

# **Методическое пособие**

**к лабораторным работам, проводимым на макетах  
Digilent NEXYS2 Board в среде разработки Xilinx ISE  
Design Suite 14.4**

# Содержание

Введение .....	3
Комбинационная и последовательностная логика .....	3
Логические вентили .....	4
Мультиплексор .....	6
Триггеры.....	6
D-триггер .....	7
Т-триггер .....	9
Программируемые логические интегральные схемы (ПЛИС) и Look-Up Tables (LUT).....	11
Язык Verilog .....	12
Базовые типы источников сигналов .....	12
Иерархия проекта .....	14
Арифметические и логические функции .....	15
Сложение и вычитание .....	15
Логический и арифметический сдвиг.....	15
Битовые логические операции .....	16
Булевы логические операции .....	17
Операторы редукции.....	17
Оператор условного выбора .....	17
Операторы сравнения.....	18
«always» блоки. Блокирующие и синхронные присвоения .....	18
Физическое проектирование устройств и ограничения Verilog .....	22
Лабораторный макет .....	22
ПЛИС Spartan3E-500 FG320.....	25
Среда разработки Xilinx ISE Design Suite 14.4 .....	30
Создание проекта.....	30
Создание и синтез модуля .....	30
Логическое моделирование устройств .....	32
Графический редактор схем .....	35
Файл назначения портов схемы физическим выводам ПЛИС (User constraints file).....	37
Список литературы.....	40

## Введение

Электронные схемы и их составляющие могут быть разделены на два ключевых типа в зависимости от общих принципов их функционирования: аналоговые (непрерывные) и цифровые (дискретные). В аналоговых цепях используется непрерывный диапазон напряжения, в противоположность дискретным уровням, которые применяются в цифровых схемах. В некоторых случаях провести чёткое разграничение между непрерывными и дискретными схемами сложно — в силу того, что как те, так и другие включают в свой состав элементы и линейного, и нелинейного характера. Примером может послужить, допустим, компаратор, используемый для аналогово-цифрового преобразования сигнала: получая на входе непрерывный диапазон напряжения, он в то же время выдает на выходе лишь один из двух возможных уровней сигнала, подобно цифровой схеме. Такая схема обычно называется не аналоговой или цифровой, а смешанной. В настоящее время сложно найти электронную схему, которая была бы полностью аналоговой, повсеместно используются цифровые технологии.

К цифровым относятся схемы, основанные на некотором количестве дискретных уровней напряжения. Они представляют собой наиболее типичную физическую реализацию булевой алгебры и составляют элементную основу всех цифровых компьютеров. Термины «цифровая схема», «цифровая система» и «логическая схема» часто при этом рассматриваются как синонимичные. Для цифровых схем характерна, как правило, двоичная система с двумя уровнями напряжения, которые соответствуют логическому нулю и логической единице соответственно. Часто первый соотносится с низким напряжением, а вторая — с высоким, хотя встречаются и обратные варианты. К числу базовых конструктивных элементов этого типа относятся: логические вентили, сумматоры, триггеры, счетчики, регистры, мультиплексоры, а также устройства с высокой степенью интеграции – микроконтроллеры, микропроцессоры, программируемые пользователем интегральные схемы (ПЛИС) и другие.

Цель лабораторных работ, проводимых на макетах Digilent NEXYS2 Board в среде разработки Xilinx ISE Design Suite 14.4 – знакомство с базовыми элементами цифровой электроники, языком формального описания электрических цепей и аппаратуры Verilog, который используется для проектирования логики ПЛИС, а также освоение лабораторного макета Digilent NEXYS2 Board и среды разработки Xilinx ISE Design Suite 14.4.

## Комбинационная и последовательностная логика

В цифровых устройствах комбинационной логикой (комбинационной схемой) называют логику функционирования устройств, в которой состояние выхода устройства однозначно определяется набором входных сигналов. Это отличает комбинационную логику от последовательностной (секвенциальной) логики, в рамках которой выходное значение зависит не только от текущего входного воздействия, но и от предыстории функционирования цифрового устройства. Другими словами, последовательностная логика предполагает наличие памяти, которая в комбинационной логике не предусмотрена. Комбинационная логика, как правило, работает асинхронно, т.е. как только на входах комбинационной схемы изменяется состояние сигналов, состояние выхода также мгновенно меняется. Логические вентили, сумматоры, мультиплексоры, шифраторы и дешифраторы служат примером комбинационной логической схемы.

Последовательностная схема, как правило, работает синхронно, т.е. в качестве одного входа схемы добавляется сигнал синхронизации (CLK), и состояние выхода устройства изменяется не мгновенно после изменения состояния входов, а только после среза (edge) сигнала синхронизации CLK. Срез может быть положительным ("фронт", posedge) или отрицательным ("спад", negedge). Триггеры, счетчики, регистры являются

последовательными логическими схемами. Таким образом, практически любое современное цифровое устройство представляет собой набор последовательно (или циклично) соединенных схем комбинационной и последовательной логики (Рис.1).

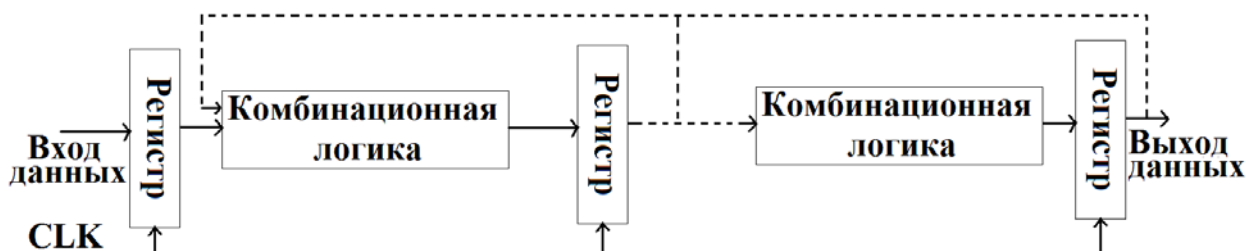


Рис.1. Пример цифрового устройства, состоящего из последовательно соединенных схем комбинационной и последовательной логики.

Входные данные записываются в регистр, далее обрабатываются комбинационной логикой (выполняются арифметические/логические операции) и снова записываются в регистр. Все регистры имеют общий вход сигнала синхронизации CLK. И только после фронта (или спада) этого входа, данные на выходе регистра перезаписываются значениями на входе. Состояние всей схемы определяется текущими значениями в регистрах. Такая реализация устройства исключает использование недостоверных (не установившихся данных) на выходе устройства, а также их изменение во время считывания т.к. значения на выходе регистра могут измениться только после перепада CLK. Как правило, сигнал синхронизации это периодическая последовательность импульсов с длительностью равной половине периода. Чем выше частота сигнала синхронизации, тем больше производительность цифрового устройства, но максимально возможная частота синхронизации ограничена минимально необходимым временем для прохождения сигналов от входа до выхода самой большой комбинационной схемы устройства.

## Логические вентили

Логический вентиль — базовый элемент цифровой схемы, выполняющий элементарную логическую операцию, преобразуя таким образом множество входных логических сигналов в единственный выходной логический сигнал. Логика работы вентиля основана на битовых операциях с входными цифровыми сигналами в качестве операндов. При создании цифровой схемы вентили соединяют между собой, при этом выход используемого вентиля должен быть подключён к одному или к нескольким входам других вентилях. В настоящее время в цифровых устройствах доминируют электронные логические вентили на базе полевых транзисторов, однако в прошлом для создания вентилях использовались и другие устройства, например, электромагнитные реле.

В цифровой электронике логический уровень сигнала представлен в виде уровня напряжения. Обычно (и далее в лабораторных работах) высоким уровнем сигнала или логической «1» считают напряжение питания схемы (на схемах обозначают VCC, VDD), а низким уровнем сигнала или логическим «0» считают напряжение 0 вольт (обычно обозначают VSS, GND). На рисунке 2 представлены наиболее часто используемые логические вентили, их обозначение в схемах и таблицы истинности, в которых показана зависимость выходных значений элементов от входных.

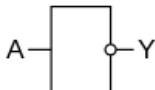
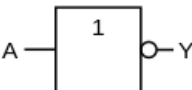
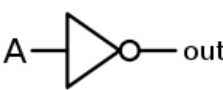


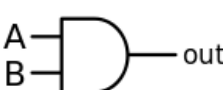
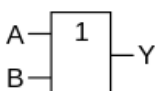
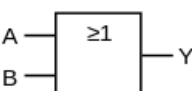
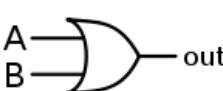
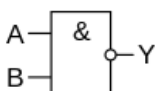

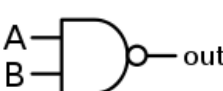
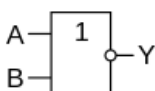
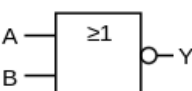
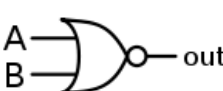
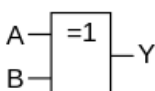
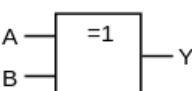

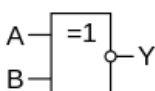
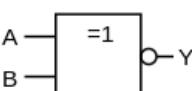
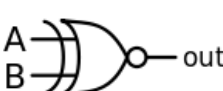
Логический вентиль	Условные графические обозначения			Функция	Таблица истинности															
	ГОСТ 2.743-91	IEC 60617-12 : 1997	US ANSI 91-1984																	
<b>НЕ</b> (англ. NOT gate)				Отрицание $Y = \overline{A}$ $Y = \neg A$ $Y = \tilde{A}$	<table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Y	0	1	1	0									
A	Y																			
0	1																			
1	0																			
<b>И</b> (англ. AND gate)				Конъюнкция $Y = A \wedge B$ $Y = A \cdot B$ $Y = A \& B$ $Y = AB$	<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1
A	B	Y																		
0	0	0																		
0	1	0																		
1	0	0																		
1	1	1																		
<b>ИЛИ</b> (англ. OR gate)				Дизъюнкция $Y = A \vee B$ $Y = A + B$	<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1
A	B	Y																		
0	0	0																		
0	1	1																		
1	0	1																		
1	1	1																		
<b>НЕИ (И-НЕ)</b> (англ. NAND gate) Элемент Шеффера				$Y = \overline{A \wedge B}$ $Y = \overline{A \cdot B}$ $Y = \overline{A \& B}$ $Y = \overline{AB}$ $Y = A B$	<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0
A	B	Y																		
0	0	1																		
0	1	1																		
1	0	1																		
1	1	0																		
<b>НЕИЛИ (ИЛИ-НЕ)</b> (англ. NOR gate) Элемент Пирса				$Y = \overline{A \vee B}$ $Y = \overline{A \vee B}$ $Y = \overline{A + B}$ $Y = A - B$	<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0
A	B	Y																		
0	0	1																		
0	1	0																		
1	0	0																		
1	1	0																		
<b>Исключающее ИЛИ</b> (англ. XOR gate) сложение по модулю 2				Строгая дизъюнкция $Y = A \underline{\vee} B$ $Y = A \oplus B$	<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0
A	B	Y																		
0	0	0																		
0	1	1																		
1	0	1																		
1	1	0																		
<b>Исключающее ИЛИ с инверсией</b> (англ. XNOR gate) равнозначность				Эквиваленция $Y = \overline{A \underline{\vee} B}$ $Y = \overline{A \oplus B}$ $Y = \overline{A \oplus B}$ $Y = A \odot B$	<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	1
A	B	Y																		
0	0	1																		
0	1	0																		
1	0	0																		
1	1	1																		

Рис.2. Логические вентили.

Из представленного на рисунке 2 набора логических вентилей, соединяя их, можно составить схему любой сложности, которая будет выполнять любую операцию над входными данными. Из двухвходовых элементов можно составить элементы с 3,4 или более входами, например, на рис. 3а показан элемент «И» на 4 входа, составленный из трех элементов с двумя входами. Часто на схемах сразу рисуют элементы с нужным количеством входов, как показано на рис. 3б. Входы элементов могут быть соединены между собой, но выходы логических вентилей никогда нельзя соединять, т.к. в случае появления логического «0» на одном выходе и логической «1» на другом получится конфликт выходов. Это приведет к неопределенному значению сигнала в месте соединения, а также может привести к перегреву и выходу из строя устройства.

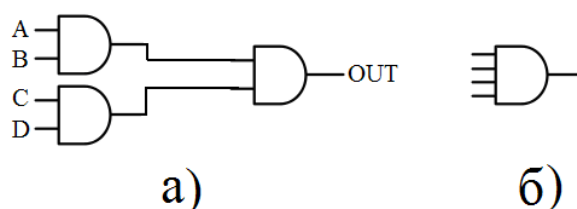


Рис. 3. а) логический элемент «И» на 4 входа, составленный из трех элементов с двумя входами; б) Обозначение в схеме логического элемента «И» на 4 входа.

## Мультиплексор

Для выполнения лабораторных работ стоит также рассмотреть еще одно комбинационное логическое устройство, называемое мультиплексором, предназначенное для управляемой передачи данных от нескольких источников информации в один выходной канал. Согласно определению, мультиплексор должен иметь один выход и две группы входов: информационные и адресные. Код, подаваемый на адресные входы, определяет, какой из информационных входов в данный момент подключен к выходному выводу. Поскольку  $n$ -разрядный двоичный код может принимать  $2^n$  значений, то, если число адресных входов мультиплексора равно  $n$ , число его информационных входов должно равняться  $2^n$ . На рисунке 4 представлено условное графическое обозначение мультиплексора и его таблица истинности.

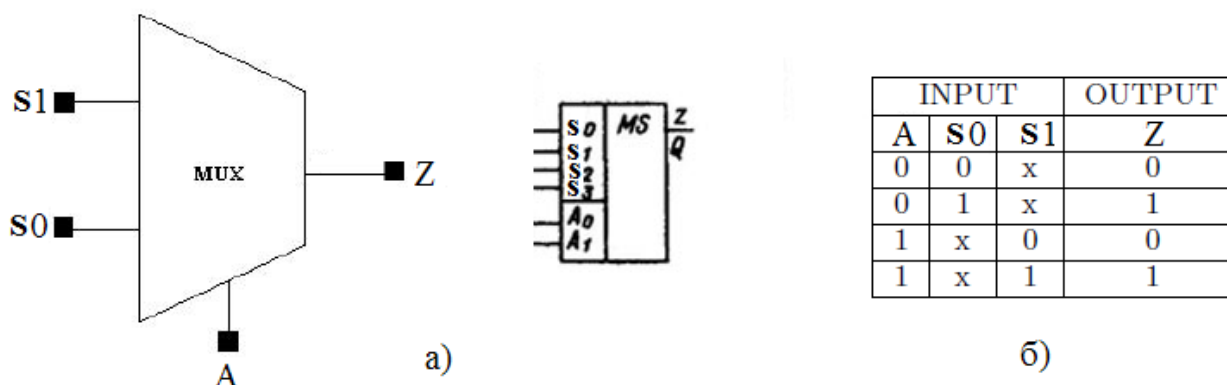


Рис. 4. а) условное графическое обозначение мультиплексора; б) таблица истинности.

В современных КМОП технологиях мультиплексор очень часто применяется и сам по себе и в составе других логических элементов, т.к. очень легко реализуется на МОП транзисторах.

## Триггеры

Триггер (от английского “trigger”) – цифровое устройство, которое может длительно находиться в одном из двух устойчивых состояний и чередовать их под воздействием внешних сигналов. При этом переход из одного состояния в другое осуществляется максимально быстро, временем переходных процессов на практике принято пренебрегать. Триггеры – это основной элемент для построения различных запоминающих устройств и счетчиков. Отличительной особенностью триггера как функционального устройства является свойство запоминания двоичной информации. Под памятью триггера подразумевают способность оставаться в одном из двух состояний и после прекращения действия переключающего сигнала. Приняв одно из состояний за «1», а другое за «0», можно считать, что триггер хранит (помнит) один разряд числа,

записанного в двоичном коде. Триггеры способны сохранять свою память только при наличии питающего напряжения. Именно по этому их принято относить к оперативной памяти. Если выключить питающее напряжение и затем его снова включить, триггер переходит в случайное состояние – он может иметь на выходе как логический ноль, так и логическую единицу. Именно поэтому, проводя проектирование схем, надо обязательно предусмотреть вопрос приведения триггера в начальное («стартовое») состояние.

Число входов у триггера определяется его структурой и выполняемыми функциями. По тому, как информация записывается в триггер, они делятся на асинхронные – информация записывается непрерывно и зависит от информационных сигналов, которые подаются на вход триггера и синхронные – информация записывается только при наличии дополнительного синхронизирующего сигнала, фактически открывающего работу триггера. Исходя из функционального назначения, триггеры классифицируют на: RS-триггеры, JK-триггеры, D-триггер и T-триггеры. Далее рассмотрим устройство и работу D и T триггеров. RS и их модификацию – JK триггеры не будем рассматривать, т.к. в цифровой технике асинхронные RS-триггеры практически не используются, а другие типы на их основе в современных КМОП технологиях не создаются, ввиду громоздкости схемы. Наиболее часто используемые D и T триггеры в современных КМОП технологиях делаются на основе «аналогового переключателя», который по сути представляет собой мультиплексор, описанный выше.

## D-триггер

Триггер задержки, который используют для создания регистров сдвига и регистров хранения, неотъемлемая часть любого микропроцессора. Схема показана на рис. 5.

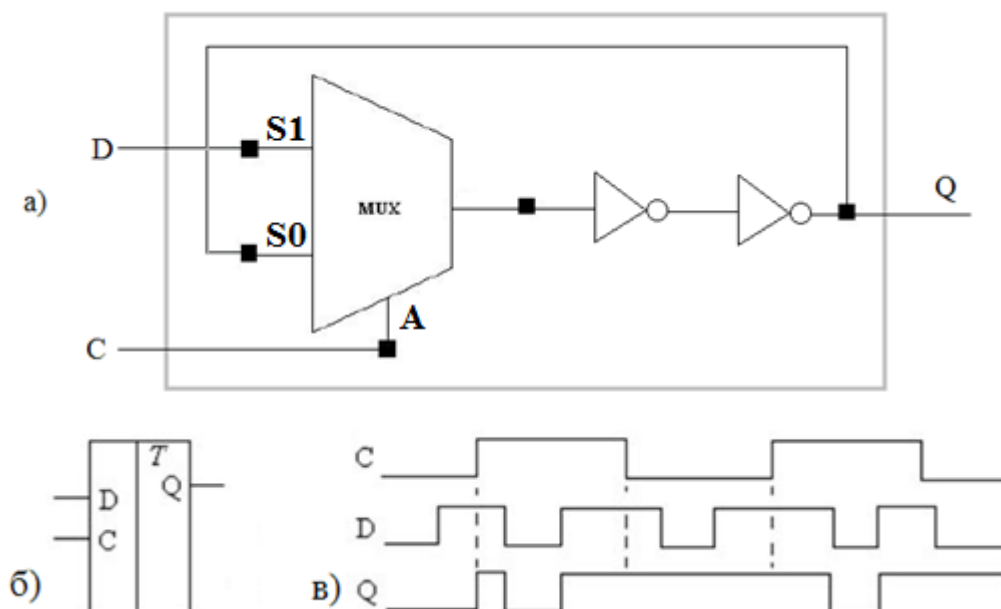


Рис. 5. D-триггер. а) Схема D-триггера; б) схематическое изображение D-триггера; в) временная диаграмма работы D-триггера.

D-триггер – неотъемлемая часть любого запоминающего устройства, регистра, счетчика в современных интегральных схемах. Они могут составлять половину или даже большую часть всех элементов цифрового устройства. В современных КМОП технологиях D-триггер делают из мультиплексора и буфера (обычно два последовательно

соединенных логических элемента НЕ), как показано на рисунке 5а. D-триггер имеет два входа – информационный D и синхронизации C. При «0» на входе C, выход Q определяется своим предыдущим состоянием, а при логической «1» на C – на выход Q проходит сигнал с входа D. Работа D-триггера соответствует таблице истинности, описанной в таблице 1. На временной диаграмме (рис. 5в) приведен принцип работы D-триггера.

Входы		Выходы
C	D	Q
0	0	Определяется предыдущим состоянием
0	1	Определяется предыдущим состоянием
1	0	0
1	1	1

Таблица 1. Таблица истинности D-триггера.

Рассмотренный D-триггер по принципу его работы еще называют «защелкой» (flip-flop). В реальных же цифровых устройствах часто используют D-триггер с динамическим сигналом синхронизации, который передает значение входа D на выход Q не постоянно, когда сигнал синхронизации C = «1», а только в момент изменения синхросигнала, по фронту (нарастанию) или же по спаду. Такой триггер легко составить из двух «защелок», одна из которых с инверсным входом сигнала синхронизации. На рисунках 6а, 6б показаны схемы, условное обозначение и временные диаграммы D-триггеров, работающих по фронту и по спаду импульса сигнала синхронизации. Также на рисунке 6в показана схема довольно часто применяющегося D-триггера с входом разрешения переключения CE.

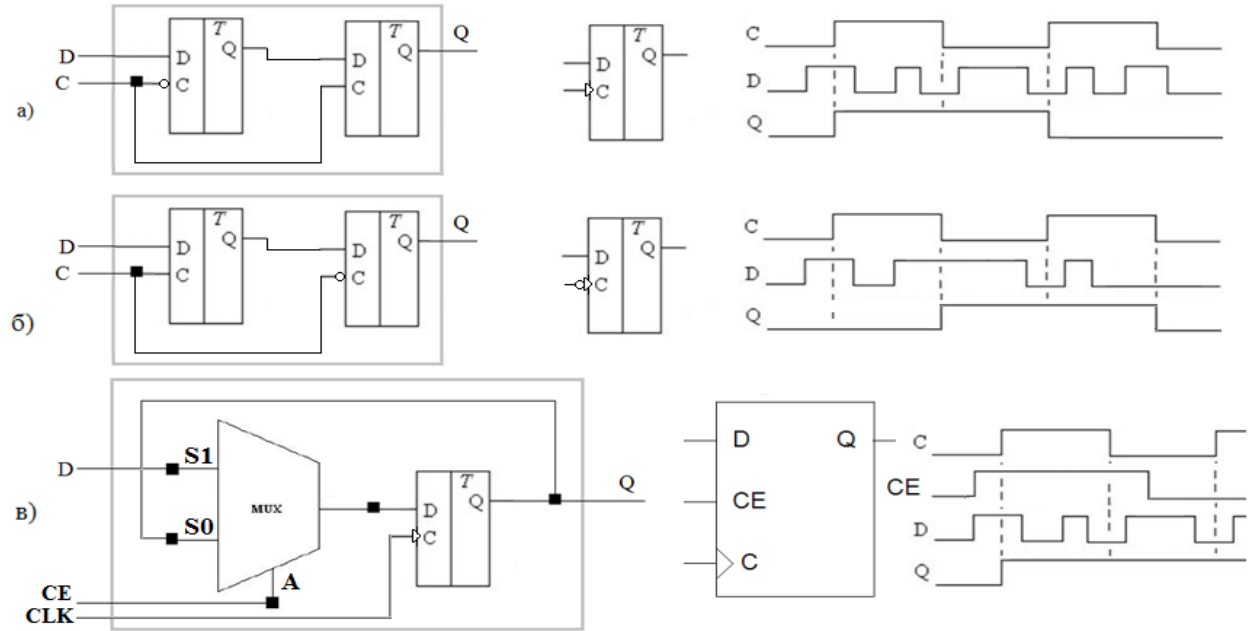


Рис. 6. а) D-триггер, работающий по фронту сигнала синхронизации; б) D-триггер, работающий по спаду сигнала синхронизации. Слева показаны схемы, посередине условное обозначение в схемах, справа временные диаграммы, демонстрирующие работу триггеров. в) D-триггер с входом разрешения переключения CE.

На рис. 7 показана схема реального промышленного триггера на МОП транзисторах, который используется для разработки цифровых устройств на технологии КМОП 65 нм. Схема немного усложнена по сравнению с 5а, но принцип работы тот же.



Оранжевыми кругами обведены инверторы (логические элементы НЕ), желтыми – мультиплексоры. Вход сигнала синхронизации управляет мультиплексорами, пропуская на выход Q либо входной сигнал D, либо предыдущее значение Q. У триггера, показанного на рисунке 7, имеется также инверсный выход QN. Вдаваться в подробности реализации устройства на МОП транзисторах сейчас не будем.

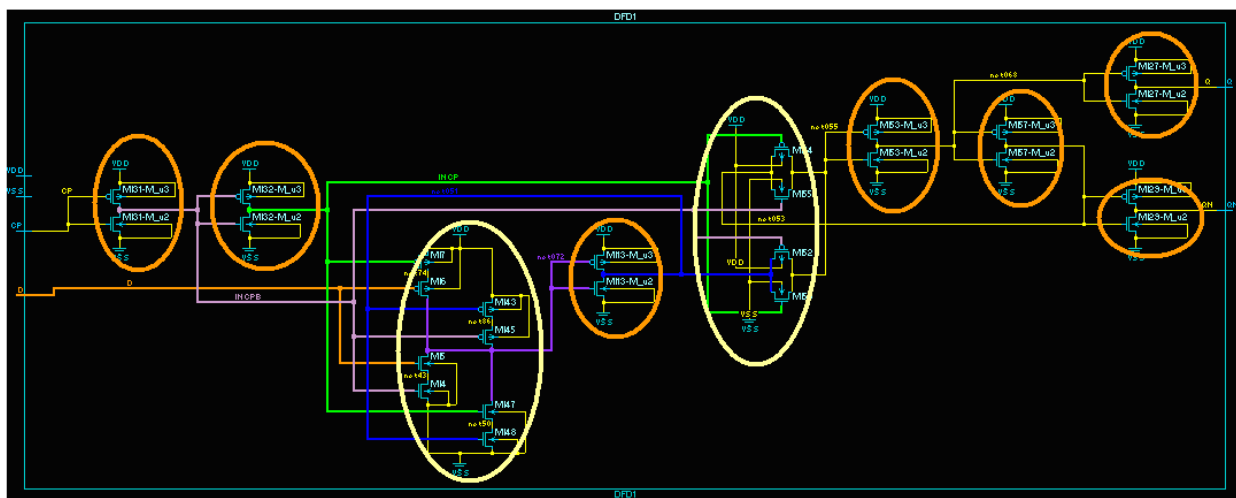


Рис. 7. Реальный промышленный D-триггер на МОП транзисторах.

## Т-триггер

Т-триггер — это счетный триггер. Принцип работы Т-триггера заключается в следующем. После поступления на вход Т импульса, состояние триггера меняется на прямо противоположное. Счётным он называется потому, что Т триггер как бы подсчитывает количество импульсов, поступивших на его вход. Жаль только, что считать этот триггер умеет только до одного. При поступлении второго импульса Т-триггер снова сбрасывается в исходное состояние.

Т триггер можно построить из D триггера. Для того чтобы превратить D триггер в счётный, необходимо ввести цепь обратной связи с инверсного выхода этого триггера на вход, как показано на рисунке 8а.

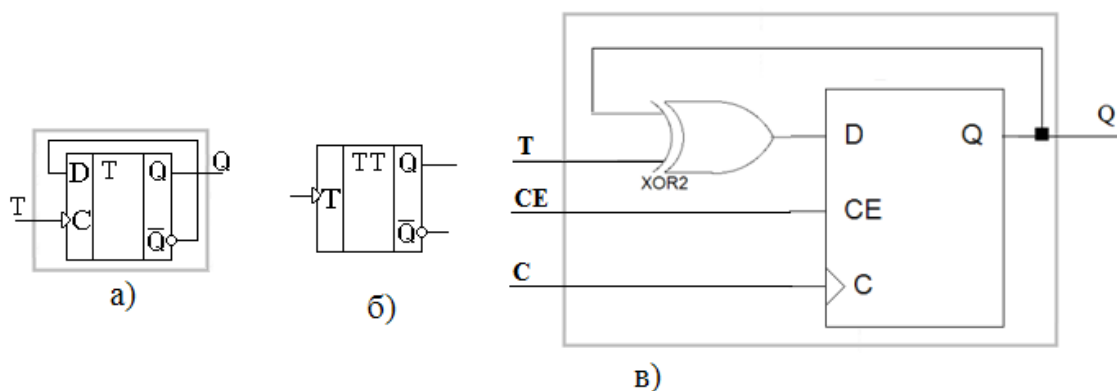


Рис. 8. Т-триггер. а) Схема Т триггера, построенная на основе D триггера; б) Условно-графическое обозначение Т-триггера; в) Схема синхронного Т-триггера.

При разработке схем синхронных двоичных счетчиков один Т-триггер соответствует одному разряду считаемого числа. Важно осуществлять одновременную

запись во все триггеры счетчика. В этом случае вход Т триггера служит только для разрешения изменения состояния на противоположное (в счетчиках – для разрешения переноса в старший разряд при заполни младших единицами), а синхронизация производится входом "С". Подобная схема Т-триггера с входом разрешения счета СЕ (в счетчиках параллельно подключается ко всем триггерам) приведена на рисунке 8в. Т-триггер состоит из D-триггера с входом разрешения переключения СЕ и логического элемента XOR2 (исключающее или), который выполняет функцию управляемого инвертора. При  $T=0$  сигнал на его выходе повторяет Q, а при  $T=1$  – инвертирует Q. Таким образом, Т-триггер при  $CE=1$  по фронту сигнала синхронизации переключается всегда, но при  $T=1$  в противоположное состояние, а при  $T=0$  в предыдущее состояние, т.е. не переключается. Временная диаграмма Т триггера приведена на рисунке 9. При построении этой временной диаграммы был использован D-триггер, работающий по спаду синхронизирующего сигнала.

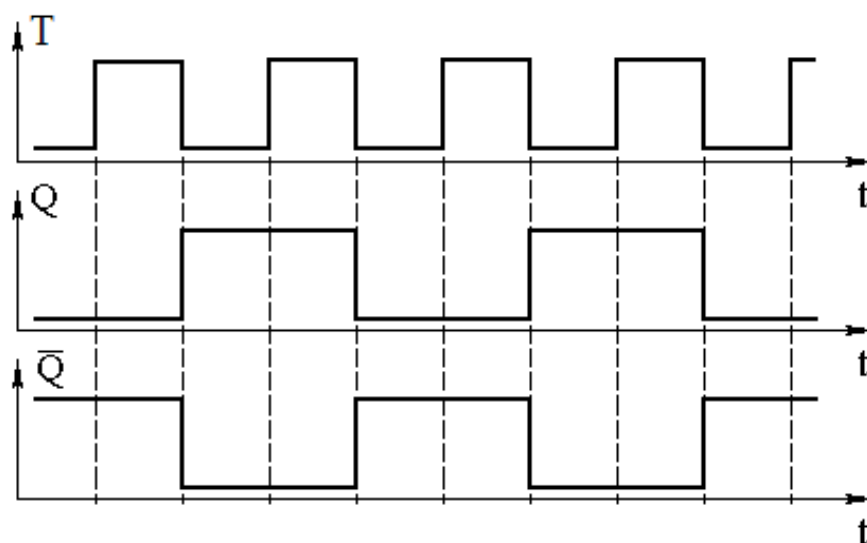


Рис. 9. Временная диаграмма Т триггера.

Т-триггеры широко применяются в схемах деления и умножения частоты. Без них была бы невозможна реализация синтезаторов частот, которые применяются в качестве задающих генераторов в передатчиках и гетеродинах в приемниках раций, мобильных телефонов и т.д. Не менее важна роль Т-триггеров в формировании сигналов синхронизации цифровых микросхем, таких как центральные процессоры компьютеров.

В лабораторных работах для создания счетчиков применяются Т-триггеры FTCE с входом **CE** (Clock Enable) и входом **CLR** асинхронного сброса в 0. Схема FTCE триггера, сделанного на основе FDCE триггера (синхронный D-триггер с входом CE, CLR) показана на рисунке 10. Особенность асинхронного входа CLR состоит в том, что сброс в 0 происходит не по фронту, а по уровню  $CLR=1$  независимо от наличия сигнала синхронизации CLK и уровней сигналов на других входах CE и T.

Т - это вход разрешения переключения. Если  $CE=1$ ,  $CLR=0$  и  $T=1$ , то по фронту сигнала синхронизации CLK триггер переключается в противоположное состояние.

( $Q \leq !Q$ ). При  $T=0$  триггер не переключается фронту CLK ( $Q \leq Q$ ).

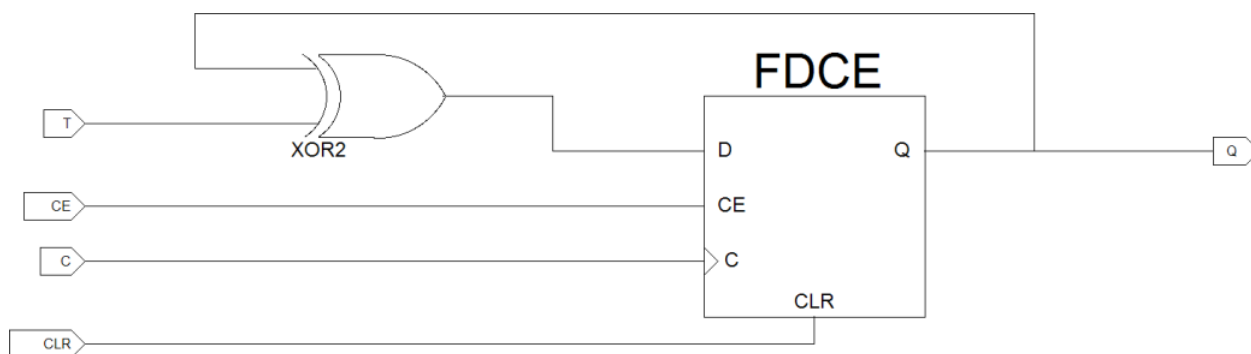


Рис. 10. Схема FTCE триггера.

## **Программируемые логические интегральные схемы (ПЛИС) и Look-Up Tables (LUT)**

Программируемая логическая интегральная схема (ПЛИС, англ. programmable logic device (PLD) или field programmable gate array (FPGA)) — электронный компонент, используемый для создания цифровых интегральных схем. В отличие от обычных цифровых микросхем, логика работы ПЛИС не определяется при изготовлении, а задаётся посредством программирования (проектирования). Для программирования используются программатор и IDE (отладочная среда проектирования), позволяющие задать желаемую структуру цифрового устройства в виде принципиальной электрической схемы или программы на специальных языках описания аппаратуры: Verilog, VHDL, AHDL и др.

ПЛИС широко используется для построения различных по сложности и по возможностям цифровых устройств, например, предназначенных для проектирования и прототипирования интегральных схем специального назначения, т.к. при мелкосерийном и единичном производстве изготовление отдельной микросхемы существенно дороже. Также ПЛИС используются в качестве программируемых логических контроллеров. Некоторые производители для своих ПЛИС предлагают программные процессоры, которые можно модифицировать под конкретную задачу, а затем встроить в ПЛИС. Тем самым обеспечивается увеличение свободного места на печатной плате, упрощается проектирование самой ПЛИС и увеличивается быстродействие ПЛИС. Современные микросхемы ПЛИС имеют более 1000 выводов.

Look-Up Table (LUT) — неотъемлемая часть любой ПЛИС, это таблицы соответствия, по сути, маленький блок памяти, в котором записана таблица истинности нужного нам устройства. Входы задают адрес, а на выходе получаем однобитные данные. Вычисление функций заменяется извлечением значения из таблиц. LUT используют, чтобы обеспечить аппаратную реконфигурируемость. Например, в лабораторных макетах Digilent NEXYS2 Board установлена ПЛИС Spartan3, которая содержит 4х-входовые LUT'ы. Каждый LUT может реализовать любую логическую функцию. То есть это по желанию может быть OR4, или AND4, или NOR4, или какая-нибудь более сложная функция, например  $Q = (A \& B) \mid (\sim C \& D)$ .

Таким образом, имея одну микросхему ПЛИС с набором множества LUT таблиц, сконфигурировав LUT таблицы, то есть, записав в них таблицы истинности нужных нам функций, мы получаем цифровое устройство, которое работает согласно записанной в него логике. По сути, с помощью LUT таблиц получают универсальное аппаратное устройство, которое может работать как угодно, согласно записанной в него логической схеме. Подробнее о работе ПЛИС Spartan3E описано в разделе «Лабораторный макет».

## Язык Verilog

Прежде, чем приступить к выполнению лабораторных работ, немного ознакомимся с синтаксисом языка формального описания цифровых схем Verilog. Этот язык используется для проектирования логики интегральных микросхем ASIC и так же ПЛИС. В настоящем методическом пособии вы ознакомитесь с основами языка Verilog, но если Вы всерьез займетесь проектированием цифровых микросхем, вам придется прочитать множество книг, в том числе на английском языке.

### Базовые типы источников сигналов

Сигналы (signal) – это электрические импульсы, которые передаются по проводникам (wires) между логическими элементами схемы. Проводники переносят информацию не производя над ней никаких вычислений. В цифровой схеме сигналы важны для передачи двоичных данных.

Один из базовых типов источника сигнала в языке Verilog – это цепь или проводник – **wire**. Таким образом, если у вас есть арифметическое или логическое выражение, вы можете ассоциировать результат выражения с именованным проводником и позже использовать его в других выражениях. Это немного похоже на переменные, только их (как провода в схеме) нельзя пересоединить на лету, нельзя поменять назначение. Значение проводника (wire) – это функция того, что присоединено к нему. Вот пример декларации однобитного проводника, написанной на языке Verilog:

```
wire a;
```

Вы можете ему назначить другой сигнал, скажем сигнал “b”, вот так:

```
wire b;  
assign a = b;
```

Или вы можете определить сигнал и сделать назначение ему одновременно в одном выражении:

```
wire a = b;
```

У вас могут быть проводники, передающие несколько битов (шины):

```
wire [3:0] c; //это четыре провода
```

Проводники передающие несколько битов информации называются “шина”, иногда “вектор”. Назначения к ним делаются так же:

```
wire [3:0] d;  
assign c = d; //“подключение” одной шины к другой
```

Количество проводников в шине определяется любыми двумя целыми числами разделенными двоеточием внутри квадратных скобок.

```
wire [11:4] e; //восьмибитная шина  
wire [255:0] f; //256-ти битная шина
```

Из шины можно выбрать некоторые нужные биты и назначить другому проводу:

```
wire g;
assign g = f[2]; //назначить сигналу “g” второй бит шины “f”
```

Кроме того, выбираемый из шины бит может определяться переменной:

```
wire [7:0] h;
wire i = f[h]; // назначить сигналу “i” бит номер “h” из шины “f”
```

Вы можете выбрать из сигнальной шины некоторый диапазон битов и назначить другой шине с тем же количеством битов:

```
wire [3:0] j = e[7:4];
```

Существует другой тип источника сигнала называемый регистр: **reg**. Регистр reg в языке Verilog скорее обозначает переменную, которая может хранить значение, чем аппаратный регистр. Если регистру постоянно присваивается значение комбинаторной (логической) функции, то он ведет себя точно как проводник (wire). Если же регистру присваивается значение в синхронной логике, например по фронту сигнала тактовой частоты, то ему, в конечном счете, будет соответствовать физический триггер или группа триггеров. Регистры описываются так же, как и проводники:

```
reg [3:0] m;
reg [0:100] n;
```

Вы можете определить массив регистров, которые обычно называют “память”:

```
reg [7:0] q [0:15]; //память из 16 слов, каждое по 8 бит
```

Еще один тип источника сигнала это постоянные сигналы или просто числа:

```
wire [12:0] s = 12; /* 32-х битное десятичное число, которое будет “обрезано” до 13 бит */
wire [12:0] z = 13'd12; //13-ти битное десятичное число
wire [3:0] t = 4'b0101; //4-х битное двоичное число
wire [7:0] q = 8'hA5; // 8-ми битное шестнадцатеричное число A5
wire [63:0] u = 64'hdeadbeefcafebab; /*64-х битное шестнадцатеричное число */
```

Если точно не определить размер числа, то оно принимается по умолчанию 32-х разрядным. Это может быть проблемой при присвоении сигналам с большей или меньшей разрядностью. Числа – это числа. Они могут использоваться во всяких арифметических и логических выражениях. Например, можно прибавить 1 к вектору “aa”:

```
wire [3:0] aa;
wire [3:0] bb;
assign bb = aa + 1;
```

Такое выражение сложения в коде на Verilog в конечном счете обернется аппаратным сумматором внутри чипа.

Для универсальности разрабатываемых устройств также часто используют глобальные параметры. Например, удобно задавать разрядность шин данных не конкретным значением, а используя параметры. Синтаксис следующий:

```
`define m 10  
wire [m:0] a;
```

Функционально это будет то же самое, что объявить шину **wire [10:0] a**.

## Иерархия проекта

Verilog позволяет группировать логику в блоки. Каждый блок логики называется “модулем” (module). Модули имеют входы и выходы, которые ведут себя как сигналы wire или reg.

Синтаксис описания модуля в языке Verilog такой:

```
module my_module //имя модуля  
(  
    input port_a, //описание входов и выходов  
    output wire[6:0]port_b, //если тип сигнала (wire/reg)не указан, то по  
    умолчанию применяется wire.  
    input [0:4]w,  
    inout wire y,  
    output reg [3:0]z  
);  
    //Далее описывается работа модуля (арифметические, логические операции,  
    присвоения...)  
  
    wire r = w[1];  
    assign port_b = 7'b0101001;  
  
    //В конце описания логики каждого модуля пишем ключевое слово endmodul  
endmodule
```

В проекте, особенно сложном, бывает много модулей, соединенных между собой. Прежде всего, нужно заметить, что обычно в проекте всегда есть один модуль самого верхнего уровня (top level). Он состоит из нескольких других модулей. Те в свою очередь могут содержать еще модули и так далее. Не обязательно, чтобы все модули были написаны на одном языке описания аппаратуры. Совсем наоборот. Довольно удобно и наглядно иметь модуль самого верхнего уровня выполненным в виде схемы, состоящей из модулей более низкого уровня. Эти модули могут быть написаны разными людьми, на разных языках (Verilog, VHDL, AHDL, и даже выполнены в виде схемы). Итак, внутри тела любого модуля, можно объявлять экземпляры других модулей и потом соединять их друг с другом проводниками. Порядок описания экземпляра модуля в Verilog такой: пишем название модуля, тип которого нам нужен, пишем название конкретно этого экземпляра модуля, описываем подключения сигналов: точка и затем имя сигнала модуля, затем в скобках имя проводника, который сюда подключен.

Например, на рисунке 3а изображен 4х-входовый логический элемент «И», состоящий из трех двухвходовых. На Verilog эта схема будет выглядеть так:

```
module AND2( input IN1, output wire OUT,  
            input IN2);  
    assign OUT = IN1 & IN2;  
endmodule
```

```

module AND4 (    input A,    output wire OUT,
                input B,
                input C,
                input D);
    wire out_1, out_2;
    AND2 inst1_and2( .IN1 (A),    .OUT (out_1),
                    .IN2 (B) );
    AND2 inst2_and2 ( .IN1 (C),    .OUT (out_2),
                    .IN2 (D) );
    AND2 inst3_and2( .IN1 (out_1), .OUT (OUT),
                    .IN2 (out_2) );
endmodule

```

## Арифметические и логические функции

Сейчас, мы уже знаем про модули, их входные и выходные сигналы и как они могут быть соединены друг с другом. Но нужно ли каждый раз, когда мы хотим, например, сложить два числа, делать такие сложные модули? Конечно нет! В этом разделе познакомимся с основными арифметическими и логическими операторами языка Verilog. С помощью комбинаторной логики посчитаем некоторые арифметические и логические функции, а синтезатор, встроенный в среду разработки сам автоматически преобразует сложные выражения, написанные на Verilog в схему, состоящую из элементарных логических вентилей, которую можно реализовать на физической аппаратуре.

### Сложение и вычитание

Вот пример модуля, который одновременно и складывает и вычитает два числа. Здесь входные операнды у нас 8-ми битные, а результат 9-ти битный. Verilog корректно сгенерирует бит переноса (carry bit) и поместит его в девятый бит выходного результата. С точки зрения Verilog входные операнды беззнаковые. Если нужна знаковая арифметика, то об этом нужно отдельно позаботиться.

```

module simple_add_sub(
    //два входных 8-ми битных операнда
    input [7:0] operandA,    output wire [8:0] out_sum,
    input [7:0] operandB,    output wire [8:0] out_dif ); /*Выходы для
    арифметических операций имеют дополнительный 9-й бит
    переполнения*/
    assign out_sum = operandA + operandB; //сложение
    assign out_dif = operandA - operandB; //вычитание
endmodule

```

### Логический и арифметический сдвиг

Сдвиг, при котором уходящий бит исчезает, не влияя на оставшиеся биты, а на месте появившегося бита записывается бит 0, называют логическим сдвигом. Рассмотрим пример работы операции сдвига. Пусть у нас есть число 10101010b (в двоичной системе).



Если сделать сдвиг влево на 1 бит, то получим число 01010100b. Если сделать сдвиг исходного числа вправо на 1 бит, то получим число 01010101b.

При арифметическом сдвиге слово рассматривается не просто как группа битов, а как целое число в дополнительном коде. При сдвиге влево ведёт себя как логический сдвиг, при сдвиге вправо уходящий бит исчезает, не влияя на оставшиеся биты, а на месте появившегося бита устанавливается бит, соответствующий знаку. Пример работы операции сдвига: Пусть у нас есть число 11111010b = -6 (в двоичной системе). Если сделать сдвиг влево на 1 бит, то получим число 11110100b = -12. Если сделать сдвиг исходного числа вправо на 1 бит, то получим число 11111101b = -3. Легко заметить, что арифметический сдвиг влево соответствует умножению на 2, а сдвиг вправо — делению на 2. Операции сдвига очень часто используются, т.к. легко реализуются в аппаратуре и быстро выполняются в процессе работы устройства.

Вот пример модуля, который выполняет сдвиги. В нашем примере результат для сдвига влево 16-ти битный. Если сдвигать влево или вправо слишком далеко, то результат получится просто ноль.

```
module simple_shift (  
  // два входных 8-ми битных операнда  
  input [7:0] operandA,      output wire [15:0] out_shl,  
  input [7:0] operandB,      output wire [7:0] out_shr,  
                              output wire [7:0] out_sar); /*логический сдвиг  
влево operandA на количество бит, указанное в operandB*/  
  
  assign out_shl = operandA << operandB;  
  /* В этом примере на сколько сдвигать определяется 3-мя битами  
второго операнда */  
  assign out_shr = operandA >> operandB[2:0];  
  
  //арифметический сдвиг вправо (сохранение знака числа)  
  assign out_sar = operandA >>> operandB[2:0];  
endmodule
```

## Битовые логические операции

Битовые операции в Verilog выглядят так же, как и в языке C. Каждый бит результата вычисляется отдельно соответственно битам операндов. Вот пример:

```
module simple_bit_logic (  
  //два входных 8-ми битных операнда  
  input [7:0] operandA,      output wire[7:0] out_bit_and,  
  input [7:0] operandB,      output wire[7:0] out_bit_or,  
                              output wire[7:0] out_bit_xor,  
                              output wire[7:0] out_bit_not);/*Выходы для  
битовых (bit-wise) логических операций*/  
  assign out_bit_and = operandA & operandB; //И  
  assign out_bit_or = operandA | operandB; //ИЛИ  
  assign out_bit_xor = operandA ^ operandB; //исключающее ИЛИ  
  assign out_bit_not = ~operandA; //НЕ  
endmodule
```



## Булевы логические операции

Булевы логические операторы отличаются от битовых операций. Так же, как и в языке C, здесь значение всей шины рассматривается как ИСТИНА если хотя бы один бит в шине не ноль или ЛОЖЬ, если все биты шины – ноль. Результат получается всегда однобитный (независимо от разрядности операндов) и его значение "1" (ИСТИНА) или "0" (ЛОЖЬ).

```
module simple_bool_logic (  
    //два входных 8-ми битных операнда  
    input [7:0] operandA,    output wire[7:0] out__bool_and,  
    input [7:0] operandB,    output wire[7:0] out__bool_or,  
                                output wire[7:0] out__bool_xor,  
                                output wire[7:0] out__bool_not); /*Выходы для  
    битовых (boolean) логических операций*/  
    assign out__bool_and = operandA && operandB;    //И  
    assign out__bool_or = operandA || operandB;    //ИЛИ  
    assign out__bool_not = !operandA;    //НЕ  
endmodule
```

## Операторы редукции

Verilog имеет операторы редукции. Эти операторы позволяют выполнять операции между битами внутри одной шины. Так, можно определить все ли биты в шине равны единице (&bus), есть ли в шине хотя бы одна единица (|bus) или четность числа единиц в шине (^bus). Рассмотрим пример:

```
module simple_reduction_logic (  
    input [7:0] operandA, //входной 8-ми битный операнд  
                                output wire out_reduction_and,  
                                output wire out_reduction_or,  
                                output wire out_reduction_xor); /* Выходы для  
    логических операций редукции*/  
    assign out_reduction_or = |operandA;  
    assign out_reduction_and = &operandA;  
    assign out_reduction_xor = ^operandA;  
endmodule
```

А вот еще полезные операторы редукции:

~|operandA обозначает, что в шине нет единиц.

~&operandA обозначает, что некоторые биты в шине равны нулю.

## Оператор условного выбора

Язык C имеет оператор "? :". С его помощью можно выбрать одно значение из двух по результату логического выражения. В Verilog тоже есть подобный оператор. Он фактически реализует мультиплексор. В данном примере на выходе мультиплексора окажется значение operandA если сигнал sel\_in единица. И наоборот. Если входной сигнал sel\_in равен нулю, то на выходе мультиплексора будет значение operandB.

```

module simple_mux (
    //входные 8-ми битные операнды
    input [7:0] operandA, output wire [7:0]out_mux, /*Выход
мультиплексора*/
    input [7:0] operandB,
    input sel_in); //входной сигнал селектора

    assign out_mux = sel_in ? operandA : operandB;
endmodule

```

## Операторы сравнения

В Verilog можно ли сравнивать "числа" (а точнее значения регистров или шин). Ниже приведен пример (все сравнения происходят с беззнаковыми числами):

```

module simple_compare (
    //входные 8-ми битные операнды
    input [7:0] operandA,
    input [7:0] operandB,
    output wire out_eq,
    output wire out_ne,
    output wire out_gt,
    output wire out_lt,
    output wire out_ge,
    output wire out_le); /*Выходы операций
сравнения*/
    assign out_eq = operandA == operandB; //равно
    assign out_ne = operandA != operandB; //не равно
    assign out_ge = operandA >= operandB; //больше или равно
    assign out_le = operandA <= operandB; //меньше или равно
    assign out_gt = operandA > operandB; //больше
    assign out_lt = operandA < operandB; //меньше
endmodule

```

## «always» блоки. Блокирующие и синхронные присвоения.

Мы уже познакомились с постоянным назначением сигналов с помощью **assign**. Фактически для описания любой комбинационной логической схемы этого достаточно. А для описания схем последовательностной логики язык Verilog имеет так называемые «always» блоки. Они позволяют выразить алгоритм так, чтобы он выглядел как последовательность действий. Для описания «always» блока используется вот такой синтаксис:

```

always @( <sensitivity_list> ) begin
    <statements>
end

```

**<sensitivity\_list>** – это список всех входных сигналов, к которым чувствителен блок. Это список входных сигналов, изменение которых влияет выходные сигналы этого блока.

«always» переводится как «всегда». Такую запись можно прочитать вот так: "Всегда выполнять выражения <statements> при изменении сигналов, описанных в списке чувствительности <sensitivity list>". В списке чувствительности имена входных сигналов разделяются ключевым словом "or":

```
always @(a or b or d) begin
    <statements>
end
```

Иногда гораздо проще и надежней включать в список чувствительности все сигналы. Это делается вот так:

```
always @* begin
    <statements>
end
```

Тогда исправляя выражения в <statements> вам не нужно задумываться об изменении списка чувствительности.

При описании выражений внутри «always» блоков, с правой стороны от знака равенства, как и раньше, можно использовать типы сигналов wire или reg, а вот с левой стороны используется только тип reg. Пример:

```
wire [3:0] a;
wire [3:0] b;
wire [3:0] c;
reg [3:0] d;
reg [3:0] e;
reg [3:0] f;
always @(a or b or c) begin
    d <= выражение, использующее входные сигналы a,b,c ;
    j <= (a + b) & c;
    f <= c[1] ? (a & b) : (a + b);
end
```

Обратите внимание, здесь мы делаем присвоение регистрам с помощью оператора "<=", который называется "не блокирующим" или синхронным присвоением. Обычно, при описании последовательностной логики используют именно синхронное присвоение в always блоках. Но синтаксис языка Verilog позволяет использовать и "блокирующее присвоение" с помощью оператора "=". Разберемся в чем разница между блокирующим и синхронным присвоением.

Блокирующее присвоение (с помощью оператора "=") называется так потому, что вычисления производятся строго в порядке, описанном в always блоке. Второе выражение вычисляется только после первого и результат второго выражения может зависеть от результата первого.

Синхронное присвоение в always блоке (оператор "<=") обозначает факт одновременного запоминания вычисленных значений в соответствующих регистрах. Выражения в always блоке выполняются не последовательно, а одновременно. И присвоение, тоже произойдет одновременно. Рассмотрим пример:

```
module test1(
    input wire clock,          output reg [3:0]x,
```

```

input wire [3:0]in,          output reg [3:0]y,
                             output reg [3:0]z );

always @(posedge clock) begin
    x <= in + 1;
    y <= x + 1;
    z <= y + 1;
end
endmodule

```

Предположим исходное состояние триггеров неизвестно (значение X). Пусть, например, входное значение шины in равно 3 и модуль тактируется частотой clock. Если такой модуль загрузить в симулятор, то получим следующую временную диаграмму:

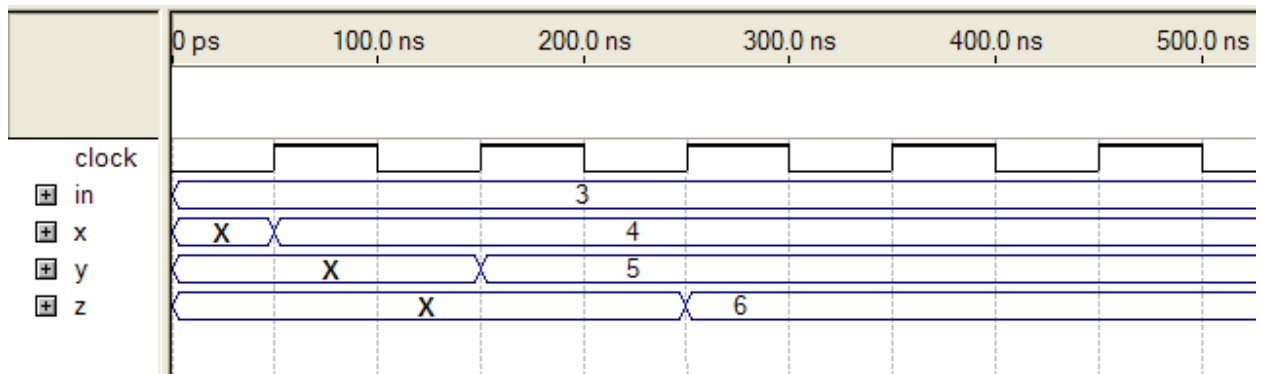


Рис. 11. Временная диаграмма - результат симуляции модуля test1.

Вместе с первым фронтом clock в регистр x будет записано значение суммы in+1 и оно равно четырем. В этот же момент времени в регистр y должна быть записана сумма x+1, но симулятор еще не знает текущего значения регистра x. Оно еще не определено. Только второй фронт clock запишет в регистр y число 5. Присвоение в регистр z так же происходит синхронно с остальными присвоениями. Но только на третьем такте в регистр z запишется число 6, так как только к этому времени симулятору будет известно значение регистра y. В реальной схеме все работает точно так же. Сигнал на выходе z получается задержанным на 2 такта относительно выхода x. Попробуем теперь изменить наш модуль. Сейчас мы будем использовать блокирующее присвоение:

```

module test2(
    input wire clock,    output reg [3:0]x,
    input wire [3:0]in,  output reg [3:0]y,
                        output reg [3:0]z);

always @(posedge clock) begin
    x = in + 1;
    y = x + 1;
    z = y + 1;
end
endmodule

```

Загрузим модуль в симулятор, и получим его временную диаграмму:

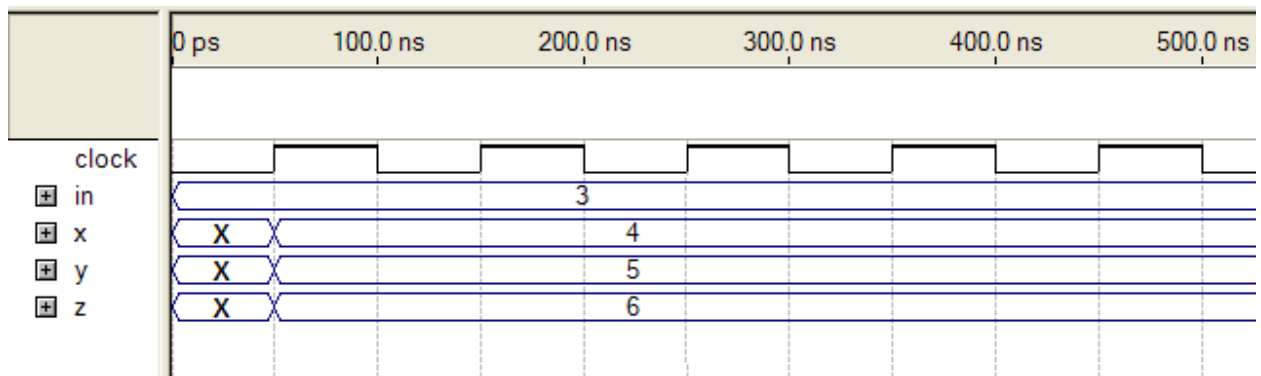


Рис. 12. Временная диаграмма - результат симуляции модуля test2.

Поведение модуля стало принципиально другим! И это неудивительно, ведь этот код описывает теперь уже другую цифровую схему. Поскольку мы использовали блокирующие присвоения, то это обозначает, что следующее выражение учитывает результат предыдущего (в порядке написания). Фактически в этом случае наша запись подразумевает следующее:  $x = in + 1$ ;  $y = in + 2$ ;  $z = in + 3$ . В регистры будут записаны суммы одновременно уже в первом такте.

Попробуем теперь изменить порядок строк наших выражений в always блоке. Синхронное присвоение даже не будем пробовать – результат не изменится, т.к. все выражения вычисляются одновременно не зависимо от того, в каком порядке они написаны в Verilog. А вот с блокирующими присвоениями не так. Тут важна очередность выражений. Поменяем местами две строки, вот так:

```
always @(posedge clock) begin
    x = in + 1;
    z = y + 1;
    y = x + 1;
end
```

Снова загрузим модуль в симулятор, и получим его временную диаграмму:

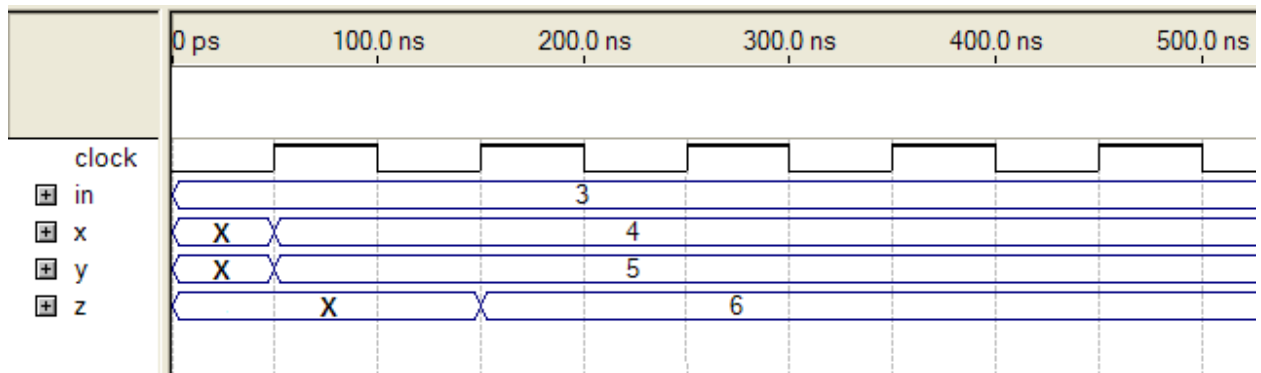


Рис. 13. Временная диаграмма - результат симуляции модуля test2 с измененным порядком присвоения регистров x, y, z в always блоке.

В регистр z будет помещено результирующая сумма на такт позже, чем в регистр y. И это правильно! По первому фронту тактовой частоты может быть вычислено точно только x и y, но нельзя вычислить z. К моменту вычисления z по первому фронту значение y не определено. А вот на втором фронте сигнала clock уже z будет посчитан.

Таким образом, мы выяснили, что Блокирующие и синхронные присвоения действуют по-разному. Синхронные присвоения " $=$ " в языке Verilog позволяют легче

описать многочисленные одновременные присвоения регистрам и учесть все задержки распространения данных внутри цифровой схемы.

## **Физическое проектирование устройств и ограничения Verilog**

Среда разработки цифровых устройств на ПЛИС как правило состоит из редактора Verilog, синтезатора, симулятора и программы для прошивки конфигурации разработанного устройства в ПЛИС. Все модули в проекте, которые написаны на Verilog оптимизируются и преобразуются синтезатором в реальную электрическую схему устройства. Синтезатор в своей работе использует библиотеку элементов, составленную производителем ПЛИС, в которой описано, какие элементы (логические вентили, триггеры или более сложные интегрированные в современные ПЛИС устройства) и в каком количестве можно реализовать на выбранной ПЛИС. Таким образом синтезатор преобразует код Verilog так, чтобы получилась схема, содержащая только имеющиеся в библиотеке элементов устройства, доступные для использования в ПЛИС. Следующим этапом выполняется «размещение и трассировка» элементов. То есть резервируются физически доступные в ПЛИС логические вентили, триггеры, соединяются между собой проводниками. В результате получается конфигурационный файл (по сути, электрическая схема устройства), который необходимо загрузить в ПЛИС, чтобы получилось работающее устройство, логика работы которого была описана в Verilog.

В приведенных выше примерах были рассмотрены основные арифметические и логические операторы языка Verilog. Конечно, кое-что упущено в этом описании, но базовые операции рассмотрены. Вы можете заметить, что некоторые операторы совсем не описаны. Это такие операторы, как умножение (\*) или деление (/), или какой-нибудь модуль, остаток от деления (%). Такие вещи нужно использовать с осторожностью в Verilog, т.к. все, что описано в модуле, синтезатор в конечном итоге преобразует в электрическую схему реального устройства, состоящую из элементарных логических вентилях и триггеров. Синтез умножения или деления может привести к снижению быстродействия схемы. Да и места в чипе они займут много. Подумайте, возможно, вместо них можно применить, например, сдвиги. Конечно, некоторые чипы ПЛИС могут иметь встроенные умножители, если синтезатор знает об этом, он может использовать их.

## **Лабораторный макет**

Лабораторная работа выполняется на макете Digilent NEXYS2 Board, работающий на ПЛИС Spartan3E-500 FG320, который имеет достаточно большой объем компонент и для выполнения существенно более сложных работ. Структурная схема макета и устройства ввода и отображения информации приведены на рисунке 14.

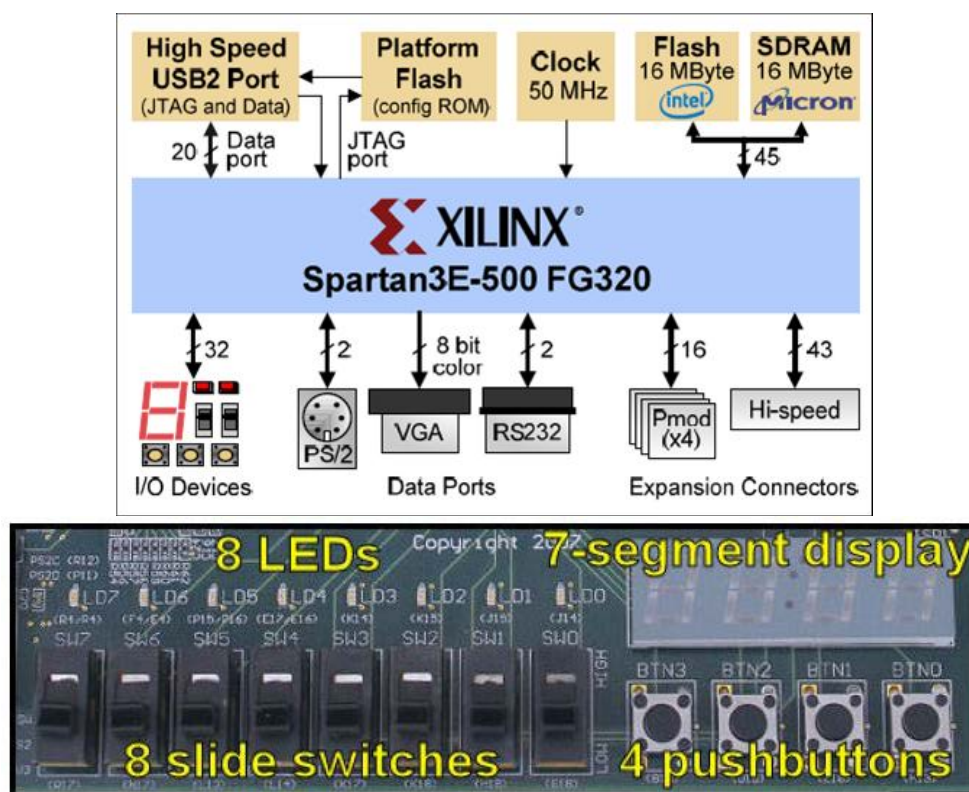


Рис. 14. Структурная схема макета NEXYS 2 и устройства ввода и отображения информации ((I/O Devices).

Светодиодный семи сегментный динамический индикатор имеет 4 цифры. Схема соединения светодиодов индикатора показана на рисунке 15.

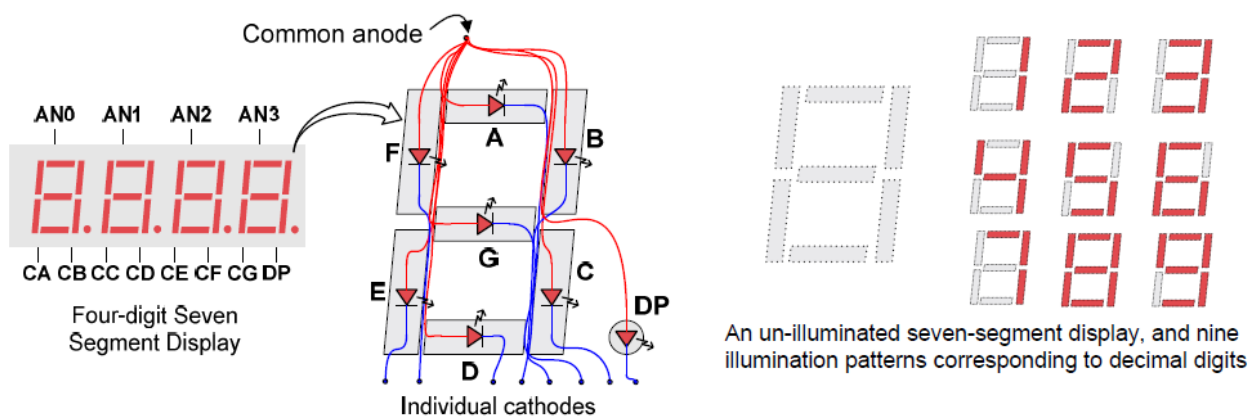


Рис. 15. Схема соединения светодиодов индикатора макета.

Аноды светодиодов каждой цифры соединены вместе и напряжение (+3.3V) подается на них через транзисторные ключи. Одноименные сегменты (катоды светодиодов) всех четырех цифр также соединены вместе, поэтому цифры необходимо включать поочередно. Если переключать цифры достаточно быстро, то создается иллюзия, что все четыре индикатора светятся одновременно, хотя на самом деле в каждый момент времени светится только один из четырех индикаторов. Такая работа индикатора позволяет сэкономить достаточно много места на плате лабораторного макета и Ю-выводы ПЛИС. Веди для того, чтобы одновременно показать все 4 цифры на индикаторах, нужно соединить ПЛИС с индикаторами 36 проводниками (8 катодов-сегментов + 1 общий анод \* 4 индикатора), а при такой реализации используется всего 12 проводников (4 анода + 8 соединенных с каждым индикатором катодов). Рассмотрим временные диаграммы поочередного включения цифр индикатора, которые показаны на рисунке 16.



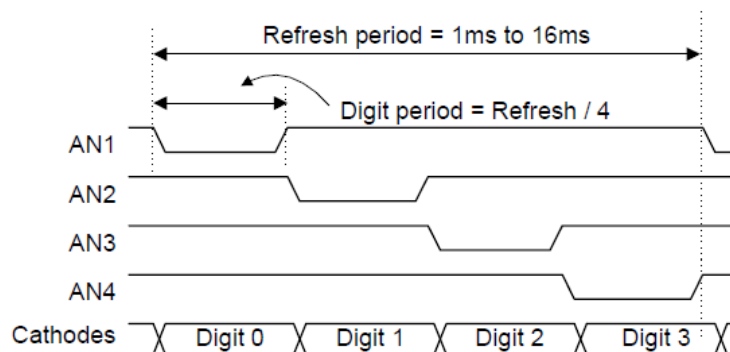


Рис. 16. Временные диаграммы поочередного включения цифр индикатора макета.

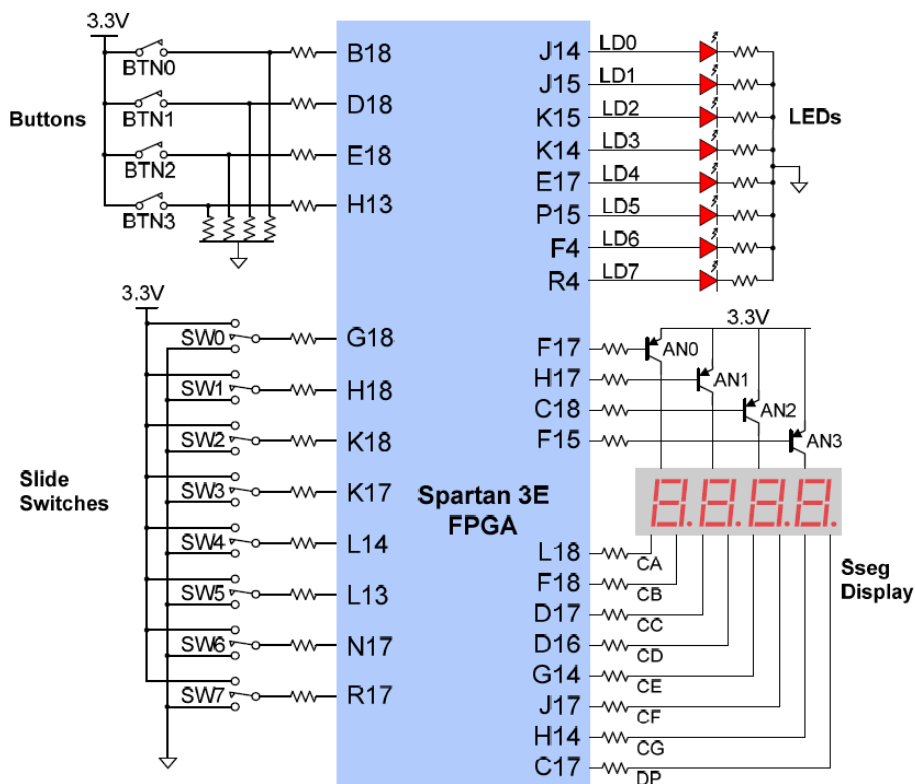


Рис. 17. Схема соединения органов управления и индикации макета NEXYS-2 с ПЛИС. В прямоугольнике Spartan 3E FPGA указаны номера контактных площадок ПЛИС макета, органов управления и индикации.

Возьмем период переключения равный, например, 16 мс. Тогда каждый из четырех индикаторов должен светиться по 4мс. Сегменты индикатора светятся, если на общий анод индикатора подан логический «0» и катод, соответствующего нужному сегменту светодиода, подключен к логическому «0» (см. схему рис.17). В первые 4мс устанавливаем значения на катодах, такие чтобы светились сегменты соответствующие нужной нам цифре и подаем логический «0» на анод первого индикатора (на других анодах «1») – светится первая цифра. Через 4мс меняем значения на катодах, чтобы получить вторую цифру, которую хотим высветить на индикаторе и подаем логический «0» на анод второго индикатора, а на все остальные аноды «1» - светится цифра на втором индикаторе, остальные погашены. Далее с 8мс по 12мс высвечиваем аналогично третью, а затем с 12мс по 16мс четвертую цифру, потом снова первую и так далее. На рисунках 17 и



18 показаны схемы соединения органов управления и индикации лабораторного макета с ПЛИС.

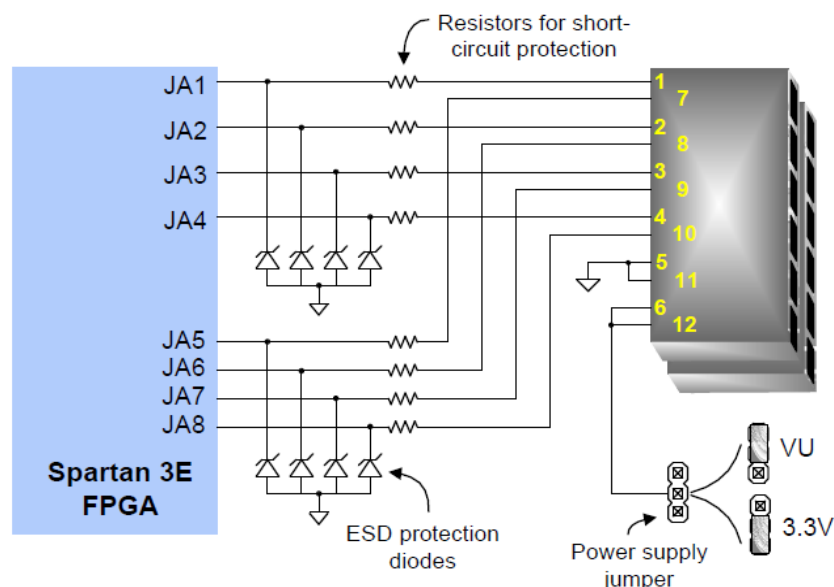


Table 3: Nexys2 Pmod Connector Pin Assignments							
Pmod JA		Pmod JB		Pmod JC		Pmod JD	
JA1: L15	JA7: K13	JB1: M13	JB7: P17	JC1: G15	JC7: H15	JD1: J13	JD7: K14 <sup>1</sup>
JA2: K12	JA8: L16	JB2: R18	JB8: R16	JC2: J16	JC8: F14	JD2: M18	JD8: K15 <sup>2</sup>
JA3: L17	JA9: M14	JB3: R15	JB9: T18	JC3: G13	JC9: G16	JD3: N18	JD9: J15 <sup>3</sup>
JA4: M15	JA10: M16	JB4: T17	JB10: U18	JC4: H16	JC10: J12	JD4: P18	JD10: J14 <sup>4</sup>

Рис. 18. Порты ввода вывода макета NEXYS 2.

Для выполнения лабораторных работ еще понадобится знать, что на лабораторном макете есть генератор сигнала синхронизации с тактовой частотой 50МГц, который подключен к выводу «B8» ПЛИС. Для корректной работы модулей выход генератора необходимо подключать к входам CLK описанных в работе счетчиков через буфер BUFG, который можно найти в разделе «Buffer» библиотеки элементов проекта.

## ПЛИС Spartan3E-500 FG320

Рассмотрим подробнее устройство ПЛИС Spartan3E-500 FG320, на основе которого сделан лабораторный макет Digilent NEXYS2 Board. ПЛИС Spartan3E-500 FG320 имеет аппаратную реализацию:

- 9312 триггеров (FD),
- 9312 табличных 4-х входных генераторов логических функций (4 inputs LUTs),
- 232 доступных для использования блоков ввода вывода (bonded IOBs) ,
- 73 кбит распределенной памяти (Slice RAM),
- 16 аппаратных модулей блочной памяти с суммарной емкостью 360 кбит
- 20 аппаратных умножителей 18x18 бит.

Плис Spartan3E-500 FG320 представляет собой интегральную микросхему в BGA корпусе, пример показан на рисунке 19.

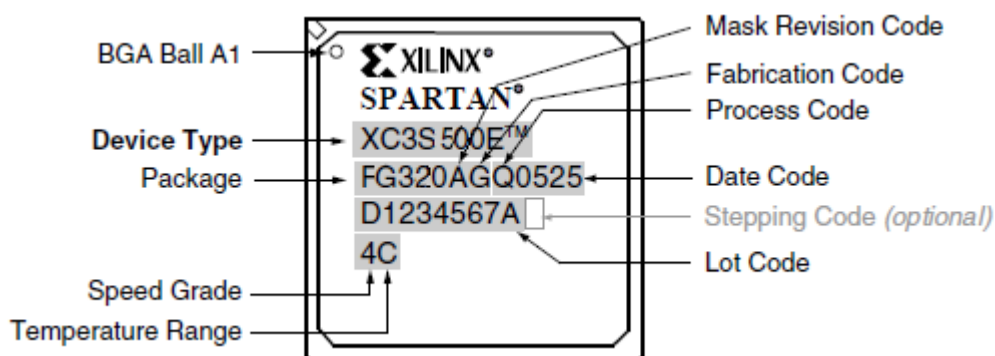


Рис. 19. ПЛИС Spartan3E-500 FG320 в BGA корпусе.

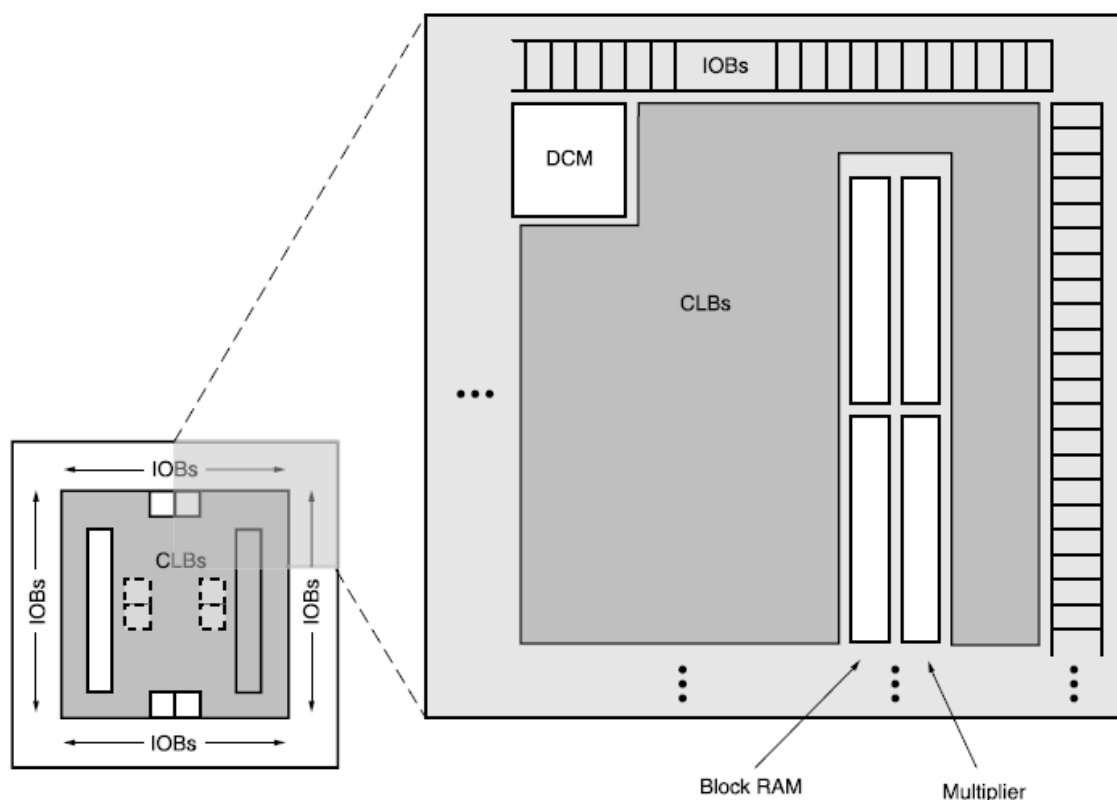


Рис.20. Архитектура ПЛИС Spartan3E.

На рисунке 20 показана архитектура ПЛИС, которая состоит из пяти основных аппаратно реализованных элементов:

1. Configurable Logic Blocks (CLBs), содержащие Look-Up таблицы (LUTs) которые предоставляют программируемую логику, а также D-триггеры. CLBs позволяют реализовать широкий диапазон различных логических функций, а также память.
2. Input/Output Blocks (IOBs) – блоки ввода/вывода, контролирующие передачу данных между выводами микросхемы и внутренней логической схемы устройства.
3. Block RAM – модули блочной памяти с суммарной емкостью 360 кбит.
4. Multiplier Blocks – аппаратные умножители 18х18 бит.
5. Digital Clock Manager (DCM) Blocks – самонастраивающиеся блоки синхронизации, предоставляют возможности для задержки, деления, умножения, сдвига по фазе сигналов синхронизации.

Configurable Logic Blocks (CLBs) составляют основной ресурс для реализации комбинационных и последовательностных логических схем. Каждый CLB содержит 4 слайса, каждый слайс содержит две четырех-входовых Look-Up таблицы (LUTs), которые

и используются для программирования логических функций и триггеров. LUT-ы могут использоваться как, например, 16x1 RAM (оперативная память) или как 16-битный регистр сдвига, или как комбинационное устройство, вычисляющее логическую или арифметическую функцию. Логические функции, описанные (например, на verilog) в проекте разрабатываемого цифрового устройства преобразуются в физические устройства, используя ресурсы CLB.

Каждый CLB содержит четыре соединенных между собой слайса, как показано на рисунке 21.

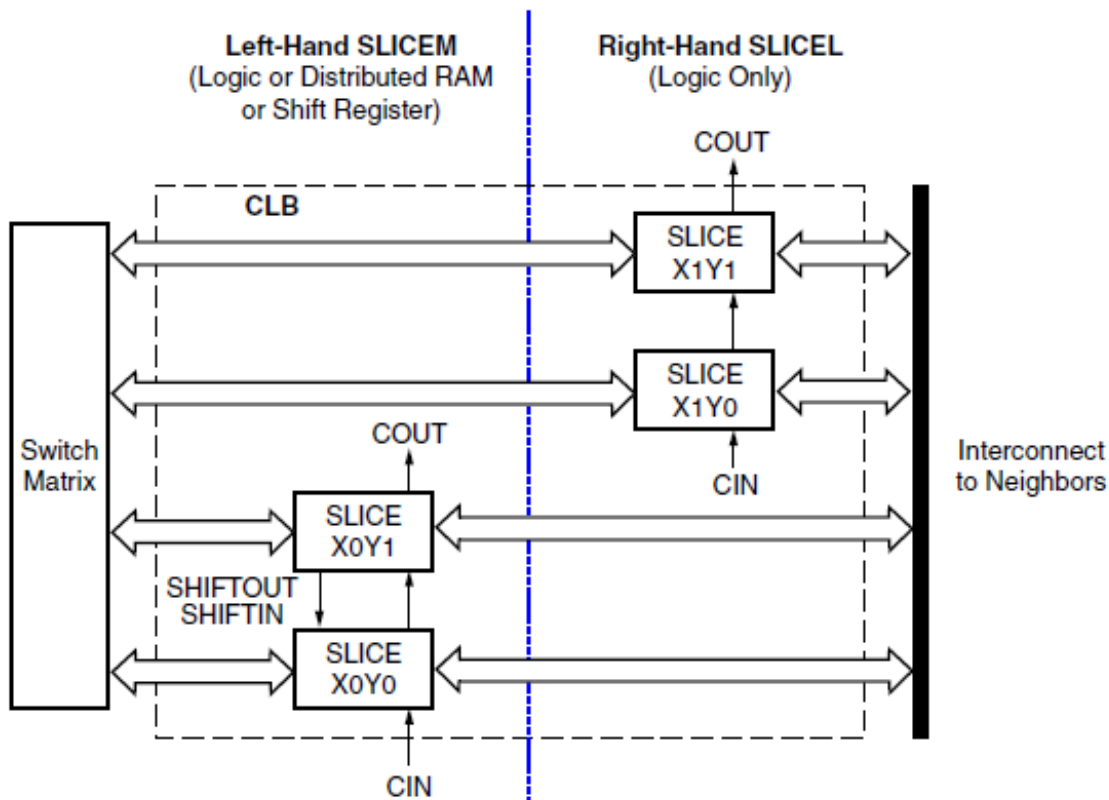


Рис. 21. Размещение слайсов в CLB.

Слайсы сгруппированы парами. Левая пара называется SLICEM. Предоставляет реализацию как логических функций, так и памяти. Правая пара называется SLICEL. Предоставляет реализацию только логических функций. Так сделано, чтобы уменьшить физический размер CLB в микросхеме ПЛИС и уменьшить стоимость самой ПЛИС. Состав SLICEM и SLICEL показан на рисунке 22.

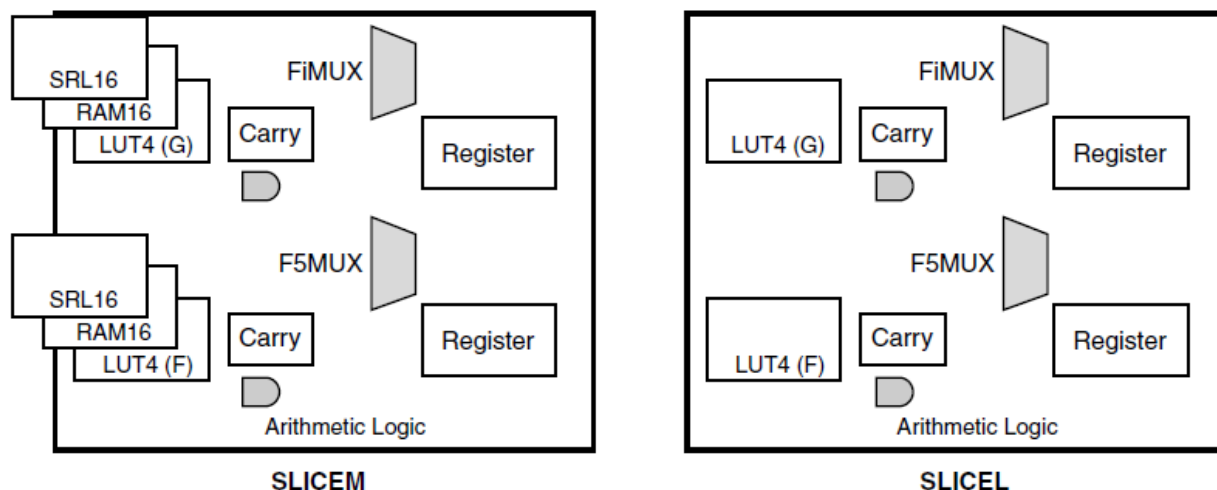


Рис. 22. Ресурсы слайсов SLICEM и SLICEL.

Look-Up таблицы (LUT) – основанные на 16-и битной оперативной памяти генераторы логических функций. LUTы в каждой паре слайсов SLICEM могут быть также сконфигурированы как оперативная память или 16-и битный регистр сдвига. На рисунке 23 показаны LUT ресурсы в слайсе. Каждый из двух LUTов в слайсе имеет 4 логических входа (A1-A4) и единственный выход (D). В LUT таблице записано значение выхода D для каждого из возможных значений входов A1-A4 – то есть таблица истинности логической функции, которую реализует LUT. Функции с большим количеством входов могут быть реализованы с использованием нескольких LUT, соединенных в каскад. Выход LUTа может быть подключен, например, к мультиплексору большой комбинационной схемы, схеме арифметического устройства с сигналом переноса, непосредственно к выходу CLB или к триггеру в CLB.

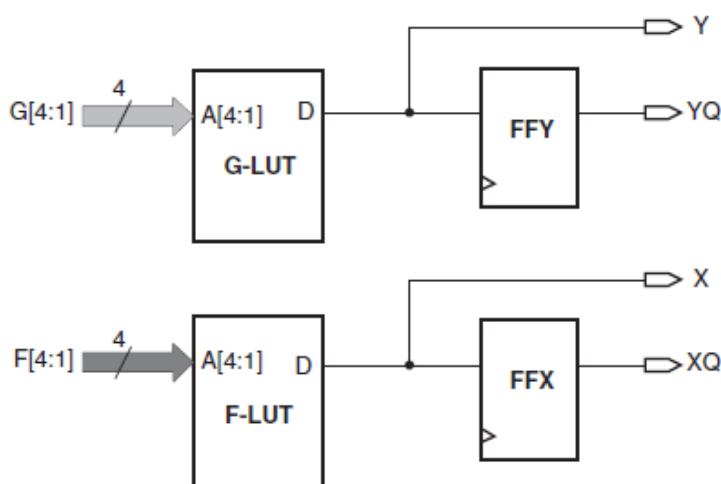


Рис. 23. LUT ресурсы в слайсе.

Еще несколько слов о том, как соединяются между собой элементы ПЛИС. Межсоединения в ПЛИС (также называется трассировка) реализуется с помощью **Switch matrix**, которая представляет собой программируемую при конфигурации ПЛИС сеть сигналов (путей) между входами и выходами функциональных элементов ПЛИС, таких как IOB, CLB, DCM, block RAM. Switch matrix позволяет соединить различные типы устройств, как показано на рисунке 24.

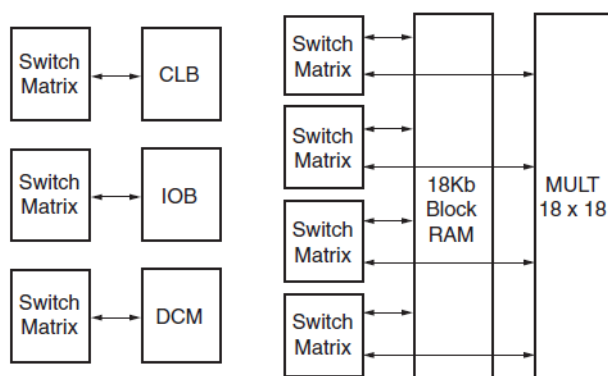


Рис. 24. Соединение разных типов устройств ПЛИС (CLBs, IOBs, DCMs, and Block RAM/Multiplier).

Фактически, ПЛИС Spartan-3E может быть представлен как массив соединенных элементов в виде колонок и рядов, как показано на рисунке 25.

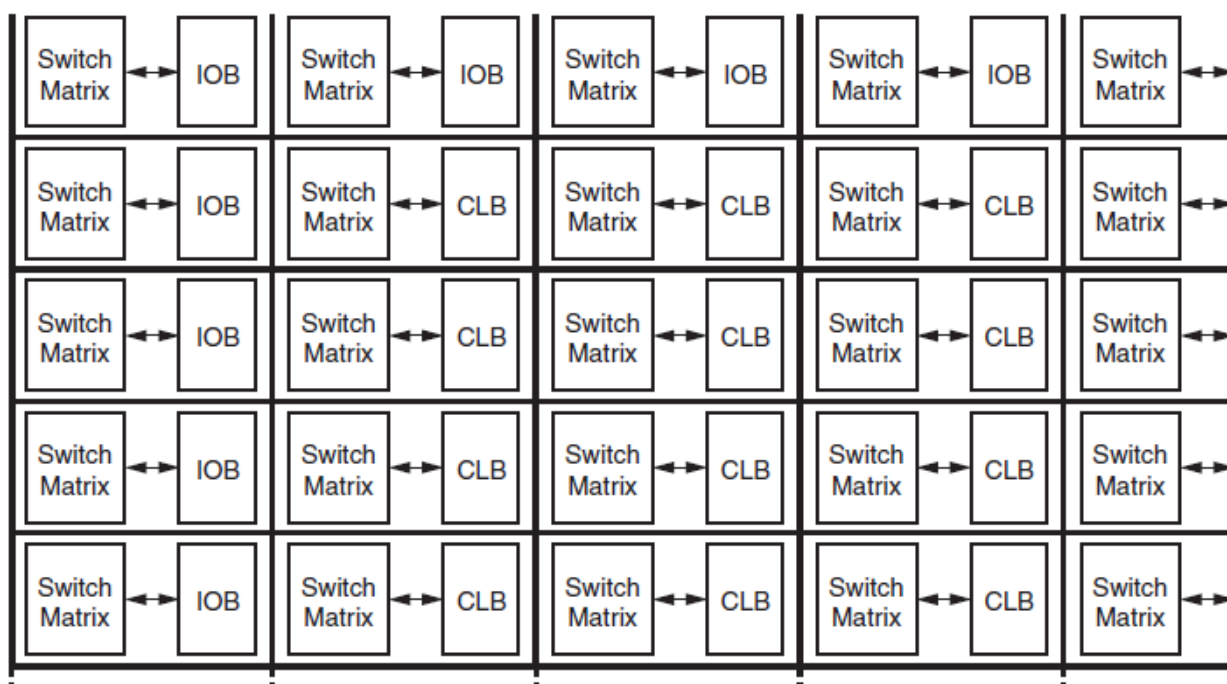


Рис. 25. Массив соединенных элементов ПЛИС Spartan-3Е.

Описанное на Verilog или в виде графической схемы цифровое устройство, разрабатываемое в среде Xilinx ISE Design Suite, в результате работы синтезатора в конечном итоге представляется в виде бинарного файла конфигурации ПЛИС. Этот файл содержит информацию о том, как должны быть сконфигурированы LUTы и как они должны соединяться между собой и внешними устройствами (switch matrix). Далее этот файл конфигурации необходимо загрузить в ПЛИС, чтобы получить реально работающее устройство. Современные ПЛИС поддерживают различные интерфейсы для конфигурации. Не будем их подробно рассматривать. Для записи ПЛИС лабораторного макета, производитель макета предоставляет программу Digilent Adept, с помощью которой ПЛИС макета “прошивается” через USB интерфейс. Причем загрузить файл конфигурации можно как непосредственно в ПЛИС, так и в энергонезависимую память, которая присутствует в макете. Во втором случае макет автоматически возьмет конфигурацию ПЛИС из памяти и восстановит свою работу после отключения/включения питания макета.

## Среда разработки Xilinx ISE Design Suite 14.4

Среда разработки Xilinx ISE Design Suite 14.4 имеет редактор Verilog, графический редактор схем, синтезатор, симулятор. Программа для прошивки конфигурации разработанного устройства в ПЛИС используется отдельная от среды разработки – Digilent Adept, которая поставляется производителем макета Digilent NEXYS2 Board. Рассмотрим пример выполнения лабораторной работы в Xilinx ISE Design Suite 14.4 от создания проекта до загрузки конфигурации ПЛИС в макет.

### Создание проекта

В папке FRTK создать папку со своим именем (только латинские символы). Запустить среду разработки ISE Design Suite 14.4. Значок на рабочем столе:



Далее в созданной папке создать проект (**File -> New project**) с именем **LabN\_номер\_лаб\_работы**. Установить настройки для ПЛИС, используемой в макете NEXYS2 (Spartan3E, XC3S500E, FG320, XST (VHDL/Verilog), Isim(VHDL/Verilog), Store all values). Пример настроек проекта показан на рисунке 26.

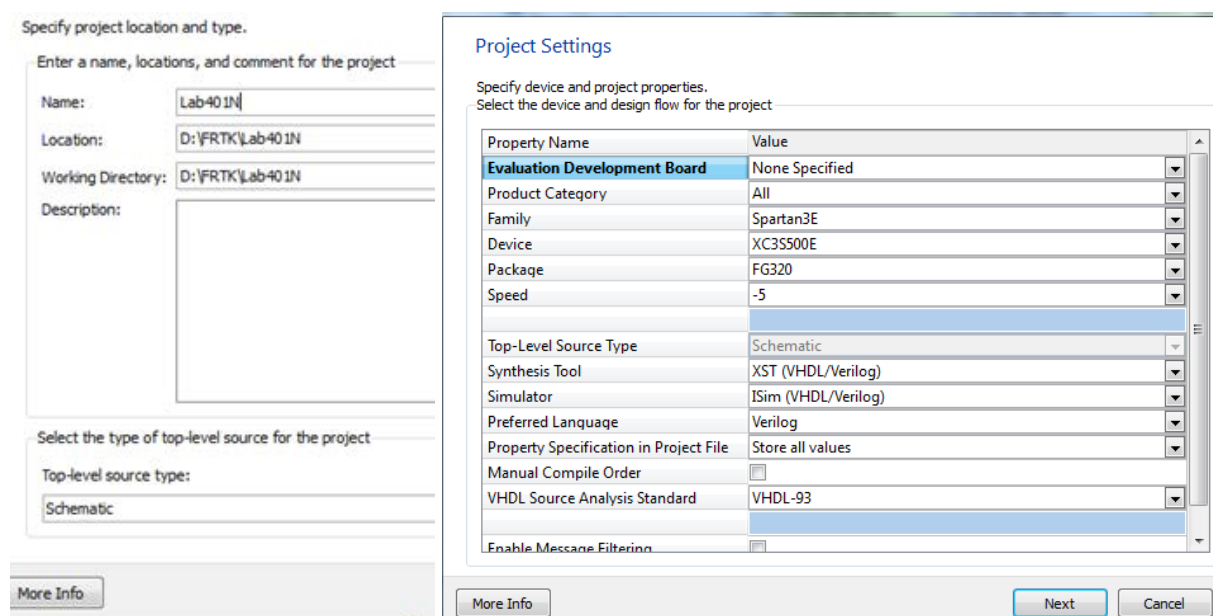


Рис.26. Пример настроек проекта для загрузки на макет Nexys2.

### Создание и синтез модуля

Рассмотрим все действия подробнее на примере модуля двоичного суммирующего счетчика с асинхронным входом сброса **VCB2CE**. Сразу следует отметить, что в среде проектирования ISE14 все операции доступны только для модуля помещенного на вершину проекта (Set as Top Module). Но проверка синтаксиса и создание символа модуля

для схематика доступны для любого модуля. Как правило, все модули пишутся на Verilog, а TOP-модуль выполняется в виде графической схемы. Фактически, достаточно только схему пометить командой **Set as Top Module**, т.к. именно ее необходимо будет синтезировать и загрузить в макет, а все используемые в ней модули синтезируются и подгружаются автоматически. Но если мы хотим посмотреть информацию о синтезе какого-то подмодуля – тогда необходимо будет именно этот модуль сделать TOP-модулем и отдельно синтезировать. После этого будет доступен просмотр синтезированной логической и технологической схемы именно выбранного модуля, а также отчеты о синтезе – число слайсов (Slices), триггеров (Flip Flops) и таблиц логических функций (LUTs), необходимых для реализации модуля.

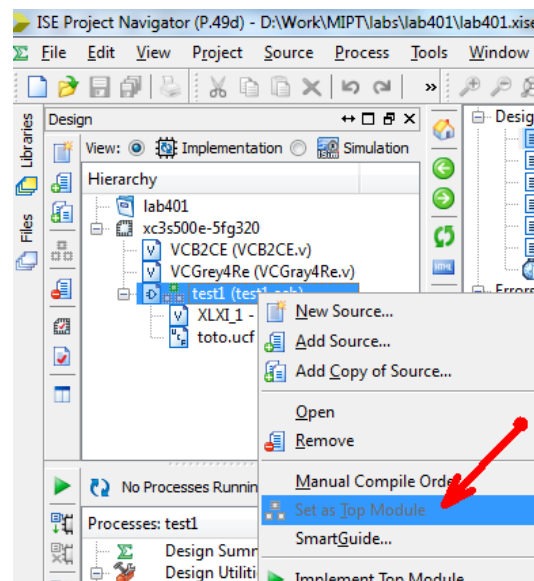
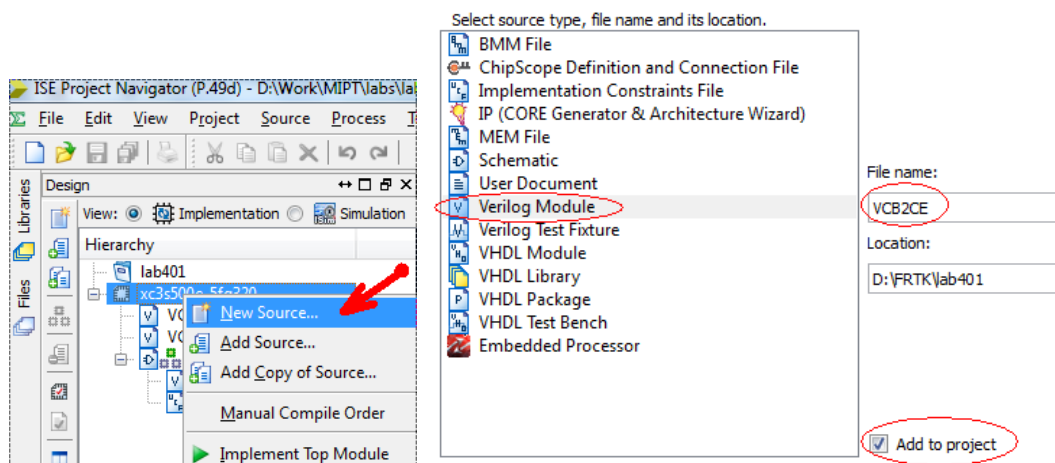


Рис.27. Выбор модуля верхнего уровня.

И так, в окне источников (Sources) создадим (New Source) заданный модуль (Verilog Module). Сделаем его главным в проекте (Set as Top Module). Введем на Verilog-е текст схемы модуля. Установим режим View: **Implementation**. Проверим синтаксис введенного текста схемы (В окне процессов (Processes) пункт Synthesize XST -> Check Syntax). Скриншоты в среде разработки показаны на рисунке 28.

#### Select Source Type





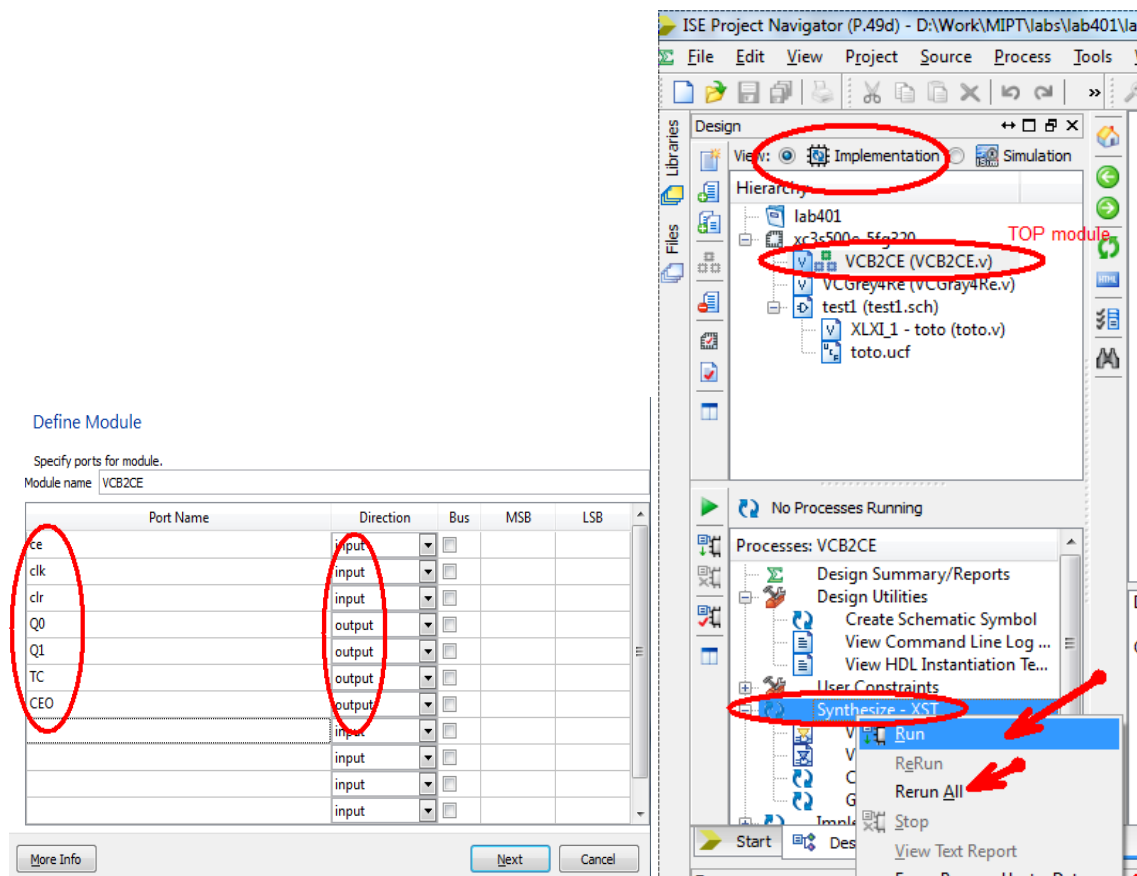


Рис.28. Скриншоты создания и синтеза модуля **VCB2CE** в среде разработки.

Далее выполним синтез модуля. Для этого надо в окне процессов (Processes) выполнить Synthesize XST. Следует исправить возможные ошибки, обратить внимание на предупреждения. Проверить соответствует ли Вашим представлениям синтезированная логическая схема этого модуля (View RTL Schematic). Посмотреть технологическую (View Technology Schematic) схему синтезированного модуля. В отчете о синтезе (View Synthesis Report) посмотреть число слайсов (Slices), триггеров (Flip Flops) и таблиц логических функций (LUTs) необходимых для реализации модуля. Создать схемотехнический символ модуля (в окне процессов Design Utilities → Create Schematic Symbol).

## Логическое моделирование устройств

Для тестирования, отладки своего разрабатываемого устройства в среде разработки Xilinx ISE Design Suite 14.4 имеется симулятор, с помощью которого можно посмотреть временные диаграммы написанного модуля, посмотреть как меняется состояние выходов модуля при изменении состояния входов. Для проведения логического моделирования Xilinx ISE Design Suite 14.4 необходимо создать модуль задания - Verilog Test Fixture, в котором описывается на языке Verilog, какой именно модуль из проекта мы хотим моделировать, на какие входы устройства будем воздействовать, периодические сигналы, начальное состояние входов и их изменение через заданный период времени.

Создадим задание на моделирование, например, для счетчика VCBmCE. Чтобы не путаться, назовем модуль с описанием задания моделирования tf\_VCBmCE.

**module** tf\_VCBmCE;



```

// Inputs
reg ce;
reg clk;
reg clr;

// Outputs
wire [2:0] Q;
wire TC;
wire CEO;

// Instantiate the Unit Under Test (UUT)
VCBmCE uut (
    .ce(ce),
    .Q(Q),
    .clk(clk),
    .TC(TC),
    .clr(clr),
    .CEO(CEO) );

// Генератор периодического сигнала синхронизации clk
parameter Tclk=20; //Период сигнала синхронизации 20 нс
always begin clk=0; #(Tclk/2); clk=1; #(Tclk/2); end

// Генератор периодического сигнала ce
parameter Tce=80; //Период сигнала ce 80 нс
always begin ce=0; #(3*Tce/4); ce=1; #(1*Tce/4); end

initial begin // Initialize Inputs
    clr = 0; //Исходное состояние входов
    #92; clr = 1; //Через 92 нс 1
    #5; clr = 0; //Через 5 нс 0
    #111; clr = 1; //Через 111 нс 1
    #70; clr = 0; //Через 70 нс 0
end
endmodule

```

Описание портов создается автоматически при добавлении файла Verilog Test Fixture к проекту (в приведенном выше примере выделено курсивом). Сигналы входов также автоматически объявляются регистрами, на которые в разделе **initial begin** необходимо задать желаемые изменения состояний через заданное время (временные диаграммы). Но до блока **initial begin** следует задать все периодические сигналы. Например, сигнал синхронизации **clk** с периодом 20 нс задан следующим образом:

```

parameter Tclk=20;
always begin clk=0; #(Tclk/2); clk=1; #(Tclk/2); end

```

Запись **#значение** означает паузу столько миллисекунд, сколько указано в **значении**. То есть запись **always begin clk=0; #(Tclk/2); clk=1; #(Tclk/2); end** фактически означает – установить **clk** в логический «0», подождать половину периода **Tclk** и установить **clk** в логическую «1», снова подождать половину периода **Tclk** и установить **clk** в логический «0».

Аналогично пишется и изменение входных сигналов в **initial begin** блоке. Задаем входные сигналы и их изменение через указанное время, а симулятор нам покажет на временной диаграмме, как в результате изменятся выходные сигналы нашего модуля. В примере **tf\_VCBmCE** мы изменяем значение входа сброса счетчика **clr**. Сигнал **clr** устанавливается равным «1» через 92 нс после начала на 5 нс, а затем еще раз через 111 нс на 70 нс. Сигналы **clk** и **ce** задаются генераторами периодических сигналов. Вот какую временную диаграмму мы получим в результате моделирования (рис. 29):

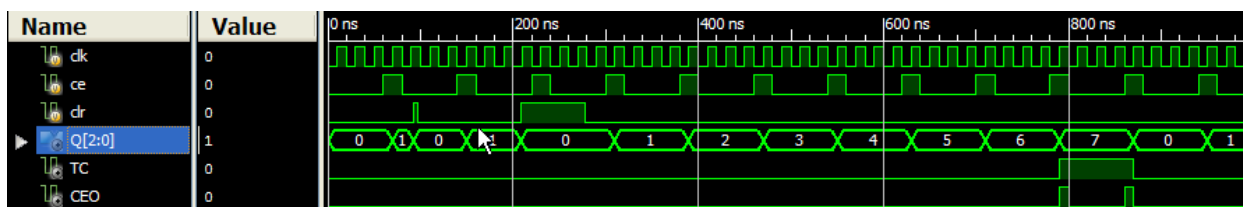


Рис. 29. Пример временных диаграмм логического моделирования модуля VCBmCE (m=3).

На приведенной временной диаграмме состояние триггеров Q[2:0] счетчика отображается в виде положительного декадного числа (Radix – Unsigned Decimal). Высокий уровень сигнала **clr**=1 сбрасывает в 0 триггеры и удерживает их в 0 независимо от сигналов на других входах (**ce** и **clk**) счетчика.

Для моделирования модуля в Xilinx ISE Design Suite 14.4 нужно предварительно установить режим **Behavioral Simulation** (рис. 2930).

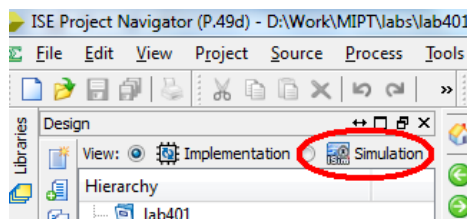


Рис.30. Скриншот выбора режима Behavioral Simulation.

Создадим для нашего модуля задание на моделирование: В окне источников (Sources) создать New source -> Verilog Test Fixture, в окне associate source выбрать модуль для моделирования (см. рисунок 31).

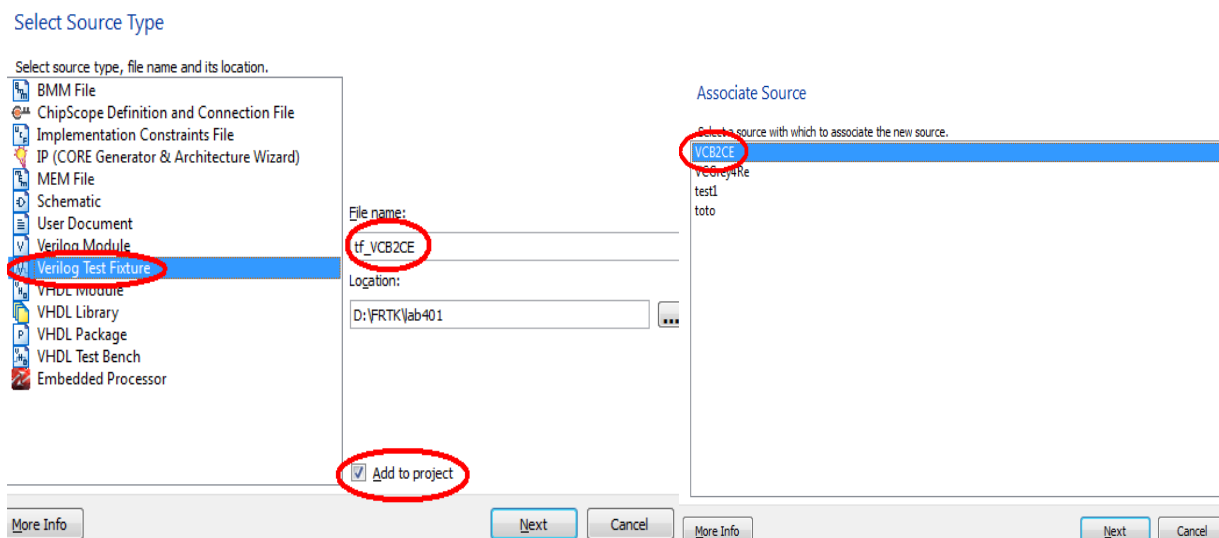


Рис.31. Скриншоты создания задания на моделирование.

В результате будет создан verilog файл с описанием задания для моделирования модуля VCB2CE. Этот файл нужно заполнить, прописав в него периодические сигналы, начальное состояние входов модуля и описать как сигналы на входах будут меняться через заданное время. Подробнее синтаксис этого файла смотрите в разделе «**Логическое моделирование устройств**».

После того, как файл задания готов, можно запустить симулятор. Выделив модуль, в окне процессов (Processes) выполнить **Simulate behavioral model** и посмотреть содержательные временные диаграммы модуля. См. скриншоты на рисунке 32.

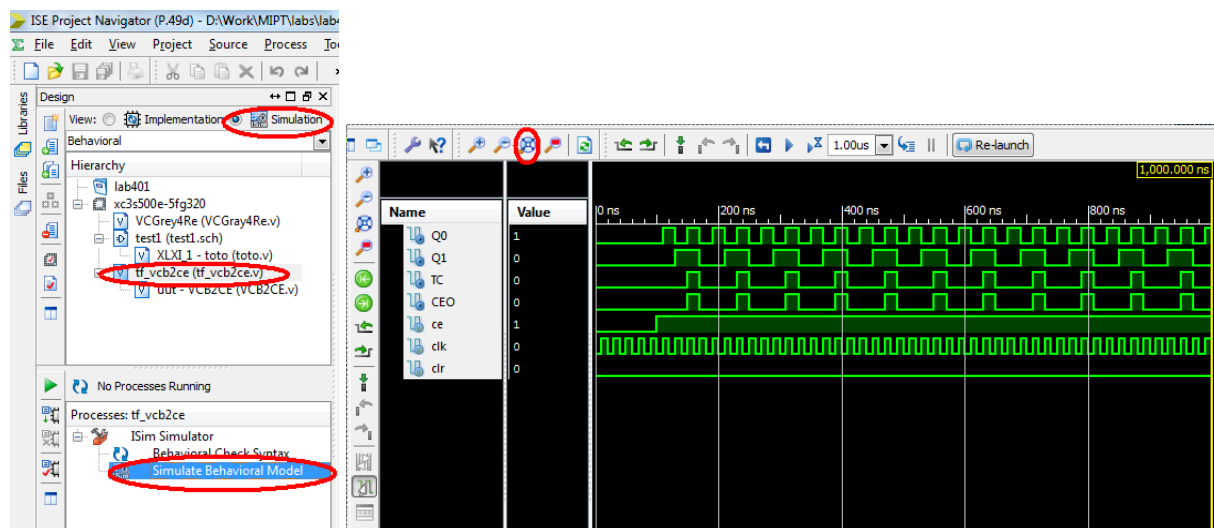


Рис.32. Скриншот запуска и работы симулятора.

По умолчанию время моделирования – 1000 наносекунд. Это время можно изменить в окне Properties Simulate Behavioral Model (см. рисунок 33).

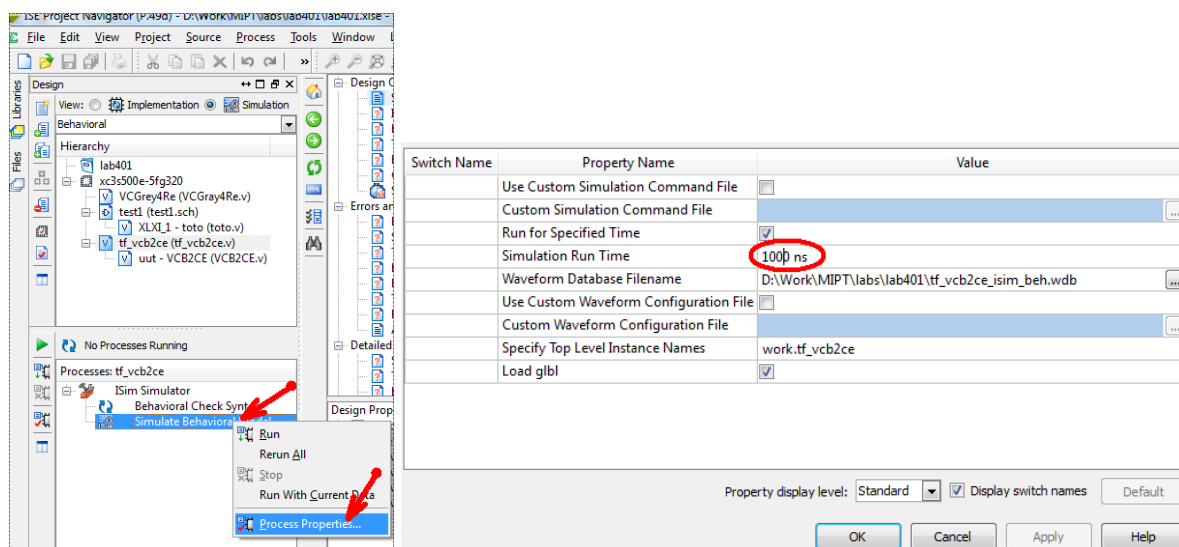


Рис.33. Установка необходимого времени моделирования.

## Графический редактор схем

Как правило, все модули пишутся на Verilog, а TOP-модуль, объединяющий все подмодули, выполняется в виде графической схемы. Чтобы создать схему надо в окне

источников (Sources) создать **New Source -> Schematic**. Скриншоты из среды проектирования показаны на рис. 34. Не забудьте перейти в режим Implementation и установить схему как TOP-модуль.

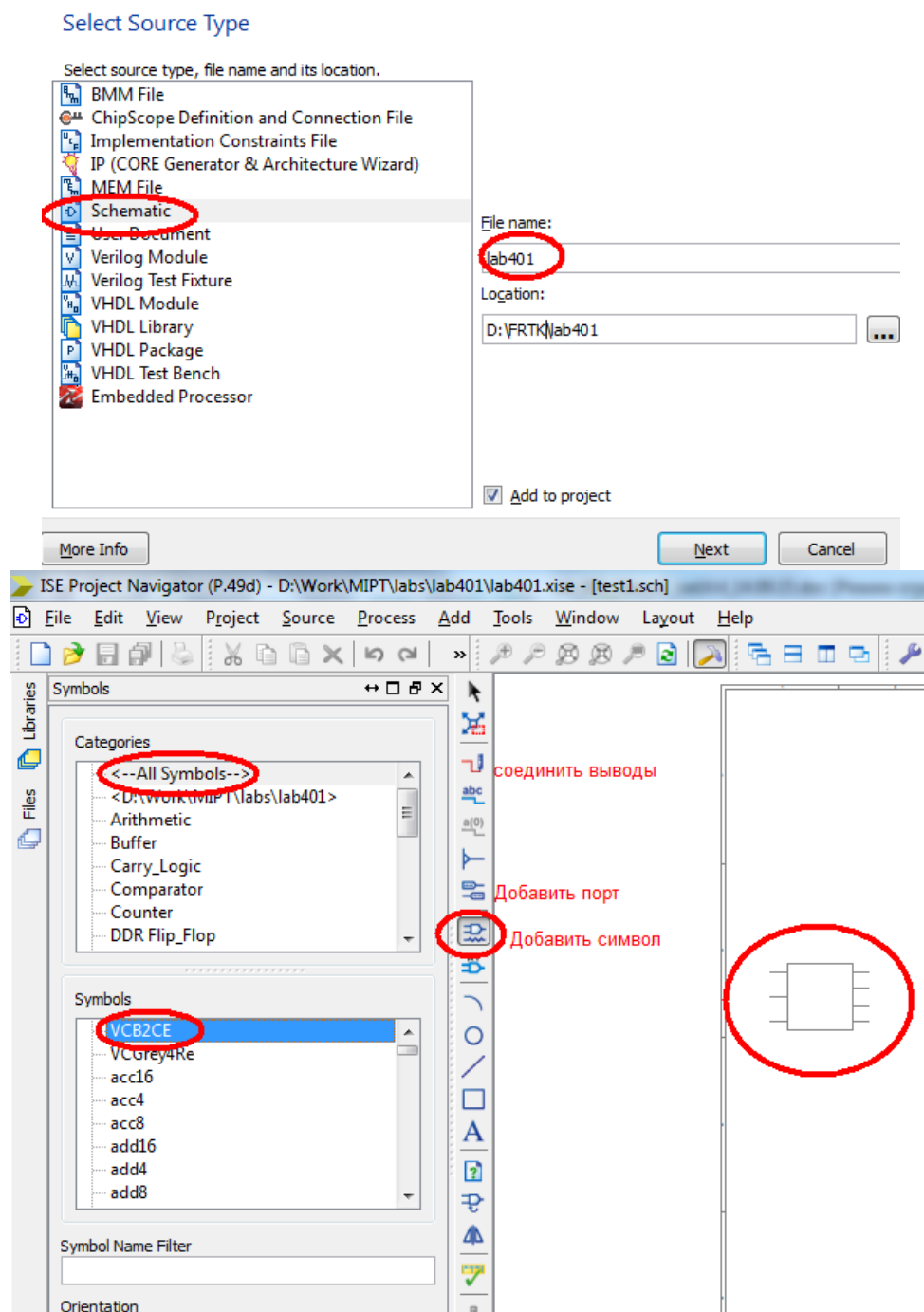


Рис.34. Скриншоты создания и редактирования схемы.

Нужно добавить на схему символы всех подмодулей и соединить их проводниками. Добавить порты ввода/вывода для физического взаимодействия с ПЛИС на макете. Задать названия сигналов на портах. Не забудьте, что сигнал синхронизации 50 МГц надо подключать к счетчикам через элемент **bufg**. Пример схемы показан на рисунке 35.

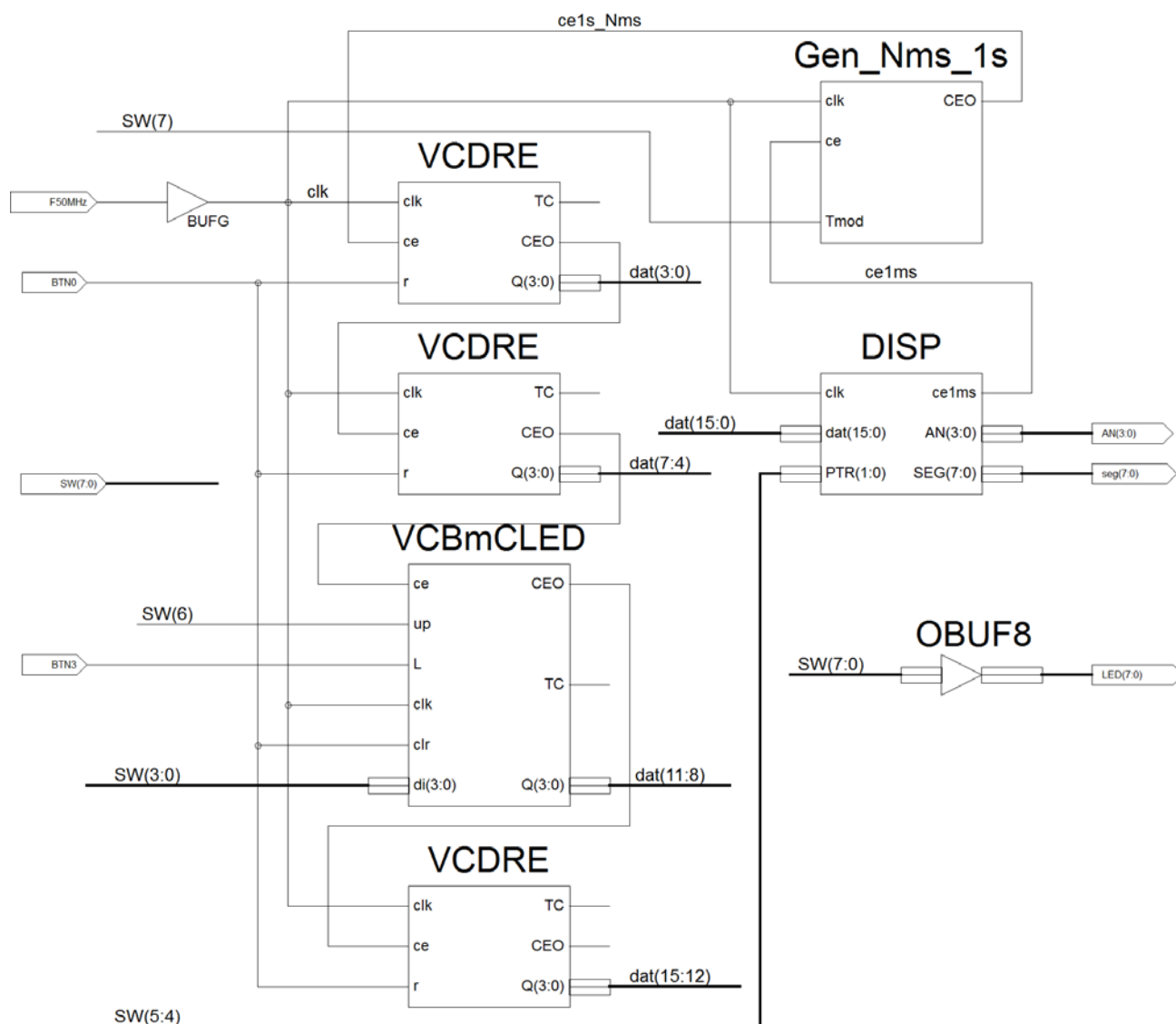


Рис.35. Пример схемы, состоящий из модулей (VCDRE, VCBmCLED...).

## Файл назначения портов схемы физическим выводам ПЛИС (User constraints file)

Для описания взаимодействия ПЛИС с другими устройствами макета используется UCF-файл, в котором должны быть прописаны связи портов схемы (TOP-модуля) с контактными площадками ПЛИС. Для добавления UCF-файла в проект нужно создать (New Source) Implementation Constraints File с именем lab\_номер\_лаб\_работы.ucf и в созданный текстовый файл ввести список связей портов схемы с контактными площадками ПЛИС. Пример ucf-файла для макета NEXIS 2:

```
-----
NET "AN<0>" LOC = "F17" ;
NET "AN<1>" LOC = "H17" ;
NET "AN<2>" LOC = "C18" ;
NET "AN<3>" LOC = "F15" ;
```

```

NET "BTN0" LOC = "B18" ;
#NET "BTN1" LOC = "D18" ;
#NET "BTN2" LOC = "E18" ;
NET "BTN3" LOC = "H13" ;

NET "F50MHz" LOC = "B8" ;#clk
NET "seg<0>" LOC = "L18" ;
NET "seg<1>" LOC = "F18" ;
NET "seg<2>" LOC = "D17" ;
NET "seg<3>" LOC = "D16" ;
NET "seg<4>" LOC = "G14" ;
NET "seg<5>" LOC = "J17" ;
NET "seg<6>" LOC = "H14" ;
NET "seg<7>" LOC = "C17" ;#DOT

NET "SW<0>" LOC = "G18" ;
NET "SW<1>" LOC = "H18" ;
NET "SW<2>" LOC = "K18" ;
NET "SW<3>" LOC = "K17" ;
NET "SW<4>" LOC = "L14" ;
NET "SW<5>" LOC = "L13" ;
NET "SW<6>" LOC = "N17" ;
NET "SW<7>" LOC = "R17" ;

NET "LED<0>" LOC = "J14" ;#LD0
NET "LED<1>" LOC = "J15" ;#LD1
NET "LED<2>" LOC = "K15" ;#LD2
NET "LED<3>" LOC = "K14" ;#LD3
NET "LED<4>" LOC = "E17" ;#LD4
NET "LED<5>" LOC = "P15" ;#LD5
NET "LED<6>" LOC = "F4" ;#LD6
NET "LED<7>" LOC = "R4" ;#LD7

#NET "TXD" LOC = "P9" ;
#NET "RXD" LOC = "U6" ;

#NET "JA1" LOC = "L15" ;#Pin1
#NET "JA2" LOC = "K12" ;#Pin2
#NET "JA3" LOC = "L17" ;#Pin3
#NET "JA4" LOC = "M15" ;#Pin4
#NET "JA7" LOC = "K13" ;#Pin7
#NET "JA8" LOC = "L16" ;#Pin8
#NET "JA9" LOC = "M14" ;#Pin9
#NET "JA10" LOC = "M16" ;#Pin10

#NET "JB1" LOC = "M13" ;#Pin1
#NET "JB2" LOC = "R18" ;#Pin2
#NET "JB3" LOC = "R15" ;#Pin3
#NET "JB4" LOC = "T17" ;#Pin4
#NET "JB7" LOC = "P17" ;#Pin7
#NET "JB8" LOC = "R16" ;#Pin8
#NET "JB9" LOC = "T18" ;#Pin9

```

```
#NET "JB10" LOC = "U18" ;#Pin10
```

```
#NET "JC1" LOC = "G15" ;#Pin1
```

```
#NET "JC2" LOC = "J16" ;#Pin2
```

```
#NET "JC3" LOC = "G13" ;#Pin3
```

```
#NET "JC4" LOC = "H16" ;#Pin4
```

```
#NET "JC7" LOC = "H15" ;#Pin7
```

```
#NET "JC8" LOC = "F14" ;#Pin8
```

```
#NET "JC9" LOC = "G16" ;#Pin9
```

```
#NET "JC10" LOC = "J12" ;#Pin10
```

```
#NET "JD1" LOC = "J13" ;#Pin1
```

```
#NET "JD2" LOC = "M18" ;#Pin2
```

```
#NET "JD3" LOC = "N18" ;#Pin3
```

```
#NET "JD4" LOC = "P18" ;#Pin4
```

```
#NET "JD7" LOC = "K14" ;#LD3
```

```
#NET "JD8" LOC = "K15" ;#LD3
```

```
#NET "JD9" LOC = "J15" ;#LD3
```

```
#NET "JD10" LOC = "J14" ;#LD3
```

В этом текстовом файле символом # «закомментированы» неиспользуемые выводы ПЛИС. Синтаксис файла простой: перечисляются связи в формате

**NET "порт\_на\_схеме" LOC = "контактная\_площадка\_ПЛИС".**

## **Создание и загрузка в макет конфигурационного файла ПЛИС**

Для создания конфигурационного файла ПЛИС нужно выделить схему, установить ее как TOP-модуль и в окне процессов запустить Generate Programming File. Все этапы синтеза и имплементации при запуске генерации файла конфигурации будут пройдены. Убедиться в отсутствии ошибок. Загрузить полученный файл в макет с помощью



программы Digilent Adept .

Скриншоты создания и загрузки в макет файла конфигурации ПЛИС смотрите на рис. 36.

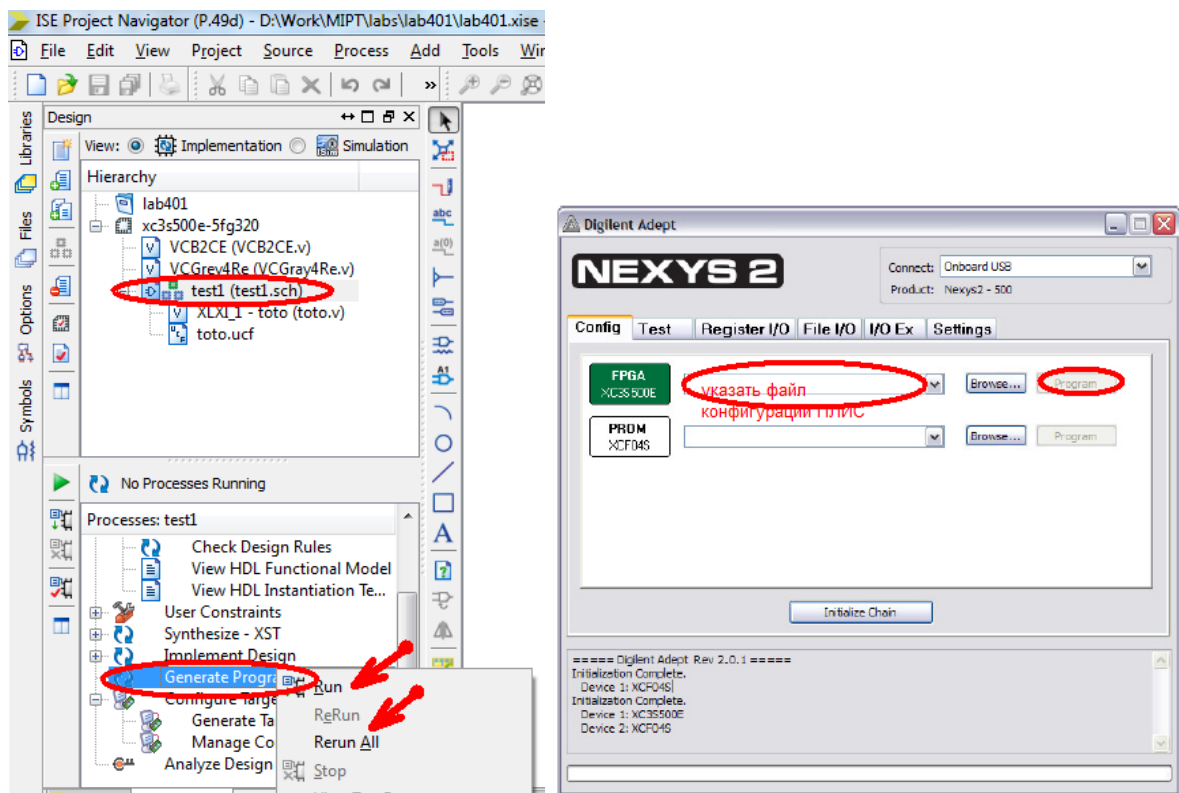


Рис.36. Скриншоты создания и загрузки в макет файла конфигурации ПЛИС.

## Список литературы

1. Материал из Википедии – Электроника.  
<https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%B8%D0%BA%D0%B0>
2. Материал из Википедии – Логический вентиль.  
<https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9%D0%B2%D0%B5%D0%BD%D1%82%D0%B8%D0%BB%D1%8C>
3. Материал из Википедии – Комбинационная логика.  
<https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B1%D0%B8%D0%BD%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0>
4. Материал из Википедии – Секвенциальная логика.  
<https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BA%D0%B2%D0%B5%D0%BD%D1%86%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0>
5. Материал из Википедии – Триггеры.  
<https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B8%D0%B3%D0%B3%D0%B5%D1%80>
6. Материал из Википедии – ПЛИС.  
<https://ru.wikipedia.org/wiki/%D0%9F%D0%9B%D0%98%D0%A1>
7. Введение в Verilog – язык описания цифровых схем. Николай К.  
[http://marsohod.org/index.php/downloads/doc\\_download/37-vvedenie-v-verilog](http://marsohod.org/index.php/downloads/doc_download/37-vvedenie-v-verilog)
8. Материал из Википедии – Код Грея.  
<https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4%D0%93%D1%80%D0%B5%D1%8F>
9. Даташит на ПЛИС Spartan-3E.