

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 проектирования электронной компонентной базы» и «Синтез
цифровых интегральных схем»
для студентов, обучающихся по направлениям
11.03.04 и 11.04.04 «Электроника и нанoeлектроника»

Работа № 1. Простая комбинационная логика. Непрерывное присваивание....	3
Работа № 2. Сложная комбинационная логика. Блокирующее присваивание	26
Работа № 3. Последовательностная логика. Неблокирующее присваивание. Параметризация.....	34
Работа № 4. Конечные автоматы.....	35
Приложение А.....	36

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:

- Icarus Verilog
- GTKWave

Работа № 1. Простая комбинационная логика. Непрерывное присваивание

Выполните задание согласно варианта, представленного в таблице 1.1:

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.
2. Для этих же функций сформируйте два файла описания на языке Verilog: с применением логических операций и в виде списка соединений.
3. Для п. 2 напишите общий testbench, произведите моделирование в ModelSim (или iVerilog + GTKWave). Пр продемонстрируйте идентичность результатов моделирования.

Таблица 1.1 – 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	$y_0 = \overline{x_0} \cdot (x_1 \oplus x_2),$

№ варианта	Логические выражения
	$y_1 = x_1 \oplus (\overline{x_0} \cdot x_2),$ $y_2 = x_0 \cdot x_1 + \overline{x_2}.$
5, 12, 19, 26	$y_0 = x_0 \oplus x_1 \oplus \overline{x_2},$ $y_1 = \overline{x_0} + \overline{x_2},$ $y_2 = x_0 + \overline{x_1} \cdot x_2.$
6, 13, 20, 27	$y_0 = \overline{x_0} + \overline{x_1},$ $y_1 = \overline{x_0} \oplus x_1,$ $y_2 = \overline{x_0} + \overline{x_1} \oplus x_2.$
7, 14, 21, 28	$y_0 = \overline{x_1} + x_2,$ $y_1 = \overline{x_0} \oplus x_2,$ $y_2 = \overline{x_0} \oplus \overline{x_1} + x_2.$

Указания:

Для утилит iVerilog + GTKWave используйте следующие команды в терминале:

```
> iverilog <file.v>
```

```
> vvp a.out
```

```
> gtkwave dump.vcd
```

Выполнение задания № 1

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*.

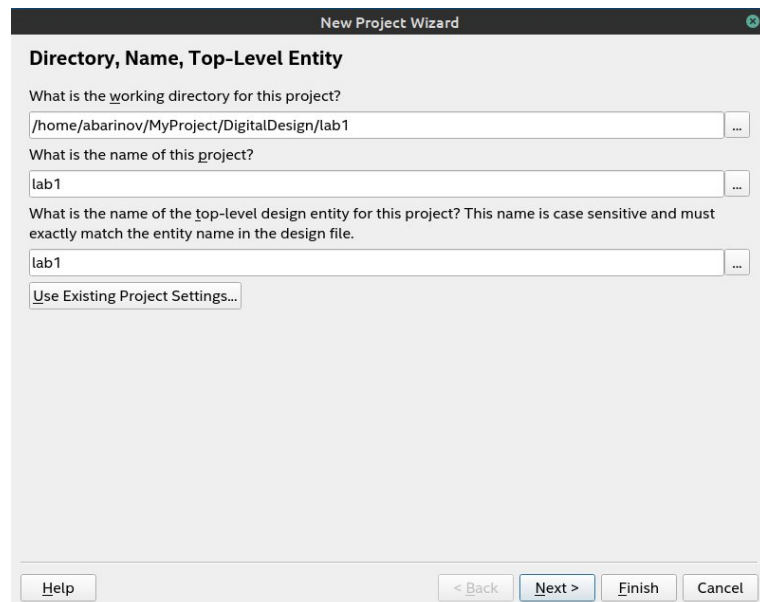


Рисунок 1.1 – Создание проекта

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

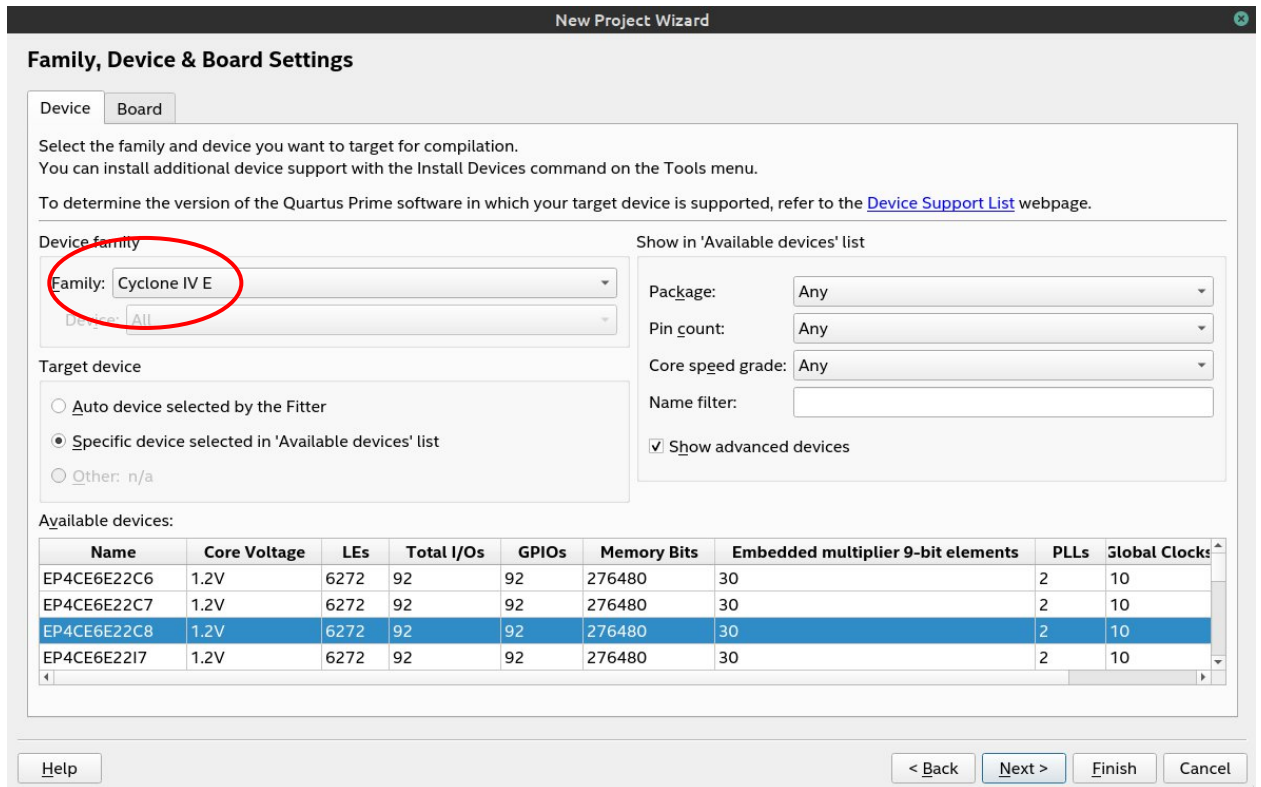


Рисунок 1.2 – Выбор ПЛИС

8. В следующем окне укажите инструмент моделирования ModelSim, а язык Verilog HDL. На текущем этапе моделирование проводится не будет, а пройдёт 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*.

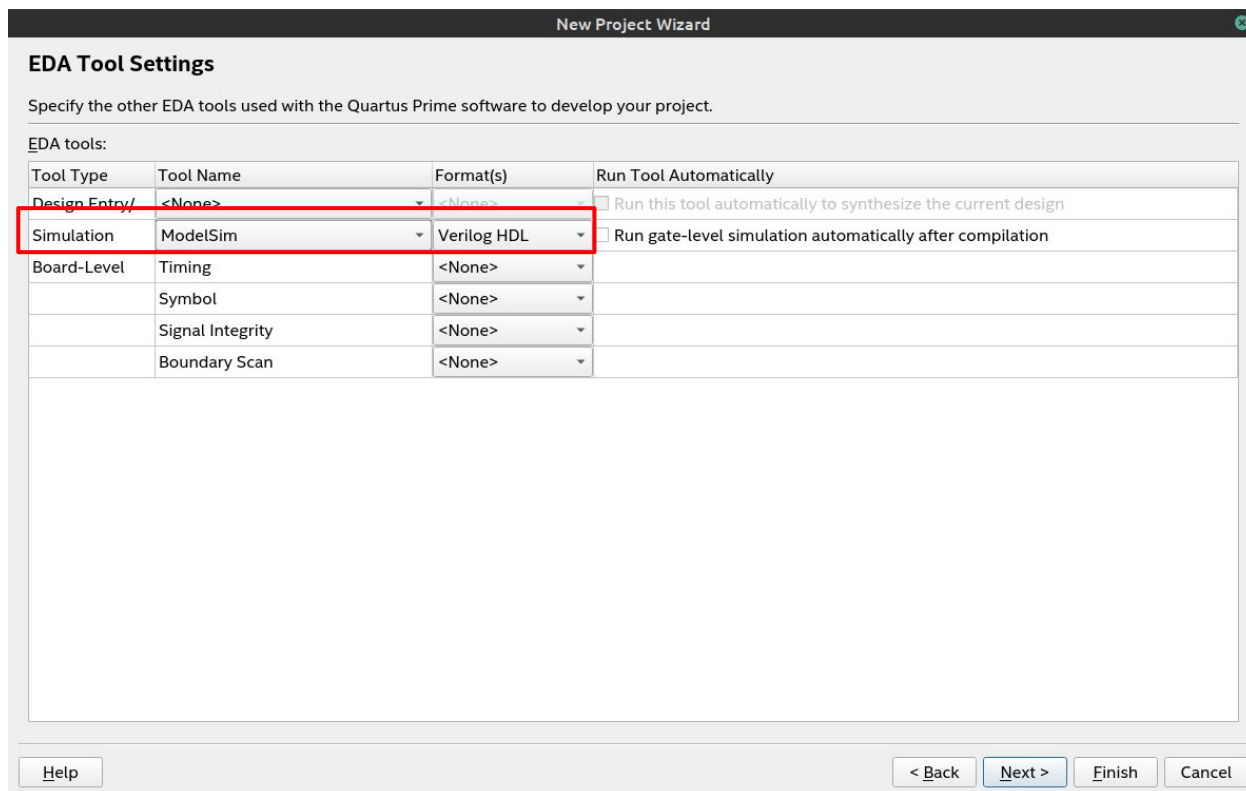


Рисунок 1.3 – Настройка инструментов

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**.

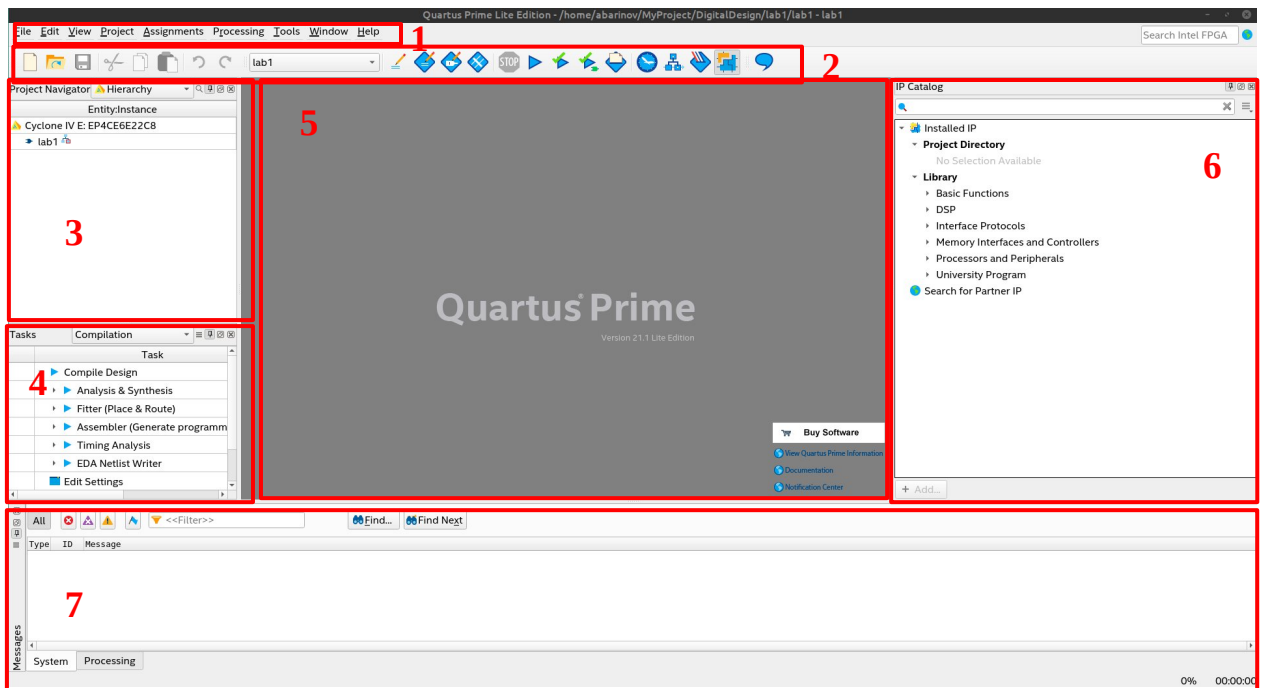


Рисунок 1.4 – Основное окно Quartus Prime:

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.

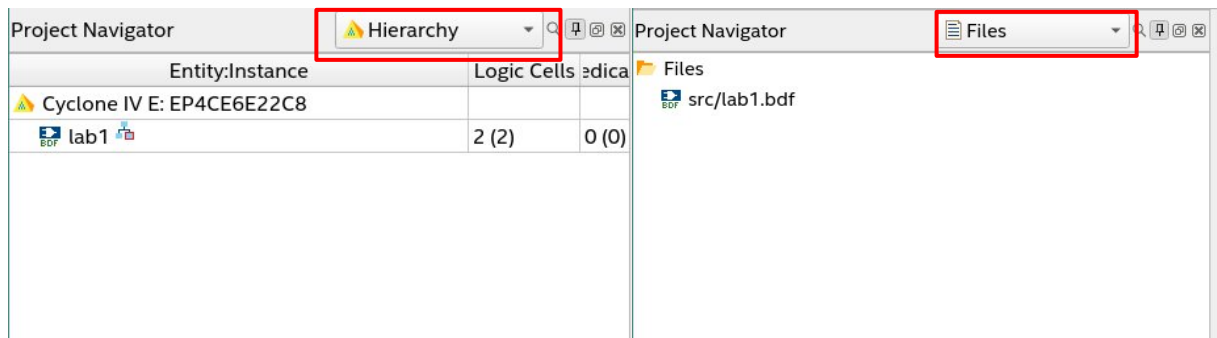


Рисунок 1.5 - Панель Project Navigator

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**.

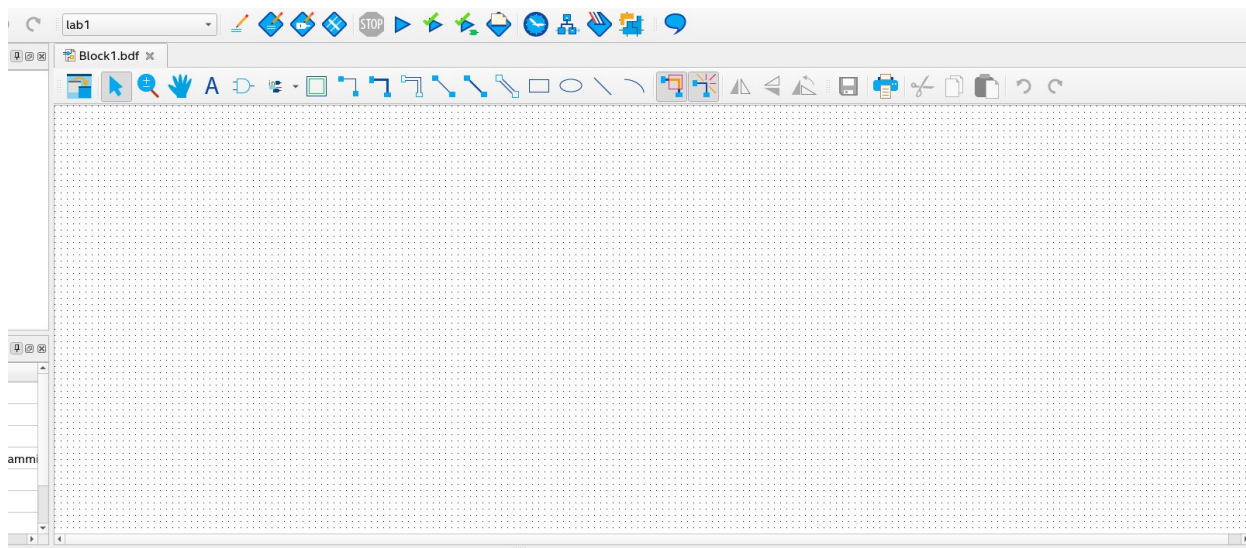





Рисунок 1.6 – Поле для рисования логической схемы

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.

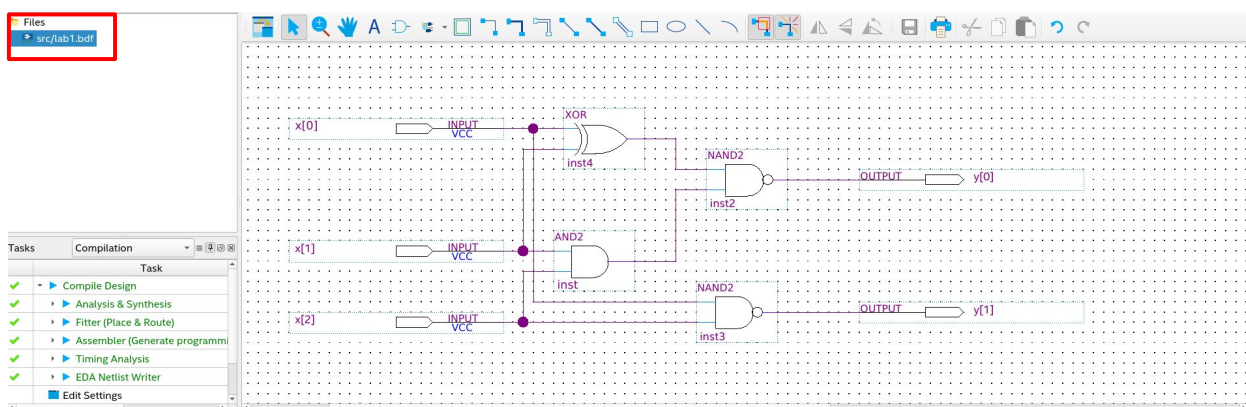

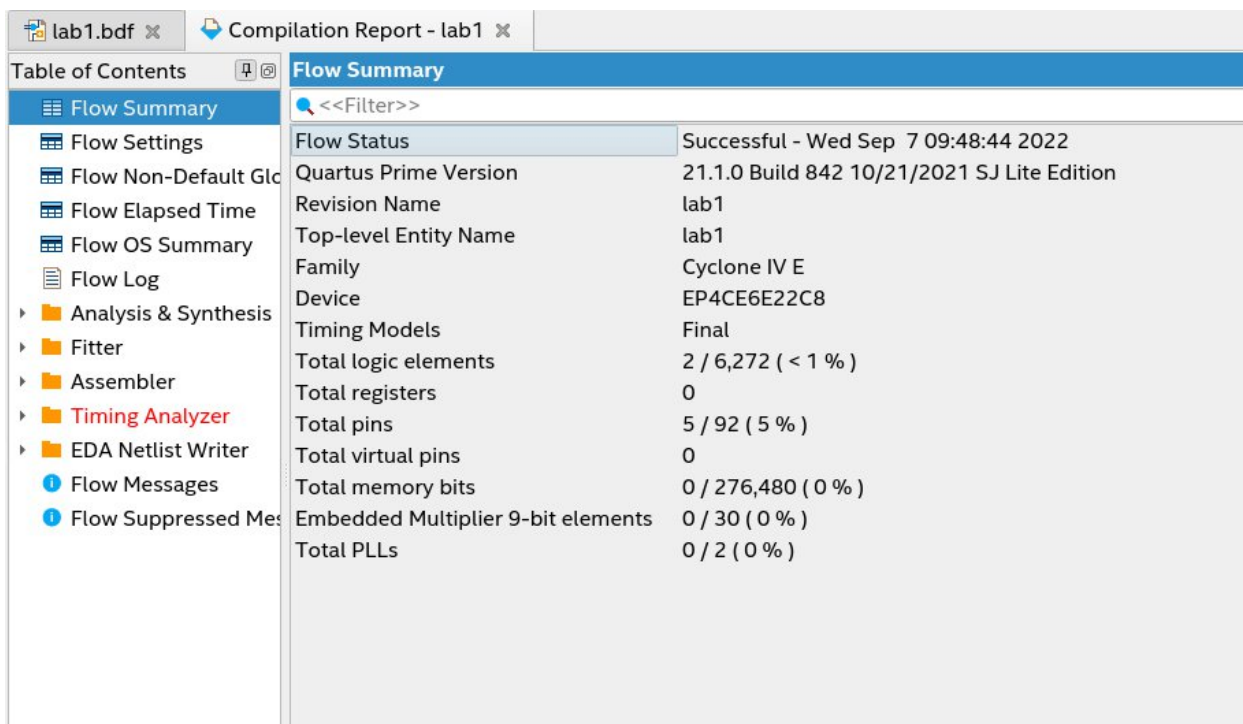


Рисунок 1.7 – Графическое изображение логической схемы

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	
<<Filter>>	
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.

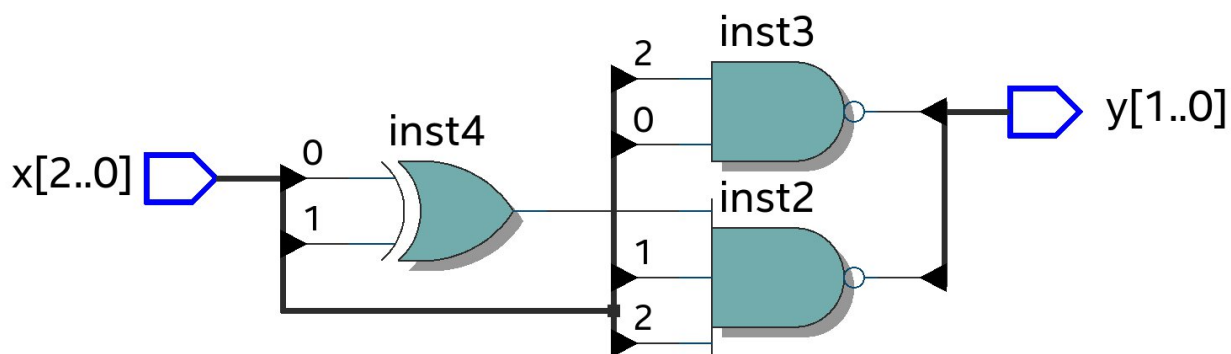


Рисунок 1.9 – RTL-представление схемы

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

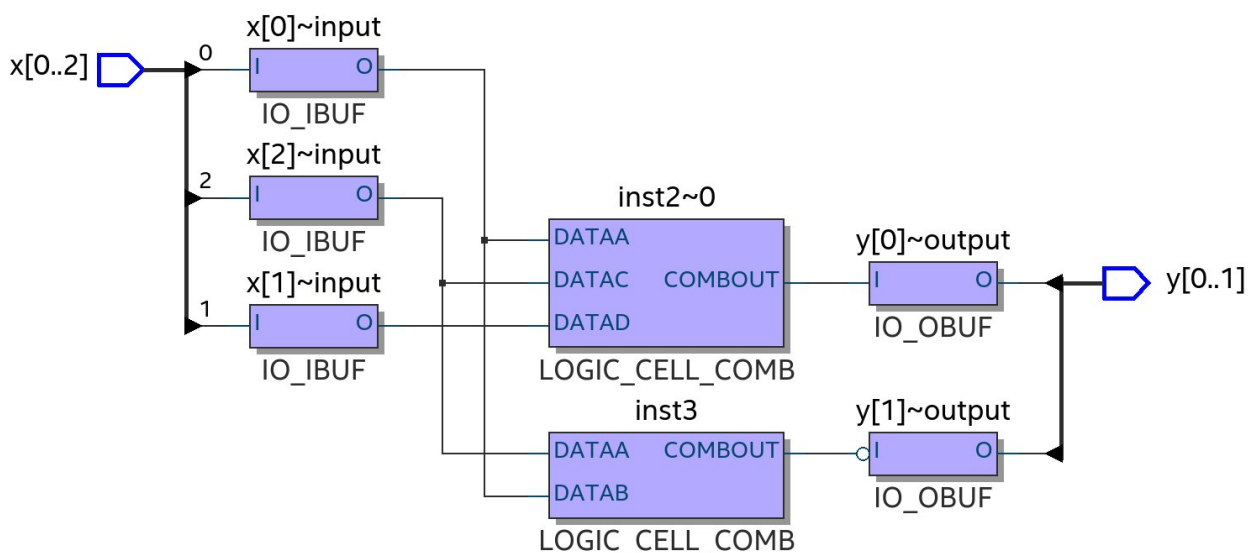


Рисунок 1.10 – Используемые логические элементы ПЛИС

11. Проанализируйте каждую из 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.

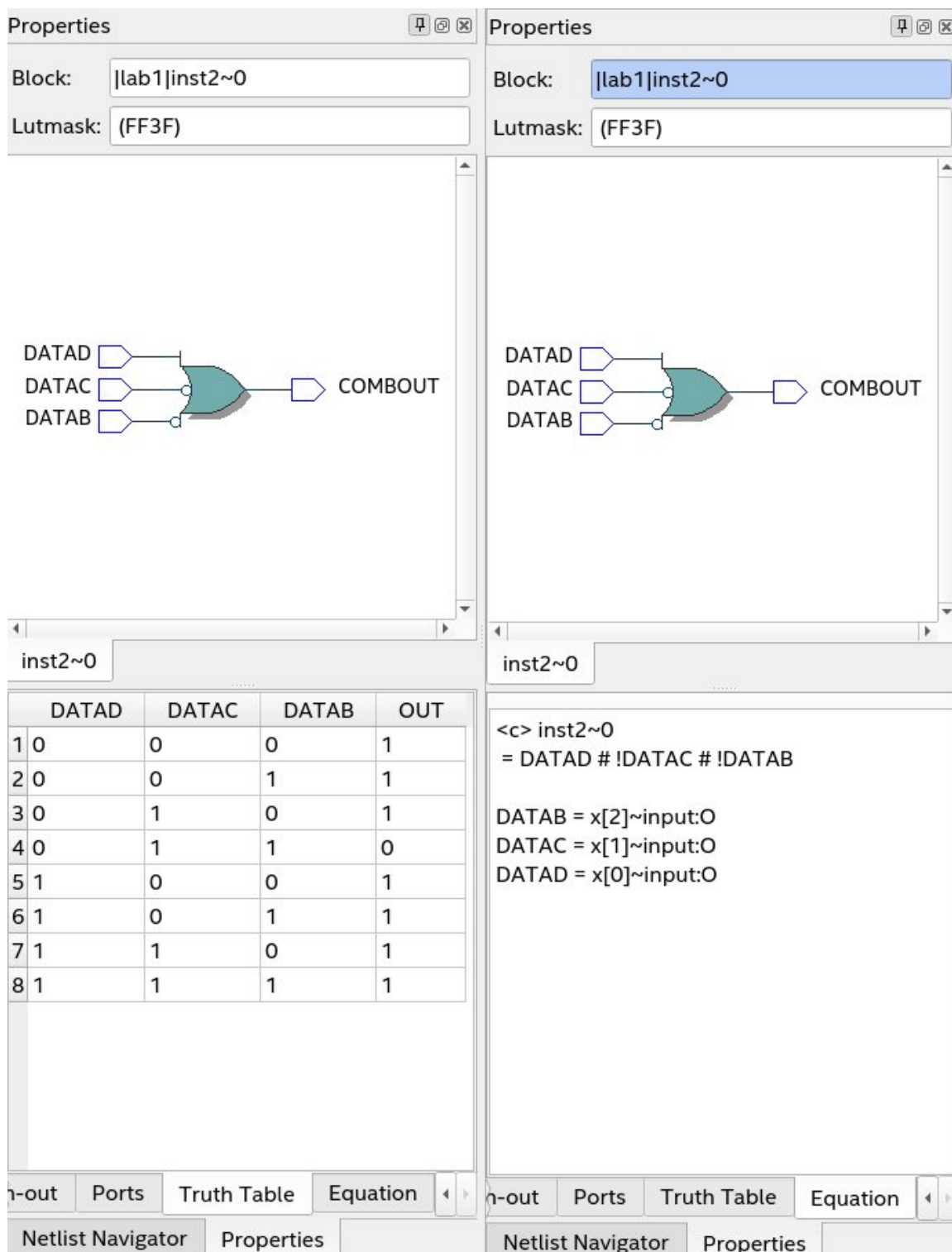


Рисунок 1.11 - Панель свойств логического элемента ПЛИС

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 LVTTL 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.

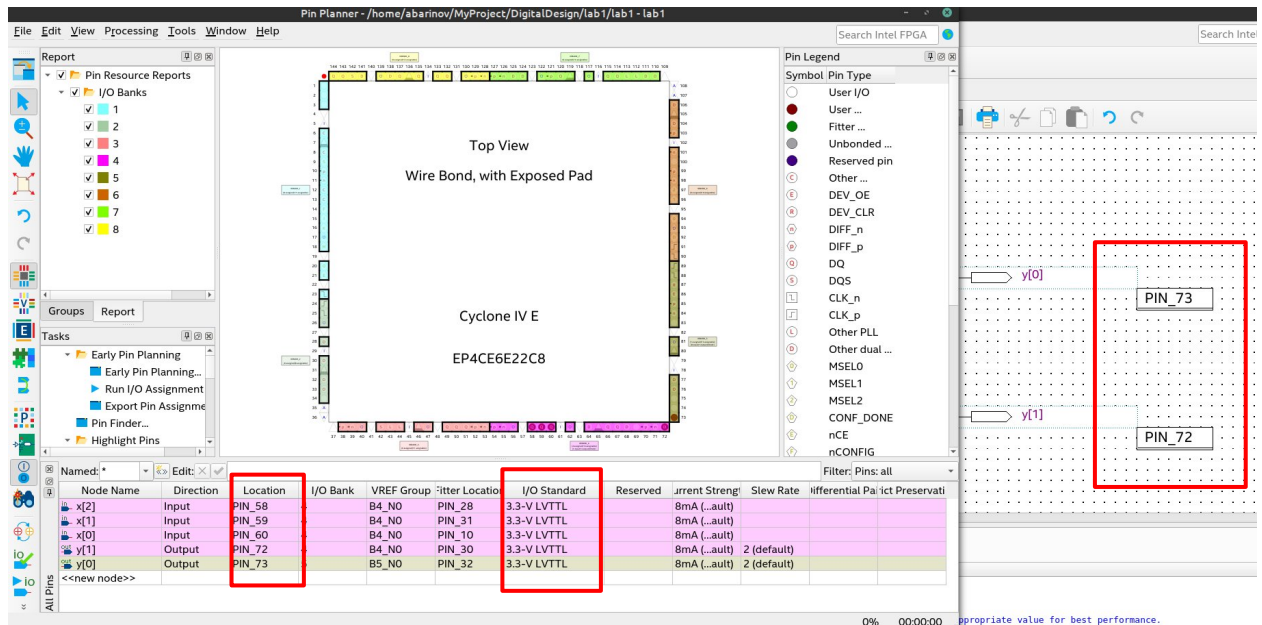


Рисунок 1.12 – Окно задания соответствия пинов Pin Planner

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*.

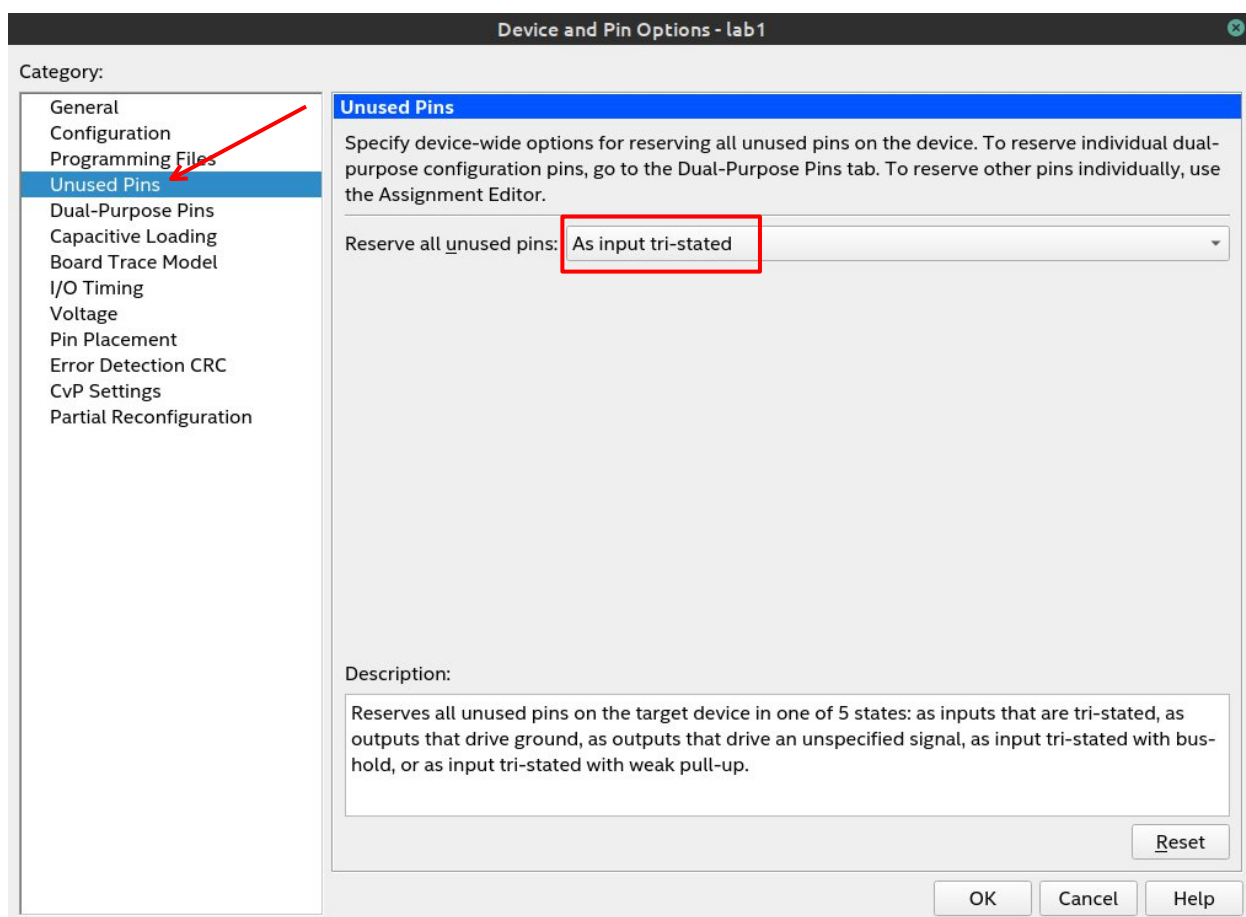


Рисунок 1.13 – Настройка неиспользуемых пинов ПЛИС

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.

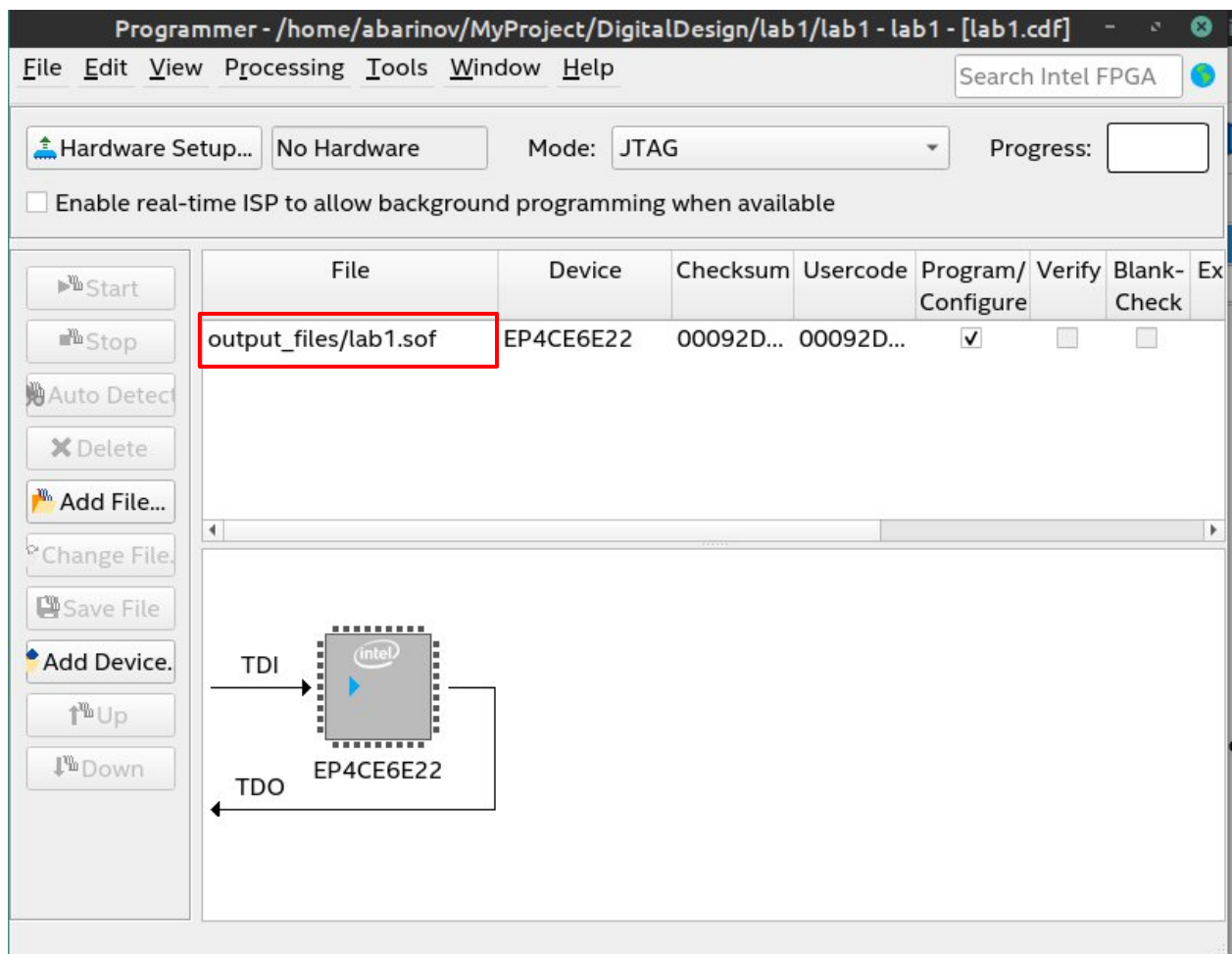


Рисунок 1.14 – Окно Programmer

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*.

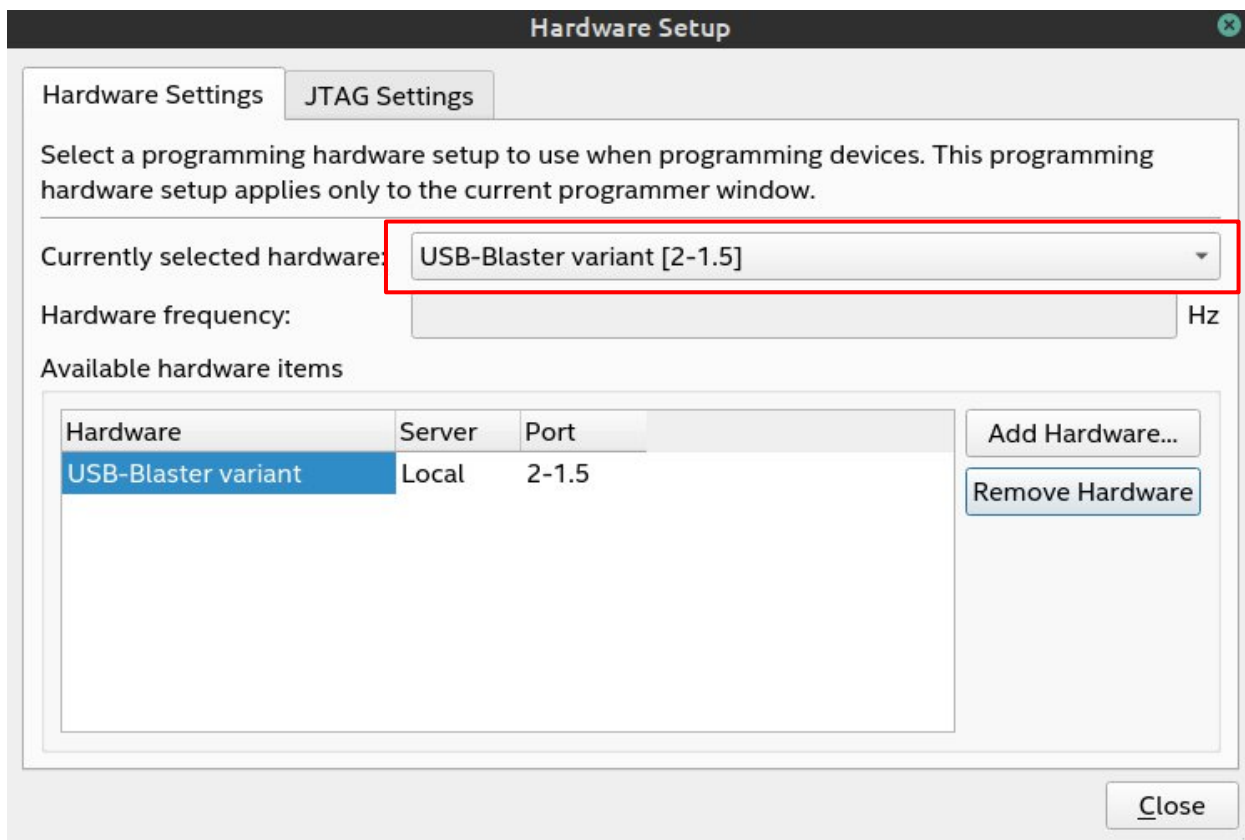


Рисунок 1.15 – Выбор программатора

8. 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.

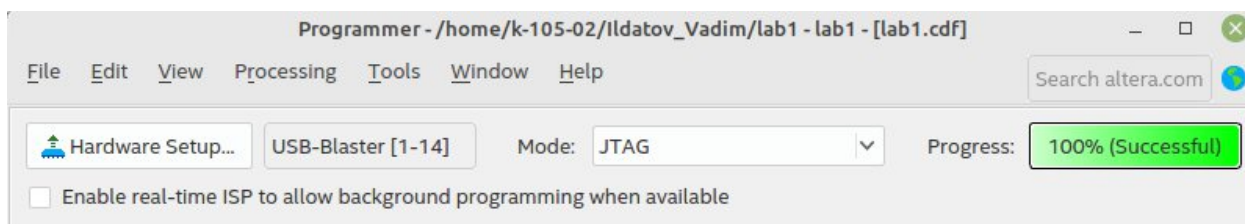


Рисунок 1.16 – Успех процедуры прошивки ПЛИС

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.

Выполнение задания № 2

1. В том же проекте создайте новый файл **File > New...** Выберите *Verilog HDL File*. На рабочей части откроется файл **Verilog1.v**.
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.v**. Убедитесь, что файл стал отображаться в Project Navigator.

Используемые цифровые примитивы для структурного описания и логические выражения для поведенческого описания представлены в таблице 1.2. При использовании цифровых примитивов в скобках указываются сначала выходы, а затем входы. Количество входов не ограничено.

Таблица 1.2 - Логические операции и вентили, используемые в Verilog-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	$y = a \cdot \bar{b}$	$y = \sim(a \& b)$	<code>nand(y, a, b);</code>
ИЛИ-НЕ	$y = a + \bar{b}$	$y = \sim(a b)$	<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	$y = a \oplus \bar{b}$	$y = \sim(a \wedge b)$	<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  [2:0] x; //входной сигнал
4  output [1:0] y; //выходной сигнал
5
6  wire    [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**), а также представляет собой трёхразрядную шину с индексацией старшего разряда (LSB, least significant bit, англ. *наименее значащий бит*) - 2, а младшего (MSB, most significant bit, англ. *наиболее значащий бит*) - 0. Аналогично в строке 5 для выходного (**output**) сигнала **y** указано, что он двухразрядный с индексом старшего разряда - 1, а младшего - 0. Для задания портов входа / выхода используется конструкция:

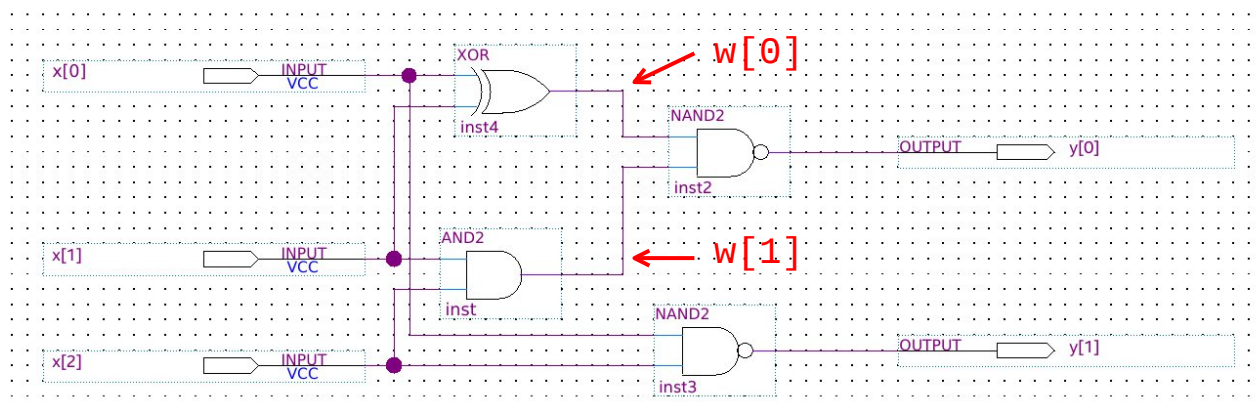
<тип_порта> [тип_сигнала] [[LSB:MSB]] <имя_порта>;

В угловых скобках указаны обязательные атрибуты, в квадратных – необязательные. Тип порта может быть входным **input**, выходным **output** и двунаправленным **inout**. Тип сигнала в основном либо провод **wire**, либо регистровый **reg**. Если тип сигнала не указан, то по умолчанию используется **wire**. Для многоразрядных сигналов (шины) необходимо указать разрядность и индексацию младшего и старшего разрядов **[LSB:MSB]**, для одноразрядного сигнала этого делать не нужно. Рекомендуется в проектах использовать единый стиль нумерации, например, для *n*-разрядной шины младший разряд - 0, старший – *n*-1.

В строке 7 введён дополнительный сигнал для наименования проводников, соединяющих логические вентили внутри схемы, которые не являются входными или выходными сигналами. Так как провод в схеме соединяет логические вентили, то он имеет тип **wire**. Внутри схемы примера всего два провода, в связи с чем сигнал **w** объявлен двухразрядным. По аналогии с портами сигналы записываются конструкцией

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

В строках 9-12 записаны цифровые примитивы согласно схеме на рисунке 1.17.



Завершает описание модуля ключевое слово **endmodule** на строке 14.

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

3. Аналогично сформируйте файл **lab1_beh.v** с описанием поведенческой модели **lab1_beh**.

Поведенческое описание отличается от структурного тем, что нет 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 1.2 с примером записи логических выражений на Verilog описание модуля будет выглядеть следующим образом.

Листинг 1.2 – Поведенческое описание логической схемы


```

1  module lab1_beh(x, y);
2
3  input  [2:0] x;
4  output [1:0] y;
5
6  wire   [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 логического выражения справа от него. Таким образом для задания простой логики используется конструкция

assign <сигнал> = <логическое_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.v**, 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 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...**

Выполнение задания № 3

1. Сформируйте в проекте файл для тестирования **tb_lab1.v**, расположите его в папке /src.

Тестбенч представляет собой отдельный отладочный модуль без входных и выходных сигналов, в котором подключается тестируемый 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 понадобится подключить к модулю тестбенча файл с тестируемым модулем. Для этого используется препроцессорная директива **``include`**

`include` "<имя_файла>"

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

`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 входных сигналов используется сигнал с типом **reg** (его значение задаётся в процессе моделирования, он представляет собой элемент памяти, который сохраняет своё значение до тех пор, пока ему не будет присвоено новое), а для сигналов, которые подключаются к выходам тестируемого модуля – тип **wire** (его значение формируется в процессе моделирования, представляет собой «проводник»).

- *подключение тестируемого модуля или модулей*

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

```
<имя_модуля> <имя_экземпляра> ( .<имя_порта> (<имя_сигнала>) [ ,  
                                .<имя_порта> (<имя_сигнала>),  
                                .<имя_порта> (<имя_сигнала>),  
                                ... ] );
```

При подключении нескольких 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 reg bt, bt;
2 wire ct;
3 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*. 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 значении, 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 C

```

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

```

Пример описания тестового модуля представлен в листинге 1.3.

Листинг 1.3 – Описание тестового модуля для схемы из примера

```

1 //препроцессорная директива для подключения файлов модулей
2 `include «lab1_beh.v»

```

```

2 `include «lab1_struct.v»
3
4 //препроцессорная директива для определения временного масштаба
5 `timescale 1ns / 1ps
6
7 module tb_lab1;
8
9 //объявляем входные данные
10 reg [2:0] xt;
11
12 //объявляем выходные данные
13 wire [1:0] yt;
14
15 //подключаем устройства
16 lab1_struct DUT1( .x( xt ), .y( yt ) );
17 lab1_beh DUT2( .x( xt ), .y( yt ) );
18
19 //задаём изменение входного сигнала 'x' с шагом в 10 нс
20 initial begin
21     x = 3'b000; //в начале моделирования сигнал 'x' равен 000
22     #10 x = 3'b001; //ждём 10 нс и меняем сигнал на 001
23     #10 x = 3'h7; //спустя ещё 10 нс сигнал равен 111
24     #10 x = 3'h4; //через 10 нс сигнал меняется на 100
25 end
26
27 //задаём время моделирования 50 нс и останавливаем его
28 initial
29     #50 $stop;
30
31 //отмечаем в терминале значения сигналов при каждом их изменении
32 //или изменении времени
33 initial begin
34     $display(«Start test»);
35     $monitor(«At time %t 'x' = %b and 'y' = %b», $time, xt, yt);
36 end
37
38 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*

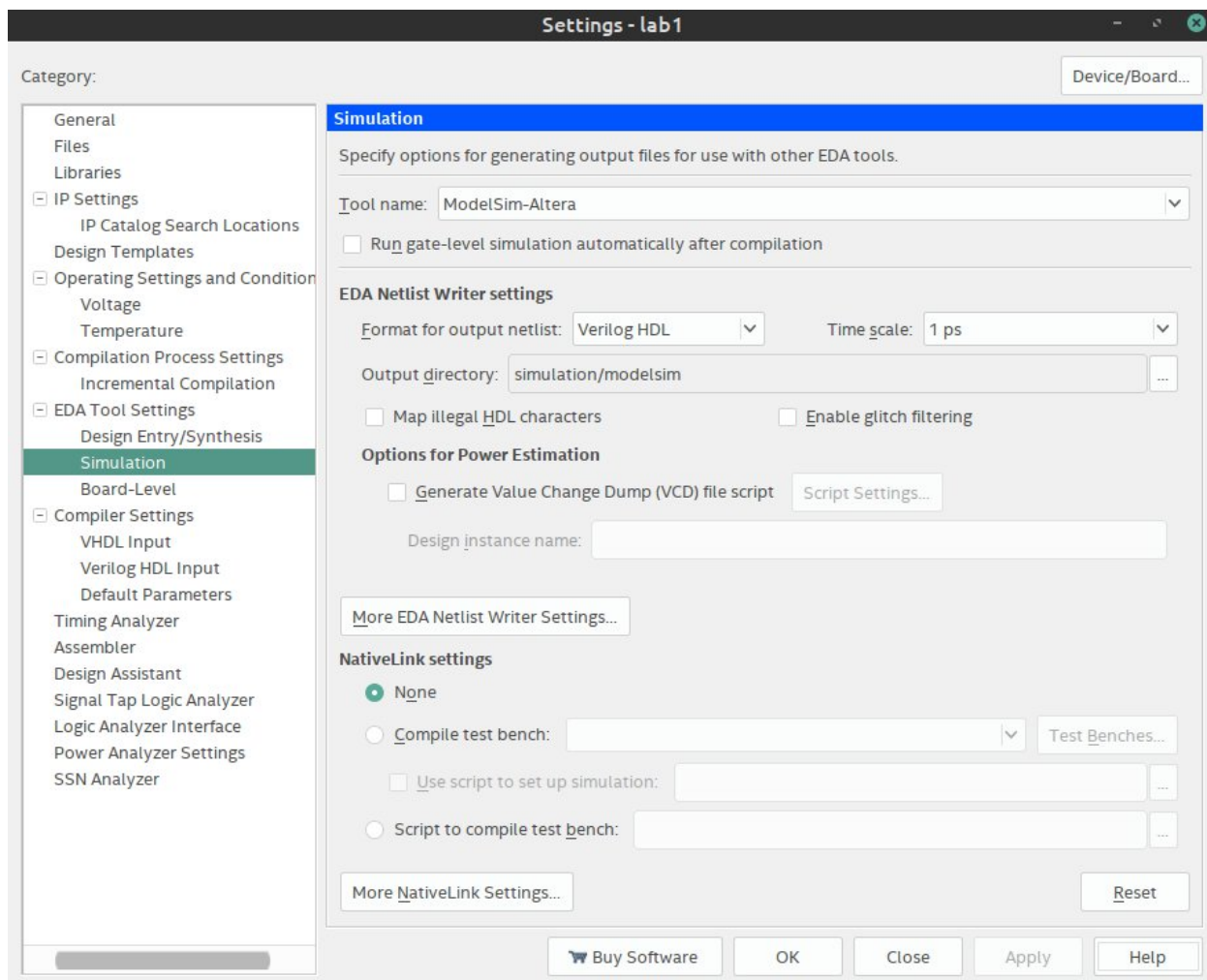


Рисунок 1.18 – Окно настроек параметров моделирования

- 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 *OK*.
- 2.7. В окне списков тестбенчей должен появиться добавленный тест. Нажмите в нём *OK*.
- 2.8. В окне настроек также нажмите *OK*.
- 2.9. Произведите синтез модуля, который необходимо протестировать.
- 2.10. Запустите моделирование **Tools > Run Simulation Tool > RTL Simulation**.

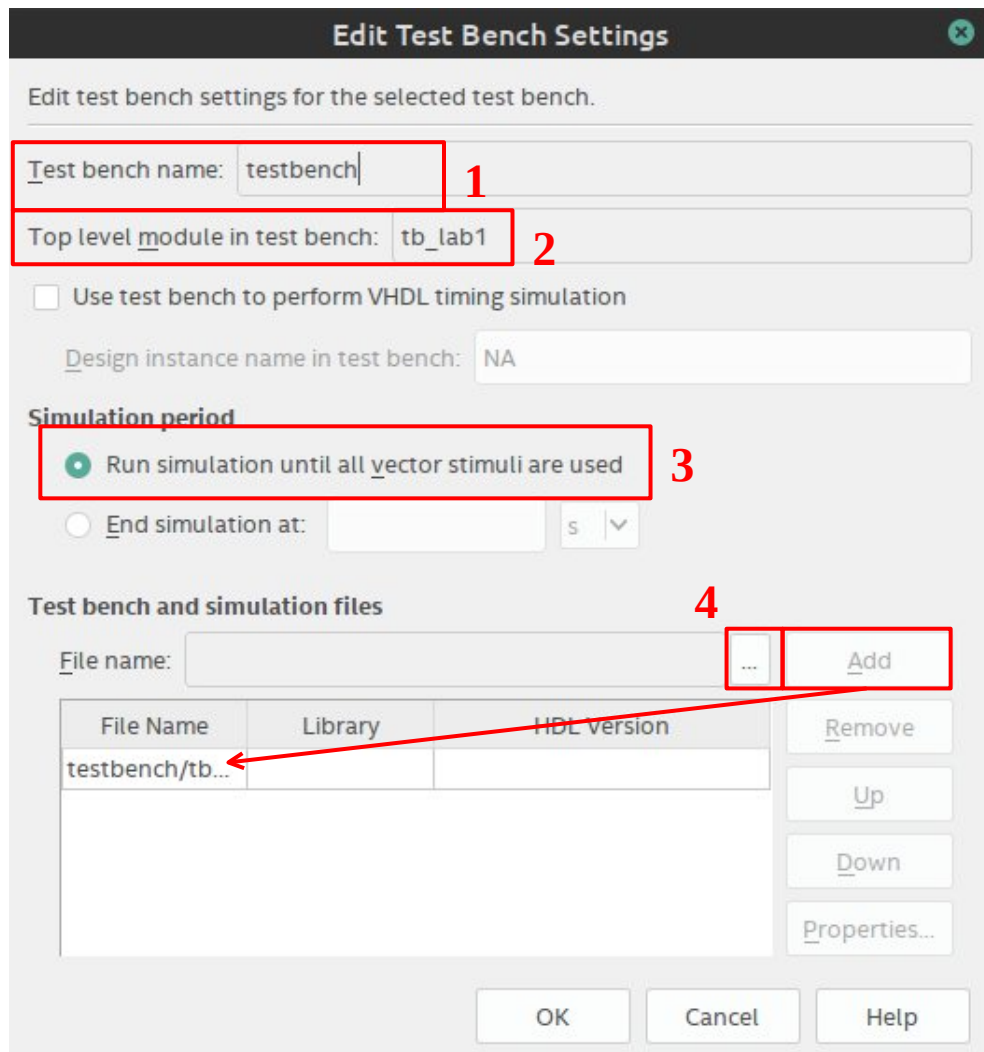



Рисунок 1.19 – Настройка тестбенча

- 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).

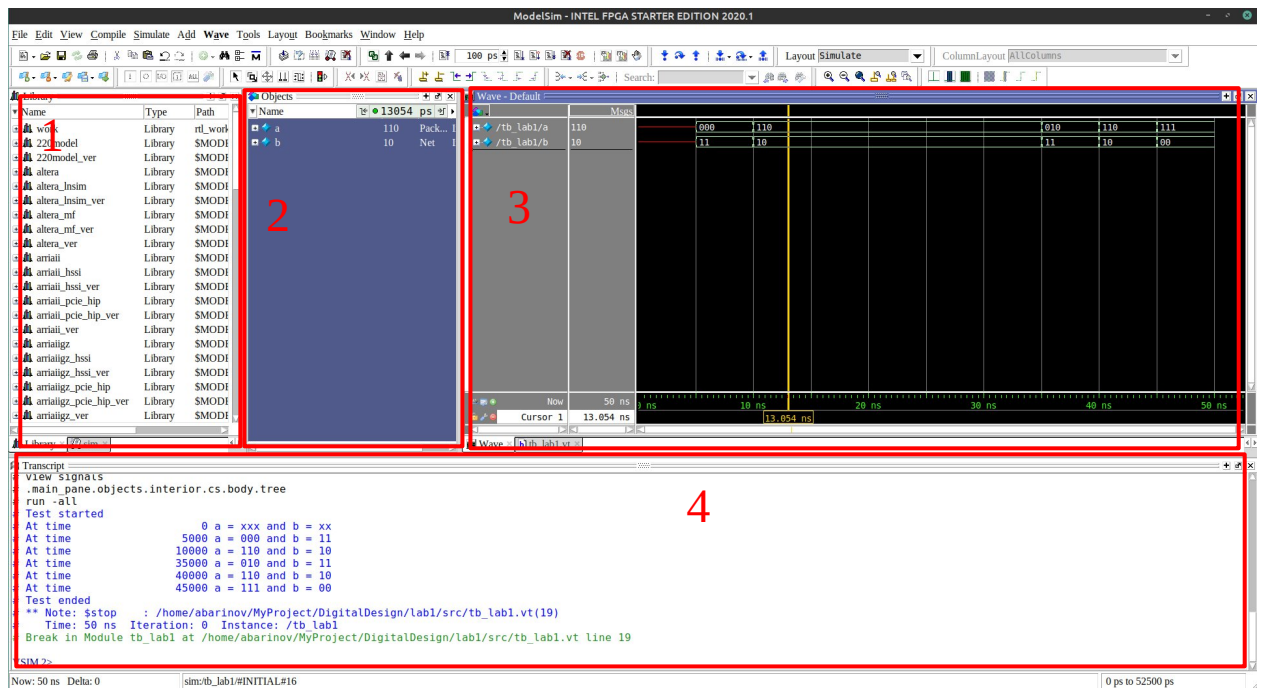


Рисунок 1.20 – Окно ModelSim:

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

Работа № 2. Сложная комбинационная логика. Блокирующее присваивание

Работа № 1 была посвящена проектированию простой комбинационной логики. Если в качестве поведенческого описания можно задать логическое выражение, то используется оператор непрерывного присваивания **assign**, а сигнал, которому присваивается значение логической функции должен иметь тип **wire**.

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

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

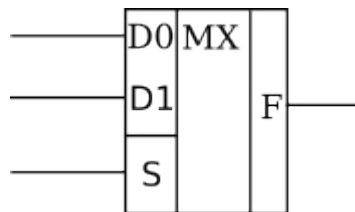


Рисунок 2.1 – Мультиплексор «из 2 в 1»

В этом случае сигнал F легко описывается оператором непрерывного присваивания

Листинг 2.1 – Описание мультиплексора «из 2 в 1»

```
1 module mux2l_assign(D, S, F);
2
3 input  [1:0] D;
4 input      S;
5 output     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 mux4l_assign(D, S, F);
2
```

```

3 input  [3:0] D;
4 input  [1:0] S;
5 output          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**

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 Verilog для такой формализации используется последовательный оператор ветвления **if...else if... else....** Этот оператор записывается внутри процедурного блока **always**.

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

always @ (a or b)

...

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. При этом, если изменений значений на входах нет, то сигнал F должен «помнить» своё значение, которое он принял ранее. Поэтому сигнал F должен иметь тип **reg**.

Следует запомнить, что все сигналы, значение которых формируется в процедурном блоке **always**, должны иметь тип **reg**.

Сам синтаксис оператора ветвления выглядит как

```

if (<условие>) begin
    <действия, если условие истинно>
end
else begin
    <действия, если условие ложно>
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    [3:0] D;
4 input    [1:0] S;
5 output reg    F;
6
7 always @ (D or S) 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** 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 Verilog

Сравнение	Запись
равенство	$A == B$
больше	$A > B$
меньше	$A < B$
больше или равно	$A \geq B$
меньше или равно	$A \leq B$

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

Оператор выбора **case**

Оператор ветвления полезен при сравнении небольшого количества условий. Если же их много, то структура оператора «лесенкой» выглядит не удачно. Для большого числа сравнений и для наглядности удобно

использовать последовательный оператор выбора **case**. Как и оператор ветвления он выполняется внутри процедурного блока **always**.

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

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

В таком случае, мультиплексор из нашего примера может быть описан как

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

```
1 module mux41_case(D, S, F);  
2  
3 input      [3:0] D;  
4 input      [1:0] S;  
5 output reg      F;  
6  
7 always @ (D or S) begin  
8     case ( S )  
9         2'b00 : F = D[0];  
10        2'b01 : F = D[1];  
11        2'b10 : F = D[2];  
12        2'b11 : F = D[3];  
13        default : F = 1'bz;  
14    endcase  
14 end  
15  
16 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 блока **always** 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 в третье высокооимпедансное Z-состояние.

Что будет, если мы не учтём все комбинации и не отметим вариант по умолчанию? В этом случае синтезатор сформирует D-защёлку – элемент памяти. Рассмотрим пример с мультиплексором «из 3 в 1». Для кодирования трёх входных состояний требуется 2-битная шина выбора (адресная шина). Но она может кодировать 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 default	с default
<pre> 1 module mux31_case(D, S, F); 2 3 input [2:0] D; 4 input [1:0] S; 5 output reg F; 6 7 always @ (D or S) 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 [2:0] D; 4 input [1:0] S; 5 output reg F; 6 7 always @ (D or S) 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 присваивания с применением специальной конструкции, именуемой *тернарный оператор*

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

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

```
1 module mux2l_tern(D, S, F);
2
3 input [1:0] D;
4 input      S;
5 output     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

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

```
1 module mux4l_tern(D, S, F);
2
3 input [3:0] D;
4 input [1:0] S;
5 output     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 [3:0] D;
4 input [1:0] S;
5 output      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** сигнал 'b' примет значение сигнала 'a', следом за этим сигнал 'c' примет значение сигнала 'b', которое уже равно 'a', потом аналогично повторит значение сигнала 'a' сигнал 'd', и в завершении выходной сигнал приобретает значение сигнала 'a'.

Листинг 2.9 – Блокирующее присваивание

```
1 module test(a, f);
2
3 input a;
4
5 output reg f;
6
7 reg b, c;
8
9 always @ (a) 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).

Задание

1. Приведите HDL-описание шифратора «из 8 в 3» и дешифратора «из 4 в 8» с использованием последовательных операторов **if...else** и **case**.
2. Проведите синтез схем и продемонстрируйте RTL-представление обоих вариантов.
3. Приведите HDL-описание приоритетного шифратора «из 3 в 8» 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.

Работа № 3. Последовательная логика. Неблокирующее присваивание. Параметризация

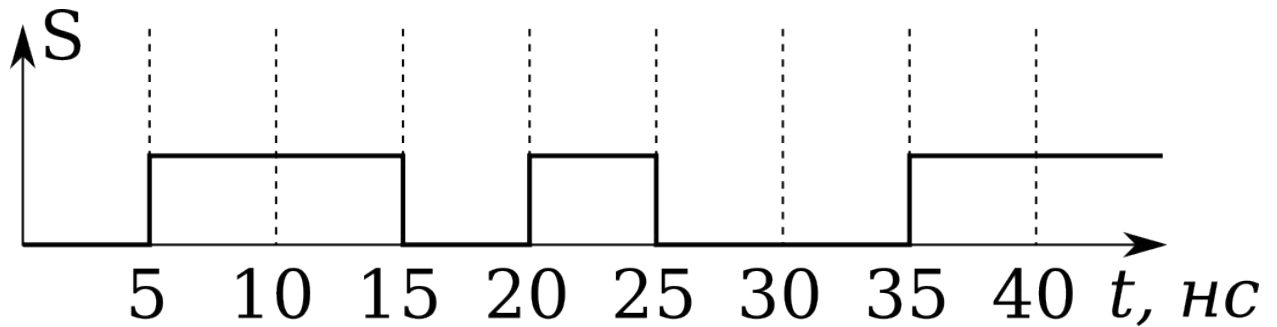
Проектирование счётчиков, регистров.

Работа № 4. Конечные автоматы

Проектирование последовательностной логики с использованием концепции конечных автоматов.

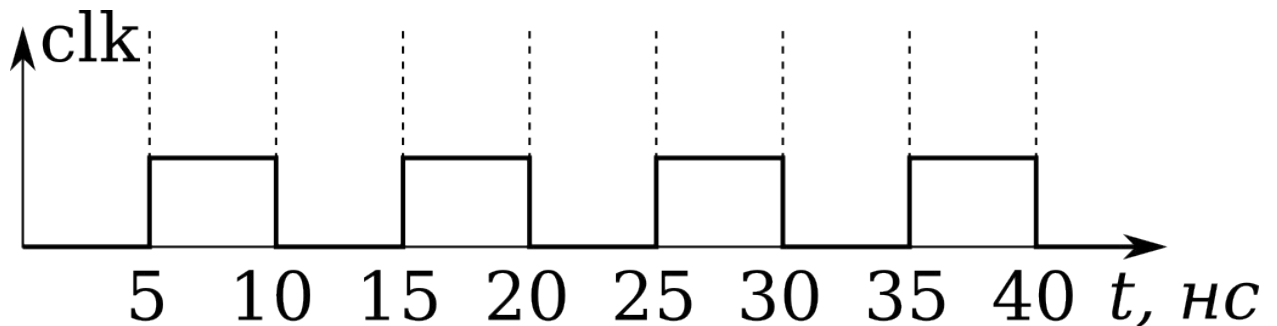
Приложение А

Задание сигналов для модуля тестирования



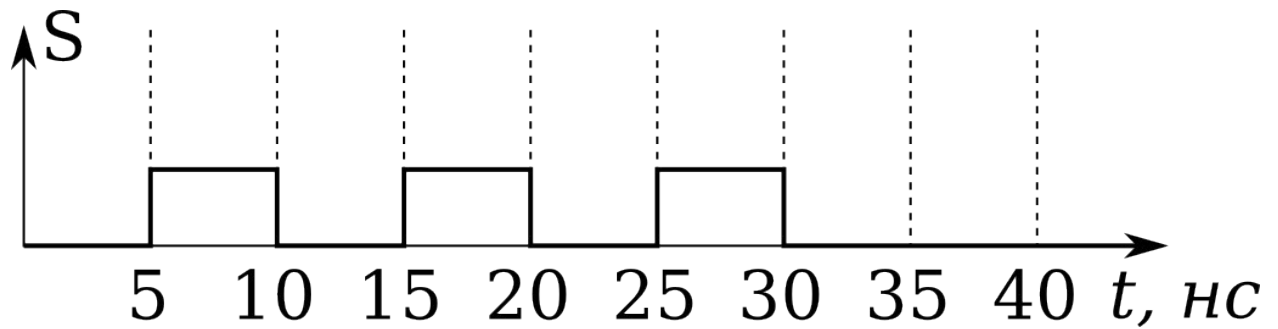
Листинг А.1 - Задание сигнала, произвольно меняющегося во времени

```
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
```

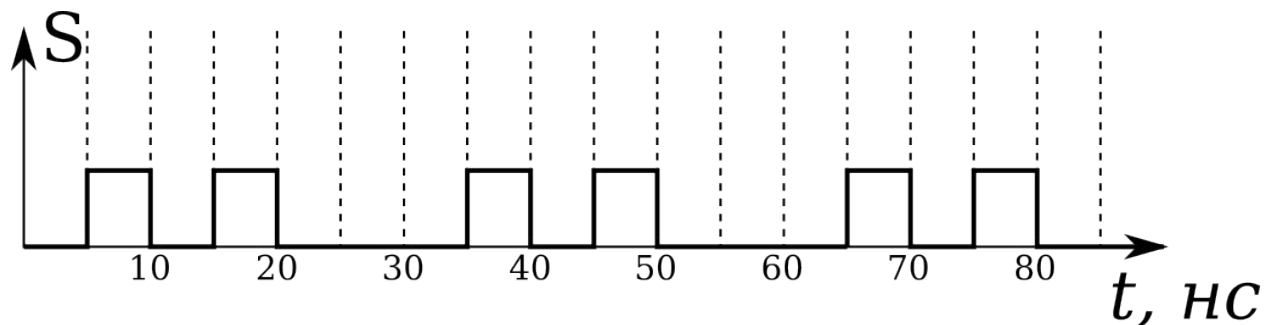


Листинг А.3 - Задание сигнала, периодически меняющегося во времени 3 раза

```

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

```