

РЕАЛИЗАЦИЯ ЦИФРОВЫХ ПРОТОКОЛОВ ПЕРЕДАЧИ ИНФОРМАЦИИ И СИСТЕМ НА КРИСТАЛЛЕ НА ПЛИС

Методические указания к лабораторным работам



Рязань 2017

УДК 681.3.06

Реализация цифровых протоколов передачи информации и систем на кристалле на ПЛИС: методические указания к лабораторным работам / Рязан. гос. радиотехн. ун-т; сост. И.С. Холопов. Рязань, 2017. 48 с.

Содержат описание синтеза на ПЛИС систем передачи данных по интерфейсам SPI, I2C, VGA, DVI и LVDS, а также основ проектирования систем на кристалле на основе процессоров семейства Nios II. Методические указания ориентированы на разработку проектов в средах Quartus II и Eclipse на ПЛИС фирмы «Altera».

Работы предназначены для магистрантов направления 11.04.01 – «Радиотехника», изучающих дисциплину «Основы проектирования систем на ПЛИС».

Автор выражает благодарность студентам гр.210 Бирюкову Антону и Макаркину Павлу за помощь в разработке макетов к лабораторным работам по изучению протоколов SPI и I2C.

Табл. 11. Ил. 25. Библиогр.: 16 назв.

Цифровой конечный автомат, протоколы SPI, I2C, VGA, DVI, LVDS, система на кристалле, процессор Nios II

Печатается по решению редакционно-издательского совета Рязанского государственного радиотехнического университета.

Рецензент: кафедра РТС РГРТУ (зав. кафедрой В.И. Кошелев)

Реализация цифровых протоколов передачи информации
и систем на кристалле на ПЛИС

Составитель Х о л о п о в Иван Сергеевич

Редактор М.Е. Цветкова
Корректор С.В. Макушина

Подписано в печать 27.02.17. Формат бумаги 60 х 84 1/16.

Бумага писчая. Печать трафаретная. Усл. печ. л. 3,0.

Тираж 50 экз. Заказ

Рязанский государственный радиотехнический университет.

390005, Рязань, ул. Гагарина, 59/1.

Редакционно-издательский центр РГРТУ.

Лабораторная работа № 1

Изучение протокола передачи данных SPI

Цель работы: синтез цифрового автомата для реализации информационного обмена с микросхемами, поддерживающими интерфейс SPI (читается как «эс-пи-ай»), на примере цифрового трехосного акселерометра ADXL345 фирмы Analog Devices.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Интерфейс SPI (от англ. *Serial Peripheral Interface* – последовательная шина для подключения внешних устройств) разработан компанией Motorola. SPI – дуплексный синхронный протокол последовательного обмена данными, реализованный по принципу *Master - Slave* (ведущий - ведомый). Обязательным условием передачи данных по шине SPI является генерация сигнала синхронизации SCLK (*Serial Clock*). Этот сигнал имеет право генерировать только *Master*. Данные по протоколу SPI передаются последовательными байтами, бит за битом, начиная со старшего бита. Максимальная длина шины SPI может достигать 5 м, а максимальная частота передаваемых по ней тактовых импульсов – 10 МГц. Некоторые микросхемы быстродействующих аналого-цифровых преобразователей (АЦП) поддерживают частоту тактовых импульсов шины SPI до 100 МГц.

Существует два основных типа подключения к шине SPI:

- 1) подключение одного устройства *Slave*;
- 2) подключение нескольких устройств *Slave*.

Электрическая схема подключения первого типа приведена на рис. 1. На время передачи или приема данных необходимо, чтобы в линии \overline{SS} (от англ. *Slave Select* – выбор *Slave*) действовал низкий логический уровень. В противном случае устройство неактивно. *Master*

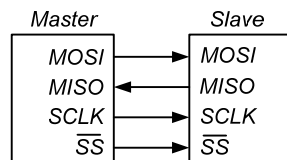


Рис. 1

передает данные по линии MOSI (от англ. *Master Output* → *Slave Input* – выход *Master* → вход устройства) синхронно с сигналом SCLK (от англ. *Serial clock* – последовательный тактовый сигнал), а *Slave* принимает биты данных по фронту принятого сигнала синхронизации (переднему или заднему в зависимости от режима работы). Устройство отправляет свои данные по линии MISO (от англ. *Master Input* → *Slave Output* – вход *Master* → выход устройства).

Электрическая схема подключения второго типа (параллельное подключение n устройств) приведена на рис. 2.

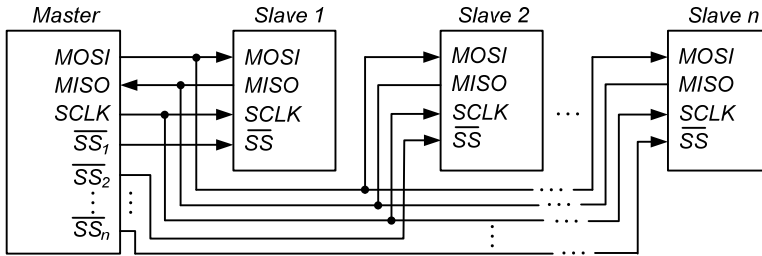


Рис. 2

Все сигнальные линии, кроме \overline{SS} , соединяются параллельно, а *Master* переводом в состояние логического нуля сигнала \overline{SS}_i , $i = 1, n$, определяет, с каким устройством он будет обмениваться данными. Главным недостатком такого подключения является необходимость в дополнительных линиях для адресации, так как общее число линий связи для подключения n устройств равно $(3 + n)$.

Альтернативные обозначения линий связи MOSI, MISO, SCLK и \overline{SS} , используемые различными производителями микросхем, сведены в табл. 1.

Варианты обозначения сигнальных линий шины SPI Таблица 1

| Основное обозначение | Альтернативное обозначение | |
|----------------------|----------------------------|---------------|
| | <i>Master</i> | <i>Slave</i> |
| MOSI | DO, SDO, DOUT | DI, SDI, DIN |
| MISO | DI, SDI, DIN | DO, SDO, DOUT |
| SCLK | DCLOCK, SPC, CLK, SCK | |
| SS | CS | |

Протокол передачи SPI заключается в выполнении операции побитного ввода и вывода данных по фронтам сигнала синхронизации. Установка данных при передаче и их чтение при приеме выполняются по противоположным фронтам тактового сигнала SCLK. Существует четыре варианта (режима) работы интерфейса SPI, которые описываются двумя параметрами [1, 2].

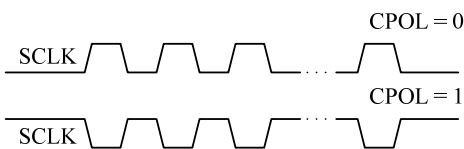


Рис. 3

1. CPOL (от англ. **Clock Polarity**) – исходный уровень сигнала синхронизации.

Если CPOL = 0, то линия SCLK до начала цикла обмена данными и

после его окончания имеет низкий логический уровень, т.е. первый фронт – нарастающий; если $CPOL = 1$ – то в линии в состоянии отсутствия обмена данными действует высокий логический уровень, т.е. первый фронт – спадающий (см. рис. 3).

2. $CPHA$ (от англ. *Clock Phase*) – фаза синхронизации. От этого параметра зависит, в какой последовательности выполняется установка и чтение данных: если $CPHA = 0$, то по первому фронту тактового сигнала $SCLK$ выполняется чтение данных, а по второму – их установка; если $CPHA = 1$ – то наоборот (см. диаграммы в табл. 2).

Номер режима SPI (от 0 до 3) принято обозначать дибитом (парой бит) $CPOL$, $CPHA$ (табл. 2).

Режимы работы SPI

Таблица 2

| Режим SPI | CPOL | CPHA | Диаграммы сигналов в линиях шины SPI |
|-----------|------|------|--------------------------------------|
| 0 | 0 | 0 | |
| 1 | 0 | 1 | |
| 2 | 1 | 0 | |
| 3 | 1 | 1 | |

Master и *Slave*, работающие в различных режимах, являются несовместимыми, что является недостатком протокола SPI.

При обмене данными по шине SPI возможны прием и чтение как одного байта, так и нескольких байт подряд. Микросхема *Slave* содержит несколько регистров, в которые можно записывать данные и/или считывать их. Каждый регистр имеет свой адрес A (упрощенно карта регистров *Slave* приведена на рис. 4); объем регистра, как правило, составляет 8 или 16 бит.

Slave

| | |
|------|----|
| A1 | A2 |
| A3 | A4 |
| ~ | |
| An-1 | An |

Рис. 4

Рассмотрим обмен данными по SPI в режиме 3 (рис. 5 - 8), поскольку именно такой режим поддерживается исследуемым в лабораторной работе датчиком ADXL345. На рис. 5 и далее будут использоваться альтернативные обозначения линий SPI относительно микросхемы *Slave*.

На время передачи данных в соответствии с протоколом SPI необходимо перевести линию CS в низкий логический

уровень. Минимальный интервал времени t_{DELAY} от заднего фронта CS до первого фронта на линии SCLK приводится в техническом описании (*Datasheet*) конкретной микросхемы *Slave* и, как правило, составляет 5...10 нс. Далее на линии *Slave* SDI(MOSI) *Master* должен сначала выставить служебный байт, после чего записать в *Slave* либо считать из него один или несколько байт данных. Служебный байт для ADXL345 состоит из бита чтения или записи R/W (**Read/Write**: '1' – чтение, '0' – запись), бита множественной или одиночной передачи данных M/S (**Multiple/Single**: '1' – передача нескольких байт подряд, '0' – передача одного байта) и шести бит адреса регистра *Slave*, с которым будет происходить обмен данными. По окончании передачи в линии CS устанавливается высокий логический уровень. Стоит отметить, что если в линии CS постоянно действует логический '0' (линия CS соединена с землей), то передача данных невозможна.

Временные диаграммы сигналов в линиях шины SPI при записи одного байта данных $D = D_7D_6D_5D_4D_3D_2D_1D_0$ в регистр *Slave* с адресом $A = A_5A_4A_3A_2A_1A_0$ приведены на рис. 5. При чтении одного байта данных *Slave* после приема служебного байта по линии SDI выставляет на линии SDO байт данных D, хранящийся в регистре с адресом A. Уровень сигнала в линии SDI в этот момент может быть как высоким, так и низким (рис. 6).

Если необходимо записать данные в несколько соседних регистров подряд (с адресами A, A+1, A+2, ...), то нужно в служебном байте установить бит M/S = '1' и указать адрес первого регистра $A = A_5A_4A_3A_2A_1A_0$. В этом случае после записи каждого байта данных адрес регистра будет автоматически инкрементироваться (увеличиваться на 1). Временные диаграммы, соответствующие записи двух байт в соседние регистры с адресами A и (A+1), приведены на рис. 7. Аналогичным образом может быть организовано чтение нескольких байт данных из соседних регистров (рис. 8).

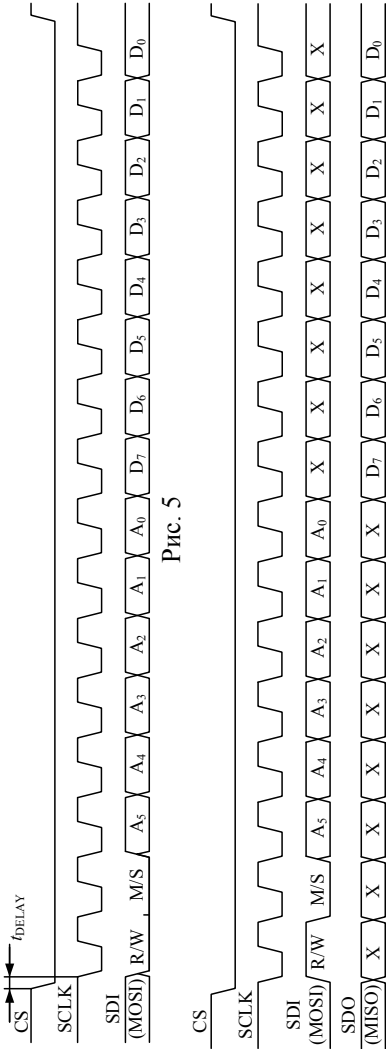


Рис. 5

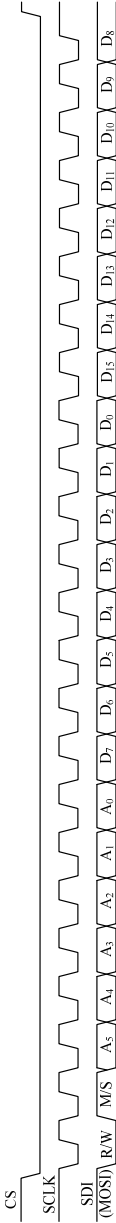


Рис. 6

Рис. 7

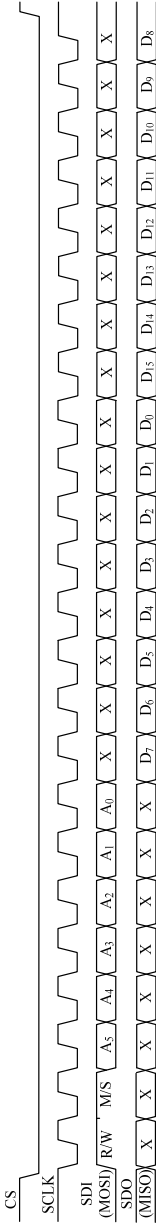


Рис. 8

Некоторые микросхемы поддерживают трехпроводной протокол SPI, в котором вместо двух разнонаправленных линий SDI и SDO используется одна двунаправленная линия SDIO. Временные диаграммы при записи данных в *Slave* в этом случае аналогичны четырехпроводному SPI, а при чтении данные D после приема служебного байта выставляются на той же линии SDIO. Для реализации двунаправленной линии средствами VHDL порт SDIO должен быть объявлен как **inout** (Z-состояние по умолчанию), а линия SDIO подключена к шине питания через подтягивающий резистор. Передача данных при этом возможна только выдачей низкого логического уровня в линию SDIO (замыканием ее на землю), а высокий логический уровень формируется благодаря подтягивающему резистору.

Реализация информационного обмена с любой микросхемой подразумевает внимательное изучение ее технического описания, которое часто приводится только на английском языке. Основные сведения о микросхеме, как правило, приводятся в аннотации (*Features*) технического описания [3, P. 1]: для исследуемого трехосного датчика ускорения (акселерометра) ADXL345 это диапазоны измерения ускорения (*Full resolution range*), напряжение питания (*Supply voltage*), цена младшего разряда (*LSB scale factor*, где аббревиатура LSB означает младший значащий бит, *Least Significant Bit*), поддерживаемые протоколы передачи данных (*Interfaces*). Далее важно найти в техническом описании электрическую схему подключения для выбранного протокола передачи данных [3, P. 15].

Карта всех регистров и описание назначения каждого из них приводятся в разделах *Register Map* [3, P. 23-27] и *Applications Information* [3, P. 28-36] соответственно.

Регистры управления микросхем часто имеют обозначение CONTROL, CTRL или CTL. Для датчика ADXL345 таким регистром является регистр POWER_CTL [3, P. 25] с адресом (*Address*) 2D в шестнадцатеричной (*Hex*) системе счисления (в языках программирования высокого уровня принято обозначение с префиксом «0x» – 0x2D) или 45 – в десятичной (*Dec*). Тип регистра (*Type*) R/W означает, что для него разрешены как чтение, так и запись данных; значение по умолчанию (после подачи на устройства питания) «00000000» = 0x00 приводится в колонке *Reset Value*. Описание назначения каждого бита данного регистра приводится в подразделе *Register 0x2D – POWER_CTL* [3, P. 25-26]. Бит *Link* позволяет произвести разделение функций датчика в активном и неактивном режимах: при значении '1' такое разделение возможно, при '0' – нет (в лабораторной работе выберем последний вариант). Высокий

логический уровень бита *AUTO_SLEEP* разрешает устройству автоматически переходить в режим пониженного энергопотребления; также установим этот бит в '0'. Бит *Measure* определяет режим работы устройства: '1' – режим измерения, '0' – режим ожидания. Значение '1' в бите *Sleep* устанавливает пониженную частоту дискретизации (темп измерений): значение 1, 2, 4 или 8 Гц определяется двумя битами *Wakeup* [3, Р. 26]. Используемый в лабораторной работе управляющий байт для регистра *POWER_CTL* – 0x08.

Регистр *BW_RATE* содержит бит *LOW_POWER* ('1' – режим низкого энергопотребления с большим уровнем шумов, '0' – нормальный режим) и 4 бита *Rate* [3, Р. 25], которые определяют частоту дискретизации датчика.

Установка '1' в биты регистра *INT_ENABLE* разрешает различные виды прерываний (см. регистр *INT_SOURCE*), а '0' – запрещает.

Регистр *DATA_FORMAT* определяет формат представления данных с датчика в регистрах с адресами 0x32-0x37. Высокий логический уровень бита *SELF_TEST* (самотестирование) изменяет реальные данные с датчика: такой режим может потребоваться при отладке или проверке достоверности показаний. Бит *SPI* устанавливает 4-проводной ('0') или 3-проводной ('1') интерфейс SPI. Установка '1' в бит *INT_INVERT* инвертирует уровни прерываний в регистре прерываний *INT_SOURCE*. Бит *FULL_RES* определяет изменение разрешения акселерометра: при '0' разрядность акселерометра фиксирована – независимо от диапазона измерений отсчет выходного сигнала по каждой оси представляется 10-ю битами (т.е. при расширении диапазона измерений увеличивается цена младшего разряда LSB); при '1' число уровней квантования увеличивается с ростом динамического диапазона (цена LSB не меняется). Бит *Justify* задает выравнивание дополнительного кода отсчета ускорения (10 - 13 бит в зависимости от диапазона измерений) по правому ('0') или левому ('1') краю в паре байт выходного сигнала для каждой оси. В первом случае знак ускорения дублируется на свободные старшие разряды, во втором случае младшие разряды дополняются не значащими нулями (см. пример для 10-разрядного кода "10 10001011" в табл. 3). Биты *Range* задают диапазон измерений акселерометра.

Влияние бита *Justify* на формат данных акселерометра Таблица 3

| Бит Justify | 16 бит в паре выходных регистров Datax0 (младший байт) и Datax1 (старший байт), $x = \{X, Y, Z\}$ | | | | | | | | | | | | | | | |
|----------------|---|---|---|---|---|---|---|---|--------|---|---|---|---|---|---|---|
| | Datax1 | | | | | | | | Datax0 | | | | | | | |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Перед чтением данных из регистров с адресами 0x32-0x37 необходимо проверить, записаны ли в них новые данные от встроенного в акселерометр АЦП, прочитав бит готовности данных *DATA_READY* регистра прерываний *INT_SOURCE*.

С учетом технического описания ADXL345 для чтения данных можно использовать следующий алгоритм работы цифрового конечного автомата (ЦКА) на ПЛИС:

1) инициализация регистров управления *POWER_CTL*, *BW_RATE*, *INT_ENABLE* и *DATA_FORMAT*, т.е. запись в них управляющих слов для обеспечения требуемого режима работы;

2) чтение регистра прерываний *INT_SOURCE*: если бит *DATA_READY* = '1', выполнить п. 3, в противном случае повторять п. 2;

3) чтение данных об ускорении вдоль осей *X*, *Y* и *Z* из регистров с адресами 0x32-0x37 в буферный регистр ПЛИС;

4) выдача данных по завершении чтения;

5) ожидание следующего периода чтения данных с датчика (величина периода определяется требуемой частотой дискретизации) и переход к п. 2.

Граф состояний ЦКА, реализующего SPI-Master для датчика ADXL345 в соответствии с приведенным выше алгоритмом, показан на рис. 9. ЦКА содержит 5 состояний: *sleep* – ожидание, *init* – инициализация, *int_status* – чтение состояния регистра прерываний, *read_data* – чтение данных в буферный регистр ПЛИС, *d_ready* – анализ готовности данных.

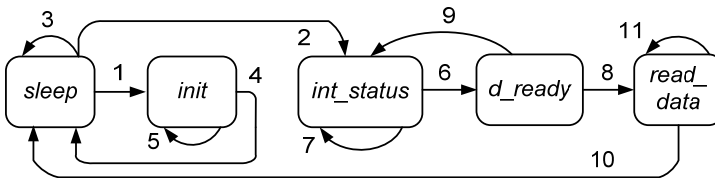


Рис. 9

На рис. 9 на направлениях смены состояний ЦКА обозначены следующие условия:

- 1 – инициализация не завершена;
- 2 – инициализация завершена, наступил новый цикл чтения данных;
- 3 – не выполнены условия 1 и 2;
- 4 – инициализирующий байт записан в регистр управления;
- 5 – условие 4 не выполнено;
- 6 – прочитано содержимое регистра прерываний;
- 7 – условие 6 не выполнено;

- 8 – данные готовы;
- 9 – данные не готовы;
- 10 – прочитаны все байты данных;
- 11 – условие 10 не выполнено.

Домашнее задание

1. Изучить основы передачи данных по протоколу SPI.
2. Ознакомиться с техническим описанием интегральной микросхемы ADXL345 [3]: схема включения; порядок обмена информацией по протоколу SPI; адреса регистров управления и значения управляющих байт для них; принцип декодирования кода ускорения.
3. По *Datasheet* определить значения управляющих байт при следующих требованиях к акселерометру: байт для регистра POWER_CTL – 0x08 = 8 (нормальное энергопотребление), частота дискретизации – 100 Гц, диапазон измерений – $\pm 2g$, разрешено прерывание по готовности данных, выравнивание данных – по левому краю для нечетного варианта и по правому – для четного; фиксированная разрядность – 10 бит, количество линий шины SPI – 4.
4. По графу состояний рис. 9 составить программы на языке описания аппаратуры VHDL для чтения данных с акселерометра ADXL345 в соответствии с требованиями пп. 1 - 3 порядка выполнения работы.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Моделирование обмена данными по протоколу SPI на примере трехосного акселерометра ADXL345

- 1.1. Создать новый проект с именем *spi_master*.
- 1.2. Составить описание пакета, в котором объявить:
 - 1) перечислимый тип *spi_states* с состояниями ЦКА SPI-Master;
 - 2) подтип *byte* – как тип **natural** в диапазоне 0-255;
 - 3) константы подтипа *byte*:
 BW_RATE, POWER_CTL, INT_ENABLE и DATA_FORMAT – для адресов регистров управления и INT_SOURCE – для адреса регистра прерываний;
 WORD_BW_RATE, ..., WORD_DATA_FORMAT – для управляющих слов длиной 1 байт, которые будут записаны в соответствующие регистры при инициализации;
- ALL_INIT_REGS и ALL_READ_REGS – для общего количества инициализируемых регистров и регистров с данными от датчика;
- DATA0 – для адреса первого регистра с данными;
- 4) константу DISCRETE типа **natural**, определяющую период дискретизации, выраженный в периодах частоты тактового сигнала *clk*.

1.3. Составить описание сущности *SPI-Master* с портами:

1) clock: **in std_logic**; -- тактовый сигнал кварцевого генератора;

2) cs: **out std_logic**; -- линия CS;

3) sclk: **out std_logic**; -- линия SCLK;

4) sdi: **out std_logic**; -- линия SDI Slave;

5) sdo: **in std_logic**; -- линия SDO Slave;

6) выходной вектор объемом 2 байта для вывода кода ускорения по одной из осей, указанной преподавателем.

1.4. В декларативной части архитектуры объявить:

тип *adxl_regs* как массив байт длиной ALL_INIT_REGS (нумерация элементов массива возрастающая, начиная с нуля);

тип *data_regs* как массив из ALL_READ_REGS элементов типа **std_logic_vector(7 downto 0)** с аналогичной нумерацией элементов;

константы типа *adxl_regs*:

INIT_ADDRESSES = (BW_RATE, ... , DATA_FORMAT) и

CTRL_DATA = (WORD_BW_RATE, ..., WORD_DATA_FORMAT).

Объявить сигналы типа **std_logic**:

new_data – признак готовности новых данных в датчике ('1' – данные готовы);

init_ready – признак окончания инициализации ('1' – инициализация завершена);

clk – внутренний тактовый сигнал (выход сигнала делителя);

sclk_ena – признак разрешения выдачи clk в линию sclk ('1' – выдача разрешена);

wd – сигнал от внутреннего таймера о начале нового цикла чтения данных ('1' – начало нового цикла);

cnt – сигнал разрешения счета бит ('1' – счет разрешен).

-- сигналы состояний ЦКА;

signal state, next_state: spi_states;

-- счетчик бит;

signal n_bit: **natural range 0 to 8*(ALL_READ_REGS + 1) - 1**;

-- число проинициализированных регистров;

signal num_init: **natural range 0 to ALL_INIT_REGS**;

-- число прочитанных регистров;

signal num_read: **natural range 0 to ALL_READ_REGS**;

1.5. Составить описание проекта делителя частоты с настроечной константой. Выходной сигнал делителя должен представлять меандр с высоким логическим уровнем в первом полупериоде. Подключить к проекту компонент делителя.

1.6. Для синтеза процессов, реализующих *SPI-Master*, можно воспользоваться шаблоном, приведенным далее:

```

-- формирование сигнала в линии SCLK
sclk <= <лог. уровень> when sclk_ena = '0' else clk;
-- процесс определения следующего состояния ЦКА
process (state, wd, init_ready, n_bit, new_data, num_read)
begin
case (state) is
  when sleep => if (init_ready = '0') then next_state <= <состояние>;
    elsif (wd = '1') then next_state <= <состояние>;
    else next_state <= <состояние>;
    end if;
  when init => if (n_bit = 15)
    then next_state <= <состояние>;
    else next_state <= <состояние>;
    end if;
  when int_status => if (n_bit = 15)
    then next_state <= <состояние>;
    else next_state <= <состояние>;
    end if;
  when d_ready =>
    if (new_data = '0') then next_state <= <состояние>;
    else next_state <= <состояние>;
    end if;
  when read_data => if (n_bit = (8*(ALL_READ_REGS + 1) - 1))
    then next_state <= <состояние>;
    else next_state <= <состояние>;
    end if;
  when others => null;
end case;
end process;
-- описание счетчика бит по заднему фронту clk с асинхронным
-- сбросом по низкому логическому уровню сигнала cnt
process (<список чувствительности>)
  <операторы для реализации счетчика>;
-- процесс формирования сигнала разрешения счета бит
process (<список чувствительности>)
begin
if (clk'event and clk = '0') then
  case (state) is
    when sleep => cnt <= '0';
    when init => cnt <= '1';
    when int_status => cnt <= <лог. уровень>;

```

```

    when read_data => cnt <= <лог. уровень>;
    when d_ready => cnt <= '0';
    when others => null;
end case;
end if;
end process;
-- внутренний таймер и смена состояний ЦКА
process (<список чувствительности>)
variable cycle: natural range 0 to DISCRETE;
begin
if (clk'event and clk = '1') then
    state <= next_state;
    if cycle = 0 then wd <= '1';
        else wd <= '0';
    end if;
    if cycle = DISCRETE - 1 then cycle := 0;
        else cycle := cycle + 1;
    end if;
end if;
end process;
-- процесс формирования сигналов в линиях SDI и CS
process (clock, state, n_bit, num_init, num_read)
-- регистры для временного хранения данных
variable buf_8: std_logic_vector(7 downto 0);
variable buf_16: std_logic_vector(15 downto 0);
begin
if (clock'event and clock = '1') then
    case (state) is
        when sleep => sdi <= '0'; cs <= '1'; sclk_ena <= '0';
        when init => cs <= '0'; sclk_ena <= '1';
            buf_16 := <биты R/W и M/S> &
                conv_std_logic_vector(INIT_ADDRESSES(num_init), 6) &
                conv_std_logic_vector(CTRL_DATA(num_init), 8);
            sdi <= buf_16(15 - n_bit);
        when int_status =>
            cs <= <лог. уровень>; sclk_ena <= <лог. уровень>;
            buf_8 := <биты R/W и M/S> &
                conv_std_logic_vector(<адрес>, 6);
            sdi <= buf_8(7 - n_bit);
        when read_data =>
            cs <= <лог. уровень>; sclk_ena <= <лог. уровень>;

```

```

        buf_8 := <биты R/W и M/S> &
        conv_std_logic_vector(<адрес>, 6);
        sdi <= buf_8(7 - n_bit);
    when d_ready => sdi <= '0';
        cs <= <лог. уровень>; sclk_ena <= <лог. уровень>;
    when others => null;
end case;
end if;
end process;
-- процесс, выполняющий чтение данных с линии SDO Slave
-- и формирование управляющих сигналов
process (<список чувствительности>)
variable data: data_regs;
begin
if (clk'event and clk = '1') then
    case (state) is
        when sleep => num_read <= 0;  new_data <= '0';
            <выход> <= <старший байт> & <младший байт>;
        when init => if (n_bit = 15) then
            if (num_init = ALL_INIT_REGS - 1)
                then init_ready <= <лог. уровень>;
            else init_ready <= <лог. уровень>;
                num_init <= num_init + 1;
            end if;
        end if;
        when int_status => if (n_bit = <номер бита>)
            then new_data <= sdo;
        end if;
        when read_data => if (n_bit > <номер бита>)
            then data(num_read)(7 - (n_bit rem 8)) := sdo;
            if (n_bit rem 8 = 7)
                then num_read <= num_read + 1;
            end if;
        end if;
        when others => null;
    end case;
end if;
end process;

```

1.7. Задать настроечную константу делителя равной 8, константу DISCRETE = 200. Откомпилировать проект; при наличии в тексте программы ошибок внести исправления.

1.8. Выполнить моделирование работы проекта. Для упрощения отладки добавить в *.vwf-файл, помимо входных и выходных сигналов сущности, текущее состояние ЦКА, служебные сигналы и переменную *cycle*. Задать частоту сигнала с кварца 25 МГц, время моделирования – 100 мкс. По умолчанию установить на линии SDO логический '0'. Запустить моделирование. Убедиться в правильности записи заданных байт в регистры управления. Выполнить перезапись *.vwf-файла результатами моделирования (П.1). Сымитировать во втором по порядку состоянии ЦКА *int_status* ответ от датчика о готовности данных. Выполнить моделирование. Сымитировать в состоянии *read_data* ненулевой ответ от *Slave* и снова выполнить моделирование. Проанализировать работу синтезированного ЦКА.

2. Проверка работы проекта с датчиком ADXL345

2.1. Установить настроечную константу делителя, чтобы обеспечить скорость обмена по шине SPI 1 Мбит/с; вернуть исходное значение константы DISCRETE.

2.2. Подключить входные и выходные сигналы проекта к выводам ПЛИС отладочного комплекта. Линиям шины SPI назначить номера выводов ПЛИС в соответствии со схемой подключения датчика ADXL345 к контактам разъема JP 1 (GPIO 0) или JP2 (GPIO 1). Выходной сигнал подключить к красным и зеленым светодиодам LEDR7 – LEDR0 и LEDG7 – LEDG0. Откомпилировать проект заново.

2.3. **Не подключая плату с ПЛИС к ПЭВМ**, соединить контакты разъема датчика с контактами разъема JP 1 (GPIO 0): VCC – с контактом 3,3 В, GND – с контактом GND. Показать преподавателю результаты подключения выводов датчика. **Предельное напряжение обратной полярности для ADXL345 составляет всего -0,3 В, поэтому при неправильной коммутации выводов VCC и GND датчик будет поврежден.**

2.4. Включить отладочный макет. Загрузить проект в ПЛИС.

2.5. Наблюдать на светодиодах изменение сигнала ускорения. Повернув датчик таким образом, чтобы нормированный модуль проекции вектора ускорения свободного падения на выбранную ось чувствительности составил ± 1 (можно воспользоваться рисунками *Figure 57* и *58 Datasheet* [3]), объяснить влияние выбранного бита *Justify* на форму представления выходных данных.

3. Индикация ускорения на семисегментных индикаторах

3.1. Добавить в программу компоненты трехвходового мультиплексора для переключения между данными с каждой из осей чувствительности датчика, семисегментного индикатора (ССИ) и блок декодера ускорения (код ускорения ADXL в соответствии с *Datasheet*

перевести в доли g и выдать значение в 100 раз большее).

3.2. Изменить темп чтения данных с акселерометра на 2 или 5 Гц.

3.3. Загрузить скомпилированный проект в ПЛИС. Убедиться в правильности его работы (индикации ускорений со знаком) для всех осей чувствительности.

3.4. По окончании работы выключить отладочный макет.

Лабораторная работа № 2

Изучение протокола передачи данных I2C

Цель работы: синтез цифрового автомата для реализации информационного обмена с микросхемами, поддерживающими интерфейс I2C (читается как «ай-ту-си»), на примере цифрового трехосного акселерометра ADXL345 фирмы Analog Devices.

Интерфейс I2C (от англ. *Inter-Integrated Circuit* – схема внутренней связи) разработан компанией Philips. I2C – последовательный полудуплексный протокол передачи данных. Для обмена используются две линии связи: линия данных SDA (от англ. *Serial Data* – последовательные данные) и линия тактового сигнала SCL (от англ. *Serial Clock* – тактовый сигнал). Шина I2C, как и SPI, организована по принципу *Master-Slave*. Протокол I2C поддерживает обмен информацией на расстоянии до 30...50 см. К одной двухпроводной шине одновременно может быть подключено до 127 устройств, различающихся 7-битным адресом (для микросхем с 10-битным адресом максимальное количество равно 1023).

Подключение к шине I2C (рис. 10) можно рассматривать как соединение выводов микросхем с открытым коллектором (высокоимпедансное Z-состояние) по схеме «монтажное И»; поскольку все устройства *Slave* включены параллельно, то выдача логического нуля хотя бы одной микросхемой на линии SDA или SCL приводит к появлению низкого логического уровня во всей линии.

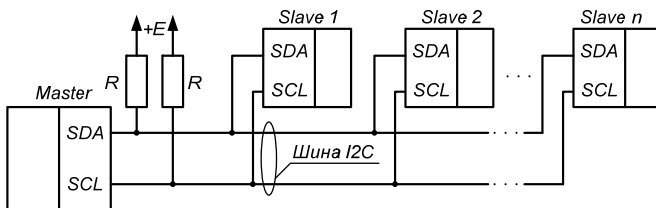


Рис. 10

Передача/прием данных по шине I2C осуществляется переводом линии SDA в низкий логический уровень. Тактовый сигнал SCL имеет право генерировать только *Master*. Максимальная частота тактовых

импульсов составляет 100 или 400 кГц. При передаче логической '1' запрещено генерировать сигнал сильной '1', поскольку, если одно из устройств *Slave* в данный момент установит на линии логический '0', это приведет к короткому замыканию (см. рис. 10) и повреждению *Master* или *Slave*. Поэтому **высокий логический уровень в линиях шины I2C следует задавать только их переводом в Z-состояние**, при этом сигнал логической '1' формируется с помощью подтягивающих резисторов R.

Начало передачи определяется сигналом *Start* (S) – переходом линии SDA из '1' в '0' при высоком логическом уровне сигнала в линии SCL (рис. 11, а). Окончание передачи определяется сигналом *Stop* (P) – переходом сигнала SDA из '0' в '1' при высоком логическом уровне SCL (рис. 11, в). При передаче информации от *Slave* *Master* генерирует тактовый сигнал в линии SCL и считывает биты с линии SDA (рис. 11, б). *Slave* выставляет новый бит при SCL = '0', *Master* считывает данный бит при SCL = '1', т.е. считывание производится не по фронту, а по уровню тактового сигнала. Этим объясняются меньшие помехоустойчивость и скорость передачи данных шины I2C по сравнению с шиной SPI. При передаче информации от *Master* к *Slave* *Master* генерирует тактовый сигнал в линии SCL и выдает биты в линию SDA. *Slave* считывает данные по высокому логическому уровню в линии SCL. Длительности '0' и '1' в периоде тактового сигнала SCL рекомендуется устанавливать в соотношении 70/30 или 60/40.

Таким образом, изменение данных на линии SDA может выполняться только при низком логическом уровне в линии SCL; если SCL = '1', то происходит их чтение. Если изменения на линии SDA происходят при SCL = '1', то это соответствует служебной команде *Start* или *Stop*.

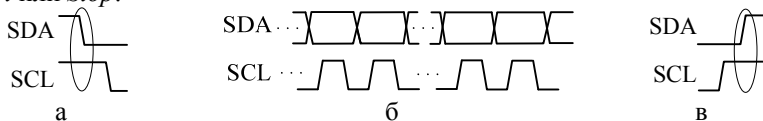


Рис. 11

Данные по протоколу I2C передаются последовательными байтами, бит за битом, начиная со старшего бита [4, С. 5-34]. Первый байт всегда направляется от *Master* к *Slave* и представляет собой физический адрес устройства SAD (от англ. *Slave address*) длиной 7 бит + 8-й бит направления передачи R/W (*Read/Write*): ('0' – запись в устройство, '1' – чтение из устройства). Бит R/W определяет, что будет делать *Slave* при передаче следующего байта: передавать или принимать данные.

Если байт данных был принят *Slave* корректно, то устройство вырабатывает бит подтверждения ACK (от англ. *Acknowledge* – подтверждение, ответ). В некоторых *Datasheet* бит ACK может быть обозначен как SAK (от англ. *Slave Acknowledge*). Если адрес SAD, переданный *Master*, совпал с адресом *Slave*, то вырабатывается ACK = '0' и передача данных продолжается. Если ACK = '1', это означает, что *Slave* с данным адресом на линии не обнаружен, либо неправильно принял байт со своим адресом, либо неработоспособен. При записи данных в *Slave* следующие за SAD байты могут быть байтами с адресами регистров *Slave* (обозначаются как SUB от англ. *Sub-address*) или байтами данных DATA. После корректного приема каждого байта *Slave* также отвечает байтом подтверждения ACK = '0'. При чтении данных из *Slave* сигнал подтверждения о приеме байта (бит '0') должен вырабатывать *Master*. Этот сигнал обозначают как MAK (*Master acknowledge*). После приема последнего байта перед выработкой сигнала *Stop Master* должен сообщить *Slave*, что больше не будет принимать от него данные. Для этого он вырабатывает на линии SDA сигнал высокого логического уровня NACK = '1' (от англ. *No acknowledge* – нет ответа).

Если необходимо записать в *Slave* несколько байт в соседние регистры, то выполняется передача адреса SUB первого регистра, а затем подряд n байт данных. Адрес SUB при записи каждого нового байта данных будет автоматически инкрементироваться. Аналогичное справедливо и для чтения нескольких байт из *Slave*.

Например, для записи двух байт в соседние регистры необходимо:

- 1) сформировать сигнал S;
- 2) отправить 7 бит адреса *Slave* SAD и бит записи R/W = '0';
- 3) получить бит подтверждения ACK;
- 4) отправить байт адреса регистра *Slave* SUB, в который требуется записать первый байт данных;
- 5) получить бит подтверждения ACK;
- 6) отправить байт данных DATA для записи в регистр с адресом SUB;
- 7) получить бит подтверждения ACK;
- 8) отправить второй байт данных для записи в регистр с адресом SUB + 1;
- 9) получить бит подтверждения ACK;
- 10) после приема последнего ACK сформировать сигнал P.

Для чтения двух байт из соседних регистров *Slave* необходимо:

- 1) сформировать сигнал S;
- 2) отправить 7 бит адреса *Slave* SAD и бит записи R/W = '0';
- 3) получить бит подтверждения ACK;

- 4) отправить байт адреса регистра *Slave* SUB, из которого требуется считать первый байт данных;
- 5) получить бит подтверждения ACK;
- 6) сформировать сигнал повторного старта **RS**;
- 7) отправить 7 бит адреса *Slave* SAD и бит чтения R/W = '1';
- 8) получить бит подтверждения ACK;
- 9) получить байт данных DATA из регистра с адресом SUB;
- 10) сформировать бит подтверждения MAK;
- 11) получить байт данных DATA из регистра с адресом SUB + 1;
- 12) после приема последнего байта данных сформировать бит NACK;
- 13) сформировать сигнал **P**.

Служебный сигнал **RS** (от англ. *Repeated Start* – повторный старт) аналогичен сигналу **S** и разделяет циклы записи и чтения при чтении данных из *Slave*.

В табл. 4. и 5 указан порядок выработки сигналов *Master* и *Slave*, а на рис. 13 и 14 приведены соответственно временные диаграммы, демонстрирующие запись и чтение двух байт подряд, начиная с регистра с адресом SUB.

В табл. 6 приведен пример с порядком установки бит на линии SDA для записи двух байт (0xF0 и 0x77) в регистры с адресами 0x0A и 0x0B, а в табл. 7 – для чтения двух байт (0x03 и 0x0D) из регистров с адресами 0x0F и 0x10. Адрес *Slave* принят равным 0x63.

Устройства *Slave* с низкой скоростью записи данных (например, электрически репрограммируемые постоянные запоминающие устройства EEPROM) требуют некоторого времени для записи байта данных, поэтому после ответа ACK *Slave* может удерживать на линии SCL низкий логический уровень (показан на рис. 12 утолщенной линией), не позволяя *Master* генерировать новые тактовые импульсы (показаны пунктиром): по принципу монтажного И на линии SCL в этом случае будет действовать логический '0'. Поэтому при работе с такими *Slave* после записи байта данных *Master* должен анализировать состояние линии SCL ('0' – занята, '1' – свободна) и при необходимости задерживать передачу следующего байта.

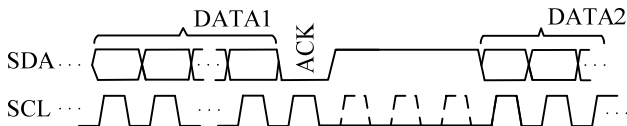


Рис. 12

Сигналы в линии SDA при записи двух байт

Таблица 4

| | | | | | | | | | | |
|--------|---|-------|-----|-----|-----|------|-----|------|-----|---|
| Master | S | SAD+W | | SUB | | DATA | | DATA | | P |
| Slave | | | ACK | | ACK | | ACK | | ACK | |

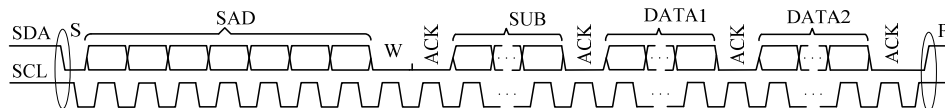


Рис. 13

Сигналы в линии SDA при чтении двух байт

Таблица 5

| | | | | | | | | | | | | | |
|--------|---|-------|-----|-----|-----|----|-------|-----|------|-----|------|------|---|
| Master | S | SAD+W | | SUB | | RS | SAD+R | | DATA | MAK | DATA | NACK | P |
| Slave | | | ACK | | ACK | | | ACK | | | | | |

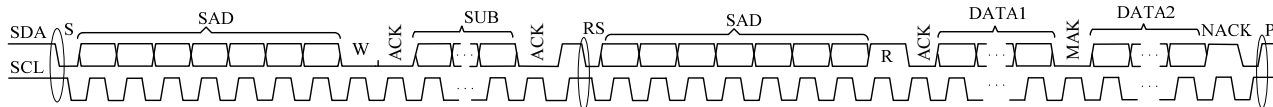


Рис. 14

Логические уровни в линии SDA при записи двух байт в Slave с адресом 0x63

Таблица 6

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|-----|---|---|---|---|---|---|---|---|--|-----|---|---|---|---|---|---|---|--|-----|---|---|---|---|---|---|---|-----|---|--|
| Master | S | 1 | 1 | 0 | 0 | 0 | 1 | 1 | W | | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | | P | |
| Slave | | | | | | | | | | Ack | | | | | | | | | | Ack | | | | | | | | | Ack | | | | | | | | Ack | | |

Логические уровни в линии SDA при чтении двух байт из Slave

Таблица 7

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|-----|---|---|---|---|---|---|---|---|-----|----|---|---|---|---|---|---|---|---|-----|--|--|--|--|--|--|--|--|
| Master | S | 1 | 1 | 0 | 0 | 0 | 1 | 1 | W | | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | | RS | 1 | 1 | 0 | 0 | 0 | 1 | 1 | R | | | | | | | | | |
| Slave | | | | | | | | | | Ack | | | | | | | | | Ack | | | | | | | | | | Ack | | | | | | | | |

Продолжение таблицы 7

| | | | | | | | | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|-----|---|---|---|---|---|---|---|---|--|------|---|
| Master | | | | | | | | | Mak | | | | | | | | | | Nack | P |
| Slave | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | | | |

В публицистическом стиле принцип работы протокола I2C подробно описан в интернет-источниках [5] и [6].

Логика работы цифрового автомата для реализации I2C *Master* для чтения данных с датчика ADXL345 схожа с принципом работы ЦКА SPI *Master*, рассмотренного в лабораторной работе № 1: сначала необходимо записать управляющие слова в контрольные регистры, а затем, анализируя бит готовности данных в регистре INT_SOURCE, с заданным интервалом дискретизации считывать данные. Особенностью ЦКА I2C *Master* является наличие дополнительных состояний: обработки бита ACK и генерации/анализа сигналов S, P, RS, ACK и NACK (рис. 15).

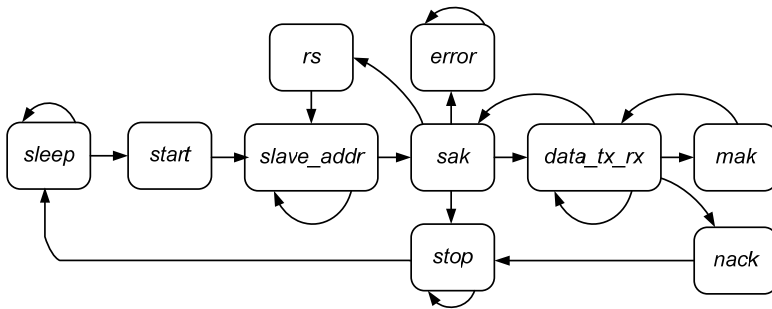


Рис. 15

Поскольку чтение содержимого нескольких соседних регистров быстрее, чем поочередное чтение одного байта из каждого регистра, в работе считываются 8 байт данных из регистров от INT_SOURCE до DATA1 (т.е. также считываются и данные из регистра DATA_FORMAT).

Домашнее задание

1. Изучить основы передачи данных по протоколу I2C.
2. Определить по *Datasheet* значения управляющих байт при следующих требованиях к акселерометру: байт для регистра POWER_CTL – 0x08 = 8 (нормальное энергопотребление), частота дискретизации – 100 Гц, диапазон измерений – $\pm 2g$, разрешено прерывание по готовности данных, выравнивание данных – по левому краю для нечетного варианта и по правому – для четного; фиксированная разрядность 10 бит, разрешен интерфейс I2C.
3. По графу состояний рис. 15 и в соответствии с логикой работы цифрового автомата из лабораторной работы № 1 составить программы на языке описания аппаратуры VHDL для чтения и обработки данных с датчика акселерометра ADXL345 в соответствии с требованиями пп. 1 и 2 порядка выполнения работы.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Моделирование обмена данными по протоколу I2C на примере трехосного акселерометра ADXL345

1.1. Создать новый проект с именем *i2c_master*.

1.2. Повторить п. 1.2 лабораторной работы № 1 со следующими изменениями.

Вместо типа *spi_states* объявить тип *i2c_states* с состояниями:

sleep – ожидание;

start – генерация сигнала S;

slave_addr – передача адреса *Slave*;

data_tx_rx – прием/передача байт данных;

stop – генерация сигнала P;

sak – анализ ответа ACK от *Slave*;

error – состояние ошибки: отсутствие ответа ACK от *Slave*;

mak – генерация сигнала MAK *Master*;

nack – генерация сигнала NACK *Master*;

rs – генерация сигнала повторного старта.

В пакете также объявить константу адреса *Slave* SLAVE_ADDRESS (без младшего бита SAD0) типа *byte*.

1.3. Составить описание сущности I2C-Master с портами:

1) *clock*: **in std_logic**; -- тактовый сигнал кварцевого генератора;

2) *cs*: **out std_logic**; -- сигнал выбора режима I2C для ADXL345;

3) *scl*: **inout std_logic**; -- линия SCL;

4) *sda*: **inout std_logic**; -- линия SDA;

5) *sad0*: **out std_logic**; -- младший бит адреса *Slave*;

6) *err*: **out std_logic**; -- признак состояния *error*;

7) выходной вектор объемом 2 байта для вывода кода ускорения по одной из осей, указанной преподавателем.

1.4. В декларативной части архитектуры объявить:

тип *adxl_regs* как массив байт длиной (ALL_INIT_REGS+1);

тип *data_regs* как массив байт длиной (ALL_READ_REGS+1);

константы типа *adxl_regs*:

INIT_ADDRESSES=(BW_RATE, ..., DATA_FORMAT, INT_SOURCE) и

CTRL_DATA=(WORD_BW_RATE, ..., WORD_DATA_FORMAT, 0).

Объявить сигналы типа **std_logic**:

new_data – признак готовности новых данных в датчике ('1' – данные готовы);

init – признак инициализации ('1' – инициализация);

clk – внутренний тактовый сигнал (1-й выход делителя);

clk_for_scl – сигнал для выдачи в линию SCL (2-й выход делителя);

gen_scl – признак разрешения выдачи *clk_for_scl* в линию SCL

('1' – выдача разрешена);

free – признак свободной линии SCL ('1' – линия свободна);

rw – чтение/запись ('1'/'0');

ack – признак ответа ACK;

wd – сигнал от внутреннего таймера о начале нового цикла чтения данных ('1' – начало нового цикла);

cnt – сигнал разрешения счета бит ('1' – счет разрешен);

RStart – сигнал для генерации повторного старта RS;

data_address – признак передачи данных ('1') или адреса ('0').

Также в архитектуре необходимо объявить:

сигнал *type_tx* перечислимого типа с элементами tx_adr, tx_data, tx_stop и tx_RS для определения действий *Master* после приема ответа ACK (соответственно передача адреса *Slave*, байта SUB или DATA, сигнала P и сигнала RS);

сигналы *num_init* и *num_read* типа *byte* для счета проинициализированных и прочитанных регистров *Slave* соответственно;

сигнал для счета бит *n_bit* типа **natural range 0 to 7**;

сигналы *state* и *next_state* для хранения состояний I2C *Master*.

1.5. Составить описание проекта делителя частоты с настроечной константой и двумя выходами. Сигнал с первого выхода делителя объявить как **out std_logic** и сформировать на нем меандр с низким логическим уровнем в первом полупериоде; сигнал со второго выхода объявить как **inout std_logic** и сформировать на нем меандр с 'Z' состоянием во второй и третьей четвертях периода. Подключить компонент делителя к проекту.

1.6. Присвоить требуемые логические уровни выходам sad0 и cs.

1.7. Для синтеза процессов, реализующих I2C-*Master*, можно воспользоваться шаблоном, приведенным ниже.

-- формирование сигнала в линии SCL

scl <= clk_for_scl when gen_scl = '1' else 'Z';

-- процесс смены состояний ЦКА

process (clk, wd)

begin

if clk'event and clk = '0' **then**

if wd = '1' **then**

state <= start;

else state <= next_state;

end if;

end if;

end process;


```

-- процесс определения следующего состояния ЦКА
process(state, ack, n_bit, num_read, num_init, new_data, rw, free, type_tx)
begin
case state is
  when sleep => next_state <= sleep;
  when start => next_state <= <состояние>;
  when slave_addr => if (n_bit = 7) then next_state <= <состояние>;
    else next_state <= <состояние>;
    end if;

  when sak => if ack = '0' then
    case (type_tx) is
      when tx_adr => next_state <= data_tx_rx;
      when tx_data => next_state <= data_tx_rx;
      when tx_stop => next_state <= stop;
      when tx_RS => next_state <= RS;
      when others => null;
    end case;
    else next_state <= error;
    end if;

  when mak => next_state <= <состояние>;
  when RS => next_state <= <состояние>;
  when nack => next_state <= <состояние>;
  when stop => if free = '0' then next_state <= stop;
    else next_state <= sleep;
    end if;

  when error => next_state <= error;
  when data_tx_rx => if (rw = '1') then -- если чтение
    if (n_bit = 7) then -- если прочитан байт
      -- если данные не готовы – прекратить чтение
      if new_data = '0' then next_state <= nack;
      else -- иначе, если прочитаны все байты
        if (num_read = ALL_READ_REGS) then
          next_state <= <состояние>;
        else next_state <= <состояние>;
        end if; -- if num_read = ...
      end if; -- if new_data = ...
      -- иначе продолжаем чтение
      else next_state <= data_tx_rx;
      end if; -- if (n_bit = 7)
    else -- если запись

```



```

when nack => err <= '0'; cnt <= '0';
                sda <= <лог. уровень>;
                gen_scl <= <лог. уровень>;
when stop => err <= '0'; cnt <= '0';
                gen_scl <= <лог. уровень>;
                sda <= '0';
when error => err <= '1'; cnt <= '0';
                sda <= 'Z'; gen_scl <= '0';
when data_tx_rx =>
    err <= '0'; cnt <= '1';
    -- если началась запись данных ...
    if (rw = '0' and n_bit = 0) then
    -- ... но шина SCL занята
        if (free = '0') then
            gen_scl <= <лог. уровень>;
        else gen_scl <= <лог. уровень>;
        end if; -- if free = ...
    end if; -- if rw = ...
    -- если выполняется чтение данных
    if (rw = '1') then
        sda <= <лог. уровень>;
    -- иначе, если выполняется запись
    else
    -- если передается адрес регистра Slave SUB
        if data_address = '0' then
            buf_8 := conv_std_logic_vector(INIT_ADDRESSES(num_init),8);
        else -- иначе, если передается байт данных DATA
            buf_8 := conv_std_logic_vector(CTRL_DATA(num_init),8);
        end if; -- if data_address = ...
        -- установка уровня на линии SDA для передачи '0' и '1'
        if (buf_8 (7-n_bit) = '0') then sda <= <лог. уровень>;
        else sda <= <лог. уровень>;
        end if; -- if (buf_8 (7-n_bit) = ...
    end if; -- if rw = ...
when others => null;
end case;
end if; -- if clock'event and clock = ...
end process;

-- счетчик бит со счетом по заднему фронту clk
-- и сбросом по cnt или free
process (clk, cnt, free)
    <операторы для счетчика бит>

```

```

-- анализ состояния линии SCL
process (clk, state)
begin
  if clk'event and clk = '1' then
    case state is
      -- анализ состояния SCL перед началом передачи данных
      when data_tx_rx => if n_bit = 0 then free <= scl;
                        else free <= '1'; end if;
      -- анализ линии SCL перед формированием условия P
      when stop => free <= scl;
      when others => free <= '1';
    end case;
  end if;
end process;

-- процесс формирования управляющих сигналов
process (clk, new_data, num_init, rw, init, type_tx)
variable cycle: natural := 0; -- переменная для счета тактов clk
variable buf: data_regs; -- буферный регистр
begin
  if (clk'event and clk = '1') then
    case state is
      when start => wd <= '0'; rw <= '0';
      -- если проинициализированы не все регистры
      if (num_init < ALL_INIT_REGS) then init <= '1';
      else init <= '0'; end if;
      when sak => ack <= <линия шины I2C>; RStart <= '1';
      -- определение типа следующих за ACK данных
      if type_tx = tx_data then
        if (init = '1') then
          if (data_address = '0') then data_address <= '1';
          else type_tx <= tx_stop;
          end if; -- if data_address = ...
        else
          type_tx <= tx_rs;
        end if; -- if init = ...
      end if; -- if type_tx =
      when slave_addr => type_tx <= tx_adr; data_address <= '0';
      when data_tx_rx => if rw = '1' then
        buf(num_read)(7-n_bit) := sda;
        new_data <= buf(0)(<номер бита>);
      end if;
      type_tx <= tx_data;
    
```

```

when sleep => rw <= '0';
    for i in 0 to ALL_READ_REGS-1 loop
        <обнуление массива векторов buf(i)>;
    end loop;
    -- формирование сигнала начала нового цикла
    if cycle = 0 then wd <= '1'; new_data <= '0';
    else wd <= '0'; end if;
    num_read <= 0; RStart <= '1';
when nack => if (new_data = '1') then
    data_out <= <выходные данные>;
    end if;
when mak => num_read <= num_read + 1;
when RS => rw <= '1'; RStart <= '0';
when stop => if init = '1' then num_init <= num_init + 1;
    end if;
when others => null;
end case;
-- счетчик тактов clk для задания интервала дискретизации
if cycle = <значение> then cycle := 0;
else cycle := cycle + 1;
end if;
end if; -- if clk'event and clk = ...
end process;
end Behavior;

```

1.8. Задать настроенную константу делителя равной 8, константу DISCRETE = 200. Откомпилировать проект; при наличии в тексте программы ошибок внести исправления.

1.9. Выполнить моделирование работы проекта. Для упрощения отладки добавить в *.vwf-файл, помимо входных и выходных сигналов сущности, текущее состояние ЦКА, служебные сигналы и переменную *cycle*. Задать частоту сигнала с кварца 24 МГц, время моделирования – 100 мкс. По умолчанию установить на линии SCL логическую '1'. Запустить моделирование. Убедиться в правильности передачи байта с адресом *Slave SAD* по линии SDA.

1.10. Выполнить перезапись *.vwf-файла результатами моделирования (II.1). Проанализировать правильность установки на линии SDA байта SUB или DATA.

1.11. Сымитировать в 9-м бите ответ ACK. Выполнить повторное моделирование.

1.12. Повторяя пп. 1.10 и 1.11, достигнуть состояния ЦКА data_tx_rx, в котором выполняется чтение данных из регистра

INT_SOURCE, и симитировать на линии SDA признак готовности данных.

1.13. Повторить моделирование. Убедиться в завершении цикла обмена сигналами NACK и P.

1.14. Симитировать ненулевой ответ от *Slave* в состояниях ЦКА data_tx_rx, в которых принимаются данные от устройства, и выполнить повторное моделирование.

2. Проверка работы проекта с датчиком ADXL345

2.1. Установить значение коэффициента делителя таким, чтобы обеспечить скорость обмена по шине I2C 400 кбит/с; изменить константу DISCRETE.

2.2. Подключить входные и выходные сигналы проекта к выводам ПЛИС отладочного комплекта. Линиям шины I2C, а также выходным портам cs и sad0 назначить номера выводов ПЛИС в соответствии со схемой подключения датчика ADXL345 к контактам разъема JP 1 (GPIO 0) или JP2 (GPIO 1). Выходной сигнал подключить к красным и зеленым светодиодам LEDR7 – LEDR0 и LEDG7 – LEDG0, порт егг – к светодиоду LEDR9. Откомпилировать проект заново.

2.3. **Не подключая плату с ПЛИС к ПЭВМ**, соединить контакты разъема датчика с контактами разъема JP 1 (GPIO 0): VCC – с контактом 3,3 В, GND – с контактом GND. Показать преподавателю результаты подключения выводов датчика. **Предельное напряжение обратной полярности для ADXL345 составляет всего -0,3 В, поэтому при неправильной коммутации выводов VCC и GND датчик будет поврежден.**

2.4. Включить отладочный макет. Загрузить проект в ПЛИС.

2.5. Наблюдать на светодиодах изменение сигнала ускорения.

2.6. Повторить п. 3 лабораторной работы № 1.

Лабораторная работа № 3

Изучение протоколов передачи видеоданных VGA, DVI и LVDS

Цель работы: синтез устройства для реализации передачи видеоданных по протоколам VGA и LVDS.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Стандарт VGA (от англ. *Video Graphics Array* – массив видеографики, читается как «ви-джи-эй») – это аналоговый стандарт передачи видеоданных, который определяет размер кадра изображения, количество цветов и частоту обновления кадров; принят в 1987 г. Шина VGA содержит как минимум три аналоговые линии R, G и B для передачи соответственно сигналов красного, зеленого и

синего с уровнем напряжения от 0 до 0,7 В и две цифровые линии для передачи импульсов строчной HSYNC (*Horizontal synchronization*) и кадровой VSYNC (*Vertical synchronization*) синхронизации.

При поступлении импульсов синхронизации луч в электронно-лучевой трубке монитора выполняет построчное сканирование, начиная с левого верхнего угла (на рис. 16 показаны номера строк и столбцов позиций луча для кадра размером 640х480 пикселей). При достижении конца строки луч переходит на строку ниже, а при достижении правого нижнего угла – снова в левый верхний угол. По окончании строки и кадра на линиях HSYNC и VSYNC формируются синхроимпульсы низкого логического уровня SYNC, переносящие луч соответственно на начало строки и на нулевую строку. Также на этих линиях должны быть предусмотрены передний (*Front Porch, FP*) и задний (*Back Porch, BP*) защитные интервалы высокого логического уровня (верхняя эпюра на рис. 17).

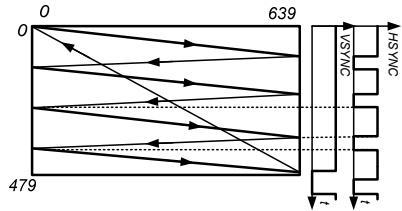


Рис. 16

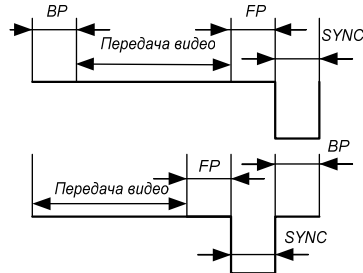


Рис. 17

При реализации на ПЛИС удобно начинать формирование синхросигнала не с *BP*, а с сигнала разрешения передачи видео (нижняя эпюра на рис. 17).

Основная (пиксельная) тактовая частота clk , выраженные в количестве ее периодов длительности логических уровней в линии HSYNC и выраженные

в количестве периодов HSYNC длительности в линии VSYNC строго регламентированы для каждого видеоформата.

Рассмотрим в качестве примера сигналы для формирования изображения 640х480 пикселей с частотой 60 Гц. Частота тактового сигнала clk для этого режима составляет 25,175 МГц, а длительности синхросигналов HSYNC и VSYNC, выраженные соответственно в периодах clk и HSYNC, приведены в табл. 8.

Длительность импульсов синхронизации

Таблица 8

| Синхросигнал | Передача видео | FP | SYNC | BP | Всего |
|--------------|----------------|----|------|----|-------|
| HSYNC | 640 | 16 | 96 | 48 | 800 |
| VSYNC | 480 | 10 | 2 | 33 | 525 |

Стандарт DVI (от англ. *Digital Visual Interface* – цифровой видеointерфейс, читается как «ди-ви-ай») – цифровой стандарт передачи видеоданных, который определяет размер кадра изображения, количество цветов и частоту обновления кадров; принят в 1999 г. DVI использует высокоскоростную технологию TMDS (*Transition Minimized Differential Signaling* – передача дифференциальных сигналов с минимизацией количества перепадов уровней): ее принцип заключается в таком кодировании последовательно передаваемых 10-битных слов, при котором количество переключений '0'-'1' и '1'-'0' в слове минимально.

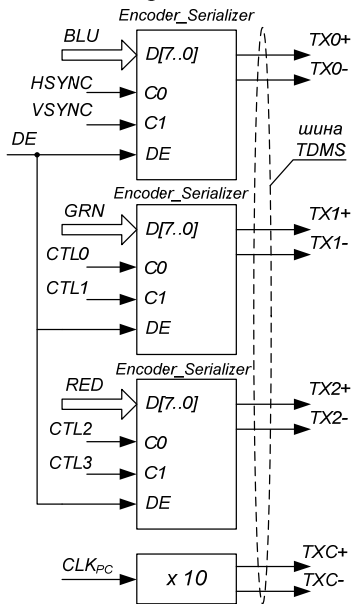


Рис. 18

При передаче видеоданных по одноканальной (*Single Link*) шине TDMS входные данные – 24 бита с информацией о цвете (по 8 бит на каналы RED, GRN и BLU), 4 бинарных управляющих сигнала CTL и импульсы HSYNC и VSYNC (рис. 18) – кодируются в блоке *Encoder_Serializer* (сдвиговый регистр-кодировщик) в 10-разрядный код, который последовательно передается по трем линиям связи типа «витая пара» (линии TX0-2) с синхронизацией тактовым сигналом CLK (линия TXC). Алгоритм кодирования 10-битных слов определяется сигналом DE (*Data enable*). На приемной стороне декодер выполняет обратную операцию, извлекая из последовательных кодов сигналы R, G, B, CTL, DE, HSYNC и VSYNC. В передатчике используется умножитель частоты, повышающий частоту тактового сигнала пиксельной частоты (*Pixel Clock*) CLK_{PC} в 10 раз: полученный тактовый сигнал CLK используется для тактирования сдвиговых регистров, а также передается по дифференциальной линии TXC.

Алгоритмы кодирования и декодирования для передачи сигналов по DVI рассмотрены в стандарте [7]. Логический уровень на линиях CTL определяет алгоритм кодирования при низком логическом уровне DE (в одноканальной шине TDMS на линиях CTL, как правило, устанавливается логический ноль).

Пример VHDL реализации передатчика DVI в соответствии с [7] приводится в интернет-источнике [8].

Стандарт LVDS (от англ. *Low-Voltage Differential Signaling* – передача низковольтного дифференциального сигнала, читается как «эл-ви-ди-эс») – цифровой стандарт передачи видеоданных, разработанный компанией National Semiconductor в 1994 г, так же, как и DVI, реализует передачу высокочастотного дифференциального низковольтного (0,3 В) видеосигнала по линиям типа «витая пара» (шина TDMS). Исходный поток параллельных данных (24 или 18 бит цвета на пиксель), а также сигналы DE, HSYNC (HS) и VSYNC (VS) конвертируются в 7-битные слова, которые последовательно передаются по дифференциальным линиям TX. Тактовая частота, передаваемая по отдельной дифференциальной паре TXC, получается умножением исходной частоты CLK_{PC} в семь раз. Логика кодирования для передачи информации об одном пикселе с детализацией цвета 24 и 18 бит приведена на рис. 19 и 20 соответственно.



Рис. 19



Рис. 20

Домашнее задание

1. Изучить основы передачи данных по протоколам VGA и LVDS.
2. Составить описание сущностей и архитектур цифровых устройств, синтезируемых в пп.1 и 2 порядка выполнения лабораторной работы.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Исследование передачи видеоданных по стандарту VGA

- 1.1. Создать новый проект с именем VGA_Timer.
- 1.2. Для соответствующих номеру бригады параметров видеопотока (табл. 9) определить по [9] пиксельную частоту и длительности логических уровней импульсов HSYNC и VSYNC.

Параметры видеопотока

Таблица 9

| № | 1 | 2 | 3 | 4 | 5 | 6 |
|---------------------|----------|----------|---------|----------|---------|---------|
| Размер кадра, пикс. | 1024x768 | 1152x864 | 800x600 | 1024x768 | 800x600 | 800x600 |
| Частота, Гц | 60 | 75 | 100 | 75 | 75 | 85 |

1.3. Подключить к проекту пакет с объявлением типа `VGA_params`, представляющего собой запись с полями `Video`, `FP`, `Sync` и `BP`, а также констант `H_TIMING` и `V_TIMING` данного типа, указав в данных полях моменты времени от начала периода сигнала синхронизации: например, для видеосигнала 640х480 пикселей с частотой кадров 60 Гц в соответствии с [9]:

`H_TIMING` = (640, 640 + 16, 640 + 16 + 96, 640 + 16 + 96 + 48);

`V_TIMING` = (480, 480 + 10, 480 + 10 + 2, 480 + 10 + 2 + 33).

1.4. Объявить сущность с настроечными константами `H` и `V` типа `VGA_params` и значениями по умолчанию `H_TIMING` и `V_TIMING` соответственно, входным портом `clk`, выходными портами `hsync` и `vsync`, портами `row` и `column` типа **natural** ограниченной разрядности для обозначения номера строки и столбца пикселя соответственно и логическим портом `video_on` для разрешения выдачи сигнала в линии `R`, `G` и `B`.

1.5. Написать содержательную часть архитектуры. Сформировать сигналы `hsync` и `vsync` с требуемыми временными интервалами для формата видеоизображения в соответствии с номером варианта. Сформировать высокий логический уровень `video_on` в моменты времени, соответствующие передаче видео, и низкий – в противном случае. Сформировать сигналы с номером столбца и строки.

1.6. Выполнить моделирование работы проекта для частоты $f_{clk} = 50$ МГц на интервале времени 3 – 5 периодов `hsync`, убедиться в его работоспособности.

1.7. Создать новый проект с именем `VGA_Video_Generator` с входными портами `clk`, `row`, `column` и `video_on` и 4-разрядными выходами `R`, `G` и `B` типа **std_logic_vector**.

1.8. Написать содержательную часть архитектуры со следующей логикой работы. При `video_on = '0'` значения `R`, `G` и `B` асинхронно обнуляются. При `video_on = '1'` по фронту `clk` генерируются коды `R`, `G` и `B` для формирования тестового кадра из 8 цветных вертикальных полос одинаковой ширины с цветами (слева направо): белый, желтый, голубой, зеленый, пурпурный, красный, синий, черный. Интенсивность сигналов (кроме черного): 75 % от максимального значения.

1.9. Выполнить моделирование работы проекта для первой строки, убедиться в его работоспособности.

1.10. Создать новый проект с именем `VGA` с входным тактовым сигналом `clock` и выходными портами `hsync`, `vsync`, `R`, `G` и `B`.

1.11. В декларативной части подключить к проекту компонент *PLL* (от англ. *Phase Locked Loop*) аппаратно реализованного генератора с петель фазовой автоподстройки частоты (ФАПЧ) для формирования

тактового сигнала требуемой пиксельной частоты из сигнала кварцев 24, 27 или 50 МГц (**П.2**). Принцип работы такого генератора подробно описан в [10]. Также подключить компоненты синтезированных в пп. 1.1 - 1.6 и 1.7 - 1.9 таймера и генератора.

1.12. С использованием структурного стиля программирования разработать содержательную часть архитектуры в соответствии с рис. 21.

1.13. Выполнить моделирование работы проекта для первой строки, убедиться в его работоспособности.

1.14. Подключить входные и выходные сигналы проекта к соответствующим выводам ПЛИС отладочного комплекта, воспользовавшись описанием к лабораторному макету *DE1 User Manual*.

1.15. Подключить монитор к разъему DB-15F отладочного макета.

1.16. Загрузить проект в ПЛИС, проверить его работоспособность.

2. Исследование передачи видеоданных по стандарту LVDS

2.1. Изменить константы `H_TIMING` и `V_TIMING` проекта `VGA_Timing` под формат видео 640x480 пикселей с частотой кадров 60 Гц (табл. 8). Принять пиксельную частоту равной 25 МГц. Изменить разрядность выходных сигналов сущности `VGA_Video_Generator` на 8.

2.2. Создать новый проект с именем `LVDS_coder` с входами `clk`, `hsync`, `vsync` и `de` типа **std_logic**, 8-разрядными входами `R`, `G` и `B` и выходами `tx0-tx3` и `txc` для кодирования согласно рис. 19. Разработать содержательную часть его архитектуры.

2.3. Создать новый проект LVDS передатчика с именем `LVDS_tx` с входом `clock` и выходами `tx0-tx3` и `txc`. Подключить к нему компоненты `VGA_Timing`, `VGA_Video_Generator` и `LVDS_coder`, а также генератор с петлей ФАПЧ `PLL`.

2.4. Составить архитектуру для реализации передатчика LVDS в соответствии со структурной схемой рис. 22 (для синтеза `PLL` с двумя выходами см. п. 7 приложения **П.2**).

2.5. Выполнить моделирование работы проекта для $f_{clock} = 50$ МГц на

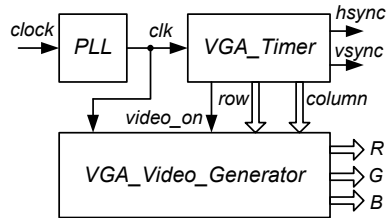


Рис. 21

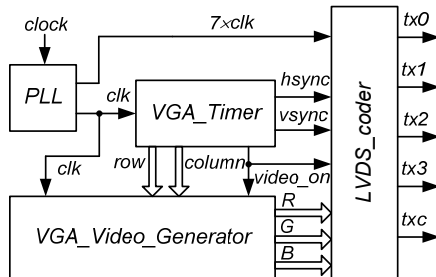


Рис. 22

интервале 3 - 4 периодов *hsync*. При необходимости вынести в файл моделирования вспомогательные сигналы. Доказать правильность работы синтезированной сущности.

Лабораторная работа № 4

Изучение основ проектирования систем на кристалле на примере процессора Nios II

Цель работы: синтез микропроцессорной (МП) системы с использованием ядра Nios II.

В конце 90-х – начале 2000-х годов на смену специализированным микросхемам пришли микроэлектронные разработки, основанные на программных моделях ядер устройств. Описания ядер на языках группы HDL позволили компоновать проекты из готовых и хорошо отлаженных модулей. Составленный таким образом проект принято называть «системой на кристалле» (от англ. «System-on-crystal», SoC). Синтезируемые МП системы можно реализовать на ПЛИС типа FPGA: фирма Xilinx предлагает процессорные ядра Microblaze и Picoblaze (32-х и 8-разрядные соответственно), фирма Altera – ядро Nios II.

Nios II – это встраиваемый программный 32-разрядный процессор общего назначения с перестраиваемой конфигурацией, оптимизированный для реализации на ПЛИС компании Altera со структурой FPGA. Nios II построен по гарвардской архитектуре, т.е. имеет две отдельные шины адреса и данных; разрядность каждой шины составляет 32 разряда. В его архитектуру также входят 32 регистра общего назначения (РОН) и 32 источника внешних прерываний. Допустимый объем внешнего адресного пространства – 2 Гб. Nios II является процессором конвейерного типа (т.е. команды в нем выполняются за один цикл частоты синхронизации), использует сокращенную систему команд (RISC) и поддерживает вычисления в формате с плавающей точкой. В процессорах RISC арифметические и логические операции выполняются над операндами, находящимися в регистрах общего назначения. Обмен информацией между регистрами и памятью МП системы осуществляется путем выполнения команд загрузки данных из регистра и сохранения данных в регистр.

Структура процессора Nios II показана на рис. 23. Блоки памяти команд и данных, помеченные как TC (от англ. *Tightly Coupled* – тесно связанные), принадлежат специальному типу памяти, обладающему малыми задержками и применяемому для критичных к быстродействию приложений. Такая память реализуется на кристалле, но вне ядра [11, С.607-620]. Так называемые теньевые регистры (от

англ. *shadow registers*) программисту не доступны – с ними может работать только МП. Типовое использование теневых регистров состоит в ускорении контекстных переключений.

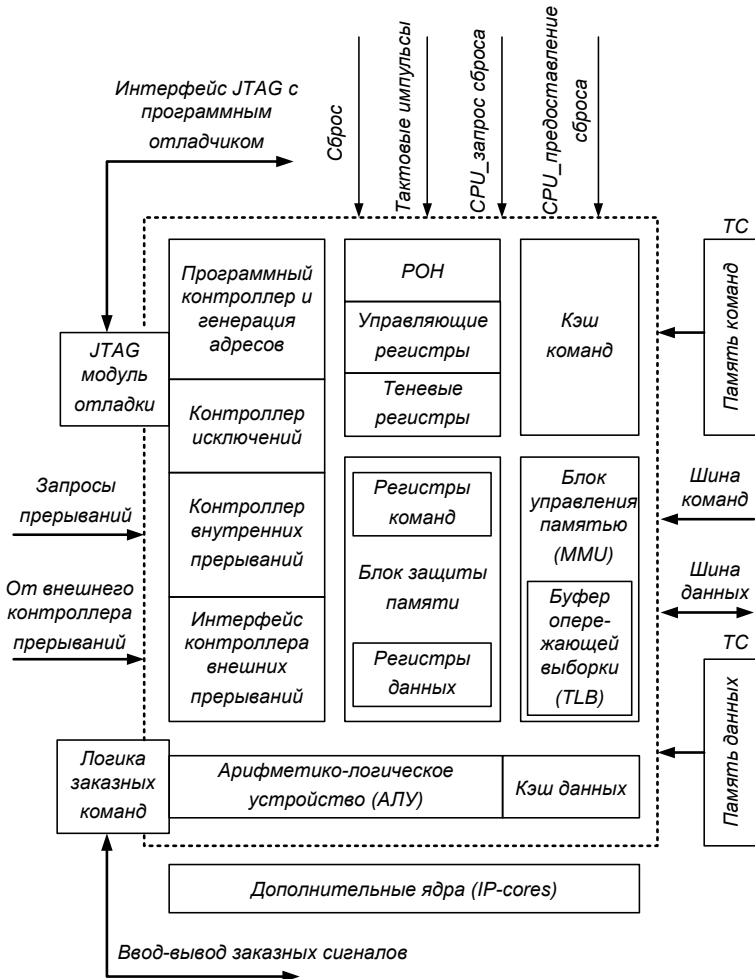


Рис. 23

Блок опережающей выборки команд необходим для конвейеризации – ускорения выполнения команд МП. Выполнение любой команды условно можно разделить на два этапа: выборку кода команды из памяти и непосредственно ее выполнение. Конвейеризация

позволяет сократить до минимума первый из этапов: при этом выборка команды из памяти и ее дешифрирование в стандартную для АЛУ форму объединяются так, что в любой момент времени несколько шагов выполняются одновременно, т.е. одна команда выбирается из памяти, вторая дешифрируется, а третья исполняется.

Наличие одновременно внутреннего и внешнего контроллеров прерываний не является противоречием, так как при работе внешнего контроллера внутренний отключается.

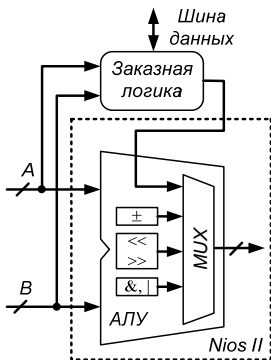


Рис. 24

Архитектура Nios II поддерживает до 256 определяемых пользователем (так называемых заказных, от англ. «*custom logic*») команд. АЛУ Nios II напрямую соединяется с логикой заказной команды, позволяя аппаратно реализовать операции, обращение к которым и использование которых происходит по аналогии со стандартными командами. Использование заказных команд позволяет добавить к АЛУ аппаратный узел собственной разработки, обрабатывающий задачи за один или несколько тактов, т.е. заменить сложную последовательность стандартных команд

МП единственной командой, которая аппаратными средствами выполняет те же действия. Добавленная пользователем заказная логика также может обращаться к памяти и логике вне Nios (рис. 24). Здесь необходимо отметить основное отличие МП, реализованного на ПЛИС, от стандартного МП. В стандартном МП скорость обработки зависит только от скорости работы самого микропроцессора (тактовой частоты), ресурсов (разрядность шин, число регистров, наличие умножителей и т. д.) и эффективности программного кода. При этом все ресурсы МП, как правило, участвуют в решении задачи последовательно, а данные из одного вычислительного процесса переносятся в другой вычислительный процесс под управлением программы. Для МП, реализованного на ПЛИС, ресурсами являются логические ячейки самой ПЛИС и формально неважно, как именно данные ресурсы связаны с МП – как отдельные вычислительные узлы (например, умножители с накоплением) или как конечные автоматы. После того как пользователь определил ресурс, требуемый для решения специфичных аппаратных задач, все остальные ресурсы ПЛИС могут быть задействованы для реализации вычислительных узлов. Также при реализации вычислений на заказной логике данные

передаются из одного вычислительного процесса в другой аппаратно, не требуя затрат времени МП [12].

К собственно процессорному ядру в процессе синтеза МП системы возможно подключение дополнительных ядер – *Intellectual Property cores (IP-cores)*: таймеров, блоков постоянной и оперативной памяти, модулей аппаратно реализованных интерфейсов (Ethernet, UART, SPI и др.), широтно-импульсных модуляторов и т.п. либо ядер, написанных пользователем.

Для Nios II доступны три конфигурации МП ядра: экономичная Nios II/e (*economy*), стандартная /s (*standart*) и быстрая /f (*fast*). Выбор конкретной конфигурации определяется ресурсами ПЛИС, сложностью вычислений и требуемым набором средств отладки. На реализацию перечисленных разновидностей ядра требуется соответственно не более 600, 1300 и 1800 логических элементов ПЛИС. Производительность МП в микросхемах семейства Stratix достигает для перечисленных вариантов 0,16, 0,75 и 1,17 DMIPS/МГц, что при работе на максимальных частотах ядер (135...150 МГц) соответствует 28, 120 и 200 DMIPS. Dhrystone MIPS, или DMIPS, – синтетический тест производительности МП; MIPS – количество миллионов инструкций в секунду (от англ. *million instructions per second*).

Подключение к процессору внешних устройств осуществляется либо с помощью отдельных шин данных для записи и для чтения, либо с использованием шины с тремя состояниями.

Для построения системы на кристалле (СнК) с процессором Nios II используется шина Avalon. Ее основными отличительными особенностями являются простота, малое количество ячеек кристалла, задействованных для логики шины, и полностью синхронные операции.

Каждая шина Avalon подключает единственного *Master* к нескольким периферийным устройствам. *Master* шины Avalon обычно выступает ядро микропроцессора Nios II. Периферийным устройством Avalon может быть, например, UART, таймер или устройство памяти. Все транзакции шины Avalon передают (перемещают) единственный байт, слово половинной разрядности (2 байта) или слово (4 байта) между одним из периферийных устройств и *Master*. Шина Avalon адаптирована для синтеза на кристалле: для взаимосвязи устройств вместо шин с тремя состояниями используются мультиплексоры.

Как правило, шина Avalon является модулем с большим количеством линий ввода-вывода. Модуль шины имеет один специальный периферийный порт интерфейса (*peripheral special interface port*, PSIP) для каждого *Slave* и один PSIP для *Master* («владельца» шины), а также содержит логику, которая выполняет следующие функции:

декодирование адреса – производит сигналы выбора кристалла для каждого периферийного устройства;

мультиплексирование шины данных – передает данные от выбранного *Slave* к *Master*;

динамическое изменение разрядности шины – для чтения или записи данных большей разрядности, чем у периферийного устройства, автоматически выполняются многократные циклы шины;

назначение номера прерывания – определяет номер и приоритет прерывания *IRQ (Interrupt request)*, когда один или более *Slave* одновременно требуют запроса прерывания.

При добавлении либо удалении *Slave* вся связанная с ним информация (PSIP) также добавляется или удаляется из шины Avalon.

Процессор Nios II поддерживает полный набор встроенных системных инструментальных средств развития: C/C++ транслятор, ассемблер и JTAG-отладчик. Ядро Nios II (в зависимости от конфигурации) включает поддержку аппаратных контрольных точек и управления через интерфейс JTAG.

Типы данных Altera Таблица 10

| Тип Altera | Аналог в C | Кол-во байт |
|----------------|----------------------------|-------------|
| alt_8 | char | 1 |
| alt_u8 | unsigned char, byte | 1 |
| alt_16 | short | 2 |
| alt_u16 | unsigned short | 2 |
| alt_32 | int | 4 |
| alt_u32 | unsigned int | 4 |

При разработке программ для Nios II Altera рекомендует применять собственные типы данных, объявленные в заголовочном файле *alt_types.h* (табл. 10).

Наиболее полное руководство по работе с процессором Nios II

(на английском языке) приводится на сайте разработчика [13, 14].

Для выполнения лабораторной работы также можно воспользоваться справочными материалами из пособий [15, 16].

Домашнее задание

1. Изучить основы архитектуры МП Nios II.

2. Воспользовавшись программой *WinFilter.exe*, определить в соответствии с табл. 11 весовые коэффициенты фильтра авторегрессии – скользящего среднего (АРСС) для фильтров нижних (ФНЧ) и верхних (ФВЧ) частот и порядка фильтра $n = 1$.

Тип цифрового фильтра и относительная частота среза Таблица 11

| № | 1 | 2 | 3 | 4 | 5 | 6 |
|---------------|-------------|-----|----------|------|-------------|------|
| Тип фильтра | ФВЧ | | | ФНЧ | | |
| Тип АЧХ | Баттерворта | | Чебышева | | Баттерворта | |
| Частота среза | 0,15 | 0,2 | 0,25 | 0,25 | 0,2 | 0,15 |

Построить переходную (ПХ) и импульсную (ИХ) характеристики АРСС-фильтра.

3. Составить программы на языке С в соответствии с требованиями пп. 2 и 3 лабораторной работы.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Синтез структуры МП системы (аппаратная часть)

1.1. Создать новый проект с именем Nios_proс. В меню **Tools** выбрать мастер настройки системы на кристалле *Qsys*. В результате будет создан новый модуль СнК – генератор тактового сигнала и сигнала сброса (*Clock Source*) с именем *clk_0*. Переименовать модуль в *clk*, вызвав контекстное меню на его имени либо нажав F2. Сохранить СнК с именем *nios_cpu*.

1.2. Дважды нажать левой кнопкой мыши (ЛКМ) на модуле *Clock Source* и в открывшемся диалоговом окне установить частоту тактового сигнала 50 МГц (50000000 Гц) в поле *Clock frequency*. Нажать *Finish*, чтобы применить изменения.

1.3. Добавить в СнК модуль МП ядра Nios II из библиотеки (*Library*) *Processors* двойным нажатием ЛКМ на Nios II Processor либо кнопку **Add** (добавить). Выбрать минимальную конфигурацию МП (Nios II/e) в диалоговом окне. Поскольку карта памяти МП системы еще не сформирована, то дальнейшее конфигурирование модуля МП временно не производить. На странице *JTAG Debug Module* (JTAG-отладчик) выбрать опцию *Level 1* для поддержки основных отладочных функций (загрузка программы по интерфейсу JTAG, программные прерывания). Нажать *Finish*. Переименовать модуль МП в *nios_cpu*.

1.4. Добавить модуль ОЗУ программ и данных: ***Memories and Memory Controllers -> On-Chip -> On-Chip Memory (RAM or ROM)***. В открывшемся диалоговом окне выбрать ОЗУ (RAM) и задать объем памяти ***Total Memory Size*** 16384 байта, размер слова данных ***Data Width*** – 32 (бита). Остальные параметры оставить без изменений. Нажать «*Finish*». Переименовать модуль в *on_chip_RAM*.

1.5. Добавить компонент аппаратного идентификатора версии System ID Peripheral из библиотеки ***Peripherals -> Debug and Performance -> System ID Peripheral***. Идентификатор обновляется при каждой генерации СнК и позволяет контролировать соответствие версий сгенерированного программного кода и СнК. Если значение идентификатора в программном обеспечении и в загруженной в ПЛИС СнК не совпадает, при загрузке программного обеспечения будет выдана ошибка. Задать имя модуля *sysid*, значение идентификатора ***System ID*** (в десятичном коде) *ГГ0N*, где ГГ – последние две цифры текущего года, N – номер бригады.

1.6. Добавить модуль UART из библиотеки *Interface protocols* -> *Serial* -> *UART (RS-232 Serial Port)* и принять его параметры по умолчанию: количество бит данных **Data bits** – 8, стоповый бит **Stop bits** – 1, бит четности отсутствует (**Parity** – *None*), скорость передачи данных **Baud Rate** – 115200 бит/с.

Поскольку все добавленные модули пока не соединены между собой, в нижней части рабочего окна мастера *Qsys* будут отображены сообщения об ошибках при синтезе МП системы.

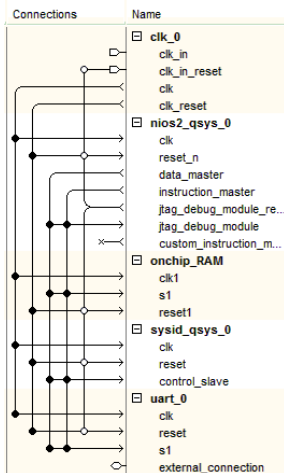


Рис. 25

1.7. Создать сеть синхронизации и сброса СнК. Для этого на вкладке *System Contents* рабочего окна *Qsys* в столбце *Connections* соединить выходы блока синхронизации и сброса *Clock Output (clk)* и *Reset Output (clk_reset)* с входами МП, памяти, идентификатора системы и UART соответственно. Соединение осуществляется нажатием ЛКМ на узле соединения линий (выполненное соединение отображается заливкой черным цветом, рис. 25).

1.8. Подключить все блоки МП СнК к системной шине Avalon: выходы *Data master* и *Instruction Master* процессора Nios II подключить к входу *s1* модулей ОЗУ и UART, а также к входу *control_slave*

идентификатора системы (рис. 25).

1.9. Экспортировать сигналы *tx* и *rx* модуля UART во внешние порты. Для этого во вкладке *System Contents* выбрать компонент UART и сигнал *external connection*, затем, нажав ЛКМ в столбце *Export*, указать название сигнала – *uart* и нажать *Enter*.

1.10. Соединить выход запроса на прерывание *IRQ* блока UART с входом *IRQ* процессора (*IRQ 31*) в столбце *IRQ*.

1.11. Выполнить настройку векторов исключений и прерываний МП (отличие между этими терминами рассмотрено в П.3). Дважды нажать ЛКМ на модуле МП Nios II/e. В открывшемся диалоговом окне в поле *Reset Vector* (адрес ячейки памяти при сбросе) выбрать параметр *Reset Vector Memory: on_chip_RAM.s1*. В поле *Exception Vector* (адрес ячейки памяти при исключении) выбрать параметр *Exception Vector Memory: on_chip_RAM.s1*. Нажать кнопку *Finish*.

Другие настройки МП кратко рассмотрены в П.4.

1.12. Назначить базовые адреса устройств МП системы и приоритет прерываний: команды **System -> Assign Base Addresses** и **System -> Assign Interrupt Numbers** соответственно.

1.13. На вкладке *HDL Example* в выпадающем списке *HDL Language* выбрать язык описания VHDL и скопировать в буфер обмена VHDL-описание компонента МП.

1.14. Сохранить конфигурацию МП СнК и закрыть окно *Qsys*.

1.15. Создать новый VHDL-файл. Объявить в нем сущность *nios_proc* с входами тактового сигнала и сброса, а также портами UART. Подключить файл с описанием МП *nios_cpu.qsys* через меню **Project -> Add/Remove Files in Project**. Подключить к проекту компонент МП. Скомпилировать проект. Подключить входные и выходные сигналы проекта к выводам ПЛИС отладочного комплекта (для формирования сигнала сброса использовать одну из кнопок *Key*) и выполнить повторную компиляцию.

2. Разработка программной части

2.1. Запустить программу Nios II Software Build Tools for Eclipse (далее по тексту – Eclipse) и в открывшемся диалоговом окне выбрать для сохранения проекта Eclipse каталог с проектом *Nios_proc*.

2.2. Для создания нового проекта использовать меню **File -> New -> Nios II Application and BSP from Template**. В открывшемся диалоговом окне настроек в качестве шаблона проекта *Template* выбрать *Blank project* (пустой проект); на панели *Target Hardware information* (сведения об аппаратной части СнК) указать:

SOPC Information File Name: ... /nios_cpu.sopcinfo;
project name: nios_cpu.

Для применения сделанных изменений нажать кнопки *Next* и *Finish*. В результате будут созданы два проекта: проект *nios_cpu_bsp* с набором драйверов, настроек и библиотек для аппаратной МП системы (*Board support package, BSP*) и шаблон проекта *nios_cpu*.

Примечание: при работе с Eclipse на ПЭВМ с операционной системой Windows 7 и выше возможно появление сообщения об ошибке о невозможности создания проекта «*Failed to execute: ./create-this-app --no-make*». В этом случае нужно выполнить запуск Eclipse с правами администратора, вызвав в меню «Пуск» контекстное меню ярлыка Eclipse и выбрав пункт «Запуск от имени администратора».

2.3. Добавить C-файл в проект. Для этого вызвать контекстное меню каталога *nios_cpu* в дереве проектов *Project Explorer*, выбрать **New->Other->Source File**, нажать *Next*, в появившемся диалоговом окне в поле *Source file* ввести имя нового файла «*nios_cpu.c*» и нажать *Finish*.

2.4. Добавить в файл *nios_cpu.c* код программы приема данных по UART, их фильтрации и выдачи результата по UART в формате с

плавающей точкой. Формат входных данных – 8-разрядный дополнительный код с фиксированной точкой: старший разряд – знаковый, фиксированная точка, 7 информационных бит дробной части.

```
// подключение заголовочного файла с описанием
// функций ввода/вывода данных и типов данных Altera
#include "sys/alt_stdio.h"
#include "alt_types.h"
static alt_u8 TxRx; // байт данных в дополнительном коде
<объявление массивов переменных для цифрового фильтра>;
// основная программа
int main() {
    while (1) { // организация бесконечного цикла
        TxRx = alt_getchar(); // чтение одного байта данных по UART
        <перевод из доп. кода в дробное число со знаком>;
        <операторы для реализации цифрового фильтра>;
        <перевод дробного числа со знаком в целое в доп. коде>;
        // запись результата в порт UART
        alt_putchar(<байт данных с результатом фильтрации>);
    } // end while
    return 0;
} // end main
```

Сохранить файл с кодом программы.

2.5. Создать образ слоя абстрагирования (*Hardware Abstraction Layer, HAL*), позволяющий взаимодействовать инструкциям высокоуровневого языка программирования C/C++ с низкоуровневыми компонентами (аппаратным обеспечением) и содержащий драйверы устройств, используемых в аппаратном проекте. Для этого в окне Eclipse выбрать пункт **Nios II -> BSP Editor**, в открывшемся диалоговом окне выбрать меню **File -> Open** и указать путь к файлу настроек SnK settings.bsp (...\\Software\\nios_cpu_bsp\\). Оптимизировать HAL для экономии объема памяти, занимаемого программой (**П.5**).

Нажать на кнопку *Generate* в окне Eclipse. Сохранить сгенерированный слой абстрагирования. Закрыть окно **BSP Editor**.

2.6. Выполнить сборку программного проекта (команда **Project -> Build All**). При необходимости исправить синтаксические ошибки в тексте программы.

2.7. Оценить объем памяти *nnn*, занимаемый программой: в окне дерева проектов выбрать .elf-файл (**nios_cpu -> Binaries -> nios_cpu.elf**) и в поле служебной информации на вкладке *Console* найти строки:

Info: (nios_cpu.elf) *nnn* Bytes program size (code + initialized data).

Info: *mmm* Bytes free for stack + heap.

Оставшаяся от выделенного в п. 1.4 объема память RAM ($mmm = [16384 - nnn]$ байт) может быть задействована для организации стека и так называемой «кучи» (от англ. *heap*) для динамического выделения памяти.

Файл с расширением *.elf* представляет собой программу для загрузки в МП Nios II (П.6).

3. Загрузка программы и ее отладка

3.1. Подключить кабель RS-232 к разъемам отладочного макета и ПЭВМ. Загрузить аппаратный проект СнК nios_proс в ПЛИС средствами Quartus II.

3.2. Выполнить загрузку программного проекта МП nios_cpi в ПЛИС средствами Eclipse. Выбрать пункт меню **Run -> Debug Configurations....** Создать новую отладочную сессию, нажав правой кнопкой мыши на меню *Nios II Hardware* и выбрав *New*. В открывшемся служебном окне в поле *Project Name* выбрать nios_cpi. Убедиться в активности соединения с микросхемой загрузчика USB-Blaster на вкладке *Target Connection* (в таблице *Processors* должна быть указана строка с процессором Nios II). В противном случае обновить список процессоров, нажав кнопку *Refresh Connections* в правой части служебного окна. Отметить пункты «*Ignore mismatched system timestamp*» (игнорировать время сборки МП системы) и «*Reset the selected target system*» (после загрузки выполнить сброс МП системы). Нажать кнопку *Debug*.

3.3. После загрузки программы в процессор выполнить анализ работы проекта с использованием программы терминала *HTerm.exe*, позволяющей принимать и передавать данные через COM-порт ПЭВМ. После запуска *HTerm.exe* подключиться к активному COM-порту ПЭВМ (как правило, это порт *Com1*) и настроить параметры передачи в соответствии с настройками модуля UART СнК (П.7).

3.4. Подать на вход процессора функцию единичного скачка, записав значение 3F (в кодировке *hex*) в строке поля *Input control* (П.7).

3.5. Проанализировать ответ МП Nios II, сравнив результаты домашних расчетов и первый байт принятых данных ПХ.

3.6. Повторить пп. 3.4 и 3.5 для получения остальных отсчетов ПХ или ИХ цифрового фильтра.

3.7. Проанализировав принятые от МП байты данных со значениями отсчетов ПХ, сделать вывод о правильности работы цифрового фильтра. При необходимости просмотра текущих значений переменных в функции *main()* использовать точки останова в Eclipse и пошаговую отладку (П.8).

3.8. Изменить арифметику программного кода на целочисленную. Повторить пп. 2.6 - 3.7. Сравнить результаты работы фильтров. Оценить сокращение занимаемого объема RAM при переходе к целочисленной арифметике.


3.9. Закомментировать в коде программы операторы и переменные для реализации фильтра и добавить строки, реализующие передачу по UART значения идентификатора системы (System ID) в виде четырех байт. Для этого подключить к проекту следующие заголовочные файлы:

```
#include "system.h"
#include "altera_avalon_sysid_qsys.h"
#include "altera_avalon_sysid_qsys_regs.h"
#include "alt_types.h"
```

Для получения System ID вызвать функцию `IORD_ALTERA_AVALON_SYSID_QSYS_ID(SYSID_BASE)` и записать возвращаемое ей значение в переменную `hardware_id` типа **static alt_u32**, где `SYSID_BASE` – базовый адрес блока *sysid*, который автоматически записывается в файл `system.h` при сборке проекта. Для разделения `hardware_id` на 4 байта использовать битовые маски и операции побитового И с последующим сдвигом вправо.

3.10. Выполнить компиляцию программного проекта; загрузить его в ПЛИС. С помощью программы терминала принять значение идентификатора СнК *sysid*, перевести в десятичный код и сравнить со значением, присвоенным в п. 1.5.

Приложение

П.1. Для перезаписи файла моделирования необходимо на странице с результатами моделирования *Simulation Report* выбрать инструмент *Waveform Editing Tool*  и в появившемся окне предупреждения выбрать пункт «*I want to overwrite my vector input file ...*» («Я желаю перезаписать исходный файл моделирования ...»).

П.2. Создание компонента ФАПЧ.

1. В меню **Tools** выбрать мастер подключения компонентов **MegaWizard Plug-In Manager**.

2. В открывшемся диалоговом окне выбрать пункт **Create a new custom megafunction variation** (*создать новую мегафункцию*).

3. В новом диалоговом окне выбрать язык описания VHDL; в списке доступных мегафункций в левой части окна раскрыть папку I/O, нажав на «+» слева от нее, и выбрать компонент **ALTCLKLOCK**; в поле «**What name do you want for the file?**» (*какое имя Вы хотите назначить файлу?*) ввести идентификатор (например, pll<выходная частота>MHz) и нажать кнопку **Next**. Появится многостраничное окно с настройками компонента ФАПЧ **ALTCLKLOCK**.

4. В окне **Parameter Settings: General/Modes** (основные настройки) в поле «**What is the frequency of the inclock0 input?**» (частота входного сигнала) ввести значение частоты выбранного кварца (24, 27 или 50 МГц).

5. В следующем окне **Parameter Settings: Scan/Inputs/Lock** (дополнительные входы) снять галочки в пунктах «**Create an 'areset' input ...**» (создать вход асинхронного сброса) и «**Create 'locked' output ...**» (создать выход для уведомления о входе системы ФАПЧ в режим синхронизации). Удаление указанных портов можно контролировать на условном графическом обозначении компонента в левом углу окна.

6. В окне «**Output clocks: clk0**» выбрать вариант «**Enter output clock frequency**» (ввод частоты выходного сигнала) и в разблокированном поле ввести требуемое значение выходной частоты. Если данное значение f_1 можно представить в виде $f_1 = n_1 f_0 / n_2$, где n_1 и n_2 – натуральные числа, меньшие 64, то в верхней части окна текстом синего цвета будет показано подтверждение «**Able to implement ...**» (возможно применить). Значения n_1 и n_2 при этом отобразятся в полях «**Clock multiplication factor**» (множитель) и «**Clock division factor**» (делитель) соответственно. В противном случае текстом красного цвета будет показано подтверждение «**Cannot implement ...**» (нельзя применить), т.е. компонент ФАПЧ с текущими настройками не может быть синтезирован. В этом случае необходимо вернуться к окну выбора входной частоты и изменить ее значение. Также в окне «**Output clocks: clk0**» можно задать скважность выходного сигнала (**Clock duty cycle**) и сдвиг его фазы (**Clock phase shift**).

7. При необходимости аналогичным п. 6 образом в полях «**Output clocks: clk1(2)**» можно задать еще две выходных частоты.

8. В окне **Summary** (результаты) можно выбрать файлы, которые будут созданы в директории проекта (обязательно должен быть выбран файл с расширением *.vhd).

П.3. Прерывание и исключение – события, означающие, что в МП системе и/или исполняемой программе возникло состояние, требующее реакции на него процессора. Как правило, прерывания и исключения приводят к принудительной передаче управления от текущей программы к специальной процедуре – обработчику прерывания/исключения. При появлении запроса на прерывание и/или при исключении текущая задача приостанавливается на время, в течение которого МП выполняет код обработчика. По его окончании МП возобновляет прерванную задачу.

Прерывания вырабатываются по сигналам управления внешних цифровых устройств или внутренних таймеров МП системы.

Исключения вырабатываются при возникновении ошибки в ходе выполнения программы (деление на ноль, корень квадратный или логарифм от отрицательного аргумента и др.).

П.4. На странице *Caches and Memory Interfaces* (кэш-память) в конфигурациях Nios II/s и Nios II/f пользователь может установить объем кэша команд (для конфигурации Nios II/e он по умолчанию равен 4 кбайт) и поддержку пакетных передач (*Bursts Transfers*) с настройкой объема пакета. В лабораторной работе память программ расположена во внутренней статической памяти МП, поэтому пакетную передачу включать не требуется (при использовании динамической памяти включение пакетных передач желательно). При использовании внешних модулей памяти (ОЗУ и ПЗУ) также рекомендуется задействовать один порт тесно связанной памяти команд (*tightly coupled instruction master port*), который используется для прямого доступа МП к модулям оперативной памяти на кристалле, что позволяет повысить производительность и избежать конфликтов (например, при одновременном доступе к общей внешней шине микропроцессорной системы шин данных и инструкций процессорного ядра).

П.5. Для оптимизации памяти на панели *hal* ветви *Common* (общие настройки) дерева настроек *Settings* оставить галочки только на пунктах «*enable_reduced_device_drivers*» и «*enable_small_c_library*» (разрешить сокращение списка драйверов и использование упрощенной библиотеки C). Поскольку программа для СнК в лабораторной работе составлена на языке C и работает в бесконечном цикле, также можно отключить поддержку C++ и опцию разрешения выхода из программы. Для этого на панели *hal* ветви *Advanced* (дополнительные настройки) снять галочки в пунктах «*enable_c_plus_plus*» (поддержка C++), «*enable_clean_exit*» и «*enable_exit*» (разрешение выхода).

Выбрать пункт «*enable_lightweight_device_driver_api*».

П.6. Файл конфигурации *.sof* («прошивка», формируемая Quartus II) загружается в конфигурационное ОЗУ ПЛИС для формирования аппаратной части СнК, а файл *.elf* («прошивка», формируемая Eclipse) загружается в ОЗУ МП, чтобы программа Nios II запустилась с нужной точки (как правило, с адреса 0x00.0).

При автономной работе СнК файлы *.sof* и *.elf* загружаются не напрямую в ПЛИС, а сначала в конфигурационное ПЗУ (в этом случае адреса *.sof* и *.elf* файлов записываются со смещением) [16]. Чтобы *.sof* и *.elf* файлы в этом случае были размещены в конфигурационной памяти ПЛИС друг за другом, в СнК используется специальная микросхема – *EPCS Serial Flash* контроллер (флэш-память). Контроллер имеет в своем

составе специальный загрузчик (*boot loader*) – программу, которая запускается при включении питания. *Boot loader* выполняет последовательную загрузку аппаратной и программных частей проекта и переводит вектор состояния МП на начало программы.

П.7. В выпадающем списке *Port* окна программы *HTerm.exe* выбрать порт COM1. В полях справа от списка портов задать скорость передачи (*Baud*), длину информационной части в битах (*Data*), количество стоповых бит в протоколе (*Stop*) и тип бита четности (*Parity*). Установить шестнадцатеричную кодировку передаваемых и принимаемых байт, выбрав *hex* в выпадающем списке *Type* панели *Input control*. Подключиться к выбранному COM-порту, нажав на кнопку *Connect*. Для отправки байта данных по UART ввести его значение в поле панели *Input control* и нажать кнопку *ASend*. В появившемся служебном окне в поле *Repetitions* задается количество повторов посылок байта (при вводе нулевого значения устанавливается режим непрерывных посылок). Байты данных, отправленные в МП систему, отображаются в поле *Transmitted data*, принятые из МП системы – в поле *Received data*.

П.8. Команды для пошаговой отладки доступны в меню **Run**. Точка останова ставится двойным нажатием ЛКМ в поле, расположенном слева от кода программы (имеет серо-синий фон). Для просмотра значений переменных можно воспользоваться комбинацией «горячих» клавиш **Alt + Shift + Q**, **V** либо служебным окном **Window -> Show View -> Variables**. Если меню **Variables** не доступно, следует зайти в меню **Other...** и найти **Variables** в подкаталоге **Debug**.

Библиографический список

1. Новицкий А. Синхронный последовательный интерфейс SPI в микроконтроллерах «от А до Я» и его реализация на примере ADuC70xx фирмы Analog Devices // Компоненты и технологии. – 2009. – № 3. – С. 53-60.
2. <http://www.gaw.ru/html.cgi/txt/interface/spi/index.htm>.
3. <http://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf>.
4. Семёнов Б.Ю. Шина I2C в радиотехнических конструкциях. – М.: Солон-Р, 2002. – 190 с.
5. <http://easyelectronics.ru/interface-bus-iic-i2c.html>
6. <http://robocraft.ru/blog/communication/780.html>
7. Digital Visual Interface. Revision 1.0. – 1999. – 76 p.
8. http://hamsterworks.co.nz/mediawiki/index.php/Dvid_test
9. <http://tinyvga.com/vga-timing.html>

10. Васильев Е.В. Цифровые радиопередающие устройства: учеб. пособие. – Рязань: РГРТУ, 2006. – 72 с.

11. Угрюмов Е.П. Цифровая схемотехника: учеб. пособие для вузов. – 3-е изд., перераб. и доп. – СПб.: БХВ-Петербург, 2010. – 816 с.

12. Каршенбойм И. Микроконтроллер для встроенного применения – NIOS. Система команд и команды, определяемые пользователем. Часть II. Команды перехода, исключения, конвейер и команды, определяемые пользователем // Компоненты и технологии. – 2002. – № 9. – С. 32-36.

13. SOPC Builder User Guide. https://www.altera.com/en_US/pdfs/literature/ug/ug_socp_builder.pdf.

14. Nios II Classic Software Developer's Handbook https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2sw_nii5v2.pdf.

14. Ефремов Н.В., Бородин А.А. Инструментальные средства проектирования и отладки систем на программируемых кристаллах компании Altera: учеб. пособие. – М.: МГУС, 2012. – 150 с.

15. Проектирование систем на кристалле на основе ПЛИС. http://e-learning.bmstu.ru/moodle/pluginfile.php/2978/mod_data/content/880/EVM_Lab1.pdf.

16. Как запрограммировать конфигурационное ПЗУ Altera типа EPCS. <http://acvarif.info/artplis/artplis3.html>.

Содержание

| | |
|--|----|
| Лабораторная работа № 1. Изучение протокола передачи данных SPI..... | 1 |
| Лабораторная работа № 2. Изучение протокола передачи данных I2C | 15 |
| Лабораторная работа № 3. Изучение протоколов передачи видеоданных VGA, DVI и LVDS | 28 |
| Лабораторная работа № 4. Изучение основ проектирования систем на кристалле на примере процессора Nios II | 34 |
| Приложение | 44 |
| Библиографический список | 47 |