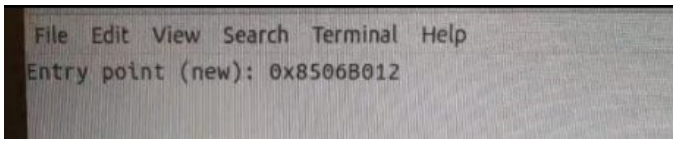


Рис. 19: Результат.

Видно, что вывод соответствует нашему драйверу в файле MyDriver.c строка 106.

```
DEBUG ((EFI_D_INFO, "Entry point (new): 0x%p\n",
My_driver_entry_point));
```

Также можно посмотреть весь список драйверов в UEFI. В моём случае это Загрузка ОС -> Быстрая загрузка -> EFI USB Device.

```
Shell> fs0:
FS0:\> drivers
```

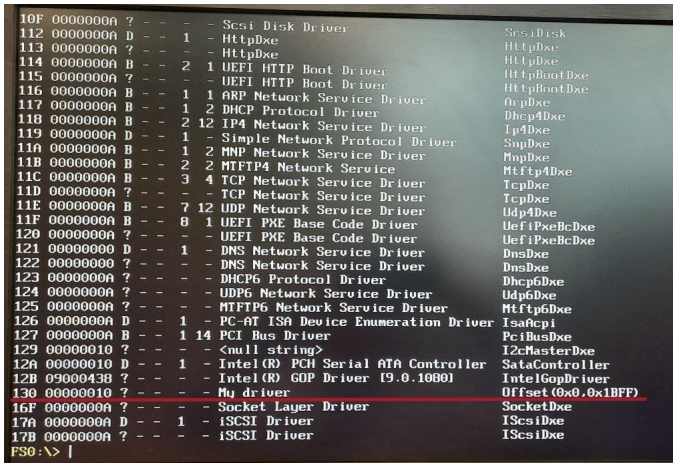


Рис. 20: Список драйверов UEFI.

Также можно проверить в самой операционной системе, для этого надо в терминале сделать следующее.

```
cd /sys/bus/pci/devices/0000\:01\:00.0
echo 1 > rom
dd if=rom of=/home/no/rom.rom
```

Тут можно сравнить rom файл после генерации и как он записался в системе.

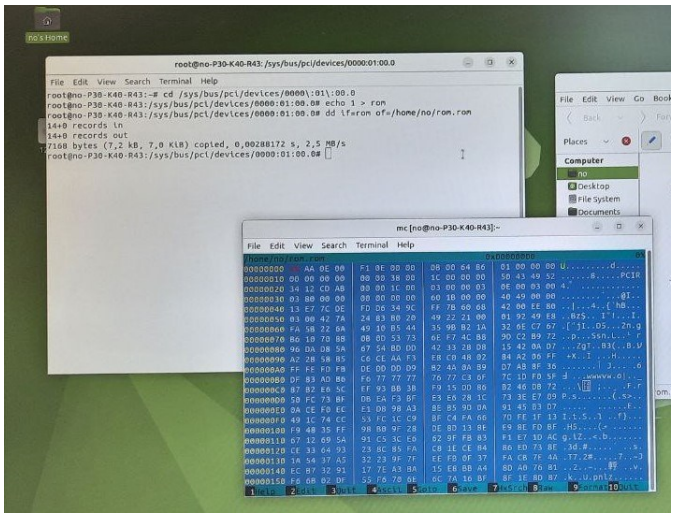


Рис. 21: Результат.

9 Вывод

В данной статье мы разобрали как можно не боятся и модернизировать пример проекта Xilinx. Как сконфигурировать драйвер и из него получить ROM файл из edk. Преобразовать файл в нужный нам формат и добавить в проект. Спасибо Александру Абрамову за помощь в части драйвера и другое.

9.1 Послесловие

Ребята, если у вас есть проекты, любой сложности, не стесняйтесь и записывайте это всё и выкладывайте в общий доступ, помогайте русскому сообществу развиваться, любой проект может внести вклад в общее дело или даже вдохновить кого-то, как это было со мной. Так как это моя первая статья, не знаю, насколько можно взять её за пример, но я не против.

Список используемой литературы

1. Expansion ROM карт PCI - "Структура EROM"
2. EDK - "Раздел Драйвер"
3. Установка EDK - "Раздел Драйвер"
4. Компилирование - "Раздел Драйвер"
5. How to run OVMF - "Раздел Драйвер"
6. How to debug OVMF with QEMU using GDB - "Раздел Драйвер"
7. Исходники проекта - "Драйвер и питоновский скрипт"

Реализация IQ-модулятора для ПЛИС

Кузнецов Данила
Telegram: @imdanilesko
E-mail: kyz324@gmail.com

Обсуждение и комментарии: link

Введение

В данной заметке рассмотрена реализация квадратурного модулятора на языке Verilog, с возможностью формирования фазомодулированных (ФМ) и линейно-частотно модулированных (ЛЧМ) сигналов на несущей частоте. Представленный модулятор может быть использован для ЦАП, принимающих данные в незнаковом формате.

Описанный блок формирует модулированный сигнал на несущей частоте и позволяет настраивать следующие характеристики сигнала:

- Амплитуды, начальные фазы и значение смещения по напряжению каждой квадратуры;
- Частоту несущей;

Теоретические основы

Данный раздел описывает связи характеристик генерируемого сигнала с входными параметрами модуля.

Модулированный сигнал на несущей частоте представляет собой произведение высокочастотного сигнала и комплексной огибающей – модулированного сигнала на нулевой частоте. Для удобства описания дальнейших формул представим их в экспоненциальной форме:

$$s_m(t) = e^{j(\omega_c t + \phi_0)} * g(t)$$

где *g(t)* – комплексная огибающая, вид которой зависит от вида модуляции, которая будет применяться. Для бинарного фазомодулированного сигнала комплексную огибающую можно представить следующим выражением:

$$g(t) = e^{j\pi W(t)}$$

где *W(t)* – информационный сигнал, последовательность нулей и единиц. Для удобства реализации модулятора и возможности реализации нескольких видов модуляции, в качестве входных данных будет использоваться не информационный сигнал, а его преобразованный вид – фазовая траектория (обозначим её как $\phi(t)$). То есть входными данными будет фаза комплексной огибающей. Таким образом, модулированный сигнал представляется в

виде:

$$s_m(t) = e^{j(\omega t+\phi_0)} * e^{j\phi(t)} = e^{j(\omega t+\phi_0+\phi(t))}$$

Представим в квадратурном виде:

$$I = \cos(\omega t + \phi_0 + \phi(t))$$

$$Q = \sin(\omega t + \phi_0 + \phi(t))$$

Очевидна необходимость вычисления синуса или косинуса. Самый простой способ – это сформировать файл, содержащий значения семплов синуса. Адресами для данного файла будут значения фазы. Количество семплов на период синуса задаст значение дискрета фазы, определяющее точность выставления фазы для квадратур. В данном модуляторе использовался файл на 1024 семпла, что соответствует дискрету фазы в $2\pi/1024$. Амплитуда синуса задаётся в соответствии с разрядностью ЦАП. В данной задаче разрядность ЦАП была 10-бит, что соответствует диапазону от 0 до 1023. Далее представлен скрипт для MATLAB, для формирования подобного файла. Файл хранится в BRAM, с двумя входами для чтения адреса и двумя выходами для значений синуса.

```
sin_lut = 'sin_lut.txt';
max = 1023;
min = 0;
step = 1024;
amp = (max-min)/2;
shift = (max+min)/2;
x = 0:2*pi/step:2*pi-2*pi/step;
y = shiftdim(round(amp*sin(x)+shift),1);
writematrix(y,sin_lut);
```

Листинг 1

С использованием входных параметров модуля:

- phase_zero – значения начальной фазы квадратур;
- phase_mod – значения модулирующей фазы;
- zero_lvl – значение напряжения постоянного смещения (как правило в самом ЦАП, либо на его выходе находится схема убирающая смещение по напряжению, преобразуя униполярный сигнал в биполярный.
- amp_coef – амплитудный коэффициент для регулировки уровня сигнала
- step – шаг перестроения дискрета фазы несущей частоты;

представленные зависимости преобразуются к следующему виду.

$$\phi_0 = \frac{2\pi}{1024} * phase_{zero}; \phi(t) = \frac{2\pi}{1024} * phaze_{mod}; \omega t = \frac{2\pi}{1024} step * t$$

Параметр «step» задаёт на сколько дискретов за такт будет меняться фаза, связывая частоту тактового сигнала с несущей частотой модулированного сигнала. Таким образом, квадратуры будут описываться следующими формулами:

$$I = AMP\_COEF\_I * \sin\left(\frac{2\pi}{1024} * PHASE\_ZERO\_I + \frac{2\pi}{1024} * STEP * t + \frac{2\pi}{1024} * PHASE\_MOD\_I\right) + LVLSET\_I$$

$$Q = AMP\_COEF\_Q * \sin\left(\frac{2\pi}{1024} * PHASE\_ZERO\_Q + \frac{2\pi}{1024} * STEP * t + \frac{2\pi}{1024} * PHASE\_MOD\_Q\right) + LVLSET\_Q$$

где «lvlset» - дополнительное слагаемое необходимое для позиционирования сигналов по сетке напряжения, высчитываемое из входного параметра «zero_lvl».

Реализация

В основе расчёта фаз лежит счётчик, начальное значение которого равно входному параметру «phase_zero», а инкремент параметру «step».

```
always @(posedge clk_in or negedge rst_in)
begin
  if ( ~rst_in )
  begin
    addr_I <= phase_zero_I;
    addr_Q <= phase_zero_Q;
  end
  else    if ( ~en_in )
  begin
    addr_I <= phase_zero_I;  //Задание
начальной фазы
```

```
    addr_Q <= phase_zero_Q;
  end
  else
  begin
    addr_I <= addr_I + step; //С каждым
такотом фаза инкрементируется на заданный шаг
    addr_Q <= addr_Q + step;
  end
end
```

Листинг 3

Затем значения ограничиваются по амплитуде и регулируется смещение по напряжению. Для амплитудного коэффициента используется представление в формате «fix-point», где самый старший бит соответствует 0 степени двойки, а самый младший $2^{-(N-1)}$, где N – разрядность коэффициента (то есть максимальное значение для 8 битного коэффициента 8'b10000000). После умножения необходимо сохранить разрядность сигнала.

```
//Сокращаем амплитуду за счёт умножения на
амплитудный коэффициент
wire [BIT_WIDTH+AMP_WIDTH-1:0] I_mull =
I_sin*amp_coef_I;
wire [BIT_WIDTH+AMP_WIDTH-1:0] Q_mull =
Q_sin*amp_coef_Q;

reg  [BIT_WIDTH-1:0] I_mull_reg, Q_mull_reg;
//Сокращаем разрядность за счёт конкатенации
always @(posedge clk_in or negedge rst_in)
begin
  if ( ~rst_in )
  begin
    I_mull_reg <= 0;
    Q_mull_reg <= 0;
  end
  else
  begin
    I_mull_reg <= I_mull[BIT_WIDTH+AMP_WIDTH-
1:AMP_WIDTH-1];
    Q_mull_reg <= Q_mull[BIT_WIDTH+AMP_WIDTH-
1:AMP_WIDTH-1];
  end
```

Листинг 4

Параллельно с умножением на коэффициент, рассчитывается требуемое смещение по напряжению. Итоговый выходной сигнал является результатом сложения рассчитанного смещения с результатом произведения.

```
/*Строка ((amp_coef_I<<BIT_WIDTH)-amp_coef_I)
аналогична умножению текущего амплитудного
коэффициента на максимальное значение сигнала
(2^BIT_WIDTH)-1.
Далее сокращаем разрядность после умножения и
дополнительно делим на 2 за счёт сдвига на
значение разрядности амплитудного
коэффициента.
В конце прибавляем единицу совершая
округление в большую сторону*/
wire [2*BIT_WIDTH:0] mid_lvl_I_rash =
((amp_coef_I<<BIT_WIDTH)-amp_coef_I)>>
(AMP_WIDTH);
wire [BIT_WIDTH-1:0] mid_lvl_I =
mid_lvl_I_rash+1;
wire [2*BIT_WIDTH:0] mid_lvl_Q_rash =
((amp_coef_Q<<BIT_WIDTH)-amp_coef_Q)>>
(AMP_WIDTH);
wire [BIT_WIDTH-1:0] mid_lvl_Q =
mid_lvl_Q_rash+1;

/*Рассчитываем значение добавки к сигналу как
разность требуемого уровня нуля синуса и
уровня нуля при текущем значении амплитудных
коэффициентов*/
wire [2*BIT_WIDTH-1:0] lvl_set_I = zero_lvl_I
- mid_lvl_I;
wire [2*BIT_WIDTH-1:0] lvl_set_Q = zero_lvl_Q
- mid_lvl_Q;

//Устанавливаем требуемый уровень нуля синуса
wire [BIT_WIDTH-1:0] I_wire =
I_mull_reg+lvl_set_I;
wire [BIT_WIDTH-1:0] Q_wire =
Q_mull_reg+lvl_set_Q;

/*Записываем итоговое значение сигнала в
выходные порты, если модулятор включен, либо
значения для нулевого уровня, если выключен*/
always @(posedge clk_in or negedge rst_in)
begin
  if (~rst_in)
  begin
    I <= 0;
    Q <= 0;
  end
  else
    if ( en_in )
    begin
      I <= I_wire;
      Q <= Q_wire;
    end
    else
    begin
      I <= zero_lvl_I;
      Q <= zero_lvl_Q;
    end
end
```

Листинг 5

Итоговая работа модуля

В рамках тестирования модуля генерируется несколько модулированных сигналов. Для фазовой модуляции задаются следующие входные параметры:

step = 1 – несущая частота сигнала будет равна частоте тактирования делённой на 1024;

phase_zero_I = 0; phase_zero_Q = 256 – Q-квадратура сдвинута по фазе относительно I-квадратуры на 90º;

zero_lvl = 512; amp_coef_I = amp_coef_Q = 8'h80 – максимальное значение амплитуды, смещение по напряжению соответствует середине диапазона напряжений ЦАП;

phase_mod принимает значения 0 и 512, что соответствует фазам 0º или 180º;

Получаем на выходе QPSK сигнал (четырёхуровневая фазовая модуляция), показанный на рисунке 1.

Для генерации сигнала ЛЧМ и тестирования функционала по управлению амплитудой используются следующие входные параметры:

step = 0 – несущая частота равна нулю, как следствие управление фазой сигнала ведётся напрямую с помощью фазовой траектории phase_mod;

phase_zero_I = 0; phase_zero_Q = 256 – Q-квадратура будет сдвинута относительно I-квадратуры на 90º;

zero_lvl_I = 256; amp_coef_I = 8'h20 – нулевой уровень I разместим на четверти диапазона напряжений, а амплитуду уменьшим в 4 раза;

zero_lvl_Q = 128; amp_coef_Q = 8'h08 – нулевой уровень I разместим на 1/8 диапазона напряжений, а амплитуду уменьшим в 4 раза, а амплитуду уменьшим в 16 раз;

Итоговый ЛЧМ сигнал показан на рисунке 2:

Заключение

Описана реализация модуля квадратурного модулятора, который может быть использован для формирования сигнала в радиотехнических системах различного назначения: как для систем связи, так и радиолокационных. Использование фазовой траектории позволяет реализовать

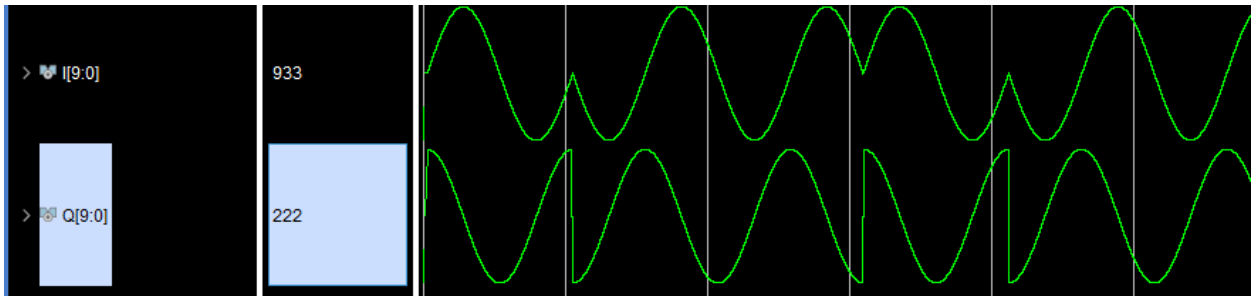


Рисунок 1: Фазомодулированный сигнал

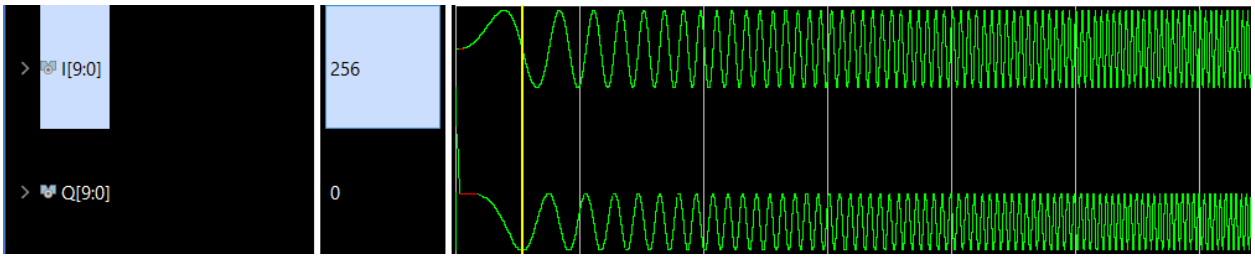


Рисунок 2: ЛЧМ сигнал

```
page <= 127;
```

множество модуляций, помимо представленных. Из особенностей работы с модулем можно выделить требования к формированию той самой фазовой траектории, что может быть проблематично для систем связи, где информационное сообщение постоянно меняется, в отличие от радиолокационных систем, где можно заранее рассчитать фазу и записать на устройство. Для простых видов модуляций (фазовой, например) этот вопрос решается добавлением дешифратора, а, например, для когерентной частотной модуляции потребуется непрерывный расчёт, что потребует достаточных вычислительных мощностей. Ознакомится с полным кодом реализации можно пройдя github.com/danilesko/IQ_mod/

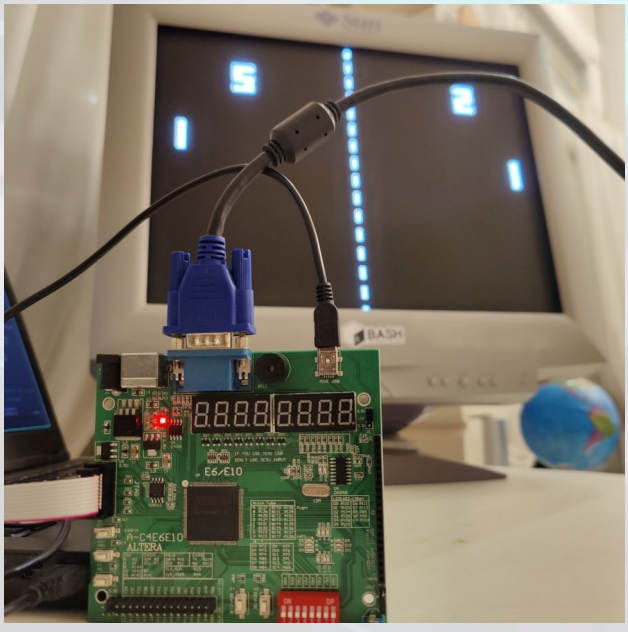
Игра в Pong на SystemVerilog

Кудинов Максим
Telegram: @m_kudinov

Обсуждение и комментарии: link

Аннотация

После написания Pong на C появилась идея реализовать аналог на SystemVerilog для FPGA. Интерес представляло сравнение подходов при программной и аппаратной реализации одной задачи. Логика игры довольно простая, так что написание подобного проекта может быть хорошим упражнением для начинающих, которые уже изучили основы программирования или описания цифровой логики.



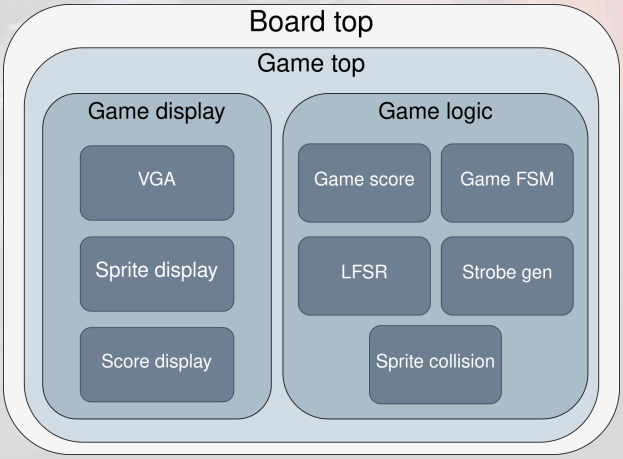
Pong на плате Zeowaa и мониторе Sun Microsystems

Введение

Игра заключается в том, что игрок с помощью двух кнопок (вверх и вниз) управляет ракеткой, пытаясь отбить шарик, а с противоположной стороны ракетка двигается автоматически. За каждый забитый шарик начисляется 1 очко. Побеждает тот, кто первый достигает определенного количества очков.

Для игры на плате используются 3 кнопки - 2 для движения и 1 для начала новой игры. Вывод изображения осуществляется через VGA.

Для удобства проект разбит на модули со следующей иерархией:



Иерархия модулей

Перед тем, как перейти к обзору каждого модуля, стоит поговорить о глобальных решениях.

Параметризация модулей

В SystemVerilog есть несколько методов задачи параметров: параметризация при объявлении модуля, header файл с **`define** и **package**.

Первый метод хорош, когда проект небольшой и параметров относительно мало, либо же требуется несколько инстансов с разными параметрами. В противном случае начинается путаница: либо задавать кучу параметров в top модуле и тащить их через весь проект к нужным модулям, либо параметризировать инстанс каждого отдельно, что долго и может привести к ошибкам, если что-то пропустить.

Гораздо удобнее, если все параметры будут в отдельном файле, который потом добавляется в нужные модули.

Первый способ так сделать - это файл с расширением **.svh**, который содержит в себе директивы компилятора **`define** и потом добавляется к нужным файлам проекта через **`include "filename"**. Пример такого файла **config.svh**:

```
`ifndef CONFIG_SVH
`define CONFIG_SVH

`define CONST_0 0
`define CONST_1 1

`endif
```

Первые две и последняя строчки называются include guards и нужны для того, чтобы компилятор выполнил содержимое файла только 1 раз в случае, если в проекте несколько **`include** одного header файла.

После чего в другом файле можно будет импортировать header и использовать эти параметры.

```
`include "config.svh"

module example_conf;
    initial begin
        $(display("%d, %d",
        `CONST_0, `CONST_1);
    end
endmodule
```

Проблема такого подхода в том, что все `define сразу окажутся в локальном именном пространстве, что может быть неудобно, если нам нужен всего один параметр из config.svh, а другие имена конфликтуют с локальными.

Для решения этих проблем в SystemVerilog есть **package**. **Package** представляет из себя именное пространство, которое может быть передано другим модулям. Пример простого **package** в проекте:

```
`ifndef BOARD_PKG_SVH
`define BOARD_PKG_SVH

package board_pkg;
    parameter KEYS_W    = 3;
    parameter LEDS_W    = 2;
    parameter VGA_RGB_W = 3;
endpackage : board_pkg

`endif

`include "board_pkg.svh"
module board_top
    import board_pkg::*;
(
// IO list
```

Не для всех САПР нужно помещать **package** в header и добавлять через **include**, а можно сразу переходить к объявлению **import**. Однако в стандарте языка сказано, что компиляция **package** должна происходить до компиляции контекста, в котором он используется [1], так что без **`include**, например, Verilator работать не будет [2].

Можно импортировать либо все содержимое **package** через **package_name::***, либо добавлять поименно через **package_name::parameter_name**, также можно разбивать параметры на несколько **package**.

В данном случае было принято решение разбить параметры на 5 различных **package**,

которые будут находиться в директории include.

```
include/
  sprite_pkg.svh
  score_pkg.svh
  board_pkg.svh
  lfsr_pkg.svh
  vga_pkg.svh
```

Автоматический сброс при загрузке на плату

При загрузке дизайна на FPGA, по умолчанию все регистры сбрасываются в 0 (в случае с Quartus), так что для корректной игры приходилось после загрузки каждый раз вручную нажимать сброс, чтобы выставить регистры в правильное начальное состояние. Но процесс сброса при старте можно автоматизировать с помощью initial и нехитрой логики.

Ключевое слово initial используется не только для верификации, в случае FPGA оно может задавать начальное значение регистров, которое они получат во время прошивки платы.

Так что можно завести отдельный регистр для сброса, который изначально будет активен, а потом отключится по положительному фронту тактового сигнала.

```
logic rst;
logic upload_rst_n;

// Сброс после загрузки
initial begin
    upload_rst_n = '0;
end

always_ff @(posedge clk_i)
    upload_rst_n <= '1;

assign rst = ~rst_n_i || ~upload_rst_n;
```

У меня на плате кнопки с активным нулем, так что для регистра я сохранил эту логику.

Теперь перейдем к рассмотрению самых интересных частей в модулях, начнем с самого верхнего – модуля board top.

Board top

Вся суть этого модуля в абстрагировании конкретных сигналов с платы от логики игры: сделать сброс, кнопки и светодиоды активными в 1, а не в 0.

На вход поступает тактовый сигнал и кнопки, на выход – сигналы VGA, параметры задаются через board_pkg

```
`include "board_pkg.svh"

module board_top
    import board_pkg::*;
(
    input  logic                  clk_i,
    input  logic                  rst_n_i,
    input  logic [   KEYS_W-1:0] keys_i,
    output logic [   LEDS_W-1:0] leds_o,
    output logic [VGA_RGB_W-1:0] vga_rgb_o,
    output logic                  vga_vs_o,
    output logic                  vga_hs_o
);
```

Далее идет автоматический сброс, который уже был показан выше, после чего инвертируем сигналы.

```
logic [KEYS_W-1:0] keys;
logic [LEDS_W-1:0] leds;

assign keys   = ~keys_i;
assign leds_o = ~leds;
```

Ну и в конце инстанцируем модуль с самой игрой, куда передаем уже инвертированные сигналы.

```
game_top i_game_top (
  .clk_i     ( clk_i     ),
  .rst_i     ( rst       ),
  .keys_i    ( keys      ),
  .leds_o    ( leds      ),
  .vga_rgb_o ( vga_rgb_o ),
  .vga_hs_o  ( vga_hs_o  ),
  .vga_vs_o  ( vga_vs_o  )
);
```

Game top

Глобально игра делится на 2 части: изменение параметров игровых объектов и

их вывод на экран. За первое отвечает модуль game logic, за второе game display.

Game logic ведет подсчет очков и изменение координат спрайтов (спрайт – небольшое изображение в игре, в нашем случае ракетки и шарик), которые потом передает в game display для вывода.

Для отрисовки спрайта нужно 4 параметра: позиция по X (левый край), позиция по Y (верх), правый край (X + ширина), низ (Y + высота). Если просто писать это как input/output logic, то для трех спрайтов придется передать 12 сигналов.

SystemVerilog позволяет собрать сигналы в структуру.

```
typedef struct packed {
  logic [X_POS_W-1:0] x_pos;
  logic [Y_POS_W-1:0] y_pos;
  logic [X_POS_W-1:0] right;
  logic [Y_POS_W-1:0] bottom;
} sprite_t;
```

Теперь мы можем создавать сразу целый спрайт с помощью типа sprite_t, но его нельзя просто передать как input. Для этого можно использовать interface.

```
`include "sprite_pkg.svh"

interface sprite_if;
    import sprite_pkg::sprite_t;
    sprite_t sprite;

    modport logic_mp (
        output  sprite
    );

    modport display_mp (
        input  sprite
    );

endinterface : sprite_if
```

С помощью интерфейса можно передавать сразу несколько сигналов, в том числе структуру. Конструкция modport задает направление сигнала с точки зрения модуля: game logic передает спрайт в game display.

Для счета тоже создается своя структура и интерфейс, потому что счет, помимо значения, тоже имеет свои координаты на экране.

Поскольку спрайтов всего 3, их можно объединить в массив структур, и тогда мы просто передаем между модулями массив sprites, вместо 12 отдельных сигналов.

```
logic     new_frame;

sprite_if sprites [N_SPRITES] ();
score_if  score               ();

game_logic i_game_logic (
    .clk_i       ( clk_i     ),
    .rst_i       ( rst_i     ),
    .keys_i      ( keys_i    ),
    .new_frame_i ( new_frame ),
    .leds_o      ( leds_o    ),
    .sprites_o   ( sprites   ),
    .score_o     ( score     )
);

game_display i_game_display (
    .clk_i       ( clk_i     ),
    .rst_i       ( rst_i     ),
    .vga_hs_o    ( vga_hs_o  ),
    .vga_vs_o    ( vga_vs_o  ),
    .vga_rgb_o   ( vga_rgb_o ),
    .new_frame_o ( new_frame ),
    .sprites_i   ( sprites   ),
    .score_i     ( score     )
);
```

Еще модули связаны сигналом new_frame, но о его значении немного позже.

Game logic

Этот модуль отвечает за перемещение ракеток и шарика, определение столкновений между шариком и ракеткой, обновление скорости и направления шарика, подсчета очков.

Новые значения сначала считаются в блоке always_comb, а потом записываются в регистр в always_ff. Так что мы используем две группы спрайтов – комбинационные (w - wire) и регистры (r - register).

```
sprite_t player_w;
sprite_t enemy_w;
sprite_t ball_w;

sprite_t player_r;
sprite_t enemy_r;
sprite_t ball_r;
```

Ракетки фиксированы по координате X, изменяется только Y. Если соответствующая кнопка нажата, и при этом ракетка не выходит за пределы поля, то мы изменяем координаты:

```
// Двигаемся вниз
if (key_down &&
   (player_r.y_pos < DOWN_LIMIT))
    player_w.y_pos = player_r.y_pos + 1'b1;

// Двигаемся вверх
if (key_up &&
   (player_r.y_pos > SCREEN_BORDER))
    player_w.y_pos = player_r.y_pos - 1'b1;
```

Дальше стояла задача написать логику перемещения “компьютера”, против которого мы и будем играть. Сделать, что координата Y ракетки равна координате Y шарика было бы слишком просто, да и играть в игру, в которую нельзя выиграть просто не интересно.

Самое просто решение - определить положение шарика относительно центра ракетки. И двигаться в этом направлении с заданной скоростью. Таким образом, обычно “компьютер” будет успевать догнать шарик и отбить, но не всегда.

```
// Если шарик выше - поднимаемся
if ((enemy_center > ball_r.y_pos) &&
    (enemy_r.y_pos > SCREEN_BORDER))
     enemy_w.y_pos = enemy_r.y_pos - 1'b1;

// Если шарик ниже - опускаемся
if ((enemy_center < ball_r.y_pos) &&
    (enemy_r.y_pos < DOWN_LIMIT))
     enemy_w.y_pos = enemy_r.y_pos + 1'b1;
```

Скорость шарика состоит из вертикальной и горизонтальной. Для простоты старший бит отвечает за направление, а остальные - за модуль скорости.

На основе старшего бита значение скорости либо вычитается, либо прибавляется к координатам. Если шарик выходит за пределы поля, то его координаты меняются на центр экрана.

```
// Если старший бит 1 - двигаемся влево
if (ball_speed_x[SPEED_W-1])
    ball_w.x_pos = ball_r.x_pos -
    X_POS_W' (ball_speed_x[SPEED_W-2:0]);
// Иначе вправо
else
    ball_w.x_pos = ball_r.x_pos +
    X_POS_W' (ball_speed_x[SPEED_W-2:0]);

// Вертикальное движение аналогично

// Если шарик за пределами экрана -
// возвращаем на центр
if ((ball_r.x_pos > SCREEN_H_RES -
                    SCREEN_BORDER) ||
    (ball_r.x_pos < SCREEN_BORDER))
begin

    ball_w.x_pos = X_POS_W'(SCREEN_H_RES/2);
    ball_w.y_pos = Y_POS_W'(SCREEN_V_RES/2);
end
```

Как можно заметить, у шарика есть отдельный регистр со значением скорости, а у ракеток мы изменяем координаты на 1 пиксель. Сделано это по той причине, что у ракеток есть более точный параметр скорости. Если задавать скорость через изменение пикселей, то появляется проблема того, что градация скоростей слишком крупная, то есть изменение скорости на 1 пиксель за раз слишком медленно, а изменение на 5 пикселей уже слишком быстро. И точно подстроить значение скорости не получается.

Для ракеток эту проблему можно решить, задав строб нужной частоты, который будет служить сигналом для обновления координат. В таком случае частота строба и будет соответствовать значению скорости.

Strobe gen

```
// Вычисляем значение для счетчика, до
// которого нужно досчитать перед стробом
localparam CNT_MAX = BOARD_CLK_MHZ *
           1_000_000     /
           STROBE_FREQ_HZ;

localparam CNT_W   = $clog2(CNT_MAX + 1);
logic [CNT_W-1:0] cnt;

always_ff @(posedge clk_i)
  if (rst_i)
    cnt <= '0;
```

```
// По стробу сбрасываем в 0
  else if (strobe)
    cnt <= '0;
  else
    cnt <= cnt + CNT_W'(1);
// Когда досчитали - отправляем строб
assign strobe = cnt == CNT_W' (CNT_MAX);
```

После чего в game logic создаем инстанс для скорости игрока и “компьютера”.

```
strobe_gen #(
        .BOARD_CLK_MHZ  ( BOARD_CLK_MHZ ),
        .STROBE_FREQ_HZ ( PLAYER_SPEED  )
    ) i_player_strobe_gen (
        .clk_i          ( clk_i         ),
        .rst_i          ( rst_i         ),
        .strobe         ( update_player )
    );

// Аналогично для enemy
```

Почему же нельзя так же сделать с шариком? Проблема в том, что перемещение по двум осям получается крайне дерганным (вместо перемещения по диагонали, шарик сначала перескочит по X, а потом через несколько кадров по Y). Но ракетки перемещаются только по Y, поэтому у них такой проблемы нет.

Далее в game logic мы вычисляем правую и нижнюю части спрайтов

```
assign player_w.right  = player_w.x_pos +
                X_POS_W' (PADDLE_WIDTH);
assign player_w.bottom = player_w.y_pos +
               Y_POS_W' (PADDLE_HEIGHT);

// Аналогично для enemy и ball
```

И, наконец, записываем все изменения в регистры.

```
always_ff @(posedge clk_i)
if (rst_i) begin
    player_r     <= INIT_ST_P;
    enemy_r      <= INIT_ST_E;
    ball_r       <= INIT_ST_B;
end else begin
    if (~game_en)
        ball_r   <= INIT_ST_B;
    else begin
    if (update_player)
        player_r <= player_w;

    if (update_enemy)
        enemy_r  <= enemy_w;

    if (new_frame_i)
        ball_r   <= ball_w;
    end
end
```

За счет того, что мы использовали **packed struct**, мы можем просто присвоить одну структуру другой, вместо того, чтобы перечислять поля в явном виде.

Тут и пригодился сигнал **new_frame_i**, который я упоминал в модуле game_top. Он служит в роли вертикальной синхронизации, чтобы координаты шарика обновлялись только после смены кадра, а не во время отрисовки части шарика, из-за чего визуально происходит “разрыв” изображения.

Далее рассмотрим механику столкновений шарика с ракеткой.

Sprite collision

Идея этого модуля была позаимствована из лабораторной работы Школы Синтеза Цифровых Схем [3].

Модуль получает на вход 2 спрайта и определяет, есть ли у них общие точки. Спрайты столкнулись, когда правые стороны обоих спрайтов находятся правее левых сторон (сравниваем стороны разных спрайтов), и обе верхние стороны выше, чем нижние.

```
// x_pos - лево, y_pos - верх
always_ff @(posedge clk_i)
if (rst_i)
    collision_o <= '0;
else
    collision_o <=
    ((rect_1.right  > rect_2.x_pos) &&
     (rect_2.right  > rect_1.x_pos) &&
     (rect_2.bottom > rect_1.y_pos) &&
     (rect_1.bottom > rect_2.y_pos));
```

Чтобы сделать игру более интересной, у ракетки есть 3 хитбокса (зоны попадания), в зависимости от сработавшего хитбокса шарик ведет себя по-разному.

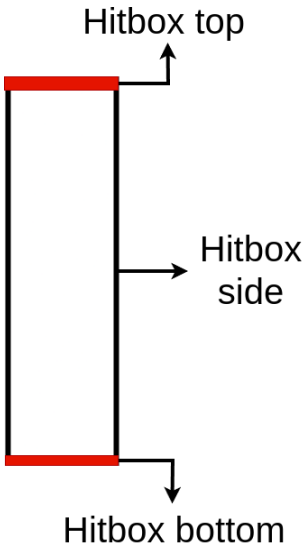


Схема хитбоксов

Итого на 2 ракетки выходит 6 хитбоксов. В этом случае также можно объединить сигналы в несколько структур.

```
sprite_t [N_HITBOXES-1:0] p_hitboxes;
sprite_t [N_HITBOXES-1:0] e_hitboxes;
```

После чего создадим интерфейсы для передачи структур в модуль коллизии.

```
sprite_if p_hit [N_HITBOXES] ();
sprite_if e_hit [N_HITBOXES] ();
sprite_if b_hit              ();
```

А далее изменим стороны изначального спрайта так, чтобы получилось 3 разных хитбокса: один без изменений для коллизии стороной, у второго изменим координату его верхней стороны на bottom - 1, что соответствует нижнему торцу, а у третьего сделаем низ как y_pos + 1, что станет верхним торцом.

```
always_comb begin
  p_hitboxes = { N_HITBOXES { player_r } };
  e_hitboxes = { N_HITBOXES { enemy_r  } };

  b_hit.sprite = ball_r;

  p_hitboxes[2].y_pos = player_r.bottom-1'b1;
  e_hitboxes[2].y_pos = enemy_r.bottom -1'b1;

  p_hitboxes[1].bottom= player_r.y_pos+ 1'b1;
  e_hitboxes[1].bottom= enemy_r.y_pos + 1'b1;
end
```

Ну и вот теперь все готово, чтобы отправить хитбоксы в **generate**, который сделает необходимое количество проверок на столкновение.

```
genvar i;
generate
    for (i = 0; i < N_HITBOXES; i++) begin :
collision_player
      assign p_hit[i].sprite = p_hitboxes[i];

      sprite_collision i_player_collision (
        .clk_i          ( clk_i            ),
        .rst_i          ( rst_i            ),
        .rect_1_i       ( p_hit        [i] ),
        .rect_2_i       ( b_hit            ),
        .collision_o    ( player_colls [i] )
        );
    end
// То же самое для ракетки “компьютера”
endgenerate
```

На выходе получаем 2 массива сигналов: player_colls и enemy_colls, которые нужны для определения направления и скорости шарика.

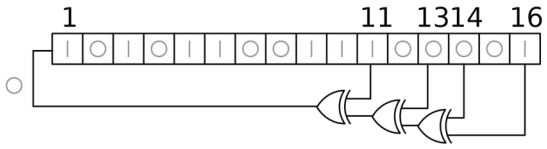
```
// Столкновение с боком ракетки игрока
if (player_colls[0]) begin
    // Направить шарик влево
    ball_speed_x_w[SPEED_W-1]   = 1'b1;
    // Задать основу скорости
    ball_speed_x_w[SPEED_W-2:1] =
                          DEFLECT_SPEED_X;
    // Добавить немного случайности
    ball_speed_x_w[0]           = rnd_num[0];
    // По вертикали направление не меняем
    ball_speed_y_w[SPEED_W-2:0] =
                          DEFLECT_SPEED_Y;
    ball_speed_y_w[1]           = rnd_num[1];
end

// Аналогично для столкновения с
// боком ракетки “компьютера”

// Если ракетка движется вверх и
// задевает шарик верхним торцом -
// направить шарик вверх и присвоить
// большую вертикальную скорость
if ((key_up && player_colls[1]) ||
    enemy_colls[1]) begin

    ball_speed_y_w[SPEED_W-1]   = 1'b1;
    ball_speed_y_w[SPEED_W-2:0] =
                        SIDE_HIT_SPEED_Y;
end

// Аналогично для столкновения с низом

// Если шарик вышел за пределы поля,
```

```
// то инициализировать скорость
if ((ball_r.x_pos >
            SCREEN_H_RES - SCREEN_BORDER) ||
    (ball_r.x_pos < SCREEN_BORDER)) begin
    ball_speed_x_w = { rnd_num[4],
                       2'b01,
                       rnd_num[7:6] };
    ball_speed_y_w = { rnd_num[5],
                       3'b000,
                       rnd_num[8]   };
end

// Если стукнулся об верх поля -
// направить вниз
if (ball_r.y_pos < SCREEN_BORDER)
    ball_speed_y_w[SPEED_W-1] = 1'b0;

// Аналогично для низа
```

Случайное число rnd_num берется из сдвигового регистра с обратной связью (Linear Feedback Shift Register - LFSR)

LFSR

Для простой генерации псевдослучайных чисел можно использовать LFSR. На вход регистра подается результат XOR между несколькими позициями, что приводит к генерации конечной и повторяемой последовательности, что, впрочем, вполне годится для случайных значений в игре.



LFSR [4]

```
parameter RND_NUM_W = 9;
parameter RND_SEED  = 1337;  // начальное
значение цепочки чисел
parameter TAPS      = 'h110; // позиции
регистра для XOR
```

```
always_ff @(posedge clk_i)
if (rst_i)
    rnd_num_o <= RND_NUM_W' (RND_SEED);
else
    rnd_num_o <= { rnd_num_o[RND_NUM_W-2:0],
^(rnd_num_o & TAPS) };
```

FSM

В игре есть очень простой конечный автомат, который ждет нажатия кнопки для начала игры, а потом ждет, пока кто-либо не наберет максимальное количество очков, после чего опять уходит в состояние ожидания игры.

```
always_comb
begin
  next = state;
  case (state)
    ST_WAIT_START:
      if (game_rst_i)
        next = ST_PLAY;
    ST_PLAY:
      if (p_score_i == MAX_SCORE ||
          e_score_i == MAX_SCORE)
        next = ST_WAIT_START;
  endcase
end

game_en_o <= (state == ST_PLAY);
```

Пока автомат в стадии игры, то он выставляет сигнал game_en_o, который используется в game_logic.

```
// Если игра перешла в активное состояние -
обнулить счет
end
else if (game_en & ~game_en_prev)
begin
  player_score_r <= '0;
  enemy_score_r  <= '0;
end
```

А сам счет вычисляется по координатам шарика.

```
// Если шарик коснулся правого края -
прибавить очко “компьютеру”
if (ball_r.x_pos > SCREEN_H_RES -
SCREEN_BORDER)
enemy_score_w = enemy_score_r + 1'b1;

// Если коснулся левого - прибавить
игроку
if (ball_r.x_pos < SCREEN_BORDER)
player_score_w = player_score_r + 1'b1;
```

Game score

Для счета был выбран шрифт 3x5 пикселей, что позволяет отобразить все цифры от 0 до 9. Чтобы счет было хорошо видно на экране, цифры надо увеличить. Простым вариантом является повторение битов сначала по горизонтали, а потом по вертикали.

```
parameter SCALE   = 10;
parameter SCORE_W = 3 * SCALE;
parameter SCORE_H = 5 * SCALE;
```

С помощью SCALE можно задавать нужный масштаб.

Пример записи цифры 0:

```
parameter [0:SCORE_H-1][0:SCORE_W-1] score0 =
          {
            { SCALE
              { {SCALE{1'b1}},
                {SCALE{1'b1}},
                {SCALE{1'b1}}
              }
            },
            { SCALE
              {
                {SCALE{1'b1}},
                {SCALE{1'b0}},
                {SCALE{1'b1}} }
            },
            { SCALE
              {
                {SCALE{1'b1}},
                {SCALE{1'b0}},
                {SCALE{1'b1}}
              }
            },
            { SCALE
              {
                {SCALE{1'b1}},
                {SCALE{1'b0}},
                {SCALE{1'b1}}
              }
            },
            { SCALE
              {
                {SCALE{1'b1}},
                {SCALE{1'b1}},
                {SCALE{1'b1}}
              }
            }
          };
```

Таким образом, получаем цифру 30 на 50 пикселей, которую можно выводить на экран.

Модуль просто выбирает нужную цифру, в зависимости от значения счета.

```
case (player_score_i)
  M_SCORE_W'(0): player_s.score_val = score0;
  M_SCORE_W'(1): player_s.score_val = score1;
  M_SCORE_W'(2): player_s.score_val = score2;
  M_SCORE_W'(3): player_s.score_val = score3;
  M_SCORE_W'(4): player_s.score_val = score4;
  // 5-9
  default:
    player_s.score_val = 'x;
```

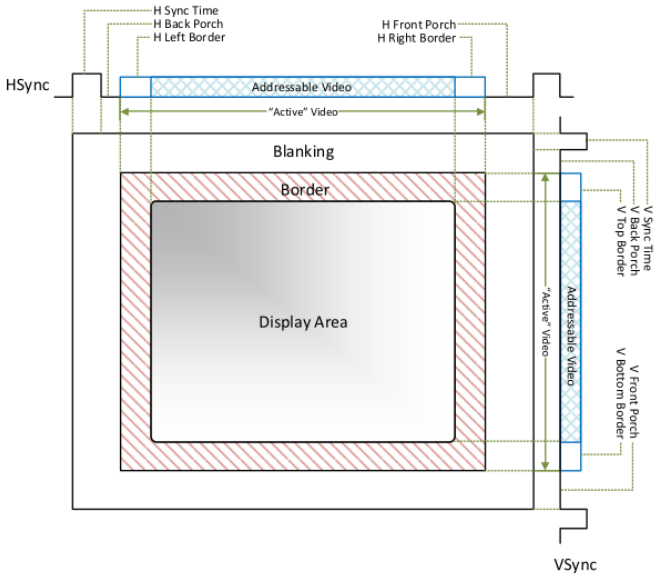
Ну вот и вся логика игры, осталось только вывести на экран.

VGA

Идея вывода изображения очень простая - перемещаемся по экрану сверху вниз и слева направо, выбирая по пикселю за раз и указывая его цвет.

VGA (Video Graphics Array) состоит из 5 сигналов: hsync для перехода на новую строку пикселей, vsync для возвращения на самый верх, и RGB для определения цвета.

Но если мы хотим вывести изображение, скажем, 640x480, то на самом деле нам понадобятся дополнительные, “невидимые” столбцы и строки, как бы за пределами экрана. Они нужны для соблюдения тайминга стандарта.



Тайминг VGA [5]

Для определения точного количества необходимых линий, к нужному разрешению добавляется передний и задний порог (front/back porch), рамка (border) и длительность сигнала hsync/vsync. Эти значения определяются желаемым разрешением, их можно найти в таблице таймингов VGA [6].

```
// HSYNC
parameter HSYNC_START = SCREEN_H_RES  +
                        H_BORDER      +
                        H_FRONT_PORCH;

parameter HSYNC_END   = HSYNC_START   +
                        HSYNC_PULSE;

parameter H_TOTAL     = HSYNC_END     +
                        H_BACK_PORCH  +
                        H_BORDER;

// VSYNC
parameter VSYNC_START =  SCREEN_V_RES +
                         V_BORDER     +
                         V_FRONT_PORCH;

parameter VSYNC_END   = VSYNC_START   +
                        VSYNC_PULSE;

parameter V_TOTAL     = VSYNC_END     +
                        V_BACK_PORCH  +
                        V_BORDER;
```

С учетом этих дополнительных линий, получаем, что для разрешения 640x480 всего нужно считать 800 столбцов и 525 строк.

Переход с одного пикселя на следующий в строке происходит по сигналу pixel clock, частота которого по стандарту равна 25.175 MHz, но на практике 25MHz работает так же хорошо, а ее очень просто получить на FPGA плате с частотой 50 MHz через обычный счетчик.

```
always_ff @(posedge clk_i)
  if (rst_i)
    pixel_clk_cnt <= '0;
  else
    pixel_clk_cnt <= pixel_clk_cnt + 1'b1;

assign pixel_clk_en = pixel_clk_cnt ==
PX_CNT_W'((BOARD_CLK_MHZ/PIXEL_CLK_MHZ)-1);
```

По сигналу pixel_clock_en обновляем текущую позицию по горизонтали, а когда дойдем до края, то возвращаемся в 0.

```
assign h_cnt_max    = h_cnt == (H_TOTAL - 1);

always_ff @(posedge clk_i)
  if (rst_i)
    h_cnt     <= '0;
  else if (pixel_clk_en)
  begin
    h_cnt     <= h_cnt + 1'b1;

    if (h_cnt_max)
      h_cnt <= '0;
  end
```

Позицию по вертикали обновляем по pixel_clk_en и максимальному значению строки (h_cnt).

```
always_ff @(posedge clk_i)
  if (rst_i)
    v_cnt     <= '0;
  else if (pixel_clk_en && h_cnt_max)
  begin
    v_cnt     <= v_cnt + 1'b1;

    if (v_cnt == (V_TOTAL - 1))
      v_cnt <= '0;
  end
```

Ну и последнее, и самое главное - записать выходные значения в регистры. Если оставить выход асинхронным, то будут артефакты при выводе на экран.

```
always_ff @(posedge clk_i)
    if (rst_i) begin
        hsync_o         <= '0;
        vsync_o         <= '0;
        pixel_x_o       <= '0;
        pixel_y_o       <= '0;
        visible_range_o <= '0;
    // Обновляем значения по pixel clock
    end else if (pixel_clk_en) begin
        // Выводим сигналы (с активным 0),
        // когда находимся в области вывода
        hsync_o <=
                  ~(h_cnt >= HSYNC_START &&
                    h_cnt < HSYNC_END);

        vsync_o <=
                  ~(v_cnt >= VSYNC_START &&
                    v_cnt < VSYNC_END);

        pixel_x_o <= h_cnt;
        pixel_y_o <= v_cnt;

        // Для простоты других модулей
        // указываем, когда координаты
        // находятся в видимой области экрана
        visible_range_o <= (
                   (h_cnt < SCREEN_H_RES) &&
```

```
                    (v_cnt < SCREEN_V_RES));
    end
```

Sprite display

Для того, чтобы отобразить что-то на экране, надо понять, что pixel_x и pixel_y VGA находятся на этом объекте, и знать цвет, который нужно передать пикселю. В нашей игре все спрайты исключительно белые, так что все сводится к логике “если мы на спрайте - текущий пиксель белый”. Модуль принимает на вход интерфейс со структурой спрайта, и смотрит, находится ли текущий пиксель VGA в пределах спрайта.

```
always_comb
begin
  vga_rgb_w   = '0;
  on_sprite_w = '0;

  if (pixel_x_i > sprite.x_pos &&
      pixel_x_i < sprite.right &&
     (pixel_y_i > sprite.y_pos &&
      pixel_y_i < sprite.bottom )) begin
      vga_rgb_w   = '1;
      on_sprite_w = '1;
  end
end
```

Score display

Отображение счета работает похожим образом, но тут прямоугольник не весь залит белым, а только частично по квадратам. Так что теперь нужно проверять не только, что мы находимся в пределах спрайта, но и учитывать - черный это квадрат или белый.

```
vga_rgb_w  = '0;
on_score_w = '0;

if (pixel_x_i >= player_s.x_pos           &&
    pixel_x_i <  player_s.x_pos + SCORE_W &&
    pixel_y_i >= player_s.y_pos           &&
    pixel_y_i <  player_s.y_pos + SCORE_H &&
    player_s.score_val
    [pixel_y_i - player_s.y_pos]
    [pixel_x_i - player_s.x_pos]) begin
        on_score_w = '1;
        vga_rgb_w  = '1;
end
// Для enemy логика аналогичная
```

Game display

Game display получает на вход массив спрайтов и счет, после чего инстанцирует модули VGA, sprite display и score display.

Для компактной обработки всех спрайтов используется generate.

```
genvar i;
generate
  for (i = 0; i < N_SPRITES; i++)
  begin : sprite_display
    sprite_display i_player (
      .clk_i       ( clk_i          ),
      .rst_i       ( rst_i          ),
      .pixel_x_i   ( vga_x_pos      ),
      .pixel_y_i   ( vga_y_pos      ),
      .on_sprite_o ( on_sprite  [i] ),
      .vga_rgb_o   ( sprite_rgb [i] ),
      .sprite_i    ( sprites_i  [i] )
    );
  end
endgenerate
```

После чего выбираем, что отрисовать в текущий момент в зависимости от того, на какие элементы игры мы попали.

```
always_comb begin
    vga_rgb_w  = '0;

    if (vga_visible_range) begin
        // Если на счете
        if (on_score)
            vga_rgb_w = score_rgb;

        // Если на каком-то из спрайтов
        for (int n = 0; n < N_SPRITES; n++)
        begin
            if (on_sprite[n])
                vga_rgb_w = sprite_rgb[n];
        end

        // Если на разделителе по центру
        if (vga_x_pos >
           (SCREEN_H_RES/2-SEPARATOR_WIDTH/2)
            &&
            vga_x_pos <
           (SCREEN_H_RES/2+SEPARATOR_WIDTH/2)
            &&
           (vga_y_pos + 9 & 31) <
            SEPARATOR_DOT_HEIGHT)
        begin
                vga_rgb_w = '1;
        end
   end
end
```

Теперь игру можно выводить на монитор!

Заключение

Ну вот и все! Конечно, был рассмотрен далеко не весь код, но я постарался описать основные идеи, на которых строится игра.

Поначалу казалось, что сделать игру в логике будет намного сложнее, чем программируя на C, но по сути большинство идей абсолютно одинаковые, просто немного в другой обертке. Главными отличиями были отказ от умножения, чтобы сократить критический путь, и необходимость самому реализовывать генерацию псевдослучайных чисел и контроллер для вывода на экран.

Если вам интересен проект, то вы можете ознакомиться с полной версией кода на GitHub [7].

Источники

1. 1800-2023 - IEEE Standard for SystemVerilog --Unified Hardware Design, Specification, and Verification Language 26.3
2. https://github.com/verilator/verilator/issues/2890#issuecomment-820379954
3. https://github.com/chipdesignschool/basics-graphics-music/blob/main/labs/14_game/game_overlap.sv
4. https://en.wikipedia.org/wiki/Linear-feedback_shift_register
5. https://digilent.com/reference/learn/programmable-logic/tutorials/vga-display-congroller/start
6. http://www.tinyvga.com/vga-timing/640x480@60Hz
7. https://github.com/max-kudinov/pong_fpga

if if'y рознь. QUARTUS vs VIVADO. SystemVerilog vs VHDL

Мальчуков А.Н.

andrey@malchukov.ru

Обсуждение и комментарии: link

Введение

В операторе if проверяется условие, однако одно и тоже условие, можно описывать разными способами. На примере ограничения модуля счёта счётчика рассматриваются три способа описания одного и того же события с анализом занимаемых при этом ресурсов на различных кристаллах ПЛИС и в разных САПР. Кроме того, в этот раз будет показана разная интерпретация САПР Vivado одинаковых описаний на языках SystemVerilog и VHDL.

Описание проектов

В задачах реализации задержки, деления частоты или ШИМ необходимо оперировать модулем счёта счётчика. Ограничение это можно реализовать через оператор if, отслеживая нужный момент тремя способами: 1. сравнение текущего значения счётчика с константой заданного количества переключений счётчика с помощью компаратора (обычное сравнение – выбор программиста); 2. исключающее ИЛИ (xor) значения счётчика и константы с определением нулевого результата (привет Assembler); 3. отслеживание достижения единичных значений заданных разрядов счётчика через конъюнкцию (and). В статье рассматриваются примеры 32-х и 64-х разрядных счётчиков.

Листинги кодов проектов

Реализации способов 1 и 3 на языках SystemVerilog и VHDL для САПР Quartus и Vivado ничем не отличаются. Во втором способе есть отличия, которое заключается в том, что Quartus (20.1.1 и ниже) не поддерживает в VHDL 2008 многовходовые логические элементы (reduction), в отличие от Vivado (2019.1).

Счётчик с ограничением модуля счёта 81008082_{16} способом 1 на SystemVerilog в модуле cnt32sv:

```
module cnt32sv (input CLK,
output logic f);
logic [31:0] delay;
always_ff @(posedge CLK)
    if (delay == 32'h81008081) begin
        f <= 1'b1;
        delay <= '0;
    end
    else begin
        f <= 1'b0;
        delay <= delay + 1'b1;
    end
endmodule
```

Счётчик с ограничением модуля счёта 81008082_{16} способом 1 на VHDL (1993) в модуле cnt32vhdl:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity cnt32vhdl is
port(CLK: in STD_LOGIC;
f: out STD_LOGIC);
end;
architecture synth of cnt32vhdl is
signal delay: STD_LOGIC_VECTOR(31 downto 0);
begin
  process (CLK) begin
    if rising_edge (CLK) then
      if delay =
"10000001000000001000000010000001" then
        f <= '1';
        delay <= (OTHERS => '0');
      else
        f <= '0';
        delay <= delay + '1';
      end if;
    end if;
  end process;
end;
```

Счётчик с ограничением модуля счёта 81008082_{16} способом 2 на SystemVerilog в модуле cnt32sv2:

```
module cnt32sv2 (
input CLK,
output logic f);
logic [31:0] delay;
always_ff @(posedge CLK)
    if (!(delay ^ 32'h81008081)) begin
        f <= 1'b1;
        delay <= '0;
    end
    else begin
        f <= 1'b0;
        delay <= delay + 1'b1;
    end
endmodule
```

Счётчик с ограничением модуля счёта 81008082_{16} способом 2 на VHDL (1993) в модуле cnt32vhdl2:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity cnt32vhdl2 is
port(CLK: in STD_LOGIC;
f: out STD_LOGIC);
end;
architecture synth of cnt32vhdl2 is
signal delay, delxor: STD_LOGIC_VECTOR(31
downto 0);
begin
  delxor <= delay xor
"10000001000000001000000010000001";
  process (CLK) begin
    if rising_edge (CLK) then
      if (delxor(0) or delxor(5) or delxor
(10) or delxor(15) or delxor(20) or delxor
(24) or delxor(28)
       or delxor(1) or delxor(6) or delxor
(11) or delxor(16) or delxor(21) or delxor
(25) or delxor(29)
       or delxor(2) or delxor(7) or delxor
(12) or delxor(17) or delxor(22) or delxor
(26) or delxor(30)
       or delxor(3) or delxor(8) or delxor
(13) or delxor(18) or delxor(23) or delxor
(27) or delxor(31)
       or delxor(4) or delxor(9) or delxor
(14) or delxor(19)) = '0' then
        f <= '1';
        delay <= (OTHERS => '0');
      else
        f <= '0';
        delay <= delay + '1';
      end if;
    end if;
  end process;
end;
```

Счётчик с ограничением модуля счёта 81008082_{16} способом 2 на VHDL (2008 для САПР Vivado) в модуле cnt32vhdl2x:

```
library IEEE;use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity cnt32vhdl2x is
port(CLK: in STD_LOGIC;
f: out STD_LOGIC);
end;
architecture synth of cnt32vhdl2x is
signal delay, delxor: STD_LOGIC_VECTOR(31
downto 0);
begin
  delxor <= delay xor
"10000001000000001000000010000001";
  process (CLK) begin
    if rising_edge (CLK) then
      if (or delxor) = '0' then
        f <= '1';
        delay <= (OTHERS => '0');
      else
        f <= '0';
        delay <= delay + '1';
      end if;
    end if;
  end process;
end;
```

Счётчик с ограничением модуля счёта 81008082_{16} способом 3 на SystemVerilog в модуле cnt32sv3:

```
module cnt32sv3 (
input CLK,
output logic f);
logic [31:0] delay;
always_ff @(posedge CLK)
    if (delay[31] & delay[24]
      & delay[15] & delay[7] & delay[1])
    begin
        f <= 1'b1;
        delay <= '0;
    end
    else begin
        f <= 1'b0;
        delay <= delay + 1'b1;
    end
endmodule
```

Счётчик с ограничением модуля счёта 81008082_{16} способом 3 на VHDL (1993) в модуле cnt32vhdl3:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity cnt32vhdl3 is
port(CLK: in STD_LOGIC;
f: out STD_LOGIC);
end;
architecture synth of cnt32vhdl3 is
signal delay: STD_LOGIC_VECTOR(31 downto 0);
begin
  process (CLK) begin
    if rising_edge (CLK) then
      if (delay(31) and delay(24) and delay
(15) and delay(7) and delay(1)) = '1' then
        f <= '1';
        delay <= (OTHERS => '0');
      else
        f <= '0';
        delay <= delay + '1';
      end if;
    end if;
  end process;
end;
```

Счётчик с ограничением модуля счёта 8080810881008082_{16} способом 1 на SystemVerilog в модуле cnt64sv:

```
module cnt64sv (
input CLK,
output logic f);
logic [63:0] delay;
always_ff @(posedge CLK)
    if (delay == 64'h8080810881008081) begin
        f <= 1'b1;
        delay <= '0;
    end
```

Счётчик с ограничением модуля счёта 8080810881008082_{16} способом 1 на VHDL (1993) в модуле cnt64vhdl:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity cnt64vhdl is
port(CLK: in STD_LOGIC;
f: out STD_LOGIC);
end;
architecture synth of cnt64vhdl is
signal delay: STD_LOGIC_VECTOR(63 downto 0);
begin
  process (CLK) begin
    if rising_edge (CLK) then
      if delay =
"1000000010000000100000010000100010000001000000000100000010000001" then
        f <= '1';
        delay <= (OTHERS => '0');
      else
        f <= '0';
        delay <= delay + '1';
      end if;
    end if;
  end process;
end;
```

Счётчик с ограничением модуля счёта 8080810881008082_{16} способом 2 на SystemVerilog в модуле cnt64sv2:

```
module cnt64sv2 (
input CLK,
output logic f);
logic [63:0] delay;
always_ff @(posedge CLK)
    if (!(delay ^ 64'h8080810881008081))
begin
        f <= 1'b1;
        delay <= '0;
    end
    else begin
        f <= 1'b0;
        delay <= delay + 1'b1;
    end
endmodule
```

Счётчик с ограничением модуля счёта 8080810881008082_{16} способом 2 на VHDL (1993) в модуле cnt64vhdl2:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity cnt64vhdl2 is
port(CLK: in STD_LOGIC;
f: out STD_LOGIC);
end;
architecture synth of cnt64vhdl2 is
signal delay, delxor: STD_LOGIC_VECTOR(63
downto 0);
begin
  delxor <= delay xor
"1000000010000000100000010000100010000001000000
0000100000010000001";
  process (CLK) begin
    if rising_edge (CLK) then
      if (delxor(0 ) or delxor(13) or delxor
(26) or delxor(39) or delxor(52)
       or delxor(1 ) or delxor(14) or delxor
(27) or delxor(40) or delxor(53)
       or delxor(2 ) or delxor(15) or delxor
(28) or delxor(41) or delxor(54)
       or delxor(3 ) or delxor(16) or delxor
(29) or delxor(42) or delxor(55)
       or delxor(4 ) or delxor(17) or delxor
(30) or delxor(43) or delxor(56)
       or delxor(5 ) or delxor(18) or delxor
(31) or delxor(44) or delxor(57)
       or delxor(6 ) or delxor(19) or delxor
(32) or delxor(45) or delxor(58)
       or delxor(7 ) or delxor(20) or delxor
(33) or delxor(46) or delxor(59)
       or delxor(8 ) or delxor(21) or delxor
(34) or delxor(47) or delxor(60)
       or delxor(9 ) or delxor(22) or delxor
(35) or delxor(48) or delxor(61)
       or delxor(10) or delxor(23) or delxor
(36) or delxor(49) or delxor(62)
       or delxor(11) or delxor(24) or delxor
(37) or delxor(50) or delxor(63)
       or delxor(12) or delxor(25) or delxor
(38) or delxor(51)) = '0' then
        f <= '1';
        delay <= (OTHERS => '0');
      else
        f <= '0';
        delay <= delay + '1';
      end if;
    end if;
  end process;
end;
```

Счётчик с ограничением модуля счёта 8080810881008082_{16} способом 2 на VHDL (2008 для САПР Vivado) в модуле cnt64vhdl2x:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity cnt64vhdl2x is
port(CLK: in STD_LOGIC;
f: out STD_LOGIC);
end;
architecture synth of cnt64vhdl2x is
signal delay, delxor: STD_LOGIC_VECTOR(63
downto 0);
begin
  delxor <= delay xor
"1000000010000000100000010000100010000001000000
0000100000010000001";
  process (CLK) begin
    if rising_edge (CLK) then
      if (or delxor) = '0' then
        f <= '1';
        delay <= (OTHERS => '0');
      else
        f <= '0';
        delay <= delay + '1';
      end if;
    end if;
  end process;
end;
```

Счётчик с ограничением модуля счёта 8080810881008082_{16} способом 3 на SystemVerilog в модуле cnt64sv3:

```
module cnt64sv3 (
input CLK,
output logic f);
logic [63:0] delay;
always_ff @(posedge CLK)
    if (delay[63] & delay[55]
      & delay[47] & delay[40] & delay[35]
      & delay[31] & delay[24] & delay[15]
      & delay[7]  & delay[1])
    begin
        f <= 1'b1;
        delay <= '0;
    end
    else begin
        f <= 1'b0;
        delay <= delay + 1'b1;
    end
endmodule
```

Счётчик с ограничением модуля счёта 808081088100808216 способом 3 на VHDL (1993) в модуле cnt64vhdl3:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity cnt64vhdl3 is
port(CLK: in STD_LOGIC;
f: out STD_LOGIC);
end;
architecture synth of cnt64vhdl3 is
signal delay: STD_LOGIC_VECTOR(63 downto 0);
begin
  process (CLK) begin
    if rising_edge (CLK) then
      if (delay(63) and delay(55)
```

```
      and delay(47) and delay(40) and delay
(35)
      and delay(31) and delay(24) and delay
(15)
      and delay(7)  and delay(1)) = '1' then
        f <= '1';
        delay <= (OTHERS => '0');
      else
        f <= '0';
        delay <= delay + '1';
      end if;
    end if;
  end process;
end;
```

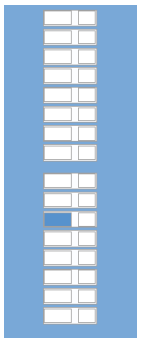


Рисунок 1. Задействованная LC для генерации GND

Результаты компиляции и их обсуждение

Приведённые выше модули были откомпилированы в САПР Quartus v.9.1 sp2, v.20.1 (далее по тексту Q9 и Q20 соответственно) и Vivado 2019.1 (далее по тексту V19). Результаты по требуемым ресурсам приведены в табл. 1.

В САПР Qurartus результаты получились предсказуемые, конъюнкция (способ 3) требует значительно меньше ресурсов, нежели способы 1 и 2. Однако, есть небольшие расхождения между Q9 и Q20 в реализации 3 способа для разрядности счётчиков 32 и 64. Эти расхождения возникли из-за разных кристаллов. В кристалле 10M08DAF484C8G Q20 дополнительно тратит LC на генерацию земли для встроенных блоков (рис. 1 и 2). Также Q9 и Q20 по-разному распределили ресурсы (рис. 3).

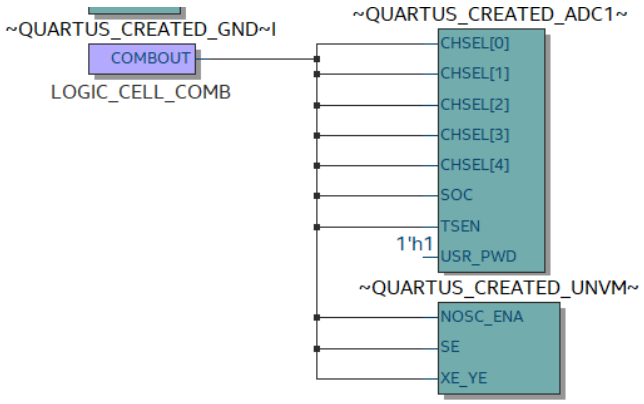


Рисунок 2. Схема генерации GND в RTL Viewer

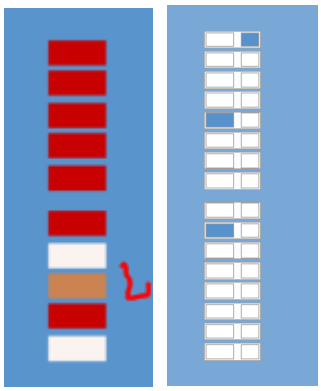


Рисунок 3. Chip planner: слева Q9, справа Q20

Таблица 1. Занимаемые ресурсы в разных кристаллах

	EPC1C3T144C7 (Q9), LC-LCR	10M08DAF484C8G (Q20), LC-LCR	XC7A35TCPG236-1 (V19), SL-SR
cnt32	66-33	49-33	9 (8)-33
cnt32-2	67-33	49-33	8-33
cnt32-3	34-33	36-33	2-33
cnt64	133-65	97-65	19 (22)-65
cnt64-2	134-65	97-65	14-65
cnt64-3	67-65	69-65	3-65

На рис. 3 chip planner в Q9 отмечает красным цветом ячейки с занятыми LC и R (триггер), оранжевым – используется только LC без R; в Q20 – слева LC, а справа R (триггер). Таким образом Q20 потратил больше ячеек, т.к. использовал две LC без R, также есть отдельная R без LC и плюс к этому LC на генерацию GND на внутренние блоки. Однако разницы между языками SystemVerilog и VHDL в Q9 и Q20 в этих модулях обнаружено не было.

В САПР V19 результат получился более «интересный», т.к. есть отличия по ресурсам между языками SystemVerilog и VHDL в первом способе для обеих разрядностей, т.е. для модулей cnt32sv и cnt32vhdl, а также cnt64sv и cnt64vhdl. Разница в интерпретации кодов со стороны V19 видна в RTL схемах (рис. 4 и 5).

Кроме этого, стоит также отметить оптимизацию второго способа со стороны V19 для счётчика разрядностью 64. Q9 и Q20 преобразует в многовходовую NOR, V19 на RTL схеме реализует «в лоб» (рис. 6).

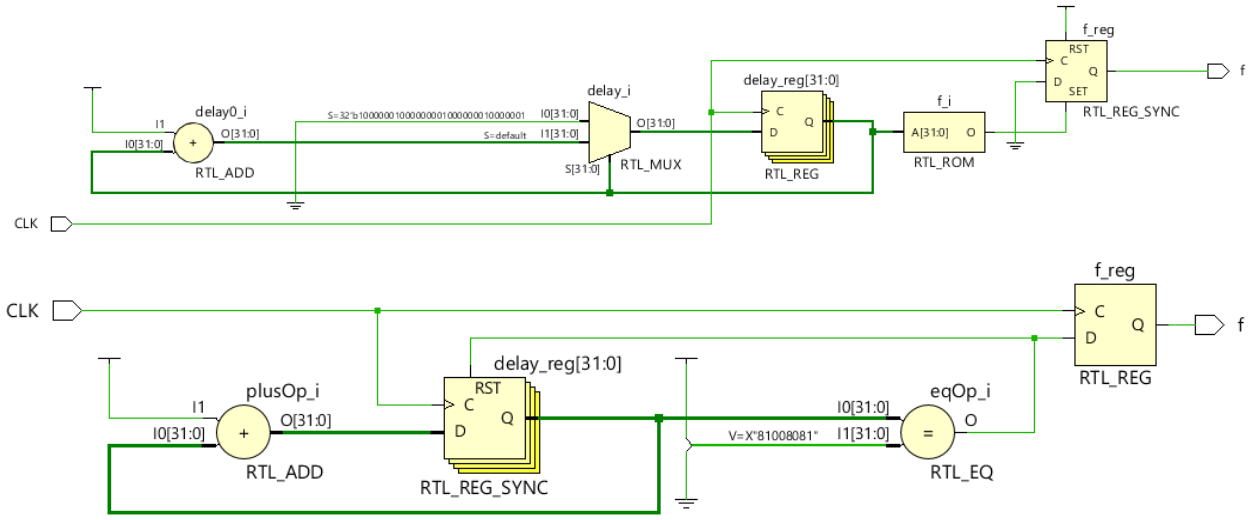


Рисунок 4. RTL схемы cnt32sv сверху, а cnt32vhdl снизу

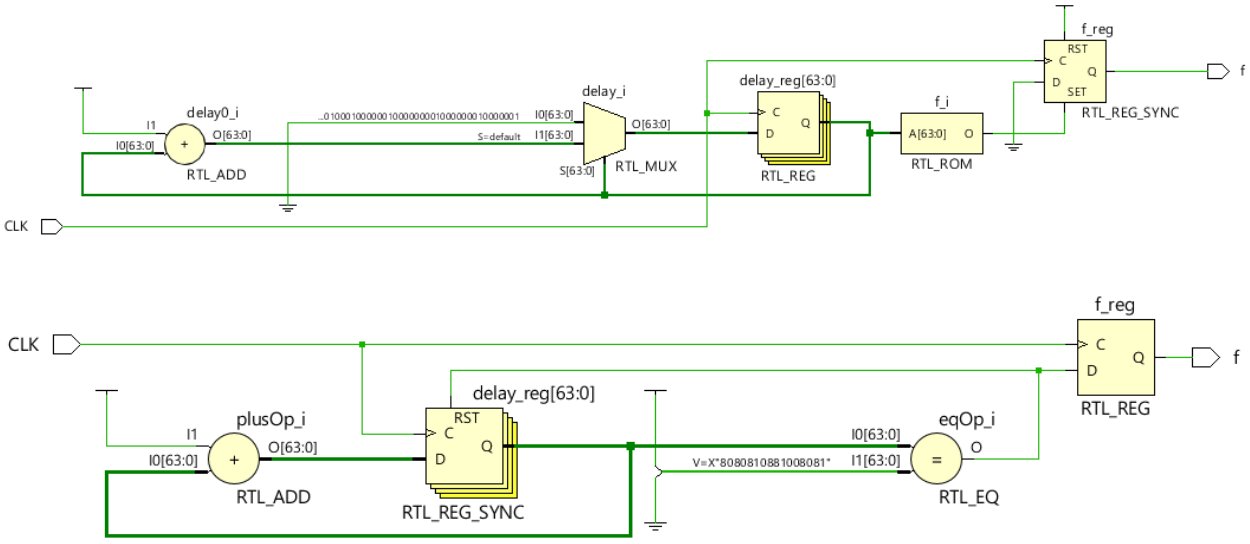


Рисунок 5. RTL схемы cnt64sv сверху, а cnt64vhdl снизу

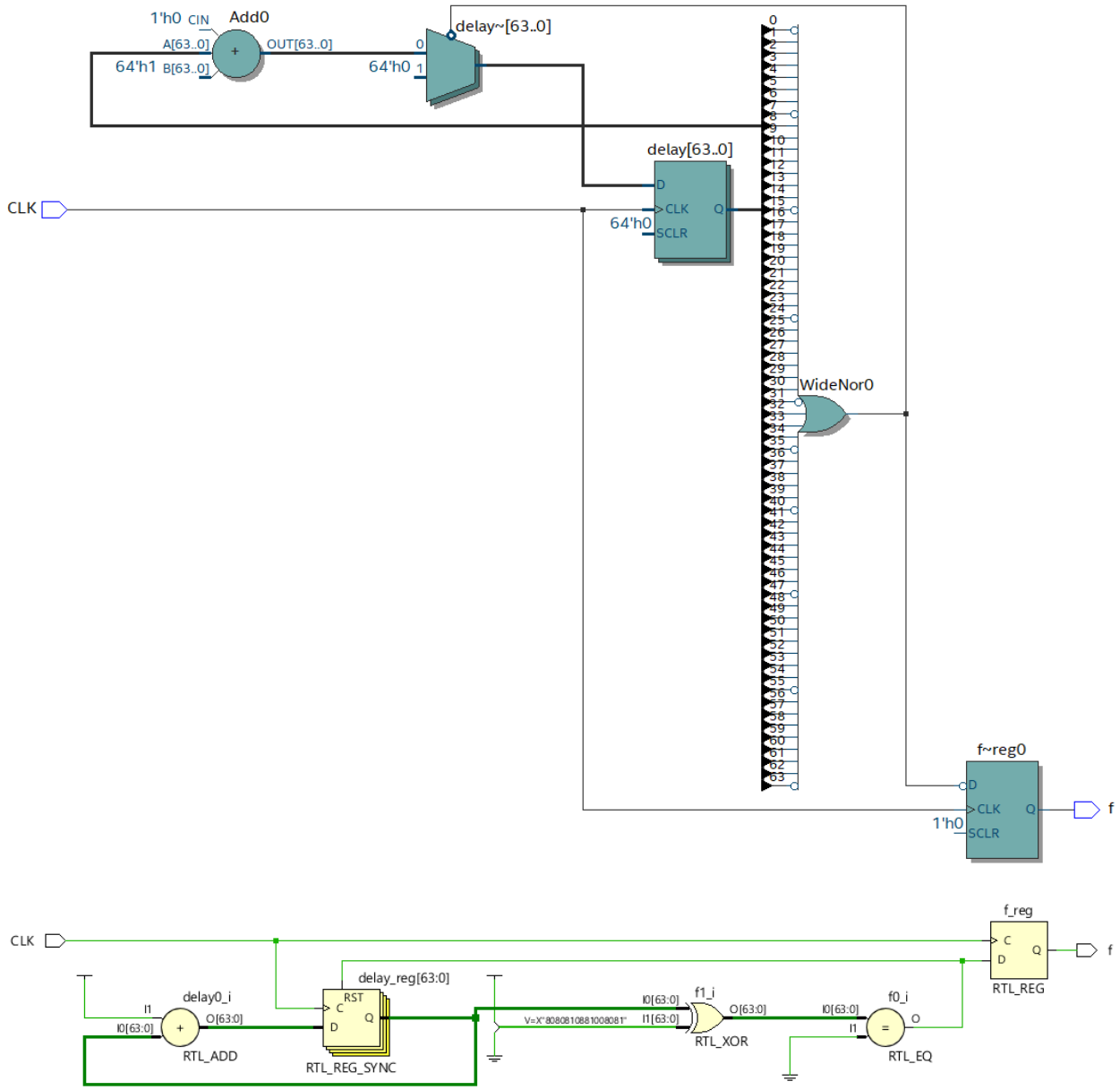


Рисунок 6. RTL схемы cnt64-2 Q20 сверху и V19 снизу

Заключение

Показано, что разные способы реализации условия оператора if требуют разное количество ресурсов, ожидаемо для этих примеров наименее затратный способ 3 – через конъюнкцию. В этот раз САПР Vivado по-разному синтезировал одинаковые устройства, описанные одинаковыми конструкциями на SystemVerilog и VHDL. Опять же, стоит отметить, что Vivado в VHDL 2008 поддерживаются многовходовые логические элементы (reduction), в отличие от Quartus.

Быстрое вычисление медианы в целых числах

Пузанов Николай

Телеграм: *@punzik*

Обсуждение и комментарии: link

Введение

В задачах статистики и цифровой обработки сигналов нередко возникает потребность в вычислении медианы некоторого массива данных. Обычно нужно вычислить медиану от нескольких значений, что реализуется относительно просто и работает быстро. Но бывают случаи, когда массив содержит тысячи или десятки тысяч элементов, а машинного времени и ресурсов не очень много. В этом случае может помочь алгоритм binmedian со средней сложностью O(n), или его модификация binapprox, которая вычисляет приближенную медиану, но гарантированно за O(n).

В этой статье рассматривается рекурсивная реализация алгоритма binmedian, вычисляющая точное значение медианы для массива целых числах конечной разрядности. Алгоритм имеет сложность строго O(n).

Исходные данные

В качестве примера возьмём массив из 15 беззнаковых чисел (рис.1).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	2	26	31	14	10	8	28	8	18	10	13	28	31	23

Рис.1: Исходный массив чисел. Верхняя строка - индексы элементов массива.

Найдём медиану обычным способом, т.е. отсортируем массив и возьмём средний элемент (рис.2).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	8	8	10	10	13	13	14	18	23	26	28	28	31	31

Рис.2: Отсортированный массив. Средний элемент выделен пунктирным прямоугольником.

Средний элемент имеет номер 7. Его значение, а соответственно и значение медианы равно 14. Это число мы будет использовать в дальнейшем для проверки корректности описанных методов.

Метод binmedian

Как известно, сортировка - это довольно затратная задача для выполнения "в железе". К счастью, существует способ избежать этой процедуры с помощью метода вычисления медианы под названием binmedian. В общем случае он позволяет вычислить точное значение медианы без использования сортировки *в среднем* за O(n). В случае целых

чисел, как будет показано ниже, с помощью этого метода вычисляется точное значение *строго* за O(n). Для вычисления медианы методом binmedian нужно построить гистограмму входного массива чисел, а затем последовательно складывать значения бинов, пока сумма не станет больше половины длины последовательности (номера центрального элемента массива).

В качестве примера построим гистограмму из четырёх бинов. Для этого разобьем диапазон чисел в массиве на четыре части. С учётом входных значений, диапазоны бинов будут следующие (каждое значение - это полуинтервал [a;b), в него входят значения большие или равные левой границе, и меньшие правой границы): `[2;10), [10;17), [17;24), [24;32).`

На рис.3 цветами выделены интервалы в отсортированном массиве.

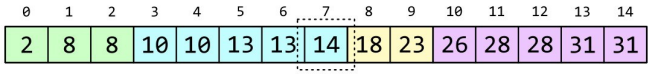


Рис.3: Отсортированный массив, разбитый на интервалы.

Вычислим значения бинов гистограммы, а затем просуммируем их слева направо до тех пор, пока сумма не превысит число `7` (индекс среднего элемента массива). Бин, на котором остановился счёт будет тем бином, в интервале которого находится медиана. В нашем случае это бин под номером `1`, т.е. медиана находится в интервале `[10;17)`. Гистограмма показана на рис.4.

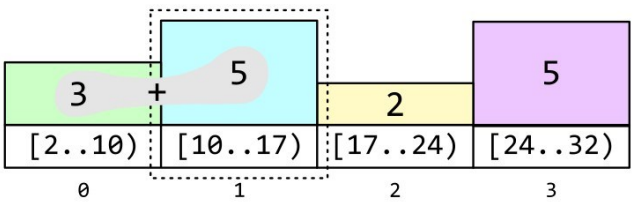


Рис.4. Гистограмма. В нижней части обозначены номера бинов, затем выше интервалы бинов, и наконец их значения. Сумма первых двух бинов равна `8`, что больше, чем индекс среднего элемента массива.

Нетрудно вычислить сложность этого метода. Количество операций равно n * k + c, где n - количество элементов в массиве, k - количество операций, необходимых для обновления гистограммы, c - расходы на подготовку гистограммы и сложение значений бинов. Т.к. k и c константы (c зависит от разрядности чисел и в общем случае не зависит от количества элементов), то время вычисления зависит (линейно) только от количества элементов массива. Таким образом сложность метода равняется O(n).

Точное вычисление медианы

Чтобы вычислить точное значение медианы, очевидно нужно сделать гистограмму такой длины, чтобы интервал каждого бина был равен единице. Тогда результирующее значение точно попадёт в значение медианы (напомню, мы производим вычисления в целых числах).

В нашем случае необходима гистограмма от минимального значения (2) до максимального

(31) с количеством бинов 30.

Чтобы упростить схему (а мы же в итоге планируем сделать это в железе?), можно взять полный интервал разрядности чисел. В этом случае адресом бина будет служить просто значение очередного числа из массива. Для нашего примера я специально выбрал числа шириной `5` бит, то есть их значения лежат в диапазоне от 0 до `31`.

Построим гистограмму. Как и в предыдущем случае, будем складывать значения бинов слева направо пока сумма не превысит `7` (рис.5). В итоге мы остановимся на бине `14` с суммой, равной `8`. Номер бина и есть значение медианы.

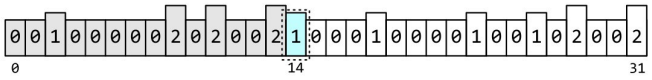


Рис.5. Гистограмма с количеством бинов, равном полному диапазону входных чисел. Индекс выделенного бина и есть искомая медиана.

Сложность алгоритма не изменилась, но увеличился размер гистограммы. А что если у нас будут числа большей разрядности? Строить гистограмму например для 32-битных данных очень расточительное занятие - понадобится массив памяти на 4 миллиарда слов.

Точное вычисление медианы для чисел с большой разрядностью

В этом случае можно разбить вычисление на несколько этапов. Сначала, используя старшие разряды чисел вычислить приближенное значение медианы. Затем, опускаясь по разрядам, прийти к точному значению. Количество этапов может быть произвольным в зависимости от разрядности чисел и требований к объёму используемой памяти.

У нас разрядность небольшая, по этому вычислим медиану в два этапа. Для этого разобьём наш интервал разрядности (5 бит) на две части. Первая часть будет состоять из 3 бит, вторая - из 2 бит.

Построим гистограмму длиной 8 используя только 3 старших бит для индексации бинов.

Так же, как и в предыдущих случаях, просуммируем бины слева направо и определим номер бина, на котором сумма превысит индекс центрального элемента. В нашем случае это бин с номером 3 (рис.6). Значение старших 3 бит медианы будет равно этому номеру, что в бинарном виде будет выглядеть как 0b011.

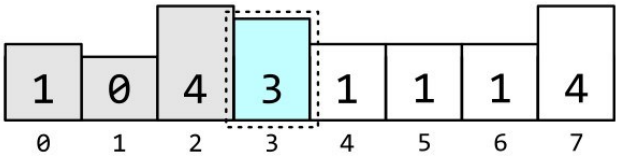


Рис.6. Гистограмма, посроенная по старшим 3 битам входных чисел. Соответствующие старшие биты медианы равны индексу бина, на котором остановился подсчёт суммы.

Теперь можно вычислить точное значение медианы, используя наше знание о значении её старших бит. Для этого построим вторую гистограмму. Для её построения будем отбирать числа, старшие биты которых равны значению, полученному на предыдущем шаге - 0b011.

Для индексации гистограммы будем использовать только младшие 2 бит этих чисел, т.е. длина гистограммы будет равняться 4.

На этом шаге мы должны ввести новую операцию, которой не было в ранее. Необходимо запомнить количество чисел, значение старших бит которых меньше значения, полученного на предыдущем шаге. Фактически это ещё один бин гистограммы с интервалом "меньше меньшего". Значение этого бина будет необходимо на последнем шаге вычислений. Остался последний шаг - просуммировать бины и остановиться на том, где сумма превысит индекс центрального элемента. Однако, в отличие от предыдущих примеров, начальное значение суммы равно не нулю, а количеству чисел, старшие биты которых меньше старших битов медианы (тот самый бин "меньше меньшего"). Нетрудно посчитать их количество - 5 штук. В результате последовательного суммирования (показано на рис.7) мы получили номер бина - 2. Это и есть значение младших бит искомой медианы - 0b10. Склеив всё вместе получим ответ - 0b01110, или 14.

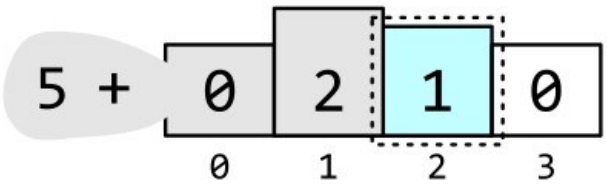


Рис.7. Гистограмма, посроенная по младшим 2 битам входных чисел. На рисунке выделен бин, индекс которого равен младшим битам значения медианы.

Несмотря на то, что число итераций удвоилось, время потраченное на вычисления всё ещё линейно зависит от количества элементов массива. Т.е. сложность алгоритма осталась той же.

Заключение

В статье был показан метод быстрого вычисления медианы для массива челых чисел с использованием алгоритма binmedian. Автором был реализован описанный алгоритм в виде RTL, который был успешно использован в ПЛИС Xilinx серий 7 и US+.

Исходный код статьи вы сможете найти по ссылке: https://github.com/punzik/fast-median-article. Там вы так же найдёте код на Python, которым вычислялись значения, приведенные в статье.

Испытательный стенд с использованием YosysHQ MCY

Сергей Б.

e-mail: sergebn@mail.ru

Телеграм: @SergeBN

Обсуждение и комментарии: link

Аннотация

Эта работа представляет небольшой обзор методологии мутационного тестирования, так как она описана на странице YosysHQ MCY. Также, во второй части этой работы я опишу использование этой методологии на примере разработки и мутационного тестирования тестового стенда для модуля расчёта суммы чисел натурального ряда.

1 Мутационное тестирование

1.1 Введение в методологию

Согласно YosysHQ MCY - это новый инструмент, помогающий цифровым дизайнерам и руководителям проектов понять и улучшить охват тестовых стендов.

Утверждение 1. *Если у вас есть тестовый стенд, и он даёт сбой, вы знаете, что у вас проблема. Но если он проходит успешно, вы ничего не знаете, если не знаете, что на самом деле тестирует ваш тестовый стенд.*

1.2 Основные принципы

Идея MCY заключается в том, чтобы проверить насколько хорош набор тестов при обнаружении ошибок в проекте путём преднамеренного внесения изменений (небольших "мутаций", которые изменяют значение одного бита в проекте) и проверки, были ли они обнаружены. Однако не гарантируется, что мутация нарушит дизайн: если, например, значение бита изменяется только в циклах, в которых связанный valid сигнал низкий, дизайн по-прежнему будет функционировать так, как задумано, и набор тестов будет корректным при его прохождении.

Прелогика 1.1. Ваш набор тестов повторяет оригинальный дизайн.

Следствие 1.1. Невозможно измерить охват мутациями для неудачного проекта. Если исходный проект уже нарушен, дальнейшее его нарушение путём введения мутации не может привести к заметному изменению состояния тестирования.

Обычно инструменты покрытия предлагают выбрать произвольный целевой

коэффициент покрытия меньше 100% основываясь на интуиции и опыте, чтобы учесть те случаи, когда мутация не имеет существенного значения. MCY стремится заставить вас настроить формальную проверку эквивалентности, которая послужит “истиной” о том, влияет ли мутация на функционирование вашего проекта. Если проверка эквивалентности может найти допустимую комбинацию входных данных, для которых выходные данные измененного дизайна отличаются от исходных, но набор тестов все равно проходит дизайн, то набор тестов не настроен для обнаружения такого типа некорректного поведения. Таким образом целью становится достижение 100%-ного охвата.

При настройке формальной эквивалентности необходимо будет указать какие изменения во входных и выходных сигналах являются функциональными различиями в соответствии с проектной документацией. В отличие от полной функциональной проверки, гораздо проще указать, что если, например, изменённый дизайн выводит “5”, когда оригинал выводит “3”, то, по крайней мере, одно из значений должно быть неправильным, чем писать формальные свойства, которые вычисляют, что результат *должен* быть “42” на основе входных данных, которые могли быть введены в течение нескольких предыдущих циклов.

Для ядра MCY нет ни какой разницы в том, как запускать тот или иной тест в общем наборе тестов. Поэтому в формат файла конфигурации не встроено никакой специальной обработки, которая помогла бы вам правильно настроить ваш проект. Если один автор пишет и набор тестов, и тест эквивалентности, различие между ними может стать размытым. При настройке теста эквивалентности, и особенно при написании [logic], что касается конфигурации, то крайне важно, чтобы вы держали эту основную идею на переднем крае своего сознания:

Основной принцип 1.1. Проверка эквивалентности - это эталонный стандарт, с которым сравнивается ваш набор тестов. Он сообщит вам, можно ли обнаружить релевантное изменение поведения в изменённом дизайне. И о вашем наборе тестов будут судить по тому, точно ли он обнаруживает это изменение или нет.

Примечание 1.1. Это не говорит о том, что проект, который проходит проверку на эквивалентность, является правильным, или проект, который не соответствует эквивалентности, является ошибочным. Поскольку, скорее всего, оригинальный дизайн не совсем свободен от ошибок, то эквивалентность ничего не может сказать вам о его корректности.

Проверка эквивалентности служит для определения того, **способен ли ваш тестовый стенд обнаруживать изменения в поведении**, которые внесла мутация.

При запуске проекта MCY рекомендуется начать с непосредственного использования этой “наивной” логики и сохранения небольшого количества мутаций. По мере того, как вы обретаете уверенность в том, что настройка работает, вы можете постепенно добавлять оптимизации, которые позволят вам запускать большее количество мутаций.

Пример показывает реализацию наивной логики:

```
sim_okay = result(”test_sim”) == ”PASS”
eq_okay = result(”test_eq”) == ”PASS”
if sim_okay and not eq_okay:
       tag(”UNCOVERED”)
elif not sim_okay and not eq_okay:
       tag(”COVERED”)
elif sim_okay and eq_okay:
       tag(”NOCHANGE”)
elif not sim_okay and eq_okay:
       tag(”EQGAP”)
else: assert 0
```

Если начальный набор тестов состоит из тестового стенда test_sim и стенда проверки эквивалентности test_eq, то приведённая логика безоговорочно запускает оба стенда. И далее первые три случая являются ожидаемыми: если проверка эквивалентности показывает, что существует разница в поведении между исходной и мутировавшей моделями, то, если тестовый стенд улавливает это, мутация покрывается, а если ему не удается ее уловить, она не покрывается; если проверка эквивалентности и тестовый стенд соглашаются с тем, что исходная и измененная конструкции ведут себя одинаково, мутация бесполезна для оценки качества тестового стенда и может быть проигнорирована.

Следите за этой четвертой категорией! В правильно настроенном проекте этого никогда не должно произойти. Если проверка эквивалентности обнаруживает, что мутация не влияет на поведение, но тестовый модуль обнаруживает ошибку, значит, один из них неверен. Либо проверка эквивалентности не обнаруживает соответствующих изменений в поведении, либо тестовый стенд не соответствует требованиям дизайна. Тщательно исследуйте эти случаи. Если поведение с рассматриваемой мутацией в порядке, настройте тестовую среду для ее прохождения. Если поведение не в порядке, измените проверку эквивалентности, чтобы обнаружить проблемное изменение. Приступайте к оптимизации логики только после того, как вы не заметите никаких случаев четвертого типа.

По соображениям производительности, вы часто захотите получить логически эквивалентный, но сильно отличающийся поток в [logic] разделе, и это может стать источником путаницы при рассмотрении более сложных оптимизированных примеров. Есть два допущения, которые можно использовать, чтобы избежать запуска некоторых тестов и все равно получить тот же результат:

Допущение 1.1. Проверка эквивалентности верна.

Следствие1.1.1. Если проверка эквивалентности завершается неудачей, мутация изменяет функционирование дизайна. Если она проходит успешно, функциональной разницы между исходным и измененным дизайном нет.

В частности, это означает, что если проверка эквивалентности пройдет успешно, нет необходимости запускать какой-либо набор тестов для этой мутации. Поскольку мутация не оказывает какого-либо существенного эффекта, она ничего не может сказать нам о способности набора тестов обнаруживать ошибки.

Допущение 1.2. Набор тестов не будет возвращать ложные сбои для рабочего проекта.

Убедитесь, что ваши тестовые стенды терпимы к небольшим изменениям в поведении, которые не имеют отношения к корректности, таким как, например, значения выходных битов, которые не используются в определенных режимах.

Исходя из этого предположения, если какой-либо тест в наборе возвращает ОШИБКУ (FAIL), нет необходимости запускать ни другие тесты из набора тестов, ни проверку эквивалентности — мутация устранена. Это позволяет вам сначала запустить короткие модульные тесты и быстро определить результат для более ”явно глючных” мутаций.

Любой случай EQGAP нарушает по крайней мере одно из этих предположений. Вот почему рекомендуется провести тестирование для данного случая в самом

начале, чтобы получить уверенность в правильности допущений. Также можно использовать rng(), чтобы проверить, что предположение справедливо только для случайного подмножества мутаций, чтобы сбалансировать появление проблем со временем выполнения теста и быть уверенным в корректности теста.

Как правило, для достижения наилучшей производительности всегда сначала запускайте самый короткий тест. Если у вас есть модульные тесты, запускайте их первыми, а интеграционные тесты - после, в порядке возрастания времени выполнения. Проверка эквивалентности часто является одним из более длительных тестов, особенно для наихудшего случая, когда проекты эквивалентны.

Пример другой логики. Если ваш набор тестов содержит несколько тестовых стендов с увеличивающейся продолжительностью выполнения: модульный тест test_unit, интеграционный тест test_sys и аппаратный тест в цикле test_hw, и проверка эквивалентности test_eq обычно это занимает больше времени, чем test_sys, но меньше времени, чем test_hw, тогда применение двух допущений для скорейшего завершения, когда это возможно, привело бы к следующей логике:

```
if result(”test_unit”) == ”FAIL”:
    tag(”COVERED”)
    return

if result(”test_sys”) == ”FAIL”:
    tag(”COVERED”)
    return

if result(”test_eq”) == ”PASS”:
    tag(”NOCHANGE”)
    return

if result(”test_hw”) == ”FAIL”:
    tag(”COVERED”)
    return

tag(”UNCOVERED”)
```

Как вы можете видеть, лежащие в основе рассуждения больше не очевидны из этого кода. Всегда помните о фундаментальном принципе и двух допущениях при работе с MCY!

2 Создание и анализ мутаций с помощью MCY

Итак, путь очерчен, дорога намечена, давайте пройдём по этому пути. Изначально я следовал наставлениям из этого руководства. Потом для себя я создал отправную точку в GITVERSE. Далее все события будут разворачиваться вокруг этого кода. Клонируйте код `git clone https://gitverse.ru/sc/serge.balakshiy/sumn_mcy.git` и модифицируйте его по своему усмотрению. Как использовать полученный код, можно прочитать в README. Здесь мы сосредоточимся на ответах к вопросам “как”, “откуда”, “почему”.

Вы также можете получить оригинальный код из mcy/examples/bitcnt.

Помня прелогику 1.1 и следствие 1.1, я выбрал модуль sum_n, представленный в статье «К формальной проверке». Этот модуль уже прошел формальную проверку.

В каталоге проекта создадим проверочный стенд sumn_sim_tb.v с самоконтролем, для которого будем измерять покрытие. Это не единственное решение для выполнения такой задачи, и стенд может быть любым видом теста, который может быть запущен из скрипта без ручного вмешательства, и может возвращать результат PASS/FAIL.

```
module sumn_sim_tb;
  reg [ 7:0] mdin;
  reg [ 7:0] ndin;
  wire [15:0] sdout;
  sumn uut (
         .m_in (mdin ),
         .n_in (ndin ),
```

```
        .s_out (sdout)
      );
  task test;
    input [ 7:0] mi;
    input [ 7:0] ni;
    input [15:0] si;
    begin
      mdin = mi;
      ndin = ni;
      #20;
      if (si === sdout)
      begin
        $display("OK sum=%d mi=%d do(ref)=do
(uut)=%d si=%d", mi, ni, sdout,si);
      end
      else
      begin
        $display("ERROR sum=%d mi=%d do(ref)
=do(uut)=%d si=%d", mi, ni, sdout,si);
        $stop;
      end
      #20;
    end
  endtask
  reg [ 7:0] N,M;
  reg [15:0] result;
  integer   seed =77;
  initial
  begin
    // Проверяем граничные наборы данных test
(0,0,0); test(0,1,0); test(1,0,0); test
(1,1,1); test(0,2,0); test(1,2,3); test
(1,3,6); test(13,80,3162); test(80,13,0);
test(1,255,32640); test(7,7,7); test
(41,41,41); test(255,255,255); test
(256,256,0);
    // Для большего покрытия присвоим
случайные значения repeat(127) begin
    N = $random(seed);
    M = $random(seed);
    result = (N>0)&(M>0)&(N>=M)?((N+M)*(N-
M+1))/2:0;
    test(M,N, result);
  end
  $display("PASS tests");
  $finish;
end
initial
begin
  $dumpfile("out.vcd");
  $dumpvars(0, sumn_sim_tb);
end
endmodule
```

Для этого стенда создана задача test, которая принимает три параметра. Два числа m_i и n_i - верхняя и нижняя граница для вычисляемой суммы и параметр si - ожидаемый, заранее вычисленный результат. Также в стенде создаётся экземпляр модуля sumn, который вычисляет результат. Вычисленный результат сравнивается с ожидаемым значением и на дисплей выводится результат работы стендовой задачи. Этот результат будет необходим для дальнейшей обработки.

Модуль sum_n, который я переименовал в sumn, представляет собой синтезируемый дизайн, который будет изменён MCY. Этот дизайн может состоять из нескольких модулей/файлов. Если ваш дизайн большой или требует много этапов формальной проверки, вам следует разделить его на несколько частей, которые будут изменены отдельно. "Модуль верхнего уровня мутации" не обязательно должен совпадать с тестируемым модулем в testbench, это может быть любой модуль в его иерархии подмодулей. Никаких других изменений в этот модуль я не вносил. И это очень простой модуль, вычисляющий сумму натурального ряда чисел от n до m. Модуль возвращает вычисленную сумму либо 0, при несоответствующем наборе входных данных в соответствии с формулой

$$\sum_{j=1,m=1}^{n>=m} S = \frac{(m+n)(n-m+1)}{2} \qquad (1)$$

```
module sumn
  (
    // input wire
    input wire [7:0]    m_in, n_in,
    // output wire output wire [15:0]
s_out );
    assign s_out = (n_in>=m_in)&(n_in>0)&
(m_in>0)? (m_in + n_in) * (n_in - m_in + 1) /
2 : 0;
endmodule
```

Запустим стенд.

```
$ iverilog -D RAND sumn.v sumn_sim_tb.v
$ vvp a.out
```

В результате должны получить следующее:

```
VCD info: dumpfile out.vcd opened for output.
OK mi= 0 ni= 0 do(ref)=do(uut)= 0 si=   0
OK mi= 0 ni= 1 do(ref)=do(uut)= 0 si=   0
...
OK mi=115 ni=246 do(ref)=do(uut)=23826
si=23826 OK mi=181 ni=133 do(ref)=do(uut)
=   0 si=   0 OK mi=245 ni= 25 do(ref)=do
(uut)= 0 si=   0
PASS tests
sumn_sim_tb.v:62: $finish called at 5600 (1s)
```

2.1 Конфигурационный файл

Внутри проекта создадим новый файл под названием config.mcy. Это основной файл конфигурации MCY. В файле конфигурации MCY есть несколько обязательных разделов, которые будут постепенно заполняться на следующих этапах. Теперь вставим первые пять заголовков для этих разделов:

```
[options]
[script]
[files]
[logic]
[report]
```

Добавим первую настройку в раздел [options]

```
[options]
size 10
```

Это количество мутаций, которые сгенерирует MCY. Это очень низкое число выбрано при настройке проекта, чтобы тестовые запуски были быстрыми. Как только все заработает правильно, следует увеличить размер набора мутаций.

2.2 Генерация изменяемого сетевого списка (Mutable netlist generation)

Yosys вносит мутации в синтезированный список соединений. На первом этапе нам нужно рассказать Yosys, как создать этот список.

В разделе [script] заполним инструкции по чтению тестируемого дизайна в Yosys:

```
[script]
read -sv sumn.v
prep -top sumn
```

Это должна быть серия read команд, по одной на файл. Если проект большой, следует включить только подмножество файлов, необходимых для сборки модуля, который должен быть изменен. Когда все источники будут прочитаны, строка prep -top sumn запускает синтез в модуле, подлежащем мутации, обозначенном -top.

Теперь можно протестировать созданный скрипт либо в интерактивном режиме в Yosys, либо сохранив его в файле script.ys (без заголовка [script]) и выполнив команду:

```
$ yosys script.ys
```

В разделе [files], содержащем список путей к файлам, следует записать файлы, требуемые скриптом:

```
[files] sumn.v
```

2.3 Скрипт тестового стенда

Файл с именем test_sumn_sim.sh. В начале он выглядит так:

```
#!/bin/bash
exec 2>&1
set -ex
MCY_SCRIPTS=/путь_к_yosys/mcy/scripts
bash $MCY_SCRIPTS/create_mutated.sh
```

Когда запускается MCY, тесты будут выполняться во временном каталоге tasks/<uuid>, и пути должны располагаться относительно этого местоположения.

Сценарий create_mutated.sh считывает файлы описания мутации, подготовленные во временном каталоге задач с помощью MCY, и

(если вызывается без дополнительных аргументов) записывает файл mutated.v, содержащий мутировавший модуль.

Далее следует пара строк, которая запускает тестовый стенд как обычно, но с заменой исходного кода (модуль sumn.v в нашем случае) на мутировавший источник (mutated.v)

```
iverilog -o sim ../../sumn_sim_tb.v mutated.v
vvp -n sim > sim.out
```

При прогоне этого стенда с мутировавшим модулем получим тот же результат, что и в листинге, приведённом выше.

Теперь напишем логику сохранения возвращённого состояния стендом в файл output.txt:

```
if grep PASS sim.out && ! grep ERROR sim.out;
then
    echo ”1 PASS” > output.txt
elif ! grep PASS sim.out && grep ERROR
sim.out; then
    echo ”1 FAIL” > output.txt
else
    echo ”1 ERROR” > output.txt
fi
exit 0
```

Единица 1 перед статусом указывает индекс теста. Для тестов со значительными затратами на настройку возможно протестировать несколько мутаций за одно выполнение, и в этом случае этот номер идентифицирует тестовый запуск. Здесь мы запускаем каждый тест индивидуально, поэтому индекс всегда равен 1.

Проверить правильность работы этой части можно следующим образом:

- создадим директории database и tasks/test внутри директории проекта. Примечание: эти каталоги будут удалены при запуске MCY, поэтому не сохраняем в них никаких важных файлов.
- добавим write_ilang database/design.il в конец файла script.ys, созданного ранее.
- выполним следующие команды:

```
yosys script.ys
cd tasks/test
echo ”1 mutate -mode none” > input.txt
SCRIPTS=/путь_к_yosys/mcy/scripts bash ../../
test_sumn_sim.sh
```

(При необходимости следует изменить путь для scripts, чтобы он соответствовал местоположению установки MCY.)

- убедимся, что файл output.txt был создан и содержит 1 PASS.
- если все работает, добавим следующий раздел в нижней части config.mcy:

```
[test test_sumn_sim]
expect PASS FAIL
run bash $PRJDIR/test_sumn_sim.sh
```

2.4 Формальный тест эквивалентности

Это самая трудоемкая часть проекта MCY, но также и то, что делает MCY особенным. Чтобы узнать, должен ли тестируемый тестовый стенд возвращать PASS или FAIL, мы настроим формальную проверку свойств, которая сможет окончательно определить, может ли мутация соответствующим образом повлиять на выходные данные модуля.

Преимущество использования формальных методов заключается в том, что они будут исчерпывающе исследовать все возможные комбинации входных данных, что является непомерно высоким для испытательного стенда моделирования для большинства нетривиальных проектов из-за комбинаторного взрыва. Но подход MCY также менее сложен, чем прямая формальная проверка дизайна, поскольку, как правило, проще описать, является ли изменение выходных данных “важным”, чем напрямую

описывать правильное поведение.

В отличие от предыдущего теста, где мы экспортировали измененный модуль с тем же интерфейсом, что и исходный модуль, чтобы мы могли легко заменить его в тестовом модуле, здесь мы будем использовать *-c* возможность получить модуль, в котором мы можем включать или отключать мутацию по желанию на основе входного сигнала mutsel. Мы также будем экспортировать в формат ILANG вместо Verilog, поскольку SBY это понимает.

Создадим файл test_sumn_eq.sh и добавим следующий скрипт:

```
#!/bin/bash
exec 2>&1
set -ex
bash $SCRIPTS/create_mutated.sh -c -o mutat-
ed.il
```

Далее мы создадим схему miter, которая сравнивает исходный и изменённый модуль. Создадим файл с именем test_sumn_eq.sv и введём следующий код:

```
module miter (
  input [ 7:0] ref_mdin,
  input [ 7:0] uut_mdin,
  input [ 7:0] ref_ndin,
  input [ 7:0] uut_ndin,
  input [15:0] result_din
               );
  wire [15:0] ref_sdout;
  wire [15:0] uut_sdout;
  sumn ref (
         .mutsel    (1'b0),
         .m_in(ref_mdin),
         .n_in(ref_ndin),
         .s_out(ref_sdout)
       );
  sumn uut (
         .mutsel    (1'b1),
         .m_in(uut_mdin),
         .n_in(uut_ndin),
         .s_out(uut_sdout)
       );
  always @*
  begin
    assume ((ref_mdin == uut_mdin)
            && (ref_ndin == uut_ndin));
    assert (ref_sdout == uut_sdout);
  end
endmodule
```

Здесь мы создаём экземпляр sumn дважды, один раз с отключенной мутацией (ref) и один раз с включенной мутацией (uut). Обратите внимание на сигнал mutsel. Далее мы добавим assert и assume заявления, которые выражают, при каких условиях мы ожидаем выходные данные, которые не будут изменены. То есть, попросту утверждаем для нашего модуля, что при одинаковых входных данных результаты вычислений должны совпадать.

Цель состоит в том, чтобы быть как можно более точным в отношении условий, при которых мы ожидаем тот же результат. Поэтому мы никогда ничего не будем проверять в случае "неправильных" входных данных. В последнем случае схема должна возвращать 0. И это будет также "правильным" поведением схемы. Тем самым мы избегаем комбинаторного взрыва входных данных при тестировании обычным способом.

Мы будем использовать SBY для проверки этих формальных свойств. Создадим файл test_sumn_eq.sby

```
[options]
mode bmc
depth 1
expect pass,fail

[engines]
smtbmc yices

[script]
read_verilog -sv test_sumn_eq.sv
read_ilang mutated.il
prep -top miter
fmcombine miter ref uut
flatten
opt -fast

[files]
test_sumn_eq.sv
mutated.il
```

В [4] уже описывалось, как настроить sby проект. Кроме того, вы можете ознакомиться с документацией SBY для получения подробной информации. Что касается

приведённых настроек, отметим следующее:

- Модуль sumn является комбинаторным, поэтому мы можем использовать проверку ограниченной модели за один шаг.
- Дополнительные шаги fmcombine, flatten и opt в разделе скрипта не обязательны, но они увеличивают скорость проверки.
- Предполагается, что все используемые файлы присутствуют в каталоге, в котором выполняется тест.

Проверить настройки sby в директории tasks/test с уже созданным input.txt возможно следующим образом:

```
cd tasks/test
ln -s ../../test_sumn_eq.sv ../../
test_sumn_eq.sby .
bash ../../test_sumn_eq.sh
sby -f test_sumn_eq.sby
```

Поскольку мы в очередной раз тестируем мутацию “ничего не делать”, это должно вернуть PASS. Если это работает правильно, мы можем завершить сценарий для запуска этого теста sby. Нам осталось извлечь возвращаемое значение и поместить его в файл output.txt. Добавим в файл test_sumn_es.sh следующее:

```
ln -fs ../../test_sumn_eq.sv ../../
test_sumn_eq.sby .
sby -f test_sumn_eq.sby
gawk ”{ print 1, \$1; }” test_sumn_eq/status
>> output.txt
exit 0
```

Еще раз проверим, что запущенный bash ../../test_sumn_eq.sh внутри tasks/test работает корректно и пишет 1 PASS в output.txt. Обратитевнимание, что скрипт добавляет данные в этот файл, и идентичная строка может уже существовать из предыдущих запусков, поэтому убедитесь, что при выполнении добавляется новая строка.

Теперь можно настроить конфигурацию для этого теста в конце config.mcy:

```
[test test_sumn_eq]
expect PASS FAIL
run bash $PRJDIR/test_sumn_eq.sh
```

2.5 Логика тегирования

Когда мы настроили два теста, нам нужно сообщить MCY, как мы хотим анализировать результаты. С двумя тестами есть только четыре возможных результата, каждому из которых мы можем присвоить метку:

- оба теста завершаются неудачей: тестовый стенд точно обнаруживает проблему, т. е. мутация устранена (COVERED).
- Тестовый модуль моделирования проходит успешно, но тест на эквивалентность завершается неудачей: тестовый модуль не находит проблему, т. е. обнаружена мутация (UNCOVERED).
- тестовый стенд моделирования успешно пройден и тест на эквивалентность пройден: мутация не вносит релевантных изменений в функциональность модуля (NOCHANGE).
- тестовый стенд моделирования завершается неудачей, но тест эквивалентности проходит: тест эквивалентности, должно быть, был настроен неправильно, и существует разрыв между формальным описанием и ожидаемым поведением (EQGAP).

Объявим эти четыре тега в разделе [options]

```
[options]
size 10
tags COVERED
UNCOVERED NOCHANGE EQGAP
```

Затем, под разделом [logic], опишем, как пометить тесты тегами:

```
sim_okay = result(”test_sumn_sim”) == ”PASS”
eq_okay = result(”test_sumn_eq”) == ”PASS”
if sim_okay and not eq_okay:
    tag(”UNCOVERED”)
elif not sim_okay and not eq_okay:
```

```
        tag("COVERED")
elif sim_okay and eq_okay:
    tag("NOCHANGE")
elif not sim_okay and eq_okay:
    tag("EQGAP")
else:
    assert 0
```

Этот раздел по существу определяет функцию Python и может использовать предопределенные функции result(«name>") (где <name> — тест, определенный в разделе [test <name>]) и tag(«name>") (для любого тега, определенного в тегах в разделе [options]). Одна мутация может быть помечена несколькими тегами или вообще не иметь тегов.

Если в проекте есть несколько тестов разной длины, можно использовать отложенное вычисление для условного запуска тестов. Для данной мутации тест выполняется только тогда, когда секция [logic] вызывает result(). (Пример приведен в бонусном разделе в конце этого руководства.)

Наконец, заполните раздел [report] следующим образом:

```
[report]
if tags("EQGAP"):
    print("Found %d mutations exposing a for-
mal gap!" % tags("EQGAP"))
if tags("COVERED")+tags("UNCOVERED"):
    print("Coverage: %.2f%%" % (100.0*tags
("COVERED")
      /(tags("COVERED")+tags("UNCOVERED"))))
```

Это снова раздел, определяющий функцию Python. Здесь функция tags («name>") может использоваться для получения количества мутаций, помеченных данным тегом. Если есть формальный разрыв, это весьма проблематично, поэтому об этом будет сообщено в первую очередь. Во-вторых, мы печатаем метрику покрытия, рассчитанную как процент охваченных мутаций от всех мутаций, которые вызывают соответствующие изменения в дизайне, т. е., как тех, которые помечены как «охваченные», так и как «непокрытые». Расчет происходит по формуле, заключённой в операторе print:

2.6 Запуск MCY

Теперь проект MCY полностью настроен. Удалим директории database и tasks, которые мы создали для тестирования, выполнив:

```
$ mcy purge
```

Затем выпоним:

```
$ mcy init
$ mcy run
```

Также можно выполнить:

```
$ make purge
$ make init
$ make mcy_run
```

В результате получим что-то такое

```
==> Run all tasks from queue
    -> Reading configuration file.
    -> Connecting to database.
    ==> Running task 45c9519e-ed7a-47df-8e1c-
a7d667436bd1 (test_sumn_sim)
1 1 mutate -mode none
        ==> Running task bd7358cf-3fe7-48c1-
b8b7-8442aee9cf3e (test_sumn_sim)
1 2 mutate -mode cnot0 -module sumn -cell
$ternary$sumn.v:9$11 -port Y -portbit 5 -
ctrlbit 0 -wire s_out -wirebit 5 -src
sumn.v:6.23-6.28 -src sumn.v:9.19-15.6
...
==> Finishing task 53933146-c455-46a6-92e9-
05572c3d5735 (test_sumn_fm)
    1 15 PASS mutate -mode inv -module sumn -
cell $gt$sumn.v:10$2 -port A -portbit 0 -src
sumn.v:10.6-10.12
    -> Finished running all tasks.
    -> Getting database statistics.
Database contains 47 cached results.
Database contains 18 cached "FAIL" results
for "test_sumn_eq".
Database contains 3 cached "PASS" results for
"test_sumn_eq".
Database contains 5 cached "PASS" results for
"test_sumn_fm".
Database contains 16 cached "FAIL" results
for "test_sumn_sim".
Database contains 5 cached "PASS" results for
"test_sumn_sim".
Tagged 16 mutations as "COVERED".
Tagged 3 mutations as "NOCHANGE".
Tagged 2 mutations as "UNCOVERED".
    -> Print report
Coverage: 88.89%
```

Поскольку изначально запрошено всего несколько тестов, они должны завершиться быстро. Выполнение в последовательном режиме (без -j аргумент) делает более очевидным, какой тест является причиной в случае ошибки.

Если этот начальный тестовый запуск завершится успешно и будет выведен показатель покрытия, вы можете увеличить количество мутаций в начале тестирования. config.mcy:

```
[options]
size 1000
```

На этот раз выполнение тестов займет больше времени, поэтому включите параллельные запуски (замените $(nproc) с учетом количества используемых ядер):

```
$ mcy reset
$ mcy run -j$(nproc)
```

reset сохранит существующие результаты для ранее протестированных мутаций, но добавит больше мутаций для достижения нового запрошенного размера.

Пока выполняются тесты, во втором терминале можно запустить (в директории базового проекта, где находится ваш config.mcy)

```
$ mcy dash
mcy dashAD Suite (base) serge@bw0xx:~/school/
prj/tests/sumnmc
 *   Serving Flask app 'mcy-dash' (lazy load-
ing)
 *   Environment: production
WARNING: This is a development server. Do not
use it in a production deployment. Use a pro-
duction WSGI server instead.
 *   Debug mode: on
WARNING: This is a development server. Do not
use it in a production deployment. Use a p
 *   Running on http://127.0.0.1:5000
```

Следом откройте указанный адрес в своем браузере, чтобы следить за ходом выполнения на панели мониторинга. Это может быть особенно интересно при выполнении тестов на удаленном сервере.

Как только тесты будут завершены, вы можете использовать:

```
$ mcy gui
```

чтобы визуально изучить «горячие точки» вашего кода, где существуют пробелы в покрытии, см. рисунок 1. В настоящее время это жестко запрограммировано для использования имен тегов «COVERED» и «UNCOVERED».

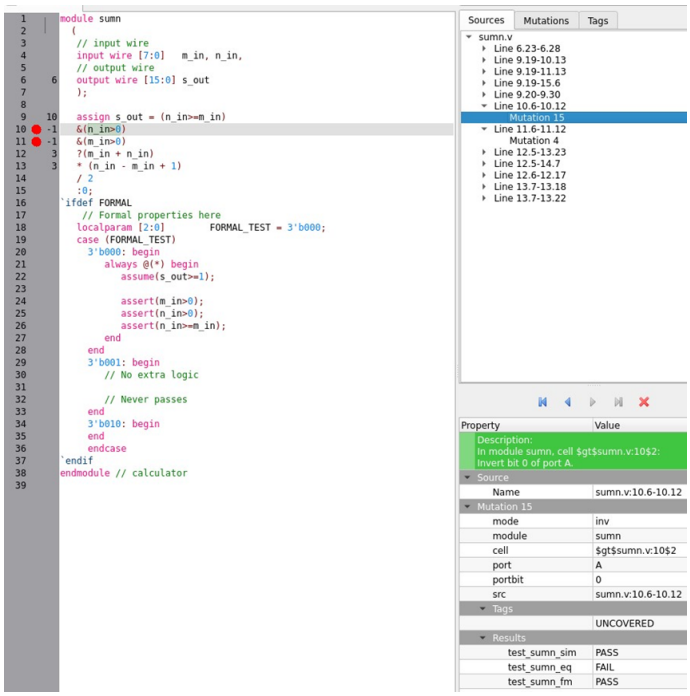


Рис. 1: "Горячие точки"

Красные точки это и есть «UNCOVERED». Кликните по ней. Справа появится мутация, которая отмечена красным. Внизу справа, в зелёном поле можно найти краткое описание мутации. Ещё ниже представлена вся мутация по портам и битам.

Аналогичное представление только для командной строки создается:

```
$ mcy source sumn.v
==> Retrieving source info
-> Reading configuration file.
-> Connecting to database.
-> Extract coverage info.
-> Display source file with info.
| module sumn
| (
    |   // input wire
    |   input wire [7:0]    m_in, n_in,
    |   // output wire
    6|  output wire [15:0] s_out
    |   );
```

```
|
    10| assign s_out = (n_in>=m_in)
! -1|   &(n_in>0)
! -1|   &(m_in>0)
    3|  ?(m_in + n_in)
    3|  * (n_in - m_in + 1)
     |  / 2
     |  :0;
| endmodule // calculator
```

Положительные числа в левом столбце указывают на мутации, помеченные как COVERED, отрицательные числа указывают на UNCOVERED.

Вы можете попытаться улучшить тестовый стенд в sumn_sim_tb.v, чтобы добиться лучшего покрытия. После изменения этого файла не забудьте аннулировать старые результаты, выполнив:

```
$ mcy purge
```

Поскольку мутации генерируются случайным образом, чем лучше ваш охват, тем больший размер требуется для поиска непроверенных случаев. Если вы достигнете 100%, попробуйте увеличить размер дальше.

2.7 Интеграция второго теста

Часто у вас будет целая коллекция тестов разного объема и строгости. Все они могут быть интегрированы в единый проект MCY, чтобы получить показатель покрытия для набора тестов в целом. В этом разделе мы добавим второй, более продолжительный, но более тщательный тестовый стенд для увеличения показателя покрытия.

test_sumn_fm это формальный тестовый стенд, который полностью проверяет, что модуль соответствует формальному определению желаемого поведения. Так как это значительно увеличивает время выполнения примера, test_sumn_fm отключен по умолчанию в config.mcy. Его можно включить или отключить, установив переменную use_formal, определенную в config.mcy.

Будет необходим скрипт для запуска теста test_sumn_fm.sh

```
#!/bin/bash
exec 2>&1
set -ex

bash $SCRIPTS/create_mutated.sh -o mutated.il

ln -s ../../test_sumn_fm.sv ../../
test_sumn_fm.sby .
sby -f test_sumn_fm.sby

gawk ”{ print 1, \$1; }” test_sumn_fm/status
>> output.txt

exit 0
```

Здесь мы не используем опцию *-c* для скрипта create_mutated, поскольку нам нужен мутированный модуль с тем же интерфейсом, что и исходный sumn модуль для замены в тестовом стенде.

Для запуска теста также необходим скрипт test_sumn_fm.sby, в котором показано как sby будет выполнять формальную проверку.

```
[options]
mode bmc
depth 1
expect pass,fail

[engines]
smtbmc boolector

[script]
read_verilog -sv test_sumn_fm.sv
read_ilang mutated.il
prep -top test_sumn_fm_tb
flatten
opt -fast

[files]
test_sumn_fm.sv
mutated.il
```

Нам понадобятся database/ и tasks/ директории для пробного запуска, но на этот раз мы можем использовать существующий

проект MCY для их создания.

Если файл database/design.il не существует, запускайте mcy init, чтобы создать его.

Далее, запускайте mcy task -k test_sim 1.

```
$ mcy task -k test_sumn_sim 1
==> Run tasks from queue
  -> Reading configuration file.
  -> Connecting to database.
 ==> Running task 3eec484d-1377-44d7-acf9-
c676a1d5e9d7 (test_sumn_sim)
 1 1 mutate -mode none
  ==> Finishing task 3eec484d-1377-44d7-acf9-
c676a1d5e9d7 (test_sumn_sim)
  1 1 PASS mutate -mode none
  -> Finished running task.
```

Обратите внимание на напечатанный uuid задачи. Войдём в каталог tasks/$uuid, созданный этой командой, и запустим `$ bash ../../test_sumn_fm.sh`, чтобы проверить правильность работы теста (он должен вернуть PASS, поскольку задача 1 всегда имеет режим mutate -mode none, который не вызывает мутаций).

Если все работает, как ожидалось, мы можем добавить этот тест в конфигурацию MCY. В config.mcy, в разделе [options], снова уменьшим размер и добавим новый тег “FMONLY”:

```
[options]
size 10
tags COVERED UNCOVERED NOCHANGE EQGAP FMONLY
```

В нижней части файла добавим новый раздел для нового теста:

```
[test test_sumn_fm]
expect PASS FAIL
run bash $PRJDIR/test_sumn_fm.sh
```

Наконец, мы скорректируем раздел [logic] для использования этого нового теста. Сначала определим переменную use_formal. Таким образом, мы можем включать и выключать этот дорогостоящий тест по своему желанию:

```
[logic]
use_formal = True
```

После выполнения двух исходных тестов, но до применения тегов, вставьте новый фрагмент кода:

```
sim_okay = result(”test_sumn_sim”) == ”PASS”
eq_okay = result(”test_sumn_eq”) == ”PASS”

if sim_okay and use_formal:
    sim_okay = (result(”test_sumn_fm”) ==
”PASS”)
    if not sim_okay:
        tag(”FMONLY”)
```

Дополнительный тест test_sumn_fm будет запускаться только в случае, когда sim_okay и use_formal истинны, т. е. тестовый стенд моделирования не выявил никаких проблем с модулем. Это вызовет дополнительную нагрузку на стенд, но также это означает, что этот длительный тест будет выполнен только для небольшой части мутаций. Мутации, для которых только формальный тест смог обнаружить проблему, помечаются “FMONLY” чтобы иметь возможность отследить, какие тесты охватывают те или иные мутации.

Проверим, правильно ли работает эта новая конфигурация:

```
$ mcy purge
$ mcy init
$ mcy run
```

В зависимости от случайно сгенерированных мутаций, могут быть некоторые мутации, помеченные как “FMONLY”, в вашем первоначальном наборе из 10. Проверьте, отображается ли следующая строка в mcy status:

```
Tagged 1 mutations as ”FMONLY”.
```

При необходимости можно сгенерировать новые мутации, повторно выполнив приведенные выше команды или увеличив количество мутаций.

Если все работает правильно, можно вернуть размер набора мутаций к его исходному значению.

```
[options]
size 1000
```

Запуск MCY теперь потребует значительно больше времени, поэтому не забудьте включить параллелизм:

```
mcy reset
mcy run -j$(nproc)
```

На этот раз станет возможно достичь 100% покрытия, поскольку формальный тестовый модуль всесторонне проверяет, корректен ли результат для любой возможной комбинации входных данных.

3 Интерпретация результатов

Допустим, мы получили вот такой результат.

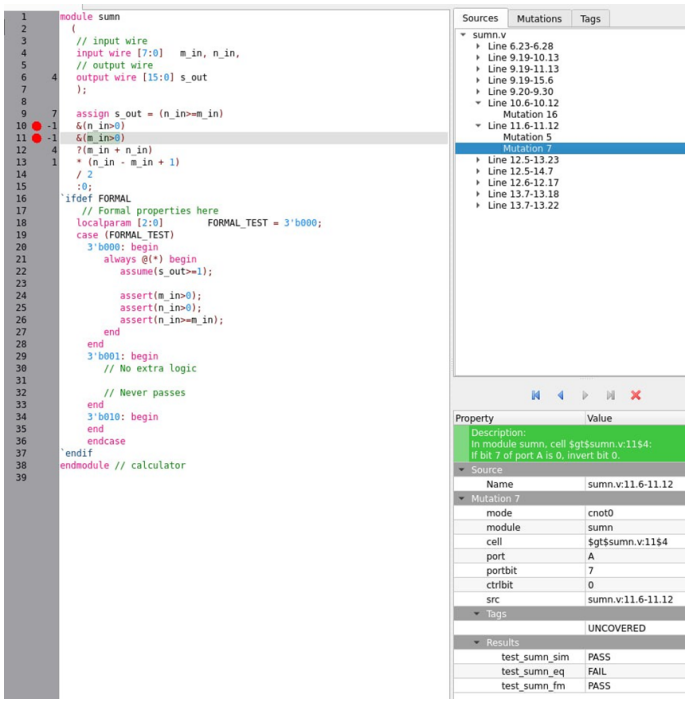


Рис. 2: Пример описания мутации

Здесь мутация помечена как “непокрытая” (“UNCOVERED”). Поле Description в нижнем правом углу показывает, что мутация внесена в операцию сравнения ($gt), в левую часть (port A). Бит 7 заменен на функцию cnot0 от бита 0 (cnot0 - это xor нуля и аргумента). Т.е., вместо бита 7 в m_in подставили бит 0, и ничего в работе модуля не изменилось (или тестбенч это не обнаружил). Следовательно мутация помечена как “не покрытая” (“UNCOVERED”). Возможно биты 0 и 7 в m_in совпали по значению.

На данном этапе можно довольно быстро достичь 100% покрытия.

```
-> Getting database statistics.
Database contains 48 cached results.
Database contains 17 cached ”FAIL” results
for ”test_sumn_eq”.
Database contains 4 cached ”PASS” results for
”test_sumn_eq”.
Database contains 2 cached ”FAIL” results for
”test_sumn_fm”.
Database contains 4 cached ”PASS” results for
”test_sumn_fm”.
Database contains 15 cached ”FAIL” results
for ”test_sumn_sim”.
Database contains 6 cached ”PASS” results for
”test_sumn_sim”.
Tagged 17 mutations as ”COVERED”. Tagged 2
mutations as ”FMONLY”.
Tagged 4 mutations as ”NOCHANGE”.
-> Print report
Coverage: 100.00%
```

Если это так, пробуем увеличить количество мутаций. Задайте новое число в

```
[options]
size 100
```

И, скорее всего, обнаружим новые “не покрытые” мутации:

```
-> Getting database statistics.
Database contains 231 cached results.
Database contains 82 cached ”FAIL” results
for ”test_sumn_eq”.
Database contains 18 cached ”PASS” results
for ”test_sumn_eq”. Database contains 7
cached ”FAIL” results for ”test_sumn_fm”.
Database contains 24 cached ”PASS” results
for ”test_sumn_fm”.
Database contains 69 cached ”FAIL” results
for ”test_sumn_sim”.
Database contains 31 cached ”PASS” results
for ”test_sumn_sim”.
Tagged 76 mutations as ”COVERED”.
Tagged 7 mutations as ”FMONLY”.
Tagged 18 mutations as ”NOCHANGE”.
Tagged 6 mutations as ”UNCOVERED”.
-> Print report
Coverage: 92.68%
```

Важно увидеть, какие мутации не вызвали ошибок, чем это обусловлено, и, что следут делать.

4 Используется ли это?

Здесь приводится небольшой обзор публикаций, найденных в интернете, так или иначе связанных с внедрением мутационного тестирования.

Хороший отчет по внедрению мутационного тестирования в компании Авито представлен в работе [1].

Пример использования в Sipios (финтех) [5].

Пример патента, в котором предлагается использовать мутационное тестирование в отсутствие целевой аппаратуры (неисправности вносятся в модели целевой аппаратуры) [3].

> Поставленная задача изобретения решается тем, что в проекты ПЛИС, реализованные на языках описания аппаратуры, намеренно вносят модели неисправностей; затем проводят тестирование с целью оценки вероятности обнаружения внесенных моделей неисправностей тестируемой аппаратурой или ПО; на языке описания аппаратуры... В работе [2]

Исследуется задача построения тестов с гарантированной полнотой для последовательностных цифровых систем. Рассматриваются тесты, нацеленные как на обнаружение в логических схемах низкоуровневых неисправностей различных типов, так и на более высокий функциональный уровень. Для построения тестов по модели логической схемы используется мутационный подход...

Полученные экспериментальные результаты позволяют заключить, что возможно эффективное использование мутационного подхода при синтезе качественных тестов для последовательностных схем: тест, построенный относительно мутаций логической схемы, дополняется последовательностями, покрывающими непокрытые переходы соответствующего конечного автомата, пока длина исходного теста не увеличится в 2–4 раза. Согласно результатам экспериментов, можно ожидать, что тест, расширенный подобным образом, будет полным относительно большого количества ошибок на функциональном уровне и, соответственно, тесты, построенные с помощью такого подхода, могут быть использованы в качестве проверки правильности функционирования систем как на высоком, так и на низком уровнях абстракции.

5 Заключение

Различные аспекты мутационного тестирования довольно широко представлены в интернете. Выполняя эту работу, я просто следовал указаниям, сопровождающим проект MCY. Мне удалось достаточно быстро создать работающий стенд, но, как, оказалось, в нём была ошибка. Эту ошибку я выявил на этапе проверки эквивалентности. Это явилось хорошим стимулом для дальнейшей работы и придало уверенности в том, что тестируемый тестовый стенд способен обнаруживать изменения в поведении, которые внесла мутация.

Список литературы

1. Александр Асмаков @AVAsmakov. «Мутационное тестирование: опыт внедрения на 1500 сервисов». В: (). URL: https://habr.com/ru/companies/avito/articles/ 650073/.
2. Е.М. Винарский А.В. Лапутенко. «СИНТЕЗ ТЕСТОВ ДЛЯ ЦИФРОВЫХ СИСТЕМ НА ВЫСОКОМ И НИЗКОМ УРОВНЯХ

АБСТРАКЦИИ». В: *ПРОЕКТИРОВАНИЕ И ДИАГНОСТИКА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ*. ВЕСТНИК ТОМСКОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА (2019). УДК 519.2. DOI: 10.17223/19988605/47/13. URL: file: /// home/serge/Downloads/sintez-testov-dlya-tsifrovyh-sistem-navysokom-i-nizkom-urovnyah-abstraktsii.pdf.

3. Недорезов Дмитрий Александрович. «СПОСОБ МУТАЦИОННОГО ТЕСТИРОВАНИЯ РАДИОЭЛЕКТРОННОЙ АППАРАТУРЫ И ЕЕ УПРАВЛЯЮЩЕГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ». В: *Владелец патента: Открытое акционерное общество "Информационные спутниковые системы "имени академика М.Ф. Решетнева"(RU)* (29 апр. 2014). URL: https://yandex.ru/patents/doc/RU2549523C1_20150427.

4. С. Балакший. «К формальной проверке». В: *https://fpga-systems.ru/* (28 янв. 2024). URL: https://fpga-systems.ru/publ/raznoe/poznavatelnoe/k_formalnoj_ proverke/16-1-0-224.

5. «Мутационное тестирование. Теория+практикум». В:*testengineer.ru*().URL: https://testengineer.ru/mutacionnoe-testirovanie-i-kak-ego-provodit/.

Опенсорс для ПЛИС... и наоборот

Гуров В.В.

va1ery@ya.ru

Обсуждение и комментарии: link

(по материалам компании Lushay Labs)

Отладочные платы, вместо микроконтроллеров содержащие программируемые логические интегральные схемы - ПЛИС, или FPGA (Field-Programmable Gate Array), уже сегодня могут стать не менее популярными, чем платы Arduino. В значительной мере этому способствует наличие на рынке доступных плат китайского производства наряду с программными средствами разработки с открытым исходным кодом (Open Source Software - OSS).

Одной из таких плат является недорогая (на момент написания статьи цена составляла порядка 2 тыс. р) плата Tang Nano 9K, разработанная на основе ПЛИС GW1NR-9 компании Gowin Semiconductor. Помимо коннекторов для подключения разнообразных дисплеев и встроенного программатора, эта плата оснащена флеш-памятью, двумя кнопками и шестью светодиодами, конфигурируемыми пользователем. Сама ПЛИС содержит 8640 логических блоков LUT4, что вполне отвечает потребностям начинающих пользователей.

Плата разработки Tang Nano 9K

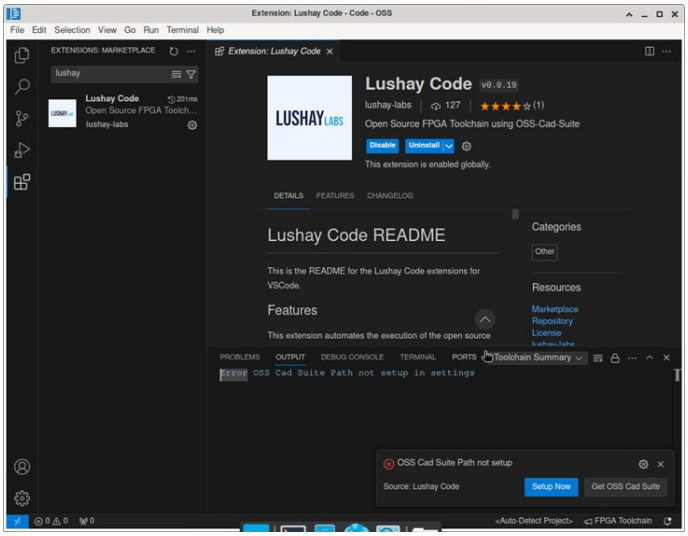
Что касается набора инструментов с открытым исходным кодом, то на сегодняшний день он включает в себя все компоненты, необходимые для каждого из этапов разработки: ПО Yosys для синтеза, NextPnR для размещения и маршрутизации цепей и Apicula, реконструирующее архитектуру и предоставляющее инструменты генерации битового потока (битстрима) для программирования ПЛИС компании Gowin, которое в свою очередь осуществляется посредством ПО openFPGALoader.

Данный набор инструментов легко конфигурируется, для чего по адресу https://

github.com/YosysHQ/oss-cad-suite-build/releases?ref=learn.lushaylabs.com достаточно выбрать и скачать архив OSS-CAD-Suite для соответствующей платформы (MacOSX, Linux, Windows), распаковать его и добавить к списку значений переменной окружения PATH абсолютный путь к находящейся в корне распакованного архива папке bin, например, так:

```
export PATH="/home/имя_пользователя/oss-cad-suite/bin:$PATH"
```

Удобнее это можно сделать, добавив к редактору Visual Studio Code расширение Lushay Code, которое позволяет задать вышеуказанный путь и запускать любой из инструментов OSS-CAD-Suite. Чтобы начать настройку данного плагина кликните на значок FPGA Toolchain в правом нижнем углу окна VS Code.



Настройка плагина Lushay Code

Ниже показано, как с их помощью можно быстро научиться создавать проекты на языке Verilog.

Начнем с простого примера двоичного счетчика, разряды которого отображаются с помощью светодиодов.

1. На плате Tang Nano 9K имеется 6 светодиодов, и если каждый будет представлять в нашем счетчике один двоичный разряд, то мы сможем отображать числа от 0 до 111111 (63 в десятичном формате).

2. В качестве входных сигналов используем тактовые импульсы, получаемые от встроенного в плату генератора.

3. Частота последнего составляет 27 МГц, поэтому если фиксировать каждый такт, то счетчик будет заполняться примерно 421 тыс. раз каждую секунду, -слишком быстро для визуализации процесса. Чтобы заставить его считать не так быстро, скажем, раз в полсекунды, нужно пропустить 13 500 тыс. тактов прежде чем увеличить значение счетчика на единицу.

Теперь, когда у нас есть общий план, давайте создадим в пустой папке проекта файл под названием counter.v и запишем в него то, что нам известно.

```
module top //объявление модуля
(
input clk, //вход - тактовый сигнал
output [5:0] led //выход - 6 светодиодов
);
localparam WAIT_TIME =
13500000; //"замедляющий" параметр WAIT_TIME
reg [5:0] ledCounter = 0; //создаем и
обнуляем шестиразрядный счетчик
endmodule //конец модуля
```

Приведенный выше код с комментариями, написанный на языке Verilog, определяет модуль под названием top с одним входом clk для тактового сигнала и шестью выходами на светодиоды led. Эти обозначения будут позднее сопоставлены с физическими выводами ПЛИС в другом файле - файле ограничений (привязок).

Формат определения выхода led имеет вид [MSB:LSB], благодаря чему группа из шести светодиодов может быть представлена не как led5, led4, ...led0, а как битовый массив размером [5:0]. Здесь крайний левый бит (с индексом 5) называется старшим (most significant bit, MSB) , а крайний правый бит (с индексом 0) - младшим значащим битом

(least significant bit, LSB).

Следующая строка определяет локальную константу WAIT_TIME, выше мы уже подсчитали, что нам потребуется 13,5 млн тактов на каждый шаг счетчика, чтобы достичь желаемой задержки в полсекунды.

Последняя строка - это и есть наш фактический счетчик, и мы снова используем тот же формат [5:0], чтобы задать переменную (регистр) шириной в 6 бит.

Но это еще не все. На самом деле нам понадобится еще один счетчик тактов. Двоичное представление числа 13 500 000 содержит 24 разряда, поэтому нам нужно создать счетчик размером [23:0].

```
reg [23:0] clockCounter = 0; //создаем и обнуляем 24-разрядный счетчик
```

Итак, мы описали все компоненты проекта. Теперь следует задать правила увеличения значений наших счетчиков. Это можно сделать с помощью блока always:

```
always @(posedge clk) begin
  clockCounter <= clockCounter + 1;
  if (clockCounter == WAIT_TIME) begin
    clockCounter <= 0;
    ledCounter <= ledCounter + 1;
  end
end
```

Блок always начинается со списка чувствительности. По сути, это условия для того, чтобы сработала описываемая им схема. В нашем случае всякий раз, когда на линии тактовой частоты появляется положительный импульс, должно произойти следующее.

Прежде всего значение счетчика тактов увеличим на единицу, а затем проверим, достигло ли оно заданного выше значения WAIT_TIME. Если так, тогда сбрасываем счетчик тактов на ноль, а к значению шестиразрядного счетчика прибавляем единицу. При этом нет необходимости перезагружать шестиразрядный счетчик, поскольку он автоматически вернется к 0 сразу, как только увеличится до 63.

Отметим, что оператор "<=" не похож на стандартный оператор присваивания в большинстве языков программирования. Данный оператор устанавливает значение для регистра clockCounter, которое будет передаваться ему только при следующем тактовом сигнале. Это означает, что если мы увеличим его от 0 до 1 в первой строке блока, значение clockCounter для оставшейся части текущего блока по-прежнему будет равно 0 и увеличится на единицу только на следующем такте. То же самое происходит, когда мы увеличиваем значение шестиразрядного счетчика, изменение которого будет замечено только при следующем тактовом сигнале.

Есть способ немедленно присвоить значение, используя вместо этого блокирующий оператор "=", но при работе с регистрами принято использовать неблокирующие операторы присваивания.

Последнее, что нужно сделать для завершения нашего модуля, - это подключить регистр к светодиодам.

```
assign led = ledCounter;
```

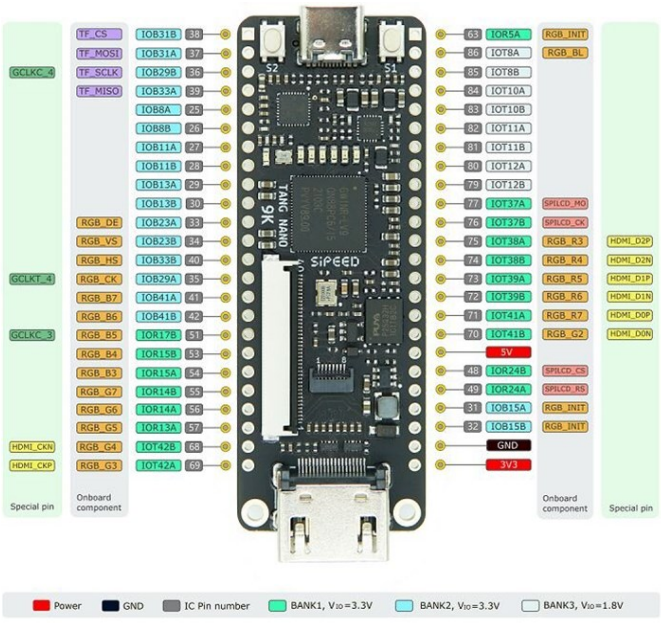
За пределами блока always мы используем assign и = для определения значения wires. Провода в отличие от регистров не могут хранить значения, поэтому нужно указать, к чему они подключены.

Окончательный код должен выглядеть следующим образом:

```
module top
(
input clk,
output [5:0] led
);
localparam WAIT_TIME = 13500000;
reg [5:0] ledCounter = 0;
```

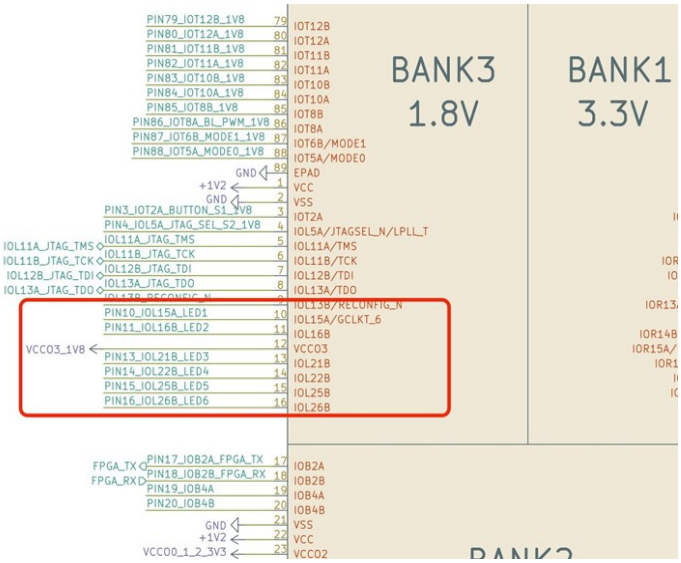
```
reg [23:0] clockCounter = 0;
always @(posedge clk) begin
  clockCounter <= clockCounter + 1;
  if (clockCounter == WAIT_TIME) begin
    clockCounter <= 0;
    ledCounter <= ledCounter + 1;
  end
end
assign led = ledCounter;
endmodule
```

Следующий файл, который нам нужно создать, - это файл counter.cst, в котором нам нужно будет привязать физические контакты микросхемы к цепям clk и led. Чтобы сделать это, нам нужно иметь некоторую информацию о портах ввода/вывода (General Purpose Input/Output, GPIO), а именно их номера. Для платы это довольно просто, все номера GPIO перечислены на следующем рисунке:



Распиновка GPIO

Но в нашем примере мы не используем GPIO-порты платы, мы используем контакты самой ПЛИС, которые подключены к встроенным компонентам (тактовому генератору и светодиодам).



Распиновка встроенных светодиодов

Как видим, встроенные светодиоды подключены к контактам микросхемы 10,11,13,14,15 и 16. Аналогично для привязки цепи clk мы находим контакт 52 в секции xtal (кварцевый генератор).

Распиновка для тактового генератора

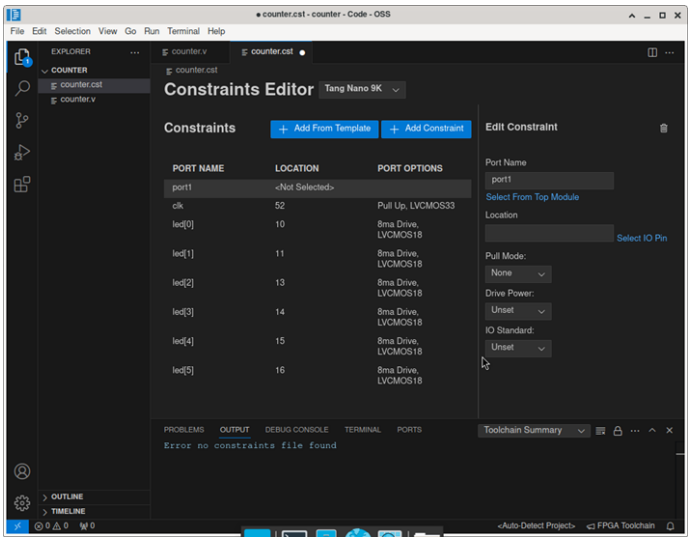
По завершении нашего мини-исследования мы можем создать cst-файл, в котором мы определим привязки следующим образом:

```
IO_LOC "clk" 52;
IO_PORT "clk" PULL_MODE=UP;
IO_LOC "led[0]" 10;
IO_LOC "led[1]" 11;
IO_LOC "led[2]" 13;
IO_LOC "led[3]" 14;
IO_LOC "led[4]" 15;
IO_LOC "led[5]" 16;
```

В этом файле мы назначаем проводу clk вывод 52 с подтягивающим резистором, а также определяем 6 разрядов нашего светодиодного выхода, используя найденные нами выше номера выводов ПЛИС. Заметим, что добавив к каждому разряду индекс бита в квадратных скобках, мы сократили наш код verilog, задав присваивание значений

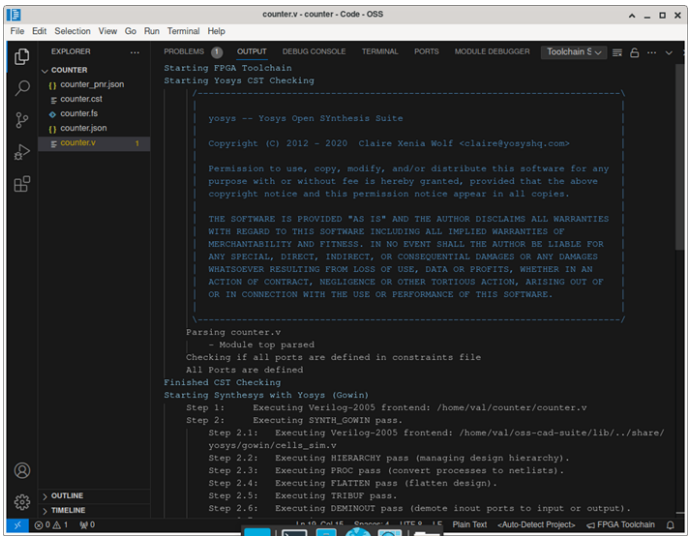
сигналов светодиодам одной строкой вместо шести.

Мастер Lushay Code позволяет легко создать файл ограничений (Constraints) из шаблона, для этого достаточно выбрать нужную плату из выпадающего списка, а затем выбрать нужные шаблоны и при необходимости отредактировать их.

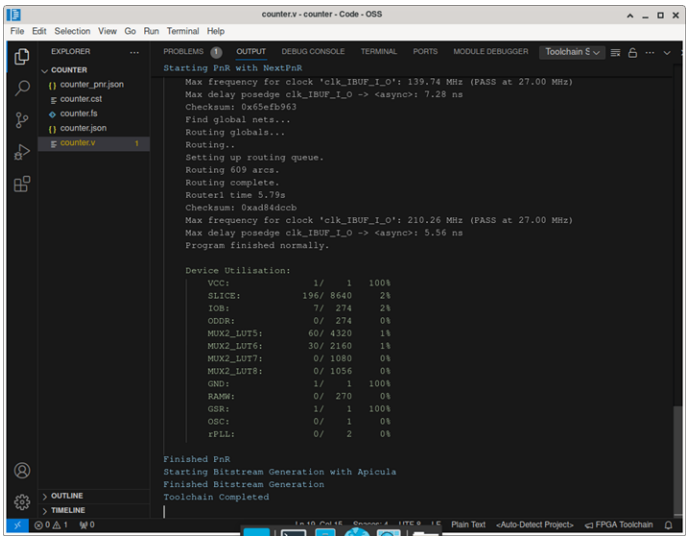


Редактор файла ограничений

Теперь все готово для запуска нашего проекта. Кликаем на уже знакомый нам значок FPGA Toolchain в правом нижнем углу и в меню команд VS Code выбираем Build Only.



Запуск фазы синтеза



Генерация битстрима завершена

Для загрузки битстрима в подключенную через USB-C плату с ПЛИС можно аналогичным образом выбрать команду Build and Program, либо Program Only если битстрим уже сгенерирован.

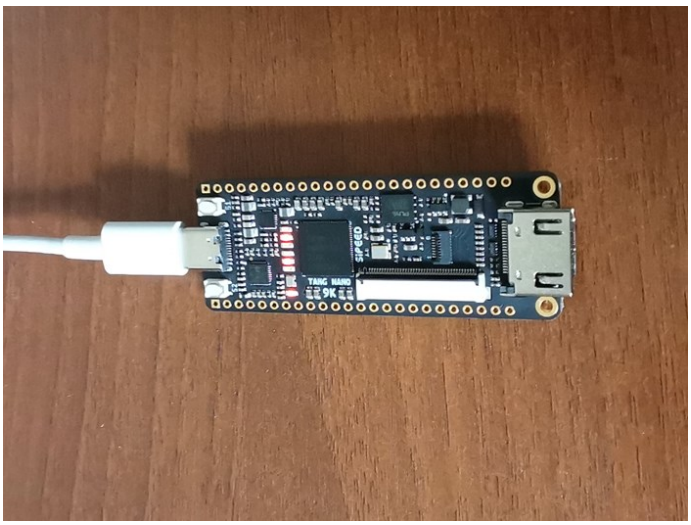
Если путь к папке oss-cad-suite/bin добавлен к переменной PATH, можно воспользоваться утилитой openFPGALoader, запустив ее в окне терминала, при необходимости предварив командой sudo.

openFPGALoader загрузил битстрим

Битстрим загружен и мы видим, что наш проект работает, правда, не совсем так как хотелось бы. Исправим логическую ошибку, заключающуюся в том, что мы не учли уровень активного сигнала для светодиодов - по умолчанию он высокий! Поэтому добавим инверсию ~ (логическое НЕ) в предпоследнюю строку модуля top.

```
assign led = ~ledCounter;
```

Наш модуль работает!



Наш модуль работает!

Источник:

1. Tang Nano 9K: Getting Setup https://learn.lushaylabs.com/getting-setup-with-the-tang-nano-9k/

Buildroot это просто

Белоусов Олег

belousov.oleg@gmail.com

Обсуждение и комментарии: link

Читая чат FPGA-Systems Embedded увидел достаточно много вопросов по использованию buildroot (далее просто BR). Но что удивило, так это ответы! Точней созданные и распространяемые мифы вокруг него. Основные: это неудобно, сложно поддерживать и развивать, долго разбираться. И вишенка на торте: “BR нужен только для сборки toolchain и rootfs. Ядро и загрузчик нужно собирать отдельно”. Я работаю с BR довольно давно (около 10 лет точно) и за это время накопил какой-то опыт, с которым и хочу поделиться. Все написанное не догма, но возможно кому-то это сможет помочь.

Предыстория. Мне для одного проекта понадобился прототип на TRX Duo. Это китайский клон SDR Red Pitaya с Zynq 7010 и ADC, DAC. Так как это клон, он использует весь набор софта от Red Pitaya, а у нее в PS части стоит Linux Alpine. Но я традиционно использую BR.

Я создал репозиторий и буду делать коммиты в реальном времени, по коммитам можно посмотреть сколько это занимает по времени (все честно!). В результате мы соберем все для запуска Linux на Zynq: toolchain, uboot, kernel и rootfs.

Репозиторий: https://github.com/strijar/TRX-Duo

Структура директорий

Есть практика когда программисты правят непосредственно BR как им нужно для сборки проекта. Это самое просто и очевидное. Но я очень советую использовать механизм external при работе с BR. Это означает, что вы никаким образом не завязаны на основное дерево исходников, т.е. никак их не меняете. Все конфиги, патчи, скрипты которые вам будут нужны - лежат отдельно. Поверьте, в будущем вам это сильно облегчит жизнь. Ну хотя бы если вы захотите более свежую версию BR.

Поэтому создаем структуру директорий:

```
br2_external/
    board/
        trx_duo/
            linux/
            uboot/
    configs/
```

В корне файлы `.gitignore`, скрипт `config.sh` и несколько файлов в `br_external`. Скрипт `config.sh` довольно примитивный, его задача скачать BR нужной нам версии, распаковать его и сконфигурировать с нашим конфигом. Там же передается переменная `$BR2_EXTERNAL` которая указывает на нашу external "систему".

Запускаем скрипт `config.sh` и получаем ошибку:

```
Makefile:1012: *** "Can't find
trx_duo_defconfig".  Останов.
```

Это нормально, так как мы еще не создали наш конфиг. Но смотрите у нас появилась директория `buildroot-2024.02` откуда и будем отталкиваться.

Трюк. Когда вы работаете с BR ему каждый раз приходится выкачивать все исходники для его сборки. Если вы часто пересобираете или у вас просто несколько разных проектов под разные устройства это может быть довольно внушительный объем трафика (да и суммарно на диске много занимает). Все скачанные архивы, по умолчанию складываются в директории `dl` внутри дерева исходников BR. Я просто создаю в своей домашней директории папку `dl` и после распаковки дерева BR создаю внутри сим.линк на нее (`cd buildroot-2024.02 && ln -s ~/dl dl`). В результате все архивы от всех сборок BR лежат в одном месте, экономя трафик и место на диске.

Конфиг по умолчанию

За основу возьмем конфиг для платы Avnet Zed. Копируем buildroot-2024.02/ configs/zynq_zed_defconfig в нашу папку br2_external/configs/trx_duo_defconfig и проверяем.

Ремарка. Я очень советую постепенный процесс настройки. Т.е. берем заведомо рабочий конфиг и идем этапами с проверкой каждого этапа - собирается и работает ли. Это спасает от ситуаций, когда вы все сделали, а что-то пошло не так и приходится все начинать сначала. Промежуточные этапы можно сохранять в бинарном виде просто архивом, чтобы потом быстро "откатиться"

Запускаем config.sh и теперь все должно пройти без ошибок. У нас появилась директория output_trx_duo, в которой и будут все результаты сборки системы. Заходим туда и собираем toochain:

```
cd output_trx_duo
make toolchain
```

Ремарка. По умолчанию BR использует собственный toolchain который собирает сам из исходников. Это занимает какое-то время (особенно если вам нужен C++ тоже). Но есть возможность использовать внешний, уже собранный toolchain. Это может быть ваш собственный. И вы тогда вы просто в конфиге прописываете путь до него. Или BR может сам скачать и поставить на выбор toolchain от ARM или Linaro. Это сильно экономит время, но есть подводные камни! Например я обнаружил, что в системе собранной с toolchain от Linaro не работает как надо OpenSSH демон, потому что есть ошибка в его libc

Конфиг uboot по умолчанию

За основу конфига uboot возьмем тот, что предлагается по умолчанию. Для этого позволим BR скачать исходники uboot и настроить его с конфигом по умолчанию:

```
make uboot-configure
```

Ремарка. У BR в make довольно много “целей”. Основные: `*-menuconfig *-dirclean *-rebuild`. Для конфигурации (например есть linux-menuconfig или uboot-menuconfig) для очистки результатов сборки и для пересборки без очистки соответственно. Много интересного покажет `make help`

Копируем build/uboot-custom/.config к себе br2_external/board/trx-duo/uboot.config и прописываем в настройках BR чтобы он использовал его:

```
make menuconfig

Bootloaders -->
    [*] U-Boot
        U-Boot configuration: Using a custom
board (def)config file
        ($(BR2_EXTERNAL_TRX_DUO_PATH)/board/
trx-duo/uboot.config)            Configuration
file path
```

Ремарка. Переменная $(BR2_EXTERNAL_TRX_DUO_PATH) формируется самим BR автоматически. Часть TRX_DUO это name из файла external.desc

Теперь, если мы будем менять что-то в настройках uboot они не потеряются если мы очистим результаты его сборки, потому что они находятся отдельно. Проверим что собирается:

```
make uboot-dirclean
make uboot
```

PS init для uboot

Мы же помним, что для запуска Zynq используется FSBL. Но и тут в BR все сделано до нас. Нам нужно просто положить в нашей директории файлы `ps7_init_gpl.c` и `ps7_init_gpl.h` и указать их в конфиге uboot. Эти файлы, кстати, проще всего достать прямо из .xsa (на самом деле это zip архив) не прибегая к формированию FSBL средствами Vitis. Копируем два этих файла в нашу папку `br2_external/board/trx-duo/uboot` и настраиваем uboot:

```
make uboot-menuconfig

ARM architecture -->
    ($(BR2_EXTERNAL_TRX_DUO_PATH)/board/trx
-duo/uboot/ps7_init_gpl.c)    Zynq/ZynqMP PS
init file(s) location
```

Потом пересобираем и обновляем наш конфиг чтобы не потерялось:

```
make uboot-rebuild
make uboot-update-defconfig
```

DTS для uboot

Ремарка. Еще одна причина почему мне нужна была самособранная система для TRX Duo в том, что у моего экземпляра почему то не работает штатный USB Serial который сидит на PS UART0 и используется как консоль. А без консоли совсем грустно! Поэтому мне нужно перенастроить uboot и linux ядро чтобы они использовали UART1. Это делается в файлах DTS. Но на мое счастье Avnet Zed по умолчанию тоже использует UART1 для консоли, так что ничего править не нужно, но я все равно сделаю собственные DTS на будущее.

Копируем build/uboot-custom/arch/arm/dts/zynq-zed.dts к себе в br2_external/board/trx-duo/uboot/trx-duo.dts и настраиваем сначала в основном конфиге:

```
make menuconfig
Bootloaders →
    ($(BR2_EXTERNAL_TRX_DUO_PATH)/board/trx-
duo/uboot/trx-duo.dts) Device   Tree Source
file paths
    (DEVICE_TREE=trx-duo)  Custom make op-
tions
```

И в конфиге uboot:

```
make uboot-menuconfig
Device Tree Control -->
    Default Device Tree for DT control: trx-
duo
    List of device tree files to include for
DT control: trx-duo
```

Пересобираем и обновляем конфиги:

```
make uboot-rebuild
make uboot-update-defconfig
make update-defconfig
```

На этом этапе у нас в папке images получился работающий загрузчик. Давайте это проверим. Возьмем SD карту с первым разделом в FAT и помеченным как загрузочный. Зальем туда файлы boot.bin u-boot.img вставляем карту в устройство, скрещиваем пальцы и включаем:

```
U-Boot SPL 2023.01 (Apr 03 2024 - 13:05:25
+0300)
Silicon version:        3
Trying to boot from MMC1
spl_load_image_fat_os: error reading image
system.dtb, err - -2

U-Boot 2023.01 (Apr 03 2024 - 13:05:25 +0300)

CPU:   Zynq 7z010
Silicon: v3.1
Model: Avnet ZedBoard board
DRAM:  ECC disabled 512 MiB
```

Дальше могут идти какие-то ошибки, но этого уже достаточно. Загрузчик работает. Следующий этап сборка ядра linux.

Конфиг linux по умолчанию

Согласно нашей идеологии вытащим конфиг для linux ядра отдельно к себе, пока ничего там править не будем, но на будущее.

```
make linux-configure
```

Копируем конфиг build/linux-custom/.config к себе br2_external/board/trx-duo/linux.config и правим в настройках:

```
make menuconfig

Kernel -->
    Kernel configuration: Using a custom
(def)config file
    Configuration file path:
    $(BR2_EXTERNAL_TRX_DUO_PATH)/board/trx-
duo/linux.config
```

Проверяем и обновляем конфиг:

```
make linux-rebuild
make update-defconfig
```

DTS для linux

Как мы уже решили, DTS будем делать собственные, поэтому берем что предлагают по умолчанию из build/linux-custom/arch/arm/boot/dts/zynq-zed.dts

и сохраняем себе в br2_external/board/trx-duo/linux/trx-duo.dts далее настраиваем конфиг:

```
make menuconfig

Kernel -->
    In-tree Device Tree Source file names:
все удалить
    Out-of-tree Device Tree Source file
paths:
     $(BR2_EXTERNAL_TRX_DUO_PATH)/board/trx-
duo/linux/trx-duo.dts
```

Проверяем сборку, обновляем конфиг:

```
make linux-rebuild
make update-defconfig
```

На этом моменте в папке images добавляются файлы uImage и собранный из DTS файл trx-duo.dtb и можно их проверить. Копируем их на нашу SD карту, создаем там-же папку extlinux и добавляем в нее файл extlinux.conf включаем:

```
Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 6.1.30-xilinx
(master@localhost.localdomain) (arm-buildroot
-linux-gnueabihf-gcc.br_real (Buildroot -
gedbd68c) 12.3.0, GNU ld (GNU Binutils) 2.40)
#1 SMP PREEMPT Wed Apr  3 13:
19:49 MSK 2024
CPU: ARMv7 Processor [413fc090] revision 0
(ARMv7), cr=18c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT
aliasing instruction cache
OF: fdt: Machine model: Avnet ZedBoard board
Memory policy: Data cache writealloc
cma: Reserved 16 MiB at 0x1f000000
```

В самом конце будет ошибка, потому что у нас нет раздела с rootfs. Сейчас этим займемся.

Post build/image файлы

Еще одно преимущество использования BR, что в итоге вы получаете один файл с образом для SD карты. Со всеми разделами и всем необходимым. Который достаточно просто залить на карту без возни с fdisk mkfs и проч. Образ создается (обычно скриптом `post-image.sh`) на основе описания из файла `genimage.cfg` Еще для работы uboot нужен файл `extlinux.conf` (там имя файла linux ядра и dtb файла) который мы закидывали вручную на предыдущем этапе. Но это можно сделать автоматически при сборке системы с помощью скрипта `post-build.sh`. Эти файлы с некоторыми вариациями уже есть в дереве исходников BR, можно взять их за основу и подправить под свои задачи. В нашем случае я это сделал за вас. Закидываем к себе в `br2_external/board/trx-duo/` и правим конфиг:

```
make menuconfig

System configuration -->
    Custom scripts to run before creating filesystem images:
    $(BR2_EXTERNAL_TRX_DUO_PATH)/board/trx-duo/post-build.sh
    Custom scripts to run after creating filesystem images:
    $(BR2_EXTERNAL_TRX_DUO_PATH)/board/trx-duo/post-image.sh
```

Собираем все вместе

```
make
```

На этом этапе в папке `images` появился наш образ `sdcard.img` Не забудем обновить конфиг:

```
make update-defconfig
```

Заливаем на карту образ. Для начала вам нужно аккуратно выяснить под каким системным именем находится ваша SD карта. У меня это /dev/sdb поэтому я делаю так:

```
sudo dd if=sdcard.img of=/dev/sdb bs=1M of-lag=sync status=progress
```

Внимание! Карту перед этим не надо монтировать. Вставляем карту в устройство и включаем:

```
Run /sbin/init as init process
EXT4-fs (mmcblk0p2): re-mounted. Quota mode: disabled.
Saving 256 bits of non-creditable seed for next boot
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Starting network: OK

Welcome to Buildroot
buildroot login:
```

Поздравляю, вы собрали и запустили Linux на Zynq только с помощью BR. Если верить коммитам в моем репозитории это заняло 1 час 46 мин от начала до конца.

Что дальше?

Теперь на основе этой уже работающей системы можно подгонять ее дальше под свои задачи. Добавлять сборку нужных пакетов (например поставить openssh для удаленного доступа) отредактировать под себя DTS (добавить светодиоды, кнопки итд) настроить под себя ядро (добавить нужные или удалить ненужные драйвера)

“А как же PL прошивка?” - спросите вы. В рамках BR этот вопрос решается как минимум двумя способами: загрузкой через uboot или загрузкой прямо внутри Linux.

Так же остались не освещенным вопросы добавления/сборки собственных пакетов и драйверов. Это делается тоже довольно просто и удобно средствами BR.

Но это уже темы для следующей статьи.

Подключение физического устройства, размещенного на ПЛИС в симулятор QEMU при помощи Ethernet

Мангушев Александр Вячеславович

e-mail: mangushev2001@yandex.ru
Telegram: @Mangushev_Alex

Обсуждение и комментарии: link

Введение

Современные системы на кристалле (SoC) стали включать в себя все больше аппаратных ускорителей (видеокодеки, модули шифрования, блоки для работы с нейросетями), которым требуется поддержка со стороны операционной системы.

Для успешного вывода чипа на рынок необходимо как можно раньше подключить всех членов команды разработки. Однако пока чип не изготовлен это становится проблематично. Для решения возникшей проблемы можно пойти несколькими способами.

Первый – разместить чип на ПЛИС и запустить на нем Linux. В таком случае мы получаем наиболее близкую к реальности платформу, однако есть ряд недостатков. Не все блоки, необходимые для работы системы можно разместить в ПЛИС, необходимо заменять их аналогами, что требует дополнительного времени. Также большинство промышленных дизайнов не удается вместить в одну ПЛИС, для чего потребуется разделение дизайна, что также негативно сказывается на времени.

Второй подход – запустить Linux на виртуальной машине и добавить туда необходимые блоки. Такой подход существенно лучше, но также не лишен недостатков. Не все устройства можно корректно сымитировать программно, например блоки, работающие с внешней периферией. Также может потребоваться более глубокое тестирование, касающееся аппаратной реализации. В таком случае можно связать виртуальную машину с потактовым симулятором с помощью DPI [1,2]. Однако блок может требовать внешней периферии, либо его симуляция может отнимать много времени.

Для решения указанных проблем было предложено следующее. Разместить тестируемый блок (блоки) в ПЛИС, и обеспечить с ними связь из виртуальной машины.

Архитектура разрабатываемого решения

Рассмотрим более подробно схему предлагаемого решения. На хостовой

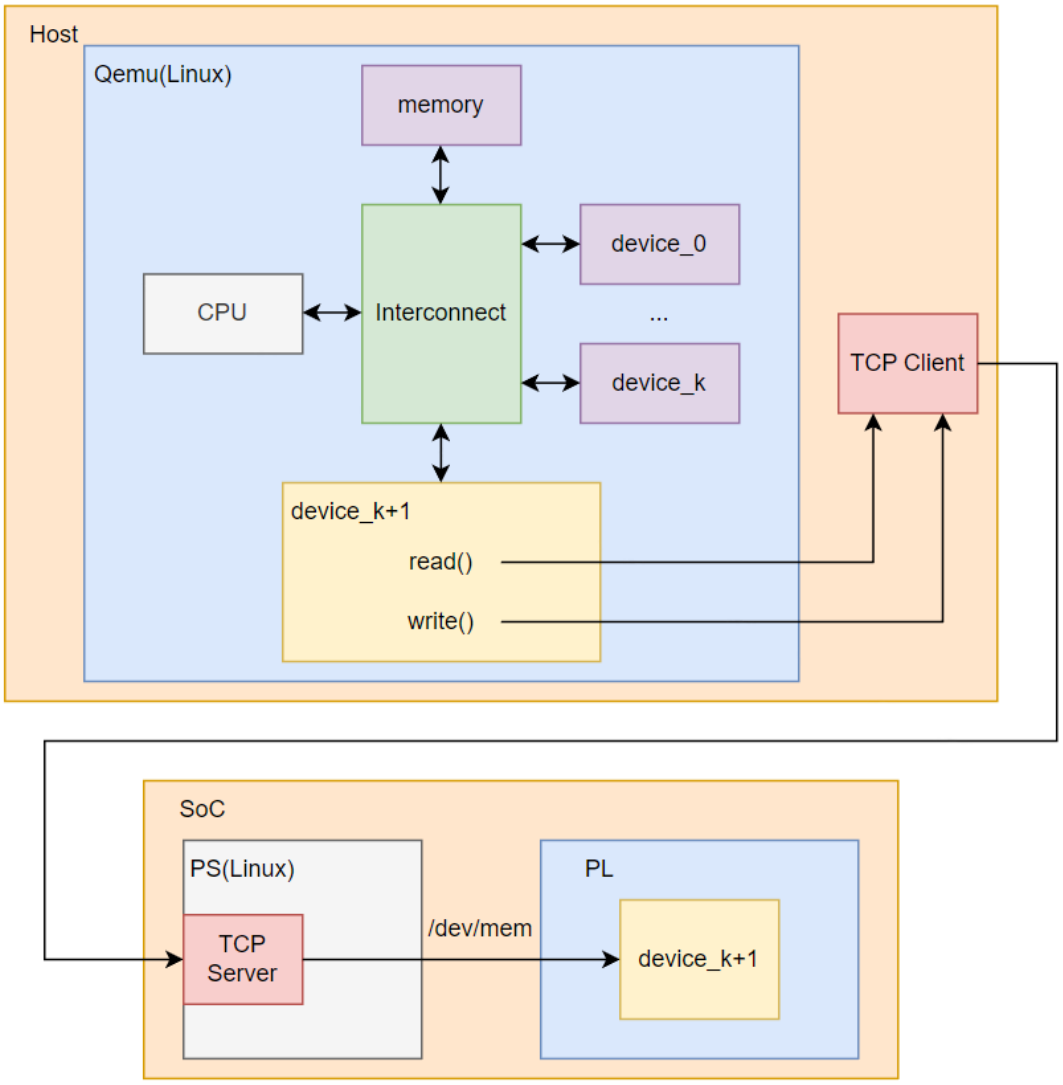


Рисунок 1 – Структурная схема решения

машине запускается QEMU, эмулирующий SoC на базе архитектуры RISCV с запущенной Linux. Добавляемое устройство не сохраняет данные и не выполняет операции, а перенаправляет запросы чтения/записи на TCP сервер. Сервер запущен на процессорной части SoC. Взаимодействие с регистрами тестового устройства реализовано с помощью /dev/mem для улучшения переносимости кода.

Эмулятор QEMU

В качестве платформы виртуализации предлагается использовать популярный open -source проект QEMU[3], который позволяет эмулировать различные процессорные архитектуры, например RISCV, ARM, x86.

QEMU собран из исходных кодов (версия v8.2.2, конфигурация --target-list=riscv64-softmmu).

Теперь можно подготовить сборку Linux с помощью Buildroot (make qemu_riscv64_virt_defconfig)

В результате получим сборку и скрипт для запуска QEMU. Остается добавить пользовательский модуль и пересобрать QEMU.

Аппаратная платформа

В качестве отладочного стенда была выбрана плата ebaz4205. На ней располагается SoC ZYNQ XC7Z010CLG400, имеющая 2 ядра Cortex A9 и ПЛИС Artix-7 на 28000 логических ячеек.

На плате располагается 256 Мбайт ОЗУ DDR3, 128Мбайт ПЗУ NAND Flash, разъем для

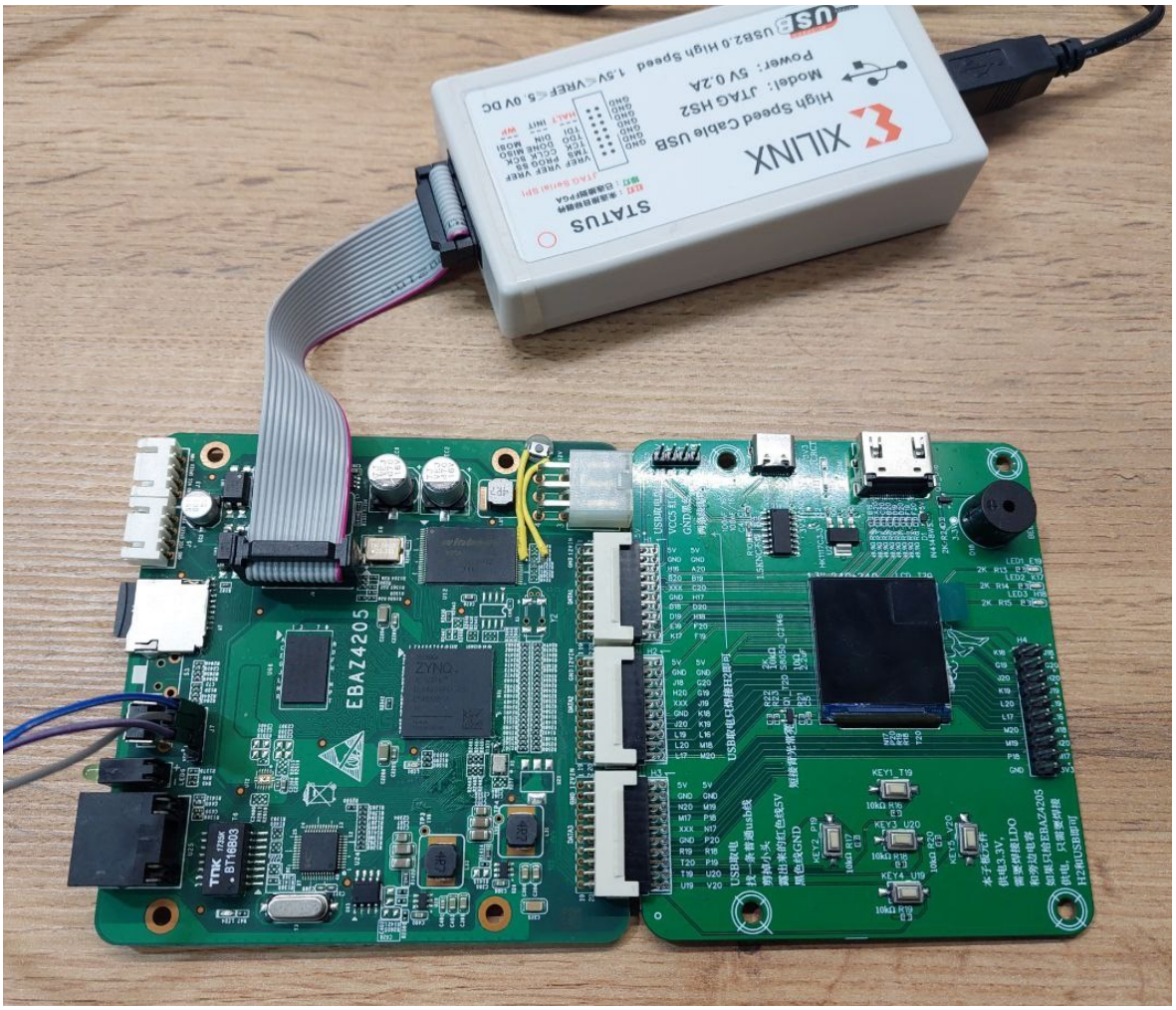


Рисунок 2 – Отладочная плата

SD карты, порт Ethernet. По умолчанию плата не совсем пригодна для комфортного использования, поэтому были выполнены следующие модификации: Подпаяна кнопка для возможности запуска системы с JTAG (по умолчанию NAND).

Интерфейс со стороны QEMU

Добавление собственного устройства в QEMU представлено в [4,5]. Имя добавляемого модуля – CUSTOM_DEV. Рассмотрим только код, реализующий основную логику (custom_dev.h в include/hw/misc/ и custom_dev.c в hw/misc).

В заголовочном файле располагается прототип функции обмена данными с ПЛИС платой и инициализации блока.

В файле custom_dev.c вначале производится объявление констант и необходимых типов. Затем объявляются функции, вызываемые при чтении/записи по адресу устройства. В них производится вызов функции обмена данными с ПЛИС. Рассмотрим эту функцию подробнее.

```
static uint64_t custom_dev_read(void *opaque,
hwaddr offset, unsigned int size)
{
  //CustomDevState *s = opaque;
  return data_exchange(offset |
                       DEV_ADDR, 0, 0);
}

static void custom_dev_write(void *opaque,
hwaddr offset, uint64_t value, unsigned int
size)
{
  //CustomDevState *s = opaque;
  data_exchange(offset | DEV_ADDR, value, 1);
}
```

Таблица 1 – Формат передачи данных

0	1	2	3	4	5	6	7	8	9
1	0	3	2	1	0	3	2	1	0
Тип		Адрес				Данные			

Аргументы – адрес регистра, данные, тип транзакции. Если производится чтение, то необходимо дождаться ответа от сервера, в противном случае можно завершать выполнение функции. Обмен данными по сети производится с помощью библиотеки socket. Формат передачи пакетов представлен в таблице 1.

Первая строка показывает нумерацию байт в передаваемом пакете. Вторая строка показывает в каком порядке байты адреса/данных/типа хранятся в передаваемом пакете. Если тип равен единице, значит пакет записи. В таблице 2 представлен формат ответа, приходящий от сервера в случае запроса чтения.

Таблица 2 – Формат ответ данных

0	1	2	3
3	2	1	0
Данные			

Для обмена данными выполняется подключение к TCP серверу и передача запроса. Для пакета чтения выполняется парсинг данных и возврат результата из функции. При пакете записи функция возвращает значение 0.

Интерфейс со стороны ПЛИС

На процессорной части Zynq в пользовательском пространстве запускается код, разворачивающий TCP сервер и ожидающий внешних подключений.

```
static uint64_t custom_dev_read(
  void *opaque, hwaddr offset,
  unsigned int size
)
```

После подключения происходит прием пакета данных и парсинг типа транзакции, адреса и данных.

```
uint32_t addr = 0, data = 0;
uint16_t wr = 0;

wr |= client_message[0] << 8;
wr |= client_message[1];

for (int i = 3, k = 2; i >= 0; i--, k++)
  addr |= (client_message[k] << (i * 8));

for (int i = 3, k = 6; i >= 0; i--, k++)
  data |= (client_message[k] << (i * 8));

printf("Msg from client: %x %x %x\n", wr,
addr, data);
```

Затем производится открытие файла /dev/mem на чтение/запись и отображение адреса в память с помощью mmap.

```
fd = open("/dev/mem", O_RDWR | O_SYNC);
map_base = mmap(
       0, map_size,
       PROT_READ | PROT_WRITE,
       MAP_SHARED,
       fd,
       addr & ~((typeof(addr))pagesize - 1)
);
```

Затем в зависимости от типа транзакции либо производим запись, либо чтение с отправкой ответа клиенту.

```
if (wr){
    *((volatile uint32_t *)virt_addr) = data;
}
else{
    read_result = *((volatile uint32_t *)
                    virt_addr);
    for (int i = 3, k = 0; i >= 0; i--, k++){
        server_message[k] = read_result >>
                            (i * 8);
    }
 if (
 send(client_sock, server_message, 4, 0) < 0)
 {
  printf("Can't send\n");
  return -1;
 }
}
```

Обращение к модулю с помощью /dev/mem позволяет сделать решение более переносимым и независящим от версии системы, на которой производится запуск.

Тестируемый аппаратный блок

Блок реализует шифрование алгоритмом AES128. Он имеет по четыре 32-х разрядных регистра для входных, выходных данных и ключа. Управление производится с помощью регистра ACSR. После того, как все данные загружены, необходимо записать в первый бит единицу. Результат готов после выставления единицы во втором бите. Блок конвейерный и выдает результат через 9 тактов. Прерывание не генерируется, для определения завершения расчета необходимо выполнять циклический опрос регистра статуса. На рисунке 3 представлен его вид в Vivado Block Design. Доступ к регистрам осуществляется с помощью интерфейса AXI4.

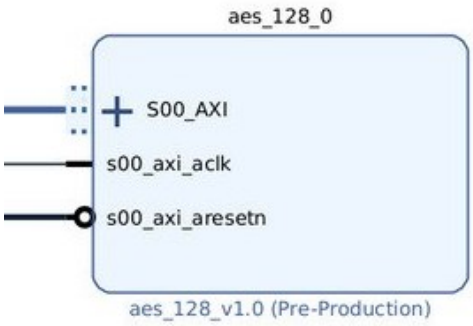


Рисунок 3 – Тестируемый аппаратный модуль вычисления шифра AES-128

Исходный код блока был взят с Github[6] и модифицирован. Исходный вариант является комбинационным, вычисление было разбито на 9 стадий. Также модуль был оформлен в виде IP блока, с доступом по интерфейсу AXI4. Подключен к процессорной системе через AXI-Interconnect порту M_AXI_GP0.

Результаты тестирования

Для тестирования работы системы было решено воспользоваться утилитой devmem, которая позволяет обращаться к физическим адресам из пользовательского пространства.

Произведем запись в регистры входных данных и ключа, а затем инициируем старт расчета.

```
#
# devmem 0x6000000 32 0x2054776F
# devmem 0x6000004 32 0x4E696E65
# devmem 0x6000008 32 0x4F6E6520
# devmem 0x600000C 32 0x54776F20
#
# devmem 0x6000010 32 0x67204675
# devmem 0x6000014 32 0x204B756E
# devmem 0x6000018 32 0x73206D79
# devmem 0x600001C 32 0x54686174
#
# devmem 0x6000030 32 0xFFFFFFFF
#
# devmem 0x6000030
0x00000002
#
# devmem 0x6000020
0x1A02D73A
# devmem 0x6000024
0x402299B3
# devmem 0x6000028
0x571420F6
# devmem 0x600002C
0x29C3505F
# uname -a
Linux buildroot 6.1.44 #11 SMP Mon Mar 25 14:30:20 MSK 2024 riscv64 GNU/Linux
```

Рисунок 4 – Взаимодействие с регистрами тестируемого модуля из Linux, запущенного в QEMU

После завершения расчета можно произвести считывание данных из регистров результата и убедиться в правильности полученных значений. Параллельно можно посмотреть вывод программы сервера, запущенной на ПЛИС и убедиться в корректности транзакций (рисунок 4).

На данный момент реализация позволяет взаимодействовать с модулями в программируемой логике только в режиме циклического опроса.

```
Listening for incoming connections.....
Client connected at IP: 172.16.7.10 and port: 54286
Msg from client: 0 46000020 0
4313b0

Listening for incoming connections.....
Client connected at IP: 172.16.7.10 and port: 49810
Msg from client: 0 46000024 0
4313b0

Listening for incoming connections.....
Client connected at IP: 172.16.7.10 and port: 49822
Msg from client: 0 46000028 0
4313b0

Listening for incoming connections.....
Client connected at IP: 172.16.7.10 and port: 46846
Msg from client: 0 4600002c 0
4313b0

Listening for incoming connections.....
^C
# uname -a
Linux buildroot 6.1.0-xilinx #1 SMP PREEMPT Wed Jan 10 14:43:08 MSK 2024 armv7l GNU/Linux
#
```

Рисунок 5 – Результат вывода программы, запущенной на Zynq

ЗАКЛЮЧЕНИЕ

В результате проделанной работы[7] была подготовлена конфигурация ПЛИС, реализующая блок аппаратного вычисления алгоритма AES-128. Для процессорной системы подготовлена сборка Linux, на которой производится запуск TCP сервера для приема запросов от симулятора. Также подготовлена сборка Linux для запуска в симуляторе QEMU.

Кроме того, в QEMU добавлен пользовательский модуль, осуществляющий взаимодействие с TCP сервером ПЛИС.

Произведено тестирование системы и выявлены недостатки и необходимые для доработки места.

СПИСОК ИСТОЧНИКОВ

1. Biagetti, G.; Falaschetti, L.; Crippa, P.; Alessandrini, M.; Turchetti, C. Open-Source HW/SW Co-Simulation Using QEMU and GHDL for VHDL-Based SoC Design. Electronics 2023, 12, 3986. https://doi.org/10.3390/electronics12183986
2. Díaz, E.; Mateos, R.; Bueno, E.J.; Nieto, R. Enabling Parallelized-QEMU for Hardware/Software Co-Simulation Virtual Platforms. Electronics 2021, 10, 759. https://doi.org/10.3390/electronics10060759
3. QEMU [Электронный ресурс] URL: https://www.qemu.org/, дата обращения 10.05.2024
4. Writing a custom device for QEMU URL: https://sebastienbourdelin.com/2021/06/16/writing-a-custom-device-for-qemu/, дата обращения 10.05.2024
5. Hello world device in QEMU URL: https://medium.com/@matanbach44/hello-world-device-in-qemu-ae69b02872f4, дата обращения 10.05.2024
6. Исходный модуль вычисления AES128 URL: https://github.com/Daniyar1239/AES-encryption-in-Verilog, дата обращения 10.05.2024
7. Описываемый проект на Github URL: https://github.com/alexmangushev/qemu_fpga_device, дата обращения 10.05.2024

Шаблон проекта испытательного стенда с использованием Yosys, Verilator, Icarus Verilog

Балакший Сергей

e-mail: sergebn@mail.ru

Телеграм: @SergeBN

Обсуждение и комментарии: link

Аннотация

Работа представляет описание создания и использования шаблона проекта для дальнейшего моделирования на основе инструментов с открытым исходным кодом (FOSS/FLOSS), таких как Yosys, Icarus Verilog, Verilator.

1 Введение

При работе с языками моделирования цифровой аппаратуры, таких как Verilog, SystemVerilog, а также с наборами инструментов YosysHQ (oss-cad-suite), iverilog, verilator возникает необходимость в написании различных тестов. Причём каждый тест имеет некоторое окружение, например, Makefile, различные скрипты, а также сам тестовый модуль. Создавать вручную для каждого теста такое окружение довольно накладно и с некоторых пор становится даже утомительно. Таким образом и возникла мотивация в написании шаблона для создания проекта нового тестового стенда. Это сродни команде “Создать новый проект” в средах различных IDE. При использовании в качестве IDE операционной системы как универсальной среды программирования, инструментом для создания нового проекта будет просто скрипт.

Далее в статье описывается процесс создания такого скрипта и его использование.

2 Создание шаблона

2.1 Шаблон для создания тестового стенда

Готовый шаблон находится в mktest_tb.sh в репозитории на GitVerse. Клонируйте его в свое рабочее окружение.

```
$ git clone https://gitverse.ru/
serge.balakshiy/mktest_tb
```

Перейдите в директорию

```
$ cd mktest_tb
```

и выполните скрипт

```
$ bash mktest.sh <test_name>
```

и переместитесь в директорию <test_name>

```
$ cd <test_name>
```

Внутри этой директории будут находиться следующие поддиректории.

```
<test_name>
   |_ bench
       |    |_ cpp
       |    |_ verilog
   |_ rtl
   |_ sim
```

Где <test_name> любое допустимое имя, определённое пользователем. Это же имя будет использовано для именования модуля, написанного на verilog. В директории rtl будут находиться файлы с описанием модуля устройства. В директории sim будет находиться файл, содержащий описание стенда. И в этой директории строится симуляция. В директории bench/cpp будут файлы на C++. И в bench/formal будут файлы, описывающие формальную проверку.

Для определения правил сборки будем использовать Makefile. Кратко опишем цели Makefile.

Цели **vtb** и **itb** строят стенд соответственно с использованием verilator и iverilog. Эти цели собираются без использования C++ и функции main. Модули и стенд, в этом случае, могут быть написаны на Verilog и/или SystemVerilog.

Цель **dot:** показать как использовать yosys для получения файлов с графическим отображением тестируемого дизайна.

Цель **exe:** собрать стенд с использованием main.cpp, хотя ничего и не делает.

Цель **exe_b** собирает стенд с детальным управлением тестируемым модулем под управлением только C++.

Цель **exe_t** аналогична предыдущей цели, но собрана с использованием дополнительного шаблонного класса TESTB, чтобы разгрузить функцию main.

2.2 Скрипт mktest_tb.sh

При запуске скрипта первое, что он потребует - это имя для нового тестового стенда. Эта способность скрипта реализована при помощи следующих строк:

```
1   if [ -n ”$1” ]
2     then
3     TESTNAME=”S1”
4   else
5     echo -n ”Введите имя папки:”
6     read TESTNAME
7   fi
8
9 AUTHOR=”Сергей Балакший, t.me/sergeBN,
@SergeBN”
```

В строке 1 проверяется наличие параметра. Если он есть, то его значение присваивается переменной TESTNAME (3). Если параметр не указан, то происходит запрос (5) и чтение введённого значения в переменную TESTNAME (6).

Далее, в строке 9 следует переменная AUTHOR. Значение этой переменной будет использовано при формировании файла README. Заполните её по своему усмотрению.

2.3 Цели itb и vtb

При задании целей сборки важно понимать как работает iverilog и как работает verilator.

В данном случае verilator делает две основных вещи.

1. Создает из исходных файлов Verilog и/или SystemVerilog файлы C++.
2. Вызывает gcc для компиляции и сборки проекта в исполняемую программу.

Второе действие verilator делает “за кадром” постольку-поскольку ему задан параметр –bynary. Этот параметр заставляет verilator строить исполняемый файл. В том числе verilator сгенерирует файл V<testname>_tb__main.cpp. Этот файл будет содержать соответствующую параметрам функцию main, необходимую для построения исполняемой симуляции.Параметр – trace

включит context -> traceEverOn(true) в main.

Таким образом становится возможным генерирование дампа данных для gtkwave.

Параметр –timing включает поддержку тактируемого времени в исполняемой модели. Это достигается включением в main строки с функцией context -> time(topp->nextTimeSlot()), которая работает в основном цикле и обеспечивает симуляцию тактами времени.

Icarus Verilog работает иначе. Также как и verilator, он использует исходные файлы Verilog с некоторыми ограничениями. Читает и интерпретирует исходные файлы. Затем генерирует скомпилированный результат в выходной файл с форматом по умолчанию vvp. Полученный файл не может быть выполнен сам по себе. Для его запуска и выполнения симуляции необходима программа vvp.

Дальнейшее действие одинаково в обоих случаях. Мы используем программу gtkwave для просмотра полученного дампа данных в виде эпюр сигналов.

Для достижения этих целей потребуется iverilog и verilator. Итак, обе обозначенные выше цели используют одни и те же исходные файлы дизайна:

- тестируемый модуль rtl/<testname>.sv;
- тестовый стенд sim/<testname>_tb.sv.

Ниже показана часть Makefile, в которой определены эти цели.

```
1 .PHONY: vtb itb
2
3   TESTNAME = :TESTNAME:
4   MTOP = $(TESTNAME)_tb
5   SRC = ../rtl/$(TESTNAME).sv $(TESTNAME)
_tb.sv
6
7   VFLAGS = --binary -Wall --trace --timing
8   VTOP = -top $(MTOP)
9
10 vtb: vsim vvcd wave
11
12  vsim: $(SRC)
13  verilator $(VFLAGS) $(VTOP) $^
14
15  vvcd:obj_dir/V$(TESTNAME)_tb
16  $^
17
18 itb: isim ivcd wave
19
20  isim: $(SRC)
21  iverilog $^
22
23  ivcd: a.out
24  vvp $^
25
26  wave: out.vcd
```

Здесь строка 1 определяет доступные цели. Переменная TESTNAME (3) содержит имя тестового модуля, определённого пользователем на этапе формирования проекта. MTOP (4) содержит имя модуля top. Переменная SRC (5) содержит список файлов проекта. Переменная VFLAGS (7) содержит флаги для verilator. VTOP (8) содержит строку с параметром -top.

В строке 10 уточняется цель vtb. При построении цели vsim (12, 13) вызывается verilator с соответствующими параметрами и строится директория obj_dir, в которой находятся сгенерированные файлы на C++. После этого цель vvcd (15, 16) вызывает исполняемый файл ./<testname> для того, чтобы выполнить симуляцию и сгенерировать данные для gtkwave. Эти данные направляются в файл out.vcd, после чего вызывается gtkwave (26, 27) для отображения эпюр сигналов, построенных при выполнении симуляции. Само имя файла “out.vcd” должно быть определено в файле testbench (<testname>_tb.sv) фунцией $dumpfile(“out.vcd”). Там же должна быть определена функция $dumpvars(), чтобы переменные и их значения записывались в выходной файл.

Цель itb достигается схожим образом, за исключением того, что на первом этапе вызывается iverilog (20, 21), который создаёт из исходных файлов набор данных в файле

по умолчанию a.out. Этот набор данных сам по себе не может быть выполнен ни на одной платформе. Поэтому на следующем шаге вызывается программа vvp (23, 24), которая выполняет скомпилированную a.out форму, созданную Icarus Verilog. Кроме этого, vvp порождает набор данных для программы gtkwave. Этот набор данных может быть порожден в нескольких форматах. Здесь используется по умолчанию формат волнового дампа -vcd. Файлы дампа VCD большие и громоздкие, но они также максимально совместимы со сторонними инструментами, которые читают дампы сигналов.

Кроме этого можно использовать формат -fst. Это современный формат дампа, который быстрее и компактнее других форматов дампа. Он поддерживает инкрементальный дамп. Аргумент -fst-speed использует более быстрые методы сжатия, которые создают выходной файл заметно большего размера. Аргумент -fst-space выполняет переупаковку файла при закрытии, для создания файла дампа минимально возможного размера. Аргументы -fst-space-speed или -fst-speed-space используют более быстрый метод сжатия и также переупаковывают файл при закрытии.

Теперь, находясь в корневой директории вновь созданного тестового стенда, выполните команду

```
$ make itb
```

В результате получим запуск программы gtkwave, которая будет отображать эпюры сигналов как на рисунке 1.

Аналогично, можно построить цель vtb. После запуска

```
$ make vtb
```

Вы увидите тот же результат, что и на рисунке 1.

2.4 Цель exe

Следующим стремлением будет создание тестового стенда под управлением verilator, но который управляется отдельным файлом sim_main.cpp. В этой части мы будем использовать C++ для полного управления стендом.

```
0   // file sim_main.cpp
1   #include ”Vt3.h”
2   #include ”verilated.h”
3   int main(int argc, char** argv) {
4       VerilatedContext* contextp = new
VerilatedContext;
5       contextp->commandArgs(argc, argv);
6       Vt3* top = new Vt3{contextp};
7
8       while (!contextp->gotFinish())
{ top->eval(); }
9       delete top;
10      delete contextp;
11      return 0;
12  }
```

Предполагается, что стенд назван t3. Тогда в строке 1 происходит включение сгенерированного файла Vt3.h. Далее подключается общий файл verilated.h (2). В

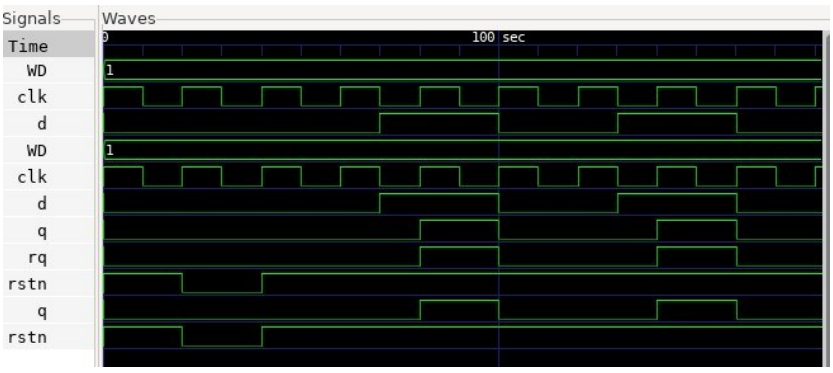


Рис. 1: Эпюры напряжений сигналов

строке 4 создается указатель conetxtp, который инициализируется объектом VerilatedContext и будет нужен для управления стендом. В строке 6 создается объект Vt3 и инициализируется указатель top на этот объект. Строка 8 задает главный цикл программы. Этот цикл будет работать до нажатия клавиш Ctrl+C или Ctrl+Z. Все, что делает этот цикл, он вызывает функцию eval(), которая вычисляет состояние сигналов на линиях стенда. В случае окончания работы стенда происходит удаление созданных объектов и выход из программы с кодом 0.

Собрать эту цель можно командой

```
$ make exe
```

Исполняемый файл Vt3 будет находиться в директории obj_dir. Запустим его в работу, находясь в директории sim.

```
$ obj_dir/Vt3
^Z
[2]+ Stopped        obj_dir/Vt3
```

Программа завершает свою работу при нажатии Ctrl+Z. Эта программа ничего не делает, так как в main и не сказано что-либо делать.

Рассмотрим, как строится данная цель.

```
1 .PHONY: exe
2
3 exe: del_dir post_exe 4
5   del_dir:
6   rm -f -r obj_dir
7
8   CSRC=../rtl/$(TESTNAME).sv ../bench/cpp/
sim_main.cpp
9   post_exe: obj_dir/V$(TESTNAME)
10  obj_dir/V$(TESTNAME): $(CSRC)
11  verilator --cc --exe --build -j 0 -Wall
$^
```

Вначале определяется цель exe. В строке 3 она уточняется. Цель del_dir удаляет созданную ранее временную рабочую директорию. Цель post_exe будет создавать запускаемую программу. Для этого в строке 8 указываются исходные файлы. В строке 9 показано, что post_exe зависит от конечного файла V<test_name>, который и строится в строках 10, 11 при помощи verilator. Где: –cc параметр, который заставляет verilator генерировать файлы C++; –exe сообщает verilator-у, что надо использовать файл оболочку sim_main.cpp и строить исполняемый файл; –build позволит verilator-у строить самостоятельно, без дополнительного вызова gcc “вручную”; -j 0 указывает использовать столько потоков, сколько есть в компьютере; -Wall включает более сильные предупреждения; $^в конце - список исходных файлов, которые нужны для сборки основной цели.

Как уже отмечалось выше, эта программа ничего не делает. И представляет интерес то, как построить программу, используя определенный пользователем файл sim_main.cpp, который содержит функцию main. И с этой задачей мы успешно справились. Переходим к рассмотрению следующего вопроса. Как заставить программу делать полезную и информативную симуляцию.

2.5 Цель exe_b

Для построения этой цели напишем новый файл sim1_main.cpp. Если на данном этапе снова выполнить цель vtb, то в директории obj_dir окажется сгенерированный verilator-ом файл V<testname>_tb__main.cpp. Об этом файле упоминалось в разделе “Цели itb и vtb”. Файл содержит интересную информацию о том, как включить трассировку и ход времени. Сейчас мы это рассмотрим подробнее.

```
1   // Verilated -*- C++ -*-
2   // DESCRIPTION: main() calling loop,
3   // created with Verilator --main
4
5   #include ”verilated.h”
6   #include ”Vt3_tb.h”
7
8 //======================
9
```

```
10  int main(int argc, char** argv, char**) {
11  // Setup context, defaults, and parse
command line
12  Verilated::debug(0);
13  const std::unique_ptr<VerilatedContext>
contextp
14  {new VerilatedContext};
15  contextp->traceEverOn(true);
16  contextp->commandArgs(argc, argv); 17
18  // Construct the Verilated model, from
Vtop.h generated
19  // from Verilating
20  const std::unique_ptr<Vt3_tb> topp
21  {new Vt3_tb{contextp.get()}};
22
23  // Simulate until $finish
24  while (!contextp->gotFinish()) {
25  // Evaluate model
26  topp->eval();
27  // Advance time
28  if (!topp->eventsPending()) break;
29  contextp->time(topp->nextTimeSlot());
30  }
31
32  if (!contextp->gotFinish()) {
33  VL_DEBUG_IF(VL_PRINTF
34  (”+ Exiting without $finish; no events
left\n”););
35  }
36
37  // Execute ’final’ processes
38  topp->final();
39
40  // Print statistical summary report
41  contextp->statsPrintSummary();
42
43  return 0;
```

Здесь видно, как verilator включает трассировку (строка 15) и ход времени (строка 29).

Включим в новый файл трассировку, просто добавив строку contextp->traceEverOn (true) (строка 13). Также будем использовать contextp->time(), но управлять ходом времени будем иначе.

```
1   // file sim1_main.cpp
2   #include ”Vt3.h”
3   #include ”verilated.h”
4   #include ”verilated_vcd_c.h”
5
6   int main(int argc, char** argv) {
7   Verilated::mkdir(”logs”);
8
9   const std::unique_ptr<VerilatedContext>
contextp{newVerilatedContext};
10  contextp->debug(0);
11  contextp->commandArgs(argc, argv);
12  contextp->randReset(2);
```

```
13  contextp->traceEverOn(true);
14
15  VerilatedVcdC* tfp = new VerilatedVcdC;
16  if(tfp==0){printf(”Error tfp\n”); return
1;}
17  const std::unique_ptr<Vt3> top{new Vt3
{contextp.get(), ”TOP”}};
18
19  top->trace(tfp, 99);    // Trace 99 lev-
els of hierarchy
20  // (or see below)
21  tfp->open(”out.vcd”);
22
23  top->rstn = !0;
24  top->clk = 0;
25  uint16_t cnt=0;
26  while (!contextp->gotFinish()) {
27  contextp->timeInc(1); // 1 timeprecision
period passes...
28  top->clk = !top->clk;
29  if (!top->clk) {
30  if (contextp->time() > 1 && contextp-
>time() < 10) {
31  top->rstn = !1; // Assert reset
32  } else {
33  top->rstn = !0; // Deassert reset
34  }
35  if (cnt < 32){
36  cnt++;
37  }else{
38  cnt = 0;
39  top->d = !top->d;
40  }
41  }
42  top->eval();
43  tfp->dump(contextp->time());
44  }
45  top->final();
46  tfp->close();
47  return 0;
48  }
```

Поскольку в исходном модуле присутствует сигнал rstn, присвоим ему начальное значение (строка 23). Также в исходном модуле есть линия clk, на которой необходимо начать ход времени. Поэтому установим clk в 0 (строка 24). А в строке 28, которая находится внутри основного цикла, будем на каждом такте цикла менять значение clk на противоположное. Эта строка (28) и будет задавать ход времени в тестируемом модуле.

В строке 15 создается объект VerilatedVcdC и указатель на него, чтобы было куда выгружать данные о работе стенда. Если все хорошо (16), то создаем модуль top с

указателем. И включаем трассировку (19), связывая модуль top с объектом ftp. После чего открываем файл “out.vcd” (21), в который и будем выгружать данные метрик.

В строке 25 определяем счетчик, при помощи которого будем управлять подачей данных на вход модуля.

Далее начинается главный цикл тестового стенда (26). Этот цикл прервется по сочетанию клавиш Ctrl+Z либо Ctrl+C. Далее (27, 28) задаем длительность периода и генерим ход времени на линии clk.

Определяем момент изменений сигналов на положительном фронте clk (29) и отсчитываем 100 периодов time() для подачи сигнала rstn на модуль (30, 31). По истечении этого времени сбрасываем rstn в !0, чтобы дать тестируемому модулю возможность работать (33).

В строках 35...40 происходит отсчет счетчиком значения 32, после чего счетчик сбрасывается в 0, а сигнал на линии top->d меняет значение на противоположное.

В строке 42 пересчитывается метрика стенда. И в строке 43 данные time() выгружаются в дамп. Цикл повторяется.

Если происходит одно из событий Ctrl+Z или Ctrl+C, то цикл завершается. Память освобождается. Программа завершается с кодом 0.

Компиляция и сборка стенда происходит при указании цели exe_b.

```
1   .PHONY: exe_b
2   exe_b: rm_obj post_exe_b build_exe
run_exe_b
3
4   rm_obj:
5   rm -f -r obj_dir
6
7   post_exe_b: $(CPPSRC)
8   verilator --cc --exe --trace --timing -
Wall $^
9
10  build_exe:
11  make -j 0 -C obj_dir -f V$(TESTNAME).mk
V$(TESTNAME)
12
13  run_exe_b:
14  echo ”\n$(TESTNAME) is runnig. To inter-
rupt do Ctl+C\n” 15  obj_dir/V$(TESTNAME)
```

Здесь цель exe_b состоит из 4-х подцелей (2).

1. Удаляются результаты предыдущей работы (4, 5).
2. Запускается в работу verilator (7, 8), которому даны следующие указания:
   - –cc генерировать выходные данные на C++;
   - –exe создать вместе с файлом-оболочкой sim1_main.cpp исполняемый файл, а не только библиотеку;
   - –trace включить трассировку;
   - –timing включить часы;
   - -Wall расширить сообщения компилятора;
   - $^- исходные файлы проекта.
3. Запускаем сгенерированный verilator-ом V<testname>.mk в директории obj_dir, отвечающий за сборку *.cpp файлов (10, 11).
4. Запускаем симуляцию (13...15). Не забываем остановить программу Ctrl+Z или Ctrl+C.

Просмотреть полученный результат можно программой gtkwave. При просмотре должны увидеть картинку, как на рисунке 2.

На рисунке хорошо видно поведение сигнала rstn. Также видно, что данные на выходе модуля (сигнад q) меняются на положительном фронте сигнала clk. Тогда как входные данные на линии d меняются раньше. Тоже происходит и при смене d на противоположное значение. При этом длительность периода q составляет 32 такта time() или 16 тактов времени clk, как это и было определено в главном цикле функции main.

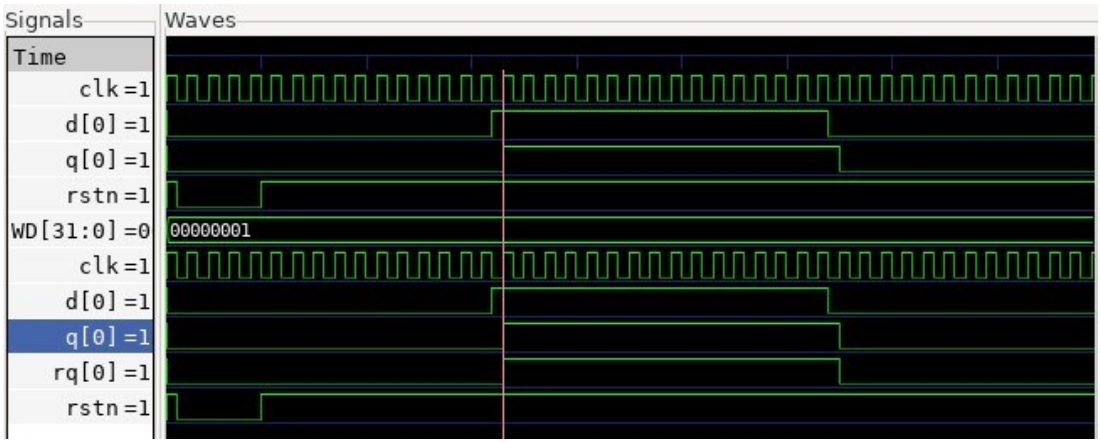


Рис. 2: Эпюры напряжений сигналов

2.6 Цель exe_t

Стремление к унификации стенда приводит к еще одной идее. Можно использовать шаблонный класс и перенести в него некоторые детали реализации. Для реализации этой идеи я воспользовался проектом Wbi2c в репозитории ZipCpu. Оттуда я взял файл testb.h, описывающий шаблонный класс TESTB. В нем я изменил лишь виртуальный метод tick(void). Теперь этот метод содержит логику управления тестируемым модулем <testname>, см. следующий листинг.

```
1   #ifndef TESTB_H
2   #define TESTB_H
3
4   #include <stdio.h>
5   #include <stdint.h>
6   #include ”verilated.h”
7   #include <verilated_vcd_c.h>
8
9   #define TBASSERT(TB,A) do { if (!(A))
{ (TB).closetrace(); } \
10  assert(A); } while(0);
11
12 template <class VA> class TESTB { 13 pub-
lic:
14  VA  *m_core; 15 VerilatedVcdC* m_trace;
16  uint64_t    m_tickcount;
17
18  TESTB(void) : m_trace(NULL), m_tickcount
(0l) {
19  m_core = new VA; 20 m_core->clk = 0;
21  m_core->rstn = !0;
22  eval(); // Get our initial values set
properly.
23  }
24
25  virtual ~TESTB(void) {
26  closetrace();
27  delete m_core; 28   m_core = NULL;
29  }
30
31  virtual void opentrace(const char
*vcdname) {
32  if (!m_trace) {
33  m_trace = new VerilatedVcdC;
34  m_core->trace(m_trace, 99);
35  m_trace->open(vcdname);
36  }
37  }
38
39  virtual void closetrace(void) {
40  if (m_trace) {
41  m_trace->close();
42  delete m_trace; 43  m_trace = NULL;
44  }
45  }
46  virtual void eval(void) {
47  m_core->eval();
48  }
49
50  virtual void tick(void) {
51  m_tickcount++;
52
53 // Убедитесь, что мы получили наши оценки
непосредственно перед 54 // началом работы,
так как в некоторых модулях могли произойти
55  // изменения, от которых зависит работа
модуля. Такая оценка 56 // заставляет эту
логику пересчитываться до начала тактового
57 // сигнала.
58  eval();
59  // Предполагается, что в тестируемом
модуле есть сигнал clk,
60  // и, что линия с сигналом d является
информационным входом 61  // в модуль.
62  // Кроме того, tick мы будем
масштабировать на 10
63  if (m_trace) m_trace->dump((vluint64_t)
(10*m_tickcount-2));
64  m_core->clk = 1; // Управляем линией clk
65  eval();
```

```
66  if (m_trace) m_trace->dump((vluint64_t)
(10*m_tickcount));
67  m_core->clk = 0; // Управляем линией clk
68  eval();
69  if (m_trace) {
70  m_trace->dump((vluint64_t)
(10*m_tickcount+5));
71  m_trace->flush();
72  }
73  if((10*m_tickcount)>10 &&
(10*m_tickcount)< 100){
74  m_core->rstn = !1; // Assert reset
75  }else {
76  m_core->rstn = !0; // Deassert reset
77  }
78  // Управляем входом d
79  if( (10*m_tickcount)%13==0){m_core->d = !
m_core->d;}
80  }
81
82  unsigned long   tickcount(void) {
83  return m_tickcount;
84  }
85  };
86
87 #endif
```

Предполагается, что в интерфейсе модуля, который будет использовать данный шаблонный класс, есть линия clk для подачи хода времени, линия rstn для сигнала “сброс”. Также предполагается, что информационным входом в модуль является линия *d*. Если в тестируемом модуле другие имена этих линий, то это надо согласовать.

Рассмотрим этот шаблон: переменная m_core указывает на моделируемый модуль; m_trace указывает на объект трассировки; m_tickcount - внутреннее время шаблона. Эти переменные инициируются в конструкторе при создании. И сразу же после инициализации переменных вызывается метод eval(), чтобы учесть начальные значения.

Метод opentrace(const char *vcdname) (31...37) принимает имя файла для выходных данных трассировки. Создает объект трассировки и открывает указанный файл.

Метод closetrace(void) (39...44) делает ровно противоположное.

Метод eval(void) (46...48) - это просто обертка, в которой происходит вызов основной функции eval().

Метод tick(void) (50...80) - это метод, в котором размещается логика управления стендом. Рассмотрим подробнее этот метод. Каждый раз при вызове этого метода происходит увеличение счетчика m_tickcout, чтобы отслеживать текущее время. Далее следует логика управления сигналами на линиях интерфейса тестируемого модуля (63...80). Эта логика аналогична уже вышеописанной логике в файле sim1_main.cpp.

Теперь можно представить файл simt_main.cpp, в котором используется данный шаблонный класс.

```
1   #include ”Vt3.h”
2   #include ”verilated.h”
3   #include ”testb.h”
4
5   int main(int argc, char** argv) {
6   VerilatedContext *contextp = new Veri-
latedContext;
7   contextp->commandArgs(argc, argv);
8   contextp->debug(0);
9   contextp->randReset(2);
10  contextp->traceEverOn(true); 11
12  TESTB<Vt3> *tb = new TESTB<Vt3>;
13
14  tb->opentrace(”out.vcd”);
15
16  while (!contextp->gotFinish()) { tb-
>tick(); }
17
18  tb->closetrace();
19  delete tb;
20  delete contextp;
21
22  return 0;
23  }
```

Функция main заметно сократилась. Отметим изменения. В строке 12 создается экземпляр тестируемого модуля и инициализируется указатель tb на этот объект. В строке 14 создается выходной файл. Строка 16 теперь содержит весь главный цикл, который состоит в том, чтобы вызывать метод tick. Этот цикл остановит Ctrl+Z или Ctrl+C. После остановки цикла происходит

уборка за собой и выход с кодом 0.

Посмотреть эпюры можно, как и в других случаях, при помощи gtkwave. Эпюры должны выглядеть как на рисунке 2.

2.7 Цель dot

Эта цель настроена на получение файлов *.dot, представляющих в графическом виде тестовый модуль. Эта цель строится применением yosys и скрипта, который находится в директории sim/ с именем <имя_проета>.ys. Допустим, что имя проекта t4, тогда скрипт будет выглядеть так:

```
1 echo on
2
3   hierarchy -top t3
4   select -module t3
5   select -list
6   select t:*
7   select -list
8   select -set new_cells %
9   select -clear
10  show -format dot -prefix t3_show t3
11  show -format dot -prefix new_cells_show -
notitle @new_cells
12  show -color maroon3 @new_cells -color
cornflowerblue p:* \
13  -notitle -format dot -prefix t3_hier
14
15 #
============================================
===========
16
17  proc -noopt
18  select -set new_cells t: t:*dff
19  show -color maroon3 @new_cells -notitle -
format dot -prefix \
20  t3_proc
21
22 #
============================================
===========
23
24  opt_expr; clean
25  select -set new_cells t:
26  show -color cornflowerblue @new_cells -
notitle -format dot \
27  -prefix t3_clean
```

В этом скрипте записаны правила для получения файлов *.dot:

- команда select используется для изменения и просмотра списка выбранных объектов: когда вызывается select -module <module_name> (строка 4) - это меняет текущую выборку из всего дизайна только на указанный <module_name> модуль и указывает на то, что любые команды, которые мы запускаем на данном этапе, будут работать только с этим модулем;
- вызов select -list (строка 5) даёт список всех объектов в модуле <module_name>, включая сам модуль, а также все провода, входы, выходы, процессы и ячейки;
- select t:* (строка 6). Здесь t: означает, что будет происходить отбор с совпадением по типу ячейки, и * означает сопоставление с чем угодно. Это простое выделение находит все ячейки, независимо от их типа. Активное выделение теперь отображается как [<module_name>]*, указывающее на некоторый подвыбор из модуля <module_name>. Это дает нам ячейки, которые мы хотим выделить для модуля <module_name> после изображения иерархии.

Мы можем присвоить выделенному элементу имя с помощью select -set (8). В нашем случае используется название new_cells, и мы указываем ему использовать текущий выбор, обозначенный символом %. Затем мы можем использовать это именованное выделение, обозначив его как @new_cells, которое мы увидим позже (11, 12). Затем мы очищаем выделение, чтобы следующие команды могли работать с полным дизайном (9). Хотя здесь это разделено, мы могли бы сделать то же самое в одной строке, вызвав select -set new_cells <module_name>/t:*. Если мы знаем, что в нашем проекте есть только один модуль, мы можем пропустить часть <module_name>. Просматривая дальше код, мы можем увидеть это с помощью select -set new_cells t:$mux t:*dff (18). Мы также можем видеть в

этой команде, что выбор необязательно должен ограничиваться одним оператором.

Многие команды также поддерживают необязательный [selection] аргумент, который может быть использован для переопределения выбранных в данный момент объектов. Мы могли бы, например, вызвать clean <module_name> с тем, чтобы clean имел возможность оперировать только с <module_name>.

Команда show создаёт (10, 11, 12, 19, 26) файлы dot с представленным дизайном в соответствии с выбором select.

Подробнее о скриптах yosys см. YosysHQ. More scripting. Сборкой занимается Makefile

```
.PHONY: dot
dot:$(TESTNAME).ys
    yosys -s $^ ../rtl/$(TESTNAME).sv
```

Здесь yosys вызывается с опцией -s, которой в качестве параметра передается имя файла скрипта. Также передается имя файла, в котором содержится рассматриваемый модуль.

В корневой директории созданного проекта выполните:

```
$ make dot
$ cd sim
```

В директории sim будут созданы файлы с расширением dot. Файлы можно просмотреть в приложении DotViewer или в каком-нибудь другом подобном. Если вы ранее уже перешли в директорию sim, прямо в ней можно выполнить

```
$ make dot
```

И вы получите тот же результат.

В дополнение к сказанному, если у вас установлен пакет symbolator, то вы можете построить графическое изображение тестируемого модуля, применив команду:

```
$ symbolator -f png --title <module_name>.v
```

Где: -f png - графический формат выходного файла; –title - параметр, указываващий построить имя модуля над графическим изображением модуля; <module_name>.v файл модуля. Для нашего модуля получается вот такой рисунок 3.

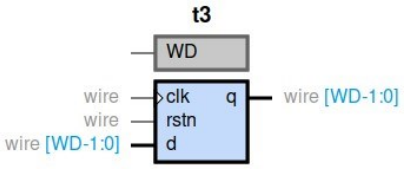


Рис. 3: Тестируемый модуль

3 Заключение

Выполняя эту работу, я научился создавать Makefile для сборки тестовых стендов. Варианты сборок:

При использовании IcarusVerilog необходимо написать стенд и модули на Verilog. Здесь Icarus Verilog - это программа iverilog, которая преобразовывает текст модели, написанный на языке Verilog, в выходной формат для дальнейшей обработки. Verilog - это язык описания модулей электронных цифровых схем, как описано в стандарте IEEE-1364. Чтобы получить представление о текущем состоянии Icarus Verilog, смотрите его домашнюю страницу здесь.

1. При использовании verilator, можно писать модули и стенд на SystemVerilog. Где verilator - это программа, которая вызывается с параметрами, аналогичными GCC, и преобразовывающая исходные тексты модулей и стенда, написанные на SystemVerilog (Verilog), в файлы на C++/SystemC. Затем эти файлы компилируются компилятором C++ (gcc/clang) в исполняемый файл. Вызов результирующего исполняемого файла выполняет моделирование проектирования.
2. SystemVerilog - это язык описания

аппаратуры, являющийся расширением языка описания оборудования IEEE 1364-2001 Verilog, помогающий создавать и проверять модели абстрактного архитектурного уровня.

- При работе с verilator можно не создавать функцию main. Она будет создана автоматически самим verilator-ом в соответствии с указанными параметрами.
- Если вы создаете свою функцию main, то вы можете более детально управлять своим стендом.

Конечно, есть еще множество возможностей для написания тестовых стендов, которые здесь не рассматривались. Но чтобы их использовать, и развивать дальше свои проекты, мне пришлось преодолеть эту первую ступеньку.

Буду рад, если данная статья поможет еще кому-либо. Также буду рад конструктивным комментариям и предложениям.

Verilator как linter в Neovim

Кудинов Максим
Telegram: @m_kudinov

Обсуждение и комментарии: link

Введение

Verilator обычно используют для симуляции Verilog/SystemVerilog кода, но помимо этого существует опция **–lint-only**, которая только указывает на синтаксические ошибки, а при добавлении **-Wall** и некоторые стилистические.

Я предпочитаю вести всю разработку из терминала, редактируя код в Neovim и запуская различные инструменты через скрипты. Преимущество Neovim перед классическим Vim заключается в том, что конфигурацию можно писать на языке Lua, и поддержка Language Server Protocol уже встроена в редактор.

Language Server Protocol (LSP) - протокол между редактором и сторонним языковым сервером, который добавляет такие возможности, как автодополнение текста, переход к объявлению переменных, переименование.

В качестве LSP для Verilog/SystemVerilog я использую Verible, он предоставляет не только возможности языкового сервера, но еще форматирование и lint.

Но его lint не распознает многие моменты, например неиспользуемые сигналы или защелки в комбинационной логике. Verilator указывает на эти моменты, и еще на многое другое, так что его lint мы и будем ставить в дополнение к Verible.

Установка Verilator

Если у вас еще не установлен Verilator, то сначала стоит поставить его. Для поддержки опции **–timing** нужна версия 5.002 и выше. Например, начиная с Ubuntu 23.04 подходящую версию можно установить из репозитория через **sudo apt install**. Если в вашем дистрибутиве пакет версии ниже, то можно установить актуальный релиз из OSS CAD SUITE.

Установка nvim-lint

Для интеграции Verilator в Neovim нужно воспользоваться плагином, который предоставляет подключение сторонних lint программ. По опыту мне больше всего понравился nvim-lint.

Установите плагин через ваш plugin manager, я еще не успел перейти на lazy, так что буду использовать packer.

```
use("mfussenegger/nvim-lint")
```

После чего надо указать типы файлов, к которым надо подключать Verilator.

```
require('lint').linters_by_ft = {
    systemverilog = {'verilator'},
    verilog = {'verilator'}
}
```

Теперь создаем автокоманду, которая будет запускать линт при сохранении файла.

```
vim.api.nvim_create_autocmd(
{ 'BufWritePost' }, {
    callback = function()
        require('lint').try_lint()
    end,
})
```

Последнее, что осталось сделать для начальной настройки - выключить lint от Verible, чтобы результаты диагностики не смешивались.

Для настройки LSP я использую nvim-lspconfig с функцией on_attach, в которой можно выключить возможности для конкретного сервера.

```
local on_attach = function(client, bufnr)
    -- other settings
    -- Disable verible diagnostics
    if client.name == "verible" then
        vim.lsp.handlers
        ["textDocument/publishDiagnostics"]
        = function() end
    end
end
```

Этих настроек хватит для простых одиночных модулей. Например, попробуем добавить лишний сигнал в FIFO.

Warning: signal is not used

Дополнительная настройка

У текущей конфигурации есть большая проблема: Verilator на анализ получает только текущий файл, игнорируя весь остальной проект. Для примера попробуем открыть проект из нескольких файлов, где еще package находятся в отдельной директории include.

Verilator не может найти файлы

Чтобы Verilator смог найти все необходимые файлы, ему нужно передать дополнительную конфигурацию. Напишем файл verilator.f в корне проекта.

```
// Включить стилистическую диагностику
-Wall

// Учитывать задержки
--timing

// Использовать FST формат для waveform
--trace-fst

// Показывать имена полей структур
--trace-structs

// Заменять X на случайное значение
--x-assign unique

// То же самое, но при старте
--x-initial unique

// Указать директорию с модулями
-Irtl

// Указать директорию с package
-Irtl/include

// Указать top модуль проекта
rtl/board_top.sv
```

Дополнительные аргументы для Verilator можно посмотреть в документации.

Как можно заметить, тут нет опции **–lint-only**, но есть опции, которые не играют роли для диагностики. Сделано это по причине, что это общий файл, который использует как редактор, так и симулятор для запуска тестбенчей. Таким образом не приходится указывать одно и то же дважды.

К сожалению, пока что Verilator не может автоматически распознать конфигурационный файл в директории, так что этот файл ему нужно передать в качестве аргумента. Для этого добавим в конфигурацию Neovim следующее.

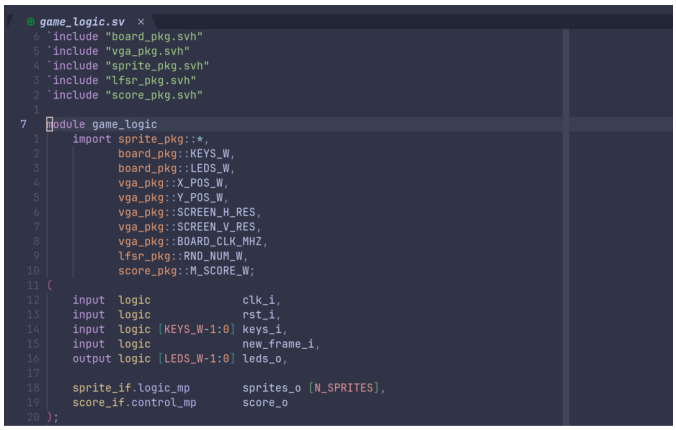
```
local verilator
verilator = require
('lint').linters.verilator

verilator.args = {
    '--lint-only',
    '-F',
    vim.fs.find("verilator.f", {
        upward = true,
        stop = "/home",
        type = "file"
    })[1]
}
```

Опция **-F** означает, что пути в файле verilator.f относительные, а не абсолютные. Для абсолютных замените на **-f**. Далее, с помощью **vim.fs.find()** происходит поиск конфигурационного файла с текущей директории и выше до /home (не включительно). **vim.fs.find** выдает лист, так что с помощью [1] берем первый элемент (да, в Lua индексация с 1).

Для того, чтобы не было ошибок при просмотре файлов без конфигурации, в home своего пользователя добавляю verilator.f с необходимыми аргументами.

```
-Wall
--timing
```

На этом настройка завершена, как можно убедиться, все ошибки из файла ушли.



Заключение

Open source инструменты для цифрового дизайна развиваются, и это не может не радовать. Verilator далеко не идеален, и некоторые языковые конструкции не поддерживаются. Согласно таблице соответствия стандарту, самая полная поддержка языка у библиотеки Slang, но она и инструменты на ее основе требуют куда большей настройки, так что пока я остановился на Verilator, который предоставляет хорошую диагностику по умолчанию.

Источники

1. https://microsoft.github.io/language-server-protocol/
2. https://github.com/chipsalliance/verible
3. https://github.com/YosysHQ/oss-cad-suite-build
4. https://github.com/mfussenegger/nvim-lint
5. https://veripool.org/guide/latest/exe_verilator.html#
6. https://chipsalliance.github.io/sv-tests-results/
7. https://github.com/MikePopoloski/slang/

Заметки ПЛИСовода

Туровский Дмитрий Николаевич

E-mail: dim7881@rambler.ru

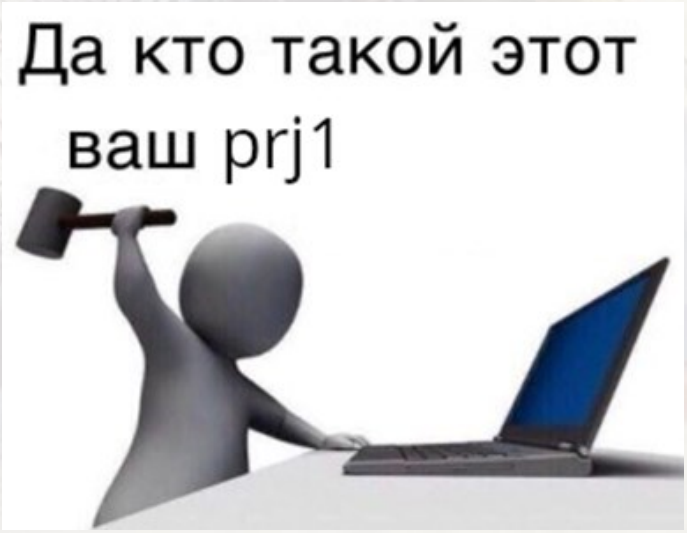
Обсуждение и комментарии: link

Согласно имеющемуся опыту и наблюдениям за проектами любителей и начинающих инженеров можно отметить наиболее общие и часто встречающиеся недочёты при создании проектов ПЛИС:

1. Неочевидные, «неговорящие» имена проектов.
2. Неполная первоначальная настройка проекта или её отсутствие.
3. Отсутствие организованного хранения файлов проектов.
4. Несовпадающие с электрической схемой имена сигналов TOP-модуля проекта.
5. Несоответствие настроек IO Standard сигналов проекта с электрической схемой.

Далее рассмотрим подробнее каждый из пунктов.

Имя проекта. Обычная вещь, над которой казалось бы, можно особо не заморачиваться на начальных этапах. В процессе пока у вас имеется только пара проектов, действительно, можно не придавать особого значения имени. Однако, далее проектов становится больше и спустя время разобраться для чего был создан проект бывает не так просто.



Поэтому желательно с самого начала придумать короткое и понятное имя, чтобы в будущем можно было быстрее разобраться, что за проект перед вами.

Первоначальная настройка проекта. В этом пункте поговорим немного о наболевшем, заострим внимание на одном аспекте в качестве примера. При первоначальной настройке проекта есть пункт выбора целевого ПЛИС, для которого создаётся проект. Выбирая целевую ПЛИС, партномер в проекте должен совпадать с маркировкой на корпусе. Повторим —

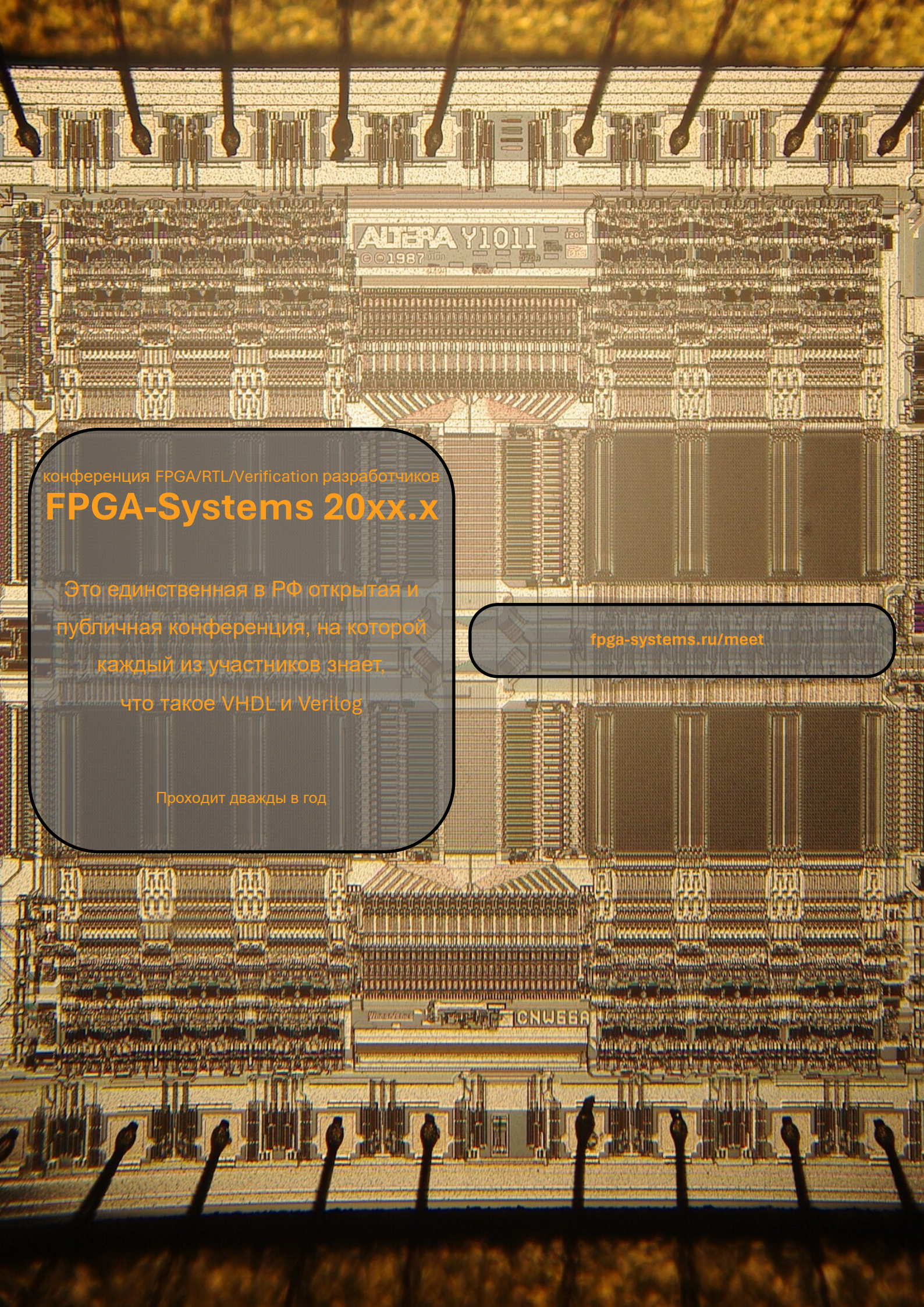
должен совпадать полностью. В противном случае вы рискуете столкнуться с ситуацией, когда полностью рабочий проект не будет работать на ПЛИС после загрузки конфигурационного файла в него, или, например, часть задействованных ip-ядер не будет работать. Поэтому уделяйте особое внимание первоначальной настройке проекта, с целью снизить трудозатраты на поиски проблем.

Отсутствие организованного хранения исходных файлов verilog-модулей, ip-ядер и прочих приведёт к беспорядку в каталоге проекта. При создании последующих проектов вам придётся копировать все файлы в новые каталоги, если появится необходимость использовать их снова. В случае имеющегося отдельного древа каталогов вам нужно будет лишь указывать пути к файлам. Поэтому стоит заранее задуматься, обратить внимание на образцы организации хранения исходных файлов и создать каталоги.

Несовпадающие с электрической схемой имена сигналов TOP-модуля проекта. Иногда можно столкнуться с ситуацией, когда имена сигналов TOP-модуля проекта названы произвольно, при этом имена либо полностью, либо почти не взаимосвязаны с именами сигналов на электрической схеме. В таком случае сложно разобраться что это за сигнал без тщательного изучения схемы от контакта ПЛИС до конечного устройства, для которого этот сигнал предназначался. Такие имена сигналов несут в себе нулевую информацию, поэтому с самого начала в TOP -модуле имена необходимо давать осмысленные, взаимосвязанные с именами сигналов на электрической схеме.

Несоответствие настроек IO Standard сигналов проекта с электрической схемой. Очень часто встречается ситуация «сферического коня в вакууме», когда ПЛИС как будто бы является изолированным устройством на плате от остальной части устройств.

Это видно, например, по нетронутым настройкам сигналов, которые остаются по умолчанию, как, например, IO Standard. Разработчиками игнорируется важнейшая информация о параметрах других устройств из электрической схемы, подключённых к ПЛИС, их уровнях напряжений и т.д. Да, выяснять эту информацию из схемы, читать документацию на другие устройства является утомительной, рутинной работой, но очень важной. ПЛИС является составной частью схемы, поэтому это нужно и важно учитывать, хотя бы для того, чтобы ускорить отладку проекта.



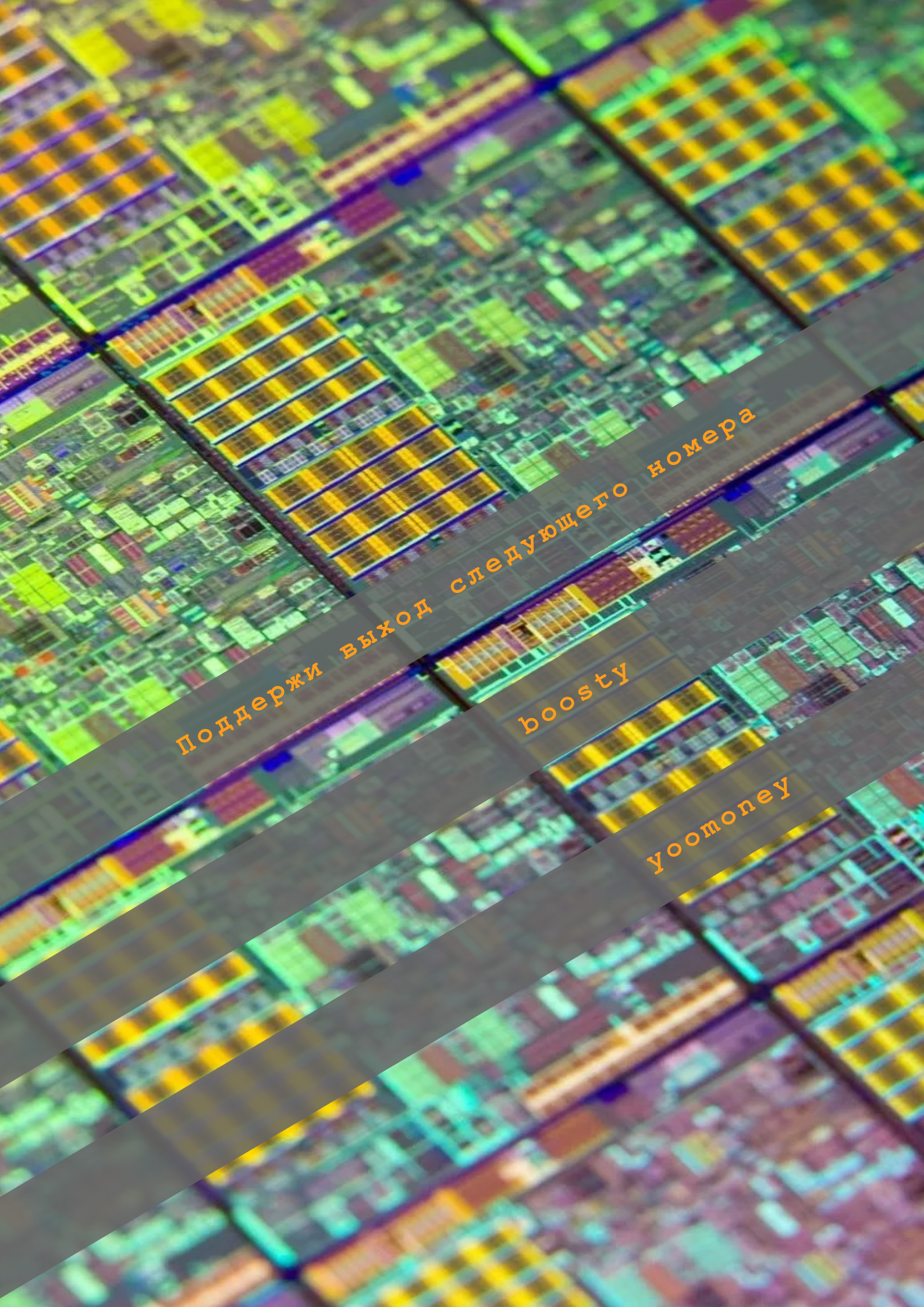
конференция FPGA/RTL/Verification разработчиков

FPGA-Systems 20xx.x

Это единственная в РФ открытая и публичная конференция, на которой каждый из участников знает, что такое VHDL и Verilog

Проходит дважды в год

fpga-systems.ru/meet


Поддержи выход следующего номера
boosty
yoomoney

Профильные телеграм каналы и чаты

@fpgasystems	основной телеграм-чат, в котором задаются вопросы по разработке на ПЛИС.
@fpgasystems_embd	телеграм-чат, где обсуждается разработка для процессорной части систем на кристалле (таких как Zynq-7000, Zynq MP, Intel SoC и др.) или софт-процессоров (например, Nios II, Microblaze, RISC-V, MIPS и др.). Также здесь можно получить ответы на вопросы, связанные с C, Linux и т. д.
@fpgasystems_verification	телеграм-чат по верификации. Здесь обсуждают вопросы, связанные с тестовыми покрытиями, интеграцией Verification IP, использованием верификационных библиотек и фреймворков (UVM, OVM, UVVM, OSVVM, AVM, eRM, URM, RVM, RMM, VMM и др.).
@fpgasystems_dsp	телеграм-чат, где обсуждаются вопросы по цифровой обработке сигналов и их последующей реализации на ПЛИС.
@fpgasystems_events	информационный канал, в котором публикуются свежие новости, анонсы мероприятий и вебинаров, выход новых видео, статей и книг, вакансии и т.д. по проблематике проектирования FPGA/RTL/Верификации
@cpu_design	телеграм-канал, который ведет амбассадор RISC-V International и рассказывает о магии процессоростроения;
@zynq7000	телеграм-канал, посвященный разработкам на Zynq-7000. Автор также рассказывает о работе и с другими платформами
@verif_for_all	телеграм канал одного из преподавателей «Школы синтеза цифровых схем», где просто и понятно рассказывается о верификации цифровых устройств;
@positiveslack	телеграм канал по проблематике ASIC, FPGA, SystemVerilog, UVM; здесь обсуждают цифровой дизайн с уклоном в верификацию
@embedoka	авторский телеграм-канал embedded-инженера
@vlsihub	авторский телеграм-канал о чипмейкерстве и ПЛИСоводстве
@enginegger	телеграм-канал, ориентированный на использование открытых инструментов для FPGA/RTL-разработок и верификации
@docstech_offical	Сайт-документация. Перевод ОФИЦИАЛЬНЫХ документов, руководств и гайдов на русский язык. https://docstech.ru

КОНТАКТЫ | CONTACTS

admin@fpga-systems.ru

Почта издателя | Publisher's e-mail

@fpgasystems_fsm

Канал обсуждения статей из журнала в телеграм
Telegram discussion group

FPGA-Systems.ru/fsm

Сайт журнала | Magazine web-page

@fpgasystems

Телеграм FPGA комьюнити | Telegram of FPGA community

Тел | Phone :: +7-929-955-68-75

FPGA-Systems Magazine :: FSM :: № BETA (state_1)
Первый журнал о программируемой логике