

**Министерство науки и высшего образования Российской Федерации**

**Национальный исследовательский университет «МЭИ»**

**Институт радиотехники и электроники им. В.А. Котельникова**

**Кафедра электроники и нанoeлектроники**

Практикум по дисциплинам

«Основы цифрового синтеза» и «Синтез цифровых интегральных схем»

для студентов, обучающихся по направлениям

11.03.04 и 11.04.04 «Электроника и нанoeлектроника»

Об отладочных платах.....	3
Работа № 1. Простая комбинационная логика. Непрерывное присваивание....	4
Работа № 2. Комбинационные схемы на уровне регистровых передач. Блокирующее присваивание.....	27
Работа № 3. Последовательностная логика. Неблокирующее присваивание. Параметризация.....	35
Работа № 4. Автоматы конечных состояний.....	46
Работа № 5. Проектирование регулярной структуры. Сумматоры.....	55
Работа № 6. Одно- и двухпортовая память.....	64
Работа № 7. Использование семисегментного индикатора. Функции.....	69
Работа № 8. Подключение дисплея LCD1602 к ПЛИС по 8-битной шине.....	70
Работа № 9. Передача данных между ПК и ПЛИС. Последовательный интерфейс U(S)ART.....	73
Работа № 10. Передача данных между ПЛИС. Последовательные интерфейсы GPIO, SPI и I2C.....	74
Работа № 11. Интерфейс VGA.....	75
Работа № 12. Подключение клавиатуры к ПЛИС. Интерфейс PS/2.....	76
Приложение А.....	77
Приложение Б.....	79

Для выполнения работ потребуется следующий набор программного обеспечения:

- Quartus Prime v. 20.1 Lite Edition
- ModelSim v. 20.1 Starter Edition
- Поддержка FPGA Cyclone IV v. 20.1

В дополнение к указанному набору полезным будет установить следующее дополнительное программное обеспечение:

- Visual Studio Code
- Icarus Verilog
- GTKWave

## **Об отладочных платах**

**A-C4E6**

**OMDAZZ**

# Работа № 1. Простая комбинационная логика.

## Непрерывное присваивание

### Создание проекта

1. Откройте Quartus Prime.
2. Создайте проект в Quartus Prime **File > New Project Wizard...**
3. В открывшемся окне нажмите *Next*.
4. Теперь необходимо указать директорию, в которой будут расположены файлы проекта, а также задать имя проекта и модуля верхнего уровня (по умолчанию они совпадают). Рекомендуется для каждого проекта формировать отдельную папку. В качестве имени директории, проекта и модуля верхнего уровня укажите **lab1**. Нажмите *Next*.

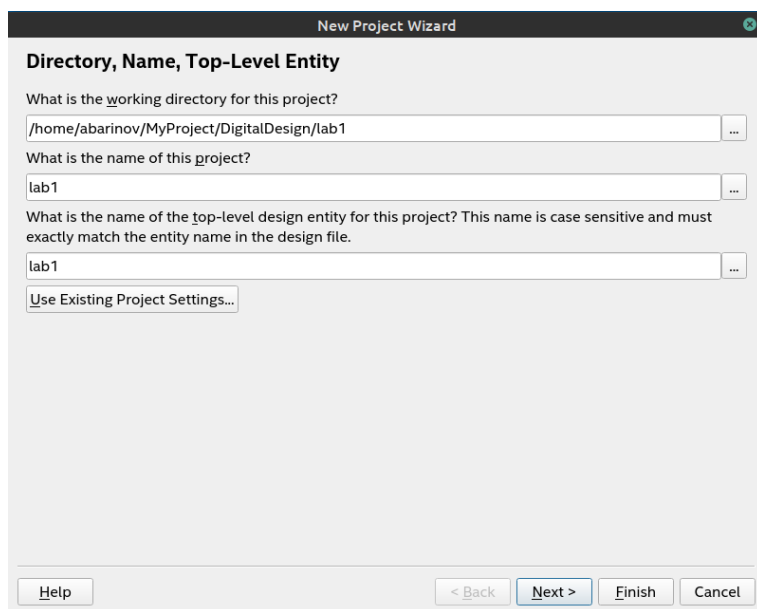


Рисунок 1.1 – Создание проекта

5. В следующем окне выберите *Empty Project* и нажмите *Next*.
6. В окне добавления файлов нажмите *Next*.
7. В окне настройки устройства ПЛИС выберите семейство устройств Cyclone IV E. Из списка доступных устройств выберите EP4CE6E22C8. Нажмите *Next*.

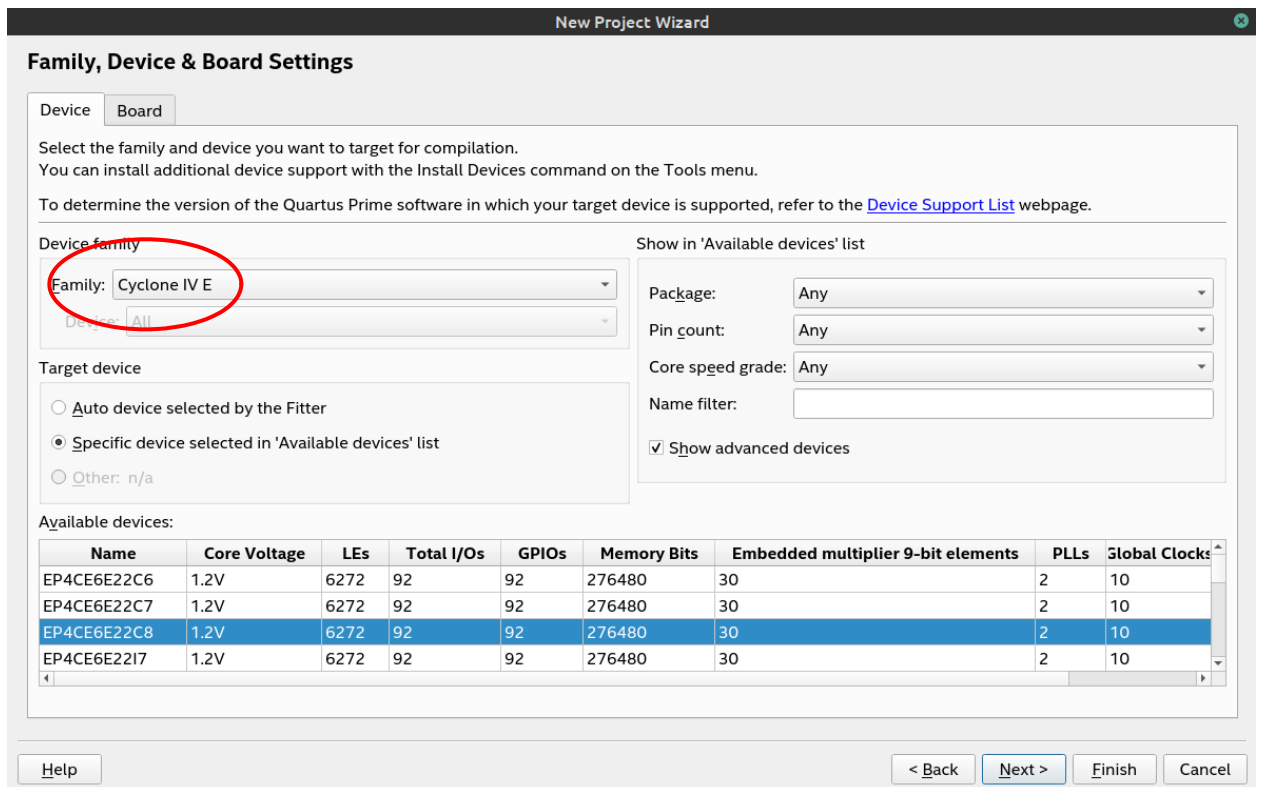


Рисунок 1.2 – Выбор ПЛИС

8. В следующем окне укажите инструмент моделирования ModelSim-Altera (**обратите внимание**, что не ModelSim!), а язык SystemVerilog HDL. На текущем этапе моделирование проводится не будет, а пройдет в задании 3, но так как проект для заданий один и тот же, то выбрать следует сейчас. Нажмите *Next*.
9. Ознакомьтесь с информацией в окне с итогами и нажмите *Finish*.

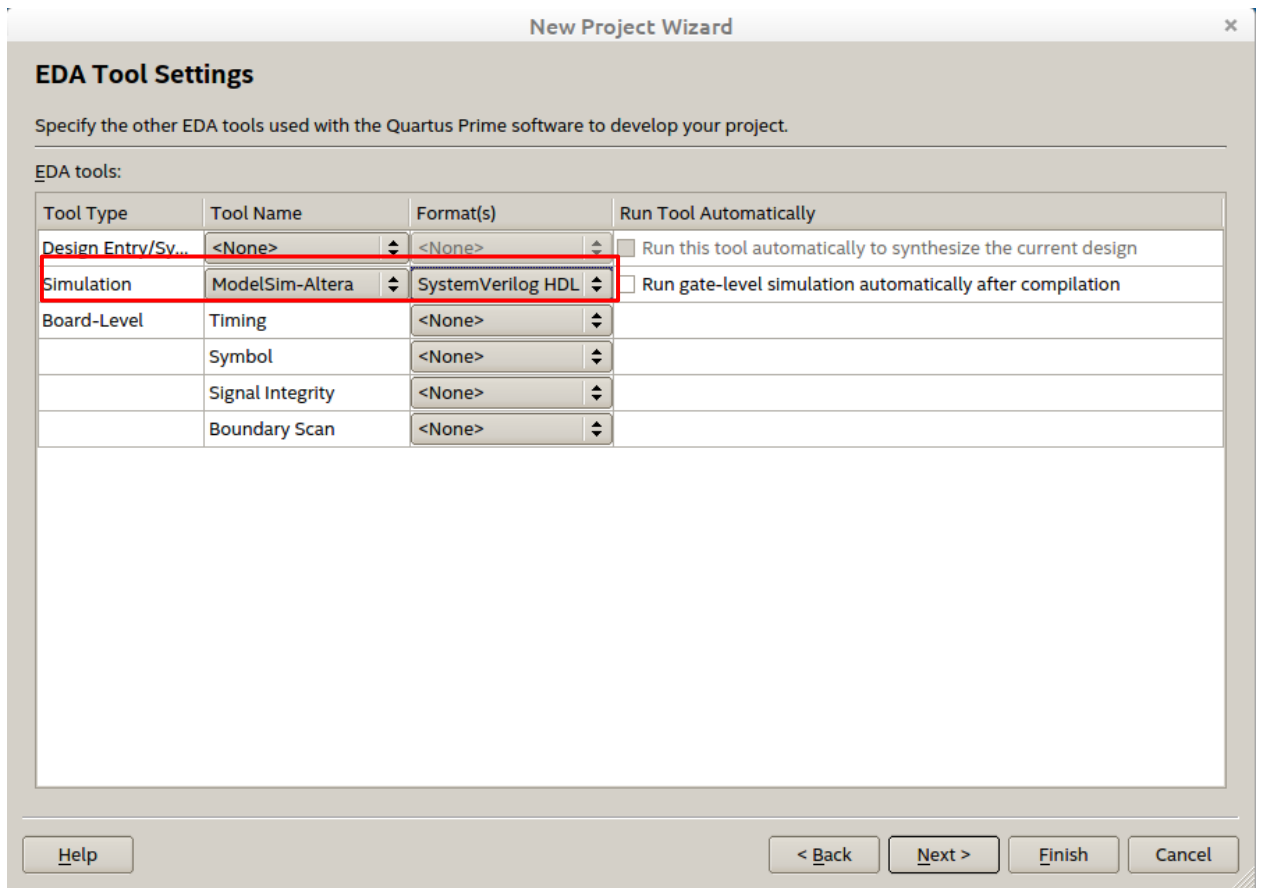


Рисунок 1.3 – Настройка инструментов

10. В результате появится окно (рисунок 1.4) Quartus с созданным проектом.
11. В дальнейшем для открытия существующего проекта необходимо использовать меню **File > Open Project...** и открыть файл **.qpf**.

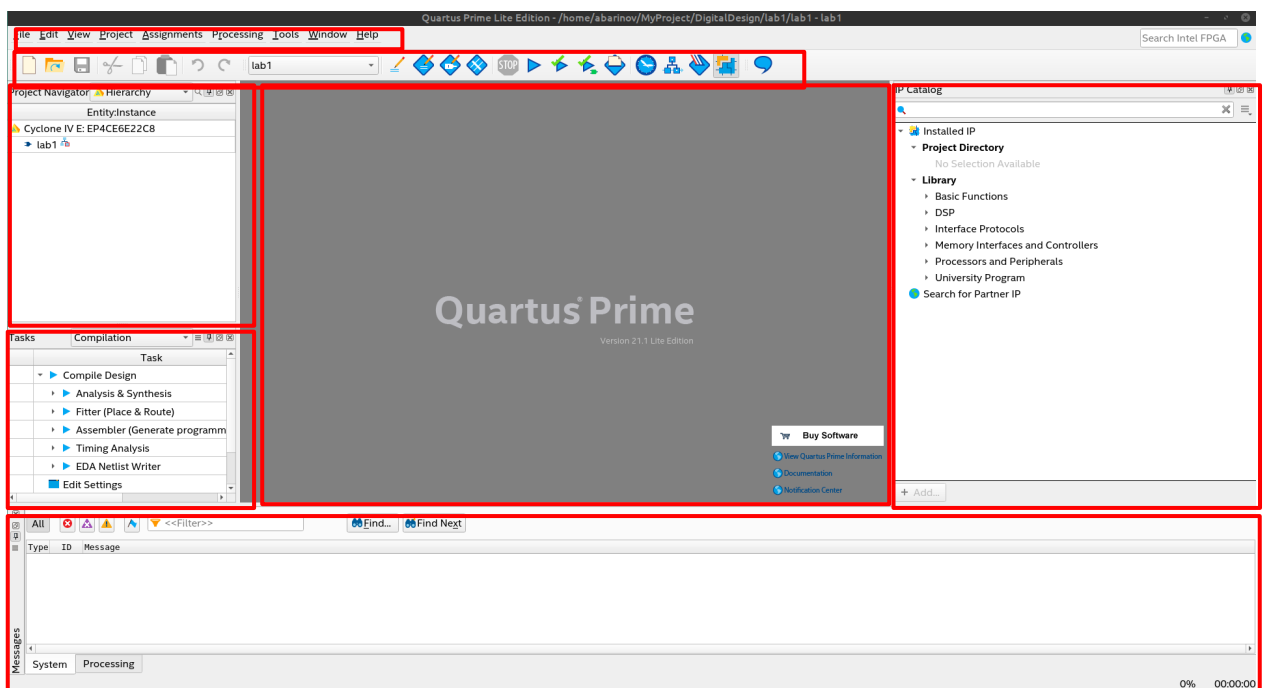


Рисунок 1.4 – Основное окно Quartus Prime:

1 – панель меню, 2 – панель с иконками, 3 – навигатор проекта, 4 – список выполняемых задач в процессе компиляции проекта, 5 – основная область, 6 – список IP-блоков (можно закрыть), 7 – журнал с сообщениями о текущем статусе, предупреждениях и ошибках в процессе компиляции проекта

## Создание BDF-файла и компиляция проекта

1. В навигаторе проекта Project Navigator измените представление с Navigator на Files.

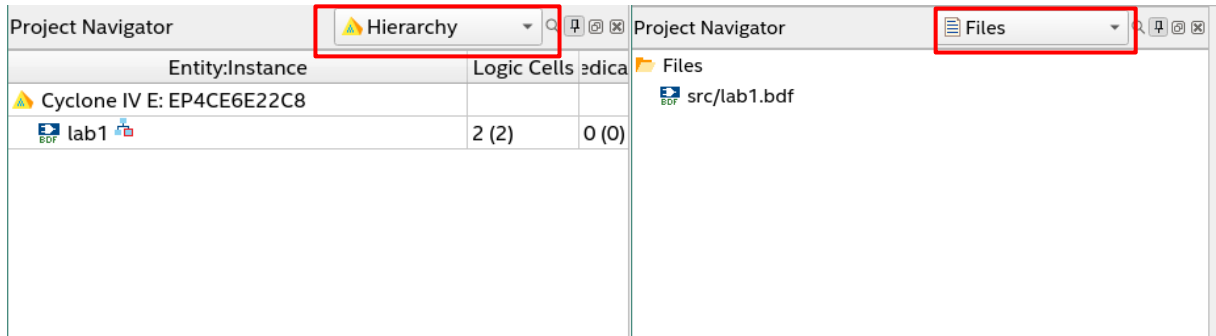


Рисунок 1.5 - Панель Project Navigator

2. Создайте файл схемного представления **File > New...**, выберите *Block Diagram / Schematic File*. Нажмите **OK**. В результате в главном окне появится область рисования схем с сеткой под именем **Block1.bdf**.

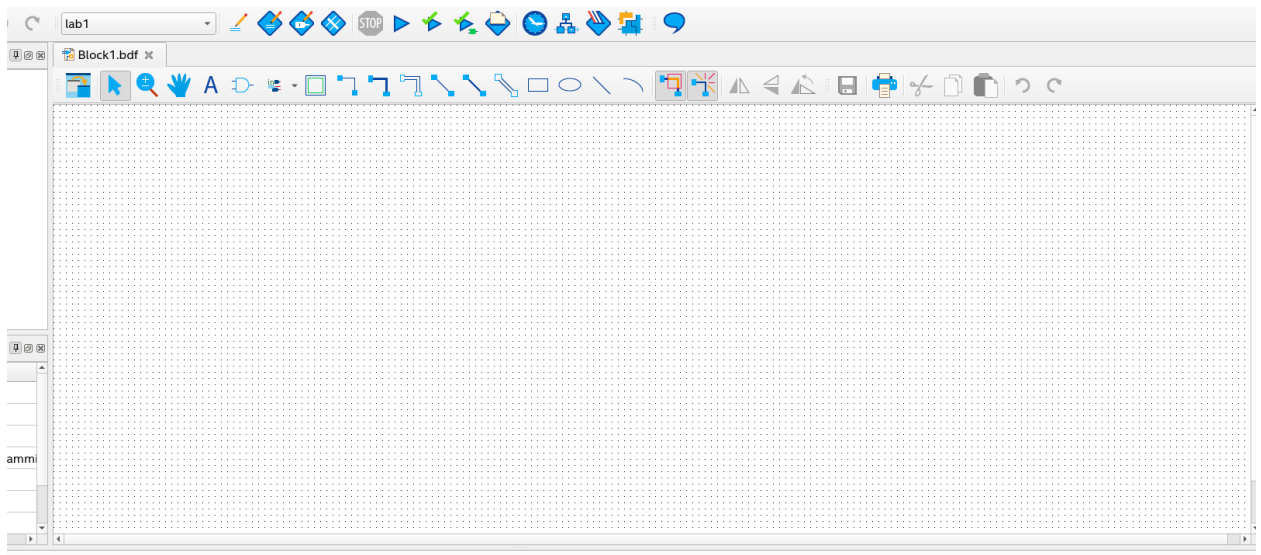





Рисунок 1.6 – Поле для рисования логической схемы

3. Инструментом **Symbol Tool**  добавьте из библиотеки *primitives > logic* необходимые логические вентили. При расположении элементов убедитесь, что их границы (штриховые области) не накладываются друг на друга.

4. Инструментом **Orthogonal Node Tool**  произведите соединение логических элементов проводниками.
5. Инструментом **Pin Tool** добавьте  входные и выходные пины. Двойным щелчком мыши по пину произведите переименование `pin_name` входных пинов на `x[0]`, `x[1]`, `x[2]`, `y[0]`, `y[1]` и `y[2]`.
6. Сохраните файл **File > Save**. При работе с проектом удобно файлы одинакового назначения сохранять в определённые папки (например, файлы HDL-описания в папку `/src`, файлы с тестами в папку `/testbench`, результаты моделирования в папку `/simulation` и так далее). Создайте папку в папке с проектом `src` и сохраните файл **lab1.bdf** в ней. Этот файл должен появиться в списке Project Navigator.

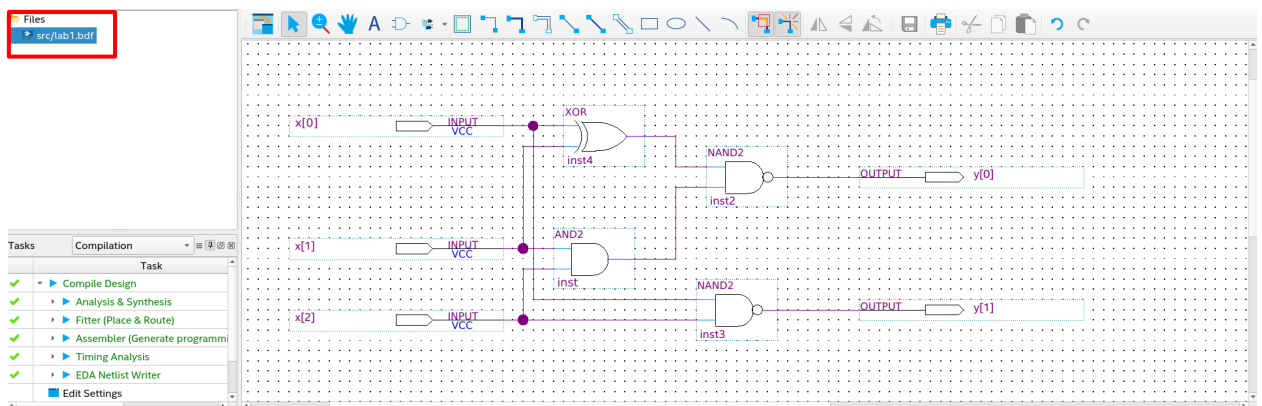



Рисунок 1.7 – Графическое изображение логической схемы

7. Произведите компиляцию проекта выбрав **Processing > Start Compilation**, либо нажав на  панели иконок. В процессе компиляции в области задач будут отображаться выполняемые операции. Если всё прошло хорошо, то в журнале сообщений не должно быть ошибок, однако будет порядка 14 предупреждений, что на текущем этапе не мешает работе.
8. После компиляции откроется окно с отчётом. Наиболее часто полезная информация из отчёта – это используемые ресурсы ПЛИС, то есть количество логических элементов, регистров, количество пинов, объёма памяти, количество умножителей и прочее, которые были задействованы в результате логического синтеза. Обратите внимание, что для используемого в описании примера схема состоит из 4 логических элементов, а в отчёте указано только 2.



Flow Summary	
Flow Status	Successful - Wed Sep 7 09:48:44 2022
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	lab1
Top-level Entity Name	lab1
Family	Cyclone IV E
Device	EP4CE6E22C8
Timing Models	Final
Total logic elements	2 / 6,272 (< 1 %)
Total registers	0
Total pins	5 / 92 (5 %)
Total virtual pins	0
Total memory bits	0 / 276,480 (0 %)
Embedded Multiplier 9-bit elements	0 / 30 (0 %)
Total PLLs	0 / 2 (0 %)

Рисунок 1.8 – Отчёт после процедуры компиляции проекта

9. Для просмотра RTL-представления схемы откройте **Tools > Netlist Viewers > RTL Viewer**. Обсудите с преподавателем, возникшее несоответствие в количестве элементов.

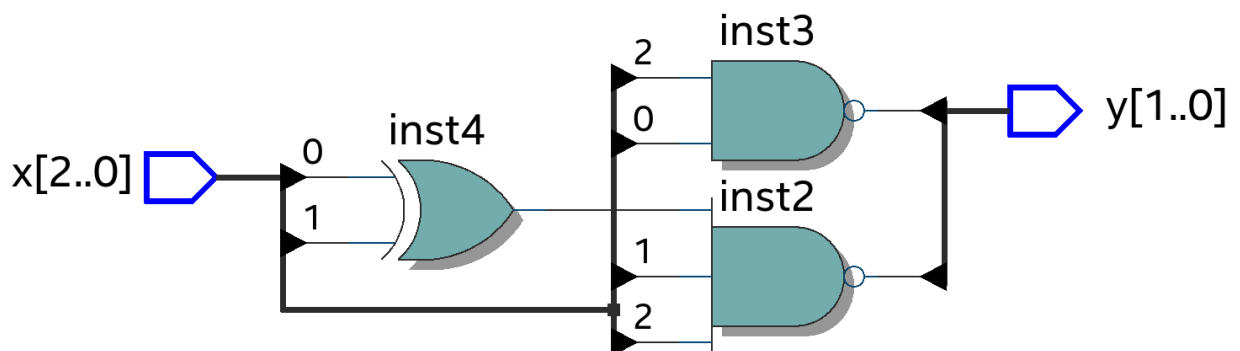


Рисунок 1.9 – RTL-представление схемы

10. Аналогично выберите **Tools > Netlist Viewers > Tecnology Map Viewer (Post-Mapping)**.

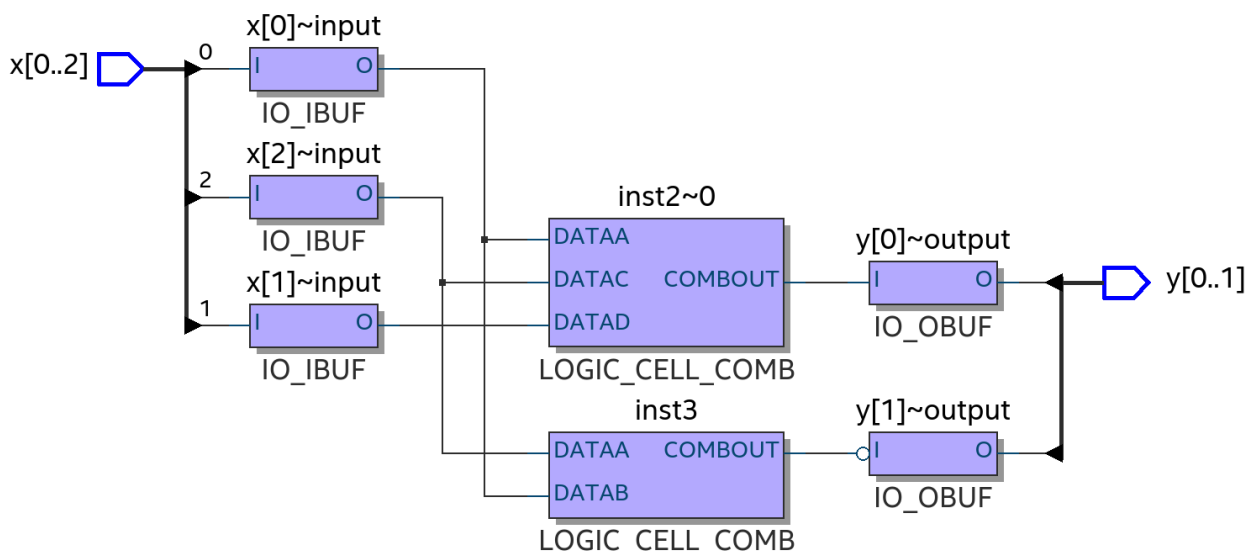


Рисунок 1.10 – Используемые логические элементы ПЛИС

- Проанализируйте каждую из  $\text{LOGIC\_CELL\_COMB}$ : правой кнопкой нажмите на логическую ячейку и в контекстном меню выберите **Properties**, затем в боковой панели выберите снизу вкладку **Truth Table**. Убедитесь, что таблица истинности совпадает с определённой Вами для каждого выхода  $y$ . Кроме того, на вкладке **Equation** можно увидеть логическое выражение, соответствующее данной таблице истинности (в нём знак  $\#$  означает операцию ИЛИ,  $!$  – инверсию,  $\&$  – И). На графическом изображении сверху можно увидеть RTL-представление реализованной логической функции.

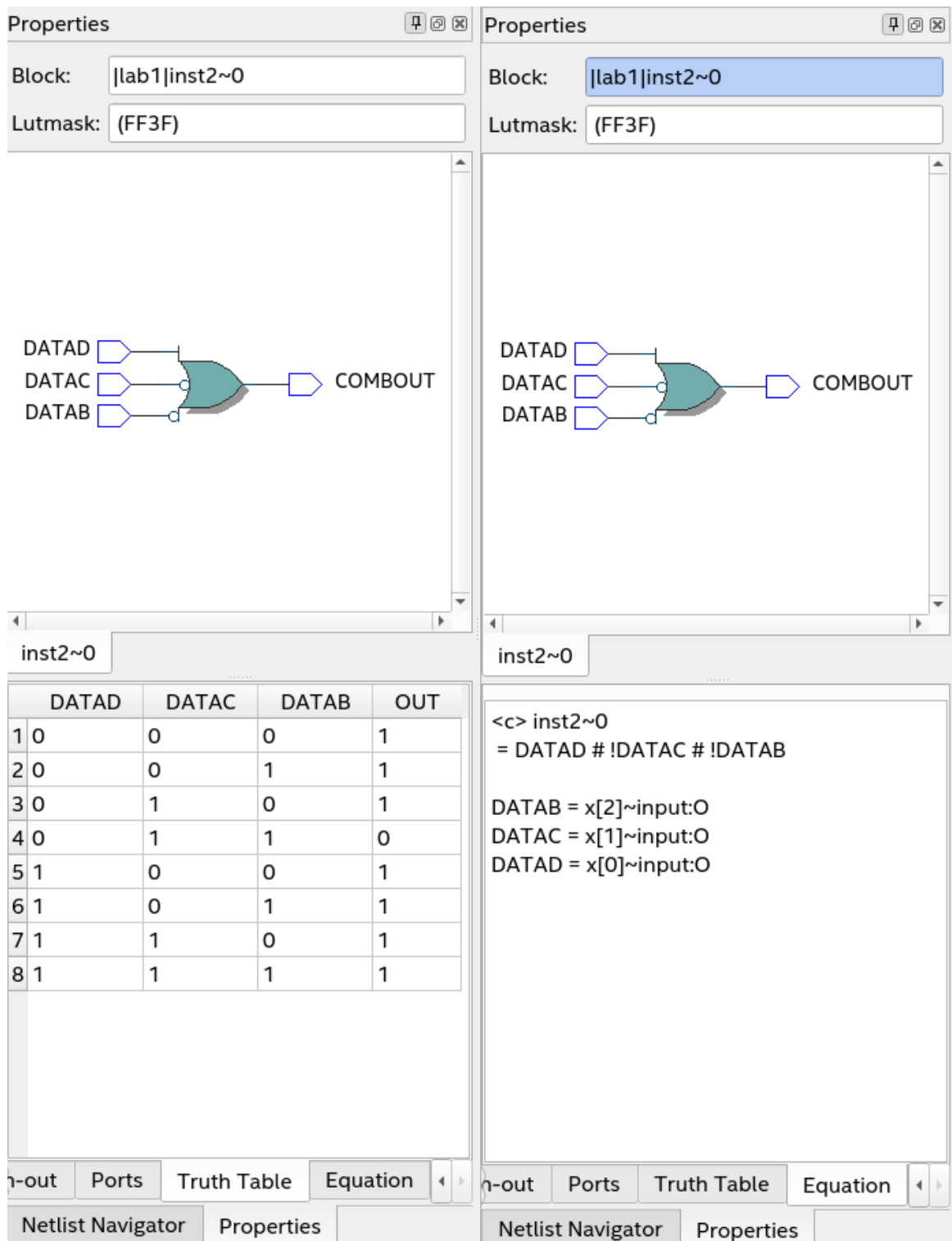


Рисунок 1.11 - Панель свойств логического элемента ПЛИС

## Прошивка ПЛИС

1. Инструментом **Assignments > Pin Planner** откройте окно назначения пинов. Для входных и выходных пинов поставьте в соответствие пины с ПЛИС, выбрав нужный в столбце Location. Тип контакта (I/O Standart) укажите 3.3-V LVTTTL для всех контактов. Для входных

сигналов рекомендуется использовать DIP переключатели. Для выходных - группу светодиодов. Соответствие пинов следует смотреть на отладочной плате. При этом на графическом представлении схемы будут появляться назначение пинов.

## 2. Закройте окно Pin Planner.

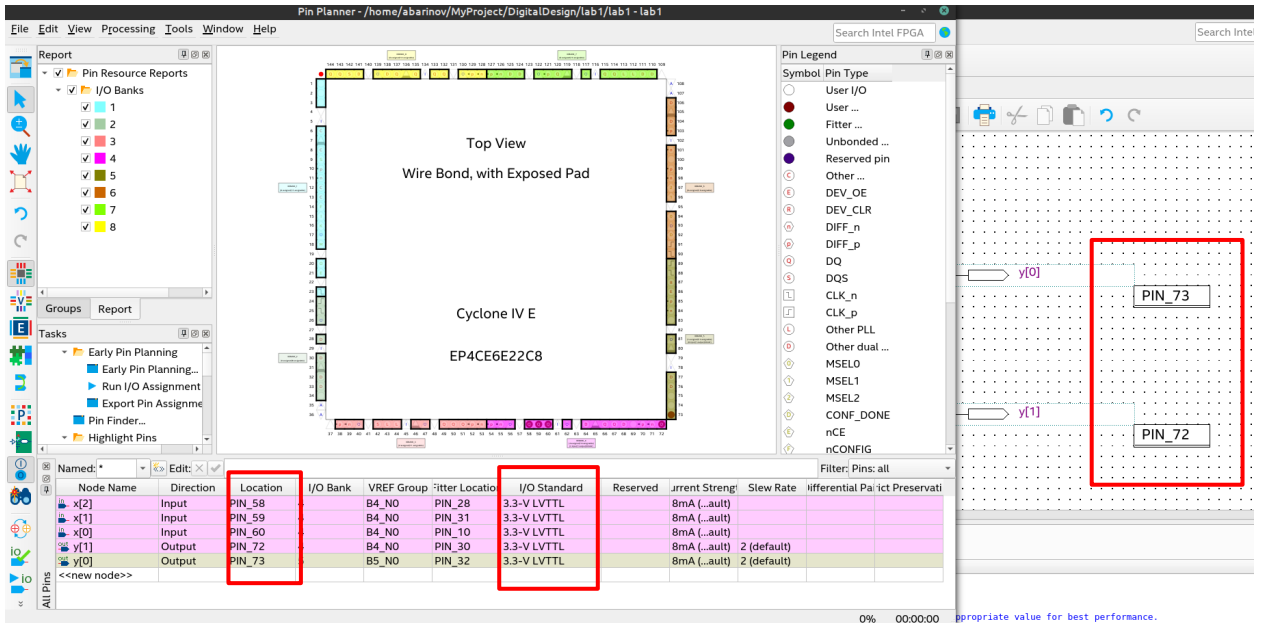


Рисунок 1.12 – Окно задания соответствия пинов Pin Planner

3. Инструментом **Assignments** > **Device** откройте окно настройки ПЛИС для перевода неиспользуемых пинов в третье состояние для избежания случайных коротких замыканий. В нём нажмите кнопку *Device and Pin Options...* В открывшемся окне выберите слева *Unused Pins* и в выпадающем меню укажите *As input tri-stated*. Нажмите *OK*, затем снова *OK*.

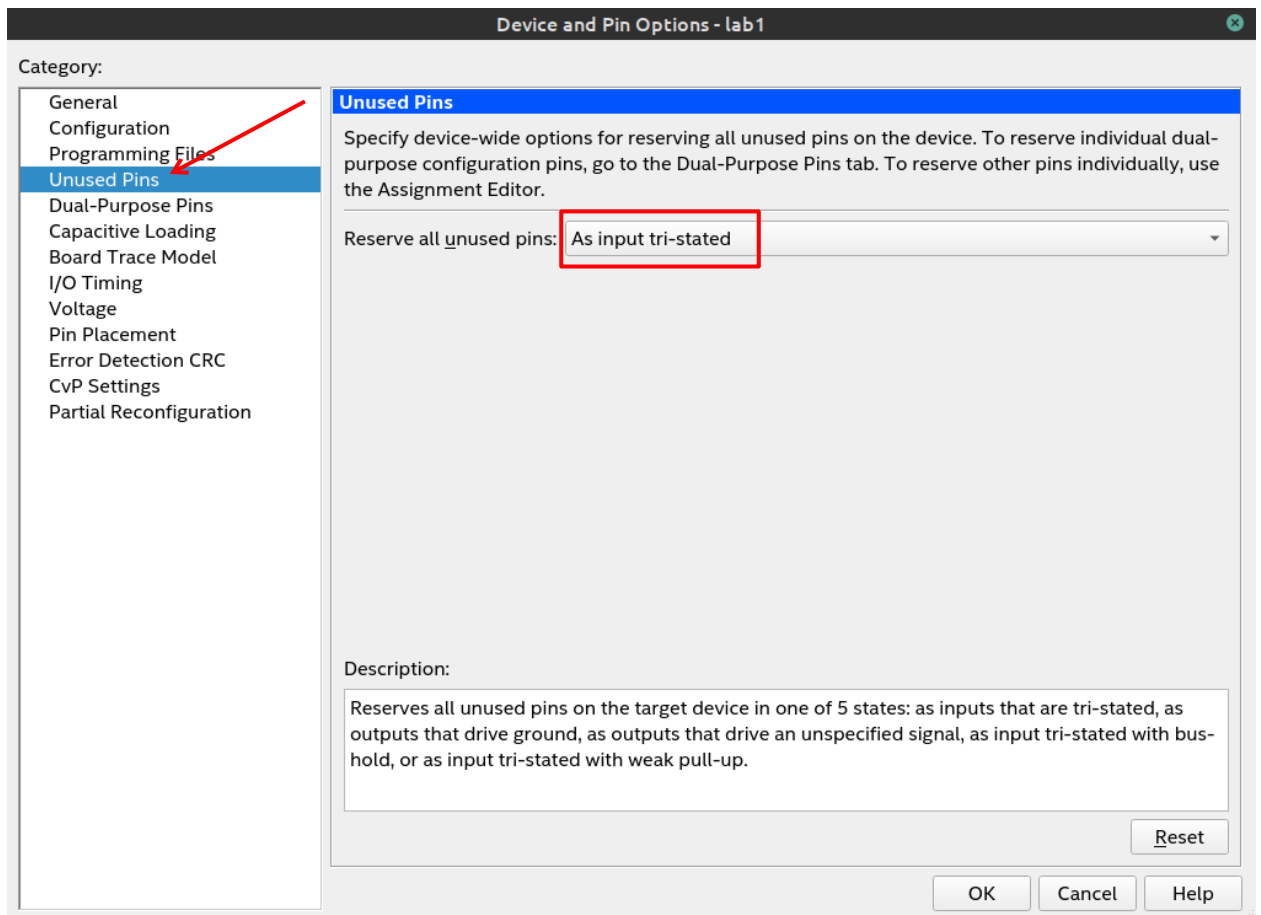


Рисунок 1.13 – Настройка неиспользуемых пинов ПЛИС

4. Подключите USB питание ПЛИС и программатор к компьютеру, а контакт программатора к контакту JTAG на ПЛИС и mini-USB к контакту mini USB на ПЛИС.
5. Снова произведите компиляцию проекта Processing > Start Compilation.
6. Инструментом **Tools > Programmer** откройте окно прошивки ПЛИС. В нём будет отображён файл прошивки **lab1.sof**, который сформировался в результате последней компиляции.

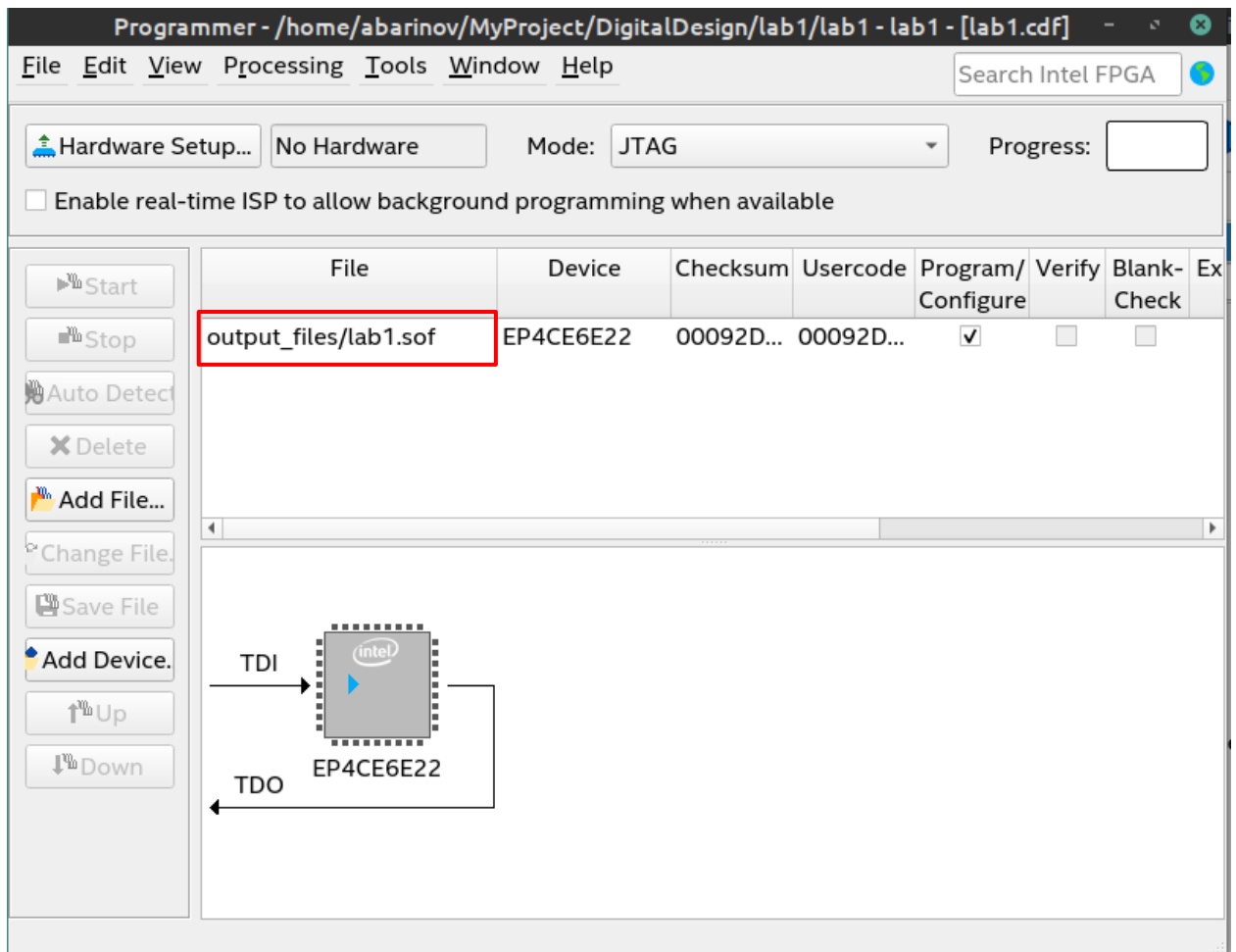


Рисунок 1.14 – Окно Programmer

7. Если прошивки ещё не было, то необходимо указать программатор. Для этого нажмите в открывшемся окне на кнопку *Hardware Setup* и в выпадающем меню выберите *USB Blaster*. Нажмите *Close*.

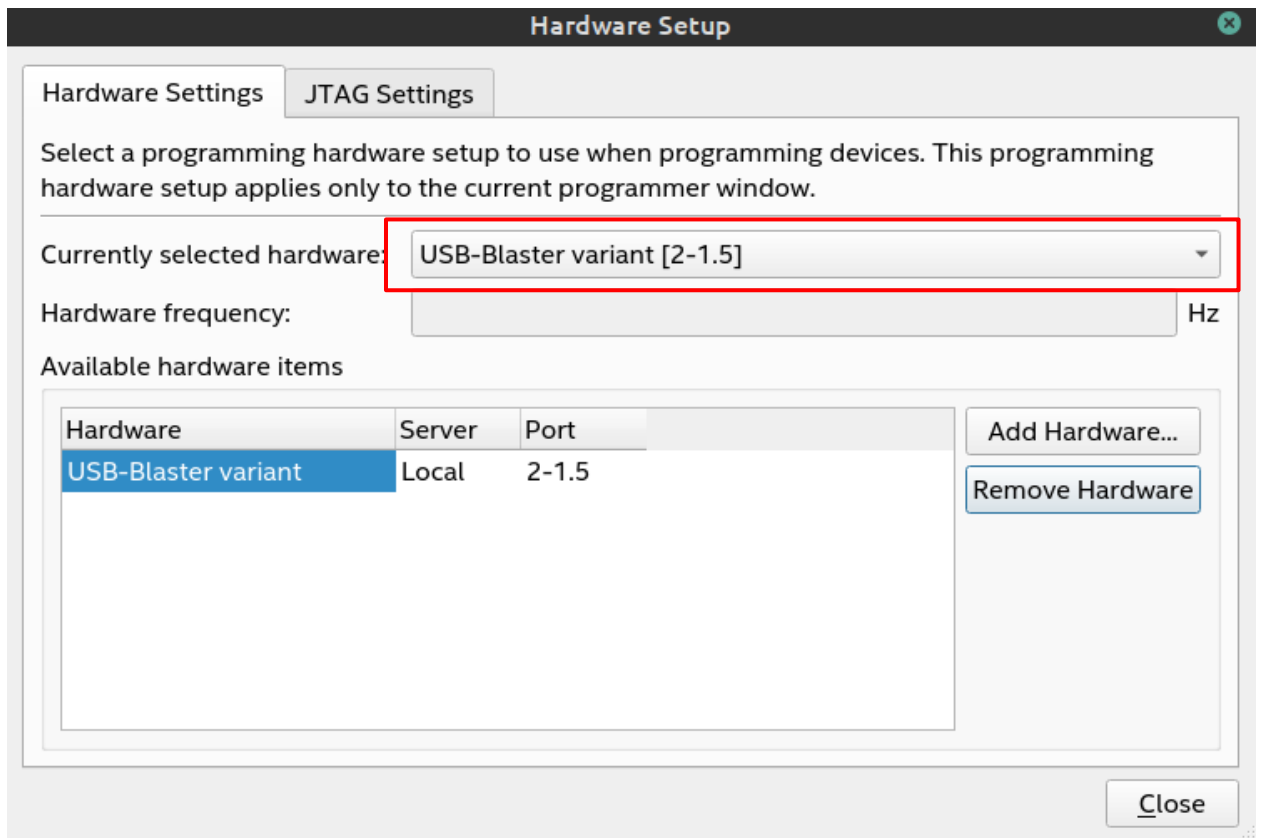


Рисунок 1.15 – Выбор программатора

- Для прошивки нажмите *Start*. Если всё прошло хорошо, то в окне Programmer должно отобразиться, что процесс завершился удачей, и ПЛИС должна включить светодиоды и начать реагировать на переключатели. В противном случае обратитесь к преподавателю.

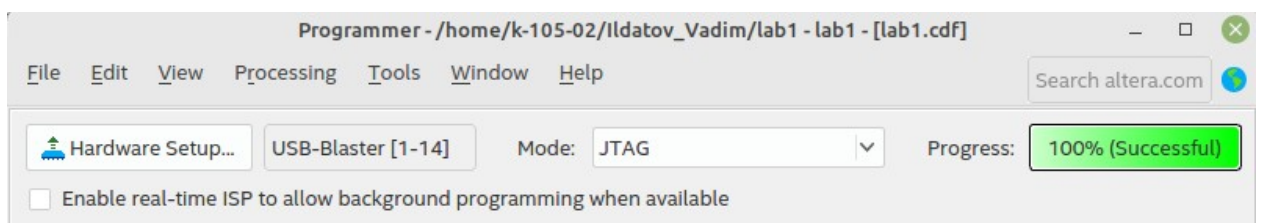


Рисунок 1.16 – Успех процедуры прошивки ПЛИС

Следует отметить, что в зависимости от времени производства отладочной платы состоянию, когда светодиод загорается, может соответствовать либо высокий уровень сигнала на выходе, либо низкий. Аналогично для переключателей: верхнему состоянию может соответствовать либо низкий уровень сигнала, либо высокий. В большинстве случаев для плат, используемых в работе, выполняется, что светодиод загорается при низком уровне сигнала на выходе, включённому состоянию

переключателей и нажатому состоянию кнопок также соответствует низкий уровень сигнала на входе. Определяется экспериментально.

## Создание файла SystemVerilog HDL

1. В том же проекте создайте новый файл **File > New...** Выберите *SystemVerilog HDL File*. На рабочей части откроется файл **SystemVerilog1.sv**.
2. Опишите в нём модуль `lab1_struct`. Сформируйте в нём структурное описание. Сохраните файл в папку `/src`. Имя файла должно совпадать с названием модуля `lab1_struct.sv`. Убедитесь, что файл стал отображаться в Project Navigator.

Используемые цифровые примитивы для структурного описания и логические выражения для поведенческого описания представлены в таблице 1.1. При использовании цифровых примитивов в скобках указываются сначала выходы, а затем входы. Количество входов не ограничено.

Таблица 1.1 - Логические операции и вентили, используемые в SystemVerilog-описании

Операция	Логическая функция	Поведенческое описание	Описание на вентильном уровне
НЕ	$y = \bar{a}$	$y = \sim a$	<code>not(y, a);</code>
И	$y = a \cdot b$	$y = a \& b$	<code>and(y, a, b);</code>
ИЛИ	$y = a + b$	$y = a   b$	<code>or(y, a, b);</code>
И-НЕ	$y = \overline{a \cdot b}$	$y = \sim( a \& b )$	<code>nand(y, a, b);</code>
ИЛИ-НЕ	$y = \overline{a + b}$	$y = \sim( a   b )$	<code>nor(y, a, b);</code>
ИСКЛЮЧАЮЩЕЕ ИЛИ	$y = a \oplus b$	$y = a \wedge b$	<code>xor(y, a, b);</code>
ИСКЛЮЧАЮЩЕЕ ИЛИ-НЕ	$y = \overline{a \oplus b}$	$y = \sim( a \wedge b )$	<code>xnor(y, a, b);</code>

Для схемы, используемой для примера на рисунке 1.7, описание модуля будет выглядеть следующим образом:

Листинг 1.1 – Структурное описание логической схемы

```

1 module lab1_struct(x, y); //объявление модуля
2
3 input logic [2:0] x; //входной сигнал
4 output logic [1:0] y; //выходной сигнал
5
6 logic [1:0] w; //внутренний сигнал
7 //список соединений
8 xor(w[0], x[0], x[1]);
9 and(w[1], x[1], x[2]);

```



```

10 nand(y[0], w[0], w[1]);
11 nand(y[1], x[0], x[2]);
12
13 endmodule

```

Здесь описание модуля начинается с ключевого слова на строке 1 `module`, имени модуля `lab1_struct`, а также указанием в скобках входных и выходных сигналов `X` и `Y`.

В строке 3 указано, что сигнал `X` является входным (`input`) типа `logic`, а также представляет собой трёхразрядную шину с индексацией младшего разряда (LSB, least significant bit, англ. *наименее значащий бит*) - 0, а старшего (MSB, most significant bit, англ. *наиболее значащий бит*) - 2. Аналогично в строке 4 для выходного (`output`) сигнала `Y` указано, что он двухразрядный типа `logic` с индексом старшего разряда - 1, а младшего - 0. Для задания портов входа / выхода используется конструкция:

```

1 <тип_порта> [тип_сигнала] [[MSB:LSB]] <имя_порта>;

```

В угловых скобках указаны *обязательные* атрибуты, в квадратных – *необязательные*. Тип порта может быть входным `input`, выходным `output` и двунаправленным `inout`. Тип сигнала в основном `logic`. Для многоразрядных сигналов (шины) необходимо указать разрядность и индексацию старшего и младшего разрядов `[MSB:LSB]`, для одноразрядного сигнала этого делать не нужно. Рекомендуется в проектах использовать единый порядок нумерации типа *big-endian*, например, для  $n$ -разрядной шины младший разряд - 0, старший –  $n-1$ .

В строке 6 введён дополнительный сигнал для наименования проводников, соединяющих логические вентили внутри схемы, которые не являются входными или выходными сигналами. Этот сигнал имеет тип данных `logic`. Внутри схемы примера всего два провода, в связи с чем сигнал `W` объявлен двухразрядным. По аналогии с портами сигналы записываются конструкцией

```

1 <тип_сигнала> [[MSB:LSB]] <имя_сигнала>;

```

В строках 8-11 записаны цифровые примитивы согласно схеме на рисунке 1.17.

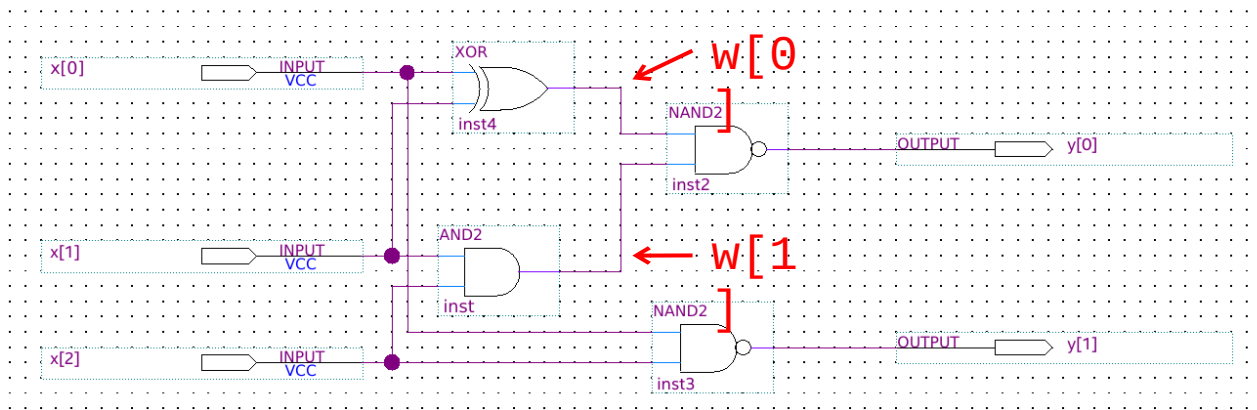


Рисунок 1.17 – Внутренние провода

Завершает описание модуля ключевое слово **endmodule** на строке 13.

По своей сути структурное описание является списком соединений (netlist).

3. Аналогично сформируйте файл **lab1\_beh.sv** с описанием поведенческой модели **lab1\_beh**.

Поведенческое описание отличается от структурного тем, что оно не предполагает необходимости описывать соединения и указывать используемые цифровые примитивы. Поведенческое описание связывает состояние на выходе схемы с состоянием на входе. С точки зрения нашей схемы для примера поведенческим описанием будет логическое выражение, связывающее сигналы  $y$  и  $x$ .

Для сигналов  $y[0]$  и  $y[1]$  такими выражениями согласно рисунков 1.7 и 1.17 будут:

$$y[0] = \overline{(x[0] \oplus x[1]) \cdot (x[1] \cdot x[2])},$$

$$y[1] = \overline{x[0] \cdot x[2]}.$$

В случае поведенческого описания согласно таблице 1.1 с примером записи логических выражений на SystemVerilog описание модуля будет выглядеть следующим образом.

Листинг 1.2 – Поведенческое описание логической схемы

```

1 module lab1_beh(x, y);
2
3 input logic [2:0] x;
4 output logic [1:0] y;
5
6 logic [1:0] w;
7
8 //используем непрерывное присваивание

```


```

9 assign y[0] = ~( ( x[0] ^ x[1] ) & ( x[1] & x[2] ) );
10 assign y[1] = ~( x[0] & x[2] );
11
12 endmodule

```

Для простой комбинационной логики, которой является наша схема для примера, используется ключевое слово **assign**, которое определяет *непрерывное присваивание* сигналу слева от знака присваивания (=) значение выражения справа от него. Таким образом для задания простой логики используется конструкция

```
1 assign <сигнал> = <выражение>;
```

4. В отличие от задания 1, в случае отсутствия необходимости прошивать ПЛИС и проводить временной анализ разрабатываемого устройства, вместо полной компиляции проекта достаточно произвести только первый этап анализа и синтеза. Для этого выберите в Project Navigator файл **lab1\_struct.sv**, вызовите контекстное меню и выберите **Set as Top-Level Entity**. **Компиляция, обработка, анализ и синтез всегда проводятся для модуля верхнего уровня!** На панели иконок выберите Start Analysis &  Synthesis для формирования RTL-представления схемы.
5. По окончании в случае отсутствия ошибок проведите анализ RTL- и технологического представлений аналогично заданию 1. Если всё сделано верно, то три описания (графическое, структурное и поведенческое) должны дать идентичные результаты.

Примечание: при работе с HDL-кодом удобно использовать специальные редакторы с подсветкой синтаксиса (например, NotePad++ для ОС Windows или Kate, Xed, Vim, Visual Studio Code и прочие для ОС GNU/Linux). В этом случае созданные файлы необходимо добавить в проект Quartus вручную, вызвав в Project Navigator контекстное меню у папки Files и выбрав **Add/Remove Files in Project...**

## Написание тестового модуля

1. Сформируйте в проекте файл для тестирования **tb\_lab1.sv**, расположите его в папке /src.

*Тестбенч* представляет собой отдельный отладочный модуль без входных и выходных сигналов, в котором подключается тестируемый модуль. Данный модуль используется для проверки кода в HDL. Задача тестбенча – сгенерировать входные сигналы для тестируемого устройства, а затем принять выходные и проанализировать их.

Обобщённая структура тестового модуля может быть представлена следующим образом:

- *подключение файлов с модулями*

Для больших проектов может быть много файлов модулей. Для того, чтобы модуль тестбенча «увидел» тестируемый модуль может понадобиться подключить к модулю тестбенча файл с тестируемым модулем. Для этого используется препроцессорная директива ``include`

```
1 `include "<имя_файла>"
```

- *задание параметров временного масштаба при помощи препроцессорной директивы ``timescale`*

```
1 `timescale <масштаб_времени> / <точность_моделирования>
```

Здесь *масштаб времени* определяет множитель, задающий изменения сигналов во времени, записанные в тестбенче. Например, если масштаб времени указан 1 нс, а временная задержка записана как `#50`, это означает, что выдерживается пауза длительностью 50 нс.

*Точность моделирования* задаёт шаг моделирования.

- *объявление входных и выходных сигналов*

Для подключения к тестируемому модулю изменяющихся во времени сигналов используется сигнал с типом данных `logic`.

- *подключение тестируемого модуля или модулей*

При подключении устройств используется подключение внешних сигналов к соответствующим портам и конструкция выглядит следующим образом

```
1 <имя_модуля> <имя_экземпляра> (.<имя_порта> (<имя_сигнала>)[ ,  
2 .<имя_порта> (<имя_сигнала>),  
3 .<имя_порта> (<имя_сигнала>),  
4 ...] );
```

При подключении нескольких идентичных модулей `<имя_экземпляра>` позволит отличить один от другого. Так как мы подключаем модули для тестирования, то распространённой практикой является наименование экземпляров как UUT (unit under test), DUT (design

under test) или CUT (circuit under test). Имена экземпляров должны различаться! Можно их нумеровать в виде DUT1, DUT2, DUT3 и так далее.

Например, требуется подключить модуль `test` с входами `a`, `b` и выходом `c` к тестируемому модулю. Тогда описание будет выглядеть так

```
1 logic at, bt, ct;  
2 test DUT ( .a(at), .b(bt), .c(ct));
```

- задание изменения входных сигналов во времени

Изменение сигналов во времени можно задавать достаточно большим количеством способов как простых, так и сложных. Сейчас лучше начать с простых. Для задания изменения сигнала используется специальный процедурный блок `initial`, который окружается процедурными скобками `begin...end` (или `fork...join`, но последние на текущий момент не требуются).

Каждый `initial`-блок запускается однократно в начале моделирования и выполняются они *параллельно*. Внутри процедурных скобок `begin...end` операции выполняются *последовательно*<sup>1</sup>. Примеры задания различных видов сигналов приведены в Приложении А.

Для каждого входного сигнала рекомендуется использовать отдельный процедурный блок `initial`. Также в файле тестбенча можно указать время моделирования, которое удобно записать в отдельный `initial`-блок в виде

```
1 initial  
2 #200 $stop; //останавливаем моделирование через 200 единиц времени
```

- задание вывода информации в терминал

Зачастую помимо вывода временных диаграмм (waveform) и их анализа, полезным бывает вывод информации об изменениях сигналов, их значениях, результатах сравнения в терминале. Для вывода текста в терминал часто используются функции `$display` и `$monitor`.

Функция `$display` выводит информацию в терминал однократно в момент её вызова. Функция `$monitor` выводит информацию в терминал в момент изменения входных сигналов («мониторит» изменения).

Синтаксис обеих функций схож с синтаксисом языка С

```
1 $display(«текст», [перечень сигналов или переменных]);
```

---

<sup>1</sup> Внутри `fork...join` - параллельно.

```
2 $monitor(«текст», <перечень сигналов или переменных>);
```

Пример описания тестового модуля представлен в листингах 1.3 и 1.4.

Листинг 1.3 – Формирование файла верхнего уровня

```
1 //препроцессорная директива для подключения файлов модулей
2 `include "lab1_struct.sv"
3 `include "lab1_beh.sv"
4
5 module top(x, y_struct, y_beh);
6
7 //объявляем входные сигналы
8 input logic [2:0] x;
9
10 //объявляем выходные сигналы
11 output logic [1:0] y_struct, y_beh;
12
13 //подключаем устройства
14 lab1_struct U1 ( .x( x ), .y( y_struct ) );
15 lab1_beh U2 ( .x( x ), .y( y_beh ) );
16
17 endmodule
```

Листинг 1.4 – Описание тестового модуля для схемы из примера

```
1 //препроцессорная директива для подключения файлов модулей
2 `include "top.sv"
3
4 //препроцессорная директива для определения временного масштаба
5 `timescale 1ns / 1ps
6
7 module tb_lab1;
8
9 //объявляем входные данные
10 logic [2:0] xt;
11
12 //объявляем выходные данные
13 logic [1:0] yt_struct, yt_beh;
14
15 //подключаем устройства
16 top DUT (.x(xt), .y_struct(yt_struct), .y_beh(yt_beh));
17
18 //задаём изменение входного сигнала 'xt' с шагом в 10 нс
19 initial begin
20     xt = 3'b000; //в начале моделирования сигнал 'xt' равен 000
21     #10 xt = 3'b001; //ждём 10 нс и меняем сигнал на 001
22     #10 xt = 3'h7; //спустя ещё 10 нс сигнал равен 111
23     #10 xt = 3'h4; //через 10 нс сигнал меняется на 100
24 end
25
26 //задаём время моделирования 50 нс и останавливаем его
27 initial
28     #50 $stop;
29
30 //отмечаем в терминале значения сигналов при каждом их изменении
31 //или изменении времени
32 initial begin
33     $display("Start test");
```

```

34     $monitor($time, " x = %b and y_struct = %b, y_beh = %b", xt,
yt_struct, yt_beh);
35     end
36
37     endmodule

```

2. Проведите моделирование модуля инструментом ModelSim. Для запуска теста в ModelSim следует пройти следующие шаги.

2.1. Откройте окно настроек **Assignments > Setting...**, вкладку *EDA Tool Settings > Simulation*

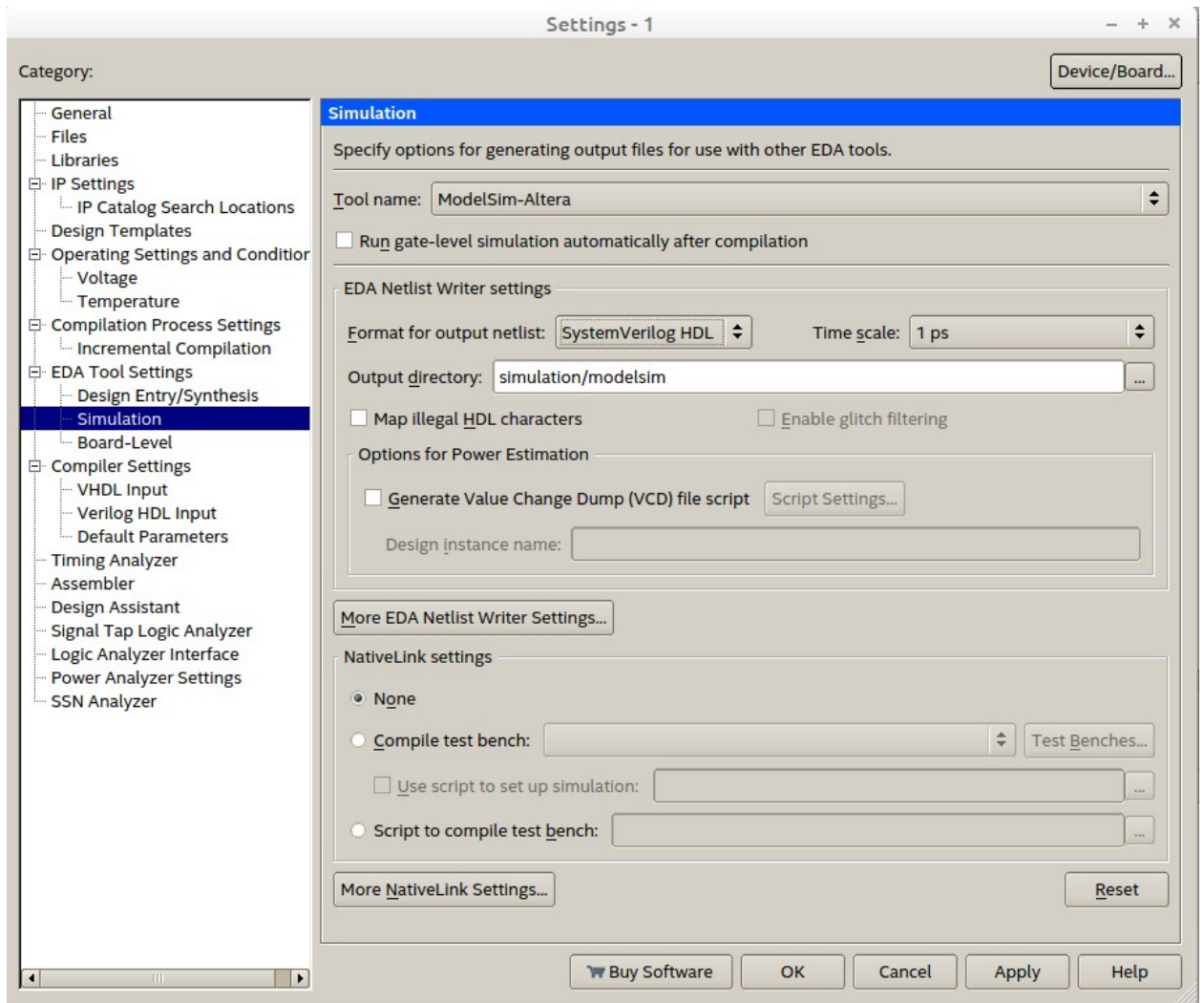


Рисунок 1.18 – Окно настроек параметров моделирования

2.2. В поле *Tool name* убедитесь, что указан ModelSim-Altera.

2.3. В блоке NativeLink settings выберите Compile test bench. Нажмите кнопку *Test Benches...*

2.4. В открывшемся окне нажмите *New...*

- 2.5. Задайте (рисунок 1.19) имя тесту (1), укажите название тестового модуля (2), отметьте моделирование пока используются все тестовые вектора (3), добавьте файл с описанием тестового модуля (4).
- 2.6. Нажмите *ОК*.
- 2.7. В окне списков тестбенчей должен появиться добавленный тест. Нажмите в нём *ОК*.
- 2.8. В окне настроек также нажмите *ОК*.
- 2.9. Произведите синтез модуля, который необходимо протестировать.
- 2.10. Запустите моделирование **Tools > Run Simulation Tool > RTL Simulation**.

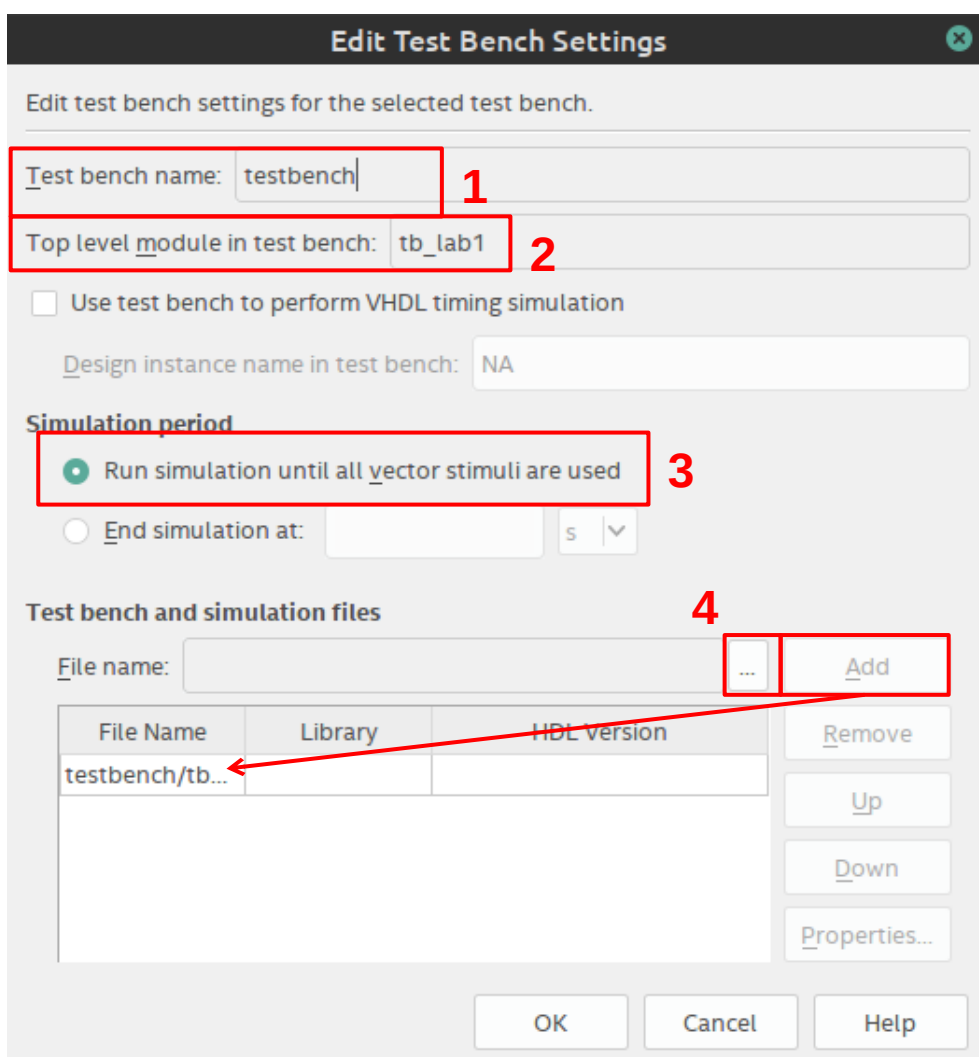



Рисунок 1.19 – Настройка тестбенча

- 2.11. Если всё прошло нормально, то должно открыться окно ModelSim с временными диаграммами и сообщениями в терминале. Как можно видеть на рисунке 1.20 в области построения временных



диаграмм появились входные и выходные сигналы с диаграммами (для того, чтобы диаграммы уместились в поле, нажмите *Zoom Full*  на панели выше или клавишу *F* при активном окне осциллограмм).

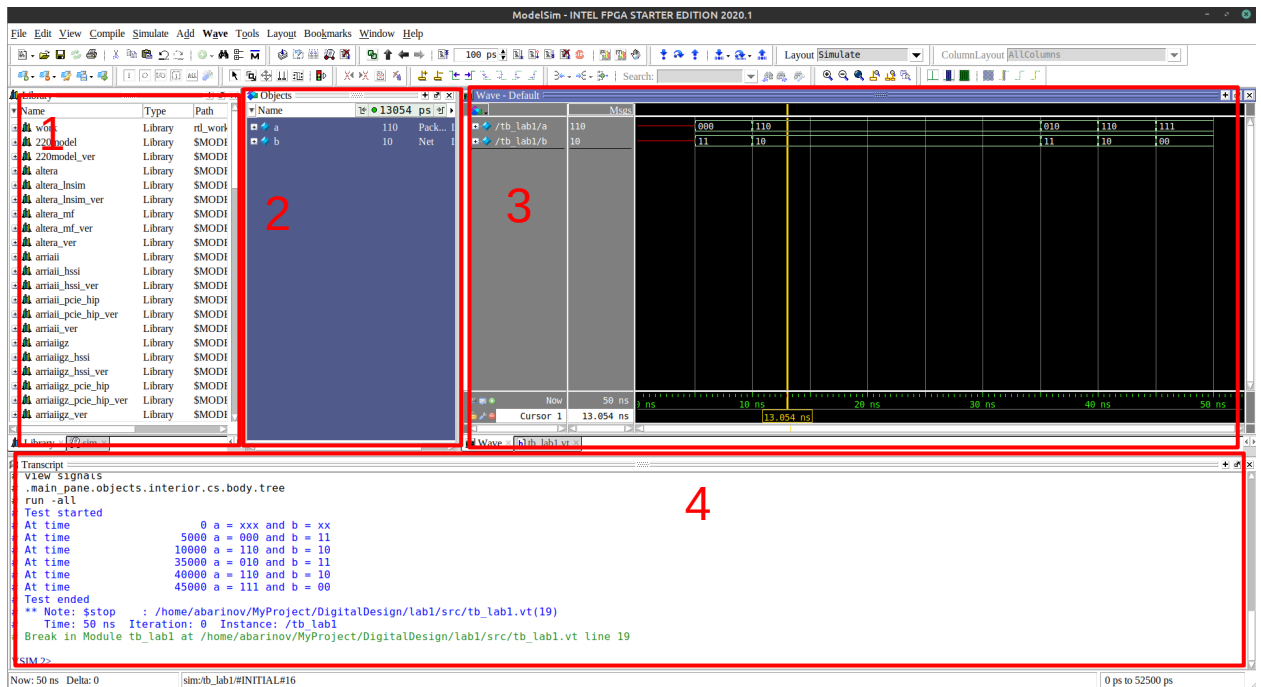


Рисунок 1.20 – Окно ModelSim:

1 – дерево библиотек с рабочей папкой work; 2 – список сигналов (входных и выходных); 3 – временные диаграммы; 4 – терминал

## Упражнения

Выполните задание согласно варианту, представленному в таблице 1.2:

1. Для заданных логических функций сформируйте в Quartus Prime BDF-файл (block diagram file), произведите для отладочной платы ПЛИС семейства Cyclone IV сопоставление пинов в Pin Planner. Составьте таблицу истинности, продемонстрируйте работу схемы на отладочной плате. *Указание:* при подключении входных и выходных портов поставьте перед ними инверторы для удобства восприятия работы платы.

2. Для этих же функций сформируйте два файла описания на языке SystemVerilog: с применением логических операций и в виде списка соединений. Объедините их в один модуль в файле top.sv.
3. Для п. 2 напишите testbench, произведите моделирование в ModelSim. Продемонстрируйте идентичность результатов моделирования для двух видов спроектированных модулей.

Таблица 1.2 – Исходные данные для задания

№ варианта	Логические выражения
1, 8, 15, 22	$y_0 = \overline{x_0} + x_1,$ $y_1 = x_0 \oplus x_1,$ $y_2 = \overline{x_0} + \overline{x_1} + x_2.$
2, 9, 16, 23	$y_0 = \overline{x_0} \oplus x_1,$ $y_1 = \overline{x_2} (x_0 + \overline{x_1}),$ $y_2 = x_0 \cdot \overline{x_1} + x_2.$
3, 10, 17, 24	$y_0 = x_0 \oplus \overline{x_1},$ $y_1 = \overline{x_0} \cdot (x_1 + x_2),$ $y_2 = (x_0 \oplus x_1) \cdot x_2.$
4, 11, 18, 25	$y_0 = \overline{x_0} \cdot (x_1 \oplus x_2),$ $y_1 = x_1 \oplus (\overline{x_0} \cdot x_2),$ $y_2 = x_0 \cdot x_1 + \overline{x_2}.$
5, 12, 19, 26	$y_0 = x_0 \oplus x_1 \oplus \overline{x_2},$ $y_1 = x_0 + \overline{x_2},$ $y_2 = \overline{x_0} + \overline{x_1} \cdot x_2.$
6, 13, 20, 27	$y_0 = \overline{x_0} + \overline{x_1},$ $y_1 = x_0 \oplus x_1,$ $y_2 = \overline{x_0} + \overline{x_1} \oplus x_2.$
7, 14, 21, 28	$y_0 = \overline{x_1} + x_2,$ $y_1 = x_0 \oplus x_2,$ $y_2 = \overline{x_0} \oplus \overline{x_1} + x_2.$

## Работа № 2. Комбинационные схемы на уровне регистровых передач. Блокирующее присваивание

Работа № 1 была посвящена проектированию простых комбинационных схем. Если в качестве поведенческого описания такой схемы можно задать логическое выражение, то используется оператор непрерывного присваивания `assign`.

Однако для разных логических схем можно использовать иное представление поведенческого описания по отношению к логической функции. Например, простой мультиплексор «из 2 в 1» (рисунок 2.1) можно описать его логической функцией

$$F = \bar{S} \cdot D_0 + S \cdot D_1.$$

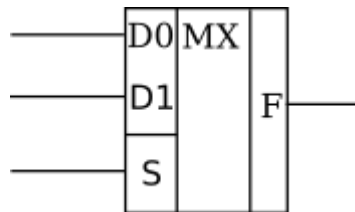


Рисунок 2.1 – Мультиплексор «из 2 в 1»

В этом случае сигнал  $F$  легко описывается оператором непрерывного присваивания

Листинг 2.1 – Описание мультиплексора «из 2 в 1»

```
1 module mux21_assign(D, S, F);
2
3 input logic [1:0] D;
4 input logic S;
5 output logic F;
6
7 assign F = ( ~S & D[0] ) | ( S & D[1] );
8
9 endmodule
```

Для мультиплексора «из 4 в 1» логическая функция будет уже длиннее

$$F = \bar{S}_0 \cdot \bar{S}_1 \cdot D_0 + S_0 \cdot \bar{S}_1 \cdot D_1 + \bar{S}_0 \cdot S_1 \cdot D_2 + S_0 \cdot S_1 \cdot D_3.$$

И такую функцию записывать сложнее.

Листинг 2.2 – Описание мультиплексора «из 4 в 1»

```
1 module mux41_assign(D, S, F);
2
3 input logic [3:0] D;
```

```

4 input logic [1:0] S;
5 output logic      F;
6
7 assign F = ( ~S[0] & ~S[1] & D[0] ) | ( S[0] & ~S[1] & D[1] ) | ( ~S[0] &
           S[1] & D[2] ) | ( S[0] & S[1] & D[3] );
8
9 endmodule

```

## Оператор ветвления `if...else`

Строго говоря, описание работы мультиплексора можно описать *принципом* его работы. В таком случае, можно представить себе, что мультиплексор является схемой выбора и формализовать его поведенческое описание в виде: *если*  $S = 00$ , *то*  $F = D_0$ , *иначе если*  $S = 01$ , *то*  $F = D_1$ , *иначе если*  $S = 10$ , *то*  $F = D_2$ , *иначе*  $F = D_3$ . В языке SystemVerilog для такой формализации используется последовательный оператор ветвления `if...else if... else....` Этот оператор записывается внутри процедурных блоков `always_comb`, `always_latch` и `always_ff`<sup>2</sup>.

В работе 1 мы познакомились с процедурным блоком `initial`, который запускается однократно в начале моделирования. В отличие от него процедурный `always`-блок выполняется всегда, когда изменяется хотя бы один сигнал, находящийся в его *списке чувствительности*. Список чувствительности содержит перечень сигналов, изменение которых должны запускать выполнение `always`-блока. Синтаксис этого процедурного блока

```

1 always_* @(a or b)
2 begin
3   ...
4 end

```

Здесь вместо `*` следует подставить `comb`, `latch` или `ff`. Значок `@` означает наступление события (event), которое задано далее в круглых скобках.

Полезно вспомнить, что комбинационные схемы определяются тем, что значения выходных сигналов зависят от комбинации значений всех входных сигналов. С точки зрения процедурного `always`-блока это означает, что в список чувствительности для комбинационных схем входят все входные сигналы. Для того чтобы их не перечислять каждый раз, используется сокращённый синтаксис

```

1 always_comb
2 begin
3   ...

```

<sup>2</sup> Если речь идёт о любых таких блоках, в тексте практикума они объединяются одним словом «`always`-блок»

4 end

В примере с мультиплексором, выход  $F$  меняет своё значение всегда, когда меняется хотя бы один из входов  $D$  или  $S$ . Значит, эти сигналы и будут в списке чувствительности.

Сам синтаксис оператора ветвления выглядит как

```
1 if (<условие>) begin
2   <действия, если условие истинно>
3 end
4 else begin
5   <действия, если условие ложно>
6 end
```

Таким образом, мультиплексор «из 4 в 1» можно описать следующим образом

Листинг 2.3 – HDL-описание мультиплексора «из 4 в 1» с оператором `if...else`

```
1 module mux41_if(D, S, F);
2
3   input  logic    [3:0] D;
4   input  logic    [1:0] S;
5   output logic    F;
6
7   always_comb begin
8     if (S == 2'b00)
9       F = D[0];
10    else if (S == 2'b01)
11      F = D[1];
12    else if (S == 2'b10)
13      F = D[2];
14    else
15      F = D[3];
16  end
17
18 endmodule
```

В данном листинге в операторе `if` используется сравнение сигнала  $S$  с тем или иным значением (операторы сравнения представлены в таблице 2.1). Процедурный блок `always_comb` срабатывает при изменении сигналов  $D$  и  $S$ , после чего происходит проверка текущего значения  $S$  и вывод сигнала  $D[S]$  на выход.

Таблица 2.1 – Операторы сравнения в SystemVerilog

Сравнение	Запись
равенство	$A == B$
не равенство	$A != B$
больше	$A > B$

меньше	A < B
больше или равно	A >= B
меньше или равно	A <= B

При работе с оператором `if...else` следует иметь в виду, что он выполняется последовательно. Это означает, что при входе в `always`-блок проверяются условия последовательно и при выполнении условия осуществляется выход из этого блока.

### Оператор выбора `case`<sup>3</sup>

Оператор ветвления полезен при сравнении небольшого количества условий. Если же их много, то структура оператора «лесенкой» выглядит неудачно. Для большого числа сравнений и для наглядности удобно использовать последовательный оператор выбора `case`. Как и оператор ветвления он выполняется внутри процедурного `always`-блока.

Работа мультиплексора из нашего примера на базе оператора выбора может быть формализована как: *в зависимости от значения сигнала S на выходе появляется одно из значений входного сигнала.*

Синтаксис оператора выбора следующий

```

1 case (<сигнал, по которому осуществляется выбор>)
2   <условие_1> : <действие, в случае условия 1>;
3   <условие_2> : begin
4     <действия, в случае условия 2>;
5     end
6   ...
7   default : <действие, если ни одно условие не срабатывает>;
8 endcase

```

В таком случае мультиплексор из нашего примера может быть описан как представлено в листинге 2.4.

Листинг 2.4 – HDL-описание мультиплексора «из 4 в 1» с оператором `case`

```

1 module mux41_case(D, S, F);
2
3   input  logic    [3:0] D;
4   input  logic    [1:0] S;
5   output logic    F;
6
7   always_comb begin
8     case ( S )
9       2'b00 : F = D[0];
10      2'b01 : F = D[1];
11      2'b10 : F = D[2];

```

<sup>3</sup> Разновидности `casex` и `casez` здесь не рассматриваются

```

12         2'b11 : F = D[3];
13     default : F = 1'bx;
14 endcase
15 end
16
17 endmodule

```

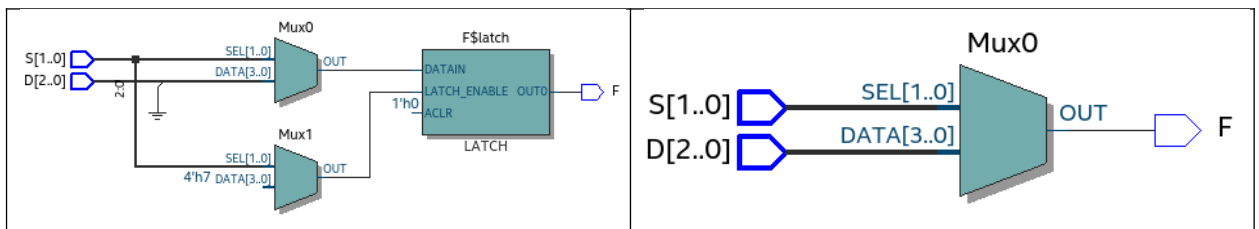
## Появление D-защёлок (D-latch)

Для мультиплексора «из 4 в 1» мы перебрали все возможные комбинации сигнала *S*, возникает вопрос о целесообразности этого пункта. Следует отметить необходимость в комбинационных схемах, разрабатываемых при помощи процедурного `always_comb`-блока, учесть все возможные комбинации входных сигналов. В случае их отсутствия в операторе `case` используется пункт `default` для учёта всех неиспользуемых комбинаций (или `else` в операторе ветвления `if...else`). В листинге 2.4 в этом случае выход мультиплексора переходит любое состояние, которое определяется синтезатором.

Что будет, если мы не учтём все комбинации и не отметим вариант по умолчанию? В этом случае синтезатор выдаст ошибку, так как процедурный блок `always_comb` не предполагает элементов памяти. Если они нужны, то используется процедурный блок `always_latch`. Рассмотрим пример с мультиплексором «из 3 в 1». Для кодирования трёх входных состояний требуется 2-битная шина выбора (адресная шина). Но она может кодировать максимально 4 состояния. Значит, одно из них не будет использоваться. Сформируем два описания мультиплексора на базе оператора `case` с и без состояния по умолчанию и приведём результат синтеза.

Листинг 2.5 – Появление D-защёлок

без <code>default</code>	с <code>default</code>
<pre> 1 module mux31_case(D, S, F); 2 3 input logic [2:0] D; 4 input logic [1:0] S; 5 output logic      F; 6 7 always_latch begin 8     case ( S ) 9         2'b00 : F = D[0]; 10        2'b01 : F = D[1]; 11        2'b10 : F = D[2]; 12    endcase 13 end 14 15 endmodule </pre>	<pre> 1 module mux31_case(D, S, F); 2 3 input logic [2:0] D; 4 input logic [1:0] S; 5 output logic      F; 6 7 always_comb begin 8     case ( S ) 9         2'b00 : F = D[0]; 10        2'b01 : F = D[1]; 11        2'b10 : F = D[2]; 12        default : F = D[0]; 13    endcase 14 end 15 16 endmodule </pre>



Как можно видеть, синтезатор при отсутствии перебора всех возможных значений сигналов в списке чувствительности процедурного блока `always` для состояния, когда  $S = 2'b11$  сформировал защёлку, так как в этом случае сигнал  $F$  должен сохранять своё последнее значение.

При разработке цифровых интегральных схем защёлки считаются нежелательным элементом, так как являются D-триггерами, синхронизируемыми по уровню. Следует стараться их избегать и всегда использовать синхронизацию по фронту. При синтезе схемы Quartus в логах предупреждает о появлении защёлок.

### Тернарный оператор

При проектировании небольших комбинационных схем, в которых как и в мультиплексоре присутствует идея выбора, полезным бывает использование непрерывного присваивание с применением специальной конструкции, именуемой *тернарный оператор*

```
1 assign <сигнал> = (<условие>) ? (<действие, если условие истинно>) : (<действие, если условие ложно>);
```

Идея работы этого оператора схожа с оператором ветвления. Например, мультиплексор «из 2 в 1» можно записать как представлено в листинге 2.6.

Листинг 2.6 – HDL-описание мультиплексора «из 2 в 1» с тернарным оператором

```
1 module mux21_tern(D, S, F);
2
3 input logic [1:0] D;
4 input logic S;
5 output logic F;
6
7 assign F = ( !S ) ? ( D[0] ) : ( D[1] );
8
9 endmodule
```

Здесь в качестве условия выступает проверка истинности инверсного значения сигнала  $S$ . Если  $S = 0$ , то  $F = D_0$ , иначе  $F = D_1$ .



Для мультиплексора «из 4 в 1» также можно использовать тернарный оператор, так как он может быть вложен в другой тернарный оператор, однако такой подход не рекомендуется для больших схем, так как затрудняется восприятие кода. Тернарный оператор полезен для замены структуры `if . . . else` с наличием одного условия.

Листинг 2.7 – HDL-описание мультиплексора «из 4 в 1» с тернарным оператором

```
1 module mux41_tern(D, S, F);
2
3 input logic [3:0] D;
4 input logic [1:0] S;
5 output logic      F;
6
7 assign F = (!S[1]) ? ((!S[0]) ? D[0] : D[1]) : ((!S[0]) ? D[2] : D[3]);
8
9 endmodule
```

Самый простой способ описать мультиплексор при условии, что количество информационных входов равно  $N = 2^M$ , где  $M$  – количество адресных входов, заключается в идее оператора выбора, в которой на выход подаётся значение того входа, номер которого определён адресным входом в десятичном представлении.

Листинг 2.8 – HDL-описание мультиплексора «из 4 в 1»

```
1 module mux41(D, S, F);
2
3 input logic [3:0] D;
4 input logic [1:0] S;
5 output logic      F;
6
7 assign F = D[S];
8
9 endmodule
```

## Блокирующее присваивание

В последовательных операторах `if...else` и `case` при проектировании комбинационной логики используется *блокирующее присваивание*. Это присваивание обозначается обычным знаком равенства (=), но без специального слова `assign`, которое определяет непрерывное присваивание и выполняется параллельно.

При использовании блокирующего присваивания все действия выполняются последовательно (оператор блокирует выполнение следующего действия до окончания выполнения предыдущего). Например, в листинге 2.9 в результате выполнения блока `always_comb` сигнал ‘b’ примет значение сигнала ‘a’, следом за этим сигнал ‘c’ примет значение сигнала ‘b’, которое

уже равно 'а', потом аналогично повторит значение сигнала 'а' сигнал 'd', и в завершении выходной сигнал приобретает значение сигнала 'а'.

Листинг 2.9 – Блокирующее присваивание

```
1 module test(a, f);
2
3 input logic a;
4
5 output logic f;
6
7 logic b, c;
8
9 always_comb begin
10     b = a;
11     c = b;
12     f = c;
13 end
14
15 endmodule
```

При проектировании *последовательностной* логики вместо блокирующего присваивания используется *неблокирующее присваивание*, которое позволяет выполняться операциям *одновременно* (не блокирует их выполнение).

## Упражнения

1. Приведите HDL-описание шифратора «из 8 в 3» и дешифратора «из 3 в 8» с использованием последовательных операторов `if...else` и `case`. Проведите синтез схем и продемонстрируйте RTL-представление обоих вариантов.
2. Приведите HDL-описание 8-битного приоритетного шифратора с использованием непрерывного присваивания `assign` с тернарным оператором, оператором ветвления `if...else` и оператором выбора `case`. Сравните их RTL-представления. Напишите для него testbench и промоделируйте работу.
3. Проведите проектирование 8-разрядного селектора для передачи 32-битных сигналов. Напишите тестовый модуль и смоделируйте работу селектора. *Указание:* селектором называют мультиплексор, у которого адресные входы закодированы в позиционном коде.

## Работа № 3. Последовательная логика. Неблокирующее присваивание. Параметризация

В отличие от комбинационных схем, в которых значения выходных сигналов определяются текущим значением входных, в последовательной (или секвенциальной) логике значения сигналов на выходе зависят не только от текущего значения сигналов на входе, но и от предыдущего значения сигналов на выходе. То есть в последовательная логика является логикой памяти. В цифровых интегральных схемах последовательная логика используется повсеместно в виде регистров, счётчиков, блоков регистровой памяти.

Сам уровень регистровых передач (англ. RTL, *register transfer level*) предполагает описание схемы как передача сигнала между регистрами с обработкой их в комбинационной логике без определения того, как регистры и комбинационная логика реализуется на вентиляльном уровне. Переход от RTL-уровня на вентиляльный осуществляется в результате синтеза поведенческого HDL-описания схемы.

### Статический D-триггер

Простейшим элементом памяти является D-триггер, синхронизируемый уровнем. Его также называют статическим D-триггером, D-защёлкой или триггером-защёлкой (рисунок 3.1). В таблице истинности (таблица 3.1) приведено формальное описание его работы.

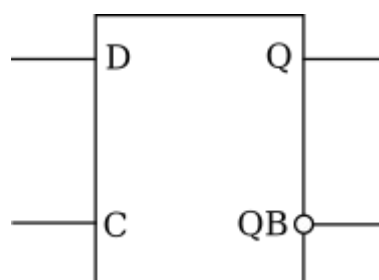


Рисунок 3.1 – D-защёлка

Таблица 3.1 – Таблица истинности статического D-триггера

$C$	$Q_n$	$QB_n$
0	$Q_{n-1}$	$QB_{n-1}$
1	$D$	$\bar{D}$

С точки зрения поведенческого описания работу статического D-триггера можно сформулировать как *выход Q будет принимать значения входа D, пока на входе C держится высокий уровень сигнала, иначе значение выхода Q не меняется; для инвертированного выхода QB (от англ. bar –*

планка) повторяется то же самое, но формируется инверсное значение сигнала *D*. HDL-описание такой формулировки приведено в листинге 3.1.

Листинг 3.1 – HDL-описание статического D-триггера

```
1 module DLatch(D, C, Q, QB);
2
3 input logic D, C;
4
5 output logic Q, QB;
6
7 always_latch begin
8     if ( C == 1'b1 ) begin
9         Q <= D;
10        QB <= ~D;
11    end
12 end
13
14 endmodule
```

Обратите внимание, так как в условии на строках 8-11 была только одна проверка на значение сигнала ‘С’ равного логической 1, но не указано, что делать при значении сигнала ‘С’ равном логическому 0, то в результате синтеза мы получим триггер-защёлку.

### Динамический D-триггер

Триггер-защёлка является нежелательным элементом в цифровой схеме. Связано это с тем, что возникает трудность временного анализа, то есть определение критических путей, максимальной частоты работы схемы и прочее. В связи с этим базовым элементом последовательностных схем является D-триггер, синхронизируемый по фронту (рисунок 3.2). Этот триггер ещё называют *динамическим* (англ. D-flip-flop). Он в отличие от статического, который синхронизируется по уровню и является триггером-защёлкой, срабатывает в момент прихода тактового синхроимпульса.

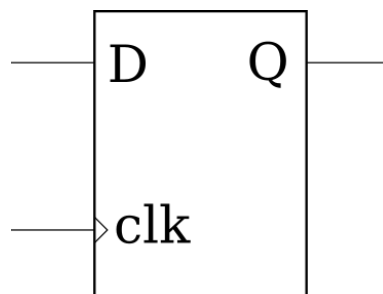


Рисунок 3.2 – Динамический D-триггер

Отличие от статического D-триггера заключается в том, что динамический срабатывает по фронту тактового синхроимпульса. То есть на выход *Q* значение входа *D* передаётся в момент переключения сигнала *clk*

из 0 в 1, либо из 1 в 0. Для динамического D-триггера используется процедурный блок `always_ff` со списком чувствительности. Для обозначения переднего фронта используется ключевое слово `posedge`<sup>4</sup>. Описание D-триггера представлено в листинге 3.2.

Листинг 3.2 – HDL-описание динамического D-триггера

```
1 module Dflipflop(D, clk, Q);
2
3 input logic D, clk;
4
5 output logic Q;
6
7 always_ff @(posedge clk) begin
8     Q <= D;
9 end
10
11 endmodule
```

### Неблокирующее присваивание

Стоит отметить, что в листингах 3.1 и 3.2 используется новый вид присваиваний – неблокирующее присваивание (`<=`). Оно используется в случае, когда необходимо параллельное, или одновременное, выполнение операций в процедурных скобках `begin...end`.

В приведённом примере с D-защёлкой неясна необходимость использования неблокирующего присваивания, так как результат синтеза при использовании блокирующего присваивания будет тем же. Для того чтобы понять, приведём пример (листинг 3.3).

Листинг 3.3 – Пример блокирующего и неблокирующего присваиваний

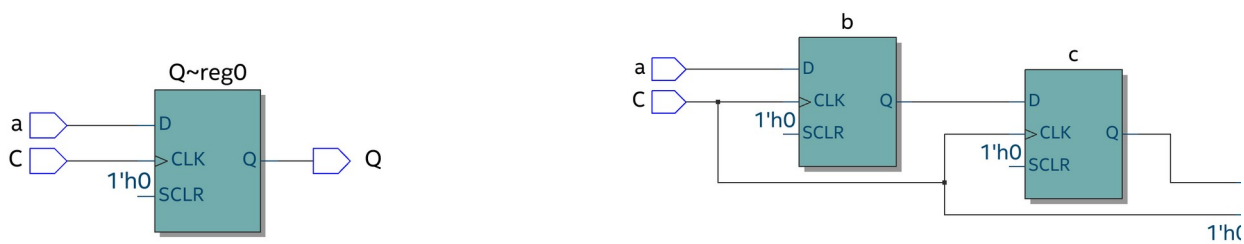
```
1 module test_block(D, clk, Q);
2
3 input logic D, clk;
4
5 output logic Q;
6
7 logic b, c;
8
9 always_ff @(posedge clk) begin
10     b = D;
11     c = b;
12     Q = c;
13 end
14
15 endmodule
```

```
1 module test_nonblock(D, clk, Q);
2
3 input logic D, clk;
4
5 output logic Q;
6
7 logic b, c;
8
9 always_ff @(posedge clk) begin
10     b <= D;
11     c <= b;
12     Q <= c;
13 end
14
15 endmodule
```

<sup>4</sup> От англ. positive edge – положительный фронт, или нарастающий фронт, переключение из 0 в 1; для перехода из 1 в 0 используется понятие отрицательного фронта или спадающего – англ. negative edge, соответственно в Verilog это `negedge clk`.

В описании модулей указано, что при появлении переднего фронта сигнала `clk` формируется значение выходного сигнала `Q`. При этом в модуле `test_block` с блокирующим присваиванием формирование сигнала `Q` идёт последовательно: сначала сигнал `b` примет значение сигнала `D`, затем сигнал `c` примет значение сигнала `b`, после этого в этом же моменте симуляции сигнал `Q` примет значение сигнала `c`. Таким образом за один такт значение сигнала `D` напрямую перейдёт на выход `Q`. Синтезатор сформирует обычный динамический D-триггер (рисунок 3.4, а).

Во втором случае в модуле `test_nonblock` формирование сигнала `Q` через промежуточные сигналы `b` и `c` сформируется параллельно. Это означает, что по окончании текущего момента симуляции (при выходе из блока `always_ff`) сигнал `b` примет значение сигнала `D`, сигнал `c` – предыдущее значение сигнала `b` (которое у него было в момент входа в блок `always_ff`), аналогично сигнал `Q` примет значение сигнала `c`, которое у него было в момент входа в блок `always_ff`. Таким образом формируется сдвиговый регистр, состоящий из трёх динамических D-триггеров (рисунок 3.2, б).



а) б)  
 Рисунок 3.2 – Результат синтеза листинга 3.3:  
 а) с блокирующим присваиванием, б) с неблокирующим присваиванием

## Параллельный регистр

С точки зрения работы регистра с параллельной записью и параллельным выводом, его работа ничем не отличается от динамического D-триггера кроме того, что входной сигнал данных и выходной сигнал становятся многоразрядными.

Перед тем, как начать описывать регистр следует отметить, что триггерные устройства имеют проблему, связанную с тем, что в начале работы следует установить их начальное значение (обычно – сброс состояния в 0). С точки зрения реализации сброса можно разделить регистры на два вида: с *асинхронным сбросом* и с *синхронным сбросом*.

При асинхронном сбросе сигнал сброса является приоритетным перед сигналом тактового синхроимпульса и может сработать в любой момент. Поведенчески работа такого регистра может быть описана как *если на входе сброса активный уровень сигнала, то на выходе регистра 0, на тактовый синхроимпульс не реагирует, иначе по приходу положительного фронта синхроимпульса на выход Q передаётся значение на входе D*. Описание такого поведения на примере 32-битного регистра представлено в листинге 3.4.

Листинг 3.4 – HDL-описание 32-битного регистра с асинхронным сбросом

```
1 module REG32(D, clk, _rst, Q);
2
3 input logic [31:0] D;
4 input logic clk, _rst;
5
6 output logic [31:0] Q;
7
8 always_ff @(posedge clk or negedge _rst) begin
9   if ( !_rst )
10    Q <= 0;
11   else
12    Q <= D;
13 end
14
15 endmodule
```

Здесь для сигнала `_rst` в списке чувствительности используется ключевое слово `negedge`, так как в SystemVerilog список чувствительности может содержать либо значения сигналов (реакция на уровень как в комбинационной логике), либо события типа перехода из 0 в 1 (`posedge`), либо из 1 в 0 (`negedge`).

При использовании синхронного с тактовым импульсом сбросом в списке чувствительности сигнал `_rst` будет отсутствовать, так как сброс регистра будет осуществляться только по приходу положительного фронта (листинг 3.5).

Листинг 3.5 – HDL-описание 32-битного регистра с синхронным сбросом

```
1 module REG32(D, clk, _rst, Q);
2
3 input logic [31:0] D;
4 input logic clk, _rst;
5
6 output logic [31:0] Q;
7
8 always_ff @(posedge clk) begin
9   if ( !_rst )
10    Q <= 0;
11   else
```

```

12     Q <= D;
13 end
14
15 endmodule

```

Регистр является универсальным устройством, на котором можно сформировать различные другие устройства: счётчик, сдвиговый регистр, элемент памяти.

## Счётчик

Счётчик отличается от регистра тем, что при приходе тактового синхрои импульса он изменяет своё значение, увеличивая его на 1 (суммирующий счётчик) или уменьшая на 1 (вычитающий счётчик), или реализуя оба вида счёта (реверсивный счётчик).

В листинге 3.6 приведено описание суммирующего счётчика.

Листинг 3.6 – HDL-описание 4-разрядного суммирующего счётчика

```

1 module cnt4(clk, _rst, Q);
2
3 input logic      clk, _rst;
4
5 output logic [3:0]      Q;
6
7 always_ff @(posedge clk, negedge _rst) begin
8     if ( !_rst )
9         Q <= 0;
10    else
11        Q <= Q + 1'b1;
12 end
13
14 endmodule

```

Такой счётчик считает от 0 до 15 (в шестнадцатеричном формате от 0x0 до 0xF). Если требуется реализовать счётчик, считающий до определённого значения, например, до 9 (двоично-десятичный счётчик, англ. BCD, *binary-coded decimal*), то необходимо добавить условие перехода в значение 0 при достижении значения 9 (листинг 3.7).

Листинг 3.7 – HDL-описание двоично-десятичного суммирующего счётчика

```

1 module cnt4_bcd(clk, _rst, Q);
2
3 input logic      clk, _rst;
4
5 output logic [3:0]      Q;
6
7 always_ff @(posedge clk, negedge _rst) begin
8     if ( !_rst )
9         Q <= 0;
10    else if ( Q < 9 )

```



```

11     Q <= Q + 1'b1;
12     else
13     Q <= 0;
14 end
15
16 endmodule

```

Таким же образом можно реализовать вычитающий и реверсивный счётчики. А также счётчик с установкой начального значения.

Следует заметить, что для примера с регистром 32-разрядным может понадобиться, например, регистр 80-разрядный, а также 4-разрядный. По логике, можно для каждого такого регистра написать отдельный модуль. Однако сам принцип работы регистра не изменится. Для того чтобы не плодить дополнительные sv-файлы, а также оптимизировать и сделать более гибким описание разумным будет применять параметризуемые модули.

## Параметризуемые модули

В указанном выше примере регистров при их описании будет меняться только одна величина – размер шины данных. Она может быть задана как параметр. В общем виде параметризуемый модуль описывается как

```

1 module <имя_модуля> #(parameter <имя_параметра_1> = <значение_параметра_1_по_умолчанию>,
2                               <имя_параметра_2> = <значение_параметра_2_по_умолчанию>,
3                               ...
4                               <имя_параметра_n> = <значение_параметра_n_по_умолчанию> )
5
6     ([перечисление_портов]);

```

Тогда наш регистр будет записан в листинге 3.8.

Листинг 3.8 – HDL-описание параметрического регистра

```

1 module REG_par #(parameter WIDTH = 32)
2     (D, clk, _rst, Q);
3
4     input  logic [WIDTH-1:0]    D;
5     input  logic                clk, _rst;
6
7     output logic [WIDTH-1:0]    Q;
8
9     always_ff @(posedge clk or negedge _rst)
10        if ( !_rst )
11            Q <= 0;
12        else
13            Q <= D;
14
15 endmodule

```

Подключение экземпляра параметризованного модуля в общем виде выглядит как

```

1 <имя_модуля> #(.<имя_параметра_1>(<значение_параметра_1>),

```

```

2         .<имя_параметра_2>(<значение_параметра_2>),
3         ...
4         .<имя_параметра_n>(<значение_параметра_n>))
5     <имя_экземпляра_модуля>
6     (<сопоставление портов>);

```

Здесь значение параметра назначается также как и порты через точку. Если значение параметра не указано, то используется значение по умолчанию. То есть экземпляры 80-разрядного и 4-разрядного регистров будут созданы как в листинге 3.9.

Листинг 3.9 – Создание экземпляров параметрических модулей

```

1 REG_par #(.WIDTH(80)) REG80 (.D(D80), .clk(clk), ._rst(_rst), .Q(Q80));
2 REG_par #(.WIDTH(4) ) REG4 (.D(D4) , .clk(clk), ._rst(_rst), .Q(Q4));

```

## Сдвиговые регистры

Часто при выполнении различных операций над двоичными данными необходимо осуществлять их сдвиг влево, вправо, циклический влево или вправо.

Из нециклических различают два вида сдвигов: *логический* и *арифметический*.

### Логический сдвиг

При логическом сдвиге влево или вправо все разряды числа сдвигаются в соответствующую сторону на указанное количество разрядов, а на месте освободившихся разрядов записывается 0.

Логический сдвиг *влево* эквивалентен *умножению* десятичного вида двоичного числа на  $2^n$ , где  $n$  – количество сдвигаемых разрядов. Например,  $3_{10} = 0011_2$ , при сдвиге двоичного числа влево на один разряд получаем  $0110_2 = 6_{10}$ .

Логический сдвиг *вправо* эквивалентен *делению* десятичного вида двоичного числа на  $2^n$ , где  $n$  – количество сдвигаемых разрядов. Например,  $32_{10} = 10000_2$ , при сдвиге двоичного числа вправо на два разряда получаем  $00100_2 = 8_{10}$ .

Операция сдвига может быть осуществлена как в виде комбинационной логики, так и в виде последовательностной.

Для комбинационной логики используются операторы << и >> для сдвига влево и вправо соответственно. Синтаксис выглядит следующим образом

<сдвинутое\_число> = <сдвигаемое\_число> << <количество\_разрядов\_сдвига>;

<сдвинутое\_число> = <сдвигаемое\_число> >> <количество\_разрядов\_сдвига>;

### *Арифметический сдвиг*

У логического сдвига обнаруживается недостаток, связанный с применением его для чисел со знаком.

Следует напомнить, что для чисел со знаком старший разряд определяет знак числа. Если он равен 0, то число, записанное справа от этого разряда, положительное и представлено в прямом коде. Если же знак равен 1, то число отрицательное и записано в дополнительном коде.

Например,  $0110_2$  положительное число, десятичное значение которого равно  $110_2 = 6_{10}$ . А уже  $1110_2$  – отрицательное. Для получения значения модуля требуется вычесть «1» и инвертировать все биты, то есть  $110_2 \rightarrow 101_2 \rightarrow 100_2 = 4_{10}$ , то есть  $1110_2 = -4_{10}$ .

Для сдвигов (операций умножения и деления) в случае чисел со знаками используется арифметический сдвиг, при котором значение старшего разряда сохраняется.

Арифметический сдвиг влево осуществляется так же, как и логический, то есть в конце приписываются нули. При арифметическом сдвиге вправо значение разряда повторяется на месте сдвинутых бит.

В SystemVerilog применяется операция <<< и >>> для арифметического сдвига влево и вправо соответственно.

Например, результатом выполнения  $-4 \lll 1$  будет  $-4_{10} = 1\_1100_2 \lll 1 = 1\_1000_2 = -8_{10}$ .

Для того чтобы реализовать арифметический сдвиг в SystemVerilog, синтезатору необходимо сообщить, что результат сдвига является знаковым, для этого используется ключевое слово **signed** при объявлении порта или сигнала. Например, регистр, осуществляющий арифметический сдвиг вправо на 1 разряд введённого числа, описан в листинге 3.10.

Листинг 3.10 – HDL-описание сдвигового регистра (арифметический сдвиг)

```

1 module regShiftRightArith #(parameter WIDTH = 8)
2     (D, clk, _rst, shift, Q);
3
4 input      logic [WIDTH-1:0]      D;
5 input      logic                  clk, _rst, shift;
6
7 output signed logic [WIDTH-1:0]    Q;
8
9 always_ff @(posedge clk, negedge _rst) begin
10  if ( !_rst )           //сбрасываем регистр при _rst = 0
11  Q <= 0;
12  else if ( !shift ) //записываем в регистр при shift = 0
13  Q <= D;
14  else                   //сдвигаем данные в регистре при shift = 1
15  Q <= Q >>> 1;
16 end
17
18 endmodule

```

Однако реализация арифметического сдвига влево в SystemVerilog работает не корректно, в связи с чем сдвиги могут быть реализованы через операции конкатенации и репликации.

Например, сдвиг логический вправо на 3 разряда для 8-битного числа представлен в листинге 3.11.

Листинг 3.11 – HDL-описание сдвигового регистра (логический сдвиг вправо)

```

1 always_ff @(posedge clk, negedge _rst)
2     ...           //логика сброса
3 else if
4     ...           //логика установки значения
5 else             //сдвигаем данные в регистре
6     Q <= {3{1'b0}, Q[7:4]};

```

## Упражнения

1. Спроектируйте двухразрядный двоично-десятичный счётчик. Напишите тестовый модуль и проведите моделирование его работы. *Указание:* спроектируйте модуль одноразрядного двоично-десятичного счётчика, на основе которого сформируйте структурное описание модуля верхнего уровня.
2. Спроектируйте двухразрядный двоично-десятичный счётчик, считающий до 60. Напишите тестовый модуль и проведите моделирование его работы. *Указание:* спроектируйте параметризуемый модуль одноразрядного двоично-десятичного счётчика, на основе которого сформируйте структурное описание модуля верхнего уровня,

используя два модуля одноразрядного счётчика – для единиц и десятков.

3. Спроектируйте делитель частоты, осуществляющий формирование тактового сигнала 1 Гц из сигнала частотой 50 МГц. Организуйте также формирование импульса (флага), который детектирует фронт генерируемого тактового сигнала.
4. Спроектируйте параметрический регистр, осуществляющего ввод начального значения и кольцевой сдвиг влево или вправо. Напишите тестовый модуль к нему и смоделируйте его работу для  $N$ -разрядного числа, где  $N$  в 3 раза превышает номер по журналу.
5. Спроектируйте ШИМ, регулирующий яркость светодиода, мерцающего при частоте 0,5 Гц. Модулятор работает на частоте 50 МГц, возможно кнопками увеличивать яркость светодиода или уменьшать её в диапазоне от 10 до 90 % с шагом 10 %. Продемонстрируйте работоспособность ШИМ на ПЛИС. *Указание:* антидребезгом для кнопок можно пренебречь, либо выполнить упр. 7.
6. На основе упр. 5 спроектируйте «пищалку» с меняющейся частотой и громкостью звучания. *Указания:*
  - 6.1. в качестве «пищалки» используйте источник звука BELL на отладочной плате,
  - 6.2. частота пицания меняется в диапазоне от 0,5 до 10 Гц с шагом 0,5 Гц,
  - 6.3. громкость звучания регулируется кнопками при помощи ШИМ.
7. Разработайте систему антидребезга для кнопок на основе счётчика.

## Работа № 4. Автоматы конечных состояний

Проектирование последовательностной логики с использованием концепции автоматов конечных состояний (англ. FSM, Finite State Mashine) позволяет представить разрабатываемое устройство как некоторую машину, работающую в различных состояниях. Примером такой машины может быть интерфейс интегральной схемы для последовательного приёма-передачи данных, который работает в таких состояниях как приём, передача, ожидание, проверка, ответ и пр.

Теория конечных автоматов предполагает наличие некоторого количества различных типов автоматов. С точки зрения проектирования интегральных схем наиболее используемыми являются автомат Мура и автомат Мили.

### Описание конечного автомата на Verilog

При описании конечного автомата рекомендуется следовать следующим правилам:

1. Каждый конечный автомат следует описывать в отдельном модуле с целью возможности лёгкости чтения кода, а также упрощения работы компилятора.
2. При кодировании состояний автомата не используйте препроцессорную директиву ``define`, используйте перечисляемый тип данных `enum`.
3. Все последовательностные `always_ff`-блоки используют неблокирующее присваивание, все комбинационные `always_comb`-блоки – блокирующие. Это позволит избежать гонок сигналов в автомате.
4. Описание конечного автомата должно легко поддаваться отладке.

Для описания регистра состояний используется последовательностный `always_ff`-блок с неблокирующими присваиваниями.

Листинг 4.1 – Описание регистра состояний

```
1 always_ff @(posedge clk, negedge _rst)
2   if ( !_rst )
3     state <= S0;
4   else
5     state <= nextstate;
```

Для определения следующего состояния используется комбинационный `always_comb`-блок с блокирующим присваиванием. Общее представление такого блока показано в листинге ниже.

Листинг 4.2 – Описание комбинационной схемы, вычисляющей следующее состояние

```
1 always_comb
2   case ( state )
3     S0 :
4       if ( input_signal )
5         nextstate = S1;
6       else
7         nextstate = S0;
8       // ...
9     S1 :
10      if ( input_signal )
11        nextstate = S2;
12      else
13        nextstate = S0;
14      // ...
15    default : nextstate = S0;
16  endcase
```

Здесь в каждом состоянии проверяется значение входного сигнала `input_signal`, после чего формируется значение следующего состояния, которое автомат примет при приходе переднего фронта тактового импульса согласно листинга 4.1.

Выходные сигналы могут так же формироваться с помощью комбинационных `always_comb`-блоков, либо при помощи оператора непрерывного присваивания `assign`.

Листинг, реализующий автомат Мура для поиска последовательности «11» в битовом потоке данных приведён ниже

Листинг 4.3 – Описание автомата Мура

```
1 module search_Moore(D, clk, _rst, Q);
2
3   input  logic D, clk, _rst;
4   output logic      Q;
5
6   enum  logic [1:0] {SAD = 2'b00, HOPE, HOORAY} state, nextstate;
7
8   //регистр состояний
9   always_ff @(posedge clk, negedge _rst)
10    if ( !_rst )
11      state <= SAD;
12    else
```

```

13     state <= nextstate;
14
15     //комбинационная функция состояний
16     always_comb
17     case ( state )
18         SAD : nextstate = (D) ? HOPE : SAD;
19         HOPE, HOORAY : nextstate = (D) ? HOORAY : SAD;
20         default : nextstate = SAD;
21     endcase
22
23     //выходная комбинационная логика
24     assign Q = (state == HOORAY) ? 1'b1 : 1'b0;
25
26 endmodule

```

Автомат Мили требует на одно состояние меньше по сравнению с автоматом Мура, так как выходное значение определяется также входными сигналами. Автомат Мили описан в листинге ниже

Листинг 4.4 – Описание автомата Мили

```

1  module search_Mealey(D, clk, _rst, Q);
2
3  input  logic D, clk, _rst;
4  output logic      Q;
5
6  enum logic {SAD = 1'b0, HOPE} state, nextstate;
7
8  //регистр состояний
9  always_ff @(posedge clk, negedge _rst)
10     if ( !_rst )
11         state <= SAD;
12     else
13         state <= nextstate;
14
15     //комбинационная функция состояний
16     always_comb
17     case ( state )
18         SAD, HOPE : nextstate = (D) ? HOPE : SAD;
19         default : nextstate = SAD;
20     endcase
21
22     //выходная комбинационная логика
23     assign Q = (D == 1'b1 && state == HOPE) ? 1'b1 : 1'b0;
24
25 endmodule

```

## Графическое задание конечного автомата

Одним из вариантов задания конечного автомата в среде Quartus Prime является графическое представление графа автомата. Для создания такого



представления следует проделать следующие шаги (на примере автомата, реализующего поиск последовательности «11» в потоке битовых данных).

1. Создайте в проекте новый файл типа State Mashine File. Будет создан smf-файл. В появившемся окне в области слева можно увидеть входные и выходные порты автомата. Нажатием правой кнопки мыши можно вызвать контекстное меню, позволяющее добавить новые порты.

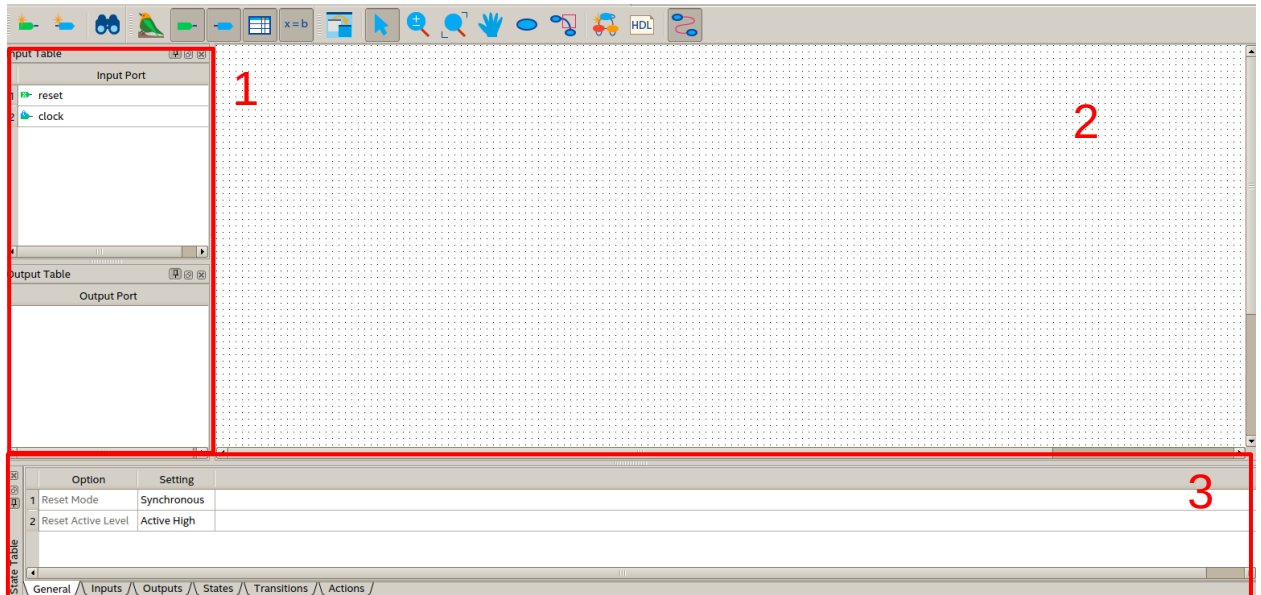



Рисунок 4.1 – Окно графического представления автомата:

1 – входные и выходные сигналы, 2 – область рисования графа автомата, 3 – область параметров

2. Для автомата, реализуемого нами, добавьте необходимые входные и выходные порты.
3. Добавьте состояния автомата. Для этого нажмите State Tool на панели меню , курсором установите состояние на свободном поле области рисования. За один раз можно добавить несколько состояний.
4. Двойным щелчком по состоянию откройте окно свойств, во вкладке General укажите название состояния, а также отметьте состояние по умолчанию (то, в которое переходит автомат по сигналу reset, оно отображается со стрелочкой).

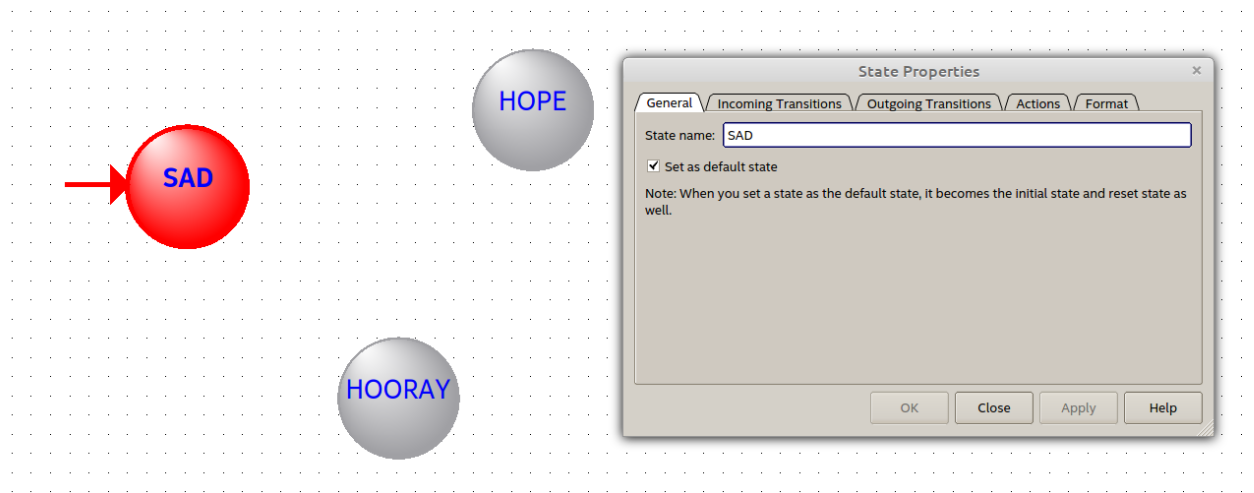



Рисунок 4.2 – Свойства состояния

5. Затем добавьте переходы между состояниями инструментом Transition Tool . Для задания петли достаточно один раз щёлкнуть по состоянию. Для задания перехода между состояниями необходимо с зажатой левой кнопкой мыши провести курсором от начального состояния к следующему.

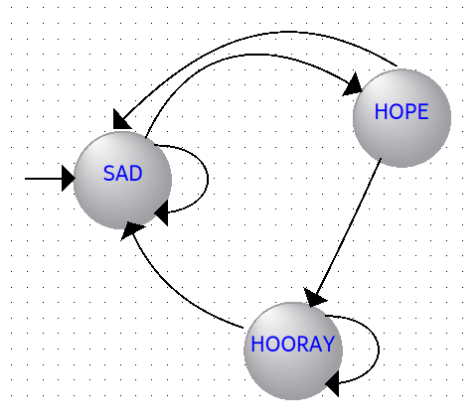


Рисунок 4.3 – Переходы между состояниями

6. Следующим шагом следует указать условия переходов из одного состояния в другое. Для этого можно использовать два способа.
- а. Первый заключается в указании условий входящих в состояние и исходящих из него переходов. Откройте окно свойств состояния и на вкладках Incoming Transitions и Outcoming Transitions введите в нотации SystemVerilog условия переходов (для отсутствия условия, введите в нотации VHDL условие OTHERS)

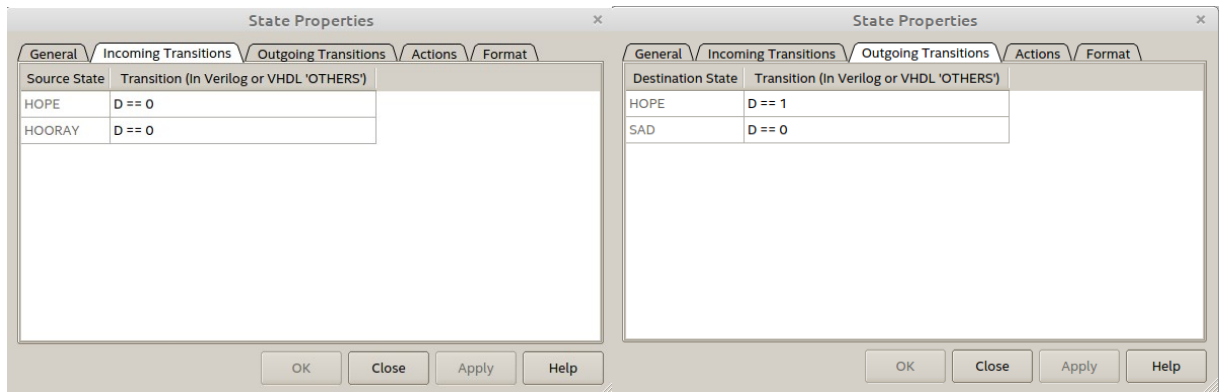


Рисунок 4.4 – Входящие и исходящие переходы состояния SAD

- б. Второй заключается в непосредственном указании условий каждого перехода. Откройте окно свойств перехода и также в нотации SystemVerilog введите условие (или в нотации VHDL укажите OTHERS).

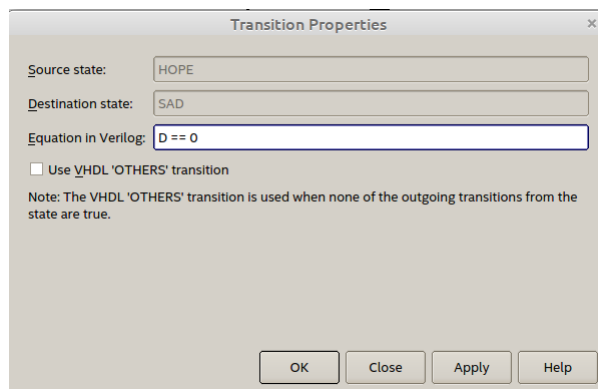


Рисунок 4.5 – Переход из состояния HOPE в состояние SAD

7. После указания всех переходов они отобразятся на графе автомата.

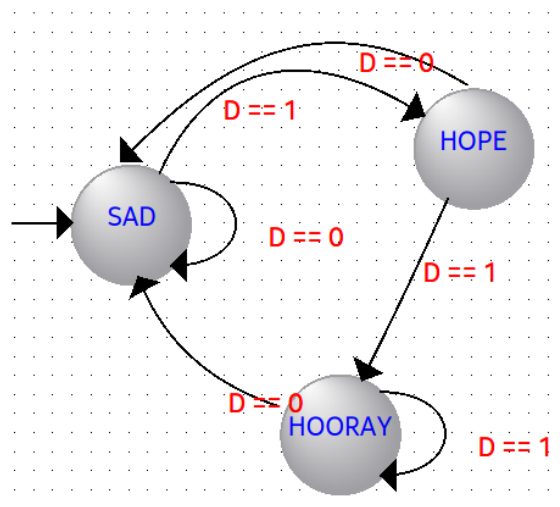


Рисунок 4.6 – Граф автомата с переходами

8. На вкладке *Actions* каждого состояния укажите значение выходных сигналов в этом состоянии.

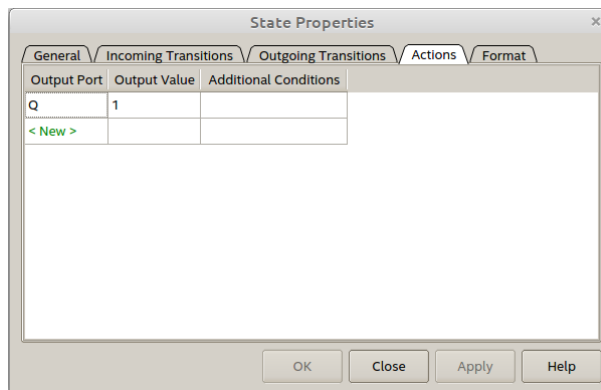



Рисунок 4.7 – Вкладка *Actions* для состояния HOORAY

9. В поле нижних вкладок на вкладке *General* можно указать, является ли сброс синхронным или асинхронным, а также задать активный уровень сигнала сброса. На вкладке *Inputs* отображаются все входные сигналы с указанием сигнала сброса и тактового сигнала. На вкладке *Outputs* отображаются все выходные сигналы. Их можно задать как регистровыми, так и не регистровыми (синхронизируется ли значение выходного сигнала через регистр), можно указать, что состояние меняется в тот же самый цикл тактового сигнала или на следующий. На вкладке *States* перечислены все состояния, в *Transitions* – все функции переходов, а на *Actions* – все функции выходов.
10. После формирования графического представления конечного автомата, для дальнейшего его тестирования необходимо сформировать его HDL-описание. Для этого нажмите кнопку *Generate HDL File*  на панели меню и в открывшемся окне выберите *SystemVerilog*. Сформируется HDL-описание автомата.

## Моделирование конечного автомата в ModelSim

Для вывода текущего состояния на осциллограмму в ModelSim следует использовать следующий ход. В описании модуля добавить регистровую переменную и при помощи процедурного оператора выбора *case* задать ей строковые значения для каждого состояния. Например,

Листинг 4.1 – Определение текстовой переменной в описании модуля

```
1 logic [63:0] textstate;  
2 always_comb  
3     case ( state )  
4         STATE1 : textstate = «STATE1»;
```

```

5     STATE2 : textstate = «STATE2»;
6     STATE3 : textstate = «STATE3»;
7     ...
8     STATEn : textstate = «STATEn»;

9  endcase

```

Следует отметить, что длина переменной определяется максимальной длительностью имени состояния, каждая буква – это один байт или 8 бит, соответственно для слова длиной 8 символов необходимо определить переменную длиной 64 бита. Конструкция в листинге 4.1 несинтезируема.

Для вывода значения состояния в тестбенче следует обратиться к внутреннему сигналу тестируемого модуля. Это осуществляется при помощи иерархического обращения в нотации

```

<экземпляр_модуля_верхнего_уровня>.<экземпляр_модуля_уровня_2>...<экземпляр_модуля_уровня_n>.<сигнал_в_экземпляре_модуля_n>

```

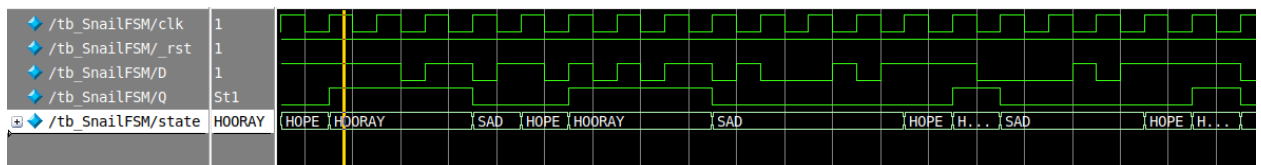
Листинг 4.2 – Обращение к внутреннему сигналу модуля

```

1  myModule DUT(.clk(clk), .rst(rst), .D(D), .Q(Q));
2  logic [63:0] state = DUT.textstate; //обращаемся к внутренней
переменной textstate экземпляра DUT модуля myModule

```

При моделировании в ModelSim на сигнале состояния `state` необходимо выбрать представление Radix > ASCII.



## Упражнения

1. Модифицируйте автоматы Мура и Мили для обнаружения уникальной последовательности 11 в битовом потоке данных.
2. Спроектируйте универсальный сдвиговый регистр, реализующий параллельный ввод 16-разрядного числа и параллельный вывод, последовательный вывод, логический сдвиг влево, логический сдвиг вправо, кольцевой сдвиг влево, кольцевой сдвиг вправо в виде автомата.
3. Спроектируйте автомат, реализующий работу светофора в следующем режиме: 40 секунд горит красный свет, затем в течение 21 секунды горит зелёный, после чего загорается жёлтый свет на 3 секунды. Продемонстрируйте его работу на плате с ПЛИС. В качестве сброса автомата используйте кнопку, а индикаторы – светодиоды. *Указание:*

для формирования тактового сигнала в 1 Гц создайте отдельный модуль делителя частоты, работающем на частоте 50 МГц (внутренняя частота ПЛИС).

4. Модифицируйте описанный выше автомат светофора одним из следующих способов (узнайте у преподавателя):
  - a. в последние 5 секунд зелёный сигнал мигает с частотой 1 Гц;
  - b. по нажатию кнопки во время красного света через 3 секунды загорается зелёный, если красному свету осталось гореть меньше 3 секунд, то нажатие кнопки игнорируется.
  - c. при переключении с красного на зелёный свет также загорается на 3 секунды жёлтый.

## Работа № 5. Проектирование регулярной структуры. Сумматоры

При проектировании цифровых схем периодически возникает необходимость формировать повторяющиеся модули или использовать итеративный подход к созданию кода. Для таких случаев подходит инструкция `generate...endgenerate`. Оборачивание кода в эту инструкцию позволяет управлять синтезом. Выделяют три типа инструкций: итеративный синтез `generate for`, условный синтез `generate if` и `generate case`. В данной работе рассматривается итеративный синтез.

### Инструкция `generate`

Синтаксис данной инструкции следующий:

```
1 //объявление переменной цикла
2 genvar <имя>;
3 //код для блока генерации
4 generate
5   for (<начальное_состояние>; <условие_остановки_цикла>; <увеличение
переменной цикла>) begin : <имя_блока>
6     //код для выполнения
7   end
8 endgenerate
```

Рассмотрим применение инструкции генерации на примере проектирования регулярных структур — сумматоров.

Как известно, сумматор выполняет функцию суммирования двух чисел. При этом многоразрядный сумматор обладает переносом разряда, формирование которого определяет быстродействие работы модуля. С точки зрения формирования сигнала переноса существует множество алгоритмов для вычисления суммы. Рассмотрим самые простые:

1. сумматор со сквозным (последовательным) переносом;
2. сумматор с ускоренным (параллельным) переносом;
3. сумматор с групповым переносом;
4. сумматор с условным переносом.

### Сумматор со сквозным переносом. Время формирования сигнала суммы и переноса

В сумматоре со сквозным переносом одноразрядные полные сумматоры соединены цепочкой переноса (рисунок 5.1).

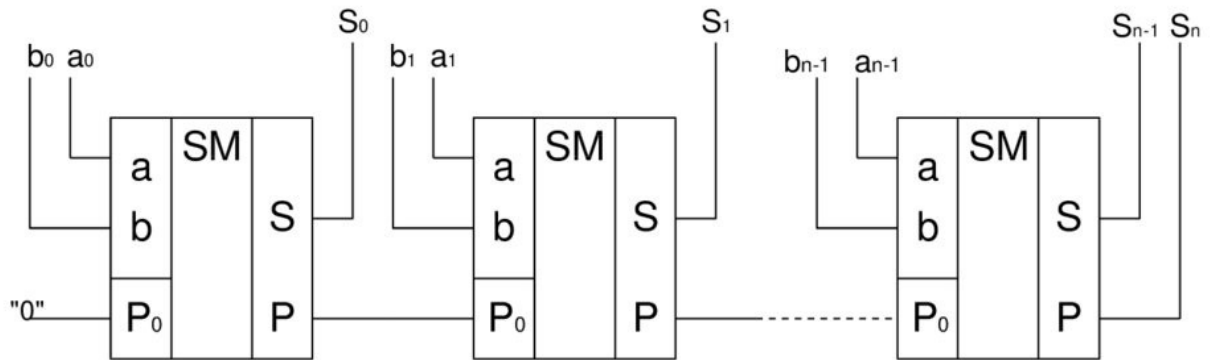


Рисунок 5.1 — Сумматор со сквозным переносом

Это приводит к тому, что каждый последующий разряд начинает формироваться после того, как сформируется сигнал переноса с предыдущего разряда. Одноразрядный полный сумматор представлен на рисунке 5.2, а его HDL-описание в листинге 5.1.

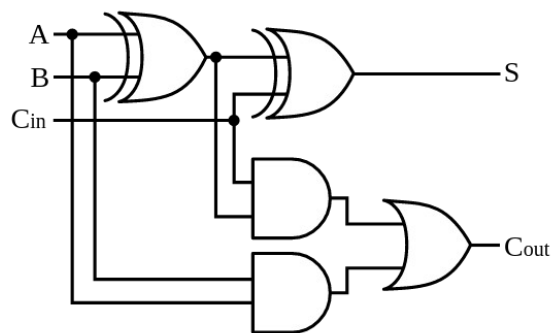


Рисунок 5.2 — Одноразрядный полный сумматор

Листинг 5.1 — HDL-описание одноразрядного полного сумматора

```

1 module FAdder (A, B, Cin, S, Cout);
2   input logic A, B, Cin;
3   output logic S, Cout;
4
5   assign S = A ^ B ^ Cin;
6   assign Cout = (A & B) | (Cin & (A ^ B));
7
8   endmodule

```

Поведенческое описание параметризованного многоразрядного сумматора может быть представлено как сумма входных значений (листинг 5.2).

Листинг 5.2 — Поведенческое HDL-описание сумматора

```

1 module Adder #(parameter WIDTH = 8) (A, B, S, Cout);
2   input logic [WIDTH-1:0] A, B;
3

```



```

4 output logic [WIDTH-1:0] S;
5 output logic Cout;
6
7 assign {Cout, S} = A + B;
8
9 endmodule

```

Результатом синтеза такого сумматора станет сумматор со сквозным переносом (рисунок 5.1), который представляет собой регулярную структуру. Структурное описание такого сумматора использует инструкцию **generate**.

Листинг 5.3 — Структурное HDL-описание многоразрядного сумматора со сквозным переносом

```

1 module FAdder_par #(parameter WIDTH = 8) (A, B, S, Cout);
2
3 input logic [WIDTH-1:0] A, B;
4 output logic [WIDTH-1:0] S;
5 output logic Cout;
6
7 logic [WIDTH:0] w; //шина для всех переносов внутри и
вне сумматора
8
9 assign w[0] = 1'b0; //нулевой перенос – это 0
10
11 genvar i; //переменная, по которой будет производиться генерация
12 generate //блок генерации
13 for(i = 0; i < WIDTH; i++)
14 begin : add_stage //генерируемое имя в виде add_stage[i].Add_bit
15 FAdder Add_bit
(.A(A[i]), .B(B[i]), .Cin(w[i]), .S(S[i]), .Cout(w[i+1]));
16 end
17 endgenerate
18
19 assign Cout = w[WIDTH]; //финальный перенос – это сигнал переноса
сумматора
20
21 endmodule

```

В результате синтезируется 8-битный сумматор, представленный на рисунке 5.3.

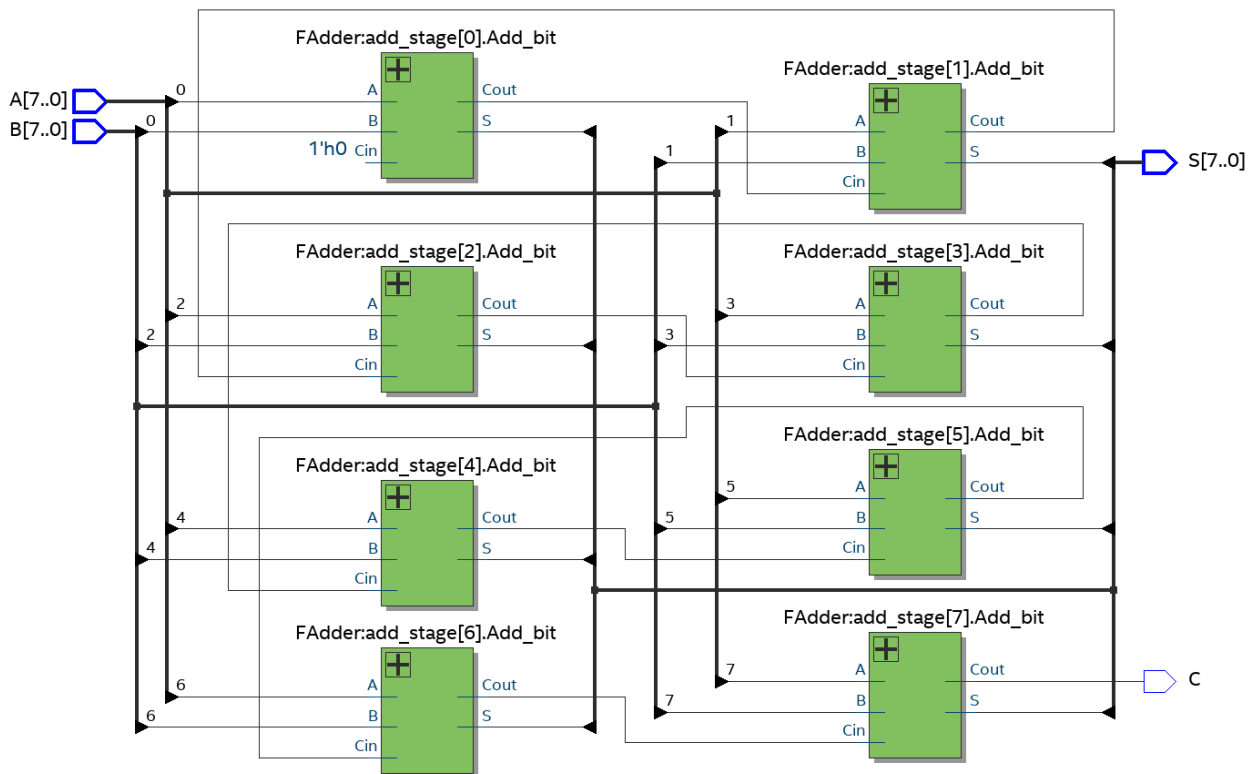


Рисунок 5.3 — Синтезированный 8-битный сумматор со сквозным переносом

Если проанализировать время формирования сигнала переноса в таком сумматоре, то время формирования переноса в младшем разряде будет равно  $T_0^C = 3 \cdot \tau$ , где  $\tau$  — время задержки распространения сигнала одного логического элемента. Задержка формирования сигнала переноса в каждом последующем разряде будет увеличиваться на две задержки, так как формирование  $A \wedge B$  во всех сумматорах завершится в один и тот же момент времени  $\tau$ . Таким образом,  $n$ -й бит переноса сформируется за

$$T_n^C = T_0^C + (n-1) \cdot 2\tau = (1+2n)\tau.$$

Сигнал суммы младшего разряда формируется за время  $T_0^S = 2\tau$ , каждый последующий разряд будет задержан на  $\tau$  после прихода сигнала переноса с предыдущего разряда. Таким образом время формирования  $n$ -го бита суммы будет определяться как

$$T_n^S = T_0^S + T_{n-1}^C + \tau = 2\tau + (1+2(n-1))\tau + \tau = 2(1+n)\tau.$$

Таким образом, время задержки формирования сигналов переноса и суммы линейно возрастает с увеличением разрядности сумматора.

**Сумматор с параллельным переносом. Время формирования сигнала суммы и переноса**

Сигнал переноса каждого разряда одноразрядного полного сумматора формируется в отдельной схеме ускоренного переноса (рисунок 5.4).

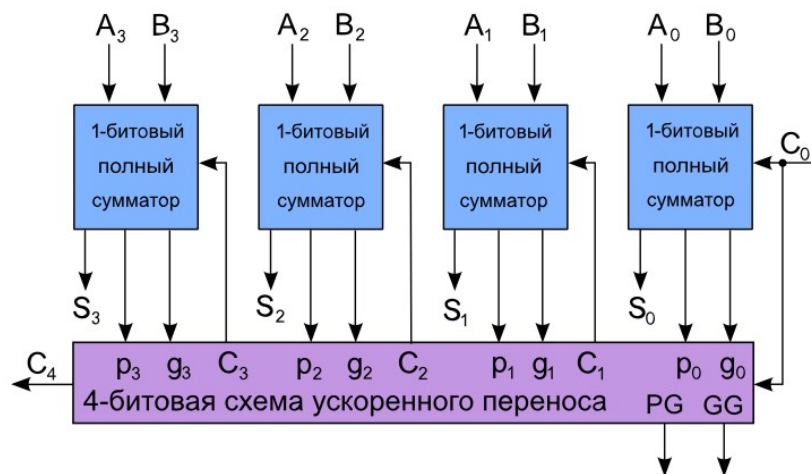


Рисунок 5.4 — Сумматор с ускоренным переносом

Схема ускоренного переноса предполагает использование заранее сформированных сигналов генерации ( $g$ ) и распространения ( $p$ )

$$g_i = a_i \cdot b_i, \quad p_i = a_i \oplus b_i, \quad c_{i+1} = g_i + p_i \cdot c_i.$$

В этом случае каждый сигнал переноса будет описываться следующими логическими выражениями:

$$c_1 = g_0 + p_0 \cdot c_0,$$

$$c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) = g_1 + p_1 g_0 + p_1 p_0 c_0,$$

$$c_3 = g_2 + p_2 c_2 = g_2 + p_2 (g_1 + p_1 c_1) = g_2 + p_2 (g_1 + p_1 (g_0 + p_0 c_0)) = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

и так далее.

Для  $n$ -разрядного сумматора сигнал переноса определяется в общем виде как

$$c_n = G G + P G \cdot c_0,$$

где  $G G = g_{n-1} + p_{n-1} g_{n-2} + p_{n-1} p_{n-2} g_{n-3} + \dots + p_{n-1} p_{n-2} \dots p_1 g_0 = \sum_{i=0}^{n-1} ((\prod_{k=i}^{n-2} p_k) \cdot g_i)$  - групповой сигнал генерации для формирования  $n$ -го разряда переноса,

$P P = \prod_{i=0}^{n-1} p_i$  - групповой сигнал распространения для формирования  $n$ -го разряда переноса.

Как видно, для формирования каждого разряда сигнала переноса необходимо выполнить три операции: сформировать сигналы  $g_i$  и  $p_i$  (выполняются одновременно за время  $\tau$ ), произвести над ними операцию И (за время  $\tau$ ) и потом произвести операцию ИЛИ (за время  $\tau$ ). Таким образом, время формирования каждого бита сигнала переноса производится за время  $T_n^C = 3\tau$ . Сигнал суммы будет в таком случае формироваться за время  $T_n^S = T_n^C + \tau = 4\tau$ .

Таким образом, время задержки формирования сигналов переноса и суммы постоянно и не зависит от разрядности сумматора.

### Сумматор с групповым переносом

Если проанализировать выражение для формирования  $n$ -го разряда переноса в сумматоре с ускоренным переносом, можно заметить, что сложность схемы переноса сильно возрастает с увеличением разрядности сумматора. Для избежания усложнения используют разбиение многоразрядного сумматора на группы меньшей разрядности (от 2 до 8) и соединяют эти группы между собой. Таким образом, можно реализовать два вида групповых сумматоров:

1. с параллельно-последовательным переносом: группу формирует сумматор с ускоренным переносом, группы соединяются последовательно (перенос из предыдущей группы является входным сигналом в текущую группу);
2. с параллельно-параллельным переносом: группу формирует сумматор с ускоренным переносом, группы также соединяются через схему с ускоренным формированием переноса (рисунок 5.5).

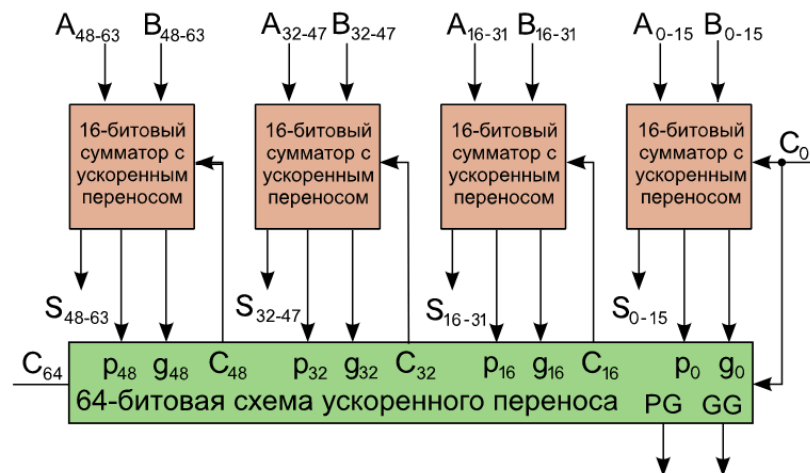


Рисунок 5.5 — Сумматор с групповым (параллельно-параллельным) переносом

Здесь присутствует дополнительная схема ускоренного переноса, которая в качестве сигналов  $p$  и  $g$  использует для формирования переноса  $i$ -й группы сигналы  $PG$  и  $GG$  предыдущей группы. Для рисунка 5.5 сигналы переноса в группы будут определяться как

$$c_{16} = G G_0 + P G_0 c_0,$$

$$c_{32} = G G_1 + P G_1 c_{16} = G G_1 + P G_1 (G G_0 + P G_0 c_0) = G G_1 + P G_1 G G_0 + P G_1 P G_0 c_0$$

и так далее по аналогии с обычным ускоренным переносом.

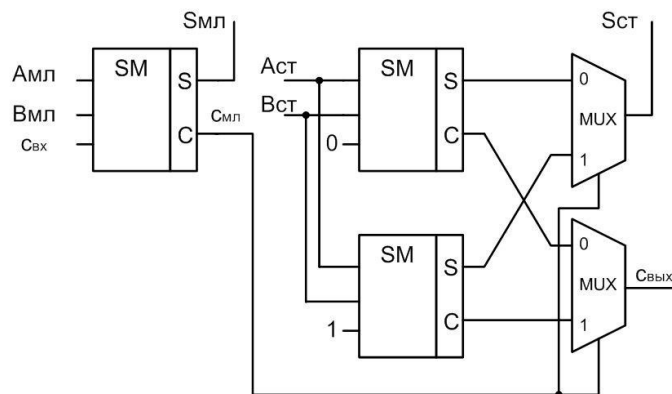
С точки зрения проектирования схема ускоренного переноса для группового переноса ни чем не отличается от обычной схемы ускоренного переноса.

### Сумматор с условным переносом

В этом случае  $n$ -разрядный сумматор делится на две равные группы: один  $(n/2)$ -разрядный сумматор младших разрядов и два  $(n/2)$ -разрядных сумматора старших разрядов. Таким образом, сумматор будет состоять из трёх сумматоров с разрядностью  $n/2$ .

Все сумматоры работают независимо друг от друга. В группе сумматоров старших разрядов один сумматор рассчитывает сумму так, будто переноса из группы младших разрядов нет, а второй — будто есть.

Сигнал переноса из группы младших разрядов появится после окончания суммирования этой группой и сигнал переноса перейдёт на мультиплексоры, которые выбирают результат суммирования одного из двух сумматоров группы старших разрядов (рисунок 5.6).



Такой сумматор позволяет увеличить скорость расчёта в 2 раза по отношению к полному сумматору. Следует отметить, что каждая группа сумматоров может быть любой: с последовательным переносом, ускоренным переносом или групповым переносом.

### Упражнение

1. Сформируйте структурное описание сумматора с последовательным переносом.
2. Оберните его входными и выходными регистрами.
3. Проведите моделирование работы сумматора, убедитесь в его работоспособности.
4. Проведите определение максимальной частоты тактового сигнала, который можно подавать на сумматор без нарушения его работоспособности. Для этого:
  - 4.1. Создайте файл `time.sdc`, задающий временные ограничения в проекте. Содержание файла представлено в листинге 5.4.

#### Листинг 5.4 — Содержание файла `time.sdc`

```
1 #создаём тактовый синхросигнал
2 create_clock -name {clock} -period 50MHz [get_ports {clk}]
3 #убираем все неопределённости
4 derive_clock_uncertainty
5 #убираем не интересующие нас задержки на регистрах
6 #входных от порта clk до проводников перед сумматором
7 #символ * спользуется для многоразрядных проводников и портов
8 set_false_path -from [get_clocks {clock}] -to [get_nets {QA[*]}]
9 set_false_path -from [get_clocks {clock}] -to [get_nets {QB[*]}]
10 set_false_path -from [get_clocks {clock}] -to [get_nets {QP}]
11 #выходных от порта clk до выходных портов
12 set_false_path -from [get_clocks {clock}] -to [get_ports {S[*]}]
13 set_false_path -from [get_clocks {clock}] -to [get_ports {C}]
```

4.2. Откройте настройки временного анализа **Assignments > Settings...** В открывшемся окне выберите вкладку **Timing Analyzer**. В ней в поле **File** выберите файл с временными ограничениями.

4.3. Скомпилируйте дизайн, нажав на *Start Compilation (Ctrl + L)*.

4.4. В появившемся окне отчёта раскройте меню **Timing Analyzer**, затем раскройте меню **Slow 1200 mV 85 C Model** и выберите меню

Fmax Summary. В появившейся таблице будет указана максимальная частота, на которой может работать сумматор. В колонке Fmax restricted указана частота, которой можно добиться на ПЛИС.

## Работа № 6. Одно- и двухпортовая память

Обобщённо память можно рассматривать как двумерный массив запоминающих элементов, каждый из которых хранит один бит данных. Содержимое памяти записывается и считывается по строкам. Строка выбирается адресом (Address). Записанные или считанные значения называются данными (Data). Матрица с N-битным адресом и M-битными данными имеет  $2^N$  строк и M столбцов. Каждая строка данных называется словом. Таким образом, матрица содержит  $2^N$  M-битных слов.

Запоминающие устройства классифицируются по способу хранения битов. Запоминающие устройства делятся на два больших класса: оперативные запоминающие устройства (ОЗУ) (RAM, память с произвольным доступом) и постоянные запоминающие устройства (ПЗУ) (ROM, память только для чтения). Современные ПЗУ не являются постоянными в строгом значении слова: они могут программироваться, т.е. информация в них может записываться. Различие между ОЗУ и ПЗУ состоит в том, что запись в ПЗУ требует больше времени, и они являются энергонезависимыми. В листинге 6.1 приведён пример описания ОЗУ размерностью 64 слова по 32 бита. У этого ОЗУ есть синхронный вход разрешения записи. Другими словами, запись в память происходит по переднему фронту тактового импульса, если сигнал разрешения записи (write enable) we находится в активном состоянии. Чтение происходит немедленно. Непосредственно после включения питания содержимое ОЗУ непредсказуемо. Схема такой памяти приведена на рисунке 6.1.

Листинг 6.1 — SystemVerilog-описание ОЗУ

```
1 module RAM8x16 #(parameter WIDTH = 8,
2                     DEPTH = 16)
3                     (DataIn, Addr, WE, clk
4                     DataOut);
5 input logic [WIDTH-1:0] DataIn;
6 input logic [$clog2(DEPTH)-1:0] Addr;
7 input logic WE;
8 input logic clk;
9 output logic [WIDTH-1:0] DataOut;
10
11 logic [WIDTH-1:0] RAM [DEPTH];
12
13 always_ff @(posedge clk) begin
14     if (!WE) DataOut <= RAM[Addr];
15     else RAM[Addr] <= DataIn;
16 end
17
```



```
18 endmodule
```

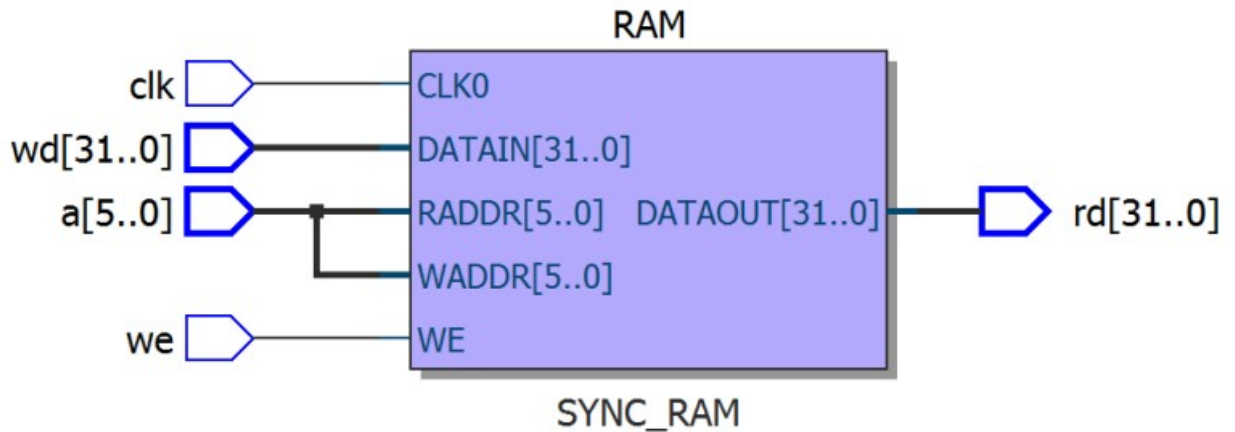


Рисунок 6.1 — Изображение ОЗУ

Пример реализации ПЗУ приведён в листинге 6.2. Для записи заданных заранее значений нужно в модуль памяти добавить блок `initial`, в котором, используя функцию `$readmemh`, указать файл с данными. Схема такой памяти приведена на рисунке 6.2. Таким же образом можно инициализировать не только ПЗУ, но и ОЗУ.

Листинг 6.2 — SystemVerilog-описание ПЗУ

```
1 module RAM8x16 #(parameter WIDTH = 8,  
2                     DEPTH = 16)  
3                     (DataIn, Addr, RE, clk,  
4                     DataOut);  
5  
6 input logic [$clog2(DEPTH)-1:0] Addr;  
7 input logic RE;  
8 input logic clk;  
9 output logic [WIDTH-1:0] DataOut;  
10  
11 logic [WIDTH-1:0] RAM [DEPTH];  
12  
13 initial begin  
14     $readmemh("RAM.txt", ROM);  
15 end  
16 always_ff @(posedge clk) begin  
17     if (RE) DataOut <= RAM[Addr];  
18 end  
19  
20 endmodule
```

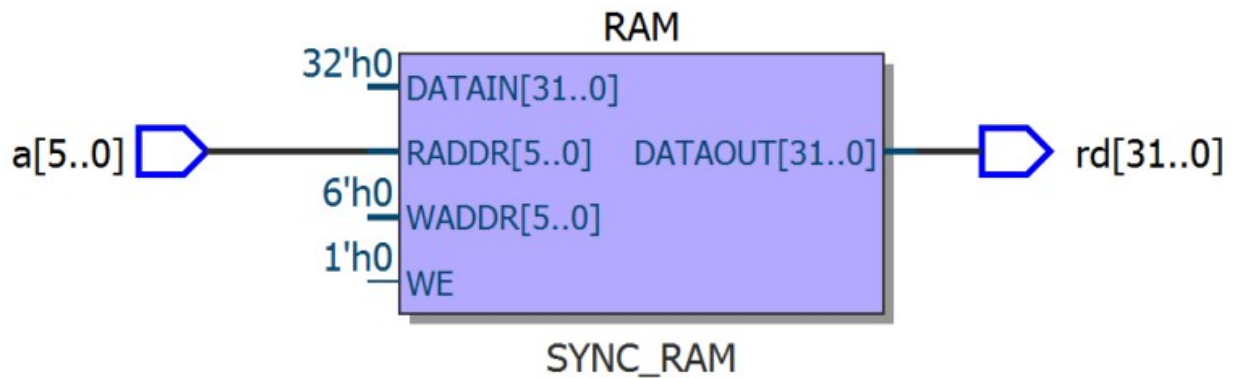


Рисунок 6.2 — Изображение ПЗУ

Для инициализации переменных в тестовый модуль можно после создания экземпляра тестируемого модуля добавить блок `initial`. Далее значения этих переменных можно менять или не менять в процедурном блоке `always`. В листинге 6.3 приведён шаблон тестирующего модуля, в котором задаётся значение входа `writable` равным нулю и нигде не меняется.

Листинг 6.3 — Шаблон тестирующего модуля

```

1  `timescale 1 ns / 1 ns
2  module tb_RAM8x16();
3
4  parameter PERIOD = 20; //период тактового сигнала 20 нс
5
6  bit clk, WE;
7  logic [WIDTH-1:0] DataIn;
8  logic [$clog2(DEPTH)-1:0] Addr;
9  logic [WIDTH-1:0] DataOut;
10
11  RAM8x16 #(
12      .WIDTH(8),
13      .DEPTH(16)
14  ) DUT (
15      .DataIn(DataIn),
16      .Addr(Addr),
17      .clk(clk),
18      .WE(WE),
19      .DataOut(DataOut));
20
21  initial begin
22      WE = 0;
23      ...
24  end
25
26  initial begin
27      clk = 0;
28      forever #(PERIOD/2) clk = ~clk;
29  end

```

```

30
31 always @(negedge clk) begin
32     Addr <= ...;
33     $display(...);
34 end
35
36 initial begin
37     $display("Start TEST");
38     #... $display("Stop TEST");
39     $stop;
40 end
41
42 endmodule

```

## Упражнение

1. Спроектируйте на языке SystemVerilog модуль 32-разрядной памяти объёмом 64 слова.
2. Напишите для неё тестирующий модуль, в котором проверьте, что в памяти находятся неопределённые значения.
3. Запишите число, отличное от нуля, по адресу N, где N – номер по журналу.
4. Проверьте, что число записалось по заданному адресу, а в остальных ячейках по-прежнему находятся неопределённые значения.
5. Спроектируйте на языке SystemVerilog второй модуль 32-разрядной ОЗУ объёмом 32 слова. Проинициализируйте этот модуль заданными значениями. Предварительно расширьте количество записываемых значений до 30 ячеек.

```

00000000
00000000
00000055
00000000
00020007
00000000
00000000
00000044
00000000
00001043
01000008
00000000
00003040

```

6. Скопируйте из проинициализированной памяти значение, записанное по адресу N, в первый блок памяти по адресу N\*3.

7. Убедитесь, что значение скопировалось по заданному адресу, а в остальных ячейках находятся неопределённые значения.
8. Спроектируйте на языке SystemVerilog модуль 32-разрядной ПЗУ. Проинициализируйте ПЗУ заданными значениями.
9. Скопируйте значение из ПЗУ с адреса b000010 в ОЗУ по адресу N.
10. Убедитесь, что значение скопировалось по заданному адресу, а в остальных ячейках находятся неопределённые значения.

# Работа № 7. Использование семисегментного индикатора. Функции

## Семисегментный дисплей

### Функции и процедуры

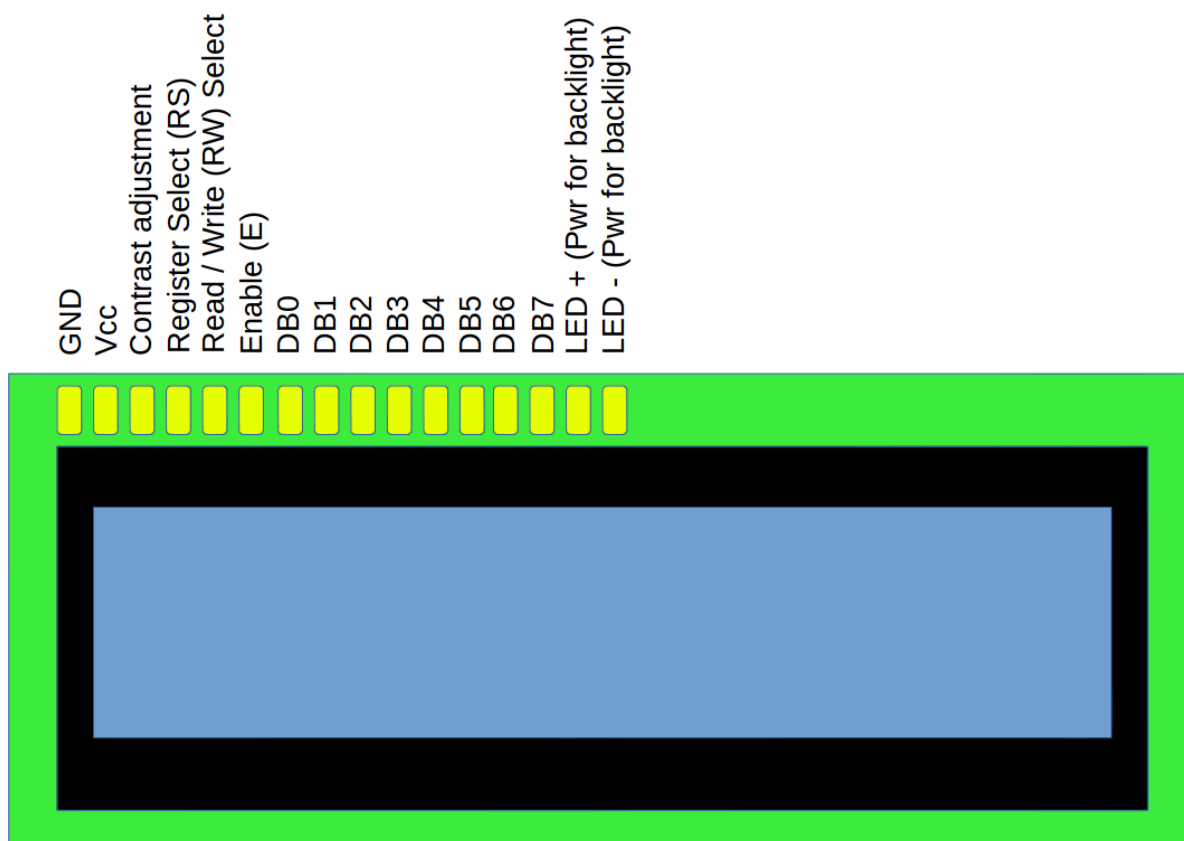
Для улучшения читаемости и сопровождения кода удобно выделять часть либо повторяющуюся, либо смысловую в отдельный блок - *функцию* (function) или *процедуру* (task). Основные отличия между этими формами заключаются в следующем:

- в функции нельзя использовать управление временем, в процедуре можно (использовать операторы задержек #, ожидания события @ и пр.);
- функция, в общем случае, возвращает какое-либо значение (поддерживается оператор return), процедура - нет;
- 

### Упражнения

1. Выведите на отладочной плате на семисегментном дисплее дату своего рождения.
2. Выведите на отладочной плате на семисегментном дисплее слово, которое будет перемещаться циклически влево каждые полсекунды.
3. Выведите на отладочной плате предложение.

## Работа № 8. Подключение дисплея LCD1602 к ПЛИС по 8-битной шине



Коды символов для дисплеев с поддержкой русского языка. Первый символ – 1, второй – номер столбца, третий – номер строки.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				Ø	Ǿ	Ɔ	`	Ɔ			Б	Ю	Ч	.	Д	Ў
1			!	1	А	Q	а	q			Г	Я	Ш		Ц	Ў
2			"	2	В	R	в	г			Ё	Б	Ъ		Щ	Ў
3			#	3	С	S	с	s			Ж	В	Ы	!!	Ѡ	ѡ
4			\$	4	D	T	d	t			Э	Г	Ь	Ɔ	Ф	Ѡ
5			%	5	E	U	e	u			И	ё	Э	Х	Ц	Ѡ
6			&	6	F	V	f	v			Й	Ж	Ю	Ɔ	Щ	ѡ
7			'	7	G	W	g	w			Л	Э	Я	I	'	Ѡ
8			(	8	H	X	h	x			П	И	€	И	''	Ѡ
9			)	9	I	Y	i	y			У	Й	€	†	˘	Ѡ
A			*	:	J	Z	j	z			Ф	К	€	↓	Ѡ	Ѡ
B			+	;	K	[	k	l			Ч	Л	"	†	Ѡ	Ѡ
C			,	<	L	φ	l	l2			Ш	М	Ѡ	†	ij	Ѡ
D			-	=	M	]m	m	l5			Ъ	Н	€	†	†	Ѡ
E			.	>	N	^	n	€			Ы	П	f	Ɔ	˘	Ѡ
F			/	?	O	_	o	e			Э	Т	€	•	o	■

### Задания

1. Подключите к отладочной плате дисплей LCD1602. На основе выданного преподавателем HDL-кода выведите на первой строке дисплея свою фамилию, а на второй - имя.
2. Модифицируйте HDL-описание, добавив блок ROM-памяти, который хранит ASCII-коды до 0x7F. Сформируйте вывод фразы, выдающей десятичный код числа, которое вводится dip-переключателями (число от 0 до 255), в виде "Input number is:".

3. Модифицируйте HDL-описание таким образом, чтобы на экране выводилось предложение в виде перемещающегося объявления слева направо.
4. Модифицируйте HDL-описание таким образом, чтобы можно было вводить текст оператору. Задайте двумя кнопками key перемещение курсора влево или вправо, третьей кнопкой - прокрутку символов для выбора. Сформируйте на основе этого вывод своего имени. *Указание: создайте модуль `debounce` (англ. `debounce` - антидребезг) для ликвидации многократного нажатия кнопок при фактическом одном нажатии.*



**Работа № 9. Передача данных между ПК и ПЛИС.  
Последовательный интерфейс U(S)ART**

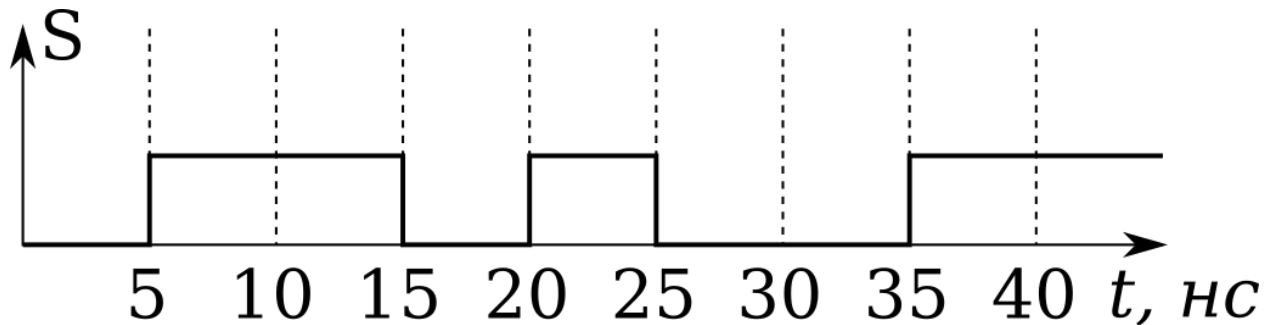
**Работа № 10. Передача данных между ПЛИС.  
Последовательные интерфейсы GPIU, SPI и I2C**

## **Работа № 11. Интерфейс VGA**

**Работа № 12. Подключение клавиатуры к ПЛИС.  
Интерфейс PS/2**

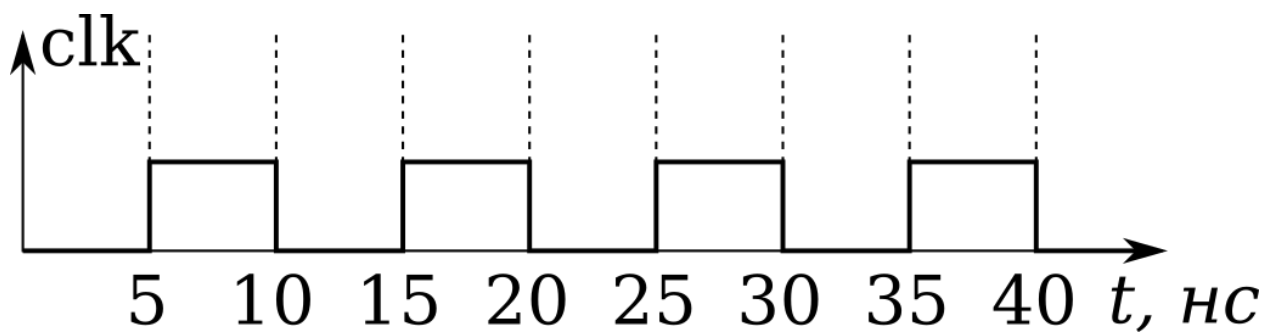
# Приложение А

## Задание сигналов для модуля тестирования



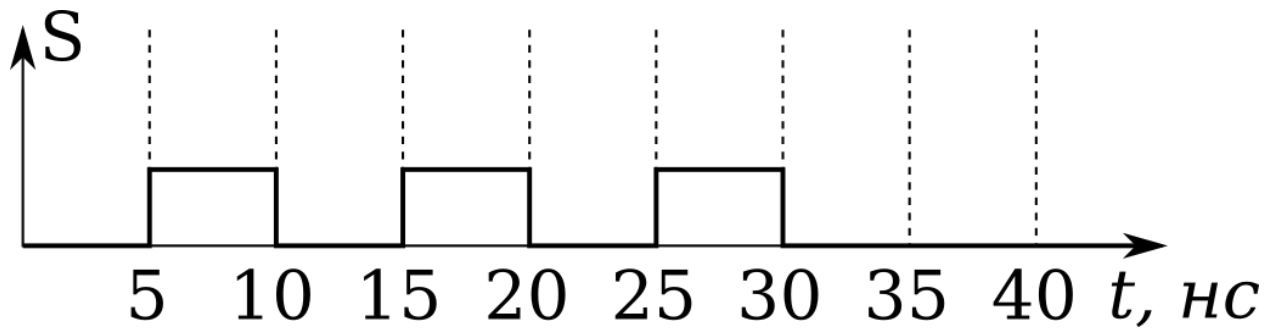
Листинг А.1 - Задание сигнала, произвольно меняющегося во времени

```
1 //указывается задержка каждого изменения сигнала S после
2 //предыдущего изменения
3 initial begin
4     S = 1'b0;
5     #5 S = 1'b1;
6     #10 S = 1'b0;
7     #5 S = 1'b1;
8     #5 S = 1'b0;
9     #10 S = 1'b1;
10 end
```



Листинг А.2 - Задание сигнала, периодически меняющегося во времени

```
1 initial begin
2     clk = 1'b0;
3     //бесконечное количество раз каждые 5 нс меняется сигнал clk
4     forever begin
5         #5 clk = ~clk;
6     end
7 end
```

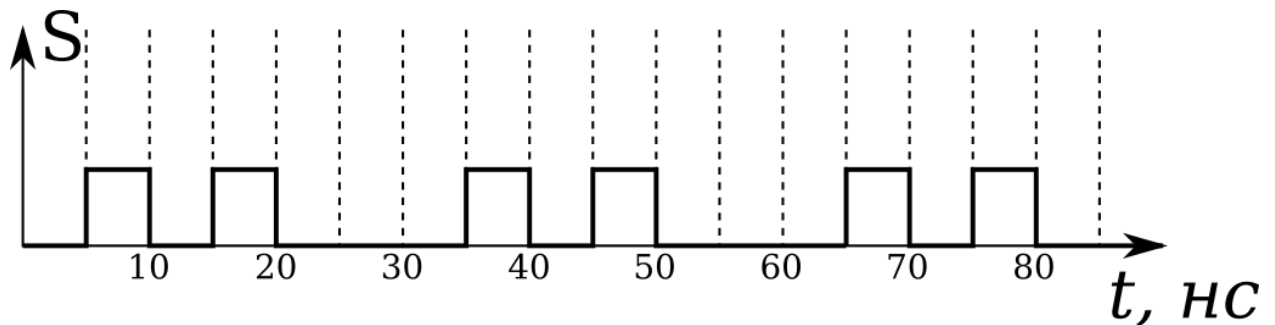


Листинг А.3 - Задание сигнала, периодически меняющегося во времени 3 раза

```

1 initial begin
2   S = 1'b0;
3   //6 раз сигнал S претерпевает изменения каждые 5 нс
4   repeat(6) #5 S = ~S;
5 end

```



Листинг А.4 - Задание сигнала произвольной формы, периодически меняющегося во времени 3 раза

```

1 initial begin
2   S = 1'b0;
3   repeat(3) begin //3 раза повторяются первые 30 нс
4     //в каждом цикле сигнал S претерпевает 4 изменения каждые 5 нс
5     repeat(4) #5 S = ~S;
6     #10;
7   end
8 end

```

Листинг А.5 - Задание сигнала произвольной формы, периодически меняющегося во времени бесконечное количество раз

```

1 initial begin
2   S = 1'b0;
3   forever begin //всегда повторяются первые 30 нс
4     //в каждом цикле сигнал S претерпевает 4 изменения каждые 5 нс
5     repeat(4) #5 S = ~S;
6     #10;
7   end
8 end

```

## Приложение Б

### Использование Tcl-скриптов для автоматизации процесса проектирования

Создание проекта, синтез, размещение и трассировка, а также временной анализ и асемблирование с последующим формированием файла прошивки ПЛИС являются операциями рутинными. Их можно описать при помощи скриптового языка TCL (Tool Command Language) в файле скрипте и автоматически выполнять весь поток проектирования на ПЛИС от создания проекта до формирования файла прошивки.

Ниже представлены листинги скриптов для отдельных этапов работы с ПЛИС.

#### Листинг Б.1 - Создание проекта сумматора Adder

```
1 project_create Adder
2 #указываем семейство и имя ПЛИС
3 set_global_assignment -name FAMILY "Cyclone IV E"
4 set_global_assignment -name DEVICE "EP4CE6E22C8"
5 #указываем sv-файл(-ы), который(-е) должны быть в проекте
6 set_global_assignment -name SYSTEMVERILOG_FILE top.sv
7 #добавляем файл с констрейнами (при необходимости)
8 set_global_assignment -name SDC_FILE time.sdc
9 #указываем имя модуля верхнего уровня
10 set_global_assignment -name TOP_LEVEL_ENTITY top
11 #закрываем проект
12 project_close
```

#### Листинг Б.2 - Открытие проекта сумматора Adder (rev.tcl)

```
1 project_open Adder
2 #указываем sv-файл(-ы), который(-е) должны быть в проекте
3 set_global_assignment -name SYSTEMVERILOG_FILE top.sv
4 #добавляем файл с констрейнами (при необходимости)
5 set_global_assignment -name SDC_FILE time.sdc
6 #указываем имя модуля верхнего уровня
7 set_global_assignment -name TOP_LEVEL_ENTITY top
8 #задаём пины для ввода значений (dip-переключатели)
9 #и тактовый синхроимпульс
10 set_location_assignment PIN_23 -to clk
11 set_location_assignment PIN_68 -to A[0]
12 set_location_assignment PIN_67 -to A[1]
13 set_location_assignment PIN_66 -to A[2]
14 set_location_assignment PIN_65 -to A[3]
15 set_location_assignment PIN_64 -to B[0]
16 set_location_assignment PIN_60 -to B[1]
17 set_location_assignment PIN_59 -to B[2]
18 set_location_assignment PIN_58 -to B[3]
```

```
19 set_instance_assignment -name IO_STANDART "3.3-V LVTTTL" -to clk
20 set_instance_assignment -name IO_STANDART "3.3-V LVTTTL" -to A[0]
21 set_instance_assignment -name IO_STANDART "3.3-V LVTTTL" -to A[1]
22 set_instance_assignment -name IO_STANDART "3.3-V LVTTTL" -to A[2]
23 set_instance_assignment -name IO_STANDART "3.3-V LVTTTL" -to A[3]
24 set_instance_assignment -name IO_STANDART "3.3-V LVTTTL" -to B[0]
25 set_instance_assignment -name IO_STANDART "3.3-V LVTTTL" -to B[1]
26 set_instance_assignment -name IO_STANDART "3.3-V LVTTTL" -to B[2]
27 set_instance_assignment -name IO_STANDART "3.3-V LVTTTL" -to B[3]
28 #компиляция проекта
29 #необходим пакет flow, по умолчанию не подгружен
30 load_package flow
31 #полный поток компиляции Quartus
32 execute_flow -compile
33 #закрываем проект
34 project_close
```

```
quartus_map top
quartus_fit top
quartus_asm top
quartus_sta top --sdc=time.sdc
quartus_sh --flow compile top
quartus_sh -t rev.tcl
quartus_pgm -l
quartus_pgm -c USB-BlasterII -a
```