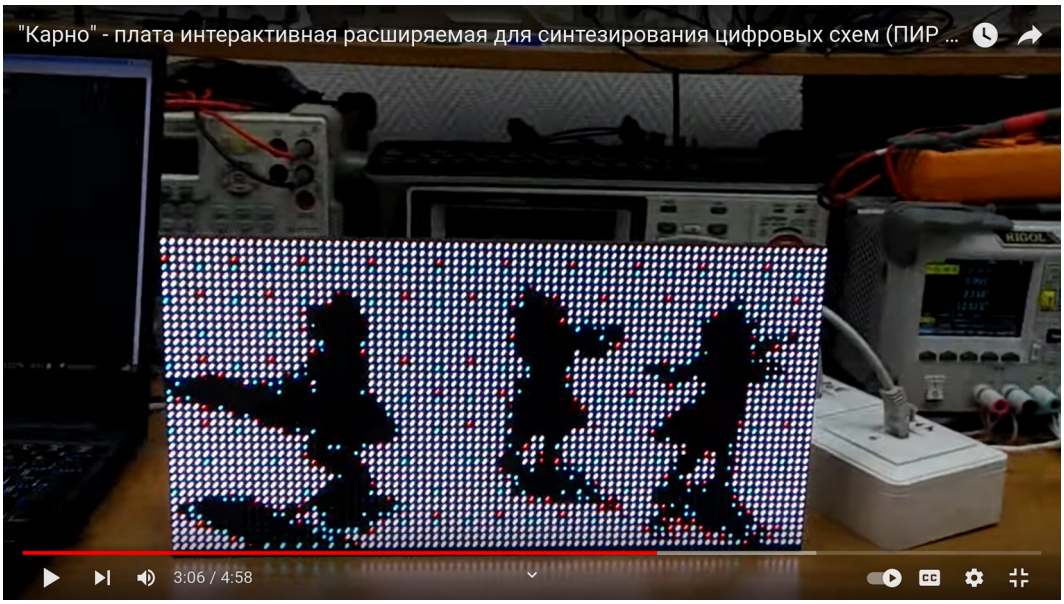
Разработка цифровой аппаратуры нетрадиционным методом: Yosys, SpinalHDL, VexRiscv

Основная прелесть использования ПЛИС, на мой взгляд, состоит в том, что разработка аппаратуры превращается в программирование со всеми его свойствами: написание и отладка кода как текста на специализированных языках описания аппаратуры (HDL); код распространяется в виде параметризованных модулей (IP-блоков), что позволяет его легко переиспользовать в других проектах; распределенная разработка обширным коллективом разработчиков с системой контроля версий, такой же, как у программистов (Git); и, как и в программировании, ничтожно низкая стоимость ошибки.

Последнее очень важно, так как если при разработке устройства классическим методом разработчик несет вполне существенные затраты на сборку и производство изделия, и любая схемотехническая ошибка или ошибка трассировки печатной платы — это всегда выход на очередную итерацию и попадание на деньги, то при работе с ПЛИС ошибки ничтожны по своей стоимости и легко устранимы. И даже если в серийном изделии обнаруживается ошибка, то её во многих случаях можно устранить очередным апгрейдом прошивки «в поле» без замены изделия. Короче, с приходом ПЛИС разработка цифровой аппаратуры все больше и больше выглядит как программирование, а это, помимо всего прочего, существенно понижает порог вхождения в тему, и все больше программистов становятся разработчиками «железа». А новые люди, в свою очередь, приносят с собой в индустрию новые подходы и принципы.

В этой статье я хочу поделиться своим небольшим опытом «программирования» микросхем ПЛИС и тем, как я постепенно погружался в тему ПЛИСоводства. Изначально я собирался написать небольшую заметку про открытый тулчейн для синтеза Yosys. Потом — про язык SpinalHDL и синтезируемое микропроцессорное ядро VexRiscv, на нём написанное. Потом — про замену микроконтроллеров микросхемами ПЛИС на примере моей отладочной платы «Карно». Но в процессе я погрузился в историю появления Hardware Description Languages (HDL), и когда я начал писать, Остапа, как это часто бывает, понесло... В общем, получилось то, что получилось.

А еще эту статью можно рассматривать как глубокое погружение в то, что происходит вот на этом новогоднем видео: https://www.youtube.com/watch?v=xr0pdGuJbXQ



СОДЕРЖАНИЕ

1. Краткий экскурс в историю

История микросхем началась в 1960 году, когда группа Джея Ласта из Fairchild Semiconductors продемонстрировала первый работающий полупроводниковый прибор, изготовленный по планарной технологии. Прибор получил название «Интегральная Схема» (или IC от «Integrated Circuit»), а в русскоязычной литературе часто используют просто слово «микросхема», независимо от степени её сложности. История создания микросхемы весьма запутанная, так как этому событию предшествовала серия открытий и изобретений, выполненных различными людьми из разных компаний и стран, поэтому вслед за изобретением микросхемы последовали патентные войны — каждый пытался откусить от пирога кусок поболее. Первая микросхема в СССР была создана в 1961 году в Таганрогском радиотехническом институте. В течении последующих 10 лет мировой радиоэлектронной промышленностью было предложено и испытано множество способов производства микросхем, почти все они полагались на планарный метод изготовления транзисторов на кремниевой, германиевой или сапфировой пластине с помощью процесса фотолитографии — итеративным процессом последовательного нанесения фоторезиста, засветки, проявления, травления, промывки, ионной имплантации и металлизации, а схема, которая образовывалась таким образом, задавалась набором фотомасок.

Примерно до средины 1980-х годов, а это почти 25 лет, все аналоговые и цифровые микросхемы, в том числе микросхемы памяти и микропроцессоров, разрабатывались исключительно путем черчения схем, сначала на бумаге, позже — с использованием САПР, которые по тем временам были весьма примитивны. Но в обоих случаях рисовалась схема электрическая принципиальная, в которой прослеживался каждый проводник. Когда схема была готова, разработчики переходили к её трассировке — формированию расположения отдельных электронных компонентов (транзисторов, резисторов и конденсаторов) на поверхности кристалла, по трассировке формировали набор фотомасок. В этом им часто помогала специальная клейкая лента-скотч Рубилит, которую тонкими полосками наклеивали на прозрачную майларовую пленку (или наоборот — удаляли лишнее), таким образом формируя узор будущей микросхемы в увеличенном формате. Этот узор фотографическим методом и с уменьшением в 100 раз переносился на фотомаску, которая далее использовалась в планарном фотолитографическом производстве.

Рис. 1. Rubylith operators, 1970, Courtesy of the Intel Corporation.

Микросхемы очень быстро росли в степени интеграции — увеличении числа транзисторов и уменьшении площади на кристалле кремния, что в геометрической прогрессии тянуло за собой увеличение сложности изготовления фотомасок и их стоимости. Очевидно, что такой метод разработки микросхем не мог обойтись без ошибок и для достижения рабочего образца приходилось выполнить порой до десятка итераций, что также не могло не сказываться на стоимости продукции. Изготовление рабочего набора фотомасок занимало значительную часть финансовых и временных затрат на производство микросхемы. Для того, чтобы минимизировать затраты и сократить время, разработчикам приходилось вручную проверять и перепроверять свои дизайны на всех этапах. В какой-то момент схемотехника достигла такой сложности, что проверить её работоспособность вручную стало практически невозможно и логичным решением этой проблемы стало внедрение автоматизированных процессов верификации и симуляции дизайнов. А это означало, что весь дизайн, от схемы до фотомасок, должен был быть теперь представлен в цифровом формате, т. е. в виде данных в файлах: схемы в виде связанного графа элементов (нетлиста), а фотомаски — в виде последовательности команд на языке машин фотоплоттеров (Gerber). Так, для представления схемы в виде соединения отдельных её элементов в электронном виде появились языки описания аппаратуры — HDL от «Hardware Description Language», а процесс разработки стали назвать «VLSI design» от «Very Large Scale Integration» (в русскоязычной литературе — СБИС от «Сверх Большая Интегральная Схема»).

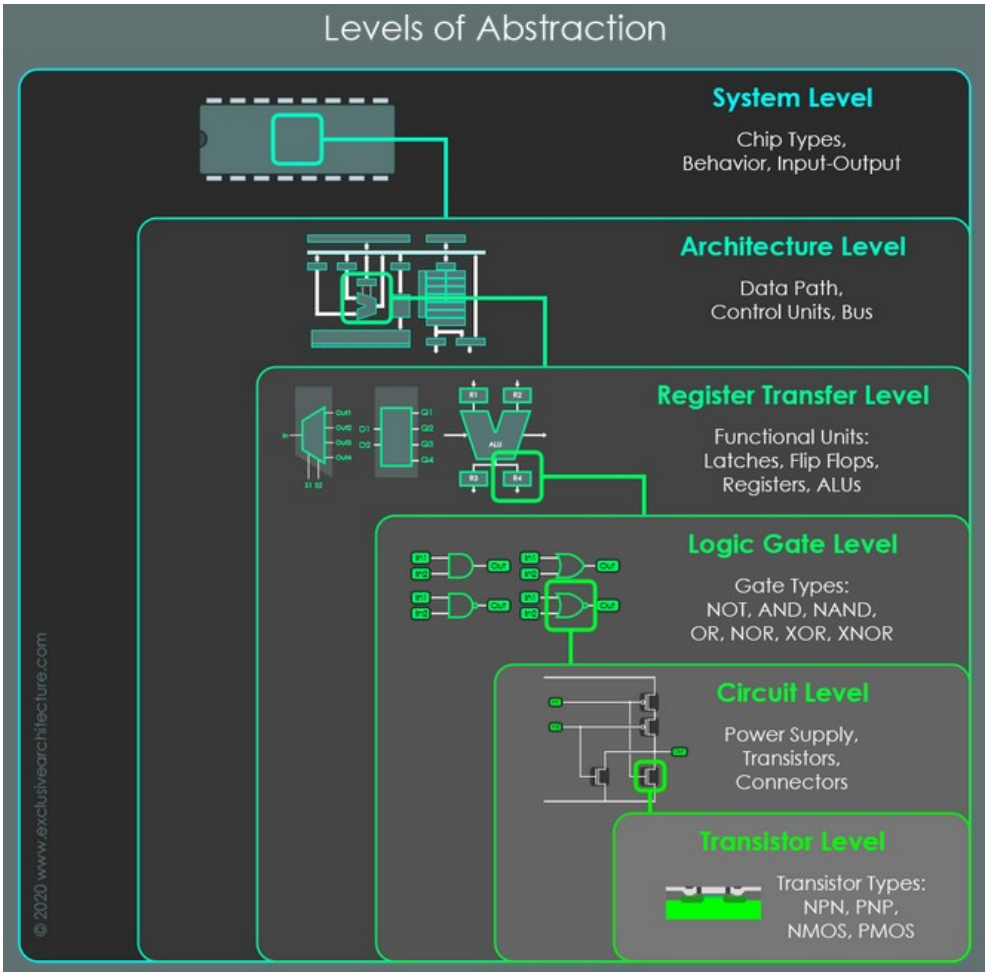


Рис 2а. Уровни абстракции в цифровой электронике.

Языки описания аппаратуры (HDL) позволяют представить прежде всего цифровую схему (хотя конечно же есть языки для описания аналоговых схем) в виде текста, подобно языкам программирования, описывают структуру схемы, потоки данных и её поведение с течением времени. История появления HDL насчитывает множество способов описания схем, но способ представления схем на уровне регистровых передач (RTL от «Register Transfer Level») впервые предложенный Гордоном Беллом в 1971 году оказался наиболее удобен для верификации сложных цифровых схем. На основе RTL был создан языке ISP (ISPL и ISPS) на котором была создана модель ЭВМ DEC PDP-8. С тех пор RTL метод используется во всех современных HDL языках для разработки и верификации VLSI дизайнов.

Что бы понять, что такое RTL, необходимо представить цифровую схему в виде отдельных регистров (D-триггеров), управляемых общим тактовым сигналом, а входы и выходы этих регистров обеспечить комбинационной логикой и связать между собой в заданную схему. Таким образом, все цифровые схемы в терминах RTL являются синхронными — данные в них перетекают из одного регистра в другой по переднему фронту тактового сигнала (иногда по спаду). Данные, перетекая между регистрами, претерпевают изменения (обработку) комбинационными схемами, которые работают асинхронно, а задержка распространения сигнала в них не учитывается на уровне RTL, т. е. условно полается, что комбинационные схемы срабатывают мгновенно и 100-процентно (что, разумеется, не так, но об этом отдельно). Ниже на рисунке 2 приведена простейшая синхронная цифровая схема, состоящая из одного регистра (триггера) и одного инвертора. Схема представляет собой делитель тактовой частоты на 2.

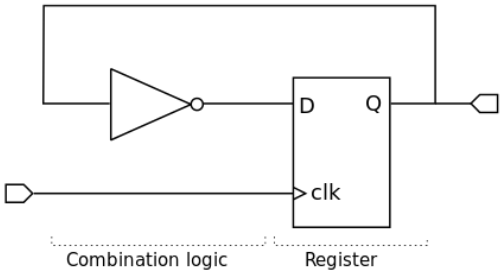


Рис. 2б. Пример элементарной цифровой синхронной схемы (RTL уровень).

Приведенная выше схема может быть описана текстом на языке описания аппаратуры следующим образом:

```
if rising_edge(clk) then
        Q <= ~D;
end if;
```

Это означает, что выходной сигнал **Q** примет значение инверсное (противоположное) входному сигналу **D** в следующем такте (сигнал сброса не показан для упрощения понимания).

Данный текст на HDL может быть не только просимулирован, но и с помощью специальных программных средств т. н. «кремниевых компиляторов» может быть автоматически преобразован сначала в связанный граф примитивов (отдельных логических элементов и транзисторов), а затем в их геометрическое представление на кристалле кремния

и в конечном счете может быть получен набор фотомасок. В современном производстве фотомаски для изготовления микросхем представляются в электронном формате GDSII, а формат Gerber занял нишу в производстве печатных плат. Процесс преобразования текста HDL в файлы фотомасок раньше было принято называть «аппаратной компиляцией», но сейчас больше используется термин «синтез цифровых схем» (logic synthesis), так как конечным результатом может быть не только геометрическая форма (фотомаска), но некий поток двоичных данных, определяющий конфигурацию микросхемы ПЛИС для реализации заданной цифровой схемы внутри неё.

История гласит, что легендарный микропроцессор Intel 80386 является первым микропроцессором компании Intel, который был полностью разработан с помощью цифрового инструментария. Для него была построена полная RTL модель, дизайн итеративно дорабатывался и симулировался потактово до тех пор, пока не было достигнуто совпадение всех симулируемых сигналов с проектными значениями, или, как принято говорить, с «эталонной моделью» (golden reference model). Вот что пишет известный блоггер Кен Ширрифф в свой статье «Examining the silicon dies of the Intel 386 processor» про историю разработки i386:

> *The design process of the 386 is interesting because it illustrates Intel's migration to automated design systems and heavier use of simulation. At the time, Intel was behind the industry in its use of tools so the leaders of the 386 realized that more automation would be necessary to build a complex chip like the 386 on schedule. By making a large investment in automated tools, the 386 team* ***completed the design ahead of schedule****. Along with proprietary CAD tools, the team made heavy use of standard Unix tools such as* `sed`*,* `awk`*,* `grep`*, and* `make` *to manage the various design databases.*

И далее:

> *The high-level design of the chip (register-level RTL) was created and refined until clock-by-clock and phase-by-phase timing were represented. The RTL was programmed in MAINSAIL, a portable Algol-like language based on SAIL (Stanford Artificial Intelligence Language). Intel used a proprietary simulator called Microsim to simulate the RTL, stating that full-chip RTL simulation was "the single most important simulation model of the 80386".*

The next step was to convert this high-level design into a detailed logic design, specifying the gates and other circuitry using Eden, a proprietary schematics-capture system. Simulating the logic design required a dedicated IBM 3083 mainframe that compared it against the RTL simulations. Next, the circuit design phase created the transistor-level design. The chip layout was performed on Applicon and Eden graphics systems...

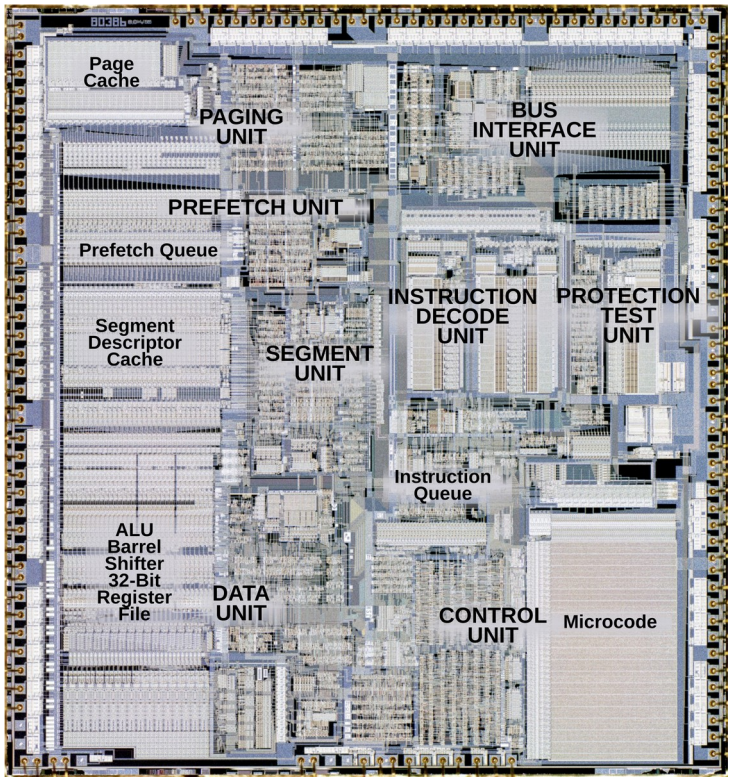


Рис.3.Фотография кристалла микропроцессора 80386 с обозначением функциональных областей. 1985г. Заимствованно из блога Кена Ширриффа, сайт http://www.righto.com/.

В современной практике широкое использование получили два языка описания аппаратуры: VHDL и Verilog. Verilog был предложен частной компанией Gateway Design Automation и входил в состав коммерческой системы разработки микросхем. Взамен проприетарному Verilog-у в недрах минобороны США был разработан VHDL и предложен как открытый стандарт. Изначально оба языка предназначались для документирования и симуляции цифровых схем, но к 1985 году стало понятно, что разрабатывать схемы «в тексте» гораздо легче, чем рисовать их, пусть даже на экране монитора. Такие схемы легче анализировать и модифицировать, легче систематизировать и каталогизировать, над одним и тем же куском кода (схемы) может работать несколько человек, но самое главное — код схемы можно переиспользовать от дизайна к дизайну, легко изменяя небольшую его часть. Код схемы даже может быть параметризован, выделен в отдельные блоки, упакован и продан. Появления HDL привело к созданию целой индустрии купли-продажи IP блоков (здесь IP от «Intellectual Property» — интеллектуальная собственность).

В 1990 году компания Cadence приобрела компанию Gateway Design Automation и сделала Verilog открытым, вскоре после чего оба языка были приняты IEEE как открытые стандарты. За почти 40-летнюю историю существования языки развивались, в них вносились серьезные изменения, добавлялись и расширялись стандартные библиотеки (VHDL), вводились и принимались новые стандарты. VHDL традиционно имел широкое хождение в авиакосмической и военной отрасли (как в США, так и в России, кстати), в то

время как Verilog больше популярен в академической среде и в небольших коммерческих компаниях. На начало 2020 годов проявилась явная тенденция к превалированию языка Verilog и к спаду популярности VHDL. С чем это связано — мне сказать трудно.

В 2002 году от языка Verilog отпочковался SystemVerilog который в 2005 году так же был принял стандартом IEEE 1800-2005, а в 2009 году язык Verilog вошел как подмножество в язык SystemVerilog и был принят как стандарт IEEE 1800-2009 SystemVerilog. Отличие Verilog от SystemVerilog примерно такое же, как отличие языков C и C++. SystemVerilog поддерживает классы и объекты, а также сложные структуры сигналов. В современной практике, когда говорят Verilog, подразумевают именно SystemVerilog, либо явно подчеркивают его старую версию стандарта.

Ниже я приведу в качестве примера схему двоичного 4-х битового полного сумматора описанную на двух языках. На всякий случай напомню, что полным сумматором называется такой сумматор, который выполняет сложение двух входных двоичных чисел заданной разрядности с учетом входного сигнала переноса и формированием на выходе как суммы, так и выходного сигнала переноса, что позволяет соединять полные сумматоры в каскады и получать сумматоры большей битности.

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

entity ripple_adder is
    port (
        a    : in  std_logic_vector(3 downto 0);
        b    : in  std_logic_vector(3 downto 0);
        cin  : in  std_logic;
        cout : out std_logic;
        s    : out std_logic_vector(3 downto 0));
end entity;

architecture rtl of ripple_adder is
    signal sum : std_logic_vector(4 downto 0);
begin
    sum  <= ('0'&a)+('0'&b)+("0000"&cin);
    cout <= sum(4);
    s    <= sum(3 downto 0);
end architecture;
```

Рис. 4. *Схема 4-х битового полного сумматора на языке VHDL.*

```
module fulladd (  input [3:0] a,
                  input [3:0] b,
                  input c_in,
                  output reg c_out,
                  output reg [3:0] sum);

    always @ (a or b or c_in) begin
    {c_out, sum} = a + b + c_in;
  end
endmodule
```

Рис. 5. Схема 4-х битового полного сумматора на языке Verilog (IEEE 1364-2001).

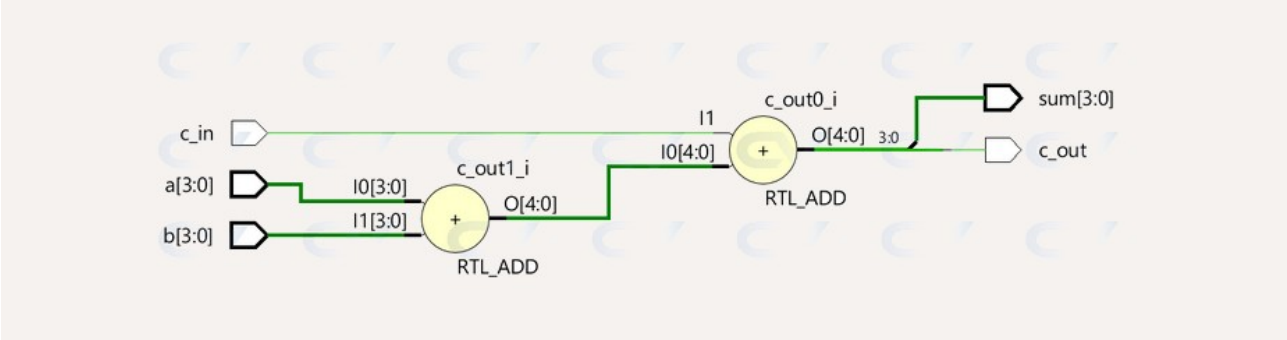


Рис. 6. Схема 4-х битового полного сумматора синтезированная из выше приведенного кода на языке Verilog.

Из приведенного выше примера видно, что VHDL гораздо более многословен, а его синтаксис не всегда понятен и очевиден. В то же время, синтаксис языка Verilog в какой-то степени похож на синтаксис языка Си. Возможно это является одной из причин падения популярности языка VHDL, так как за последние годы в тему разработки цифровых схем вливается всё больше программистов.

2. Появления программируемых логических устройств

Идея сделать логические схемы программно реконфигурируемыми пришла вместе с появлением микросхем памяти (ОЗУ, ПЗУ) в середине 1960х, так как любую микросхему памяти можно рассматривать как логическое устройство, преобразующее набор входящих сигналов, поступающих на линии адреса, в выходящие — снимаемые с линий данных. В этом смысле любая микросхема памяти является аппаратной таблицей преобразования («lookup table», «таблица истинности») и может быть использована для построения перепрограммируемых дешифраторов. И действительно, такое её применение часто использовалось в процессорах больших ЭВМ древности для построения дешифраторов команд, да и по сей день в современных микропроцессорах дешифратор команд часто строится на перепрограммируемой ПЗУ. Построение дешифратора команд на основе перепрограммируемой ПЗУ является поистине великим изобретением, так как позволяет относительно легко исправлять ошибки в системе команд (но не всегда, привет Intel-у) либо

вести отладку процессора загружая в память дешифратора специально подготовленные тестовые последовательности.

Использование перепрограммируемых ПЗУ для построения логических схем, разумеется, имеет свои ограничение. Пожалуй, самое очевидное из них — это гигантское число ячеек памяти, необходимое для построения дешифраторов с большим числом входов и выходов. Вспомним, что в 60-х и 70-х годах прошлого века компьютеры были большие, а память у них была крохотной и жутко дорогой. Поэтому в начале 70-х годов сразу несколько компаний, среди которых были такие известные и ныне гиганты полупроводниковой индустрии, как Texas Instruments, General Electric и National Semiconductor, представили на рынок свои варианты микросхем, содержащих массивы логических элементов, соединения между которыми можно было в некоторой степени конфигурировать программно — путем одноразового пережигания перемычек, т. е. использовался тот же самый принцип, что и при программировании ПЗУ. А в некоторых изделиях «прошивку» можно было даже стирать ультрафиолетом и перепрограммировать перемычки повторно. Такие изделия получили название «Programmable Logic Array» — PLA или PAL.

В микросхемах PAL использовалась регулярная структура (рис. 7) из логических элементов «И», обычно рисуемых на схемах горизонтально, выходы которых соединялись логическими элементами «ИЛИ», рисуемых вертикально, таким образом образуя «сумму произведений» (sum of product), что соответствует дизъюнктивной нормальной форме (ДНФ), к которой, согласно булевой алгебре, может быть сведена любая булева формула. Данные структуры в микросхемах PAL, многократно повторяясь, образовывали массив, используя который можно было реализовать любую булеву функцию.

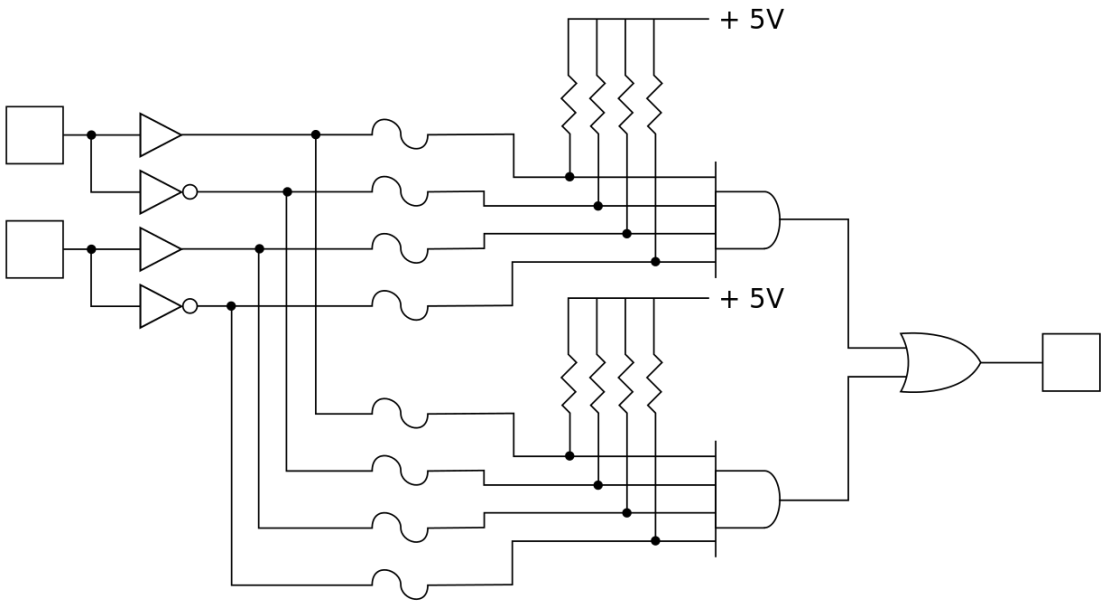


Рис. 7. Упрощенная схема PAL — программируемого логического массива. Программируемые элементы в этой схеме это «фьюзы» - пережигаемые и восстанавливаемые предохранители.

Для программирования микросхем PLA/PAL в 1983 года компанией Monolithic Memories, Inc был разработан специальный язык описания программируемой логики — PALASM, который позволял преобразовать булевы функции и таблицы переходов состояний в таблицу пережигаемых «фьюзов». Компания MMI производила свой вариант микросхемы программируемой логики PAL16R4, появление такого программного инструмента сделало

эту микросхему очень популярной, а сам язык PALASM превратился в стандарт «де факто», который использовали подавляющее большинство разработчиков, имеющих дело с PAL.

```
PAL16R4 PAL              PAL DESIGN SPECIFICATION
CNT4SC
4 bit counter with synchronous clear
Michael Holley and Dave Pellerin
Clk  Clear  NC  NC  NC  NC  NC  NC  NC  GND
 OE  NC     NC /Q3 /Q2 /Q1 /Q0  NC  NC  VCC

  Q3 :=  Clear
       + /Q3 * /Q2 * /Q1 * /Q0
       +  Q3 *  Q0
       +  Q3 *  Q1
       +  Q3 *  Q2

  Q2 :=  Clear
       + /Q2 * /Q1 * /Q0
       +  Q2 *  Q0
       +  Q2 *  Q1

  Q1 :=  Clear
       + /Q1 * /Q0
       +  Q1 *  Q0

  Q0 :=  Clear
       + /Q0

FUNCTION TABLE
OE Clear Clk   /Q0 /Q1 /Q2 /Q3
------------------------------
 L   H    C     L   L   L   L
 L   L    C     H   L   L   L
 L   L    C     L   H   L   L
 L   L    C     H   H   L   L
 L   L    C     L   L   H   L
 L   H    C     L   L   L   L
------------------------------
```

Рис. 8. Пример реализации 4-х битного счетчика для PAL16R4 на языке PALASM.

В 1985 году компания Lattice Semiconductor представила свой, расширенный и углубленный вариант микросхемы PAL, которую также можно было перепрограммировать многократно электрическим путем по технологии ЭППЗУ (EEPROM). Эта микросхема имела некоторые структурные усовершенствования: в структуру были введены так называемые «макроячейки» («macrocels») содержащие помимо логических «И» и «ИЛИ» еще регистры (D-триггеры) и мультиплексоры для обхода регистров. Также появилась возможность организовывать обратные связи — подавать сигнал с выхода триггера на вход этой же или других ячеек. Схемы программируемой логики такого типа получили название «Generic Logic Array» (GAL). На рис 9. приведена структурная схема «макроячейки» Lattice GAL22V10, сама микросхема содержала десять таких «макроячеек» (22 входа и 10 выходов).

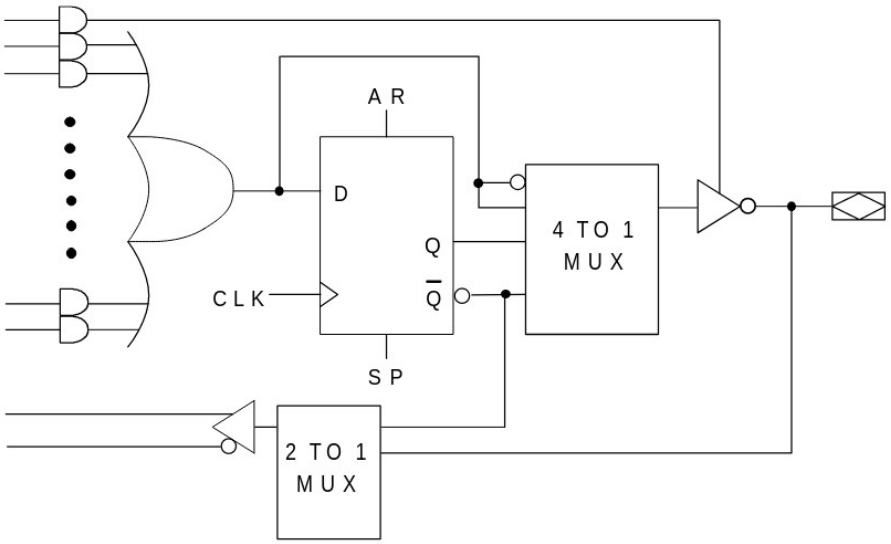


Рис. 9. Структура макроячейки (Output Logic Macro Cell) микросхемы GAL22V10.

Микросхема Lattice GAL22V10 была и остается очень популярной в схемах управления и автоматики, а также в качестве связывающей логики (glue logic). Микросхема производилась до 2010 года, а её аналог Atmel ATF22V10 производится и по сей день. Для программирования этой микросхемы был разработан специализированный HDL язык CUPL и среда разработки Atmel WinCUPL. Эта среда позволяет синтезировать из кода на языке CUPL простые цифровые схемы для большого ряда PAL/GAL микросхем или экспортировать его в формате PALASM для применения в других программных продуктах.

Недавно, занимаясь «цифровым ретро», в процессе которого мне требовалось отображать состояние шины адреса и данных для 8-ми битной машины, я с удивлением для себя обнаружил, что в стандартном наборе логики 7400 отсутствует такое полезное устройство, как дешифратор 4-х битовых двоичных чисел в их шестнадцатеричное представление для отображения на 7-ми сегментном индикаторе. Проблема быстро решилась с помощью микросхемы ATF22V10 и небольшого кусочка кода для CUPL, позаимствованного из репозитория Дуга Габбарда, за что ему огромное спасибо. Полный проект и схему подключения микросхемы GAL22V10 (она же ATF22V10) к 7-ми сегментному индикатору можно получить по ссылке: https://sourceforge.net/projects/dual-bcd-to-hex-7-seg-driver/?source=navbar

Добавлю, что опенсорсная реализация компилятора с языка CUPL/PALASM называется Portable GAL Assembler — GALasm и доступна в репозитории на Github-е. GALasm разработал Алессандро Зумо в начале 2000-х годов, но не смотря на такую «древность» его код прекрасно компилируется и работает в современных ОС на основе Linux и BSD.

Также хочу отметить, что на Хабре есть интересная статья «PAL, GAL и путешествие в цифровое Ретро» от пользователя **@alecv** о структурах и истории PLA, PAL и GAL, с примерами кода на PALASM.

Изделия PLA, PAL и GAL бурно развивались, количество вентилей (логических элементов) и «макроячеек» стремительно увеличивалось, добавлялись новые логические блоки, и позже такие устройства стали называть Programmable Logic Device (PLD) и Complex Programmable Logic Device (CPLD), а в русскоязычной литературе — «Программируемые Логические Устройства». Подавляющее большинство ПЛУ (CPLD) имеют встроенную «перепрошиваемую» память, определяющую связи элементов внутри микросхемы. Так как

процесс перепрошивки подразумевает восстановление и повторное прожигание «фьюзов», то количество итераций записи в такие устройства ограничено — от нескольких десятков до нескольких сотен циклов.

Параллельно с ПЛУ шло развитие программируемой логики и по другому пути. Вместо отдельных вентилей с программируемыми связями, стали использовать массив достаточно крупных блоков логики — «Configurable Logic Blocks» (CLBs), «Logical Units» (LUs), «Logical Cells» (LCs) или «макроблоков». Терминология здесь сильно разнится и зависит от фантазии продаванов конкретного производителя микросхем. Каждый макроблок в таких изделиях имеет в составе программируемую таблицу преобразования («lookup table» или LUT) с двумя, тремя или четырьмя входами, а так же регистр и набор мультиплексоров. Макроблоки располагали в прямоугольную структуру, поверх которой располагалась матрица коммутации. Схема коммутации между входами и выходами макроблоков в таких устройствах задается ячейками статической памяти, выходы которых подключаются к транзисторным ключам матрицы коммутации. Для того чтобы внутри устройства образовалась какая-то схема соединений между макроблоками и сами макроблоки приобрели определенные свойства, необходимо записать данные в требуемые статические ячейки памяти, т. е. загрузить в микросхему битовый поток конфигурации. Такие устройства получили название «Field Programmable Gate Array» (FPGA), в русскоязычной литературе — «Программируемые Логические Интегральные Схемы» (ПЛИС). Структурная схема макроблока (логической ячейки) классической микросхемы ПЛИС приведена на рисунке 10 ниже.

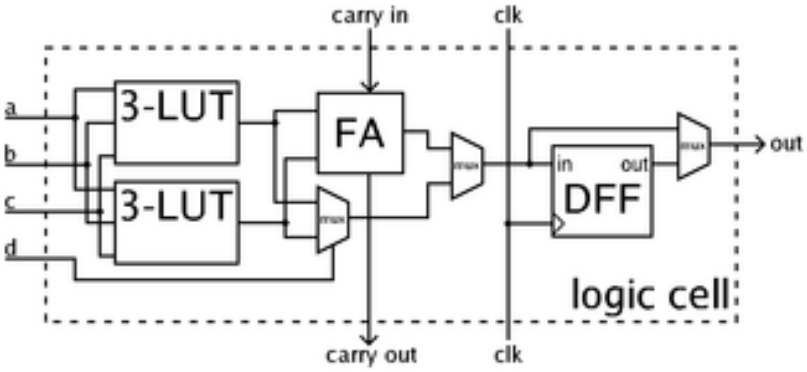


Рис. 10. Типовая структура «логического блока» классической ПЛИС на основе 4-LUT. На схеме: 3-LUT — трехвходовая таблица преобразования, FA — полный сумматор, DFF — D-триггер (регистр).

Устройства из массива макроблоков такой структуры имеют определенные преимущества над массивом логических вентилей PLA/PAL и PLD. Если посмотреть на структуру макроблока, то можно определить сходство макроблока с элементарной синхронной схемой, изображенной на рис. 2, которая лежит в основе RTL — комбинационная логика на входе (LUT + сумматор) и регистр (DFF) на выходе. А значит из таких кирпичиков можно складывать синхронные схемы обработки данных и вычислительные блоки.

Пару слов про LUT («lookup table»). LUT представляет собой аппаратную программно реконфигурируемую булеву функцию (это может быть «И», «ИЛИ», «НЕ» или любая другая). Реализуется LUT обычно в виде иерархической структуры мультиплексоров, входы нижнего уровня которых подключены к ячейкам статической памяти, задающим выполняемую функцию, а входы управления — к входным сигналам, над которыми эта функция применяется.

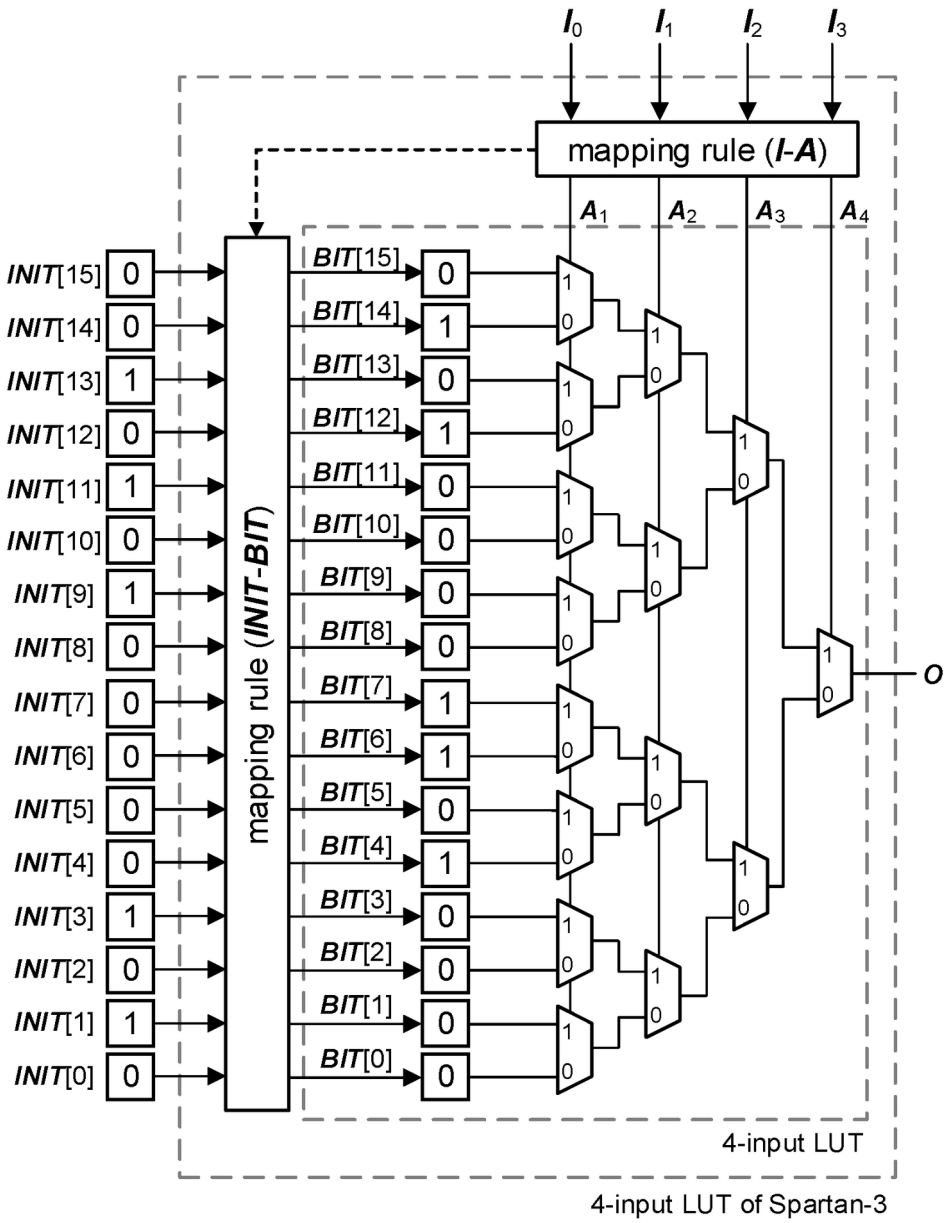


Рис. 11. Типовая схема 4-LUT для многих микросхем ПЛИС.

На схеме рис. 11 приведена типовая схема четырехвходовой «lookup table» (4-LUT). Линии управления мультиплексорами **I0**, **I1**, **I2** и **I3** являются входными двоичными сигналами для блока 4-LUT, а логическая операция, которая над ними выполняется, задается шестнадцатью битами статической памяти (SRAM) обозначенных на схеме как **BIT[0]**...**BIT[15]**, состояние которых определяется в момент загрузки конфигурации в микросхему ПЛИС, т. е. в момент её инициализации. Результат операции подается на выход **O**. На такую таблицу 4-LUT можно посмотреть и с другой стороны — это просто блок статической памяти, который содержит шестнадцать однобитовых ячеек, адресуемых линиями I0...I3. Из схемы видно, что 4-LUT состоит из двух 3-LUT блоков, соединенных мультиплексором верхнего уровня, каждый 3-LUT состоит их двух 2-LUT, также объединенных мультиплексором уровнем пониже, и так далее. Современные ПЛИС могут содержать разное количество LUT разной структуры, вплоть до 7-LUT.

Отличительной особенностью микросхем ПЛИС от ПЛУ (PLA/PAL/PLD/CPLD) долгое время была необходимость в наличии внешнего блока памяти, обычно NOR flash, для сохранения битового потока конфигурации ПЛИС, который должен быть загружен в микросхему ПЛИС при её включении в процессе инициализации. Этим же и определялся основной их недостаток — ПЛИС требует некоторого весьма существенного (до 100 мс) времени для выхода на рабочий режим. В этом смысле микросхемы ПЛУ более удобны, так как в них конфигурация внутренних связей всегда сохраняется состоянием прошитых «фьюзов» даже при отключении питания, а значит ПЛУ готовы к работе почти мгновенно после подачи питания или сброса. В некоторых случаях микросхемы ПЛУ использовались для загрузки конфигурации в микросхемы ПЛИС.

Однако многие современные микросхемы ПЛИС либо уже содержат внутри себя блок NOR flash памяти для сохранения конфигурации (хоть это и не решает проблемы медленного выхода на рабочий режим), либо также как ПЛУ используют технологию перепрограммируемых перемычек («фьюзов»). Последнее относится к изделиям с относительно небольшим числом макроблоков, так как электрически перепрограммируемые «фьюзы» занимают достаточно большую площадь на кристалле, которую целесообразней использовать для «материи» ПЛИС.

Первопроходцем в создании микросхем ПЛИС принято считать компанию Altera (ныне входит в состав Intel). Altera начала в 1983 году с создания достаточно простых PLA/PLD устройств, которые эволюционировали в сложные FPGA. Другим известным производителем микросхем ПЛИС является компания Xilinx (принадлежит AMD), которая вышла на рынок со своими PLD изделиями в 1985 году и на данный момент Xilinx является лидером рынка программируемой логики. Если я не ошибаюсь, то на момент написания данной статьи рекордсменом по числу макроблоков на одном кристалле является микросхема Virtex UltraScale+ VU19P пр-ва Xilinx, которая содержит умопомрачительные **9 миллионов «logical cells»** и огромное количество разнообразных «hard blocks».

Рис. 12. AMD/Xilinx Virtex UltraScale+ VU19P. 2020г.

Надо отметить, что в современных микросхемах ПЛИС помимо макроблоков (логических блоков, LUT-ов) присутствуют т. н. «hard blocks» — блоки специализированной логики и обработки данных, такие как: блоки преобразования частот (PLL), умножители и делители (MUL/DIV) и целые ALU, блоки обработки сигналов (DSP), блоки распределенной памяти (BRAM), блоки динамической памяти (SDRAM), сериализаторы-десериализаторы (SERDES), блоки цифровых интерфейсов и шин (DDR, PCIe, USB), высокоскоростные

трансиверы и т. д. С некоторых пор в микросхемы ПЛИС начали активно интегрировать вычислительные ядра ARM.

Таким образом, современные ПЛИС — это сложные гибридные многофункциональные устройства, средствами которых можно решать тяжелые специализированные задачи по высокоскоростной обработке сигналов; строить высокопроизводительные вычислительные системы для симуляции сложных физических процессов или симуляции других специализированных цифровых и аналоговых схем (т. н. ASIC - «Application Specific Integrated Circuits») и их верификации при разработке и производстве.

По некоторым данным (Wikipedia), при симуляции ASIC средствами ПЛИС исходят из следующих эмпирических соотношений: площадь симулируемого кристалла будет в 40 раз меньше выбранной для симуляции микросхемы ПЛИС, скорость симуляции составит примерно 1/3 от скорости работы ASIC, а потребляемая мощность ПЛИС будет в 12 раз больше.

Также отмечу, что ряд производителей микропроцессоров (AMD, Intel), начали внедрять в свои высокопроизводительные микропроцессорные изделия «материю» ПЛИС, связь с которой обеспечивается специализированными высокоскоростными интерфейсами внутри одного кристалла. Как применять на практике такие изделия и для решения каких именно задача — мне пока не понятно.

3. Производители микросхем ПЛИС

Я буду использовать термин ПЛИС (или англоязычный вариант — FPGA) как собирательный для всех видов программируемой логики, основанной на макроблоках или макроячейках, так как в современной действительности грань между CPLD и FPGA устройствами сильно размыта, а принципы работы с ними мало чем отличаются.

Прежде всего надо сказать, что каждый производитель микросхем ПЛИС разрабатывает и распространяет свой инструментарий (toolchain или IDE или просто «тул») — целую среду разработки, позволяющую выполнить всё: от разработки схем на нескольких HDL языках (SystemVerilog и VHDL поддерживаются всеми), до верификации, симуляции, визуализации, синтеза и заливки полученной конфигурации. Но важно понимать, что этот инструментарий поддерживает синтез только для микросхемы ПЛИС конкретного производителя и, как правило, является отдельным коммерческим продуктом, который реализуется за отдельные, весьма серьезные деньги. Разумеется, существуют условно-бесплатные ограниченные лицензии на эти тулы. Эти лицензии распространяемые в исключительно «образовательных целях» вполне пригодны не только для обучающихся, но и для вполне опытных разработчиков. Однако, если Вы пожелаете использовать какие-то специфичные «hard blocks» присутствующие в выбранной для Вашего дизайна микросхеме ПЛИС, то скорее всего придется раскошелиться.

Еще один важный момент состоит в том, что на рынке присутствует с десяток производителей микросхем ПЛИС, каждый предлагает огромный модельный ряд своих изделий. Если в рамках одного производителя, точнее даже в рамках одной серии микросхем от одного производителя, присутствует хоть какая-то совместимость, то между микросхемами и тулами разных производителей совместимости нет никакой, а порой даже находятся отличия в синтаксисе HDL языков (несмотря на то, что есть стандарты). Иными словами, если разработчик пристрастился к продукции, скажем, производства Xilinx, то перейти на использование и перенести свои разработки на изделия Alter-ы будет катастрофически непростой задачей. Проблемой здесь являются эти самые «hard blocks», которые специфичны для конкретной микросхемы и конкретного тула конкретного

производителя. «Hard blocks» очень сильно упрощают разработку, а во многих случаях без них вообще нельзя обойтись. Для части таких блоков у разных производителей могут быть эквивалентные замены (как например блоки PLL и SERDES есть у всех), но это не значит, что при переходе от одного вендора к другому вы сможете взять и просто перекомпилировать (пересинтезировать) свой дизайн с использованием других блоков. Нет, вам придется хорошо поработать руками и головой — переписать код и плотно протестировать его.

Далее я перечислю список известных производителей микросхем ПЛИС, некоторые из популярных моделей, а также среды разработки (тулы).

1. **Xilinx, Inc**. Куплена компанией AMD в 2022г.

Известна следующими линейками микросхем ПЛИС: Vertex (топовые), Kintex (среднего уровня) и Artix — для тех кому попроще. Также широко известна серия гетерогенных изделий Zynq-7000 содержащая целую систему-на-кристалле с вычислительными ядрами ARM в комбинации с «фабрикой» ПЛИС.

Широко известна также серия «классических» ПЛИС микросхем SPARTAN 2/3/4/5/6/7, часть из которых уже снята с производства.

Основной тул: Vivado Design Suite.

2. **Altera, Inc**. Куплена и поглощена компанией Intel в 2015г, но в октябре 2023г Intel заявил, что собирается выделить бизнес по производству ПЛИС в отдельное независимое подразделение с выходом на IPO.

Известна следующими сериями микросхем ПЛИС: MAX (CPLD); Cyclon I, II, III, IV — классические ПЛИС; Stratix — ПЛИС с высокоскоростными интерфейсами; Cyclon V — гетерогенные с ядрами ARM Cortex-A9; Топовые изделия: Cyclon 10, Arria 10, Stratix 10.

Основной тул: Intel Quartus Prime. Quartus Prime Lite Edition доступен для скачивания бесплатно, но требуется регистрация и получение лицензии.

Устаревший, но имеющий очень широкое хождение тул Altera Quartus II, для которого есть бесплатная версия — Quartus II Web. Если не ошибаюсь, Quartus II работает только с микросхемами, произведенными до 2015г.

3. **Microsemi Corporation** (бывшая Actel), в 2018г поглощенная компанией **Microchip Technology Inc.** Специализируется на радиационно-стойких ПЛИС для авиации и космоса, в связи с чем изделия этого производителя содержат небольшое количество логических блоков. Редкий гость в наших краях.

Известна следующими сериями микросхем: IGLOO, PolarFire и PolarFire SoC — содержит ядра RISC-V.

Основной тул: Libero SoC Design Suite

4. **Lattice Semiconductor, Inc**. Один из старейших производителей микросхем ПЛИС на рынке (основана в 1983г). Производит классические ПЛИС малой и средней плотности (до 100K LUs), низким энергопотреблением и со встроенной конфигурационной памятью.

Известна следующими сериями микросхем: iCE40, ECP5 и ECP5-5G, MachXO (CPLD). Все эти серии микросхем имеют широкое хождение среди студентов и любителей самоделок и на то есть особая причина — поддержка открытым тулчейном Yosys, но об этом позже.

Основной тул: Lattice Diamond. Предоставляет возможность запросить бесплатную краткосрочную лицензию для «образовательных целей».

5. **Efinix, Inc.** Относительно молодой производитель (основана в 2012 году) с китайскими корнями. Специализируется на микросхемах ПЛИС малой, средней и высокой плотности — 4K-500K LEs. Также редкая птица в наших краях.

Известна следующими сериями микросхем: Trion (T8) и Trion Titanium — содержит ядра RISC-V.

Основной тул: Efinity Software. Доступен бесплатно для скачивания, выдается лицензия на 1 год сопровождения (бесплатные апгрейды) при приобретении комплекта разработчика Xyloni Development Kit.

6. **GOWIN Semiconductors.** Еще один производитель с китайскими корнями. Специализируется на микросхемах ПЛИС малой и средней плотности, но со своей «изюминкой». В виду своей дешевизны и доступности с Aliexpress, изделия этого производителя имеют широкое хождение в среде самоделкиных. Популярность ПЛИС этого производителя растет. Доступен в России.

Известен следующими микросхемами: LittleBee GW1N (от 1K до 8K LUs), Arora GW2A-18 (20K LUs) и GW2A-55 (54K LUs). Эти ПЛИС также содержат большое количество блоковой статической памяти (BSRAM), а серия GW1NR содержит до 64 Mbits динамической (SDRAM) памяти и доступную пользователю Flash память. Поддерживаются открытым тулчейном Yosys.

Основной тул: GOWIN EDA. Доступен бесплатно, но требуется регистрация и получение лицензии. Доступно в России через посредника.

Под «классическими» я понимаю такие ПЛИС, которые внутри построены на регулярных макроблоках, структура которых приведена на рис. 10 или близкая к ней. Современные изделия от Xilinx и Altera сильно отошли от этой изящной структуры.

4. Синтез цифровых схем для ПЛИС

Не ошибусь если скажу, что появления микросхем программируемой логики PLD/CPLD и FPGA, а это середина-конец 1980-х, произошло в самый подходящий для этого момент. К этому времени уже существовали различные HDL языки; разработка микросхем с использованием HDL и верификация их дизайнов перед запуском производства широко входила в практику. Поэтому для конфигурирования программируемой логики почти сразу начали использовать HDL языки. Для этого был создан инструментарий, позволяющий выполнять «синтез» схем, т. е. преобразование схемы с языков HDL в набор битов конфигурации микросхемы ПЛИС, реализующий заданную схему.

Как же выглядит синтез цифровых схем для ПЛИС на практике? Вкратце, последовательность действий разработчика выглядит следующим образом:

Этап 0. Составление спецификации и эталонной модели

Как и при разработке программного обеспечения, разработка цифровых схем начинается с постановки задачи. Для этих целей проектировщик описывает свойства будущего изделия сначала в виде структурной диаграммы (рис. 13), которая постепенно, сверху вниз детализируется (принцип «top-down design») с использованием микроархитектурных диаграмм (см. пример на рис. 14). Из микроархитектурных диаграмм прослеживаются все присутствующие в изделии сигналы, регистры и логические блоки. По микроархитектурным диаграммам строятся временные диаграммы, описывающие все возможные состояния входных и выходных сигналов и их изменение с течением времени.

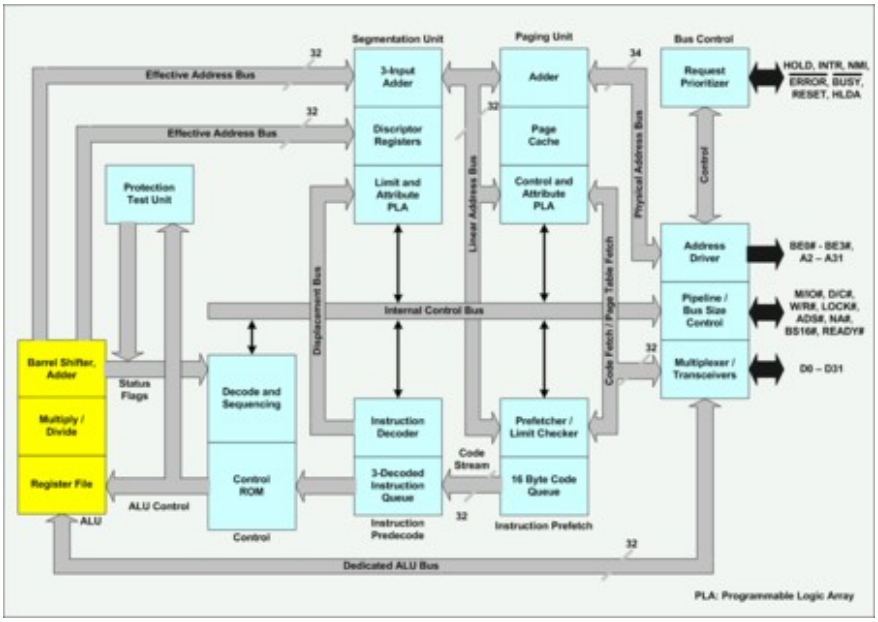


Рис. 13. Блочная диаграмма иллюстрирующая структуру микропроцессора Intel 80386DX.

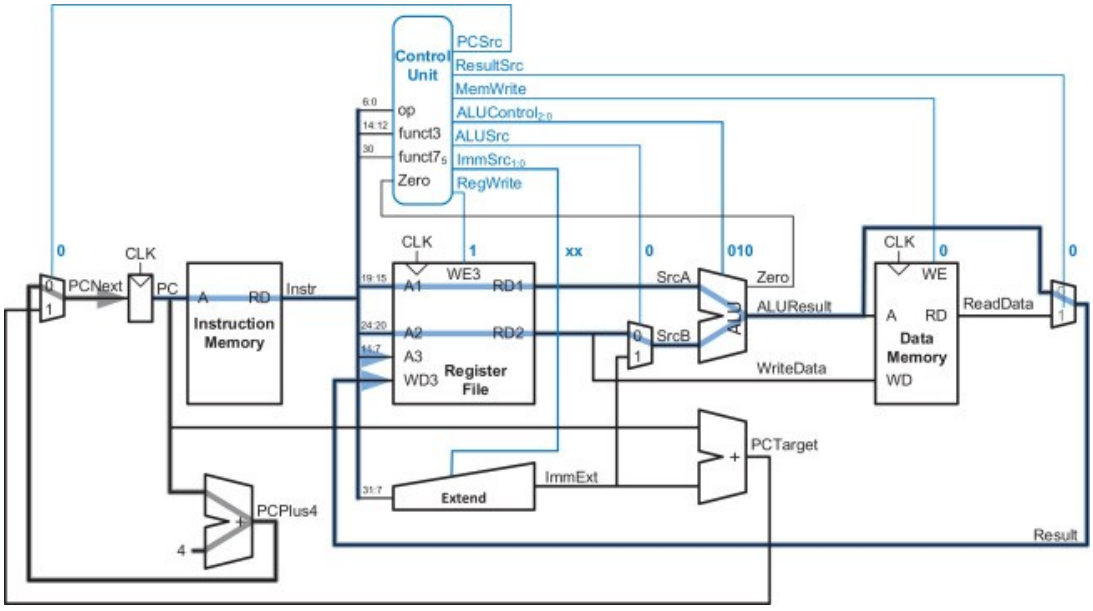


Рис 14. Микроархитектурная диаграмма однотактового микропроцессора RV32I. На микроархитектурной диаграмме просматриваются все сигналы, регистры и комбинационные блоки устройства.

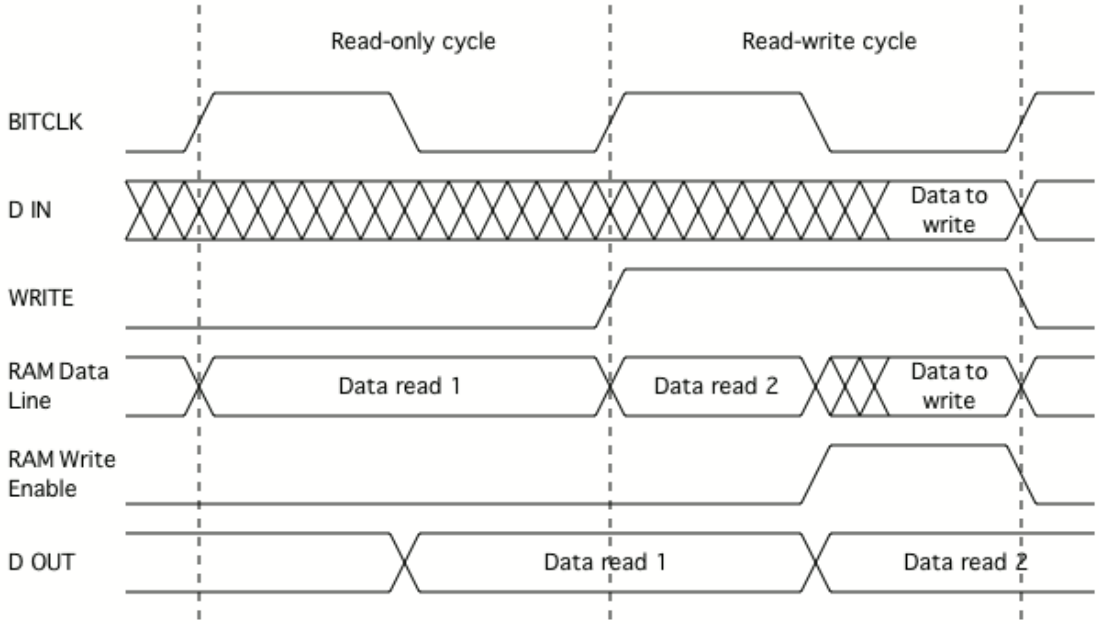


Рис 15. Пример временной диаграммы при доступе в память.

Когда готов набор микроархитектурных и соответствующих временных диаграмм, производят создание «эталонной модели» (golden reference model). Для этого часто используют обычные языки программирования или язык SystemC, который представляет собой набор классов для C++. Смысл этого этапа состоит в том, чтобы, во-первых промоделировать систему в целом и понять какие у неё могут быть свойства, какая теоретически достижимая производительность и вообще проверить идею. Во-вторых, чтобы использовать полученную эталонную модель далее для верификации дизайна.

Этап 1. Выбор ПЛИС

Имея на руках спецификации, диаграммы и модель будущего изделия, разработчик определяется с производителем и моделью ПЛИС, на которой он собирается решать поставленную задачу. В этом деле очень много субъективизма и личных предпочтений. Кто-то исторически привык к ПЛИС-ам Altera; кто-то считает, что изделия Xilinx — лучшие, так как микросхемы этого производителя на данный момент самые насыщенные по функционалу; кто-то довольствуется аскетизмом классических ПЛИС от Lattice или дешевизной GOWIN. Главное, чтобы ресурсов ПЛИС, т. е. число LEs, LUTs, блоков памяти и прочих «строительных блоков» имелось с хорошим (читай — двойным) запасом. А для того, чтобы как-то оценить количество требуемых ресурсов, нужны микроархитектурные диаграммы.

Совет от себя лично — для начинающих рекомендую обратить внимание на микросхемы ПЛИС Lattice iCE40 и ECP5, а также GOWIN GW1N и GW2A (LittleBee и Arora).

Этап 2. Приобретение средств разработки

Разработчик определяется с приобретением средств разработки (тулом) и его установки на рабочий ПК (или на сервер). Все современные тулы поддерживают ОС Windows и Linux и могут быть использованы как в интерактивном режиме (IDE), так и в пакетном (командная строка и набор скриптов). Параллельно с этим приобретается

отладочная плата (или несколько разных), содержащая выбранную модель микросхемы ПЛИС и устройство для её программирования — как правило, JTAG программатор. Многие отладочные платы сейчас уже идут со встроенным программатором с USB разъемом, что очень удобно. Если для Вашей модели ПЛИС вдруг не оказалось отладочной платы со встроенным программатором, то стоит приобрести отдельный JTAG адаптер, например USB Blaster — этот программатор (JTAG адаптер) поддерживается огромным спектром как коммерческих тулов, так и «опенсорсных» (СПО). Вообще, подойдет любой USB<->Serial адаптер на основе популярной микросхемы FTDI FT2232.

Этап 3. Составление плана распиновки

Изучается спецификация (даташит) на выбранную микросхему ПЛИС и создается описание её внешних выводов — каждому внешнему сигналу назначается один или несколько pad-ов (пинов). Это описание представляется в текстовом файле, обычно имеющего расширение **.lpf** или **.cst**. Для создания этого файла в коммерческих тулах имеются специальные средства, позволяющие выбрать модель микросхемы из списка поддерживаемых и визуально распределить сигналы по выводам корпуса микросхемы. Использование таких визуальных средств не всегда удобно, иногда проще создать и редактировать текстовый файл вручную, а параметры pad-ов подсмотреть в даташите на конкретную микросхему.

Ниже, для примера, я приведу небольшую выдержку LPF файла из своего проекта с микросхемой ПЛИС Lattice серии ECP5 (25K LEs). Файл karnix_hub12_cabga256.lpf, структуру которого мы рассмотрим более подробно чуть ниже, содержит примерно следующее:

```
FREQUENCY PORT "io_clk25" 25.0 MHz;
LOCATE COMP "io_clk25" SITE "B9";
IOBUF PORT "io_clk25" IO_TYPE=LVCMOS33;

LOCATE COMP "io_rgb[0]" SITE "A13";              # LED0
IOBUF PORT "io_rgb[0]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_rgb[1]" SITE "A14";              # LED1 - WORK
IOBUF PORT "io_rgb[1]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_rgb[2]" SITE "A15";              # LED2 - TEST
IOBUF PORT "io_rgb[2]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_rgb[3]" SITE "B14";              # LED3
IOBUF PORT "io_rgb[3]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_gpio[0]" SITE "L1";              # HUB_00
IOBUF PORT "io_gpio[0]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_gpio[0]" PULLMODE=NONE DRIVE=16;
LOCATE COMP "io_gpio[1]" SITE "L2";              # HUB_01
…
```

Этап 4. Написание кода

Разработчик пишет код цифровой схемы на языках HDL — традиционно на Verilog (SystemVerilog) или VHDL. Нетрадиционно — на Chisel, SpinalHDL или nMigen/AmaranthHDL. Существует приличное количество экзотических HDL языков, но почти все они в конечном счете сводятся к генерации кода на языках Verilog или VHDL.

Очень часто при разработке используются библиотечные или приобретенные функциональные блоки (IP блоки) — большие куски кода на HDL, которые требуют адаптации и интеграции в новый дизайн. Языки HDL позволяют создавать (описывать) блоки параметризованными, что сильно упрощает повторное их использование.

Этап 5. Верификация и симуляция

Код отлаживается, верифицируется и симулируется (реализуется потактово). Производится это либо встроенными средствами коммерческого тула от поставщика микросхем ПЛИС, либо приобретенным коммерческими тулами специального назначения, как-то ModelSim и Questa. Крупные компании по разработке микросхем часто используют свои доморощенные тулы для верификации и симуляции, как в примере с Intel 80386 — помните слова Кена Ширриффа про *sed*, *awk* и *grep* ? В академической среде, а так же среди самоделкиных и небольших компаний разработчиков имеют широкое хождение «опенсорсные» тулы, такие как Icarus Verilog и Verilator. Весь процесс разработки, верификации и симуляции итеративно повторяется до тех пор, пока не будет получен дизайн, удовлетворяющий требованиям спецификации или соответствующий «эталонной модели», в которой, кстати, тоже могут быть ошибки.

Следует заметить, что в микроэлектронной промышленности широко проповедуется принцип «Design for Verification» — некий свод правил для инженеров разработчиков и проектировщиков, обеспечивающий выход годной продукции с первого раза. Т.е. дизайн поступает на фабрику только тогда, когда пройдены все этапы верификации и симуляции и есть твердая уверенность в отсутствии ошибок. Причем, «фабы» тоже проверяют поступающие к ним дизайны на соответствие уже своим правилам — «Design Rule Check» (DRC check).

Этап 6. Синтез

Из полученного кода на HDL, средствами выбранного тула производится синтез т. н. netlist-а. Netlist — это файл (обычно формата EDIF, BDIF или JSON) описывающий представление разработанной цифровой схемы из «атомарных» блоков, которые присутствуют в выбранной микросхеме ПЛИС. Такие блоки обычно называют «Standard cells». Синтезированный netlist при желании можно визуализировать в виде цифровой схемы.

Этап 7. Оптимизация и STA

Далее производится оптимизация netlist-а — из него удаляются лишние блоки, типовые структуры комбинируются и замещаются на упрощенные/эквивалентные или на специальные «hard» блоки, специфичные для выбранной ПЛИС. Вся эта процедура выполняется автоматически специальными утилитами, входящими в состав тула. Существует множество алгоритмов оптимизации, они вызываются один за другим из скриптов, повторяются в цикле с разными параметрами и т. д. Опытные разработчики предпочитают писать свои скрипты для оптимизации своих дизайнов.

На этапе оптимизации netlist-а подключаются алгоритмы «Static Timing Analisys» (STA) которые достаточно точно позволяют рассчитать временные задержки распространения сигналов в синтезированной схеме, а также позволяют выполнить ряд оптимизаций по улучшению этого показателя (т. е. уменьшить задержки). Часто бывает так, что на этом этапе дальнейшая обработка netlist-а невозможна, так как в нём присутствуют логические цепочки, рассчитанное время распространения сигнала в которых превышает заданное при разработке схемы значение (нарушение спецификации). В этом случае разработчику приходится возвращаться и перепроектировать схему — переписывать код на HDL, применять другие решения, отказываясь от громоздких комбинационных схем или

оптимизировать их вручную разбивая регистрами (т. е. выполнять «конвейеризацию»), после чего повторять весь процесс верификации, симуляции, синтеза и оптимизации.

Этап 8. Расстановка и трассировка

Оптимизированный netlist поступает на вход следующей утилите, которая выполняет процесс расстановки блоков по местам и маршрутизацию сигнальных линий их связывающих, это т. н. процесс «place and route» (PnR или «плэйсер»). В этом процессе каждому блоку из netlist-а определяется фиксированное место внутри микросхемы ПЛИС, а блоки связываются между собой доступными средствами матрицы коммутации, к которой также подключаются внешние вывода (pad-ы) микросхемы и тактовые сигналы. На этом этапе происходит очень тяжелый и затратный по времени и ресурсам процесс оптимизации, но уже не структурной, а топологической — утилита PnR ищет оптимальное расположение блоков и их связей, так чтобы не нарушить дизайн и его временные характеристики.

Результатом работы утилиты плэйсера является топологическое представление схемы внутри ПЛИС (тоже файл формата EDIF или JSON). После работы плэйсера временны*е* характеристики дизайна, как правило, улучшаются, но бывает и наоборот — всё становится только хуже и разработчику приходится возвращаться к редизайну на HDL. Иногда проблемы с расстановкой можно решить изменением числа «рандомизации» (seed), так как алгоритмы плэйсера не являются строго детерминированными и часто основываются на случайных последовательностях. Т.е. покрутив «seed» несколько раз, можно попытаться добиться результата, удовлетворяющего требованиям STA, и это будет работать.

Надо отметить, что трассировка тактовых сигналов, как правило, производится отдельно от остальных цифровых сигналов, и для них в микросхемах ПЛИС предусматриваются специальные глобальные линии («global nets») с минимальными задержками и искажениями фронтов. Если в дизайне используются генерируемые внутри него тактовые сигналы, то такие сигналы требуют особого внимания — их требуется вывести на глобальные линии, для чего используется специальный синтаксис и специализированные «hard» блоки, предназначенные для этой цели. Также глобальными линиями принято трассировать сигналы сброса (resets) и некоторые общие для множества блоков схемы сигналы разрешения (enable signals). Иногда утилита синтеза автоматически распознает такие сигналы и маркирует их специальным образом, чтобы плэйсер на этапе размещения и трассировки использовал глобальные линии, но это не всегда работает верно и требует внимания разработчика.

Синтезатор, оптимизатор и плэйсер обычно сообщают в log-ах о том, какие решения были приняты в процессе и какие из сигналов и каким способом были трассированы. Также в log-ах могут появляться предупреждающие сообщения о сомнительных моментах дизайна, которые могут в конечном счете работать совсем не так, как это запланировал разработчик, поэтому log нужно внимательно читать и анализировать.

В больших и сложных дизайнах для FPGA (и особенно для ASIC) обычно присутствует еще один этап, предшествующий вызову плэйсера, который называется «floorplanning» и в процессе которого для плэйсера задают набор правил и ограничений. Например, разработчик может посчитать необходимым размещение каких-то отдельных блоков рядом друг с другом или возле выходных сигналов (возле pad-ов) или поближе к линиям питания. Но в большинстве случаев этот процесс планирования расстановки проистекает автоматически внутри плэйсера и дает приемлемый результат.

Этап 9. Генерация битстрима

Завершающим этапом синтеза является генерация «битстрима» — последовательности бит которую можно загрузить в микросхему ПЛИС. Для этого используется отдельная утилита в составе тула, она принимает на вход JSON файл от плэйсера и генерирует на выходе двоичный файл (обычно с расширением .bit, .bin или .fs) готовый к загрузке в ПЛИС. В большинстве случаев это чисто технический этап выполняемый автоматически и не требующий взаимодействия с разработчиком.

Этап 10. Загрузка в ПЛИС и проверка на железе

После того как разработчик получил «битстрим», его можно и нужно загрузить в NOR flash память на отладочной плате чтобы микросхема ПЛИС считала его при следующей инициализации. Производится это с помощью JTAG программатора (см. выше) и специальной утилиты. В коммерческих тулах такая утилита входит в поставку. Существуют и «опенсорсные» утилиты, например openFPGALoader.

Очень частно, при активной отладке дизайна и прогонке его «на железе», имеет смысла загружать битстрим не в NOR flash, а прямо в конфигурационные регистры SRAM микросхемы ПЛИС. Это позволяет, во-первых, экономить число операций записи во flash, во-вторых, это существенно быстрее и позволяет экономить время разработчику. Но тут надо не забывать о том, что при сбросе (инициализации), микросхема ПЛИС всегда считывает конфигурацию из NOR flash, т. е. заново перезапишет содержимое своей конфигурационной памяти, а загруженный в неё ранее битстрим будет потерян. Если разработчик случайно выпускает этот момент из вида, то это часто приводит к разного рода «необъяснимым» эффектам в поведении отлаживаемого дизайна. Поэтому, мой небольшой совет начинающим — если что-то идет не так, перезалейте Ваш битстрим в NOR flash и выполните сброс по питанию!

5. Yosys Open SYnthesis Suite

Про существование программируемой логики я лично узнал еще в конце 90-х, будучи студентом, но микросхему ПЛИС живьём увидел много лет спустя - в 2010 или где-то около того. Если не ошибаюсь, то была микросхема про-ва Altera серии Cyclon II на небольшой плате в комплекте с кварцевым генератором и стабилизаторами питания. Больше на плате ничего не было, все выводы ПЛИС были попросту выведены на штыревые разъёмы, расположенные по четырем сторонам платы. Разумеется, у меня возникло желание попробовать «запрограммировать» её, но я тут же столкнулся с рядом трудностей. Во-первых, требовался какой-то JTAG программатор, а эти устройства в то время стоили баснословных денег (и софт для них тоже). Во-вторых, всё сообщество российских (да и западных) разработчиков-плисоводов плотно сидело на ОС Windows и интенсивно ёрзало мышами по экрану. Для меня, человека, привыкшему работать в командной строке удаленно по SSH и люто ненавидящему не только «винду», но и весь проприетарный софт, это вызывало стойкое отвращение (разумеется, мне часто приходилось использовать «винду», но только лишь для того, чтобы запустить PuTTY). Я несколько раз выкачивал Quartus II под Linux и пытался устанавливать его на сервер разработки, но сталкивался с какими-то нерешаемыми трудностями — понять, как пользоваться этим мульти-гигабайтным тулом из shell-а было не просто, так как детальной инструкций никто не написал, а все форумы в то время были завалены инструкциями со скриншотами окошек. Короче, ни Quartus, ни Vivado, ни любой другой проприетарный тул я осилить так и не смог. А может быть, просто не захотел.

Зато в 2018 году я узнал про существование полностью open source решения для синтеза цифровых схем и это сильно прибавило мне энтузиазма. Интерес к теме программирования ПЛИС также подогревался начавшей бурно цвести темой RISC-V (по-русски читается как «риск-5», а не как вакцина от Ковида) — полностью открытой микропроцессорной архитектуре, вычислительные ядра которой уже можно было синтезировать для ПЛИС. Так я повторно начал погружаться в тему, но уже совсем с другим, нетрадиционным подходом.

Основная утилита в этом опенсорсном решении, Yosys, представляет собой тул для выполнения синтеза, т. е. преобразования описания цифровой схемы из текста на языке Verilog в netlist с последующим выполнением процедур оптимизации, используя библиотеку Berkeley ABC. Если быть более точным, то Yosys — это всего лишь фронтэнд, преобразующий текст на языке Verilog (позже добавили поддержку VHDL) в структуры, которые может обрабатывать библиотека ABC, а вся магия происходит именно в ней. Библиотека ABC появилась очень давно и уходит своими корнями в 1980-е годы. Всё это время основное применение библиотека ABC находила в академических застенках для моделирования, симуляции и верификации, и лишь появление утилиты Yosys в 2014 году сделало доступным всю мощь ABC для широких масс. Вместе Yosys, ABC, NextPNR и еще ряд других открытых проектов устроили настоящую революцию в индустрии синтеза цифровых схем в конце 2010-х годов, где преобладало засилье закрытых, проприетарных, бесконечно раздутых и заоблачно дорогих программных пакетов с входом «только для своих».

И Yosys это не только для ПЛИС. На данный момент на основе цепочки утилит, во главе которых находится Yosys, построен пакет открытых программ OpenLane/OpenROAD, предназначенный для автоматизации разработки и автоматической трассировки микросхем (ASIC), с помощью которого любой желающий может полностью выполнить дизайн цифровой микросхемы и даже бесплатно заказать её в производство в составе «Multi Project

Wafer». MPW — это способ производства, когда на одной кремниевой пластине («вафле») размещается несколько разных проектов с целью удешевления производства прототипов.

Интересно и то, что Yosys изначально это бакалаврская дипломная работа Клиффорда Вулфа, выполненная при работе над другим проектом.

Wol13] Wolf, Clifford: Design and Implementation of the Yosys Open SYnthesis Suite. 2013. – Bachelor Thesis, Vienna University of Technology.

Отличная документация, написанная Вулфом по утилите Yosys, находится тут.

Как я отметил выше, Yosys — это своего рода верхушка айсберга, под которой скрывается целая цепочка связанных воедино опенсорсных проектов с целью предоставить широкой общественности вариант законченного альтернативного решения для синтеза цифровых схем и не только. Ниже я попытаюсь кратко разобрать некоторые, самые важные, из них.

Arachne-pnr («паук»). Утилита-плэйсер и первая попытка реализовать полностью законченный опенсорсный тул синтеза для ПЛИС, автором является Коттон Сид. Сид связал проекты IceStorm и Yosys и дописал к ним недостающее звено — «плэйсер», в результате получил тул для синтеза под микросхемы ПЛИС Lattice iCE40. На данный момент «паук» больше не поддерживается, весь его функционал полностью перекрывается другим, более мощным инструментом — NextPNR.

IceStorm. Проект по документированию внутреннего устройства микросхемы ПЛИС Lattice iCE40. Проект выполнен Мэтиасом Лассером (Mathias Lasser) и всё тем же Клиффордом Вулфом в 2015 году.

Идея проекта следующая. Всем очень хочется иметь открытый тул для синтеза, но внутренняя структура микросхем ПЛИС тщательно скрывается их производителями «за семью патентами», что сильно препятствует созданию такого полезного тула. Не беда. Возьмем самую простую по структуре микросхему ПЛИС, возьмем проприетарный тул от производителя и будем синтезировать много «случайных» схем, а к полученному битстриму применим статистический анализ. Ну почти как арабский математик Аль-Кинди, который в 9-м веке придумал статистический метод взлома шифров. И это сработало! Первая микросхема ПЛИС, которую полностью документировали Мэт и Клифф, была iCE40 производства Lattice Semiconductors.

С результатами этой ошеломляющей работы Клифф выступил на регулярном съезде Der Chaos Computer Club (видео). В своем выступлении Клифф демонстрировал слайды, в которых сравнивал качество и скорость синтеза созданного ими инструмента с проприетарным. Результат потрясающий — несмотря на некоторое отставание в оптимальности использования ресурсов микросхемы, опенсорсный тул работал в разы быстрее проприетарного!

Чуть позже в рамках проекта Project Trellis была полностью документирована микросхема ПЛИС Lattice ECP5 со всеми её «hard» блоками. А это очень интересная по своим возможностям микросхема, позволяющая синтезировать в ней ядро RISC-V с MMU и гонять на этом ядре настоящий Linux.

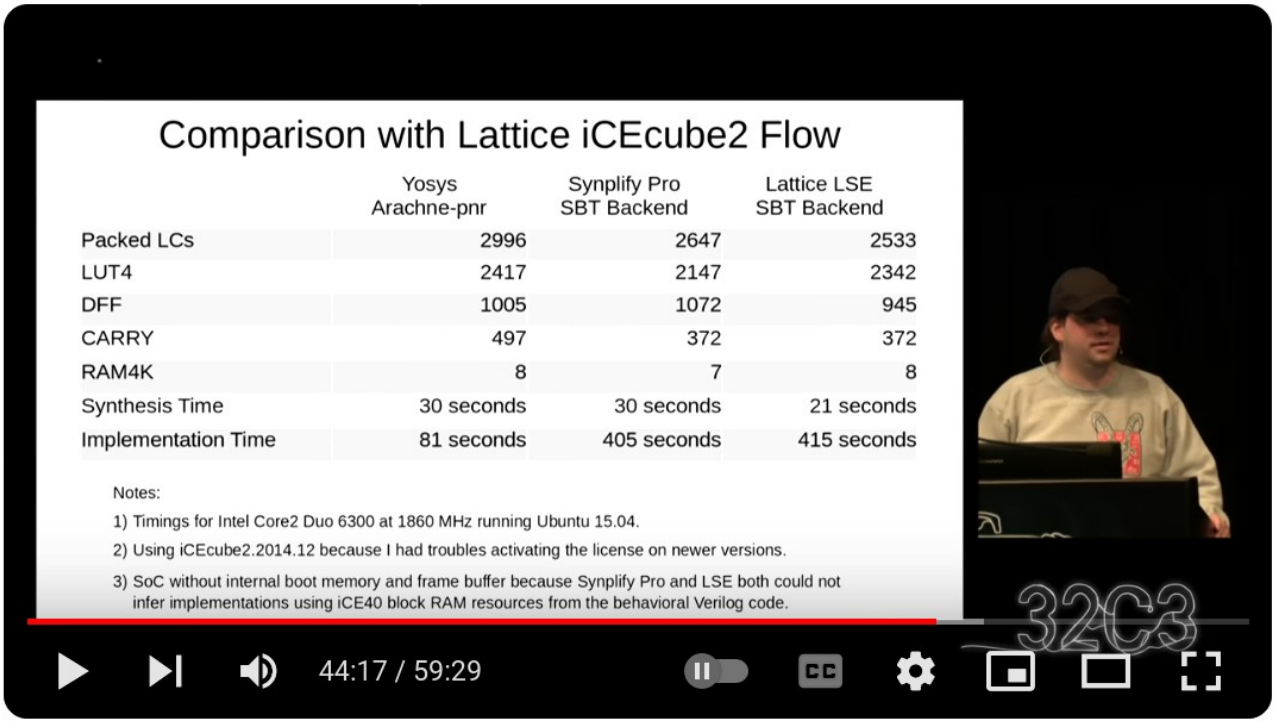


Рис. 16. Фрагмент выступления Клиффорда Вулфа с демонстрацией эффективности созданного им инструмента для синтеза под ПЛИС Lattice iCE40. На слайде «Implementation time» - время выполнения трассировки.

Разумеется, идею Клиффа и Мэта тут же подхватило сообщество, и процесс не пошел,

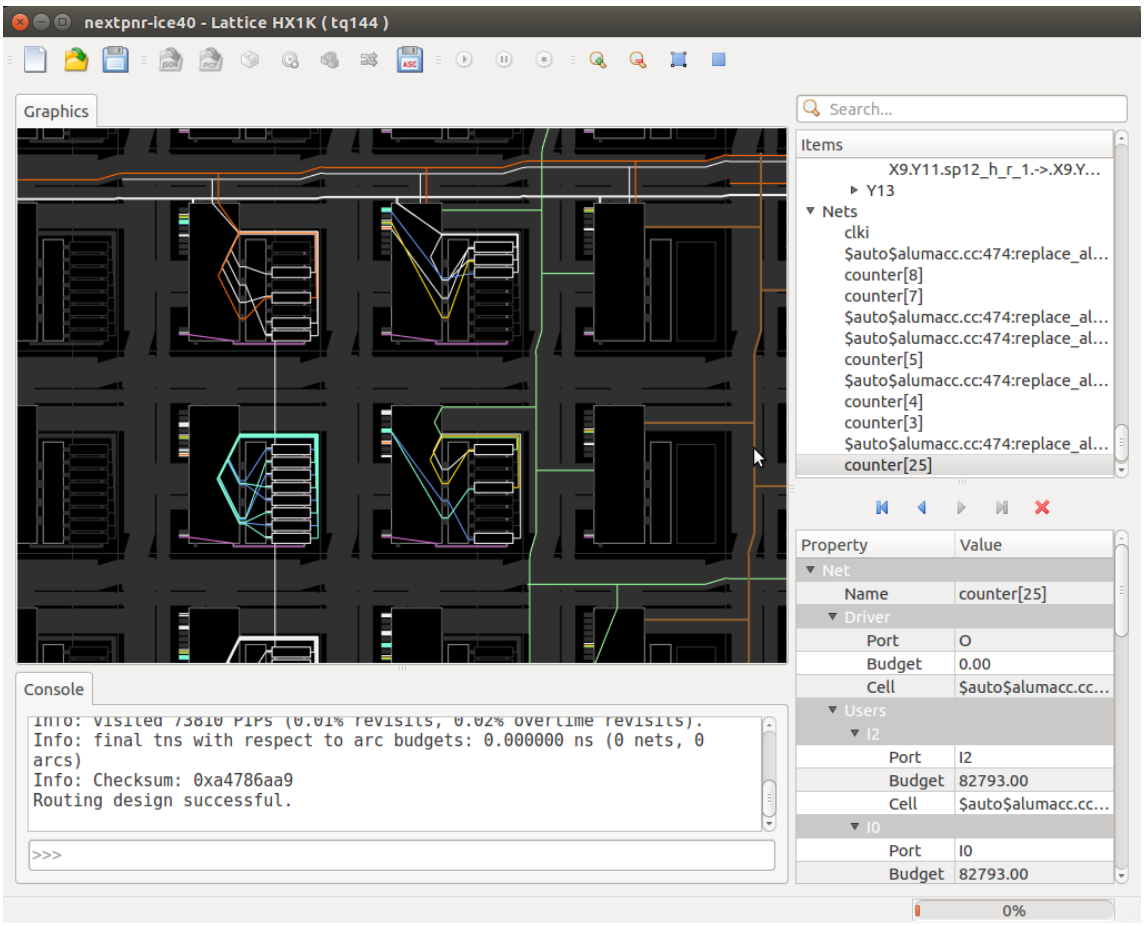


Рис. 17. Визуализация результата работы утилиты NextPNR при синтезе для микросхемы ПЛИС Lattice iCE40HX1K.

openFPGALoader — универсальная утилита-программатор для загрузки битстрима. Поддерживает огромное количество устройств-программаторов, микросхем ПЛИС и ПЛУ (не только тех, что присутствуют в ChipDB). Имеет возможность гибкой подстройки сигнальных линий JTAG, позволяя использовать программаторы, даже если их нет в списке поддерживаемых. Позволяет загружать битстрим как в NOR flash, с которой работает ПЛИС, так и в RAM самой ПЛИС. Может быть использована для прошивки большого количества различных моделей микросхем NOR flash памяти. Утилита openFPGALoader уже присутствует в репозиториях пакетов почти всех известных дистрибутивов Linux и *BSD.

На данный момент существует над-проект YosysHQ который покрывает все вопросы синтеза, верификации, симуляции, трассировки и создания документации для микросхем ПЛИС с использованием решений с открытым исходным кодом. Авторы и активные коммиттеры этого проекта организовали свою компанию (YosysHQ GmbH) и предлагают услуги консалтинга, а также ряд коммерческих продуктов на основе созданных ими опенсорсных решений.

6. Как установить утилиты тулчейна Yosys

Самый простой способ — это скачать и установить готовый пакет бинарных программ «OSS CAD Suite» подготовленный разработчиками из YosysHQ, который оформлен двумя способами: в виде docker контейнера и в виде тарболла. Бинарный пакет собирается каждую

ночь (nightly builds) и выкладывается в репозиторий, поэтому тут нужно быть осторожным, чтобы случайно не попасть на «сломанную» версию пакета (если что-то пошло не так, то нужно попробовать более раннюю версию пакета). На данный момент «еженощно» формируются билды для нескольких платформ, вот выдержка из README репозитория:

> ***linux-x64***
>
> *Any personal Linux based computer should just work, no additional packages are needed to be installed on system to make OSS CAD Suite working. Distributed libraries are based on Ubuntu 20.04, but everything is packaged in such a way so it can be used on any Linux distribution.*
>
> ***darwin-x64***
>
> *Any macOS 10.14 or later with Intel CPU should use this distribution package.*
>
> ***darwin-arm64***
>
> *Any macOS 11.00 or later with M1 CPU should use this distribution package.*
>
> ***windows-x64***
>
> *This architecture is supported for Windows 10 and 11, but older 64-bit version of Windows 7, 8 or 8.1 should work.*
>
> ***linux-arm***
>
> *ARM based Linux devices such as Raspberry Pi 3, 4 or 400 can use this distribution package.*
>
> ***linux-arm64***
>
> *ARM64 based Linux devices using 64bit CPU as in Raspberry Pi 4 and 400 (with 64bit version of OS installed), and also laptops like the MNT Reform 2 can use this distribution package.*
>
> ***linux-riscv64***
>
> *RiscV-64 based Linux devices should use this distribtuion package, but please note that this is currently* ***untested***

Установка через docker, пожалуй, наиболее простой вариант, так как у YosysHQ имеется свой «Hub» на сайте Docker-a, на котором выкладываются docker файлы для всех имеющихся вариантов пакета. Это значит, что выкачать и установить контейнер, собранный для платформы **linux-x64**, можно командой вида:

```
$ docker pull yosyshq/cross-linux-x64
```

Установить из тарболла тоже не сложно. Достаточно сходить в репозиторий по ссылке:

`https://github.com/YosysHQ/oss-cad-suite-build/releases/latest`

и скачать требуемую версию пакета (.tgz файл объемом около 500МБ). После чего распаковать скачанный пакет в домашний каталог или любое другое удобное место, прописать пути к исполняемым файлам в $PATH и подготовить окружение:

Для Linux и MacOS:

```
export PATH="<extracted_location>/oss-cad-suite/bin:$PATH"
```

Для FreeBSD:

```
setenv PATH "<extracted_location>/oss-cad-suite/bin:$PATH"
```

далее подготовить переменные окружения специальным скриптом:

```
source <extracted_location>/oss-cad-suite/environment
```

Для пользователей Windows:

из текущей оболочки:

```
<extracted_location>\oss-cad-suite\environment.bat
```

или создать новое окно с оболочкой:

```
<extracted_location>\oss-cad-suite\start.bat
```

Те, кому религия не позволяет использовать готовые бинарные пакеты, а я придерживаюсь именно этого течения, могут попробовать собрать весь стек утилит (или какую-то его часть) из исходных кодов. Мне удавалось успешно собирать почти весь стек под ОС ALT Linux p10 (x64) и под ОС FreeBSD 13.2 (amd64). Но это тема отдельного, очень длинного разговора.

После того как Вы установили Yosys тулчейн любым из способов, необходимо проверить, запускаются ли у Вас соответствующие утилиты:

```
rz@devbox:~$ yosys -V
Yosys 0.36+58 (git sha1 ea7818d31, gcc 7.5.0-3ubuntu1~18.04 -fPIC -Os)

rz@devbox:~$ openFPGALoader -V
openFPGALoader v0.11.0
```

Для работы с ПЛИС Lattice ECP5 потребуются следующие утилиты:

```
rz@devbox:~$ nextpnr-ecp5 -V
"nextpnr-ecp5" -- Next Generation Place and Route (Version nextpnr-0.6-157-g4a402519)

rz@devbox:~$ ecppack --version
Project Trellis ecppack Version 1.2.1-14-g488f4e7

rz@devbox:~$ ecpbram
Project Trellis - Open Source Tools for ECP5 FPGAs
ecpbram: ECP5 BRAM content initialization tool

rz@devbox:~$ ecpprog --help
```

```
Simple programming tool for FTDI-based Lattice ECP JTAG programmers.
Usage: ecpprog [-b|-n|-c] <input file>
       ecpprog -r|-R<bytes> <output file>
       ecpprog -S <input file>
       ecpprog -t
```

Для работы с ПЛИС Lattice iCE40 потребуются следующие утилиты:

```
rz@devbox:~$ nextpnr-ice40 -V
"nextpnr-ice40" -- Next Generation Place and Route (Version nextpnr-0.6-157-g4a402519)

rz@devbox:~$ icepack -h

Usage: icepack [options] [input-file [output-file]]

rz@devbox:~$ icebram

Usage: icebram [options] <from_hexfile> <to_hexfile>
       icebram [options] -g [-s <seed>] <width> <depth>

Replace BRAM initialization data in a .asc file. This can be used
for example to replace firmware images without re-running synthesis
and place&route.

rz@devbox:~$ iceprog --help
Simple programming tool for FTDI-based Lattice iCE programmers.
Usage: iceprog [-b|-n|-c] <input file>
       iceprog -r|-R<bytes> <output file>
       iceprog -S <input file>
       iceprog -t
```

Для работы с ПЛИС GOWIN LittleBee и Arora потребуются следующие утилиты:

```
rz@devbox:~$ nextpnr-gowin -V
"nextpnr-gowin" -- Next Generation Place and Route (Version nextpnr-0.6-157-g4a402519)

rz@devbox:~$ gowin_bba -h
usage: gowin_bba [-h] -d DEVICE [-i CONSTIDS] [-o OUTPUT]

Make Gowin BBA

rz@devbox:~$ gowin_pack -h
usage: gowin_pack [-h] -d DEVICE [-o OUTPUT] [-c] [-s CST] [--allow_pinless_io] [--
jtag_as_gpio] [--sspi_as_gpio] [--mspi_as_gpio]
                  [--ready_as_gpio] [--done_as_gpio] [--reconfign_as_gpio]

Pack Gowin bitstream
```

Также для работы Вам потребуется ряд вспомогательных утилит:

```
rz@devbox:~$ make -v
GNU Make 4.1
Built for x86_64-pc-linux-gnu

rz@devbox:~$ netlistsvg --version
1.0.2

rz@devbox:~$ xdot -h
usage: xdot [-h] [-f FILTER] [-n] [-g GEOMETRY] [--hide-toolbar] [file]
xdot.py is an interactive viewer for graphs written in Graphviz's dot language.
```

Для проверки синтаксиса, верификации и симуляции Вам потребуются:

```
rz@devbox:~$ iverilog -V
Icarus Verilog version 10.1 (stable) ()

rz@devbox:~$ gtkwave -V
GTKWave Analyzer v3.3.117 (w)1999-2023 BSI
```

```
rz@devbox:~$ verilator --version
Verilator rev v5.020-157-g2b4852048
```

Ну и последний, но важный момент. Чтобы утилита **openFPGALoader** могла иметь доступ к USB программатору в операционной системе Linux, требуется добавить набор правил для **systemd**. Для этого необходимо скачать файл с правилами https://github.com/trabucayre/openFPGALoader/blob/master/99-openfpgaloader.rules, поместить его в каталог **/etc/udev/rules.d/** и перезагрузить операционную систему. Выполнять это нужно от имени суперпользователя (root-а).

7. Пример использования открытого тулчейна Yosys

Настало время попробовать Yosys в деле. По законам жанра здесь должен был бы следовать пример а-ля «hello, world», то есть мигание светодиодом или что-то в этом роде. Но я хотел бы направить читателя несколько другим путём и познакомить с очень интересным проектом по обучению широких масс основам цифрового синтеза и верификации. Этот проект так и называется «Школа синтеза цифровых схем», его идейным вдохновителем является Юрий Панчул (@YuriPanchul), наш соотечественник, который несколько десятков лет трудится в именитых электронных компаниях в «Кремниевой Долине» и имеет огромный опыт в этой сфере. У Юрия есть свой блог на Habr-е, где он регулярно выкладывает тематические статьи и посты о том, как устроена эта индустрия, какие в ней превалируют тенденции и какие актуальные проблемы присутствуют.

Со слов Юрия, по долгу службы ему часто приходится проводить собеседования молодых специалистов, претендующих на должность «digital hardware engineer» и в какой-то момент у него родилась идея сделать свой набор небольших лабораторных работ на языке SystemVerilog, чтобы использовать их, во-первых, на «собесе» для проверки знаний, а во-вторых — для повышения квалификации своих коллег из смежных ведомств. Этот набор лабораторных работ называется «**basics-graphics-music**» и он свободно доступен в репозитории Юрия на Github-е по ссылке: https://github.com/yuri-panchul/basics-graphics-music. Этот же набор лабораторных работ активно используется в «Школе синтеза».

Сам проект «basics-graphics-music» устроен так, что позволяет его использовать с большим спектром тулов (Quartus, Vivado, Gowin EDA), а также с большим количеством отладочных и учебных плат, содержащих микросхемы ПЛИС, и он по праву может называться первым портабельным (portable) репозиторием обучающих примеров для программирования ПЛИС. Для поддержки проект имеется список рассылки: **portable-hdl-examples@googlegroups.com**.

Еще из интересных особенностей данного сборника лабораторных работ можно отметить их идеологическую выверенность стиля кода и последовательность подачи материала. Лабораторные работы начинаются с демонстрации самых базовых принципов комбинационной логики (правила де Моргана) и описания машин-состояния (FSM) и доходят до весьма сложных (но небольших по объёму кода) примеров работы с VGA графикой, синтезированием и детектированием музыки.

Недавно в проект была добавлена поддержка тулчейна Yosys для нескольких плат с микросхемами ПЛИС Lattice и Gowin — работа, к которой я приложил свою руку.

Среди множества отладочных плат, поддерживаемых проектом «basics-graphics-music» присутствует, в том числе, разработанная мной и моими коллегами из ООО «Фабмикро» плата «ПИР СЦХ-254 Карно» («Karnaugh Interactive Extendable ASIC Simulation Board»)

предназначенная для обучения азам цифрового синтеза и экспериментов с синтезируемыми микропроцессорными ядрами RISC-V.

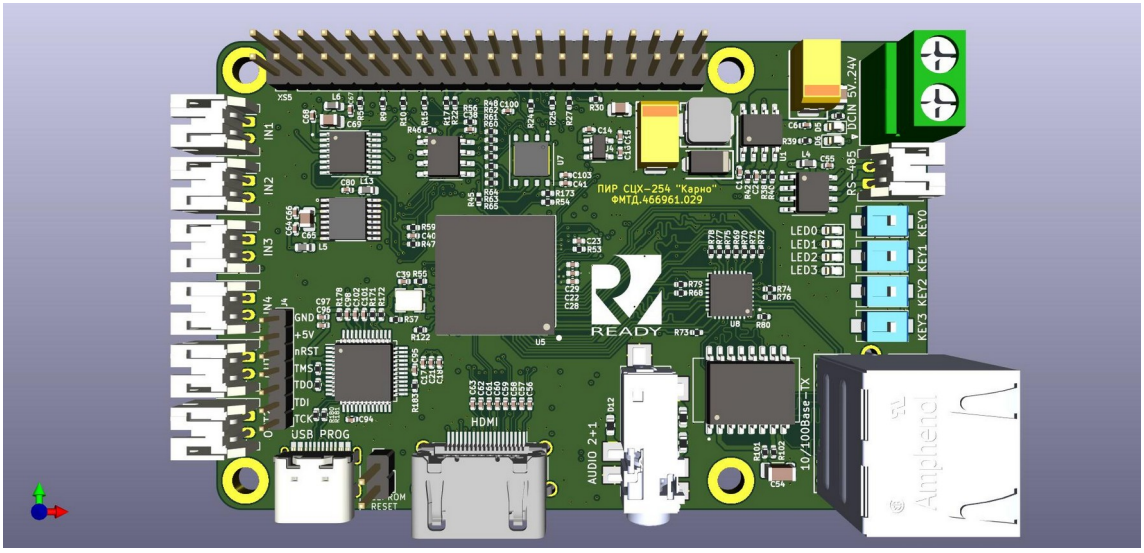


Рис. 18. Плата «Карно» - интерактивная расширяемая для синтезирования цифровых схем. 3D модель. Выполнена в KiCAD 7. OSHW проект.

Это полностью «open source and open hardware» проект на базе ПЛИС Lattice ECP5 с 25K логических блоков, содержит ряд интересных периферийных устройств: многоканальный ЦАП и АЦП, блок статической SRAM памяти объемом 512КБ, FastEthernet, HDMI, кнопки и светодиоды и кое-что еще. Плата «Карно» имеет встроенный JTAG программатор.

Плата выполнена в САПР KiCAD 7 и весь проект со схемой, трассировкой платы и Gerber файлами, доступен для скачивания с Github-а по ссылке: https://github.com/Fabmicro-LLC/Karnix_ASB-254 . Для демонстрации возможностей платы «Карно» я записал и выложил на YouTube новогоднее видео ссылка на которе дана в начале этой стетьи.

Далее я буду приводить примеры работы с тулчейном Yosys с её участием. Но как я отмечал выше, выбор платы не сильно принципиален, так как проектом «basics-graphics-music» поддерживаются ряд других, легко доступных отладочных плат, среди которых можно отметить OrangeCrab и TangNano-9K.

7.1. Установка и настройка «basics-graphics-music»

Итак, приступим к рассмотрению нескольких лабораторных работ. Клонируем репозиторий «basics-graphics-music» в любой удобный рабочий подкаталог:

```
rz@devbox:~$ git clone https://github.com/yuri-panchul/basics-graphics-music.git
Cloning into 'basics-graphics-music'...

rz@devbox:~$ cd  basics-graphics-music
rz@devbox:~/basics-graphics-music$
```

Произведем настройку среды синтеза для нашей платы, для этого запустим скрипт **check_setup_and_choose_fpga_board.bash** следующим образом:

```
rz@devbox:~/basics-graphics-music$ bash check_setup_and_choose_fpga_board.bash

check_setup_and_choose_fpga_board.bash: The currently selected FPGA board: karnix_ecp5_yosys. Please
select an FPGA board amoung the following supported:
 1) alinx_ax4010                      25) karnix_ecp5_yosys
 2) arty_a7                           26) nexys4
 3) arty_a7_pmod_mic3                 27) nexys4_ddr
 4) basys3                            28) nexys_a7
 5) c5gx_audio                        29) omdazz
 6) c5gx_vga666                       30) omdazz_pmod_mic3
 7) c5gx_vga_pmod                     31) orangecrab_ecp5_yosys
 8) cmod_a7                           32) piswords6
 9) de0                               33) qmtech_c4_starter
10) de0_cv                            34) rzrd
11) de0_nano_soc_vga666               35) rzrd_pmod_mic3
12) de0_nano_soc_vga_pmod             36) saylinx
13) de0_nano_vga666                   37) saylinx_pmod_mic3
14) de0_nano_vga_pmod                 38) tang_nano_20k
15) de10_lite                         39) tang_nano_9k
16) de10_lite_tm1638_arduino          40) tang_nano_9k_gowin_yosys
17) de10_nano_vga666                  41) tang_primer_20k_dock
18) de10_nano_vga_mister              42) tang_primer_20k_dock_alt
19) de10_nano_vga_pmod                43) tang_primer_20k_dock_gowin_yosys
20) de1_soc                           44) tang_primer_20k_lite
21) de2_115                           45) zeowaa
22) dk_dev_3c120n                     46) zeowaa_wo_dig_0
23) emooc_cc                          47) zybo_z7
24) ice40hx8k_evb_yosys               48) exit
Your choice (a number): 25
```

Выберем плату «karnix_ecp5_yosys», для этого введем её порядковый номер из списка (в данном случае — 25) и нажмем «ввод», скрипт продолжит выполнение и выдаст следующие сообщения:

```
check_setup_and_choose_fpga_board.bash: FPGA board selected: karnix_ecp5_yosys

check_setup_and_choose_fpga_board.bash: Created an FPGA board selection file: "/home/rz/basics-
graphics-music/fpga_board_selection"
Would you like to create the new run directories for the synthesis of all labs in the package, based
on your FPGA board selection? We recommend to do this if you plan to work with Quartus GUI rather
than with the synthesis scripts. [y/n]
```

Я нажму **n**, так как не использую Quartus. Скрип завершит работу с сообщением:

```
Configuring for Lattice ECP5...
OK
rz@devbox:~/basics-graphics-music$
```

Настройка среды готова. Посмотрим какие лабораторные работы нам доступны:

```
rz@devbox:~/basics-graphics-music$ ls labs
01_and_or_not_xor_de_morgan  08_shift_register     15_uart                           22_syst_ws
02_mux                       09_7segment_word      16_round_robin_arbiter            23_i2s_synthesizer
03_decoder                   10_hex_counter        17_geiger_muller_radiation_counter 24_i2s_music
04_priority_encoder          11_note_recognition   18_pow5_single_cycle              50_crash_course
05_7seven_segment_letter     12_snail_fsm          19_pow5_pipelined                 90_valid_ready_etc_preview
06_vga                       13_music_recognition  20_pow5_pipelined_valid           91_systemverilog_homework_preview
07_binary_counter            14_game               21_pow5_pipelined_valid_solution  common
```

7.2 Лабораторная работа «01_and_or_not_xor_de_morgan»

Данная лабораторная работа показывает, как выглядят цифровые схемы, построенные только на комбинационной логике, выраженной на языке SystemVerilog.

Комбинационные схемы — это асинхронные (нетактируемые) схемы состоящие только из логических элементов, таких как «И», «ИЛИ», «НЕ», «исключающее ИЛИ» и т. д.

Такие схемы подчиняются законам булевой алгебры, в том числе поддаются различным преобразованиям и упрощениям, к ним применимы правила де Моргана.

Зайдем в каталог с лабораторной работой и посмотрим, что нам предлагается:

```
rz@devbox:~/basics-graphics-music$ cd labs/01_and_or_not_xor_de_morgan

rz@devbox:~/basics-graphics-music/labs/01_and_or_not_xor_de_morgan$ ls -l

-rwxrwxr-x 1 rz rz  495 Jan  5 22:39 01_clean.bash
-rwxrwxr-x 1 rz rz  495 Jan  5 22:39 02_simulate_rtl.bash
-rwxrwxr-x 1 rz rz  495 Jan  5 22:39 03_synthesize_for_fpga.bash
-rwxrwxr-x 1 rz rz  495 Jan  5 22:39 04_configure_fpga.bash
-rwxrwxr-x 1 rz rz  495 Jan  5 22:39 05_run_gui_for_fpga_synthesis.bash
-rwxrwxr-x 1 rz rz  495 Jan  5 22:39 06_choose_another_fpga_board.bash
-rwxrwxr-x 1 rz rz  495 Jan  5 22:39 07_synthesize_for_asic.bash
-rwxrwxr-x 1 rz rz  495 Jan  5 22:39 08_visualize_asic_synthesis_results_1.bash
-rwxrwxr-x 1 rz rz  495 Jan  5 22:39 09_visualize_asic_synthesis_results_2.bash
drwxrwxr-x 2 rz rz 4096 Jan 11 22:54 run
-rw-rw-r-- 1 rz rz 1142 Jan  5 22:39 tb.sv
-rw-rw-r-- 1 rz rz 2614 Jan  5 22:39 top.sv
```

Каждый каталог с лабораторной работой содержит файлы: **top.sv** и **tb.sv** (расширение **.sv** означает текст на SystemVerilog), а также стандартный набор bash скриптов для выполнения операций синтеза (для ПЛИС и ASIC), запуска среды IDE и визуализации результатов. Далее я продемонстрирую только процесс синтеза и загрузки в ПЛИС. Но прежде давайте заглянем в top.sv.

```
rz@devbox:~/basics-graphics-music/labs/01_and_or_not_xor_de_morgan$ cat top.sv
```

В самом начале файла мы видим описание интерфейса модуля **top**, который предоставляет нам окно во внешний мир — доступ к устройствам на плате, среди которых массив светодиодов, массив кнопок, порт UART и ряд других. Интерфейс модуля top общий для всех лабораторных работ в этом проекте, но отдельные примеры (лабы) могут использовать только часть предоставляемых возможностей.

```
`include "config.svh"

module top
# (
    parameter clk_mhz = 50,
              w_key   = 4,
              w_sw    = 8,
              w_led   = 8,
              w_digit = 8,
              w_gpio  = 100
)
(
    input                         clk,
    input                         slow_clk,
    input                         rst,

    // Keys, switches, LEDs

    input        [w_key   - 1:0] key,
    input        [w_sw    - 1:0] sw,
    output logic [w_led   - 1:0] led,

    // A dynamic seven-segment display

    output logic [          7:0] abcdefgh,
    output logic [w_digit - 1:0] digit,

    // VGA

    output logic                 vsync,
    output logic                 hsync,
    output logic [          3:0] red,
    output logic [          3:0] green,
    output logic [          3:0] blue,

    input                        uart_rx,
    output                       uart_tx,

    input                        mic_ready,
    input        [         23:0] mic,
```

```
    output       [         15:0] sound,

    // General-purpose Input/Output

    inout        [w_gpio  - 1:0] gpio
);
```

Далее следует установка значений по умолчанию для всех внешних сигналов, не используемых в данном примере:

```
    // assign led      = '0;
       assign abcdefgh = '0;
       assign digit    = '0;
       assign vsync    = '0;
       assign hsync    = '0;
       assign red      = '0;
       assign green    = '0;
       assign blue     = '0;
       assign sound    = '0;
       assign uart_tx  = '1;
```

Затем следует сам пример, разбираемый в рамках выполнения данной лабораторной работы. В данном случае нам демонстрируют, как описываются два входных однобитных сигнала **a** и **b**, которые привязываются к кнопкам **key[]** (многобитный сигнал) и однобитный сигнал результата **result**. Между сигналами **a**, **b** и **result** устанавливается однозначная логическая связь — логическая операция «Исключающее ИЛИ» (XOR). Выходной сигнал **result** привязывается к светодиоду **led[0]**. К светодиоду **led[1]** привязывается результат этой же логической операции, но иным способом — минуя промежуточные сигналы **a** и **b**.

```
    wire a = key [0];
    wire b = key [1];

    wire result = a ^ b;

    assign led [0] = result;

    assign led [1] = key [0] ^ key [1];
```

Далее следуют упражнения для самостоятельного выполнения, в данном случае на знание логических операций и правил де Моргана:

```
    // Exercise 1: Change the code below.
    // Assign to led [2] the result of AND operation.
    //
    // If led [2] is not available on your board,
    // comment out the code above and reuse led [0].

    // assign led [2] =

    // Exercise 2: Change the code below.
    // Assign to led [3] the result of XOR operation
    // without using "^" operation.
    // Use only operations "&", "|", "~" and parenthesis, "(" and ")".

    // assign led [3] =

    // Exercise 3: Create an illustration to De Morgan's laws:
    //
    // ~ (a & b) == ~ a | ~ b
    // ~ (a | b) == ~ a & ~ b
```

Попробуем синтезировать схему и загрузить её в ПЛИС. Запуск всего конвейера для получения битстрима осуществляется одной командой:

```
rz@devbox:~/basics-graphics-music/labs/01_and_or_not_xor_de_morgan$ bash 03_synthesize_for_fpga.bash

Configuring for Lattice ECP5...
OK
```

Далее последует длинный, длинный лог результата работы всех утилит участвующих в процессе.

В самом конце, если все прошло удачно и без ошибок, мы увидим следующее:

```
Info: Program finished normally.
ecppack --compress --freq 38.8 --input 01_and_or_not_xor_de_morgan_out.config --bit
01_and_or_not_xor_de_morgan.bin

Would you like to upload to FPGA ? [y/N]
```

Нас спрашивают, хотим ли мы загрузить полученный битстрим в ПЛИС прямо сейчас, на что я отвечаю **N** (нет). Скрипт заканчивает свою работу сообщив нам имя файла битстрима:

```
Resulting binary file is:
-rw-rw-r-- 1 rz rz 99064 Jan 12 23:49
/home/rz/basics-graphics-music/labs/01_and_or_not_xor_de_morgan/run/01_and_or_not_xor_de_morgan.bin
```

7.3 Загрузка битстрима в микросхему ПЛИС

Настало время подключить нашу отладочную плату «Карно» к машине, на которой мы ведем разработку и запустить скрипт для прошивки:

```
rz@devbox:~/basics-graphics-music/labs/01_and_or_not_xor_de_morgan$ bash 04_configure_fpga.bash
Configuring for Lattice ECP5...
OK

Would you like to upload to FPGA ? [y/N] y
```

Скрипт спрашивает, действительно ли мы желаем выполнить загрузку в ПЛИС? Отвечаем **y**.

```
This board supports the following upload methods:
[o] for openFPGALoader using JTAG based on FTDI FT2232
[e] for ecpprog using JTAG based on FTDI FT2232
Which upload method do you prefer ? o
```

Скрипт просит выбрать, каким программатором производить загрузку битстрима, **openFPGALoader** или **ecpprog**. Выбираем первый, т. е. вводим **o.**

```
Upload method for this board is: openloader

Where to upload, SRAM or Flash ? [s/F] s
```

Далее скрипт спрашивает, будем ли мы загружать битстрим в SRAM или в NOR Flash. В данном случае выбираем SRAM, т. е. вводим: **s**

Если плата подключена и USB программатор на ней детектировался системой, то мы получим следующий вывод от утилиты **openFPGALoader**:

```
make upload_openloader
make[1]: Entering directory '/home/rz/basics-graphics-music/labs/01_and_or_not_xor_de_morgan/run'
openFPGALoader -v --ftdi-channel 0  01_and_or_not_xor_de_morgan.bin

No cable or board specified: using direct ft2232 interface
Jtag frequency : requested 6.00MHz   -> real 6.00MHz
found 1 devices
index 0:
        idcode 0x1111043
        manufacturer lattice
        family ECP5
        model  LFE5UM-25
        irlength 8
File type : bin
Open file: DONE
Parse file: DONE
bitstream header infos
Part: LFE5U-25F-6CABGA256
idcode: 41111043
IDCode : 41111043
```

```
displayReadReg
        Config Target Selection : 0
        Done Flag
        Std PreAmble
        No err
Enable configuration: DONE
SRAM erase: DONE
Loading: [==================================================] 100.00%
Done
userCode: 00000000
Disable configuration: DONE
displayReadReg и
        Config Target Selection : 0
        Done Flag
        Std PreAmble
        No err
```

Сразу по окончанию работы утилиты **openFPGALoader**, микросхема ПЛИС на плате будет сконфигурирована и Ваш дизайн (лабораторная работа) активирован в ней.

Для того чтобы проверить, как работает дизайн, нажмем кнопки KEY0 и KEY1 сначала по отдельности, потом вместе. Светодиоды LED0 и LED1 должны реагировать соответственно дизайну.

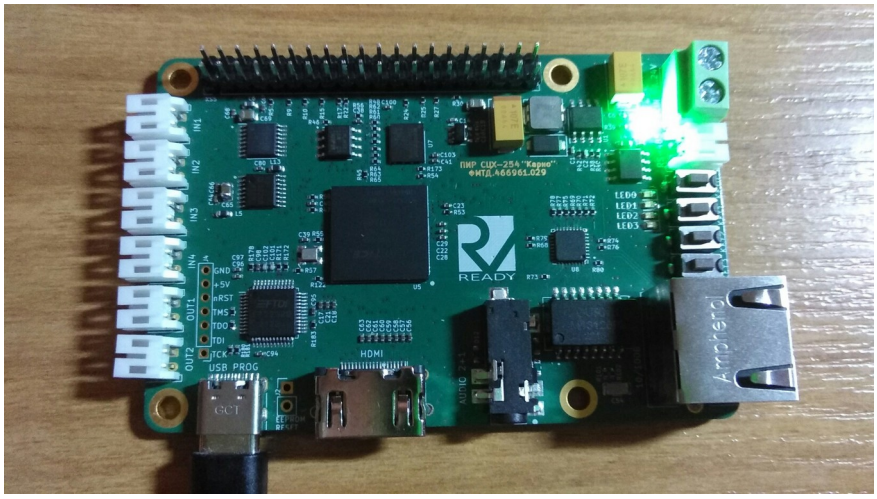


Рис. 19. ПИР СЦХ-254 «Карно» в действии.

Если Вы проводите синтез на удаленной системе (на сервере), к которой нет возможности подключить отладочную плату через USB, то загрузку битстрима можно осуществлять с любой локальной машины. Для этого на неё достаточно установить утилиту **openFPGALoader**. Как я уже упоминал выше, эта утилита доступна почти во всех дистрибутивах Linux и BSD, и установить её можно из системного репозитория, например командой:

```
$ sudo apt-get install openfpgaloader
```

После чего выполнять загрузку битстрима путем вызова утилиты openFPGALoader напрямую, предварительно скопировав бинарный файл с битстримом на локальную систему. Вызов утилиты осуществляется следующим образом:

```
# для загрузки в SRAM
$ openFPGALoader -v --ftdi-channel 0 01_and_or_not_xor_de_morgan.bin

# для загрузки в NOR flash
$ openFPGALoader -v -f --ftdi-channel 0 01_and_or_not_xor_de_morgan.bin
```

8. Типовой Makefile для синтеза с помощью Yosys

Для желающих попробовать Yosys я приведу небольшой **Makefile** с описанием последовательности (конвейера) запуска утилит для того, чтобы превратить цифровую схему из исходного описания на языке SystemVerilog в битстрим, готовый к заливке в микросхему ПЛИС. Данный файл предназначен для сборки под плату «Карно» с ПЛИС Lattice ECP5.

Итак, описываем проект: название, исходные файлы и зависимости:

```
NAME = 01_and_or_not_xor_de_morgan
INC = /home/rz/basics-graphics-music/labs/common
BOARD = karnix_ecp5_yosys
READ_VERILOG += -p "read_verilog -sv
/home/rz/basics-graphics-music/labs/01_and_or_not_xor_de_morgan/top.sv"
DEPS += /home/rz/basics-graphics-music/labs/01_and_or_not_xor_de_morgan/top.sv
```

Описываем тип корпуса микросхемы и файл распиновки внешних сигналов:

```
LPF = board_specific.lpf
DEVICE = 25k
PACKAGE = CABGA256
```

Запускаем по очереди: **yosys** — для синтеза, **nextpnr-ecp5** — для размещения и трассировки, **ecppack** — для упаковки в битстрим:

```
all: $(NAME).bin

$(NAME).bin: $(PCF) $(DEPS)
        yosys -p "verilog_defaults -add -I$(INC)" $(READ_VERILOG) -p "synth_ecp5 -json $
(NAME).json -top board_specific_top"
        nextpnr-ecp5 --package $(PACKAGE) --$(DEVICE) --json $(NAME).json --textcfg $
(NAME)_out.config --lpf $(LPF) --lpf-allow-unconstrained
        ecppack --compress --freq 38.8 --input $(NAME)_out.config --bit $(NAME).bin
```

Загрузка битстрима в ПЛИС:

```
upload_openloader:
ifeq ("$(FLASH_METHOD)", "flash")
        openFPGALoader -v --ftdi-channel $(FTDI_CHANNEL) -f --reset $(NAME).bin
else
        openFPGALoader -v --ftdi-channel $(FTDI_CHANNEL) $(NAME).bin
endif
```

Полный текст заготовок Makefile-а для своего проекта можно взять из проекта «basics-graphics-music» из следующих файлов:

Для ПЛИС Lattice ECP5: **boards/karnix_ecp5_yosys/Makefile**

Для ПЛИС Lattice iCE40: **boards/ice40hx8k_evb_yosys/Makefile**

Для ПЛИС Gowin GW1N: **boards/tang_nano_9k_gowin_yosys/Makefile**

Для ПЛИС Gowin GW2A: **boards/tang_primer_20k_dock_gowin_yosys/Makefile**

9. Файл описания внешних сигналов и ограничений (LPF, PCF, CST)

Каждый проприетарный тул имеет свой формат файла для описания внешних связей и типа внешних сигналов микросхемы ПЛИС. Так, в Gowin EDA для ПЛИС GW1N и GW2A используется файл формата .CST, в Lattice Diamond для ПЛИС iCE40 используется файл .PCF, а для ПЛИС ECP5 это файл .LPF.

В тулчейне Yosys файл с описанием внешних сигналов необходим утилите плэйсеру — NextPNR, для каждого вида микросхем ПЛИС существует своя версия этой утилиты: **nextpnr-ice40**, **nextpnr-ecp5**, **nextpnr-gowin** и т. д. Разработчики утилиты NextPNR не стали изобретать какой-то свой унифицированный формат файла, а сделали поддержку того формата, который принят для данной микросхемы ПЛИС. С одной стороны, это очень удобно при переносе проектов с проприетарных тулов. С другой, при переносе проекта с одной микросхемы ПЛИС на другую в рамках Yosys возникает необходимость заново создавать файл описания. На сколько это удобно — судите сами, но я бы предпочел, чтобы был еще один универсальный формат.

Разберем формат файла .LPF чуть более детально и посмотрим на некоторые из часто используемых директив для описания сигналов и ограничений в ПЛИС Lattice ECP5:

Конструкция **LOCATE COMP "xxx" SITE "yyy";** привязывает внутренний сигнал **xxx** к выводу **yyy** микросхемы, например:

```
LOCATE COMP "CLK" SITE "B9"; // 25MHz crystal oscillator
или
LOCATE COMP "GPIO[0]" SITE "B9"; // Connected to IO_6
```

Конструкция **IOBUF PORT "xxx" IO_TYPE=LVCMOS33;** указывает на то, что для сигнала **xxx** необходимо использовать аппаратных блок **IOBUF** - двунаправленный буфер (другие варианты не поддерживаются). Также эта директива указывает на то, что для данного сигнала нужно использовать буфер типа Low-voltage CMOS на 3.3V. Возможные варианты: LVCMOS33, LVCMOS25, LVCMOS18, LVCMOS15, LVCMOS12, LVCMOS33D, LVCMOS25D, LVCMOS15D, LVCMOS12D, LVCMOS18D .

Пример:

```
IOBUF PORT "CLK" IO_TYPE=LVCMOS33;
```

Помимо **IO_TYPE** в ограничениях для данного вывода микросхемы можно указать необходимость в резисторе подтяжки. Делается это следующим образом:

```
PULLMODE=NONE
или
PULLMODE=UP
или
PULLMODE=DOWN
```

Также можно указать силу тока выходного каскада (драйвера) через директиву **DRIVE**, которая может принимать одно из значений: 4,6,8,10,12 и 16 которые соответствуют току в мА. Пример:

```
LOCATE COMP "io_gpio[0]" SITE "L1";
IOBUF PORT "io_gpio[0]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_gpio[0]" PULLMODE=NONE DRIVE=16;
```

Еще одна полезная директива **FREQUENCY NET** указывает на то, что данный сигнал используется для тактирования, следом указывается минимально разрешенная частота. Это подсказка плэйсеру, о том, что данный сигнал требуется трассировать особым образом и проверять задержки (тайминги) на возможность работы при заданной частоте. Пример:

```
FREQUENCY NET "clk" 25.00000 MHz;
```

Это пожалуй все, что необходимо знать начинающему про файл LPF. Более подробно со списком поддерживаемых директив можно ознакомиться в техническом описании на соответствующую микросхему ПЛИС.

Полный файл описания внешних сигналов и ограничений для различных ПЛИС и плат можно получить из проекта «basics-graphics-music»:

Для ПЛИС Lattice ECP5: **boards/karnix_ecp5_yosys/board_specific.lpf**

Для ПЛИС Lattice iCE40: **boards/ice40hx8k_evb_yosys/board_specific.pcf**

Для ПЛИС Gowin GW1N: **boards/tang_nano_9k/board_specific.cst**

Для ПЛИС Gowin GW2A: **boards/tang_primer_20k_dock_gowin_yosys/board_specific.cst**

10. Особенности синтаксиса Yosys

На момент написания этой статьи, из проекта «basics-graphics-music» тулчейном Yosys поддерживаются не все лабораторные работы. Это связано с тем, что синтаксис языка SystemVerilog весьма обилен и разнообразен и, как это часто бывает с любыми компиляторами, поддержка сложных синтаксических конструкций несколько отличается от тула к тулу.

Так получилось и с Yosys. Если попытаться синтезировать лабораторную работу **02_mux**, то можно получить от утилиты **yosys** следующее сообщение об ошибке:

```
rz@devbox:~/basics-graphics-music/labs/01_and_or_not_xor_de_morgan$ cd ../02_mux/
rz@devbox:~/basics-graphics-music/labs/02_mux$

rz@devbox:~/basics-graphics-music/labs/02_mux$ bash 03_synthesize_for_fpga.bash
Configuring for Lattice ECP5...
OK

...

 Yosys 0.36+58 (git sha1 ea7818d31, gcc 7.5.0-3ubuntu1~18.04 -fPIC -Os)


-- Running command `verilog_defaults -add -I/home/rz/basics-graphics-music/labs/common' --

-- Running command `read_verilog -sv /home/rz/basics-graphics-music/labs/02_mux/top.sv' --

1. Executing Verilog-2005 frontend: /home/rz/basics-graphics-music/labs/02_mux/top.sv
Parsing SystemVerilog input from `/home/rz/basics-graphics-music/labs/02_mux/top.sv' to AST
representation.
/home/rz/basics-graphics-music/labs/02_mux/top.sv:4: ERROR: Unimplemented compiler directive or
undefined macro `error_This_module_requires_support_for_multidimantional_arrays.
Makefile:76: recipe for target '02_mux.bin' failed
make: *** [02_mux.bin] Error 1
```

Заглянув в файл **top.sv** в строках 3-5 мы увидим следующую директиву препроцессору заключенную в директиву **ifdef**:

```
`ifdef YOSYS
    `error_This_module_requires_support_for_multidimantional_arrays
`endif
```

Не сложно догадаться, что директивы с таким именем в Yosys нет и быть не может. Использование такой директивы это хак — способ остановить работу препроцессора Yosys с ошибкой и дать пользователю понять, что дальнейшая обработка его дизайна данным тулом

невозможна, потому что **Yosys, на момент написания этой статьи не поддерживал многомерные массивы**.

Этот код условной компиляции (ifdef) с ошибкой был добавлен в код лабораторной работы мной, когда я занимался портированием проекта на тулчейн Yosys. Вообще в коде лабораторной работы **02_mux** какой-то особой надобности в многомерных массивах нет, и её можно было бы смело переписать, используя только двумерные массивы (с ними проблем в Yosys нет), но после совещания с коллективом я пришел к выводу, что стоит оставить оригинальный код, так как студентам нужно продемонстрировать использование многомерных массивов сигналов в SystemVerilog.

Замечу, что **02_mux** — это единственная лабораторная работа, которая не собирается (не синтезируется) тулчейном Yosys, код остальных лабораторных работ исправлен директивами условной «компиляции».

Отсутствие поддержки многомерных массивов — это не единственная проблема Yosys. Ниже я продемонстрирую еще несколько часто встречающихся конструкций языка SystemVerilog, которые не поддерживаются.

10.1 Макро YOSYS

Прежде всего надо сказать, что Yosys устанавливает макро YOSYS, что позволяет делать условную сборку проекта. Выглядит это совершенно обыденно:

```
`ifdef YOSYS
    /* some Yosys specific code */
`else
    /* code for other tools */
`endif
```

10.2 Многомерные массивы сигналов

Многомерные массивы сигналов (multi-dimentional arrays) поддерживаются только до двух уровней. Пример массива размером 1024 для ячеек памяти размерностью 32 бит, который может быть синтезирован Yosys-ом:

```
reg [1023:0] mem [31:0];
```

или

```
reg [1023:0][31:0] mem;
```

Описать память с банками уже не выйдет, т.е. вот такой синтаксис не сработает:

```
reg [3:0][1023:0][31:0] mem;
```

10.3 Функция возведения в степень ($POW) и оператор **

Отсутствует встроенная функция возведения в степень ($POW) и оператор **. Данный код не работает:

```
assign x = y ** 2;
```

Для второй степени это легко обходится следующим образом:

```
  assign x = y * y;
```

Для других степеней придется писать свой код.

10.4 Функция не может иметь доступ к сигналу, описанному за её пределами

Функция (function) может оперировать только тем, что ей передано в параметрах. Доступ к сигналам и регистрам, которые находятся снаружи, изнутри функции запрещен.

Например, в лабе **13_music_recognition** содержалась следующая последовательность функций, которая использовала сигнал **distance**, описанный выше в рамках модуля **top**. Это код Yosys-ом **не** синтезируется:

```
logic [19:0] distance;

//-----------------------------------------------------------------------

function [19:0] check_freq_single_range (input [18:0] freq_100);

   check_freq_single_range =    distance > low_distance  (freq_100)
                              & distance < high_distance (freq_100);
endfunction


//-----------------------------------------------------------------------

function [19:0] check_freq (input [18:0] freq_100);

   check_freq =   check_freq_single_range (freq_100 * 4)
                | check_freq_single_range (freq_100 * 2)
                | check_freq_single_range (freq_100    );

endfunction
```

Код был переписан таким образом, чтобы сигнал **distance** передавался в качестве одного из входных сигналов в функции его использующие:

```
//-----------------------------------------------------------------------

function [19:0] check_freq_single_range (input [18:0] freq_100, input [19:0]
distance);

   check_freq_single_range =    distance > low_distance  (freq_100)
                              & distance < high_distance (freq_100);
endfunction

//-----------------------------------------------------------------------

function [19:0] check_freq (input [18:0] freq_100, input [19:0] distance);

   check_freq =   check_freq_single_range (freq_100 * 4 , distance)
                | check_freq_single_range (freq_100 * 2 , distance)
                | check_freq_single_range (freq_100     , distance);

endfunction
```

10.5 Сигналы нулевой или отрицательной размерности

Тут все просто, сигналы нулевой и, тем более, отрицательной размерности не поддерживаются и на попытку объявить такие сигналы Yosys выдает соответствующее сообщение об ошибке. Многие могут задаться вопросом, а зачем разработчику могут понадобиться такие бессмысленные сигналы? Ответ прост: такие сигналы обычно появляются в результате исполнения различных преобразований с использованием параметров конфигурации модулей.

Для иллюстрации возьмем модуль **top** любой из лабораторных работы. В определении этого модуля (см. 7.2) имеется ряд сигналов, размерность которых задана с изменяемым параметром, например:

```
input        [w_key   - 1:0] key,
```

где **w_key** — это параметр, определяющий число доступных на плате кнопок, передается он сюда сверху из **board_specific_top.sv**. Если на какой-либо из плат нет своих кнопок, то параметр **w_key** будет установлен в ноль и определение сигнала **input** выродится в что-то типа:

```
input        [0   - 1:0] key,
```

То есть будет попытка определить сигнал отрицательной размерности.

Такие тулы как Quartus и Vivado спокойно проглатывают такой синтаксис, а вот Yosys — нет.

10.6 Сложные битовые манипуляции иногда могут выдавать ошибку о loopback-ах

С этой ошибкой пока не удалось полностью разобраться. Ниже я приведу код из лабораторной работы **04_priority_encoder**, который пришлось переработать:

```
`ifdef YOSYS

    wire [w - 1:0] c;

    genvar i;

    generate
        for (i = 0; i < w; i = i + 1) begin
            if (i == 0)
                assign c [0] = 1'b1;
            else
                assign c [i] = ~ in [i - 1] & c [i - 1];
        end
    endgenerate

`else

    wire [w - 1:0] c = { ~ in [w - 2:0] & c [w - 2:0], 1'b1 };

`endif
```

10.7 Глобализация сигналов

В Yosys поддерживаются следующие псевдомодули для "глобализации" сигналов, т. е. способ дать подсказку синтезатору и плэйсеру о том, что определенный сигнал требуется трассировать используя глобальные линии.

Для ПЛИС iCE40:

```
SB_GB clk_buf
(.USER_SIGNAL_TO_GLOBAL_BUFFER(CLK), .GLOBAL_BUFFER_OUTPUT(slow_clk));
```

Для ПЛИС ECP5:

```
DCCA clk_buf (.CLKI(CLK), .CLKO(slow_clk), .CE(1));
```

Для ПЛИС Gowin:

```
BUFG clk_buf (.I(CLK), .O(slow_clk));
```

Для ПЛИС Gowin имеется следующий нюанс. Утилита плэйсер **nextpnr-gowin**, выполняющая размещение и трассировку для ПЛИС Gowin, не понимает эту фичу и отказывается выполнять работу. То есть этап на стадии работы утилиты **yosys** проходит и нетлист синтезируется успешно, но **nextpnr-gowin** данную фичу не поддерживает и в логе можно увидеть следующее информационное сообщение:

```
./gowin/arch.cc: log_info("BUFG isn't supported\n");
```

Однако, **nextpnr-gowin** все равно пытается разрулить искусственные тактовые сигналы оптимальным путем. Для примера, я завел счетчик-делитель и взял один из его битов в качестве **slow_clk**. Вот что я увидел в логе:

```
Info: Max frequency for clock 'div[22]': 315.86 MHz (PASS at 27.00 MHz)```
```

То есть **nextpnr-gowin** определил, что сигнал **div[22]** используется для тактирования и вывел его на глобальную линию, что видно по максимальной частоте!

На данный момент это всё, что мне известно об особенностях синтаксиса тулчейна Yosys. Если кто-то из читателей обладает дополнительной информацией — буду рад обсудить в комментариях и добавлю в текст статьи.

11. Анализируем сообщения от тулов

В ходе работы утилиты **yosys** и **nextpnr** выдают огромный поток информации о протекающем процессе, в котором, помимо сообщений о синтаксических ошибках, могут быть сообщения требующие внимания, особенно если ваш дизайн работает не так, как вы запланировали/предполагали. В этом случае следует внимательно изучить сообщения об оптимизации и сокращении цепей (nets), триггеров (DFFs — «D flip-flops») и модулей (modules). Возможно, что в вашем дизайне присутствует логическая ошибка, в результате которой часть «важных» сигналов становится логически бесполезными и оптимизатор их просто удалит за ненадобностью. Если такой сигнал/триггер обнаруживается среди сообщений перечисленных ниже, требуется тщательным образом проанализировать текст модулей с его участием и, возможно, переписать более прозрачно.

11.1 Сообщения от утилиты yosys

В выводе от утилиты **yosys** стоит обратить внимание на следующие сообщения:

1. На различные предупреждения:

```
Warning: Yosys has only limited support for tri-state logic at the moment. (/home/rz/basics-
graphics-music/labs/common/tm1638_board.sv: 303)
```

Да, **yosys** поддерживает состояние высокого импеданса для сигнальных линий (**tri-state** или **z**), но эта поддержка имеет свои нюансы, о чем нам и сообщает синтезатор.

Или вот такое предупреждение:

```
Warning: Resizing cell port board_specific_top.slow_clk_buf.CE from 32 bits to 1 bits.
```

Здесь синтезатор сообщает нам, что объявленный сигнал **slow_clk_buf** размерностью 32 бита был урезан до одного бита, так как этого вполне достаточно.

2. Сообщения об удаленных неиспользуемых модулях и неиспользуемых сигналах:

```
13.3.5. Analyzing design hierarchy..
Top module:  \board_specific_top
...
Removing unused module `\seven_segment_display'.
Removing unused module `\vga'.
Removing unused module `\digilent_pmod_mic3_spi_receiver'.
Removing unused module `\shift_reg'.
Removing unused module `\slow_clk_gen'.
Removing unused module `\inmp441_mic_i2s_receiver'.
Removing unused module `\strobe_gen'.
Removing unused module `\counter_with_enable'.
Removing unused module `\i2s_audio_out'.
Removing unused module `\tm1638_sio'.
Removing unused module `\tm1638_board_controller'.
Removing unused module `\top'.
Removed 12 unused modules.
```

Лабораторная работа подключает некоторое количество инструментальных модулей из этого же репозитория, но не использует их, а значит эти модули будут убраны.

3. Сообщения об удаленных неиспользуемых процессах:

```
13.4.11. Executing PROC_CLEAN pass (remove empty switches from decision trees).
Removing empty process
`TRELLIS_FF.$proc$/usr/local/bin/../share/yosys/ecp5/cells_sim.v:0$403'.
Found and cleaned up 2 empty switches in
`\TRELLIS_FF.$proc$/usr/local/bin/../share/yosys/ecp5/cells_sim.v:350$402'.
Removing empty process
`TRELLIS_FF.$proc$/usr/local/bin/../share/yosys/ecp5/cells_sim.v:350$402'.
Removing empty process
`DPR16X4C.$proc$/usr/local/bin/../share/yosys/ecp5/cells_sim.v:0$377'.
Found and cleaned up 1 empty switch in
`\DPR16X4C.$proc$/usr/local/bin/../share/yosys/ecp5/cells_sim.v:285$354'.
Removing empty process
`TRELLIS_DPR16X4.$proc$/usr/local/bin/../share/yosys/ecp5/cells_sim.v:0$320'.
Found and cleaned up 1 empty switch in
`\TRELLIS_DPR16X4.$proc$/usr/local/bin/../share/yosys/ecp5/cells_sim.v:221$296'.
Removing empty process
`TRELLIS_DPR16X4.$proc$/usr/local/bin/../share/yosys/ecp5/cells_sim.v:213$295'.
Found and cleaned up 1 empty switch in `$paramod$c57045fef6efabbb852bcb6522a23329a7270ab8\
slow_clk_gen.$proc$/home/rz/basics-graphics-music/labs/common/slow_clk_gen.sv:35$406'.
Removing empty process `$paramod$c57045fef6efabbb852bcb6522a23329a7270ab8\slow_clk_gen.
$proc$/home/rz/basics-graphics-music/labs/common/slow_clk_gen.sv:35$406'.
Cleaned up 5 empty switches.
```

Пожалуй, здесь все сообщения, кроме последнего, касаются процессов в подключенных библиотечных модулях, которые не были использованы в схеме и, соответственно, подлежат удалению.

Последнее сообщение относится к «медленному клоку», который формируется в модуле-обертке, но не задействован в модуле данной лабораторной работы, поэтому тоже будет удален.

4. Сообщения о сокращении размерности сигналов:

```
13.14. Executing WREDUCE pass (reducing word size of cells).
Removed top 2 bits (of 4) from port Y of cell board_specific_top.$not$/home/rz/basics-
graphics-music/boards/karnix_ecp5_yosys/board_specific_top.sv:91$4 ($not).
Removed top 2 bits (of 4) from port A of cell board_specific_top.$not$/home/rz/basics-
graphics-music/boards/karnix_ecp5_yosys/board_specific_top.sv:91$4 ($not).
Removed top 2 bits (of 4) from wire board_specific_top.top_key.
Removed top 2 bits (of 4) from wire board_specific_top.top_led.
```

Синтезатор анализирует, как используются биты в многобитных сигналах и, если они не используются, то синтезатор отправит их под сокращение, то есть урежет размерность.

5. Статистика по утилизации ресурсов

```
13.47. Printing statistics.

=== board_specific_top ===

   Number of wires:                 33
   Number of wire bits:            125
   Number of public wires:          33
   Number of public wire bits:     125
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                  1
     LUT4                            1
```

Статистика тоже имеет важное и полезное значение. Разработчик всегда должен иметь представление о том, какое количество сигналов, триггеров, процессов или ячеек памяти в его дизайне и, если это теоретическое значение начинает расходится с тем, что **yosys** выдает в результате синтеза, это явный признак логической ошибки, либо неверного использования языковых конструкций языка Verilog.

В приведенном выше сообщении об используемых ресурсах для лабораторной работы **01_and_or_not_xor_de_morgan** видно, что весь дизайн лабораторной работы уложился в одну макроячейку (cell) из одного LUT-4, так как в данной лабораторной работе была реально задействована только комбинационная логика из нескольких логических элементов, что покрывается возможностями одного LUT-4.

А сейчас проведем следующий эксперимент: добавим в самый конец модуля этой же лабораторной работы код для регистра-счетчика **counter** размерностью 8 бит, будем увеличивать его каждый раз когда нажата клавиша **KEY[2]**, а старший бит **counter[7]** выведем на **LED[2]**, т. е. добавим следующий код:

```
logic [7:0] counter;

assign led[2] = counter[7];

always_ff @ (posedge clk or posedge rst)
        if(rst)
```

```
                counter <= 0;
        else if(key[2])
                counter <= counter + 8'd1;
```

Запустим синтез, дождемся его завершения (а он обязан завершиться без ошибки), загрузим в ПЛИС и протестируем. Как бы долго мы не удерживали клавишу KEY[2], состояние светодиода LED[2] меняться не будет, что очень странно, ведь синтез завершился без ошибок! Для решения этой проблемы сначала посмотрим на статистику по задействованным ресурсам, выдаваемую утилитой **yosys**, здесь мы увидим следующее:

```
13.47. Printing statistics.

=== board_specific_top ===

   Number of wires:                 34
   Number of wire bits:            133
   Number of public wires:          34
   Number of public wire bits:     133
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                  1
     LUT4                            1
```

Мы обнаружим, что наш измененный дизайн всё еще укладывается в один LUT-4 и в нём нет никаких триггеров, а такого быть не должно. Но как так получилось?! Анализируем вывод **yosys** более детально и обнаруживаем следующее предупреждение:

```
13.9. Executing OPT_CLEAN pass (remove unused cells and wires).
Finding unused cells or wires in module \board_specific_top..
Warning: Driver-driver conflict for \i_top.counter [7] between cell $flatten\i_top.
$procdff$454.Q and constant 1'0 in board_specific_to p: Resolved using constant.
Removed 12 unused cells and 35 unused wires.
```

Здесь **yosys** сообщает нам, что он увидел конфликт между двумя источниками — сигналом **i_top.counter [7]** и константой **1'0** которые пытаются задавать («драйвить») один и тот же сигнал и в этом конфликте **константа выиграла**, а значит всё, что далее связано с сигналом **i_top.counter[7]** было по цепочек удалено за ненадобностью!

Откроем файл с кодом модуля, внимательно просмотрим его с самого начала и обнаружим вот такой кусочек кода:

```
`ifndef VERILATOR

generate
    if (w_led > 2)
    begin : unused_led
        assign led [w_led - 1:2] = '0;
    end
endgenerate

`endif
```

Но что делает этот код и зачем он тут? Если вкратце, то этот код «приземляет» неиспользуемые линии **led**, если модуль синтезируется, а не симулируется, точнее он присваивает неиспользуемым ранее битам сигнала **led** значение **0.** Этот код добавлен другим разработчиком и мы, не обратив на него внимания и добавив свой код в конец модуля, получили нерабочий модуль, который успешно синтезируется без ошибок!

Ну что же, исправим это дело — временно изменим условие **ifdef** так, чтобы этот обнуляющий код не подключался и пересоберем модуль еще раз. В результате мы получим следующую статистику от утилиты **yosys**:

```
13.47. Printing statistics.

=== board_specific_top ===

   Number of wires:                 49
   Number of wire bits:            159
   Number of public wires:          49
   Number of public wire bits:     159
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                 15
     CCU2C                           4
     LUT4                            3
     TRELLIS_FF                      8
```

Хо-хо! Теперь мы видим, что наш дизайн синтезировался в схему, содержащую восемь триггеров (блоков TRELLIS_FF), что соответствует нашему регистру-счетчику, и четыре блока CCU2C, из которых построен сумматор. Загрузим полученный битстрим в ПЛИС и убедимся, что индикатор LED2 реагирует на нажатия кнопки KEY2.

Примерно таким образом приходится действовать разработчику при поиске логических засад в дизайне, над которым он работает, и в этом деле **yosys** старается помочь в меру своих возможностей.

11.2 Сообщения от утилиты nextpnr

Теперь разберем некоторые важные сообщения от утилиты **nextpnr**. Напомню, что утилита **nextpnr** (для ПЛИС ECP5 она называется **nextpnr-ecp5**) производит размещение блоков, полученных в результате синтеза, внутри микросхемы ПЛИС и трассировку сигнальных цепей между ними. В процессе своей работы утилита **nextpnr** производит ряд оптимизаций в поиске оптимального расположения с минимизацией объема задействованных ресурсов микросхемы ПЛИС, а также производит привязку линий ввода вывода к внешним выводам микросхемы. В процессе оптимизации эта утилита выполняет статический анализ задержек (STA), вычисляет критические (самые «длинные») пути в схеме и рассчитывает максимально допустимые частоты, исходя из свойств блоков выбранной микросхемы ПЛИС и полученной топологии размещения.

Прежде чем мы приступим, давайте ознакомимся с терминологией, используемой в утилите **nextpnr**. Ниже приведена небольшая выдержка из NextPnR FAQ:

Терминология базы данных дизайна:

- **Cell (ячейка)**: логическая сущность или физический блок внутри нетлиста. Упаковщик, являющийся частью NextPnR, комбинирует или иным способом модифицирует ячейки, а плэйсер размещает их на доступных Базовых Элементах (Bels).
- **Port (порт)**: вход или выход ячейки (cell), может быть связан только с одной цепью (net).
- **Net (цепь)**: связь между портами в нетлисте, прямой аналог электрической цепи. Цепь может быть разведена с помощью одного или нескольких проводников (wires) внутри микросхемы. Цепи всегда имеют размерность в один двоичный сигнал (один бит). Многобитные цепи всегда разбиваются на составные части.
- **Source (источник)**: Выходной порт ячейки (cell), управляющий заданной цепью (net).

- **Sink (сток)**: Входной порт ячейки (cell), управляемый заданной цепью (net).
- **Arc (дуга)**: Пара источник-приемник.

Терминология архитектурной базы данных:

- **Bel («базэл»)**: Базовый Элемент — функциональный блок микросхемы ПЛИС, такой как: «логическая ячейка» (LC), ячейка ввода/вывода (IO cell), блочная память (block RAM) и прочие аппаратные блоки ПЛИС. На каждый «базэл» может быть размещена только одна ячейка (cell).
- **Pin (вывод)**: Внешний «пин» ввода/вывод «базэла» (Bel), постоянно соединяется одним проводником внутри ПЛИС.
- **Pip («ПТС»)**: Программируемая точка связи, соединяющая несколько проводников (wires) в заданном направлении.
- **Wire (проводник)**: Фиксированное физическое соединение внутри микросхемы ПЛИС, между ПТС и/или выводами «базэлов».
- **Alias («алиас»)**: Специальная ПТС, представляющая постоянное (неизменяемое) соединение между двумя проводниками.
- **Group (группа)**: Может содержать все выше приведенные сущности: bels, pips, wires и другие группы.
- **BelBucket**: Группа «базэлов» образующая «покрытие множества» (см. «Задачу о покрытии множества»).

Терминология процессов:

- **Packing (упаковка)**: Процесс группирования ячеек, полученных от синтезатора, в более крупные «логические ячейки».
- **Placing (размещение)**: Процесс поиска и размещения упакованных ячеек в базовые элементы (базэлы).
- **Routing (трассировка)**: Процесс поиска и объединения цепей проводниками.

Еще термины:

- **Binding (привязка)**: Привязка цепей к проводникам и ячеек к базовым элементам.
- **Path (путь)**: Все дуги, соединяющие вывод одного FF триггера («флип-флопа») с основным входом другого FF триггера.

7Давайте посмотрим, какую же полезную информацию мы можем добыть из вывода **nextpnr**. Сразу после запуска утилита выводит информационное сообщение о том, сколько и каких ресурсов присутствует в топологии, которая поступила ей на вход от синтезатора:

```
Info: Logic utilisation before packing:
Info:     Total LUT4s:        11/24288     0%
Info:         logic LUTs:      3/24288     0%
Info:         carry LUTs:      8/24288     0%
Info:           RAM LUTs:      0/ 3036     0%
Info:          RAMW LUTs:      0/ 6072     0%

Info:      Total DFFs:         8/24288     0%
```

Первое число в третьей графе — это количество задействованных ресурсов каждого вида. Второе число — это максимально доступное количество ресурса для выбранного типа и корпуса микросхемы ПЛИС. Как я уже упоминал, на плате «Карно» установлена микросхема ПЛИС ECP5 с **25K** логических ячеек. Обозначение «25K» это чисто маркетинговая уловка, на

самом деле логических ячеек в данной микросхеме немного меньше, а именно - 24288 штук. Как видим, не только ребята из Юго-Восточной Азии любят приукрасить действительность.

Далее мы видим привязку сигнальных линий к выводам («ногам») микросхемы:

```
Info: Packing IOs..
Info: pin 'VGA_VS$tr_io' constrained to Bel 'X0/Y44/PIOD'.
Info: pin 'VGA_R[3]$tr_io' constrained to Bel 'X72/Y8/PIOA'.
...
Info: pin 'UART_TX$tr_io' constrained to Bel 'X58/Y0/PIOB'.
Info: pin 'UART_RX$tr_io' constrained to Bel 'X58/Y0/PIOA'.
Info: pin 'SW[1]$tr_io' constrained to Bel 'X29/Y0/PIOB'.
Info: pin 'SW[0]$tr_io' constrained to Bel 'X29/Y0/PIOA'.
Info: pin 'LED[3]$tr_io' constrained to Bel 'X67/Y0/PIOA'.
Info: pin 'LED[2]$tr_io' constrained to Bel 'X67/Y0/PIOB'.
...
Info: pin 'KEY[3]$tr_io' constrained to Bel 'X62/Y0/PIOB'.
Info: pin 'KEY[2]$tr_io' constrained to Bel 'X62/Y0/PIOA'.
..
Info: pin 'GPIO[11]$tr_io' constrained to Bel 'X0/Y32/PIOD'.
Info: pin 'GPIO[10]$tr_io' constrained to Bel 'X0/Y32/PIOC'.
...
```

Иногда можно увидеть следующее предупреждающее сообщение:

```
Warning: IO 'some_output' is unconstrained in LPF and will be automatically placed
```

Таким способом **nextpnr** сообщает разработчику о том, что в его дизайне, в модуле **top**, присутствует внешний сигнал **some_output**, который не описан в LPF, а следовательно, плэйсер не знает, как его привязать к внешнему выводу микросхемы, и этот сигнал будет привязан автоматически к первому подходящему (случайному) выводу. На такие сообщения нужно обращать внимание и не допускать присутствие в схеме внешних сигналов, не привязанных к физическим выводам. Несоблюдение этого правила может приводить к серьезным утечкам тока (если этот случайный вывод оказался на «земле»), как следствие - нестабильная работа микросхемы ПЛИС и выход её из строя!

После выполнения упаковки, утилита **nextpnr** может выдать сообщение о выполненной привязке сигналов к глобальным линиям:

```
Info: Promoting globals...
Info:     promoting clock net CLK$TRELLIS_IO_IN to global network
```

Здесь утилита сообщает о том, что она посчитала, что сигнал **CLK** является тактовой цепью и он будет трассирован с использованием глобальной цепи. Не всегда **nextpnr** распознает тактовые цепи и цепи сброса корректно, поэтому нужно следить за тем, как размещаются такие сигналы и предпринимать определенные действия — имеется способ маркировать такие сигналы, см. главу «*10.7 Глобализация сигналов*». Иногда бывает так, что **nextpnr** совершенно напрасно считает какой-нибудь из промежуточных сигналов тактовым и тоже размещает его в глобальных линиях, понапрасну тратя ограниченный ресурс. Способа избавиться от такого эффекта я пока не нашел, но заметил, что в имени сигнала лучше не использовать строки «clk» и «CLK».

Далее **nextpnr** еще раз выдает статистику о результатах упаковки и размещения, но в этот раз в статистике уже присутствуют наименования аппаратных блоков конкретной микросхемы, которые требуется задействовать для размещения схемы:

```
Info: Device utilisation:
Info:             TRELLIS_IO:    39/  197    19%
Info:                   DCCA:     1/   56     1%
Info:                 DP16KD:     0/   56     0%
```

```
Info:             MULT18X18D:     0/   28     0%
Info:                 ALU54B:     0/   14     0%
Info:                EHXPLLL:     0/    2     0%
Info:                EXTREFB:     0/    1     0%
Info:                   DCUA:     0/    1     0%
Info:               PCSCLKDIV:     0/    2     0%
Info:                 IOLOGIC:     0/  128     0%
Info:                SIOLOGIC:     0/   69     0%
Info:                     GSR:     0/    1     0%
Info:                  JTAGG:     0/    1     0%
Info:                   OSCG:     0/    1     0%
Info:                  SEDGA:     0/    1     0%
Info:                     DTR:     0/    1     0%
Info:                 USRMCLK:     0/    1     0%
Info:                 CLKDIVF:     0/    4     0%
Info:               ECLKSYNCB:     0/   10     0%
Info:                 DLLDELD:     0/    8     0%
Info:                  DDRDLL:     0/    4     0%
Info:                 DQSBUFM:     0/    8     0%
Info:        TRELLIS_ECLKBUF:     0/    8     0%
Info:           ECLKBRIDGECS:     0/    2     0%
Info:                   DCSC:     0/    2     0%
Info:             TRELLIS_FF:     8/24288     0%
Info:           TRELLIS_COMB:    17/24288     0%
Info:           TRELLIS_RAMW:     0/ 3036     0%
```

Данный результат является промежуточным, он подается на вход оптимизатора и наступает самое интересное — процесс «утрясывания» и «перетасовывания» схемы, удаления из неё лишних частей и сокращение длинных связей. Этот процесс стохастический, т. е. в нем присутствует элемент случайности.

После выполнения оптимизации утилита **nextpnr** рассчитает максимально теоретически возможную частоту для тактовых сигналов, о чем сообщит следующим сообщением:

```
Info: Max frequency for clock '$glbnet$CLK$TRELLIS_IO_IN': 457.04 MHz (PASS at 12.00 MHz)
```

В данном случае нам сообщают, что максимально возможная частота тактового сигнала составит 457,02 МГц и она превышает (pass) указанный в ограничениях лимит в 12,00 МГц. Иными словами, все ок и беспокоиться не о чем. Но часто бывает и наоборот — рассчитываемая максимально допустима частота становится ниже заданной для данного дизайна и тогда придется либо перерабатывать дизайн и понижать требования к частотам, либо… дождаться окончания трассировки. Дело в том, что на этапе трассировки тоже работает оптимизация и в некоторых случаях трассировщик может еще немного уменьшить задержки.

Аналогично тактовым сигналам производятся расчеты «критического пути» для остальных сигналов в схеме и вычисление задержки для них. В конце концов, после выполнения трассировки, утилита выдаст статистику по имеющимся критическим путям:

```
Info: Max frequency for clock '$glbnet$CLK$TRELLIS_IO_IN': 371.20 MHz (PASS at 12.00 MHz)

Info: Max delay <async>                            -> <async>                           : 2.86 ns
Info: Max delay <async>                            -> posedge $glbnet$CLK$TRELLIS_IO_IN: 1.83 ns
Info: Max delay posedge $glbnet$CLK$TRELLIS_IO_IN -> <async>                           : 1.75 ns
```

Здесь мы видим, что в процессе трассировки максимально возможная частота для сигнала **CLK** уменьшилась с 457,04 МГц до 371,20 МГц, но все еще проходит заданное ограничение в 12,0 МГц. На этом этапе непрохождение заданного лимита по частоте является фатальным и если это случается, то утилита **nextpnr** останавливается с ошибкой.

Успешное завершение утилиты **nextnpr** сигнализируется сообщением:

```
Info: Program finished normally.
```

Для наглядности, ниже я приведу вывод сообщений для одного из моих дизайнов, в котором как раз присутствует ситуация, когда рассчитываемая максимальная частота в критических путях не проходит заданное ограничение (ниже лимита) и расскажу, как я эту проблему решил.

Мой дизайн **VexRiscvWithHUB12ForKarnix** (см. главу 18) использует синтезируемое ядро VexRiscv, синтезируемый Ethernet контроллер и еще некоторое количество синтезируемой логики для организации видео фреймбуфера для работы со светодиодными матрицами. Для выполнения задачи мне требовалось, чтобы вычислительное ядро работало на частоте не менее 60 МГц. Для этого был задействован блок PLL на вход которого подается частота 25 МГц от генератора, распаянного на плате, преобразуется в 60 МГц и подается на вычислительное ядро.

После окончания первой стадии (размещения), утилита **nextpnr** выдала следующую статистику по критическим путям:

```
Info: Max frequency for clock '$glbnet$io_lan_rxclk$TRELLIS_IO_IN': 62.89 MHz (PASS at 25.00 MHz)
Info: Max frequency for clock '$glbnet$io_lan_txclk$TRELLIS_IO_IN': 65.24 MHz (PASS at 25.00 MHz)
Info: Max frequency for clock             '$glbnet$core_pll_CLKOP': 38.30 MHz (FAIL at 60.00 MHz)

Info: Max delay <async>                                  -> posedge $glbnet$core_pll_CLKOP
: 9.00 ns
Info: Max delay <async>                                  -> posedge
$glbnet$io_lan_rxclk$TRELLIS_IO_IN: 2.88 ns
Info: Max delay posedge $glbnet$core_pll_CLKOP           -> <async>
: 18.67 ns
Info: Max delay posedge $glbnet$core_pll_CLKOP           -> posedge
$glbnet$io_lan_rxclk$TRELLIS_IO_IN: 5.51 ns
Info: Max delay posedge $glbnet$core_pll_CLKOP           -> posedge
$glbnet$io_lan_txclk$TRELLIS_IO_IN: 4.07 ns
Info: Max delay posedge $glbnet$io_lan_rxclk$TRELLIS_IO_IN -> posedge $glbnet$core_pll_CLKOP
: 1.84 ns
Info: Max delay posedge $glbnet$io_lan_txclk$TRELLIS_IO_IN -> <async>
: 4.24 ns
Info: Max delay posedge $glbnet$io_lan_txclk$TRELLIS_IO_IN -> posedge $glbnet$core_pll_CLKOP : 1.60
ns
```

из которой видно, что расчет для сигнала **core_clk_CLKOP** дает максимальную частоту 38.30 МГц.

Дождавшись окончания трассировки, я получил следующую статистику по критическим путям:

```
Info: Max frequency for clock '$glbnet$io_lan_rxclk$TRELLIS_IO_IN': 88.19 MHz (PASS at 25.00 MHz)
Info: Max frequency for clock '$glbnet$io_lan_txclk$TRELLIS_IO_IN': 74.08 MHz (PASS at 25.00 MHz)
ERROR: Max frequency for clock             '$glbnet$core_pll_CLKOP': 56.58 MHz (FAIL at 60.00 MHz)

…
0 warnings, 1 error
```

Видно, что оптимизатор в процессе трассировки хорошо поработал и ужал задержку так, что максимально возможная частота для сигнала **core_pll_CLKOP** поднялась аж до 56,58 МГц, но всё еще не дотягивает до 60,0 МГц.

К сожалению, в моём случае перерабатывать дизайн было крайне нежелательно, и я решил поиграться с параметром **--seed** утилиты **nextpnr-ecp5** (задается в Makefile). Это

параметр инициализации генератора псевдослучайных чисел, используемых для первоначального случайного размещения. Я заметил, что изменяя этот параметр можно в некоторых пределах варьировать результаты размещения и трассировки — для них получаются немного отличающиеся задержки в критических путях. Таким образом, где-то на 15-й попытке мне удалось подобрать такое число и получить такое размещение, что максимальная частота для сигнала **core_pll_CLKOP** перевалила за 60,0 МГц и я получил рабочий битстрим. Ниже вывод статистики от **nextpnr**:

```
Info: Max frequency for clock '$glbnet$io_lan_rxclk$TRELLIS_IO_IN': 82.36 MHz (PASS at 25.00 MHz)
Info: Max frequency for clock '$glbnet$io_lan_txclk$TRELLIS_IO_IN': 67.20 MHz (PASS at 25.00 MHz)
Info: Max frequency for clock             '$glbnet$core_pll_CLKOP': 61.35 MHz (PASS at 60.00 MHz)
```

Еще один способ решить эту же проблему — это увеличить параметр **`--speed`** который задает «грейд» (качество исполнения) микросхемы ПЛИС. На плате «Карно» использована микросхема 6-го грейда, но увеличив **speed** до 7-го я получил вполне рабочий битстрим, который был успешно проверен рядом тестов. Да, такой способ более опасный, может подвести в самый не подходящий момент, я рекомендую прибегать к нему только в случаях отладки и проведения экспериментов, но не для конечного изделия.

12. Синтезируемая ЭВМ

Идея собрать вычислительную машину полностью из микросхем программируемой логики пришла людям в головы сразу, как только появились достаточно объемные по ресурсам микросхемы ПЛУ и ПЛИС. На сайте Wikipedia на страничке посвященной ПЛИС есть упоминание о том, что идея разработки первой «синтезируемой» ЭВМ была предложена неким Стивом Кассельманом в 1987 году и, при финансовой поддержке центра управления ВМФ США (Naval Surface Warfare Department), такая машина была успешно создана и запатентована в 1992 году. Машина содержала 600 000 программируемых вентилей (т. е. логических элементов, не макроячеек и не LUT-ов), следовательно логично предположить, что эта машина была построена на микросхемах PAL и PLD. К сожалению, какой-либо дополнительной информации об этом изобретении мне найти не удалось.

Тем не менее, это событие тут же подстегнуло развитие совершенно нового направления в цифровой электронике — создание синтезируемых микропроцессорных ядер (или «soft-core»). Так, в 1990-х годах все именитые производители ПЛИС отметились выпуском своих 8-ми битных синтезируемых ядер: PicoBlaze от Xilinx, Nios от Altera, LatticeMicro8 от Lattice Semi. Все эти синтезируемые ядра были построены по RISC архитектуре и требовали совсем небольшое (по современным меркам) количество ресурсов. Для примера, ядро *LatticeMicro8* требовало всего **200 LUT-ов**. В начале 2000-х годов все они получили апгрейд: *MicroBlaze* — 32/64 бит RISC, *Nios II* - 32 бит RISC и *LatticeMicro32* — тоже 32 бит RISC.

Первоначально синтезируемые микропроцессорные ядра распространялись как отдельно лицензируемые коммерческие продукты (IP-блоки) в составе фирменного IDE, но к концу 2000-х годов все они либо были опубликованы под свободными лицензиями, либо получили опенсорсные клоны и аналоги. Среди чисто опенсорсных ядер стоит упомянуть *OpenRISC (or1k)* — проект Дамьяна Лампрета (Damjan Lampret), основанного в далеком 1999 году, одного из ранних предшественников и конкурентов RISC-V. OpenRISC поддерживается коммерческой организацией OpenCores, но проповедует принципы «*open-source hardware*» (OSHW) и имеет определенную популярность по сей день, не смотря на то, что многие его сторонники перешли на сторону RISC-V.

В те же 2000-е годы для всех известных синтезируемых ядер был портирован GNU C Compiler Toolchain, что широко распахнуло дверь в мир синтезируемых микропроцессоров

перед энтузиастами, любителями и профессиональными программистами. Примерно в это же время для архитектур синтезируемых ядер были портированы различные операционные системы — *FreeRTOS*, Linux и ряд других.

Что же из себя представляет синтезируемый микропроцессор? Как несложно догадаться, это текст на языке HDL, чаще на Verilog, реже на VHDL, но встречается и такое, что разные микроархитектурные части одного ядра могут реализовываться на различных языках, и даже не на Verilog или VHDL, а на специально разработанном языке.

Хорошим примером синтезируемого опенсорсного микропроцессора может служить *VexRiscv* — синтезируемое ядро, написанное на специальном HDL языке под названием *SpinalHDL*. Разработчиком софт-ядра VexRiscv и языка SpinalHDL является швейцарец Шарль Папон. Свою разработку он представил на конкурс *RISC-V SoftCPU Contest* проводимого RISC-V Foundation в 2018 году, где занял первое место и получил приз в $6000 USD за достижение максимальной производительности на ПЛИС Lattice и Microsemi. Далее мы попытаемся разобраться, в чем особенности VexRiscv и зачем автору потребовалось изобретать свой собственный язык описания аппаратуры для создания софт-ядра.

12.1 Синтезируемое вычислительное ядро VexRiscv

VexRiscv — это полностью open source реализация вычислительно ядра архитектуры RISC-V. Это очень гибко конфигурируемое ядро, его можно ужать до применения на очень ограниченном ресурсе и программировать только на ассемблере для «голого железа», а можно легко расширить, включив поддержку кэширования инструкций и данных, поддержку MMU, FPU, атомарных инструкций и соорудить на нем многопроцессорную систему для запуска ОС Linux прямо внутри ПЛИС. Шарль Папон, разработчик VexRiscv, вдохновленный достижениями быстроразвивающейся софтверной индустрии, перенес и применил некоторые её принципы на разработку цифровой аппаратуры, а именно — идею объектно-ориентированных плагинов (**plugin**). В рамках VexRiscv все его компоненты — это плагины, которые подключаются к разным стадиям конвейера. Плагины экспортируют свои структуры данных (сигнальные линии), которые автоматически становятся доступны во всех местах конвейера. Плагины могут предоставлять различные сервисы (**service**) другим плагинам. Плагины можно заменять и расширять. При таком подходе у автора получился гибко настраиваемый микропроцессор, в котором всё, вплоть от счетчика команд (**PC**) и регистрового файла до менеджера обработки «опасностей» (hazards), может быть заменено, расширено и углублено. Но самое интересное, что в погоне за гибкостью и функциональностью автору удалось не потерять в производительности, за что он и бы удостоен приза.

Ниже перечислены некоторые особенности VexRiscv, взятые со страницы README проекта:

1. Ядро VexRiscv поддерживает инструкции следующих расширений спецификации RISC-V:
   - RV32I — стандартный минимальный набор с целочисленными операциями;
   - [M] — целочисленные умножения и деления;
   - [A] — атомарное исполнение операций изменения ячеек памяти;
   - [F] — операции с плавающей запятой (floating-point);
   - [D] — операции с подвижной запятой двойной точности (double-precision floating-point);
   - [C] — сжатые инструкции (compressed instruction set), позволяют существенно уменьшить объем программного кода.

2. Ядро VexRiscv может иметь конвейер от двух до пяти ступеней: [Fetch], Decode, Execute, [Memory], [WriteBack] — ступени, обозначенные в квадратных скобках, могут быть отключены.
3. Замеры производительности ядра показывают от 1.44 DMIPS/MHz до 1.57 DMIPS/MHz, в зависимости от набора включенных фич.
4. VexRiscv оптимизирован для синтеза в ПЛИС и при этом не использует ни единого вендорозависимого IP-блока, т. е. является самодостаточным и легко переносимым.
5. VexRiscv поддерживает шины AXI, Avalon и Wishbone, что позволяет легко строить на базе этого ядра сложные многопроцессорные системы-на-кристалле.
6. Содержит опциональный блок MUL/DIV.
7. Опциональный блок 32-х и 64-х битного FPU.
8. Опциональный и настраиваемый кеш данных (D$) и инструкций (I$).
9. Опциональный MMU для виртуализации памяти.
10. Опциональная поддержка JTAG, что позволяет вести отладку через **OpenOCD** и **GDB**.
11. Опциональная поддержка прерываний и исключений согласно *RISC-V Privileged ISA Specification v1.10*.
12. Две реализации инструкции сдвига: полный однотактовый «full barrel shifter» и многотактовый «shiftNumber».

Результаты синтеза и замеры производительности разных конфигураций ядра VexRiscv:

```
VexRiscv small (RV32I, 0.52 DMIPS/Mhz, no datapath bypass) ->
    Artix 7     -> 240 Mhz 556 LUT 566 FF
    Cyclone V   -> 194 Mhz 394 ALMs
    Cyclone IV  -> 174 Mhz 831 LUT 555 FF
    iCE40       -> 85 Mhz 1292 LC


VexRiscv small and productive with I$ (RV32I, 0.70 DMIPS/Mhz, 4KB-I$)  ->
    Artix 7     -> 220 Mhz 730 LUT 570 FF
    Cyclone V   -> 142 Mhz 501 ALMs
    Cyclone IV  -> 150 Mhz 1,139 LUT 536 FF
    iCE40       -> 66 Mhz 1680 LC


VexRiscv linux balanced (RV32IMA, 1.21 DMIPS/Mhz 2.27 Coremark/Mhz, with cache trashing,
4KB-I$, 4KB-D$, single cycle barrel shifter, catch exceptions, static branch, MMU,
Supervisor, Compatible with mainstream linux) ->
    Artix 7     -> 180 Mhz 2883 LUT 2130 FF
    Cyclone V   -> 131 Mhz 1,764 ALMs
    Cyclone IV  -> 121 Mhz 3,608 LUT 2,082 FF
```

От себя добавлю, что мне удавалось разгонять микропроцессорное ядро VexRiscv в варианте «small and productive with I$» на плате «Карно» до частоты 80 МГц. Более «жирные» ядра стабильно работают на частоте 50МГц на этой же плате.

12.2 Синтезируемые системы-на-кристалле на базе VexRiscv

Прежде чем мы углубимся во внутреннее устройство ядра VexRiscv, нужно сказать вот о чем. Наличие одного лишь синтезируемого микропроцессора мало чего дает пользователю, так как этот микропроцессор требуется обеспечить кодом исполняемой программы и каким-то количеством оперативной памяти для её работы, а чтобы эта программа могла выполнять какие-то полезные действия, требуется оснастить микропроцессор устройствами ввода/вывода. Собственно, любая ЭВМ, по классике — это **процессор**, соединенный с

памятью и **устройствами ввода/вывода** посредством **шины**. Все это вместе помещенное на один кристалл принято называть «Системой-на-Кристалле» или СнК (от англ. «System-on-Chip» или SoC). Таковыми являются подавляющее большинство современных микропроцессоров и микроконтроллеров всех уровней и мастей. Микропроцессоров в чистом виде давно не осталось! Очевидно, что из «материи» ПЛИС можно собирать виртуальные системы-на-кристалле, которые принято называть «синтезируемой СнК» (synthesizable SoC). SpinalHDL вместе с вычислительным ядром VexRiscv предоставляют неплохой набор средств для построения синтезируемых СнК различной сложности, используя периферийные IP-блоки из библиотечной поставки SpinalHDL, и в этом смысле связка SpinalHDL+VexRiscv является самодостаточным решением, хотя и позволяет использовать любые сторонние IP-блоки, написанные на Verilog и VHDL. Вообще, будет правильнее сказать, что ядро VexRiscv является одним из библиотечных IP-блоков в составе фреймворка SpinalHDL.

Шарль Папон не поленился и снабдил пользователей ядра VexRiscv несколькими готовыми заготовками (или шаблонами) для написания своих синтезируемых СнК, это **Murax SoC**, **Briey SoC** и **SaxonSoc**. Первые две СнК входят в репозиторий VexRiscv и при сборке можно выбрать один из вариантов. *SaxonSoc* же более поздняя и более сложная реализация синтезируемой СнК, нацеленная, во-первых, на эксперименты с *Banana Memory Bus* (BMB), во-вторых, на «прощупывание» новых концепций построения СнК на основе «генераторов». Про BMB хочу сказать, что это тоже разработка Шарля Папона — весьма передовая шина, лишенная некоторых недостатков широко используемых шин AXI4 от ARM и Tilelink от SiFive, она проще, решает те же проблемы и является «FPGA friendly».

Рассмотрим каждую из этих СнК поподробней.

Murex SoC

Murex SoC — самая минималистияная СнК по потребляемым ресурсам ПЛИС, может синтезироваться без каких-либо внешних компонентов и предоставляет следующие возможности:

- Работает с ядрами VexRiscv конфигурации RV32I[M].
- Имеет JTAG debugger (Eclipse/GDB/OpenOCD ready).
- Поддерживает On-chip RAM (т. е. RAM, синтезированную внутри ПЛИС).
- Один вход прерываний без контроллера.
- Периферийная шина APB3.
- 32 пиновый GPIO порт.
- Один 16-ти битный prescaler, два 16-ти битных таймера.
- UART с поддержкой FIFO для TX и RX.

СнК Murax SoC предназначена, по большей части, для тестирования самих ядер VexRiscv, но может быть использован для построение несложных микроконтроллерных систем для решения задач промышленной автоматизации. Далее я продемонстрирую один из вариантов такой системы, а также покажу, как можно расширить этоту СнК дополнительными IP-блоками (контроллером прерываний, блоком статической памяти и FastEthernet).

Полная СнК Murax SoC с ядром VexRiscv с «bypass» стадиями при синтезировании для ПЛИС Lattice iCE40 умещается в **2800** логических ячеек и работает на частоте 66 МГц.

Briey SoC

Briey SoC — более навороченная СнК. Посмотрев на структурную схему Briey SoC, изображенную на рис. 20, можно легко заметить, что в основе этой СнК находится популярный кросс-коннект (шина) AXI4, с одной стороны к которому подключается ядро VexRiscv (двумя портами — для инструкций и данных раздельно), с другой — различная периферия и мосты на другие шины. Отличительная особенность данной СнК от Murax SoC состоит в том, что она позволяет подключать к вычислительному ядру различные расширения, а также кеши инструкций и данных. А еще Briey SoC имеет готовый интерфейс для подключения динамической памяти (SDRAM).

Но основное достоинство Briey SoC — это популярная шина AXI4, для которой не сложно найти практически любой периферийный IP-блок, как коммерческий, так и опенсорсный. В составе SpinalHDL, кстати, имеется достаточно богатая библиотека компонентов (IP-блоков), в том числе: FastEthernet (MII), I2C, JTAD, SIO, SPI, UART, USB, VGA, PLIC, мосты в другие шины (APB3, AMBA3, AMBA4, Wishbone) и много чего еще.

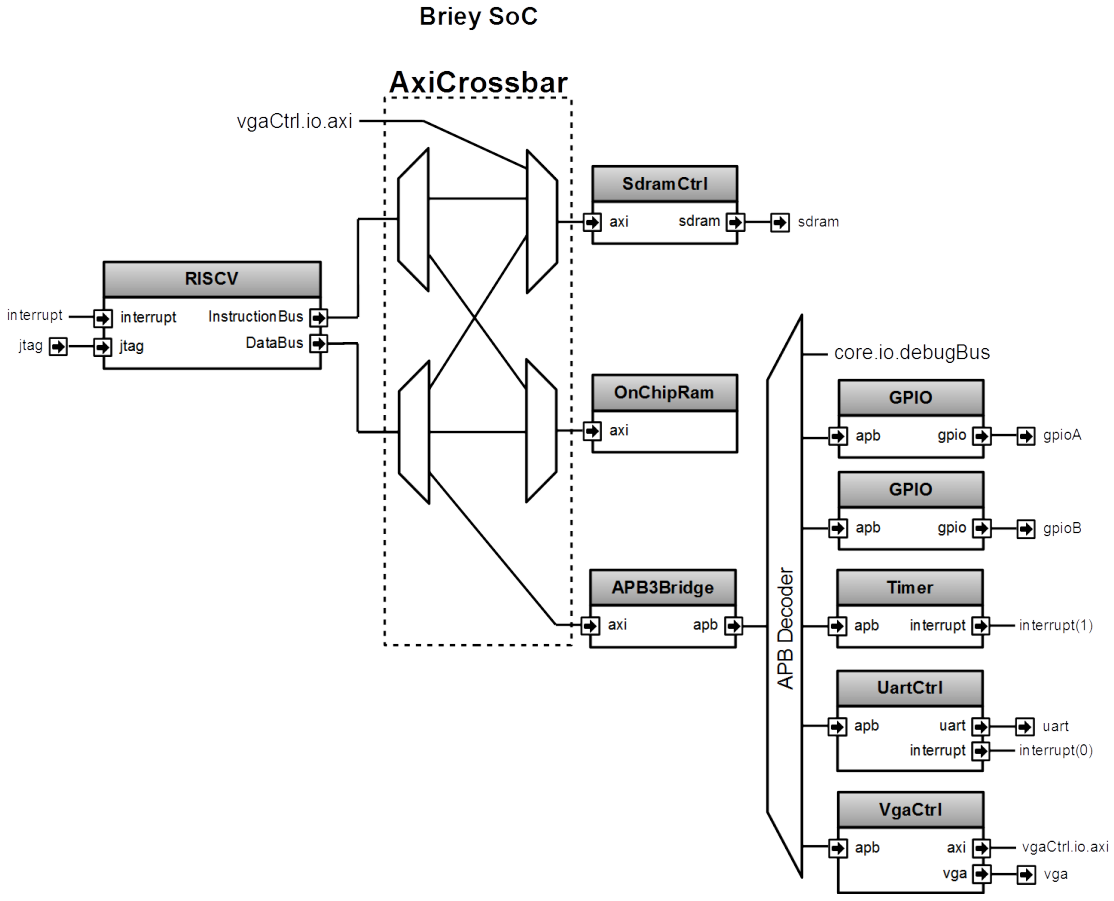


Рис. 20. Структура Briey SoC в составе VexRiscv.

Более подробное описание Briey SoC можно найти в документации по ссылке: https://spinalhdl.github.io/SpinalDoc-RTD/dev/SpinalHDL/miscelenea/lib/briey/hardware_toplevel.html

SaxonSoc

SaxonSoc — многопроцессорная СнК позволяющая запускать Linux и, наверное, этим все сказано. Замечу лишь, что в основе SaxonSoc находится разработанный автором свой кросс-коннект Banana Memory Bus, к которому подключаются периферийные устройства, а также мосты в другие шины. Один из вариантов конфигурации SaxonSoc представлен на рис. 21.

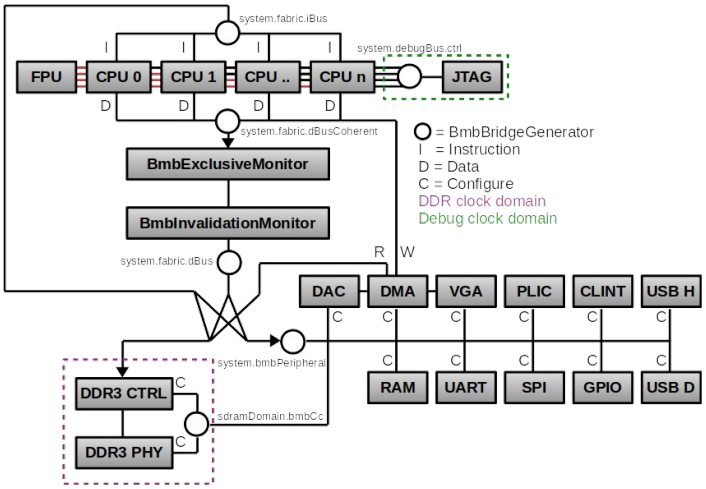


Рис. 21. Вариант конфигурации SaxonSoc.

SaxonSoc находится в отдельном репозитории по адресу: `https://github.com/SpinalHDL/SaxonSoc`

Видео демонстрация SaxonSoc от автора: VexRiscv running Linux, Doom and OpenTTD on FPGA

Вообще, понятие СнК в рамках связки SpinalHDL/VexRiscv очень размытое. Можно взять любую из этих трех готовых СнК, убрать из неё всё ненужное, добавить нужное и получить структуру синтезируемой ЭВМ под свои задач. Далее мы этим и займемся, но сначала рассмотрим, что же такое SpinalHDL и чем он так хорош.

13. Язык описания аппаратуры SpinalHDL

Язык описания аппаратуры SpinalHDL был разработан в 2014 году. Это язык позволяющий описывать цифровые схемы на уровне регистровых передач (RTL). SpinalHDL представляет собой набор классов и библиотек для языка программирования Scala, который в свою очередь входит в экосистему ЯП Java, иными словами SpinalHDL — это фреймворк. Но не спешите воротить нос. При работе со SpinalHDL программировать на Scala вам скорее всего не придется, так как от Scala в SpinalHDL остались, пожалуй, только фигурные скобки. SpinalHDL перегружает и переопределяет действия многих операторов, делая синтаксис очень близким к VHDL, но при этом существенно менее многословным и лаконичным, т. е. текста писать придется существенно меньше. Вот что сам автор говорит про изобретенный им язык:

> *Традиционно языки HDL [имеется в виду Verilog и VHDL] предлагают небольшой выбор высокоуровневых абстракций, что приводит к длинным и повторяющимся описаниям сигнальных линий. Разработка сложной аппаратуры на таких языках требует много времени, а отладка и сопровождение очень затруднено из-за огромного количества запутанных сигналов. SpinalHDL позволяет разработчикам контролировать сложность посредством высокоуровневых абстракций, используя модели объектно-ориентированного и функционального программирования, создавая краткое самодокументируемое описание аппаратуры…*
>
> – Шарль Папон, автор и создатель SpinalHDL.

От себя добавлю, что в среде разработчиков на SystemVerilog и VHDL есть очень много ритуальных действий, например, описания цепей асинхронного сброса, создание отдельного процесса **process/always** для каждого регистра и т. д. Всего этого в SpinalHDL нет.

SpinalHDL был не первым HDL на основе Scala. Существует и успешно развивается такой HDL язык как Chisel — брат близнец, изначально созданный в UC Berkeley и долгое время используемый в академической среде. Chisel в какой-то момент был подхвачен компанией SiFive, теми же людьми, что разрабатывали архитектуру RISC-V, и сейчас имеет определенное хождение в кругах ASIC разработчиков (вычислительные ядра от SiFive написаны на Chisel). В этой связи многие задают автору языка SpinalHDL вопрос — «а нафига, если есть/был Chisel ?». И вот его ответ:

> *При создании SpinalHDL я, безусловно, был вдохновлен языком Chisel, но не подумайте, что SpinalHDL является прямым его ответвлением [fork-ом]. Chisel создавался и используется в основном для разработки микросхем (ASIC), он находится в состоянии активной разработки с постоянно изменяющимся API. SpinalHDL же нацелен больше на разработку для ПЛИС и, в общем-то, уже «заморожен» [не претерпевает изменений]. Язык активно сопровождается — устраняются ошибки, иногда добавляются новые фичи, но основное ядро языка останется неизменным на ближайшее будущее.*
>
> – Шарль Папон, автор и создатель SpinalHDL.

SpinalHDL — это метаязык. Это означает, что код, написанный на SpinalHDL, должен быть скомпилирован и исполнен, а в процессе исполнения он генерирует текст на другом традиционном HDL языке. То есть, после запуска программы на SpinalHDL, результатом её работы становится текстовый файл (или набор файлов) на языке Verilog или VHDL (поддерживаются оба), которые далее подаются на вход синтезатору, например, Yosys.

SpinalHDL создавался для работы с уже существующими инструментами и богатой кодовой базой. В языке имеется возможность интегрировать в дизайн любой модуль написанный на Verilog или VHDL, обернув его в так называемый IP «черный ящик» («blackbox»), что позволяет далее использовать его как компонент SpinalHDL.

SpinalHDL содержит все необходимые средства для моделирования, симуляции и верификации дизайна. Вспомним, что SpinalHDL это Scala — очень мощный ООП. Это позволяет вести разработку сложных дизайнов на одном единственном языке, не прибегая к помощи других инородных средств, как это часто бывает среди разработчиков, использующих традиционные HDL — моделирование на C++, масса скриптов на Perl, Shell и Tcl и уж потом код на Verilog или VHDL.

Подобным же свойством обладает другой широко известный HDL язык — SystemC. Этот HDL, аналогично SpinalHDL, является набором классов к языку C++ и также является метаязыком — очень мощный инструмент, используемый верификаторами. На мой взгляд, код на SystemC — это что-то ужасное: он не читаем, его очень много, синтаксис нелогичен и пестрит макросами. При решении этих же задач на SpinalHDL код получается существенно короче, проще для понимания и просто красивее. На Habr-е, кстати, есть отличная статья про SystemC от пользователя **@Daffodil**:

Разработка цифровой аппаратуры на C++/SystemC глазами SystemVerilog программиста

Всё это дает толчок к созданию языков мета-мета уровня. Например, известен язык (фреймворк) LiteX, который является библиотекой для Python, позволяющей описывать

цифровые схемы используя готовые блоки, в том числе на SpinalHDL, и получать синтезируемый код.

Общими мазками процесс разработки на SpinalHDL выглядит следующим образом:

1. Разработчик описывает свой RTL дизайн на SpinalHDL и получает **.scala** файл (или файлы).

2. Компилирует **.scala** код с помощью **Scala Build Tools** в исполняемый **JVM**.

3. Исполняет полученный байт-код с помощью Java машины и получает на выходе набор файлов на традиционном Verilog (или VHDL).

4. Подает сгенерированные Verilog/VHDL файлы на вход синтезирующему тулу (Yosys, Quartus, Vivado и т. д.), получая на выходе битстрим для ПЛИС.

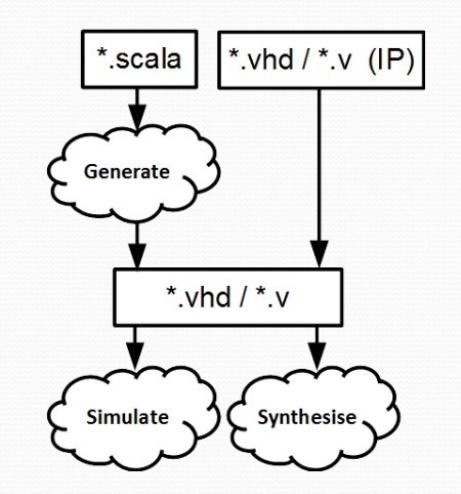


Рис. 22. Процесс разработки на SpinalHDL в общих мазках.

Автор языка утверждает, что разработка на SpinalHDL не вносит в логику схемы никакой дополнительной избыточности (no logic overhead), а сгенерированный Verilog/VHDL код может быть успешно симулирован и верифицирован в том числе стандартными тулами. В сгенерированном коде сохраняются именования сигналов, регистров и модулей, описанные в коде на SpinalHDL.

Немного забегая вперед, для того чтобы продемонстрировать насколько SpinalHDL удобный и лаконичный язык, заточенный именно под RTL, я приведу пример схемы и кода, взятых прямо из документации. Предположим, что у нас имеется следующая цифровая схема (рис. 23), в которой присутствуют комбинационная логика и два регистра-счетчика: один с асинхронным сбросом, другой без такового.

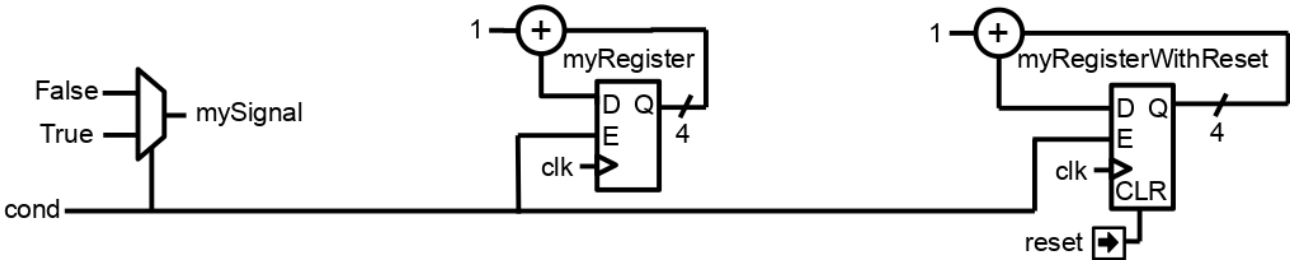


Рис. 23. Пример цифровой схемы. Взято из документации на SpinalHDL.

При разработке на традиционных HDL языках разработчик обычно описывает несколько отдельных «процессов» с указанием списка «чувствительности» (перечень сигналов,

которые, по его мнению, будут изменяться в процессе), при этом для комбинационной и последовательностной логики процессы как правило разделяют. Так же, на каждый из двух регистров описывается отдельный процесс, так как в данном случае регистры разных типов (со сбросом и без). Таким образом, у нас получается код на VHLD из трех процессов:

```
signal mySignal : std_logic;
signal myRegister : unsigned(3 downto 0);
signal myRegisterWithReset : unsigned(3 downto 0);

process(cond)
begin
    mySignal <= '0';
    if cond = '1' then
        mySignal <= '1';
    end if;
end process;

process(clk)
begin
    if rising_edge(clk) then
        if cond = '1' then
            myRegister <= myRegister + 1;
        end if;
    end if;
end process;

process(clk,reset)
begin
    if reset = '1' then
        myRegisterWithReset <= 0;
    elsif rising_edge(clk) then
        if cond = '1' then
            myRegisterWithReset <= myRegisterWithReset + 1;
        end if;
    end if;
end process;
```

На SpinalHDL эта же самая цифровая схема будет описана следующим образом:

```
val mySignal             = Bool()
val myRegister           = Reg(UInt(4 bits))
val myRegisterWithReset  = Reg(UInt(4 bits)) init(0)

mySignal := False
when(cond) {
    mySignal            := True
    myRegister          := myRegister + 1
    myRegisterWithReset := myRegisterWithReset + 1
}
```

Видно, что, во-первых, в коде на SpinalHDL нет процессов (они скрыты внутри типов конструкторов соответствующих переменных). Во-вторых, весь код комбинационной и последовательностной логики описывается в одном блоке под одним условным оператором when(). Я уверен, что для большинства разработчиков, давно пишущих код на Verilog и/или VHDL, такое решение будет просто шокирующим. На SpinalHDL такой стиль является нормой, он позволяет существенно сократить число строк кода, а сам код сделать более понятным и читаемым.

13.1 Базовые конструкции языка SpinalHDL

Детально описывать все конструкции и возможности SpinalHDL я не буду, а пройдусь по основным моментам, чтобы у читателя составилось общее представление о языке. Заинтересовавшиеся читатели могут найти полный текст документации по ссылке:

https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html

В SpinalHDL приняты аналогичные правила работы с сигналами, как в Verilog и VHDL, а именно:

1. Все сигналы и регистры работают параллельно, порядок присваивания не имеет значения.
2. Присваивание комбинационному сигналу выполняется (становится доступно) в этом же такте.
3. Присваивание регистру выполняется в следующем такте того домена, к которому принадлежит регистр.
4. При многократном присваивании в один и тот же сигнал или регистр, в расчет берется самое последнее присваивание (принцип «last assignment wins»).
5. С любым регистром или сигналом можно работать как с объектом.

Типы данных

SpinalHDL является строго типизированным языком, в котором определены пять базовых типов и два составных типа. Базовые типы это: **Bool**, **Bits**, **UInt** для беззнаковых, **SInt** для знаковых и **Enum**. Два составных типа это **Bundle** — сложная структура сигналов, и **Vec** — массив однотипных сигналов. Для базовых типов можно указывать размерность в битах.

Для преобразования типов данных существуют методы вида **.asBits()**, **.asUInt()**, **.asSInt()** и **.asBool()**.

Регистры

Для описания регистров в SpinalHDL используется специальные конструкции:

Reg(type) — описывает регистр указанного базового типа **type**.

RegInit(resetValue) — описывает регистр с начальным значением **resetValue**.

RegNext(nextValue) — описывает регистр, который каждый новый такт принимает значение, вычисляемое выражением **nextValue**.

RegNextWhen(nextValue, cond) — аналогично **RegNext**, но при условии, что выражение **cond** является истиным.

Присваивание :=

В SpinalHDL для синтезируемых переменных используется один тип оператора присваивания. Будет ли он блокирующим или неблокирующим определяется типом переменной. Существует модификация оператора присваивания: <> - позволяет присваивать сложные структуры поименованных сигналов (бандлы), учитывая направленность сигналов внутри бандла.

В вышеприведенном примере переменная **mySignal** описывает сигнал размерностью один бит, а переменные **myRegister** и **myRegisterWithReset** описывают регистры по 4 бита. Дополнительный модификатор **init()** указывает на то, что данный регистр имеет асинхронный сброс и устанавливает значение регистра по умолчанию. Установка дефолтных значений возможна для большинства современных ПЛИС и такой код успешно синтезируется.

Сравнение === и =/=

В SpinalHDL для сравнения двух сигналов используется оператор ===. Оператор с противоположной логикой имеет вид =/=. Пример:

```
io.flag  := (counter === 0) | io.cond1
```

Конкатенация ##

```
io.result := io.high ## io.low
```

Арифметические и логические операторы

Как и в других HDL языках, в SpinalHDL имеется множество операторов для описания арифметических и логических операций над сигналами. В таблицах ниже приведена часть из них:

Operator	Description	Return
x + y	Addition	T(max(w(x), w(y) bits)
x - y	Subtraction	T(max(w(x), w(y) bits)
x * y	Multiplication	T(w(x) + w(y) bits)
x > y	Greater than	Bool
x >= y	Greater than or equal	Bool
x < y	Less than	Bool
x <= y	Less than or equal	Bool
x >> y	Arithmetic shift right, y : Int	T(w(x) - y bits)
x >> y	Arithmetic shift right, y : UInt	T(w(x) bits)
x << y	Arithmetic shift left, y : Int	T(w(x) + y bits)
x << y	Arithmetic shift left, y : UInt	T(w(x) + max(y) bits)
x.resize(y)	Return an arithmetic resized copy of x, y : Int	T(y bits)

Operator	Description	Return type
!x	Logical NOT	Bool
x && y x & y	Logical AND	Bool
x || y x | y	Logical OR	Bool
x ^ y	Logical XOR	Bool
~x	Logical NOT	Bool
x.set[()]	Set x to True	Unit (none)
x.clear[()]	Set x to False	Unit (none)
x.setWhen(cond)	Set x when cond is True	Bool
x.clearWhen(cond)	Clear x when cond is True	Bool
x.riseWhen(cond)	Set x when x is False and cond is True	Bool
x.fallWhen(cond)	Clear x when x is True and cond is True	Bool

Модификаторы in, out, master, slave

Модификаторы применяются к синтезируемым переменным (сигналам) и позволяют определить их логическое назначение, а также семантику для оператора <>.

Компоненты

В SpinalHDL компонент — это законченный логический блок цифровой схемы (аналогично **module** в Verilog). Любой компонент является классом-наследником от класса **Component**. Компонент может содержат член класса **io** типа **Bundle** — сложная структура, описывающая сигналы ввода/вывода данного компонента. Например, компонент (модуль) из схемы представленной на рис. 24 на SpinalHDL будет выглядеть следующим образом:

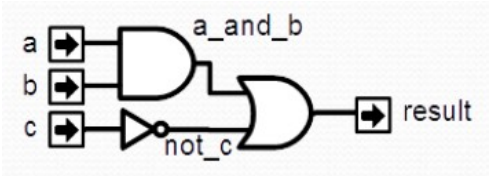


Рис. 24. Комбинационная схемы как компонент.

```
class MyComponent extends Component {
	val io = new Bundle {
		val a = in Bool
		val b = in Bool
		val c = in Bool
		val result = out Bool
		}
	val a_and_b = Bool
	a_and_b := io.a & io.b
	val not_c = ! io.c
	io.result := a_and_b | not_c
}
```

Этот код на SpinalHDL в результате генерации даст нам следующий код на VHDL:

```
entity MyComponent is
	port(
	io_a : in std_logic;
	io_b : in std_logic;
	io_c : in std_logic;
	io_result : out std_logic
	);
end MyComponent;

architecture arch of MyComponent is
	signal a_and_b : std_logic;
	signal not_c : std_logic;
begin
	io_result <= (a_and_b or not_c);
	a_and_b <= (io_a and io_b);
	not_c <= (not io_c);
end arch;
```

Области

Для удобства структурирования кода внутри компонента его части можно разбивать на области, используя конструкцию **Area {...}**, которая имеет следующий вид:

```
class TopLevel extends Component {
	//…
	val logicArea = new Area {
		val flag = Bool
	}
	val fsmArea = new Area {
		when(logicArea.flag) {
			//…
		}
	}
}
```

Инкапсулирование компонентов

Один компонент может быть инкапсулирован внутрь другого, при этом связь интерфейсных сигналов осуществляется оператором одиночного **:=** или группового <> присваивания. Пример:

```
class SubComponent extends Component{
	val io = new Bundle {
		val input = in Bool
		val result = out Bool
	}
	...
}
class TopLevel extends Component {
	val io = new Bundle {
		val a = in Bool
```

```
                val b = in Bool
                val result = out Bool
        }
        val sub = new SubComponent
        sub.io.input := io.a
        io.result := sub.io.result | io.b
}
```

Оператор when(), elsewhen() и otherwise()

Данные операторы позволяют выполнять сравнение и условное назначение сигналов:

```
when(io.conds(0)){
        io.result := 2
        when(io.conds(1)){
                io.result := 1
        }
} elsewhen(1) {
        io.result := 3
} otherwise {
        io.result := 0
}
```

Оператор switch()

```
switch(state) {
        is(MyEnum.state0) {
                ...
        }

        is(MyEnum.state1) {
                ...
        }

        default{
                ...
        }
}
```

Оператор mux()

Упрощенная запись для описания мультиплексоров:

```
val bitwiseSelect = UInt(2 bits)
val bitwiseResult = bitwiseSelect.mux(
  0 -> (io.src0 & io.src1),
  1 -> (io.src0 | io.src1),
  2 -> (io.src0 ^ io.src1),
  default -> (io.src0)
)
```

Память Mem()

Массивы ячеек памяти могут быть определены специальным компонентом **Mem**(), который имеет встроенные методы для синхронного и асинхронного доступа:

```
//Memory of 1024 Bools
val syncRam = Mem(Bool, 1024)
val asyncRam = Mem(Bool, 1024)

//Write them
syncRam(5) := True
asyncRam(5) := True

//Read them
val syncRam = mem.readSync(6)
val asyncRam = mem.readAsync(4)
```

Функции и пользовательские операторы

В SpinalHDL можно определять и переопределять операции над объектами (сигналами, регистрами и сложными структурами данных). Предположим, что у нас имеется некий комплексный сигнал **Color**, состоящий из трех компонент: **r**, **g** и **b**. Тогда можно определить класс для работы с этим сигналом цветности, а внутри него определить оператор + для покомпонентного суммирования цветов. Такой класс приведен ниже.

```
case class Color(channelWidth: Int) extends Bundle {
	val r,g,b = UInt(channelWidth bits)

	def +(that: Color): Color = {
		val result = Color(channelWidth)
		result.r := this.r + that.r
		result.g := this.g + that.g
		result.b := this.b + that.b
		return result
	}
}
```

В приведенном выше примере используется параметризация размерности составляющих сигналов, которая определяется параметром **channelWidth** при инстанциировании сигнала класса **Color**.

13.2 FSM, потоки, конвейеры, тактовые домены

Машины состояний (FSM)

Машины состояний в SpinalHDL могут быть определены как традиционным методом с помощью **Enum** и **switch**, так и специализированными средствами (классы **State** и **StateMachine**), позволяющими легко создавать вложенные и множественно-вложенные FSM с минимальным количеством кода. SpinalHDL не предъявляет каких-либо требований к стилю изложения FSM, их можно и нужно описывать так, как это делается на языках программирования. Внутри обработки состояний можно смешивать как сигналы, так и регистры, что контрастирует с принятыми принципами описания FSM в мире Verilog/VHDL. Ниже приведены два примера с использованием класса **StateMachine**.

```
// FSM style A

val fsm = new StateMachine{
	io.result := False
	val counter = Reg(UInt(8 bits)) init (0)
	val stateA : State = new State with EntryPoint{
		whenIsActive (goto(stateB))
	}
	val stateB : State = new State{
		onEntry(counter := 0)
		whenIsActive {
			counter := counter + 1
			when(counter === 4){
				goto(stateC)
			}
		}
		onExit(io.result := True)
	}
	val stateC : State = new State{
		whenIsActive (goto(stateA))
	}
}


// FSM style B

val fsm = new StateMachine{
```

```
        val stateA = new State with EntryPoint
        val stateB = new State
        val stateC = new State
        val counter = Reg(UInt(8 bits)) init (0)
        io.result := False

        stateA
               .whenIsActive (goto(stateB))

        stateB
               .onEntry(counter := 0)
               .whenIsActive {
                      counter := counter + 1
                      when(counter === 4){
                             goto(stateC)
                      }
               }
               .onExit(io.result := True)
        stateC
               .whenIsActive (goto(stateA))
}
```

Мне лично больше по душе вариант **A** из-за наличия фигурных скобок — они четко дают понять, где заканчивается обработка одного состояния и начинается обработка следующего. Вариант **B** это чистый ООП подход из длинных конструкций соединенных оператором точка. :-)

Интерфейс потока Flow

Класс **Flow** позволяет описать простой valid/payload протокол взаимодействия по шине, в котором передающая сторона (master устройство) подает данные на комплексный сигнал payload, при этом устанавливая valid, а принимающая сторона (slave устройство) считывает данные с payload и не имеет возможности остановить работу шины. Внутри библиотек SpinalHDL описание класса Flow выглядит примерно следующим образом:

```
case class Flow[T <: Data](payloadType: T) extends Bundle {
       val valid = Bool
       val payload = cloneOf(payloadType)
}
```

Потоки можно соединять в цепочки операторами >> и << образуя сложные комбинационные схемы, либо операторами <-< или >-> образуя конвейеры (т. е. разделенные регистрами).

Пример использования потока **Flow**:

```
case class FlowExample() extends Component {
  val io = new Bundle {
    val request = slave(Flow(Bits(8 bit)))
    val answer = master(Flow(Bits(8 bit)))
  }
  val storage = Reg(Bits(8 bit))

  val fsm = new StateMachine {
    io.answer.setIdle()

    val idle: State = new State with EntryPoint {
      whenIsActive {
        when(io.request.valid) {
          storage := io.request.payload
          goto(sendEcho)
        }
      }
    }

    val sendEcho: State = new State {
      whenIsActive {
        io.answer.push(storage)
```

```
        goto(idle)
      }
    }
  }
}
```

В данном примере описана машина состояний, которая постоянно принимает данные по линии **request** и возвращает их же в ответе **answer**, сохраняя полученные данные во внутренний регистр. Так как сигналы request и answer объявлены как потоки, то весь код машины состояний может быть заменен одним простым выражением:

```
io.answer <-< io.request
```

Интерфейс потока Stream

Класс **Stream** позволяет определить протокол вида valid/ready/payload с «рукопожатием», в котором принимающая сторона (slave) начинает обработку данных **payload** по сигналу **valid**, при этом поток останавливается до подачи принимающей стороной сигнала готовности **ready**. Внутри библиотек SpinalHDL поток Stream определяется следующим образом:

```
case class Stream[T <: Data](payloadType: T) extends Bundle {
       val valid = Bool
       val ready = Bool
       val payload = cloneOf(payloadType)
}
```

В цифровой схемотехнике такой протокол очень часто используется, например, для работы с FIFO или с простой периферией которая требует ожидания выполнения запроса.

Пример использования потока **Stream**:

```
class StreamFifo[T <: Data](dataType: T, depth: Int) extends Component {
  val io = new Bundle {
    val push = slave Stream (dataType)
    val pop = master Stream (dataType)
  }
  ...
}
```

Как и в случае с потоком **Flow**, потоки **Stream** можно объединять операторами << и >> или <-< и >->, создавая таким образом сложные конвейеры обработки данных. По мимо этого у потока **Stream** имеется ряд управляющих методов, позволяющих останавливать и перезапускать обработку данных в конвейере (**haltWhen**, **throwWhen** и т.д).

В документации на SpinalHDL приводится такой пример кода:

```
case class RGB(channelWidth : Int) extends Bundle {
  val red   = UInt(channelWidth bits)
  val green = UInt(channelWidth bits)
  val blue  = UInt(channelWidth bits)

  def isBlack : Bool = red === 0 && green === 0 && blue === 0
}

val source = Stream(RGB(8))
val sink   = Stream(RGB(8))
sink <-< source.throwWhen(source.payload.isBlack)
```

Эквивалентная схема к этому коду приведена на рис. 25 ниже.

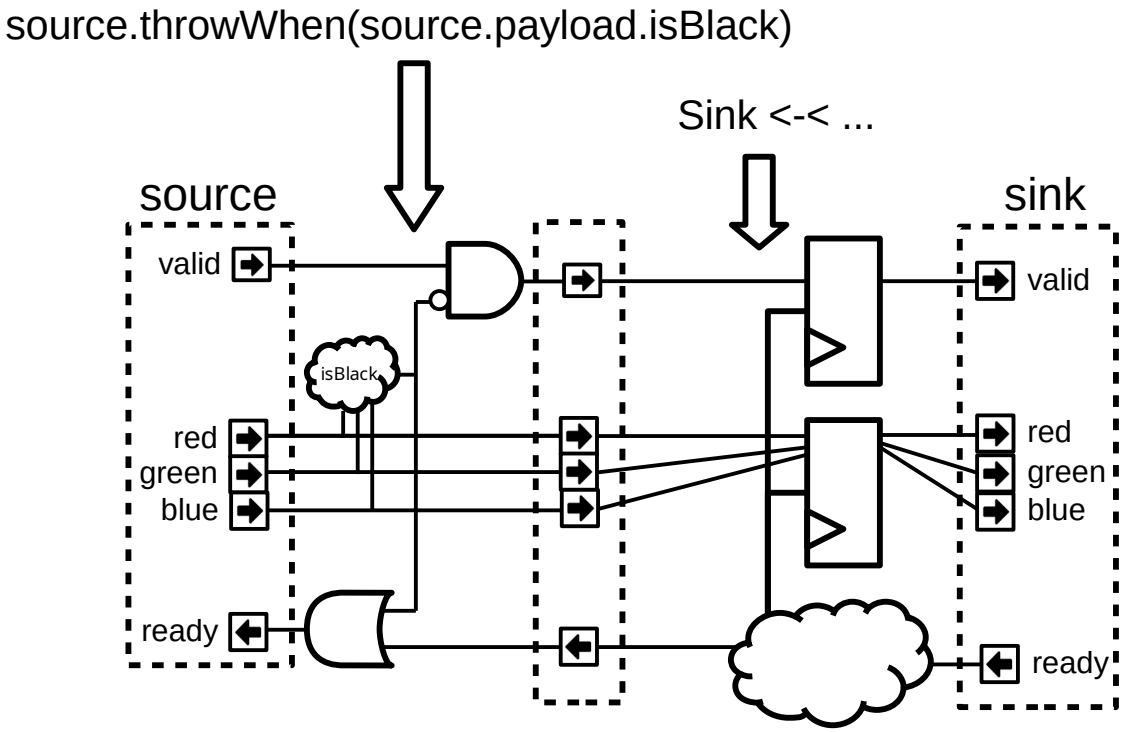


Рис. 25. Пример построения конвейера обработки данных с использованием потока Stream.

Построение конвейерного байпаса с применением LatencyAnalysis()

Часто в цифровых схемах при построении конвейерной обработки возникает необходимость запустить часть данных в обход нескольких ступеней конвейера, число которых неизвестно или может изменяться в процессе работы над схемой, например, как изображено на рис. 26.

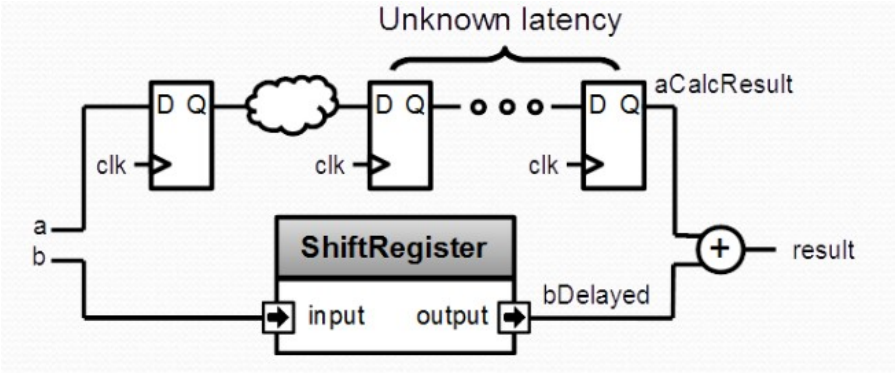


Рис. 26. Конвейер неизвестной длины с байпассом на сдвиговом регистре.

Для того, чтобы обеспечить обход такого конвейера, в SpinalHDL имеется функция **LatencyAnalysis**(), вычисляющая кратчайший путь по описанной схеме, выраженный в количестве тактов. Тогда построение байпаса для такого конвейера будет выглядеть следующим образом:

```
val a = UInt(8 bits)
val b = UInt(8 bits)
val aCalcResult = complicatedLogic(a)
```

```
val aLatency = LatencyAnalysis(a,aCalcResult)
val bDelayed = Delay(b,cycleCount = aLatency)
val result = aCalcResult + bDelayed
```

Здесь **complicatedLogic** представляет собой объект, в котором скрывается схема конвейера: это может быть **Component**, **Area** или какая-то еще сущность. Функция **Delay**() добавляет последовательно указанное количество регистров, рассчитанное функцией **LatencyAnalysis**(), создавая таким образом сдвиговый регистр нужной глубины для обхода конвейера.

Тактовые домены и CDC

Любой цифровой дизайн является простым до тех пор, пока в нём не появляются участки схемы, тактируемые от разных источников тактового сигнала, причем это могут быть сигналы «как бы» одной и той же частоты, но если они поддерживаются не одним и тем же «фундаментом» (fundamental), то в тактовых сигналах двух разных источников очень быстро накапливается разность фаз, что ведет к появлению проблем метастабильности и потере данных. Сам процесс передачи данных из одного участка схемы в другой с отличным источником тактового сигнала принято называть «пересечением тактовых доменов» («Clock Domain Crossing» или CDC). В ряде случаев проблемы CDC можно решить достаточно просто, введя серию последовательных буферных регистров, но часто на стыке между двумя такими блоками требуется установка синхронизирующих устройств, например — FIFO со счетчиком на основе кода Грея. Не стану вдаваться в подробности решения этой непростой задачи, вместо этого приведу ссылку на известную статью Клиффорда Каммингса «Clock Domain Crossing (CDC) Design & VerificationTechniques Using SystemVerilog», в которой автор предлагает несколько различных решений, в том числе и с помощью FIFO.

Решение проблем CDC может выглядеть весьма громоздким, а разработчику приходится всё время держать в голове этот момент и писать код так, чтобы случайно не потерять момент из виду и не напортачить. Помимо этого, часто код приходится рубить на куски ifdef-ами — на синтезируемый и симулируемый. В итоге код превращается в отличную вермишель.

Чтобы справиться с вермишелью, SpinalHDL предлагает следующее решение — весь код нужно ассоциировать с каким-то соответствующим тактовым доменом, после чего, при генерации SpinalHDL будет автоматически подставлять требуемые тактовые сигналы и сигнал сброса в нужные участки кода на Verilog/VHDL, а также предпринимать действия по недопущению ошибки при CDC. Для этого имеется следующий инструментарий:

Класс **ClockDomain** — описывает параметры нового тактового домена, его сигналы тактирования и сброса. Надо заметить, что в SpinalHDL сигналы сброса и тактирования, как неразлучная пара, всегда ходят вместе.

Класс **ClockingArea** — в него включают весь код, относящийся к определенному тактовому домену.

Пример описания домена:

```
// Configure new clock domain
val myClockDomain = ClockDomain(
  clock  = io.clk,
  reset  = io.resetn,
  config = ClockDomainConfig(
    clockEdge        = RISING,
    resetKind        = ASYNC,
    resetActiveLevel = LOW
  )
)

// Define an Area which uses myClockDomain
val myArea = new ClockingArea(myClockDomain) {
```

```
    val myReg = Reg(UInt(4 bits)) init(7)

    myReg := myReg + 1

    io.result := myReg

    // ...
  }
```

Структура **ClockDomainConfig** задает ряд параметров, определяющих поведение SpinalHDL при генерации кода для указанного домена: тип фронта тактового сигнала и тип сигнала сброса.

По умолчанию, если привязка кода к тактовым доменам не выполнена, то весь код автоматически привязывается к дефолтному домену **ClockDomain** со следующими настройками:

- Clock : rising edge
- Reset : asynchronous, active high
- No clock enable

CDC переход между двумя тактовыми доменами выполненный с помощью буферных регистров может выглядеть так:

```
class CrossingExample(clkA : ClockDomain,clkB : ClockDomain) extends Component {
  val io = new Bundle {
    val dataIn  = in Bool()
    val dataOut = out Bool()
  }

  // sample dataIn with clkA
  val area_clkA = new ClockingArea(clkA) {
    val reg = RegNext(io.dataIn) init(False)
  }

  // BufferCC to avoid metastability issues
  val area_clkB = new ClockingArea(clkB) {
    val buf1   = BufferCC(area_clkA.reg, False)
  }

  io.dataOut := area_clkB.buf1
}
```

Функция **BufferCC**() добавляет в генерируемый код два и более буферных регистра, тактируемых от тактового сигнала соответствующего домена, и работает только для однобитных сигналов. Для более сложных переходов имеется класс **StreamFifoCC**, с помощью которого можно легко описать CDC переход для сигналов любой сложности. Пример:

```
val clockA = ClockDomain(???)
val clockB = ClockDomain(???)
val streamA,streamB = Stream(Bits(8 bits))
//...
val myFifo = StreamFifoCC(
  dataType  = Bits(8 bits),
  depth     = 128,
  pushClock = clockA,
  popClock  = clockB
)
myFifo.io.push << streamA
myFifo.io.pop  >> streamB
```

На этом, пожалуй, закончим наш краткий обзор языка SpinalHDL и перейдем к практическим упражнениям.

13.3 Установка SpinalHDL

Далее я буду рассматривать вариант подготовки среды для работы из командной строки в UNIX системах с использованием опенсорсного тулчейна Yosys и симулятора Verilator. Процесс установки и проверки работоспособности этих тулов описан в главе «*6. Как установить утилиты тулчейна Yosys*». SpinalHDL работает как на Linux, так и в BSD системах.

Как уже отмечалось выше, SpinalHDL это набор библиотек классов для языка программирования Scala, а это значит, что процесс сборки проекта целесообразно выполнять с помощью Scala Build Tools — **sbt,** все примеры и шаблоны автор SpinalHDL приводит для SBT. Для Scala, в свою очередь, требуется Java DK. Поэтому последовательно установим Java, Scala и SBT.

```
rz@devbox:~$ sudo apt-get update
rz@devbox:~$ sudo apt-get install openjdk-8-jdk
rz@devbox:~$ sudo apt-get install scala
rz@devbox:~$ sudo apt-get install sbt
```

Если в вашем дистрибутиве Linux по умолчанию нет SBT, то нужно разобраться, как его добавить. Для старых версий Ubuntu это делается так:

```
rz@devbox:~$ echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a
/etc/apt/sources.list.d/sbt.list
rz@devbox:~$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
2EE0EA64E40A89B84B2DF73499E82A75642AC823
rz@devbox:~$ sudo apt-get update
rz@devbox:~$ sudo apt-get install sbt
```

На ОС FreeBSD 13.x установить всё это добро можно одной командой, так как Java, Scala и SBT уже присутствуют в репозитории с пакетами:

```
rz@butterfly:~ % sudo pkg install scala sbt
```

В случае возникновения затруднений с установкой SBT, рекомендую к прочтению статью на Хабре «Введение в sbt» от пользователя **@antaresm**.

Далее необходимо клонировать из репозитория шаблон SpinalHDL проекта, после чего зайти в каталог **SpinalTemplateSbt** и запустить генерацию Verilog кода для этого демонстрационного (шаблонного) проекта. SBT добудет все необходимые зависимости и установит их локально для пользователя, от которого выполняется команда, примерно вот так:

```
rz@devbox:~ $ git clone https://github.com/SpinalHDL/SpinalTemplateSbt.git
Cloning into 'SpinalTemplateSbt'...
...
Resolving deltas: 100% (203/203), done.

rz@devbox:~/SpinalTemplateSbt $ sbt "runMain projectname.MyTopLevelVerilog"
[info] welcome to sbt 1.6.0 (OpenJDK BSD Porting Team Java 1.8.0_392)
[info] loading settings for project spinaltemplatesbt-build from plugins.sbt ..
[info] loading project definition from /usr/home/rz/SpinalTemplateSbt/project
[info] loading settings for project projectname from build.sbt ...
[info] set current project to projectname (in build file:/usr/home/rz/SpinalTemplateSbt/)
[info] running (fork) projectname.MyTopLevelVerilog
[info] [Runtime] SpinalHDL v1.10.0    git head : 270018552577f3bb8e5339ee2583c9c22d324215
[info] [Runtime] JVM max memory : 4512.0MiB
[info] [Runtime] Current date : 2024.01.24 03:53:05
[info] [Progress] at 0.000 : Elaborate components
```

```
[error] [Thread-2] INFO net.openhft.affinity.Affinity - Using dummy affinity control
implementation
[info] [Progress] at 0.190 : Checks and transforms
[info] [Progress] at 0.271 : Generate Verilog
[info] [Done] at 0.323
[success] Total time: 2 s, completed Jan 24, 2024 3:53:06 AM
```

В результате, в файле **./hw/gen/MyTopLevel.v** мы получим код на языке Verilog:

```
rz@devbox:~/SpinalTemplateSbt $ ll ./hw/gen/MyTopLevel.v
-rw-r--r--  1 rz  rz  726 Jan 24 03:53 ./hw/gen/MyTopLevel.v


rz@devbox:~/SpinalTemplateSbt $ cat ./hw/gen/MyTopLevel.v

// Generator : SpinalHDL v1.10.0    git head : 270018552577f3bb8e5339ee2583c9c22d324215
// Component : MyTopLevel
// Git hash  : 51249d4c6e34bfcda55c0332356bd15cf5adbc1d

`timescale 1ns/1ps

module MyTopLevel (
  input  wire          io_cond0,
  input  wire          io_cond1,
  output wire          io_flag,
  output wire [7:0]    io_state,
  input  wire          clk,
  input  wire          reset
);

  reg        [7:0]    counter;

  assign io_state = counter;
  assign io_flag = ((counter == 8'h00) || io_cond1);
  always @(posedge clk or posedge reset) begin
    if(reset) begin
      counter <= 8'h00;
    end else begin
      if(io_cond0) begin
        counter <= (counter + 8'h01);
      end
    end
  end


endmodule
```

Этот код соответствует следующему исходному коду на SpinalHDL находящемуся в файле **./hw/spinal/projectname/MyTopLevel.scala**

```
rz@devbox:~/SpinalTemplateSbt $ cat ./hw/spinal/projectname/MyTopLevel.scala
package projectname

import spinal.core._

// Hardware definition
case class MyTopLevel() extends Component {
  val io = new Bundle {
    val cond0 = in  Bool()
    val cond1 = in  Bool()
    val flag  = out Bool()
    val state = out UInt(8 bits)
  }

  val counter = Reg(UInt(8 bits)) init 0

  when(io.cond0) {
    counter := counter + 1
  }

  io.state := counter
  io.flag := (counter === 0) | io.cond1
}
```

```
object MyTopLevelVerilog extends App {
  Config.spinal.generateVerilog(MyTopLevel())
}

object MyTopLevelVhdl extends App {
  Config.spinal.generateVhdl(MyTopLevel())
}
```

Аналогично можно выполнить генерацию кода VHDL:

```
rz@devbox:~/SpinalTemplateSbt $ sbt "runMain projectname.MyTopLevelVhdl"

rz@devbox:~/SpinalTemplateSbt $ ll ./hw/gen/MyTopLevel.v hdl
-rw-r--r--  1 rz  rz  14724 Jan 24 03:51 ./hw/gen/MyTopLevel.vhd
```

Код на VHDL получается очень огромный, так как SpinalHDL включит в него массу инструментальных (вспомогательных) функций. Я подозреваю, что автор языка SpinalHDL изначально ориентировался на генерацию Verilog и его языковые конструкции, поэтому SpinalHDL выдает коротенький Verilog и страшно длинный VHDL.

Далее мы разберем этот шаблонный пример несколько более детально. Я, в след за автором SpinalHDL, тоже буду ориентироваться на Verilog, уж извините.

13.4 Разбор шаблона SpinalTemplateSbt

Структура дерева каталогов этого шаблонного проекта, который по умолчанию называется **projectname**, достаточно сложная, но в нём присутствует всё, для того чтобы начать разрабатывать свою первую цифровую схему, верифицировать, симулировать и синтезировать её.

Начнем с описания некоторых настроечных файлов.

Файл **./build.sbt** — конфигурационный и сборочный файл для SBT, содержит набор переменных, задающих версию языка Scala и версию библиотеки SpinalHDL, которая должна быть использована для компиляции данного проекта. Внутри файла **build.sbt** можно изменить имя проекта с используемого по умолчанию **projectname** на что-то своё (например, на **myproject**), но делать это совершенно не обязательно — это имя не влияет на конечный результат. Если Вы изменили имя проекта, то нужно создать новый рабочий каталог **./hw/spinal/myproject/** и все ваши исходные коды на SpinalHDL складывать в него. Во всех файлах исходных кодов после этого следует указывать **package myproject** в самой первой строке.

Файл **./build.sc** — аналогичный для системы сборки **Mill**. Так как мы будем ориентироваться на SBT, то этот файл нам не потребуется и его можно смело удалить.

Каталог **./hw/spinal/projectname/** — рабочий каталог с исходными кодами проекта. По умолчанию в нём находятся следующие файлы:

```
rz@devbox:~/SpinalTemplateSbt $ ll ./hw/spinal/projectname/
total 16
-rw-r--r--  1 rz  rz  338 Jan 24 03:48 Config.scala
-rw-r--r--  1 rz  rz  580 Jan 24 03:48 MyTopLevel.scala
-rw-r--r--  1 rz  rz  698 Jan 24 03:48 MyTopLevelFormal.scala
-rw-r--r--  1 rz  rz  887 Jan 24 03:48 MyTopLevelSim.scala
```

Файл **./hw/spinal/projectname/Config.scala** — содержит настройки для Scala и должен всегда присутствовать в рабочем каталоге проекта. Если Вы переименовали свой проект в **myproject**, то необходимо скопировать этот файл в новый рабочий каталог и заменить в нем **package projectname** на **package myproject.**

Далее рассмотрим, какие файлы могут присутствовать в рабочем каталоге.

Файл **MyTopLevel.scala** — в нём должен присутствовать код компонента (модуля) самого верхнего уровня для вашего дизайна и ряд входных точек для синтеза: объекты **MyTopLevelVerilog** и **MyTopLevelVhdl**. Имена входных точек передаются в SBT в командной строке, например:

```
rz@devbox:~/SpinalTemplateSbt $ sbt "runMain projectname.MyTopLevelVerilog"
или
rz@devbox:~/SpinalTemplateSbt $ sbt "runMain projectname.MyTopLevelVhdl"
```

Файл **MyTopLevelFormal.scala** — может содержать код для выполнения формальной верификации дизайна и входную точку для неё. Для выполнения верификации необходимо, чтобы был установлен проект **SymbiYosys** и его основная утилита **sby**. Вообще, верификация — это тема отдельного длинного разговора, так что пока не будем её касаться. Запуск верификации осуществляется командой:

```
rz@devbox:~/SpinalTemplateSbt $ sbt "runMain projectname.MyTopLevelFormal"
```

Файл **MyTopLevelSim.scala** — содержит код и точку входа для запуска и выполнения симуляции. Для выполнения симуляции должен быть установлен пакет **Verilator**. Эту интересную тему мы немного коснемся чуть ниже при работе с VexRiscv.

Запуск симуляции осуществляется командой:

```
rz@devbox:~/SpinalTemplateSbt $ sbt "runMain projectname.MyTopLevelSim"
```

Теперь посмотрим на файл с синтезируемым кодом. В заголовке этого файла мы видим стандартные для Scala элементы, описывающие имя пакета программ:

```
package projectname
```

и перечень подключаемых библиотек:

```
import spinal.core._
```

В данном случае подключаются все библиотеки основного ядра SpinalHDL. Далее, при работе с VexRiscv, мы добавим сюда библиотеки софт-ядра, а также некоторые свои компоненты.

Далее идет описание компонента верхнего уровня, который должен быть унаследован от класса **Component** и его интерфейса ввода/вывода:

```
// Hardware definition
case class MyTopLevel() extends Component {
  val io = new Bundle {
    val cond0 = in  Bool()
    val cond1 = in  Bool()
    val flag  = out Bool()
    val state = out UInt(8 bits)
  }
```

В данном случае, для примера, предлагается цифровая схема с двумя входящими однобитными сигналами **cond0** и **cond1**, и двумя выходными сигналами **flag** и **state**. Сигнал **flag** однобитный, а сигнал **state** имеет ширину 8 бит. Этому описанию соответствует следующий код на языке Verilog, из которого мы видим, что SpinalHDL неявно включает в компонент сигнал тактирования **clk** и асинхронного сброса **reset**:

```
module MyTopLevel (
  input  wire          io_cond0,
  input  wire          io_cond1,
  output wire          io_flag,
  output wire [7:0]    io_state,
  input  wire          clk,
  input  wire          reset
);
```

Далее следует описание последовательностной и комбинационной логики:

```
  val counter = Reg(UInt(8 bits)) init 0

  when(io.cond0) {
    counter := counter + 1
  }

  io.state := counter
  io.flag := (counter === 0) | io.cond1
```

В данном случае создается 8-битный регистр-счетчик со сбросом **counter**, значение которого увеличивается на 1 каждый такт при наличии на входе **cond0** лог «1». Выходной сигнал **flag** устанавливается в лог «1», если значение счетчика равно нулю и входной сигнал **cond1** установлен в лог «1».

Этот код транслируется в следующий код на языке Verilog:

```
  reg        [7:0]    counter;

  assign io_state = counter;
  assign io_flag = ((counter == 8'h00) || io_cond1);
  always @(posedge clk or posedge reset) begin
    if(reset) begin
      counter <= 8'h00;
    end else begin
      if(io_cond0) begin
        counter <= (counter + 8'h01);
      end
    end
  end
```

Видно, что SpinalHDL, следуя принятым в Verilog традициям, сначала размещает код для комбинационной логики, а затем добавляет код для последовательностной: запускается процесс обработки по передним фронтам тактового сигнала **clk** и сигнала сброса **reset**, внутри процесса добавляется код для асинхронного сброса. Генерируемый код на языке Verilog получается вполне читаемый и может быть далее использован как модуль в составе другого проекта, либо подан на вход синтезатору Yosys для получения битстрима для ПЛИС.

13.5 Подготовка Makefile-а для синтеза из SpinalHDL в битстрим

Договоримся называть «генерацией» процесс преобразования кода из языка SpinalHDL в код на языке Verilog. Чтобы из кода на языке SpinalHDL получить битстрим для ПЛИС на плате «Карно», необходимо последовательно выполнить следующие действия:

1. Выполнить генерацию текстов Verilog из SpinalHDL с помощью утилиты **sbt**. На выходе получим один или несколько файлов Verilog (файлы с расширением **.v)**.
2. Выполнить синтез нетлиста из текстов Verilog с помощью утилиты **yosys**. На выходе получим нетлист в формате JSON (файл с расширением **.json**).
3. Выполнить оптимизацию, размещение и трассировку нетлиста утилитой **nextpnr-ecp5**. На выходе получим текстовый файл размещения в формате Trellis (файл с расширением **.config**).
4. Выполнить генерацию битстрима из файла размещения. На выходе получим двоичный файл (расширение **.bit**), готовый к загрузке в ПЛИС.

Воспользуемся наработками из главы «*8. Типовой Makefile для синтеза с помощью Yosys*», слегка упростим его для понимания, и получим следующий **Makefile**:

```
rz@devbox:~/SpinalTemplateSbt$ cat Makefile

NAME = projectname
SPINALHDL = ./hw/spinal/projectname/MyTopLevel.scala
VERILOG = ./hw/gen/MyTopLevel.v
LPF = karnix_cabga256.lpf
DEVICE = 25k
PACKAGE = CABGA256
FTDI_CHANNEL = 0 ### FT2232 has two channels, select 0 for channel A or 1 for channel B
#
FLASH_METHOD := $(shell cat flash_method 2> /dev/null)
UPLOAD_METHOD := $(shell cat upload_method 2> /dev/null)

all: $(NAME).bit

$(VERILOG): $(SPINALHDL)
        sbt "runMain projectname.MyTopLevelVerilog"

$(NAME).bin: $(LPF) $(VERILOG)
        yosys -v2 -p "synth_ecp5 -abc2 -top MyTopLevel -json $(NAME).json" $(VERILOG)
        nextpnr-ecp5 --package $(PACKAGE) --$(DEVICE) --json $(NAME).json --textcfg $
(NAME)_out.config --lpf $(LPF) --lpf-allow-unconstrained
        ecppack --compress --freq 38.8 --input $(NAME)_out.config --bit $(NAME).bit


upload_openloader:
ifeq ("$(FLASH_METHOD)", "flash")
        openFPGALoader -v --ftdi-channel $(FTDI_CHANNEL) -f --reset $(NAME).bit
else
        openFPGALoader -v --ftdi-channel $(FTDI_CHANNEL) $(NAME).bit
endif

clean:
        @rm $(NAME).json $(NAME)_out.config $(NAME).bit $(VERILOG)
```

По мимо этого, для успешной генерации битстрима, нам необходимо создать .LPF файл с описанием внешних выводов микросхемы ПЛИС, распаянной на плате «Карно» и передать его имя в утилиту **nextpnr-ecp5**. Создадим файл **karnix_cabga256.lpf** исходя из следующих данных:

1. На плате «Карно» тактовый генератор частотой **25.0** МГц подключен к выводу **B9** микросхемы ПЛИС, обозначим этот вывод для входного тактового сигнала **clk**.
2. Задействуем вывод **E13**, который на плате «Карно» подведен к кнопке **KEY3** и подтянут к лог «0», как входной сигнал асинхронного сброса **reset**.
3. Задействуем выводы **B13** и **C13**, к которым подключены кнопки **KEY0** и **KEY1**, как входные сигналы **io_cond0** и **io_cond1** соответственно.

4. Задействуем вывод **A13**, к нему подключен светодиод **LED0**, как выходной сигнал **io_flag**.
5. Задействуем выводы **L1**, **L2**, **M1**, **M2**, **K4**, **K5**, **L4** и **L4** как биты выходного сигнала **io_state[7:0]**. Эти выводы распаяны на плате «Карно» на линии **GPIO_00** — **GPIO_07**.
6. Все выводы имеют уровни напряжений +3,3В, т. е. соответствуют типу уровней **LVCMOS33**.

Тогда наш .LPF файл примет следующий вид:

```
rz@devbox:~/SpinalTemplateSbtForKarnix$ cat karnix_cabga256.lpf

FREQUENCY PORT "clk" 25.0 MHz;
LOCATE COMP "clk" SITE "B9";
IOBUF PORT "clk" IO_TYPE=LVCMOS33;

LOCATE COMP "reset" SITE "E13";
IOBUF PORT "reset" IO_TYPE=LVCMOS33;

LOCATE COMP "io_cond0" SITE "B13";             # KEY0
IOBUF PORT "io_cond0" IO_TYPE=LVCMOS33;

LOCATE COMP "io_cond1" SITE "C13";             # KEY1
IOBUF PORT "io_cond1" IO_TYPE=LVCMOS33;

LOCATE COMP "io_flag" SITE "A13";             # LED0
IOBUF PORT "io_flag" IO_TYPE=LVCMOS33;

LOCATE COMP "io_state[0]" SITE "L1";              # GPIO_00
IOBUF PORT "io_state[0]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_state[0]" PULLMODE=NONE DRIVE=16;
LOCATE COMP "io_state[1]" SITE "L2";              # GPIO_01
IOBUF PORT "io_state[1]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_state[1]" PULLMODE=NONE DRIVE=16;
LOCATE COMP "io_state[2]" SITE "M1";              # GPIO_02
IOBUF PORT "io_state[2]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_state[2]" PULLMODE=NONE DRIVE=16;
LOCATE COMP "io_state[3]" SITE "M2";              # GPIO_03
IOBUF PORT "io_state[3]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_state[3]" PULLMODE=NONE DRIVE=16;
LOCATE COMP "io_state[4]" SITE "K4";              # GPIO_04
IOBUF PORT "io_state[4]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_state[4]" PULLMODE=NONE DRIVE=16;
LOCATE COMP "io_state[5]" SITE "K5";              # GPIO_05
IOBUF PORT "io_state[5]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_state[5]" PULLMODE=NONE DRIVE=16;
LOCATE COMP "io_state[6]" SITE "L4";              # GPIO_06
IOBUF PORT "io_state[6]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_state[6]" PULLMODE=NONE DRIVE=16;
LOCATE COMP "io_state[7]" SITE "L5";              # GPIO_07
IOBUF PORT "io_state[7]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_state[7]" PULLMODE=NONE DRIVE=16;
```

Теперь можно собирать проект командой **make** и ожидать окончания сборки, которое сигнализируется следующим сообщением:

```
Info: Program finished normally.
ecppack --compress --freq 38.8 --input projectname_out.config --bit projectname.bit
```

Если все прошло удачно, то можно подключить плату «Карно», загрузить битстрим в микросхему ПЛИС командой **make upload** и протестировать нашу цифровую схему нажимая кнопки KEY0 и KEY1 — мы должны наблюдать за реакцией на светодиоде LED0, а на выходах GPIO получать число срабатываний в двоичной форме. Результирующий битстрим будет находиться в файле **projectname.bit** .

Небольшое замечание по поводу сигналов сброса. На плате «Карно» уже имеется кнопка «**RESET**». Её нажатие приводит к системном сбросу микросхемы ПЛИС и инициации процесса считывания конфигурации из NOR flash памяти, что равносильно сбросу питания. Сигнал «**reset**», который мы определили выше и подвели на копку KEY3 — это сброс схемы пользователя внутри ПЛИС, фактически её нажатие будет приводить к обнулению только счетчика синтезированной цифровой схемы.

В следующей главе мы немного поэкспериментируем с кодом на SpinalHDL и проанализируем информацию об ошибках, выдаваемых компилятором Scala и генератором кода SpinalHDL. А пока что приведу ссылку на репозиторий модифицированного шаблонного проекта с Makefile-ом и .LPF для платы «Карно»:

https://github.com/Fabmicro-LLC/SpinalTemplateSbtForKarnix.git

13.6 Анализируем вывод сообщений SpinalHDL

Запустив процесс сборки, первым делом мы увидим вывод от компилятора Scala, который будет собирать наш проект вместе с библиотеками классов в составе SpinalHDL:

```
rz@devbox:~/SpinalTemplateSbt$ make

sbt "runMain projectname.MyTopLevelVerilog"
[info] welcome to sbt 1.6.0 (Ubuntu Java 11.0.9.1)
[info] loading settings for project spinaltemplatesbt-build from plugins.sbt ...
[info] loading project definition from /home/rz/SpinalTemplateSbt/project
[info] loading settings for project projectname from build.sbt ...
[info] set current project to projectname (in build file:/home/rz/SpinalTemplateSbt/)
[info] running (fork) projectname.MyTopLevelVerilog
```

Если компиляция прошла успешно, т. е. синтаксических ошибок с точки зрения Scala не выявлено, то SBT запустит на исполнение собранный байткод с помощью JVM и мы начнем получать сообщения от SpinalHDL о ходе процесса генерации:

```
[info] [Runtime] SpinalHDL v1.10.0    git head : 270018552577f3bb8e5339ee2583c9c22d324215
[info] [Runtime] JVM max memory : 8294.0MiB
[info] [Runtime] Current date : 2024.01.26 21:22:22
[info] [Progress] at 0.000 : Elaborate components
[info] [Progress] at 0.725 : Checks and transforms
[info] [Progress] at 0.958 : Generate Verilog
[info] [Done] at 1.140
[success] Total time: 5 s, completed Jan 26, 2024, 9:22:23 PM
```

Давайте попробуем внести в код приведенного выше примера какую-нибудь ошибку, скажем, попробуем присвоить переменной (сигналу) типа **UInt** значение типа **Bool**. Для этого исправим код следующим образом:

```
val counter = Reg(UInt(8 bits)) init 0

io.state := True

when(io.cond0) {
  counter := counter + 1
}
```

После запуска процесса сборки мы сразу получим сообщение от компилятора Scala следующего вида:

```
rz@devbox:~/SpinalTemplateSbt$ make

sbt "runMain projectname.MyTopLevelVerilog"
[info] welcome to sbt 1.6.0 (Ubuntu Java 11.0.9.1)
[info] loading settings for project spinaltemplatesbt-build from plugins.sbt ...
[info] loading project definition from /home/rz/SpinalTemplateSbt/project
```

```
[info] loading settings for project projectname from build.sbt ...
[info] set current project to projectname (in build file:/home/rz/SpinalTemplateSbtForKarnix/)
[info] compiling 1 Scala source to /home/rz/SpinalTemplateSbtForKarnix/target/scala-2.12/classes ...
[error] /home/rz/SpinalTemplateSbtForKarnix/hw/spinal/projectname/MyTopLevel.scala:16:12: overloaded
method value := with alternatives:
[error]   (value: String)Unit <and>
[error]   (rangesValue: (Any, Any),_rangesValues: (Any, Any)*)Unit <and>
[error]   (that: spinal.core.UInt)(implicit loc: spinal.idslplugin.Location)Unit
[error]  cannot be applied to (spinal.core.Bool)
[error]   io.state := True
[error]            ^
[error] one error found
[error] (Compile / compileIncremental) Compilation failed
[error] Total time: 7 s, completed Jan 26, 2024, 10:50:44 PM
Makefile:15: recipe for target 'hw/gen/MyTopLevel.v' failed
make: *** [hw/gen/MyTopLevel.v] Error 1
```

Здесь Scala выдает четкое сообщение о том, где и когда произошла ошибка. Не забудем вернуть код в исходное состояние и продолжим эксперименты.

В процессе генерации SpinalHDL осуществляет ряд проверок нашего дизайна на логические проблемы. В частности, он может выявлять «защелки» и «петли» — это когда выход логической цепи завернут на её вход. Такие конструкции не должны присутствовать в нормальных цифровых схемах и создают ряд проблем (за исключением очень редких случаев, когда разработчик делает это целенаправленно). Сложность дизайна делает такие ошибки неочевидными и малозаметными, и они могут легко просочиться в готовое изделие, здесь SpinalHDL стоит на страже наших интересов.

Попробуем умышленно добавить «петлю» в код и посмотреть, как выглядит реакция SpinalHDL. Для этого исправим файл **hw/spinal/projectname/MyTopLevel.scala** так, чтобы тело компонента содержало следующий код:

```
val counter = Reg(UInt(8 bits)) init 0
val latch = Bool()

latch := ~io.flag

when(io.cond0) {
  counter := counter + 1
}

io.state := counter
//io.flag := (counter === 0) | io.cond1
io.flag := (counter === 0) | io.cond1 | latch
```

После запуска команды **make** мы увидим длинную портянку с различными сообщениями, среди которых будут сообщения об ошибке следующего вида:

```
[info]
**********************************************************************************************
[info] [Warning] Elaboration failed (2 errors).
[info]           Spinal will restart with scala trace to help you to find the problem.
[info]
**********************************************************************************************
[info] [Progress] at 0.902 : Elaborate components
[info] [Progress] at 0.918 : Checks and transforms
[error] Exception in thread "main" spinal.core.SpinalExit:
[error]  Error detected in phase PhaseCheckCombinationalLoops
[error] ********************************************************************************
[error] ********************************************************************************
[error] COMBINATORIAL LOOP :
[error]   Partial chain :
[error]     >>> (toplevel/io_flag : out Bool) at projectname.MyTopLevel$
$anon$1.<init>(MyTopLevel.scala:10) >>>
[error]     >>> (toplevel/latch :  Bool) at projectname.MyTopLevel.<init>(MyTopLevel.scala:15) >>>
[error]     >>> (toplevel/io_flag : out Bool) at projectname.MyTopLevel$
$anon$1.<init>(MyTopLevel.scala:10) >>>
```

```
[error]   Full chain :
[error]     (toplevel/io_flag : out Bool)
[error]     ! Bool
[error]     (toplevel/latch :  Bool)
[error]     Bool || Bool
[error]     (toplevel/io_flag : out Bool)
[error] ********************************************************************************
```

Здесь SpinalHDL подсказывает нам номера строк кода (**MyTopLevel.scala:10** и **MyTopLevel.scala:15**) в которых определены переменные, участвующие в создании «петли».

К слову сказать, утилита синтеза **yosys** тоже способна распознавать такие «петли», о чем выдает предупреждающее сообщения. Поэтому, давайте попробуем умышлено допустить ошибку, которую **yosys** не заметит, но SpinalHDL точно не пропустит. Для этого нам достаточно неполно определить состояние выходного сигнала **io_flag**, например вот так:

```
val counter = Reg(UInt(8 bits)) init 0

when(io.cond0) {
  counter := counter + 1
}

io.state := counter
//io.flag := (counter === 0) | io.cond1
when(counter === 0 | io.cond1) {
  io.flag := True
}
```

Жирным шрифтом выделен участок проблемного кода. После запуска сборки мы получим от SpinalHDL следующее сообщение об ошибке:

```
[info]
********************************************************************************************
[info] [Warning] Elaboration failed (2 errors).
[info]           Spinal will restart with scala trace to help you to find the problem.
[info]
********************************************************************************************
[info] [Progress] at 0.948 : Elaborate components
[info] [Progress] at 0.968 : Checks and transforms
[error] Exception in thread "main" spinal.core.SpinalExit:
[error]  Error detected in phase PhaseCheck_noLatchNoOverride
[error] ********************************************************************************
[error] ********************************************************************************
[error] LATCH DETECTED from the combinatorial signal (toplevel/io_flag : out Bool), defined at
[error]     projectname.MyTopLevel$$anon$1.<init>(MyTopLevel.scala:10)
[error]     projectname.MyTopLevel.<init>(MyTopLevel.scala:7)
[error]     projectname.MyTopLevelVerilog$.$anonfun$new$3(MyTopLevel.scala:28)
[error]     spinal.sim.JvmThread.run(SimManager.scala:51)
[error] ********************************************************************************
```

Как и в предыдущий раз, SpinalHDL сообщает нам сначала номер строки, в которой определена переменная, создающая проблему, а далее номера строк блоков кода, где эта переменная используется. К сожалению, в данном случае SpinalHDL не даст точного указания на номер ошибочной строки кода, так как такой строки не существует — весь код корректный, некорректным является логика дизайнера.

Давайте исправим ошибку добавив обработку недостающего состояния следующим глупым, но наглядным способом:

```
io.state := counter
//io.flag := (counter === 0) | io.cond1
when(counter === 0 | io.cond1) {
  io.flag := True
} otherwise {
  io.flag := False
}
```

Здесь жирным текстом выделен добавленный код, исправляющий логическую ошибку. Запустим сборку и убедимся, что компиляция и генерация проходит успешно:

```
rz@devbox:~/SpinalTemplateSbt$ make

sbt "runMain projectname.MyTopLevelVerilog"
[info] welcome to sbt 1.6.0 (Ubuntu Java 11.0.9.1)
[info] compiling 1 Scala source to /home/rz/SpinalTemplateSbt/target/scala-2.12/classes ...
[info] running (fork) projectname.MyTopLevelVerilog
[info] [Runtime] SpinalHDL v1.10.0    git head : 270018552577f3bb8e5339ee2583c9c22d324215
[info] [Runtime] JVM max memory : 8294.0MiB
[info] [Runtime] Current date : 2024.01.26 22:40:43
[info] [Progress] at 0.000 : Elaborate components
[info] [Progress] at 0.692 : Checks and transforms
[info] [Progress] at 0.900 : Generate Verilog
[info] [Done] at 1.036
[success] Total time: 13 s, completed Jan 26, 2024, 10:40:44 PM
```

SpinalHDL может выдавать еще ряд ошибок, но приведенные выше две являются самыми распространенными, и с ними сталкивается любой новичок буквально при первой же попытке собрать свой проект. Надеюсь, мой беглый разбор поможет читателю избежать таких ситуаций, и он не остановится в своём желании пробовать SpinalHDL.

13.7 Симуляция и верификация в SpinalHDL

Несколько слов следует сказать о том, как реализована симуляция и верификация в SpinalHDL. Так как SpinalHDL на выходе генерирует Verilog код, то к его симуляции и верификации могут быть применены все имеющиеся традиционные средства. Однако, с некоторых пор в SpinalHDL был добавлен API, позволяющий создавать испытательные стенды (testbenches) прямо внутри дизайна, используя всю мощь языка Scala. Данный API позволяет читать/изменять сигналы внутри «подопытного» компонента, распараллеливать и, наоборот, объединять процессы симуляции, ожидать наступление определенного состояния, распечатывать информацию о процессе симуляции, состоянии регистров и сигналов испытуемого компонента и т. д.

Для выполнения симуляции SpinalHDL использует внешние средства, такие как Verilator и Icarus Verilog, установку этих тулов мы обсуждали ранее. Ниже я приведу пример тривиального тестбенча из документации на SpinalHDL.

В данном примере в качестве «подопытного кролика» предлагается простой компонент **Indentity** с параметром определяющим размерность своих сигналов в битах**.** Этот компонент имеет один входной сигнал **a** и один выходной сигнал **z**, который всегда повторяет входной сигнал. Код такого компонента выглядит следующим образом:

```
import spinal.core._

// Identity takes n bits in a and gives them back in z
class Identity(n: Int) extends Component {
  val io = new Bundle {
    val a = in Bits(n bits)
    val z = out Bits(n bits)
  }

  io.z := io.a
}
```

Предположим, что нам требуется испытать (верифицировать) работу компонента **Identity** для случая, когда размерность сигналов составляет три бита. Тогда испытательный стенд для него может выглядеть так:

```
import spinal.core.sim._

object TestIdentity extends App {
  // Use the component with n = 3 bits as "dut" (device under test)
  SimConfig.withWave.compile(new Identity(3)).doSim{ dut =>
    // For each number from 3'b000 to 3'b111 included
    for (a <- 0 to 7) {
      // Apply input
      dut.io.a #= a
      // Wait for a simulation time unit
      sleep(1)
      // Read output
      val z = dut.io.z.toInt
      // Check result
      assert(z == a, s"FAIL: Got $z, expected $a")
    }
    println(s"SUCCESS!")
  }
}
```

Данный тестбенч определяет испытуемый компонент **dut** типа **Identity(3)** и верифицирует его путем последовательного перебора всех допустимых входных значений **dut.a** и проверкой выходных значений **dut.z**, которые обязаны совпадать. Если будет выявлено несовпадение, то в лог будет выдано сообщение об ошибке (assert) и верификация будет остановлена. Если же перебор всех вариантов завершится успешно, то мы увидим в логе сообщение «SUCCESS!».

Добавим этот код в отдельный файл **hw/spinal/projectname/TestIdentity.scala** и запустим верификацию следующей командой:

```
rz@devbox:~/SpinalTemplateSbt $ sbt "runMain projectname.TestIdentity"

[info] welcome to sbt 1.6.0 (OpenJDK BSD Porting Team Java 1.8.0_392)
[info] loading settings for project spinaltemplatesbt-build from plugins.sbt ...
[info] loading project definition from /usr/home/rz/SpinalTemplateSbt/project
[info] loading settings for project projectname from build.sbt ...
[info] set current project to projectname (in build file:/usr/home/rz/SpinalTemplateSbt/)
[info] running (fork) projectname.TestIdentity
[info] [Runtime] SpinalHDL v1.10.1    git head : 2527c7c6b0fb0f95e5e1a5722a0be732b364ce43
[info] [Runtime] JVM max memory : 4512.0MiB
[info] [Runtime] Current date : 2024.03.15 03:33:52
[info] [Progress] at 0.000 : Elaborate components
[info] [Progress] at 0.176 : Checks and transforms
[info] [Progress] at 0.259 : Generate Verilog
[info] [Done] at 0.304
[info] [Progress] Simulation workspace in
/usr/home/rz/SpinalTemplateSbt/./simWorkspace/Identity
[info] [Progress] Verilator compilation started
[info] [info] Found cached verilator binaries
[info] [Progress] Verilator compilation done in 311.979 ms
[info] [Progress] Start Identity test simulation with seed 1948229477
[info] SUCCESS!
[info] [Done] Simulation done in 7.414 ms
[success] Total time: 3 s, completed Mar 15, 2024 3:33:53 AM
```

Здесь мы наблюдаем как SBT сгенерировал нам код для Verilator-а (обычно это код на языке Си, он располагается в подкаталоге **./simWorkspace/**), запустил компиляцию этого кода и поставил его на исполнение. Успешный результат симуляции мы видим в виде текстового сообщения:

```
[info] SUCCESS!
```

Но это еще не всё. В подкаталоге **./simWorkspace/Identity/test/** мы получим файл с временной диаграммой всех перебираемых в процессе верификации вариантов сигналов или состояний:

```
rz@devbox:~/SpinalTemplateSbtForKarnix $ ll simWorkspace/Identity/test/
total 4
```

```
-rw-r--r--  1 rz  rz  548 Mar 15 03:33 wave.vcd
```

Визуализировать (просмотреть в графическом виде) этот файл можно утилитой **GTKWave**:

```
rz@devbox:~/SpinalTemplateSbt $ gtkwave simWorkspace/Identity/test/wave.vcd
```

Так как испытуемый компонент **Identity** содержит исключительно комбинационную схему, то результат визуализации не очень интересен — не имеет разворота во времени. Далее, при работе с вычислительным ядром VexRiscv, я продемонстрирую как провести потактовую симуляцию ядра при исполнении программы, и мы посмотрим на некоторые временные диаграммы.

14. Синтез вычислительного ядра VexRiscv

14.1 Знакомство с репозиторием VexRiscv

Теперь, когда мы немного освоились с синтаксисом языка SpinalHDL, подготовили и проверили среду, научились находить и исправлять типовые ошибки, можно приступить к изучению синтезируемой ЭВМ на базе ядра VexRiscv.

Репозиторий с кодом VexRiscv находится по адресу: https://github.com/SpinalHDL/VexRiscv

Клонируем его и посмотрим, что там внутри:

```
rz@devbox:~$ git clone https://github.com/SpinalHDL/VexRiscv
Cloning into 'VexRiscv'...
...
Resolving deltas: 100% (9730/9730), done.

rz@devbox:~$ cd  VexRiscv

rz@devbox:~/VexRiscv$ tree -d -L 5
.
├── assets
├── doc
│   ├── gcdPeripheral
│   │   ├── img
│   │   └── src
│   │       └── main
│   │           ├── c
│   │           └── scala
│   ├── nativeJtag
│   ├── smp
│   └── vjtag
├── project
├── scripts
│   ├── Murax
│   │   ├── arty_a7
│   │   ├── iCE40-hx8k_breakout_board
│   │   │   └── img
│   │   ├── iCE40-hx8k_breakout_board_xip
│   │   │   └── img
│   │   └── iCE40HX8K-EVB
│   └── regression
└── src
    ├── main
    │   ├── c
    │   │   ├── common
    │   │   ├── emulator
    │   │   │   ├── build
    │   │   │   └── src
    │   │   └── murax
    │   │       ├── hello_world
    │   │       └── xipBootloader
    │   ├── ressource
```

```
│   │   └── hex
│   └── scala
│       ├── spinal
│       │   └── lib
│       └── vexriscv
│           ├── demo
│           ├── ip
│           ├── plugin
│           └── test
└── test
```

Из этого разветвленного дерева подкаталогов нас на первых порах будут интересовать только подкаталоги ./**scripts/Murax**, **./src/main/scala** и **./src/main/c**.

Подкаталог ./**scripts/Murax** содержит ряд скриптов для сборки нескольких примеров с использованием СнК Murax для нескольких плат с микросхемами ПЛИС.

Подкаталог **./src/main/scala** содержит код на языке SpinalHDL, в том числе реализацию ядра VexRiscv, нескольких вариантов СнК на его базе и код периферийных устройств задействованных в этих СнК.

Подкаталог **./src/main/c** содержит исходные коды на языке Си нескольких примеров программ, которые можно исполнить на данном синтезируемом микропроцессоре.

Перед тем, как мы приступим к сборке демонстрационного проекта, необходим предпринять некоторые подготовительные действия.

14.2 Установка компилятора для архитектуры RISC-V

Сборка проекта на базе VexRiscv усложнена тем фактом, что помимо генерации и синтезирования аппаратуры нам еще потребуется компилировать программы на языке Си, получать исполняемый машинный код и размещать его в памяти синтезированного микропроцессора. Для того, чтобы компилировать Си программы для архитектуры RISC-V нам потребуется установить один из доступных опенсорсных компиляторов, а именно «GNU C Toolchain for RISC-V». Есть несколько способов установить RISC-V GCC toolchain.

Первый способ — это клонировать репозиторий https://github.com/riscv/riscv-gnu-toolchain и выполнить сборку самостоятельно. В этом случае придется установить ряд дополнительных библиотек и разрешить ряд проблем с зависимостями. Для пользователей ОС Linux и *BSD это не должно являться проблемой. При самостоятельной сборке желательно указать путь для установки компилятора **`--prefix=/opt/`**. Рецепт сборки выглядит так:

```
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain riscv-gnu-toolchain
cd riscv-gnu-toolchain

ARCH=rv32im
rmdir -rf $ARCH
mkdir $ARCH; cd $ARCH
../configure  --prefix=/opt/$ARCH --with-arch=$ARCH --with-abi=ilp32
sudo make -j100500
```

И ждать два дня.

Второй способ — это скачать готовый (pre-built) вариант компилятора с сайта SiFive, но для этого потребуется пройти регистрацию на сайте компании и указать свой адрес электронной почты. Для скачивания нужно пройти по ссылке:

```
https://www.sifive.com/software/  => SiFive Freedom SDK for Metal => Download
```

Скачанный пакет желательно распаковать в каталог **/opt/**, либо добавить символическую ссылку.

Ну и третий, самый быстрый, способ — установить компилятор GCC из репозитория пакетов дистрибутива вашей ОС, так как с некоторых пор он присутствует почти во всех дистрибутивах ОС Linux и *BSD. Чтобы установить из репозитория:

Для ОС FreeBSD:

```
$ sudo pkg install riscv64-none-elf-gcc
```

Для ОС Linux:

```
$ sudo apt-get install gcc-riscv64-linux-gnu
```

Для того, чтобы проверить работоспособность компилятора, выполним сборку приложения «**hello_world**» из репозитория VexRiscv:

```
rz@devbox:~$ cd VexRiscv/src/main/c/murax/hello_world

rz@devbox:~/VexRiscv/src/main/c/murax/hello_world$ make
mkdir -p build/src/
/opt/riscv//bin/riscv64-unknown-elf-gcc -c -march=rv32i  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing   -o build/src/main.o src/main.c
/opt/riscv//bin/riscv64-unknown-elf-gcc -S -march=rv32i  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing   -o build/src/main.o.disasm src/main.c
mkdir -p build/src/
/opt/riscv//bin/riscv64-unknown-elf-gcc -c -march=rv32i  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing -o build/src/crt.o src/crt.S -
D__ASSEMBLY__=1
/opt/riscv//bin/riscv64-unknown-elf-gcc -march=rv32i  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing -o build/hello_world.elf build/src/main.o
build/src/crt.o -march=rv32i  -mabi=ilp32 -nostdlib -lgcc -mcmodel=medany -nostartfiles -
ffreestanding -Wl,-Bstatic,-T,./src/linker.ld,-Map,build/hello_world.map,--print-memory-
usage -Lbuild
Memory region         Used Size  Region Size  %age Used
             RAM:         896 B         2 KB     43.75%
/opt/riscv//bin/riscv64-unknown-elf-objcopy -O ihex build/hello_world.elf
build/hello_world.hex
/opt/riscv//bin/riscv64-unknown-elf-objdump -S -d build/hello_world.elf >
build/hello_world.asm
/opt/riscv//bin/riscv64-unknown-elf-objcopy -O verilog build/hello_world.elf
build/hello_world.v
```

Если сборка не выполняется, то необходимо выяснить путь к компилятору в вашей системе командой **which**, например:

```
rz@butterfly:~/VexRiscv/src/main/c/murax/hello_world % which riscv64-none-elf-gcc
/usr/local/bin/riscv64-none-elf-gcc
```

и отредактировать ./**makefile**, заменив значения в переменных **RISCV_NAME** и **RISCV_PATH** соответственно показаниям утилиты **which**:

```
RISCV_NAME ?= riscv64-none-elf
RISCV_PATH ?= /usr/local/
```

Замечание для пользователей FreeBSD:

В *BSD системах сборку нужно осуществлять командой **gmake** (GNU make), а не **make** (BSD make), так как это две разные несовместимые утилиты. При работе со SpinalHDL я настоятельно рекомендую сделать локальный симлинк для **gmake** в пользовательский **~/bin/make** и установить его первым в поиске, как-то так:

```
rz@butterfly:~ % ln -s /usr/local/bin/gmake ~/bin/make

rz@butterfly:~ % setenv PATH ~/bin:$PATH

rz@butterfly:~ % echo $PATH
/home/rz/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin:/home/rz/bin
```

Такое решение радикально упростит жизнь, так как SpinalHDL постоянно использует **GNUтый make**. После работы достаточно закрыть сеанс (терминал), чтобы вернуть переменную PATH в исходное состояние и старый **BSD make** вернется.

14.3 Подготавливаем Makefile, LPF и toplevel.v

Читатель, наверное, уже обратил внимание на то, что в дереве каталогов присутствуют примеры только для отладочных плат на базе ПЛИС Xilinx Artix-7 и Lattice iCE40, нас же интересует сборка для Lattice ECP5 в составе платы «Карно» (ну или любой другой). Поэтому нам необходимо выполнить еще ряд подготовительных действий: создать рабочий подкаталог **Karnix** для платы «Карно», в котором разместить **Makefile** для процедуры сборки (компиляция, генерация, синтез, формирование битстрима), **LPF** файл с описанием сигналов ввода/вывода для нашей платы в соответствии с дизайном который мы собираемся синтезировать и файл с модулем-оберткой toplevel.v для связи внешних сигналов софт-ядра VexRiscv с сигналами на нашей платы.

1. Создадим рабочий каталог:

```
rz@devbox:~/VexRiscv$ mkdir scripts/Murax/Karnix
rz@devbox:~/VexRiscv$ cd scripts/Murax/Karnix

и перейдем в него:

rz@devbox:~/VexRiscv/scripts/Murax/Karnix$
```

2. Позаимствуем **Makefile** из репозитория **SpinalTemplateSbtForKarnix** и добавим в него вызов сборки Си программы «hello_world». Готовый Makefile будет выглядеть следующим образом:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ cat Makefile

NAME = murax_hello_world
VEXRISCV = ../../../src/main/scala/vexriscv/demo/Murax.scala
VERILOG = ../../../Murax.v
BIN = ../../../Murax.v*.bin
HEX = ../../../src/main/c/murax/hello_world/build/hello_world.hex
CSRC = ../../../src/main/c/murax/hello_world/src/*.[ch]
LPF = karnix_cabga256.lpf
DEVICE = 25k
PACKAGE = CABGA256
FTDI_CHANNEL = 0 ### FT2232 has two channels, select 0 for channel A or 1 for channel B
#
FLASH_METHOD := $(shell cat flash_method 2> /dev/null)
UPLOAD_METHOD := $(shell cat upload_method 2> /dev/null)

all: $(NAME).bit

compile: $(HEX)

$(HEX): $(CSRC)
```

```
        (cd ../../../src/main/c/murax/hello_world/; make)

generate: $(VERILOG) $(BIN)

$(VERILOG) $(BIN): $(HEX) $(VEXRISCV)
        (cd ../../..; sbt "runMain vexriscv.demo.Murax_karnix")

$(NAME).bit: $(LPF) $(VERILOG) toplevel.v
        yosys -v2 -p "synth_ecp5 -abc2 -top toplevel -json $(NAME).json" $(VERILOG)
toplevel.v
        nextpnr-ecp5 --package $(PACKAGE) --$(DEVICE) --json $(NAME).json --textcfg $
(NAME)_out.config --lpf $(LPF) --lpf-allow-unconstrained
        ecppack --compress --freq 38.8 --input $(NAME)_out.config --bit $(NAME).bit


upload:
ifeq ("$(FLASH_METHOD)", "flash")
        openFPGALoader -v --ftdi-channel $(FTDI_CHANNEL) -f --reset $(NAME).bit
else
        openFPGALoader -v --ftdi-channel $(FTDI_CHANNEL) $(NAME).bit
endif

clean:
        -rm *.json *.config *.bit $(VERILOG) $(BIN) $(HEX)
```

3. Аналогичным образом позаимствуем из репозитория **SpinalTemplateSbtForKarnix** файл с описанием внешних сигналов **karnix_cabga256.lpf** и разместим его в этом рабочем каталоге. Здесь нам потребуется добавить сигнальные линии для отладочного UART, для кнопок KEY и для светодиодов LED. Результирующий LPF файл будет выглядеть следующим образом:

```
FREQUENCY PORT "io_clk25" 25.0 MHz;
LOCATE COMP "io_clk25" SITE "B9";
IOBUF PORT "io_clk25" IO_TYPE=LVCMOS33;

LOCATE COMP "io_key[0]" SITE "B13";
IOBUF PORT "io_key[0]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_key[1]" SITE "C13";
IOBUF PORT "io_key[1]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_key[2]" SITE "D13";
IOBUF PORT "io_key[2]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_key[3]" SITE "E13";
IOBUF PORT "io_key[3]" IO_TYPE=LVCMOS33;

LOCATE COMP "io_led[0]" SITE "A13";              # LED0
IOBUF PORT "io_led[0]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_led[1]" SITE "A14";              # LED1 - WORK
IOBUF PORT "io_led[1]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_led[2]" SITE "A15";              # LED2 - TEST
IOBUF PORT "io_led[2]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_led[3]" SITE "B14";              # LED3
IOBUF PORT "io_led[3]" IO_TYPE=LVCMOS33;

LOCATE COMP "io_uart_debug_txd" SITE "E12";              # UART_DEBUG_TXD
IOBUF PORT "io_uart_debug_txd" IO_TYPE=LVCMOS33;
LOCATE COMP "io_uart_debug_rxd" SITE "D12";              # UART_DEBUG_RXD
IOBUF PORT "io_uart_debug_rxd" IO_TYPE=LVCMOS33;
LOCATE COMP "io_rst_n" SITE "R8";                # FPGA_RESET
IOBUF PORT "io_rst_n" IO_TYPE=LVCMOS33 DRIVE=4;
```

14.4 Добавляем точку входа для генерации СнК Murax для платы «Карно»

Отредактируем файл **./src/main/scala/vexriscv/demo/Murax.scala,** описывающий синтезируемую систему-на-кристалле Murax, и добавим в него точку входа для генерации кода под плату «Карно». Для этого добавим в самый конец файла следующие строки кода на языке SpinalHDL:

```
object Murax_karnix{
  def main(args: Array[String]) {
    val hex = "src/main/c/murax/hello_world/build/hello_world.hex"
    SpinalVerilog(Murax(MuraxConfig.default(false).copy(coreFrequency = 25 MHz,
onChipRamSize = 96 kB, onChipRamHexFile = hex)))
  }
}
```

Здесь мы описываем «оберточный» объект **Murax_karnix** из метода **main()** которого будет вызываться генерация кода Verilog. В качестве параметров в конструктор объекта **Murax()** можно передать структуру с настройками, среди которых требуется указать частоту тактового генератора на плате (25 МГц), указать объем ОЗУ который потребуется синтезировать (в данном случае **96 КБ**) и путь к HEX файлу, содержимое которого будет размещено в это ОЗУ при процедуре конфигурировании микросхемы ПЛИС.

Микросхему Lattice ECP5 25K позволяет синтезировать ОЗУ объемом до 112 КБ из распределенной памяти BRAM, но часть из этой памяти будет задействована под регистры микропроцессора и прочие регистры цифровой аппаратуры, поэтому указываем требуемый размер немного меньше максимального возможного.

14.5 Модифицируем код программы «hello_world»

Код демонстрационной программы «hello_world» располагается в подкаталоге **./src/main/c/murax/hello_world/src/**, зайдем в него и осмотримся.

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ cd ../../../src/main/c/murax/hello_world/src
rz@devbox:~/VexRiscv/src/main/c/murax/hello_world/src$ ls -l

-rw-rw-r-- 1 rz rz 1642 Feb  3 19:46 crt.S
-rw-rw-r-- 1 rz rz  178 Feb  3 19:46 gpio.h
-rw-rw-r-- 1 rz rz  303 Feb  3 19:46 interrupt.h
-rw-rw-r-- 1 rz rz 2163 Feb  3 19:49 linker.ld
-rw-rw-r-- 1 rz rz 1182 Feb  3 21:55 main.c
-rw-rw-r-- 1 rz rz  483 Feb  3 21:38 murax.h
-rw-rw-r-- 1 rz rz  217 Feb  3 19:46 prescaler.h
-rw-rw-r-- 1 rz rz  296 Feb  3 19:46 timer.h
-rw-rw-r-- 1 rz rz 1263 Feb  3 21:40 uart.h
```

Как видно, программа состоит из ряда заголовочных файлов (**.h**), главного файла **main.c** с кодом программы, файла **crt.S** с кодом инициализации «C run-time», написанного на ассемблере и файла конфигурации редактора объектных связей (линковщика) — **linker.ld**.

Для начала, нам нужно отредактировать файл конфигурации линковщика и указать каким объемом ОЗУ мы располагаем и какого размера стек мы хотим использовать. Откроем файл **linker.ld**, найдем следующие строки и внесем изменения, выделенные жирным шрифтом:

```
MEMORY {
  RAM      (rwx): ORIGIN = 0x80000000, LENGTH = 96k
}

_stack_size = DEFINED(_stack_size) ? _stack_size : 2048;
_heap_size = DEFINED(_heap_size) ? _heap_size : 0;
```

Далее нам нужно добавить код инициализации UART в текст программы. По какой-то причине автор программы «hello_world» этого не сделал, видимо понадеявшись на значения по умолчанию для платы Artix, но эти значения не подходят для нашей платы Karnix из-за того, что на нашей плате использован другой тактовый генератор. Откроем файл **main.c** и отредактируем тело функции **main()** чтобы оно приняло следующий вид:

```
void main() {
```

```
        Uart_Config uart_config;

        uart_config.dataLength = UART_DATA_8;
        uart_config.parity = UART_PARITY_NONE;
        uart_config.stop = UART_STOP_ONE;

        uint32_t rxSamplePerBit = UART_PRE_SAMPLING_SIZE + UART_SAMPLING_SIZE +
UART_POST_SAMPLING_SIZE;

        uart_config.clockDivider = SYSTEM_CLOCK_HZ / UART_BAUD_RATE / rxSamplePerBit - 1;
        uart_applyConfig(UART, &uart_config);


        GPIO_A->OUTPUT_ENABLE = 0x0000000F;
        GPIO_A->OUTPUT = 0x00000001;
        const int nleds = 4;
        const int nloops = 1000000;
        while(1){
                println("Hello world, this is VexRiscv!");
                for(unsigned int i=0;i<nleds-1;i++){
                        GPIO_A->OUTPUT = 1<<i;
                        delay(nloops);
                }
                for(unsigned int i=0;i<nleds-1;i++){
                        GPIO_A->OUTPUT = (1<<(nleds-1))>>i;
                        delay(nloops);
                }
        }
}
```

Из приведенного выше текста на языке Си видно, что программа «hello_world» циклически зажигает и гасит четыре светодиода и выдает в порт UART сообщение «Hello world, this is VexRiscv» — всё согласно сложившимся традициям.

Добавим недостающие константы в файл **uart.h**:

```
rz@devbox:~/VexRiscv/src/main/c/murax/hello_world/src$ head -20 uart.h
#ifndef UART_H_
#define UART_H_

#define UART_PRE_SAMPLING_SIZE  1
#define UART_SAMPLING_SIZE      3
#define UART_POST_SAMPLING_SIZE 1
#define UART_BAUD_RATE       115200

#define UART_PARITY_NONE        0
#define UART_PARITY_EVEN        1
#define UART_PARITY_ODD         2

#define UART_STOP_ONE           0
#define UART_STOP_TWO           1

#define UART_DATA_5             4
#define UART_DATA_6             5
#define UART_DATA_7             6
#define UART_DATA_8             7
#define UART_DATA_9             8
```

И в файл **murax.h**:

```
rz@devbox:~/VexRiscv/src/main/c/murax/hello_world/src$ head -10 murax.h
#ifndef __MURAX_H__
#define __MURAX_H__

#include "timer.h"
#include "prescaler.h"
#include "interrupt.h"
#include "gpio.h"
#include "uart.h"

#define SYSTEM_CLOCK_HZ 25000000 // system clock on Karnix board in Hz
```

Проверим собираемость программы «hello_world»:

```
rz@devbox:~/VexRiscv/src/main/c/murax/hello_world/src$ cd ..

rz@devbox:~/VexRiscv/src/main/c/murax/hello_world$ make clean
rm -rf build/src
rm -f build/hello_world.elf
rm -f build/hello_world.hex
rm -f build/hello_world.map
rm -f build/hello_world.v
rm -f build/hello_world.asm
find build -type f -name '*.o' -print0 | xargs -0 -r rm
find build -type f -name '*.d' -print0 | xargs -0 -r rm

rz@devbox:~/VexRiscv/src/main/c/murax/hello_world$ make
mkdir -p build/src/
/opt/riscv//bin/riscv64-unknown-elf-gcc -c -march=rv32i  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing   -o build/src/main.o src/main.c
/opt/riscv//bin/riscv64-unknown-elf-gcc -S -march=rv32i  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing   -o build/src/main.o.disasm src/main.c
mkdir -p build/src/
/opt/riscv//bin/riscv64-unknown-elf-gcc -c -march=rv32i  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing -o build/src/crt.o src/crt.S -
D__ASSEMBLY__=1
/opt/riscv//bin/riscv64-unknown-elf-gcc -march=rv32i  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing -o build/hello_world.elf build/src/main.o
build/src/crt.o -march=rv32i  -mabi=ilp32 -nostdlib -lgcc -mcmodel=medany -nostartfiles -
ffreestanding -Wl,-Bstatic,-T,./src/linker.ld,-Map,build/hello_world.map,--print-memory-
usage -Lbuild
Memory region         Used Size  Region Size  %age Used
             RAM:        2756 B        96 KB      2.80%
/opt/riscv//bin/riscv64-unknown-elf-objcopy -O ihex build/hello_world.elf
build/hello_world.hex
/opt/riscv//bin/riscv64-unknown-elf-objdump -S -d build/hello_world.elf >
build/hello_world.asm
/opt/riscv//bin/riscv64-unknown-elf-objcopy -O verilog build/hello_world.elf
build/hello_world.v
```

Здесь нас интересует, чтобы компилятор успешно собрал файл **build/hello_world.hex** и размер области памяти не превышал **96КБ** синтезируемых в СнК.

Для тех, кто не хочет тратить время на прохождение всего вышеописанного приключения, приведу ссылку на клон репозитория VexRiscv содержащего ветку с кодом для поддержки платы ПИР СЦХ-254 «Карно»:

https://github.com/Fabmicro-LLC/VexRiscvWithKarnix/tree/karnix

14.6 Сборка СнК Murax и ядра VexRiscv

Ну что же, наконец у нас всё готово и можно попытаться запустить сборку СнК на базе ядра VerRiscv. Перейдем в рабочий каталог для нашей платы и запустим **make**:

```
rz@devbox:~/VexRiscv/src/main/c/murax/hello_world$ cd ../../../../../scripts/Murax/Karnix

rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ make
```

Первым делом мы будем наблюдать как SBT запустит компиляцию кода на языке Scala и SpinalHDL:

```
(cd ../../..; sbt "runMain vexriscv.demo.Murax_karnix")
[info] welcome to sbt 1.6.0 (Ubuntu Java 11.0.9.1)
[info] loading settings for project vexriscv-build from plugins.sbt ...
[info] loading project definition from /home/rz/VexRiscv/project
[info] loading settings for project root from build.sbt ...
[info] set current project to VexRiscv (in build file:/home/rz/VexRiscv/)
```

```
[info] running (fork) vexriscv.demo.Murax_karnix
```

Компилятор Scala отработал без ошибок и SBT поставил полученный байт-код на исполнение на Java машине:

```
[info] [Runtime] SpinalHDL v1.10.1    git head : 2527c7c6b0fb0f95e5e1a5722a0be732b364ce43
[info] [Runtime] JVM max memory : 8294.0MiB
[info] [Runtime] Current date : 2024.02.04 00:37:10
[info] [Progress] at 0.000 : Elaborate components
...
[info] [Progress] at 2.469 : Checks and transforms
[info] [Progress] at 3.398 : Generate Verilog
[info] [Warning] 157 signals were pruned. You can call printPruned on the backend report to
get more informations.
[info] [Done] at 4.832
[success] Total time: 11 s, completed Feb 4, 2024, 12:37:15 AM
```

В результате исполнения байт-кода мы получаем набор Verilog файлов. Давайте остановим процесс и посмотрим, что это за файлы:

```
^C

rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ ls -l ../../../
total 1316
-rw-rw-r-- 1 rz rz   1079 Feb  3 19:46 LICENSE
-rw-rw-r-- 1 rz rz 351198 Feb  4 11:50 Murax.v
-rw-rw-r-- 1 rz rz 221184 Feb  4 11:50 Murax.v_toplevel_system_ram_ram_symbol0.bin
-rw-rw-r-- 1 rz rz 221184 Feb  4 11:50 Murax.v_toplevel_system_ram_ram_symbol1.bin
-rw-rw-r-- 1 rz rz 221184 Feb  4 11:50 Murax.v_toplevel_system_ram_ram_symbol2.bin
-rw-rw-r-- 1 rz rz 221184 Feb  4 11:50 Murax.v_toplevel_system_ram_ram_symbol3.bin
-rw-rw-r-- 1 rz rz  63826 Feb  3 22:13 README.md
drwxrwxr-x 2 rz rz   4096 Feb  3 19:46 assets
-rw-rw-r-- 1 rz rz    621 Feb  3 19:46 build.sbt
-rw-rw-r-- 1 rz rz    998 Feb  3 19:46 build.sc
…
```

Файл **Murax.v** содержит код нашего СнК и всех его составных частей, в том числе ядро VexRiscv, преобразованное в текст на языке Verilog. Если заглянуть во внутрь этого файла, то мы увидим главный модуль **Murax** с описанием его внешних сигналов:

```
module Murax (
  input  wire          io_asyncReset,
  input  wire          io_mainClk,
  input  wire          io_jtag_tms,
  input  wire          io_jtag_tdi,
  output wire          io_jtag_tdo,
  input  wire          io_jtag_tck,
  input  wire [31:0]   io_gpioA_read,
  output wire [31:0]   io_gpioA_write,
  output wire [31:0]   io_gpioA_writeEnable,
  output wire          io_uart_txd,
  input  wire          io_uart_rxd
);
```

Файлы с расширением **.bin** содержат данные для инициализации синтезируемого ОЗУ (RAM) и, по сути, это наша программа «hello_world» в виде последовательности двоичных 8-ми битных слов в текстовом виде. Файлы **.bin** подключаются внутри файла **Murex.v** следующими строками кода:

```
  initial begin
    $readmemb("Murax.v_toplevel_system_ram_ram_symbol0.bin",ram_symbol0);
    $readmemb("Murax.v_toplevel_system_ram_ram_symbol1.bin",ram_symbol1);
    $readmemb("Murax.v_toplevel_system_ram_ram_symbol2.bin",ram_symbol2);
    $readmemb("Murax.v_toplevel_system_ram_ram_symbol3.bin",ram_symbol3);
  end
```

Данный комплект файлов, вместе с оберточным файлом **toplevel.v**, передается в утилиту **yosys** для выполнения синтеза. Еще раз введем команду **make** и продолжим сборку:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ make

yosys -v2 -p "synth_ecp5 -abc2 -top toplevel -json murax_hello_world.json" ../../../Murax.v
toplevel.v
1. Executing Verilog-2005 frontend: ../../../Murax.v
2. Executing Verilog-2005 frontend: toplevel.v
…
```

Здесь мы видим, как на вход утилиты **yosys** передаются два файла: **../../../Murax.v** и **toplevel.v**, и она начинает их обрабатывать, используя свой Verilog-2005 фронтэнд.

В процессе синтеза, утилита **yosys** выдаст нам ряд предупреждений о том, что не все биты комплексного сигнала **gpio_read** привязаны к источникам (драйверам):

```
3.10. Executing OPT_CLEAN pass (remove unused cells and wires).
3.11. Executing CHECK pass (checking for obvious problems).
Warning: Wire toplevel.\murax.system_gpioACtrl.io_gpio_read_buffercc.io_dataIn [31] is used
but has no driver.
Warning: Wire toplevel.\murax.system_gpioACtrl.io_gpio_read_buffercc.io_dataIn [30] is used
but has no driver.
Warning: Wire toplevel.\murax.system_gpioACtrl.io_gpio_read_buffercc.io_dataIn [29] is used
but has no driver.
...
Warning: Wire toplevel.\murax.system_gpioACtrl.io_gpio_read_buffercc.io_dataIn [5] is used
but has no driver.
Warning: Wire toplevel.\murax.system_gpioACtrl.io_gpio_read_buffercc.io_dataIn [4] is used
but has no driver.
```

Это объясняется тем, что мы задействовали всего первых четыре бита регистра **gpioA** из 32-х доступных (см файл toplevel.v), которые мы подвязали к линиям кнопок **io_key[0]-io_key[3]**. Остальные не подвязанные биты будут автоматически подтянуты к логическому «0».

Утилита **yosys** завершается без ошибок и выдает нам кое-какую статистику о своей деятельности:

```
3.50. Executing CHECK pass (checking for obvious problems).
3.51. Executing JSON backend.
Warnings: 28 unique messages, 28 total
End of script. Logfile hash: 697ada3929, CPU: user 29.56s system 0.80s, MEM: 151.63 MB peak
Yosys 0.36+58 (git sha1 ea7818d31, gcc 7.5.0-3ubuntu1~18.04 -fPIC -Os)
Time spent: 21% 32x opt_clean (7 sec), 21% 11x techmap (6 sec), ...
```

Вслед за **yosys** в конвейере сборки вызывается утилита **nextpnr-ecp5**. Напомню, что она получает на вход файл с описанием нетлиста в формате JSON, синтезированного утилитой **yosys** и занимается оптимизацией, размещением и трассировкой синтезированной схемы на базовых элементах целевой микросхемы ПЛИС.

```
nextpnr-ecp5 --package CABGA256 --25k --json murax_hello_world.json --textcfg
murax_hello_world_out.config --lpf karnix_cabga256.lpf --lpf-allow-unconstrained
Info: constraining clock net 'io_clk25' to 25.00 MHz
```

Здесь утилита **nextpnr-ecp5** сообщает нам, что она обнаружила сигнал **io_clk25**, для которого присутствует ограничение по частоте не ниже **25.00 МГц**. Это значение далее будет использоваться в процессе STA как целевое.

Далее утилита **nextpnr-ecp5** выводит нам статистику о задействованных схемой ресурсах целевой микросхемы ПЛИС до выполнения оптимизации и размещения, т. е. в том виде как это поступило от утилиты **yosys**:

```
Info: Logic utilisation before packing:
Info:     Total LUT4s:      2272/24288     9%
Info:         logic LUTs:   1818/24288     7%
Info:         carry LUTs:    238/24288     0%
```

```
Info:           RAM LUTs:    144/ 3036     4%
Info:          RAMW LUTs:     72/ 6072     1%

Info:      Total DFFs:      1379/24288     5%
```

Обратите внимание, что весь СнК вместе с вычислительным ядром занимает всего **1379** D-триггеров и **2272** ячеек LUT-4.

Далее утилита **nextpnr-ecp5** сообщает, как произошло размещение внешних сигналов, которые жестко определены в LPF файле.

```
Info: Packing IOs..
Info: pin 'io_uart_debug_txd$tr_io' constrained to Bel 'X58/Y0/PIOB'.
Info: pin 'io_uart_debug_rxd$tr_io' constrained to Bel 'X58/Y0/PIOA'.
Info: pin 'io_led[3]$tr_io' constrained to Bel 'X67/Y0/PIOA'.
Info: pin 'io_led[2]$tr_io' constrained to Bel 'X67/Y0/PIOB'.
Info: pin 'io_led[1]$tr_io' constrained to Bel 'X65/Y0/PIOB'.
Info: pin 'io_led[0]$tr_io' constrained to Bel 'X65/Y0/PIOA'.
Info: pin 'io_key[3]$tr_io' constrained to Bel 'X62/Y0/PIOB'.
Info: pin 'io_key[2]$tr_io' constrained to Bel 'X62/Y0/PIOA'.
Info: pin 'io_key[1]$tr_io' constrained to Bel 'X60/Y0/PIOB'.
Info: pin 'io_key[0]$tr_io' constrained to Bel 'X60/Y0/PIOA'.
Info: pin 'io_core_jtag_tms$tr_io' constrained to Bel 'X0/Y23/PIOB'.
Info: pin 'io_core_jtag_tdo$tr_io' constrained to Bel 'X0/Y23/PIOD'.
Info: pin 'io_core_jtag_tdi$tr_io' constrained to Bel 'X0/Y23/PIOC'.
Info: pin 'io_core_jtag_tck$tr_io' constrained to Bel 'X0/Y23/PIOA'.
Info: pin 'io_clk25$tr_io' constrained to Bel 'X38/Y0/PIOA'.
```

После чего утилита выполняет упаковку остальных сигналов схемы:

```
Info: Packing constants..
Info: Packing carries...
Info: Packing LUTs...
Info: Packing LUT5-7s...
Info: Packing FFs...
Info:     542 FFs paired with LUTs.
Info: Generating derived timing constraints...
```

В процессе упаковки обнаруживаются сигналы тактирования, которые утилита **nextpnr-ecp5** предлагает разместить в глобальных линиях:

```
Info: Promoting globals...
Info:     promoting clock net io_clk25$TRELLIS_IO_IN to global network
Info:     promoting clock net io_core_jtag_tck$TRELLIS_IO_IN to global network
```

После размещения утилита **nextpnr-ecp5** еще раз выдает статистику задействованных ресурсов микросхемы ПЛИС, но уже в более детальном виде:

```
Info: Device utilisation:
Info:             TRELLIS_IO:    15/  197     7%
Info:                   DCCA:     2/   56     3%
Info:                 DP16KD:    48/   56    85%
Info:             MULT18X18D:     0/   28     0%
Info:                 ALU54B:     0/   14     0%
Info:                EHXPLLL:     0/    2     0%
Info:                EXTREFB:     0/    1     0%
Info:                   DCUA:     0/    1     0%
Info:              PCSCLKDIV:     0/    2     0%
Info:                IOLOGIC:     0/  128     0%
Info:               SIOLOGIC:     0/   69     0%
Info:                    GSR:     0/    1     0%
Info:                  JTAGG:     0/    1     0%
Info:                   OSCG:     0/    1     0%
Info:                  SEDGA:     0/    1     0%
Info:                    DTR:     0/    1     0%
Info:                USRMCLK:     0/    1     0%
Info:                CLKDIVF:     0/    4     0%
Info:              ECLKSYNCB:     0/   10     0%
Info:                DLLDELD:     0/    8     0%
Info:                 DDRDLL:     0/    4     0%
Info:                DQSBUFM:     0/    8     0%
Info:        TRELLIS_ECLKBUF:     0/    8     0%
```

```
Info:           ECLKBRIDGECS:     0/    2     0%
Info:                   DCSC:     0/    2     0%
Info:             TRELLIS_FF:  1379/24288     5%
Info:           TRELLIS_COMB:  2346/24288     9%
Info:           TRELLIS_RAMW:    36/ 3036     1%
```

Здесь **DCCA** — это число глобальных линий для тактовых сигналов, **DP16KD** — число блоков распределенной памяти (каждый по 16Кбит), **TRELLIS_FF** — число D-триггеров, а **TRELLIS_COMB** — число логических (комбинационных) элементов LUT-4. Важно понимать, что данные аппаратные блоки специфичны для микросхемы ПЛИС Lattice ECP5, для других микросхем названия блоков и их количество будет иным.

Утилита **nextpnr-ecp5** производит расчет максимальных задержек для тактовых сигналов и выдает нам следующую информацию:

```
Info: Max frequency for clock         '$glbnet$io_clk25$TRELLIS_IO_IN': 49.99 MHz (PASS at 25.00
MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 142.92 MHz (PASS at 12.00
MHz)
```

Далее запускается процесс оптимизации и трассировки, который занимает достаточно большое время и требователен к вычислительным ресурса системы, на которой происходит сборка.

```
Info: Routing globals...
...
Info: Critical path report for clock '$glbnet$io_clk25$TRELLIS_IO_IN' (posedge -> posedge):
```

По окончанию процесса трассировки утилита nextpnr-ecp5 еще раз вычисляет задержки и выдает нам полученную информацию:

```
Info: Max frequency for clock         '$glbnet$io_clk25$TRELLIS_IO_IN': 69.80 MHz (PASS at 25.00
MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 149.30 MHz (PASS at 12.00
MHz)
```

Работа утилиты **nextpnr-ecp5** завершается сообщением вида:

```
2 warnings, 0 errors

Info: Program finished normally.
```

После плэйсера **nextpnr-ecp5** по цепочке запускается утилита **ecppack**, которая быстро и немногословно формирует битстрим для микросхемы ПЛИС Lattice ECP5:

```
ecppack --compress --freq 38.8 --input murax_hello_world_out.config --bit murax_hello_world.bit
```

Процесс сборки окончен, результирующий битстрим находится в файле **murax_hello_world.bit** в текущем рабочем каталоге:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ ls -l
total 8120
-rw-rw-r-- 1 rz rz    1345 Feb  3 22:08 Makefile
-rw-rw-r-- 1 rz rz   11486 Feb  3 19:47 karnix_cabga256.lpf
-rw-rw-r-- 1 rz rz  257823 Feb  4 13:05 murax_hello_world.bit
-rw-rw-r-- 1 rz rz 6246585 Feb  4 13:05 murax_hello_world.json
-rw-rw-r-- 1 rz rz 1785563 Feb  4 13:05 murax_hello_world_out.config
-rw-rw-r-- 1 rz rz    1158 Feb  3 21:51 toplevel.v
```

14.7 Запускаем СнК Murax на ПЛИС и тестируем «hello_world»

Перед тем как мы подключим плату «Карно» и запустим прошивку, нам необходимо установить еще одну утилиту — эмулятор терминала, чтобы мы могли наблюдать за тем, какие данные передаются в отладочный порт UART. Напомню, что программа «hello_world» передает в этот порт строку символов приветствия.

В качестве эмулятора терминала предлагаю использовать утилиту **minicom**. Она присутствует во всех известных дистрибутивах ОС Linux и FreeBSD, поэтому установить её совсем не сложно:

Для FreeBSD:

```
rz@butterfly:~ % sudo pkg install minicom
```

Для ОС Linux:

```
rz@devbox:~$ sudo apt-get install minicom
```

Напомню, что на плате «Карно» присутствуют два последовательных порта доступных через USB. Порт 0 работает в режиме JTAG и используется для прошивки NOR flash и микросхемы ПЛИС. Порт 1 является отладочным и может быть использован пользователем по своему усмотрению. Программа «hello_world» будет выдавать строку приветствия именно в этот порт со скоростью **115200** бод без аппаратного контроля (RTS/CTS Flow Control OFF) и стандартного формат фрейма **8N1**.

Подключим плату «Карно» к рабочему компьютеру, откроем на нем еще одно окно терминала и запустим в нем утилиту **minicom**:

Для FreeBSD:

```
rz@butterfly:~ % sudo minicom -b 115200 -D /dev/ttyU1
```

Для ОС Linux:

```
rz@devbox:~$ sudo minicom -b 115200 -D /dev/ttyUSB1
```

Зайдем в настройки нажав Ctrl-A+O, зайдем в меню «Serial port setup» и отключим RTS/CTS нажатием клавиши «F»:

F - Hardware Flow Control : No

Через меню «**Save setup as dfl**» сохраним текущие настройки как настройки по умолчанию.

Теперь можно запустить загрузку битстрима в ПЛИС:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ make upload
```

В процессе загрузки битстрима утилита **openFPGALoader** будет сообщать о ходе выполнения:

```
Jtag frequency : requested 6.00MHz   -> real 6.00MHz
Open file DONE
Parse file DONE
Enable configuration: DONE
SRAM erase: DONE
Detail:
Jedec ID          : ef
```

```
memory type       : 70
memory capacity   : 18
flash chip unknown: use basic protection detection
Erasing: [==================================================] 100.00%
Done
Writing: [==================================================] 100.00%
Done
Refresh: DONE
```

Сообщение DONE означает успешное завершение. Если вместо этого наблюдается сообщение типа:

```
write to flash
No cable or board specified: using direct ft2232 interface
unable to open ftdi device: -3 (device not found)
JTAG init failed with: unable to open ftdi device
```

это означает, что утилита **openFPGALoader** не смогла выполнить подключение по USB к программатору. Наиболее вероятная причина такого исхода — отсутствие прав доступа к устройству. Проверьте подключение кабеля и попробуйте запустить процесс загрузки от пользователя **root**.

Если процесс завершился успешно, то на плате «Карно» мы сразу будем наблюдать, как один за другим по цепочке зажигаются и гаснут светодиоды, а в окне терминала **minicom** будем наблюдать сообщения приветствия:

```
Welcome to minicom 2.8

OPTIONS: I18n
Compiled on Jan 29 2024, 16:10:11.
Port /dev/ttyU1, 03:09:59

Press CTRL-A Z for help on special keys

Hello world, this is VexRiscv!
Hello world, this is VexRiscv!
Hello world, this is VexRiscv!
Hello world, this is VexRiscv!
```

Для завершение сессии в эмуляторе терминала **minicom** следует нажать последовательность: **Ctrl-A, X, Enter**.

15. Устройство синтезируемого СнК Murax и ядра VexRiscv

Теперь, когда у нас есть возможность модифицировать, собирать и пересобирать СнК Murax и Сишный код программы для него, настало время погрузиться в детали и посмотреть, как код СнК выглядит на языке SpinalHDL, как задается конфигурация ядра VexRiscv и как к нему подключается различная периферия.

15.1 Структура СнК Murax

Напомню, что код СнК Murax находится в файле **./src/main/scala/vexriscv/demo/Murax.scala**, выше мы уже добавляли свой код в этот файл — дополнительную точку входа для сборки. Сейчас посмотрим немного глубже.

СнК Murax по структуре очень похож на СнК Briey, структура которого изображена рис. 20, но вместо **AxiCrossbar** в СнК Murax используется **PipelinedMemoryBus**.

Весь код СнК Murax описывается одним классом **Murax**, который является наследником класса **Component**, т. е. Murax является аппаратным компонентом. Конструктор класса Murax имеет параметр **config**, который является структурой типа **MuraxConfig**, содержащей дефолтные параметры СнК (тактовую частоту, размер ОЗУ и т.д.)

```
case class Murax(config : MuraxConfig) extends Component{
  import config._
```

Как и у всякого компонента, у Murax есть внешние сигналы ввода/вывода, сигналы сброса и тактирования:

```
  val io = new Bundle {
    //Clocks / reset
    val asyncReset = in Bool()
    val mainClk = in Bool()

    //Main components IO
    val jtag = slave(Jtag())

    //Peripherals IO
    val gpioA = master(TriStateArray(gpioWidth bits))
    val uart = master(Uart())

    val xip = ifGen(genXip)(master(SpiXdrMaster(xipConfig.ctrl.spi)))
  }
```

Конструкция **ifGen(genXip)** добавит генерацию следующего за ней кода, если **genXip** будет установлен в значение **True**. XiP это флэш память с SPI интерфейсом и функцией «eXecution In Place» — возможностью исполнения кода прямо из флэш памяти. В данном случае она не задействована.

Murax содержит два домена тактирования:

```
  val systemClockDomain = ClockDomain(
    clock = io.mainClk,
    reset = resetCtrl.systemReset,
    frequency = FixedFrequency(coreFrequency)
  )

  val debugClockDomain = ClockDomain(
    clock = io.mainClk,
    reset = resetCtrl.mainClkReset,
    frequency = FixedFrequency(coreFrequency)
  )
```

Домен тактирования **debugClockDomain** содержит JTAG интерфейс и все что с ним связано. Вся остальная периферия, а также ядро VexRiscv находятся в домене **systemClockDomain**. Далее будем рассматривать только домен **systemClockDomain**.

Центральным элементом в системном домене является шина, которая называется **pipelinedMemoryBus**, управляет которой шинный арбитр **MuraxMasterArbiter**:

```
val mainBusArbiter = new MuraxMasterArbiter(pipelinedMemoryBusConfig, bigEndianDBus)
```

Арбитр имеет три порта: два ведомых (slave) порта для подключения к ядру VexRiscv (порт инструкций **iBus** и порт данных **dBus**) и один порт мастер **MainBus**.

```
val mainBusMapping = ArrayBuffer[(PipelinedMemoryBus,SizeMapping)]()
```

Подключение ядра VexRiscv к арбитру осуществляется следующим затейливым образом. Сначала инстанциируется класс **VexRiscv** с передачей ему большой структуры с настройками **VexRiscvConfig** — она содержит описание того, какие плагины и функции должны быть включены/выключены:

```
//Instanciate the CPU
val cpu = new VexRiscv(
  config = VexRiscvConfig(
    plugins = cpuPlugins += new DebugPlugin(debugClockDomain, hardwareBreakpointCount)
  )
)
```

После этого, в цикле перебираются все подключенные в данный момент плагины из которых состоит ядро VexRiscv, и каждый плагин линкуется с шинами **iBus** и **dBus** - они являются объектами типа **Stream**. Напомню, что **Stream** состоит из комплексного сигнала **cmd** для передачи запроса и комплексного сигнала **rsp** для ответа. Линковка происходит операторами <>, << или >>, вот так:

```
//Checkout plugins used to instanciate the CPU to connect them to the SoC
val timerInterrupt = False
val externalInterrupt = False
for(plugin <- cpu.plugins) plugin match{
  case plugin : IBusSimplePlugin =>
    mainBusArbiter.io.iBus.cmd <> plugin.iBus.cmd
    mainBusArbiter.io.iBus.rsp <> plugin.iBus.rsp
  case plugin : DBusSimplePlugin => {
    if(!pipelineDBus)
      mainBusArbiter.io.dBus <> plugin.dBus
    else {
      mainBusArbiter.io.dBus.cmd << plugin.dBus.cmd.halfPipe()
      mainBusArbiter.io.dBus.rsp <> plugin.dBus.rsp
    }
  }
  case plugin : CsrPlugin        => {
    plugin.externalInterrupt := externalInterrupt
    plugin.timerInterrupt := timerInterrupt
  }
  case plugin : DebugPlugin         => plugin.debugClockDomain{
    resetCtrl.systemReset setWhen(RegNext(plugin.io.resetOut))
    io.jtag <> plugin.io.bus.fromJtag()
  }
  case _ =>
}
```

Для специфических плагинов, таких как **CsrPlugin** и **DebugPlugin** производится линковка дополнительных сигналов, в данном случае это сигналы **externalInterrupt, timerInterrupt и io.jtag**.

С другой стороны арбитра к шине **MainBus** подключается шина набортного ОЗУ **ram** типа **MuraxPipelinedMemoryBusRam** и мост **apbBridge** типа **PipelinedMemoryBusToApbBridge**:

```
//****** MainBus slaves ********

val mainBusMapping = ArrayBuffer[(PipelinedMemoryBus,SizeMapping)]()
val ram = new MuraxPipelinedMemoryBusRam(
  onChipRamSize = onChipRamSize,
  onChipRamHexFile = onChipRamHexFile,
  pipelinedMemoryBusConfig = pipelinedMemoryBusConfig,
  bigEndian = bigEndianDBus
)
mainBusMapping += ram.io.bus -> (0x80000000l, onChipRamSize)

val apbBridge = new PipelinedMemoryBusToApbBridge(
  apb3Config = Apb3Config(
    addressWidth = 20,
    dataWidth = 32
  ),
  pipelineBridge = pipelineApbBridge,
  pipelinedMemoryBusConfig = pipelinedMemoryBusConfig
)
mainBusMapping += apbBridge.io.pipelinedMemoryBus -> (0xF0000000l, 1 MB)
```

Шине ОЗУ выделяется область адресного пространства, начиная с **0x80000000** и длиной **onChipRamSize**. Переменная **onChipRamSize** — это член класса, которая задается при инстанциировании класса **Murax** в точке входа. В нашем случае, для платы «Карно», она

установлена в 96 КБ (см. главу «*14.4 Добавляем точку входа для генерации СнК Murax для платы «Карно*»).

Для моста **apbBridge** резервируется адресное пространство, начиная с **0xF0000000** и размером **1 МБ**. Через мост **apbBridge** подключается вся стандартная периферия:

```
//******** APB peripherals *********
val apbMapping = ArrayBuffer[(Apb3, SizeMapping)]()

val gpioACtrl = Apb3Gpio(gpioWidth = gpioWidth, withReadSync = true)
io.gpioA <> gpioACtrl.io.gpio
apbMapping += gpioACtrl.io.apb -> (0x00000, 4 kB)

val uartCtrl = Apb3UartCtrl(uartCtrlConfig)
uartCtrl.io.uart <> io.uart
externalInterrupt setWhen(uartCtrl.io.interrupt)
apbMapping += uartCtrl.io.apb  -> (0x10000, 4 kB)

val timer = new MuraxApb3Timer()
timerInterrupt setWhen(timer.io.interrupt)
apbMapping += timer.io.apb     -> (0x20000, 4 kB)

...
```

Каждому компоненту на шине **apbBrigde** резервируется своё адресное пространство для регистров управления. Из кода выше видно, что регистры периферийного порта **gpioA** будут расположены начиная с адреса **0xF0000000 + 0x00000**, а регистры UART будут находиться с адреса **0xF0000000 + 0x10000**, и т. д.

На этом код СнК Murax заканчивается.

Если мы заходим создать свой периферийный компонент, то нам потребуется сделать его наследником класса **Component**, содержащего интерфейс определяемый классом **Apb3**, после чего подключить его к шине **apbBrigde** в этом же месте и аналогичным образом. Далее я покажу, как это сделать на нескольких примерах.

15.2 Структура вычислительного ядра VexRiscv

Как уже было отмечено выше, вычислительное ядро VexRiscv — это обычный компонент, т. е. является наследником класса **Component**, описан в файле **./src/main/scala/vexriscv/VexRiscv.scala** и инстанциируется строкой кода вида:

```
val cpu = new VexRiscv( config = VexRiscvConfig( ...) )
```

где параметр **config** задает настройки вычислительно ядра, описываемые классом **VexRiscvConfig**. Посмотрим на структуру этого конфигурационного класса, чтобы понять, какие «цапы» мы можем крутить и в какой степени мы можем изменять структуру ядра VecRiscv:

```
case class VexRiscvConfig(){
  var withMemoryStage = true
  var withWriteBackStage = true
  val plugins = ArrayBuffer[Plugin[VexRiscv]]()

...
```

На первый взгляд, настроек немного: две булевых переменных **withMemoryStage** и **withWriteBackStage**, и переменная **plugins** описывающая массив абстрактных интерфейсных объектов типа **Plugin[VexRiscv]**. Булевы переменные задают возможность подключения двух ступеней конвейера: Memory и WriteBack. Напомню, что ядро VexRiscv является конвейерной реализацией архитектуры RISC-V, в которой конвейер может содержать от двух до пяти

ступеней: [Fetch], Decode, Execute, [Memory], [WriteBack]. Таким образом, ступени Memory и WriteBack можно включать/выключать, изменяя переменные **withMemoryStage** и **withWriteBackStage**. Что касается ступени Fetch, то она подключается при необходимости как плагин добавлением в массив **plugins**. Плагины — это то, что делает VexRiscv расширяемым, интересным и уникальным. На рис. 27 приведено обобщенное изображение структуры конвейера VexRiscv и того, как некоторые плагины распределены между его ступенями.

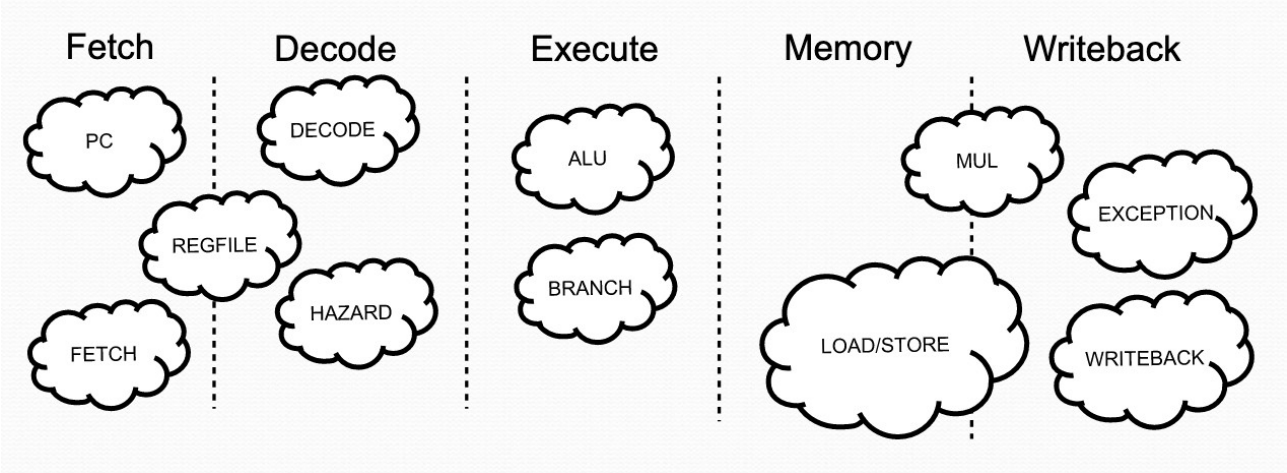


Рис. 27. Структура конвейера VexRiscv в общих чертах.

15.3 Плагины вычислительного ядра VexRiscv

Посмотрим, какими плагинами на данный момент мы располагаем и какими индивидуальными настройками они обладают. Весь код плагинов находится в подкаталоге **./src/main/scala/vexriscv/plugin/**, по одному файлу на плагин:

```
rz@devbox:~/VexRiscv$ ls -w 120 ./src/main/scala/vexriscv/plugin/

AesPlugin.scala                     FpuPlugin.scala               NoPipeliningPlugin.scala
BranchPlugin.scala                  HaltOnExceptionPlugin.scala   PcManagerSimplePlugin.scala
CfuPlugin.scala                     HazardPessimisticPlugin.scala Plugin.scala
CsrPlugin.scala                     HazardSimplePlugin.scala      PmpPlugin.scala
DBusCachedPlugin.scala              IBusCachedPlugin.scala        PmpPluginNapot.scala
DBusSimplePlugin.scala              IBusSimplePlugin.scala        RegFilePlugin.scala
DebugPlugin.scala                   IntAluPlugin.scala            ShiftPlugins.scala
DecoderSimplePlugin.scala           MemoryTranslatorPlugin.scala
SingleInstructionLimiterPlugin.scala
DivPlugin.scala                     Misc.scala                    SrcPlugin.scala
DummyFencePlugin.scala              MmuPlugin.scala
StaticMemoryTranslatorPlugin.scala
EmbeddedRiscvJtag.scala             Mul16Plugin.scala             VfuPlugin.scala
ExternalInterruptArrayPlugin.scala  MulDivIterativePlugin.scala   YamlPlugin.scala
Fetcher.scala                       MulPlugin.scala
FormalPlugin.scala                  MulSimplePlugin.scala
```

Детальное описание большинства плагинов можно найти в README файле в репозитории VexRiscv по ссылке: https://github.com/SpinalHDL/VexRiscv/blob/master/README.md#plugins

Ниже я приведу описания только части наиболее важных и часто используемых плагинов с некоторыми поясняющими комментариями. С этими плагинами мы далее проведем ряд экспериментов.

DBusSimplePlugin — создает шину данных для вычислительного ядра, без кеширования данных. Позволяет подключать ядро к шинам вида Axi4Shared, Avalon, Withbone, AhbLite3, Bmb, PipelinedMemoryBus. Последняя является самой простой и используется в СнК Murax.

Ниже приведено определение класса **DBusSimplePlugin**, из которого можно понять, какими фичами обладает данный плагин.

```
class DBusSimplePlugin(catchAddressMisaligned : Boolean = false,
                       catchAccessFault : Boolean = false,
                       earlyInjection : Boolean = false,
                              /*, idempotentRegions : (UInt) => Bool = (x) => False*/
                       emitCmdInMemoryStage : Boolean = false,
                       onlyLoadWords : Boolean = false,
                       withLrSc : Boolean = false,
                       val bigEndian : Boolean = false,
                       memoryTranslatorPortConfig : Any = null) extends Plugin[VexRiscv] with
                              DbusAccessService
```

Видно, что **DBusSimplePlugin** имеет порт для подключения транслятора адресного пространства памяти (MMU), который тоже является отдельным плагином.

DBusCachedPlugin — создает шину данных для вычислительно ядра, но с организацией кеша данных. Позволяет подключать ядро к шинам: Axi4Shared, Avalon, Withbone, Bmb и PipelinedMemoryBus. Используется в СнК Briey с шиной Axi4Shared. Как и его упрощенный собрат имеет порт для подключения транслятора адресного пространства памяти.

Определение класса DBusCachedPlugin и его параметров следующее:

```
class DBusCachedPlugin(val config : DataCacheConfig,
                       memoryTranslatorPortConfig : Any = null,
                       var dBusCmdMasterPipe : Boolean = false,
                       dBusCmdSlavePipe : Boolean = false,
                       dBusRspSlavePipe : Boolean = false,
                       relaxedMemoryTranslationRegister : Boolean = false,
                       csrInfo : Boolean = false,
                       tightlyCoupledAddressStage : Boolean = false)  extends Plugin[VexRiscv] with
                              DBusAccessService with DBusEncoding
                              Service with VexRiscvRegressionArg
```

Данный плагин, используя сервис от плагина **CsrPlugin**, может экспортировать читаемый **csrInfo** регистр **0xCC0** с двумя полями [7:0] и [27:20] содержащий информацию о кеше:

```
if(csrInfo){
  val csr = service(classOf[CsrPlugin])
  csr.r(0xCC0, 0 ->  U(cacheSize/wayCount),  20 ->  U(bytePerLine))
}
```

IBusSimplePlugin — обеспечивает шину для передачи инструкций в вычислительное ядро, без организации кеша. Позволяет подключать ядро к шинам вида Avalon, Withbone, AhbLite3, Bmb, PipelinedMemoryBus и Axi4ReadOnly. Используется в СнК Murax. По сути, данный плагин является Fetch ступенью конвейера.

Описание класса **IBusSimplePlugin** и его параметров выглядит следующим образом:

```
class IBusSimplePlugin(    resetVector : BigInt,
                           val cmdForkOnSecondStage : Boolean,
                           val cmdForkPersistence : Boolean,
                           val catchAccessFault : Boolean = false,
                               prediction : BranchPrediction = NONE,
                               historyRamSizeLog2 : Int = 10,
                               keepPcPlus4 : Boolean = false,
                               compressedGen : Boolean = false,
```

```
    val busLatencyMin : Int = 1,
    val pendingMax : Int = 7,
        injectorStage : Boolean = true,
    val rspHoldValue : Boolean = false,
    val singleInstructionPipeline : Boolean = false,
    val memoryTranslatorPortConfig : Any = null,
        relaxPredictorAddress : Boolean = true,
        predictionBuffer : Boolean = true,
        bigEndian : Boolean = false,
        vecRspBuffer : Boolean = false
) extends IbusFetcherImpl
```

Данный плагин имеет множество интересных параметров. Например, параметр **resetVector** задает адрес в памяти, с которого вычислительное ядро начнет исполнение программы; параметр **compressedGen** позволяет подключить код для декодера сжатого набора инструкций RV32C; параметр **prediction** позволяет подключать код предсказателя ветвлений, тоже в виде отдельного плагина.

IBusCachedPlugin — обеспечивает шину для передачи инструкций в вычислительный модуль с использованием кеша. Функционал этого плагина аналогичен **IBusSimplePlugin.** Объявление класса **IBusCachedPlugin** выглядит следующим образом:

```
class IBusCachedPlugin(resetVector : BigInt = 0x80000000l,
                       relaxedPcCalculation : Boolean = false,
                       prediction : BranchPrediction = NONE,
                       historyRamSizeLog2 : Int = 10,
                       compressedGen : Boolean = false,
                       keepPcPlus4 : Boolean = false,
                       val config : InstructionCacheConfig,
                       memoryTranslatorPortConfig : Any = null,
                       injectorStage : Boolean = false,
                       withoutInjectorStage : Boolean = false,
                       relaxPredictorAddress : Boolean = true,
                       predictionBuffer : Boolean = true)  extends IbusFetcherImpl
```

DecoderSimplePlugin — обеспечивает другие плагины сервисом дешифрации инструкций.

```
class DecoderSimplePlugin(catchIllegalInstruction : Boolean = false,
                          throwIllegalInstruction : Boolean = false,
                          assertIllegalInstruction : Boolean = false,
                          forceLegalInstructionComputation : Boolean = false,
                          decoderIsolationBench : Boolean = false,
                          stupidDecoder : Boolean = false) extends Plugin[VexRiscv] with
DecoderService
```

Имеет опции для включения/отключения отлова нелегальных (нереализованных) инструкций: если параметр **catchIllegalInstruction** установлен в True, то ядро будет формировать trap и соответствующее исключение, если на вход дешифратора попадет инструкция несоответствующая текущему набору.

Добавление инструкций в дешифратор осуществляется вызовом метода **decoderService.add** следующим образом:

```
decoderService.add(
      //Bit pattern of the new instruction
      key = M"0000011----------000-----0110011",

      //Decoding specification when the 'key' pattern is recognized in the instruction
      List(
        IS_SIMD_ADD             -> True, //Inform the pipeline that the current instruction
is a SIMD_ADD instruction
        REGFILE_WRITE_VALID     -> True, //Notify the hazard management unit that this
instruction writes to the register file
```

```
        BYPASSABLE_EXECUTE_STAGE -> True, //Notify the hazard management unit that the
instruction result is already accessible in the EXECUTE stage (Bypass ready)
        BYPASSABLE_MEMORY_STAGE  -> True, //Same as above but for the memory stage
        RS1_USE                  -> True, //Notify the hazard management unit that this
instruction uses the RS1 value
        RS2_USE                  -> True  //Same than above but for RS2.
      )
    )
```

RegFilePlugin — обеспечивает стандартный регистровый файл согласно спецификации RISC-V. Описание класса **RegFilePlugin** следующее:

```
class RegFilePlugin(regFileReadyKind : RegFileReadKind,
                    zeroBoot : Boolean = false,
                    x0Init : Boolean = true,
                    writeRfInMemoryStage : Boolean = false,
                    readInExecute : Boolean = false,
                    syncUpdateOnStall : Boolean = true,
                    rv32e : Boolean = false,
                    withShadow : Boolean = false //shadow registers aren't transition hazard
                              free
                   ) extends Plugin[VexRiscv] with RegFileService
```

Данный плагин имеет ряд опций для оптимизации. Параметр **regFileReadyKind** задает тип используемых ячеек памяти для регистров: SYNC или ASYNC. Первый вариант (SYNC) — будут задействованы синхронные D-триггеры (задержка в 1 такт), хорошо ложится на стандартные LUT ячейки ПЛИС. Вариант ASYNC позволяет использовать асинхронные ячейки памяти SRAM и BRAM (без задержки), подходит для проектирования микросхем (ASIC).

HazardSimplePlugin — реализует очень важный для конвейерных процессоров механизм называемый «Hazard», суть которого состоит в том, чтобы отслеживать взаимозависимости между разными инструкциями, исполняемыми на разных ступенях конвейера в один и тот же момент времени. В случае выявления зависимостей, данный плагин может придержать (остановить) исполнение инструкции в ступени дешифратора (Decoder) или, если есть такая возможность, передать результат с более поздних ступеней, чтобы не задерживать исполнение. Описание класса **HazardSimplePlugin** и его параметров приведено ниже:

```
class HazardSimplePlugin(bypassExecute : Boolean = false,
                         bypassMemory: Boolean = false,
                         bypassWriteBack: Boolean = false,
                         bypassWriteBackBuffer : Boolean = false,
                         pessimisticUseSrc : Boolean = false,
                         pessimisticWriteRegFile : Boolean = false,
                         pessimisticAddressMatch : Boolean = false) extends Plugin[VexRiscv]
with HazardService
```

Параметры **bypassExecute**, **bypassMemory**, **bypassWriteBack** разрешают передачу результата из соответствующих ступеней на стадию **Decoder**.

IntAluPlugin — плагин, реализующий целочисленное АЛУ. Встраивается в стадию Execute и реализует такие инструкции как ADD, SUB, SLT, SLTU, XOR, OR, AND, LUI и AUIPC. Это очень простой плагин не имеющий параметров, определение класса **IntAluPlugin** выглядит следующим образом:

```
class IntAluPlugin extends Plugin[VexRiscv]
```

LightShifterPlugin и **FullBarrelShifterPlugin** — эти плагины реализуют инструкции побитного сдвига SLL, SRL и SRA. Плагин **LightShifterPlugin** является простой многотактовой реализацией сдвига, выполняющий задачу за N тактов, где N — число бит на которое требуется выполнить сдвиг. **FullBarrelShifterPlugin** — выполняет полный сдвиг на заданное число бит за один такт. Полная реализация является очень ресурсоемкой (требует большого количества логических элементов) и вносит большую задержку в исполнение, а значит существенно уменьшает максимальную тактовую частоту ядра. Определения классов:

```
class FullBarrelShifterPlugin(earlyInjection : Boolean = false) extends Plugin[VexRiscv]
```

и

```
class LightShifterPlugin extends Plugin[VexRiscv]
```

FpuPlugin — плагин, реализующий инструкции с плавающей запятой («Floating Point Unit» - FPU). Блок FPU может быть как встроенным в конвейер, так и использовать внешний FPU блок (внешний сопроцессор). Описание класса **FpuPlugin** выглядит следующим образом:

```
class FpuPlugin(externalFpu : Boolean = false,
                var simHalt : Boolean = false,
                val p : FpuParameter) extends Plugin[VexRiscv] with VexRiscvRegressionArg
```

Параметр **p** является сложной структурой с настройками блока FPU, определение этой структуры выглядит следующим образом:

```
case class FpuParameter( withDouble : Boolean,
                         asyncRegFile : Boolean = false,
                         mulWidthA : Int = 18,
                         mulWidthB : Int = 18,
                         schedulerM2sPipe : Boolean = false,
                         sim : Boolean = false,
                         withAdd : Boolean = true,
                         withMul : Boolean = true,
                         withDivSqrt : Boolean = false,
                         withDiv : Boolean = true,
                         withSqrt : Boolean = true,
                         withShortPipMisc : Boolean = true)
```

В файле README.md репозитория VexRiscv приводится детальное описание устройства блока FPU и его настроек. Ознакомиться можно по ссылке: https://github.com/SpinalHDL/VexRiscv/blob/master/README.md#fpu

PcManagerSimplePlugin — простой плагин реализует регистр счетчика команд (PC). Данный плагин в текущей версии ядра не используется, а регистр счетчика команд интегрирован в базовый функционал ядра, в класс **VexRiscv**.

BranchPlugin — данный плагин осуществляется предсказание ветвлений с целью оптимизации хода исполнения программы, так как каждый «промах» (загрузка неверной инструкции в декодер) может иметь стоимость в 2 или 4 такта. Этот плагин уменьшает негативное влияния команд ветвления на кеш инструкций. Плагин реализует все инструкции ветвления: JAL, JALR, BEQ, BNE, BLT, BGE, BLTU и BGEU. Определение класса **BranchPlugin**:

```
class BranchPlugin(earlyBranch : Boolean,
                   catchAddressMisaligned : Boolean = false,
                   fenceiGenAsAJump : Boolean = false,
```

```
                    fenceiGenAsANop : Boolean = false,
                    decodeBranchSrc2 : Boolean = false) extends Plugin[VexRiscv] with
PredictionInterface
```

В VexRiscv реализованы следующие алгоритмы предсказания ветвлений, которые задаются параметром prediction и плагина **IBusSimplePlugin** или **IBusCachedPlugin** (напомню, что эти два плагина реализуют Fetch ступень конвейера):

- NONE — без предсказания, любое изменение регистра PC инструкциями ветвления приводит к полному набору «штрафных» циклов (от 2 до 4).
- STATIC — спекулятивное предсказание, фактически всегда выбирается одна и та же ветвь выполнения инструкции ветвления. Если предсказание сбылось, то «штраф» уменьшается до одного дополнительного такта, иначе «штраф» будет полным.
- DYNAMIC — для предсказания используется статистика, т. н. BHT кеш, который определяет какое из направлений ветвления будет более вероятным.
- DYNAMIC_TARGET — использует для предсказания т. н. «целевой буфер ветвления с прямым отображением» (BTB) в ступени Fetch, который сохраняет значение регистра PC, где расположена инструкция ветвления, значение PC по которому был осуществлен переход и два бита истории/вероятности. Это наиболее эффективный вариант предсказания ветвлений, так как если предсказание угадано, то никаких «штрафных» тактов не происходит. Существенным недостатком является очень длинная комбинационная цепочка тянущаяся от кеша предсказаний до регистра счетчика команд, проходящая через интерфейс ветвлений. Эта комбинационная цепочка существенно ограничивает максимальную тактовую частоту ядра.

На этом я закончу обзор плагинов вычислительного ядра VexRiscv, желающие узнать больше — могут почерпнуть информацию из документации, расположенной в репозитории этого проекта, ссылки на которую я приводил уже несколько раз выше по тексту.

16. Эксперименты с ядром VexRiscv и СнК Murax

Теперь, когда мы немного ознакомились с внутренним устройством вычислительного ядра VexRiscv, настало время немного поэкспериментировать с различными параметрами ядра VexRiscv и СнК Murax, то есть начать его модифицировать под свои нужны.

Далее я буду последовательно модифицировать код мастер репозитория VexRiscv, демонстрируя как действовал я, добавляя новые возможности в СнК и постепенно адаптируя его под поставленные задачи. Все внесенные мной изменения можно получить из отдельного репозитория на Github-е: по ссылке: https://github.com/Fabmicro-LLC/VexRiscvWithKarnix.git

16.1 Оптимизация на примере плагина RegFilePlugin

Напомню, что у плагина **RegFilePlugin** имеется параметр, позволяющий задавать тип используемой памяти для организации регистрового файла. В СнК Murax по умолчанию используется синхронный вариант построения регистров, т. е. **regFileReadyKind** установлена

в значение **SYNC.** Так как микросхема ПЛИС Lattice ECP5 имеет распределенные блоки двухпортовой статической памяти, из которых можно соорудить асинхронный регистровый файл, то резонно проверить насколько это может быть (или не быть) эффективным.

Отредактируем файл **Murax.scala** — найдем в нем добавление плагина, и заменим строку кода:

```
new RegFilePlugin(
  regFileReadyKind = plugin.SYNC,
  zeroBoot = false
),
```

на строку:

```
new RegFilePlugin(
  regFileReadyKind = plugin.ASYNC,
  zeroBoot = false
),
```

Запустим пересборку проекта командой **make** и понаблюдаем за статистикой утилизируемых ресурсов и рассчитываемых максимальных значений тактовых частот. Вот что мы должны увидеть:

Для режима ASYNC:

```
Info: Device utilisation:
..
Info:             TRELLIS_FF:  1325/24288     5%
Info:           TRELLIS_COMB:  2284/24288     9%
Info:           TRELLIS_RAMW:    36/ 3036     1%

Info: Max frequency for clock         '$glbnet$io_clk25$TRELLIS_IO_IN': 74.47 MHz (PASS at
25.00 MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 128.97 MHz (PASS at
12.00 MHz)
```

Сравним эти же показатели для режима SYNC который используется по умолчанию:

```
Info:             TRELLIS_FF:  1379/24288     5%
Info:           TRELLIS_COMB:  2346/24288     9%
Info:           TRELLIS_RAMW:    36/ 3036     1%

Info: Max frequency for clock         '$glbnet$io_clk25$TRELLIS_IO_IN': 69.80 MHz (PASS at
25.00 MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 149.30 MHz (PASS at
12.00 MHz)
```

Видно, что в режиме ASYNC уменьшился расход Flip-Flop триггеров и комбинационных ячеек LUT-4. Также видно, что максимально допустимая частота для основного тактового сигнала **io_clk25** существенно подросла — с 69.80 МГц до 74.47. МГц. Это означает, что мы, теоретически, имеем возможность увеличить частоту тактирования ядра до 74 МГц или даже немного выше! Это можно сделать, не изменяя кварцевого осциллятора, путем преобразования частоты с помощью встроенного в микросхему ПЛИС блока PLL.

16.2 Увеличиваем тактовую частоту ядра используя встроенный PLL

Если кратко, то PLL или «Phase-locked Loop» («Фазовая автоподстройка частоты», «ФАПЧ») это такой блок аппаратуры, который, используя ряд аналоговых ухищрений, позволяет делить и умножать тактовые частоты в определенных пределах и с определенной точностью. Микросхема ПЛИС Lattice ECP5 содержит два таких аппаратных блока, которые называются **EHXPLLL**. Тулчейн Yosys поддерживает эти блоки, а значит, мы можем задействовать один из них для преобразования частоты, поступающей от кварцевого осциллятора распаянного на плате «Карно» номиналом 25.0 МГц в (почти) любую другую

частоту. В этом примере я продемонстрирую, как добавить PLL, настроить его, получить на выходе тактовый сигнал частотой 75.0 МГц и тактировать от него СнК Murax и вычислительное ядро в его составе.

PLL — это достаточно сложный блок аппаратуры, имеющий ряд параметров, изменяя которые можно получать разные характеристики выходного тактового сигнала. Чтобы облегчить работу дизайнеру, в составе тулчейна Yosys имеется утилита **ecppll**, задав которой пару исходных параметров (значения частоты входного и выходного сигнала), можно получить код на языке Verilog. Этот код при синтезе с помощью Yosys подключит и задействует один из аппаратных блоков EHXPLLL.

Запустим **ecppll** со следующими параметрами:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ ecppll --clkin_name in_clk25 -i 25 --clkout0_name
out_clk75 -o 75 -f pll25to75.v

Pll parameters:
Refclk divisor: 1
Feedback divisor: 3
clkout0 divisor: 8
clkout0 frequency: 75 MHz
VCO frequency: 600
```

Здесь мы указываем утилите номиналы входной и выходной частоты, а также наименования сигнальных линий для входа и выхода. Утилита **ecppll** сформирует и запишет код на языке Verilog для модуля PLL, с использованием встроенного блока EHXPLLL, в **файл pll25to75.v**. Посмотрим на его содержимое:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ cat pll25to75.v

// diamond 3.7 accepts this PLL
// diamond 3.8-3.9 is untested
// diamond 3.10 or higher is likely to abort with error about unable to use feedback signal
// cause of this could be from wrong CPHASE/FPHASE parameters
module pll
(
    input in_clk25, // 25 MHz, 0 deg
    output out_clk75, // 75 MHz, 0 deg
    output locked
);
(* FREQUENCY_PIN_CLKI="25" *)
(* FREQUENCY_PIN_CLKOP="75" *)
(* ICP_CURRENT="12" *) (* LPF_RESISTOR="8" *) (* MFG_ENABLE_FILTEROPAMP="1" *) (*
MFG_GMCREF_SEL="2" *)
EHXPLLL #(
        .PLLRST_ENA("DISABLED"),
        .INTFB_WAKE("DISABLED"),
        .STDBY_ENABLE("DISABLED"),
        .DPHASE_SOURCE("DISABLED"),
        .OUTDIVIDER_MUXA("DIVA"),
        .OUTDIVIDER_MUXB("DIVB"),
        .OUTDIVIDER_MUXC("DIVC"),
        .OUTDIVIDER_MUXD("DIVD"),
        .CLKI_DIV(1),
        .CLKOP_ENABLE("ENABLED"),
        .CLKOP_DIV(8),
        .CLKOP_CPHASE(4),
        .CLKOP_FPHASE(0),
        .FEEDBK_PATH("CLKOP"),
        .CLKFB_DIV(3)
    ) pll_i (
        .RST(1'b0),
        .STDBY(1'b0),
        .CLKI(io_clk25),
        .CLKOP(clk75),
        .CLKFB(clk75),
        .CLKINTFB(),
        .PHASESEL0(1'b0),
        .PHASESEL1(1'b0),
        .PHASEDIR(1'b1),
```

```
        .PHASESTEP(1'b1),
        .PHASELOADREG(1'b1),
        .PLLWAKESYNC(1'b0),
        .ENCLKOP(1'b0),
        .LOCK(locked)
        );
endmodule
```

Видно, что утилита **ecppll** произвела расчет значений большого числа параметров и подставила их в модуль **EHXPLLL**. Описание всех параметров доступно в документации на микросхему, но рассчитать их значения вручную будет не просто.

Теперь осталось подключить этот файл в **toplevel.v** и произвести коммутацию тактовых сигналов: внешний сигнал **io_clk25** необходимо подать на вход **in_clk25** модуля **pll**, а выходной сигнал **out_clk75** через дополнительный проводник **clk75** подсоединить к сигналу тактирования **mainClk** модуля **Murax**. На Verilog-е это выглядит так:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ cat toplevel.v

`timescale 1ns / 1ps

`include "pll25to75.v"

module toplevel(
    input   io_clk25,
    input   [3:0] io_key,
    output  [3:0] io_led,
    input   io_core_jtag_tck,
    output  io_core_jtag_tdo,
    input   io_core_jtag_tdi,
    input   io_core_jtag_tms,
    output  io_uart_debug_txd,
    input   io_uart_debug_rxd
  );

  assign io_led[0] = io_gpioA_write[0];
  assign io_led[1] = io_gpioA_write[1];
  assign io_led[2] = io_gpioA_write[2];
  assign io_led[3] = io_gpioA_write[3];

  wire [31:0] io_gpioA_read;
  wire [31:0] io_gpioA_write;
  wire [31:0] io_gpioA_writeEnable;

  assign io_gpioA_read[0] = io_key[0];
  assign io_gpioA_read[1] = io_key[1];
  assign io_gpioA_read[2] = io_key[2];
  assign io_gpioA_read[3] = io_key[3];

  wire clk75;

  pll i_pll(.in_clk25(io_clk25), .out_clk75(clk75));

  Murax murax (
    .io_asyncReset(io_key[3]),
    //.io_mainClk (io_clk25),
    .io_mainClk (clk75),
    .io_jtag_tck(io_core_jtag_tck),
    .io_jtag_tdi(io_core_jtag_tdi),
    .io_jtag_tdo(io_core_jtag_tdo),
    .io_jtag_tms(io_core_jtag_tms),
    .io_gpioA_read       (io_gpioA_read),
    .io_gpioA_write      (io_gpioA_write),
    .io_gpioA_writeEnable(io_gpioA_writeEnable),
    .io_uart_txd(io_uart_debug_txd),
    .io_uart_rxd(io_uart_debug_rxd)
  );

endmodule
```

Далее, укажем программе «hello_world» на тот факт, что тактовая частота процессора изменилась, это требуется для правильного расчета задержек в программе. Для этого в файле **./src/main/c/murax/hello_world/src/murax.h** заменим значение константы **SYSTEM_CLOCK_HZ** на **75000000L** и выполним пересборку Си программы:

```
rz@devbox:~/VexRiscv/src/main/c/murax/hello_world$ make clean && make
```

Не забываем установить **regFileReadyKind = plugin.ASYNC** для плагина **RegFilePlugin** (см. предыдущую главу), а также указать для СнК Murax новую частоту в параметре **coreFrequency = 75 Mhz**, после чего запускаем сборку проекта и наблюдаем:

```
rz@devbox:~/VexRiscvWithKarnix/scripts/Murax/Karnix$ make

...
Info: Device utilisation:
Info:             TRELLIS_IO:    15/  197     7%
Info:                   DCCA:     2/   56     3%
Info:                DP16KD:    48/   56    85%
...
Info:               EHXPLLL:     1/    2    50%
...
Info:             TRELLIS_FF:  1325/24288     5%
Info:           TRELLIS_COMB:  2270/24288     9%
Info:           TRELLIS_RAMW:    36/ 3036     1%
...
Info: Routing globals...
Info:     routing clock net $glbnet$io_core_jtag_tck$TRELLIS_IO_IN using global 0
Info:     routing clock net $glbnet$clk75 using global 1
...
Info: Max frequency for clock                          '$glbnet$clk75': 75.55 MHz (PASS at
75.00 MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 124.66 MHz (PASS at
12.00 MHz)
```

Хохо! Наш дизайн прошел контроль STA по самому краешку — 75 МГц из 75.55 МГц разрешенных.

Подключаем плату «Карно», загружаем битстрим командой **make upload** и наблюдаем за тем, как «пляшут» светодиоды в три раза быстрее прежнего.

16.3 Увеличиваем производительность инструкций сдвига

В главе «*15.3 Плагины вычислительного ядра VexRiscv*» я упоминал о том, что автор VexRiscv снабдил своё вычислительное ядро двумя реализациями (двумя плагинами) инструкций побитового сдвига: **LightShifterPlugin** — многотактовая реализация, выполняющая сдвиг на N бит за N тактов, и **FullBarrelShifterPlugin** — полная реализация («бочковой сдвигатель»), выполняющая операции сдвига на N бит за 1 такт. Очевидно, что последний является более интересным с точки зрения производительности, но имеет существенный недостаток — потребляет большое количество логических элементов и имеет достаточно длинную цепочку комбинационной логики (длинный критический путь), что может существенно увеличить время работы стадии конвейера Execute и сократить максимальную частоту тактирования ядра. По умолчанию в СнК Murax используется облегченный вариант, сделано это с целью экономии ресурсов микросхемы и демонстрации минимально возможного варианта. Давайте же перенастроим наше вычислительное ядро на использование полной реализации сдвигателя и посмотрим, во что это выливается.

Прежде чем мы приступим к модификации аппаратуры, нам необходимо добавить небольшой код на языке Си в программу «hello_world» для измерения производительности инструкций сдвига. Для этого воспользуемся уже имеющимся в СнК Murax блоком таймеров **MuraxApb3Timer,** который состоит из двух 16-ти битных таймеров **timerA** и **timerB** и одного общего 16-ти битного прескейлера (делителя частоты). Проинициализируем прескейлер так, чтобы таймер вел расчет в миллисекундах, т. е. занесем в прескейлер значение **(SYSTEM_CLOCK_HZ / 1000000 - 1)**. Далее, добавим в код функции **delay()** ассемблерную инструкцию сдвига на 16, а время исполнения функции **delay()** будем измерять одним из таймеров и выводить в порт UART в виде шестнадцатеричного числа.

Файл **main.c** программы «hello_world» примет следующие изменения:

```
void delay(uint32_t loops){
        for(int i=0;i<loops;i++){
                //int tmp = GPIO_A->OUTPUT;
                asm volatile ("slli t1,t1,0x10");
        }
}

void printhex(unsigned int number){
        unsigned int mask = 0xf0000000;
        static char hex_digits[11] = {0,0,0,0,0,0,0,0,'\r','\n','\0'};

        for(int i = 0; i < 8; i++) {
                int digit = (number & mask) >> 4*(7-i);
                if(digit < 0x0a)
                        hex_digits[i] = digit + 0x30;
                else
                        hex_digits[i] = (digit - 0x0a) + 'A';
                mask = mask >> 4;
        }

        print(hex_digits);
}

const int nleds = 4;
const int nloops = 1000000;
TIMER_PRESCALER->LIMIT = 0xFFFF; // Set max possible clock divider
while(1){
        unsigned int shift_time;
        println("Hello world, this is VexRiscv!");
        for(unsigned int i=0;i<nleds-1;i++){
                GPIO_A->OUTPUT = 1<<i;
                TIMER_A->CLEARS_TICKS = 0x00020002;
                TIMER_A->LIMIT = 0xFFFF;
                delay(nloops);
                shift_time = TIMER_A->VALUE;
        }
        for(unsigned int i=0;i<nleds-1;i++){
                GPIO_A->OUTPUT = (1<<(nleds-1))>>i;
                delay(nloops);
        }

        printhex(shift_time);
}
```

Жирным шрифтом показаны новые или измененные строки кода. Запускаем сборку и проверяем что у нас получилось:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ make clean && make
```

Если все прошло успешно, т. е. в программе нет синтаксических ошибок, то произойдет компиляция, синтез и полная сборка битстрима.

Как обычно, подключаем плату «Карно» и загружаем битстрим командой **make upload**. После этого подключаемся к отладочному UART с помощью терминала **minicom** и наблюдаем сообщения вида:

rz@butterfly:~ % sudo minicom -b 115200 -D /dev/ttyU1

```
Hello world, this is VexRiscv!
000001AC
Hello world, this is VexRiscv!
000001AC
Hello world, this is VexRiscv!
000001AB
Hello world, this is VexRiscv!
000001AB
```

Выводимое шестнадцатеричное число — это количество отсчетов таймера, которое произошло за момент выполнения модифицированной функции **delay**(). Мы можем приближенно пересчитать во сколько тактов исполняется одна итерация цикла внутри этой функции: (0xFFFF+1) * 0x01AC / 1000000 = **28** тактов на цикл. Где 0xFFFF — делитель (прескейлер), а 1000000 число итераций цикла.

Теперь модифицируем СнК Murax и вместо **LightShifterPlugin** подключим **FullBarrelShifterPlugin**. Для этого отредактируем файл **Murax.scala**, найдем в нём переменную **cpuPlugins**, которой присваивается инициализированный массив плагинов, закомментируем плагин **LightShifterPlugin** и добавим **FullBarrelShifterPlugin**:

```
cpuPlugins = ArrayBuffer(
      new IBusSimplePlugin(
      ...
      ),
      new SrcPlugin(
        ...
      ),
      //new LightShifterPlugin,
      new FullBarrelShifterPlugin,
      new HazardSimplePlugin(
        ...
      ),
```

Еще раз пересоберем проект командой **make clean && make**, по окончанию сборки загрузим битстрим командой **make upload**, подключим **minicom** и наблюдаем сообщения вида:

```
Hello world, this is VexRiscv!
000000C7
Hello world, this is VexRiscv!
000000C7
Hello world, this is VexRiscv!
000000C6
Hello world, this is VexRiscv!
000000C6
Hello world, this is VexRiscv!
000000C7
```

Видно, что число отсчетов таймера значительно сократилось, а значит функция **delay()** стала работать быстрее. Пересчитаем в количество тактов: (0xFFFF+1) * 0x00C7 / 1000000 = **13**. Т.е. время исполнения одной итерации цикла внутри **delay()** сократилось на 15 тактов. И это полностью соответствует теории: машинная команда **slli t1,t1,0x10** которая составляет тело цикла, выполняет сдвиг регистра **t1** (временный регистр #1) влево на 16 бит. В первом случае она исполняется за 16 тактов, во втором — за один такт, т. е. на 15 тактов быстрее!

А сейчас посмотрим на статистику, полученную при синтезе.

При использовании плагина **LightShifterPlugin**, выполняющего облегченный вариант сдвига на N бит за N тактов:

```
Info: Device utilisation:
Info:             TRELLIS_IO:    15/  197     7%
Info:                   DCCA:     2/   56     3%
Info:                 DP16KD:    48/   56    85%
...
```

```
Info:                EHXPLLL:     1/    2    50%
...
Info:             TRELLIS_FF:  1325/24288     5%
Info:           TRELLIS_COMB:  2255/24288     9%
Info:           TRELLIS_RAMW:    36/ 3036     1%

Info: Max frequency for clock                           '$glbnet$clk75': 77.29 MHz (PASS at
75.00 MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 118.46 MHz (PASS at
12.00 MHz)
```

При использовании «полного бочкового сдвигателя», реализуемого плагином **FullBarrelShifterPlugin**, получаем следующую статистику:

```
Info: Device utilisation:
Info:             TRELLIS_IO:    15/  197     7%
Info:                   DCCA:     2/   56     3%
Info:                 DP16KD:    48/   56    85%
...
Info:                EHXPLLL:     1/    2    50%
...
Info:             TRELLIS_FF:  1353/24288     5%
Info:           TRELLIS_COMB:  2447/24288    10%
Info:           TRELLIS_RAMW:    36/ 3036     1%

Info: Max frequency for clock                           '$glbnet$clk75': 79.73 MHz (PASS at
75.00 MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 124.01 MHz (PASS at
12.00 MHz)
```

Интересным здесь является то, что не смотря на значительно больший расход ресурсов (дополнительно 28 триггеров и 192 логических блоков LUT-4), максимально допустимая тактовая частота синтезированной схемы даже немного возросла — с 77,29 МГц до 79,73 МГц.

Такое поведение тулчейна мне объяснить сложно. Моё предположение состоит в том, что код «бочкового сдвигателя» хорошо оптимизируется и синтезатором и плейсером. Отчасти это подтверждается статистикой, полученной после выполнения размещения (до оптимизации маршрутов) — уже на этом этапе видно, что «бочковой сдвигатель» более эффективен с точки зрения STA.

Статистика до оптимизации:

Для **LightShifterPlugin**:

```
Info: Max frequency for clock                           '$glbnet$clk75': 49.10 MHz (FAIL at
75.00 MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 156.32 MHz (PASS at
12.00 MHz)
```

Для **FullBarrelShifterPlugin**:

```
Info: Max frequency for clock                           '$glbnet$clk75': 51.05 MHz (FAIL at
75.00 MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 124.73 MHz (PASS at
12.00 MHz)
```

17. Добавляем свои аппаратные блоки (IP-блоки)

Конфигурация СнК Murax и ядра VexRiscv в его составе, которая представлена по умолчанию в репозитории VexRiscv, служит для целей демонстрации и имеет очень

ограниченные возможности. Настало время посмотреть, как можно расширить функционал Murax и создать свою СнК, подходящую для решения широкого спектра практических задач.

17.1 Микросекундный машинный таймер MTIME

Для дальнейшего исследования и экспериментов нам понадобится микросекундный машинный таймер, который монотонно возрастает и никогда не переполняется (или делает это очень редко). С таким таймером удобно работать при проведении измерений производительности, как, например, в предыдущей главе. Добавить такой таймер не сложно, язык SpinalHDL уже содержит все необходимые примитивы, требуется только сложить их вместе и подвязать получившийся IP-блок к периферийной шине **Apb3**. Сейчас мы это и проделаем.

Схема нашего машинного микросекундного таймера будет содержать два регистра-счетчика. Первый счетчик, **counter** размером 8 бит, будет отсчитывать число тактов до полной микросекунды, опираясь на значение параметра **clockMHz**, который будет задаваться извне и содержать значение частоты системного клока в мегагерцах. В нашем случае, при частоте **clockMHz =** 75.0 МГц, этот счетчик будет принимать максимальное значение 75 - 1 = 74. Второй регистр-счетчик, **microCounter** размерностью 32 бита, это собственно наш микросекундный таймер, который будет прирастать на единицу каждый раз когда первый счетчик достигает значения **clockMHz — 1**. Сигнал **io.micros** будет ссылаться на значение счетчика **microCounter** и представлять внешний интерфейс (сигнал) нашего машинного таймера. Вот как это выглядит на языке SpinalHDL:

```
package mylib

import spinal.core._
import spinal.lib._
import spinal.lib.bus.amba3.apb._
import spinal.lib.bus.misc._

case class MachineTimer(clockMHz: Int = 25) extends Component {
  val io = new Bundle {
    val micros = out UInt(32 bits)
  }

  val counter = Reg(UInt(8 bits))
  val microCounter = Reg(UInt(32 bits))
  io.micros := microCounter

  counter := counter + 1

  when (counter === clockMHz - 1) {
    counter := 0;
    microCounter := microCounter + 1
  }

  def driveFrom(busCtrl : BusSlaveFactory, baseAddress : Int = 0) () = new Area {
    busCtrl.read(io.micros, baseAddress)
  }
}
```

Метод **driveFrom()** обеспечивает привязку интерфейсного сигнала **io.micros** к шине **BusSlaveFactory** при чтении слова по адресу **baseAddress**. **BusSlaveFactory** является внешней шиной для моста **Apb3**, смотрящей в сторону периферии, а наше новое устройство будет являться слэйвом на этой шине.

Для привязки нашего нового компонента потребуется создать еще один, оберточный, компонент **Apb3MachineTimer** - он будет инкапсулировать в себя компонент **MachineTimer** и интерфейсный класс **Apb3SlaveFactory. Apb3SlaveFactory** является интерфейсом для моста **Apb3.** Оберточный компонент также будет содержать одну строку кода,

обеспечивающую связь между новым компонентом и **Apb3**. Код для этого оберточного компонента на SpinalHDL выглядит следующим образом:

```
case class Apb3MachineTimer(clockMHz : Int = 25) extends Component {
  val io = new Bundle {
    val apb = slave(Apb3(Apb3Config(addressWidth = 8, dataWidth = 32)))
  }

  val busCtrl = Apb3SlaveFactory(io.apb)
  val mtCtrl = MachineTimerCtrl(clockMHz)

  mtCtrl.driveFrom(busCtrl)() // connect component to the bus
}
```

Строго говоря, этот оберточный компонент не является обязательным, но тогда нам пришлось бы вытащить этот код на верхний уровень, в Murax, что не очень эстетично.

Разместим весь этот код в одном файле **./src/main/scala/mylib/MachineTimer.scala**, при этом подкаталог **./src/main/scala/mylib** потребуется создать.

Теперь, для того чтобы подключить наш новый компонент к шине Apb3 внутри СнК Murax, наравне с другой периферией, необходимо добавить его в ассоциативный список, содержащийся в переменной **apbMapping**, указав адрес привязки **0xB0000**. Вставим выделенный жирным код сразу после подключения уже существующего таймера:

```
    val timer = new MuraxApb3Timer()
    timerInterrupt setWhen(timer.io.interrupt)
    apbMapping += timer.io.apb     -> (0x20000, 4 kB)

    val machineTimer = Apb3MachineTimer(coreFrequency.toInt / 1000000)
    apbMapping += machineTimer.io.apb   -> (0xB0000, 4 kB)
```

Таким образом, регистр нашего микросекундного таймера будет доступен из программы при чтении 32-х битного слова по адресу: 0xF0000000 + 0xB0000 = **0xF00B0000**. Для того, чтобы при сборке SBT нашел код нашего нового компонента, поможем ему в этом, добавив в заголовке файла Murax.scala следующу строку:

```
import mylib.Apb3MachineTimer
```

Чтобы проверить, как работает созданный нами машинный микросекундный таймер, добавим в файл **src/murax.h** макро **MTIME** для доступа к регистру:

```
#define MTIME   (*(volatile unsigned long*)(0xF00B0000))
```

А в код программы «hello_world» добавим измерение времени выполнения функции delay() с помощью микросекундного таймера. Модифицированный Си код в файле **src/main.c** приведен ниже:

```
while(1){
        unsigned int shift_time;
        unsigned int t1, t2;
        println("Hello world, this is VexRiscv!");
        for(unsigned int i=0;i<nleds-1;i++){
                GPIO_A->OUTPUT = 1<<i;
                TIMER_A->CLEARS_TICKS = 0x00020002;
                TIMER_A->LIMIT = 0xFFFF;
                t1 = MTIME;
                delay(nloops);
                t2 = MTIME;
                shift_time = TIMER_A->VALUE;
        }
        for(unsigned int i=0;i<nleds-1;i++){
                GPIO_A->OUTPUT = (1<<(nleds-1))>>i;
                delay(nloops);
        }
```

```
                printhex(shift_time);
                printhex(t2 - t1);
        }
```

Как обычно, выполняем сборку проекта командой **make clean && make**, устраняем все синтаксические ошибки и опечатки, подключаем плату «Карно» и загружаем полученный битстрим. Запускаем эмулятор терминала **minicom** и наблюдаем в порту следующие строки:

```
Hello world, this is VexRiscv!
000000C7
0002A516
Hello world, this is VexRiscv!
000000C6
0002A515
Hello world, this is VexRiscv!
000000C6
0002A515
```

Здесь первое число — это измерение числа тактов (умноженное на 65536), выполненное таймером TIMER_A, а второе число — результат замеров с помощью микросекундного таймера. Число 0002A515 есть не что иное как 173333 микросекунд, которые требуются для выполнения функции **delay(1000000)**.

Теперь, когда у нас есть микросекундный таймер, мы можем легко соорудить сервисные функции **delay_us()** и **delay_ms()**, осуществляющие задержку исполнения программы на заданное количество микро- и миллисекунд:

```
void delay_us(unsigned int us){
        unsigned int t = MTIME;
        while(MTIME - t < us);
}

void delay_ms(unsigned int ms){
        for(int i = 0; i < ms; i++)
                delay_us(1000);
}
```

17.2 Подключаем микросхему SRAM

На плате «Карно» имеется распаянная и подключенная к ПЛИС микросхема статической памяти K6R4016V1D-10 объемом 16x256 КБ (512 КБ) и было бы очень неплохо задействовать эту память в своих программах, исполняемых на синтезированном ядре. Эта микросхема имеет стандартный набор сигнальных линий:

- **A0-A9** (линии выбора строки) и **A10-A17** (линии выбора столбца) вместе формируют адрес 16-битного слова, т. е. микросхема содержит всего 262144 слов по 16 бит.
- **D0-D15** — линии данных.
- **#CS** — сигнал выбора микросхемы, лог «0» — выбрана.
- **#WE** — сигнал разрешения записи в память, лог «0» - запись, лог «1» - чтение.
- **#OE** — сигнал разрешения выставить данные на выходную шину данных. Лог «0» - данные на выходе активны, лог «1» - линии данных находятся в состоянии высокого импеданса.
- **#LB** — сигнал, разрешающий запись младшего байта.
- **#UB** — сигнал, разрешающий запись старшего байта.

Опишем эти сигнальные линии в LPF файле, чтобы ими можно было пользоваться, добавим в файл **karnix_cabga256.lpf** следующие строки:

```
LOCATE COMP "io_sram_cs" SITE "H15";            # SRAM #CS
IOBUF PORT "io_sram_cs" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_we" SITE "K14";            # SRAM #WE
IOBUF PORT "io_sram_we" IO_TYPE=LVCMOS33;
```

```
LOCATE COMP "io_sram_oe" SITE "J14";            # SRAM #OE
IOBUF PORT "io_sram_oe" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_bhe" SITE "J16";           # SRAM #BHE
IOBUF PORT "io_sram_bhe" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_ble" SITE "J15";           # SRAM #BLE
IOBUF PORT "io_sram_ble" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[0]" SITE "L16";        # SRAM D00
IOBUF PORT "io_sram_dat[0]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[1]" SITE "L15";        # SRAM D01
IOBUF PORT "io_sram_dat[1]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[2]" SITE "M16";        # SRAM D02
IOBUF PORT "io_sram_dat[2]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[3]" SITE "M15";        # SRAM D03
IOBUF PORT "io_sram_dat[3]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[4]" SITE "K13";        # SRAM D04
IOBUF PORT "io_sram_dat[4]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[5]" SITE "K12";        # SRAM D05
IOBUF PORT "io_sram_dat[5]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[6]" SITE "L13";        # SRAM D06
IOBUF PORT "io_sram_dat[6]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[7]" SITE "L12";        # SRAM D07
IOBUF PORT "io_sram_dat[7]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[8]" SITE "N16";        # SRAM D08
IOBUF PORT "io_sram_dat[8]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[9]" SITE "P15";        # SRAM D09
IOBUF PORT "io_sram_dat[9]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[10]" SITE "L14";       # SRAM D10
IOBUF PORT "io_sram_dat[10]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[11]" SITE "M14";       # SRAM D11
IOBUF PORT "io_sram_dat[11]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[12]" SITE "P16";       # SRAM D12
IOBUF PORT "io_sram_dat[12]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[13]" SITE "R16";       # SRAM D13
IOBUF PORT "io_sram_dat[13]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[14]" SITE "M13";       # SRAM D14
IOBUF PORT "io_sram_dat[14]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_dat[15]" SITE "N14";       # SRAM D15
IOBUF PORT "io_sram_dat[15]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[0]" SITE "N13";       # SRAM A00
IOBUF PORT "io_sram_addr[0]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[1]" SITE "P14";       # SRAM A01
IOBUF PORT "io_sram_addr[1]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[2]" SITE "R15";       # SRAM A02
IOBUF PORT "io_sram_addr[2]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[3]" SITE "T15";       # SRAM A03
IOBUF PORT "io_sram_addr[3]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[4]" SITE "P13";       # SRAM A04
IOBUF PORT "io_sram_addr[4]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[5]" SITE "R14";       # SRAM A05
IOBUF PORT "io_sram_addr[5]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[6]" SITE "R13";       # SRAM A06
IOBUF PORT "io_sram_addr[6]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[7]" SITE "T14";       # SRAM A07
IOBUF PORT "io_sram_addr[7]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[8]" SITE "R12";       # SRAM A08
IOBUF PORT "io_sram_addr[8]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[9]" SITE "T13";       # SRAM A09
IOBUF PORT "io_sram_addr[9]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[10]" SITE "M12";      # SRAM A10
IOBUF PORT "io_sram_addr[10]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[11]" SITE "N12";      # SRAM A11
IOBUF PORT "io_sram_addr[11]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[12]" SITE "M11";      # SRAM A12
IOBUF PORT "io_sram_addr[12]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[13]" SITE "N11";      # SRAM A13
IOBUF PORT "io_sram_addr[13]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[14]" SITE "P11";      # SRAM A14
IOBUF PORT "io_sram_addr[14]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[15]" SITE "P12";      # SRAM A15
IOBUF PORT "io_sram_addr[15]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[16]" SITE "K16";      # SRAM A16
IOBUF PORT "io_sram_addr[16]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_sram_addr[17]" SITE "K15";      # SRAM A17
IOBUF PORT "io_sram_addr[17]" IO_TYPE=LVCMOS33;
```

И передадим эти сигналы из модуля **toplevel** в модуль **Murax** в оберточном файле **toplevel.v**, как показано в выдержке ниже:

```
module toplevel(
    input   io_clk25,
    input   [3:0] io_key,
    output  [3:0] io_led,
    input   io_core_jtag_tck,
    output  io_core_jtag_tdo,
    input   io_core_jtag_tdi,
    input   io_core_jtag_tms,
    output  io_uart_debug_txd,
    input   io_uart_debug_rxd,
    output  io_sram_cs,  // #CS
    output  io_sram_we,  // #WE
    output  io_sram_oe,  // #OE
    output  io_sram_bhe, // #UB
    output  io_sram_ble, // #LB
    output  [17:0] io_sram_addr, // A0-A17
    inout   [15:0] io_sram_dat   // D0-D15
  );

...

  Murax murax (
    .io_asyncReset(io_key[3]),
    //.io_mainClk (io_clk25),
    .io_mainClk (clk75),
    .io_jtag_tck(io_core_jtag_tck),
    .io_jtag_tdi(io_core_jtag_tdi),
    .io_jtag_tdo(io_core_jtag_tdo),
    .io_jtag_tms(io_core_jtag_tms),
    .io_gpioA_read       (io_gpioA_read),
    .io_gpioA_write      (io_gpioA_write),
    .io_gpioA_writeEnable(io_gpioA_writeEnable),
    .io_uart_txd(io_uart_debug_txd),
    .io_uart_rxd(io_uart_debug_rxd),
    .io_sram_cs(io_sram_cs),
    .io_sram_we(io_sram_we),
    .io_sram_oe(io_sram_oe),
    .io_sram_bhe(io_sram_bhe),
    .io_sram_ble(io_sram_ble),
    .io_sram_addr(io_sram_addr),
    .io_sram_dat(io_sram_dat)
  );
```

17.2.1 Разрабатываем контроллер SRAM

А теперь подумаем вот о чем. У нас есть типовой набор сигналов для интерфейса к микросхеме SRAM. Чтобы не таскать далее по всему коду эту пачку сигналов, имеет смысл описать её как комплексный сигнал, и тогда нам будет достаточно передавать одну переменную. На SpinalHDL такой комплексный сигнал можно описать следующим образом.

Создадим класс **SramInterface** с двойным наследованием: от класса **Bundle** (сложная структура сигналов) и от класса **IMasterSlave**. Последний позволяет изменять направления действия сигналов в зависимости от амплуа компонента, который будет использовать наш интерфейсный класс: «мастер» (ведущий) или «слэйв» (ведомый).

Так как наш интерфейс по большей части будет выступать в амплуа «мастер», то мы переопределим абстрактный метод **asMaster()**, он будет использовать сервисные функции **out()** и **inout(),** унаследованные от класса **Bundle,** для изменения направленности сигналов.

Для того, чтобы сделать наш интерфейс чуть более гибким, нам необходимо добавить к нему пару параметров, а именно — параметры, задающие размерность шины адреса и шины данных. Сделаем это с помощью простой структуры **SramLayout**:

```
case class SramLayout(addressWidth: Int, dataWidth : Int){
  def bytePerWord = dataWidth/8
```

```
  def capacity = BigInt(1) << addressWidth
}
```

Тогда описание нашего интерфейсного класса примет следующий вид:

```
case class SramInterface(g : SramLayout) extends Bundle with IMasterSlave{
  val addr = inout(Analog(Bits(g.addressWidth bits)))
  val dat = inout(Analog(Bits(g.dataWidth bits)))
  val cs  = Bool
  val we  = Bool
  val oe  = Bool
  val ble  = Bool
  val bhe  = Bool

  override def asMaster(): Unit = {
    out(cs,we,oe,ble,bhe)
    inout(dat, addr)
  }
}
```

Поместим все это дело в файл **./src/main/scala/mylib/Sram.scala**, то есть в тот же каталог, где располагается ранее созданный компонент MachineTimer. В заголовке файла также укажем имя пакета и перечень используемых библиотек:

```
package mylib

import spinal.core._
import spinal.lib._
import spinal.lib.bus.simple._
import spinal.lib.io.TriState
```

Теперь можно описать комплексный сигнал **sram** на входе компонента **Murax**, представляющего синтезируемый СнК. Отредактируем файл **Murax.scala** следующим образом:

Во-первых, подключим наши новые классы, добавив в заголовок следующую строку:

```
import mylib.Apb3MachineTimer
import mylib.{SramInterface,SramLayout,PipelinedMemoryBusSram}
```

Во-вторых, найдем описание внешних сигналов компонента **Murax** и добавим новый сигнал **sram**, сразу указав параметры шин адреса и данных:

```
case class Murax(config : MuraxConfig) extends Component{
  import config._

  val io = new Bundle {
    //Clocks / reset
    val asyncReset = in Bool()
    val mainClk = in Bool()

    //Main components IO
    val jtag = slave(Jtag())

    ...

    val sram = master(SramInterface(SramLayout(addressWidth = 18, dataWidth = 16)))
  }
```

С интерфейсом разобрались. Теперь нужно реализовать компонент «контроллера» микросхемы SRAM, который, с одной стороны, будет выступать «мастером» и взаимодействовать с микросхемой памяти через описанный выше интерфейс, а с другой — выступать в роли «слэйв» устройства на шине **MainBus**. Напомню, что шина **MainBus** является в СнК Murax основой для интерконнекта с его частями и определяется классом **PipelinedMemoryBus**. Тогда код «заготовки» нашего контроллера, назовем его **PipelinedMemoryBusSram**, будет выглядеть следующим образом:

```
case class PipelinedMemoryBusSram(pipelinedMemoryBusConfig: PipelinedMemoryBusConfig,
                                  sramLayout : SramLayout) extends Component{
  val io = new Bundle{
    val bus = slave(PipelinedMemoryBus(pipelinedMemoryBusConfig))
    val sram = master(SramInterface(sramLayout))
  }
  ...
  // здесь будет имплементация контроллера SRAM
}
```

Настало время подумать о логике работы нашего контроллера микросхемы SRAM.

Для тех, кто не в курсе особенностей, микросхема статической памяти (Static Random Access Memory) работает асинхронно, то есть не имеет тактирования — чтение и запись в ячейки осуществляется после подачи на микросхему набора разрешающих сигналов (#CS — chip-select, #WE — write-enable) и занимает какое-то время. Это время, требуемое микросхеме на выполнения операции, обозначается в спецификации как «Read Cycle Time» ($\mathbf{t_{rc}}$)и «Write Cycle Time» ($\mathbf{t_{wc}}$). Заглянув в спецификацию на микросхему K6R4016V1D мы увидим, что оба этих параметра имеют одинаковое значение — от **10нс**, а это означает, что данная микросхема способна выполнять почти **100 млн** операций записи и чтения в секунду. Микросхема оперирует 16-ти битными словами, т. е. за одну операцию записывает или считывает по 16 бит данных, которые подаются или снимаются с шины данных (сигналы D0-D15), а адрес слова, которое будет записано или считано, предварительно подается на шину адреса (сигналы A0-A17).

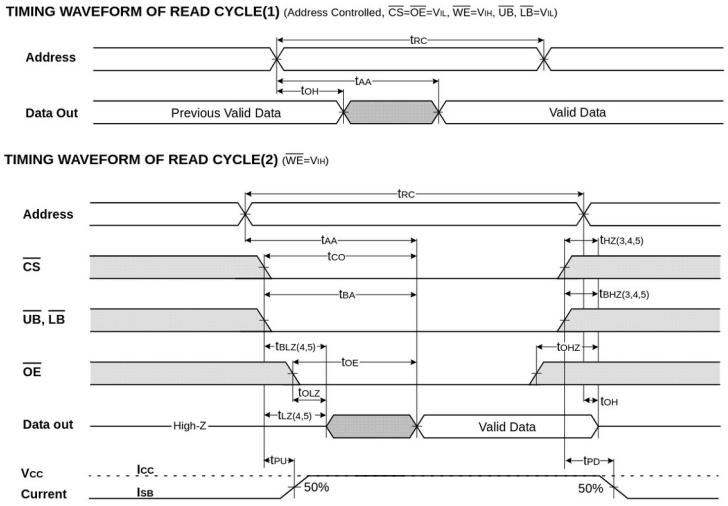


Рис. 28. Временные диаграммы для цикла чтения микросхемы SRAM K6R4016V1D.

Если посмотреть на временные диаграммы на рис. 28, то можно увидеть, что адрес подается с небольшим опережением сигналов разрешения, но если длительность удержания сигналов достаточно большая, то адрес можно подавать вместе с разрешающими сигналами. В нашем дизайне частота тактового сигнала составляет 75 МГц или иными словами, тактовый сигнал имеет длительность **13нс**. Этого времени, теоретически, должно быть достаточно, чтобы осуществлять операции чтения и записи в микросхему SRAM за один такт, подавая все необходимые сигналы сразу. То есть за один системный такт мы могли бы читать или записывать по 16 бит данных в SRAM. Однако, опыт показал, что на частоте 60 МГц и выше, при чтении возникают множественный битфлипы (искажения данных), при этом запись на частоте 75 МГц производится без ошибок. Я не стал вдаваться в подробности почему так происходит, но пришел к выводу, что для надежного чтения каждого 16-битного слова в реальности потребуется два такта.

Вычислительное ядро VexRiscv является 32-х битным, а это означает, что ядро за один такт считывает или записывает в шину **PipelinedMemoryBus** слова размерностью 32 бита и тут возникает дополнительная сложность. Чтобы выполнять операции чтения/записи с данной микросхемой SRAM, нам потребуется разбивать операцию записи на две части (два такта), а операцию чтения — на четыре. При чтении придется «склеивать» полученный результат в 32-битное слово перед выдачей его в шину.

Еще один момент состоит в том, что вычислительное ядро может потребовать записать не все 32 бита, а только часть из них, например младшие 8 бит (младший байт) или старшие 8 бит любого полуслова, или комбинацию из них. Для того, чтобы наш контроллер SRAM понял, какую именно часть данных требуется записать, шина **PipelinedMemoryBus** передает набор из четырех сигналов-масок **io.bus.cmd.mask[3:0]** — наличие лог «1» в соответствующем бите маске сигнализирует о том, что соответствующий байт (один из четырех байтов 32-х битного слова) требуется записать, а остальные оставить без изменения.

Таким образом, в реализации нашего контроллера SRAM вырисовывается автомат состояний, работающий по следующему алгоритму:

При записи в SRAM (**io.bus.cmd.write === True)**:

1. Автомат начинает работу в состоянии **#0**, в котором он ожидает сигнала готовности **io.bus.cmd.valid** от шины.
2. Получив сигнал готовности, автомат устанавливает управляющие сигналы **io.sram.oe = False** и **io.sram.we = False**; параллельно с этим выставляет на шину адреса микросхемы **io.sram.addr** адрес младшего 16-ти битного слова, полученного с внешней шины **io.bus.cmd.address**; на шину данных микросхемы **io.sram.dat** - младшие 16 бит данных; на линии **io.sram.ble** и **io.sram.bhe** подает значения масок из **io.bus.cmd.mask(0)** и **io.bus.cmd.mask(1)** соответственно; и переводит автомат в состояние **#1**.
3. В состоянии **#1** автомат выставляет на шину адреса **io.sram.addr** адрес старшего 16-ти битного слова; на шину данных **io.sram.dat** - старшие 16 бит данных; на линии **io.sram.ble** и **io.sram.bhe** подает значения масок **io.bus.cmd.mask(2)** и **io.bus.cmd.mask(3)** соответственно. В этом же состоянии автомат переводит своё состояние в **#0** и сигнализирует о завершении выполнения операции установкой сигнала **io.bus.cmd.ready** в **True**.

Цикл записи требует всего два такта (два состояния).

При чтении из SRAM (**io.bus.cmd.write === False)** все становится немного сложнее:

1. Автомат так же начинает работу в состоянии #**0**, в котором он ожидает сигнала готовности **io.bus.cmd.valid** от шины.
2. Получив сигнал готовности, автомат устанавливает управляющие сигналы **io.sram.oe = False**, **io.sram.we = True** и выставляет на шину адреса **io.sram.addr** адрес младшего 16-ти битного слова, полученного с шины **io.bus.cmd.address**, а данные с шины **io.sram.dat** копирует во внутренний (временный) регистр **rsp_data** в его младшие 16 бит, и переходит в состояние **#1**.

3. В состоянии **#1** автомат продолжает удерживать на шине **io.sram.addr** адрес младшего 16-ти битного слова и продолжает копировать данные во внутренний регистр, т. е. выполняет пустой цикл и переходит в состояние **#2**.
4. В состоянии **#2** автомат выставляет на шину адреса **io.sram.addr** адрес старшего 16-ти битного слова, полученного с шины **io.bus.cmd.address**, а данные с шины **io.sram.dat** копирует во внутренний регистр **rsp_data**, в его старшие 16 бит и переходит в состояние **#3**.
5. В состоянии **#3** автомат продолжает удерживать на шине **io.sram.addr** адрес старшего 16-ти битного слова, продолжает копировать данные во внутренний регистр. При этом автомат в этом же цикле формируется сигнал завершения операции **io.bus.cmd.ready** в **True**, а также устанавливает сигнал готовности данных **io.bus.rsp.valid**.

   Таким образом, цикл чтения требует четыре такта (четыре состояния автомата).

Здесь требуется сделать еще пару замечаний по работе алгоритма:

1. Выходной сигнал **io.bus.rsp.valid** должен буферизироваться через регистр (в нашем случае это будет регистр **rsp_valid).** Это требуется для того, чтобы разнести во времени сигналы **io.bus.cmd.ready** и **io.bus.rsp.valid** из-за особенностей реализации шины **PipelinedMemoryBus**. Если этого не сделать, то получив оба сигнала в одном такте шина остановится (перейдет в состояние STALL) и система прекратит работу.
2. Выходные данные при цикле чтении должны браться из временного регистра **rsp_data** и передаваться на шину **io.bus.rsp.data**.
3. При установке управляющих сигналов для микросхемы SRAM нужно учитывать, что все управляющие сигналы имеют инверсное значение (active low), т. е. эти сигналы требуется инвертировать.

Переведя описанный выше алгоритм на язык SpinalHDL, получим следующий код для контроллера SRAM, описываемого классом **PipelinedMemoryBusSram**:

```
case class PipelinedMemoryBusSram(pipelinedMemoryBusConfig: PipelinedMemoryBusConfig,
                                  sramLayout : SramLayout) extends Component{
  val io = new Bundle{
    val bus = slave(PipelinedMemoryBus(pipelinedMemoryBusConfig))
    val sram = master(SramInterface(sramLayout))
  }

  val state = Reg(UInt(3 bits)) init(0)
  val rsp_data = Reg(Bits(pipelinedMemoryBusConfig.addressWidth bits)) init(0)
  val rsp_valid = Reg(Bool()) init(False)

  io.sram.cs := ~io.bus.cmd.valid
  io.sram.we := True
  io.sram.oe := True
  io.sram.bhe := True
  io.sram.ble := True

  io.bus.rsp.data := rsp_data;
  io.bus.rsp.valid := rsp_valid
  io.bus.cmd.ready := (state === 1 && io.bus.cmd.write) || (state === 3 && !
io.bus.cmd.write)

  when (io.bus.cmd.valid) {
    when (io.bus.cmd.write) { // Write
      io.sram.we := False // active low
      when (state === 0) { // Write low 16 bits
        io.sram.addr := io.bus.cmd.address(sramLayout.addressWidth downto 2).asBits ## B"0"
        io.sram.dat := io.bus.cmd.data(15 downto 0).asBits
        io.sram.ble := ~io.bus.cmd.mask(0)
        io.sram.bhe := ~io.bus.cmd.mask(1)
        state := 1
```

```
      } otherwise { // Write high 16 bits
        io.sram.addr := io.bus.cmd.address(sramLayout.addressWidth downto 2).asBits ## B"1"
        io.sram.dat := io.bus.cmd.data(31 downto 16).asBits
        io.sram.ble := ~io.bus.cmd.mask(2)
        io.sram.bhe := ~io.bus.cmd.mask(3)
        state := 0
      }
    } otherwise { // Read
      io.sram.ble := False
      io.sram.bhe := False
      io.sram.oe := False
      when (state === 0) {
        io.sram.addr := io.bus.cmd.address(sramLayout.addressWidth downto 2).asBits ## B"0"
        rsp_data(15 downto 0) := io.sram.dat // buffer low 16 bits - first time
        state := 1
        rsp_valid := False
      } elsewhen (state === 1) {
        io.sram.addr := io.bus.cmd.address(sramLayout.addressWidth downto 2).asBits ## B"0"
        rsp_data(15 downto 0) := io.sram.dat // buffer low 16 bits - second time
        state := 2
      } elsewhen (state === 2) {
        io.sram.addr := io.bus.cmd.address(sramLayout.addressWidth downto 2).asBits ## B"1"
        rsp_data(31 downto 16) := io.sram.dat // buffer high 16 bits - first time
        state := 3
      } otherwise {
        io.sram.addr := io.bus.cmd.address(sramLayout.addressWidth downto 2).asBits ## B"1"
        rsp_data(31 downto 16) := io.sram.dat // buffer high 16 bits - second time
        rsp_valid := True // Signal data is READY
        state := 0
      }
    }
  } otherwise { // not Valid
    state := 0
    rsp_valid := False
  }
}
```

Поместим этот код в файл **Sram.scala** и попробуем запустить сборку командой **make clean && make**. Если никаких синтаксических ошибок не возникло, то на стадии генерации кода Verilog (т. е. при исполнении байт-кода на JVM), мы должны получить длинный список из приведенных ниже сообщений:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ make clean && make
...
[info] [Progress] at 2.396 : Checks and transforms
...
[error] Exception in thread "main" spinal.core.SpinalExit:
[error]  Error detected in phase PhaseCheck_noLatchNoOverride
[error] ********************************************************************************
[error] ********************************************************************************
[error] NO DRIVER ON (toplevel/io_sram_cs : out Bool), defined at
[error]     mylib.SramInterface.<init>(Sram.scala:16)
[error]     vexriscv.demo.Murax$$anon$1.<init>(Murax.scala:176)
[error]     vexriscv.demo.Murax.<init>(Murax.scala:162)
[error]     vexriscv.demo.Murax_karnix$$anonfun$main$8.apply(Murax.scala:556)
[error]     vexriscv.demo.Murax_karnix$$anonfun$main$8.apply(Murax.scala:556)
[error]     spinal.sim.JvmThread.run(SimManager.scala:51)
[error] ********************************************************************************
[error] ********************************************************************************
[error] NO DRIVER ON (toplevel/io_sram_we : out Bool), defined at
[error]     mylib.SramInterface.<init>(Sram.scala:17)
[error]     vexriscv.demo.Murax$$anon$1.<init>(Murax.scala:176)
[error]     vexriscv.demo.Murax.<init>(Murax.scala:162)
[error]     vexriscv.demo.Murax_karnix$$anonfun$main$8.apply(Murax.scala:556)
[error]     vexriscv.demo.Murax_karnix$$anonfun$main$8.apply(Murax.scala:556)
[error]     spinal.sim.JvmThread.run(SimManager.scala:51)
[error] ********************************************************************************
...
[error] Nonzero exit code returned from runner: 1
[error] (Compile / runMain) Nonzero exit code returned from runner: 1
[error] Total time: 32 s, completed Feb 14, 2024, 11:47:45 PM
```

Сообщение об ошибке вида «**NO DRIVER ON**» говорит о том, что соответствующий сигнал объявлен и используется, но для него не назначен источник (нет «драйвера»), то есть нет схемы, формирующей этот сигнал. В нашем случае это легко объясняется тем, что мы описали интерфейсные сигналы для микросхемы SRAM, подключили сигнальные линии к самой микросхеме, но не подключили их к контроллеру **PipelinedMemoryBusSram**, да и сам контроллер не подключили к шине **MainBus**.

Ну что же, исправим это недоразумение — отредактируем файл **Murax.scala**, найдем в нём участок кода, в котором производится подключение «слэйв» устройств к шине **MainBus** путем добавления их в ассоциативный массив **mainBusMapping**:

```
//****** MainBus slaves ********
val mainBusMapping = ArrayBuffer[(PipelinedMemoryBus,SizeMapping)]()

val ram = new MuraxPipelinedMemoryBusRam(
  onChipRamSize = onChipRamSize,
  onChipRamHexFile = onChipRamHexFile,
  pipelinedMemoryBusConfig = pipelinedMemoryBusConfig,
  bigEndian = bigEndianDBus
)
mainBusMapping += ram.io.bus -> (0x80000000l, onChipRamSize)

...
```

И добавим сюда наш контроллер микросхемы SRAM:

```
val sramCtrl = new PipelinedMemoryBusSram(
  pipelinedMemoryBusConfig = pipelinedMemoryBusConfig,
  sramLayout = SramLayout(addressWidth = 18, dataWidth = 16)
)
sramCtrl.io.sram <> io.sram
mainBusMapping += sramCtrl.io.bus -> (0x90000000l, 512 kB)
```

При добавлении нового компонента декодеру шины сообщается, что контроллеру SRAM будет доступно адресном пространстве **512KB @ 0x90000000**. Напомню, что адресное пространство «бортовой» RAM находится в диапазоне **96KB @ 0x80000000**.

Запустим еще раз сборку и убедимся, что всё прошло гладко и без ошибок:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ make

(cd ../../..; sbt "runMain vexriscv.demo.Murax_karnix")
[info] welcome to sbt 1.6.0 (Ubuntu Java 11.0.9.1)
…
[info] running (fork) vexriscv.demo.Murax_karnix
[info] [Runtime] SpinalHDL v1.10.1    git head : 2527c7c6b0fb0f95e5e1a5722a0be732b364ce43
[info] [Runtime] JVM max memory : 8294.0MiB
[info] [Runtime] Current date : 2024.02.15 00:24:16
[info] [Progress] at 0.000 : Elaborate components
[info] [Warning] This VexRiscv configuration is set without software ebreak instruction
support. Some software may rely on it (ex: Rust). (This isn't related to JTAG ebreak)
[info] [Warning] This VexRiscv configuration is set without illegal instruction catch
support. Some software may rely on it (ex: Rust)
[info] [Progress] at 2.281 : Checks and transforms
[info] [Progress] at 3.035 : Generate Verilog
…
Info: Logic utilisation before packing:
Info:     Total LUT4s:      2637/24288    10%
Info:         logic LUTs:   2149/24288     8%
Info:         carry LUTs:    272/24288     1%
Info:           RAM LUTs:    144/ 3036     4%
Info:          RAMW LUTs:     72/ 6072     1%

Info:      Total DFFs:      1428/24288     5%
…
Info: Packing constants..
Info: Packing carries...
Info: Packing LUTs...
```

```
Info: Packing LUT5-7s...
Info: Packing FFs...
Info:     675 FFs paired with LUTs.
Info: Generating derived timing constraints...
Info:     Input frequency of PLL 'i_pll.pll_i' is constrained to 25.0 MHz
Info:     Derived frequency constraint of 75.0 MHz for net clk75
Info: Promoting globals...
Info:     promoting clock net clk75 to global network
Info:     promoting clock net io_core_jtag_tck$TRELLIS_IO_IN to global network
Info: Checksum: 0x8aa53f27

Info: Device utilisation:
Info:             TRELLIS_IO:    54/  197    27%
Info:                   DCCA:     2/   56     3%
Info:                 DP16KD:    48/   56    85%
Info:             MULT18X18D:     0/   28     0%
Info:                 ALU54B:     0/   14     0%
Info:                EHXPLLL:     1/    2    50%
...
Info:            TRELLIS_FF:  1428/24288     5%
Info:          TRELLIS_COMB:  2749/24288    11%
Info:          TRELLIS_RAMW:    36/ 3036     1%
…
Info: Max frequency for clock                            '$glbnet$clk75': 80.34 MHz (PASS at
75.00 MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 130.21 MHz (PASS at
12.00 MHz)
…
2 warnings, 0 errors

Info: Program finished normally.
ecppack --compress --freq 38.8 --input murax_hello_world_out.config --bit
murax_hello_world.bit
```

Главное, на что следует обращать во всей этой статистике то, что наш дизайн всё еще проходит по STA и максимально допустимая тактовая частота составляет 80,34 МГц, т. е. имеется некоторый запас «прочности».

17.2.2 Тестируем память и контроллер SRAM

Перед тем, как задействовать полученную область памяти в программе, нам потребуется некая функция, которая проведет тест всего пространства SRAM памяти объемом 512 КБ и выяснит, рабочая у нас SRAM или нет — вполне возможна такая ситуация, при которой микросхема SRAM перестанет справляться с задачей, т. е. попросту не будет успевать выполнять операции чтения/записи. Такая ситуация может возникнуть, если продолжить увеличивать тактовую частоту системного домена.

Строго говоря, организовать качественное тестирование памяти — это достаточно сложная задача, но мы не будем заниматься перфекционизмом, а воспользуемся простым алгоритмом — будем записывать псевдослучайную последовательность в ячейки памяти последовательно до конца области, после чего проделаем цикл чтения и сравнения читаемых данных с этой же псевдослучайной последовательностью. Ниже приведен вариант такой функции тестирования:

```
#define SRAM_SIZE       (512*1024)
#define SRAM_ADDR_BEGIN 0x90000000
#define SRAM_ADDR_END   (0x90000000 + SRAM_SIZE)

// ...

int sram_test_write_random_ints(void) {
        volatile unsigned int *mem;
        unsigned int fill;
        int fails = 0;

        fill = 0xdeadbeef; // random seed
        mem = (unsigned int*) SRAM_ADDR_BEGIN;
```

```
        print("Filling SRAM at: ");
        printhex((unsigned int)mem);

        while((unsigned int)mem < SRAM_ADDR_END) {
                *mem++ = fill;
                fill += 0xdeadbeef; // generate pseudo-random data
        }

        fill = 0xdeadbeef; // random seed
        mem = (unsigned int*) SRAM_ADDR_BEGIN;

        print("Checking SRAM at: ");
        printhex((unsigned int)mem);

        while((unsigned int)mem < SRAM_ADDR_END) {
                if(*mem != fill) {
                        print("SRAM check failed at: ");
                        printhex((unsigned int)mem);
                        print("expected: ");
                        printhex((unsigned int)fill);
                        print("got: ");
                        printhex((unsigned int)*mem);
                        fails++;
                }
                mem++;
                fill += 0xdeadbeef; // generate pseudo-random data
        }

        if((unsigned int)mem == SRAM_ADDR_END)
                print("SRAM total fails: ");
                printhex((unsigned int)fails);

        return fails++;
}
```

В приведенном выше коде на языке Си, мы для удобства объявляем набор констант для описания области памяти — её начальный адрес, конечный адрес и размер. В функции **sram_test_write_random_ints(),** мы сначала циклом заполняем тестируемую область 32-х битными числами, которые вычисляются по известной нам формуле — первоначальное число **0xDEADBEEF** каждую итерацию цикла прибавляется к накопленной сумме в переменной **fill**, формируя псевдослучайное число. Затем производится чтение данных из этой же области и сравнение считанного числа с рассчитанным значением. Функция возвращает количество выявленных несовпадений, т. е. ошибок записи или чтения.

Добавим вызов функции **sram_test_write_random_ints()** в начало функции **main()** программы «hello_world», сразу после процедуры инициализации UART:

```
void main() {
        Uart_Config uart_config;

        uart_config.dataLength = UART_DATA_8;
        uart_config.parity = UART_PARITY_NONE;
        uart_config.stop = UART_STOP_ONE;

        uint32_t rxSamplePerBit = UART_PRE_SAMPLING_SIZE + UART_SAMPLING_SIZE +
UART_POST_SAMPLING_SIZE;

        uart_config.clockDivider = SYSTEM_CLOCK_HZ / UART_BAUD_RATE / rxSamplePerBit - 1;
        uart_applyConfig(UART, &uart_config);

        sram_test_write_random_ints();

       ...
```

Выполняем сборку всего проекта командой **make clean && make**. После устранения ошибок и получения битстрима, загружаем его в плату «Карно» командой **make upload**, запускаем эмулятор терминала **minicom** и наблюдаем за тем, как отрабатывает тест SRAM памяти. Если всё в порядке, то в порт будут выводиться следующие сообщения:

```
Filling SRAM at: 90000000
Checking SRAM at: 90000000
SRAM total fails: 00000000
Hello world, this is VexRiscv!
000000C6
000282CD
Hello world, this is VexRiscv!
000000C6
000282CE
Hello world, this is VexRiscv!
000000C6
000282CD
```

Если же микросхема памяти сбоит или отсутствует, мы будем наблюдать сообщения вида:

```
Filling SRAM at: 90000000
SRAM check failed at: 90000000
expected: DEADBEEF
got: BEEF0000
SRAM check failed at: 90000004
expected: BD5B7DDE
got: 7DDE0000
SRAM check failed at: 90000008
expected: 9C093CCD
got: 3CCD0000
```

17.2.3 Используем SRAM с функцией malloc

Осталось задействовать добытый кусок памяти в Си программах. Самый простой способ сделать это - просто читать и писать в область адресного пространства начиная с **0x90000000**, используя прямой указатель. Более сложный вариант — задействовать этот блок памяти для «кучи» («heap»), то есть позволить библиотечной функции **malloc()** распределять блоки в этой области памяти. Давайте разберемся как это реализуется.

Функция **malloc()** входит в состав стандартной библиотеки ввода/вывода, которую принято называть **libc**, а её определение традиционно находится в заголовочном файле **stdlib.h**. Давайте вставим в код программы «hello_world» вызов этой функции и посмотрим, что у нас получится. Отредактируем файл **main.c**, включим в него заголовочный файл **stdlib.h**, вызовем функцию **malloc()** из тела функции **main()** и распечатаем возвращаемый результат:

```
#include <stdint.h>
#include <stdlib.h>
...

void main() {
        ...
        sram_test_write_random_ints();

        char *test = malloc(1024);
        println("Malloc test:");
        printhex((unsigned int)test);

        ...

}
```

Помимо этого, нам нужно объяснить компилятору, чтобы на стадии линковки программы «hello_world» он подключал библиотеку **libc_nano.a** — это упрощенная версия библиотеки **libc**, предназначенная для вычислительных систем с небольшим объёмом ОЗУ. Для этого в файле сборки **makefile** добавим следующую строку:

```
rz@devbox:~/VexRiscv/src/main/c/murax/hello_world$ cat makefile

...
INC  =
LIBS = -lc_nano
```

Запустив сборку программы командой **make**, мы увидим следующее сообщение об ошибке:

```
/opt/riscv64-unknown-elf-gcc-8.3.0-2019.08.0-x86_64-linux-ubuntu14/bin/../lib/gcc/riscv64-
unknown-elf/8.3.0/../../../../riscv64-unknown-elf/bin/ld: /opt/riscv64-unknown-elf-gcc-
8.3.0-2019.08.0-x86_64-linux-ubuntu14/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/../../../../
riscv64-unknown-elf/lib/rv32i/ilp32/libc.a(lib_a-sbrkr.o): in function `.L0 ':
sbrkr.c:(.text+0x18): undefined reference to `_sbrk'
```

Оказывается, что для работы функции **malloc()** требуется некоторая функция с именем **_sbrk()**, которой у нас нет. Что это за функция и где её взять? Быстрый ответ на это вопрос — написать самостоятельно.

Функция **malloc()** получает на вход один параметр — размер блока памяти который нужно выделить программе. Она выполняет эффективный поиск на «куче» незадействованного блока памяти требуемого размера, используя двоичное дерево поиска (или иной алгоритм — имеется много реализаций malloc). В результате она возвращает указатель на выделенный блок памяти или NULL, если выделить заданный блок не удалось. Функция **malloc()** выделяет запрашиваемые блоки из области, которую приято называть «кучей» или «heap» — это специально зарезервированное редактором объектных связей (линкером) адресное пространство.

Как правило, «куча» располагается в верхней части адресного пространства программы, после всех других областей, а за «кучей» следует свободная, неиспользуемая память. Делается это для того, чтобы «кучу» можно было динамически расширять по мере надобности, постепенно отодвигая условную линию разделения областей (сегментов) «кучи» и свободной памяти. Чтобы контролировать это перемещение (рост «кучи»), функция **malloc()** вызывает функцию **_sbrk()** (производное от «segment break») каждый раз, когда в «куче» не хватает места для выполнения очередного запроса на распределение блока памяти. Фактически, все, что делает функция **_sbrk()**, это запоминает во внутреннем указателе адрес ячейки памяти, где находится граница областей; проверяет, можно ли её расширить (сдвинуть на заданное количество байт); если можно, то запоминает новое значение и возвращает его в функцию **malloc()**. В современных операционных системах с виртуальной памятью функция **_sbrk()** реализуется не простым механизмом описанным выше, а через системный вызов **mmap()**. В старых UNIX истемах попытка программы осуществить доступ за пределы используемой области памяти отслеживалось операционной системой и автоматически рассматривалось как необходимость выделить программе (процессу) дополнительной памяти к уже имеющейся. Если в системе нет свободной памяти для процесса, то такой процесс снимается с исполнения. Но нас это всё не касается, так как в данном случае мы работаем с «голым железом».

Так как у нас нет виртуальной памяти и нет операционной системы, то мы можем реализовать функцию **_sbrk()** тем способом, который нам удобен, а именно — заведем указатель на начало свободной памяти и будем его постепенно передвигать на запрашиваемое количество байт. Как только указатель перейдет за пределы доступной физической памяти, наша функция **_sbrk()** вернет NULL, сигнализируя таким образом функции **malloc()** о том, что вся память израсходована. Осталось выяснить, где мы можем расположить «кучу» и как это сделать.

Помимо «кучи», линкер также резервирует адресное пространство для стека, для изменяемых и не изменяемых данных программы, а также для самого текста (кода) программы — всё это называется структурой размещения программы в памяти. Данная структура определяется настройками линковщика, которые, в нашем случае для программы «hello_world» находятся в файл **linker.ld** и используются при сборки программы.

Заглянув в файл **linker.ld**, мы увидим, что:

1. Линкер будет производить размещение всех областей программы в адресном пространстве начиная с 0x80000000 и размером 96КБ.
2. Линкер по умолчанию, если не определена переменная **_stack_size**, выделяет под стек 2048 байт.
3. Линкер по умолчанию, если не определена переменная **_heap_size**, выделяет под «кучу» 0 байт, т. е. «кучи» нет.
4. Линкер будет производить размещение областей в следующей, не очень удобной последовательности (от младших адресов к старшим):
   - ._vector — таблица векторов, состоящая из одного JUMP;
   - .text(crt.o) — текст «C run-time»;
   - ._user_heap — «куча» размером 0 байт по умолчанию;
   - ._stack — стек программы размером 2048 байт;
   - .data — область неинициализированных данных программы;
   - .rodate — область инициализированных неизменяемых данных программы;
   - .text — текст программы «hello_world».

Неудобство такой структуры размещения состоит в том, что «куча» здесь окружена другими областями и не может быть безопасно расширена путем перемещения указателя, не повредив стек, область которого следует сразу за «кучей». Поэтому, перед тем как имплементировать функцию **_sbrk()**, мы немного переработаем структуру размещения, а именно — передвинем «кучу» в самый конец адресного пространства и присвоим ей название **_ram_heap**. Это будет область для «кучи» в пределах синтезируемой RAM. Размер этой области мы оставим равным нулю, функция **_sbrk()** расширит её при первом же вызове на требуемое количество байт.

Наш новый файл **linker.ld** будет выглядеть следующим образом:

```
rz@devbox:~/VexRiscv/src/main/c/murax/hello_world$ cat src/linker.ld
OUTPUT_FORMAT("elf32-littleriscv", "elf32-littleriscv", "elf32-littleriscv")
OUTPUT_ARCH(riscv)
ENTRY(crtStart)

MEMORY {
  RAM      (rwx): ORIGIN = 0x80000000, LENGTH = 96k
}

_stack_size = DEFINED(_stack_size) ? _stack_size : 2048;
_ram_heap_size = DEFINED(_ram_heap_size) ? _ram_heap_size : 0;

SECTIONS {

  ._vector ORIGIN(RAM): {
    *crt.o(.start_jump);
    *crt.o(.text);
  } > RAM

  .data :
  {
    *(.rdata)
    *(.rodata .rodata.*)
    *(.gnu.linkonce.r.*)
    *(.data .data.*)
    *(.gnu.linkonce.d.*)
    . = ALIGN(8);
    PROVIDE( __global_pointer$ = . + 0x800 );
    *(.sdata .sdata.*)
    *(.gnu.linkonce.s.*)
    . = ALIGN(8);
    *(.srodata.cst16)
    *(.srodata.cst8)
    *(.srodata.cst4)
```

```
    *(.srodata.cst2)
    *(.srodata .srodata.*)
  } > RAM

  .bss (NOLOAD) : {
               . = ALIGN(4);
               /* This is used by the startup in order to initialize the .bss secion */
               _bss_start = .;
    *(.sbss*)
    *(.gnu.linkonce.sb.*)
    *(.bss .bss.*)
    *(.gnu.linkonce.b.*)
    *(COMMON)
               . = ALIGN(4);
               _bss_end = .;
  } > RAM

  .rodata         :
  {
    *(.rdata)
    *(.rodata .rodata.*)
    *(.gnu.linkonce.r.*)
  } > RAM

  .noinit (NOLOAD) : {
      . = ALIGN(4);
     *(.noinit .noinit.*)
      . = ALIGN(4);
  } > RAM

  .memory : {
    *(.text);
    end = .;
  } > RAM

  .ctors          :
  {
    . = ALIGN(4);
    _ctors_start = .;
    KEEP(*(.init_array*))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
    . = ALIGN(4);
    _ctors_end = .;
    PROVIDE ( END_OF_SW_IMAGE = . );
  } > RAM

  ._stack (NOLOAD):
  {
    . = ALIGN(16);
    PROVIDE (_stack_end = .);
    . = . + _stack_size;
    . = ALIGN(16);
    PROVIDE (_stack_start = .);
  } > RAM

  ._ram_heap (NOLOAD):
  {
    . = ALIGN(8);
    PROVIDE ( end = . );
    PROVIDE ( _end = . );
    PROVIDE ( _ram_heap_start = .);
    . = . + _ram_heap_size;
    PROVIDE ( _ram_heap_end = ALIGN(ORIGIN(RAM) + LENGTH(RAM) ,8) );
  } > RAM
}
```

Настало время имплементировать функцию **_sbrk()**. Напомню, что основной нашей целью было сделать так, чтобы функция **malloc()** распределяла блоки памяти из внешней SRAM памяти — области, которая располагается с адреса **0x90000000**. В то же время, у нас имеется неиспользуемая область в синтезируемой RAM памяти, где-то там выше стека. Было бы не плохо, если бы наша программа могла проверить доступность внешней SRAM и если

она работает без ошибок, то задействовать её под «кучу». Если же SRAM выдает ошибки при тестировании (либо отсутствует), то для «кучи» следует использовать свободную область из синтезированной RAM. Иными словами, нам требуется функция инициализации указателя границы «кучи» для **_sbrk()**, которую мы будем вызывать с параметром в зависимости от результата тестирования SRAM.

Чтобы ссылаться на области, которые определены в файле **linker.ld**, в файле **main.c** заведем следующие переменны:

```
// Below is some linker specific stuff
extern unsigned int end; /* Set by linker.  */
extern unsigned int _ram_heap_start; /* Set by linker.  */
extern unsigned int _ram_heap_end; /* Set by linker.  */
extern unsigned int _stack_start; /* Set by linker.  */
extern unsigned int _stack_size; /* Set by linker.  */
```

Это переменные представляют собой адреса размещения областей в памяти, они соответствуют тому, что описано в файле **linker.ld**.

Далее, заведем несколько своих указателей, с которыми будет работать наша функция **_sbrk()**:

```
unsigned char* sbrk_heap_end = 0; /* tracks heap usage */
unsigned int* heap_start = 0; /* programmer define heap start */
unsigned int* heap_end = 0; /* programmer defined heap end */
```

Переменная **sbrk_heap_end** — это и будет наш указатель на границу раздела, то есть он всегда указывает на то место, где заканчивается «куча». Переменные **heap_start** и **heap_end** будут ограничивать пределы роста «кучи».

Теперь определим функцию **init_sbrk()** для инициализации этих переменных и тоже добавим её в файл **main.c**:

```
void init_sbrk(unsigned int* heap, int size) {

        if(heap == NULL) {
                heap_start = (unsigned int*)& _ram_heap_start;
                heap_end = (unsigned int*)& _ram_heap_end;
        } else {
                heap_start = heap;
                heap_end = heap_start + size;
        }

        sbrk_heap_end = (char*) heap_start;
}
```

Ну и добавим функцию **_sbrk()** следующего содержания:

```
void* _sbrk(unsigned int incr) {

        unsigned char* prev_heap_end;

        if (sbrk_heap_end == 0) {
                // In case init_sbrk() has not been called
                // use on-chip RAM by default
                heap_start = & _ram_heap_start;
                heap_end = & _ram_heap_end;
                sbrk_heap_end = (char*) heap_start;
        }

        prev_heap_end = sbrk_heap_end;

        if((unsigned int)(sbrk_heap_end + incr) >= (unsigned int)heap_end) {

                println("_sbrk() OUT OF MEM:");
                print("sbrk_heap_end = ");
                printhex((unsigned int)sbrk_heap_end);
```

```
                print("heap_end = ");
                printhex((unsigned int)heap_end);
                print("incr = ");
                printhex((unsigned int)incr);

                return ((void*)-1); // error - no more free memory
        }

        sbrk_heap_end += incr;

        return (void *) prev_heap_end;
}
```

Единственное, что делает функция **_sbrk()** — это перемещает (увеличивает) указатель **sbrk_heap_end** на значение **incr**, предварительно проверив пределы области памяти в которых «куче» дозволено быть.

Осталось вызвать функцию **init_sbrk()** из тела функции **main()**. Для этого добавим следующий код:

```
void main() {
        Uart_Config uart_config;

        uart_config.dataLength = UART_DATA_8;
        uart_config.parity = UART_PARITY_NONE;
        uart_config.stop = UART_STOP_ONE;

        uint32_t rxSamplePerBit = UART_PRE_SAMPLING_SIZE + UART_SAMPLING_SIZE +
UART_POST_SAMPLING_SIZE;

        uart_config.clockDivider = SYSTEM_CLOCK_HZ / UART_BAUD_RATE / rxSamplePerBit - 1;
        uart_applyConfig(UART, &uart_config);

        if(sram_test_write_random_ints() == 0) {
                init_sbrk((unsigned int*)SRAM_ADDR_BEGIN, SRAM_SIZE);
                println("Enabled heap on SRAM");
        } else {
                init_sbrk(NULL, 0);
                println("Enabled heap on on-chip RAM");
        }

        char *test = malloc(1024);
        println("Malloc test:");
        printhex((unsigned int)test);
...
```

Собираем проект и устраняем синтаксические ошибки. При сборке будет выполнена компиляция Си кода, и мы увидим следующие сообщения:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ make clean && make

...
(cd ../../../src/main/c/murax/hello_world/; make)
make[1]: Entering directory '/home/rz/VexRiscvWithKarnix/src/main/c/murax/hello_world'
mkdir -p build/src/
/opt/riscv//bin/riscv64-unknown-elf-gcc -c -march=rv32i  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing   -o build/src/main.o src/main.c
/opt/riscv//bin/riscv64-unknown-elf-gcc -S -march=rv32i  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing   -o build/src/main.o.disasm src/main.c
/opt/riscv//bin/riscv64-unknown-elf-gcc -march=rv32i  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing -o build/hello_world.elf build/src/main.o
build/src/crt.o -march=rv32i  -mabi=ilp32 -nostdlib -lgcc -mcmodel=medany -nostartfiles -
ffreestanding -Wl,-Bstatic,-T,./src/linker.ld,-Map,build/hello_world.map,--print-memory-
usage -Lbuild -lc
Memory region         Used Size  Region Size  %age Used
             RAM:        9760 B        96 KB      9.93%
/opt/riscv//bin/riscv64-unknown-elf-objcopy -O ihex build/hello_world.elf
build/hello_world.hex
/opt/riscv//bin/riscv64-unknown-elf-objdump -S -d build/hello_world.elf >
build/hello_world.asm
/opt/riscv//bin/riscv64-unknown-elf-objcopy -O verilog build/hello_world.elf
build/hello_world.v
```

```
make[1]: Leaving directory '/home/rz/VexRiscvWithKarnix/src/main/c/murax/hello_world'
(cd ../../..; sbt "runMain vexriscv.demo.Murax_karnix")
[info] welcome to sbt 1.6.0 (Ubuntu Java 11.0.9.1)
```

Размер бинарного кода Си программы заметно подрос: с 2756 до 9760 байт. Это место заняли функция **malloc()** и сопутствующие ей материалы из **libc**.

Загружаем получившийся битстрим в ПЛИС и наблюдаем в порту следующие сообщения:

```
Checking SRAM at: 90000000
SRAM total fails: 00000000 special keys
Enabled heap on SRAM
Malloc test:
90000008
Hello world, this is VexRiscv!
000000C6
000282CD
Hello world, this is VexRiscv!
000000C6
000282CE
```

Видно, что тест SRAM прошел успешно и «куча» инициализировалась в области SRAM, что подтверждается указателем на первый блок памяти, выделенный функцией malloc(): **0x90000008**.

С этого момента мы можем смело использовать **malloc()** и **free()**.

17.2.4 Добавляем блоки вычислителей Div и Mul

Сейчас, когда у нас имеется функционирующий **malloc()**, мы могли бы попытаться использовать все остальные функции из библиотеки **libc**, но спешу предупредить, что не все так просто. Дело в том, что **libc** очень сильно завязана на использование математических инструкций целочисленного умножения и деления (MUL и DIV), которые на данный момент отсутствуют в нашей конфигурации ядра VexRiscv, соответственно эти инструкции отключены в опциях компилятора и при попытке собрать Си программу, скажем, с использованием функции **vsnprintf()**, приведет к выводу большого списка ошибок об отсутствующих встроенных функций **__mulsi3()** и **__divsi3()**. Эти функции являются обертками для машинных инструкций умножения и деления.

Технически, мы можем попросить компилятор эмулировать умножение и деление программно, для этого в makefile-е в опциях линкеру (**LDFLAGS**) нужно убрать параметр **-nostdlib** и добавить **-lgcc**, это заставит его подключать набор функций целочисленной математики. Но, во-первых, это увеличит объём кода, во-вторых, производительность такого решения будет очень низкой. И в-третьих, зачем нам программная эмуляция целочисленного умножения и деления, если мы можем легко добавить эти инструкции в набор команд VexrRiscv, включив соответствующие плагины.

Всё что нам требуется, это в файле **Murax.scala** найти инициализацию массива с плагинами и добавить в него два плагина: **MulPlugin** и **DivPlugin**. Изменения в **Murax.scala** выглядят следующим образом:

```
new FullBarrelShifterPlugin,
new MulPlugin,
new DivPlugin,
new HazardSimplePlugin(
  bypassExecute = false,
```

```
      bypassMemory = false,
      bypassWriteBack = false,
      bypassWriteBackBuffer = false,
      pessimisticUseSrc = false,
      pessimisticWriteRegFile = false,
      pessimisticAddressMatch = false
    ),
```

Жирным шрифтом выделены две строки, которые требуется добавить.

Перед тем как мы запустим сборку, нам нужно разрешить компилятору использовать инструкции из расширения **M**, для этого нужно чтобы ему передавался параметр **-march=rv32im** (сейчас ему передается -march=rv32i — без буквы **m**). Сделать это можно отредактировав **makefile** для Си программы «hello_world», изменив переменную MULDIV=no на **MULDIV=yes**.

Собственно, на этом вся магия заканчивается. Для того, чтобы проверить работоспособность функций **libc**, добавим в текст Си программы в тело функции **main()** вывод строки приветствия, используя функцию **vsnprintf()**:

```
while(1){
        unsigned int shift_time;
        unsigned int t1, t2;

        //println("Hello world, this is VexRiscv!");
        char str[128];
        vsnprintf(str, 128, "Hello world, this is VexRiscv!\r\n", NULL);
        print(str);

        for(unsigned int i=0;i<nleds-1;i++){
        ...
```

Как обычно, командой **make clean && make** запустим сборку и посмотрим с какими параметрами происходит линковка:

```
...
/opt/riscv//bin/riscv64-unknown-elf-gcc -march=rv32im  -mabi=ilp32 -DNDEBUG -g -Os -MD -
fstrict-volatile-bitfields -fno-strict-aliasing -o build/hello_world.elf build/src/main.o
build/src/crt.o -march=rv32im  -mabi=ilp32 -nostdlib -lgcc -mcmodel=medany -nostartfiles -
ffreestanding -Wl,-Bstatic,-T,./src/linker.ld,-Map,build/hello_world.map,--print-memory-
usage -Lbuild -lc_nano
Memory region         Used Size  Region Size  %age Used
             RAM:        9152 B        96 KB      9.31%
...
```

Все ок, используется флаг `-march=rv32im`, линковка Си программы и сборка всего проекта завершается успешно. Загружаем битстрим в ПЛИС и убеждаемся, что сообщение приветствия также хорошо выводится в отладочный порт, как и в предыдущем примере.

17.2.5 Задействуем функцию printf

Самой полезной из всех, без сомнения, является функция **printfs()**, которая является оберткой для более универсальной функции **vsnprintf()**. С функцией **vsnprintf()** мы только что разобрались и для её работы более ничего не требуется. А вот для того, чтобы заработала функция **printf()**, ей необходимо обеспечить механизм для вывода строки на «стандартное устройство вывода». В операционных системах для этих целей используется системный вызов **write()** с файловым дескриптором **#1**, но у нас нет операционной системы. Как тут

быть? Всё просто. Нужно определить ряд оберточных функций, в том числе функцию для вывода строки в последовательный порт, а именно:

- **_write()** - вызывается библиотечными функциями, когда требуется вывести блок данных (строку) на стандартное устройство вывода. В нашем случае таким устройством будет служить отладочный порт UART, поэтому тело функции **_write()** определим как обертку у уже имеющейся у нас функции **uart_write()** которая отправляет символ в порт UART.
- **_close()** - закрывает файловый дескриптор. В нашем случае это будет пустая функция.
- **_lseek()** - позиционирует курсор внутри открытого файла. Тоже определим как пустую функцию.
- **_read()** - читает блок данных из стандартного ввода. На данный момент определим её как пустую, ничего не делающую функцию. Если нам потребуется использовать функции scanf() или getc(), то эту функцию придется написать.
- **_fstat()** - выдает информацию об открытом файле. Определим как пустую функцию.
- **_isatty()** - позволяет выяснить, является ли файл именем терминалом (возвращает 1 если файл является терминалом, иначе возвращает 0). Будем всегда возвращать 1 — пусть библиотека **libc** думает что имеет дело с терминалом.

Добавим реализацию описанных выше функций «заглушек» в файл **main.c**:

```
#include <stdio.h>
#include <sys/stat.h>

...

int _write (int fd, const void *buf, size_t count) {
        int i;
        char* p = (char*) buf;
        for(i = 0; i < count; i++) { uart_write(UART, *p++); }
        return count;
}

int _read (int fd, const void *buf, size_t count) { return 1; }
int _close(int fd) { return -1; }
int _lseek(int fd, int offset, int whence) { return 0 ;}
int _isatty(int fd) { return 1; }

int _fstat(int fd, struct stat *sb) {
        sb->st_mode = S_IFCHR;
        return 0;
}
```

В теле функции **main()** заменим вывод приветствия на использование функции **printf()**:

```
 while(1){
       ...
       //println("Hello world, this is VexRiscv!");
       //char str[128];
       //vsnprintf(str, 128, "Hello world, this is VexRiscv!\r\n", NULL);
       //print(str);
       printf("Hello world, this is VexRiscv!\r\n");
```

Собираем, прошиваем битстрим в ПЛИС и наслаждаемся работой одной из самых сложных функций стандартной библиотеки **libc** — функцией **printf()**.

Ну и традиционно, статистика после компиляции и линковки Си программы:

```
/opt/riscv//bin/riscv64-unknown-elf-gcc -march=rv32im  -mabi=ilp32 -DNDEBUG -g -Os -MD -fstrict-
volatile-bitfields -fno-strict-aliasing -o build/hello_world.elf build/src/main.o build/src/crt.o -
march=rv32im  -mabi=ilp32 -nostdlib -lgcc -mcmodel=medany -nostartfiles -ffreestanding -Wl,-
Bstatic,-T,./src/linker.ld,-Map,build/hello_world.map,--print-memory-usage -Lbuild -lc_nano
Memory region         Used Size  Region Size  %age Used
```

```
RAM:        9728 B        96 KB      9.90%
```

17.3 Подключаем контроллер прерываний PLIC

Почти любое периферийное оборудование может быть настроено для формирования потока асинхронных событий, которые принято называть аппаратными прерываниями или IRQ (от «**I**nterrupt **Re**e**Q**uest»). Использование аппаратных прерываний для работы с периферийными устройствами позволяет освободить центральный процессор (вычислительное ядро) от необходимости тратить понапрасну циклы для опроса регистров готовности или состояния периферийных устройств, они (устройства) сами сообщают о своей готовности путем формирования аппаратного прерывания — устанавливают сигнал на линиях прерывания в определенное состояние. Аппаратное прерывание обычно обрабатывается процессором путем приостанова выполнения главной программы и передачи управления подпрограмме-обработчику, которую принято называть ISR (от «Interrupt Service Routine») или «IRQ handler». В обработчике происходит быстрое взаимодействие с аппаратурой, например, ввод/вывод блока данных, чтение регистра статуса и т. д., после чего управление возвращается к главной программе, к следующей инструкции в месте её прерывания.

17.3.1 Разрабатываем простейший контроллер прерываний MicroPLIC

В синтезируемом ядре VexRiscv поддерживается два варианта обработки прерываний согласно спецификации RISC-V: «прямой» и «векторный». «Прямой» вариант, это когда все виды исключений и прерываний имеют один общий обработчик, адрес которого содержится в машинном CSR регистре (от «Control and State Register») **mtvec**. Этот обработчик анализирует содержимое другого машинного CSR регистра **mcause** и в, зависимости от его значения, производит вызов соответствующей процедуры обработки. «Векторный» вариант — когда на каждый вид исключения имеется свой обработчик, а таблица векторов располагается по адресу, содержащемуся во все том же CSR регистре **mtvec**. Выбор способа, которым будут обрабатываться прерывания, осуществляется установкой 0-го бита в этом же регистре **mtvec**. Если этот бит равен 0 — то используется «прямой» способ, иначе - «векторный».

Традиционно в RISC архитектурах предпочтение отдается «прямому» способу, и далее все наши рассуждения будут исходить из этого. Следует заметить, что все виды прерываний (аппаратные и программные) в архитектуре RISC-V принято называть «исключением» (exceptions). Кстати, процесс старта (reset) вычислительного ядра тоже является исключением.

Если мы посмотрим на список подключаемых плагинов в СнК Murax, то увидим следующую строку:

```
new CsrPlugin(CsrPluginConfig.smallest(mtvecInit = if(withXip) 0xE0040020l else
0x80000020l)),
```

Плагин **CsrPlugin**, помимо поддержки CSR регистров, также отвечает за реализацию исключений (прерываний) и здесь при подключении этого плагина производится первоначальная настройка содержимого регистра **mtvec**. В нашем случае, так как мы не используем XiP, он устанавливается в значение **0x80000020** — т. е. выбирается «прямой» способ обработки исключений и единый обработчик будет находится по адресу с начала области RAM **со смещением 0x20** байт. В этом месте компилятор GCC расположит нам код инициализации («C Run-time» — или CRT) и обработчик исключений **trap_entry**. Код этой

«ловушки» прост — он сохраняет значения важных регистров на стеке и передает управление функции **irqCallback()** для дальнейшей обработки исключения или прерывания.

Этим же плагином **CsrPlugin** в VexRiscv определено два сигнала аппаратных прерываний: **timerInterrupt** и **externalInterrupt**. Отделить аппаратное прерывание от внутреннего исключения можно по старшему биту CSR регистра **mcause** — если он установлен в «1», то произошло аппаратное прерывание. К сигналу прерывания **timerInterrupt** обычно подключается системный таймер, а к **externalInterrupt** подключают какое-то внешнее периферийное устройство.

Если требуется обрабатывать прерывания от нескольких устройств, то на сигнал **externalInterrupt** подключается контроллер прерываний (**PLIC** - «Platform-Level Interrupt Controller»), который как бы расширяет количество входных линий прерывания за счет того, что он запоминает в своих внутренних регистрах источники внешних прерываний, активных на данный момент, и предоставляет программный механизм для того, чтобы а) выяснить от какого источника поступило прерывание и б) обработать прерывания в каком-то порядке, т. е. приоритизировать источники.

По умолчанию в СнК Murax отсутствует какой-либо контроллер прерываний, а на входную линия вектора **externalInterrupt** подключен сигнал прерывания от UART (последовательного порта). Так как мы собираемся расширять наш СнК и постепенно добавлять в него периферийные устройства, то встает необходимость обеспечить подключение линий прерываний от этих устройств. Иными словами, нам потребуется свой маленький PLIC.

Создать простейший PLIC, без приоритизации и очередей, несложно, достаточно завести несколько регистров для сохранения состояния входных линий прерываний и регистр для их маскирования. Код такого контроллера на языке SpinalHDL приведен ниже.

```
package mylib

import spinal.core._
import spinal.lib._
import spinal.lib.Counter
import spinal.lib.bus.amba3.apb.{Apb3, Apb3Config, Apb3SlaveFactory}

class Apb3MicroPLICCtrl(irq_nums: Int = 32) extends Component {
        val io = new Bundle {
                val apb = slave( Apb3( addressWidth = 16, dataWidth = 32))
                val externalInterrupt = out Bool()
                val IRQLines = in Bits(irq_nums bits)
        }

        // Define control words and connect it to APB3 bus
        val busCtrl = Apb3SlaveFactory(io.apb)

        val IRQEnabled = Reg(Bits(irq_nums bits)) init(0)
        busCtrl.readAndWrite(IRQEnabled, address = 0)

        val IRQPending = Reg(Bits(irq_nums bits)) init(0)
        busCtrl.readAndWrite(IRQPending, address = 1 * (irq_nums / 8))

        busCtrl.read(io.IRQLines, address = 2 * (irq_nums / 8))

        val IRQPolarity = Reg(Bits(irq_nums bits)) init(0)
        busCtrl.readAndWrite(IRQPolarity, address = 3 * (irq_nums / 8))

        val IRQLastValue = Reg(Bits(irq_nums bits)) init(0)
        busCtrl.read(IRQLastValue, address = 4 * (irq_nums / 8))

        def setIRQ(irq_line: Bool, irq_num: Int): Unit = {
                if(irq_num >= irq_nums) {
                        throw new Exception("MicroPLIC: Cannot add IRQ line with number: " +
irq_num + " as it is too big!");
```

```
                }

                io.IRQLines(irq_num) := irq_line
        }

        for (i <- 0 to (irq_nums - 1)) {
                IRQPending(i).setWhen( IRQEnabled(i) &&
                                !(io.IRQLines(i) === IRQLastValue(i)) &&
                                (io.IRQLines(i) === IRQPolarity(i))
                )
        }

        io.externalInterrupt := !(IRQPending === 0) // trigger external interrupt if there's
any pending bits set

        IRQLastValue := io.IRQLines // recall last state of all IRQ lines
}
```

Данный PLIC выполнен в виде класса **Apb3MicroPLICCtrl**, который является компонентом и имеет следующий интерфейс:

- **apb** — представляет собой интерфейс к периферийной шине **Apb3**, к которой данный компонент будет подключаться в режиме «слэйва».
- **IRQLines** — многобитный входной сигнал от источников прерываний, подключаемых к контроллеру. По-умолчания задана размерность 32 бита, что позволяет отслеживать прерывания от 32-х периферийных устройств.
- **externalInterrupt** — выходной сигнал прерывания для подключения контроллера к вычислительному ядру.

Внутри контроллер содержит следующие регистры, которые отображаются на адресное пространство машины и доступны программно:

- **IRQEnabled** — содержит битовые маски разрешения прерываний от обслуживаемых источников. Состояние лог «1» означает, что прерывание от данного источника разрешено.
- **IRQPending** — содержит битовые флаги, индицирующие о том, что соответствующий источник сформировал сигнал прерывания и ждет его обработки.
- **IRQPolarity** — содержит конфигурационные флаги, указывающие на полярность входных линий запросов прерываний от источников. Если в соответствующем бите содержится лог «0», то прерывание от данного источника регистрируется при переходе линии прерывания из «high» в «low» (по спаду), иначе — по переходу «low» в «high» (по фронту).
- **IRQLastValue** — регистр для отслеживания изменений состояний входных линий запросов прерывания.

Из кода контроллера видно, что вся его логика работы состоит в том, чтобы по каждому тактовому сигналу сравнивать состояния входных линий запросов прерываний с их предыдущими значениями и если обнаруживается изменение, согласно полярности, то устанавливается соответствующий бит в регистре **IRQPending**. Если в регистре **IRQPending** установлен хотя бы один бит в лог «1», то на выходе контроллера формируется сигнал **externalInterrupt**, который будучи подключенным к вычислительному ядру вызовет обработку вектора прерываний (вызов ISR). Для подключения линий запроса прерывания от источников имеется метод **setIRQ()**, принимающий на вход два параметра: первый параметр это сигнал запроса на прерывание от источника, второй — номер бита, ассоциируемого с данным источником.

Разумеется, такой незатейливый контроллер имеет массу недостатков, но он вполне годен для большинства задач и проверен в бою.

Разместим текст нашего контроллера в файле **./src/main/scala/mylib/MicroPLIC.scala**, а в код СнК Murax добавим следующее:

В заголовок файла **Murax.scala** добавим строку подключения библиотеки:

```
import mylib.Apb3MicroPLICCtrl
```

В ассоциативный массив **apbMapping** добавим новый элемент, подобно тому, как мы делали при добавлении машинного таймера **MachineTimer** в главе «17.*1 Микросекундный машинный таймер MTIME*»:

```
//******** APB peripherals *********
val apbMapping = ArrayBuffer[(Apb3, SizeMapping)]()
...
val plic = new Apb3MicroPLICCtrl()
apbMapping += plic.io.apb     -> (0x60000, 64 kB)
externalInterrupt := plic.io.externalInterrupt
plic.io.IRQLines := 0
```

Теперь пересадим порт UART, описываемый компонентом **Apb3UartCtrl,** на нулевой канал (нулевой бит) нашего контроллера PLIC, а системный таймер **MuraxApb3Timer** на 3-й канал:

```
val uartCtrl = Apb3UartCtrl(uartCtrlConfig)
uartCtrl.io.uart <> io.uart
///externalInterrupt setWhen(uartCtrl.io.interrupt)
plic.setIRQ(uartCtrl.io.interrupt, 0)
apbMapping += uartCtrl.io.apb  -> (0x10000, 4 kB)

val timer = new MuraxApb3Timer()
//timerInterrupt setWhen(timer.io.interrupt)
plic.setIRQ(timer.io.interrupt, 3)
apbMapping += timer.io.apb     -> (0x20000, 4 kB)
```

При необходимости, аналогичным образом на этот PLIC тут же можно завести прерывания от ряда линий **Apb3Gpio**.

И последний штрих — разрешим ядру VexRiscv обрабатывать внешние прерывания. Для этого найдем в коде СнК Murax переменную:

```
//val externalInterrupt = False
val externalInterrupt = Bool()
```

и заменим на её на объявление сигнала типа Bool, как показано выделенным шрифтом.

Традиционно выполняем сборку проекта командой **make clean && make** и добиваемся устранения синтаксических ошибок.

Статистика после выполнения синтеза и размещения выглядит следующим образом:

```
Info: Device utilisation:
Info:              TRELLIS_IO:    54/  197    27%
Info:                    DCCA:     2/   56     3%
Info:                DP16KD:    48/   56    85%
...
Info:               EHXPLLL:     1/    2    50%
...
Info:             TRELLIS_FF:  1528/24288     6%
Info:           TRELLIS_COMB:  2815/24288    11%
Info:           TRELLIS_RAMW:    36/ 3036     1%

Info: Max frequency for clock                          '$glbnet$clk75': 84.40 MHz (PASS at
75.00 MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 158.78 MHz (PASS at
12.00 MHz)
```

17.3.2 Задействуем контроллер прерываний MicroPLIC из Си программы

Наш PLIC готов к работе. Его регистры будут доступны с адресного пространства **0xF0060000**. Для удобства взаимодействия с ним добавим соответствующие константы в Си код программы «hello_world»:

В файл **murax.h**:

```
#include "plic.h"
...
#define PLIC            ((PLIC_Reg*)(0xF0060000))
```

Создадим новый заготовочный файл **plic.h** следующего содержания:

```
#ifndef _PLIC_H_
#define _PLIC_H_

#include <stdint.h>

typedef struct
{
  volatile uint32_t ENABLE;
  volatile uint32_t PENDING;
  volatile uint32_t IRQLINE;
  volatile uint32_t POLARITY;
  volatile uint32_t IRQLAST;
} PLIC_Reg;


#define PLIC_IRQ_UART0          0x00000001
#define PLIC_IRQ_TIMER0         0x00000008

#endif /* _PLIC_H_ */
```

Далее при работе с прерываниями нам потребуются примитивы для чтения, записи и изменения битов в регистрах управления машиной или CSR (Control and Status Register). Для этого создадим заголовочный файл с именем **riscv.h** и добавим в него следующие макроопределения:

```
rz@devbox:~/VexRiscv/src/main/c/murax/hello_world$ cat src/riscv.h

#ifndef _RISCV_H_
#define _RISCV_H_

//exceptions
#define CAUSE_ILLEGAL_INSTRUCTION 2
#define CAUSE_MACHINE_TIMER 7
#define CAUSE_SCALL 9

//interrupts
#define CAUSE_MACHINE_EXTERNAL 11


#define MEDELEG_INSTRUCTION_PAGE_FAULT (1 << 12)
#define MEDELEG_LOAD_PAGE_FAULT (1 << 13)
#define MEDELEG_STORE_PAGE_FAULT (1 << 15)
#define MEDELEG_USER_ENVIRONNEMENT_CALL (1 << 8)
#define MIDELEG_SUPERVISOR_SOFTWARE (1 << 1)
#define MIDELEG_SUPERVISOR_TIMER (1 << 5)
#define MIDELEG_SUPERVISOR_EXTERNAL (1 << 9)

#define MIP_STIP (1 << 5)
#define MIE_MTIE (1 << CAUSE_MACHINE_TIMER)
#define MIE_MEIE (1 << CAUSE_MACHINE_EXTERNAL)
```

```
#define MSTATUS_UIE         0x00000001
#define MSTATUS_SIE         0x00000002
#define MSTATUS_HIE         0x00000004
#define MSTATUS_MIE         0x00000008
#define MSTATUS_UPIE        0x00000010
#define MSTATUS_SPIE        0x00000020
#define MSTATUS_HPIE        0x00000040
#define MSTATUS_MPIE        0x00000080
#define MSTATUS_SPP         0x00000100
#define MSTATUS_HPP         0x00000600
#define MSTATUS_MPP         0x00001800
#define MSTATUS_FS          0x00006000
#define MSTATUS_XS          0x00018000
#define MSTATUS_MPRV        0x00020000
#define MSTATUS_SUM         0x00040000
#define MSTATUS_MXR         0x00080000
#define MSTATUS_TVM         0x00100000
#define MSTATUS_TW          0x00200000
#define MSTATUS_TSR         0x00400000
#define MSTATUS32_SD        0x80000000
#define MSTATUS_UXL         0x0000000300000000
#define MSTATUS_SXL         0x0000000C00000000
#define MSTATUS64_SD        0x8000000000000000

#define SSTATUS_UIE         0x00000001
#define SSTATUS_SIE         0x00000002
#define SSTATUS_UPIE        0x00000010
#define SSTATUS_SPIE        0x00000020
#define SSTATUS_SPP         0x00000100
#define SSTATUS_FS          0x00006000
#define SSTATUS_XS          0x00018000
#define SSTATUS_SUM         0x00040000
#define SSTATUS_MXR         0x00080000
#define SSTATUS32_SD        0x80000000
#define SSTATUS_UXL         0x0000000300000000
#define SSTATUS64_SD        0x8000000000000000


#define PMP_R     0x01
#define PMP_W     0x02
#define PMP_X     0x04
#define PMP_A     0x18
#define PMP_L     0x80
#define PMP_SHIFT 2

#define PMP_TOR   0x08
#define PMP_NA4   0x10
#define PMP_NAPOT 0x18

#define RDCYCLE 0xC00 //Read-only cycle Cycle counter for RDCYCLE instruction.
#define RDTIME 0xC01 //Read-only time Timer for RDTIME instruction.
#define RDINSTRET 0xC02 //Read-only instret Instructions-retired counter for RDINSTRET
instruction.
#define RDCYCLEH 0xC80 //Read-only cycleh Upper 32 bits of cycle, RV32I only.
#define RDTIMEH 0xC81 //Read-only timeh Upper 32 bits of time, RV32I only.
#define RDINSTRETH 0xC82 //Read-only instreth Upper 32 bits of instret, RV32I only.


#define csr_swap(csr, val) \
({ \
    unsigned long __v = (unsigned long)(val); \
    __asm__ __volatile__ ("csrrw %0, " #csr ", %1" \
                : "=r" (__v) : "rK" (__v)); \
    __v;                                \
})

#define csr_read(csr) \
({ \
    register unsigned long __v; \
    __asm__ __volatile__ ("csrr %0, " #csr \
                : "=r" (__v)); \
    __v; \
})

#define csr_write(csr, val) \
({ \
    unsigned long __v = (unsigned long)(val); \
```

```
    __asm__ __volatile__ ("csrw " #csr ", %0" \
                : : "rK" (__v)); \
})

#define csr_read_set(csr, val) \
({ \
    unsigned long __v = (unsigned long)(val); \
    __asm__ __volatile__ ("csrrs %0, " #csr ", %1" \
                : "=r" (__v) : "rK" (__v)); \
    __v; \
})

#define csr_set(csr, val) \
({ \
    unsigned long __v = (unsigned long)(val); \
    __asm__ __volatile__ ("csrs " #csr ", %0" \
                : : "rK" (__v)); \
})

#define csr_read_clear(csr, val) \
({ \
    unsigned long __v = (unsigned long)(val); \
    __asm__ __volatile__ ("csrrc %0, " #csr ", %1" \
                : "=r" (__v) : "rK" (__v)); \
    __v; \
})

#define csr_clear(csr, val) \
({ \
    unsigned long __v = (unsigned long)(val); \
    __asm__ __volatile__ ("csrc " #csr ", %0" \
                : : "rK" (__v)); \
})

static inline unsigned long __attribute__((const)) cpuid() {
        unsigned long res;
        asm ("csrr %0, mcpuid" : "=r"(res));
        return res;
}

static inline unsigned long __attribute__((const)) impid() {
        unsigned long res;
        asm ("csrr %0, mimpid" : "=r"(res));
        return res;
}

#endif // _RISCV_H_
```

В данной конфигурации VexRiscv у нас имеется всего один вектор прерывания, общий, как для всех аппаратных прерываний, связанных с **externalInterrupt**, так и для генерируемых вычислительным ядром исключений. Поэтому для того, чтобы обрабатывать аппаратные прерывания от конкретного устройства в обработчике прерываний (для Си программ это функция **irqCallback()**), необходимо сначала выяснить, что является источником прерывания, проверив старший бит машинного слова **mcause —** исключение или прерывание от внешнего устройства. Если источником является сигнал **externalInterrupt**, то далее следует проверить регистр **PLIC->PENDING**, содержащий флаги сработанных прерываний, и в соответствии с флагами самостоятельно вызывать обработчик для нужного устройства. Таким образом, код обработки прерываний может выглядеть следующим образом:

Подключим заголовочный файл с описанием примитивов CSR:

```
#include "riscv.h"
```

Объявим пару глобальных переменных для подсчета числа прерываний:

```
volatile int total_irqs = 0;
volatile int extint_irqs = 0;
```

Объявим функцию обработки исключений, которая распечатает нам код исключения и «зависнет» в бесконечном цикле:

```
void crash(int cause) {

        print("\r\n*** EXCEPTION: ");
        printhex(cause);
        print("\r\n");

        while(1);
}
```

Объявим функцию обработки прерываний от внешних устройств:

```
void externalInterrupt() {

        unsigned int pending_irqs = PLIC->PENDING;

        print("EXTIRQ: pending flags = ");
        printhex(pending_irqs);

        unsigned int a = UART->DATA;
        printhex(a);

        PLIC->PENDING &= 0x0; // clear all pending ext interrupts
}
```

Объявим функцию — вектор прерываний:

```
void irqCallback() {

        // Interrupts are already disabled by machine

        int32_t mcause = csr_read(mcause);
        int32_t interrupt = mcause < 0;    // HW interrupt if true, exception if false
        int32_t cause     = mcause & 0xF;

        if(interrupt){
                switch(cause) {
                        case CAUSE_MACHINE_TIMER: {
                                print("\r\n*** irqCallback: machine timer irq ? WEIRD!\r\
n");
                                break;
                        }
                        case CAUSE_MACHINE_EXTERNAL: {
                                externalInterrupt();
                                extint_irqs++;
                                break;
                        }
                        default: {
                                print("\r\n*** irqCallback: unsupported exception cause: ");
                                printhex((unsigned int)cause);
                        } break;
                }
        } else {
                crash(cause);
        }

        total_irqs++;
}
```

Весь этот код поместим в файл **main.c**.

Теперь в тело основного цикла программы, в функцию **main()**, добавим код разрешения прерываний от всех устройств:

```
        char *test = malloc(1024);
        println("Malloc test:");
        printhex((unsigned int)test);

        // Configure interrupt controller
        PLIC->POLARITY = 0xffffffff; // Set all IRQ polarity to High
        PLIC->PENDING = 0; // Clear pending IRQs
        PLIC->ENABLE = 0xffffffff; // Enable all ext interrupts

        // Configure UART IRQ sources: bit(0) - TX interrupts, bit(1) - RX interrupts
        UART->STATUS |= (1<<1); // Allow only RX interrupts

        csr_set(mstatus, MSTATUS_MIE); // Enable Machine interrupts
       ...
```

и распечатку значений накопленного числа прерываний:

```
        while(1){
                unsigned int shift_time;
                unsigned int t1, t2;
                //println("Hello world, this is VexRiscv!");
                //char str[128];
                //vsnprintf(str, 128, "Hello world, this is VexRiscv!\r\n", NULL);
                //print(str);
                printf("Hello world, this is VexRiscv!\r\n");
                printf("PLIC: pending = %0X, total_irqs = %d, extint_irqs = %d\r\n", PLIC-
>PENDING, total_irqs, extint_irqs);
        ...
```

Традиционно выполняем сборку проекта командой **make clean && make**, загружаем битстрим, запускаем **minicom** и наблюдаем следующие сообщения:

```
Checking SRAM at: 90000000
SRAM total fails: 00000000
Enabled heap on SRAM
Malloc test:
90000008
EXTIRQ: pending flags = 00000001
0001004D
Hello world, this is VexRiscv!
PLIC: pending = 0, total_irqs = 0, extint_irqs = 0
000000C6
000282CD
Hello world, this is VexRiscv!
PLIC: pending = 0, total_irqs = 0, extint_irqs = 0
000000C6
```

Теперь, если в порт отправить какие-нибудь данные, а для этого достаточно просто понажимать клавиши в эмуляторе терминала, то мы будем наблюдать следующие сообщения:

```
Hello world, this is VexRiscv!
PLIC: pending = 0, total_irqs = 0, extint_irqs = 0
EXTIRQ: pending flags = 00000001
00010073
EXTIRQ: pending flags = 00000001
00010064
EXTIRQ: pending flags = 00000001
00010066
EXTIRQ: pending flags = 00000001
00010073
000000C7
000282CD
Hello world, this is VexRiscv!
PLIC: pending = 0, total_irqs = 4, extint_irqs = 4
```

Видно, что аппаратные прерывания приводят к вызову функции **irqCallback()**, а за ней функции **externalInterrupt()**, о чем говорит нам битовое поле **pending flags = 00000001**, где нулевой бит установлен в лог «1» и отвечает за контроллер UART.

*Замечание: если при посылке данных в порт UART не формируются прерывания, то следует проверить опцию аппаратного контроля RTS/CTS в программе **minicom**, она должна быть отключена. В противном случае **minicom** не будет посылать данные в порт, дожидаясь готовности физически отсутствующего сигнала.*

17.4 Подключаем контроллер FastEthernet (MAC)

Плата «Карно» снабжена интерфейсом для подключения к сети с помощью микросхемы LAN8710, которая представляет собой PHY трансивер стандарта FastEthernet (100Base-T). Трансивер — это такое устройство, которое осуществляет преобразование цифрового сигнала от устройства в аналоговый сигнал для передачи его в линию связи и обратно, т. е. выполняет модуляцию и демодуляцию сигналов, их кодирование и декодирование. Трансивер LAN8710 оснащен MII интерфейсом (Media-independent Interface) для взаимодействия с вычислительной системой, в составе которой он работает. Линии MII интерфейса трансивера напрямую подключены к сигнальным линиям ПЛИС, что позволяет осуществлять работу с сетью прямо внутри микросхемы ПЛИС, иными словами — позволяет синтезировать MAC контроллер и интегрировать его в вычислительную систему, т. е. в наш вариант СнК Murax. Наличие сетевого интерфейса открывает богатые возможности к творчеству и решению практических задач, поэтому задействовать этот блок периферийной аппаратуры является крайне важным. Разрабатывать свой MAC контроллер мы не станем, так как это достаточно сложная задача для начинающих, и для описания всего процесса мне пришлось бы написать статью сопоставимого размера. Вместо этого мы задействуем уже готовый компонент из библиотеки SpinalHDL, а точнее — целый набор компонентов. Но прежде следует сказать несколько слов о том, как устроена работа с сетями Ethernet.

17.4.1 Как работает Ethernet

Сети Ethernet — это совокупный набор стандартов проводной пакетной связи без подтверждения и без гарантии доставки пакетов, предложенный компанией Xerox и разработанный в небезызвестном исследовательском центре Xerox Palo Alto Research Center (Xerox PARC). Первый коммерческий вариант Ethernet был принят как стандарт IEEE 802.3 в 1982 году, он использовал в качестве среды передачи сигнала «толстый» коаксиальный кабель (см рис. 29) и имел максимальную скорость передачи 10 Мбит/сек (в то время модемная связь на скорости 9600 бод была вершиной чудес). Этот стандарт получил сокращенное название 10Base-5. Чуть позже он был упрощен (удешевлен) и адаптирован для использования «тонкого» коаксиального кабеля с волновым сопротивлением 50 Ом (см. рис. 30). Этот стандарт получил название 10Base-2.

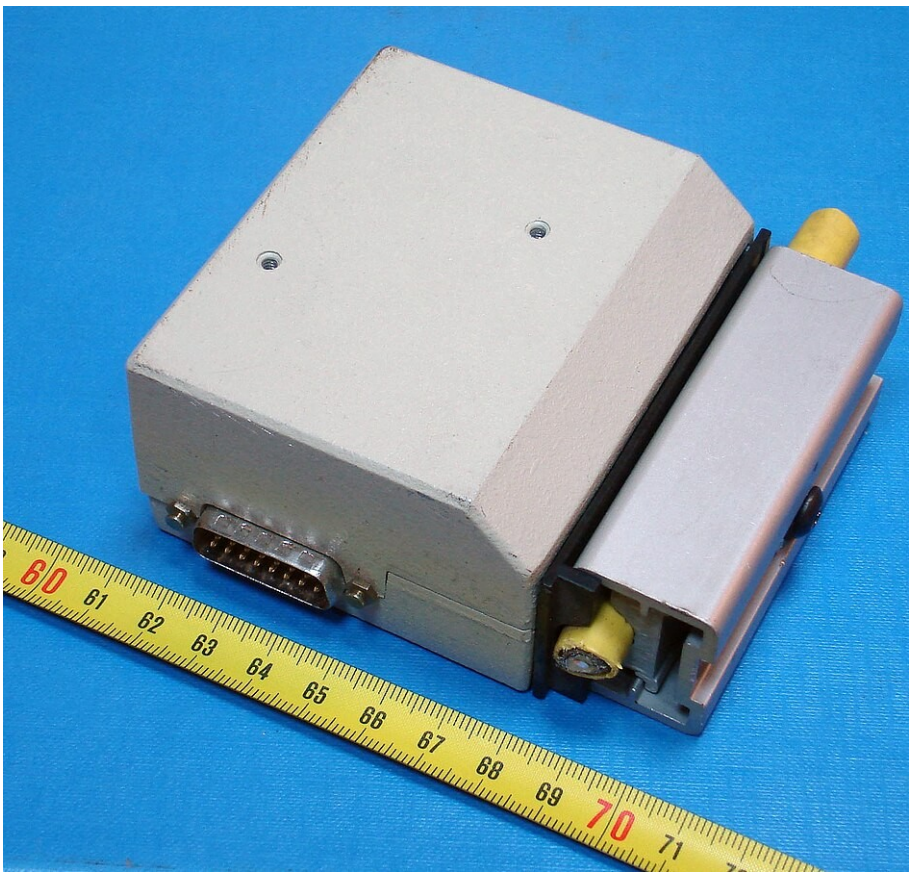


Рис. 29. Трансивер сети Ethernet (10Base-5) буквально «врезался» в коаксиальный кабель.

В обоих стандартах среда (медный провод) представляет собой «общую комнату для переговоров», а значит, в один момент времени «говорить» может только одна станция, остальные в этот момент должны только «слушать». Чтобы выйти на связь, т. е. передать пакет данных, станция обязана какое-то время понаблюдать за активностью в «переговорной комнате» и как только в ней установится «тишина» — начать подавать несущую и через некоторое время передать пакет, предварительно закодировав его, используя специально разработанный метод. После чего станция должна быстро вернуться в состояние прослушивания. В процессе передачи станция так же обязана прослушивать саму себя и сравнивать то, что она передала с тем, что она принимает. Это необходимо для того, чтобы понять, не возникла ли коллизия — ситуация, когда две (и более) станции ведут передачу данных в один и тот же момент времени. Если было детектировано состоянии коллизии, то станция должна «замолчать» на некоторое случайное время, после чего повторить попытку передачи пакета с самого начала. Если во второй попытке также была детектирована коллизия, то станция также обязана «замолчать», удвоив время ожидания и повторить попытку и так далее. Если в течении вразумительного времени станции не удалось передать пакет без коллизии, то это фиксируется как ошибка и пакет выбрасывается, а пользователю (приложению, инициирующему передачу данных) формируется соответствующий сигнал. Если в процессе передачи пакета коллизий не возникло, то считается, что пакет успешно отправлен. Никаких подтверждений на отправленный пакет не принимается и на физическом и канальном уровне не предусматривается. Такой способ доступа к среде (к «переговорной комнате») был назван «**Carrier-sense multiple access with collision detection media access control**» (CSMA/CD MAC) — множественный доступ к среде с определением наличия несущей и детектированием коллизий. Сейчас это упрощенно принято называть MAC, что не совсем правильно, но, как говорится, «who cares?».

Рис. 30. Трансивер Ethernet стандарта 10Base-2 подключался к «тонкому» коаксиальному кабелю через разъем типа BNC.

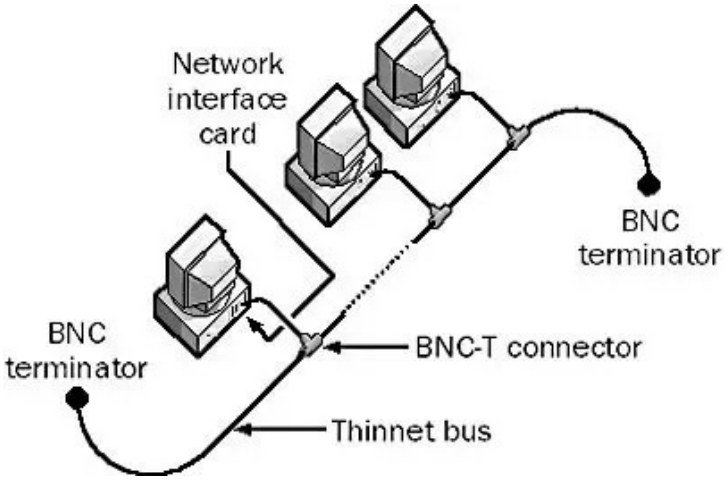


Рис. 31. Схема построения сети Ethernet стандарта 10Base-2.
К коаксиальному кабелю с помощью «тройников» (BNC-T) подключаются станции. Кабель на обоих концах терминируется сопротивлением 50 Ом (BNC terminator).

В 1990 году Ethernet был адаптирован для использования кабеля типа «витая пара» — четыре проводника, попарно свитых друг с другом, одна пара на прием, другая — на передачу данных, а для их коммутации было предложено использовать простое «повторяющее» устройство — Ethernet HUB, которое повторяло принятый сигнал из одного порта во все остальные. Использование такой схемы позволило разделить приемный и передающий каналы, а значит, существенно уменьшить влияние помех и снизить задержку на передачу пакета. Пропускная способность все еще сильно зависела от количества одновременно подключенных к сети станций — чем больше станций, тем больше вероятность возникновения коллизии. Этот стандарт получил название 10Base-T. Быстро стало понятно, что если Ethernet HUB снабдить небольшим интеллектом — т. е. заставить его

проверять адрес назначения пакета и транслировать его только в тот порт, который ведет к станции-приемнику, то можно сильно разгрузить сеть и сделать так, чтобы пропускная способность не зависела от числа станций в сети. Такое устройство назвали **Level-2 Ethernet Switch** (или L2 Ethernet-коммутатор). Современные Ethernet-коммутаторы — это существенно более сложные устройства: в протокол и в функции коммутаторов добавили понятие «виртуальной переговорной комнаты» (VLAN), снабдили функциями маршрутизаторов (Level-3), фильтров, анализа пакетов и т. д.

В 1995 году появляется стандарт FastEthernet (100Base-TX), принятый как IEEE 802.3u и работающий на скорости 100 Мбит/сек по все тем же двум свитым вместе парам, при этом усилились требования к качеству кабеля и его характеристикам (Cat5).

В 1999 году был прият стандарт GigabitEthernet (1000Base-T), известный под номером IEEE 802.3ab. Как следует из названия, данный стандарт позволяет передавать данные со скоростью 1000 Мбит/сек и использует уже четыре свитые пары — причем все четыре одновременно на прием и на передачу, а для отделения передаваемого сигнала от принимаемого используются алгоритмы «эхо подавления» (echo cancellation). Требования к кабелю возрастают еще сильнее (Cat5e, Cat6). В GigabitEthernet при первом включении станции проводят «переговоры» и договариваются, кто из них будет работать «мастером», а кто «слэйвом» - от этого зависит источник синхросигнала и способы управления потоком.

В 2016 году появляются сразу несколько стандартов: 2.5GBase-T, 5GBase-T, 10GBase-T, 25GBase-T и 40GBase-T со скоростями передачи 2.5, 5, 10, 25 и 40 Гбит/сек соответственно.

Несмотря на существование более высокоскоростных стандартов, стандарты FastEthernet (100Base-TX) и GigabitEthernet (1000Base-T) являются самыми популярными по сей день за счет дешевизны и непритязательности к инфраструктуре и кабельному хозяйству — витая пара категории Cat5e, это самый распространенный вид кабеля во всем мире. А стандарт FastEthernet за счет более низкой скорости обмена несложно реализовать в простых микроконтроллерных устройствах. Так, используемый для FastEthernet интерфейс MII рассчитан на синхронный обмен с хост-устройством на частоте 25 МГц, в то время как RGMII для GigabitEthernet работает на частоте 125 МГц и требует передачи блока по 4 бита как по фронту, так и по спаду (DDR), что очень сильно усложняет реализацию MAC контроллера и делает мало пригодным использование GigabitEthernet на дешевых микроконтроллерных устройствах, неспособных формировать сигналы с такой тактовой частотой. По этой причине подавляющее большинство IoT устройств снабжается встроенным MAC контроллерами с MII интерфейсом для подключения к 100Base-TX сети.

Теперь рассмотрим, как осуществляется подключение станции (устройства) к сети Ethernet на примере 100Base-TX. На рис. 32 изображена структурная схема такого соединения с использованием MII и RMII интерфейса. Отличие RMII от MII состоит в том, что в RMII сокращено число линий данных в два раза за счет удвоения частоты до 50 МГц, в остальном они функционируют одинаково.

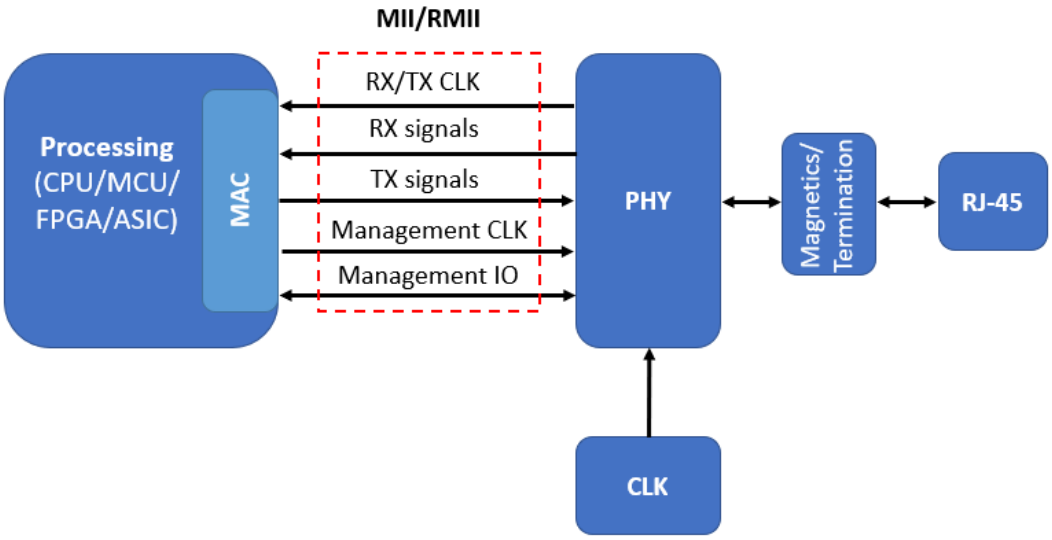


Рис. 32. Структурная схема подключения устройства к сети 100Base-TX с помощью MII/RMII интерфейса.

На стороне хост-устройства выделяется специальный цифровой блок аппаратуры, который принято называть MAC. Обычно это IP-блок, входящий в состав микросхемы микроконтроллера или более сложной СнК. В случае с платой «Карно» MAC мы будем синтезировать внутри микросхемы ПЛИС. Задача MAC состоит в том, чтобы передавать данные пользователя (приложения) в трансивер PHY и обратно по цифровой шине из 4х бит (или 2-х бит в случае RMII). При этом для приема и передачи используются раздельные шины TX_D[0:3] и RX_D[0:4], каждая со своим сигналом тактирования TX_CLK и RX_CLK. Источником тактового сигнала для обеих шин является трансивер (PHY), который для передачи использует свой независимый тактовый генератор (CLK = 25 МГц), а при приеме — восстанавливает тактовый сигнал из линии. Таким образом, на стыке между MAC и PHY возникает пересечение тактовых доменов CoreCLK<->TX_CLK и CoreCLK<->RX_CLK. Для управления передачей шина TX снабжается сигналом TX_EN, который формируется MAC в тот момент, когда он готов начать передачу данных. Аналогично, для приема шина RX снабжается сигналом RX_DV (receive data valid), который передается от PHY к MAC в момент, когда трансивер обнаружил в линии несущую и начал прием данных. Прием и передача пакета осуществляется непрерывно, т. е. как только поступил сигнал RX_DV, MAC контроллер обязан принять всё до последнего полубайта, поступление которого сигнализируется снятием сигнала RX_DV. Помимо этого, шина приема содержит еще три сигнала: RX_COL — сигнализирует о возникновении коллизии, RX_ER — ошибка при приеме и RX_CRS — индикатор наличия несущей. Все эти сигналы необходимы для того, чтобы MAC мог выполнить требования CSMA/CD.

Задача PHY гораздо более сложная — произвести кодирование полученных данных и их модуляцию в линию и наоборот. К счастью, существуют множество готовых микросхем Ethernet трансиверов (PHY) для всех стандартов и разновидностей интерфейсов, поэтому данного вопроса касаться не будем.

Помимо двух шин обмена данных (TX_D и RX_D), Ethernet трансиверы обычно имеют отдельную шину управления «Management Data Input/Output», состоящую из двух сигналов: Management IO (MDIO) — для передачи команд и считывания состояний регистров и Management CLK (MDC) — для формирования тактовых сигналов. С помощью шины управления MAC может задавать режимы работы трансивера (например, Full/Half-duplex,

скорость в линии 10 или 100 Мбит/с), а также считывать его состояние и узнавать подробности о возникающих ошибках. Взаимодействие с трансивером по шине управления MDIO походит на работу с периферийными устройствами по шине типа I2C, но имеет ряд отличий — хост-устройство передает по шине 5 бит адрес подчиненного трансивера (их на шине может быть до 31 шт), 5 бит номера регистра трансивера и 16 бит данных, которые в этот регистр требуется записать (или считать). При обмене по шине MDIO максимальная частота тактового сигнала MDC не должна превышать 2,5 МГц. Временные диаграммы MDIO интерфейса приведены ниже на рис. 33.

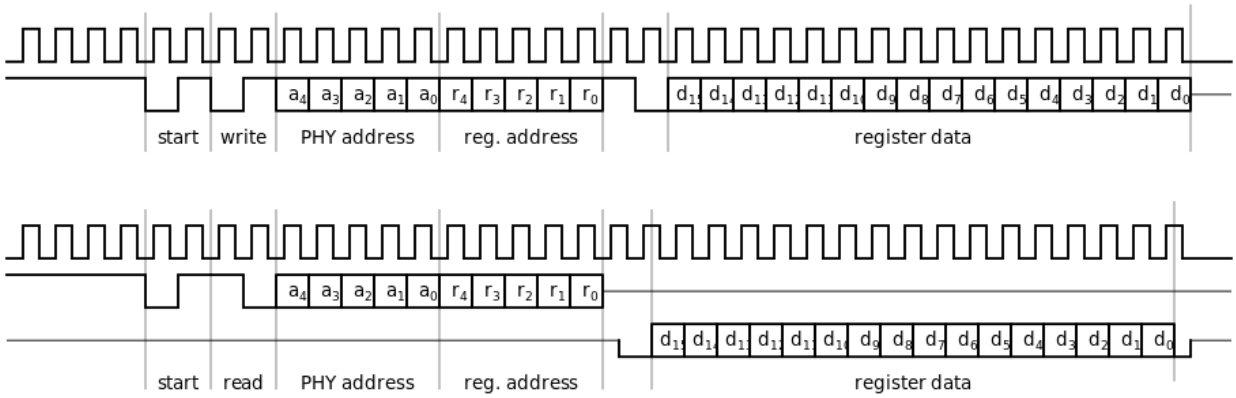


Рис. 33. Временные диаграммы циклов записи (вверху) и чтения (внизу) регистра трансивера по управляющей шине MDIO.

Перечень регистров трансивера и их назначение не специфицированы и реализуются на усмотрение производителя микросхемы трансивера. Но есть сложившиеся традиции — многие малоизвестные производители микросхем PHY повторяют протокол и формат регистров других, более известных производителей с целью совместимости.

Ethernet трансиверы могут формировать отельный сигнал прерывания (LAN_nINT) при изменении статуса — т. е. установке стабильного соединения (окончании процедуры «handshake») или смене параметров линии.

В простых устройствах большой необходимости в шине управления MDIO нет, так как большинство Ethernet трансиверов имеют настройки по умолчанию, позволяющие произвести автоматическую настройку параметров линии без участия MAC, а в интерфейсе MII уже присутствует всё необходимое для приема/передачи данных по сети. Поэтому поддержку шины MDIO мы реализовывать не будем. При необходимости читатель может легко реализовать MDIO программным путем, манипулируя двумя линиями GPIO.

Для того, чтобы принять пакет целиком, MAC должен иметь встроенный буфер размером не меньше, чем максимальный размер Ethernet пакета, который называется «фреймом». Для 100Base-TX максимальный размер фрейма составляет **1522** байта, четыре последний байта фрейма это контрольная сумма (FCS — frame check sequence), которую MAC обязан проверить после приема и перед выдачей данных пользователю. Аналогично требуется отдельный буфер на передачу — в него пользователь помещает пакет, который MAC выдает в трансивер. При передаче MAC обязан рассчитать контрольную сумму и добавить её к данным фрейма при передаче в PHY.

Каждый Ethernet фрейм содержит 6 байт адреса принимающей станции, 6 байт адреса станции отправителя, два байта **EtherType**, определяющих содержимое полезной нагрузки и до **1500** байт данных самой нагрузки. Опционально фрейм может содержать один или два тэга по четыре байта каждый. Присутствие тэгов определяется настройкой коммутатора,

первые два байта тэга всегда содержат число **0x8100,** что рассматривается как зарезервированный **EtherType** для тэгированных фреймов, а следующие два байта — номер VLAN-а («виртуальной комнаты»). Мы не будем рассматривать тэгированные фреймы и способ их формирования, так как, во-первых, это тема отдельного длинного разговора, а во-вторых, для нашей цели (реализация MAC в ПЛИС) это не существенно. На рис. 34 ниже представлен формат нетэгированного Ethernet фрейма.

Layer	Preamble	Start frame delimiter (SFD)	MAC destination	MAC source	802.1Q tag (optional)	Ethertype (Ethernet II) or length (IEEE 802.3)	Payload	Frame check sequence (32-bit CRC)	Interpacket gap (IPG)
	7 octets	1 octet	6 octets	6 octets	(4 octets)	2 octets	42–1500 octets	4 octets	12 octets
Layer 2 Ethernet frame	(not part of the frame)		← 64–1522 octets →						(not part of the frame)
Layer 1 Ethernet packet & IPG	← 72–1530 octets →								← 12 octets →

Рис. 34. Формат Ethernet фрейма.

17.4.2 Подключаем компонент MacEth

Итак, к MAC контроллеру для FastEthernet в составе нашей платы «Карно» мы можем предъявить следующие требования:

- поддержка сигналов интерфейса MII: RX_D[3:0], RX_CLK, RX_ER, RX_DV, RX_COL, RX_CRS, TX_D[3:0], TX_EN, TX_CLK;
- наличие встроенных буферов для приема и передачи фреймов в виде FIFO;
- поддержка пересечения тактовых доменов, отдельно для тракта приема и тракта передачи;
- поддержка управляющих регистров, отображаемых на общее адресное пространство, сброс TX/RX FIFO, определение количества слов свободного места в TX FIFO и количества слов готовых для чтения из RX FIFO;
- поддержка прерываний для TX — при освобождении FIFO (конец передачи), для RX — при приеме фрейма.
- поддержка регистров для чтения и записи в FIFO;
- поддержка регистров статуса (количество переданных фреймов, количество ошибок и коллизий);
- автоматический расчет контрольной суммы (FCS).

Как отмечалось выше, в составе SpinalHDL имеется весь необходимый набор компонентов для реализации MAC контроллера с описанными выше требования, расположены они в каталоге **lib/src/main/scala/spinal/lib/com/eth/** репозитория SpinalHDL:

```
rz@devbox:~$ cd ~/SpinalHDL

rz@devbox:~/SpinalHDL$ ls -l lib/src/main/scala/spinal/lib/com/eth/
total 40
-rw-rw-r-- 1 rz rz  841 Feb  1 09:14 BmbMacEth.scala
-rw-rw-r-- 1 rz rz 5158 Apr  3  2021 Mac.scala
```

```
-rw-rw-r-- 1 rz rz 9271 Feb  1 09:14 MacRx.scala
-rw-rw-r-- 1 rz rz 9858 Feb  1 09:14 MacTx.scala
-rw-rw-r-- 1 rz rz 4091 Feb  1 09:14 Phy.scala
```

Файл **Mac.scala** содержит управляющую логику контроллера, его главный компонент — **MacEth**. Также в этом файле содержится структура **MacEthParameter**, описывающая параметры MII интерфейса**.**

Файлы MacRx.scala и MacTx.scala содержат реализацию тракта приёма и передачи по интерфейсу MII и FIFO буферы.

Файл Phy.scala содержит описание интерфейса MII, разделенного на три компонента: **MiiRx**, **MiiTx** и **Mdio**.

Для того чтобы подключить компонент **MacEth** к шине **Apb3,** нам потребуется описать свой компонент-обертку. Назовем его **Apb3MacEthCtrl** и разместим в файл **./src/main/scala/mylib/Apb3MacEthCtrl.scala** репозитория VexRiscv рядом с другими нашими компонентами:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ cat
../../../src/main/scala/mylib/Apb3MacEthCtrl.scala
package mylib

import spinal.core._
import spinal.lib._
import spinal.lib.bus.amba3.apb.{Apb3, Apb3Config, Apb3SlaveFactory}
import spinal.lib.eda.altera.QSysify
import spinal.lib.com.eth._

object Apb3MacEthCtrl{
  def getApb3Config = Apb3Config(
    addressWidth = 16,
    dataWidth = 32,
    selWidth = 1,
    useSlaveError = false
  )
}

case class Apb3MacEthCtrl(p : MacEthParameter) extends Component{
        val io = new Bundle{
                val apb =  slave(Apb3(Apb3MacEthCtrl.getApb3Config))
                val interrupt = out Bool()
                val mii = master(Mii( MiiParameter( MiiTxParameter( dataWidth =
p.phy.txDataWidth, withEr = false), MiiRxParameter( dataWidth = p.phy.rxDataWidth))))
        }

        val txCd = ClockDomain(io.mii.TX.CLK)
        val rxCd = ClockDomain(io.mii.RX.CLK)
        val mac = new MacEth(p, txCd, rxCd)
        val phy = PhyIo(p.phy)
        phy <> mac.io.phy

        txCd.copy(reset = mac.txReset) on {
                val tailer = MacTxInterFrame(dataWidth = p.phy.txDataWidth)
                tailer.io.input << phy.tx

                io.mii.TX.EN := RegNext(tailer.io.output.valid)
                io.mii.TX.D := RegNext(tailer.io.output.data)
        }

        rxCd on {
                phy.rx << io.mii.RX.toRxFlow().toStream
        }

        val busCtrl = Apb3SlaveFactory(io.apb)
        val bridge = mac.io.ctrl.driveFrom(busCtrl)
        io.interrupt := bridge.interruptCtrl.pending
}
```

Как видно из кода, компонент **Apb3MacEthCtrl** инкапсулирует два других компонента: **MacEth** и **PhyIo**, в нем создаются два тактовых домена **txCd** и **rxCd**, которые передаются в компонент **MacEth** и осуществляются связи интерфейсных сигналов. Компонент **Apb3MacEthCtrl** имеет три интерфейсных сигнала: **apb** — для стыковки с шиной **Apb3** вычислительного ядра, сигнал **mii** — для соединения с трансивером и сигнал **interrupt** для формирования сигнала прерывания. Сигнал **interrupt** мы далее назначим на канал **#2** ранее разработанного контроллера прерываний (компонент **MicroPLIC**).

Подключение компонента **Apb3MacEthCtrl** к шине **Apb3** выполняется традиционным способом:

Сначала добавим внешний комплексный сигнал **mii** в компонент **Murax**:

```
case class Murax(config : MuraxConfig) extends Component{
  import config._

  val io = new Bundle {
    ...
    //Peripherals IO
    val gpioA = master(TriStateArray(gpioWidth bits))
    val uart = master(Uart())
    val mii = master(Mii(MiiParameter(MiiTxParameter(dataWidth =
config.macConfig.phy.txDataWidth, withEr = false), MiiRxParameter( dat
aWidth = config.macConfig.phy.rxDataWidth))))
   ...
  }
```

Далее добавим компонент **Apb3MacEthCtrl** в ассоциативный массив **apbMapping** в файле **Murax.scala,** также как это было сделано ранее для компонентов **Apb3MicroPLICCtrl** и **Apb3MachineTimer** и привяжем внешние сигналы **io.mii**. Код подключения выглядит следующим образом:

```
    val macCtrl = new Apb3MacEthCtrl(macConfig)
    apbMapping += macCtrl.io.apb     -> (0x70000, 64 kB)
    macCtrl.io.mii <> io.mii
    plic.setIRQ(macCtrl.io.interrupt, 2)
```

Данный компонент будет иметь регистры управления, отображаемые на адресное пространство 0xF0000000 + 0x70000 = **0xF0070000**, а линия прерывания будет формировать прерывание по каналу **#2** (бит 2 в регистрах) контроллера **MicroPLIC**.

Помимо этого, нам еще потребуется заполнить параметрами конфигурационную структуру **MacEthParameter**. Для этого добавим новый параметр **macConfig** в определение класса **MuraxConfig()**:

```
case class MuraxConfig(coreFrequency : HertzNumber,
                       onChipRamSize      : BigInt,
                       onChipRamHexFile   : String,
                       pipelineDBus       : Boolean,
                       pipelineMainBus    : Boolean,
                       pipelineApbBridge  : Boolean,
                       gpioWidth          : Int,
                       macConfig          : MacEthParameter,
                       uartCtrlConfig     : UartCtrlMemoryMappedConfig,
                       xipConfig          : SpiXdrMasterCtrl.MemoryMappingParameters,
                       hardwareBreakpointCount : Int,
                       cpuPlugins         : ArrayBuffer[Plugin[VexRiscv]])
```

Заполним структуру и разместим рядом с параметром **uartCtrlConfig** в реализации объекта **MuraxConfig()**:

```
macConfig = MacEthParameter(
  phy = PhyParameter(
    txDataWidth = 4,
    rxDataWidth = 4
  ),
  rxDataWidth = 32,
  rxBufferByteSize = 4096,
  txDataWidth = 32,
  txBufferByteSize = 4096
),
```

В структуре **MacEthParameter** параметры **rxBufferByteSize** и **txBufferByteSize** определяют размер буферов FIFO для входных и выходных Ethernet фреймов и в нашем случае имеют размер достаточный для сохранения двух обычных фреймов. Jumbo фреймы поддерживать не будем в виду ограниченного объема набортной RAM памяти.

Параметры **rxDataWidth** и **txDataWidth** задают размер слова FIFO. В нашем случае размер слова обуславливается размерностью шины — 32 бита. Это означает, что при работе с FIFO все операции будут выполнятся в 32-х битных словах, а не в байтах. На это следует обратить внимание при написании драйвера.

Вложенная структура **PhyParameter** задает параметры MII интерфейса — по 4 бита на прием и передачу.

Чтобы этот код успешно компилировался, не забываем подключить соответствующие библиотеки в заголовке файла **Murax.scala**:

```
import mylib.Apb3MicroPLICCtrl
import mylib.Apb3MacEthCtrl
import spinal.lib.com.eth._
```

Остался последних штрих: добавить описание внешних интерфейсных сигналов MII в файл **karnix_cabga256.lpf** и в файл-обертку **toplevel.v**.

Добавим следующие директивы в файл **karnix_cabga256.lpf:**

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ grep io_mii karnix_cabga256.lpf

LOCATE COMP "io_mii_mdio" SITE "B1";            # LAN_MDIO
IOBUF PORT "io_mii_mdio" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_mdio" PULLMODE=NONE;
LOCATE COMP "io_mii_mdc" SITE "A2";             # LAN_MDC
IOBUF PORT "io_mii_mdc" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_mdc" PULLMODE=NONE;
LOCATE COMP "io_mii_nint" SITE "B3";            # LAN_nINT
IOBUF PORT "io_mii_nint" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_nint" PULLMODE=NONE;
LOCATE COMP "io_mii_nrst" SITE "A3";            # LAN_nRST
IOBUF PORT "io_mii_nrst" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_nrst" PULLMODE=NONE;
FREQUENCY PORT "io_mii_TX_CLK" 25.0 MHz;
LOCATE COMP "io_mii_TX_CLK" SITE "B4";          # LAN_TXCLK
IOBUF PORT "io_mii_TX_CLK" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_TX_CLK" PULLMODE=NONE;
LOCATE COMP "io_mii_TX_EN" SITE "A4";           # LAN_TXEN
IOBUF PORT "io_mii_TX_EN" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_TX_EN" PULLMODE=NONE;
LOCATE COMP "io_mii_TX_D[0]" SITE "B5";         # LAN_TXD0
IOBUF PORT "io_mii_TX_D[0]" IO_TYPE=LVCMOS33;
```

```
IOBUF PORT "io_mii_TX_D[0]" PULLMODE=NONE;
LOCATE COMP "io_mii_TX_D[1]" SITE "A5";         # LAN_TXD1
IOBUF PORT "io_mii_TX_D[1]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_TX_D[1]" PULLMODE=NONE;
LOCATE COMP "io_mii_TX_D[2]" SITE "B6";         # LAN_TXD2
IOBUF PORT "io_mii_TX_D[2]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_TX_D[2]" PULLMODE=NONE;
LOCATE COMP "io_mii_TX_D[3]" SITE "A6";         # LAN_TXD3
IOBUF PORT "io_mii_TX_D[3]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_TX_D[3]" PULLMODE=NONE;
FREQUENCY PORT "io_mii_RX_CLK" 25.0 MHz;
LOCATE COMP "io_mii_RX_COL" SITE "B2";          # LAN_COL
IOBUF PORT "io_mii_RX_COL" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_RX_COL" PULLMODE=NONE;
LOCATE COMP "io_mii_RX_CRS" SITE "C1";          # LAN_CRS
IOBUF PORT "io_mii_RX_CRS" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_RX_CRS" PULLMODE=NONE;
LOCATE COMP "io_mii_RX_ER" SITE "C2";           # LAN_RXER
IOBUF PORT "io_mii_RX_ER" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_RX_ER" PULLMODE=NONE;
LOCATE COMP "io_mii_RX_DV" SITE "B7";           # LAN_RXDV
IOBUF PORT "io_mii_RX_DV" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_RX_DV" PULLMODE=NONE;
LOCATE COMP "io_mii_RX_CLK" SITE "F1";          # LAN_RXCLK
IOBUF PORT "io_mii_RX_CLK" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_RX_CLK" PULLMODE=NONE;
LOCATE COMP "io_mii_RX_D[0]" SITE "D1";         # LAN_RXD0
IOBUF PORT "io_mii_RX_D[0]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_RX_D[0]" PULLMODE=NONE;
LOCATE COMP "io_mii_RX_D[1]" SITE "E2";         # LAN_RXD1
IOBUF PORT "io_mii_RX_D[1]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_RX_D[1]" PULLMODE=NONE;
LOCATE COMP "io_mii_RX_D[2]" SITE "E1";         # LAN_RXD2
IOBUF PORT "io_mii_RX_D[2]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_RX_D[2]" PULLMODE=NONE;
LOCATE COMP "io_mii_RX_D[3]" SITE "F2"; #       LAN_RXD3
IOBUF PORT "io_mii_RX_D[3]" IO_TYPE=LVCMOS33;
IOBUF PORT "io_mii_RX_D[3]" PULLMODE=NONE;
```

Отредактируем файл **toplevel.v** и добавим проброс сигналов **io_mii_*** из модуля **toplevel** в модуль **Murax**:

```
module toplevel(
    input   io_clk25,
    ...
    input   [3:0] io_mii_RX_D, // MII RX data lines
    input   io_mii_RX_DV, // MII RX data valid
    input   io_mii_RX_CLK, // MII RX clock
    input   io_mii_RX_ER, // MII RX error flag
    input   io_mii_RX_CRS, // MII RX carrier sense flag
    input   io_mii_RX_COL, // MII RX collision detection flag
    output  [3:0] io_mii_TX_D, // MII TX data lines
    output  io_mii_TX_EN, // MII TX data enable
    input  io_mii_TX_CLK, // MII TX clock
  );
   ...

  Murax murax (
    ...
    .io_mii_RX_D(io_mii_RX_D),
    .io_mii_RX_CLK(io_mii_RX_CLK),
    .io_mii_RX_DV(io_mii_RX_DV),
    .io_mii_RX_ER(io_mii_RX_ER),
    .io_mii_RX_CRS(io_mii_RX_CRS),
    .io_mii_RX_COL(io_mii_RX_COL),
    .io_mii_TX_D(io_mii_TX_D),
    .io_mii_TX_CLK(io_mii_TX_CLK),
    .io_mii_TX_EN(io_mii_TX_EN),
  );
```

Собираем проект командой **make clean && make**, устраняем синтаксические ошибки, смотрим на статистику в результате синтеза и наблюдаем как ресурсы ПЛИС обильно пошли в расход:

```
Info: Device utilisation:
Info:             TRELLIS_IO:    69/  197    35%
Info:                   DCCA:     4/   56     7%
Info:                 DP16KD:    52/   56    92%
Info:             MULT18X18D:     4/   28    14%
Info:                 ALU54B:     0/   14     0%
Info:                EHXPLLL:     1/    2    50%
...
Info:             TRELLIS_FF:  2360/24288     9%
Info:           TRELLIS_COMB:  4256/24288    17%
Info:           TRELLIS_RAMW:    36/ 3036     1%
...
Info: Routing globals...
Info:     routing clock net $glbnet$io_core_jtag_tck$TRELLIS_IO_IN using global 0
Info:     routing clock net $glbnet$io_mii_RX_CLK$TRELLIS_IO_IN using global 1
Info:     routing clock net $glbnet$io_mii_TX_CLK$TRELLIS_IO_IN using global 2
Info:     routing clock net $glbnet$clk using global 3
...
Info: Max frequency for clock    '$glbnet$io_mii_TX_CLK$TRELLIS_IO_IN': 73.72 MHz (PASS at
25.00 MHz)
Info: Max frequency for clock    '$glbnet$io_mii_RX_CLK$TRELLIS_IO_IN': 113.21 MHz (PASS at
25.00 MHz)
Info: Max frequency for clock                                '$glbnet$clk': 75.26 MHz (PASS at
75.00 MHz)
Info: Max frequency for clock '$glbnet$io_core_jtag_tck$TRELLIS_IO_IN': 135.35 MHz (PASS at
12.00 MHz)
...
Info: Program finished normally.
```

У нас готов контроллер MAC. В следующей главе мы рассмотрим, как сделать простейший драйвер и выполнить обмен пакетами с DHCP сервером из Си программы «hello_world».

17.4.3 Разрабатываем драйвер для компонента MacEth

Слово «драйвер», наверное, это слишком громкое слово для такого проекта, тем не менее, нам потребуется создать три функции, которые вполне сойдут за минималистичный драйвер. Это будут функции: **mac_init()** — для инициализации (сброса) MAC контроллера, **mac_tx()** — для передачи Ethernet фрейма и **mac_rx()** — для считывания принятого фрейма из буфера MAC контроллера. Также мы подвесим код обработки прерываний, поступающих от MAC контроллера через MicroPLIC.

Перед тем как начать реализацию драйвера, разберем программный интерфейс (API), предоставляемый модулем **MacEth**. Как уже отмечалось, тракты приема и передачи работают независимо друг от друга, а значит их можно рассматривать как два отдельных устройства, каждое из которых имеет свой FIFO буфер длиной в два обычных Ethernet фрейма, элементом которого является 32-х битное слово. Для управления FIFO буферами **MacEth** предоставляет один общий регистр управления **CTRL**; один общий регистр **STAT** для статистики ошибок; раздельные регистры **TX/RX** для записи/чтения данных; и раздельные регистры **TX_AVAIL** и **RX_AVAIL** для выяснения текущей заполненности FIFO. Все регистры **MacEth** находятся в адресном пространстве, начиная с **0xF0070000**, их можно представить приведенной ниже структурой. Описание этой структуры мы поместим в новый заголовочный файл **mac.h** в каталоге с кодом программы «hello_world», а базовый адрес области с регистрами контроллера опишем с помощью макро **MAC**:

```
#define MAC     ((MAC_Reg*)(0xF0070000))
```

```
typedef struct
{
        volatile uint32_t CTRL;
        volatile uint32_t res1[3];
        volatile uint32_t TX;
        volatile uint32_t TX_AVAIL;
        volatile uint32_t res2[2];
        volatile uint32_t RX;
        volatile uint32_t res3[2];
        volatile uint32_t STAT;
        volatile uint32_t RX_AVAIL;
} MAC_Reg;
```

Регистр **CTRL** имеет смещение 0 байт и содержит биты следующего назначения:

- Бит 0 — запись лог «1» вызывает сброс аппаратуры передающего тракта и обнуление FIFO.
- Бит 1 — чтение лог «1» сигнализирует готовность и наличие места в FIFO для записи минимум одного слова.
- Бит 2 — запись лог «1» включает выравнивание данных на передающих линиях MII по границе 16 бит. Мне не совсем понятно, зачем это может понадобиться, предполагаю, что имеются трансиверы, которые имеют большее число линий данных, чем 4 для стандарта MII.
- Бит 4 — запись лог «1» вызывает сброс аппаратуры принимающего тракта и обнуление FIFO.
- Бит 5 — чтение лог «1» сигнализирует о наличии в FIFO входного тракта полностью принятого фрейма.
- Бит 6 — запись лог «1» включает выравнивание данных по 16 бит на приемных линиях интерфейса MII.
- Бит 7 — чтение лог «1» сигнализирует о том, что FIFO буфер входного тракта полностью заполнен и есть вероятность переполнения (потери части данных принятого фрейма).

Регистр **TX** имеет смещение 8 байт. Запись 32-х битного слова в этот регистр помещает слово в FIFO передающего тракта.

Регистр **TX_AVAIL** имеет смещение 12 байт — чтение из этого регистра возвращает количество свободных слов, которые еще можно поместить в буфер FIFO передающего тракта.

Регистр **RX** имеет смещение 20 байт — чтение из этого регистра изымает одно слово принятого фрейма из FIFO в принимающем тракте.

Регистр **STAT** имеет смещение 28 байт — содержит следующие поля:

- Биты [7:0] — содержат счетчик числа ошибок по приему (ERRORS).
- Биты [15:8] — содержат счетчик числа ошибок по передачи (DROPS).

Регистр **RX_AVAIL** имеет смещение 32 байта — чтение этого регистра возвращает число слов, содержащихся в FIFO приемного тракта, готовых к изъятию.

Опишем все это дело следующими макроопределениями:

```
#define MAC_CTRL_TX_RESET       0x00000001
#define MAC_CTRL_TX_READY       0x00000002
#define MAC_CTRL_TX_ALIGN_EN    0x00000004
```

```
#define MAC_CTRL_RX_RESET       0x00000010
#define MAC_CTRL_RX_PENDING     0x00000020
#define MAC_CTRL_RX_ALIGN_EN    0x00000040
#define MAC_CTRL_RX_FULL        0x00000080

#define MAC_STAT_ERRORS         0x000000ff
#define MAC_STAT_DROPS          0x0000ff00
```

Далее для удобства работы с аппаратными регистрами добавим в этот же заголовочный файл следующие инструментальные inline функции:

```
inline uint32_t mac_writeAvailability(MAC_Reg *reg){
        return reg->TX_AVAIL;
}

inline uint32_t mac_readAvailability(MAC_Reg *reg){
        return reg->RX_AVAIL;
}

inline uint32_t mac_readDrops(MAC_Reg *reg){
        return reg->STAT >> 8;
}

inline uint32_t mac_readErrors(MAC_Reg *reg){
        return reg->STAT & 0xff;
}

inline uint32_t mac_rxPending(MAC_Reg *reg){
        return reg->CTRL & MAC_CTRL_RX_PENDING;
}

inline uint32_t mac_rxFull(MAC_Reg *reg){
        return reg->CTRL & MAC_CTRL_RX_FULL;
}

inline uint32_t mac_txReady(MAC_Reg *reg){
        return reg->CTRL & MAC_CTRL_TX_READY;
}

inline uint32_t mac_getCtrl(MAC_Reg *reg){
        return reg->CTRL;
}

inline uint32_t mac_setCtrl(MAC_Reg *reg, uint32_t val){
        return reg->CTRL = val;
}

inline uint32_t mac_getRx(MAC_Reg *reg){
        return reg->RX;
}

inline uint32_t mac_pushTx(MAC_Reg *reg, uint32_t val){
        return reg->TX = val;
}
```

Добавим заголовки для трех функций драйвера:

```
void mac_init(void);
int mac_rx(uint8_t* mac_buf);
int mac_tx(uint8_t *mac_buf, int frame_size);
```

Добавим макроопределения для кодов ошибок, выдаваемых драйвером:

```
#define MAC_ERR_OK              0
#define MAC_ERR_RX_FIFO         1
#define MAC_ERR_ARGS            2
#define MAC_ERR_RX_TIMEOUT      3
```

Доработаем заголовочный файл **mac.h,** добавив условную компиляцию для ограничения избыточных включений, подключим файл **stdint.h** с описанием используемых типов данных и включим вывод отладочных сообщений определив макро MAC_DEBUG:

```
#ifndef _MAC_H_
#define _MAC_H_

#include <stdint.h>

#define MAC_DEBUG       1 // включим вывод отладочных сообщений

// … здесь располагается приведенный выше кодирование

#endif // _MAC_H_
```

У нас готов заголовочный файл для драйвера. Теперь приступим к реализации.

Сначала реализуем функцию инициализации аппаратуры (сброса), так как она очень простая. Для сброса достаточно установить в лог «1» соответствующие биты, выждать какое-то время (не менее 10 мс) и перевести в лог «0» для перехода устройства в рабочее состояние. Поместим следующий код инициализации в файл **mac.c** в том же каталоге:

```
extern void delay_us(unsigned int);

void mac_init(void) {
        mac_setCtrl(MAC, MAC_CTRL_TX_RESET | MAC_CTRL_RX_RESET);
        delay_us(10000);
        mac_setCtrl(MAC, 0);
}
```

Задержка осуществляется функций **delay_us()** которую мы ранее имплементировали в основном файле программы — в **main.c**, поэтому просто сошлемся на неё как на **extern** функцию.

Добавим код вспомогательной функции **mac_printf()** для вывода отладочной информации, используя внутренний буфер. Этот код можно будет легко отключить установкой макро-переменной **MAC_DEBUG**:

```
#define MAC_DEBUG       1

#ifdef MAC_DEBUG
#include <stdarg.h>

extern void print(char *);

char mac_dbg_msg[256];

void mac_printf(const char* fmt, ...) {
        va_list args;
        va_start(args, fmt);
        vsnprintf(mac_dbg_msg, 256, fmt, args);
        va_end(args);
        print(mac_dbg_msg);
}
#endif
```

Далее реализуем функцию для отправки данных **mac_tx()**. Эта функция будет принимать два параметра: **mac_buf** — указатель на буфер в памяти машины, содержащий подготовленный к отправке Ethernet фрейм и **frame_size** — размер фрейма в байтах.

Используемый нами MAC контроллер **MacEth** имеет следующую специфику при отправке фрейма:

- перед началом отправки фрейма необходимо дождаться готовности устройства;
- перед записью любого слова в FIFO необходимо дождаться освобождения места в нём;
- первое слово, которое помещается в FIFO, обязано содержать размер фрейма, исчисляемый в **битах**;
- в след за этим словом в FIFO помещаются данные фрейма слово за словом по 32 бита, данные в словах размещаются в формате «network» (big endian).

Таким образом, в функции **mac_tx()** нам необходимо сначала выполнить проверку валидности входных данных, рассчитать число передаваемых бит, после чего дождаться готовности аппаратуры и FIFO, передать первое слово, содержащее количество бит в фрейме и далее, в цикле перемещая по четыре байта, отправить весь фрейм в FIFO. Выглядеть это может следующим образом:

```
int mac_tx(uint8_t *mac_buf, int frame_size) {

        if(mac_buf == NULL || frame_size < 0 || frame_size >= 2048) {
                #ifdef MAC_DEBUG
                mac_printf("mac_tx() wrong args: mac_buf = %p, frame_size = %d\r\n",
mac_buf, frame_size);
                #endif
                return -MAC_ERR_ARGS;
        }

        uint32_t bits = frame_size*8;
        uint32_t words = (bits+31)/32;

        #ifdef MAC_DEBUG
        mac_printf("mac_tx() sending MAC frame size = %d (%d words)\r\n", frame_size,
words);
        #endif

        // wait for MAC controller to get ready to send
        while(!mac_txReady(MAC));

        // wait for space in TX FIFO
        while(mac_writeAvailability(MAC) == 0);

        mac_pushTx(MAC, bits); // first word in FIFO is number of bits in following packet

        uint32_t byte_idx = 0;
        uint32_t word = 0;
        uint32_t words_sent = 0;
        uint8_t *p = mac_buf;
        uint32_t bytes_left = frame_size;

        while(bytes_left) {

                word |= ((*p++) & 0xff) << (byte_idx * 8);

                if(byte_idx == 3) {
                        while(mac_writeAvailability(MAC) == 0);
                        mac_pushTx(MAC, word);
                        word = 0;
                        words_sent++;
                }

                byte_idx = (byte_idx + 1) & 0x03;

                bytes_left--;
        }

        // Write remaining tail
        if(byte_idx != 0) {
                while(mac_writeAvailability(MAC) == 0);
                mac_pushTx(MAC, word);
                words_sent++;
        }
```

```
        #ifdef MAC_DEBUG
        mac_printf("mac_tx() %02x:%02x:%02x:%02x:%02x:%02x <- %02x:%02x:%02x:%02x:%02x:%02x
"
                "type: 0x%02x%02x, byte_idx = %d, words sent = %d, frame_size = %d\r\n",
                mac_buf[0], mac_buf[1], mac_buf[2], mac_buf[3], mac_buf[4], mac_buf[5],
                mac_buf[6], mac_buf[7], mac_buf[8], mac_buf[9], mac_buf[10], mac_buf[11],
                mac_buf[12], mac_buf[13],
                byte_idx, words_sent, frame_size);
        #endif

        return MAC_ERR_OK;
}
```

Следует обратить внимание на то, что в FIFO записываются слова по четыре байта, а значит возможны случаи, когда размер фрейма не будет кратным числу 4. Такой случай в приведенном коде обработан отдельно путем проверки переменной **byte_idx**, она указывает на номер байта в передаваемом слове.

С отправкой фрейма разобрались. Теперь займемся изъятием из FIFO приемного тракта полученного фрейма. Специфика работы контроллера **MacEth** здесь следующая:

- перед чтением данных из FIFO приемного тракта необходимо дождаться появления в нём данных;
- первое слово, считанное из FIFO, содержит размер принятого фрейма, исчисляемый в **битах**, а значит нужно считать его и вычислить количество слов для цикла чтения самого фрейма;
- произвести считывание всего фрейма из FIFO словами по 32 бита, данные в котором поступают в формате «network» (big endian);

Специальная обработка случая, когда длина изымаемого фрейма не кратна 4 байтам, здесь не требуется, достаточно корректно вычислить число циклов чтения (число считываемых слов).

Реализация функции **mac_rx()** может выглядеть следующим образом:

```
int mac_rx(uint8_t* mac_buf) {

        uint32_t bytes_read = 0;

        #ifdef MAC_DEBUG
        mac_printf("mac_rx() begin\r\n");
        #endif

        if(mac_rxPending(MAC)) {

                uint32_t bits = mac_getRx(MAC);
                uint32_t words = (bits+31)/32;
                uint32_t bytes_left = (bits+7)/8;
                uint8_t* p = mac_buf;
                uint32_t word;

                if(bytes_left > 2048) {
                        mac_printf("mac_rx() RX FIFO error, bytes_left = %d bytes (%d bits)\
r\n", bytes_left, bits);
                        mac_init();
                        return -MAC_ERR_RX_FIFO;
                }

                #ifdef MAC_DEBUG
                mac_printf("mac_rx() reading %d bytes (%d bits)\r\n", bytes_left, bits);
                #endif

                while(bytes_left) {

                        int i = 1000;

                        while(mac_rxPending(MAC) == 0 && i-- > 1000);
```

```
                        if(i == 0) {
                                #if MAC_DEBUG
                                mac_printf("mac_rx() timeout, bytes_left = %d\r\n",
bytes_left);
                                #endif
                                return -MAC_ERR_RX_TIMEOUT;
                        }

                        word = mac_getRx(MAC);

                        for(i = 0; i < 4 && bytes_left; i++) {
                                *p++ = (uint8_t) word;
                                word = word >> 8;
                                bytes_left--;
                                bytes_read++;
                        }

                }

                #ifdef MAC_DEBUG
                mac_printf("mac_rx() %02x:%02x:%02x:%02x:%02x:%02x <- %02x:%02x:%02x:%02x:
%02x:%02x "
                                "type: 0x%02x%02x, bytes_left = %d, "
                                "words = %d, bytes_read = %d, last word = 0x%08x\r\n",
                        mac_buf[0], mac_buf[1], mac_buf[2], mac_buf[3], mac_buf[4],
mac_buf[5],
                        mac_buf[6], mac_buf[7], mac_buf[8], mac_buf[9], mac_buf[10],
mac_buf[11],
                        mac_buf[12], mac_buf[13],
                        bytes_left, words, bytes_read, word);
                #endif
        }

        #ifdef MAC_DEBUG
        mac_printf("mac_rx() done, bytes read = %d\r\n", bytes_read);
        #endif

        return bytes_read;
}
```

Функция **mac_rx()** принимает в качестве единственного параметра указатель на область памяти, в которую следует поместить изымаемый фрейм из FIFO приемного тракта. Предполагается, что пользователь заранее позаботился о том, чтобы выделить (статически или на «куче») буфер достаточного объема для размещения фрейма. Напомню, что стандартный Ethernet фрейм не может превышать 1522 байта.

Обычно следующим шагом в написании драйвера является создание обработчика прерываний от устройства. Используемый нами компонент **MacEth** умеет формировать прерывание в момент, когда у него в FIFO буфере приемного тракта появляются данные для чтения. Сигнал **interrupt** оберточного компонента **Apb3MacEthCtrl** мы ранее уже подвязали к 2-му каналу нашего контроллера прерываний. Но в данном примере с целью упрощения мы пойдем другим путем — будем опрашивать состояние входного FIFO из функции **delay()**, которая циклически вызывается в главном цикле программы, и если в FIFO что-то имеется, то вызывать функцию **mac_rx()**. Такой подход существенно упростит код и позволит избавиться от ряда неприятных артефактов, связанных с обработкой прерываний. Напомню, что цель данной главы является продемонстрировать работоспособность компонента **MacEth**, а не написание правильного и всесторонне выверенного драйвера.

Добавим в код функции delay(), находящейся в основном файле программы — **main.c**, а также заведем статический буфер для принимаемого фрейма:

```
static char mac_rx_buf[2048];
```

```
void delay(uint32_t loops){

        if(mac_rxPending(MAC)) {
                mac_rx(mac_rx_buf);
        }

        for(int i=0;i<loops;i++){
                //int tmp = GPIO_A->OUTPUT;
                asm volatile ("slli t1,t1,0x10");
        }
}
```

Принятые данные мы обрабатывать не будем, но при включенном макро **MAC_DEBUG** мы будем наблюдать в отладочном порту сообщения от функции **mac_rx()**, в том числе распечатку заголовка принятого Ethernet фрейма. Этого вполне достаточно для демонстрационных целей.

Последний штрих в написании драйвера — вызов функции инициализации MAC контроллера. Добавим её в код функции main() сразу после конфигурирования UART:

```
csr_set(mstatus, MSTATUS_MIE); // Enable Machine interrupts

// Configure UART0 IRQ sources: bit(0) - TX interrupts, bit(1) - RX interrupts
UART->STATUS |= (1<<1); // Allow only RX interrupts

mac_init();

GPIO_A->OUTPUT_ENABLE = 0x0000000F;
```

Не забываем добавить **#include <mac.h>** в заголовок файла **main.c**. Собираем проект как обычно, с помощью **make clean && make**, устраняем ошибки компиляции, но не спешим с загрузкой в ПЛИС. Ведь для того, чтобы получить какой-то Ethernet пакет, неплохо было бы сначала послать в сеть какой-то пакет, и здесь вполне логичным является отправка запроса Discover в DHCP сервер. Формированием примитивного DHCP запроса мы займемся в следующей главе.

17.4.4 Отправляем запрос в DHCP сервер

Работа устройства с сетью в современных реалиях не мыслима без поддержки стека протоколов TCP/IP, а его функционирование не возможно без IP адреса. Получение IP адреса является первоочередной задачей любого сетевого устройства и осуществляется это обычно посредством широко используемого протокола Dynamic Host Configuration Protocol (DHCP). В рамках данной статьи я не буду рассказывать о том, как запустить весь TCP/IP стек на нашем синтезированном СнК, хоть это и очень интересная тема, скажу лишь, что это возможно и совсем несложно, если задействовать готовый стек lwIP. Здесь же мы ограничимся созданием примитивного запроса к DHCP серверу на получение IP адреса.

Протокол DHCP является достаточно сложным и многогранным механизмом управления сетью. Одной из его основных задач является выделение вновь подключаемым к сети устройствам уникальных IP адресов и снабжение их конфигурационной информацией, достаточной для работы в сети. Выполняет эту функцию специальное программное обеспечение — DHCP сервер, один или несколько таких серверов устанавливаются и настраиваются для обслуживания запросов от DHCP клиентов в каждый широковещательный сегмент сети Ethernet.

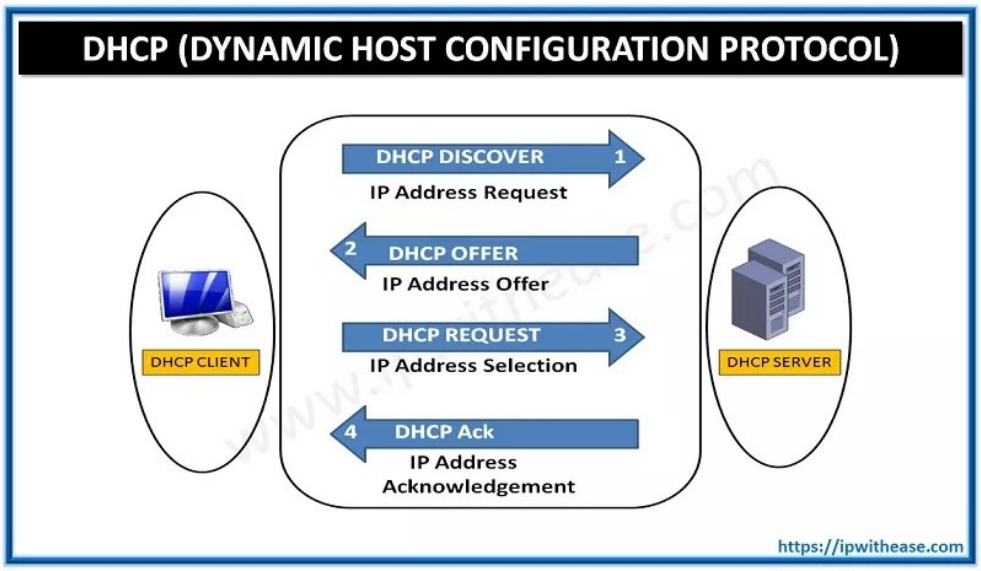


Рис. 35. Обмен между клиентом и сервером по протоколу DHCP на стадии первичного подключения.

Первоначальный запрос к DHCP серверу поступает от клиент с пустым (нулевым) IP адресом на широковещательный IP адрес **255.255.255.255** в виде IP/UDP пакета, при этом используются следующие номера UDP портов: **68** — для клиента и **67** — для сервера. В теле запроса клиент указывает тип запроса (например **Discover**) и список конфигурационных параметров, которые он хочет получить от сервера. Отправив запрос, клиент дожидается ответа от сервера. Сервер на первоначальный запрос **Discover** формирует ответ-предложение — **Offer**, указывая список предлагаемых к использованию параметров, в том числе IP адрес и сетевую маску, а также запрашиваемые опциональные параметры (адрес маршрутизатора, адрес DNS сервера, доменное имя и т. д.). Клиент применяет полученные параметры, т. е. соглашается с предложением, о чем уведомляет сервер другим запросом — **Request**, на что сервер отвечает подтверждением — **Acknowledge**. Типовая схема взаимодействия клиента и сервера по протоколу DHCP представлена на рис. 35. Реализовывать весь обмен по протоколу DHCP нам не придется. Чтобы получить какой-то ответ от DHCP сервера, нам достаточно реализовать только запрос **Discover**, это уже создаст какой-то обмен по сети и позволит нам оценить работоспособность задействованного MAC контроллера и созданного нами примитивного драйвера.

Ethernet фрейм с DHCP запросом имеет следующую общую структуру:

- Заголовок Ethernet фрейма — содержит MAC адреса станции назначения и станции отправителя. В запросе **Discover** адрес назначения устанавливается в **ff:ff:ff:ff:ff:ff**, что означает «отправить всем».
- IP заголовок — содержит нулевой IP адреса хоста отправителя и широковещательный IP адрес получателя (**255.255.255.255**).
- UDP заголовок — содержит номера портов клиента (**68**) и сервера (**67**).
- DHCP запрос — содержит уникальный номер запроса **XID**, тип запроса и список параметров. По уникальному **XID** клиент и сервер идентифицируют запросы/ответы друг друга.

Для создания DHCP запроса нам потребуются структуры MAC, IP и UDP заголовков, а также структура самого DHCP запроса.

Структура для MAC заголовка:

```
typedef struct
{
        uint8_t hw_dst_addr[6];
        uint8_t hw_src_addr[6];
        uint16_t etype;
} MAC_Header;
```

Структура для IP заголовка:

```
typedef struct {
        uint8_t ver_ihl;
        uint8_t dscp_ecn;
        uint16_t pkt_len;
        uint16_t ident;
        uint16_t flags_frags;
        uint16_t ttl_proto;
        uint16_t header_csum;
        uint32_t src_addr;
        uint32_t dst_addr;

} __attribute__((__packed__)) IP_Header;
```

Структура для UDP заголовка:

```
typedef struct {
        uint16_t src_port;
        uint16_t dst_port;
        uint16_t pkt_len;
        uint16_t crc;
} __attribute__((__packed__)) UDP_Header;
```

Структура для DHCP запроса:

```
typedef struct {
        uint8_t op;
        uint8_t htype;
        uint8_t hlen;
        uint8_t hops;
        uint32_t xid;
        uint16_t secs;
        uint16_t flags;
        uint32_t ciaddr;
        uint32_t yiaddr;
        uint32_t siaddr;
        uint32_t giaddr;
        char chaddr[16];
        char sname[64];
        char file[128];
        char magic[4];
        char opt[10];
} __attribute__((__packed__)) DHCP_Message;
```

Структура отправляемого сообщения содержит все вышеперечисленные заголовки:

```
typedef struct {
        MAC_Header mac;
        IP_Header ip;
        UDP_Header udp;
        DHCP_Message dhcp;
} __attribute__((__packed__)) MAC_IP_UDP_DHCP_Message;
```

Для того чтобы заполнить структуру IP заготовка нам потребуется вычислить контрольную сумму. Позаимствуем с сайта Wikipedia готовую функцию для этих целей:

```
uint16_t ip_check_sum(uint16_t *addr, int len)
{
```

```
        int nleft = len;
        uint16_t *w = addr;
        uint32_t sum = 0;
        uint16_t answer = 0;

        while (nleft > 1)  {
                sum += *w++;
                nleft -= 2;
        }

        if (nleft == 1) {
                *(uint8_t *)(&answer) = *(uint8_t *)w ;
                sum += answer;
        }

        sum = (sum >> 16) + (sum & 0xffff);
        sum += (sum >> 16);
        answer = ~sum;
        return(answer);
}
```

В UDP заголовке тоже присутствует поле контрольной суммы, но для IP протокола версии 4 это поле опционально, т. е. будем помещать в него нули.

Для заполнения MAC заголовка нам потребуется два MAC адреса — отправителя (наше устройство) и получателя (сервер). В качестве получателя в протоколе DHCP регламентируется использовать широковещательный MAC адрес вида **ff:ff:ff:ff:ff:ff**. В качестве MAC адреса нашего устройства выберем случайный адрес, удовлетворяющий требованиям Local Administered Address стандарта IEEE 802: **06:aa:bb:cc:dd:ee**.

```
uint8_t hw_my_addr[6] = { 0x06, 0xaa, 0xbb, 0xcc, 0xdd, 0xee };
uint8_t hw_brd_addr[6] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };
```

Для заполнения некоторых полей в структурах нам понадобится пара макроопределений для перестановки последовательности байт в слове:

```
#define SWAP32(X)       __builtin_bswap32((X))
#define SWAP16(X)       __builtin_bswap16((X))
```

Теперь мы можем описать функцию для заполнения DHCP сообщения и его отправки в сеть. На странице в Wikipedia, посвященной протоколу DHCP, приведен детальный пример пакета с запросом Discover, воспользуемся этими данными:

```
int dhcp_send_discover(void) {

        static uint32_t xid = 0x12345678;
        int frame_size = 0;

        MAC_IP_UDP_DHCP_Message *msg = (MAC_IP_UDP_DHCP_Message*) malloc(2048);

        if(msg == NULL) {
                printf("dhcp_send_discover: failed to allocate buffer\r\n");
                return -100;
        }

        bzero(msg, 2048);

        msg->dhcp.op = 1; msg->dhcp.htype = 1; msg->dhcp.hlen = 6; msg->dhcp.hops = 0;
        msg->dhcp.xid = SWAP32(xid++); msg->dhcp.secs = 0; msg->dhcp.flags = 0;
        msg->dhcp.magic[0]=0x63; msg->dhcp.magic[1]=0x82; msg->dhcp.magic[2]=0x53; msg-
>dhcp.magic[3]=0x63;
        msg->dhcp.opt[0]=0x35; msg->dhcp.opt[1]=1; msg->dhcp.opt[2]=1; // Discovery
        msg->dhcp.opt[3]=0x37; msg->dhcp.opt[4]=4; // Parameter request list
        msg->dhcp.opt[5]=1; msg->dhcp.opt[6]=3; // mask, router
        msg->dhcp.opt[7]=15; msg->dhcp.opt[8]=6; // domain name, dns
        msg->dhcp.opt[9]=0xff; // end of params
        memcpy(msg->dhcp.chaddr, hw_my_addr, 6);

        frame_size += sizeof(msg->dhcp) + sizeof(msg->udp);
        msg->udp.pkt_len = SWAP16(frame_size);
```

```
        msg->udp.src_port = SWAP16(68); msg->udp.dst_port = SWAP16(67);

        frame_size += sizeof(msg->ip);
        msg->ip.src_addr = 0x0; msg->ip.dst_addr = 0xffffffff;
        msg->ip.ver_ihl = 0x45; msg->ip.ttl_proto = SWAP16((255 << 8) + 17);
        msg->ip.pkt_len = SWAP16(frame_size);
        msg->ip.header_csum = ip_check_sum((uint16_t*)&(msg->ip), sizeof(msg->ip));

        frame_size += sizeof(msg->mac);
        memcpy(msg->mac.hw_dst_addr, hw_brd_addr, 6);
        memcpy(msg->mac.hw_src_addr, hw_my_addr, 6);
        msg->mac.etype = SWAP16(0x0800);

        int ret = mac_tx((uint8_t*)msg, frame_size);

        free(msg);

        return ret;
}
```

Приведенная выше функция **dhcp_send_discover()**, используя **malloc()**, выделяет память под структура типа **MAC_IP_UDP_DHCP_Message**. Далее последовательно заполняет её, начиная с самого конца (с тела DCHP запроса), рассчитывает контрольную сумму IP заголовка, устанавливает IP и MAC адреса и отправляет полученный Ethernet фрейм в сеть, используя функцию драйвера **mac_tx()**. После отправки выделенная область памяти высвобождается библиотечной функцией **free()**. В качестве уникального идентификатора **XID** используется число **0x12345678**, которое заносится в статическую переменную и увеличивается на единицу каждый раз при вызове функции.

Помещаем весь приведенный выше код в отдельный файл **dhcp.c** в том же каталоге, где располагается остальной код программы «hello_world». В заголовке файла **dhcp.c** подключаем используемые библиотеки и не забываем про заголовочный файл нашего драйвера **mac.h**:

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mac.h"
```

Теперь осталось добавить вызов функции **dhcp_send_discover()** из функции **main()** программы «hello_world». Добавим описание функции в заголовок файла **main.c**:

```
volatile int total_irqs = 0;
volatile int extint_irqs = 0;

int dhcp_send_discover(void);
```

Вызов **dhcp_send_discover()** расположим в конце тела цикла **while**, после распечатки времени исполнения операции сдвига:

```
        ...
        printhex(shift_time);
        printhex(t2 - t1);

        dhcp_send_discover();
```

Собираем проект, устраняем синтаксические ошибки и загружаем в ПЛИС, подключаем кабель Ethernet, запускаем **minicom** и наблюдаем сообщения:

```
Filling SRAM at: 90000000
Checking SRAM at: 90000000
SRAM total fails: 00000000
Enabled heap on SRAM
```

```
Malloc test:
90000008
Hello world, this is VexRiscv!
PLIC: pending = 0, total_irqs = 0, extint_irqs = 0
EXTIRQ: pending flags = 00000004
mac_rx() begin
mac_rx() reading 64 bytes (512 bits)
mac_rx() 06:aa:bb:cc:dd:ee <- c0:25:e9:ad:2b:1a type: 0x0806, bytes_left = 0, words = 16,
bytes_read = 64, last word = 0x00000000
mac_rx() done, bytes read = 64
000000C6
0002A516
mac_tx() sending MAC frame size = 292 (73 words)
mac_tx() fEXTIRQ: pending flags = 00000004
f:ff:ff:ff:ff:ff <- 06:aa:bb:cc:dd:ee type: 0x0800, byte_idx = 0, words sent = 73,
frame_size = 292
Hello world, this is VexRiscv!
PLIC: pending = 0, total_irqs = 2, extint_irqs = 2
mac_rx() begin
mac_rx() reading 66 bytes (528 bits)
mac_rx() 06:aa:bb:cc:dd:ee <- c0:25:e9:ad:2b:1a type: 0x0800, bytes_left = 0, words = 17,
bytes_read = 66, last word = 0x0000007e
mac_rx() done, bytes read = 66
EXTIRQ: pending flags = 00000004
mac_rx() begin
mac_rx() reading 219 bytes (1752 bits)
mac_rx() ff:ff:ff:ff:ff:ff <- c0:25:e9:ad:2b:1a type: 0x0800, bytes_left = 0, words = 55,
bytes_read = 219, last word = 0x0000000c
mac_rx() done, bytes read = 219
EXTIRQ: pending flags = 00000004
mac_rx() begin
mac_rx() reading 64 bytes (512 bits)
mac_rx() 06:aa:bb:cc:dd:ee <- c0:25:e9:ad:2b:1a type: 0x0806, bytes_left = 0, words = 16,
bytes_read = 64, last word = 0x00000000
mac_rx() done, bytes read = 64
000000C8
0002AB3C
mac_tx() sending MAC frame size = 292 (73 words)
mac_tx() ff:ff:ff:ff:ff:ff <- 06:aa:bb:cc:dd:ee type: 0x0800, byte_idx = 0, words sent = 73,
frame_size = 292
Hello world, this is VexRiscv!
PLIC: pending = 0, total_irqs = 4, extint_irqs = 4
000000C7
0002A516
mac_tx() sending MAC frame size = 292 (73 words)
mac_tx() ff:ff:ff:ff:ff:ff <- 06:aa:bb:cc:dd:ee type: 0x0800, byte_idx = 0, words sent = 73,
frame_size = 292
Hello world, this is VexRiscv!
PLIC: pending = 0, total_irqs = 4, extint_irqs = 4
EXTIRQ: pending flags = 00000004
mac_rx() begin
mac_rx() reading 346 bytes (2768 bits)
mac_rx() 06:aa:bb:cc:dd:ee <- c0:25:e9:ad:2b:1a type: 0x0800, bytes_left = 0, words = 87,
bytes_read = 346, last word = 0x00000add
mac_rx() done, bytes read = 346
mac_rx() begin
mac_rx() reading 346 bytes (2768 bits)
mac_rx() 06:aa:bb:cc:dd:ee <- c0:25:e9:ad:2b:1a type: 0x0800, bytes_left = 0, words = 87,
bytes_read = 346, last word = 0x00000199
mac_rx() done, bytes read = 346
000000C6
0002A516
```

Жирным шрифтом выделена информация о переданных и принятых Ethernet фреймах, содержащих запрос **Discover** и ответ **Offer** от DHCP сервера. В моём случае сервер имеет MAC адрес **c0:25:e9:ad:2b:1a** и это мой домашний маршрутизатор. Если присмотреться, то можно увидеть фреймы с запросами **ARP** (EtherType = 0x0806), адресованные нашему устройству, но ответить мы на них пока что не можем.

Проведем небольшое тестирование - погоняем наше устройство некоторое время в таком режиме запрос-ответ, чтобы понять не зависнет ли драйвер или контроллер. Через 15 минут продолжаем наблюдать в отладочном порту обмен с DHCP сервером:

```
Hello world, this is VexRiscv!
PLIC: pending = 0, total_irqs = 885, extint_irqs = 885
mac_rx() begin
mac_rx() reading 346 bytes (2768 bits)
mac_rx() 06:aa:bb:cc:dd:ee <- c0:25:e9:ad:2b:1a type: 0x0800, bytes_left = 0, words = 87,
bytes_read = 346, last word = 0x00000760
mac_rx() done, bytes read = 346
000000C6
0002A516
mac_tx() sending MAC frame size = 292 (73 words)
mac_tx() ff:fEXTIRQ: pending flags = 00000004
f:ff:ff:ff:ff <- 06:aa:bb:cc:dd:ee type: 0x0800, byte_idx = 0, words sent = 73, frame_size =
292
Hello world, this is VexRiscv!
PLIC: pending = 0, total_irqs = 886, extint_irqs = 886
```

По счетчику прерываний видно, что уже было принято **886** фреймов. Делаем вывод о том, что MAC контроллер и написанный нами драйвер работают стабильно.

Следующим логическим шагом было бы подключение TCP/IP стека, но эту тему я оставлю для отдельной статьи.

18. Потактовая симуляция СнК Murax

В главе «*13.7 Симуляция и верификация в SpinalHDL*» я обещал продемонстрировать, как выполняется потактовая симуляция компонента в SpinalHDL. Настало время это сделать, к тому же есть повод.

В процессе отладки компонента **PipelinedMemoryBusSram** для работы с внешней микросхемой SRAM у меня возникли определенные трудности с реализацией цикла чтения. Напомню, что на плате «Карно» имеется микросхема статической памяти, содержащая 256K ячеек по 16 бит каждая, адресация ячеек которой происходит пословно. Для выбора старшего или младшего байта микросхема имеет два сигнала **#UB** и **#LB** (у компонента **PipelinedMemoryBusSram** это будут сигналы **io.sram_bhe** и **io.sram_ble**). Запись 32-х битного слова в SRAM осуществляется за два такта, а вот чтение за два такта выполнить не удалось — очень часто на шине данных образовывался «мусор» и чтобы нивелировать эту проблему мне пришлось выделять по два такта при чтении каждого 16-ти битного слова. При реализации этого алгоритма у меня возникла некоторая путаница в голове, и чтобы её развеять, мне захотелось посмотреть, как и в какой последовательности приходят сигналы в разрабатываемый компонент **PipelinedMemoryBusSram** и что компонент выдает в ответ. Для этого пришлось освоить, как выполняется потактовая симуляция в SpinalHDL.

По сути, симуляция в SpinalHDL ничем не отличается от верификации. При верификации мы подаем на вход компонента синтетические данные и сравниваем результат с эталонной моделью. При симуляции мы делаем только половину работы — формируем требуемое нам количество тактовых импульсов и наблюдаем за работой составных частей компонента, т. е. просто печатаем или сохраняем интересующие нас сигналы для последующего анализа. Верифицировать и симулировать можно как отдельные компоненты системы, так и всю систему целиком. Правда, верифицировать такую систему, как вычислительное ядро с работающей программой, крайне затратно по времени, и к такому решению прибегают нечасто. А вот просимулировать работу системы в какие-то заданные интервалы времени и понаблюдать за состоянием определенных сигналов — это всегда полезно.

Ну что же, попробуем просимулировать всю нашу синтезируемую систему-на-кристалле **Murax** в течении 10000 тактов и понаблюдаем за сигналами на стыке компонента **PipelinedMemoryBusSram** и **mainBus**, или, если быть более точным, за сигналами **sramCtrl.io.bus** в составе области тактирования **system.** Основная сложность здесь состоит в том, как добраться до интересующих нас сигналов из виртуального испытательного стенда и как их распечатать в лог. Но об этом чуть позже, а для начала опишем испытательный стенд для всей системы:

```
import spinal.sim._
import spinal.core.sim._

object Murax_karnix_Sim {
  def main(args: Array[String]) {

    SimConfig.withWave.compile{
      val dut = new Murax(MuraxConfig.default(withXip = false).copy(
              coreFrequency = 75.5 MHz,
              onChipRamSize = 96 kB ,
              //pipelineMainBus = true,
              onChipRamHexFile = "src/main/c/murax/hello_world/build/hello_world.hex"
      ))

      // More DUT preparations should be put here
      dut

    }.doSim    {

      dut =>

      // Create our own clock domain using external signals mainClk and asyncReset
      val myClockDomain = ClockDomain(dut.io.mainClk, dut.io.asyncReset)

      // Fork process that drives our clock
      myClockDomain.forkStimulus(period = 10)

      // We are in reset state at the beginning
      myClockDomain.assertReset()

      // Simulate next 1K clock cycles
      for(idx <- 0 to 9999) {
        if(idx > 1) { myClockDomain.deassertReset() }
       myClockDomain.waitRisingEdge()
        // … some printouts should be put here
     }
    }
  }
}
```

Как и при верификации, здесь создается новый объект **Murax_karnix_Sim**, который будет выступать точкой входа при вызове сборочной утилиты SBT. **SimConfig** задает параметры симуляции — в данном случае симулятор, помимо прочего, будет формировать файл с временной диаграммой (напомню, что симуляция выполняется внешним тулом **Verilator**, который должен быть установлен в системе пользователя). Далее создается новая область **myClockDomain** для домена тактирования и в ней в цикле формируются тактовые импульсы. В нулевом такте происходит установка сигнала сброса вызовом метода **myClockDomain.assertReset()**, а в 1-м и последующих тактах сигнал сброса снимается вызовом метода **myClockDomain.deassertReset()**. Формирование тактовых сигналов происходит путем вызова метода **myClockDomain.waitRisingEdge()**, который как бы «дожидается» появления переднего фронта тактового сигнала, но на самом деле он же его и формирует. Собственно, в этом же цикле мы и должны анализировать состояние сигналов и выводить интересующие нас моменты в лог.

Разберемся, какие сигналы мы будем анализировать и выводить. Прежде всего нас интересует комплексный сигнал **sramCtrl.io.bus** и его составные части: **.cmd.valid**,

.cmd.ready, **.cmd.write**, **.cmd.address**, **.cmd.mask**, **.cmd.data**, **.rsp.valid** и **.rsp.data** (см. главу «*17.2.1 Разрабатываем контроллер SRAM*»). Если заглянуть в файл **Murax.scala**, то можно обнаружить, что компонент **sramCtrl** (и все его сигналы) инкапсулирован в область тактирования **system**. Поэтому глобально доступное имя интересующих нас сигналов будет иметь вид: **system.sramCtrl.io.bus.cmd.valid**, **system.sramCtrl.io.bus.cmd.ready** и т. д.

Помимо сигналов отлаживаемого компонента, нам было бы не плохо понимать какую машинную инструкцию в данный момент выполняет ядро и какой адрес сейчас загружен в счетчик команд PC. В компоненте **VexRiscv**, реализующий вычислительное ядро, имеются сигналы **lastStageInstruction** и **lastStagePc**, содержащие интересующие нас данные. В СнК **Murax** компонент **VexRiscv** инкапсулирован в ту же область тактирования **system** и поименован как **cpu**. Поэтому адресовать интересующие нас сигналы можно аналогичным образом: **system.cpu.lastStagePc** и **system.cpu.lastStageInstruction**.

Теперь разберемся, как получить доступ к сигналам из испытательного стенда и как их распечатать в лог. Чтобы сигналы компонента были доступны внутри стенда, их нужно «опубликовать» вызовом метода **.simPublic()** у каждого из сигналов. Выше мы создали переменную **dut** типа **Murax** в качестве испытуемого компонента. Сошлемся относительно неё и опубликуем все интересующие нас сигналы следующим образом:

```
    // More DUT preparations should be put here
    dut.system.cpu.lastStageInstruction.simPublic()
    dut.system.cpu.lastStagePc.simPublic()
    dut.system.sramCtrl.io.bus.cmd.valid.simPublic()
    dut.system.sramCtrl.io.bus.cmd.write.simPublic()
    dut.system.sramCtrl.io.bus.cmd.mask.simPublic()
    dut.system.sramCtrl.io.bus.cmd.ready.simPublic()
    dut.system.sramCtrl.io.bus.cmd.address.simPublic()
    dut.system.sramCtrl.io.bus.cmd.data.simPublic()
    dut.system.sramCtrl.io.bus.rsp.valid.simPublic()
    dut.system.sramCtrl.io.bus.rsp.data.simPublic()

    dut

  }.doSim {
```

Чтобы распечатать состояния сигналов, будем использовать стандартную функцию **println()** с форматировщиком **.format()**. При выводе каждый сигнал (переменную) требуется преобразовать к нужному типу: к числу, к булевому значению или к строке символов. Вывод в лог интересующих нас сигналов будет выглядеть так:

```
            println("[cycle: %8d, pc: %08x, instr: %08x] dut.system.sramCtrl: cmd.valid =
%s, cmd.ready = %s, cmd.write = %s, cmd.mask
= %08x, cmd.address = %08x, cmd.data = %08x, rsp.valid = %s, rsp.data = %08x".format(
              idx,
              dut.system.cpu.lastStagePc.toLong,
              dut.system.cpu.lastStageInstruction.toLong,
              dut.system.sramCtrl.io.bus.cmd.valid.toBoolean,
              dut.system.sramCtrl.io.bus.cmd.ready.toBoolean,
              dut.system.sramCtrl.io.bus.cmd.write.toBoolean,
              dut.system.sramCtrl.io.bus.cmd.mask.toLong,
              dut.system.sramCtrl.io.bus.cmd.address.toLong,
              dut.system.sramCtrl.io.bus.cmd.data.toLong,
              dut.system.sramCtrl.io.bus.rsp.valid.toBoolean,
              dut.system.sramCtrl.io.bus.rsp.data.toLong
            ))
```

Этот код должен располагаться внутри цикла, после «ожидания» переднего фронта тактового сигнала.

Собственно, этого уже достаточно, чтобы запустить симуляцию и получить результат. Но, анализировать 10000 строк лога не очень интересно, поэтому имеет смысл внести в код условие, ограничивающее вывод только интересующего нас момента. Напомню, что нас интересует процесс считывания 32-х битного слова из SRAM. Чтобы ограничить вывод, введем условие по адресу ячейки памяти, к которой осуществляется доступ. Напомню, что область SRAM находится в диапазоне **0x90000000 — 0x900400ff**.

```
if(!dut.system.sramCtrl.io.bus.cmd.write.toBoolean &&
   dut.system.sramCtrl.io.bus.cmd.address.toLong >= 0x90000000l &&
   dut.system.sramCtrl.io.bus.cmd.address.toLong <= 0x900400ffl) {

    println("...".format(
      …
    ))
}
```

Условие вида **!dut.system.sramCtrl.io.bus.cmd.write.toBoolean** проверяет, что в текущем такте выполняется операция чтения (т. е. «**не запись»**).

Добавляем весь приведенный код в самый конец файла **Murax.scala** и готовимся запустить симуляцию.

Перед запуском симуляции нам нужно сделать кое-что еще — требуется подправить исходный код Си программы «hello_world» так, чтобы в первых циклах работы программы сразу был доступ к области SRAM на чтение. Иначе, наша симуляция из 10000 тактов завершится быстрее, чем программа попытается выполнить интересующее нас действие и в логе мы ничего полезного не обнаружим. Добавим следующий код в файл **main.c**:

```
uart_config.clockDivider = SYSTEM_CLOCK_HZ / UART_BAUD_RATE / rxSamplePerBit - 1;
uart_applyConfig(UART, &uart_config);

char a[8];
a[0] = *(char*)(0x90000000 + 3);
a[1] = 0;
println(a);

if(sram_test_write_random_ints() == 0) {
```

То есть считаем **один байт** из ячейки **0x90000003**. Здесь я умышленно ставлю адрес не кратный степени двойки чтобы проверить как это выглядит на стороне разрабатываемого компонента, работает ли это вообще или произойдет программное исключение. Также нас интересует, какие биты будут установлены в сигнале маски **.io.bus.cmd.mask**.

Компилируем Си программу командой **make clean && make** из каталога **./src/main/c/murax/hello_world:**

```
rz@devbox:~/VexRiscv/src/main/c/murax/hello_world$ make clean && make
...
Memory region         Used Size  Region Size  %age Used
             RAM:       21104 B        96 KB     21.47%
/opt/riscv//bin/riscv64-unknown-elf-objcopy -O ihex build/hello_world.elf
build/hello_world.hex
/opt/riscv//bin/riscv64-unknown-elf-objdump -S -d build/hello_world.elf >
build/hello_world.asm
/opt/riscv//bin/riscv64-unknown-elf-objcopy -O verilog build/hello_world.elf
build/hello_world.v
```

Переходим в каталог **./scripts/Murax/Karnix** и запускаем симуляцию следующей командой:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ (cd ../../../; sbt "runMain
vexriscv.demo.Murax_karnix_Sim")
```

Если сборка прошла успешно и проект откомпилировался для запуска в Verilator-е, мы увидим следующие сообщения:

```
[info] welcome to sbt 1.6.0 (Ubuntu Java 11.0.9.1)
[info] loading settings for project vexriscv-build from plugins.sbt ...
[info] loading project definition from /home/rz/VexRiscv/project
[info] loading settings for project root from build.sbt ...
[info] set current project to VexRiscv (in build file:/home/rz/VexRiscv/)
[info] running (fork) vexriscv.demo.Murax_karnix_Sim
[info] [Runtime] SpinalHDL v1.10.1    git head : 2527c7c6b0fb0f95e5e1a5722a0be732b364ce43
[info] [Runtime] JVM max memory : 8294.0MiB
[info] [Runtime] Current date : 2024.03.17 00:27:40
[info] [Progress] at 0.000 : Elaborate components
[info] [Warning] This VexRiscv configuration is set without software ebreak instruction
support. Some software may rely on it (ex: Rust). (This isn't related to JTAG ebreak)
[info] [Warning] This VexRiscv configuration is set without illegal instruction catch
support. Some software may rely on it (ex: Rust)
[info] [Progress] at 2.478 : Checks and transforms
[info] [Progress] at 3.453 : Generate Verilog
[info] [Warning] toplevel/system_cpu/RegFilePlugin_regFile : Mem[32*32 bits].readAsync can
only be write first into Verilog
[info] [Warning] toplevel/system_cpu/RegFilePlugin_regFile : Mem[32*32 bits].readAsync can
only be write first into Verilog
[info] [Warning] 195 signals were pruned. You can call printPruned on the backend report to
get more informations.
[info] [Done] at 4.903
[info] [Progress] Simulation workspace in /home/rz/VexRiscv/./simWorkspace/Murax
[info] [Progress] Verilator compilation started
[info] [info] Found cached verilator binaries
[info] [Progress] Verilator compilation done in 1373.702 ms
[info] [Progress] Start Murax test simulation with seed 923932021
[info] [cycle:     4885, pc: 80001bac, instr: 0037c783] dut.system.sramCtrl: cmd.valid =
true, cmd.ready = false, cmd.write = false, cmd.mask = 00000008, cmd.address = 90000003,
cmd.data = 88888888, rsp.valid = false, rsp.data = 00000000
[info] [cycle:     4886, pc: 80001bac, instr: 0037c783] dut.system.sramCtrl: cmd.valid =
true, cmd.ready = false, cmd.write = false, cmd.mask = 00000008, cmd.address = 90000003,
cmd.data = 88888888, rsp.valid = false, rsp.data = 00000000
[info] [cycle:     4887, pc: 80001bac, instr: 0037c783] dut.system.sramCtrl: cmd.valid =
true, cmd.ready = false, cmd.write = false, cmd.mask = 00000008, cmd.address = 90000003,
cmd.data = 88888888, rsp.valid = false, rsp.data = 00000000
[info] [cycle:     4888, pc: 80001bac, instr: 0037c783] dut.system.sramCtrl: cmd.valid =
true, cmd.ready = true, cmd.write = false, cmd.mask = 00000008, cmd.address = 90000003,
cmd.data = 88888888, rsp.valid = false, rsp.data = 00000000
[info] [Done] Simulation done in 1645.485 ms
[success] Total time: 14 s, completed Mar 17, 2024, 12:27:49 AM
```

Последние четыре строки, выделенные жирным шрифтом — это и есть вывод функции **println()** в процессе симуляции. Здесь мы видим, что операция чтения из ячейки с адресом **0x90000003** была выполнена за четыре машинных такта — так отработал компонент **PipelinedMemoryBusSram**.

Из вывода видно, что операцию чтения запросила машинная инструкция с кодом **0x0037c783** расположенная в основной памяти по адресу **0x80001bac**. Давайте заглянем в файл **build/hello_world.asm** содержащий ассемблерный код программы «hello_world» и посмотрим, что это за инструкция. Ниже приведен фрагмент этого файла:

```
        a[0] = *(char*)(0x90000000 + 3);
80001ba8:       900007b7                lui     a5,0x90000
80001bac:       0037c783                lbu     a5,3(a5) # 90000003
<_ram_heap_end+0xffe8003>
```

Жирный шрифтом выделена именно эта инструкция: **lbu a5,3(a5)** — «load byte unsigned», она читает один байт (без знака) из ячейки памяти со смещением 3 байта относительно адреса, содержащегося в регистре **a5,** и помещает значение в этот же регистр **a5**.

Из лога симуляции также видно, что при чтении сигнал масок был установлен в значение **0x00000008**, то есть установлен бит в 3-ем разряде, что сигнализирует компоненту о необходимости (и достаточности) читать только 3-й байт. Текущая имплементация компонента **PipelinedMemoryBusSram** игнорирует значения масок при чтении, извлекает из памяти и выдает все 32 бита данных, что занимает целых четыре такта. Очевидно, что при байтовых операциях такое поведение компонента является избыточным и операцию можно сократить до двух тактов. Желающие улучить работу компонента **PipelinedMemoryBusSram** могут присылать мне PR на Github-е или патчи по электронной почте.

В результат симуляции всей системы также будет записан в файл **./simWorkspace/Murax/test/wave.vcd** , визуализировать который можно утилитой GTKWave следующим образом:

```
rz@devbox:~/VexRiscv/scripts/Murax/Karnix$ gtkwave ../../../simWorkspace/Murax/test/wave.vcd
```

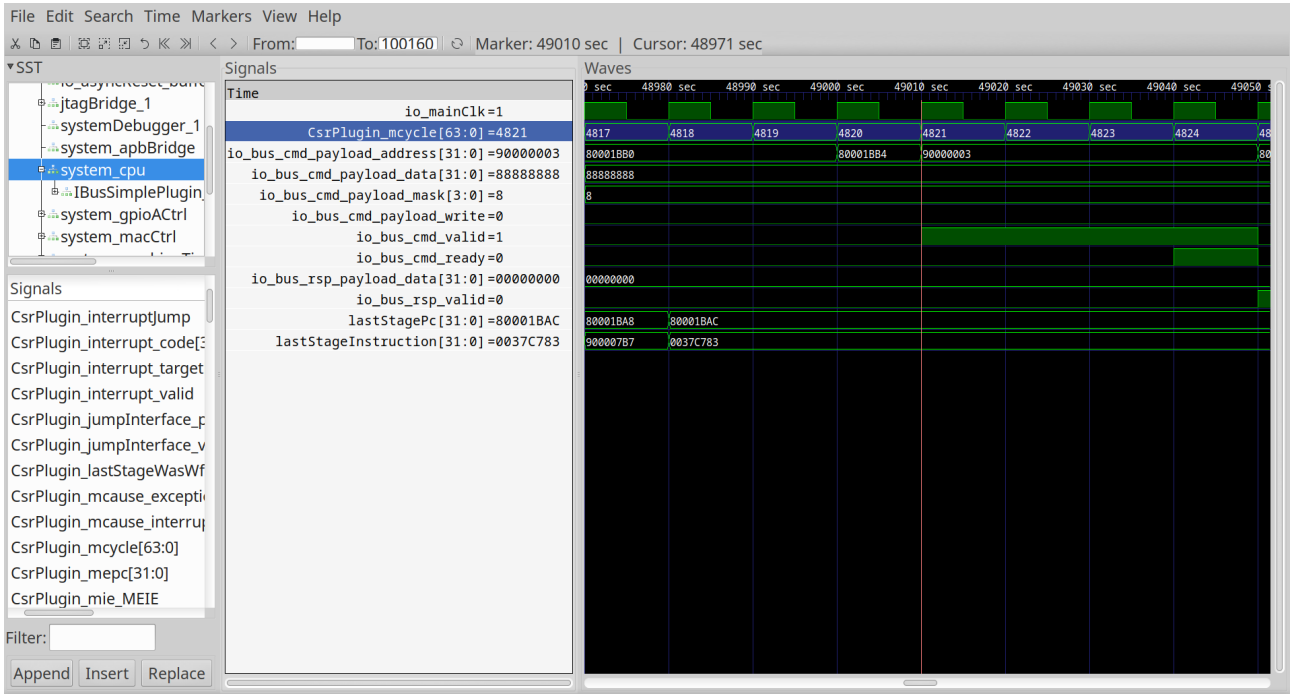


Рис. 36. Визуализация сигналов в виде временных диаграмм в GTKWave.

Разница в **(4885 - 4821) = 64** машинных цикла объясняется тем, что ядро VexRiscv при старте требует некоторое количество машинных тактов на инициализацию.

19. Более сложный пример: VexRiscvWithHUB12ForKarnix

В качестве заключения мне хотелось бы продемонстрировать более сложный пример синтезируемой системы-на-кристалле на базе всё того-же Murax, которая вполне годится для использования в промышленных решениях. Это отдельный проект, который я назвал **VexRiscvWithHUB12ForKarnix**. Данный проект был специально разработан для взаимодействия со светодиодными матрицами формата HUB12 и HUB75e по единому набору сигнальных линий. Светодиодные матрицы этого формата оснащены 16-ти пиновым штыревым разъемом IDC-16 для передачи данных в сдвиговые регистры. Эти регистры требуется непрерывно обновлять для динамического формирования изображения на матрице. Китайская промышленность выпускает большое количество различных светодиодных матриц формата «HUB» с различным набором светодиодов, сдвиговых регистров, с несовместимым форматом загрузки и несовместимым набором сигналов. Для того, чтобы взаимодействовать с такими матрицами, пользователю приходится приобретать специализированные контроллеры с большим набором разъемов — по одному на каждый тип матрицы или даже разные контроллеры. Обойти проблему несовместимости различных типов матриц я решил путем реализации синтезируемого в ПЛИС компонента — контроллера HUB интерфейсов. Разработанный мной контроллер HUB, в зависимости от выбранного типа подключенной матрицы перенастраивает сигнальные линии из общего набора (16 линий GPIO) и реализует требуемый протокол загрузки сдвиговых регистров. Контроллер получает данные из сформированного в набортной RAM фреймбуфера и непрерывно, с максимально возможной скоростью (опыт показал, что максимальная частота тактирования сдвиговых регистров для используемой нами матрицы составляет ~20 МГц), загружает их в матрицу. Пользовательскому ПО остается только формировать изображение в памяти фреймбуфера, имеющего линейную структуру. Иными словами, вычислительное ядро свободно и не задействовано в формировании изображения на матрице. Проект (и изделие на его основе) в несколько ином виде имел коммерческое применение.

Новогоднее видео, ссылка на которое приведена в самом начале статьи, является демонстрацией работы VexRiscvWithHUB12ForKarnix. Все исходные коды и материалы для сборки и повторения установки, показанной на видео доступны из репозитория по ссылке:

https://github.com/Fabmicro-LLC/VexRiscvWithHUB12ForKarnix.git

В проекте VexRiscvWithHUB12ForKarnix реализованы следующие синтезируемые аппаратные блоки (компоненты):

- Поддержка SRAM формата 16x256K.

  Файл: **./src/main/scala/spinal/lib/mem/Sram.scala**

- Контроллер прерываний (PLIC0).

  Файл: **./src/main/scala/mylib/MicroPLIC.scala**

- Два последовательных порта (UART0 и UART1) с поддержкой прерываний на TX и RX.

  Файл: **./src/main/scala/spinal/lib/com/uart/Apb3UartCtrl.scala**

- Два 32-ти битных таймера (TMER0 и TIMER1) с 32-х битным прескейлером и поддержкой прерываний по окончанию счета.

Файл: **./src/main/scala/mylib/Apb3Timer.scala**

- 32-х битный ШИМ таймер (PWM0).

  Файл: **./src/main/scala/spinal/lib/bus/simple/PWM.scala**

- Сторожевой таймер (WD0).

  Файл: **./src/main/scala/mylib/Apb3WatchDog.scala**

- Машинный 1мкс таймер (MTIME0) доступный через CSR.

  Файл: **./src/main/scala/mylib/MachineTimer.scala**

- Порт дискретных линий ввода/вывода на 32 сигнала (GPIO0).

  Файл: **${SpinalHDL}./lib/src/main/scala/spinal/lib/io/Gpio.scala**

- Контроллер шины I2C (I2C0).

  Файл: **./src/main/scala/mylib/MicroI2C.scala**

- SPI интерфейс для управления многоканальным ЦАП в составе платы «Карно» (AUDIODAC0).

  Файл: **./src/main/scala/spinal/lib/com/spi/Apb3SpiMasterCtrl.scala**

- Контроллера светодиодных матриц стандарта HUB12 и HUB75e (HUB0) которые подключаются к плате «Карно» через 16 линий GPIO.

  Файл: **./src/main/scala/mylib/HUB.scala**

- Контроллера FastEthernet (MAC0);

  Файл: **./src/main/scala/mylib/Apb3MacEthCtrl.scala**

В проекте VexRiscvWithHUB12ForKarnix реализованы следующие программные возможности:

- Тестирование SRAM и использование блока SRAM для формирования «кучи».
- Взаимодействие с микросхемой EEPROM по шине I2C для сохранения настроек (в том числе IP адреса, типа подключенной матрицы и её разрешение).
- Драйвер MAC контроллера.
- Поддержка стека протоколов TCP/IP на базе библиотеки **lwIP** (Light Weight IP). Полный стек протоколов, включая ARP, IP, ICMP, UDP, TCP, HTTP и DHCP, требует около 45КБ ОЗУ. Сокращенный вариант стека в составе ARP, IP, ICMP, UDP и DHCP умещаются в 25КБ ОЗУ.
- Поддержка протокола Modbus/RTU (через последовательный корт) и Modbus/UDP (поверх IP).
- Работа со светодиодными матрицами стандарта HUB12 и HUB75e через аппаратный видео фреймбуфер.
- Драйвер с циклическим буфером для проигрывания звука на ЦАП через аппаратный компонент AUDIODAC0.

- В протокол Modbus добавлены расширения для записи блока данных в аудио ЦАП и видео во фреймбуфер HUB контроллера.

При синтезе данная система потребляет следующие ресурсы:

```
Info: Device utilisation:
Info:             TRELLIS_IO:   100/  197    50%
Info:                   DCCA:     3/   56     5%
Info:                 DP16KD:    55/   56    98%
Info:             MULT18X18D:     4/   28    14%
Info:                 ALU54B:     0/   14     0%
Info:                EHXPLLL:     1/    2    50%
Info:                EXTREFB:     0/    1     0%
Info:                   DCUA:     0/    1     0%
Info:              PCSCLKDIV:     0/    2     0%
Info:                IOLOGIC:     0/  128     0%
Info:               SIOLOGIC:     0/   69     0%
Info:                    GSR:     0/    1     0%
Info:                 JTAGG:     0/    1     0%
Info:                  OSCG:     0/    1     0%
Info:                 SEDGA:     0/    1     0%
Info:                   DTR:     0/    1     0%
Info:               USRMCLK:     0/    1     0%
Info:               CLKDIVF:     0/    4     0%
Info:              ECLKSYNCB:     0/   10     0%
Info:                DLLDELD:     0/    8     0%
Info:                 DDRDLL:     0/    4     0%
Info:                DQSBUFM:     0/    8     0%
Info:        TRELLIS_ECLKBUF:     0/    8     0%
Info:           ECLKBRIDGECS:     0/    2     0%
Info:                   DCSC:     0/    2     0%
Info:             TRELLIS_FF:  3690/24288    15%
Info:           TRELLIS_COMB: 16880/24288    69%
Info:           TRELLIS_RAMW:  1064/ 3036    35%

Info: Max frequency for clock '$glbnet$io_lan_rxclk$TRELLIS_IO_IN': 115.37 MHz (PASS at
25.00 MHz)
Info: Max frequency for clock '$glbnet$io_lan_txclk$TRELLIS_IO_IN': 100.35 MHz (PASS at
25.00 MHz)
Info: Max frequency for clock             '$glbnet$core_pll_CLKOP': 82.24 MHz (PASS at 75.00
MHz)
```

Результат тестирования показал, что максимальная частота тактирования ядра, позволяющая стабильно работать при полной нагрузке, составляет **60 МГц** для микросхем 6-го «грейда» и **80 МГц** для микросхем 8-го «грейда».

В качестве демонстрации всей мощи полученной системы, используя вышеописанные возможности на плате «Карно», мною было реализовано стриминговое проигрывание видео ролика «Bad apple» с выводом изображения на светодиодную матрицу разрешением 80x40 пикселов (интерфейс HUB75e) и проигрыванием звука на один из ЦАП, распаянных на плате. Схема собранной установки для проигрывания показана на рис. 37.

Процесс проигрывания известного видеоролика выглядел следующим образом:

1. Видеоролик был выкачан с Youtube, пережат утилитой FFMPEG в разрешение 80x40 и записан в локальный файл в формате BGR8. Звуковой ряд был сохранен в отдельный файл в формате PCM_S16LE и частотой дискретизации 16кГц.
2. В имеющийся Perl скрипт для управления матрицами по протоколу Modbus было добавлено расширение, позволяющее считывать видео и аудио, разбивать на небольшие пакеты и отправлять их по Modbus/UDP на плату «Карно».

3. На стороне «Карно», из принимаемых Modbus/UDP пакетов изымались данные и записывались либо в видео фреймбуфер, либо в циклический аудиобуфер.
4. Скрипт в ответ от «Карно» получает по Modbus/UDP значение отражающее текущее заполнение циклического аудиобуфера (количество свободного места в нём).
5. Основываясь на степени заполненности аудиобуфера, скрипт регулирует интенсивность отправляемых в сторону «Карно» данных путем системного вызова sleep(), не допуская переполнения буфера и его полного опорожнения.

Рис.37. Плата «Карно» с загруженным проектом VexRiscvWithHUB12ForKarnix принимает из сети пакеты Modbus/UDP, изымает из них аудио и видео данные и проигрывает их на имеющейся аппаратуре.

Все материалы для желающих повторить проигрывание видео таким затейливым способом на плате «Карно» или поработать с матрицами формата HUB, находятся в подкаталоге **./scripts/Perl** репозитория VexRiscvWithHUB12ForKarnix. Там же располагается README файл с инструкцией по запуску скрипта.

20. Эпилог

В процессе работы с микросхемами ПЛИС я все больше прихожу к выводу, что современные микроконтроллеры неудобны в использовании, и если бы не существенная разница в цене, то микроконтроллеры уже давно прозябали бы на свалке истории. Одна из традиционных сфер применения МК — это системы с экстремально низким потреблением электроэнергии или устройства с батарейным питанием. Но и в эту вотчину уже зашли ПЛИС. Например, известна серия CPLD микросхем Lattice MachXO2 с ультранизким потреблением, встроенной перепрограммируемой постоянной памятью и встроенным осциллятором. Микросхемы этой серии просты, но имеют достаточное количество ресурсов для синтезирования вычислительной системы с некоторым количеством адаптированной под задачу аппаратуры. Такие микросхемы позволяют создавать «кастомный» микроконтроллер всего лишь на одной микросхеме с низким уровнем потребления тока (единицы мА) и питанием от 1.2В.

Кто-то может возразить: «а как же ЦАП и АЦП?». Традиционно эти блоки стыковки с аналоговой аппаратурой присутствуют во всех МК и отсутствуют в недорогих микросхемах

ПЛИС. Да, ПЛИС — чисто цифровые устройства, но и тут не все однозначно. ЦАП легко реализуется на ПЛИС с помощью ШИМ и несложного НЧ фильтра на выходе. А что же с АЦП? Недавно я узнал про «All-Digital ADC» или **A2D** — способ организации АЦП посредством только цифровой аппаратуры внутри ПЛИС (или ASIC), используя один дифференциальный сигнал LVDS (две линии) и небольшой пассивной обвязки. Принцип простой — вход LVDS обычно реализуется в виде компаратора, который сравнивает напряжения двух линий и формирует логический «0» или «1» внутри ПЛИС. Если подвести входной аналоговый сигнал к одной из линий LVDS (положительной), а к второй (отрицательной) подключить конденсатор и управлять его зарядом с помощью ШИМ, то компаратор внутри ПЛИС будет сравнивать уровень заряда конденсатора с уровнем входного аналогового сигнала. Несложная схема обработки с «оверсэмплингом» дает полностью цифровой АЦП. На рис. 38 показана схема устройства такого АЦП.

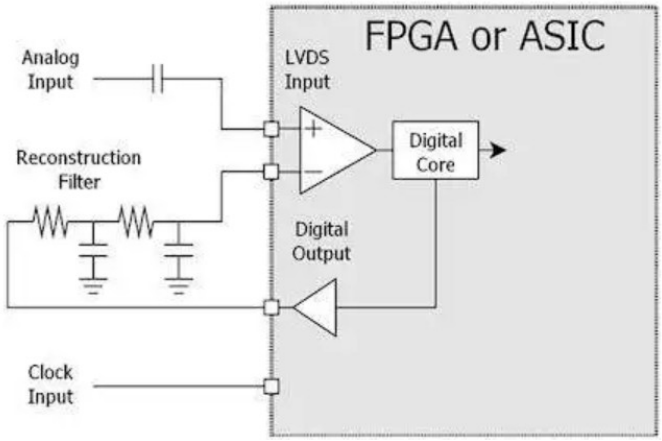


Рис. 38. Схема полностью цифрового АЦП (All-Digital ADC).

Возьмусь предположить, что со временем, чем дешевле будет становиться «кремний», тем дальше микросхемы ПЛИС будут проникать в нашу повседневную жизнь и тем больше вытеснять микроконтроллеры из оборота. Единственное, что сильно удерживает МК на «плаву», это устоявшиеся привычки разработчиков и инертность мышления, а также накопленная база ПО и инструментария.

На рынке встроенных систем долгое время доминировали МК двух архитектур — AVR и ARM (про PIC и 8051 как-то давно позабылось). Но в последние годы в эту область наметилось серьезное проникновение открытой архитектуры RISC-V, что ознаменовалось выходом микроконтроллеров серии GD32 и ESP32-C3. Отечественные производители также отметились своими изделиями на базе 32-битного RISC-V — полностью отечественный МК MIK32 производства зеленоградского АО «Микрон» заслуживает внимания. Но как мы увидели, 32-битные ядра RISC-V отлично синтезируются в ПЛИС и показывают весьма неплохую производительность. Для сравнения, тот же MIK32 имеет тактовую частоту 32 МГц, что в два раза ниже предложенного мной варианта на базе синтезируемого VexRiscv. А по цене MIK32 выше, чем стоимость микросхемы ПЛИС на нашей плате «Карно», при том, что ПЛИС — это гораздо больше, чем готовый микроконтроллер!

Всем дочитавшим эту статью до конца — огромное спасибо! И простите за то, что получилось так много.

PS: Плату «Карно» можно изготовить самостоятельно — Gerber файлы, BOM и схема в формате PDF присутствуют в репозитории. Или приобрести с ozon.ru уже собранную и протестированную.

Благодарности

Юрию Панчулу (@YuriPanchul) и команде "Школа цифрового синтеза" за отличный обучающий проект **basics-graphics-music**.

Дмитрию (@DmitryZlobec) за идеи по плате "Карно".

Виктору Сергееву за сборку серии плат "Карно" на страшном и ужасно глюкавом Autotronik BA385V2.

Евгению Короленко за прогон примеров на плате "Карно", вычитку статьи и исправление ошибок.

Ссылки на репозитории

Плата «Карно» (KiCAD): https://github.com/Fabmicro-LLC/Karnix_ASB-254
Проект «basics-graphics-music»: https://github.com/yuri-panchul/basics-graphics-music
YosysHQ: https://github.com/YosysHQ/yosys
Yosys Manual: https://yosyshq.readthedocs.io/projects/yosys/en/latest/index.html#yosys-manual
SpinalHDL: https://github.com/SpinalHDL/SpinalHDL
SpinalHDL Docs: https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html
SpinalTemplateSbtForKarnix: https://github.com/Fabmicro-LLC/SpinalTemplateSbtForKarnix
VexRiscv: https://github.com/SpinalHDL/VexRiscv
VexRiscvWithKarnix: https://github.com/Fabmicro-LLC/VexRiscvWithKarnix
VexRiscvWithHUB12ForKarnix: https://github.com/Fabmicro-LLC/VexRiscvWithHUB12ForKarnix