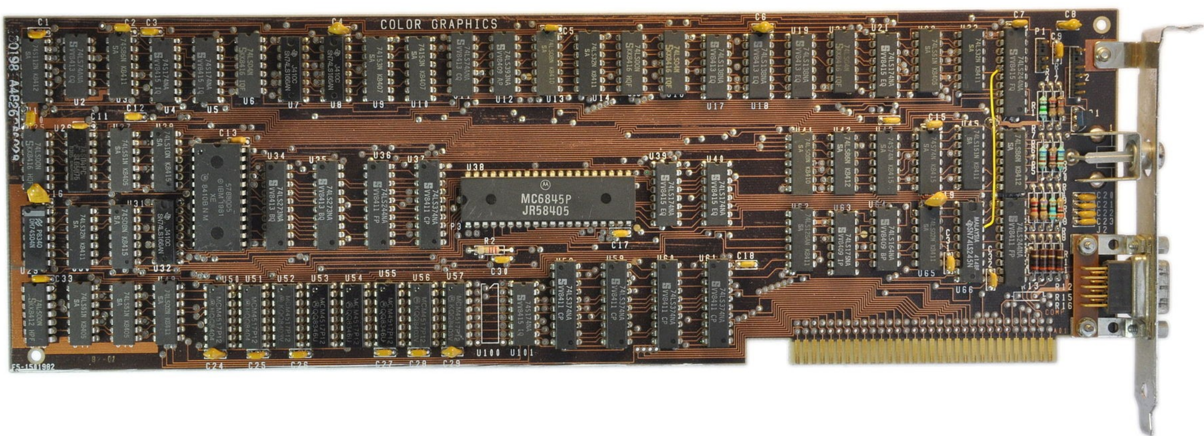
Разработка цифровой аппаратуры нетрадиционным методом: CGA видеоадаптер на SpinalHDL

В предыдущей статье я рассказывал как построить свою полностью открытую вычислительную систему на опенсорсных решениях — некую синтезируемую систему-на-кристалле, которая будет адаптирована под ваши задачи, в которой всё до последнего триггера находится под вашим управлением и не зависит ни от рыночной конъюнктуры, ни от политических решений. В этой статье я расскажу и покажу как добавить к этой СнК простейший видеоадаптер под тип старого доброго Color Graphics Adapter (CGA), но с современным (HDMI) интерфейсом, с графическим и текстовым режимами и аппаратным скроллингом для плавной прокрутки изображений. Как и в предыдущей статье, речь пойдет о проектировании аппаратуры на языке SpinalHDL и синтезе её для микросхем ПЛИС, используя опенсорсный тул. Все эксперименты я буду проводить на плате «Карно» с ПЛИС Lattice серии ECP5, но весь мой код, за исключением части касающейся настроек PLL, будет аппаратно независимым и легко адаптируемым под любой тип микросхем ПЛИС из поддерживаемых тулчейном Yosys/NextPNR.



IBM Color/Graphics Adapter образца 1981 г.

СОДЕРЖАНИЕ

1. Зачем всё это ?....................3
2. High-Definition Media Interface....................4
3. Электрические сигналы в разъеме HDMI....................5
4. TMDS кодирование сигнала в DVI-D и HDMI(R)....................7
5. Видеоформаты и видео тайминги....................14
6. Синтезируем изображение на экране монитора....................18
6.1. Выбор видеоформата....................19
6.2. Создаем интерфейсный класс HDMIInterface....................20
6.3. Создаем класс основного компонента KarnixTestHDMITopLevel....................22
6.4. Разбираемся с тактовыми сигналами....................23
6.5. Формируем «муар»....................25
6.6. Подключаем TMDS энкодеры....................27
6.7. Собираем и запускаем KarnixTestHDMI на ПЛИС....................29
7. Color Graphics Adapter....................31
7.1. Устройство CGA адаптера....................31
7.2. Видео тайминги и видеоформаты CGA адаптера....................36
7.3. Адаптируем CGA под современные реалии....................37
8. Разработка CGA подобного видеоадаптера....................40
8.1. Графический режим, он самый простой....................47
8.2. Интеграция CGA адаптера в синтезируемую СнК....................49
8.3. Простейший код на Си для тестирования графического режима....................51
8.4. Измеряем скорости записи в видеопамять....................56
8.5. Программная реализация функция Bitblit и замер её производительности....................61
9. Добавляем текстовый режим....................69
9.1. Знакогенератор....................69
9.2. Отображение текста....................72
9.3. Тестируем текстовый режим....................73
9.4. Мигающий курсор....................75
9.5. Плавная вертикальная прокрутка (скроллинг) изображения....................81
10. Специальные эффекты....................85
10.1. Высокочастотное смешивание цветов....................85
10.2. Динамическая перезагрузка палитры в процессе отрисовки....................88
10.3. Совмещение двух видеорежимов — текстового и графического....................92
10.4. Вишенка на торте....................97
11. Заключение....................99
12. Что дальше ?....................99
13. Ссылки....................100

1. Зачем всё это ?

Сразу попытаюсь ответить на резонный вопрос: «нафига козе баян и в чем профит?»

В предыдущей статье я рассказывал как построить свой СнК микроконтроллерного типа на базе софтверного ядра VexRiscv с системой команд RV32IM (32 бит RISC-V), с собственной конфигурацией аппаратуры для использования в промышленной и домашней автоматизации, и для всего того, что сейчас принято называть «IoT». Представьте, что Вы собрались сделать свой прибор для мониторинга температуры в котельной в вашем загородном доме. Есть много решений этой несложной задачи, большинство из которых - проприетарно-облачные, но мы таких решений не приемлем, так ведь? ;) Самым простым самобытным решением тут видится использование Arduino Nano или чего-то аналогичного с несложной программой на Си. И таких решений на просторах Github-а тоже предостаточно. Но что если Вам захотелось выводить параметры датчиков или отобразить график изменения температуры на большой экран или на телевизор в гостиной комнате, ну чтобы похвастаться перед гостями вашим уникальным решением. В этом случае Вам потребуется одноплатная ЭВМ с аппаратными возможностями не хуже чем Raspberry Pi, с HDMI или VGA интерфейсом, и с ОС Linux на борту. В целом, такое решение пригодно, но во-первых, не является дешевым. Во-вторых — не очень-то надежное, так как MMC/MicroSD карты, с которых обычно производится загрузка этих «одноплатников», при бросках напряжения мрут как мухи. В-третьих, в случае сбоя питания выход на рабочий режим занимает достаточно продолжительное время по причине долгой загрузки ОС Linux. Ну и в-четвертых, Вам придется заморочиться с тяжеловесным программированием под Linux и X11, попутно изучив пару-тройку фреймворков, что для такой простой задачи как отображение трех цифр или пары графиков может оказаться слишком трудозатратным. Или вот другой например, табло с маршрутами на автобусной остановке. Та же RPi тут уже плохо подходит, так как не пригодна для работы в условиях низких температур и чтобы её использовать приходится городить огород. Или пульт управления каким нибудь технологическим оборудованием в АСУТП (т. н. HMI). Тут нужна и работа при больших перепадах температур, и быстрый выход на рабочий режим при сбое, и надежность требуется слегка повыше чем у MicroSD карты. Короче, микроконтроллерное устройство способное выводить изображение на HDMI монитор или телевизор легко решит такие задачи и стоимость такого решения будет вполне приемлемой.

Но сам я решил заняться прикручиванием монитора к плате «Карно» из несколько других побуждений — мне стало интересно, на сколько это сложно сформировать видеосигнал по интерфейсу HDMI не используя готовых и дорогостоящих микросхем-энкодеров, то есть самостоятельно закодировать видеосигнал средствами ПЛИС да так, чтобы любой современный телевизор смог бы его раскодировать и нормально отобразить. Иными словами, занялся я этим вопросом из академического интереса.

2. High-Definition Media Interface

Несмотря на присутствие практически в любой современной аудио, видео и компьютерной аппаратуре, стандарт High-Definition Multimedia Interface (HDMI) является проприетарным и почти полностью закрытым. Доступ к документам его регламентирующим ограничен в рамках участников консорциума «HDMI Adopters» основателями которого являются известные нам производители аудио/видео аппаратуры - Sony, Panasonic, Hitachi, Thompson, Toshiba, Philips и Silicon Image. Википедия пишет, что сейчас в консорциум входит более 1700 участников, участие в консорциуме является платным и стоит немалых денег. Я не юрист, но в сети есть разъяснения насчет того, что любой производитель реализующий стандарт HDMI (R) обязуется отчислять «роялти» в пользу HDMI Licensing, LLC с каждого проданного чипа или устройства поддерживающего этот стандарт, независимо о того, является ли он участником консорциума или скачал спецификацию с торрентов.

Стандарт HDMI(R) позволяет передавать цифровой аудио и видео сигнал в одном направлении - от источника (source) к приемнику (sink), а также обмениваться служебной информацией между источником и приемником в обоих направлениях. Цифровое аудио может передаваться как в сжатом (compressed), так и обычном (uncompressed) формате, видео передается не сжатым. Еще одной (неприятной) особенностью HDMI является то, что передаваемые данные, видео и аудио, могут быть зашифрованы с помощью приватных ключей, используя разработанную компанией Intel технологию High-bandwidth Digital Content Protection (HDCP), основной целью которой является защита от копирования контента при передаче от источника (например, от DVD плейера) к приемнику (видеопроектору). Чтобы получить доступ к ключам, необходимо приобретать дополнительную лицензию от Digital Content Protection LLC, регулярно производить лицензионные отчисления и следовать ряду строгих ограничений. И несмотря на то, что эти электронные ключи давно «утекли» и общедоступны, Intel грозится засудить всех и каждого, кто посмеет произвести совместимое с HDCP устройство.

Но не всё так страшно. Стандарт HDMI (R) появился в начале 2000-х, в эпоху бурного перехода от аналогового (PAL/NTSC и VGA) видео к цифровому и для обеспечения совместимости с устройствами предыдущего поколения в спецификацию HDMI внесена обязательная поддержка DVI, а точнее DVI-D — его цифровой версии. Для тех, кто не застал это мимолетное видение, поясню что DVI это открытый (и на данный момент устаревший) стандарт интерфейса для подключения видеомониторов, проекторов и телевизоров к ПК появившийся в самом конце 1990-х годов. Его целью было заменить полностью аналоговый VGA интерфейс на цифровой вариант. Если в VGA интерфейсе (тут имеется в виду сам разъем D-sub DE-15) сигналы компонентов цветности «red», «green» и «blue» передавались по трем парам проводов и кодировались уровнями напряжений, то в DVI-D эти же три компонента передаются уже в виде последовательности бит по трем дифференциальным TMDS парам, по одной на компонент цвета, и кодируются цифровым методом. Еще одна дифференциальная TMDS пара используется для передачи тактового сигнала следования пикселей (pixclk), чего в VGA никогда не было. От VGA в стандарте DVI была унаследована сетка частот и разрешений VESA, хотя и сильно расширена. Минимальным и всеми поддерживаемым является разрешение 640x480 @ 60 Гц с частотой pixclk = 25.175 МГц (более подробно частотную сетку мы обсудим чуть ниже). Помимо этого, в DVI был введен еще один канал передачи цифровых данных — Display Data Channel (DDC) который предназначен для обмена конфигурационной (E-DID) и прочей служебной информацией между источником и приемником. Для DDC в качестве протокола транспортного уровня используется I^2C подобный протокол с двумя сигнальными линиями — DDC clock и DDC data.

Так чем же это нам может помочь? А тем, что стандарт HDMI с электрической точки зрения это DVI-D, более того, HDMI обеспечивает определенный уровень совместимости с DVI-D и на транспортном уровне. Да, у HDMI есть свои методы кодирования отличные и не совместимые с DVI-D, есть возможность передавать данные с двойной скоростью (DDR), есть цифровой звук и возможность шифровать контент, но всё это нам и не требуется. Ну разве что кроме звука, но это совершенно отдельная история. Иными словами мы можем смотреть на наш видео интерфейс как на вариант исполнения DVI-D упакованный в HDMI разъем Type A.

На Reddit-е есть пост с бурным обсуждением вопроса: насколько легальным является использование в своих изделиях HDMI разъема для передачи DVI-D сигнала и при этом никому не отстёгивая. Все участники дискуссии сходятся во мнении, что так как метод кодирования данных по DVI-D принадлежит организации DDWG Promoters Group, которая не имеет ничего общего с «HDMI Adopters» и в свою очередь является открытым стандартом, то и заносить в HDMI Licensing, LLC ничего не требуется. Это если не использовать на изделии логотип HDMI и не утверждать о наличии совместимости с этим стандартом, разумеется.

Но хватит беллетристики, приступим к делу. Как Вы уже поняли, реализовывать мы будем кодирование видеосигнала по стандарту DVI-D. Любое упоминание HDMI далее в тексте статьи будет подразумевать коннектор вида изображенного на рис.1 (спереди), и не более того.

Рис. 1. Пассивный переходник с DVI-D (сзади) на HDMI (спереди).

3. Электрические сигналы в разъеме HDMI

У интерфейса DVI имеется свой вид разъемов, причем нескольких типов, отличающихся по количеству задействованных контактов: DVI-I и DVI-D, каждый в варианте «Single Link» и «Dual Link». Интерфейс DVI-I содержит помимо цифровых еще и аналоговые линии сигналов цветности R, G и B. Модификация «Single Link» содержит только один

комплект сигналов передачи данных (три дифференциальных линий TMDS Data[2:0]) и работает на частотах до 165 МГц позволяя передавать изображение формата 1920x1200@60, в то время как модификация «Dual Link» имеет два комплекта таких же сигналов (тактовый сигнала TMDS Clock все так же один) работающих параллельно на той же частоте и позволяющих передавать изображение формата 2560x1600@60. Разъем DVI содержит сигналы DDC clock и DDC data для обмена информацией о поддерживаемых форматах (E-DID) и настройках, а также линии питания +5В и «земли».

Из всего этого многообразия в интерфейс HDMI перекочевало только три дифференциальных линии TMDS Data[2:0]+ / TMDS Data[2:0]-, дифференциальная линия тактирования TMDS Clock+ / TMDS Clock-, сигналы DDC clock / DDC data и линия +5В. HDMI интерфейс был расширен дополнительным сигналом Consumer Electronics Control (CEC) — последовательная однопроводная двунаправленная шина для передачи управляющих сигналов на HDMI устройства принимаемых приемником видеосигнала (sink) с пульта дистанционного управления, а также был добавлен сигнал Hot Plug Detect (HPD) для определения подключения и отключения интерфейса «на горячую». Таким образом разъем HDMI Type A версии 1.0 получил набор сигналов приведенных в таблице 1. На рис. 2 приведен ход нумерации контактов в этом разъеме.

Таблица 1. Наименование и расположение сигналов в разъеме HDMI Type A версии 1.0-1.3a

№ контакта	Наименование сигнала	№ контакта	Наименование сигнала
1	TMDS Data2+	2	TMDS Data2 Shield
3	TMDS Data2−	4	TMDS Data1+
5	TMDS Data1 Shield	6	TMDS Data1−
7	TMDS Data0+	8	TMDS Data0 Shield
9	TMDS Data0−	10	TMDS Clock+
11	TMDS Clock Shield	12	TMDS Clock−
13	CEC	14	Reserved
15	SCL (I2C serial clock for DDC)	16	SDA (I2C serial data for DDC)
17	Ground	18	+5 V (up to 50 mA)
19	Hot Plug Detect (HPD)	SHIELD	GND

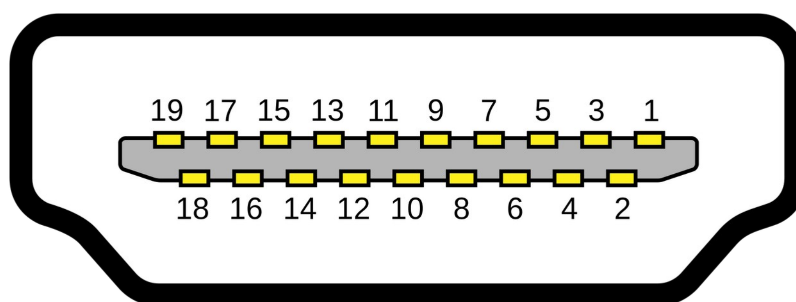


Рис. 2. Нумерация контактов в разъем HDMI Type A.

Сигнал HPD, как правило, жестко формируется схемой подключения путем включения резистора подтяжки номиналом 10K на «землю» на стороне источника и резистором подтяжки 5.1K к линии +5В на стороне приемника видеосигнала. При отсутствии

физического соединения источник будет получать лог «0» по данной линии, а при подключении приемника - лог «1» (+3.3В).

Сигнал CEC представляет линию типа «открытый коллектор», формат передачи данных по которой чем-то походит на I2C, но специфичен для HDMI. Данные передаются фреймами, каждый из которых снабжен адресом размерностью 4 бита, что позволяет адресовать до 15 отдельных устройств (адрес 15 — широковещательный).

Сигналы шины DDC, как уже отмечалось выше, представляют собой обычную шину I2C работающую на скорости 100 кбит/сек и в основном использующуюся для считывания с приемника видеосигнала информации о поддерживаемых форматах (блок E-DID или E-EDID), более подробно об этом можно прочитать в соответствующей статье на сайте Википедии. В HDMI(R) шина DDC также активно используется для реализации шифрования по протоколу HDCP.

На плате «Карно» сигналы HPD, CEC и шины DDC заведены в микросхему ПЛИС и могут быть использованы по желанию пользователя. В рамках данной статьи мы не будем их использовать, так как они не участвуют в формировании видеосигнала.

4. TMDS кодирование сигнала в DVI-D и HDMI(R)

В DVI-D данные о цветности пикселей передаются по трем дифференциальным парам/линиям, которые называются Transition Minimized Differential Signaling (TMDS). Каждая отвечает за передачу данных об одном из трех цветовых компонентов пикселя: линия TMDS_data[0] - для синего компонента, TMDS_data[1] — для зеленого и TMDS_data[2] — для красного компонента соответственно. Данные по линиям TMDS в стандарте DVI-D передаются последовательно с частотой в 10 раз выше частоты следования пикселей, которая передается отдельной диффпарой TMDS_clock. В стандарте HDMI(R) частота передачи данных отличается от DVI-D и равна пятикратной частоте TMDS_clock (в 5 раз выше), так как используется технология Double Data Rate — данные защелкиваются по переднему фронту и по спаду тактового сигнала. Структурная схема TMDS линка приведена на рис. 3а, она одинакова для DIV-D и для HDMI(R).

С электрической точки зрения TMDS это два провода, обычно свитых между собой для помехозащищенности, передача «нулей» и «единиц» по которым осуществляется изменением направления движения электрического тока в замкнутой цепи линии. Такой метод передачи данных называется Current Mode Logic, а в русскоязычной литературе его часто называют «токовой петлей». TMDS линия на стороне источника и приемника обычно подтянута к линиям питания +3,3В и терминирована на резистор номиналом 50 Ом. Такой способ передачи данных (с помощью тока) позволяет увеличить помехозащищенность линии и снизить потребляемую мощность расходуемую на передачу. Несмотря на то, что обычно при передаче данных методом «токовой петли» предполагается наличие полной связи по постоянному току (DC coupled), во всех современных высокоскоростных стандартах передачи данных принято развязывать линии по постоянному току, т. е. в линиях присутствуют последовательно включенные конденсаторы. Делается это в том числе и для увеличения надежности линии. Наличие развязки, в свою очередь, приводит к тому, что по линии нельзя передавать длинные последовательности «единиц» или «нулей», так как длительное прикладывание постоянного потенциала может полностью разрядить (или наоборот — зарядить до максимума) развязывающие конденсаторы и через них перестанет протекать электрический ток. Чтобы избавиться от такого эффекта, в TMDS, как и в других высокоскоростных линиях, применяется кодирование данных, позволяющее исключить длинные постоянные последовательности (пять и более последовательных «единиц» или «нулей»), а разница между количеством «единиц» и «нулей» за все время не может быть

больше или меньше двух. Таким образом в линии удерживается баланс «нулей» и «единиц», линия становится нейтральной по постоянному току и требует меньшей частотной полосы (bandwidth) для передачи на высоких скоростях.

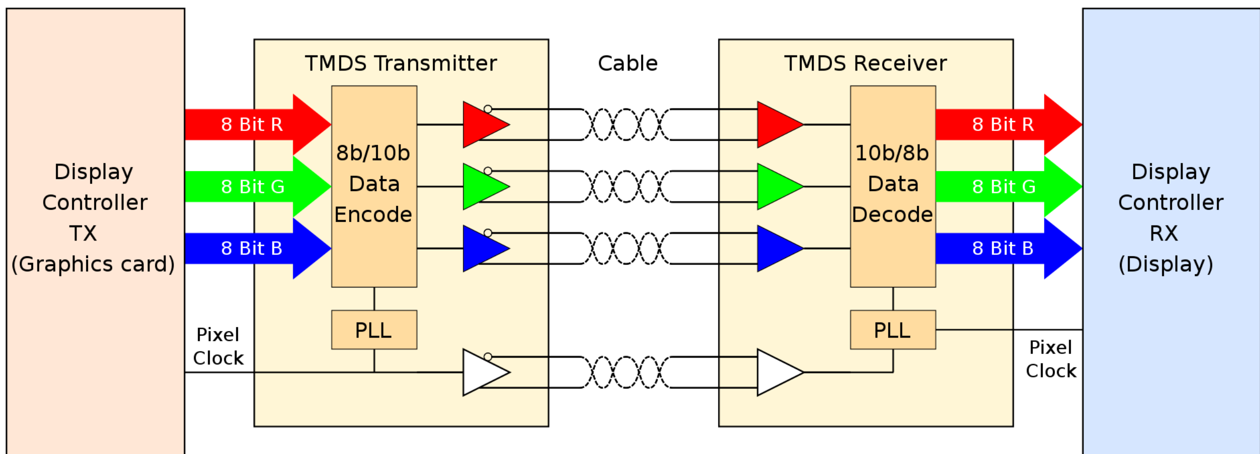


Рис. 3а. Схема TMDS линка для передачи данных о цветности (RGB), синхронизации и аудио (только в HDMI).

Обычно в серьезных изделиях для формирования токового интерфейса TMDS применяют специальные микросхемы «TMDS трансиверов», но часто бывает так, что на этих трансиверах экономят и подключают комплиментарные линии TMDS напрямую к CMOS линиям микросхемы, формирующей видеосигнал. На рис. 3б приведена выдержка из схемы формирования TMDS сигнала платы «Карно» иллюстрирующая именно такой удешевленный вариант включения. О том почему и как это становится возможным я опишу чуть далее.

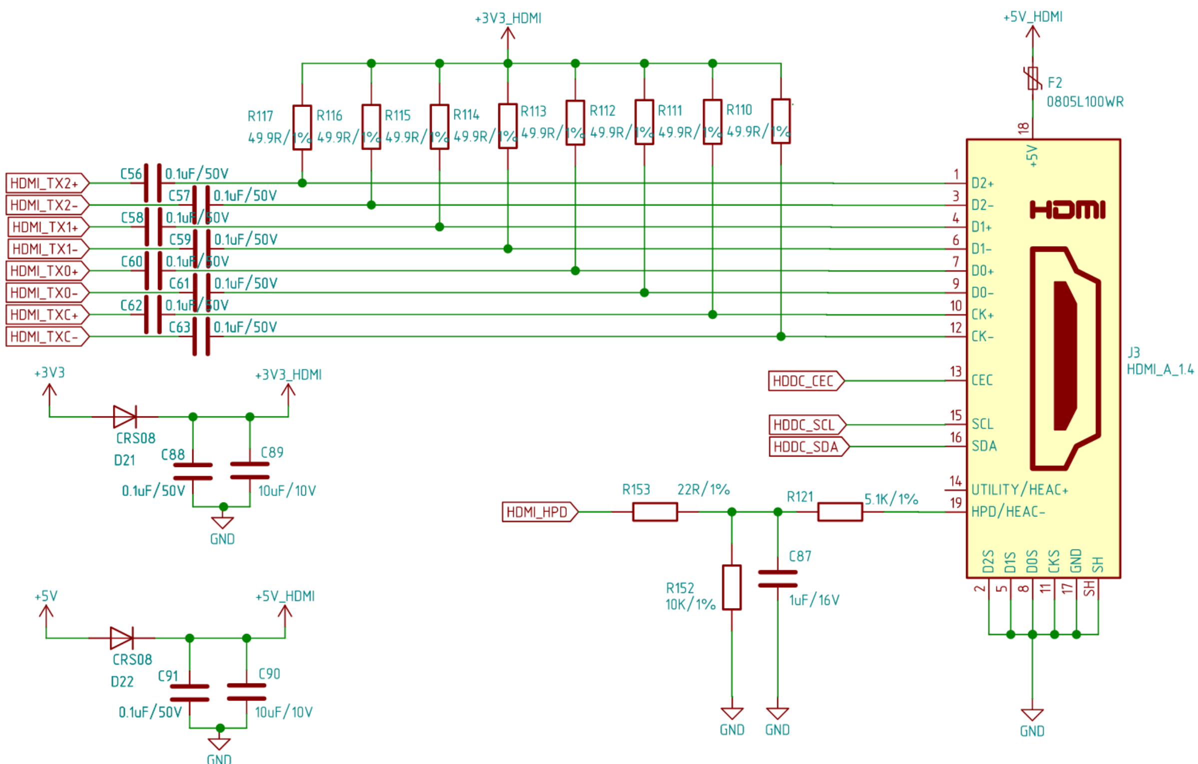


Рис. 3б. Выдержка из схемы электрической принципиальной HDMI интерфейса на плате «Карно».

При передаче по линиям TMDS используется два варианта кодирования данных: «8b/10b» для передачи данных о цветности, и «2b/10b» для передачи двух служебных битов данных называемых **c0** и **c1**. То есть каждые 8 бит информации о цвете преобразуются в 10 бит, которые последовательно передаются в линию. Аналогично обстоит дело со служебными битами. Выбор между тем, какой вариант кодирования будет использован, зависит от того, что именно передается в данный момент времени — видео-данные или какие-то служебные данные.

Важное замечание! В TMDS используется свой вариант кодирования «8b/10b», существенно отличающийся от широко используемого метода «8b/10b encoding» предложенного фирмой IBM в 1983г., но так как оба они имеют одинаковые названия, то это часто вводит в заблуждение как разработчиков, так и авторов многих публикаций! Метод предложенный IBM, помимо баланса по постоянному току, позволяет решить еще несколько важных задач: во-первых, по кодированному потоку данных на стороне приемника можно легко восстановить частоту тактирования (clock recovery), во-вторых, данные передаются с избытком, а значит более устойчивы к ошибкам. Но используемый в TMDS метод кодирования не позволяет восстанавливать частоту тактирования и, предположительно, по этой причине в TMDS тактовый сигнал передается отдельной дифференциальной парой. Стоит заметить, что стандартный IBM-овский метод «8b/10b» кодирования используется в PCI-Express, в DisplayPort, в Ethernet и во многих других современных высокоскоростных интерфейсах, почему разработчики DVI-D придумали свой «велосипед» — остается только догадываться.

В стандарте DVI-D служебные биты представляют состояние сигналов строчной (HSYNC) и кадровой (VSYNC) развертки, передача которых осуществляется в «голубом» (TMDS_data0) канале. Служебные биты информации, передаваемые в двух других каналах в рамках стандарта DVI-D не используются и там обычно передаются нули. В стандарте HDMI(R) два других канала используются для передачи служебных битов, индицирующих признак «островка данных», указывающего на то, что далее в полезных битах, вместо цветности, передается цифровой звук или «auxilary data».

В спецификации к одной из микросхем HDMI энкодеров мне попалась наглядная схема (рис. 4) демонстрирующая последовательность различных периодов с данными, передаваемых по TMDS в стандарте HDMI(R) для формата кадра 720x480. Всего используется три вида периодов передаваемой по линиям TMDS: темно-серым цветом обозначен период следования видео-данных - это момент когда передаются цветности пикселей, светло-серым обозначены периоды управляющих данных - пиксели не передаются, и голубым - период «островка с данными» - идет передача звука или данных пользователя. Как видно из схемы, всего для каждого кадра передается 525 строк, из них видимых только 480, а 45 строк являются «теневыми» и служат для передачи служебных данных или звука. В каждой строке передается 858 «символов», из которых 720 несут информацию о цветности, а остальные 138 — тоже «теневые». Тип периода определяется состоянием служебных битов c0 и c1 в трех каналах.

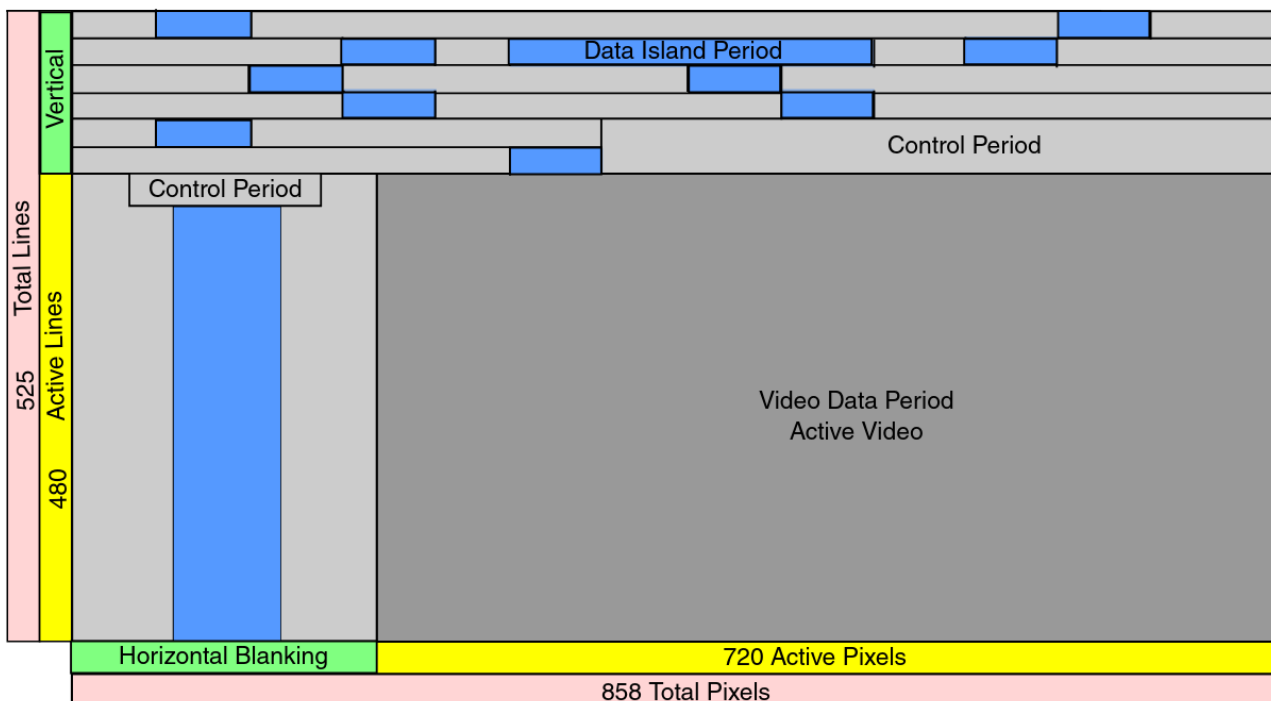


Рис. 4. Периоды следования данных в TMDS для кадра разрешением 720x480.

Но вернемся к DVI-D. Таким образом, для того, чтобы формировать видеосигнал нам необходимо научиться кодировать данные по стандарту TMDS в формате «8b/10b» для видеоизображения и «2b/10b» для сигналов HSYNC/VSYNC. На просторах Google Docs имеется статья от безымянного автора, озаглавленная «Understanding HDMI & TMDS Encoding», в которой подробно расписаны эти алгоритмы кодирования. Я не будут пересказывать всю статью, пройдусь лишь кратко по основным моментам:

1. Кодирование осуществляется отдельно и параллельно для каждого из трех каналов TMDS_data0, TMDS_data1 и TMDS_data2.

2. Кодирование служебных сигналов **c0** и **c1** осуществляется простой таблицей — при необходимости передать состояние битов выбирается и передается один из четырех 10-ти битных «символов»:

c0	c1	bits
0	0	10’b0010101011
0	1	10’b0010101010
1	0	10’b1101010100
1	1	10’b1101010101

Рис. 5. Кодирование служебных битов данных в TMDS (2b/10b).

3. Кодирование видео-данных производится по следующему алгоритму:

3.1. Исходя из числа «единиц» в передаваемом байте, выбирается один из методов «XOR encoding» или «XNOR encoding»:
3.1.1. если количество единиц менее 4-х, то используется «XOR encoding»;
3.1.2. если количество единиц более 4-х, то используется «XNOR encoding»;
3.1.3. если единиц ровно 4, то:
3.1.3.1. если нулевой бит данных == 1, то используется «XOR encoding»;
3.1.3.2. если нулевой бит данных == 0, то используется «NOR encoding».

3.2. К входным данным применяется один из выбранных методов:
3.2.1. «XOR encoding»:

```
out_data[0] := in_data[0];
out_data[1] := in_data[1] XOR out_data[0];
out_data[2] := in_data[2] XOR out_data[1];
...
out_data[7] := in_data[7] XOR out_data[6];
```

3.2.2. «XNOR encoding»:

```
out_data[0] := in_data[0];
out_data[1] := in_data[1] XNOR out_data[0];
out_data[2] := in_data[2] XNOR out_data[1];
...
out_data[7] := in_data[7] XNOR out_data[6];
```

3.3. Добавляется 9-й бит out_data[8], равный «1» если используется «XOR encoding» или «0» если «XNOR encoding».

3.4. Приведенный выше алгоритм не гарантирует сохранение DC баланса, что необходимо для работы схемы. Для этого постоянно вычисляется текущее значение баланса и если оно превышает +2 или становится ниже -2, то девять бит ранее полученных данных инвертируются.

3.5. Добавляется десятый бит out_data[9] указывающий на то, было ли выполнено инвертирование или нет.

Хорошее описание алгоритмов кодирования и разбор прочих внутренностей HDMI/DVI/TMDS, созданный по мотивам безымянного автора, изложил Jeremy See в своей статье «[Tutorial 6: HDMI Display Output]()».

Ниже приведен один из вариантов [реализации TMDS энкодера]() на языке Verilog, позаимствованный с сайта fpga4fun.com, ссылку на который прислал мне пользователь Хабра @DmitryZlobec, что и подтолкнуло меня заняться исследованием этой темы.

```
module TMDS_encoder(
        input clk,
        input [7:0] VD,  // video data (red, green or blue)
        input [1:0] CD,  // control data: C0, C1
        input VDE,       // video data enable, to choose between CD (when VDE=0) and VD (when VDE=1)
        output reg [9:0] TMDS = 0
);

wire [3:0] Nb1s = VD[0] + VD[1] + VD[2] + VD[3] + VD[4] + VD[5] + VD[6] + VD[7];
wire XNOR = (Nb1s>4'd4) || (Nb1s==4'd4 && VD[0]==1'b0);
wire [8:0] q_m = {~XNOR, q_m[6:0] ^ VD[7:1] ^ {7{XNOR}}, VD[0]};

reg [3:0] balance_acc = 0;
wire [3:0] balance = q_m[0] + q_m[1] + q_m[2] + q_m[3] + q_m[4] + q_m[5] + q_m[6] + q_m[7] - 4'd4;
wire balance_sign_eq = (balance[3] == balance_acc[3]);
wire invert_q_m = (balance==0 || balance_acc==0) ? ~q_m[8] : balance_sign_eq;
wire [3:0] balance_acc_inc = balance - ({q_m[8] ^ ~balance_sign_eq} & ~(balance==0 ||
balance_acc==0));
wire [3:0] balance_acc_new = invert_q_m ? balance_acc-balance_acc_inc : balance_acc+balance_acc_inc;
wire [9:0] TMDS_data = {invert_q_m, q_m[8], q_m[7:0] ^ {8{invert_q_m}}};
wire [9:0] TMDS_code = CD[1] ? (CD[0] ? 10'b1010101011 : 10'b0101010100) : (CD[0] ? 10'b0010101011 :
10'b1101010100);

always @(posedge clk) TMDS <= VDE ? TMDS_data : TMDS_code;
always @(posedge clk) balance_acc <= VDE ? balance_acc_new : 4'h0;

endmodule
```

Забегая немного вперед скажу, что я попытался переписать этот энкодер TMDS на языке SpinalHDL, но честно говоря результат получался «не очень». Во-первых, изображение получалось с искажениями и я никак не мог найти ошибку, а во-вторых код вышел сильно многословный и в какой-то момент я просто запутался в нем и вообще начал подозревать, что используемая в плате «Карно» схема включения HDMI не рабочая. Поэтому, чтобы не заниматься «поиском черной кошки в темной комнате» я решил далее использовать приведенный выше код энкодера на языке Verilog как «черный ящик», для чего подготовил обертку на SpinalHDL, но об этом чуть позже. Наверное, сейчас, с более глубоким пониманием происходящих процессов, мне стоит попробовать переписать код энкодера на SpinalHDL еще раз, чисто из спортивного интереса.

Но это еще не всё. Осталось выяснить, как из дискретного сигнала «нулей» и «единиц», формируемого модулем TMDS_encoder, сделать дифференциальный сигнал «токовой петли». Оказалось, что это достаточно просто — достаточно формировать два комплиментарных CMOS сигнала, обозначим их TMDS_data+ «положительный» и TMDS_data- «отрицательный». Когда нам необходимо передать логическую «1», то мы будет подтягивать TMDS_data+ к линии питания +3,3В, то есть устанавливать его в лог «1», а TMDS_data- к «земле» (GND). И наоборот, если нужно передать по дифф паре лог «0», то меняем значения на выходах комплиментарных сигналов на противоположные. Так как в линиях HDMI интерфейса на печатной плате у нас установлены развязывающие конденсаторы, то смена лог «1» на лог «0» будет менять полярность потенциала приложенного к этим конденсаторам, они будут то разряжаться, то заряжаться, а значит в цепи дифференциальной пары будет протекать электрический ток - то в одном направлении,

то в другом, в зависимости от того, передаем мы лог «1» или «0». На схеме рис. 3б сигналы HDMI_TX0+, HDMI_TX0-, HDMI_TX1+, HDMI_TX1- и HDMI_TX2+, HDMI_TX2- напрямую подключены к микросхеме ПЛИС на выводы типа LVCMOS33 (+3,3В CMOS) и формируют три «условно дифференциальные» пары TMDS для передачи данных, а сигналы HDMI_TXC+ и HDMI_TXC- формируют пару для передачи тактового сигнала. Вот такой простой фокус!

Для того, чтобы комплиментарные сигналы формировались одновременно, т. е. чтобы уменьшить фазовый шум («jitter»), во многих микросхемах ПЛИС имеется специальный аппаратный блок называемый **OBUFDS**, он принимает на вход один дискретный сигнал, и формирует на выходе два других, комплиментарных или дифференциальных, сигнала. Если в конкретной ПЛИС или в туле поддержки **OBUFDS** не имеется, то можно попытаться симулировать **OBUFDS** следующим простым кодом на Verilog, понадеявшись на то, что плейсер при трассировке сигналов расположит линии **O** и **OB** рядом и у них будет одинаковая длина и задержка распространения:

```
module OBUFDS(
    input I,
    output O,
    output OB
);

    assign O = I;
    assign OB = ~ I;
endmodule
```

5. Видеоформаты и видео тайминги

Перед тем как мы приступим к написанию своего тестового модуля для отображения «чего нибудь», нам еще предстоит выяснить, как формируется изображение на экране монитора, т. е. какие данные необходимо передавать и в какой момент времени, а для этого нам придется вспомнить, как формировалось изображение в VGA, который в свою очередь унаследовал ряд свойств от телевизионных стандартов прошлого — NTSC и PAL.

Стандарт Video Graphics Array (VGA) появился в 1987 году в графической подсистеме IBM PS/2 — то было второе, и не не очень удачное, поколение «IBM PC» (или второй подход IBM к теме создания «Персонального Компьютера»). В то время широко использовались видеомониторы и телевизоры на основе электронно-лучевых трубок (ЭЛТ), в которых пучок электронов (луч) отклоняясь в магнитном поле формируемом системой электромагнитных катушек, последовательно пробегал по всей поверхности экрана покрытого слоем люминофора (материал, попадание на который пучка электронов вызывает свечение) и вызывал кратковременное засвечивание точек, что при частоте кадров 60 или 70 Гц воспринималось человеческим глазом как полноценная картинка. Для того, чтобы перемещать луч в ЭЛТ присутствовало две системы катушек — одна отклоняла пучок электронов в горизонтальном направлении и управлялась системой «горизонтальной развертки», вторая — в вертикальном, соответственно управлялась система «вертикальной развертки». Когда пучок электронов достигал края экрана по горизонтали или по вертикали, требовалось некоторое время для того, чтобы перемагнитить катушки и вернуть луч в начало строки или в начало экрана. В этот момент, называемый периодом «обратного хода луча» («Blanking Time» или «период затенения»), изображение на экране не формировалось, но видеосигнал все равно передавался и содержал ряд управляющих импульсов. Сигналы «горизонтальной развертки» и «вертикальной развертки» (HSYNC и VSYNC) передавались как раз в этот момент и указывали системам развертки на то, что требуется вернуть луч в исходное положение. Помимо HSYNC и VSYNC в аналоговом телевидении NTSC/PAL/SECAM присутствовал ряд других сигналов которые также передавались в момент обратного хода луча, но в VGA, на сколько мне известно, они не перекочевали. Тем не менее, весь принцип формирования изображения в VGA, а далее в DVI и HDMI, полностью заимствован из аналогового телевидения, в том числе идея «периодов затенения» в конце каждой строки и в конце каждого кадра.

Замечание: в северо-американском телевизионном стандарте NTSC дело обстояло еще немного сложнее — горизонтальная развертка производилась через строку (interlaced), а кадровая повторялась два раза со смещением в одну строку по вертикали, а кодирование цветности задавалось изменением фазы сигнала (а не амплитуды как в PAL), но эти детали нам сейчас не интересны.

Как отмечалось выше, видеосигнал в VGA передается аналоговым способом по пяти сигнальным линиям: Red, Green, Blue, HSYNC и VSYNC, этот набор сигналов сокращенно называют RGBHV. Линии R, G и B несут информацию о цветности передаваемой в текущей

момент точки закодированную амплитудой (чем выше — тем больше яркость данного компонента), а сигналы HSYNC и VSYNC являются отдельными «стробами» - кратковременными импульсами, по переднему фронту или спаду (в зависимости от особенностей системы) которых начинается соответствующий период затенения. Частоту с которой передается информация о цветности точек называют «частотой следования пикселей» или «частотой точечной развертки» и часто обозначают как PIXCLK. В VGA частота следования точек интегрирована внутрь аналоговых сигналов RGB и восстанавливается специальной схемой на принимающем устройстве (телевизоре, видеомониторе, видеопроекторе). Эта частота строго зависит от формата передаваемого кадра и частоты кадровой развертки. И здесь мы вплотную подходим к такой теме как «видеоформаты» и «видео тайминги».

Изначально в VGA использовалось небольшое количество видеоформатов, определяемых по количеству отображаемых (видимых на экране) точек и частоте кадровой развертки: 640x480@60 Гц (VGA), 800x600@60 Гц (SVGA), 1024x768@60 Гц (XGA), и 1280×1024@85 Гц (SXGA) — в скобках дано устоявшееся за данным форматом название. Позже этот список был существенно расширен, часть форматов определяются открытыми стандартами VESA и ANSI/CTA-861, часть является проприетарными и присутствуют только в видеоаппаратуре определенных производителей.

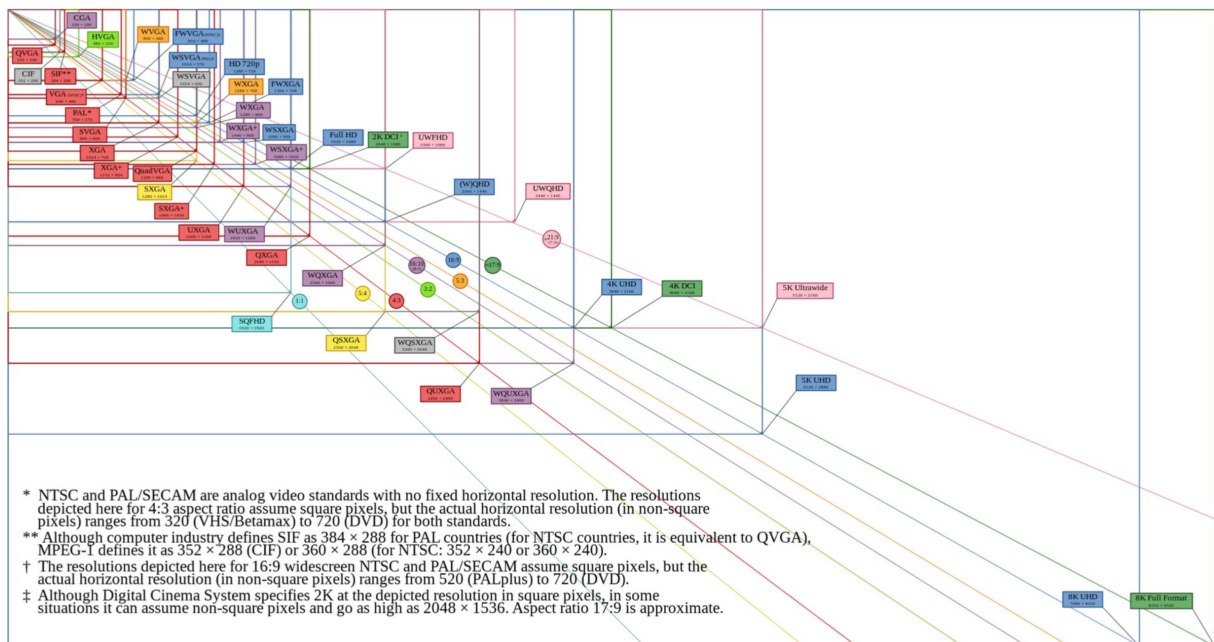


Рис. 5. Различные форматы кадра от 320x200 (CGA) до 8192x4320 (8K Full Format) и их соотношение.

Видимая часть видеоформата (видео кадра) это еще не всё. Как мы обсуждали выше, существует большое количество «темных» точек, которые передаются в период затенения вместе с управляющими сигналами. Далее я буду использовать термин «период затенения», так как термин «период обратного хода луча» потерял всякий смысл, ибо электронного луча давно с нами нет, но вот его мнимый след остался. Казалось бы этот рудимент стоило ликвидировать, но в процессе разработки своего видеоадаптера я понял, что идея «затенения» очень удобна и широко используется не только для передачи служебных/управляющих сигналов (или цифрового звука и пользовательских данных в HDMI[(R)]), но она делает аппаратуру кодирования/декодирования немного проще и позволяет

избегать различных «глитчей» - артефактов изображения вызванных задержками загрузки данных в регистры и задержками отрисовки изображения в видеопамять. К «глитчам» мы еще вернемся, а сейчас рассмотрим как же распределяются «видимые» и «темные» точки в процессе формирования кадра.

На рис 6. показаны различные периоды в процессе формирования видеоизображения, весь набор которых принято называть «видео таймингами». Передача данных видеоизображения производится строками которые называют «сканлайны» (scan-line). Первые несколько строк кадра попадают в период называемый «vertical back porch» - в этот период передаются «темные» (не отображаемые) точки. Далее следуют строки попадающие в период «drawing area», содержащиеся в них данные выводятся на экран, но не все, а только те части строк, которые не попадают в периоды «horizontal back porch», «horizontal front porch» и «horizontal sync» - эти периоды внутри каждой строки также содержат «темные» точки. После того как период активной области отрисовки «drawing area» закончился, следует период «verical front porch», а за ним «vertical sync».

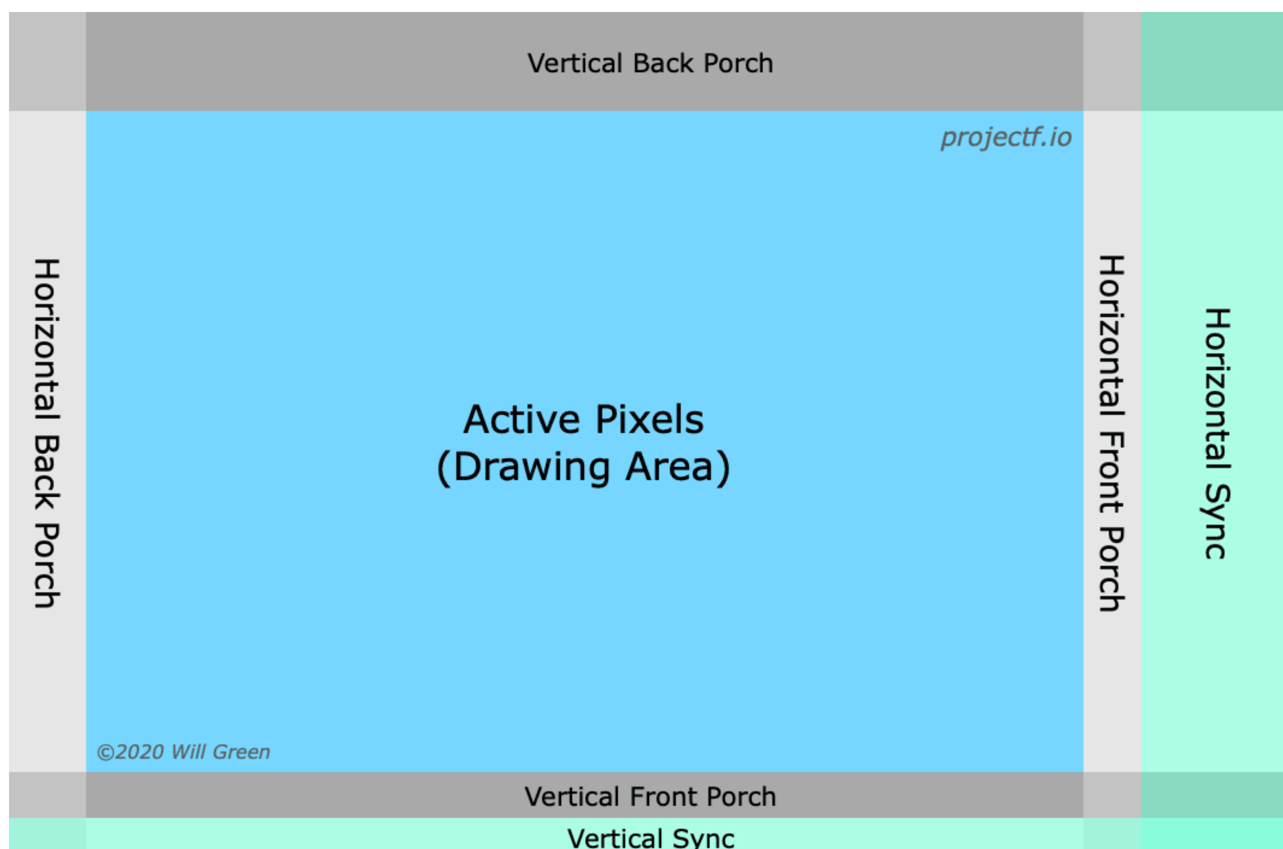


Рис. 6. Видео тайминги (периоды) в процессе формирования видео кадра.

Периоды «horizontal back porch» и «horizontal front porch» часто используются логикой схемы для подгрузки данных в регистры сдвига, так чтобы это было незаметно на экране (т. е. чтобы избежать «глитчей»). Периоды «verical back porch» и «vertical front porch» используются программным обеспечением или цифровой схемой занимающейся выводом изображения в видео память (т. н. «битблитер» от «bit blt» или «bit block transfer»), которые отрабатывают в тот момент, когда изображение остается неизменно на экране, т. е. в момент затенения. Если битблитер не успевает полностью закончить отрисовку, то это визуально проявляется как еще один «глитч» - мерцание и небольшое вертикальное смещение частей изображения, особенно если происходит вывод анимационного видео сигнала (движущихся изображений).

Периоды «horizontal sync» и «vertical sync» это моменты времени когда активны сигналы HSYNC и VSYNC соответственно.

Периоды «horizontal back porch», «horizontal front porch» и «horizontal sync» измеряются в точках (пикселах), а время их продолжительности - это есть число точек поделенное на частоту следования точек PIXCLK.

Периоды «vertical back porch», «vertical front porch» и «verical sync» определяются по числу строк сканирования (сканлайнов). Каждая строка сканирования это сумма длительности всех точек по всем четырем периодам: длительность «horizontal back porch» + длительность «horizontal front porch» + длительность «drawing area» + длительность «horizontal sync».

В некоторых системах формирования видео сигнала периоды затенения «horizontal back porch» и «horizontal front porch» объединяются в один «horizontal porch». Аналогично «vertical back porch» и «vertical front porch» объединяются в один «vertical porch». Делается это в том случае, когда с точки зрения системы не важен момент следования периодов затенения, а важна их общая длительность.

Зная параметры таймингов и частоту следования кадров можно рассчитать частоту следования точек PIXCLK. И на оборот, зная частоту PIXCLK и параметры таймингов можно вычислить частоту кадровой развертки.

Не смотря на то, что такие параметры как «horizontal back/front porch», «horizontal sync», «vertical back/front porch» и «vertical sync» могут изменятся в широких пределах, и видео аппаратура это допускает, они достаточно жестко определены и стандартизированы. Для каждого видеоформата может быть определено более одного варианта видео таймингов и ниже таблице 2 приведены тайминги для некоторых популярных форматов.

Таблица 2. Тайминги некоторых популярных видеоформатов принятых в стандарте VGA.

Format	Pixel Clock (MHz)	Horizontal (in Pixels)				Vertical (in Lines)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
640x480, 60Hz	25.175	640	16	96	48	480	10	2	33
640x480, 72Hz	31.500	640	24	40	128	480	9	3	28
640x480, 75Hz	31.500	640	16	96	48	480	11	2	32
640x480, 85Hz	36.000	640	32	48	112	480	1	3	25
800x600, 56Hz	38.100	800	32	128	128	600	1	4	14
800x600, 60Hz	40.000	800	40	128	88	600	1	4	23
800x600, 72Hz	50.000	800	56	120	64	600	37	6	23
800x600, 75Hz	49.500	800	16	80	160	600	1	2	21
800x600, 85Hz	56.250	800	32	64	152	600	1	3	27
1024x768, 60Hz	65.000	1024	24	136	160	768	3	6	29
1024x768, 70Hz	75.000	1024	24	136	144	768	3	6	29
1024x768, 75Hz	78.750	1024	16	96	176	768	1	3	28
1024x768, 85Hz	94.500	1024	48	96	208	768	1	3	36

* В данной таблице параметры «Front Porch» и «Back Porch» переставлены местами, физически период «Front Porch» всегда следует за видимой частью видео сигнала!!!

В качестве примера можно вспомнить, что задание параметров видеоформата и таймингов в конфигурационных файлах **Xorg** оконной системы X-Window осуществляется строками вида:

```
Modeline "1920x1080x60" 148.50  1920 2008 2052 2200  1080 1084 1089 1125 +hsync
+vsync
```

где после идентификатора формата, приведенного в двойных кавычках, следует число соответствующее частоте PIXCLK (в МГц), далее следуют тайминги выраженные в пикселях и строках в накопительной форме. Общий формат строки Modeline следующий:

```
PixClk Hact Hact+Hfp Hact+Hfp+Hsw Hact+Hfp+Hsw+Hbp Vact Vact+Vfp
Vact+Vfp+Vsw Vact+Vfp+Vsw+Vbp
```

Где:

- Hact - Разрешение видимой части по сканлайна (по горизонтали)
- Hfp - Horizontal front porch
- Hsw - Horizontal synch pulse width
- Hbp - Horizontal back porch
- Vact - Разрешение видимой части по вертикали
- Vfp - Vertical front porch
- Vsw - Vertical synch pulse width
- Vbp - Vertical back porch
- PixClk - частота следования пикселов = (Hact+Hfp+Hsw+Hbp) * (Vact+Vfp+Vsw+Vbp)

Параметры +hsync и +vsync указывают на полярность сигналов HSYNC и VSYNC, где «+» показывается что сигнал воспринимается по переднему фронту, а «-» - по спаду.

Но вернемся к цифровым видео интерфейсам. В цифре период отрисовки изображения принято называть «Data Enable» - т. е. наличие/присутствие данных, в некоторых руководствах его также называют «Display Enable» и даже «Display On». В цифровых видео интерфейсах (например RGB24) этот признак обычно выведен в отдельный сигнал **DE** который устанавливается в логическую «1» при передаче данных о цветности пикселей и в «0» при следовании периода затенения. В DVI-D и HDMI данного сигнала нет, а признак наличия данных передается с помощью кодовых «символов». Если следуют данные изображения, то передаются слова закодированные методом «8b/10b» по описанному в предыдущей главе алгоритму. Если же следует период затенения, то передается один из четырех кодовых «символов» закодированных по «2b/10b» и содержащий состояния управляющих сигналов **c0** и **c1** (HSYNC и VSYNC). Всё остальное остается так же как и в VGA!

6. Синтезируем изображение на экране монитора

Раз у нас имеется понимание необходимых требований, чтобы закодировать и отправить видеосигнал на телевизор или экран монитора по HDMI/DVI-D интерфейсу, логичным шагом будет попробовать что-то отобразить на экране, то есть разработать простой тестовый компонент для проверки работоспособности алгоритма кодирования и вообще всей этой теории. Для простоты реализации такого компонента, вместо отображения битмапа, что может потребовать достаточно много времени и ресурсов (памяти), имеет смысл синтезировать какой-то простой и узнаваемый «муар». Для этого не нужно большого объема памяти, а достаточно пары регистров-счетчиков, пары сдвиговых регистров и немного комбинационной логики.

6.1. Выбор видеоформата

Перед там как синтезировать «муар» нам следует подумать какого формата видео изображение мы будем передавать и с какими таймингами. Основная проблема здесь состоит в том, чтобы правильно выбрать формат поддерживаемый приемником (монитором, телевизором) подключенным в данный момент к HDMI разъему — как Вы догадываетесь, разные устройства поддерживают разный набор форматов и нам следует выбрать тот, который точно поддерживается. Согласно принятым подходам, нам следовало бы осуществить обмен по шине DDC и запросить список поддерживаемых форматов из приемника, выбирать подходящий и использовать его. Но как я уже упоминал, реализовывать DDC в рамках данной статьи мы не будем. Вместо этого подумаем над вопросом - а какой видеоформат мы вообще в состоянии реализовать на данной микросхеме ПЛИС и на данной плате? Во-первых, мы должны исходить из имеющихся возможностей по формированию тактовых частот: частоты PIXCLK и частоты следования битов, которая должна быть в 10 раз больше PIXCLK. Во-вторых, нам необходимо принять во внимание физические возможности микросхемы ПЛИС пропустить через себя данные передаваемые на этих частотах. Будем исходить из того, что у нас на плате «Карно» имеется источник стабильной частоты 25.0 МГц поддерживаемый внешним кварцевым генератором и микросхема ПЛИС которая может «обрабатывать данные» на частоте не более 400 МГц.

Если посмотреть на таблицу 2 со списком «стандартных» видеоформатов которые поддерживаются, как правило, большинством производителей современных видеомониторов и телевизоров, то можно увидеть, что под приведенные выше требования попадают только видеоформаты разрешением 640x480 и 800x600. Да и то не все, а только те, у которых частота PIXCLK менее 40 МГц, а именно: форматы с частотой 25,175 МГц, 31,5 МГц, 36,0 МГц и 38,1 МГц. Теперь нужно подумать, как получить стабильную частоту равную или близкую с точностью до 0,5% - именно такой допуск на отклонение частоты приводится в стандарте DVI. В микросхеме ПЛИС серии ECP5 имеется блок PLL (ФАПЧ) с помощью которого можно преобразовать одну (входную) частоту в другую (выходную) методом умножения и деления. Чтобы получить список возможных частот и соответствующих делителей (параметров настройки PLL), можно воспользоваться утилитой **ecppll**:

```
rz@devbox:~$ ecppll -i 25 -o 25.175
Pll parameters:
Refclk divisor: 1
Feedback divisor: 1
clkout0 divisor: 24
clkout0 frequency: 25 MHz
VCO frequency: 600
```

Параметр **-i** задает входную частоту (в МГц) на блок PLL и в случае с платой «Карно» это всегда будет частота равная 25,0 МГц. Параметр **-o** указывает желаемую частоту на выходе PLL. Утилита рассчитывает самый ближайший возможный вариант конфигурации PLL и отображает настройки блока для получения этой частоты.

Воспользовавшись утилитой **ecppll** я составил Таблицу 3 близких частот для потенциальных кандидатов в видеоформаты и рассчитал погрешность (отклонения) от целевой частоты.

Таблица 3. Перечень возможных частот для PIXCLK получаемых с выхода PLL.

clkin, МГц	Refclk div	FB dev	Clkout dev	PIXCLK, МГц	Target Clk, МГц	Error, %
25,0	1	1	24	25,0	25,175	0,7
25,0	4	5	19	31,25	31,5	0,8
25,0	7	10	17	35,7143	36,0	0,8
25,0	2	3	16	37,5	38,1	1,5

Исходя из минимальной погрешности выбор падает на формат с частотой 25,175 МГц, т. е. 640x480 @ 60 Гц, но реальная частота PIXCLK у нас будет 25,0 МГц. Этот вариант хорош еще и тем, что для получения частоты PIXCLK блок PLL может и не потребоваться, так как достаточно просто взять входную частоту 25,0 МГц с кварцевого генератора. Замечу, что тут мы немного не попадаем в допуски определяемые стандартом, так как погрешность получается немного больше (0,7% вместо 0,5%), но забегая вперед скажу, что я не смог найти ни одного устройства, которое бы отказалось принимать сигнал с PIXCLK = 25,0 МГц и это никак не сказывается на качестве изображения.

Раз уж мы заговорили о настройках PLL, то сразу рассчитаем параметры PLL для получения десятикратной частоты (x10) для формирования битов по линиям TMDS:

```
rz@devbox:~$ ecppll -i 25 -o 250
Pll parameters:
Refclk divisor: 1
Feedback divisor: 10
clkout0 divisor: 2
clkout0 frequency: 250 MHz
VCO frequency: 500
```

6.2. Создаем интерфейсный класс HDMIInterface

С видеоформатом и таймингами определились, настройки PLL рассчитали, пора писать код. Напомню, что наша задача на данном этапе состоит в том, чтобы сформировать на экране монитора любую осмысленную картинку и тем самым убедиться в правильности теории, в возможности её практической реализации на конкретной ПЛИС и в работоспособности интерфейса HDMI на плате «Карно».

На каком языке будем писать код, на Verilog? Или всё таки на SpinalHDL? Предлагаю дальше всё писать на SpinalHDL, а Verilog оставить для низкоуровневых модулей, где требуется выполнить оптимизацию, как например с модулем **TMDS_encoder** код которого приведен в главе 4. Ниже я продемонстрирую как легко подключаются модули из Verilog в SpinalHDL.

Замечание. Далее я буду работать со всё тем же репозиторием **VexRiscvWithKarnix**, а новый код буду помещать в ветку **karnix_extended**. Полный путь к репозиторию: https://github.com/Fabmicro-LLC/VexRiscvWithKarnix/tree/karnix_extended

Начнем с того, что добавим описание сигнальных линий в файле LPF, то есть сделаем привязку интерфейсных сигналов к выводам микросхемы ПЛИС, согласно схеме платы «Карно». Если Вы используете другую плату с аналогичной ПЛИС, то Вам придется самостоятельно изучить схему платы, выяснить на каких номерах выводов ПЛИС располагаются требуемые нам сигналы и внести соответствующие исправления (проставить правильные номера выводов).

Итак, добавляем в файл **karnix_cabga256.lpf** следующие строки с описанием сигналов:

```
LOCATE COMP "io_hdmi_tmds_p[0]" SITE "A11";
IOBUF PORT "io_hdmi_tmds_p[0]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_hdmi_tmds_p[1]" SITE "D11";
IOBUF PORT "io_hdmi_tmds_p[1]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_hdmi_tmds_p[2]" SITE "B11";
IOBUF PORT "io_hdmi_tmds_p[2]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_hdmi_tmds_n[0]" SITE "A12";
IOBUF PORT "io_hdmi_tmds_n[0]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_hdmi_tmds_n[1]" SITE "E11";
IOBUF PORT "io_hdmi_tmds_n[1]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_hdmi_tmds_n[2]" SITE "C11";
IOBUF PORT "io_hdmi_tmds_n[2]" IO_TYPE=LVCMOS33;
LOCATE COMP "io_hdmi_tmds_clk_p" SITE "B12";
IOBUF PORT "io_hdmi_tmds_clk_p" IO_TYPE=LVCMOS33;
LOCATE COMP "io_hdmi_tmds_clk_n" SITE "C12";
IOBUF PORT "io_hdmi_tmds_clk_n" IO_TYPE=LVCMOS33;
```

Далее нам необходимо описать интерфейсный класс со структурой сигнальных линий HDMI/DVI-D которые мы только что добавили в LPF. Сделать это не сложно, его код будет выглядеть вот так:

```
case class HDMIInterface() extends Bundle with IMasterSlave{
        val tmds_p = Bits(3 bits)
        val tmds_n = Bits(3 bits)
        val tmds_clk_p = Bool()
        val tmds_clk_n = Bool()

        override def asMaster(): Unit = {
                out(tmds_p, tmds_n, tmds_clk_p, tmds_clk_n)
        }
}
```

В вышеприведенном коде описывается структура из трех линий **tmds_p[2:0]** и трех линий **tmds_n[2:0]** для положительных и отрицательных линий дифференциальных сигналов используемых для передачи данных. Аналогично описывается сигнал **tmds_clk** — две линии в дифф паре, положительная и отрицательная. Метод **asMaster** будем использовать для того, чтобы установить соответствующие сигнальные линии как выходные. Вообще, синтаксис языка SpinalHDL позволяет сразу задавать направления сигналов, но в интерфейсных классах это лучше делать специальным методом — вдруг когда нибудь наш класс будет задействован в декодере HDMI сигнала, т. е. с противоположными направлениями этих же сигналов.

Поместим этот код в файл **./src/main/scala/mylib/HDMI.scala** вместе с кодом оберток для «черных ящиков» модулей **TMDS_encoder**, **OBUFDS** и **DCCA** написанных на Verilog, добавим импорт стандартных библиотек и получим файл следующего содержания:

```
package mylib

import spinal.core._
import spinal.lib._
import spinal.lib.io.TriState

case class HDMIInterface() extends Bundle with IMasterSlave{
        val tmds_p = Bits(3 bits)
```

```
        val tmds_n = Bits(3 bits)
        val tmds_clk_p = Bool()
        val tmds_clk_n = Bool()

        override def asMaster(): Unit = {
                out(tmds_p, tmds_n, tmds_clk_p, tmds_clk_n)
        }
}

case class TMDS_encoder() extends BlackBox{
        val clk = in Bool()
        val VD = in Bits(8 bits)
        val CD = in Bits(2 bits)
        val VDE = in Bool()
        val TMDS = out Bits(10 bits)
}

case class OBUFDS() extends BlackBox{
        val I = in Bool()
        val O, OB = out Bool()
}

case class DCCA() extends BlackBox{
  val CLKI = in  Bool()
  val CLKO = out  Bool()
  val CE = in  Bool()
}
```

Примечание. Класс **DCCA** это обертка для «черного ящика» с одноименным названием, который используется в ПЛИС серии ECP5 для отправки сигнала по глобальным линиям тактирования. **DCCA** потребуется нам для того, чтобы глобально разрулить сигнал PIXCLK и его 10-ти кратно умноженную копию. Аналогичные механизмы присутствуют во всех ПЛИС, но могут называться по разному, например BUFG (global clock buffer).

6.3. Создаем класс основного компонента KarnixTestHDMITopLevel

Теперь займемся написанием основного компонента высокого уровня для нашего теста, назовем его **KarnixTestHDMITopLevel** и будем помещать код в файл:
./src/main/scala/mylib/KarnixTestHDMI.scala.

Будем исходить из того, что у пользователя могут быть различные видеомониторы и телевизоры с различающимися таймингами, поэтому в заголовке компонента сразу заведем все параметры для таймингов и весь код будем писать параметрически, т. е. используя эти параметры вместо констант. Это в дальнейшем позволит легко портировать готовый компонент под другие видеоформаты, а также позволит легко поэкспериментировать с таймингами не меняя основной код компонента. Начнем описывать наш компонент с заголовка класса:

```
case class KarnixTestHDMITopLevel(
        horiz_back_porch: Int = 48,
        horiz_active: Int = 640,
        horiz_front_porch: Int = 16,
        horiz_sync: Int = 96,
        vert_back_porch: Int = 33,
```

```
        vert_active: Int = 480,
        vert_front_porch: Int = 10,
        vert_sync: Int = 2
  ) extends Component{
```

В заголовке объявляются все восемь параметров для таймингов и тут же присваиваются им значения по-умолчанию согласно выбранному видеоформату (640x480 @ 60Гц, pixclk = 25.0МГц).

Компонент **KarnixTestHDMITopLevel** будет иметь два интерфейсных сигнала, **clk25** — опорный тактовый сигнал с кварцевого генератора и **hdmi** — комплексный сигнал описываемый интерфейсным классом **HDMIInterface**. В коде это выглядит следующим образом:

```
    val io = new Bundle {
        val clk25 = in Bool()
        val hdmi = master(HDMIInterface())
    }
```

6.4. Разбираемся с тактовыми сигналами

Далее объявим сигнальные линии для всех тактовых сигналов используемых в нашем компоненте, а их, включая выше объявленные **io.clk25** и **io.hdmi.tmds_clk**, будет целых пять:

```
    val pixclk_in = Bool()
    val pixclk = Bool()
    val pixclk_x10 = Bool()
```

С сигналом **pixclk_x10** все более-менее понятно — это сигнал несущий десятикратно умноженную с помощью PLL частоту **clk25**. А вот с сигналом **pixclk** придется немного поразбираться. Напомню, что мы собирались использовать опорную частоту **clk25** равную 25.0 МГц в качестве сигнала пиксельной развертки — **pixclk**. Однако, **pixclk_x10** как продукция от **clk25** пропущенного через PLL имеет существенный недостаток: блок PLL в составе ПЛИС Lattice (да и у многих других тоже) не гарантирует совпадение фаз для входного и получаемого на выходе сигналов. Более того, фаза выходного сигнала может и будет со временем слегка «дрейфовать (т. е. будет накапливаться небольшая разность фаз, то увеличиваясь, то уменьшаясь). Для нас это означает, что передавать битовый поток по линиям **tmds** с частотой **pixclk_x10**, которая по фазе не совпадает с частотой следования «символов» (**pixclk**), нельзя так как на принимающей стороне будут накапливаться ошибки и приемник в какой-то момент потеряет синхронизацию и отключится. Мой опыт это подтвердил — одна из первых моих проб была выполнена используя входной **clk25** в качестве **pixclk**, далее эта частота умножалась с помощью PLL и обе использовались для кодирования битового потока TMDS. Результат был таков: видеомонитор на несколько секунд отображал изображение, после чего терял синхросигнал секунд на 10, потом опять кратковременно отображал изображение и снова терял сигнал и т. д. Анализ сигналов **pixclk** и **pixclk_x10** на осциллографе показал полное несовпадение фаз и дрейф.

Решение проблемы синфазности **pixclk** и **pixclk_x10** решается достаточно хитро. Необходимо сначала получить **pixclk_x10** с помощью PLL, после чего от него произвести сигнал **pixclk** методом деления через обычный счетчик. Казалось бы - всё просто, но тут есть еще один нюанс: полученный таким образом «искусственный» тактовый сигнал теоретически может иметь очень высокий джиттер (будет сильно колебаться и отставать по

фазе) вызванный тем, что сигнальные линии соединяющие блоки внутри микросхем ПЛИС имеют разные пути и как следствие разные задержки. Эта проблема решается тем, что искусственный тактовый сигнал должен быть трассирован внутри ПЛИС специальным образом — используя глобальные линии, специально выделенные для таких целей. Чтобы «зарулить» наш искусственно полученный **pixclk** на глобальные линии, в ПЛИС ECP5 применяется специальный встроенный блок DCCA. Этот блок имеет один вход, к которому подключается искусственный клок, и один выход который имеет соединение с одной из глобальных линий. Иными словами, нам требуется выполнить следующее:

- Взять опорный сигнал **clk25**, прогнать его через PLL и получить десятикратный по частоте **pixclk_x10**.
- Далее из **pixclk_x10** методом деления через счетчик получить промежуточный сигнал **pixclk_in**.
- Прогнать **pixclk_in** через хард блок DCCA и получить требуемый нам **pixclk**.

И это еще не всё. Работать с сигналами **pixclk_x10** и **pixclk** нам придется в разных тактовых доменах, что требуется учитывать. Благо, в этом деле SpinalHDL почти всё сделает за нас автоматически, достаточно правильно описать блоки кода и отнести их к соответствующим тактовым доменам. Таким образом получаем следующий код для клоков:

```
     /* Route artificial TMDS clock using global lines, i.e. DCCA (ECP5 specific) */
    val dcca = new DCCA()
    dcca.CLKI := pixclk_in
    dcca.CE := True
    pixclk := dcca.CLKO

    /* Use ECP5 hard PLL block to multiply board provided 25.0 MHz to get 250.0 MHz for TMDS
encoder.
     * All PLL parameters are generated by ecppll utility.
     */
    val tmds_pll = new EHXPLLL( EHXPLLLConfig(clkiFreq = 25.0 MHz, mDiv = 1, fbDiv = 10, opDiv = 2,
opCPhase = 0) )
    tmds_pll.io.CLKI := io.clk25
    tmds_pll.io.CLKFB := tmds_pll.io.CLKOP
    tmds_pll.io.STDBY := False
    tmds_pll.io.RST := False
    tmds_pll.io.ENCLKOP := True
    tmds_pll.io.ENCLKOS := False
    tmds_pll.io.ENCLKOS2 := False
    tmds_pll.io.ENCLKOS3 := False
    tmds_pll.io.PLLWAKESYNC := False
    tmds_pll.io.PHASESEL0 := False
    tmds_pll.io.PHASESEL1 := False
    tmds_pll.io.PHASEDIR := False
    tmds_pll.io.PHASESTEP := False
    tmds_pll.io.PHASELOADREG := False
    pixclk_x10 := tmds_pll.io.CLKOP

    val dviClockDomain = ClockDomain(
        clock = pixclk,
        config = ClockDomainConfig(resetKind = BOOT),
        frequency = FixedFrequency(25.0 MHz)
    )

    val tmdsClockDomain = ClockDomain(
        clock = pixclk_x10,
        config = ClockDomainConfig(resetKind = BOOT),
        frequency = FixedFrequency(250.0 MHz)
    )
```

В приведенном выше коде компонент **EHXPLLL** представляет собой обертку для встроенного блока PLL, этот компонент входит в библиотеку **spinal.lib.blackbox.lattice.ecp5._** которую потребуется подключить оператором **import**. Настройки PLL сгенерированы автоматически утилитой **ecppll**.

Следом за PLL описываются два дополнительных тактовых домена: домен **dviClockDomain** тактируемый от **pixclk** будет содержать код реализующий логику формирования видео изображения, а домен **tmdsClockDomain** будет содержать код для формирования битовой последовательности (т. е. кодирования сигнала) и тактироваться от **pixclk_x10**.

Формирование промежуточного тактового сигнала **pixclk_in** выполним с помощью счетчика в тактовом домене **tmdsClockDomain** следующим образом:

```
    val tmds_area = new ClockingArea(tmdsClockDomain) {

        /* Generate 25 MHz PIXCLK by dividing pixclk_x10 by 10 */

        val clk_div = Reg(UInt(4 bits)) init(0)
        val clk = Reg(Bool())

        clk_div := clk_div + 1

        when(clk_div === 4) {
            clk := True
        }

        when(clk_div === 9) {
            clk := False
            clk_div := 0
        }

        pixclk_in := clk

        …
    }
```

Остается еще один тактовый сигнал который нам требуется сформировать — дифференциальный сигнал **tmds_clk** производный от **pixclk**. Этот сигнал в нашем компоненте представлен двумя линиями **io.hdmi.tmds_clk_p** и **io.hdmi.tmds_clk_n**. Напомню, что для формирования дифференциального сигнала используется встроенный хард блок **OBUFDS**, оберточный компонент к которому мы уже добавили в файл **HDMIInterface.scala**. Компонент **OBUFDS** принимает на вход один «однополярный» сигнал **I** и формирует на выходе два комплиментарных сигнала **O** и **OB**. Сигнал **tmds_clk** относится к тактовому домену **dviClockDomain**, соответственно получаем следующий код для формирования **tmds_clk:**

```
    val dvi_area = new ClockingArea(dviClockDomain) {

       …

        /* Produce TMDS clock differential signal which is PIXCLK, i.e. 25.0 MHz, not 250.0 MHz !!!
*/
        val tmds_clk = OBUFDS()
        tmds_clk.I := pixclk
        io.hdmi.tmds_clk_p := tmds_clk.O
        io.hdmi.tmds_clk_n := tmds_clk.OB

    }
```

На этом с тактовыми сигналами закончим и перейдем к сути — формированию изображения.

6.5. Формируем «муар»

Для формирования любого видео изображения нам потребуется где-то хранить компоненты цвета отображаемой в данный момент точки, в то время пока она побитно передается энкодером, это будут регистры **red**, **green** и **blue** размером 8 бит каждый. Также

нам потребуются сигнальные линии для отражения текущего состояния сигналов HSYNC, VSYNC и DE (Data Enabled), назовем их **hSync**, **vSync** и **de**. Добавим описание регистров и сигналов в область кода для тактового домена **dviClockDomain** тактируемого от **pixclk**, то есть после строки:

```
    val dvi_area = new ClockingArea(dviClockDomain) {
```

добавим код:

```
        /* Define RGB regs, HV and DE signals */
        val red = Bits(8 bits)
        val green = Bits(8 bits)
        val blue = Bits(8 bits)
        val hSync = Bool()
        val vSync = Bool()
        val vBlank = Bool()
        val de = Bool()
```

Так же нам потребуется два регистра счетчика для учета текущего положения «луча», т. е. экранной координаты текущей отображаемой точки. Назовем их **CounterX** и **CounterY**, а размерность вычислим исходя из параметров тайминга. Для удобства, опишем два параметра **horiz_total_width** и **vert_total_height** которые будут выражать размер строки в пикселах и размер кадра по высоте в строках, а размерность регистров **CounterX** и **CounterY** зададим как логарифм по основанию 2 округленный в верх от **horiz_total_width** и **vert_total_height** соответственно. Иными сломами, размерность регистров **CounterX** и **CounterY** всегда будет минимально достаточная для покрытия выбранного видеоформата.

```
        /* Generate picture using X and Y counters */

        /* Convenience params */
        val horiz_total_width = horiz_back_porch + horiz_active + horiz_front_porch + horiz_sync
        val vert_total_height = vert_back_porch + vert_active + vert_front_porch + vert_sync

        val CounterX = Reg(UInt(log2Up(horiz_total_width) bits))
        val CounterY = Reg(UInt(log2Up(vert_total_height) bits))
```

Регистр **CounterX** будем увеличивать каждый такт и обнулять его если он достиг размера строки, т. е. равен значению **horiz_total_width - 1**. Регистр **CounterY** будем увеличивать только тогда, когда **CounterX** достиг размера строки, а обнулять **CounterY** будем если он достиг значения **vert_total_height - 1**. Иными словами, **CounterX** будет перебирать все точки в строке, а **CounterY** будет перебирать все строки (включая затененные):

```
        CounterX := (CounterX === horiz_total_width - 1) ? U(0) | CounterX + 1

        when(CounterX === horiz_total_width - 1) {
            CounterY := ((CounterY === vert_total_height - 1) ? U(0) | CounterY + 1)
        }
```

Двоичные сигналы **hSync**, **vSync** и **de** тоже выразим через текущие значения регистров **CounterX** и **CounterY** соотнося их с параметрами таймингов:

```
        /* Produce HSYNC, VSYNC and DE based on back/front porches */

        hSync := (CounterX >= horiz_back_porch + horiz_active + horiz_front_porch) &&
                 (CounterX < horiz_back_porch + horiz_active + horiz_front_porch + horiz_sync)

        vSync := (CounterY >= vert_back_porch + vert_active + vert_front_porch) &&
                 (CounterY < vert_back_porch + vert_active + vert_front_porch + vert_sync)

        de := (CounterX >= horiz_back_porch && CounterX < horiz_back_porch + horiz_active) &&
              (CounterY >= vert_back_porch && CounterY < vert_back_porch + vert_active)
```

Далее все просто — если в текущий момент установлен флаг **de**, сигнализирующий о том что в данный момент передается видимая часть экранной области, то вычисляем значения **red**, **green** и **blue** для текущей точки исходя из текущих значений **CounterX** и **CounterY** путем взятия разных битов их этих регистров с последующим выравниванием влево. Фактически у нас получается деление по модулю с последующим умножением, что должно отображаться на экране в виде цветных полос соответствующей ширины и высоты. Если флаг **de** равен нулю, т. е. текущий момент находится в периоде затенения, то в регистры **red**, **green** и **blue** будем загружать нули:

```
when(de) {

    /* Period of visible area - Synthesize picture */
    red := CounterY(5 downto 4) ## B"000000"
    green := CounterX(6 downto 5) ## B"000000"
    blue := CounterX(7) ## B"0000000"

} otherwise {
    /* Period of blanking */
    red := 0
    green := 0
    blue := 0
}
```

На этом формирование «муара» заканчивается.

6.6. Подключаем TMDS энкодеры

Теперь нам останется подключить три TMDS энкодера, по одному на каждый канал цветности, отправить в них формируемые значения из регистров **red**, **green** и **blue** соответственно, а результат кодирования побитно выдвинуть в дифференциальные линии HDMI интерфейса **io.hdmi.tmds_p[2:0]** и **io.hdmi.tmds_n[2:0]**. Вспомним, что модуль **TMDS_encoder**, оформленный в виде «черного ящика», принимает на вход следующие сигналы:

```
val clk = in Bool()
val VD = in Bits(8 bits)
val CD = in Bits(2 bits)
val VDE = in Bool()
```

и формирует один выходной сигнал:

```
val TMDS = out Bits(10 bits)
```

где **clk** это сигнал тактовой частоты, за один такт которой энкодер преобразует 8 бит данных на входе **VD** или 2 бита управляющих сигналов **CD**, в зависимости о состояния сигнала **VDE**, в 10 битный сигнал **TMDS** представляющий собой «символ» для передачи в линию. В нашем случае блоки энекодеров будут тактироваться от **pixclk** (25.0МГц), входные данные VDE будут поступать из **red**/**green**/**blue** соответственно, а управляющие два бита **CD** буду заполнятся из сигналов **hSync** и **vSync**, но только для «голубого» канала. Для «красного» и «зеленого» эти управляющие биты всегда будут равны **0**. См. главу «*4. TMDS и кодирование сигнала в DVI-D и HDMI[(R)]*».

Получаем следующий код для подключения TMDS энкодеров, разместится он в области кода для тактового домена **dviClockDomain** сразу за кодом формирования изображения «муара»:

```
        /* Do TMDS encoding */

        /* Pass each color reg through external TMDS encoder to get TMDS regs filled */

        val encoder_R = TMDS_encoder()
        encoder_R.clk := pixclk
        encoder_R.VD := red
        encoder_R.CD := B"00"
        encoder_R.VDE := de

        val encoder_G = TMDS_encoder()
        encoder_G.clk := pixclk
        encoder_G.VD := green
        encoder_G.CD := B"00"
        encoder_G.VDE := de

        val encoder_B = TMDS_encoder()
        encoder_B.clk := pixclk
        encoder_B.VD := blue
        encoder_B.CD := vSync ## hSync /* Blue channel carries HSYNC and VSYNC controls */
        encoder_B.VDE := de
```

Код для формирования битовой последовательности для выдачи на дифференциальные линии разместим в области кода для домена **tmdsClockDomain**, так как биты будут формироваться с частотой **pixclk_x10**. Выглядит он следующим образом:

```
    val tmds_area = new ClockingArea(tmdsClockDomain) {
       …

        /* Produce G, R and B data bits by shifting each TMDS register.
           Use BufferCC() to cross clock domains.  */

        val TMDS_shift_red = Reg(Bits(10 bits)) init(0)
        val TMDS_shift_green = Reg(Bits(10 bits)) init(0)
        val TMDS_shift_blue = Reg(Bits(10 bits)) init(0)
        val TMDS_mod10 = Reg(UInt(4 bits)) init(0)
        val TMDS_shift_load = Reg(Bool()) init(False)

        TMDS_shift_red := TMDS_shift_load ? BufferCC(dvi_area.encoder_R.TMDS) | TMDS_shift_red(9 downto 1).resized
        TMDS_shift_green := TMDS_shift_load ? BufferCC(dvi_area.encoder_G.TMDS) | TMDS_shift_green(9 downto 1).resized
        TMDS_shift_blue := TMDS_shift_load ? BufferCC(dvi_area.encoder_B.TMDS) | TMDS_shift_blue(9 downto 1).resized
        TMDS_mod10 := ((TMDS_mod10 === U(9)) ? U(0) | TMDS_mod10 + 1)
        TMDS_shift_load := TMDS_mod10 === U(9)
```

По сути это три сдвиговых регистра, по одному на каждый канал цвета, размерностью 10 бит. Данные в эти регистры загружаются из TMDS энкодеров, но так как энкодеры находятся в другом тактовом домене (тактируются от сигнала **pixclk**), то они сначала проходят через библиотечную функцию **BufferCC()**, она скрытно добавит в код несколько последовательных регистров чтобы избавиться от потенциальных проблем при пересечении тактовых доменов.

Далее сформируем дифференциальные сигналы используя компонент OBUFDS:

```
        /* Produce differential signals using hard OBUFDS block */

        val tmds_0 = OBUFDS()
        tmds_0.I := TMDS_shift_blue(0)
        io.hdmi.tmds_p(0) := tmds_0.O
        io.hdmi.tmds_n(0) := tmds_0.OB

        val tmds_1 = OBUFDS()
        tmds_1.I := TMDS_shift_green(0)
        io.hdmi.tmds_p(1) := tmds_1.O
        io.hdmi.tmds_n(1) := tmds_1.OB
```

```
        val tmds_2 = OBUFDS()
        tmds_2.I := TMDS_shift_red(0)
        io.hdmi.tmds_p(2) := tmds_2.O
        io.hdmi.tmds_n(2) := tmds_2.OB
```

Чтобы компонент **KarnixTestHDMITopLevel** генерировался в отдельный Verilog файл, необходимо добавить немного синтаксического сахара, а именно — добавить точку входа. Выглядит это так:

```
object KarnixTestHDMIVerilog{
  def main(args: Array[String]) {
    SpinalVerilog(KarnixTestHDMITopLevel().setDefinitionName("KarnixTestHDMITopLevel"))
  }
}
```

И это будет конец файла **./src/main/scala/mylib/KarnixTestHDMI.scala**. Полный листинг этого файла можно получит из репозитория по ссылке:

https://github.com/Fabmicro-LLC/VexRiscvWithKarnix/blob/karnix_extended/src/main/scala/mylib/KarnixTestHDMI.scala

6.7. Собираем и запускаем KarnixTestHDMI на ПЛИС

Процесс сборки, т. е. получения из кода на языке SpinalHDL битстрима готового для загрузки в микросхему ПЛИС, состоит из нескольких этапов. В предыдущих публикация я подробно описывал этот процесс, далее я быстро пройдусь по основным моментам.

1. Первым делом необходимо сгенерировать код для синтезатора на языке Verilog, то есть преобразовать и SpinalHDL в Verilog. Делается это командой:

```
$ sbt "runMain mylib.KarnixTestHDMIVerilog"
```

На выходе генератора будет новый файл **KarnixTestHDMITopLevel.v**.

2. Полученный файл **KarnixTestHDMITopLevel.v** вместе с другими файлами содержащими код на языке Verilog (а их у нас еще два: **TMDS_encoder.sv** и **OBUFDS.sv**), необходимо подать на вход синтезатору Yosys командой вида:

```
$ yosys -v2 -p "synth_ecp5 -abc9 -top KarnixTestHDMITopLevel -json
KarnixTestHDMITopLevel.json" KarnixTestHDMITopLevel.v TMDS_encoder.sv OBUFDS.sv
```

На выходе будет огромный файл **KarnixTestHDMITopLevel.json** содержащий нетлист, т. е. схему состоящую из логических элементов и хард-блоков.

3. Полученный в результате синтеза нетлист вместе с файлом конфигурации **karnix_cabga256.lpf** необходимо подать на вход плейсера и получить оптимизированный файл конфигурации ПЛИС. Достигается это командой вида:

```
$ nextpnr-ecp5 --seed 1122 --speed 8 --25k --parallel-refine --lpf karnix_cabga256.lpf --
package CABGA256 --json KarnixTestHDMITopLevel.json --textcfg KarnixTestHDMITopLevel.json
```

4. Файл конфигурации ПЛИС из текстового вида преобразовать в битстрим следующей командой:

```
$ ecppack --svf bin/KarnixTestHDMITopLevel_25F.svf  KarnixTestHDMITopLevel_25F.config
KarnixTestHDMITopLevel_25F.bit
```

5. Теперь файл **KarnixTestHDMITopLevel_25F.bit** можно прошивать в ПЛИС утилитой **openFPGALoader**:

```
$ openFPGALoader -f bin/KarnixTestHDMITopLevel_25F.bit
```

Сразу после завершения прошивки флэш памяти, микросхема ПЛИС будет сброшена и переконфигурирована, разработанная нами цифровая схема заработает и отобразит на экран HDMI монитора или телевизора изображение тестового «муара», фотография которого приведена на рис. 7.

Чтобы упростить процесс сборки я подготовил отдельный сборочный файл **Makefile.TestHDMI**, его можно получить из репозитория по ссылке:

https://github.com/Fabmicro-LLC/VexRiscvWithKarnix/blob/karnix_extended/scripts/KarnixExtended/Makefile.TestHDMI

Замечу, что при работе с репозиторием команды сборки должны выполняться из подкаталога **./VexRiscvForKarnix/scripts/KarnixExtended** внутри этого репозитория (ветка **karnix_extenede**). Тогда запуск синтеза производится одной командой:

```
rz@devbox:~/VexRiscvForKarnix/scripts/KarnixExtended$ make -f Makefile.TestHDMI compile
```

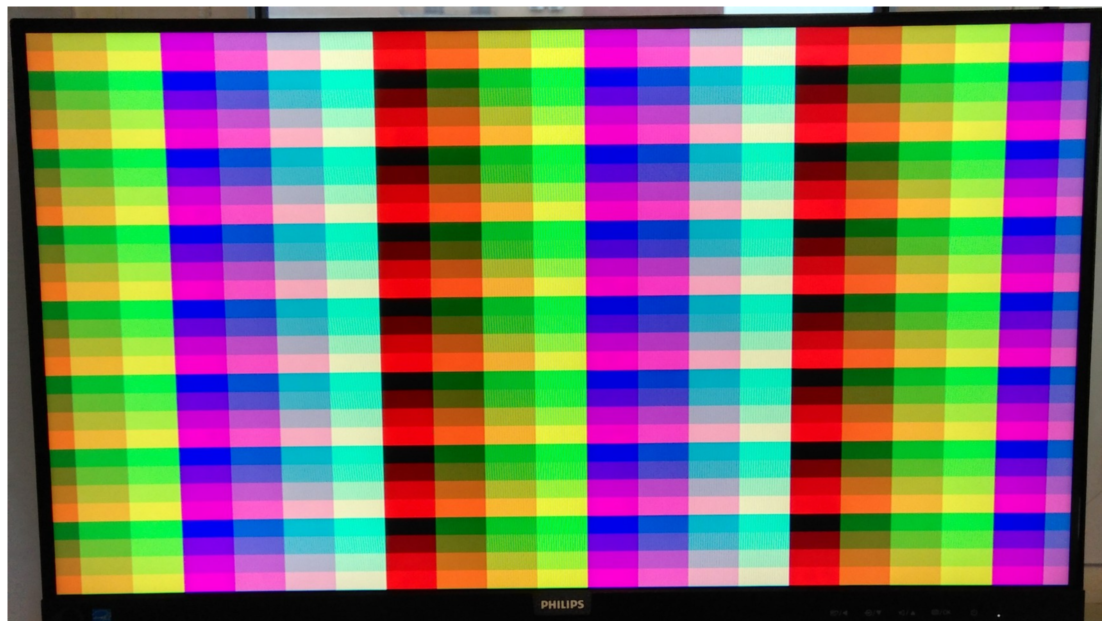


Рис. 7. Тестовое изображение (муар) на экране монитора синтезированное цифровой схемой KarnixTestHDMI на плате «Карно».

7. Color Graphics Adapter

В 1980 году гигант компьютерной индустрии, фирма IBM приняла решение выйти на рынок персональных компьютеров со своим видением этого вопроса. В отличии от традиционного для IBM подхода, который можно выразить словами «на всякую хрень мы придумаем свою еще большую дичь», специально созданная группа разработчиков, работавшая независимо от головного офиса и выведенная из-под влияния бюрократического аппарата IBM, вела тайную разработку опираясь в основном на опыт и чутье инженеров, нежели на корпоративные правила и традиции. Это чутье подсказывало им, что машина должна быть достаточно дешевой и легко расширяемой, а это означало что, во-первых, её нужно собирать из имеющихся в широком доступе компонентов, и во-вторых, архитектура машины должна быть полностью открытой, что по тем временем было весьма неординарным решением, особенно для IBM. Так и сделали. Машина, получившая название IBM Personal Computer model 5150 (или просто IBM PC которую мы сейчас называем ПК — персональный компьютер), была разработана в кратчайшие сроки (менее чем за 1 год), вышла в свет в августе 1981 года и произвела своего рода фурор в секторе SOHO (small office, home office) - мелкие фирмёшки, практикующие на дому врачи, юристы, бухгалтеры и прочие предприниматели увидели в ней большой потенциал. Однако IBM PC прошел совсем незамеченным среди креативной молодежи и любителей видео-игр. А все потому, что IBM PC первого релиза содержал видеоподсистему обеспечивающую работу только в текстовым монохромном режиме - MDA (Monochrome Display Adapter). Помимо того, что видеоадаптер MDA не предоставлял графики, он еще не позволял подключить IBM PC к обычному телевизору, как это было у других ПЭВМ того времени. Вместо этого к IBM PC подключался специализированный видеомонитор, который нужно было приобрести за отдельные деньги. Быстро осознав свой промах, IBM следом выпустила другой видеоадаптер — CGA (Color Graphics Adapter). Этот адаптер был снабжен сразу двумя разъемами - один для видеомонитора (RGBI интерфейс с разъемом DB-9), другой для обычного американского телевизора стандарта NTSC (разъем RCA). И не смотря на то, что у самой IBM видеомонитор с интерфейсом RGBI на тот момент отсутствовал (дисплей IBM 5153 появился только в марте 1983 года), на рынке присутствовали видеомониторы других производителей с которыми адаптер CGA худо-бедно мог работать. Таким образом компании IBM удалось закрыть сразу оба сектора — бизнес и домашне-развлекательный. Правда, с программным обеспечением и играми для IBM PC в то время была полная беда, но сейчас речь не об этом.

Замечание. Справедливости ради стоит отметить, что видеоадаптер MDA имел частоту строчной развертки равную 18,43 кГц, что несколько выше чем 15,7 кГц у CGA. За счет более высокой частоты развертки, изображение на экране MDA монитора выглядело более приятно глазу.

7.1. Устройство CGA адаптера

Рассмотрим по-подробнее что же представлял из себя видеоадаптер IBM CGA. Физически адаптер CGA представлял собой печатную плату размерами ~340x101мм с краевым разъемом для слота Industrial Standard Architecture (ISA), усеянную стандартной логикой из микросхем серии SN74xx. В центре платы находится специализированная микросхема контроллера ЭЛТ (CRT controller) - Motorola MC6845, рядом с ней расположена микросхема ПЗУ со знакогенератором, а под ними располагаются восемь микросхем динамической памяти MCM4517P12 для видео фреймбуфера суммарным объемом 16 килобайт, т.е. каждая микросхема по 1x16kBits. Интересно то, что контроллер MC6845 ничего не знает про графические режимы, его задача - отсчитывать знакоместа и

формировать выходной поток данных, необходимый для отображения на дисплей текущего символа, исходя из данных поступающих со знакогенератора. Разработчики адаптера CGA с помощью стандартной логики и хитрой настройки регистров контроллера, ухитрились подавать на его входы данные не из знакогенератора, а из видео памяти если адаптер работал в графическом режиме.

Микросхема MC6845 не была законченным видео процессором и в любом случае требовала «обвязки», тем не менее б**о**льшая часть работы по генерации изображения возлагалась именно на неё, в том числе: отображение курсора, который мог иметь различную форму; реализация «плавной» (по-пиксельной) и «грубой» (по-символьной) вертикальной прокрутки изображения (vertical scrolling); а также поддержка «светового пера» (Light Pen). Контроллер ЭЛТ MC6845 использовался и в других ПЭВМ того времени, таких как BBC Micro и Amstrad CPC.

На рис. 8 приведено изображение внутреннего устройства микросхемы MC6845, из него не сложно предположить как работал этот видео контроллер ЭЛТ. Если коротко, то в контроллере было несколько конфигурационных 8-ми битных регистров пронумерованных от R0 до R17 и несколько регистров-счетчиков. Конфигурационные регистры задавали параметры видео-таймингов выраженные в **символах** размером по 8 точек в ширину и по **N** строк в высоту (где **N** - не более 32-х строк). Регистры R0-R3 задавали горизонтальные тайминги, а регистры R4-R9 - вертикальные. Регистры-счетчики вели учет положения луча в текущий момент времени таким образом выражая номер текущего отображаемого символа по горизонтали и по вертикали. На блок-схеме эти регистры обозначены как **Horizontal CRT** и **Scan Line CRT** (вспоминаем счетчики CounterX и CounterY в нашем коде из главы *«6.5. Формируем «муар»*). В контроллере присутствовал еще один регистр-счетчик который учитывал текущий номер отображаемой строки внутри символа, обозначенный на схеме как **Character Row CRT**. Основываясь на содержимом этих трех счетчиков и принимая во внимание данные конфигурационных регистров, контроллер непрерывно адресовал на чтение ячейки видеопамяти, выставляя адрес ячейки на линии MA[13:0], и формировал на линиях RA[4:0] номер отображаемой строки внутри символа. 8-ми битные ASCII коды символов, поступающие из видеопамяти, комбинировались с данными RA[4:0] (номером строки внутри символа) и подавались на входы адреса микросхемы ПЗУ знакогенератора. Из неё извлекалась 8-ми битная последовательность пикселей и вместе с формируемыми контроллером сигналами синхронизации HSYNC, VSYNC и DE отправлялась далее в логику CGA адаптера, где преобразовывалась в уровни сигналов RGBI и отображалась на дисплее или экране телевизора. Интерфейс RGBI это дискретные линии Red, Green, Blue и Intensity выведенные на разъем типа DB-9, всего позволяют представить 16 цветовых комбинаций. Фактически, каждый из каналов R, G и B управлял одной из электронных пушек внутри ЭЛТ, по этому RGBI интерфейс еще назывался интерфейсом прямого управления ЭЛТ (Direct-Drive CRT).

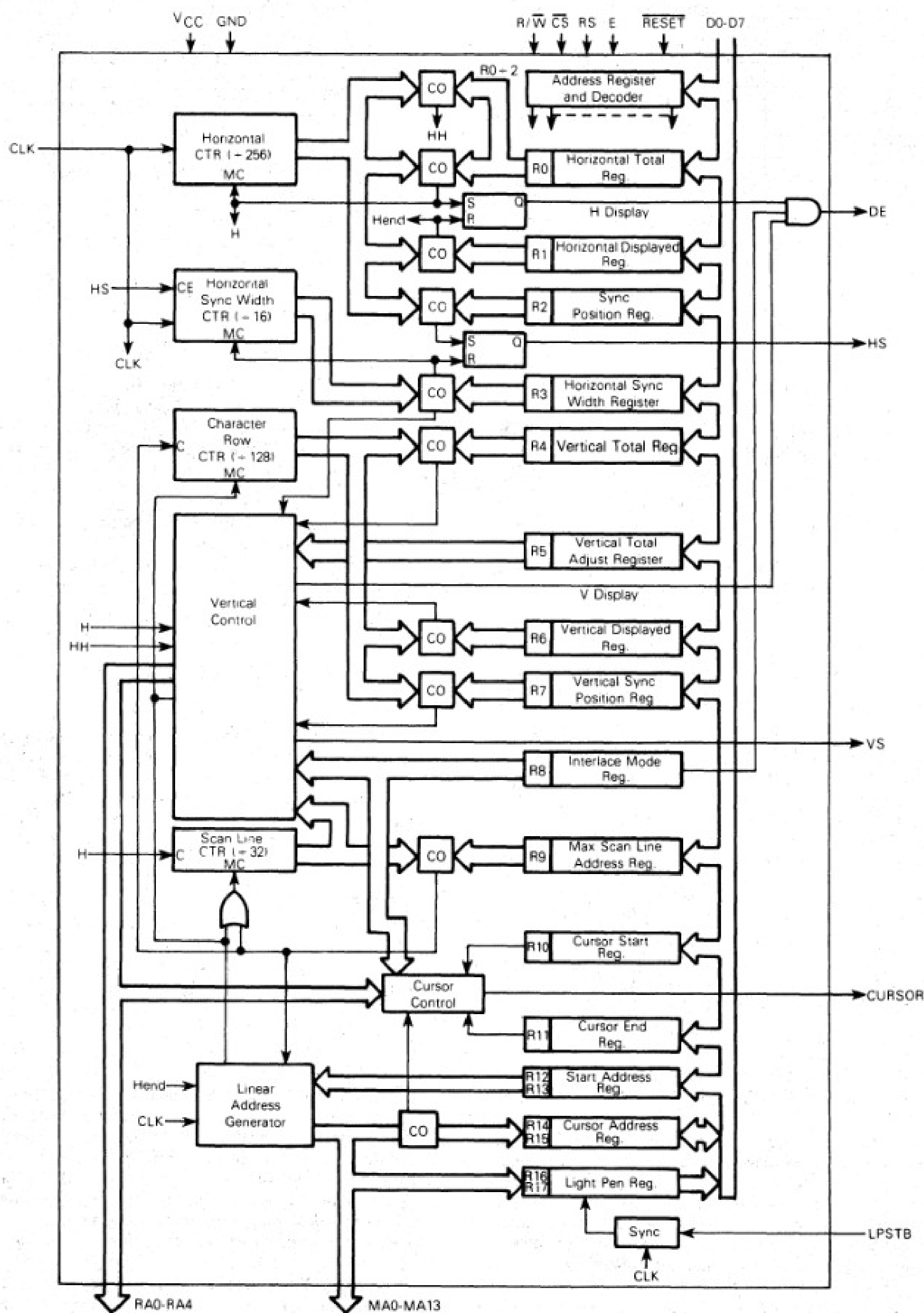


Рис. 8. Блочная схема устройства ЭЛТ контроллера MC6845.
Фрагмент спецификации от 1977г.

Из интересных моментов о работе CGA адаптера стоит отметить следующее.

Опорная частота в CGA адаптере равнялась **14,318181** МГц или 4-х кратному значению частоты несущей цвета в NTSC. Дабы минимизировать сложность схем синхронизации, частота процессора на IBM PC и XT устанавливалась в значение **4,772** МГц, что равнялось 1/3 от опорной частоты CGA. Выбирать частоту центрального процессора в

зависимости от частоты пиксельной развертки экрана была распространенной практикой для большинства ПЭВМ того времени.

В CGA адаптере использовалась однопортовая динамическая память. Это означает, что доступ к ячейкам памяти в один и тот же момент времени может получить только одно устройство. Так как контроллер ЭЛТ работал непрерывно, то разработчикам CGA адаптера пришлось применить схему разделения доступа во времени - контроллер осуществлял доступ к видеопамяти по переднему фронту тактового сигнала, после чего, на вторую половину такта, переводил линии адреса и данных в состояние высокого импеданса (HighZ). Это предоставляло возможность доступа к видеопамяти со стороны центрального процессора по спаду тактового импульса, то есть во второй половине такта, но требовало точной синхронизации действий между контроллером, центральным процессором и схемой обновления динамической памяти (в IBM PC обновлением SDRAM занимался DMA контроллер, цикл обновления через который происходил раз в 15мс). Если частота отображения пикселей не совпадала с тактовой частотой процессора, то доступ к видеопамяти от процессора приводил к появлению коротеньких штрихов случайно разбросанных по области экрана (т. н. «снег»). Аналогичный «глитч» появлялся если DMA не успевал обновить данные в SDRAM. И именно такой эффект «снега» проявлялся при работе в текстовом режиме высокого разрешения, 80x25 символов, где частота выборки из видеопамяти удваивалась и не совпадала с частотой шины центрального процессора, то есть возникала проблема перехода между тактовыми доменами. В BIOS на IBM PC это обходилось тем, что доступ к видеопамяти осуществлялся только в момент обратного хода луча (в момент затенения), а в момент прокрутки всей страницы экрана вывод на дисплей отключался. Эта проблема была решена в более поздних версиях адаптера, а так же во всех изделиях-клонах от других производителей, за счет более сложного механизма синхронизации и использовании микросхем двухпортовой памяти.

Контроллер MC6845 не имел никакой поддержки управления цветностью, так как формирование непосредственно видеосигнала RGBI не входило в его обязанности. Разработчики CGA адаптера озадачились этим моментом и реализовали возможность отображать символы 16-ю различными цветами отдельно для каждого символа и с 16-ю цветами для цвета фона знакоместа. Фактические цвета отображаемые на экране задавались фиксированным набором регистров палитры. Еще был введен режим мерцания символа — цвет текста периодически заменялся на цвет фона и обратно. Чтобы обеспечить такой функционал, каждый символ в видеопамяти представлялся двумя байтами - старший байт отводился под атрибут: старший полубайт задавал цвет фона знакоместа, младший — цвет текста, самый старший бит указывал на включение/выключение мерцания символа. А младший байт содержал ASCII код (codepage 437). Вспомним, что ЭЛТ контроллер выдавал на свою шину MA[13:0] номер символа который использовался как адрес ячейки видеопамяти. Этот адрес в CGA адаптере был расширен одним дополнительным битом, чтобы в видеопамяти появилось место под атрибут. В зависимости от видео режима, видеопамять CGA позволяла хранить от четырех до восьми страниц текста с атрибутами.

Как уже было отмечено выше, в контроллере MC6845 нет поддержки графического режима. Графический режим в CGA адаптере был реализован путем использования линий RA[4:0], формируемых контроллером для выбора номера строки символа, их задействовали для переключения между блоками (банками) видеопамяти, то есть каждая последующая строка находилась в своём адресном пространстве (video plane). Для реализации графического видео режима из 200 строк достаточно было задействовать один младший бит RA[0] установив высоту символов равной две строки. В этом режиме вся видеопамять делилась на два блока («плейна») - сначала шел блок с четными строками размером 8КБ, потом следовал такой же блок с нечетными строками. Такая организация видеопамяти в

графических режимах создавала ряд сложностей и дополнительную вычислительную нагрузку на ПО при отображении графики на CGA адаптере.

CGA адаптер позволял переключать регистры палитры «на лету» выбирая одну из шести палитр. В некоторых случаях, при наличии достаточного количества циклов процессора и за счет точного контроля, можно было увеличить число одновременно отображаемых на экране цветов. Некоторые игры на IBM PC пользовались такой возможностью.

Таблица символов знакогенератора (рис. 9) в CGA адаптере жестко зашивалась в ПЗУ и её нельзя было ни считать, ни переопределить программно. Всего в ПЗУ размещалось три набора символов: 8x14, 8x8 и 8x8 bold. По умолчанию использовался шрифт 8x8 bold, так как только он нормально отображался на композитном (NTSC) выходе. Выбор шрифта на плате оригинального IBM CGA адаптера осуществлялся с помощью джамперов.

Рис. 9. Набор символов кодовой страницы 437 — стандартный набор ASCII плюс «акцентированные» символы.

Еще одной интересной особенностью CGA адаптера было то, что изображение на RCA разъеме (композитный NTSC) поддавалось эффекту «артифактинга» - когда цвета двух рядом стоящих пикселей смешиваются и дают совсем другой цвет. Это позволяло, за счет некоторого ухудшения четкости изображения получить графическое изображение с б**о**льшим числом цветов. Данный эффект использовался во многих играх, но он не работал на RGBI мониторах. Ниже на рис. 10. приведено два изображения заставки одной и той же игры выведенное на RGBI монитор и на NTSC телевизор. Значительно позже, энтузиасты-исследователи и «демосценнеры» подбирая хитрые настройки контроллера ЭЛТ и отображая на экране различные «паттерны» в режиме высокого разрешения смогли отобразить с помощью CGA адаптера 1024 различных цвета на композитном мониторе.

Рис. 10. Пример эффекта «артифактинг» на CGA: слева — вывод на RGBI монитор, справа — вывод композитного сигнала на NTSC телевизор.

Интересное видео на тему внутреннего устройства IBM Color Graphics Adapter, а также исследование некоторых его «глитчей» доступно на Ютуб канале пользователя **PCRetroTech** по ссылке: https://www.youtube.com/watch?v=IQ2UeIx1qIA

7.2. Видео тайминги и видеоформаты CGA адаптера

Согласно руководству IBM Color/Graphics Monitor Adater в адаптере CGA было два текстового режима работы: режим среднего разрешение **40x25** символов и режим высокого разрешения **80x25** символов. Каждый символ представлялся матрицей из **8x8** видимых элементов (PELs в терминах IBM, т. е. «picture elements» или пикселей/точек). Если перевести эти размеры из символов в точки, то получим два эквивалентных графических режима разрешением **320x200** и **640x200**.

Для **графического режима 320x200** (как и текстового 40x25) частота PIXCLK устанавливалась равной половине опорной, т. е. **7,1590905** МГц. Максимальное количество точек сканирования по горизонтали составляло **456**, из них отображаемых было только **320**. Контроллер ЭЛТ был настроен на выдачу сигнала HSYNC с позиции **360**, однако реальный сигнал HSYNC выходил с CGA адаптера только спустя **32** точки и всегда имел длительность **64** точки. Не смотря на то, что эти 32 точки смещения попадали в область затенения, они отображались как «бордюр» слева и с права от видимой области, по 16 точек, одного из 16 цветов. Таким образом схема горизонтального тайминга была следующей: 320 + 40 + 32 + 64 = 456 точек, что давало частоту строчной развертки H-freq = **15699.759** Гц.

Вертикальная развертка для режима 320x200 складывалась из **200** видимых сканлайнов (строк), сигнал VSYNC начинался с **226**-й строки и продолжался **16** строк. Как и в случае со строчной разверткой, строки с **242** по **262** отображали бордюр сверху и снизу видимой области по 10 строк. Схема вертикального тайминга представлялась следующей: 200 + 26 + 16 + 20 = 262 сканлайнов, что давало частоту кадровой развертки V-freq = **59.927** кГц.

В графическом режиме 320x200 каждый байт строки (сканлайна) содержал информацию о четырех пикселах: четные строки располагались в общем адресном пространстве начиная с 0xB8000, а нечетные — с 0xBA000. Всего потреблялось 16000 байт видеопамяти.

Для **графического режима 640x200** (как и текстового 80x25) частота PIXCLK устанавливалась равной опорной частоте, т. е. **14,318181** МГц, а все горизонтальные тайминги удваивались. В этом режиме каждый пиксель мог иметь только один цвет задаваемый регистром палитры или отображаться цветом фона. Каждый байт строки содержал информацию о восьми пикселах. Как и для режима 320x200, четные строки располагались в общем адресном пространстве начиная с 0xB8000, нечетные — с 0xBA000. Также потреблялось 16000 байт видеопамяти.

В CGA адаптере был предусмотрен еще один **графический режим 160x100** точек позволяющий отображать по 16 цветов на точку, который назывался «Low-Resolution Color/Graphics Mode» и также потреблявший 16000 байт видеопамяти. В этом режиме цветность задавалась не через регистр палитры, а напрямую битами I, R, G и B расположенными в байтах видеопамяти. Данный режим формировался путем двойного отображения каждого PEL в строке и двойным сканирование строк, т. е. PEL в данном случае представлял собой четыре пикселя (размером 2x2).

Во всех графических режимах пиксели в памяти следовали слева направо от старших битов к младшим, то есть в старших битах находились информация о цветности пикселей расположенных левее.

7.3. Адаптируем CGA под современные реалии

Кратко пробежавшись по основным функциям CGA адаптера можно сказать следующее:

1. CGA адаптер тактируется от центрального процессора, умножая его частоту на 3 для получения PIXCLK равной ~14,318 МГц в режиме высокого разрешения, и умножая на 3/2 получая PIXCLK равную ~7,159 МГц в режиме среднего разрешения.
2. CGA адаптер умеет отображать два текстовых режима разрешением 40x25 и 80x25 символов, каждый символ по 8x8 пикселей. Символ в памяти занимает два байта: один байт ASCII кода и один байт атрибута задающего 16 цветов для текста, 8 цветов для фона и бит мерцания. Видеопамяти хватает на 4 или 8 страниц текста в зависимости от видео режима.
3. CGA адаптер имеет два графических режима разрешением 320x200 (4 цвета) и 640x200 (2 цвета). В обоих случая задействовано 16000 байт видеопамяти.
4. CGA адаптер может грубо прокручивать (скроллить) текст вверх и вниз в пределах страницы.
5. CGA адаптер умеет отображать мерцающий курсор в виде «подчеркивания», при этом размер курсора и его положение внутри знакоместа может изменяться.
6. CGA адаптер умеет отображать «бордюр» размером 16 точек по бокам и 10 точек сверху и снизу вокруг видимой области изображения. Бордюр мог быть только одного из 16-ти цветов.
7. CGA адаптер снабжался 16 килобайт динамической памяти используемой под видео фреймбуфер.

Встает вопрос: что из вышеперечисленного мы реально можем имплементировать на плате «Карно» с ПЛИС Lattice ECP5 объемом около 25K логических блоков? И что вообще имеет смысл имплементировать ?

Начнем с опорной частоты. Как мы уже выяснили в предыдущих главах при разработке теста KarnixHDMITest, получить опорную частоту 14,318 МГц или даже близкую к ней у нас вряд ли выйдет. А если выйдет, то современные мониторы и телевизоры всё равно не смогут отобразить такой видео сигнал. В нашем случае лучший вариант все же будет использовать 25,0 МГц в качестве опорной частоты, а в качестве основного формата использовать разрешение 640x480 пикселей.

Двигаемся далее. Снабдить наш адаптер памятью объемом 16 КБ не является проблемой. Теоретически мы можем легко синтезировать фреймбуфер объемом даже 64 КБ используя блоки распределенной памяти BRAM размером по 16 килобит, их в данной микросхеме ПЛИС содержится 56 штук, что суммарно дает **114688** байт **трехпортовой синхронной памяти**. Но тут нужно учитывать, что BRAM также используется и под другие нужды синтезируемой вычислительной системы, в том числе под регистровый файл, под кэш инструкций и данные, под основную оперативную память для программы начального старта. В результате некоторых изысканий я пришел к выводу, что наиболее реалистичный план распределения BRAM выглядит так:

- Основное ОЗУ (RAM) для кода, стэка и данных: 72 КБ — этого вполне достаточно, чтобы разместить ядро небольшой операционной системы, проинициализировать и сделать доступной SRAM (на плате «Карно» её 512КБ) или SDRAM при наличии таковой.
- Кэш инструкций и данных: 2 КБ + 2 КБ.
- MAC буфер для Ethernet фреймов: 2 КБ + 2 КБ.
- CGA адаптер:
  - видео фреймбуфер: 20 КБ;
  - регистры палитры: 16*32 бит = 64 байта;
  - ПЗУ знакогенератора: 256 символов шириной 8 пикселов и высотой 16 строк = 4096 байт.
- Остальное, по мелочи, рассредоточится между регистровым файлом, буферами UART, SPI, DAC и т.д.

На видеоформат 640x480 хорошо ложится текстовый режим из **30 строк по 80 символов**, где каждый символ состоит из 16 строк/сканлайнов по 8 точек. Учитывая, что подавляющее большинство современных телевизоров и видеомониторов имеют соотношение сторон 14:9, то удвоенная высота символа (по отношению к ширине) скомпенсирует его растяжение по горизонтали и придаст символу вид близкий к классическому. Как и в оригинальном CGA будем снабжать каждый символ на экране отдельным байтом с атрибутом задающим цвет текста и цвет фона, для чего нам может потребоваться 4800 байт на одну страницу текста (при 20КБ видеопамяти это целых 4 страницы текста). Но забегая несколько вперед скажу, что так как мы имеем дело с 32-х битной вычислительной системой у которой шина доступа к памяти тоже имеет ширину 32 бита, то для упрощения логики декодирования адреса символа и его атрибута (которые потребуется извлекать из памяти одновременно), будет целесообразнее если под символ мы тоже отведем 32 бита. Как и в случае с CGA, нулевой байт будет содержать ASCII код символа, первый байт — его атрибут, а остальные байты пока оставим без дела. При таком раскладе у нас получается размер одной страницы текста равный 9600 байт, а в лимит 20КБ вместится две полных страницы текста по 30 строк плюс хвостик из 4-х строк (всего 64 строки текста).

Знакогенератор нашего видеоадаптера, как и в случае с CGA адаптером, будет содержать матрицы для 256 символов размером 8 пикселей на 16 строк (всего 16 байт), т. е. для одной кодовой страницы потребуется 4096 байт. Очевидно, что кодовая страница «codepage 437» от IBM используемая в CGA в современных реалиях не актуальна - нас интересуют только стандартные ASCII символы с кодами 0-127, кириллица и псевдографика. Так как я веду разработку в среде Unix-подобной операционной системы, то логичным будет использовать кодовую страницу KOI8-R. К тому-же, в ОС FreeBSD есть готовые наборы консольных шрифтов, в том числе для символов размером 8x16 и таблицей KOI8-R: файл со шрифтами **/usr/share/syscons/fonts/koi8-r-8x16.fnt** закодированный утилитой **uuencode** содержит то, что нам требуется. Как он туда попал и как его раскодировать я расскажу и покажу позже при разработке знакогенератора.

Теперь поразмышляем над графическим режимом. Простой подсчет показывает, что для хранения видеоизображения размером 640x480 может потребоваться 38400 байт при однобитовом цвете и 76800 байт при двухбитовом. Ни один из этих вариантов нам не подходит, так как не вписывается в 20КБ лимит, а значит картинку придется четвертовать, т. е. поделить по ширине и по высоте на два, а видимый элемент PEL будет состоять из четырех пикселей организованных в матрицу 2x2. Так мы приходим к графическому видео режиму **320x240** по два бита на PEL, который займет 19200 байт видеопамяти, и это очень

близко к тому, что было в CGA. Теоретически, мы даже сможем имплементировать режим «высокого разрешения» размером 640x240 по одному биту на PEL.

У контроллера ЭЛТ MC6845 в составе CGA адаптера была возможность задавать с помощью конфигурационного регистра смещение в видеопамяти, начиная с которого производилась выборка данных для отображения на дисплей. Это позволяло осуществлять грубую прокрутку текста (по одной строке символов) вертикально и по одному символу горизонтально. Теоретически, можно было осуществлять плавную прокрутку (по одному PEL) и в графическом режиме, но так как объема видеопамяти было недостаточно для хранения какой-либо существенной части «невидимого» изображение в графическом режиме, то данная возможность почти не использовалась. В то же время, в текстовом режиме, данная возможность широко использовалась для уменьшения объема копируемых данных видеопамяти при вертикальном скроллинге текста. У более поздних видеоадаптеров для IBM PC (EGA и VGA) существовала возможность задать смещение для начала отображаемого изображения в регистре видеопроцессора в сканлайнах (а не в символах), что позволяло производить плавное вертикальное смещение изображения в текстовом и графическом режимах. Чтобы понять, как выглядит плавная прокрутка текста, можно посмотреть следующее видео. Такая плавная вертикальная прокрутка изображения может быть полезна для реализации не только видеоигр, но и для текстового терминала на подобие DEC VT320, что очень приятно глазу.

В нашей реализации видеоадаптера мы попытаемся реализовать такой вариант плавной (по одному сканлайну) вертикальной прокрутки изображения для графического и текстового режимов, при этом регистр смещения будет иметь размерность 10 бит, что позволит нам прокрутить все 64 строки текста (64*16 = 1024 сканлайнов) одним изменением значения этого регистра.

Реализовать мигающий курсор не представляет большой сложности, это всего два дополнительных конфигурационных регистра и два регистра для хранения координаты курсора.

От бордюра отказываемся сразу, так как видимой области экрана нам хватает только на полезное изображение и на бордюр места просто не остается - современные цифровые мониторы не позволяют отображать что-либо в периоды затенения.

8. Разработка CGA подобного видеоадаптера

Исходя из вышесказанного у нас складывается техническое здание на разработку CGA подобного видеоадаптера. Да, это будет далеко не настоящий CGA и его не вставить в слот ISA старушки IBM PC, но для решения поставленной в самом начале статьи задачи этого и не требуется. Напомню, что основная цель разработки — снабдить синтезируемый в ПЛИС микроконтроллер средством отображения текстовой и графической информации на современном телевизоре или мониторе подключаемому через HDMI (DVI-D) интерфейс.

Продолжим модифицировать и усовершенствовать проект **VexRiscvWithKarnix** в который мы ранее добавили тест отображающий «муар» через HDMI/DVI-D. Наш будущий CGA адаптер должен интегрироваться в одну из ранее разработанных синтезируемых СнК на базе **Briey** или **Murax**, для чего он должен быть снабжен одним из доступных шинных интерфейсов. В данном случае я выбираю интерфейс **Apb3**, так как он несложен в реализации и поддерживается обеими СнК. Далее все модификации я буду проводить для СнК на базе Briey, а точнее - её адаптированной к плате «Карно» версии, которую я назвал **BrieyForKarnix**.

Компонент реализующий наш будущий CGA адаптер назовем **Apb3CGA4HDMICtrl** и разместим его код в отдельном файле. Для этого перейдем в подкаталог **../scripts/KarnixExtended** репозитория, создадим новый файл **../../src/main/scala/mylib/CGA4HDMI.scala** с текстом на языке SpinalHDL, который является производной от языка Scala. Компонент будет входить в состав пакета (библиотеки) **mylib**, как и все остальные разрабатываемые в демонстрационных целях компоненты, обозначим этот момент в самом начале файле следующей строкой кода:

```
package mylib
```

Добавим строки для подключения используемых внешних библиотек:

```
import spinal.core._
import spinal.lib._
import spinal.lib.Counter
import spinal.lib.bus.amba3.apb.{Apb3, Apb3Config, Apb3SlaveFactory}
import spinal.lib.misc.HexTools
import mylib._
```

Библиотеки в составе **spinal.lib.bus.amba3.apb** содержат все необходимое для подключения к шине **Apb3**. Библиотека **spinal.lib.misc.HexTools** содержит утилиты для работы с бинарными файлами в формате Intel HEX — это потребуется нам для подгрузки знакогенератора из готового файла. Библиотека **mylib** содержит требуемые нам компоненты **HDMIInterface**, TMDS**_encoder**, **OBUFDS** и **DCCA —** точно так же как и для кода тестового «муара».

Добавим код описания заголовка компонента **Apb3CGA4HDMICtrl**:

```
case class Apb3CGA4HDMICtrl(
        horiz_back_porch: Int = 32,
        horiz_active: Int = 640,
        horiz_front_porch: Int = 32,
        horiz_sync: Int = 96,
        vert_back_porch: Int = 16,
        vert_active: Int = 480,
        vert_front_porch: Int = 27,
        vert_sync: Int = 2,
        charGenHexFile: String = "font8x16x256.hex"
      ) extends Component {
  val io = new Bundle {
    val apb              = slave(Apb3(addressWidth = 16, dataWidth = 32))
```

```
    val hdmi             = master(HDMIInterface())
    val pixclk_x10       = in Bool()
    val vblank_interrupt = out Bool()
  }

  // здесь будет много кода
}
```

В отличии от компонента **KarnixTestHDMITopLevel**, компонент **Apb3CGA4HDMICtrl** нашего CGA адаптера не будет содержать PLL, вместо этого мы передадим ему на вход уже умноженный тактовый сигнал **pixclk_x10** частотой 250 МГц, а код PLL вынесем уровнем выше - в тело описания СнК. Второе отличие — наличие параметра содержащего строку с именем файла для знакогенератора. Еще одно отличие в интерфейсной части — компонент **Apb3CGA4HDMICtrl** будет формировать выходной дискретный сигнал **vblank_interrupt** для контроллера прерываний, позволяя ему формировать прерывания вычислительного ядра в момент начала обратного хода луча (затенения). Наличие такого прерывания от адаптера сильно облегчает работу программиста и позволяет избавиться от ряда «глитчей» при обновлении данных в видеопамяти. И последнее — наличие комплексного сигнала **apb** для пристыковки к шине **Apb3** внутри СнК. Сразу после интерфейсной части добавим код для подключения к шине **Apb3**:

```
    val busCtrl = Apb3SlaveFactory(io.apb)
```

Далее определим два управляющих 32-х битных слова **cgaCtrlWord** и **cgaCtrl2Word**, и настроим декодер для них:

```
      val cgaCtrlWord = busCtrl.createReadAndWrite(Bits(32 bits), address = 48*1024+64)
init(B"32'x80000000")
      val cgaCtrl2Word = busCtrl.createReadAndWrite(Bits(32 bits), address = 48*1024+68)
init(B"32'xfc0f0000")
```

Метод **createReadAndWrite()** позволяет создать и подключить к шине ячейки памяти заданного размера (в нашем случае 32 бита), и сформировать декодер для обращения к ним на чтение и запись по адресу, указанном в поле **address**. Задаваемый адрес является смещением от базового адреса, к которому привязан сигнал **io.apb**, эту привязку мы сделаем позже уровнем выше при конфигурировании СнК. Здесь управляющие слова **cgaCtrlWord** и **cgaCtrl2Word** будут находится на шине со смещением 48*1024+64 и 48*1024+68 байт соответственно. Такое смещение выбрано потому, что первые 48К мы зарезервируем под адресное пространство видеопамяти, а еще 64 байта — под регистры палитры. Конструктор **init()** позволяет заполнить значения регистров (управляющих слов) по умолчанию которыми будут инициализироваться эти регистры после инициализации ПЛИС.

Следом распишем внутренние управляющие сигналы из состава управляющих слов:

```
  val video_enabled = cgaCtrlWord(31).addTag(crossClockDomain)
  val blanking_enabled = cgaCtrlWord(30).addTag(crossClockDomain)
  val video_mode = cgaCtrlWord(25 downto 24).addTag(crossClockDomain)
  val scroll_v_dir = cgaCtrlWord(10).addTag(crossClockDomain)
  val scroll_v = cgaCtrlWord(9 downto 0).asUInt.addTag(crossClockDomain)

  val cursor_x = cgaCtrl2Word(6 downto 0).asUInt.addTag(crossClockDomain)
  val cursor_y = cgaCtrl2Word(13 downto 8).asUInt.addTag(crossClockDomain)
  val cursor_bottom = cgaCtrl2Word(19 downto 16).asUInt.addTag(crossClockDomain)
  val cursor_top = cgaCtrl2Word(23 downto 20).asUInt.addTag(crossClockDomain)
  val cursor_blink = cgaCtrl2Word(26 downto 24).asUInt.addTag(crossClockDomain)
  val cursor_blink_enabled = cgaCtrl2Word(27).addTag(crossClockDomain)
  val cursor_color = cgaCtrl2Word(31 downto 28).asUInt.addTag(crossClockDomain)
```

Здесь мы берем по одному или несколько битов из управляющих слов и назначаем их как отдельные сигналы. Метод **addTag()** добавляет в сигнал «тэг» **crossClockDomain** указывающий на то, что обращение к данному сигналу может проходить через пересечение тактовых доменов, а значит генератору SpinalHDL следует предпринимать определенные действия при формировании выходного кода на Verilog.

Немного прокомментируем назначение этих сигналов.

Управляющее слово **cgaCtrlWord**:

- **blanking_enabled** - отключает отображение картинки на монитор. Если равен «1», то в выходном видеосигнале будут одни нули (черный экран).
- **video_mode** — двухбитовый сигнал определяющий текущий видеорежим: 2“b00 — текстовый, 2“b01 — графический. Остальные значения зарезервированы и при их включении экран будет окрашиваться полностью в красный цвет.
- **scroll_v_dir** — задает направление вертикальной прокрутки (смещения) изображения: 0 — вверх, 1 — вниз.
- **scroll_v** — сигнал размерностью 10 бит задает число линий сканирования на которое нужно прокрутить (сместить) по вертикали изображение.

Управляющее слово **cgaCtrlWord2**:

- **cursor_x** — сигнал размерностью 7 бит задает координату курсора по X (в символах).
- **cursor_y** — сигнал размерностью 6 бит задает позицию курсора по оси Y (в строках текста).
- **cursor_top** — сигнал размерностью 4 бита определяет верхнюю стоку в составе символа с которой начинается отображение курсора.
- **cursor_bottom** — сигнал размерностью 4 бита определяет нижнюю строку в составе символа на которой заканчивается отображение курсора.
- **cursor_blink** — сигнал размерностью 3 бита определяет делитель частоты кадровой развертки для вычисления частоты мигания курсора.
- **cursor_blink_enabled** — сигнал разрешает мигание курсора. Если установлен в «0», то курсор отображается как статический прямоугольник. Иначе — мигает с частотой согласно делителю **cursor_blink**.
- **cursor_color** — сигнал размерностью 4 бита определяющий цвет курсора выбираемый из 16 регистров палитры.

К свойствам курсора и методу его отображения мы еще вернемся в специально посвященной этой теме главе.

Настало время описать видеопамять (видео фреймбуфер) и подвязать декодер адреса к нему. Сделаем это следующим образом: сначала опишем буфер статической памяти требуемого размера, этот буфер на стадии синтеза будет собран из ячеек трехпортовой блоковой памяти BRAM.

```
val fb_mem = Mem(Bits(32 bits), wordCount = (320*(240+16)*2) / 32)
```

Здесь мы указываем размер одной ячейки блока (размер слова) равный 32 бита и число этих слов: 320*(240+16)*2) / 32 = 5120 что соответствует 20480 байт.

Заведем дискретный сигнал **fb_access** который будет устанавливаться в лог «1» при наличии на на шине **Apb3** адреса из диапазона 48К который мы отвели под видеопамять:

```
    val fb_access = io.apb.PENABLE && io.apb.PSEL(0) && io.apb.PADDR < 48*1024
```

Теперь используя сигнал **fb_access** опишем декодер, подключающий блок видеопамяти к шине **Apb3**:

```
when(fb_access) {
    io.apb.PRDATA := fb_mem.readWriteSync(
        address = (io.apb.PADDR >> 2).resized,
        data  = io.apb.PWDATA.resized,
        enable  = fb_access,
        write  = io.apb.PWRITE,
        mask  = 3
    )
    io.apb.PREADY := RegNext(fb_access)
  }
```

Метод **readWriteSync()** позволяет сделать доступ к блоку памяти синхронным, т. е. по тактовому импульсу (напомню, что обычно статическая память работает асинхронно). Ответный сигнал **io.apb.PREADY** формируется в следующем такте, за тактом, в котором производится доступ к ячейкам блока памяти, это делается добавлением одного последовательного триггера/регистра с помощью функции **RegNext()**.

Аналогичным образом добавим массив **palette_mem** для регистров палитры. Всего у нас будет 16 регистров, каждый по 32 бита. Младшие 24 бита каждого регистра палитры это комбинация цветовых компонентов RGB, старшие 8 бит не используются. При отображении графики или текста мы будем задавать цвет в виде индекса в этом массиве. Это позволит нам во-первых, выводить изображение любым цветом из 17 млн вариантов, во-вторых — менять цветовую гамму «на лету» путем загрузки новых значений в регистры палитры.

```
  val palette_mem = Mem(Bits(32 bits), wordCount = 16)

  val  palette_access  =  io.apb.PENABLE  &&  io.apb.PSEL(0)  &&  ((io.apb.PADDR  &  U"xffc0")  ===
U"xc000") // 49152
  when(palette_access) {
    io.apb.PRDATA := palette_mem.readWriteSync(
        address = (io.apb.PADDR >> 2).resized,
        data  = io.apb.PWDATA.resized,
        enable  = palette_access,
        write  = io.apb.PWRITE,
        mask  = 3
    )
    io.apb.PREADY := RegNext(palette_access)
  }
```

Оригинальный CGA адаптер от IBM также имел 16 регистров палитры, покрывающих все 16 вариантов цветов доступных к отображению на мониторах RGBI (комбинация битов цветности R, G, B и бита яркости I). На рис.11 приведена полная стандартная палитра использовавшаяся в IBM CGA.

Full CGA 16-color palette			
0	black #000000	8	dark gray #555555
1	blue #0000AA	9	light blue #5555FF
2	green #00AA00	10	light green #55FF55
3	cyan #00AAAA	11	light cyan #55FFFF
4	red #AA0000	12	light red #FF5555
5	magenta #AA00AA	13	light magenta #FF55FF
6	brown #AA5500	14	yellow #FFFF55
7	light gray #AAAAAA	15	white #FFFFFF
Note: Color hex values shown are 8-bit RGB equivalents, internally CGA is 4-bit RGBI			

Рис. 11. Стандартная палитра цветов видеоадаптера IBM Color/Graphics Adapter.

Далее добавим описание тактового сигнала **pixclk** как производную от **pixclk_x10**, а так же код описания тактовых доменов, позаимствовав всё это у компонента **KarnixTestHDMITopLevel**:

```
val pixclk_in = Bool() /* Artificially synthesized clock */
val pixclk = Bool() /* Artificially synthesized clock globally routed (ECP5 specific) */
val pixclk_x10 = Bool() /* x10 multiplied clock */

/* Route artificial TMDS clock using global lines, i.e. DCCA (ECP5 specific) */
val dcca = new DCCA()
dcca.CLKI := pixclk_in
dcca.CE := True
pixclk := dcca.CLKO

val dviClockDomain = ClockDomain(
  clock = pixclk,
  config = ClockDomainConfig(resetKind = BOOT),
  frequency = FixedFrequency(25.0 MHz)
)

val tmdsClockDomain = ClockDomain(
  clock = io.pixclk_x10,
  config = ClockDomainConfig(resetKind = BOOT),
  frequency = FixedFrequency(250.0 MHz)
)
```

Также добавим код из области **dvi_area** для управляющих сигналов формирования изображения. Данный код не полный, в последующих главах мы будем расширять секцию **dvi_area** добавляя новый функционал в наш видеоадаптер.

```
val dvi_area = new ClockingArea(dviClockDomain) {

  /* Define RGB regs, HV and DE signals */
  val red = Bits(8 bits)
  val green = Bits(8 bits)
  val blue = Bits(8 bits)
```

```
  val hSync = Bool()
  val vSync = Bool()
  val vBlank = Bool()
  val de = Bool()

  /* Generate counters */

  /* Convenience params */
  val horiz_total_width = horiz_back_porch + horiz_active + horiz_front_porch + horiz_sync
  val vert_total_height = vert_back_porch + vert_active + vert_front_porch + vert_sync

  /* Set of counters */
  val CounterX = Reg(UInt(log2Up(horiz_total_width) bits))
  val CounterY = Reg(UInt(log2Up(vert_total_height) bits))
  val CounterF = Reg(UInt(7 bits)) // Frame counter, used for cursor blink feature


  CounterX := (CounterX === horiz_total_width - 1) ? U(0) | CounterX + 1

  when(CounterX === horiz_total_width - 1) {
      CounterY := ((CounterY === vert_total_height - 1) ? U(0) | CounterY + 1)
  }


  when(CounterX === 0 && CounterY === 0) {
    CounterF := CounterF + 1
  }

  /* Produce HSYNC, VSYNC and DE based on back/front porches */
  hSync := (CounterX >= horiz_back_porch + horiz_active + horiz_front_porch) &&
           (CounterX < horiz_back_porch + horiz_active + horiz_front_porch + horiz_sync)

  vSync := (CounterY >= vert_back_porch + vert_active + vert_front_porch) &&
           (CounterY < vert_back_porch + vert_active + vert_front_porch + vert_sync)

  de := (CounterX >= horiz_back_porch && CounterX < horiz_back_porch + horiz_active) &&
        (CounterY >= vert_back_porch && CounterY < vert_back_porch + vert_active)

  vBlank := (CounterY < vert_back_porch) || (CounterY >= vert_active + vert_back_porch)



  /* Generate display picture */

  // ...
  // CGA implementation will be put here!
  // ...

  /* Do TMDS encoding */

  /* Pass each color reg through external TMDS encoder to get TMDS regs filled */

  val encoder_R = TMDS_encoder()
  encoder_R.clk := pixclk
  encoder_R.VD := red
  encoder_R.CD := B"00"
  encoder_R.VDE := de

  val encoder_G = TMDS_encoder()
  encoder_G.clk := pixclk
  encoder_G.VD := green
  encoder_G.CD := B"00"
  encoder_G.VDE := de

  val encoder_B = TMDS_encoder()
  encoder_B.clk := pixclk
  encoder_B.VD := blue
  encoder_B.CD := vSync ## hSync /* Blue channel carries HSYNC and VSYNC controls */
  encoder_B.VDE := de

  /* Produce TMDS clock differential signal which is PIXCLK, i.e. 25.0 MHz, not 250.0 MHz !!! */
  val tmds_clk = OBUFDS()
  tmds_clk.I := pixclk
  io.hdmi.tmds_clk_p := tmds_clk.O
  io.hdmi.tmds_clk_n := tmds_clk.OB

}
```

Следом добавим также позаимствованный код формирования TMDS сигнала и делитель частоты для **pixclk_in** (см. описание процесса формирования **pixclk** в главе «*6.4. Разбираемся с тактовыми сигналами*», то есть скопируем всю область кода **tmds_area**:

```
  val tmds_area = new ClockingArea(tmdsClockDomain) {

    /* Generate 25 MHz PIXCLK by dividing pixclk_x10 by 10 */

    val clk_div = Reg(UInt(4 bits)) init(0)
    val clk = Reg(Bool())

    clk_div := clk_div + 1

    when(clk_div === 4) {
      clk := True
    }

    when(clk_div === 9) {
      clk := False
      clk_div := 0
    }

    pixclk_in := clk


    /* Produce G, R and B data bits by shifting each TMDS register.
       Use BufferCC() to cross clock domains.  */

    val TMDS_shift_red = Reg(Bits(10 bits)) init(0)
    val TMDS_shift_green = Reg(Bits(10 bits)) init(0)
    val TMDS_shift_blue = Reg(Bits(10 bits)) init(0)
    val TMDS_mod10 = Reg(UInt(4 bits)) init(0)
    val TMDS_shift_load = Reg(Bool()) init(False)

    TMDS_shift_red := TMDS_shift_load ? BufferCC(dvi_area.encoder_R.TMDS) | TMDS_shift_red(9 downto 1).resized
    TMDS_shift_green := TMDS_shift_load ? BufferCC(dvi_area.encoder_G.TMDS) | TMDS_shift_green(9 downto 1).resized
    TMDS_shift_blue := TMDS_shift_load ? BufferCC(dvi_area.encoder_B.TMDS) | TMDS_shift_blue(9 downto 1).resized
    TMDS_mod10 := ((TMDS_mod10 === U(9)) ? U(0) | TMDS_mod10 + 1)
    TMDS_shift_load := TMDS_mod10 === U(9)

    /* Produce differential signals using hard OBUFDS block */

    val tmds_0 = OBUFDS()
    tmds_0.I := TMDS_shift_blue(0)
    io.hdmi.tmds_p(0) := tmds_0.O
    io.hdmi.tmds_n(0) := tmds_0.OB

    val tmds_1 = OBUFDS()
    tmds_1.I := TMDS_shift_green(0)
    io.hdmi.tmds_p(1) := tmds_1.O
    io.hdmi.tmds_n(1) := tmds_1.OB

    val tmds_2 = OBUFDS()
    tmds_2.I := TMDS_shift_red(0)
    io.hdmi.tmds_p(2) := tmds_2.O
    io.hdmi.tmds_n(2) := tmds_2.OB
  }
```

В самом конце заполним биты 19, 20 и 21 управляющего слова **cgaCtrlWord**, отображающие текущее состояние управляющий сигналов **hSync**, **vSync** и **vBlank**. Нам придется прогнать эти сигналы через двойной буфер для решения проблемы пересечения тактовых доменов:

```
  cgaCtrlWord(21 downto 19) := BufferCC(dvi_area.hSync ## dvi_area.vSync ## dvi_area.vBlank)
  io.vblank_interrupt := cgaCtrlWord(19)

  // Note, ## is concatination operator
```

И сформируем выходной сигнал **io.vblank_interrupt** для контроллера прерываний как копию сигнала **dvi_area.vBlank**.

На этом заканчиваются подготовительные работы и мы можем приступить к реализации функциональных блоков нашего CGA подобного видеоадаптера.

8.1. Графический режим, он самый простой

Для любого дискретного видеоадаптера формирование изображения в графическом режиме это непрерывный процесс который сводится к тому, чтобы изымать из нужных ячеек памяти данные и подавать их на видеовыход в формате понятном видеомонитору. Наш CGA подобный видеоадаптер не исключение. Чтобы изымать данные из памяти нам необходимо формировать адрес ячейки исходя из текущих значений счетчиков **CounterX** и **CounterY**. Напомню, что для удобства видеопамять в нашем адаптере реализована в виде 32-х битных слов с пословной (не побайтовой) адресацией, а каждое слово в графическом режиме 320x240 содержит информацию о 16-ти PEL-ах (отображаемых макро-пикселях размером 2x2), по два бита на PEL, что дает нам 20 слов на одну строку из 320 PEL-ов. Эти два бита в свою очередь являются индексом в массиве из 16 регистров палитры цветов, значение регистра и есть цвет PEL-а заданный в формате R8G8B8:

```
red := palette_mem(color).asBits(7 downto 0)

green := palette_mem(color).asBits(15 downto 8)

blue := palette_mem(color).asBits(23 downto 16)
```

где **color** — двухбитовый индекс цвета, а **red**, **green** и **blue** — 8-ми битовые составляющие компоненты цвета, которые мы будем передавать в TMDS энкодер. Всего регистров палитры 16 штук, но для графического режима 320x240 рабочими являются только первые четыре регистра, так как в этом режиме у нас всего два бита на индекс.

При вычислении адреса ячейки нам нужно учесть, что отображаемый PEL это не есть точка/пиксель на экране, который в данный момент отрисовывается дисплеем. Так как частота следования пикселей **pixclk** задана из расчета 640 видимых в строке пикселей и 480 видимых сканлайнов, то формула для расчета адреса слова будет следующей:

word_address = **CounterY** / 2 * 20 + **CounterX** / 2 / 16.

То есть нужно поделить **CounterX** и **CounterY** на два чтобы перейти от координат пикселя к координатам PEL-а, умножить **CounterY** на **20** слов по 16 PEL, еще раз поделить **CounterX** на **16** чтобы выбрать слово в строке и сложить эти произведения. Звучит монструозно, но на языке SpinalHDL это будет выглядеть весьма просто:

```
word_address := (CounterY(9 downto 1) * 20 + CounterX(9 downto 5)).resized
```

Вычислив адрес **word_address**, считать 32-х битное слово **word** из блока синхронной статический видеопамяти **fb_mem** можно следующим образом:

```
word := fb_mem.readSync(address = word_address, enable = true, clockCrossing = true)
```

Далее нам остается изъять из слова **word** индекс **color** одного из 16 PEL-ов, в зависимости от текущего значения **CounterX**:

```
val color = UInt(4 bits)

color := (word << (CounterX(4 downto 1) << 1))(31 downto 30).asUInt.resized
```

вычислить значения **red**, **green** и **blue** исходя из данных палитры, и отправить их в TMDS энкодер, код которого ранее был позаимствован у компонента **KarnixTestHDMITopLevel** (см. код выше после комментария /* Do TMDS encoding */).

Во всем этом деле есть одна небольшая сложность — синхронная статическая память, чтение данных из которой осуществляется за один такт: выставив адрес **word_address** в текущем такте, полезные данные в регистре **word** мы обнаружим только в следующем такте. А это означает, что начинать считывать слово из видеопамяти следует заранее, как минимум за один такт (один пиксель) до того как данные из него потребуются для формирования изображения. Чтобы решить эту проблему, во-первых заведем флаг **word_load** для разрешения считывания из видеопамяти при отображении каждого **31-го пикселя**:

```
word_load := (CounterX >= 0 && CounterX < U(horiz_back_porch + horiz_active)) &&
             (CounterY < vert_back_porch + vert_active) && ((CounterX & U(31)) === U(31))
```

Во-вторых, чтобы значение регистра **word_address** вычислялось с опережением на одно слово (то есть на 32 пикселя или 16 PEL-ов), установим тайминг **horiz_back_porch** так, чтобы он задавал отступ от начала видимой части строки в 32 пикселя. Собственно это значение (**horiz_back_porch = 32**) и указано в качестве значения по-умолчанию в заголовке компонента. А чтобы сигнализировать блоку памяти о том, что пора загрузить новое слово, присвоим сигналу **enable** в методе **readSync()** значение флага **word_load**:

```
word := fb_mem.readSync(address = word_address, enable = word_load, clockCrossing = true)
```

Собрав все выше сказанное воедино, получим следующий код для формирования изображения в графическом режиме (**video_mode** равен **1**):

```
/* Generate display picture */

val blanking_not_enabled = !blanking_enabled
val word_load = Bool()
val word = Reg(Bits(32 bits))
val word_address = UInt(13 bits)

word_load := False
word_address := 0

// 32 bit word of current character/PEL data
word := fb_mem.readSync(address = word_address, enable = word_load, clockCrossing = true)

switch(video_mode) {

  is(B"00") { // Text mode: 80x30 characters each 8x16 pixels
    // To be implemented...
  }

  is(B"01") { // Graphics mode: 320x240, 2 bits per PEL with full color palette

    // Load flag active on each 30 and 31 pixel of 32 bit word
    word_load := (CounterX >= 0 && CounterX < U(horiz_back_porch + horiz_active)) &&
                 (CounterY < vert_back_porch + vert_active) && ((CounterX & U(30)) === U(30))
```

```
        // Index of the 32 bit word in framebuffer memory: addr = y/2 * 20 + x/2/16
        word_address := (CounterY(9 downto 1) * 20 + CounterX(9 downto 5)).resized

        when(de && blanking_not_enabled) {

          val color = UInt(4 bits)

          color := (word << (CounterX(4 downto 1) << 1))(31 downto 30).asUInt.resized

          red := palette_mem(color).asBits(7 downto 0)
          green := palette_mem(color).asBits(15 downto 8)
          blue := palette_mem(color).asBits(23 downto 16)

        } otherwise {
          red := 0
          green := 0
          blue := 0
        }
      }

      default { // Display red screen for unsupported video modes
          red := 255
          green := 0
          blue := 0
      }
    }
```

Разместим его в область **dvi_area** сразу после комментария /* Generate display picture */ и можно сказать, что наш видеоадаптер почти готов к эксплуатации.

8.2. Интеграция CGA адаптера в синтезируемую СнК

Для того, чтобы как-то задействовать получившееся устройство, нам необходимо выполнить интеграцию нового компонента **Apb3CGA4HDMICtrl** в СнК. Далее я буду рассказывать как подключить наш CGA подобный адаптер к СнК на базе **Briey** (входит в состав VexRiscv), хотя его с таким же успехом можно подключить к СнК на базе Murax или Litex. А если скомпилировать код SpinalHDL в Verilog, то его можно будет подключить к любой системе-на-кристалле с поддержкой шины **Apb3** (фактически — любой микроконтроллер).

Итак, возьмем готовый пример СнК **Briey** из репозитория VexRiscv, файл **./src/main/scala/vexriscv/demo/Briey.scala**, скопируем его в файл **BrieyForKarnix.scala** в этом же подкаталоге и отредактируем. Для начала нам потребуется переименовать все классы **Briey*** в **BrieyForKarnix*** чтобы не создавать конфликтов и путаницы. Далее необходимо подключить созданные нами компоненты (классы), добавив еще одну строку **import** в заголовке файла к уже имеющимся:

```
import mylib.{HDMIInterface, Apb3CGA4HDMICtrl}
```

Далее, в интерфейсную часть компонента **BrieyForKarnix** добавить комплексный сигнал **hdmi** типа **HDMIInterface**, содержащий сигнальные линии HDMI/DVI-D интерфейса, а также дополнительный тактовый сигнал **pixclk_x10**:

```
class BrieyForKarnix(val config: BrieyForKarnixConfig) extends Component{
   ...
    val io = new Bundle {
    ...
    val hdmi = master(HDMIInterface())
    val pixclk_x10 = in Bool()
```

```
  }
```

В реализацию компонента **BrieyForKarnix** в область кода **axi** добавим наш CGA адаптер:

```
  val axi = new ClockingArea(axiClockDomain) {
    ...
    val cgaCtrl = new Apb3CGA4HDMICtrl()
    io.hdmi := cgaCtrl.io.hdmi
    cgaCtrl.io.pixclk_x10 := io.pixclk_x10
```

Если используется контроллер прерываний, то подключим к одному из его каналов сигнал **vblank_interrupt** от видеоадаптера:

```
    plic.setIRQ(cgaCtrl.io.vblank_interrupt, 7)
```

И добавим CGA адаптер в декодер адреса шины **Apb3** со смещением **0x40000**:

```
    val apbDecoder = Apb3Decoder(
      master = apbBridge.io.apb,
      slaves = List(
        ...
        cgaCtrl.io.apb -> (0x40000, 64 kB),
        …
      )
    )
  }
```

С учетом базового адреса шины **Apb3**, который равен 0xF0000000, базовый адрес регистров управления нашим видеоадаптером будет равен **0xF0040000**.

Осталось добавить внешние линии для комплексного сигнала **hdmi** для вывода сигналов на контакты микросхемы ПЛИС и описать PLL для получения десятикратной частоты **pixclk_x10**. Выполним это в компоненте **BrieyForKarnixTopLevel**.
В интерфейсной части добавим:

```
case class BrieyForKarnixTopLevel() extends Component{
    val io = new Bundle {
        ...
        val hdmi = master(HDMIInterface())
        ...
    }
```

В теле компонента добавим PLL **hdmi_pll** и соединим полученный тактовый сигнал **hdmi_pll.io.CLKOP** с сигналом **briey.io.pixclk_x10** в компоненте СнК:

```
    val hdmi_pll = new EHXPLLL( EHXPLLLConfig(clkiFreq = 25.0 MHz, mDiv = 1, fbDiv = 10, opDiv = 2,
opCPhase = 0) ) // 250.0 MHz
    hdmi_pll.io.CLKI := io.clk25
    hdmi_pll.io.CLKFB := hdmi_pll.io.CLKOP
    hdmi_pll.io.STDBY := False
    hdmi_pll.io.RST := False
    hdmi_pll.io.ENCLKOP := True
    hdmi_pll.io.ENCLKOS := False
    hdmi_pll.io.ENCLKOS2 := False
    hdmi_pll.io.ENCLKOS3 := False
    hdmi_pll.io.PLLWAKESYNC := False
    hdmi_pll.io.PHASESEL0 := False
```

```
hdmi_pll.io.PHASESEL1 := False
hdmi_pll.io.PHASEDIR := False
hdmi_pll.io.PHASESTEP := False
hdmi_pll.io.PHASELOADREG := False
briey.io.pixclk_x10 := hdmi_pll.io.CLKOP
```

Настройки PLL позаимствуем у разработанного ранее тестового компонента **KarnixTestHDMITopLevel**.

Ну и последний штрих — соединяем комплексный сигнал **hdmi** от компонента СнК с внешним миром:

```
io.hdmi <> briey.io.hdmi
```

Теперь можно проводить сборку. Генерацию кода Verilog из SpinalHDL можно выполнить следующей командой:

```
rz@devbox:~/VexRiscvForKarnix$ sbt "runMain vexriscv.demo.BrieyForKarnixVerilog"
```

Для выполнения полного цикла синтеза и получения битстрима лучше воспользоваться существующим Makefile-ом в каталоге **./scripts/KarnixExtended** переделав его для нового СнК **BrieyForKarnix**. Я не стану приводить здесь текст этого сборочного файла, так как он достаточно объемный и не относится к сути дела, вместо этого приведу ссылку на него в репозитории на Github-е:

https://github.com/Fabmicro-LLC/VexRiscvWithKarnix/blob/karnix_extended/scripts/KarnixExtended/Makefile.Briey

С использованием сборочного файла синтез выполняется одной командой:

```
$ make -f Makefile.Briey generate compile
```

Выполним эту команду несколько раз, устраним синтаксические ошибки и добьемся формирования битстрима в файле:

```
rz@devbox:~/VexRiscvForKarnix/scripts/KarnixExtended$ ll bin/BrieyForKarnixTopLevel_25F.bit
-rw-rw-r-- 1 rz rz 712177 Jul 11 22:54 bin/BrieyForKarnixTopLevel_25F.bit
```

8.3. Простейший код на Си для тестирования графического режима

Для того, чтобы как-то протестировать наш CGA видеоадаптер нам потребуется написать небольшую библиотечку позволяющую работать с его регистрами, то есть то, что обычно называют Hardware Abstraction Layer (HAL). Начнем с того, что добавим в существующий код для программы «karnix_extended_test» новый заголовочный файл **cga.h** с описанием структуры регистров CGA адаптера и набора макроопределений для доступа к битовым полям этих регистров:

```
rz@devbox:~/VexRiscvForKarnix/src/main/c/karnix_extended_test$ cat src/cga.h

#ifndef __CGA_H__
#define __CGA_H__

#define CGA_VIDEO_WIDTH         320
#define CGA_VIDEO_HEIGHT        240
#define CGA_FRAMEBUFFER_SIZE    (CGA_VIDEO_WIDTH*CGA_VIDEO_HEIGHT*2/8)

#include <stdint.h>
#pragma pack(1)
typedef struct
{
  uint8_t FB[CGA_FRAMEBUFFER_SIZE];                       // Framebuffer
  uint8_t unused1[48*1024-CGA_FRAMEBUFFER_SIZE];          //
  volatile uint32_t PALETTE[16];                          // offset 48K
  volatile uint32_t CTRL;                                 // 48K + 64
  volatile uint32_t CTRL2;                                // 48K + 128
  uint8_t unused2[12200];                                 //
  uint8_t CHARGEN[4096];                                  // offset 60K
} CGA_Reg;
#pragma pack(0)

#define CGA_MODE_TEXT           0
#define CGA_MODE_GRAPHICS1      1

#define CGA_CTRL_VIDEO_EN               (1 << 31)
#define CGA_CTRL_BLANKING_EN            (1 << 30)
#define CGA_CTRL_VIDEO_MODE_SHIFT       24
#define CGA_CTRL_VIDEO_MODE             (3 << CGA_CTRL_VIDEO_MODE_SHIFT)
#define CGA_CTRL_HSYNC_FLAG             (1 << 21)
#define CGA_CTRL_VSYNC_FLAG             (1 << 20)
#define CGA_CTRL_VBLANK_FLAG            (1 << 19)
#define CGA_CTRL_V_SCROLL_DIR           (1 << 10)
#define CGA_CTRL_V_SCROLL_SHIFT         0
#define CGA_CTRL_V_SCROLL               (0x03ff << CGA_CTRL_V_SCROLL_SHIFT)

#define CGA_CTRL2_CURSOR_X_SHIFT        0
#define CGA_CTRL2_CURSOR_X              (0xff << CGA_CTRL2_CURSOR_X_SHIFT)
#define CGA_CTRL2_CURSOR_Y_SHIFT        8
#define CGA_CTRL2_CURSOR_Y              (0xff << CGA_CTRL2_CURSOR_Y_SHIFT)
#define CGA_CTRL2_CURSOR_BOTTOM_SHIFT   16
#define CGA_CTRL2_CURSOR_BOTTOM         (0x0f << CGA_CTRL2_CURSOR_BOTTOM_SHIFT)
#define CGA_CTRL2_CURSOR_TOP_SHIFT      20
#define CGA_CTRL2_CURSOR_TOP            (0x0f << CGA_CTRL2_CURSOR_TOP_SHIFT)
#define CGA_CTRL2_CURSOR_BLINK_SHIFT    24
#define CGA_CTRL2_CURSOR_BLINK          (0x07 << CGA_CTRL2_CURSOR_BLINK_SHIFT)
#define CGA_CTRL2_CURSOR_BLINK_EN_SHIFT 27
#define CGA_CTRL2_CURSOR_BLINK_EN       (1 << CGA_CTRL2_CURSOR_BLINK_EN_SHIFT)
#define CGA_CTRL2_CURSOR_COLOR_SHIFT    28
#define CGA_CTRL2_CURSOR_COLOR          (0x0f << CGA_CTRL2_CURSOR_COLOR_SHIFT)
```

Отредактируем заголовочный файл **src/soc.h** и добавим в него базовый адрес для доступа к регистрам CGA адаптера рядом с остальной периферией перед, этим добавив подключение файла **cga.h**:

```
#include "cga.h"
...
#define TIMER_PRESCALER ((Prescaler_Reg*)0xF0020000)
#define TIMER_INTERRUPT ((InterruptCtrl_Reg*)0xF0020010)
#define TIMER0          ((Timer_Reg*)0xF0020000)
#define TIMER1          ((Timer_Reg*)0xF0021000)
#define UART0           ((Uart_Reg*)(0xF0010000))
```

```
...
#define CGA              ((CGA_Reg*)(0xF0040000))
...
```

Далее создадим файл **src/cga.c** и поместим в него код нескольких вспомогательных функций:

```
void cga_set_video_mode(int mode) {
        CGA->CTRL &= ~CGA_CTRL_VIDEO_MODE;
        CGA->CTRL |= (mode << CGA_CTRL_VIDEO_MODE_SHIFT) & CGA_CTRL_VIDEO_MODE;

        printf("cga_set_video_mode: mode = %d, ctrl = %p\r\n", mode, CGA->CTRL);
}

void cga_fill_screen(char color) {
        uint32_t *fb = (uint32_t*) CGA->FB;

        color = color & 0x3;

        uint32_t filler = (color << 30) | (color << 28) | (color << 26) | (color << 24);
                 filler |= (filler >> 8) | (filler >> 16) | (filler >> 24);

        for(int i = 0; i < CGA_FRAMEBUFFER_SIZE / (8*4); i += (8*4)) {
                *fb++ = filler;
                *fb++ = filler;
                *fb++ = filler;
                *fb++ = filler;
                *fb++ = filler;
                *fb++ = filler;
                *fb++ = filler;
                *fb++ = filler;
        }
}

void cga_wait_vblank(void) {
        while(!(CGA->CTRL & CGA_CTRL_VBLANK_FLAG));
}

void cga_set_palette(uint32_t c[16]) {
        memcpy((void *)CGA->PALETTE, c, 16 * 4);
}
```

Добавим описание этих функций в заголовочный файл **src/cga.h**:

```
void cga_set_video_mode(int mode);
void cga_fill_screen(char color);
void cga_wait_vblank(void);
void cga_set_palette(uint32_t c[16]);
```

Функция **cga_set_video_mode(int mode)** устанавливает режим работы видеоадаптера (текстовый или графический), в качестве параметра **mode** может быть передано значение константы **CGA_MODE_TEXT** или **CGA_MODE_GRAPHICS1**. На данный момент у нас реализован только графический режим **CGA_MODE_GRAPHICS1**.

Функция **cga_wait_vblank(void)** ожидает момента начала затенения (обратного хода луча), позволяет избавиться от некоторых глитчей при доступе к видеопамяти. Желательно вызывать её каждый раз при отрисовке в видеопамять, при этом важно успеть закончить работу с видеопамятью до начала вывода следующего кадра.

Функция **cga_fill_screen(char color)** заполняет весь экран одним цветом индекс **color** которого передается в качестве параметра.

Функция **cga_set_palette(uint32_t c[16])** позволяет загрузить массив со значением всех 16-ти регистров палитры. Важно понимать, что после инициализации аппаратуры нашего видеоадаптера в регистрах палитры могут находится случайные значения (или нули), по этому попытка вывода на экран без загрузки палитры может дать неожиданный результат, либо отображать черный экран.

Имея в руках такой набор HAL примитивов мы уже можем что-то отобразить на экран путем заполнения видеопамяти константой или путем копирования в область видеопамяти из другой области статической памяти. Последовательность действий при этом следующая:

- Установить графический режим CGA_MODE_GRAPHICS1;
- загрузить CGA палитру;
- ожидать начала затенения;
- заполнить экран каким нибудь одним цветом.

Создадим наш первый тест основываясь на заполнении видеопамяти одним цветом (константой). Для этого отредактируем файл **src/main.c**, добавим приведенный ниже код в тело функции **main()** сразу после кода инициализации SRAM. Вместо стандартной «сине-зеленой» палитры мы будем использовать более приятную глазу «красно-сине-зеленую», таблица со значениями для которой приведена в массиве **rgb_palette**:

```
void main() {

        ...

        printf("Hardware init\r\n");

        // Test SRAM and initialize heap for malloc to use SRAM if tested OK
        if(sram_test_write_random_ints(10) == 0) {
           ...
        } else {
                printf("SRAM %s!\r\n", "disabled");
        }

        // Init CGA: enable graphics mode and load color palette
        cga_set_video_mode(CGA_MODE_GRAPHICS1);

        static uint32_t rgb_palette[16] = {
                        0x00000000, 0x000000f0, 0x0000f000, 0x00f00000,
                        0x0000f0f0, 0x00f000f0, 0x00f0f000, 0x00f0f0f0,
                        0x000f0f0f, 0x000f0fff, 0x000fff0f, 0x00ff0f0f,
                        0x000fffff, 0x00ff0fff, 0x00ffff0f, 0x00ffffff,
                        };

        cga_set_palette(rgb_palette);

        cga_wait_vblank();

        cga_fill_screen(2); // use color #2 (green)

        ...
```

Теперь можно скомпилировать тестовое приложение командой **make**:

```
rz@devbox:~/VexRiscvForKarnix/src/main/c/karnix_extended_test$ make clean && make
```

Результат успешной компиляции будет примерно следующий:

```
Memory region         Used Size  Region Size  %age Used

             RAM:       64976 B        72 KB     88.13%

/opt/riscv64/bin/riscv64-unknown-elf-objcopy -O ihex build/karnix_extended_test.elf build/karnix_extended_test.hex

/opt/riscv64/bin/riscv64-unknown-elf-objcopy -O binary build/karnix_extended_test.elf build/karnix_extended_test.bin

hexdump -v -e '/4 "%08X\n"' < build/karnix_extended_test.bin > build/karnix_extended_test.hexx

make inc_build_num

make[1]: Entering directory '/home/rz/VexRiscvForKarnix/src/main/c/karnix_extended_test'

make[1]: Leaving directory '/home/rz/VexRiscvForKarnix/src/main/c/karnix_extended_test'

/opt/riscv64/bin/riscv64-unknown-elf-objdump -S -d build/karnix_extended_test.elf > build/karnix_extended_test.asm

/opt/riscv64/bin/riscv64-unknown-elf-objcopy -O verilog build/karnix_extended_test.elf build/karnix_extended_test.v
```

Это сообщает нам, что файл с машинным кодом **build/karnix_extended_test.hexx** готов для *реинтеграции* в битстрим. Выполнить реинтеграцию можно следующей командой из рабочего каталога для синтеза:

```
rz@devbox:~/VexRiscvForKarnix/scripts/KarnixExtended$ make -f Makefile.Briey compile

ecpbram -v -i bin/BrieyForKarnixTopLevel_random_25F.config -o bin/BrieyForKarnixTopLevel_25F.config -f ../../BrieyForKarnixTopLevel_random.hexx -t ../../src/main/c/karnix_extended_test/build/karnix_extended_test.hexx

Padding to_hexfile from 15220 words to 18432

Loaded pattern for 32 bits wide and 18432 words deep memory.

Extracted 1152 bit slices from from/to hexfile data.

ecppack --svf bin/BrieyForKarnixTopLevel_25F.svf bin/BrieyForKarnixTopLevel_25F.config bin/BrieyForKarnixTopLevel_25F.bit
```

Несколько слов о том, что такое *реинтеграция в битстрим*. Напомню, что у нас есть уже готовый файл с битстримом **bin/BrieyForKarnixTopLevel_25F.bit** полученный в результате синтеза из Verilog кода, который в свою очередь генерируется из SpinalHDL. В коде описывающий СнК имеется определение области ОЗУ (синтезированная RAM) и её заполнение готовым машинным кодом. Этот код читается из файла **../../BrieyForKarnixTopLevel_random.hex** на стадии синтеза и интегрируется в результирующий битстрим для заданной микросхемы ПЛИС. Таким образом, после старта и инициализации микросхемы ПЛИС, в синтезированной RAM образуется готовая к

исполнению программа, именно её исполняет вычислительное ядро. После того, как мы изменили код этой программы на языке Си и пересобрали новый бинарный файл, обычно следует выполнить полный цикл синтеза, чтобы новый код был заново интегрирован в битстрим. Однако этот процесс может занимать весьма существенное время. Чтобы не заниматься пересинтезированием проекта каждый раз при изменении исполняемой программы, в тулчейне имеется утилита **ecpbrm** позволяющая подменить содержимое синтезированной RAM в готовом битстриме на новое. Это позволяет существенно сократить время на сборку проекта если аппаратная его часть не изменялась, а требуется заменить только машинный код. Именно эту процедуру выполняет команда **make compile**, в результате выполнения которой формируется файл бистрима **bin/BrieyForKarnixTopLevel_25F.bit** с уже новым машинным кодом, размещенным в синтезированной RAM.

Настало время подключить плату «Карно», загрузить в неё получившийся файл с битстримом и проверить окрасился ли экран монитора в зеленый цвет. Загрузить получившийся битстрим можно следующей командой:

```
rz@devbox:~/VexRiscvForKarnix/scripts/KarnixExtended$ make -f Makefile.Briey prog
```

8.4. Измеряем скорости записи в видеопамять

Перед тем как продолжить изыскания с получившимся простейшим видеоадаптером, неплохо бы выяснить производительность видеопамяти на запись. Для этой цели немного модифицируем предыдущий тест: сразу после загрузки регистров палитры добавим цикл с многократным заполнением видеопамяти случайным цветом и выполним замер времени его исполнения. Время исполнения будем замерять функцией **get_mtime()** которая по сути возвращает текущее значение встроенного в ядро миллисекундного таймера:

```
#ifdef CGA_MEM_TEST1
printf("Executing CGA video framebuffer write performance test...\r\n");

csr_clear(mstatus, MSTATUS_MIE); // Disable Machine interrupts during test
uint32_t cga_t0 = get_mtime();
for(int i = 0; i < 1000; i++) {
        cga_fill_screen(rand()); // use random color
}
uint32_t cga_t1 = get_mtime();
csr_set(mstatus, MSTATUS_MIE); // Enable Machine interrupts after test

printf("CGA framebuffer write perf: %ld uS after 1000 frames\r\n", cga_t1 - cga_t0);
#endif
```

В начале файла **main.c** определим константу **CGA_MEM_TEST1** для включения (выключения) этого теста CGA адаптера:

```
#define CGA_MEM_TEST1
```

Выполним сначала сборку Си программы, затем реинтеграцию машинного кода в битстрим, и еще раз выполним загрузку битстрима в плату Карно, перед этим подключив терминал к виртуальному UART порту #1 платы чтобы наблюдать за выводом отладочных сообщений в терминал:

Для ОС FreeBSD:

```
rz@butterfly:~ % sudo minicom -b 115200 -D /dev/ttyU1
```

Для ОС Linux:

```
rz@devbox:~$ sudo minicom -b 115200 -D /dev/ttyUSB1
```

В окне терминала наблюдаем следующий вывод тестовой программы:

```
Karnix ASB-254 test prog. Build 00876, date/time: Sep  3 2024 15:46:26
Copyright (C) 2021-2024 Fabmicro, LLC., Tyumen, Russia.

Hardware init
Filling SRAM at: 0x90000000, size: 524288 bytes...
Checking SRAM at: 0x90000000, size: 524288 bytes...
...
Filling SRAM at: 0x90000000, size: 524288 bytes...
Checking SRAM at: 0x90000000, size: 524288 bytes...
Enabling SRAM...
SRAM enabled!
Filling video RAM at: 0xf0040000, size: 19200 bytes...
Checking video RAM at: 0xf0040000, size: 19200 bytes...
cga_set_video_mode: mode = 1, ctrl = 0x81080000
Executing CGA video framebuffer write performance test...
CGA framebuffer write perf: 320568 uS after 1000 frames
...
```

Результат измерения скорости записи в видеопамять указан в выделенной строке, а именно: **320568 мкс** за **1000** кадров или **0,321 мс** на кадр, что примерно соответствует 3115 FPS при тактовой частоте ядра 60 МГц. Цифра, на первый взгляд, достаточно большая, но какая она должна быть на самом деле ?

Давайте из любопытства посмотрим на ассемблерный код функции **cga_fill_screen()** который сгенерировал нам компилятор GCC:

```
8000622e <cga_fill_screen>:
8000622e:       890d                    andi    a0,a0,3
80006230:       01e51793                slli    a5,a0,0x1e
80006234:       01c51713                slli    a4,a0,0x1c
80006238:       8fd9                    or      a5,a5,a4
8000623a:       01a51713                slli    a4,a0,0x1a
8000623e:       8fd9                    or      a5,a5,a4
80006240:       0562                    slli    a0,a0,0x18
80006242:       8fc9                    or      a5,a5,a0
80006244:       0187d713                srli    a4,a5,0x18
80006248:       0087d693                srli    a3,a5,0x8
8000624c:       0107d613                srli    a2,a5,0x10
80006250:       8ed1                    or      a3,a3,a2
80006252:       8fd5                    or      a5,a5,a3
80006254:       8f5d                    or      a4,a4,a5
80006256:       f00407b7                lui     a5,0xf0040
8000625a:       f0045637                lui     a2,0xf0045
8000625e:       ae060613                addi    a2,a2,-1312 # f0044ae0 <_ram_heap_end+0x70032ae0>
80006262:       a011                    j       80006266 <cga_fill_screen+0x38>
80006264:       87b6                    mv      a5,a3
80006266:       c398                    sw      a4,0(a5)
80006268:       c3d8                    sw      a4,4(a5)
8000626a:       c798                    sw      a4,8(a5)
8000626c:       c7d8                    sw      a4,12(a5)
8000626e:       cb98                    sw      a4,16(a5)
80006270:       cbd8                    sw      a4,20(a5)
80006272:       cf98                    sw      a4,24(a5)
```

```
80006274:       02078693                addi    a3,a5,32 # f0040020 <_ram_heap_end+0x7002e020>
80006278:       cfd8                    sw      a4,28(a5)
8000627a:       fec795e3                bne     a5,a2,80006264 <cga_fill_screen+0x36>
8000627e:       8082                    ret
```

Здесь тело цикла, обозначенное выше жирным шрифтом, содержит восемь последовательных инструкций записи 32-х битных регистров в память, всего 10 инструкций (т. е. 10 машинных тактов) на запись блока из 32 байт. Если пересчитать это во время исполнения, то получим 10 * (19200 / 32) / 60 000 000 = ~ 100 мкс на один кадр, что сильно отличается от практических замеров — более чем в три раза. Это объясняется тем, что доступ к видеопамяти осуществляется с задержкой и процессор выполняет холостые циклы в ожидании выполнения операции записи.

Следующий тест — это измерение скорости копирования из внутренней синтезируемой RAM в область видеопамяти. Напомню, что синтезированная RAM работает на максимально возможной частоте равной частоте вычислительного ядра и позволяет за один такт считывать или записывать сразу 32 бита данных. Выясним, какую скорость копирования видео кадров мы можем ожидать от нашей системы. Для этого скопируем предыдущий тест и немного модернизируем его, назовём этот тест **CGA_MEM_TEST2**:

```
#ifdef CGA_TEST2
printf("Executing CGA video framebuffer copy from RAM performance test...\r\n");

static char test_buffer[960];

for(int i = 0; i < 960; i++)
       test_buffer[i] = rand();

csr_clear(mstatus, MSTATUS_MIE); // Disable Machine interrupts during test
uint32_t cga_t2 = get_mtime();
for(int i = 0; i < 1000; i++) {
        for(int j = 0; j < 20; j++)
                memcpy(CGA->FB + j * 960, test_buffer, 960);

}
uint32_t cga_t3 = get_mtime();
csr_set(mstatus, MSTATUS_MIE); // Enable Machine interrupts after test

printf("CGA framebuffer copy from RAM perf: %ld uS after 1000 frames\r\n", cga_t3 - cga_t2);
#endif
```

Скомпилируем новый бинарный файл с машинным кодом, реинтегрируем его в битстрим и загрузим в ПЛИС. В результате в терминале получим следующий вывод:

```
Karnix ASB-254 test prog. Build 00879, date/time: Sep  3 2024 17:54:03
Copyright (C) 2021-2024 Fabmicro, LLC., Tyumen, Russia.

Hardware init
Filling SRAM at: 0x90000000, size: 524288 bytes...
Checking SRAM at: 0x90000000, size: 524288 bytes...
...
Filling SRAM at: 0x90000000, size: 524288 bytes...
Checking SRAM at: 0x90000000, size: 524288 bytes...
Enabling SRAM...
SRAM enabled!
Filling video RAM at: 0xf0040000, size: 19200 bytes...
Checking video RAM at: 0xf0040000, size: 19200 bytes...
cga_set_video_mode: mode = 1, ctrl = 0x81080000
Executing CGA video framebuffer write performance test...
CGA framebuffer write perf: 320568 uS after 1000 frames
```

```
Executing CGA video framebuffer copy from RAM performance test...
CGA framebuffer copy from RAM perf: 357479 uS after 1000 frames
...
```

Видно что данный тест дает нам время исполнения в среднем **0,357 мс** на кадр. Видно, что показания теста CGA_MEM_TEST2 отличаются от показаний CGA_MEM_TEST1 примерно на **37 мкс** на кадр в сторону увеличения, что вполне логично — в коде добавилось чтение значения из памяти. Давайте посмотрим на ассемблерный код. Так как горячая часть кода этого теста представляет собой тело функции **memcpy()**, то будем смотреть сразу в него. И вот, что мы увидим:

```
8000c744 <memcpy>:
8000c744:       00a5c7b3                xor     a5,a1,a0
8000c748:       0037f793                andi    a5,a5,3
8000c74c:       00c50733                add     a4,a0,a2
8000c750:       00079663                bnez    a5,8000c75c <memcpy+0x18>
8000c754:       00300793                li      a5,3
8000c758:       02c7e263                bltu    a5,a2,8000c77c <memcpy+0x38>
8000c75c:       00050793                mv      a5,a0
8000c760:       0ae57c63                bgeu    a0,a4,8000c818 <memcpy+0xd4>
8000c764:       0005c683                lbu     a3,0(a1)
8000c768:       00178793                addi    a5,a5,1
8000c76c:       00158593                addi    a1,a1,1
8000c770:       fed78fa3                sb      a3,-1(a5)
8000c774:       fee7e8e3                bltu    a5,a4,8000c764 <memcpy+0x20>
8000c778:       00008067                ret
8000c77c:       00357693                andi    a3,a0,3
8000c780:       00050793                mv      a5,a0
8000c784:       00068e63                beqz    a3,8000c7a0 <memcpy+0x5c>
8000c788:       0005c683                lbu     a3,0(a1)
8000c78c:       00178793                addi    a5,a5,1
8000c790:       00158593                addi    a1,a1,1
8000c794:       fed78fa3                sb      a3,-1(a5)
8000c798:       0037f693                andi    a3,a5,3
8000c79c:       fe9ff06f                j       8000c784 <memcpy+0x40>
8000c7a0:       ffc77693                andi    a3,a4,-4
8000c7a4:       fe068613                addi    a2,a3,-32
8000c7a8:       06c7f463                bgeu    a5,a2,8000c810 <memcpy+0xcc>
8000c7ac:       0005a383                lw      t2,0(a1)
8000c7b0:       0045a283                lw      t0,4(a1)
8000c7b4:       0085af83                lw      t6,8(a1)
8000c7b8:       00c5af03                lw      t5,12(a1)
8000c7bc:       0105ae83                lw      t4,16(a1)
8000c7c0:       0145ae03                lw      t3,20(a1)
8000c7c4:       0185a303                lw      t1,24(a1)
8000c7c8:       01c5a883                lw      a7,28(a1)
8000c7cc:       02458593                addi    a1,a1,36
8000c7d0:       0077a023                sw      t2,0(a5)
8000c7d4:       ffc5a803                lw      a6,-4(a1)
8000c7d8:       0057a223                sw      t0,4(a5)
8000c7dc:       01f7a423                sw      t6,8(a5)
8000c7e0:       01e7a623                sw      t5,12(a5)
8000c7e4:       01d7a823                sw      t4,16(a5)
8000c7e8:       01c7aa23                sw      t3,20(a5)
8000c7ec:       0067ac23                sw      t1,24(a5)
8000c7f0:       0117ae23                sw      a7,28(a5)
8000c7f4:       02478793                addi    a5,a5,36
8000c7f8:       ff07ae23                sw      a6,-4(a5)
8000c7fc:       fadff06f                j       8000c7a8 <memcpy+0x64>
8000c800:       0005a603                lw      a2,0(a1)
8000c804:       00478793                addi    a5,a5,4
8000c808:       00458593                addi    a1,a1,4
8000c80c:       fec7ae23                sw      a2,-4(a5)
8000c810:       fed7e8e3                bltu    a5,a3,8000c800 <memcpy+0xbc>
8000c814:       f4e7e8e3                bltu    a5,a4,8000c764 <memcpy+0x20>
8000c818:       00008067                ret
```

Видно, что основное тело цикла копирования состоит из восьми последовательных операций чтения в регистры из исходного блока памяти (синтезируемой RAM), следом за которыми идут восемь операций записи регистров в блок память назначения (видеопамять). Иными словами, к операции записи теста CGA_MEM_TEST1 добавились еще восемь операций записи и одна операция сложения, в пересчете на время исполнения это добавит около 9 тактов * 30 итераций * 20 циклов / 60 МГц = 90 мкс. На практике же мы получаем увеличение времени исполнения на гораздо меньшую величину - на 37 мкс на каждый кадр. Объяснить это можно работой кэша данных, который может заполняться полезными данными в то время, пока другая часть шины AXI/APB3 ожидает записи в видеопамять.

И еще один, третий, тест который мы проведем — это копирование блока данных из области внешней статической памяти в область видеопамяти. Дело в том, что на плате «Карно» установлена микросхема внешней статической памяти имеющая шину данных в 16 разрядов, а значит шине AXI для доступа к 32-х битным словам требуется выполнять по две транзакции. Более того, практика показала, что используемая микросхема SRAM не может работать на частоте 60 МГц и требует задержки в один дополнительный цикл при операциях чтения. Таким образом, суммарное время доступа к 32 битам расположенным в SRAM будет **в четыре раза больше** чем к 32 битам расположенным в синтезированной RAM.

Давайте модифицируем предыдущий тест, изменим область расположения исходных данных (SRAM располагается с адреса 0x90000000), и посмотрим на сколько всё плохо. Или может быть кэш сгладит эту проблему ? Код третьего теста (**CGA_MEM_TEST3**) приведен ниже:

```
	#ifdef CGA_MEM_TEST3
	printf("Executing CGA video framebuffer copy from external SRAM performance test...\r\n");

	char *sram_test_buffer = (char*) 0x90000000;

	for(int i = 0; i < 960; i++)
		sram_test_buffer[i] = rand();

	csr_clear(mstatus, MSTATUS_MIE); // Disable Machine interrupts during test
	uint32_t cga_t4 = get_mtime();
	for(int i = 0; i < 1000; i++) {
		for(int j = 0; j < 20; j++)
			memcpy(CGA->FB + j * 960, sram_test_buffer, 960);

	}
	uint32_t cga_t5 = get_mtime();
	csr_set(mstatus, MSTATUS_MIE); // Enable Machine interrupts after test

	 printf("CGA framebuffer copy from external SRAM perf: %ld uS after 1000 frames\r\n", cga_t5
- cga_t4);
	#endif
```

После компиляции, реинтеграции и загрузки битстрима в ПЛИС получим следующий результат:

```
Karnix ASB-254 test prog. Build 00884, date/time: Sep  3 2024 19:39:30
Copyright (C) 2021-2024 Fabmicro, LLC., Tyumen, Russia.

Hardware init
Filling SRAM at: 0x90000000, size: 524288 bytes...
Checking SRAM at: 0x90000000, size: 524288 bytes...
...
Filling SRAM at: 0x90000000, size: 524288 bytes...
Checking SRAM at: 0x90000000, size: 524288 bytes...
Enabling SRAM...
SRAM enabled!
```

```
Filling video RAM at: 0xf0040000, size: 19200 bytes...
Checking video RAM at: 0xf0040000, size: 19200 bytes...
cga_set_video_mode: mode = 1, ctrl = 0x81080000
Executing CGA video framebuffer write performance test...
CGA framebuffer write perf: 320568 uS after 1000 frames
Executing CGA video framebuffer copy from RAM performance test...
CGA framebuffer copy from RAM perf: 357480 uS after 1000 frames
Executing CGA video framebuffer copy from external SRAM performance test...
CGA framebuffer copy from external SRAM perf: 357487 uS after 1000 frames
```

И что же ми видим ? Время копирования блока данных из внешней SRAM в видеопамять точно такое же, как и при копировании аналогичного блока из синтезируемой RAM, и составляет **357 мкс** на кадр. Вот так фокус!

Какие выводы можно сделать из полученных результатов? Вероятнее всего, при прямой отрисовке битмапов (спрайтов) в видеопамять всё будет очень печально, однако время полного копирования экрана равное 357 мкс дает нам надежду на возможность использования двойного буфера для вывода графических изображений, так как время затенения (обратного хода луча) составляет **3,84 мс**, а значит у нас есть шанс обновить содержимое видеопамяти за период затенения и избежать появление неприятных глазу артефактов.

Примечание. В СнК Briey вычислительное ядро сконфигурировано на использование сокращенных инструкций («C extention» или «compressed instruction set»), поэтому часть инструкций в приведенном выше машинном коде имеет размер 16 бит, а не 32 бита.

8.5. Программная реализация функция Bitblit и замер её производительности

Bit blit (BITBLT, BIT BLT, BitBLT, Bit BLT, Bit Blt и т. д.) — производная от «bit block transfer», так называют функцию для объединения двух и более областей видеоизображения (битмапов) в одно, используя булевую операцию. Обычно функция Bitblit принимает на вход два массива данных, выполняет операцию над ними и записывает в третий. Часто одним из массивов является область видеопамяти уже заполненная каким-то изображением, второй массив с помощью булевой операции объединяется с первым и помещается в ту же область видеопамяти, таким образом получают наложение одного изображение на другое. Расширенный вариант Bitblit может принимать третий массив который содержит маску указывающую какие из элементов подлежат наложению/объединению.

Функция Bitblit может быть реализована как программно, так и аппаратно. Например, графический чипсет на ПЭВМ Commodore Amiga позволял аппаратно объединять до трех битмапов применяя к ним любую из 256-ти возможных 3-входовых булевых операций, что давало огромные возможности разработчикам видео игр, упрощало их труд и ускоряло процесс отображения графики. В современных видеоадаптерах битовые операции к данным цвета не применяются, так как результат не дает привычного глазу визуального эффекта смешения цветов, вместо этого используют другие алгоритмы (alpha compositing или alpha blending). Тем не менее Bitblit, в том или ином виде, присутствует во всех современных

видеоадаптерах. Ученые затрудняются ответить на вопрос кто и когда первым применил Bitblit, но некоторые археологи уверены, что это было сделано в Xerox PARC при разработке графического интерфейса для компьютера Xerox Alto.

Видеоадаптер IBM CGA не обладал возможностью аппаратно выполнять операции над битовыми картами и разработчикам графических программ приходилось самостоятельно реализовывать функцию программного считывания, модификации и записи слов видеопамяти. Очевидно, что чем быстрее отрабатывает эта функция, тем больше времени остается у программы на остальные дела (или тем большее количество битмапов она может вывести на экран), поэтому основным критерием оценки реализации Bitblit является среднее число тактов затрачиваемое процессором на обработку/вывод одного пикселя (или PEL-а).

Прежде чем приступить к разработке функции Bitblit необходимо договориться о формате представления изображения в битмапах и об ограничениях, которые вытекают из этого формата. Классическим форматом представления изображения в памяти является представление с разбиением на битовые плоскости. В таком формате каждый бит индекса цвета записывается в отдельную область памяти называемую «битовая плоскость» (bit plane). Для хранения изображения используется столько же битовых плоскостей, сколько битов требуется для представления одного значения индекса цвета. К примеру, если цвет пикселя кодируется четырьмя битами (значениями от 4“b0000 до 4“b1111), то для хранения изображения используют четыре отдельных битовых плоскости, каждая содержит по одному биту из каждого пикселя. Такой формат был удобен при реализации аппаратуры в видеоадаптерах древности, таких как EGA или VGA, а также на Amiga и Atari ST. Это позволяло легко проводить различные булевые операции над отдельными битами, сдвигать их при необходимости, а также хранить отдельные плоскости в раздельных микросхемах памяти (в то время микросхемы памяти были «однобитными»). Поэтому, чтобы не тратить лишние машинные циклы на преобразование (а это процесс назывался «bit slicing»), изображения хранили уже в распакованном, разбитом на отдельные плоскости, состоянии.

Другим способом представление битмапа является представление в виде горизонтальных строк пикселей, биты цветности которых следуют друг за другом непрерывно. Такой формат называется «упакованный» (packed). В современных видеосистемах цвет пикселя требует от 4-х до 6-ти байт (то есть по 8, 10 и 12 бит на компонент цвета) и тоже относится считается «упакованным». В оригинальном CGA в графическом режиме 320x200 использовался упакованный формат, при этом в один байт помещалась информация о 4-х PEL-ах, по два бита на индекс.

В нашем CGA-подобном видеоадаптере мы храним информацию об индексе цвета в упакованном виде, при этом доступ к данным видеопамяти внутри адаптера осуществляется словами по 32 бита за один такт. Отсюда логичным было бы хранить битмап в виде кусочков по 32 бита (4 байта), но это же целых 16 пикселей! Такая гранулярность изображения не очень удобна, она не позволит нам отображать текст и небольшие пиктограммы. Поэтому, немного усложним себе задачу — будем хранить изображение битмапа кусочками по 4 пикселя (то есть байтами), а функцию Bitblit научим оперировать с битовыми картами любого размера.

Таким образом наша функция вывода битмапа, назовем её **cga_bitblit()**, будет работать с двумя битовыми картами — параметры **src_img** и **dst_img** указывают на области памяти типа **uint8_t***. Также **cga_bitblit()** будет принимать на вход два целочисленных параметра с координатами **x** и **y** для задания места вывода в экранных координатах (PEL-ах). Еще два целочисленных параметра **src_width** и **src_height** для указания ширины (в точках) и высоты (в строках) выводимой битовой карты, а также два целочисленных параметра **dst_width** и **dst_height** для задания размеров экранной области на которую ссылается **dst_img**. Прототип функции **cga_bitblit()** будет выглядеть следующим образом:

```
void cga_bitblit(uint8_t *src_img, uint8_t *dst_img, int x, int y, int src_width, int
               src_height, int dst_width, int dst_height);
```

Теперь немного поговорим о реализации функции **cga_bitblit()**. В целом, она достаточно простая: два цикла, один внутри другого, итерируют по строкам (**row**) и по столбцам (**col**). Внутри цикла итерирующего по столбцам происходит считывание данных из битмапа и из видеопамяти, производится расчет битовой маски **pixel_mask**, наложение данных с её помощью, и запись результирующих данных **pixel_word** обратно в видеопамять. Но тут есть пара нюансов. Во-первых, надо учитывать что оптимальный доступ к памяти в нашей вычислительной системе осуществляется словами по 32 бит. Это означает, что при реализации вывода мы должны будем считывать по одному слову размером 32 бита из видеопамяти и из битмапа, выполнять манипуляции с этими данными и записывать результат обратно в видеопамять целиком по 32 бита. Иными словами, мы должны будем вести обработку сразу 16-ти пикселей, по два бита каждый. Если производить манипуляции отдельно для каждого пикселя, то эффективность работы функции вывода понизится более чем в 16 раз, а это для нас недопустимо. Во-вторых, нужно учитывать границы экрана — совершенно бессмысленно производить манипуляции и выводить изображение если оно выходит за пределы видимой экранной области и тем более не попадает в область видеопамяти. Проверку и ограничение границ требуется сделать один раз в самом начале, а не в теле цикла. Слегка оптимизированный вариант реализации функции **cga_bitblit()** приведен ниже:

```
void cga_bitblit(uint8_t *src_img, uint8_t *dst_img, int x, int y, int src_width, int src_height,
                int dst_width, int dst_height) {

        uint32_t* dst = (uint32_t*)dst_img;
        uint8_t* src = (uint8_t*)src_img;

        int pixel_idx, pixel_offset, word_idx, word_idx_prev = -1;
        uint32_t pixel_word, pixel_mask;

        int src_pixel_offset, src_word_idx;
        int src_stride = (src_width >> 2);
        uint32_t src_word;

        int col_start = 0;
        int col_end = src_width;

        // Y border limits
        if(y >= dst_height || y + src_height <= 0)
                return;
```

```
	if(y < 0) {
		src += -y * src_stride;
		src_height += y;
		y = 0;
	}

	if(y + src_height > dst_height)
		src_height = dst_height - y;

	// X border limits
	if(x >= dst_width || x + src_width <= 0)
		return;

	if(x < 0)
		col_start = -x;

	if(x + src_width > dst_width)
		col_end = dst_width - x;

	int pixel_idx_0 = y * dst_width + x;

	for(int row = 0; row < src_height; row++) {

		for(int col = col_start; col < col_end; col++) {

			pixel_idx = pixel_idx_0 + col;
			pixel_offset = (15 - (pixel_idx & 0xf)) << 1;

			word_idx = pixel_idx >> 4; // 16 pixels per 32 bit word

			pixel_mask = 0x03 << pixel_offset;

			if(word_idx != word_idx_prev) {
				if(word_idx_prev != -1)
					dst[word_idx_prev] = pixel_word;

				pixel_word = dst[word_idx];
			}

			// Get image data
			src_word_idx = col >> 2; // 4 pixels per byte
			src_pixel_offset = (3 - (col & 0x03)) << 1;
			src_word = src[src_word_idx];

			pixel_word &= ~pixel_mask;

			if(pixel_offset >= src_pixel_offset)
				src_word <<= (pixel_offset - src_pixel_offset);
			else
				src_word >>= (src_pixel_offset - pixel_offset);

			pixel_word |= src_word & pixel_mask;

			word_idx_prev = word_idx;
		}

		src += src_stride;
		pixel_idx_0 += dst_width;
	}
	dst[word_idx] = pixel_word;
}
```

Чтобы протестировать работу функцию **cga_bitblit()** и её производительность выполним ряд тестов. Первый тест будет состоять из прямого заполнения видеопамяти без предварительной буферизации, добиваясь максимальной скорости заполнения. Для этого в цикле с помощью **cga_bitblit()** будем заполнять весь экран одним и тем же битмапом (спрайтом), а с каждым кадром будем менять спрайт по кругу, таким образом анимируя

изображение чтобы было видно артефакты если таковые возникнут. Я подготовил серию из одиннадцати спрайтов размером 16х16 пикселей приведенных на рис. 12 ниже.

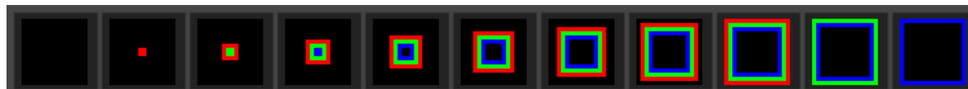


Рис. 12. Серия спрайтов размером 16х16 точек 4 цвета для тестирования функции cga_bitblit().

Исходный код функции **cga_video_test1()** для тестирования прямой записи в видеопамять приведен ниже.

```
static int _x = 0, _y = 0;
static int _sprite_idx = 0;

void cga_video_test1(void) {

        uint32_t t0 = get_mtime() & 0xffffffff;


        for(int y = 0; y < 240; y += 16)
                for(int x = 0; x < 320; x += 16)
                        cga_bitblit((uint8_t*)cga_sprites[_sprite_idx], CGA->FB, x + _x, y + _y, 16, 
16,
                                        CGA_VIDEO_WIDTH, CGA_VIDEO_HEIGHT);


        uint32_t t1 = get_mtime() & 0xffffffff;

        printf("cga_video_test1: t = %lu uS\r\n", t1 - t0);

        _sprite_idx = (_sprite_idx + 1) % 11;
}
```

Данная функция имеет три статических (глобальных) переменных: **_sprite_idx** задает номер отображаемого спрайта в серии, а **_x** и **_y** задают смещение по соответствующим осям координат. Изменяя **_x** и **_y** из главного цикла программы можно перемещать всё изображение по экрану.

Вставим вызов функции **cga_video_test1()** из главного цикла программы, соберем и реинтегрируем бинарный файл, загрузим битстрим в ПЛИС и посмотрим что у нас получилось. В терминале как обычно будем наблюдать сначала результаты предыдущих тестов:

```
Karnix ASB-254 test prog. Build 00905, date/time: Sep  4 2024 20:14:29
Copyright (C) 2021-2024 Fabmicro, LLC., Tyumen, Russia.

Hardware init
Filling SRAM at: 0x90000000, size: 524288 bytes...
Checking SRAM at: 0x90000000, size: 524288 bytes...
...
Enabling SRAM...
SRAM enabled!
Filling video RAM at: 0xf0040000, size: 19200 bytes...
Checking video RAM at: 0xf0040000, size: 19200 bytes...
cga_set_video_mode: mode = 1, ctrl = 0x81080000
Executing CGA video framebuffer write performance test...
CGA framebuffer write perf: 320568 uS after 1000 frames
Executing CGA video framebuffer copy from RAM performance test...
CGA framebuffer copy from RAM perf: 357479 uS after 1000 frames
```

```
Executing CGA video framebuffer copy from external SRAM performance test...
CGA framebuffer copy from external SRAM perf: 357487 uS after 1000 frames
...
```

После чего мы увидим результат исполнения функции **cga_video_test1()**, который циклически отображает время в микросекундах требуемое на заполнение всей экранной области одним и тем же спрайтом.

```
cga_set_video_mode: mode = 1, ctrl = 0x81000000
cga_video_test1: t = 35912 uS
cga_video_test1: t = 35920 uS
cga_video_test1: t = 35919 uS
cga_video_test1: t = 35917 uS
cga_video_test1: t = 36095 uS
cga_video_test1: t = 36255 uS
...
```

А на экране монитора при этом будем наблюдать анимированный «муар» (см. рис. 13).

Переведя измеренное время заполнения экранной области (**36 мс**) в частоту, получим примерно **28 FPS**. Результат вполне достойный, учитывая что вся аппаратура была нами же синтезирована без всяких хитростей и оптимизаций. Но если присмотреться к анимированному изображению на мониторе, то можно легко заметить один неприятный эффект — при анимации часть изображения имеет сдвиг (или «излом») вызванный тем, что мы осуществляем запись в экранную область в тот момент когда наш видеоадаптер производит отрисовку изображения на мониторе. Это типичный «глитч» для такого метода работы с видео фреймбуфером. Избавиться от него можно двумя путями: 1) изменять содержимое видеопамяти только в период затенения, или 2) проводить отрисовку изображения в отдельный буфер в памяти, а по окончанию отрисовки дождаться момента начала периода затенения и быстро скопировать всё изображение из буфера в видеопамять.

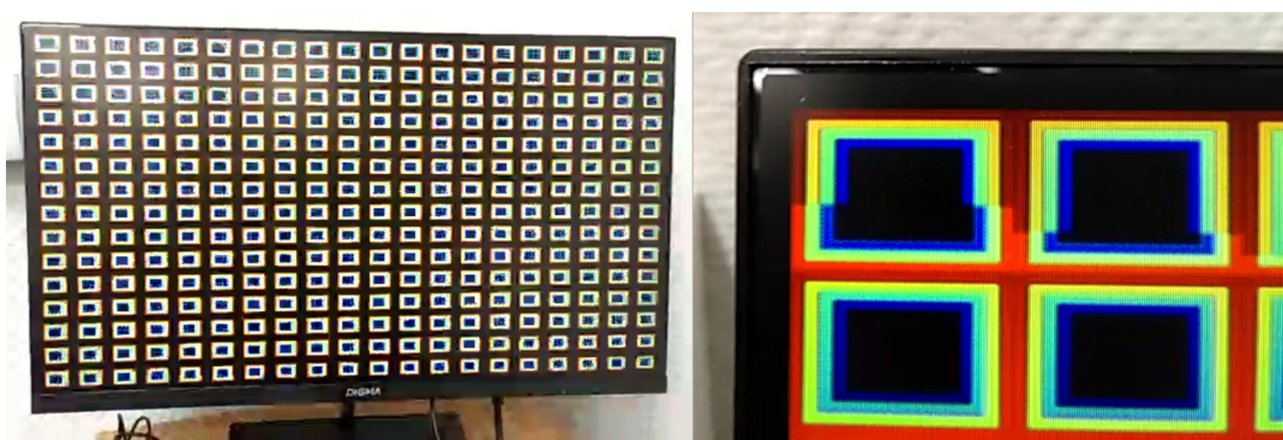


Рис. 13. Изображение полученное в результате работы функции cga_video_test1(). Слева — общий вид. Справа — участок с «изломом» картинки.

Первый вариант нам не подходит просто потому, что время отрисовки всего экрана (36 мс) на порядок больше времени периода затенения (3,84 мс). Поэтому приступим к реализации следующего теста с техниками двойной буферизации и ожиданием момента

начала периода затенения. Ниже приведен текст функции **cga_video_test2()** использующей область внешней SRAM памяти с адреса **0x90001000** для временного буфера в который выполняется отрисовка изображения всего экрана. По завершению отрисовки вызывается оберточная функция **cga_wait_vblank()** для ожидания начала момента начала обратного хода луча, после чего содержимое временного буфера полностью копируется в видеопамять вызовом функции **memcpy()**:

```
void cga_video_test2(void) {

        uint32_t t0 = get_mtime() & 0xffffffff;

        for(int y = 0; y < 240; y += 16)
                for(int x = 0; x < 320; x += 16)
                          cga_bitblit((uint8_t*)cga_sprites[_sprite_idx], (uint8_t*)0x90001000, x +
_x, y + _y, 16, 16,
                                        CGA_VIDEO_WIDTH, CGA_VIDEO_HEIGHT);

        cga_wait_vblank();

        memcpy(CGA->FB, (uint8_t*)0x90001000, 19200);

        uint32_t t1 = get_mtime() & 0xffffffff;

        printf("cga_video_test2: t = %lu uS\r\n", t1 - t0);

        _sprite_idx = (_sprite_idx + 1) % 11;
}
```

В главном цикле программы заменим вызов **cga_video_test1()** на **cga_video_test2()**, пересоберем и загрузим новый битстрим в ПЛИС. В результате исполнения данного тесты мы получим в терминале следующие сообщения:

```
Karnix ASB-254 test prog. Build 00907, date/time: Sep  4 2024 22:02:05
Copyright (C) 2021-2024 Fabmicro, LLC., Tyumen, Russia.

Hardware init
Filling SRAM at: 0x90000000, size: 524288 bytes...
Checking SRAM at: 0x90000000, size: 524288 bytes...
...
Enabling SRAM...
SRAM enabled!
Filling video RAM at: 0xf0040000, size: 19200 bytes...
Checking video RAM at: 0xf0040000, size: 19200 bytes...
...
Hardware init done
cga_video_test2: t = 47012 uS
cga_video_test2: t = 47016 uS
cga_video_test2: t = 47009 uS
cga_video_test2: t = 47020 uS
```

Присмотревшись к экрану видим, что «излом» изображения пропал (рис. 14), но при этом частота обновления кадров заметно снизилась и составляет уже **21 FPS** (47 мс на кадр). Объясняется это тем, что ожидание начала периода затенения занимает существенное время — фактически мы ожидаем пока видеоадаптер закончит обновление изображения на экране, то есть выведет в монитор целый кадр (около 10 мс). Задержкой вызванной функцией копирования **memcpy()** можно пренебречь, напомню что она составляет **0,35 мс**.

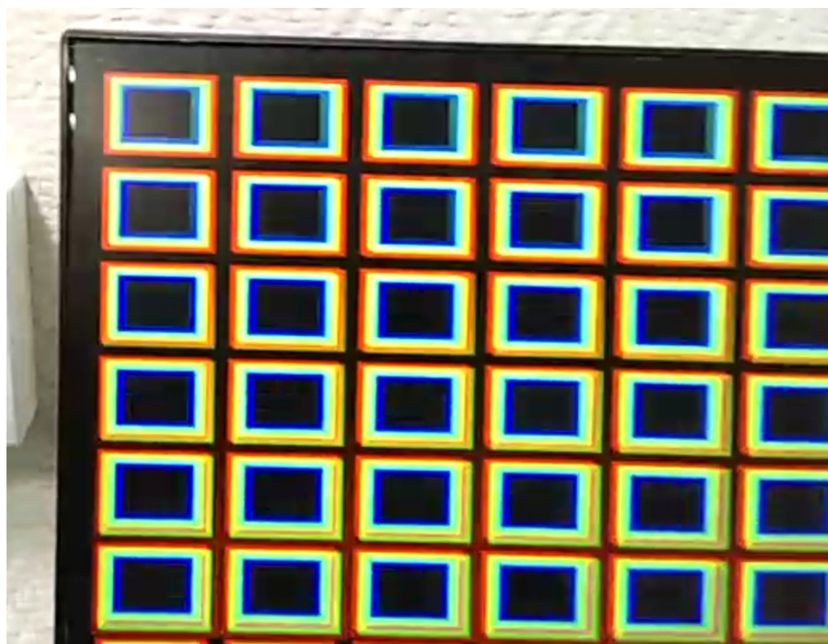


Рис. 14. Фрагмент изображения полученного в результате работы функции cga_video_test2(). «Излом» картинки отсутствует.

Видео с демонстрацией работы теста заполнения видеопамяти с двойной буферизацией можно посмотреть по ссылке: https://www.youtube.com/watch?v=b2RDPzVEt38

На этом с тестированием графического режима нашего CGA-подобного видеоадаптера мы закончим. На данном этапе мы получили кое-какие цифры о производительности разработанной видео подсистемы, а также создали набор примитивов для вывода изображений для языка Си. Мы еще вернемся к графическому режиму когда будем реализовывать плавную вертикальную прокрутку изображения, а пока двинемся дальше.

9. Добавляем текстовый режим

В предыдущих главах мы пришли в выводу о том, что в нашем CGA-подобном видеоадаптере должен быть реализован текстовый режим отображения информации с разрешением 30 строк по 80 знакомест в каждой, при этом каждое знакоместо задается в видеопамяти 32-мя битами, значащими из которых являются только младшие 16 бит кодирующие номер символа (0-255) и цветовые атрибуты - цвет фона и цвет текста, по 4 бита каждый. Для хранения отображаемого текста в текстовом режиме будем использовать всё тот же блок описываемый переменной **fb_mem** размером в 20КБ. Всего в видеопамяти адаптера сможет разместиться 64 строки текста или две страницы с небольшим хвостиком. Размер знакоместа (символа) будет иметь 8 точек в ширину и 16 линий сканирования в высоту, что соответствует полному видеоформату 640x480. Это означает, что частотные характеристики для текстового режима остаются такими же как и для графического, что сильно упрощает аппаратуру — все схемы формирования тактовых сигналов остаются прежними.

Формирование символов будем выполнять аппаратно используя образы символов «зашитые» в процессе синтеза в область памяти которую будем называть «знакогенератор». Вообще, знакогенератор является основой любого текстового видеоадаптера, вся остальная аппаратура является своего рода обвязкой и надстройкой над знакогенератором, поэтому реализацию текстового режима начнем со знакогенератора.

9.1. Знакогенератор

Напомню, что в оригинальном CGA адаптере знакогенератор располагался в микросхеме ПЗУ, не поддавался программной модификации и вообще не был никак доступен из программы. Наша реализация знакогенератора, с целью экономии ресурсов ПЛИС, тоже будет недоступна программно, но это дело можно будет легко поправить.

В файле **./src/main/scala/mylib/CGA4HDMI.scala**, сразу после определения **fb_mem** и её привязки к шине, опишем новый блок памяти для знакогенератора следующим образом:

```
// Define memory block for character generator
val chargen_mem = Mem(Bits(8 bits), wordCount = 16*256 ) // font8x16 x 256
```

Блок **chargen_mem** будет содержать слова шириной 8 бит в количестве 16*256 = 4096 штук, или 4КБ. Каждый бит слова будет представлять маску образа символа (глиф): наличие «1» в бите маски указывает на то, что данный пиксель символа следует отрисовывать цветом заданным для текста, а «0» - соответственно цветом фона. Таблица цветов будет задаваться все той же палитрой, описываемой блоком **palette_mem**, содержащей 16 регистров по 32 бита задающих реальный цвет в формате xBGR (красный «R» расположен в младших 8-ми битах).

Для того, чтобы заполнить блок **chargen_mem** данными знакогенератора, воспользуемся следующей инструментальной функцией из библиотеки **spinal.lib.misc.HexTools**:

```
HexTools.initRam(chargen_mem, charGenHexFile, 0x0l)
```

Данная функция вторым параметром получает переменную указывающую на имя локального файла. В нашем случает это параметр **charGenHexFile**, передаваемый компоненту **Apb3CGA4HDMICtrl** при его инстанциировании, по-умолчанию ему присваивается строка **"font8x16x256.hex"** (см. самое начало главы «*8. Разработка CGA подобного видеоадаптера*»). Это означает, что нам следует поместить двоичные данные знакогенератора, в IntelHEX формате, в файл с этим именем в самый корень подкаталога с проектом. При сборке проекта SpinalHDL загрузит и распарсит содержимое этого файла и сгенерирует соответствующий код на языке Verilog для заполнения переменной **chargen_mem** статическими данными. В главе «*7.3. Адаптируем CGA под современные реалии*» я упоминал, что в составе дистрибутива операционной системы FreeBSD имеется большой набор растровых шрифтов используемых для эмуляции видеотерминала.

В старые добрые времена, когда компьютеры были большими, а память была маленькой, взаимодействие с большим компьютером производилось через видеотерминалы — отдельные устройства содержащие клавиатуру и ЭЛТ монитор. Всё, что пользователь вводил с клавиатуры, как правило, тут же отправлялось в большой компьютер, а ответ компьютера отображался на мониторе в текстовом виде. Позже, с повсеместным проникновением персональных компьютеров, функции видеотерминала стали эмулировать программно, используя встроенную клавиатуру, видеоадаптер и ЭЛТ монитор ПЭВМ. Операционная система FreeBSD тоже умеет эмулировать видеотерминал, что позволяет пользователю взаимодействовать с системой в чисто текстовом виде через командную строку без привлечения графических оболочек (обычно такой режим используют на серверах для их администрирования). Когда мы закончим с реализацией текстового режима для нашего CGA-подобного видеоадаптера, с его помощью мы тоже сможем эмулировать простейший видеотерминал.

Классические видеотерминалы не поддерживали кодирование символом методом UTF-8, каждый отображаемый символ на видеотерминале задавался одним 8-ми битным байтом. Для того, чтобы отображать на экране видеотерминала символы помимо стандартного набора ASCII, который описывает диапазон символов с кодами от 0 до 127, национальный набор символов или символы псевдографики, был разработан большой набор кодировок (кодовых таблиц символов). Отличались они в основном представлением символов с кодами в диапазоне от 128 до 255. Для видеотерминалов предназначенных для работы с ЭВМ совместимых с DEC PDP-11 в нашей стране продолжительное время использовалась кодировка KOI7, на основе которой позже была разработана кодировка KOI8-R содержащая кириллические символы. Кодировку KOI8-R разработал известный программист и первопроходец Интернета в России, сотрудник первого российского интернет-провайдера «ДЕМОС», Андрей Александрович Чернов. Он также состоял в FreeBSD Core Team с 1993 по 2000 годы.

Подключимся к системе FreeBSD и посмотрим в содержимое подкаталога **/usr/share/syscons/fonts/**. В нём мы увидим множество растровых шрифтов различных кодировок с различным размером символа. Сделаем выборку по размеру **8x16** и по кодировке **koi8**:

```
rz@butterfly:~ % freebsd-version
13.3-RELEASE-p2

rz@butterfly:~ % ll /usr/share/syscons/fonts | grep 8x16 | grep koi8
-r--r--r--  1 root  wheel   5676 Mar 12 04:20 koi8-r-8x16.fnt
-r--r--r--  1 root  wheel   5677 Mar 12 04:20 koi8-rb-8x16.fnt
-r--r--r--  1 root  wheel   5677 Mar 12 04:20 koi8-rc-8x16.fnt
-r--r--r--  1 root  wheel   5676 Mar 12 04:20 koi8-u-8x16.fnt
```

Отличным кандидатом является файл **koi8-r-8x16.fnt**.

Если развернутой ОС FreeBSD под рукой не оказалось, то всегда можно заглянуть в её дерево исходных кодов и скачать файл с растровым шрифтом прямо из онлайн репозитория: https://cgit.freebsd.org/src/plain/share/syscons/fonts/koi8-r-8x16.fnt

Файл **koi8-r-8x16.fnt** содержит внутри текст в формате **uuencode** — это способ представления бинарного файла с помощью ограниченного набора печатных символов. Такой способ кодирования раньше широко использовался для вложения бинарных файлов в тело сообщения, отправляемого по электронной почте, в список рассылки или в новостную группу USENET. Подобный метод используется и по сей день для отправки сообщений электронной почты, только алгоритм кодирования немного изменился и называется Base64 — оба алгоритма очень похожи, но несовместимы друг с другом.

Но вернемся к нашему файлу с растровым шрифтом. Для того, чтобы его декодировать (перевести из текстового формата uuencode в двоичный), необходимо воспользоваться утилитой **uudecode** или аналогичной. В ОС FreeBSD данная утилита входит в стандартный набор утилит, а в ОС на основе ядра Linux данную утилиту можно получить установкой в систему пакета с именем **sharutils** следующей командой:

```
$ sudo apt install sharutils
```

Параметры вызова утилиты **uudecode** очень простые — необходимо указать имя выходного (бинарного) файла через опцию **-o <filename.bin>**, а рядом указать имя исходного закодированного файла. Либо выдать содержимое закодированного файла во входной поток этой утилите и далее она сама во всём разберется. Иными словами, декодировать файл с растровым шрифтом можно следующей командой:

```
$ uudecode -o koi8-r-8x16.bin /usr/share/syscons/fonts/koi8-r-8x16.fnt

$ ls -al koi8-r-8x16.bin
-rw-r--r--  1 rz  rz  4096 Sep  8 04:35 koi8-r-8x16.bin
```

Как видно, бинарный файл **koi8-r-8x16.bin** содержит ровно 4 КБ растровых данных (16*256).

Следующим шагом нам потребуется перевести бинарный файл в формат IntelHEX понимаемый утилитой в составе SpinalHDL пакета **HexTools**. Формат InetlHEX был изобретен в 1973 году, как можно догадаться, компанией Intel и предназначался для хранения и загрузки данных перепрограммируемых ПЗУ. Формат текстовый, что позволяло передавать прошивки ПЗУ по электронной почте не заморачиваясь с их кодированием. Формат IntelHEX оказался настолько удобен (короткие строки в шестнадцатеричном формате, каждая строка содержит адрес в памяти, данные и контрольную сумму), что закодированные им бинарные файлы можно распечатывать на листах бумаги и распространять программное обеспечение таким образом. Собственно, в 70-х и 80-х годах радиолюбители, электронщики и программисты так и поступали — публиковали в тематических журналах большие дампы своих программ и прошивок ПЗУ, представленных в этом формате.

Чтобы перевести любой файл в формат IntelHEX мы воспользуемся утилитой **objdump**, она присутствует почти во всех современных компиляторах, в том числе в GCC и в LLVM. Формат вызова команды следующий:

```
$ objcopy -I binary -O ihex koi8-r-8x16.bin font8x16x256.hex

$ ls -al font8x16x256.hex
-rw-rw-r-- 1 rz rz 11533 Sep  8 00:06 font8x16x256.hex
```

Что-ж, файл **font8x16x256.hex** с растровым шрифтом мы получили, так что двигаемся дальше.

9.2. Отображение текста

При разработке графического режима для нашего CGA адаптера в реализации компонента **Apb3CGA4HDMICtrl** у нас есть оператор **switch(video_mode)** внутри которого одна из веток (video_mode == 2“b00) оставлена в виде заготовки предназначенной для будущей реализации текстового режима:

```
is(B"00") { // Text mode: 80x30 characters each 8x16 pixels
  // To be implemented...
}
```

Настало время заполнить этот пробел. Сначала добавим код для формирования флага загрузки слова из видеопамяти и загрузку самого слова:

```
// Load flag active on each 6th and 7th pixel
word_load := (CounterX >= U(horiz_back_porch - 8) &&
              CounterX < U(horiz_back_porch + horiz_active)) &&
              (CounterY < vert_back_porch + vert_active + 16) &&
              ((CounterX & U(6)) === U(6))

// Index of the 32 bit word in framebuffer memory: addr = (y / 16) * 80 + x/8
word_address := ((CounterY - 16)(9 downto 4) * 80 +
                 (CounterX - (horiz_back_porch - 8))(9 downto 3)).resized
```

Как и при реализации графического режима здесь мы выжидаем подходящего момента для загрузки нового слова из видеопамяти содержащего ASCII код отображаемого символа и его атрибутов. Таким моментом является каждый последний пиксель последней строки знакоместа.

Следующим шагом мы определим сигнал **char_row** описывающий номер строки знакоместа, и сигнал **char_mask**, представляющий позицию отображаемого пикселя в строке знакоместа в виде битовой маски. Сигнал **char_mask** будет содержать одну единицу циклически перемещающуюся в пределах строки знакоместа (т. е. в пределах 8-ми бит):

```
val char_row = CounterY(3 downto 0)
val char_mask = Reg(Bits(8 bits))

when((CounterX & 7) === 7) {
  char_mask := B"10000000"
} otherwise {
  char_mask := B"0" ## char_mask(7 downto 1)
}
```

Далее определим еще пару сигналов. Сигнал **char_idx** будет содержать номер отображаемого в данный момент символа из таблицы знакогенератора (то есть его ASCII код), взятый из 8-ми младших битов загруженного слова **word**.

```
val char_idx = word(7 downto 0)
```

Также выделим сигналы индекса цвета для текста (биты с 11-го по 8-й) и индекса цвета фона (биты с 19-го по 16-й):

```
val char_fg_color = word(11 downto 8).asUInt
val char_bg_color = word(19 downto 16).asUInt
```

Загрузим из знакогенератора глиф (**char_data**) текущего отображаемого символа. Вычислить его адрес не сложно, достаточно «склеить» вместе биты номер символа (**char_idx**) и номер строки внутри символа (**char_row**):

```
val char_data = chargen_mem((char_idx ## char_row).asUInt).asBits
```

Теперь вычислим три компонента цветности **red**, **green** и **blue** для текущего пикселя исходя из полученных выше данных. Для этого мы будем использовать сигналы **char_fg_color** и **char_bg_color** как индексы для получения данных из регистров палитры **palette_mem**. Если бит в глифе **char_data**, на который указывает маска **char_mask**, установлен в «1», то текущий пиксель будем отображать цветом **char_fg_color**, иначе — цветом **char_bg_color**. Если же текущий пиксель не попадает в видимую область, о чем сообщает нам сигнал **de**, то **red**, **green** и **blue** будем заполнять нулями. Все выше сказанное в коде это выглядит следующим образом:

```
when(de) {

  when((char_data & char_mask) =/= 0) {
    // Draw character
    red := palette_mem(char_fg_color).asBits(7 downto 0)
    green := palette_mem(char_fg_color).asBits(15 downto 8)
    blue := palette_mem(char_fg_color).asBits(23 downto 16)
  } otherwise {
    // Draw background
    red := palette_mem(char_bg_color).asBits(7 downto 0)
    green := palette_mem(char_bg_color).asBits(15 downto 8)
    blue := palette_mem(char_bg_color).asBits(23 downto 16)
  }

} otherwise {
  red := 0
  green := 0
  blue := 0
}
```

Далее компоненты **red**, **green** и **blue** попадают на вход TMDS энкодера и передаются в HDMI интерфейс. Эта часть схемы компонента у нас уже была написана при разработке графического режима и она остается без изменений.

На этом реализация базового функционала для текстового режима заканчивается. После проведения ряда тестов для текстового режима, я покажу как можно добавить в него плавную вертикальную прокрутку текста и мигающий курсор.

9.3. Тестируем текстовый режим

После внесения изменений в описание компонента **Apb3CGA4HDMICtrl** необходимо полностью пересобрать проект, устранить синтаксически ошибки и убедиться в его собираемости командой **make -f Makefile.Briey generate compile**.

Для тестирования работоспособности текстового режима модифицируем программу **karnix_extended_test**, добавим в файл **src/main.c** функцию **cga_video_test3()**:

```
void cga_video_test3(void) {
        uint32_t *fb = (uint32_t*) CGA->FB;
```

```
	uint32_t t0 = get_mtime() & 0xffffffff;

	cga_wait_vblank();

	for(int y = 0; y < CGA_TEXT_HEIGHT_TOTAL; y++)
		for(int x = 0; x < CGA_TEXT_WIDTH; x++) {
			if(x < y)
				fb[y * 80 + x] = ' ';
			else
				fb[y * 80 + x] = ('0' + x % 80) & 0xff;

			fb[y * 80 + x] |= (x & 0x3) << 8;

			fb[y * 80 + x] |= ((y&3) << 16);
		}

	uint32_t t1 = get_mtime() & 0xffffffff;

	printf("cga_video_test3: text mode t = %lu uS, _y = %d\r\n", t1 - t0, _y);
}
```

Данная функция производит заполнение нулевой текстовой страницы видеопамяти символами от 0 до ~ при этом постоянно меняя цвет текста и цвет фона, результирующее изображение экрана приведено на рис. 15.

Заменим установку графического режима на текстовый, заменив следующие строки кода в теле функции **main()**:

```
cga_set_video_mode(CGA_MODE_GRAPHICS1);
```

на

```
cga_set_video_mode(CGA_MODE_TEXT);
```

И добавим вызов функции **cga_video_test3()** в тело цикла **while()** вместо вызова функции **cga_video_test2()**.

После компиляции, реинтеграции, сборки и загрузки битстрима в терминале получим следующие сообщения:

```
Karnix ASB-254 test prog. Build 00922, date/time: Sep 15 2024 22:49:59
Copyright (C) 2021-2024 Fabmicro, LLC., Tyumen, Russia.

…
cga_set_video_mode: mode = 0, ctrl = 0x80000000
cga_video_test3: text mode t = 4852 uS
cga_video_test3: text mode t = 12637 uS
cga_video_test3: text mode t = 12553 uS
cga_video_test3: text mode t = 12555 uS
cga_video_test3: text mode t = 12562 uS
cga_video_test3: text mode t = 12554 uS
```

Из которых видно, что заполнение одной страницы текста (вместе с ожиданием периода затенения) занимает 12,5 мс. Если немного присмотреться, то можно заметить, что время исполнения функции **cga_video_test3()** в самой первой строке заняло 4,8 мс — это тот самый

редкий случай, когда вызов функции совпал с периодом затенения и время его ожидания сократилось почти до нуля.

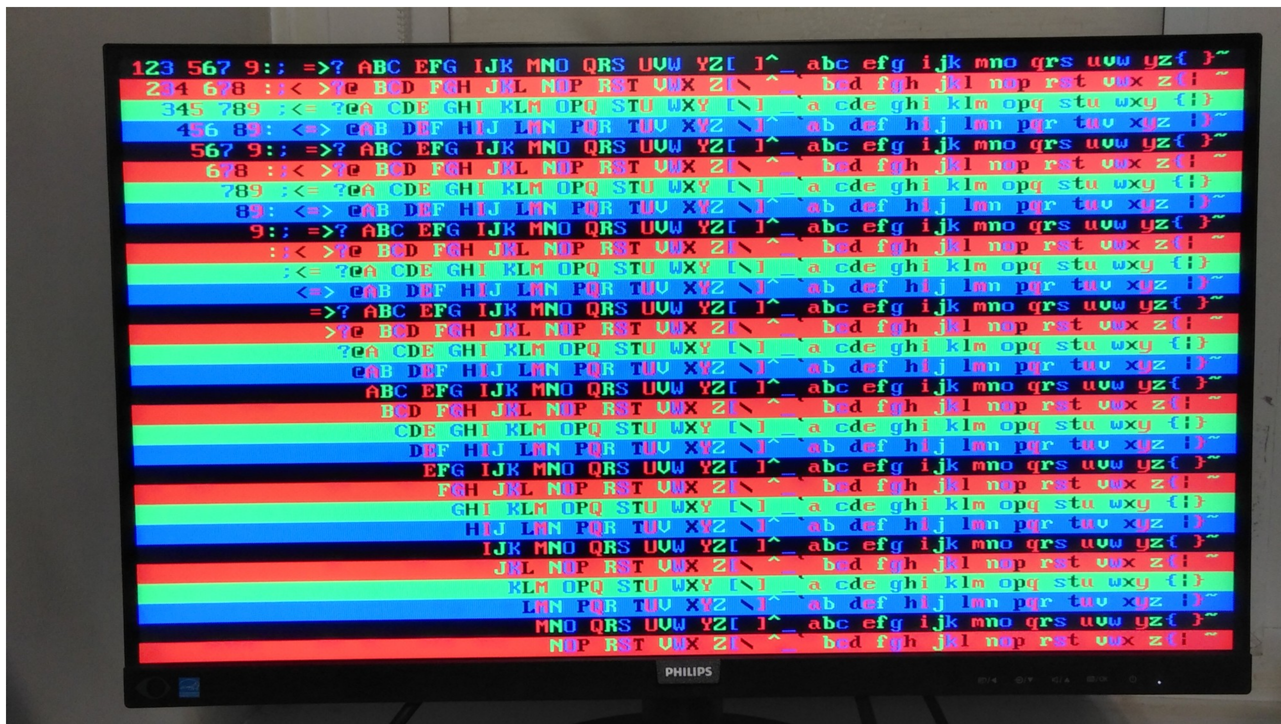


Рис. 15. Тестирование текстового режима.

9.4. Мигающий курсор

Улучшим текстовый режим нашего CGA-подобного видеоадаптера добавлением аппаратной поддержки управляемого (адресуемого) мигающего курсора. У курсора в текстовых режимах имеется три основные функции:

1. Показать пользователю куда на просторах экрана будет осуществлен вывод следующего символа или осуществлено отображение последнего введенного с клавиатуры символа.

2. Индикация режимов ввода.

3. Выбор символа (или объекта) на экране для осуществления манипуляции с ним (выбор символа для редактирования).

С технической точки зрения, аппаратный курсор это два регистра в которых записаны текущие координаты курсора: позиция символа в строке (координата X) и номер строки (координата Y). Аппаратура видеоадаптера формирует изображение курсора в виде блока или подчеркивания и накладывает его на изображение текста. Как правило курсор реализуют мигающим с целью во-первых, быстро привлечь внимание пользователя, а во-вторых, чтобы отличить его от других похожих символов, например от символа «забой» с ASCII кодом 255.

Рис. 16. Блочный мигающий курсор.

Небольшое отступления про видеотерминалы и курсоры. Первые видеотерминалы (или Video Display Units — VDUs) на основе электронно-лучевых трубок начали появляться с середины 1950-х и использовались только для отображения графической информации. Первая компьютерная игра «Spacewar!» была создана как раз для такого векторного дисплея на ЭВМ PDP-1 в 1962 году. Первый текстовый видеотерминал IBM 2260 появился в 1964-м. Его «логика» была собрана на транзисторах, а память размером от 240 до 960 байт (в зависимости от модели) для хранения отображаемой информации - на акустических линиях задержки. Видеотерминал IBM 2260 представлял собой блочное устройство предназначенное отображать на экране страницу текста целиком. Его первые модели даже не оснащались клавиатурой, так как были предназначены только для отображения информации. IBM 2260 не имел поддержки управляемого курсора.

Идею отображения «специального символа» на поверхности экрана в 1968 году предложил Дуглас Энгельбарт — исследователь из Стэнфорда занимавшийся исследованиями в области человеко-машинных интерфейсов (HMI). Он же является изобретателем первой компьютерной мыши.

Первый видеотерминал с управляемым курсором стал DEC VT05 (1970) в котором видеопамять была построена уже на микросхемах динамической памяти. Собственно появление дешевой памяти упакованной в небольшой объём послужило серьезным толчком к развитию средств ввода и отображения информации. На видеотерминале VT05 курсор имел форму «подчеркивания» - в виде одиночной полосы в самом низу знакоместа, так как такой курсор было несложно реализовать примитивными средствами в его схемотехнике, построенной на первых логических микросхемах. Усовершенствованная модель этого терминала, VT52, стала стандартом «де-факто» для символьных (последовательных) терминалов и по сей день является обязательно поддерживаемым типом терминала во всех Unix системах. Видеотерминал VT100 позволял не только программно (со стороны ЭВМ) перемещать курсор в произвольное место экрана, но и программно узнавать его текущее положение, которое могло быть изменено пользователем с помощью специальных клавиш перемещения курсора (стрелок) на клавиатуре.

Последующие модели блочных видеотерминалов от IBM серии 3270 (1971 год) были весьма более сложными устройствами и курсор на них выглядел по разному в зависимости от режима ввода: при вставке текста (insert mode) курсор выглядел как утолщенный мигающий знак подчеркивания, а при замене (overwrite) — как мигающий прямоугольный блок. Также на IBM-овских терминалах иногда использовался т. н. «half-block» курсор — заполненный блок размером в половину символа, начиная с нижней строки.

Почти на всех терминалах курсор был мигающим, частота мигания обычно составляла 3,75 Гц или 7,5 Гц. На многих терминалах форму (стиль) курсора можно было изменять на специальной странице настроек на которую терминал попадал при старте (при нажатии

определенной комбинации клавиш). Позже, на многих терминалах форму курсора можно было менять программно с помощью ANSI ESC-последовательностей - специальных строк символов начинающихся со служебного символа 0x1B (или ^[или «символа Escape»). В более поздних моделях терминалов с помощью ESC-последовательностей можно было менять не только форму, но и цвет курсора, включать или отключать режим мигания, изменять частоту мигания. Многие программные эмуляторы терминалов всё еще поддерживают этот функционал. Например, попробуйте из оболочки **bash** ввести следующую команду:

```
$ echo -ne "\x1b[\x33 q";
```

и курсор изменится на «мигающее подчеркивание». А команда

```
$ echo -ne "\x1b]12;#00FF00\a"
```

сделает его олдскульно зелёным.

Но вернемся к CGA адаптеру от IBM. Когда разрабатывался PC, в его функционал закладывалась возможность использования персонального компьютера как универсального терминала к «большим машинам», а значит в видеоадаптере должна быть аппаратная поддержка курсора с возможность задавать форму курсора. Благо, контроллер ЭЛТ MC6845 уже имел встроенную поддержку курсора, форма которого задавалась значением двух регистров — регистра «начало курсора» (R10) и регистра «окончание курсора» (R11), хранящих значение линий сканирования внутри знакоместа. По-умолчанию эти регистры имели значения 6 и 7 соответственно, что отображалось в виде утолщенного подчеркивания (две последние линии знакоместа). Частота мигания курсора на CGA адаптере была фиксированной и составляла 3,745 Гц, формировалась она в обход контроллера ЭЛТ с помощью делителя частоты VSYNC на 16. Не смотря на то, что контроллер ЭЛТ имел встроенную поддержку мигания с разной частотой, эта возможность в CGA не использовалась. Ниже приведена интересная выдержка с сайта «StackExchange» на тему частоты мигания курсора и мигания текста на адаптерах MDA и CGA:

MDA, original card from IBM, reverse-engineered from schematics :

The cursor blink signal is not handled by the Motorola 6845 CRTC. It is handled by dividing VSYNC with 74LS393 by 16, so it is on for 8 frames and off for 8 frames. As the original MDA has 16.257 MHz pixel clock, 882 dot clocks per HSYNC, and 370 lines per frame, the VSYNC rate is 49.816 Hz and cursor blinks at 3.114 Hz. Blinking text blinks at the cursor rate divided down by 2.

CGA, original card from IBM, reverse engineered from schematics :

Practically same as MDA but with different vertical rate causing different blink rate. VSYNC is externally divided by 16 with 74LS393 for cursor blink and again by two more for text blink. As the original CGA card runs from motherboard oscillator signal, the pixel clock is 315/22 MHz, and as there are 912 dot clocks per HSYNC, and 262 lines per frame, the VSYNC rate is 59.923 Hz, and cursor blinks at 3.745 Hz. Text blink again with cursor rate divided by 2.

Зачем инженерам из IBM потребовалось изменить схему мигания курсора со встроенной в ЭЛТ контроллер на свою — не совсем ясно. Моё предположение состоит в том, что разработчикам требовалось обеспечить мигание текста, так как такая фича была на всех блочных терминалах для подключения к большим машинам, но данной функции не предусматривалось контроллером MC6845. Поэтому схемы мигания текста и курсора (а они логически связаны между собой) сделаны в обход контроллера с использованием микросхем дискретной логики.

Как уже говорилось выше, аппаратно курсор представляет собой два регистра хранящие в себе координаты знакоместа для отображения следующего введенного символа. В главе «*8. Разработка CGA подобного видеоадаптера*» мы уже выделили под эти регистры два регистровых поля **cursor_x** и **cursor_y** в общем регистре управления нашим контроллером **cgaCtrl2Word**. Там же выделены два поля **cursor_bottom** и **cursor_top** для задания формы курсора. Фактически значения в этих регистрах показывают с какой строки сканирования, внутри знакоместа, следует начинать отображать курсор и на какой — заканчивать. Также у нас есть регистр **cursor_blink** задающий делитель частоты мигания и однобитовый регистр **cursor_blink_enabled** разрешающий режим мигания. Для задания цвета курсора (в оригинальном CGA такой возможности не было) мы выделили регистровое поле **cursor_color**. Для наглядности я еще раз приведу определение этих регистровых полей:

```
val cursor_x = cgaCtrl2Word(6 downto 0).asUInt.addTag(crossClockDomain)
val cursor_y = cgaCtrl2Word(13 downto 8).asUInt.addTag(crossClockDomain)
val cursor_bottom = cgaCtrl2Word(19 downto 16).asUInt.addTag(crossClockDomain)
val cursor_top = cgaCtrl2Word(23 downto 20).asUInt.addTag(crossClockDomain)
val cursor_blink = cgaCtrl2Word(26 downto 24).asUInt.addTag(crossClockDomain)
val cursor_blink_enabled = cgaCtrl2Word(27).addTag(crossClockDomain)
val cursor_color = cgaCtrl2Word(31 downto 28).asUInt.addTag(crossClockDomain)
```

Для реализации функции мигания введем 7-ми битный счетчик числа кадров **CounterF**, а регистровое поле **cursor_blink** будем использовать как номер бита (индекс) в этом счетчике для вычисления оного из двух состояний курсора в момент отрисовки кадра: «курсор отображается» и «курсор не отображается». Таким образом у нас получится целых 7 вариантов частоты для мигания курсора — от половины частоты кадровой развертки в 30 Гц (cursor_blink = 0) до 0,8125 Гц (cursor_blink = 6).

```
val CounterF = Reg(UInt(7 bits)) // Frame counter, used for cursor blink feature

when(CounterX === 0 && CounterY === 0) {
  CounterF := CounterF + 1
}
```

Введем флаг **blink_flag** который будет указывать следует ли отображать курсор или нет. Далее мы задействуем этот флаг непосредственно при отображении символа — будем замещать символ на курсор если флаг установлен. Добавим в этот флаг условие для формирования стиля курсора (размера курсора по высоте): будем сравнивать счетчик строк сканирования **char_row** с регистрами **cursor_top** и **cursor_bottom**:

```
var blink_flag = CounterF(cursor_blink) && cursor_blink_enabled &&
                 char_row >= cursor_top && char_row <= cursor_bottom
```

Иными словами, линия сканирования знакоместа будет заполнятся цветом курсора если все условия выполняются. Иначе, будет заполнятся глифом текущего символа.

Введем две переменные **x** и **y**, исключительно для удобства, которые будут ссылаться на часть битов счетчиков **CounterX** и **CounterY**, показывая позицию текущего отображаемого знакоместа по X и по Y:

```
var x = (CounterX - horiz_back_porch)(9 downto 3) // current ray column position
var y = (CounterY - vert_back_porch)(9 downto 4) // current ray raw position
```

Теперь у нас всё готово для того, чтобы написать код для формирования трех сигналов **red**, **green** и **blue** в текстовом режиме с учетом отображения курсора:

```
when(x === cursor_x && y === cursor_y && blink_flag) {
  // Implement blinking cursor
  red := palette_mem(cursor_color).asBits(7 downto 0)
  green := palette_mem(cursor_color).asBits(15 downto 8)
  blue := palette_mem(cursor_color).asBits(23 downto 16)
} elsewhen((char_data & char_mask) =/= 0) {
  // Draw character
  red := palette_mem(char_fg_color).asBits(7 downto 0)
  green := palette_mem(char_fg_color).asBits(15 downto 8)
  blue := palette_mem(char_fg_color).asBits(23 downto 16)
} otherwise {
  // Draw background
  red := palette_mem(char_bg_color).asBits(7 downto 0)
  green := palette_mem(char_bg_color).asBits(15 downto 8)
  blue := palette_mem(char_bg_color).asBits(23 downto 16)
}
```

Заменим на этот код ранее написанный вариант внутри блока **switch(video_mode)** для случая **is(B"00")**. Полностью код этого блока будет выглядеть так:

```
    is(B"00") { // Text mode: 80x30 characters each 8x16 pixels

      // Load flag active on each 6th and 7th pixel
      word_load := (CounterX >= U(horiz_back_porch - 8) && CounterX < U(horiz_back_porch + 
horiz_active)) &&
                   (CounterY < vert_back_porch + vert_active + 16) && ((CounterX & U(6)) === U(6))

      // Index of the 32 bit word in framebuffer memory: addr = (y / 16) * 80 + x/8
      word_address := ((CounterY - 16)(9 downto 4) * 80 +
                       (CounterX - (horiz_back_porch - 8))(9 downto 3)).resized

      val char_row = CounterY_(3 downto 0)
      val char_mask = Reg(Bits(8 bits))

      val char_idx = word(7 downto 0)
      val char_fg_color = word(11 downto 8).asUInt
      val char_bg_color = word(19 downto 16).asUInt

      when((CounterX & 7) === 7) {
        char_mask := B"10000000"
      } otherwise {
        char_mask := B"0" ## char_mask(7 downto 1)
      }

      val char_data = chargen_mem((char_idx ## char_row).asUInt).asBits

      when(de) {
```

```
          var x = (CounterX - horiz_back_porch)(9 downto 3) // current ray column position
          var y = (CounterY - vert_back_porch)(9 downto 4) // current ray raw position
          var blink_flag = CounterF(cursor_blink) && cursor_blink_enabled &&
                           char_row >= cursor_top && char_row <= cursor_bottom

          when(x === cursor_x && y === cursor_y && blink_flag) {
            // Implement blinking cursor
            red := palette_mem(cursor_color).asBits(7 downto 0)
            green := palette_mem(cursor_color).asBits(15 downto 8)
            blue := palette_mem(cursor_color).asBits(23 downto 16)
          } elsewhen((char_data & char_mask) =/= 0) {
            // Draw character
            red := palette_mem(char_fg_color).asBits(7 downto 0)
            green := palette_mem(char_fg_color).asBits(15 downto 8)
            blue := palette_mem(char_fg_color).asBits(23 downto 16)
          } otherwise {
            // Draw background
            red := palette_mem(char_bg_color).asBits(7 downto 0)
            green := palette_mem(char_bg_color).asBits(15 downto 8)
            blue := palette_mem(char_bg_color).asBits(23 downto 16)
          }

        } otherwise {
          red := 0
          green := 0
          blue := 0
        }
      }
```

На этом аппаратная часть реализации курсора заканчивается. Если сейчас произвести пересборку аппаратной части проекта командами **make -f Makefile.Briey generate compile**, то после загрузки битстрима в ПЛИС мы увидим на экране всё то же поле из разноцветных строк символов, формируемых функцией **cga_video_test3()**, но в верхнем левом углу будет мигать белый прямоугольник размером в знакоместо — это и есть наш курсор!

Для удобства работы с курсором из программы добавим следующие функции в файл **./src/main/c/karnix_extended_test/src/cga.c**:

```
void cga_set_cursor_xy(int x, int y) {
        CGA->CTRL2 &= ~CGA_CTRL2_CURSOR_X;
        CGA->CTRL2 |= (x & 0xff) << CGA_CTRL2_CURSOR_X_SHIFT;
        CGA->CTRL2 &= ~CGA_CTRL2_CURSOR_Y;
        CGA->CTRL2 |= (y & 0xff) << CGA_CTRL2_CURSOR_Y_SHIFT;
}


void cga_set_cursor_style(int top, int bottom) {
        CGA->CTRL2 &= ~CGA_CTRL2_CURSOR_TOP;
        CGA->CTRL2 |= (top & 0x0f) << CGA_CTRL2_CURSOR_TOP_SHIFT;
        CGA->CTRL2 &= ~CGA_CTRL2_CURSOR_BOTTOM;
        CGA->CTRL2 |= (bottom & 0x0f) << CGA_CTRL2_CURSOR_BOTTOM_SHIFT;
}


inline int cga_get_cursor_x(void) {
        return (CGA->CTRL2 >> CGA_CTRL2_CURSOR_X_SHIFT) & 0xff;
}


inline int cga_get_cursor_y(void) {
        return (CGA->CTRL2 >> CGA_CTRL2_CURSOR_Y_SHIFT) & 0xff;
}
```

И их определения в заголовочный файл **./src/main/c/karnix_extended_test/src/cga.h:**

```
void cga_set_cursor_xy(int x, int y);
void cga_set_cursor_style(int top, int bottom);
inline int cga_get_cursor_x(void);
inline int cga_get_cursor_y(void);
```

Для проверки, добавим изменение позиции курсора и его стиля в функцию **cga_video_test3()** в главном файле программы **./src/main.c** тестовой программы **karnix_extended_test**:

```
void cga_video_test3(void) {
        ...
        cga_wait_vblank();
        cga_set_cursor_xy(2, 1);
        cga_set_cursor_style(10, 14);
        ...
}
```

После компиляции тестовой Си программы, выполнения реинтеграции бинарного кода в битстрим и загрузки его в ПЛИС, мы увидим это же поле из цветных символьных строк, но курсор переместится в третью позицию на второй строке и изменит свою форму на «жирное подчеркивание».

На этом закончим с курсором и займемся плавной вертикальной прокруткой изображения. Замечу лишь, что курсор у нас всё равно получился немного неполноценный, но об этом пусть читатель напишет в комментариях и представит свои варианты решения.

9.5. Плавная вертикальная прокрутка (скроллинг) изображения

Под плавной вертикальной прокруткой изображения (smooth vertical scrolling) обычно понимается аппаратно поддерживаемое программно задаваемое вертикальное смещение изображения на произвольное число линий сканирования с шагом в одну линию. Иными словами, это такая аппаратная фича, которая позволяет программно перемещать «окно» видимой части изображения внутри видео буфера с точность в одну линию сканирования. Эта фича может быть реализована как для обоих режимов работы видеоадаптера - графического и текстового, так и индивидуально, например только для графического.

Многие видеотерминалы DEC и Wyse имели режим плавной вертикальной прокрутки изображения примерно с середины 1970-х, в DEC VT100 (1978г) уже присутствовала эта фича. На IBM PC аппаратная поддержка плавной вертикальной прокрутки появилась начиная с видеоадаптера IBM VGA (1987), но данная фича не использовалась в системном ПО (ни в BIOS, ни в MS-DOS), текст пролистывался вверх по экрану традиционным способом — с шагом в одно знакоместо, и большинство пользователей IBM PC/AT про существование этой фичи даже не догадывалось. В своей практике я очень редко сталкивался с устройствами, которые умели бы осуществлять плавный вертикальный скроллинг текстового изображения. Первый раз я увидел плавную прокрутку изображения в текстовом режиме на учебной ЭВМ УКНЦ Электроника МС 0511 в далеком 1989 году и это произвело на меня серьезное впечатление. Позже мне довилось пользоваться терминалом Wyse 60, где также присутствовала плавная прокрутка. А вот в эмуляциях терминалов в различных UNIX системах такой фичи не было долгое время.

Наша аппаратная реализация аппаратного скроллинга (или точнее будет сказать — программно задаваемого смещения) будет работать как для текстового, так и для графического режимов. Алгоритм её действия очень прост - при вычислении адреса ячейки данных, на основе которых формируется выходной видеосигнал, будем прибавлять смещение пропорциональное значению регистра смещения **scroll_v**. Применим следующий подход: вместо регистра **CounterY** для расчета адреса ячейки данных будем использовать другой регистр, **CounterY_,** значение которого будет смещено на **scroll_v** строк сканирования вверх или вниз в зависимости от значения битового регистра **scroll_v_dir**:

```
val CounterY_ = (scroll_v_dir ? (CounterY - BufferCC(scroll_v)) | (CounterY + BufferCC(scroll_v)))
```

Заметим, что регистры **scroll_v** и **CounterY_** относятся к разным тактовым доменам. Для предотвращения проблем связанных с пересечением тактовых доменов, значение регистра **scroll_v** прогоняется через буфер **BufferCC**. Теоретически, регистр **scroll_v_dir** также следовало бы прогнать через буфер, но так как он однобитовый и изменяется достаточно редко, то большого смысла в этом нет.

Таким образом, поправка кода видеоадаптера для графического режима будет выглядеть следующим образом:

```
is(B"01") { // Graphics mode: 320x240, 2 bits per PEL with full color palette

  ...
  // Index of the 32 bit word in framebuffer memory: addr = y/2 * 20 + x/2/16
  word_address := ((CounterY_ - 16)(9 downto 1) * 20 + CounterX(9 downto 5)).resized
```

Для текстового режима следует выполнить еще ряд действий: со смещением вычислить номер отображаемой стоки **char_row** и координату **y**, используемую для отображения курсора. Поправка для текстового режима выглядит так:

```
is(B"00") { // Text mode: 80x30 characters each 8x16 pixels

  ...
  // Index of the 32 bit word in framebuffer memory: addr = (y / 16) * 80 + x/8
  word_address := ((CounterY_ - 16)(9 downto 4) * 80 +
                   (CounterX - (horiz_back_porch - 8))(9 downto 3)).resized

  val char_row = CounterY_(3 downto 0)
  ...
  when(de && blanking_not_enabled) {

    var x = (CounterX - horiz_back_porch)(9 downto 3) // current ray column position
    var y = (CounterY_ - vert_back_porch)(9 downto 4) // current ray raw position
    ...
```

Собственно, это все изменения необходимые для добавления аппаратной поддержки плавной вертикальной прокрутки изображения: добавить одну строку для вычисления **CounterY_** и далее по тексту заменить **CounterY** на **CounterY_** для вычисления адреса загрузки и координат курсора.

Замечание: не стоит вносить вертикальное смещение в расчет **CounterY**, так как это приведет к неправильному формированию сигналов **hSync** и **vSync**!

Осталось протестировать как работает плавная вертикальная прокрутка. Добавим в тело основного цикла функции **main()** в файле **./src/main.c** небольшой фрагмент кода, который будет увеличивать или уменьшать значение регистра **scroll_v** в зависимости от нажатой кнопки, используя новую функцию **cga_set_scroll()**:

```
#if CGA_VIDEO_TEST == CGA_TEST_TEXT
{
        static int _scroll = 0;

        if(GPIO->INPUT & GPIO_IN_KEY0)
                _scroll--;

        if(GPIO->INPUT & GPIO_IN_KEY3)
                _scroll++;

        cga_wait_vblank();
        cga_set_scroll(_scroll);
}
#endif
```

В заголовке файла **./src/main.c** установим макро **CGA_VIDEO_TEST** в значение **CGA_TEST_TEXT** и добавим реализацию функции **cga_set_scroll()** в файл **./src/cga.c**. Данная функция устанавливает значение регистра смещения **scroll_v** и флага направления **scroll_v_dir** в зависимости от полярности входного параметра **scrl**:

```
/*
 * Set vertical scroll register value to scrl. Resulting effect is:
 * negative value - scroll up scrl scan-lines,
 * positive value - scroll down scrl scan-lines,
 * zero value - no scroll.
 */
void cga_set_scroll(int scrl) {
        CGA->CTRL &= ~CGA_CTRL_V_SCROLL;
        if(scrl >= 0) {
                CGA->CTRL &= ~CGA_CTRL_V_SCROLL_DIR;
                CGA->CTRL |= (scrl << CGA_CTRL_V_SCROLL_SHIFT) & CGA_CTRL_V_SCROLL;
        } else {
                CGA->CTRL |= CGA_CTRL_V_SCROLL_DIR;
                CGA->CTRL |= ((-scrl) << CGA_CTRL_V_SCROLL_SHIFT) & CGA_CTRL_V_SCROLL;
        }
}
```

Не забываем добавить заголовок этой функции в файл **./src/cga.h**:

```
void cga_set_scroll(int scrl);
```

После сборки и загрузки битстрима мы опять увидим на экране цветные полосы с текстом, а нажимая кнопки KEY0 и KEY3 сможем перемещать текст вверх и вниз циклически на 1024 строки сканирования. Вместе с текстом перемещается и мигающий курсор, при этом если первая страница текста уходит за пределы видимости, с ней уходит и курсор. Еще раз подчеркну, что таким способом мы перемещаем не сам текст в видео буфере, а «окно» отображаемого текста.

Часто для отображения терминала необходимо осуществлять прокрутку самого текста в видео буфере. Это тоже можно сделать плавным методом (по одной линии сканирования). Для этого прокрутку в рамка одного знакоместа (15 линий) будем осуществлять заданием смещения «окна», а на каждую 16-ю линию будем копировать/перемещать текст в видео буфере вверх или вниз, добавляя пустую строку и возвращать окно в исходное (нулевое) положение. Исходный код двух таких функций, **cga_text_scroll_up()** и **cga_text_scroll_down()**, приведен ниже:

```
void cga_text_scroll_up(int scroll_delay) {
        uint32_t *fb = (uint32_t*) CGA->FB;

        CGA->CTRL2 &= ~CGA_CTRL2_CURSOR_BLINK_EN;
        CGA->CTRL &= ~CGA_CTRL_V_SCROLL_DIR;

        for(int i = 0; i < 16; i++) {
                cga_wait_vblank();
                CGA->CTRL &= ~CGA_CTRL_V_SCROLL;
                CGA->CTRL |= (i & 0x0f) << CGA_CTRL_V_SCROLL_SHIFT;
                delay_us(scroll_delay);
        }

        cga_wait_vblank();

        for(int col = 0; col < CGA_TEXT_WIDTH; col++) {
                uint32_t tmp = fb[col];
                for(int row = 0; row < CGA_TEXT_HEIGHT_TOTAL - 1; row++) {
                                fb[row * CGA_TEXT_WIDTH + col] = fb[row * CGA_TEXT_WIDTH + col +
CGA_TEXT_WIDTH];
                }
                fb[(CGA_TEXT_HEIGHT_TOTAL - 1) * CGA_TEXT_WIDTH + col] = tmp;
        }

        CGA->CTRL &= ~CGA_CTRL_V_SCROLL;
        CGA->CTRL2 |= CGA_CTRL2_CURSOR_BLINK_EN;
}

void cga_text_scroll_down(int scroll_delay) {
        uint32_t *fb = (uint32_t*) CGA->FB;

        CGA->CTRL2 &= ~CGA_CTRL2_CURSOR_BLINK_EN;
        CGA->CTRL |= CGA_CTRL_V_SCROLL_DIR;

        for(int i = 0; i < 16; i++) {
                cga_wait_vblank();
                CGA->CTRL &= ~CGA_CTRL_V_SCROLL;
                CGA->CTRL |= (i & 0x0f) << CGA_CTRL_V_SCROLL_SHIFT;
                delay_us(scroll_delay);
        }

        cga_wait_vblank();

        for(int col = 0; col < CGA_TEXT_WIDTH; col++) {
                uint32_t tmp = fb[(CGA_TEXT_HEIGHT_TOTAL - 1) * CGA_TEXT_WIDTH + col];
                for(int row = CGA_TEXT_HEIGHT_TOTAL - 1; row > 0; row--) {
                                fb[row * CGA_TEXT_WIDTH + col] = fb[row * CGA_TEXT_WIDTH + col -
CGA_TEXT_WIDTH];
                }
                fb[col] = tmp;
        }

        CGA->CTRL &= ~CGA_CTRL_V_SCROLL;
        CGA->CTRL2 |= CGA_CTRL2_CURSOR_BLINK_EN;
}
```

Обе функции принимают на вход один параметр **scroll_delay**, задающий задержку в микросекундах между смещениями окна, это позволит регулировать скорость прокрутки текста на экране, добиваясь комфортного глазу эффекта.

Для тестирования этого вида прокрутки добавим в главный цикл функции main() следующий код:

```
#if CGA_VIDEO_TEST == CGA_TEST_TEXT_SCROLL
{
        if(GPIO->INPUT & GPIO_IN_KEY0)
                cga_text_scroll_up(500);

        if(GPIO->INPUT & GPIO_IN_KEY3)
                cga_text_scroll_down(500);
}
#endif
```

И установим макро **CGA_VIDEO_TEST** в значение **CGA_TEST_TEXT_SCROLL** в заголовке файла **./src/main.c**.

После компиляции, реинтеграции и загрузки битстрима нажимаем кнопки KEY0 и KEY3, и наблюдаем быструю и плавную прокрутку текста на экране, при этом курсор остается на своём месте. Фактически, у нас почти готов свой терминал для отображения текстовых сообщений в потоковом режиме в духе MS-DOS.

10. Специальные эффекты

Мой рассказ про разработку видеоадаптера из прошлого был бы неполным без описания ряда визуальных эффектов (или «спецэффектов») к которым часто прибегали разработчики игр и приложений того времени, дабы хоть немного возвыситься над ограничениями вызванными скудной аппаратурой видеоадаптеров. И скажу Вам, что любое такое достижение вызывало восторг у пользователя. Основной интерес конечно же представляли различные трюки позволяющие увеличить число одновременно отображаемых на экране цветов. Напомню, что адаптер IBM CGA в графическом режиме позволял отобразить на экране одновременно пиксели только четырех различных цветов из весьма скудной палитры. В главе «*7.1. Устройство CGA адаптера*» упоминалось, что оригинальный CGA имел выход телевизионного сигнала стандарта NTSC при выводе которого на обычный телевизор можно было получать дополнительные цветовые комбинации в результате смешивания цветов рядом стоящих пикселей. Многие игры прибегали к этому свойству видеосигнала чтобы отобразить красивую заставку к игре (см. рис. 10) видимую хотя бы на домашнем телевизоре. Этот визуальный эффект назывался «composite artifact colors» (цвета композитных артефактов) и был он не единственным.

10.1. Высокочастотное смешивание цветов

Повторить описанный выше визуальный эффект на мониторе с HDMI/DVI-D интерфейсом не представляется возможным, но мы можем получить похожий эффект другим способом — смешивая два цвета путем попеременной смены части изображения с высокой частотой, быстрее чем может зафиксировать глаз человека. Как мы убедились, вычислительной мощности используемой нами синтезированной вычислительной системы вполне достаточно, чтобы выполнять полную смену изображения в видео буфере за время обратного хода луча, а это значит, что мы можем изменять изображение на экране с частотой близкой к 60 Гц. Комбинируя три имеющихся у нас основных цвета (предполагаем, что

черный цвет не участвует в этом процессе), мы можем дополнительно получить еще три: «желтый» (yellow) — как комбинацию «красного» и «зеленого», «голубой» (cyan) — как комбинацию «зеленого» и «синего», и «розовый» (magenta) — как комбинацию «красного» и «синего». Здесь я беру названия цветов в кавычки, так как все они могут быть переопределены в таблице палитры на произвольные цвета из набора задаваемого 24-мя битами (богатая палитра - это фича нашей реализации видеоадаптера), при этом трюк со смешиванием всё так же будет работать и выдавать какой-то производный цвет. Если задействовать еще и «черный», то количество «артефактных» цветов составит шесть, а в сумме с основными цветами — десять. Еще интересным моментом является то, что этот трюк работает как для графического режима, так и для текстового. В текстовом режиме мы можем одновременно отображать 16 цветов стандартным способом, что, в теории, может дать нам аж 120 артефактных цвета!

Давайте попробуем простой способ смешать три основных цвета - красный, зеленый и синий. Сделаем это в текстовом режиме: будем выводить на экран горизонтальные строки текста шести различных цветов — трёх базовых и трех артефактных. Для смешивания, будем регулярно перерисовывать два экрана на которых отличаются базовые цвета в нижних трёх частях, т. е. там где будет происходить смешивание. Ниже приведен фрагмент кода, его можно вставить в тело главного цикла функции **main()** предварительно установив значение макропеременной **CGA_VIDEO_TEST** в **CGA_TEST_VIDEOFX_TEXTFLIC:**

```
#define CGA_VIDEO_TEST    CGA_TEST_VIDEOFX_TEXTFLIC

...

#if CGA_VIDEO_TEST == CGA_TEST_VIDEOFX_TEXTFLIC
{
        uint32_t *fb = (uint32_t*) CGA->FB;

        cga_wait_vblank();

        for(int i = 0; i < 80*5; i++) {
                fb[80 *  0] = 0x00000131;
                fb[80 *  5] = 0x00000232;
                fb[80 * 10] = 0x00000333;
                fb[80 * 15] = 0x00000134;
                fb[80 * 20] = 0x00000235;
                fb[80 * 25] = 0x00000336;
                fb++;
        }

        fb = (uint32_t*) CGA->FB;

        cga_wait_vblank();

        for(int i = 0; i < 80*5; i++) {
                fb[80 *  0] = 0x00000131;
                fb[80 *  5] = 0x00000232;
                fb[80 * 10] = 0x00000333;
                fb[80 * 15] = 0x00000234;
                fb[80 * 20] = 0x00000335;
                fb[80 * 25] = 0x00000136;
                fb++;
        }
}
#endif
```

Пересобрав код и загрузив битстрим в микросхему ПЛИС мы получим на экране изображение приведенное на рис. 17.

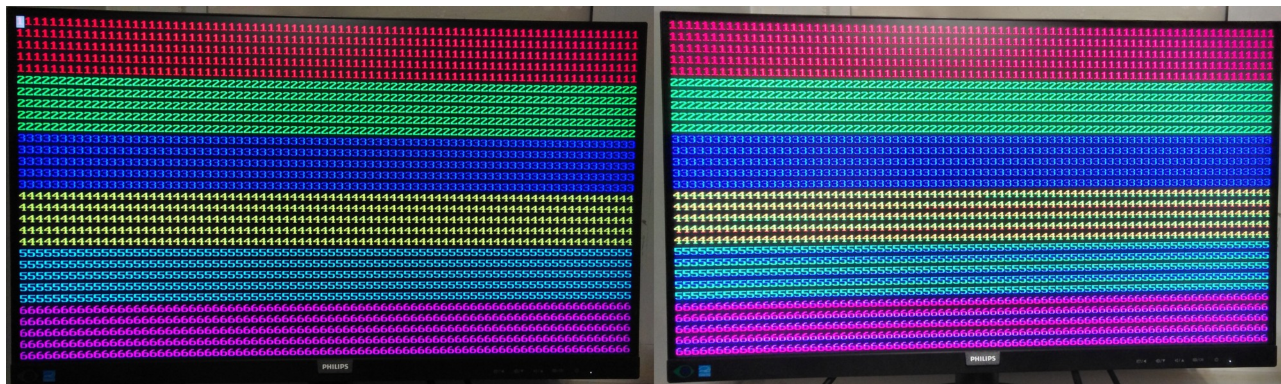


Рис. 17. Получение дополнительных трех цветов путем высокочастотного смешивания. Слева — смешивание в момент затенения, и справа — без ожидания периода затенения. На рисунке справа изображение размытое и видны артефакты.

Важно. Для того, что бы получить четкое изображение без артефактов, необходимо выполнять смену «картинок» в тот момент, когда сигнал на экран монитора не выдается, то есть в момент периода затенения. Для этого перед выводом текста (или графики) в коде стоит вызов функции **cga_wait_vblank()**. Если данную функцию не вызвать, то очень часто формирование изображения на экране будет пересекаться с процессом изменения этого изображения в видео памяти, что приведет к различным неприятным глазу артефактам. И напротив, с ожиданием обратного хода луча, изображение получается четким и равномерно окрашенным в желтый, голубой и розовый цвета соответственно. Откровенно говоря, при съёмке этого трюка с экрана монитора на камеру моего смартфона, я ожидал увидеть стробоскопический эффект под тип тех, что наблюдаются при съёмки старых ЭЛТ телевизоров - мигающая черная полоса на экране. Но стробоскопического эффекта не проявилось и изображение на камере очень четкое, без мерцаний. А вот если присмотреться к экрану монитора невооруженным глазом, то легкое мерцание всё же можно заметить.

Недостатком такого способа увеличения количества отображаемых цветов является и то, что программе придется постоянно хранить (и формировать) две слегка отличающиеся копии изображения, а так же придется постоянно тратить процессорное время на отслеживание события начала периода затенения и на копирование соответствующего изображения в видеопамять. Но это, пожалуй, самый простой способ слегка увеличить число отображаемых цветов. На оригинальной IBM PC этот способ смешивания цветов невозможен в виду низкой производительности вычислительной системы, максимально к чему такое отображение могло бы привести - это к неприятно мигающему изображению на экране.

10.2. Динамическая перезагрузка палитры в процессе отрисовки

В нашей реализации CGA-подобного видеоадаптера имеется возможность изменять регистры палитры и задавать любой цвет на выбор из тех, что можно задать тремя составляющими компонентами R, G и B размерностью по 8 бит каждый. Еще у нас есть возможность контролировать положение «луча» через битовые сигналы **vi_area.hSync** и **dvi_area.vSync**, которые программно доступны в регистре **cgaCtrlWord[21:20]**. Напомню, что сигнал **hSync** устанавливается в момент, когда «луч» начинает свой ход с конца в начало линии сканирования, а сигнал **vSync** — с конца в начало экрана. Что если после отображения каждой строки сканирования (на каждый положительный **hSync**) мы будем изменять значение регистра палитры. Это должно привести к тому, что каждая линия сканирования будет отображаться разным цветом, и количество цветов ограничено лишь возможностями регистра палитры. Давайте попробуем раскрасить обычный текст в радужные цвета.

Для начала добавим в файл **./src/cga.c** функцию **cga_text_print()** для отображения текстовой строки в заданной позиции экрана и с заданными атрибутами (из атрибутов у нас цвет фона и цвет текста):

```
void cga_text_print(uint8_t *framebuffer, int x, int y, int fg_color, int bg_color, char *text, int text_size)
{
	if(!text)
		return;

	uint32_t *fb = (uint32_t*) framebuffer;
	uint32_t attributes = (fg_color << 8) | (bg_color << 16);

	fb += CGA_TEXT_WIDTH * y + x;

	for(int i = 0; text[i]; i++)
		*fb++ = attributes | text[i];

}
```

В этой функции нет ничего особенного, она просто копирует строку в заданную область видеопамяти.

Используя этот примитив опишем функцию **cga_video_test4()** для отображения какого-нибудь осмысленного текста на экране залитого черным фоном. Как обычно, обернем эту функцию в конструкцию **#ifdef**, чтобы её можно было быстро отключить:

```
#if (CGA_VIDEO_TEST == CGA_TEST_VIDEOFX_DYNPALETTE)
void cga_video_test4(void) {
	char *text1 = "Hello, Habr!";
	char *text2 = "Here're some special video effects for ya.";
	char *text3 = "Press KEY1 or KEY2 to roll rainbow around.";
```

```
        cga_wait_vblank();

        cga_fill_screen(0); // clear screen

        cga_text_print(CGA->FB, (CGA_TEXT_WIDTH - strlen(text1)) / 2, 13, 1, 0, text1,
strlen(text1));
        cga_text_print(CGA->FB, (CGA_TEXT_WIDTH - strlen(text2)) / 2, 14, 1, 0, text2,
strlen(text2));
        cga_text_print(CGA->FB, (CGA_TEXT_WIDTH - strlen(text3)) / 2, 16, 1, 0, text3,
strlen(text3));
}
#endif
```

Функция **cga_video_test4()** очищает экранную область (заполняет нулями) и выводит посреди экрана три строки текста цветом номер **1**. Тут важен не сам цвет (по умолчанию это будет красный), а его индекс, который на самом деле является номером регистра палитры. От содержимого регистра с этим номером будет зависеть то, каким именно цветом будет вестись отображение данного текста.

Вставим вызов этой функции в тело функции **main()** перед самым началом главного цикла. В заголовке этого же файла заменим значение макропеременной **CGA_VIDEO_TEST** на новое **CGA_TEST_VIDEOFX_DYNPALETTE**.

```
#define CGA_VIDEO_TEST CGA_TEST_VIDEOFX_DYNPALETTE

...
int main(void)
{
        ...

        #if (CGA_VIDEO_TEST == CGA_TEST_VIDEOFX_DYNPALETTE)
        cga_video_test4();
        #endif

        while(1) {
                ...
        }
```

Ну, а теперь самое интересное. В теле главного цикла вставим следующий код:

```
                #if CGA_VIDEO_TEST == CGA_TEST_VIDEOFX_DYNPALETTE
                {
                        static uint32_t colorfx_rainbow[] = {
                                // Straight
                                0x000000ff, 0x000055ff, 0x0000aaff, 0x0000ffff,
                                0x0000ffaa, 0x0000ff2a, 0x002bff00, 0x0080ff00,
                                0x00d4ff00, 0x00ffd400, 0x00ffaa00, 0x00ff5500,
                                0x00ff0000, 0x00ff0055, 0x00ff00aa, 0x00ff00ff,
                                // Reversed
                                0x00ff00ff, 0x00ff00aa, 0x00ff0055, 0x00ff0000,
                                0x00ff5500, 0x00ffaa00, 0x00ffd400, 0x00d4ff00,
                                0x0080ff00, 0x002bff00, 0x0000ff2a, 0x0000ffaa,
                                0x0000ffff, 0x0000aaff, 0x000055ff, 0x000000ff,
                        };
```

```
		static int colorfx_offset = 6;

		int colorfx_idx = colorfx_offset;

		while(!(CGA->CTRL & CGA_CTRL_VSYNC_FLAG)); // 1

		for(int i = 0; i < 525/2; i++) {
			while(!(CGA->CTRL & CGA_CTRL_HSYNC_FLAG)); // 2
			CGA->PALETTE[1] = colorfx_rainbow[colorfx_idx]; // 3
			colorfx_idx = (colorfx_idx + 1) % 32; // 4
			while(CGA->CTRL & CGA_CTRL_HSYNC_FLAG); // 5

			while(!(CGA->CTRL & CGA_CTRL_HSYNC_FLAG)); // 6
			while(CGA->CTRL & CGA_CTRL_HSYNC_FLAG); // 7
		}

		if(GPIO->INPUT & GPIO_IN_KEY1)
			colorfx_offset++;

		if(GPIO->INPUT & GPIO_IN_KEY2)
			colorfx_offset--;
	}
#endif
```

Я объясню что тут происходит чуть дальше. Сейчас компилируем, реинтегрируем машинный код в битстрим и загружаем полученный битстрим в микросхему ПЛИС. Вот что мы увидим на экране монитора после старта:

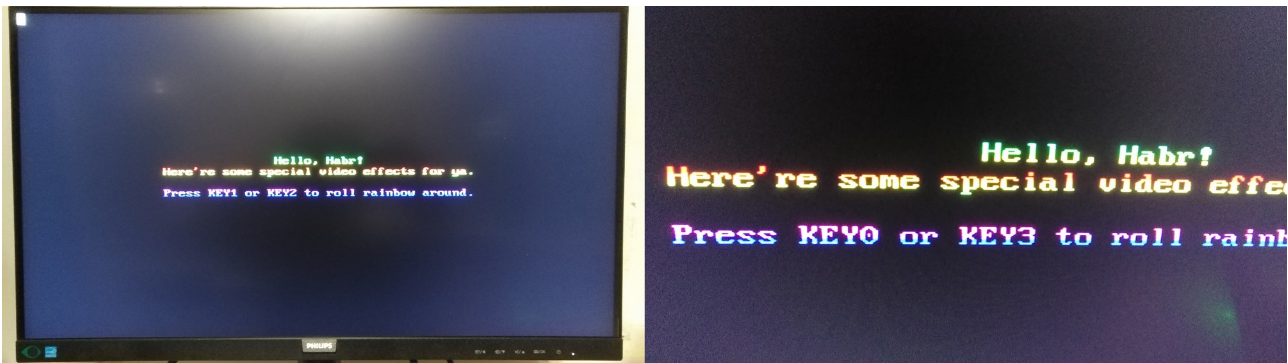


Рис. 18. Раскрашивание текста в радужные цвета построчно (по одной линии сканирования) путем отслеживания сигнала горизонтальной развертки и своевременного обновления значения в регистре палитры.

Ссылка на видео: https://www.youtube.com/watch?v=UlG--Ezm3rM

А увидим мы три строки текста раскрашенные в радужные цвета с градиентным изменением цвета — то есть несколькими оттенками желтого, зеленого, красного, розового и фиолетового. Если мы сейчас нажмем кнопку KEY1 или KEY2, то этот цветовой градиент начнет перемещаться вверх или вниз создавая эффект перетекающего цвета. У неискушенного пользователя может сложиться ложное ощущение, что перед ним не текстовый режим, а графический, причем с очень большим числом одновременно отображаемых цветов. Но мигающий в левом верхнем углу белый курсор говорит о том, что текущий видеорежим текстовый.

Давайте разбираться. Для начала вспомним, что в режиме 640x480 кадр содержит 525 строк сканирования, из них 480 являются отображаемыми (видимыми), а остальные — служебными: 2 строки на сигнал вертикальной развертки HSYNC; 27 строк на «передний порог» (front porch), который следует сразу за VSYNC; и 16 строк на «задний порог» (back porch), который следует за «передним порогом». Наша задача состоит в том, чтобы сначала поймать начало кадра (аппаратный сигнал **vSync**) и далее для каждой строки оперативно подгружать в регистр палитры номер 1 значение цвета представленное в формате xBGR. Эти значения задаются небольшой таблицей **colorfx_rainbow**, всего в таблице содержится 32 точки цвета описывающие «радугу» - сначала по возрастанию, от красного к фиолетовому, потом обратно по убыванию. Мы изменяем регистр палитры номер 1 потому, что именно этим цветовым индексом выведен текст в видеопамять. Текст выведенный другими индексами цвета (0, 2 и 3) изменяться не будет и соответственно будет отображаться обычным способом.

Теперь к коду. Для удобства я проставил номера строк к ключевым моментам. Строка помеченная цифрой «1» уже встречалась нам ранее — это ожидание момента начала обратного хода луча по вертикали, т. е. сигнала вертикальной развертки **vSync**. Сразу после этого запускается цикл отрисовки из 525/2 = 262 итераций, в каждой итерации загружается одно и то же значение цветности для двух последовательных строк сканирования. Можно было бы загружать цветность и для каждой строки сканирования отдельно, то тогда градиент будет очень сжатым и табличку с «радугой» придется увеличить в два раза, чтобы добиться хорошего визуального эффекта. Перед тем как загрузить значение в регистр палитры для последующей строки, необходимо дождаться момента когда «луч» достигнет края текущей строки и начнет обратный ход по горизонтали. Это сигнализируется активным уровнем сигнала строчной развертки **hSync**. В строке кода «2» мы как раз ждем появления бита **CGA_CTRL_HSYNC_FLAG** в регистре управления. Сразу после чего в строке коде «3» загружаем новое значение в регистр палитры номер 1 и рассчитываем новый индекс для следующего элемента массива «радуги» в строке кода «4». Далее в строке кода «5» зависаем в ожидании конца сигнала **hSync**. Ожидание конца действия сигнала строчной развертки нужно для того, чтобы убедиться в том, что «луч» перешел к следующей строке. В строках кода «6» и «7» происходит тоже самое, что и в строках «2» и «5» - ожидание начала следующего **hSync** и его завершения. Это производится отображение второй строки. Изменение регистра палитры тут не происходит, так как нам требуется чтобы использовалось предыдущее значение цветности. Иными словами — рисуем одним и тем же цветом по два раза. Далее в коде идет проверка нажатия клавиш KEY1 и KEY2, и изменения переменной **colorfx_offset**, которая влияет на начальное значение индекса **colorfx_idx**, т. е. задает его смещение в таблице «радуги». Таким образом, с точки зрения видеоадаптера, на экране у нас всего два цвета — цвет с индексом **0** и **1**. Цвет с индексом **0** не меняется и постоянно отображается черным. А цвет с индексом **1** непрерывно перегружается.

Вот такой затейливый спецэффект можно получить имея в запасе достаточное количество вычислительных ресурсов (тактов процессора), позволяющих отслеживать положение «луча». Раскрою небольшой секрет — мощности нашей вычислительной системы достаточно даже для того, чтобы отслеживать положение луча и по-горизонтали, что

позволяет раскрашивать в миллионы цветов не только строки, но горизонтальные блоки пикселей, получая еще более интересные спецэффекты.

Недостаток такого спецэффекта очевиден — почти полная занятость процессора отслеживанием положения «луча», любое отвлечение от этого процесса приведет к сбою синхронизации и смешению цветов в единую серую массу. Теоретически, можно переделать данный алгоритм на работу через прерывание по сигналу **vblank** и немного разгрузить процессор, но так как вызов обработчика прерываний может запаздывать по ряду причин, то это скажется на качестве изображения. Всё это не значит, что такой спецэффект совершенно бесполезен. Напротив, им очень часто пользовались игроделы и демо-мэйкеры в 8-ми битную эпоху на таких машинах как Atari 800, Commodore C64 и ряда других.

10.3. Совмещение двух видеорежимов — текстового и графического

А сейчас, пристегните ремни — начинается самое интересное. Как Вы думаете, что произойдет, если мы будем непрерывно, с частотой кадровой развертки 60 Гц, переключаться между текстовым и графическим режимом, при этом сохраняя и подменяя содержимое видеопамяти в соответствии с включаемым режимом ? Догадаться тут конечно не сложно, но зачем гадать, давайте попробуем!

Итак, у нас есть два видеорежима. Текстовый — в нём мы будем отображать на экран какой нибудь бессмысленный разноцветный текст («Lorem Ipsum» как раз подойдет). В этом режиме нам доступно 16 цветов раздельно для фона и текста. И графический — в нём мы будем отрисовывать анимированный битмап в четырех цветах.

Для того, чтобы вывести раскрашенный форматированный текст, немного модифицируем функцию **cga_text_print()** - добавим в неё поддержку небольшого числа ESC-последовательностей для перемещения знакоместа для следующего выводимого символа, а также сделаем обработку специальных символов: **\t** — табуляция, **\n** — перевод строки и **\r** — возврат каретки. После изменений функция **cga_text_print()** слегка разрастется и будет выглядеть так:

```
void cga_text_print(uint8_t *framebuffer, int x, int y, int fg_color, int bg_color, char *text)
{
        if(!text)
                return;

        uint32_t *fb = (uint32_t*) framebuffer;
        uint32_t attributes = (fg_color << 8) | (bg_color << 16);

        fb += CGA_TEXT_WIDTH * y + x;

        for(int i = 0; text[i]; i++) {
                if(text[i] == '\n') {
                        fb += CGA_TEXT_WIDTH;
                } else if(text[i] == '\r') {
                        fb -= (fb - (uint32_t*)framebuffer) % CGA_TEXT_WIDTH;
                } else if(text[i] == '\t') {
                        for(int j = 0; j < 8; j++)
                                *fb++ = attributes | 0x20;
                } else if(text[i] == 0x1b) {
                        i++;
                        switch(text[i]) {
                                case 'F': {
                                        i++;
```

```
                                        attributes &= ~ 0x0000ff00;
                                        attributes |= atoi(&text[i]) << 8;
                                        while(text[++i] != ';');
                                } break;
                                case 'B': {
                                        i++;
                                        attributes &= ~ 0x00ff0000;
                                        attributes |= atoi(&text[i]) << 16;
                                        while(text[++i] != ';');
                                } break;
                                case 'D': {
                                        i++;
                                        fb += CGA_TEXT_WIDTH * atoi(&text[i]);
                                        while(text[++i] != ';');
                                } break;
                                case 'U': {
                                        i++;
                                        fb -= CGA_TEXT_WIDTH * atoi(&text[i]);
                                        while(text[++i] != ';');
                                } break;
                                case 'L': {
                                        i++;
                                        fb -= atoi(&text[i]);
                                        while(text[++i] != ';');
                                } break;
                                case 'R': {
                                        i++;
                                        fb += atoi(&text[i]);
                                        while(text[++i] != ';');
                                } break;
                        }
                        continue;
                } else
                        *fb++ = attributes | text[i];
        }
}
```

Поддерживаемые ESC-последовательности следующие:

```
#define ESC_UP          "\033U" // Move cursor upward 1 line
#define ESC_DOWN        "\033D" // Move cursor downward 1 line (same as \n)
#define ESC_LEFT        "\033L" // Move cursor to the left 1 char
#define ESC_RIGTH       "\033R" // Move cursor to the right 1 char
#define ESC_FG          "\033F" // Set foreground color
#define ESC_BG          "\033B" // Set background color
```

Не забываем добавить эти макро в файл **./src/cga.h**.

Далее, каждую порцию изображения будем предварительно формировать в отдельных буферах: текстовую в **videobuf_for_text**, а графическую - в **videobuf_for_graphics** соответственно. Буфера расположим где-нибудь в области статического ОЗУ, т. е. с адреса **0x90000000**. Составим функцию **cga_video_test5()** которая будет заполнять эти два буфера и поместим её в файл **./src/main.c**. Также не забываем переключить макропеременную **CGA_VIDEO_TEST** на **CGA_TEST_VIDEOFX_HYBRID**.

```
#if (CGA_VIDEO_TEST == CGA_TEST_VIDEOFX_HYBRID)

static unsigned char *videobuf_for_text = (unsigned char*)0x90001000;

static unsigned char *videobuf_for_graphics = (unsigned char*)0x90006000;


void cga_video_test5(void) {

        char *lorem_ipsum =
```

```
		ESC_FG "1;" "Lorem ipsum" ESC_FG "15;" " dolor sit amet, consectetur adipiscing elit, "
		"sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim "
		"veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. "
		"Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat "
		"nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia "
		"deserunt mollit anim id est laborum.\r\n\n\t\t"
			"+-----------------------------------+" ESC_DOWN "1;" ESC_LEFT "38;"
			"|                                   |" ESC_DOWN "1;" ESC_LEFT "38;"
			"|                                   |" ESC_DOWN "1;" ESC_LEFT "38;"
			"|                                   |" ESC_DOWN "1;" ESC_LEFT "38;"
			"|                                   |" ESC_DOWN "1;" ESC_LEFT "38;"
			"|                                   |" ESC_DOWN "1;" ESC_LEFT "38;"
			"|                                   |" ESC_DOWN "1;" ESC_LEFT "38;"
			"|                                   |" ESC_DOWN "1;" ESC_LEFT "38;"
			"|                                   |" ESC_DOWN "1;" ESC_LEFT "38;"
			"+-----------------------------------+\r\n\n"

		ESC_BG "3;" "Contrary to popular belief, Lorem Ipsum is not simply random text. It has roots "
		"in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard "
		"McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the "
		"more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the "
		"cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum "
		"comes from sections 1.10.32 and 1.10.33 of \"" ESC_FG "1;" "de Finibus Bonorum et Malorum"
		ESC_FG "15;" "\" (The Extremes of Good and Evil) by Cicero, written in 45 BC. This book is a "
		"treatise on the theory of ethics, very popular during the Renaissance. The first line of "
		"Lorem Ipsum, \"Lorem ipsum dolor sit amet..\", comes from a line in section 1.10.32.";

	memset(videobuf_for_text, 0, 20*1024);
	cga_text_print(videobuf_for_text, 0,  0, 15, 0, lorem_ipsum);

	memset(videobuf_for_graphics, 0, 20*1024);

	int i = 0;
	for(int y = 64; y < 128; y += 16)
		for(int x = 68; x < 212; x += 16)
			cga_bitblit((uint8_t*)cga_sprites[(_sprite_idx + i) % 11],
				videobuf_for_graphics, x, y, 16, 16,
				CGA_VIDEO_WIDTH, CGA_VIDEO_HEIGHT);

	_sprite_idx++;
}
#endif
```

Приведенная выше функция **cga_video_test5()** печатает текст «Lorem Ipsum» в буфер **videobuf_for_text** слегка припудривая его разными цветами и форматированием с помощью ESC-последовательностей. Также она формирует в центре экрана стильное прямоугольное «окно» из символов «+», «-» и «|». В это окно будут отображаться анимированные спрайты, которые этой же функцией выводятся в буфер **videobuf_for_graphics** с соответствующим смещением чтобы попасть в то же место экрана где располагается текстовое окно.

Данные для отображения готовы. Теперь запрограммируем сам эффект. Для этого в главный цикл в функции **main()** вставим следующий код:

```
#if CGA_VIDEO_TEST == CGA_TEST_VIDEOFX_HYBRID
{
        cga_wait_vblank();                                // 1
        cga_set_video_mode(CGA_MODE_TEXT);                // 2
        memcpy(CGA->FB, videobuf_for_text, 20*1024);      // 3
        cga_wait_vblank_end();                            // 4

        cga_video_test5();                                // 5

        cga_wait_vblank();                                // 6
        cga_set_video_mode(CGA_MODE_GRAPHICS1);           // 7
        memcpy(CGA->FB, videobuf_for_graphics, 20*1024);  // 8
        cga_wait_vblank_end();
}
#endif
```

Стоит немного пояснить происходящий процесс. В данном коде производятся следующие действия:

1. Ожидание периода затенения. Все манипуляции с изображением на экране и режимами видеоадаптера нужно стараться выполнять в момент, пока изображение на экран не выводится.
2. Устанавливается текстовый видеорежим.
3. Быстро копируется содержимое ранее подготовленного **текстового** видеобуфера в реальный видеобуфер адаптера.
4. Ожидается конец периода обратного хода луча. С этого момента видеоадаптер начинает формировать текстовое изображение на экране.
5. Пока видеоадаптер занят формированием текстового изображения, программа вызывает функцию **cga_video_test5()** чтобы она сформировала новое изображение в буферах **videobuf_for_text** и **videobuf_for_graphics**. Напомню, что у нас отображаются анимированные спрайты и сам текст тоже может изменяться.
6. Ожидание периода затенения для следующего кадра.
7. Быстро копируется содержимое ранее подготовленного **графического** видеобуфера в реальный видеобуфер адаптера.

8. Ожидается конец периода обратного хода луча. С этого момента видеоадаптер начинает формировать графическое изображение на экране.
9. Процесс повторяется циклически.

Компилируем наш код, устраняем синтаксические ошибки, формируем битстрим, загружаем его в микросхему ПЛИС и наблюдаем на экране следующее анимированное изображение:

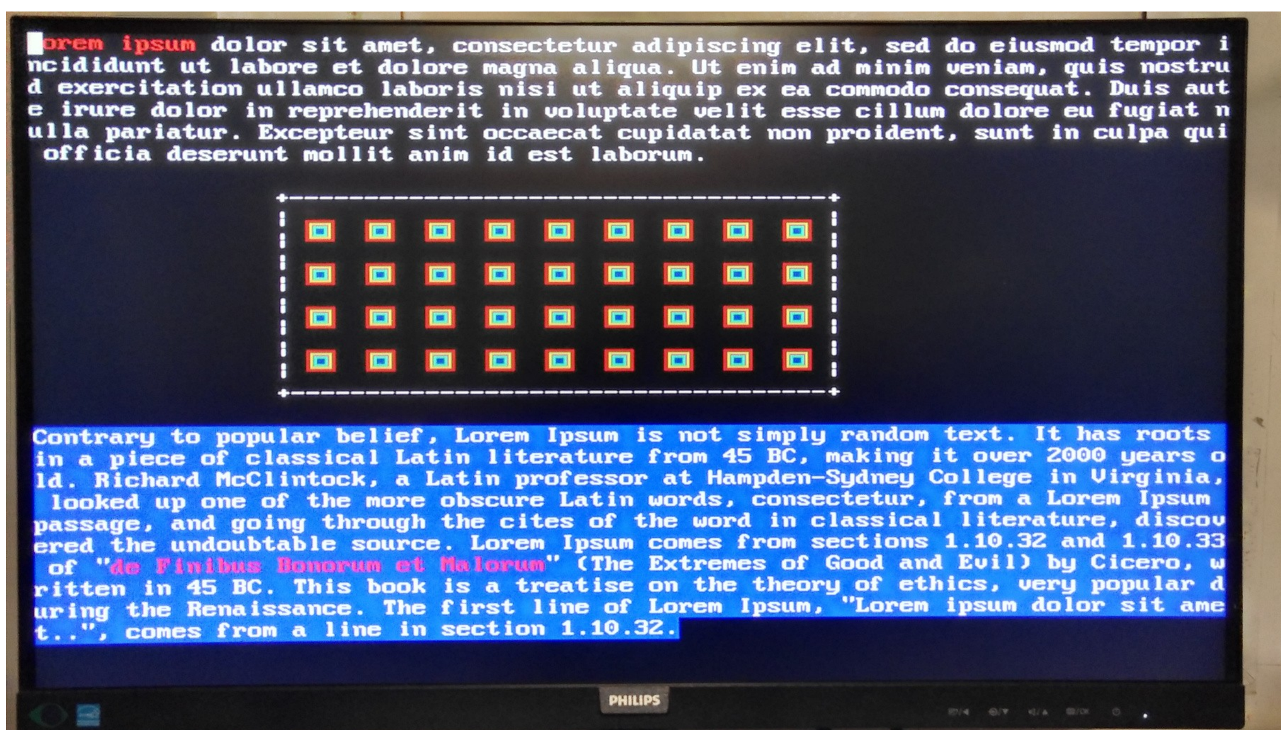


Рис. 19. Комбинирование текста и графики путем высокочастотного переключения выдеорежимов.

Более внушительно выглядит записанное видео демонстрации этого спецэффекта: https://www.youtube.com/watch?v=kjmBhLRtKTI

Этот эффект я когда-то давно подсмотрел в одной программе для MS-DOS на IBM PC/AT. В ней было много текстовых меню и другого разного текста который был сдобрен аналитической инфографикой. Судя по размеру используемого шрифта, программа отображала текст в текстовом режиме, а для отображения инфографики переключалась в графический режим 640x200 и все это происходило с большой частотой. На CGA и EGA частота строчной развертки в этих двух режимах (Mode 2 и Mode 6) была одинаковая и составляла ~15.7кГц, поэтому непрерывное переключение видеорежимов было почти незаметно глазу. Но я быстро определил в чем был подвох! :)

На этом, пожалуй, закончим со спецэффектами. Это далеко не все «недокументированные» возможности которые можно выжать из столь простого

видеоадаптера, но должен же я оставить читателю какое-то поле для исследовательской деятельности. Пишите в комментариях какие еще интересные спецэффекты на Ваш взгляд можно реализовать на этой аппаратуре.

10.4. Вишенка на торте

Приведенный ниже код реализует одновременно три спецэффекта: плавную вертикальную прокрутку текста, радужную построчную раскраску текста и высокочастотное переключение между текстовым и графическим видеорежимами для анимации спрайтов (перемещающихся «пульсирующих квадратов»).

```
#if (CGA_VIDEO_TEST == CGA_TEST_DEMO)
static unsigned char *videobuf_for_text = (unsigned char*)0x90001000;
static unsigned char *videobuf_for_graphics = (unsigned char*)0x90006000;

float my_sine(float x)
{
        x = ((int)(x * 57.297469) % 360) * 0.017452; // fmod(x, 2*PI)
       float res = 0, pow = x, fact = 1;

        for(int i = 0; i < 5; ++i) {
                res+=pow/fact;
                pow*=-1*x*x;
                fact*=(2*(i+1))*(2*(i+1)+1);
        }
        return res;
}

void cga_video_demo(void) {

        char *greetings =
                ESC_FG "5;"
                "\t\tGreetings to all users of Habr!\r\n"
                "\n"
                "\tHope you  enjoyed  reading  my post on  developing  CGA-like  video\r\n"
                "\tsubsystem for  FPGA  based synthesizable microcontroller  and liked\r\n"
                "\tthe VFXs demoed.\r\n"
                "\n"
                "\tThis work was inspired by those few who proceed coding for resource\r\n"
                "\tscarse systems  gaining  impossible  from them  using minuscule yet\r\n"
                "\tpowerful handcrafted tools.\r\n"
                "\n"
                "\t\tKudos to you dear old-time hacker fellows!\r\n"
                "\n"
                "\n"
                "\t\tSpecial credits to:\r\n"
                "\t\t==================\r\n"
                "\t" ESC_FG "14;" "Yuri Panchul" ESC_FG "5; for his work on basics-graphics-music 
Verilog labs\r\n"
                "\t" ESC_FG "14;" "Dmitry Petrenko" ESC_FG "5; for inspirations and ideas for Karnix 
FPGA board\r\n"
                "\t" ESC_FG "14;" "Vitaly Maltsev" ESC_FG "5; for helping with bitblit code 
optimization\r\n"
                "\t" ESC_FG "14;" "Victor Sergeev" ESC_FG "5; for demoscene inpirations\r\n"
                "\t" ESC_FG "14;" "Evgeny Korolenko" ESC_FG "5; for editing and testing\r\n"
                "\n"
                "\t\tThanks:\r\n"
                "\t\t=======\r\n"
                "\n"
                "\t" ESC_FG "14;" "@Manwe_Sand" ESC_FG "5; for his Good Apple for BK-0011M and other 
BK-0010 stuff\r\n"
                "\t" ESC_FG "14;" "@KeisN13" ESC_FG "5; for organizing FPGAs community in 
Russia\r\n"
                "\t" ESC_FG "14;" "@frog" ESC_FG "5; for CC'24 and keeping Russian demoscene 
running...\r\n"
                "\n"
                "\t\t*\t*\t*\n\n\n\n";
        memset(videobuf_for_text, 0, 20*1024);
        cga_text_print(videobuf_for_text, 0,  0, 15, 0, greetings);
```

```
        cga_set_cursor_xy(60, 10);

        memset(videobuf_for_graphics, 0, 20*1024);


        for(int y = 0; y < 240; y += 48)
                for(int x = 0; x < 320; x += 48)
                        cga_bitblit((uint8_t*)cga_sprites[(_sprite_idx) % 11],
videobuf_for_graphics,
                                x + 64 * my_sine((x+_sprite_idx)/80.0),
                                y + 64 * my_sine((x+_sprite_idx)/80.0),
                                16, 16, CGA_VIDEO_WIDTH, CGA_VIDEO_HEIGHT);

        _sprite_idx++;
}
#endif
```

И в главном цикле в функции **main()**:

```
                #if CGA_VIDEO_TEST == CGA_TEST_DEMO
                {
                        static int _scroll = 480;
                        static uint32_t colorfx_rainbow[] = {
                                // Straight
                                0x000000ff, 0x000055ff, 0x0000aaff, 0x0000ffff,
                                0x0000ffaa, 0x0000ff2a, 0x002bff00, 0x0080ff00,
                                0x00d4ff00, 0x00ffd400, 0x00ffaa00, 0x00ff5500,
                                0x00ff0000, 0x00ff0055, 0x00ff00aa, 0x00ff00ff,
                                // Reversed
                                0x00ff00ff, 0x00ff00aa, 0x00ff0055, 0x00ff0000,
                                0x00ff5500, 0x00ffaa00, 0x00ffd400, 0x00d4ff00,
                                0x0080ff00, 0x002bff00, 0x0000ff2a, 0x0000ffaa,
                                0x0000ffff, 0x0000aaff, 0x000055ff, 0x000000ff,
                        };

                        static int colorfx_offset = 6;

                        int colorfx_idx = colorfx_offset;

                        cga_wait_vblank();
                        cga_set_video_mode(CGA_MODE_TEXT);
                        cga_set_scroll(_scroll++);
                        memcpy(CGA->FB, videobuf_for_text, 20*1024);
                        cga_wait_vblank_end();

                        for(int i = 0; i < 480/2; i++) {
                                while(!(CGA->CTRL & CGA_CTRL_HSYNC_FLAG));
                                CGA->PALETTE[5] = colorfx_rainbow[colorfx_idx];
                                colorfx_idx = (colorfx_idx + 1) % 32;
                                while(CGA->CTRL & CGA_CTRL_HSYNC_FLAG);

                                while(!(CGA->CTRL & CGA_CTRL_HSYNC_FLAG));
                                while(CGA->CTRL & CGA_CTRL_HSYNC_FLAG);
                        }

                        cga_wait_vblank();
                        cga_set_video_mode(CGA_MODE_GRAPHICS1);
                        cga_set_scroll(0);
                        memcpy(CGA->FB, videobuf_for_graphics, 20*1024);
                        cga_wait_vblank_end();

                        cga_video_demo();
                }
                #endif
```

Результат представлен на видео: https://www.youtube.com/watch?v=5aY6kXc1IaU

На мой взгляд получилось неплохое «intro» в духе классической демосцены из 80-х.

Возможно ли было такое на IBM PC с CGA адаптером ? Скорее всего нет, но не из-за ограничений видеоадаптера, а по причине низкой производительности 16-ти битного микропроцессора i8086. Всё таки 32-х битная RISC машина в нашей синтезированной СнК на частоте 60 МГц позволяет многое.

11. Заключение

Написание данной статьи заняло у меня более трех месяцев, еще столько же занял кодинг и подготовка материалов. За это время мне довелось побывать в Санкт-Петербурге на фестивале «Chaos Constructions 2024» посвященного ретро-компьютерам, ретро-кодингу и демосцене. На фестивале я узнал про существование виртуальных платформ, в частности про TIC-80 — эдаких вымышленных идеальных 8-ми битных компьютеров для которых создаются демо, графика, музыка (и даже игры). Они также принимают участие в соревнованиях и для них выделена специальная категория. На мой взгляд, гораздо интересней иметь такую «fantasy console» на базе ПЛИС, а не в Web-браузере. С одной стороны, возможность синтезировать себе аппаратуру открывает ряд новых возможностей для творчества, с другой — сохраняет дух классической демосцены: оптимизация кода на ассемблере, борьба за каждый такт процессора и за каждый LUT, выжимание из аппаратуры максимума, ведь ПЛИС имеют свои физические ограничения. В этом свете мне кажется, что наша плата «Карно» оптимально укомплектована и может быть взята за основу для фентезийной ретро-консоли.

12. Что дальше ?

Разумеется Тетрис!

Хочется сделать простенькую игру в стиле 80-х с переливающейся двухмерной графикой и музыкой на три ШИМ канала.

Кстати о звуке. Есть желание прояснить возможность создания средствами ПЛИС энкодера для звука инкапсулируемого в HDMI(R), при этом не прибегая к использованию проприетарных IP блоков.

И третье направление исследовании — создание простой многозадачной операционной системы с виртуальной памятью. Такая ОС уже имеется — это Xv6, современная реализация классической операционной системы Unix версии 6. **Xv6** уже портирована на архитектуру RISC-V и её адаптация для нашей СнК не должна занять много времени.

PS: Тетрис, без музыки, сделать оказалось совсем не сложно. Спасибо пользователю **svedev0** с Github-а за его минималистичную реализацию игры на чистом Си из одного файла:
https://github.com/svedev0/tetris-c

Ссылка на видео с демонстрацией игры TetRISC-V на плате «Карно»:
https://www.youtube.com/watch?v=WuLQwD38eDc

13. Ссылки

1. Репозиторий с проектом СнК с CGA-подобным видеоадаптером для платы «Карно»:
https://github.com/Fabmicro-LLC/VexRiscvWithKarnix/tree/karnix_extended

2. Makefile для сборки находится тут:
https://github.com/Fabmicro-LLC/VexRiscvWithKarnix/tree/karnix_extended/scripts/KarnixTetris

3. Исходные коды игры TetRISC-V на языке «Си» для синтезируемого СнК:
https://github.com/Fabmicro-LLC/VexRiscvWithKarnix/tree/karnix_extended/src/main/c/karnix_tetris