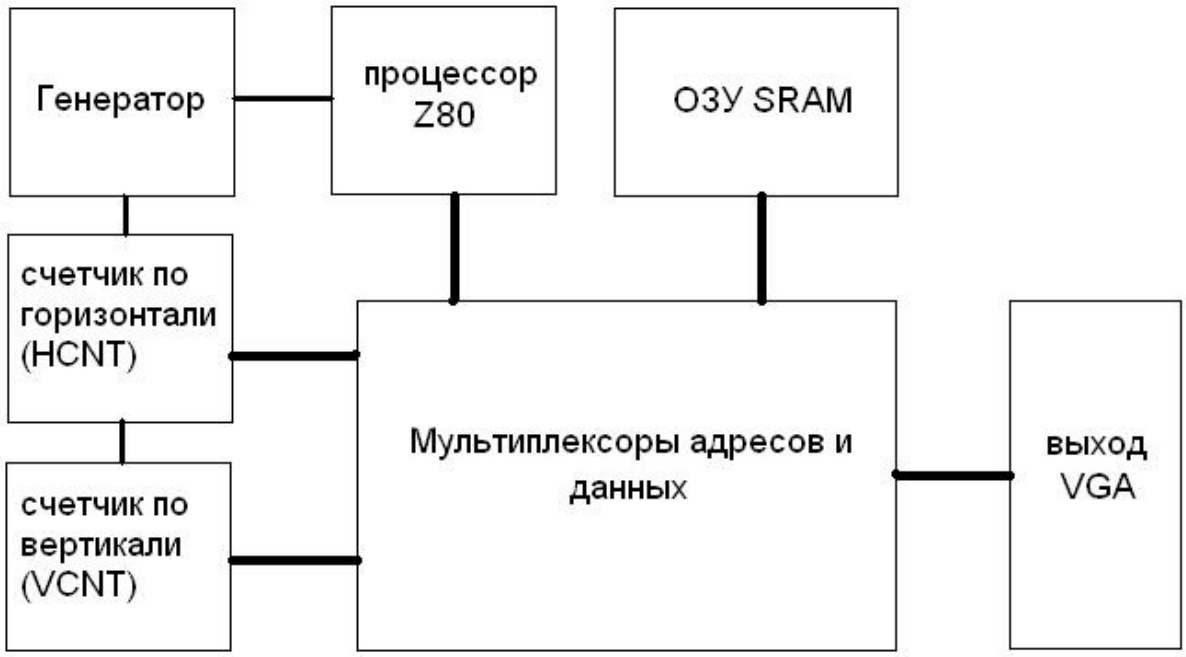
Проектирование ПК «ZX-Spectrum» на ПЛИС

Туториал для «чайников» и не только…

www.zx.pk.ru
Ewgeny7

В этом мануале я попытаюсь рассказать о принципах построения столь дорогих сердцам многих ретро-компьютеров на ПЛИС на примере простейшего ZX-Spectrum 48. Описанный здесь вариант построения микропроцессорной системы на ПЛИС далеко не единственный, но возможно наиболее простой для понимания. Для полноты примера к этому талмуду прилагается готовый проект.

Итак, вспомним, из чего собственно состоит простейший спектрум, например – Ленинград-1. Процессор, ОЗУ, ПЗУ, тактовый генератор, счетчики, мультиплексоры… В общем, примерно это можно изобразить так:



На рисунке в блок-схему уже внесены изменения, рассчитанные на создание проекта в ПЛИС. Например, тип ОЗУ обозначен как SRAM, в то время как в «настоящем» Ленинграл-1 тип памяти DRAM. Да и видеовыход мы будем делать сразу же применительно к стандарту VGA. Также на рисунке не указано ПЗУ, поскольку оно у нас будет «прибито» практически прямо к процессору.

Для проекта мы используем готовый модуль процессора T80, свободно валяющийся в сети. Но, поскольку нет в мире совершенства, мы используем не первый попавшийся, а максимально «реальный» модуль в редакции **syd**'а. Помимо исправленной работы некоторых команд, там добавлены дополнительные возможности по рулению состоянием процессора. Но мы это использовать не будем.

Также, для привинчивания клавиатуры PS/2 будет использован готовый модуль zxkbd, состоящий из двух файлов ☺.

Предполагается, что вы уже более-менее знакомы с Quartus и VHDL. И умеете создавать в «мастере», к примеру, модули PLL и LPM_ROM. Они нам понадобятся.

Итак, поехали.

1. Генераторы и счетчики.

Данный проект сделан применительно к девборде **u9_Reverse**, выполненной на основе чипа EP3C10E144 фирмы Альтера. Примененное ОЗУ – static RAM 512кб.

Создаем новый проект (или открываем прилагающийся).
Заголовок в головном файле проекта самый обычный:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.ALL;
```

Обзовем проект как-нибудь по-простому и привычному, например – speccy ☺.
Создаем описание всех пинов ввода/вывода, которые нам понадобятся:

```
entity speccy is
	port (
		-- CLOCK
		CLK_50MHz		: in std_logic;
		-- VGA
		VGA_R0			: out std_logic;
		VGA_R1			: out std_logic;
		VGA_R2			: out std_logic;
		VGA_G0			: out std_logic;
		VGA_G1			: out std_logic;
		VGA_G2			: out std_logic;
		VGA_B0			: out std_logic;
		VGA_B2			: out std_logic;
		VGA_B1			: out std_logic;
		VGA_VSYNC		: out std_logic;
		VGA_HSYNC		: out std_logic;
		-- SRAM
		SRAM_A			: out std_logic_vector(18 downto 0);
		SRAM_D			: inout std_logic_vector(7 downto 0);
		SRAM_nOE		: out std_logic;
		SRAM_nWE		: out std_logic;
		-- PS2
		PS2_KBCLK		: inout std_logic;
		PS2_KBDAT		: inout std_logic
);

end speccy;
```

Здесь мы описываем вход тактового сигнала с внешнего генератора 50Мгц, пины видеовыхода на VGA-разъем (по 3 бита на цветовую компоненту), выходы синхроимпульсов на монитор, сигналы на SRAM, клавиатуру, внешний пин «сброса».

В разделе архитектуры опишем сигналы для счетчиков:

```
signal clock            : std_logic;
signal clk_cnt          : std_logic_vector(1 downto 0);
signal hcnt             : std_logic_vector(8 downto 0);
signal vcnt             : std_logic_vector(9 downto 0);
signal flash            : std_logic_vector(4 downto 0);
```

Используя «мастер», создадим модуль PLL для преобразования входной частоты 50Мгц в необходимый нам пикселькклок 14Мгц.

```
component altpll0 is
port(
        inclk0          : IN STD_LOGIC  := '0';
        c0              : OUT STD_LOGIC
);
end component;
```

и тут же подключим его к сигналам проекта

```
PLL: altpll0
port map(
        inclk0          => CLK_50MHz,
        c0              => clock
);
```

Как видим, тактовая частота с внешнего генератора теперь поступает на вход модуля PLL, а с его выхода **c0** мы снимаем готовый пикселькклок **clock** частотой уже 14Мгц.

Небольшое пояснение относительно 14Мгц. У нас пикселькклок вдвое выше по частоте чем в реал-Ленинград-1. Дело в том, что мы будем делать видеовыход на VGA-монитор, а не телевизор. Простейший способ подключения к монитору состоит в том, что мы будем выводить строки изображения с удвоенной (относительно ТВ) частотой, при этом для сохранения фрейма спектрум-экрана каждая строка изображения будет выводиться два раза подряд. Т.е., для ТВ строки выводятся в последовательности 1-2-3-4-5-6… и так далее, у нас же – 1-1-2-2-3-3… Это простейший «скандаблер», дающий на выходе частоту кадровой развертки 50Гц при строчной частоте примерно 31кГц.
Строки выводятся в два раза быстрей, но их вдвое больше. За счет этого сохраняется фрейм спектрума. Отсюда следует, что и пикселькклок должен следовать с удвоенной частотой.

Для формирования кадра нам требуются счетчик «по горизонтали» (счетчик пикселей) **hcnt** и счетчик строк изображения **vcnt**. Порывшись в документациях на стандарт VGA, находим что для нашего случая длина одной строки изображения вместе с синхроимпульсом и прочими гашениями должна составить 32 микросекунды. Отсюда находим количество точек по горизонтали для нашего пикселькклока 14Мгц – 448 точек. Еще немного помучив калькулятор, находим, что количество строк для 50Гц должно составлять 624 строки. Поскольку строки у нас «двоятся», то формируемых строк должно быть 312 соответственно. Вспоминаем, что у

«оригинальных» аглицких спектрумов именно столько строк во фрейме. Из наших клонов столько строк изображения формируют не-пентагоны (Скорпион, КАЙ…). Путем несложных вычислений (50 *312 / 320) убеждаемся, что Пентагоновский фрейм имеет реальную частоту кадров 48,75Гц. Но мы его использовать не будем.

Итак, для подсчета 448 точек по горизонтали нам потребуется 9-разрядный счетчик. Для подсчета 624 фактических строк нам потребуется 10-разрядный счетчик. Что, собственно, и отображено в описании сигналов.

```
process (clock, hcnt)
begin
	if (clock'event and clock = '0') then
		if hcnt = 447 then
			hcnt <= "000000000";
		else
			hcnt <= hcnt + 1;
		end if;
	end if;
end process;
```

Счет ведется по фронту сигнала **clock**, по достижении значения 447 на следующем такте счетчик сбрасывается. Итого, от 0 до 447 получаем 448 точек на строку.

```
process (clock, hcnt, vcnt)
begin
if (clock'event and clock = '0') then
	if hcnt = 328 then
		if vcnt(9 downto 1) = 311 then
			vcnt(9 downto 1) <= "000000000";
		else
			vcnt <= vcnt + 1;
		end if;
	end if;
end if;
end process;
```

Здесь же счетчик тактируется сигналом **hcnt**. Точнее, когда **hcnt** достигает значения 328 то ближайший фронт **clock** увеличит значение **vcnt** на единицу.
Для удобства при дальнейших манипуляциях младший бит (0) счетчика **vcnt** игнорируется, поскольку он тупо перебирает пары одинаковых строк. Поэтому в дальнейшем что-то значащими разрядами для нас будут старшие (9..1) биты.

Для создания тактовой частоты для процессора применим счетчик **clk_cnt** на два разряда. Это даст нам частоты 7 и 3,5Мгц для тактирования процессора:

```
process (clock, clk_cnt)
begin
	if (clock'event and clock = '0') then
		clk_cnt <= clk_cnt + 1;
	end if;
end process;
```

Также, для будущего применения сразу создадим счетчик для формирования «моргалки» сигнала **flash**:

```
flash <= (flash + 1) when (vcnt(9)'event and vcnt(9)='0');
```

Пятиразрядный счетчик **flash** при тактировании старшим разрядом **vcnt** дает нам импульсы частотой: 50Гц/32=1,56Гц. Что вполне удобоваримо для мигания курсора.

Ну и до кучи сделаем генератор сигнала **int** для процессора. Опишем новые сигналы:

```
signal int              : std_logic;
signal cpu_int_n        : std_logic;
```

и опишем логику генератора:

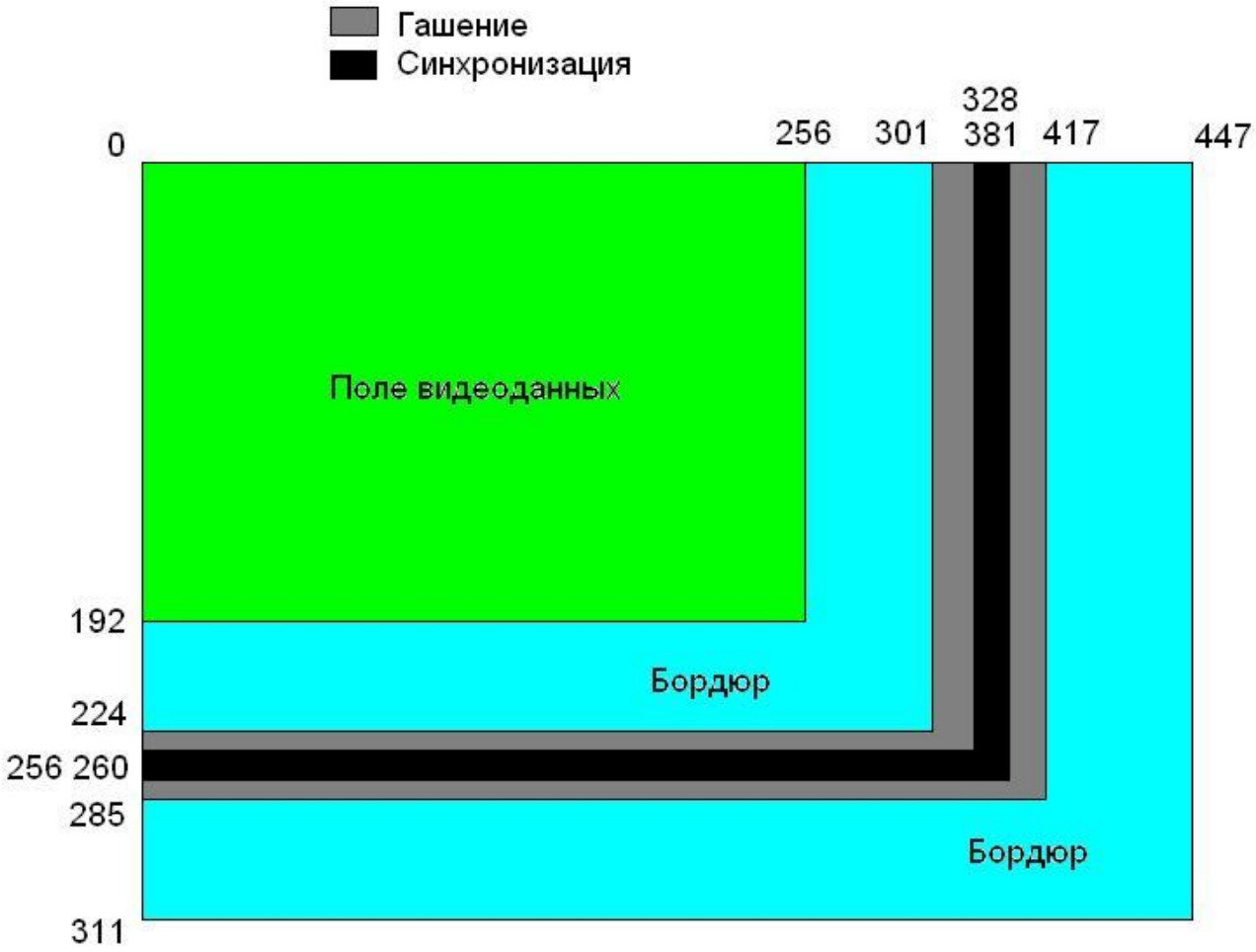
```
process (clock, vcnt, hcnt)
begin
	if (clock'event and clock = '1') then
		if (vcnt(9 downto 1) = 239 and hcnt = 316) then
			int <= '1';
		else
			int <= '0';
		end if;
	end if;
end process;
```

Сигнал **int** служит «стартером» для запуска импульса. Параметры 239 (строка фрейма) и 316 (пиксель от начала строки) определяют положение начала импульса на поле фрейма.

```
process (int, hcnt)
begin
	if (int'event and int = '1') then
		cpu_int_n <= '0';
	end if;
	if hcnt = 388 then
		cpu_int_n <='1';
	end if;
end process;
```

Сигнал **cpu_int_n** является уже непосредственно сигналом для процессора. Он активируется сигналом **int**, а выключается уже сигналом **hcnt**. Таким образом, длительность сигнала прерывания равна: 388 – 316 = 72 экранных пикселей. Сколько это будет в микросекундах можно подсчитать:
Длительность одной строки равна 32 мкс, 448 точек. Для 72 точек будет длительность сигнала – 32 / 448 * 72 = 5.14 мкс. Маловато, но сделано это преднамеренно – рассчитайте сами параметры для полноценных 8 мкс ☺.

Для наглядности и понимания, откуда здесь взялись все эти цифры, я нарисовал картинку, описывающую создаваемый нами фрейм:



2. Синхроимпульсы и служебные сигналы.

Займемся синтезом синхроимпульсов.
Стандарт VGA подразумевает импульсы отрицательной полярности определенной длины. Длину импульсов я когда-то рассчитывал, с тех пор пользуюсь готовыми цифрами. Ради интереса можете пересчитать сами. Попутно сделаем служебные сигналы screen (поле видеоданных) и blank (гашение).

Создаем сигналы:

```
signal hsync        : std_logic;
signal vsync        : std_logic;
signal screen       : std_logic;
signal blank        : std_logic;
```

Параметры для описываемых сигналов можно подглядеть на вышеприведенном рисунке фрейма.

Пишем процедуры:

```
process(clock, hcnt)
begin
	if (clock'event and clock = '1') then
		if hcnt = 328 then hsync <= '0';
		elsif hcnt = 381 then hsync <= '1';
		end if;
	end if;
end process;
```

Тут всё должно быть понятно. Триггер, устанавливаемый и сбрасываемый по состоянию счетчика пикселей **hcnt**. Кадровая синхронизация сделана аналогично, только вместо счетчика пикселей используется счетчик строк **vcnt**:

```
process (clock, vcnt)
begin
	if (clock'event and clock = '1') then
		if vcnt(9 downto 1) = 256 then vsync <= '0';
		elsif vcnt(9 downto 1) = 260 then vsync <= '1';
		end if;
	end if;
end process;
```

Напомню, что сигналы **hsync** и **vsync** – инверсны (активный «0»).
С синхрой разобрались, делаем сигнал **blank**. Этот сигнал активируется когда счетчики находятся в «зоне гашения» и синхронизации, т.е. он сообщает, что в этот момент никакие цветовые данные выводиться на монитор не должны:

```
process (clock, hcnt, vcnt)
begin
	if (clock'event and clock = '1') then
		 if (hcnt > 301 and hcnt < 417) or (vcnt(9 downto 1) > 224 and
vcnt(9 downto 1) < 285) then
			blank <= '1';
		else
			blank <= '0';
		end if;
	end if;
end process;
```

Сигнал screen сообщает, что пришла пора выводить на монитор содержимое видеопамяти. Имея в наличии сигналы screen и blank, нам легко определить момент когда нужно выводить данные о цвете бордюра. Ибо этот момент наступает, когда и screen, и blank неактивны.

```
process (clock, hcnt, vcnt)
begin
	if (clock'event and clock = '1') then
		if (hcnt < 256 and vcnt(9 downto 1) < 192) then
			screen <= '1';
		else
			screen <= '0';
		end if;
	end if;
end process;
```

Здесь всё просто, сигнал **screen** ограничен «квадратом» 0..191 по вертикали и 0..255 по горизонтали. Забегая вперед скажу, что этот сигнал является лишь предварительным, реально управлять выводом видеоданных будет «модифицированный» **screen1**. Но это будет попозже.

3. Подключение ОЗУ и ПЗУ

Подключение памяти начнем с ПЗУ. С помощью «мастера» создадим модуль lpm_rom с 16 килобайтами памяти. В качестве файла данных используем файлик 48.ROM из любого эмулятора спектрума. Здесь есть одно НО – мастер потребует предоставить ему файл в Intel Hex формате. Это несложно сделать преобразовав ромфайл с помощью программы WinHex (или еще какой-нибудь). В результате получим модуль lpm_rom0.vhd, который и подключим к проекту:

```
component lpm_rom0 is
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (13 DOWNTO 0);
		clock		: IN STD_LOGIC ;
		q		: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
	);
end component;

ROM: lpm_rom0
port map(
		address		=> cpu_a_bus(13 downto 0),
		clock		=> clock,
		q		=> rom_do
);
```

Соответственно, для подключения создадим новые сигналы.

```
signal rom_do		: std_logic_vector(7 downto 0);
signal cpu_a_bus	: std_logic_vector(15 downto 0);
```

Сигнал-шина **rom_do** будет поставлять нам данные с этого ПЗУ, а рулить адресом будет уже процессорная шина **cpu_a_bus**. Для ПЗУ нам надо 14 разрядов адресной шины, но мы сразу сделаем ее 16-разрядной, поскольку скоро будем привинчивать процессор. И сразу же добавим сигнал выборки ПЗУ **rom_sel**. Поскольку при работе процессора на младшей четвертинке адресного пространства вместо ОЗУ должно автоматом подключаться ПЗУ, то:

signal rom_sel : std_logic;

Руление сигналом будет реализовано старшими битами адресной шины процессора:

rom_sel <= '1' when (cpu_a_bus(15 downto 14) = "00") else '0';

Едем далее. ОЗУ. Для работы ОЗУ нам понадобится сразу несколько сигналов. Один – сигнал, информирующий что наступил цикл чтения видеоданных самим видеогенератором:

signal vid_sel : std_logic;

Этот сигнал активируется каждый экранный байт по два раза – чтение пикселов изображения, и чтение цветовых атрибутов. Как расположить эти моменты чтения «внутри» байта собственно значения не имеет, возьмем от балды. Поскольку три младших бита счетчика **hcnt** как раз и определяют биты внутри экранного байта, то привязку сделаем именно к этому счетчику:

vid_sel <= '1' when (hcnt(2 downto 1) = "10" and clock = '0') else '0';

Здесь видно, что я выбрал значения этих битов 4 и 5. Когда **hcnt(2 downto 0)** будет равно 4 (b'100) – мы прочитаем байт изображения (точек), когда 5 (b’101) – байт атрибутов цвета.
Также обязательно сделаем привязку к состоянию **clock = '0'**. Это необходимо для реализации «окна прозрачного доступа» к ОЗУ. Принцип работы этого окна я опишу позже.

Теперь нам придется сразу добавить несколько сигналов управления с процессора – запрос операций с памятью (cpu_mreq_n), сигнал чтения (cpu_rd_n) и сигнал записи (cpu_wr_n):

signal cpu_mreq_n : std_logic;
signal cpu_wr_n : std_logic;
signal cpu_rd_n : std_logic;

Для непосредственно руления памятью создавать новые сигналы не будем, сразу напишем логику управления пинами:

SRAM_D <= cpu_do_bus when (vid_sel = '0' and cpu_mreq_n = '0' and cpu_wr_n = '0') else "ZZZZZZZZ";

Т.е. данные будут напрвляться на ОЗУ только в случае если цикл чтения данных видеогенератором не активен, и процессор затребовал запись (**cpu_wr_n = '0'**)

данных в ОЗУ (**cpu_mreq_n = '0'**). Напомню, что сигналы управления у процессора Z80 – инверсные, активный «0».

SRAM_nOE <= '0' when (vid_sel = '1') or (cpu_mreq_n = '0' and cpu_rd_n = '0') else '1';

Чтение данных с ОЗУ активируется или когда активен цикл чтения видеогенератором, или когда процессор затребовал чтение из памяти. Всё просто и логично.

SRAM_nWE <= '0' when (vid_sel = '0' and cpu_mreq_n = '0' and cpu_wr_n = '0') else '1';

С записью также нет сложностей, видим, что сигнал записи активен когда не активен видеогенератор и процессор дал команду записи в память.

Теперь разберемся с потоком данных из ОЗУ в процессор. Делаем буферные сигналы, передающие данные на процессор и с процессора:

```
signal cpu_di_bus       : std_logic_vector(7 downto 0);
signal cpu_do_bus       : std_logic_vector(7 downto 0);
```

Теперь мы можем вставить мультиплексор, переключающий на процессор два потока данных – из ПЗУ (по активности сигнала выборки ПЗУ **rom_sel**) и из ОЗУ. Точнее, нужно прибавить еще один поток – «все биты – единицы». Это нужно в случае, если процессор обратится к каким-либо внешним портам, а тут косяк – их просто нету ☺. В такой ситуации процессор должен получить с шины данных байт h'FF. Делаем:

```
cpu_di_bus <=   rom_do when (rom_sel = '1' and cpu_mreq_n = '0') else
                SRAM_D when (rom_sel = '0' and cpu_mreq_n = '0') else
                "11111111";
```

Здесь, надеюсь, всё понятно?
Когда мы будем прикручивать клавиатуру, или еще какой порт/модуль, то этот блок мультиплексора будет «толстеть» новыми строчками кода.

А впрочем, давайте уже сразу добавим порт h'FE, все равно он нам пригодится для раскраски бордюра. А при добавлении клавиатуры мы будем и читать из него. Лепим:

```
signal port_fe_sel      : std_logic;
signal port_fe          : std_logic_vector(7 downto 0);
signal cpu_iorq_n       : std_logic;
```

Здесь мы создали сразу два сигнала. Первый сигнал – это сигнал выборки порта h'FE из общего адресного пространства. Второй – непосредственно сам порт h'FE. Реально разрядность у него конечно меньше восьми бит, но компилятор сам уберет все лишние разряды при синтезе. Третий сигнал – процессорный, «запрос операций с внешними устройствами». Как всегда, инверсный. Пишем логику работы:

port_fe_sel <= '1' when (cpu_a_bus(7 downto 0) = x"FE" and cpu_iorq_n = '0') else '0';

port_fe <= cpu_do_bus when (port_fe_sel = '1' and (cpu_wr_n'event and cpu_wr_n = '0'));

С сигналом выборки надеюсь вопросов не возникает. С процессом записи в порт разберемся чуть подробнее.

Перевожу строчку кода на русский язык: «записать в порт **port_fe** данные с шины **cpu_wr_n** когда порт выбран, и только в момент ниспадающего фронта сигнала записи с процессора». Ниспадающий фронт сигнала **cpu_wr_n** выбран не случайно, это событие возникает когда процессор уже приготовил свои данные и выдал их на шину. Т.е. внешнее устройство спокойно может эти данные забирать.

Теперь наступает самая сложная для детального понимания часть – формирование сигналов шины адреса для ОЗУ. Блок реализован на двух мультиплексорах и выглядит так:

```
process (vid_sel, vcnt, hcnt, cpu_a_bus)
begin
	if vid_sel = '1' then
		case hcnt(0) is
			when '0' => SRAM_A <= "000" & "010" & vcnt(8 downto 7) &
			vcnt(3 downto 1) & vcnt(6 downto 4) & hcnt(7 downto 3);
			when '1' => SRAM_A <= "000" & "010110" &
			vcnt(8 downto 4) & hcnt(7 downto 3);
		end case;
	else
		SRAM_A <= "000" & cpu_a_bus;
	end if;
end process;
```

Хлебнув пивка, начинаем разбирать что же тут сильно вумный дядько написал такое. Начнем с внешнего мультиплексора, сделанного на конструкции **if..then..else..end if**. Здесь переключение делается сигналом выборки цикла видеогенератора **vid_sel**. Если цикл неактивен, то по **else** на шину адреса ОЗУ поступают три ноля в старшие биты и шина адреса с процессора. Шина процессора у нас шестнадцатибитная, а разрядность ОЗУ – 19 бит. Поэтому нулями мы просто отсекаем неиспользуемую память. В этом случае процессор спокойно себе общается с ОЗУ и не парится.

Если же видеоцикл активен, то в действие включается второй мультиплексор, реализованный разнообразия ради на конструкции **case…when…end case**.
В свою очередь, в этом блоке рулит уже сигнал **hcnt(0).** Напрягаем свой склероз, и вспоминаем что я там говорил про сигнал **vid_sel**. Что он активен, когда три младшие бита счетчика пикселей **hcnt** равны по значению цифрам 4 или 5. Вот этот «довесок» в виде **hcnt(0)** как раз и определяет конкретно, какая цифра сейчас в этих битах. Если разряд равен '0' (а значение счетчика – 4), то на шину адреса выплёвывается сложная конструкция из запутанных битов счетчиков **hcnt** и **vcnt**. Этим на шину выставляется адрес в ОЗУ, где хранится байт изображения.
Если значение равно '1', то на шине – тоже запутка, но уже другая. И теперь считывается байт атрибутов цвета.

Вообще-то, этот блок руления адресами характерен для создания любого видеогенератора. Похожие блоки я использовал в **«Орионе-2010», «u10_Spetz»**. Разница в основном только в сигналах счетчиков.

Давайте разберемся в этих сигналах. Накопав в сети всякоразные доки и скопипастив нужное, смотрим сюда:

Не правда ли, что-то это напоминает?
Дествительно, это весьма похоже на наши строчки кода, описывающие адреса в видеопамяти. С одной только поправкой – циферки битов для счетчика **v** (в нашем случае – **vcnt**) на рисунке меньше на единицу, по сравнению с нашим кодом. Почему это так – я уже писал выше. Вспоминайте ☺. Маленькая подсказка – на рисунке описана «шифрация» для ТВ-выхода.

На этом мучения с памятью заканчиваются. Пора прикошачивать процессор. Чем мы сейчас и займемся.

3. Подключаем процессор

Процессор Z80 для ПЛИС реализован в версии на языке VHDL (T80) и на Verilog (TV80). Воспользуемся первым. Для этой реализации спектрума я выбрал проект T80 в творческой обработке syd'a, как самый безглючный.

Копируем в папку проекта speccy папку с софтядром T80. Подключаем необходимые модули. Их шесть:

T80s.vhd
T80.vhd
T80_ALU.vhd
T80_MCode.vhd
T80_Pack.vhd
T80_Reg.vhd

Эти модули собственно и представляют собой готовый к работе процессор Z80 с раздельными шинами данных для входных и выходных сигналов.

Процепляем головной модуль процессора T80s.vhd к нашему проекту:

```
component T80s is
	generic (
		Mode		: integer := 0;
		T2Write		: integer := 1;
		IOWait		: integer := 1 );
	port (
		RESET_n		: in std_logic;
		CLK_n		: in std_logic;
		WAIT_n		: in std_logic;
		INT_n		: in std_logic;
		NMI_n		: in std_logic;
		BUSRQ_n		: in std_logic;
		M1_n		: out std_logic;
		MREQ_n		: out std_logic;
		IORQ_n		: out std_logic;
		RD_n		: out std_logic;
		WR_n		: out std_logic;
		RFSH_n		: out std_logic;
		HALT_n		: out std_logic;
		BUSAK_n		: out std_logic;
		A		: out std_logic_vector(15 downto 0);
		DI		: in std_logic_vector(7 downto 0);
		DO		: out std_logic_vector(7 downto 0);
		RestorePC_n	: in std_logic );
end component;

Z80:T80s
port map (
	RESET_n		=> '1',
	CLK_n		=> clk_cpu,
	WAIT_n		=> '1',
	INT_n		=> cpu_int_n,
	NMI_n		=> '1',
	BUSRQ_n		=> '1',
	M1_n		=> open,
	MREQ_n		=> cpu_mreq_n,
	IORQ_n		=> cpu_iorq_n,
	RD_n		=> cpu_rd_n,
	WR_n		=> cpu_wr_n,
	RFSH_n		=> open,
	HALT_n		=> open,
	BUSAK_n		=> open,
	A		=> cpu_a_bus,
	DI		=> cpu_di_bus,
	DO		=> cpu_do_bus,
	RestorePC_n	=> '1'
);
```

Все подключенные сигналы нам уже знакомы. Выход (точнее, вход) процессора **RESET_n** пока оставляем просто подключенным к '1'. Туда же подтягиваем входы **WAIT_n**, **NMI_n**, **BUSRQ_n**. Поскольку торможения процессора у нас не предусмотрено, немаскируемое прерывание не применяется, захвата шин внешними устройствами не предвидится – эти сигналы нам не нужны. Обойдемся без них, повесив соответствующие входы на единицу. В таком виде процессор будет готов к работе и стартует сразу после запуска проекта.

Больше относительно подключения процессора мне добавить и нечего. Пускай это будет самая короткая глава в этом мануале ☺.

И перейдем к следующей, достаточно запутанной главе о реализации видеовыхода.

4. Видеовыход и чтение видеоданных.

Для начала разберемся, откуда берутся видеоданные для вывода их на монитор.

Создаём сразу целую пачку сигналов:

```
signal screen1          : std_logic;
signal vid_0_reg        : std_logic_vector(7 downto 0);
signal vid_1_reg        : std_logic_vector(7 downto 0);
signal vid_b_reg        : std_logic_vector(7 downto 0);
signal vid_c_reg        : std_logic_vector(7 downto 0);
signal vid_dot          : std_logic;
signal r,rb             : std_logic;
signal g,gb             : std_logic;
signal b,bb             : std_logic;
```

Начну с хвоста. Сигналы с загадочными именами **r**, **g**, **b** – как нетрудно догадаться, сигналы цветности. Они у нас будут передавать эти самые сигналы на ножки ПЛИСы для дальнейшего их лицезрения на мониторе. Сигналы **rb**, **gb**, **bb** – из той же серии, только они будут передавать информацию о яркости цветовых сигналов.
Vid_dot – сигнал, содержащий информацию о текущем пикселе изображения.
vid_0_reg, vid_1_reg – регистры, в которые производится чтение из видеопамяти ОЗУ. Они являются буферными, поскольку далее из них информация поступает в регистры **vid_b_reg** и **vid_c_reg**.
Сигнал **screen1**. Я уже упоминал о нем, сейчас расскажу для чего он нужен.
Для того, чтобы вывести на экран считанную из ОЗУ информацию, нам надо умудриться ее успеть считать до того, как она появится на экране. Мы применим такой способ – в самом начале фрейма, когда все счетчики обнулены, и сигнал screen «включится», мы выводить на экран еще ничего не будем. Потому, что само считывание информации у нас произойдет только на 4 и 5 тактах видеогенератора. Получив из памяти данные в буферные регистры изображения **vid_0_reg** и цвета **vid_1_reg** , мы дождемся окончания этого этого экранного байта и стремительно сделаем две вещи – скопируем информацию из буферных регистров в «рабочие» регистры **vid_b_reg** и **vid_c_reg**, и перепишем состояние сигнала **screen** в «рабочий» сигнал **screen1**. С этого момента у нас включается отображение

видеоданных на монитор. В конце строки экранной области произойдет аналогичное – сначала «отключится» **screen**, но на экран еще будет выводиться информация, а на следующем байте отключится и **screen1**. Всё. После этого включается отображение цвета бордюра и далее по списку.

Итак, как у нас читаются видеоданные. Вспоминаем, процессор у нас спокойно работает с ОЗУ, но ВНЕЗАПНО активируется сигнал **vid_sel**. Происходит это на 4 и 5 тактах счетчика **hcnt(2 downto 0)**. При этом исключительно в фазе '0' пикселькlока. В эти моменты у нас отключаются все сигналы с процессора на ОЗУ, на сигнал чтения **SRAM_nOE** подается '0' (т.е. запрос на считывание информации), на шину адреса ОЗУ выставляется комбинация сигналов счетчиков **hcnt** и **vcnt**, которая в зависимости от состояния разряда **hcnt(0)** несет информацию о байте изображения или атрибутов. Время замееееедлееенннооо течет, ОЗУ выставляет на шине данных запрошенную информацию… Пикселькlок меняет фронт на '1'… В этот момент:

```
process (clock, hcnt)
begin
	if hcnt(2 downto 0) = "100" then
		if (clock'event and clock = '1') then
			vid_0_reg <= SRAM_D;
		end if;
	end if;
end process;
```

… защелкивается информация о байте изображения,

```
process (clock, hcnt)
begin
	if hcnt(2 downto 0) = "101" then
		if (clock'event and clock='1') then
			vid_1_reg <= SRAM_D;
		end if;
	end if;
end process;
```

… или байта цвета.

Сигнал **vid_sel** снова отключается, и процик подключается к шинам ОЗУ. Поскольку процессор у нас тактируется также пикселькlоком (точнее, «поделенным» пикселькlоком), но по фронту '0', а не '1', то он просто «не замечает» этого кратковременного отключения от шин. После деактивации сигнала **vid_sel** проходит целых полклока, в течении которых ОЗУ спокойно успевает по выставленному процессором адресу выдать на шину данные (или записать их). Этот минимальный временной промежуток равен при частоте клока в 14Мгц (полклока = 28Мгц) – 35,71нс. А время доступа для примененного ОЗУ – максимум 10нс. Т.е. мы спокойно можем увеличивать частоту пикселькlока в 2 и более раз. Вот таким образом и реализовано то самое окно «прозрачного доступа». Есть и другие способы его реализации, я описал лишь самое простое решение для имеющегося «железа».

Итак, время тикает, заканчивается текущий экранный байт. На последнем такте **hcnt(2 downto 0)** происходит следующее:

```
process (hcnt, clock)
begin
	if hcnt(2 downto 0) = "111" then
		if (clock'event and clock = '1') then
			vid_b_reg  <= vid_0_reg;
			vid_c_reg  <= vid_1_reg;
			screen1    <= screen;
		end if;
 end if;
end process;
```

Это как раз то, о чем я говорил выше. Видеоданные переписаны в рабочие регистры изображения, сигнал **screen1** дает добро на вывод информации. Неустанно работает сдвиговый регистр, передающий сигналу **vid_dot** с каждым тактом по очередному биту регистра изображения **vid_b_reg**:

```
process (hcnt, vid_b_reg)
begin
	case hcnt(2 downto 0) is
		when "000" => vid_dot <= vid_b_reg(7);
		when "001" => vid_dot <= vid_b_reg(6);
		when "010" => vid_dot <= vid_b_reg(5);
		when "011" => vid_dot <= vid_b_reg(4);
		when "100" => vid_dot <= vid_b_reg(3);
		when "101" => vid_dot <= vid_b_reg(2);
		when "110" => vid_dot <= vid_b_reg(1);
		when "111" => vid_dot <= vid_b_reg(0);
	end case;
end process;
```

И вот мы подходим к финалу этой страшной истории.
Мы имеем сейчас информацию о текущей точке изображения в **vid_dot**, и информацию о раскраске в **vid_c_reg**. Надо бы это как-нибудь использовать ☺.

Находим в интернетах (или в умных книжках) информацию о содержимом байта аттрибутов.
Первые три бита у нас отвечают за цвет INK, следующие три бита – PAPER, шестой бит у нас рулит яркостью, и наконец седьмой – признак «мыргания» FLASH.
Замешаем всю эту информацию в процедуру.
Попутно учтем тот неоспоримый факт, что вся эта цветовая требуха выводится только тогда, когда активен сигнал screen1. Если он не активен, то ситуацией уже рулит сигнал blank. Если он равен '1', то мы не выводим на монитор ничего, кроме синхроимпульсов, если же равен нулю, то из регистра port_FE выводим три младших бита, определяющих цвет бордюра. Также, необходимо предусмотреть ситуацию с «ярким черным», когда байт изображения после всех процедур смешивания имеет черный цвет, но бит яркости в атрибутах включен. В этой ситуации необходимо игнорировать яркость чтобы на экране монитора не появлялся серый «мусор».

Итак, перекрестясь, лепим:

```
process(screen1, blank, hcnt, vid_dot, vid_c_reg, clock, flash)
variable selector: std_logic_vector(2 downto 0);
begin
selector:=vid_dot & flash(4) & vid_c_reg(7);
if (clock'event and clock = '1') then
  if blank = '0' then
    if screen1 = '1' then
      case selector is
        when "000"!"010"!"011"!"101" => b <= vid_c_reg(3);
                                        bb <= (vid_c_reg(3) and vid_c_reg(6));
                                         r <= vid_c_reg(4);
                                         rb <= (vid_c_reg(4) and vid_c_reg(6));
                                         g <= vid_c_reg(5);
                                        gb <= (vid_c_reg(5) and vid_c_reg(6));
        when "100"!"001"!"111"!"110" => b <= vid_c_reg(0);
                                        bb <= (vid_c_reg(0) and vid_c_reg(6));
                                         r <= vid_c_reg(1);
                                         rb <= (vid_c_reg(1) and vid_c_reg(6));
                                         g <= vid_c_reg(2);
                                        gb <= (vid_c_reg(2) and vid_c_reg(6));
    end case;
  else
        b <= port_fe(0);
        r <= port_fe(1);
        g <= port_fe(2);
        rb <= '0';
        gb <= '0';
        bb <= '0';
  end if;
 else
        b <= '0';
        r <= '0';
        g <= '0';
        rb <= '0';
        gb <= '0';
        bb <= '0';
 end if;
end if;
end process;
```

Смотрится грозно, но ничего особо страшного здесь нет. Разбор происходящего мы уже сделали чуть выше, поэтому давайте допишем последние операторы:

```
VGA_R2 <= r;
VGA_G2 <= g;
VGA_B2 <= b;
VGA_R1 <= rb;
VGA_G1 <= gb;
```

```
VGA_B1 <= bb;
VGA_R0 <= 'Z';
VGA_G0 <= 'Z';
VGA_B0 <= 'Z';
```

Здесь всё просто, мы выводим на пины сформированные сигналы цвета. Поскольку на u9_Reverse цвет сделан трёхбитным, то самому младшему битику мы присваиваем значение 'Z', т.е. он вообще никак не будет учавствовать в раскраске картинки на мониторе.

Скомпилировав всё, что мы создали на протяжении этого мануала, и залив конфиг в ПЛИС девборды, можно будет как эта бездушная железяка оживет, мелькнет пёстрый квадрат содержимого видеопамяти, проскочит черный квадрат с красными полосками теста системы, экран очистится и мы увидим рождение зверушки:

Вдоволь насмотревшись и уронив скупую слезу, пошлем жену за клавиатурой, тёщу за пивом и собаку за тапочками. Нам предстоит подключение клавиатуры!

5. Подключение PS/2 клавиатуры

Клавиатурный модуль состоит из двух файлов. Первый работает с клавиатурой на «низком уровне», т.е. непосредственно принимает сигналы с клавиатуры, преобразует их в сканкоды и передает второму, который уже непосредственно эмулирует матрицу 40-кнопочной спектрумовской клавиатуры и в соответствии с полученными сканкодами «нажимает кнопочки» на матрице.
Добавляем эти файлы в проект.

Объявляем и подключаем новый модуль:

```
component zxkbd is
   port(
         clk                  :in std_logic;
         reset                :in std_logic;
         res_k                :out std_logic;
         ps2_clk              :in std_logic;
         ps2_data             :in std_logic;
         zx_kb_scan           :in std_logic_vector(7 downto 0);
         zx_kb_out            :out std_logic_vector(4 downto 0);
         k_joy                :out std_logic_vector(4 downto 0);
         f                    :out std_logic_vector(12 downto 1);
         num_joy              :out std_logic
);
end component;

zxkey: zxkbd
port map(
   clk                  => clock,
   reset                => '0',
   res_k                => res_n,
   ps2_clk              => PS2_KBCLK,
   ps2_data             => PS2_KBDAT,
   zx_kb_scan           => kb_a_bus,
   zx_kb_out            => kb_do_bus,
   k_joy                => open,
   f                    => open,
   num_joy              => open
);
```

Как видим, добавляются новые сигналы. Ну, **PS2_KBCLK** и **PS2_KBDAT** у нас уже зарегистрированы как пины. Их объявлять не нужно.
Сигналы (шины) **kb_a_bus** и **kb_do_bus** предназначены для получения модулем старших адресных линий процессора для сканирования матрицы, и выдачи пятибитового кода состояния кнопок по запрошенным линиям матрицы.
Выход **res_n** наконец-то добавит в проект официальный сигнал сброса ☺.

Объявляем сигналы:

```
signal res_n            : std_logic;
signal kb_a_bus         : std_logic_vector(7 downto 0);
signal kb_do_bus        : std_logic_vector(4 downto 0);
```

Выводим в клавиатурный модуль шину адреса процессора:

```
kb_a_bus <= cpu_a_bus(15 downto 8);
```

Можно было и обойтись без сигнала **kb_a_bus**, просто подведя **cpu_a_bus(15 downto 8)** прямо к модулю, но мне вот захотелось сделать именно так ☺.

Так, теперь нам надо умудриться прочитать данные с клавиатуры. Процессор это делает обращаясь к целому вееру портов, оканчивающихся на h'FE. Воспользуемся тем, что селектор этого порта у нас уже сделан:

```
cpu_di_bus <=     rom_do when (rom_sel = '1' and cpu_mreq_n = '0') else
                  SRAM_D when (rom_sel = '0' and cpu_mreq_n = '0') else
                  "111" & kb_do_bus when (port_fe_sel = '1') else
                  "11111111";
```

Как видите, ничего хитрого, просто добавил строчку чтения данных из порта FE, когда процессор к ним обращается.

И напоследок – подключим выдаваемый клавиатурным модулем на халяву сигнал сброса res_n. Нам собственно сбрасывать и нечего, кроме процессора. Изменим строчку в его подключении:

```
RESET_n           => res_n,
```

Компилируем. Заливаем. Ждем появления сообщения через века от сэра Клайва Синклера. Жмем любую кнопочку. О, чудо! Теперь будет с чем поиграться ☺. Напомню напоследок, что кнопка сброса – это «Scroll Lock».

За сим заканчиваю этот мануал. Надеюсь, он пойдет вам на пользу, или как минимум – доставит удовольствие от прочтения.

Всегда Ваш – **Ewgeny7**

ewgeny7@speccy.su

www.zx.pk.ru

2011г.