

HDL, RTL и FPGA

Цифровая схемотехника. День 1

Yuri Panchul, Senior Hardware Design Engineer, MIPS
Stanislav Zhelnio, Hardware Design Engineer, IVA Technologies

MIPS



Используемые сокращения

- HDL - Hardware Description Language
 - Язык описания аппаратуры
 - Язык на котором описывают, симулируют и верифицируют цифровые схемы. В нашем случае - Verilog-2001
- RTL - Register Transfer Level
 - Уровень регистровых передач
 - Методология описания схем на HDL, которая позволяет в полуавтоматическом режиме формировать фотошаблоны, отправляемые на полупроводниковое производство
- ASIC - Application-Specific Integrated Circuit
 - Заказная специализированная интегральная схема (заказная ИМС)
 - Примером заказной ИМС является чип в вашем телефоне, на котором работает Android
- SoC - System on Chip
 - Система-на-Кристалле (СнК)
 - Заказная ИМС, в которой присутствует одно или несколько процессорных ядер, память и иные компоненты, которые внутри одного чипа формируют законченную вычислительную систему
- FPGA - Field-Programmable Gate Array
 - ПЛИС — программируемая логическая интегральная схема
 - Конфигурируемая микросхема, которая может быть использована как альтернатива созданию заказной ИМС

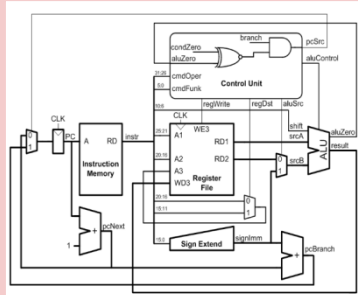
Пример двойственности аппаратной части и ПО 1/2

Микроконтроллер (embedded chip, ASIC, SoC)

Процессор

Написан на
Verilog

Выполняет
инструкции



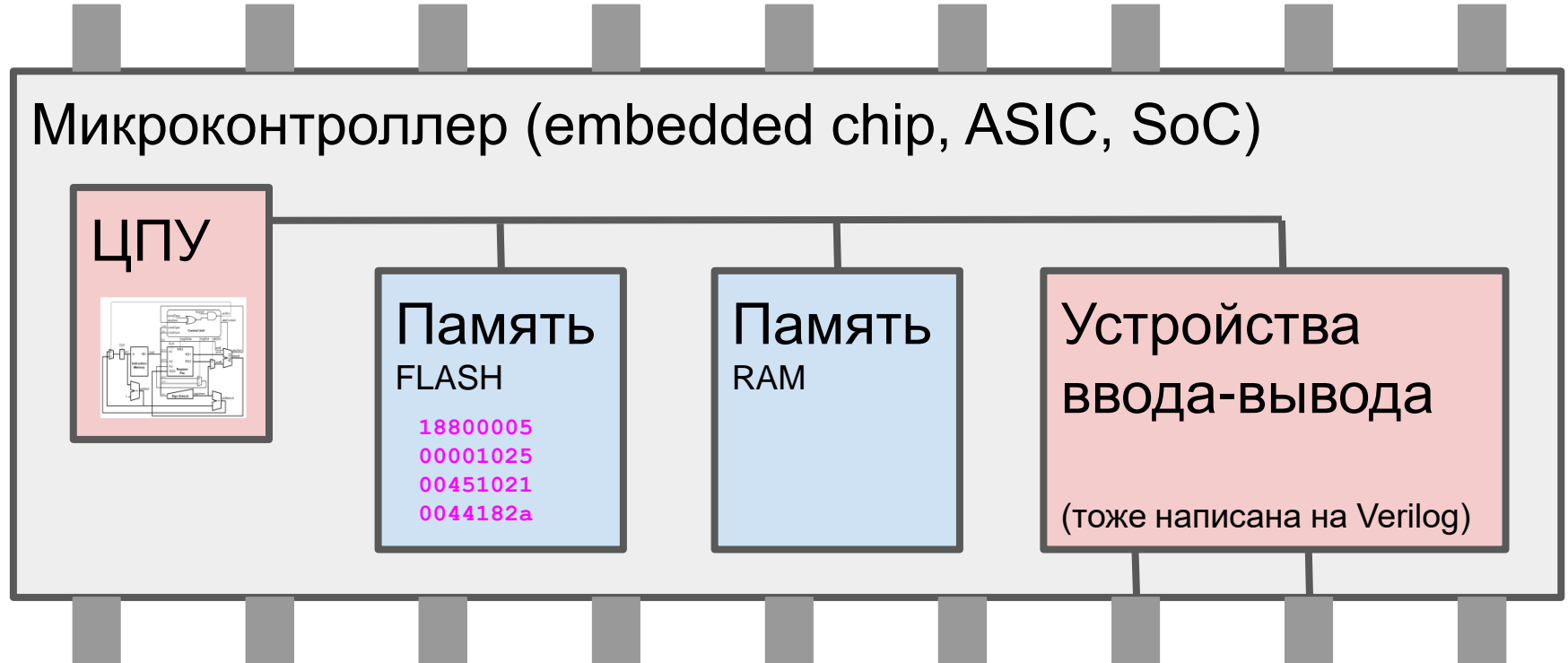
Память

Содержит программу -
последовательность инструкций

Скомпилирована из исходных
кодов на C или ином языке

18800005
00001025
00451021
0044182a
5460fffe
00451021
03e00008
00000000

Пример двойственности аппаратной части и ПО 2/2



Программа: от кода на С к машинным кодам

C:

```
int f (int a, int b)
{
    int s = 0;

    while (s < a)
        s += b;

    return s;
}
```

Ассемблер:

```
sum:
    blez    $4, exit
    move    $2, $0

    addu    $2, $2, $5

loop:
    slt     $3, $2, $4
    bnel    $3, $0, loop
    addu    $2, $2, $5

exit:
    jr      $31
    nop
```

**Машинный
код**

```
18800005
00001025

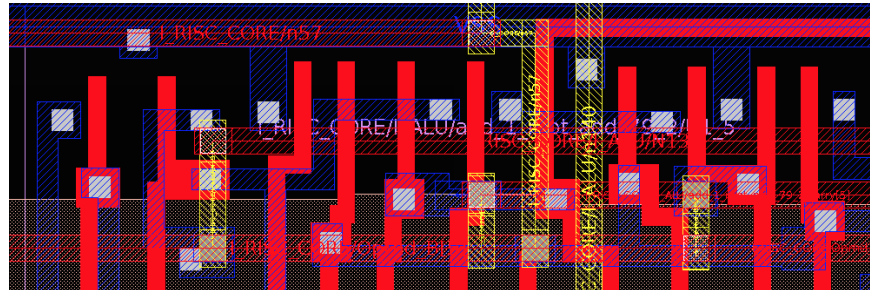
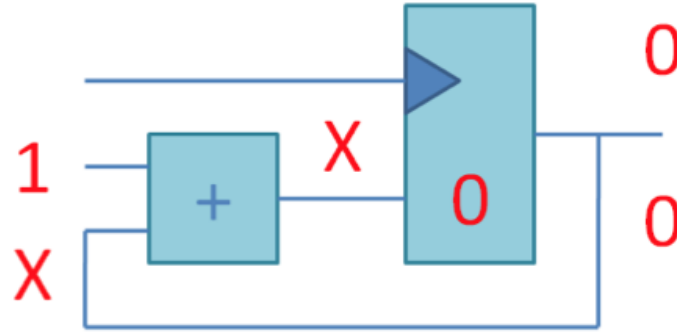
00451021

0044182a
5460fffe
00451021

03e00008
00000000
```

Схемотехника: от Verilog к транзисторам (упрощенно)

```
module counter
(
  input  clock,
  input  reset,
  output logic [1:0] n
);
  always @(posedge clock)
  begin
    if (reset)
      n <= 0;
    else
      n <= n + 1;
  end
endmodule
```

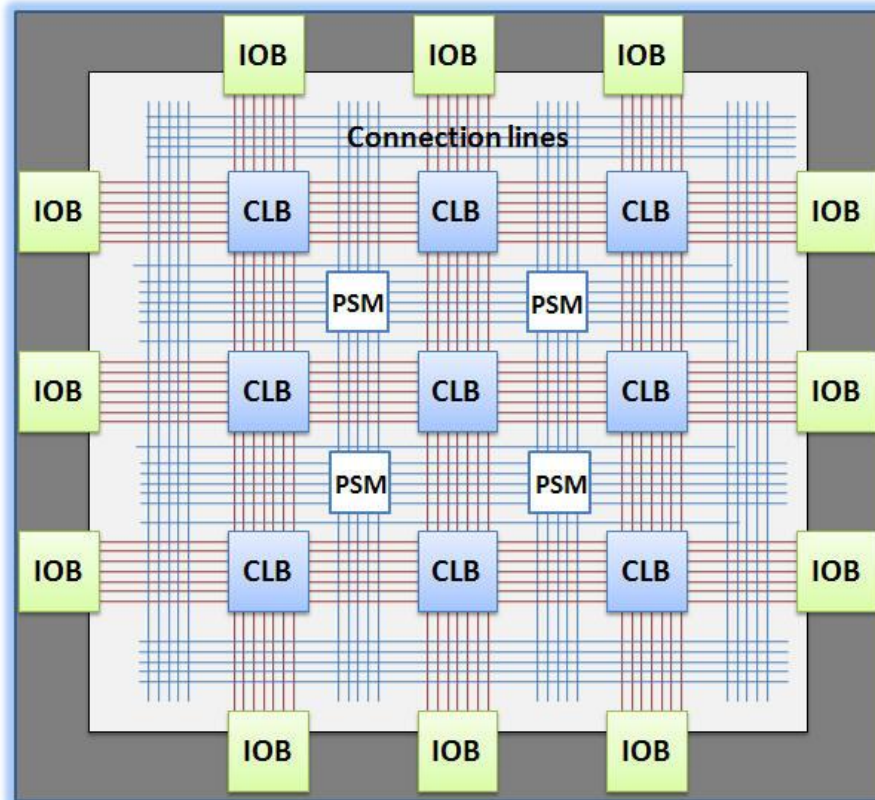


Что такое FPGA? Простыми словами

Матрица из логических ячеек

Одна ячейка может стать вентилем AND, другая OR, следующая — одним битом памяти

Внутри обычной FPGA нет процессора, но она может быть сконфигурирована так, чтобы работать как процессор



IOB
Input Output Block

CLB
Configurable
Logic Block

PSM
Programable
Switch Matrix

Connection lines
Single, Long
Double, Direct

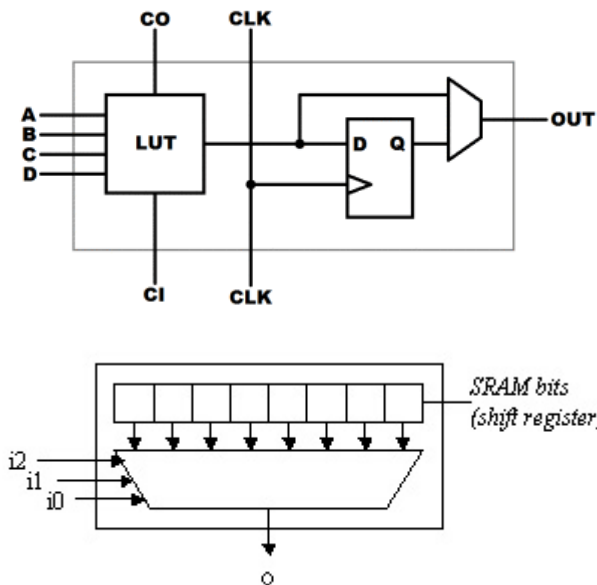
Что внутри у логической ячейки?

Внутри каждой логической ячейки есть один (или несколько) мультиплексоров (mux)

Аналогом мультиплексора в мире ПО является оператор «IF» - он позволяет делать выбор в зависимости от входных условий

Мультиплексоры подключены в битах памяти, загрузка информации в которую производится при конфигурировании («программировании») FPGA

Это позволяет выстраивать целые схемы внутри FPGA, просто меня содержимое конфигурационной памяти



Двенадцать столпов цифровой схемотехники

- Gate - вентиль
- Area - площадь
- Delay - задержка
- Parallelism - параллелизм
- Module - модуль
- Testbench - тестовое окружение
- Clock - тактовый сигнал
- Reset - сигнал сброса
- D-Flip-Flop - D-триггер
- Power - энергопотребление
- Finite State Machine - конечный автомат
- Pipeline - конвейер

Темы, относящиеся к первому дню отмечены зеленым, день 2 — синим и день 3 - желтым

В цифровой схемотехнике можно выделить

- Combinational logic

- Комбинационная логика
- Используется для вычисления арифметических и логических функций

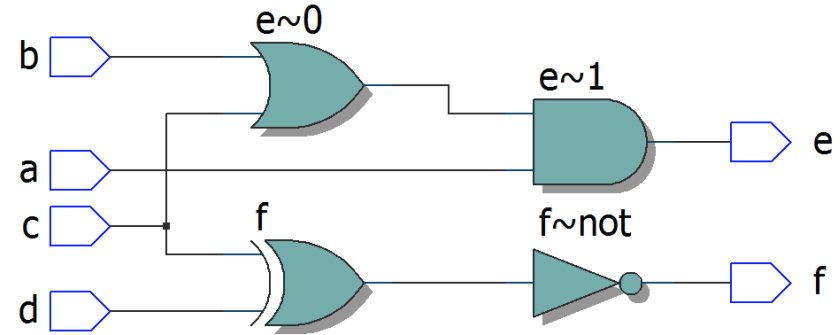
- Sequential logic

- Последовательностная логика
- Обеспечивает хранение данных и итерационные вычисления

- Первая работа посвящена комбинационной логике

Комбинационная логика

- Выходные сигналы группы логических вентилей зависят только от её входных сигналов
- Установление значений на выходах занимает некоторое время
- Такая группа вентилей называется «комбинационным облаком» (combinational cloud)
- Используется для вычисления логических и арифметических функций



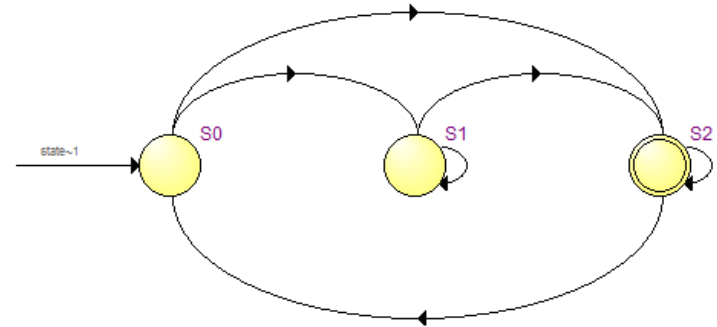
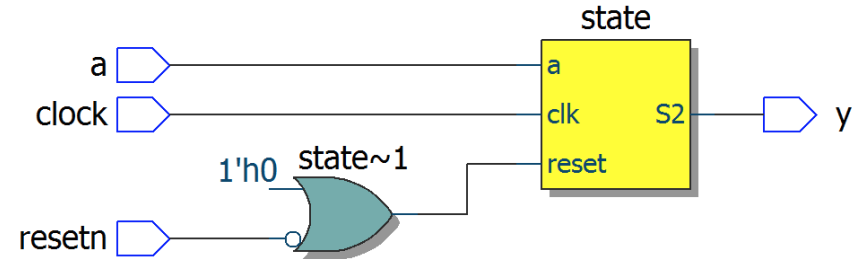
```
module top
(
    input  a, b, c, d,
    output e, f

    assign e = a & (b | c);
    assign f = ~ (c ^ d);

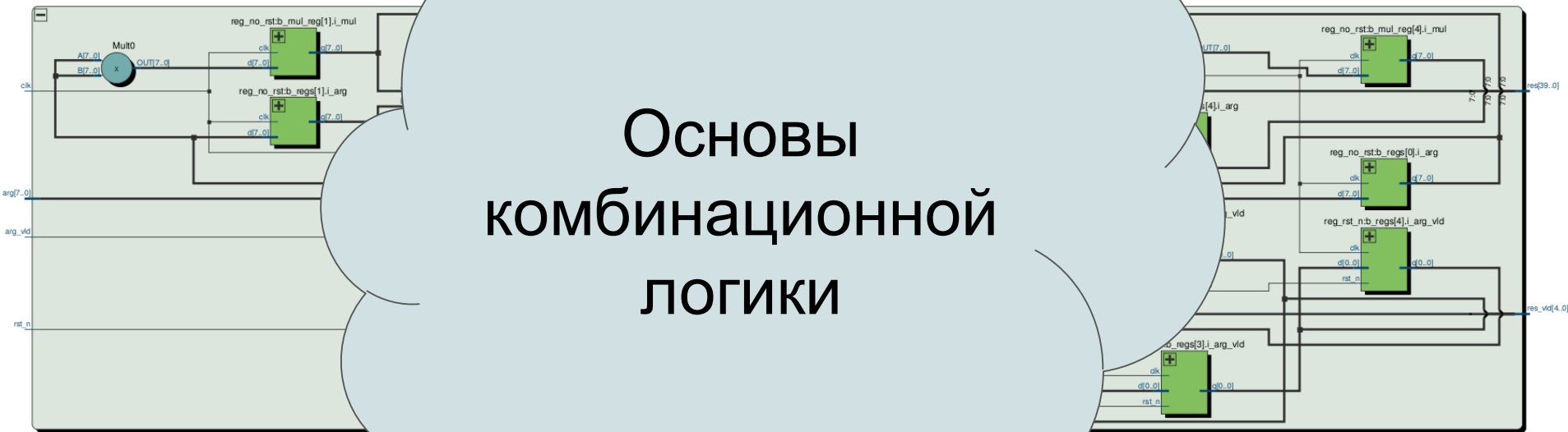
endmodule
```

Последовательностная логика

- Выходные сигналы зависят не только от входных, но также от внутреннего состояния (state), информация о котором сохранена внутри регистров (register)
- Изменение этого состояния выполняется синхронно с тактовым сигналом (clock)
- Это позволяет проектировать более сложные схемы, с запоминанием промежуточных значений и циклами

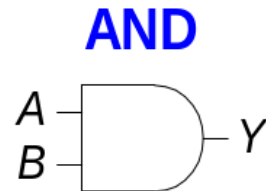


Основы комбинационной логики



Вентили — логические элементы

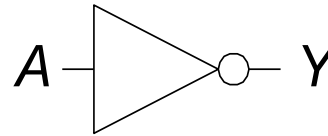
- Основные кирпичики комбинационных схем:
- **И (AND), ИЛИ (OR), НЕ (NOT), И-НЕ (NAND)** и т.д.
- Из базовых логических элементов выстраиваются более сложные схемы:
- **мультиплексоры, дешифраторы, сумматоры** и т.д.
- Каждому логическому элементу соответствует свой оператор языка Verilog или их сочетание:
- **& , | , ~ , ~\$** и т.д.



Базовые логические элементы (1 / 3)

- Для каждого вентиля можно составить таблицу истинности
- Таблица истинности показывает зависимость сигнала на выходе (**Y**) от сигналов на входе (**A**, **B**, etc)
- Самые простые вентили: **НЕ (NOT)** и **Буфер (BUF)**

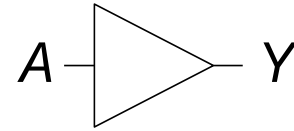
NOT



$$Y = \overline{A}$$

| A | Y |
|---|---|
| 0 | 1 |
| 1 | 0 |

BUF



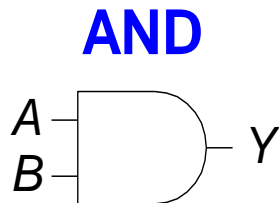
$$Y = A$$

| A | Y |
|---|---|
| 0 | 0 |
| 1 | 1 |

Базовые логические элементы(2 / 3)

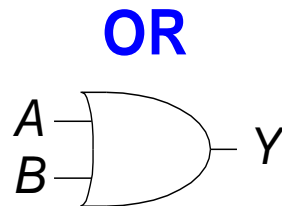
Вентили с 2 входами:

- **И (AND)**
- **ИЛИ (OR)**
- **Исключающее ИЛИ (XOR)**



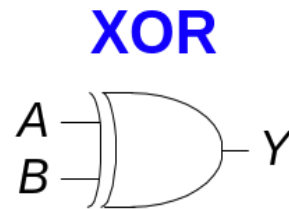
$$Y = AB$$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



$$Y = A + B$$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |



$$Y = A \oplus B$$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Базовые логические элементы(3 / 3)

Вентили с инвертированным
выходом:

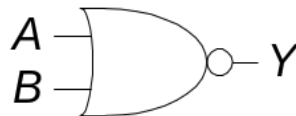
NAND



$$Y = \overline{AB}$$

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

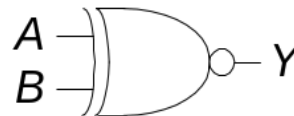
NOR



$$Y = \overline{A + B}$$

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

XNOR



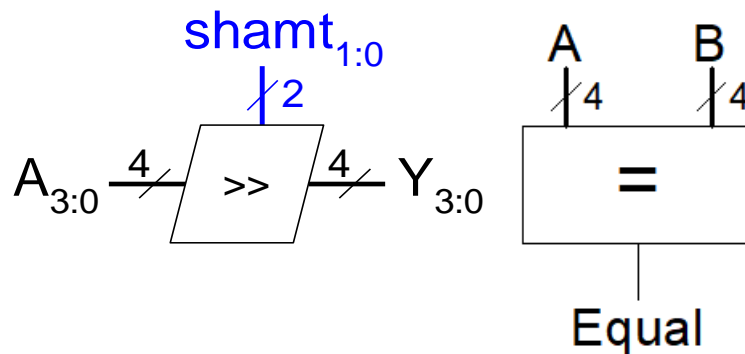
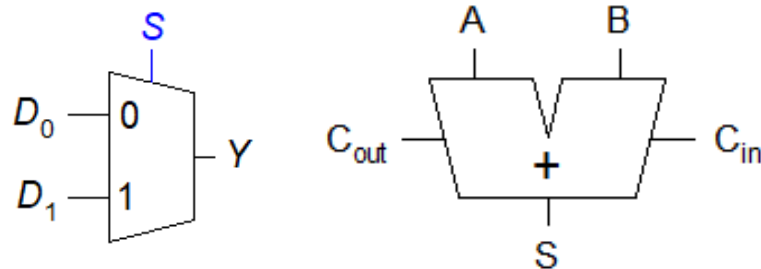
$$Y = \overline{A \oplus B}$$

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

- **И-НЕ (AND)**
- **ИЛИ-НЕ (OR)**
- **Исключающее ИЛИ-НЕ (XOR)**

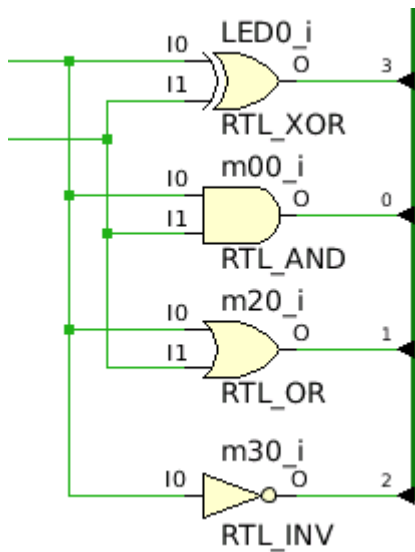
Более сложные комбинационные схемы

- Строятся на базе базовых элементах
- Более подробно описаны в книге "Цифровая схемотехника и архитектура компьютера", Дэвид М. Хэррис и Сара Л. Хэррис, 2013



Вентили на языке Verilog (1 / 2)

- описываются с помощью операторов: **&**, **|**, **~** и т.д.



```
// Basic gates AND, OR and NOT
```

```
assign LED [0] = a & b;
```

```
assign LED [1] = a | b;
```

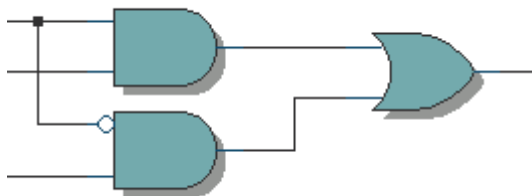
```
assign LED [2] = ~ a;
```

```
// XOR gates (useful for adders, comparisons,  
// parity and control sum calculation)
```


```
assign LED [3] = a ^ b;
```

Вентили на языке Verilog (1 / 2) ... и их приоритет

- Очень похоже на обычный язык программирования
- Но результатом является схема!



| | |
|--------------|-------------------------------------|
| ~ | Отрицание |
| *, /, % | Умножение, деление, остаток |
| +, - | Сложение, вычитание |
| <<, >> | Сдвиг |
| <<<, >>> | Арифметический сдвиг |
| <, <=, >, >= | Сравнение (больше-меньше) |
| ==, != | Сравнение на равенство |
| &, ~& | И, И-НЕ |
| ^, ~^ | Исключающее ИЛИ, исключающее ИЛИ-НЕ |
| , ~ | ИЛИ, ИЛИ-НЕ |
| ?: | Тернарный оператор |



Упражнение 1: Симуляция комбинационной схемы

Lab 1. Упражнения

1. Запуск комбинационной схемы в симуляторе

- a. Ключевые слова: ports, continuous assignments, testbench, delay, waveforms
- b. Используется ModelSim PE Student Edition от Mentor Graphics

2. Синтез комбинационной схемы

- a. Ключевые слова: Logic synthesis, constrains, synthesized schematics, FPGA configuration
- b. Используется Quartus Lite Edition и отладочная плата Terasic DE10-Lite

Подготовительные действия

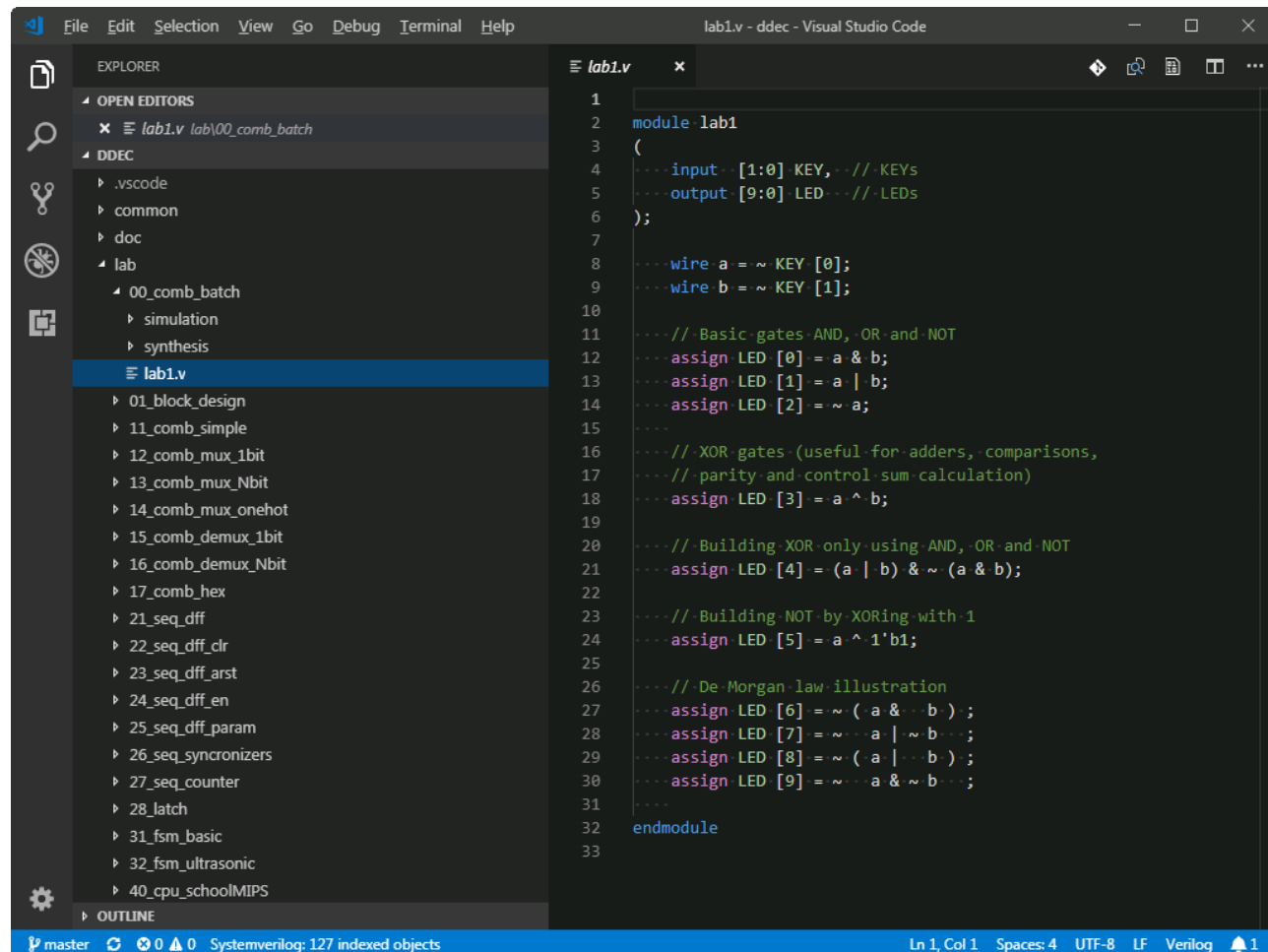
1. Клонировать репозиторий GitHub

<https://github.com/zheInio/ddec>

2. Запустите редактор Visual Studio Code
3. Откройте в нем загруженный (клонированный) каталог
4. Откройте файл:

[ddec/lab/00_comb_batch/lab1.v](#)

Lab1.v



```
1
2 module lab1
3 (
4     ....input  [1:0] KEY, ....KEYS
5     ....output [9:0] LED....LEDS
6 );
7
8     ....wire a = ~ KEY [0];
9     ....wire b = ~ KEY [1];
10
11     ....//Basic gates AND, OR and NOT
12     ....assign LED [0] = a & b;
13     ....assign LED [1] = a | b;
14     ....assign LED [2] = ~ a;
15     ....
16     ....//XOR gates (useful for adders, comparisons,
17     ....//parity and control sum calculation)
18     ....assign LED [3] = a ^ b;
19
20     ....//Building XOR only using AND, OR and NOT
21     ....assign LED [4] = (a | b) & ~ (a & b);
22
23     ....//Building NOT by XORing with 1
24     ....assign LED [5] = a ^ 1'b1;
25
26     ....//De Morgan law illustration
27     ....assign LED [6] = ~ ( a & b ) ;
28     ....assign LED [7] = ~ ~ a | ~ b ;
29     ....assign LED [8] = ~ ( a | b ) ;
30     ....assign LED [9] = ~ ~ a & ~ b ;
31     ....
32 endmodule
33
```


Комбинационная схема - 1

- lab_01/src/lab1_hdl/lab1.v
- Блоком проекта является “module”
- У модуля есть
 - Имя
 - список портов (ports)
- У каждого порта есть направление:
 - input, output или inout
- У каждого порта есть ширина / размера вектора
 - “[9:0]” означает “10 бит от LED [9] до LED [0]”
 - Если ширина не обозначена, предполагается, что она равна 1 бит, например “input ABC”

```
module lab1
(
    input  [1:0] KEY,
    output [9:0] LED
);
```

Комбинационная схема - 2

```
wire a = ~ KEY [0];
```

```
wire b = ~ KEY [1];
```

- lab_01/src/lab1_hdl/lab1.v
- Сигналы («переменные») объявленные как “wire” позволяют создавать промежуточные соединения с другими портами, другими сигналами “wire” или сигналами, объявленными как “reg”, которые будут обсуждаться далее
- “wire” может быть шириной 1 или более бит, например: “wire [4:0] A”
- Присваивания значений сигналам типа “wire” является **непрерывным**
 - Изменение значение выражения справа от знака присваивание вызывает изменение значение сигнала слева от знака
- В результате оптимизации, выполняемой средствами синтеза, “wire” могут быть исключены из итоговой схемы

Комбинационная схема - 3

- lab_01/src/lab1_hdl/lab1.v
- Ключевое слово “assign” позволяет применить непрерывное присваивание к уже объявленному порту или сигналу “wire”
- Внимание: порядок объявления непрерывных присваиваний не имеет значения, они «срабатывают» в момент изменения значения в правой части выражения
- При симуляции HDL правые части выражений с помощью планировщика, каждая задача которого вычисляет значение одного из выражений из правой части оператора присваивания
- Результатом синтеза HDL является электрическая схема (граф, netlist)

```
// Basic gates AND, OR and NOT
```

```
assign LED [0] = a & b;
```

```
assign LED [1] = a | b;
```

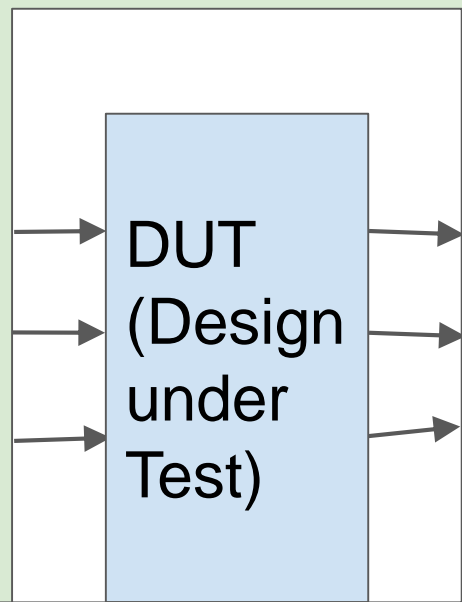
```
assign LED [2] = ~ a;
```

Тестовое окружение - 1

```
module testbench;
```

- lab_01/src/lab1_hdl/simulation/testbench.v
- Модуль testbench (TB) предназначен только для симуляции, его синтез никогда не выполняется
- Внутри TB содержит специальную программу с помощью которой выполняется тестирование модуля
- Тестируемый модуль, экземпляр которого создаётся внутри TB часто называют DUT (Design under Test)

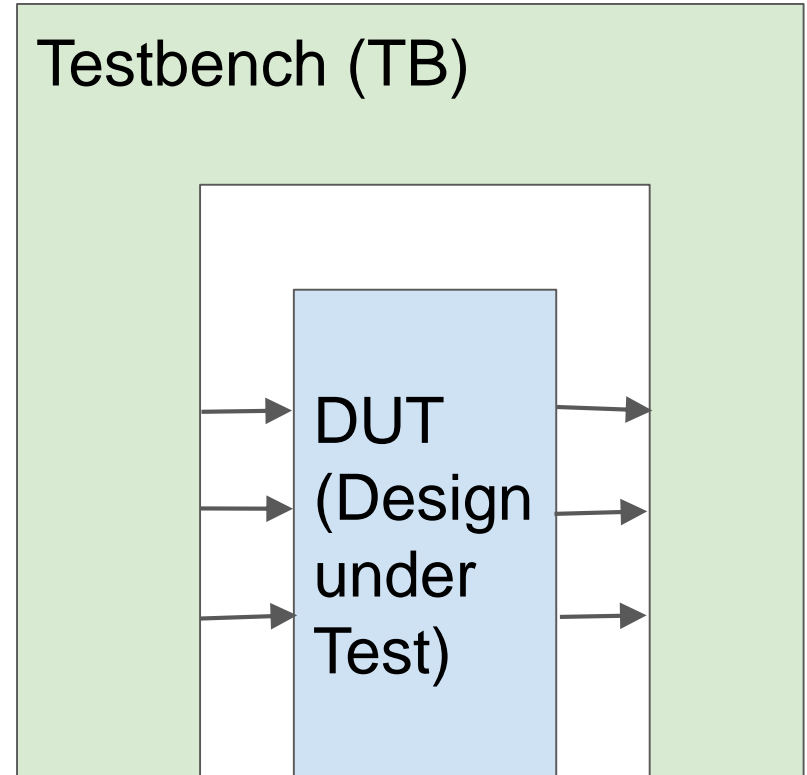
Testbench (TB)



Тестовое окружение - 2

```
module testbench;
```

- lab_01/src/lab1_hdl/simulation/testbench.v
- Testbench
 - Создаёт экземпляр DUT
 - Формирует сигналы на его входах
 - Проверяет сигналы на выходах
- Из Testbench можно вызывать функции, реализованные на других языках программирования (например, C), что позволяет формировать более сложное тестовое окружение



Тестовое окружение - 3

- При создании модуля его порты подключаются к переменным, объявленным внутри testbench
- Экземпляр модуля похож на вызов функции, но он им не является:
 - Экземпляр DUT модуля физически помещается внутри модуля testbench
 - Соединение переменных внутри ТВ с портами DUT является непрерывным, как если бы использовалось ключевое слово “assign”.

- lab_01/src/lab1_hdl/simulation/t
estbench.v

```
// input and output test signals
```

```
reg  [1:0] key;
```

```
wire [9:0] led;
```

```
// creating the instance of the
```

```
// lab1 - module name
```

```
// dut - instance name ('dut'
```

```
lab1 dut ( key, led );
```

Тестовое окружение - 4

- lab_01/src/lab1_hdl/simulation/testbench.v
- В данном случае входные сигналы тестируемого модуля формируются простейшим способом:
 - Переменной, которая подключена к порту модуля DUT присваивается значение
 - Конструкция #delay сообщает симулятору о необходимости вставить задержку продолжительностью в указанное количество временных промежутков (time units)

```
initial
```

```
begin
```

```
key = 2'b00;
```

```
#10;
```

```
key = 2'b01;
```

```
#10;
```

```
key = 2'b10;
```

```
#10;
```

```
key = 2'b11;
```

```
#10;
```

```
end
```

Тестовое окружение - 5

- lab_01/src/lab1_hdl/simulation/testbench.v
- Ключевое слово “initial” используется для того, чтобы указать симулятору, что следующее выражение необходимо выполнить в начале симуляции
- “begin/end” используются для того, чтобы сгруппировать последовательно выполняющиеся выражения в один блок
- Присваивание, выполняемые внутри блока “begin/end” с помощью оператора “=” называется **блокирующим**. Такие присваивания обрабатываются последовательно, **не** параллельно (“блокируют выполнение”)

initial

begin

key = 2'b00;

#10;

key = 2'b01;

#10;

key = 2'b10;

#10;

key = 2'b11;

#10;

end

Тестовое окружение - 6

- lab_01/src/lab1_hdl/simulation/testbench.v
- Выражение `2'b10` является константой:
“два бита шириной, двоичное, значение 10”.
- Размер временного промежутка (time unit) в выражении `#10`, а также точность симуляции по шкале времени задаются директивой ``timescale`. Этот временной промежуток встретится нам на временных диаграммах (waveforms).
- Если указать следующую директиву в начале файла testbench, то `#10` будет означать “10 наносекунд”:

`initial`

`begin`

`key = 2'b00;`

`#10;`

`key = 2'b01;`

`#10;`

`key = 2'b10;`

`#10;`

`key = 2'b11;`

`#10;`

`end`

``timescale 1 ns / 100 ps`

Тестовое окружение - 7

- lab_01/src/lab1_hdl/simulation/testbench.v

```
reg [1:0] key;
```

- Сигналу, объявленному как **reg** может быть присвоено значение внутри блока “begin/end”
- Выражение \$monitor внутри блока initial используется для того, чтобы симулятор выводил в лог сообщений обо всех изменениях переменных (сигналов), указанных в качестве параметров этого выражения

```
initial
```

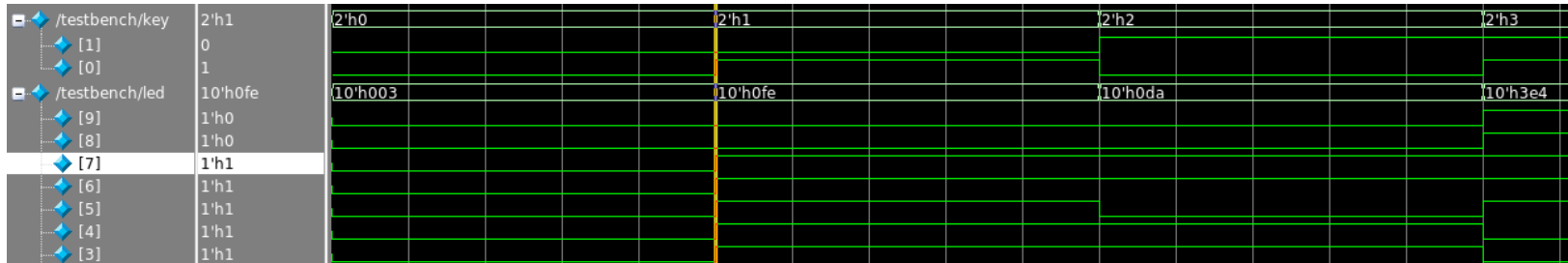
```
$monitor("key=%b led=%b", key, led);
```

Тестовое окружение - 8

`initial`

`$dumpvars;`

- `lab_01/src/lab1_hdl/simulation/testbench.v`
- `$dumpvar` служит используется для того, чтобы симулятор во время своей работы формировал временную диаграмму (waveform)
- Waveform — это база данных с информацией о том, как изменялись сигналы во время сеанса отладки
- Для просмотра временных диаграмм используется специальная программа (waveform viewer), которая может быть как отдельным приложением (GTKWave), так и быть интегрированной в симулятор Verilog



Скрипт для запуска симуляции

- lab_01/src/lab1_hdl/simulation/01_simulate_with_modelsim.bat
- Создает отдельный каталог для временных файлов, формируемых симулятором ModelSim во время работы
- Вызывает программу симулятор командой *vsim*.
- *vsim* считывает команды, которые необходимы для выполнения симуляции, из скрипта, написанного на языке TCL (Tool Command Language).
- Для Linux можно использовать аналогичный shell-скрипт

```
rem recreate a temp folder for  
rd /s /q sim  
md sim  
cd sim  
  
rem start the simulation  
vsim -do ../modelsim_script.tcl
```

TCL скрипт для ModelSim

- lab_01/src/lab1_hdl/simulation/modelsim_script.tcl
- TCL (произносится как "тикль" или "ти-си-эль") — это скриптовый язык, используемый с 1990х во многих пакетах разработки (EDA tools, EDA - Electronic Design Automation)
- Не обязательно знать его в совершенстве, часто достаточно скопировать чей-то скрипт к себе в проект

```
# create modelsim working library
vlib work
```

```
# compile all the Verilog sources
vlog ../testbench.v ../../lab1.v
```

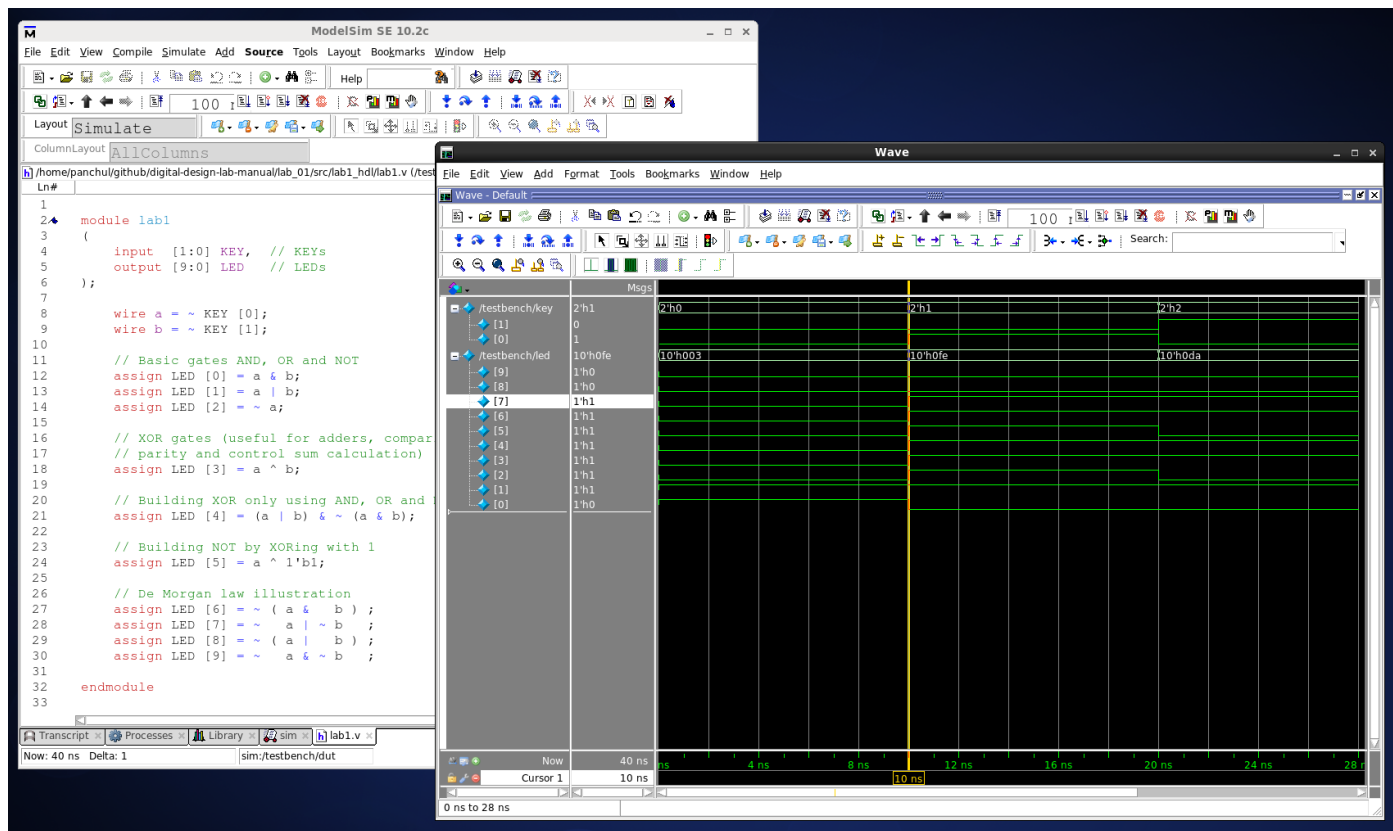
```
# open the testbench module for simulation
vsim -novopt work.testbench
```

```
# add all testbench signals to time diagram
add wave sim:/testbench/*
```

```
# run the simulation
run -all
```

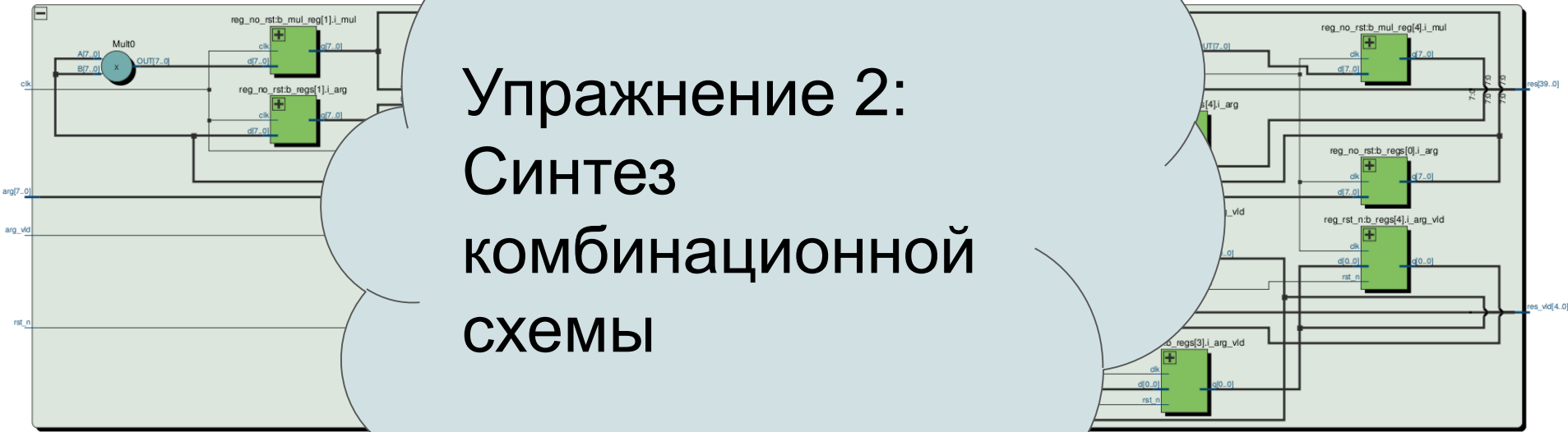
```
# expand the signals time diagram
wave zoom full
```

Теперь можно запустить симуляцию



...ИЗМЕНИТЬ ЧТО-ТО В КОДЕ И ЗАПУСТИТЬ ЕЁ ПОВТОРНО

Упражнение 2: Синтез комбинационной схемы



Основной файл проекта Quartus

- lab_01/src/lab1_hdl/synthesis/lab1.qpf
- Может быть пустым
- Quartus записывает в него информацию о времени запуска и версии приложения (используется службой поддержки Intel / Altera)
- Все настройки синтеза сохраняются не в .qpf, а в других файлах: .qsf и .sdc.

```
QUARTUS_VERSION = "17.0"  
DATE = "12:11:55  November 06, 2017"  
PROJECT_REVISION = "lab1"
```


TCL скрипт настроек синтеза - 1

- lab_01/src/lab1_hdl/synthesis/lab1.qsf
- Для проекта достаточно 4 директив (приведены ниже), а также настроек распиновки (следующий слайд)
- Нет необходимости знать все используемые внутри файла .qsf
- Часто достаточно скопировать чужой скрипт к себе в проект и использовать минимально необходимый набор

```
set_global_assignment -name DEVICE 10M50DAF484C7G
set_global_assignment -name TOP_LEVEL_ENTITY lab1
set_global_assignment -name VERILOG_FILE ../../lab1.v
set_global_assignment -name PROJECT_OUTPUT_DIRECTORY .
```

TCL скрипт настроек синтеза - 2

- lab_01/src/lab1_hdl/synthesis/lab1.qsf
- Распиновка — это информация о том, какой сигнал подключен к какому выводу FPGA.
- Софт синтеза уже «знает» о том, где расположен вывод (ножка, пин) **PIN_B8** у чипа **10M50DAF484C7G**, соответствующая настройка «сообщает» ему о том, что с этим выводом нужно соотнести порт KEY[0] — на который приходит сигнал с кнопки

```
set_location_assignment PIN_B8 -to KEY[0]
set_location_assignment PIN_A7 -to KEY[1]
set_location_assignment PIN_A8 -to LED[0]
set_location_assignment PIN_A9 -to LED[1]
. . . . .
```

TCL скрипт настроек синтеза - 3

- lab_01/src/lab1_hdl/synthesis/lab1.qsf
- Также в файле **.qsf** содержится информация о типе выходного каскада (I/O standard) для каждой из используемых ножек чипа
- В данном курсе от вас не требуется знать перечень данных типов и их характеристика
- При установке настроек IO_STANDARD порты можно задавать с помощью шаблона "*"

```
set_instance_assignment -name IO_STANDARD \  
    "3.3 V Schmitt Trigger" -to KEY*
```

```
set_instance_assignment -name IO_STANDARD "3.3-V LVTTL" -to LED*  
. . . . .
```

Перед запуском синтеза

- lab_01/src/lab1_hdl/synthesis/make_project.bat
- Intel / Altera Quartus Lite Edition создаёт множество временных файлов
- Удобнее всего запускать синтез в отдельном каталоге
- Перед этим в данный каталог необходимо скопировать файлы проекта
- Для этого используется скрипт
make_project.bat [для Windows] или make_project.sh [для Linux]

```
#!/bin/sh
```

```
rd /s /q project
```

```
mkdir project
```

```
copy *.qpf project
```

```
copy *.qsf project
```

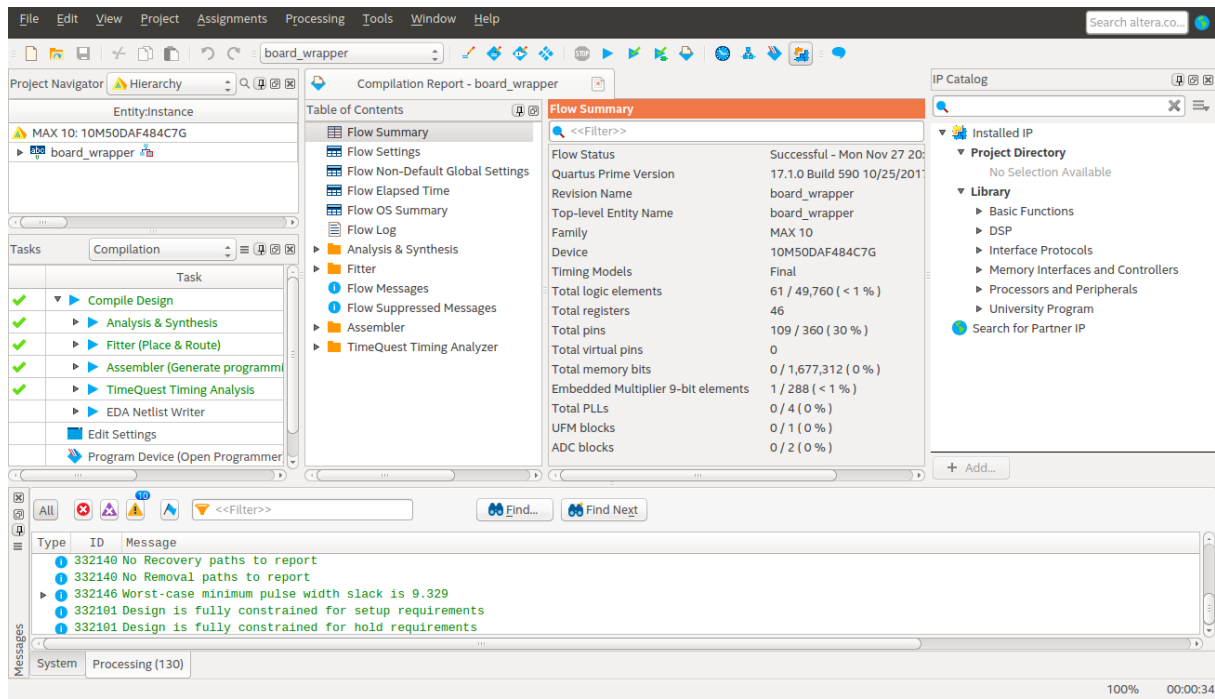
```
rm -rf project
```

```
mkdir project
```

```
cp *.q[ps]f project
```

- File:Open Project
- Не перепутайте с File:Open
- Найти файл проекта во временном каталоге
- Не перепутайте с оригинальным файлом проекта
- Двойной клик по “Compile Design” в окне Task

Запуск синтеза в Intel Quartus

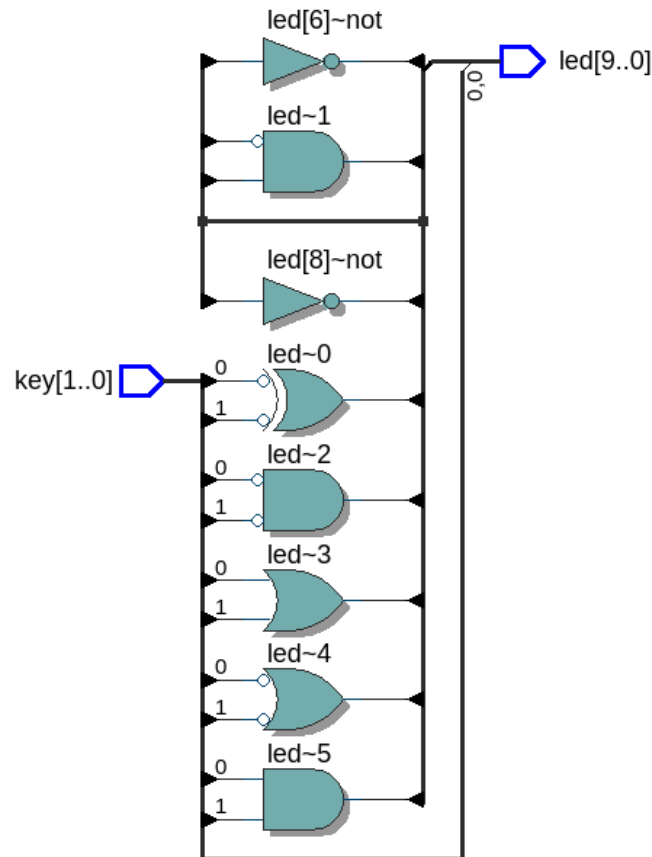


Netlist Navigator

top:1 +

▶ top

- У Quartus есть специальный режим просмотра, в котором по результатам анализа RTL-кода отображается соответствующая ему схема (**RTL-Viewer**). Эта схема строится до того, как RTL-код будет отображён на логические ячейки FPGA

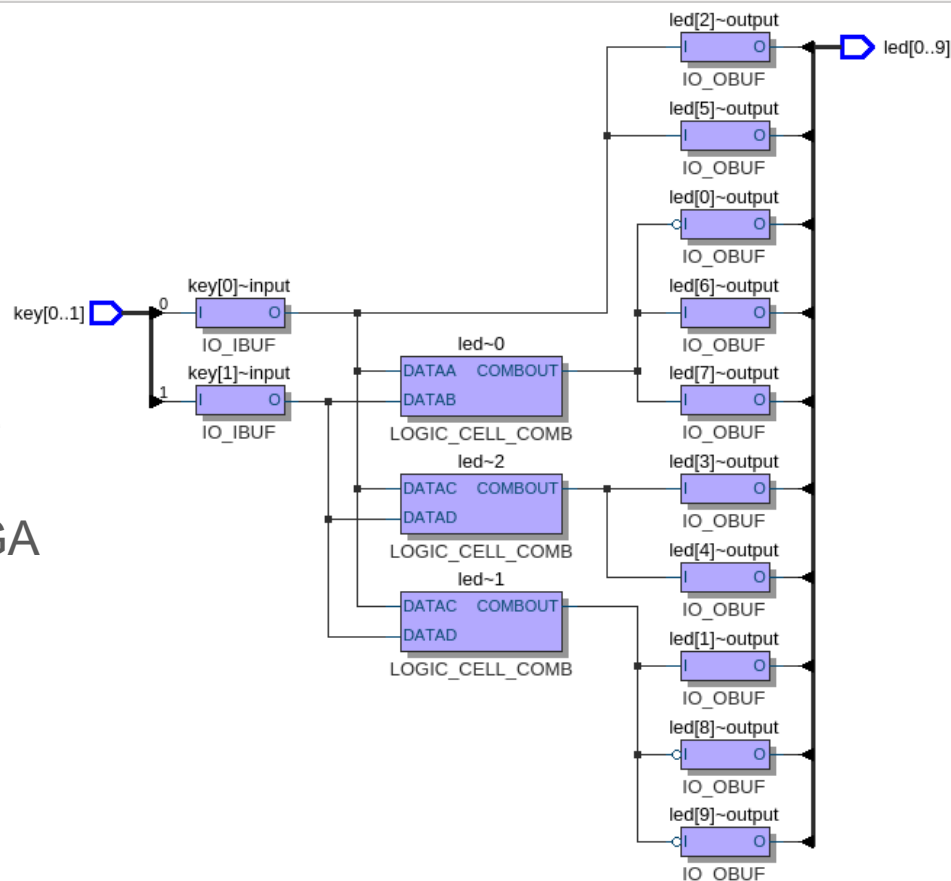


Netlist Navigator

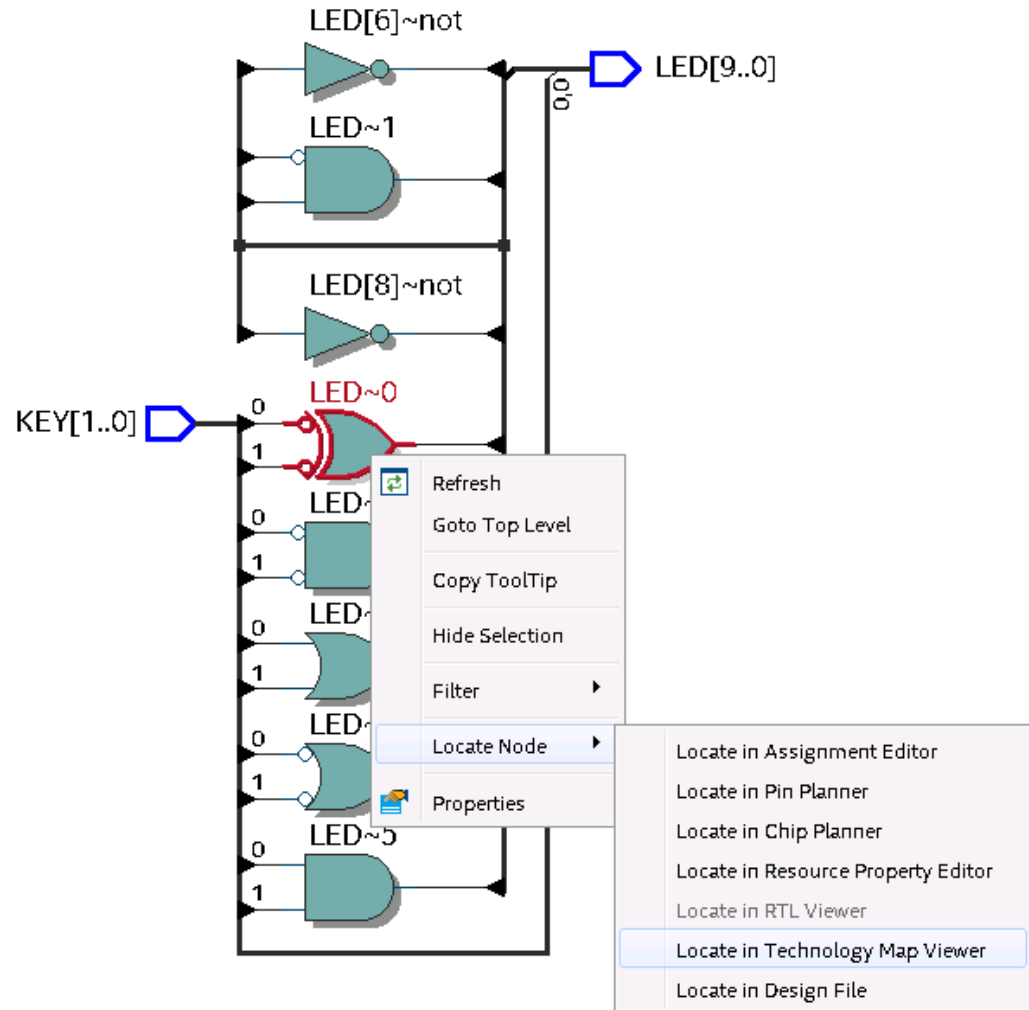
top:1 +

top

- Также доступен режим просмотра, в котором отображается схема, на которой каждому выражению в RTL-коде соответствует своя логическая ячейка FPGA - **Technology Map Viewer**. Сравните это изображение с предыдущим слайдом

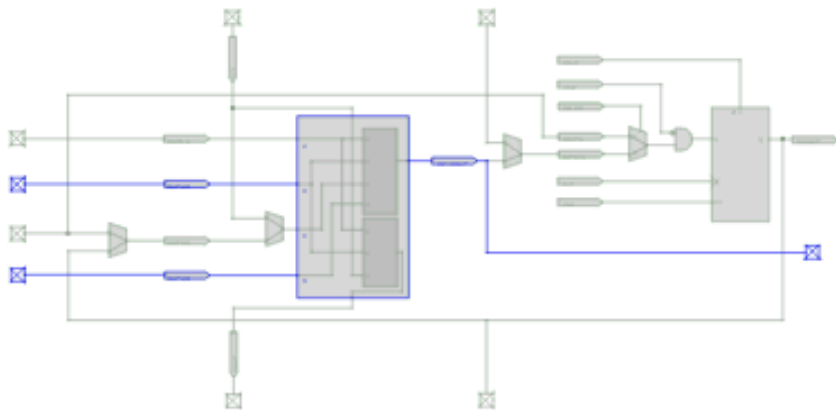


- Для каждого логического элемента можно посмотреть, каким образом он отображается на логическую ячейку



Для логической ячейки доступны:

- таблица истинности LUT и схема, которая в ней реализована (**Properties**)
- Схема логической ячейки с активными сигналами (**Resource Property Editor**)



Block: |lab1|LED~2

Lutmask: {33CC}

DATAB COMBOUT
DATAD

LED~2

| | DATAD | DATAB | OUT |
|---|-------|-------|-----|
| 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 1 |
| 3 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 |

n Fan-out Ports Truth Table

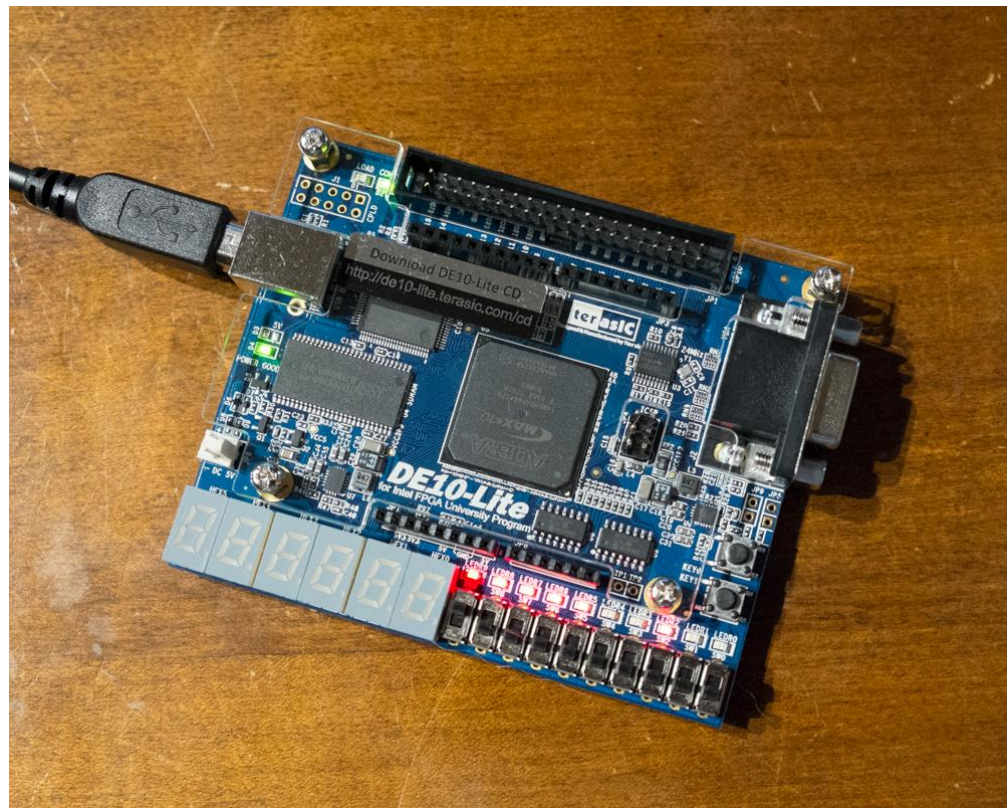
LED~0
DATAB COMBOUT
DATAD
LOGIC_CELL_COMB

LED~2
DATAB COMBOUT
DATAD
LOGIC_CELL_COMB

LED~1
DATAB COMBOUT
DATAD
LOGIC_CELL_COMB

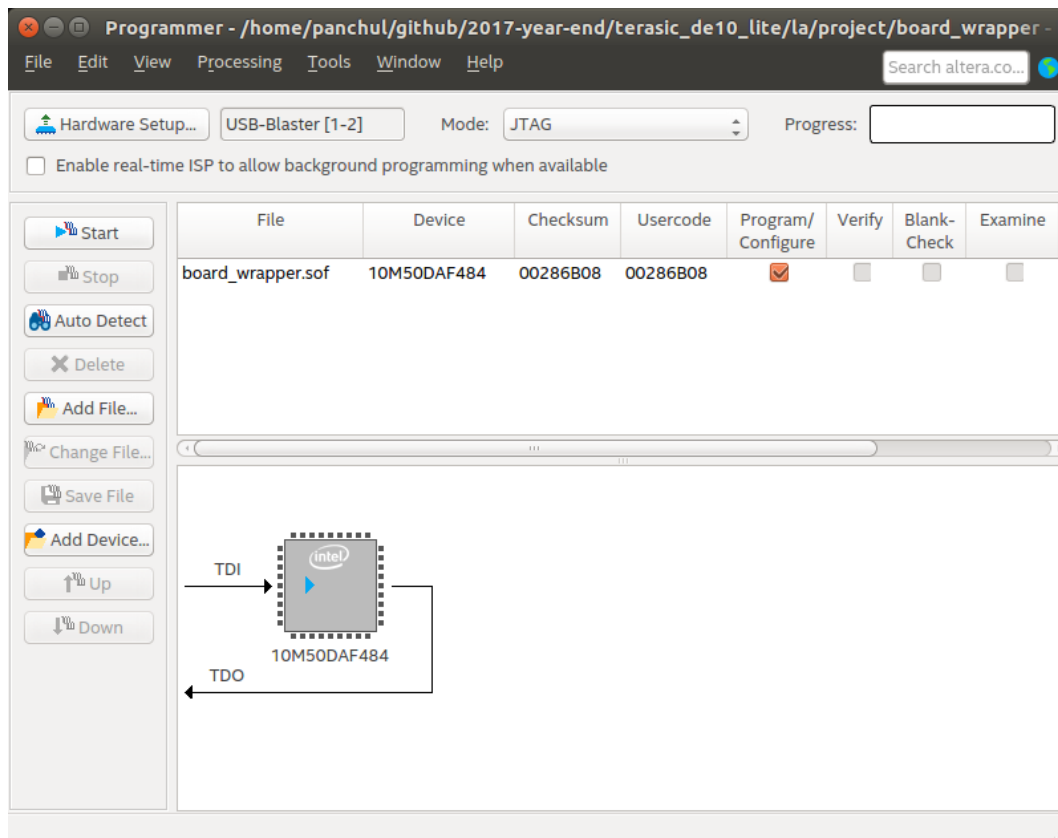
Отладочная плата Terasic DE10-Lite

- Приемлемая цена: \$85
- "Академическая" цена \$55
- Intel FPGA MAX 10
10M50DAF484C7G
- 50K логических ячеек,
достаточно для процессора
уровня MIPSfpga
- 64MB SDRAM
- Для программирования
подключается напрямую к ПК



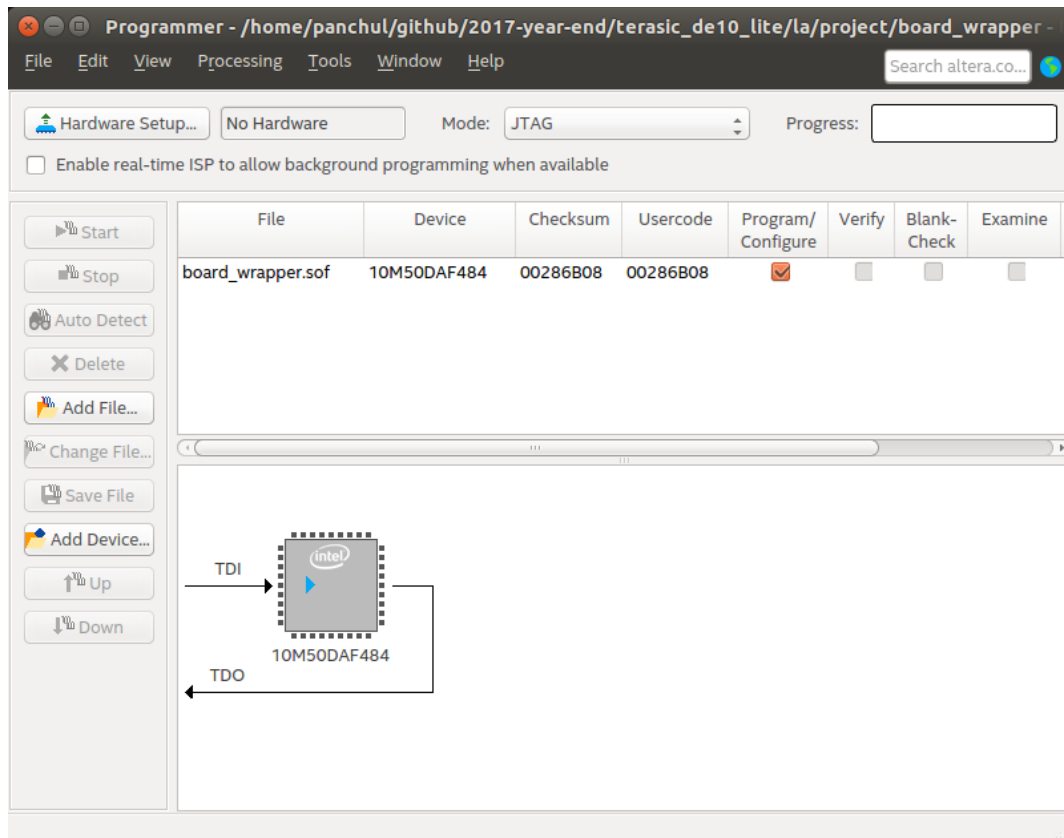
Программирование отладочной платы - 1

- Запустите “Program Device” из окна Tasks
- Должно появиться окно Programmer
- Если рядом с кнопкой “Hardware Setup” отображается “USB-Blaster” плата подключена и идентифицирована программой
- В этом случае просто нажмите на кнопку “Start”



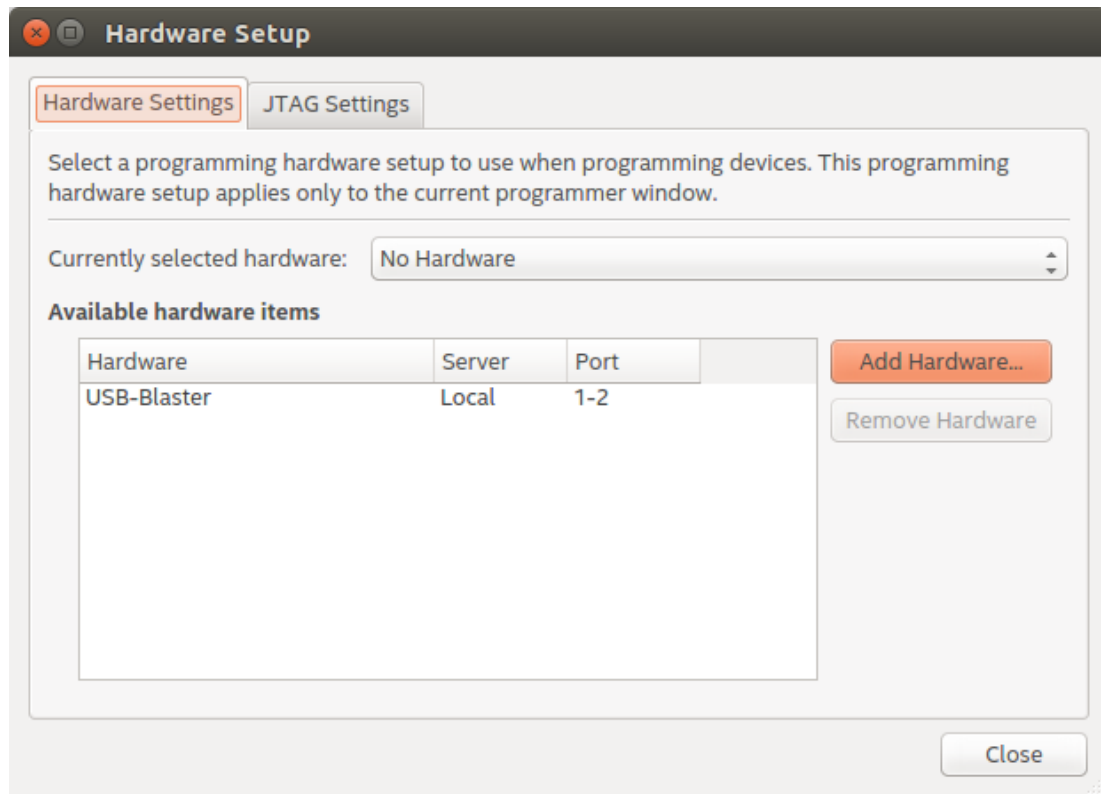
Программирование отладочной платы - 2

- Если отображается “No hardware”, нажмите “Hardware Setup”



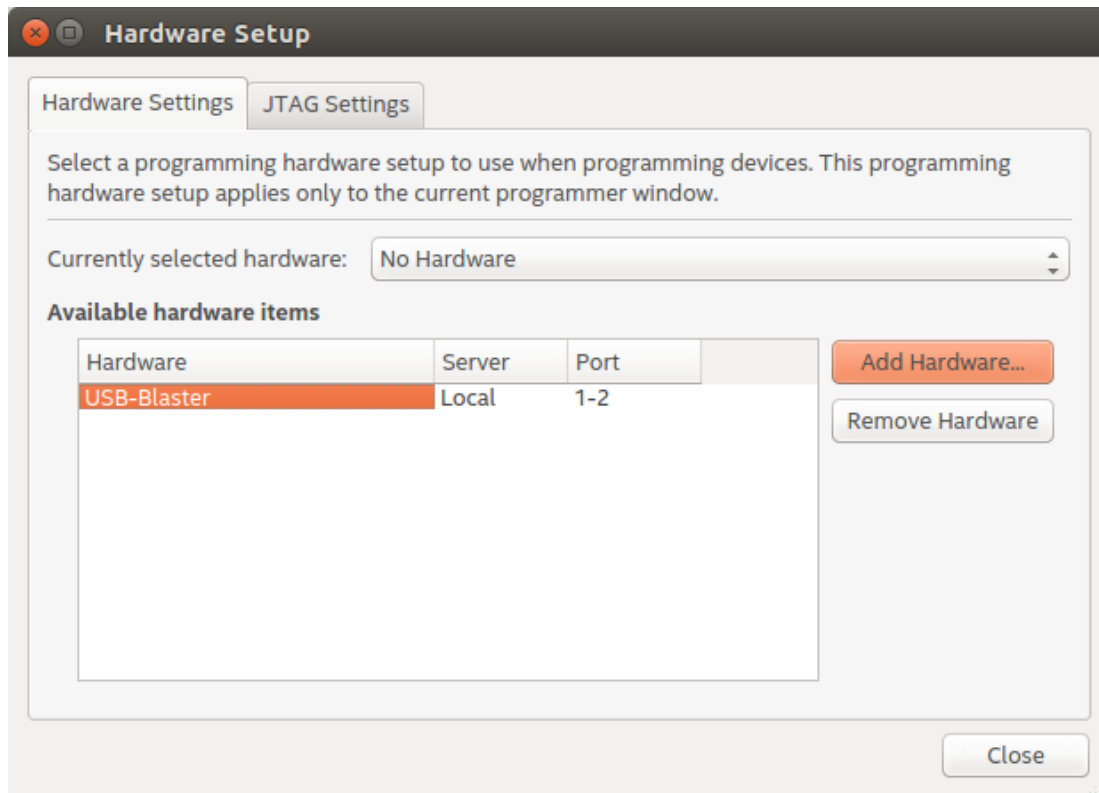
Программирование отладочной платы - 3

- В окне “Hardware Setup” должен отображаться “USB-Blaster”
- Если список пуст, проверьте подключение кабеля
- Если плата подключена, но не отображается, проверьте, установлен ли драйвер USB-Blaster



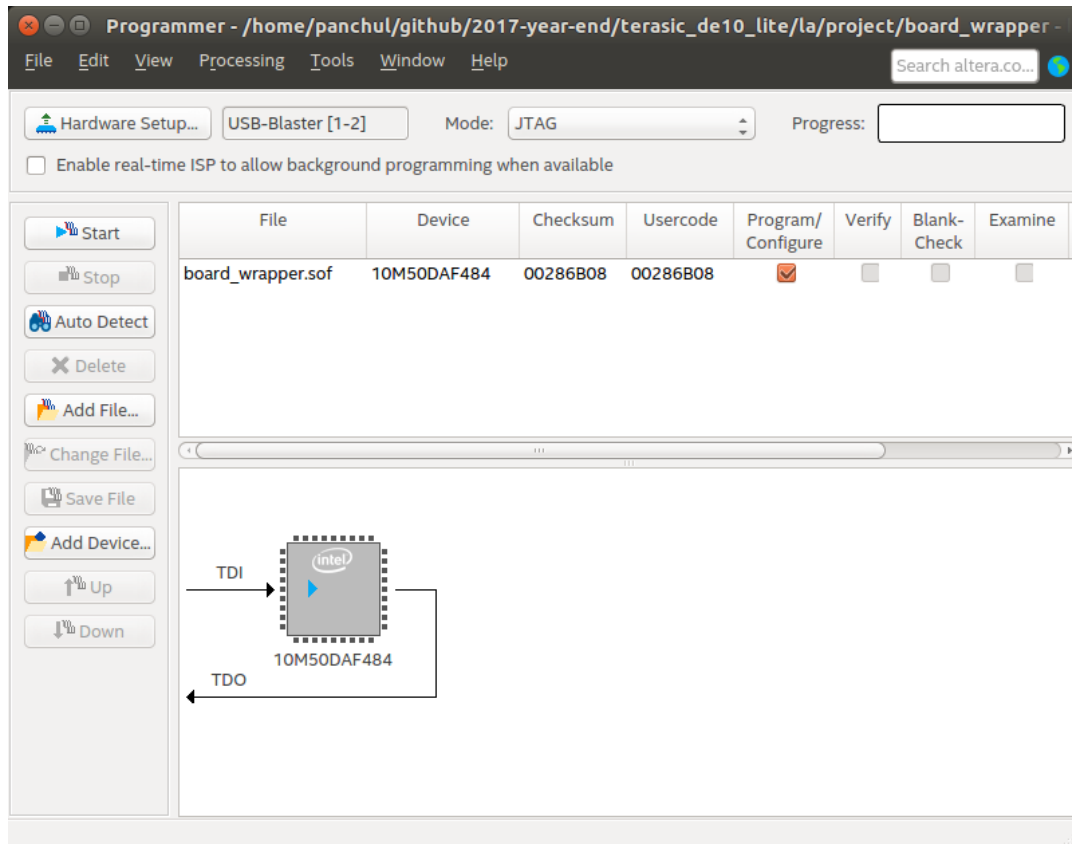
Программирование отладочной платы - 4

- Двойной клик по “USB-Blaster”, после чего кнопка “Close”



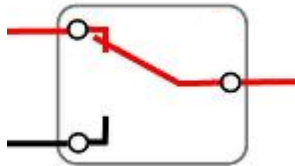
Программирование отладочной платы - 5

- После этих действий проблема должна быть устранена
- Just press “Start” button and see the “Progress” indicator becoming green.
- If “Progress” gets stuck, unplug USB cable, plug it in again and repeat.



Добавьте новый элемент комбинационной схемы - MUX

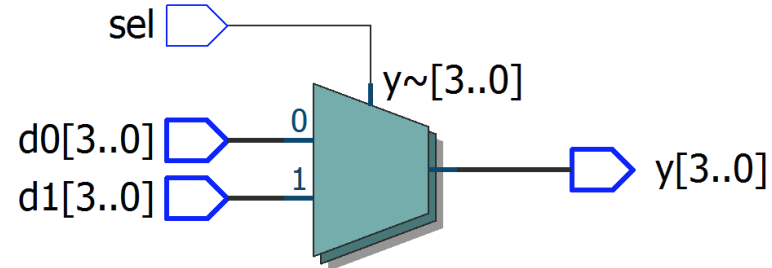
- Мультиплексор (multiplexer, mux) позволяет выбрать одно значение (сигнал) из нескольких - как переключатель
- Самый простор способ добавить мультиплексор — это использовать тернарный оператор `?:`
- Внесите изменение в проект и перезапустите синтез конфигурации FPGA



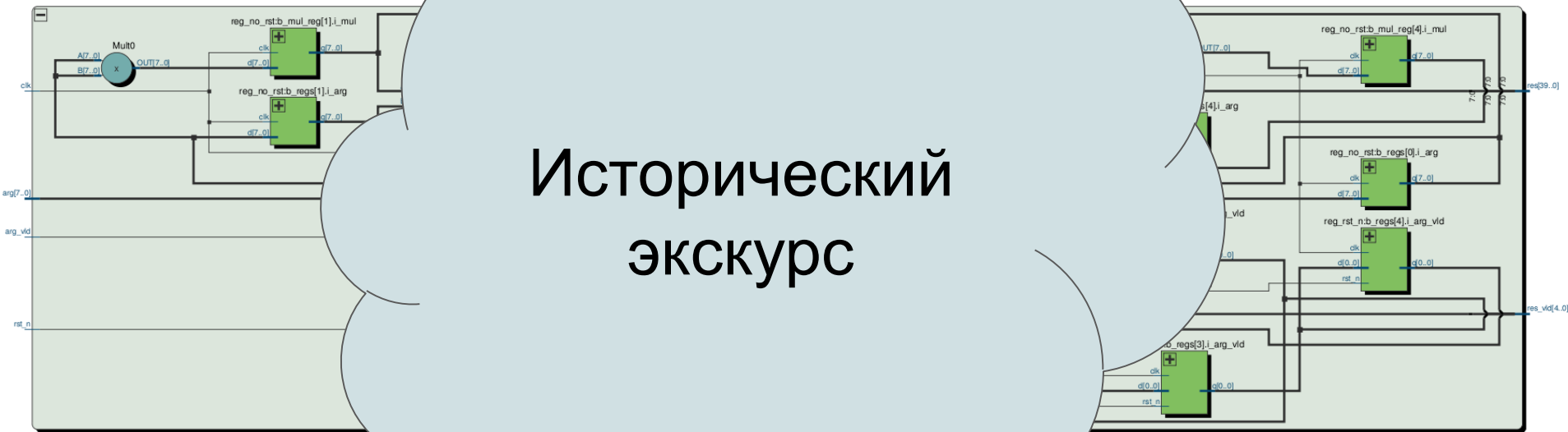
```
module mux_2_1
(
    input  [1:0] d0,
    input  [1:0] d1,
    input          sel,
    output [1:0] y
);

    assign y = sel ? d1 : d0;

endmodule
```

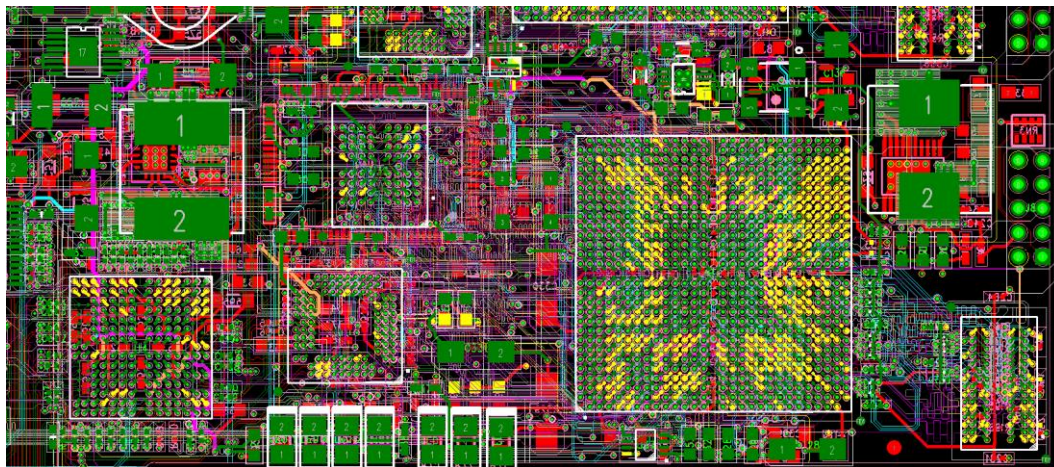


Исторический экскурс



Графическое проектирование

- До того, как языки Verilog и VHDL получили распространение, в разработке микросхем использовалось графическое проектирование
- Оно и сейчас используется для разработки принципиальных схем электронного оборудования - на основе принципиальной схемы разрабатывается проект печатной платы (PCB)

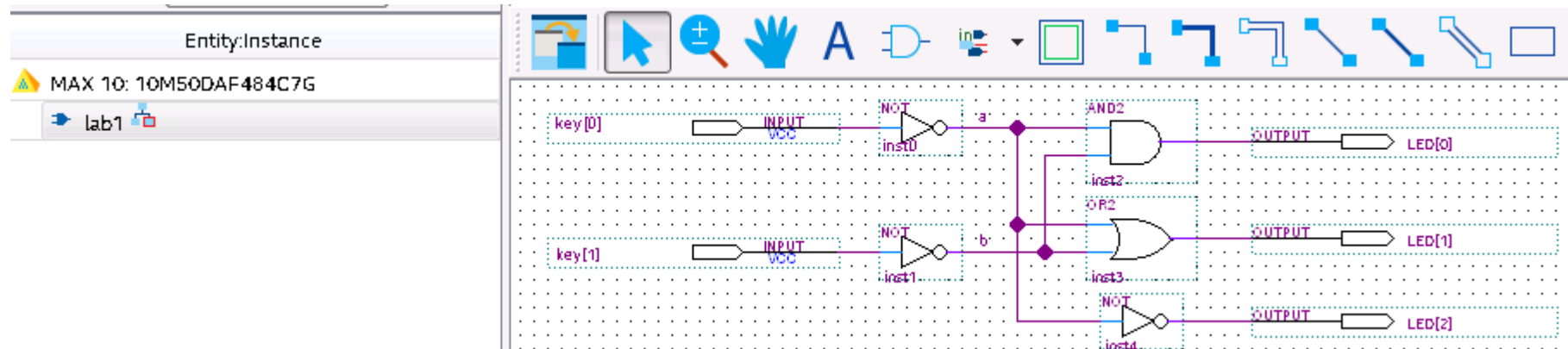


Графическое проектирование на практике (1 / 2)

- Используя Quartus Lite откройте проект:

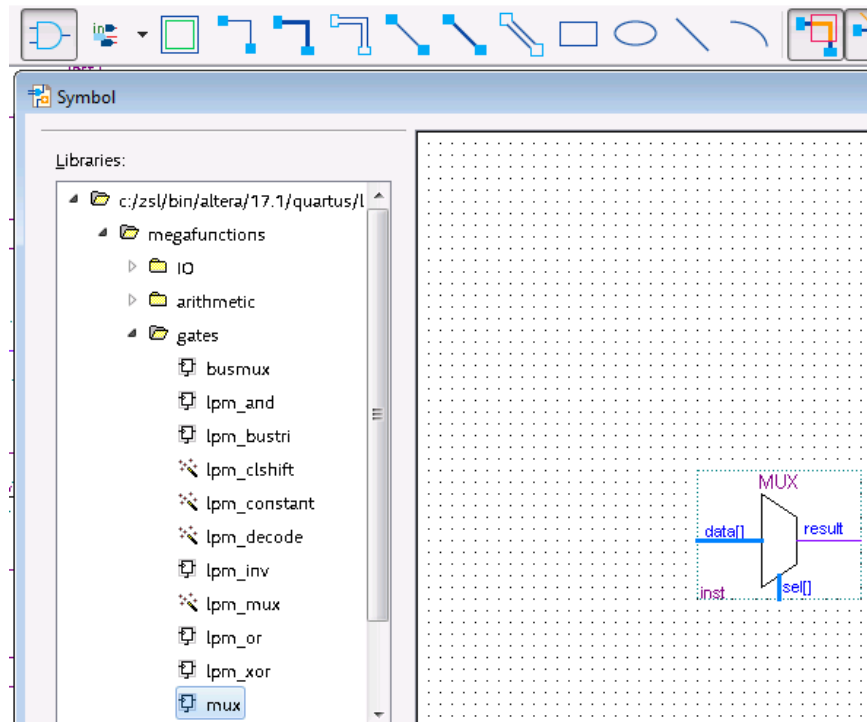
ddec/lab/01_block_design/lab1.qpf

- Выполните синтез данного проекта и запрограммируйте плату DE10-Lite



Графическое проектирование на практике (1 / 2)

- Добавьте на схему мультиплексор, чтобы ее работа была такой же, как и в предыдущем случае, когда вы вносили изменения в Verilog
- Внесите изменение в проект и перезапустите синтез конфигурации FPGA
- Удобно !? Вы бы смогли поддерживать так по-настоящему большой проект с тысячами вентилей?



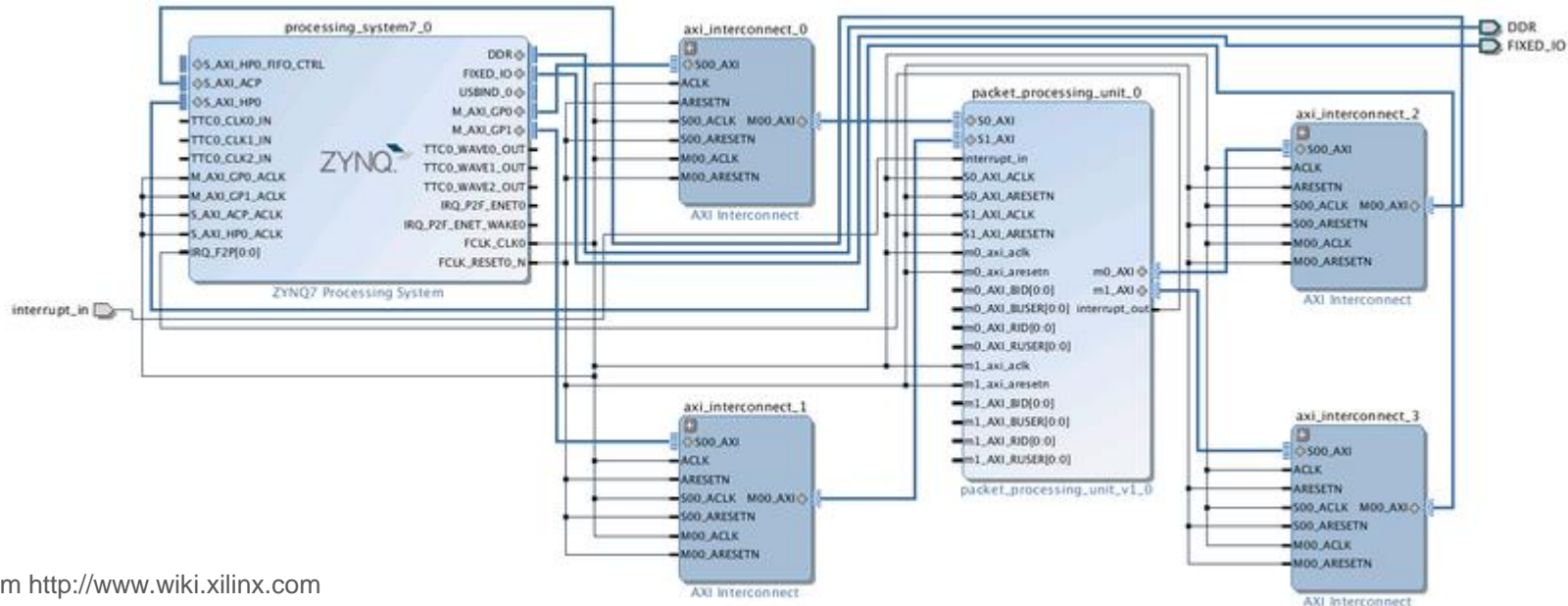
Графическое проектирование - итоги

- Минусы:

- непереносимость форматов схем между ПО разных производителей

- Плюсы:

- удобно для интеграции сложных модулей на верхнем уровне проекта
(Xilinx Vivado Block Design)



Makefile

и автоматизация
нудной работы



Зачем это нужно или non-project mode

- Средства симуляции и синтеза создают в своих каталогах множество временных файлов, которые мешают работе с системами контроля версий (например **git**)
- Постепенно разработчики выработали следующую стратегию работы: проекты, в которых работа с системами контроля версий необходима, ведутся в т.н. non-project mode:
 - когда исходные коды хранятся в отдельном каталоге, под наблюдением системы контроля версий
 - синтез и симуляция выполняются в отдельных временно создаваемых каталогах
 - при (почти) каждом запуске симуляции или синтеза необходимо создавать временные каталоги, файлы проектов и т.д.
 - Эти действия можно выполнять с помощью **bash** или **bat** скриптов, но они являются условно переносимыми между **Windows** и **Linux**
 - **make** – программа, изначально созданная для автоматизации сборки проекта, все необходимые настройки которой хранятся в т.н. **Makefile**

Основы работы с Makefile (1 / 7)

- Перейдите в каталог [ddec/lab/02_Makefile](#)
- Откройте файл **Makefile**
- В этом файле описаны процедуры – цели (**target**), которые программа `make` выполняет во время своей работы
- Каждая цель объявлена в начале строки (без пробелов), после ее имени стоит двоеточие
- В приведенном примере **first** и **second** – это цели
- В строчках, следующих за именем цели, после символа табуляции (важно!) указаны команды, которые будут выполнены для данной цели

- Откройте терминал
- Выполните команды:

make first
make second

```
first:
    echo "This is a first target"
```

```
second:
    echo "This is a second target"
```


Основы работы с Makefile (2 / 7)

- Цели могут зависеть друг от друга
- В этом случае запуск целей выполняется с учетом зависимостей
- Для make критерием того, что цель уже выполнена является наличие в текущем каталоге файла, имя которого совпадает с именем цели
- Если такого файла нет, команды, прописанные для цели будут выполняться всегда
- Для переноса строки можно использовать символ \
- Откройте терминал
- Выполните команды:
make third
make fourth

```
first:
    echo "This is a first target"

second:
    echo "This is a second target"

third: first second
    echo "This is a third target"
    echo "It depends on first and second"

fourth: \
    first \
    second
```

Основы работы с Makefile (3 / 7)

- Для того, чтобы узнать, какие команды будут запущены **make** без запуска этих команд можно использовать параметр **-n**
- Выполните команды:
make -n third
make -n fourth
- Для вывода сообщений из Makefile можно использовать функцию **\$(info ...)**
- Выполните команду:
make fifth

```
fifth:
    $(info This is a way to print some message from Makefile)
    @true
```
- Если содержимое различных Makefile совпадает, то общий код можно вынести в отдельный файл и включить в другие файлы с помощью директивы **include**. Пример такого кода:
ddec/lab/13_comb_mux_Nbit/Makefile

Основы работы с Makefile (4 / 7)

- Опции (аргументы), с которыми запускается команда можно вынести в отдельные переменные
- Доступ к таким переменным осуществляется как **\$(имя_переменной)**
- К одной переменной с помощью операции **+=** можно добавить несколько значений
- Выполните команды:
make sixth
make seventh

```
SIXTH_OPT = Just_some_program_option
```

```
sixth:
```

```
    echo $(SIXTH_OPT)
```

```
SEVENTH_OPT = An options can be
```

```
SEVENTH_OPT += concatenated
```

```
SEVENTH_ARG = end mixed
```

```
seventh:
```

```
    echo $(SEVENTH_OPT) $(SEVENTH_ARG)
```

Основы работы с Makefile (5 / 7)

- Операция `?=` позволяет установить значение переменной в случае, если оно не было установлено ранее
`EIGHTH_OPT ?= default_arg`
- Выполните команды (windows):
`make eighth`
`set EIGHTH_OPT=new_value`
`make eighth`
`eighth:`
`echo $(EIGHTH_OPT)`

`eighth_and_half: EIGHTH_OPT = some_other_arg`
`eighth_and_half: eighth`
- Выполните команды (linux bash):
`make eighth`
`set EIGHTH_OPT=new_value make eighth`
- Заменой параметров можно пользоваться, описывая зависимости:
`make eighth_and_half`

Основы работы с Makefile (6 / 7)

- При описании целей их можно представить в виде списка
- Узнать в этом случае какая именно цель была вызвана можно с помощью переменного `$@`
- Выполните команды:
`make de10_lite`
`make de0_cv`

```
NINTH_OPT = de10_lite de0_cv
```

```
$(NINTH_OPT):  
    echo $@
```

Основы работы с Makefile (7 / 7)

- Функция **`$(wildcard ...)`** позволяет получить список каталогов/файлов имя которых соответствует заданному шаблону. Если каталог не указан в составе шаблона, то поиск ведется в текущем каталоге
- Директивы **`ifeq`** и **`ifneq`** можно использовать в качестве условных операторов. В данном примере с помощью **`ifneq`** выполняется сравнение пустой строки и результатов работы функции **`$(wildcard ...)`** - если файлы **`test_*.v`** были найдены, то будет выполнен код внутри **`ifneq .. endif`**
- Выполните команду:
`make tenth`
- Опция **`make -C`** позволяет выполнить **`make`** в другом каталоге. Пример такого использования: **`ddec/lab/Makefile`**

```
TENTH_OPT = $(wildcard test_*.v)
tenth:
    ifneq (,$(wildcard test_*.v))
        echo "Verilog files found:"
    endif
    echo $(TENTH_OPT)
```

Знакомство с non-project mode Makefile

- Откройте файл **ddec/lab/11_comb_simple/Makefile**
- Откройте файл **ddec/common/lab/Makefile**
- В терминале перейдите в каталог **ddec/lab/11_comb_simple**
- Выполните команду **make**
- Выполните команды
 - make -n sim**
 - make -n sim_gui**
 - make -n sim_cmd**
 - make -n synth**и несколько других на ваш выбор с параметром **-n**
- Если в какой-то момент вы осознаете, что не понимаете, что «делает» **Makefile**, то наиболее простой путь выяснить это – запустить **make** с параметром **-n**, после чего уточнить в справке на конкретную вызываемую утилиту что делает та или иная опция

Основные опции утилит (1 / 2)

- **vsim** – запуск симуляции в **Modelsim**:
 - **-do имя_файла** – путь к tcl скрипту с внутренними командами **Modelsim**
 - **-c** не запускать GUI, симуляция в консольном режиме
 - **-onfinish stop** – при обработке **\$finish** выполнить остановку, запросить выход из программы у пользователя, в консольном режиме симуляция завершается без запроса
- **iverilog** - запуск компилятора **Icarus Verilog**
 - **-s имя_модуля** – top модуль симуляции
 - **-g2005-sv** – использовать опции стандарта SystemVerilog 2005
 - **-I** – путь к каталогу, в котором будет производится поиск файлов при обработке макроса include
- **vvp** – запуск симулятора Icarus Verilog
- **gtkwave** – запуск программы для просмотра временных диаграмм Icarus Verilog

Основные опции утилит (2 / 2)

- **vivado** – запуск **Xilinx Vivado**:
 - **-source имя_файла** – путь к tcl скрипту с внутренними командами
 - **project_name.xpr** запустить **Vivado** и открыть проект **project_name**
- **quartus** - запуск **Intel Quartus**
 - **project_name.qpf** запустить **Quartus** и открыть проект **project_name**
- **quartus_sh** – запуск **Intel Quartus** в консольном режиме
 - **--flow compile project_name** – выполнить сборку проекта **project_name**
- **quartus_pgm** – запуск программатора **Intel Quartus** в консольном режиме
- примеры использования:
 - **ddec/common/board/de10_lite/Makefile**
 - **ddec/common/board/nexys4_ddr/Makefile**

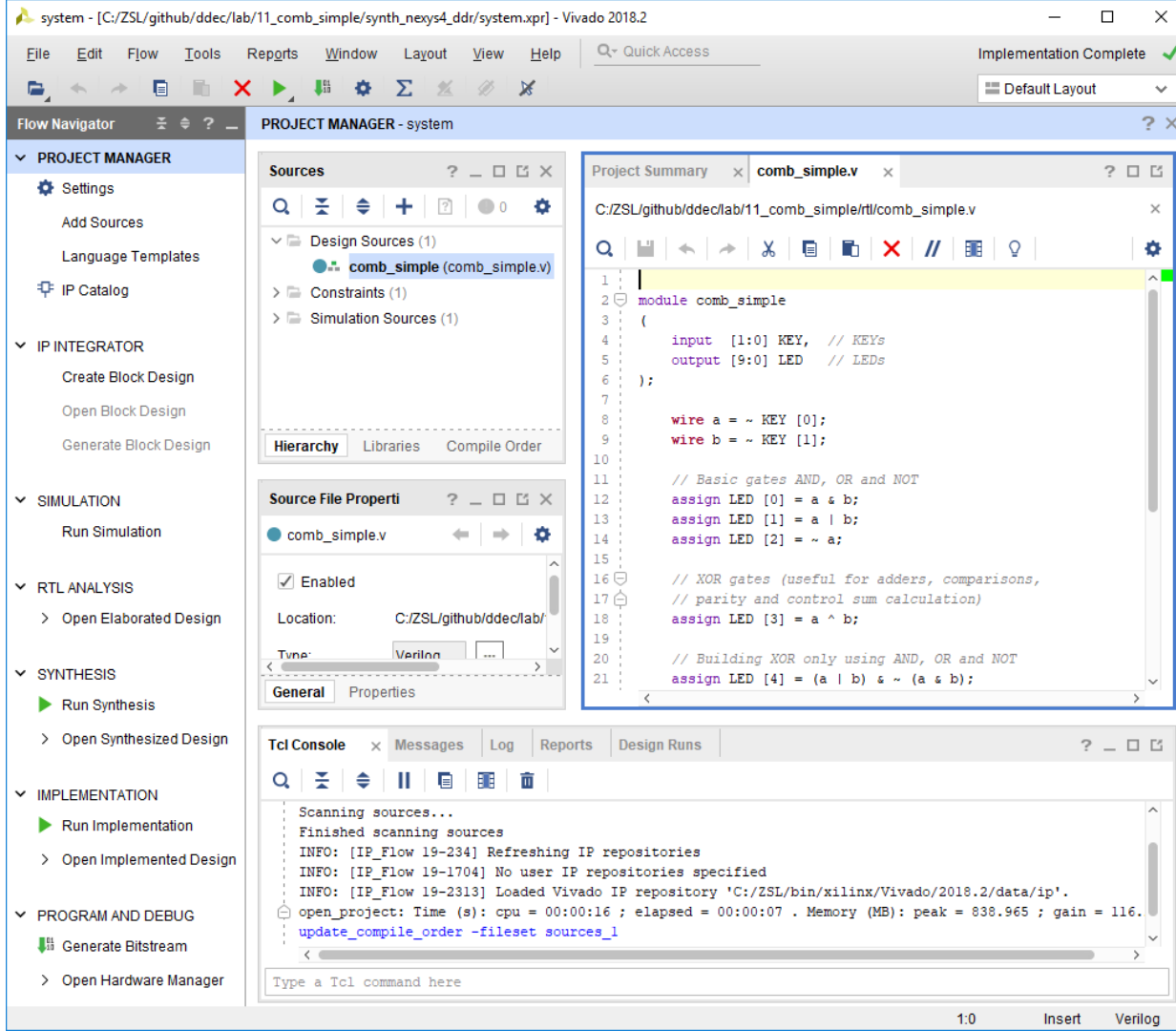
Xilinx Vivado

мощно и неторопливо



- **Xilinx Vivado** – пакет программ, использующийся для симуляции и синтеза при работе с ПЛИС Xilinx

- Преимущество: возможность автоматизировать абсолютно любое действие в виде скрипта TCL



- Представление **Elaborated Design** используется для просмотра схемы, построенной по результатам анализа RTL-кода проекта

system - [C:/ZSL/github/lab11_comb_simple/synth_nexys4_ddr/system.xpr] - Vivado 2018.2

File Edit Flow Tools Reports Window Layout View Help Quick Access Implementation Complete

Flow Navigator ELABORATED DESIGN - xc7a100tcs9324-1 (active)

PROJECT MANAGER

- Settings
- Add Sources
- Language Templates
- IP Catalog
- IP INTEGRATOR
 - Create Block Design
 - Open Block Design
 - Generate Block Design
- SIMULATION
 - Run Simulation
- RTL ANALYSIS
 - Open Elaborated Design
 - Report Methodology
 - Report DRC
 - Report Noise
 - Schematic
- SYNTHESIS
 - Run Synthesis
 - Open Synthesized Design
- IMPLEMENTATION
 - Run Implementation
 - Open Implemented Design

Netlist

- comb_simple
 - Nets (16)
 - KEY (2)
 - LED (10)
 - <const1>
 - a
 - b
 - m30
 - Leaf Cells (14)
 - a_i (RTL_INV1)
 - b_i (RTL_INV1)
 - LED0_i (RTL_XOR0)
 - LED0_i_0 (RTL_AND1)
 - LED0_i_1 (RTL_XOR0)
 - m0_i (RTL_INV1)
 - m1_i (RTL_OR0)
 - m2_i (RTL_INV1)
 - m3_i (RTL_AND1)

Cell Properties

m1_i

Name: m1_i

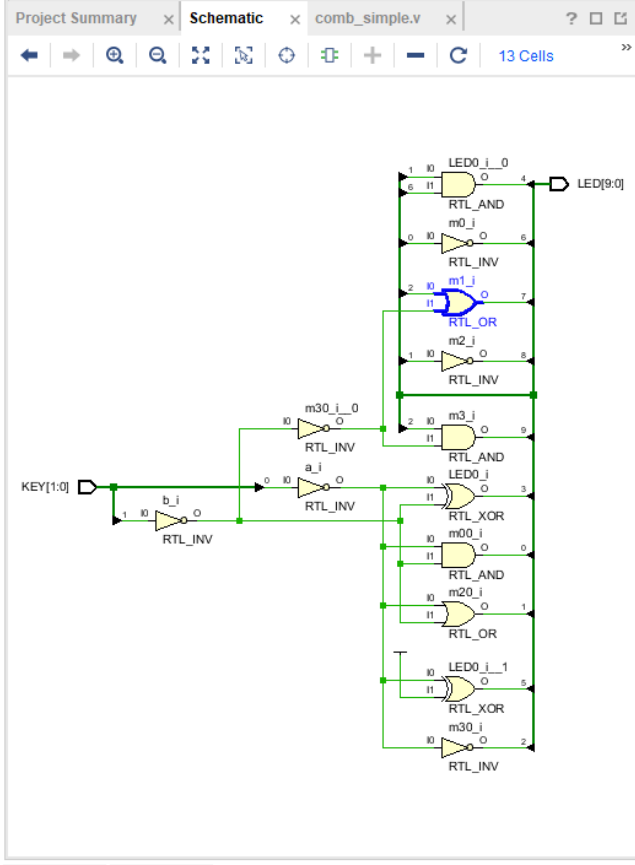
Reference name: RTL_OR0

Type: RTL Gate

Number of cell pins: 3

General Properties Nets

Project Summary Schematic comb_simple.v 13 Cells



- Представление Implemented Design – Schematic для просмотра результатов синтеза и конфигурации отдельных логических ячеек

system - [C:/ZSL/github/ddec/lab11_comb_simple/synth_nexys4_ddr/system.xpr] - Vivado 2018.2

File Edit Flow Tools Reports Window Layout View Help Quick Access Implementation Complete Default Layout

Flow Navigator

- PROJECT MANAGER
 - Settings
 - Add Sources
 - Language Templates
 - IP Catalog
- IP INTEGRATOR
- SIMULATION
- RTL ANALYSIS
- SYNTHESIS
- IMPLEMENTATION
 - Run Implementation
 - Open Implemented Design
 - Constraints Wizard
 - Edit Timing Constraint
 - Report Timing Summary
 - Report Clock Network
 - Report Clock Interaction
 - Report Methodology
 - Report DRC
 - Report Noise
 - Report Utilization
 - Report Power
 - Schematic

IMPLEMENTED DESIGN - xc7a100tcs9324-1 (active)

Sources Netlist

- comb_simple
 - Nets (20)
 - KEY (2)
 - KEY_IBUF (1)
 - LED (10)
 - LED_OBUF (7)
 - Leaf Cells (18)
 - KEY_IBUF[0]_inst (IBUF)
 - KEY_IBUF[1]_inst (IBUF)
 - LED_OBUF[0]_inst (OBUF)
 - LED_OBUF[0]_inst_i_1 (LUT2)
 - LED_OBUF[1]_inst (OBUF)
 - LED_OBUF[1]_inst_i_1 (LUT2)
 - LED_OBUF[2]_inst (OBUF)
 - LED_OBUF[3]_inst (OBUF)
 - LED_OBUF[3]_inst_i_1 (LUT2)
 - LED_OBUF[4]_inst (OBUF)
 - LED_OBUF[4]_inst_i_1 (LUT2)
 - LED_OBUF[5]_inst (OBUF)

Cell Properties

LED_OBUF[1]_inst_i_1

Name: LED_OBUF[1]_inst_i_1

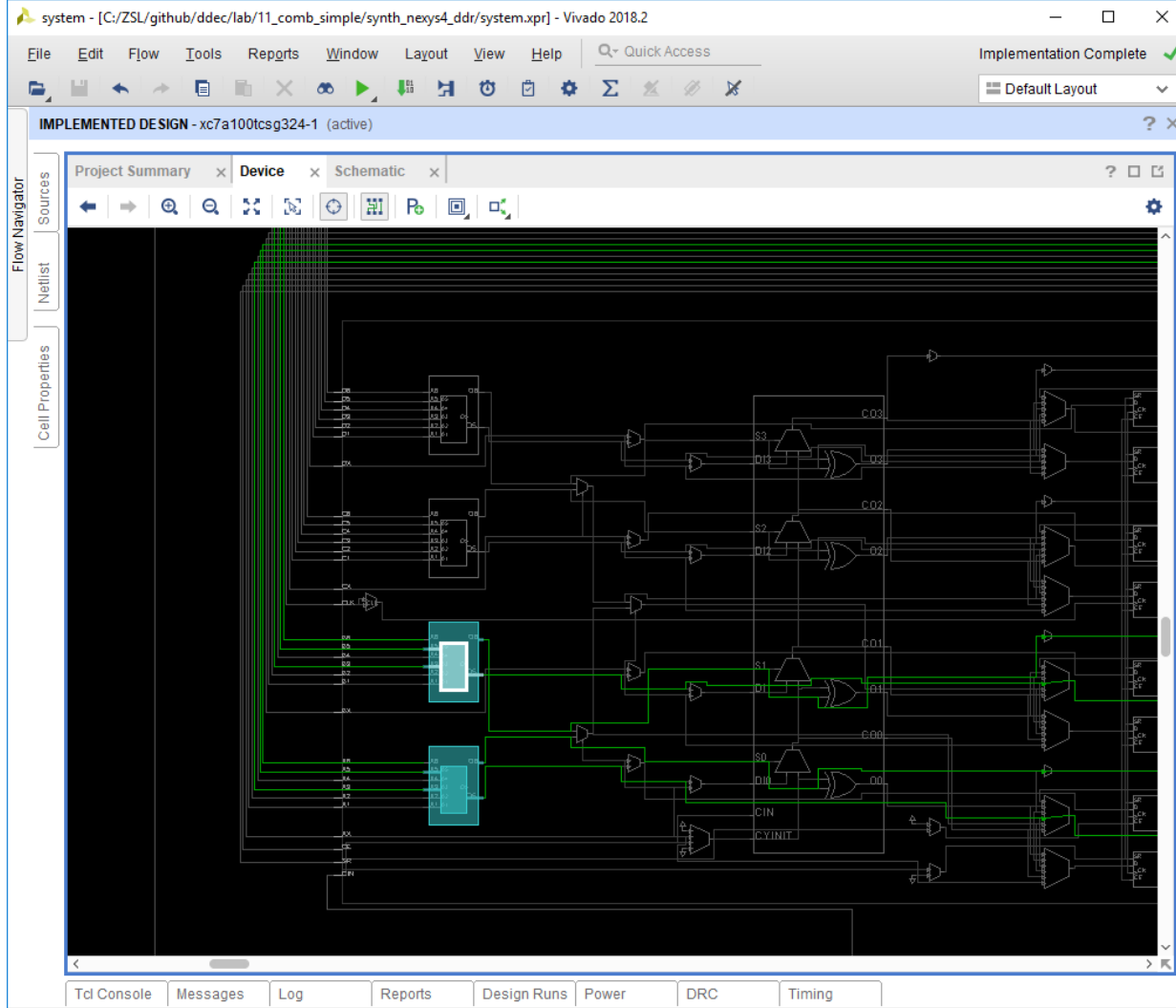
Reference name: LUT2

Type: LUT

Schematic

18 Cells

- В том же Implemented Design доступен режим представления в виде схемы, на которой показаны задействованные логические ячейки и соединения между ними



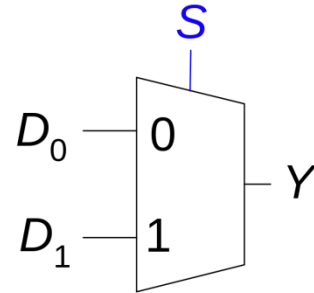
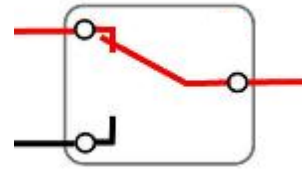
Мультиплексор на Verilog HDL

или дюжина способов
описать одно и то же



Мультиплексор – немного теории

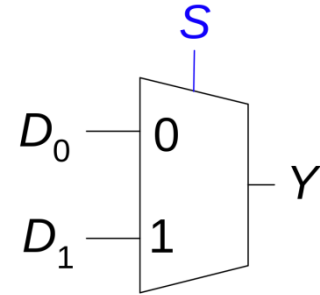
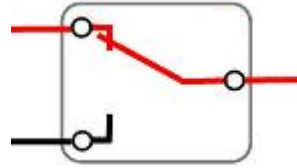
- Мультиплексор (multiplexer, mux) позволяет выбрать одно значение (сигнал) из нескольких - как переключатель
- Мы рассмотрим несколько примеров реализации мультиплексора
- Каждый из них использует разные конструкции Verilog HDL – любой язык проще изучать по примерам
- На практике можно использовать любой вариант реализации
- Каждый пример рассмотрен в виде отдельного модуля, на практике не обязательно выделять мультиплексор в отдельный модуль



| S | D_1 | D_0 | Y | S | Y |
|-----|-------|-------|-----|-----|-------|
| 0 | 0 | 0 | 0 | 0 | D_0 |
| 0 | 0 | 1 | 1 | 1 | D_1 |
| 0 | 1 | 0 | 0 | | |
| 0 | 1 | 1 | 1 | | |
| 1 | 0 | 0 | 0 | | |
| 1 | 0 | 1 | 0 | | |
| 1 | 1 | 0 | 1 | | |
| 1 | 1 | 1 | 1 | | |

Мультиплексор – немного теории

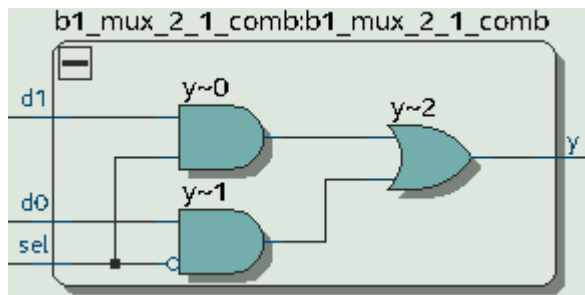
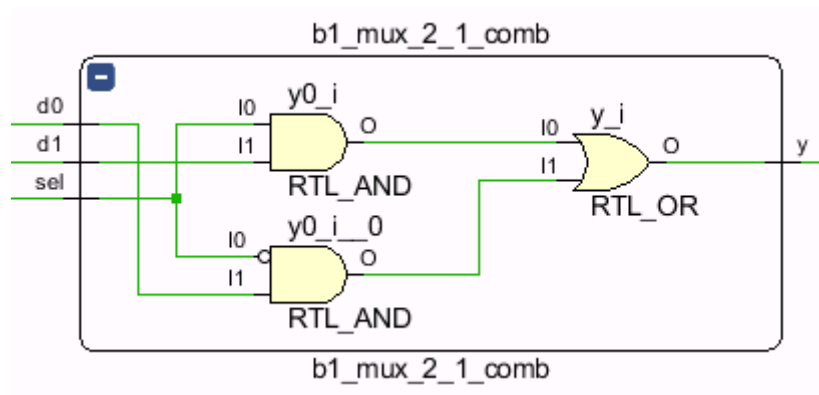
- Мультиплексор (multiplexer, mux) позволяет выбрать одно значение (сигнал) из нескольких - как переключатель
- Мы рассмотрим несколько примеров реализации мультиплексора, запоминать все – **не** обязательно
- Каждый из них использует разные конструкции Verilog HDL – любой язык проще изучать по примерам
- Каждый пример рассмотрен в виде отдельного модуля, на практике не обязательно выделять мультиплексор в отдельный модуль
- Перейдите в каталог [dddec/lab/12_comb_mux_1bit](#)



| S | D ₁ | D ₀ | Y | S | Y |
|---|----------------|----------------|---|---|----------------|
| 0 | 0 | 0 | 0 | 0 | D ₀ |
| 0 | 0 | 1 | 1 | 1 | D ₁ |
| 0 | 1 | 0 | 0 | | |
| 0 | 1 | 1 | 1 | | |
| 1 | 0 | 0 | 0 | | |
| 1 | 0 | 1 | 0 | | |
| 1 | 1 | 0 | 1 | | |
| 1 | 1 | 1 | 1 | | |

Однобитный мультиплексор – вариант 1

- На базе логических операторов



```
module b1_mux_2_1_comb
```

```
(
```

```
    input  d0,
```

```
    input  d1,
```

```
    input  sel,
```

```
    output y
```

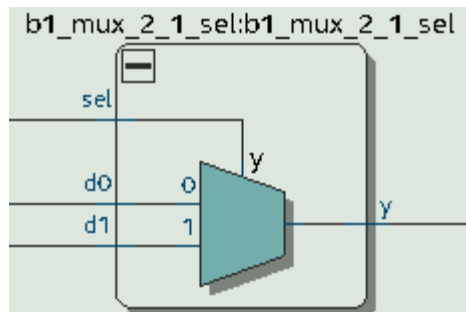
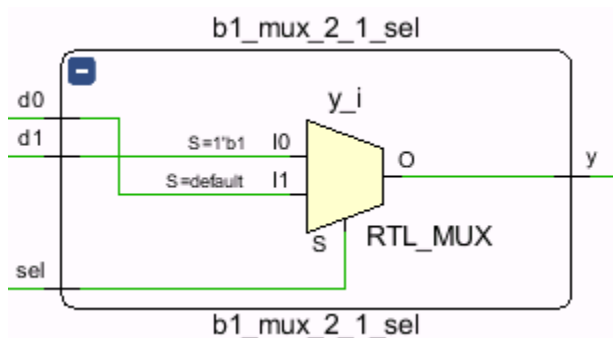
```
);
```

```
    assign y = (sel & d1) | (~sel & d0);
```

```
endmodule
```

Однобитный мультиплексор – вариант 2

- На базе тернарного оператора



```
module b1_mux_2_1_sel
(
    input  d0,
    input  d1,
    input  sel,
    output y

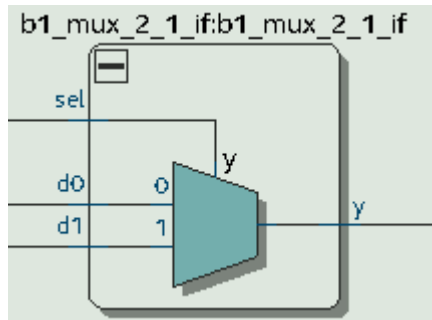
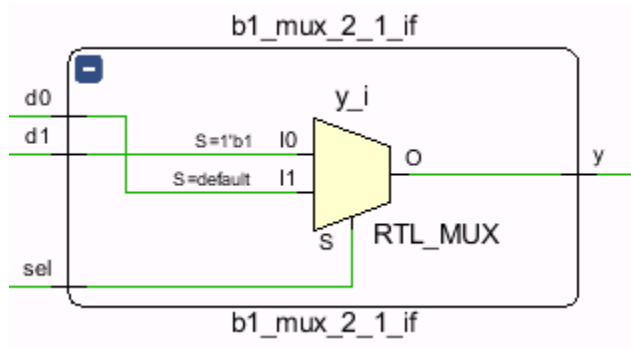
);

    assign y = sel ? d1 : d0;

endmodule
```

Однобитный мультиплексор – вариант 3

- На основе условного выражения **if-else**



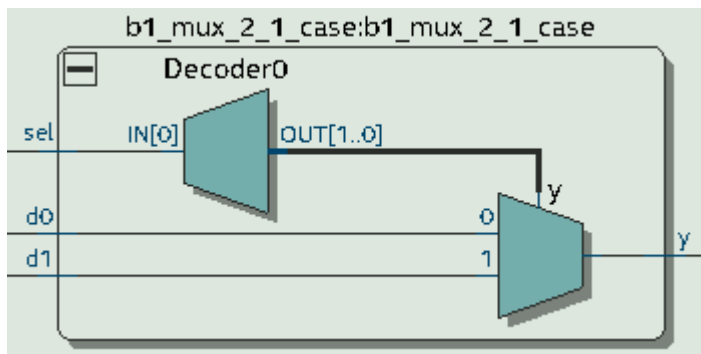
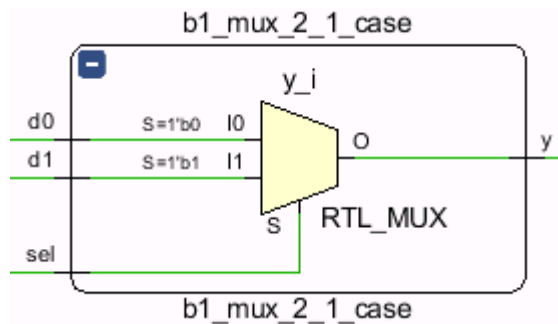
```
module b1_mux_2_1_if
(
    input  d0,
    input  d1,
    input  sel,
    output reg y

);

    always@(*)
    begin
        if(sel)
            y = d1;
        else
            y = d0;
        end
    endmodule
```

Однобитный мультиплексор – вариант 4

- На базе конструкции **case**



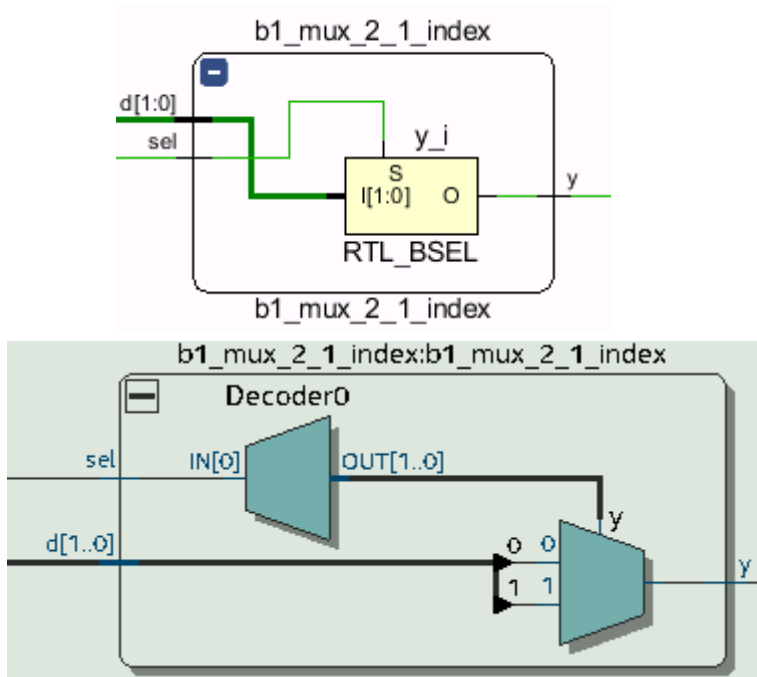
```
module b1_mux_2_1_case
(
    input  d0,
    input  d1,
    input  sel,
    output reg y

);

    always@(*)
    begin
        case (sel)
            0: y = d0;
            1: y = d1;
        endcase
    end
endmodule
```

Однобитный мультиплексор – вариант 5

- На основе доступа к элементу вектора по его номеру

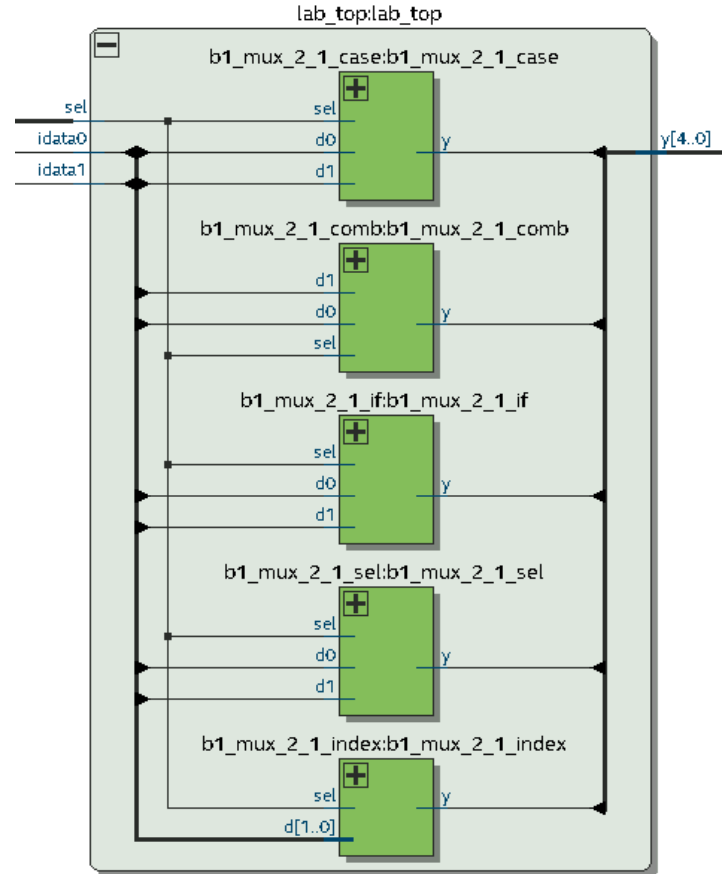


```
module b1_mux_2_1_index
(
    input  [1:0] d,
    input          sel,
    output         y
);
    assign y = d[sel];
endmodule
```

Однобитный мультиплексор – проверка

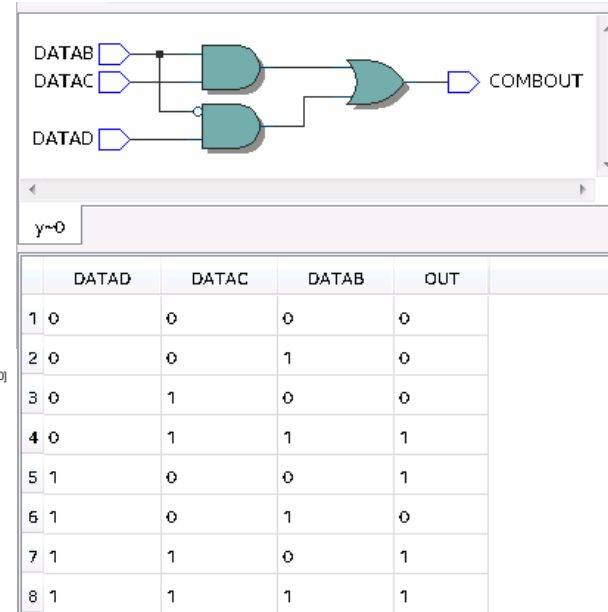
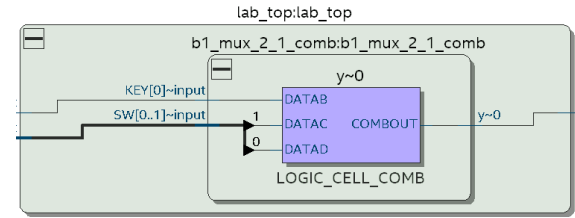
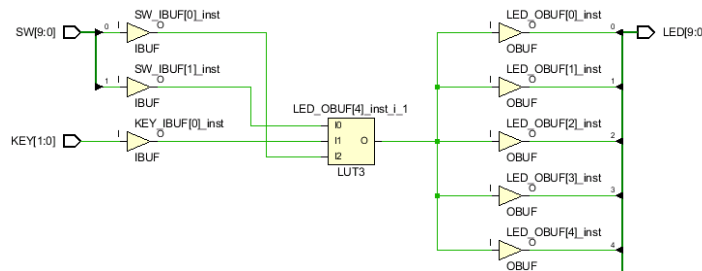
- Проверим, как будут работать разные мультиплексоры, объединив их в один модуль
- Перейдите в каталог **ddec/lab/12_comb_mux_1bit**
- Запустите проверку работы модуля в режиме симуляции:
make sim

```
# a=0 b=1 sel=x y=xx0xx
# a=0 b=1 sel=0 y=00000
# a=0 b=1 sel=1 y=11111
# a=0 b=0 sel=1 y=00000
# a=0 b=1 sel=1 y=11111
```



Однобитный мультиплексор – синтез

- Выполните синтез проекта и откройте его:
make synth
make open
- Откройте синтезированный проект в режиме просмотра **Technology Map Viewer**
- Так как таблица истинности для всех вариантов идентична, средства синтеза оптимизируют дизайн и оставляют только один мультиплексор на одном LUT
- Подключите отладочную плату и проверьте работу примера:
make load



Многопортовый N-битный MUX – вариант 1

- Перейдите в каталог [ddec/lab/13_comb_mux_Nbit](#)
- С фиксированным числом портов
- Используется выражение **case** с полным перечнем всех возможных вариантов адреса **sel**
- В любой комбинационной схеме для все возможных сочетаний входных сигналов должны быть предусмотрены выходные сигналы – иначе при синтезе в схему будут добавлены **защелки** - **latch** (рассматриваются отдельно)

```
module b2_mux_3_1_case_full
(
    input    [1:0] d0, d1, d2,
    input    [1:0] sel,
    output reg [1:0] y
);
// All possible values of input signals should be provided
// or an inferred latch will be generated!
always @(*) begin
    case (sel)
        2'b00: y = d0;
        2'b01: y = d1;
        2'b10: y = d2;
        2'b11: y = d2;
    endcase
end
endmodule
```

Многопортовый N-битный MUX – вариант 2

- С фиксированным числом портов
- Вместо перечисления всех возможных вариантов внутри **case**, задается значение по умолчанию вне **case**, но внутри этого же блока **begin - end**

```
module b2_mux_3_1_case
(
    input    [1:0] d0, d1, d2,
    input    [1:0] sel,
    output reg [1:0] y
);
    // All the possible input-output pairs should be provided!
    // The first way to provide all possible values of input signals
    // is to add the default value out of the 'case' block
    always @(*) begin
        y = d2;
        case (sel)
            2'b00: y = d0;
            2'b01: y = d1;
            2'b10: y = d2;
        endcase
    end
endmodule
```

Многопортовый N-битный MUX – вариант 3

- С фиксированным числом портов
- Вместо перечисления всех возможных вариантов внутри **case** задается значение по умолчанию с использованием **default**

```
module b2_mux_3_1_case_default
(
    input    [1:0] d0, d1, d2,
    input    [1:0] sel,
    output reg [1:0] y
);
    // All the possible input-output pairs should be provided!
    // The second way to provide all possible values of input signals
    // is to use the 'default' inside the 'case' block
    always @(*)
        case (sel)
            2'b00 : y = d0;
            2'b01 : y = d1;
            2'b10 : y = d2;
            default: y = d2;
        endcase
endmodule
```

Многопортовый N-битный MUX – вариант 4

- С фиксированным числом портов
- С параметризуемой шириной данных
- Используется **casez**
- Оператор **casez** позволяет использовать символ **?** в значении «**любое значение бита**»
- Пример использования параметризуемого модуля:

```
module b2_mux_3_1_casez
#(
    parameter WIDTH = 1
)(
    input      [WIDTH-1:0] d0, d1, d2,
    input      [      1:0] sel,
    output reg [WIDTH-1:0] y
);

    always @(*)
        casez (sel)
            2'b00: y = d0;
            2'b01: y = d1;
            2'b1?: y = d2;
        endcase
endmodule
```

Пример использования параметризируемого модуля

- У модуля 1 параметр

```
b2_mux_3_1_casex #(2) mux
(
    .d0(idata0),
    .d1(idata1),
    .d2(idata2),
    .sel(sel),
    .y(odata3)
);
```

- У модуля несколько параметров

```
bN_mux_N_1
#(
    .DATA_WIDTH ( 2 ),
    .ADDR_WIDTH ( 2 )
)
bN_mux_N_1
(
    .d    ( input_data ),
    .sel  ( sel         ),
    .y    ( odata4      )
);
```

ddec/lab/13_comb_mux_Nbit/rtl/lab_top.sv

Многопортовый N-битный MUX – вариант 5

- Используется синтезируемое подмножество языка **SystemVerilog**
- **SystemVerilog** допускает использование упакованных массивов в списке портов модуля
- Полученный в результате мультиплексор может быть конфигурирован как по ширине данных, так и по количеству портов

```
module bN_mux_N_1
#(
    parameter DATA_WIDTH = 4,
              ADDR_WIDTH = 4,
    parameter INPT_WIDTH = 2**ADDR_WIDTH
)(
    // packed array in a port list is a SystemVerilog feature
    input  [INPT_WIDTH-1:0][DATA_WIDTH-1:0] d,
    input                                     [ADDR_WIDTH-1:0] sel,
    output                                     [DATA_WIDTH-1:0] y
);
    assign y = d[sel];

endmodule
```

Упакованный массив SystemVerilog

- В документации упоминается как packed array

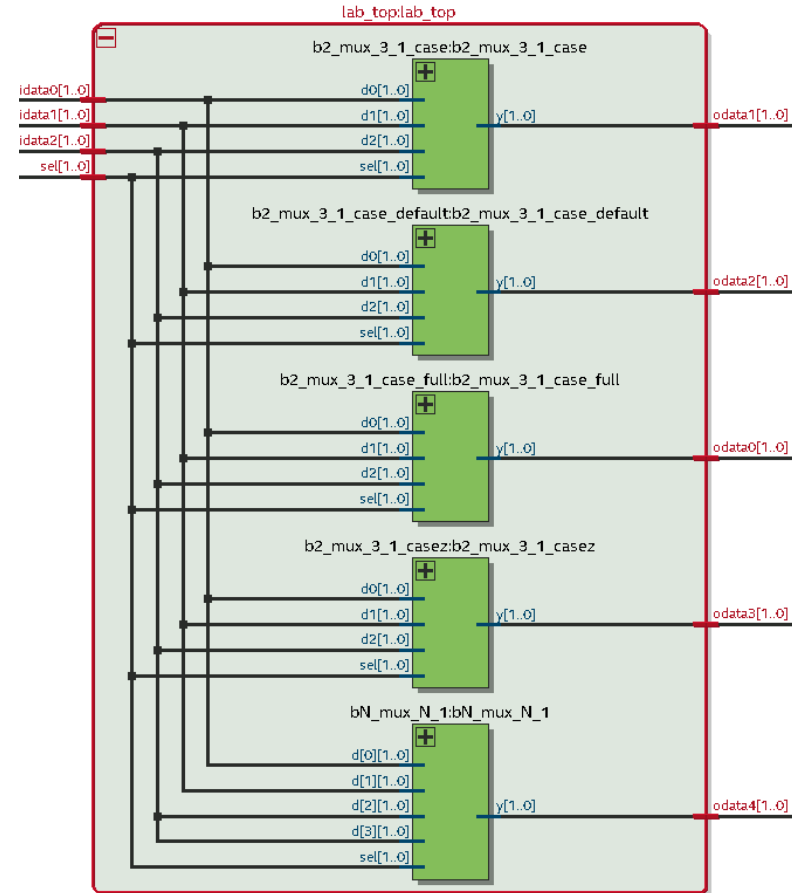
```
wire [ 7:0] d  
wire [3:0][1:0] a;  
assign a = d;
```

| d[7:0] | | | |
|-----------|-----------|-----------|-----------|
| d[7:6] | d[5:4] | d[3:2] | d[1:0] |
| a[3][1:0] | a[2][1:0] | a[1][1:0] | a[0][1:0] |

Многопортовый N-битный MUX – тест и синтез

Повторите проверки для N-портового MUX:

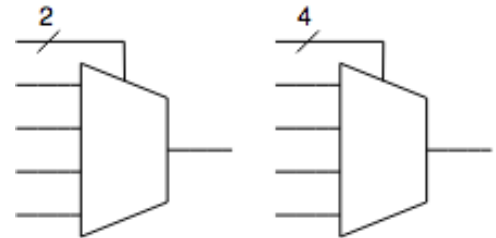
- Перейдите в каталог **ddec/lab/13_comb_mux_Nbit**
- Запустите проверку работы модуля в режиме симуляции: **make sim**
- Выполните синтез проекта и откройте его:
make synth
make open
- Откройте синтезированный проект в режиме просмотра **Technology Map Viewer**
- Что можно сказать по результатам симуляции и синтеза?
- Подключите отладочную плату и проверьте работу примера: **make load**



One-hot MUX

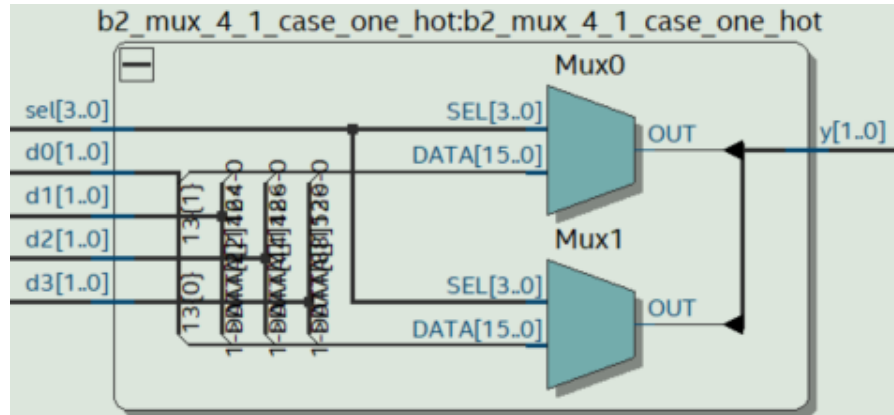
- Унитарная (**one-hot**) шина адреса – ширина шины адреса равна количеству входных каналов мультиплексора
- Для выбора активного канала на соответствующий ему бит шины адреса устанавливается в единицу, остальные биты при этом устанавливаются в ноль
- Позволяет получить более эффективную аппаратную реализацию при небольшом количестве каналов
- Мультиплексор может быть преобразован в **one-hot** в ходе оптимизации, выполняемой средствами синтеза
- Перейдите в каталог ddec/lab/14_comb_mux_onehot

| binary | one-hot |
|--------|---------|
| 2'b00 | 4'b0001 |
| 2'b01 | 4'b0010 |
| 2'b10 | 4'b0100 |
| 2'b11 | 4'b1000 |



One-hot MUX – вариант 1

- С фиксированным числом портов
- Можно реализовать параметризованную ширину данных
- На основе оператора **case**

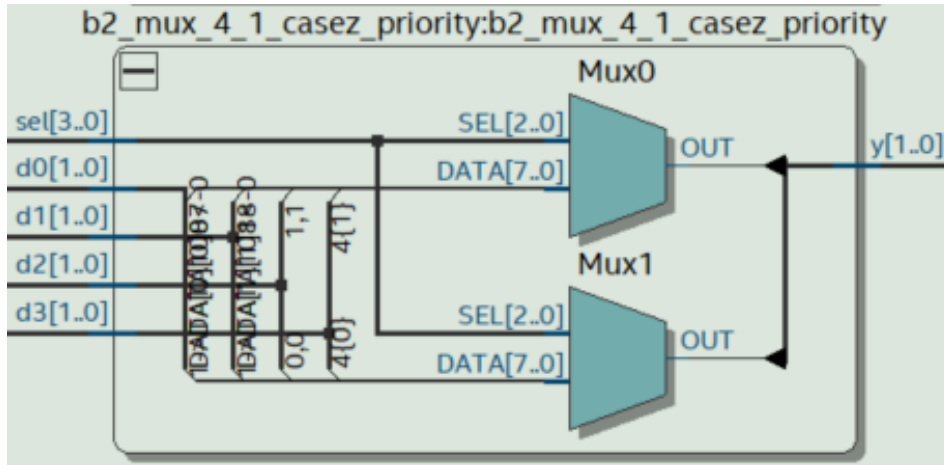


```
module b2_mux_4_1_case_one_hot
(
    input    [1:0] d0,
    input    [1:0] d1,
    input    [1:0] d2,
    input    [1:0] d3,
    input    [3:0] sel,
    output reg [1:0] y
);
```

```
always @(*)
    case (sel)
        4'b0001: y = d0;
        4'b0010: y = d1;
        4'b0100: y = d2;
        4'b1000: y = d3;
        default: y = d0;
    endcase
endmodule
```

One-hot MUX – вариант 2

- Фиксированное количество портов
- Можно реализовать параметризованную ширину данных
- С реализацией схемы приоритетов: на шине адреса может присутствовать несколько единиц
- На основе оператора **casez**



```
module b2_mux_4_1_casez_priority
(
    input    [1:0] d0,
    input    [1:0] d1,
    input    [1:0] d2,
    input    [1:0] d3,
    input    [3:0] sel,
    output reg [1:0] y
);

always @(*)
    casez (sel)
        4'b0001 : y = d0;
        4'b001? : y = d1;
        4'b01?? : y = d2;
        4'b1??? : y = d3;
        default : y = d0;
    endcase
endmodule
```

One-hot MUX – вариант 3

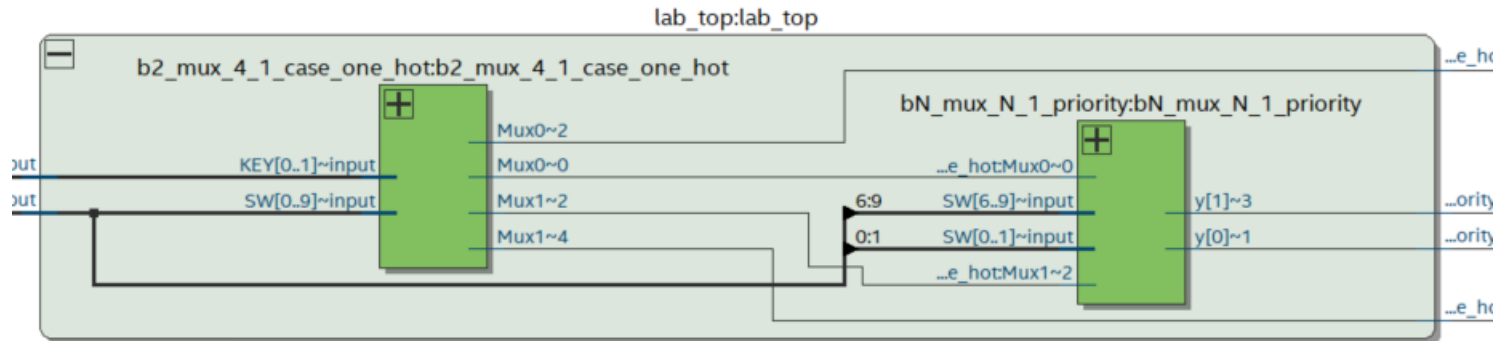
- Используется синтезируемое подмножество языка **SystemVerilog**
- **SystemVerilog** допускает использование упакованных массивов в списке портов модуля
- Полученный в результате мультиплексор может быть конфигурирован как по ширине данных, так и по количеству портов

```
module bN_mux_N_1_priority
#(
    parameter DATA_WIDTH = 4,
              INPUT_SIZE = 4
)()
    // packed array in a port list is a SystemVerilog feature
    input      [INPUT_SIZE-1:0][DATA_WIDTH-1:0] d,
    input      [INPUT_SIZE-1:0]                  sel,
    output reg  [DATA_WIDTH-1:0] y
);
    always @(*) begin
        y = d[0];
        for (int i = 0; i < INPUT_SIZE; i = i+1 ) begin
            if(sel[i])
                y = d[i];
        end
    end
end
endmodule
```

One-hot MUX – тест и синтез

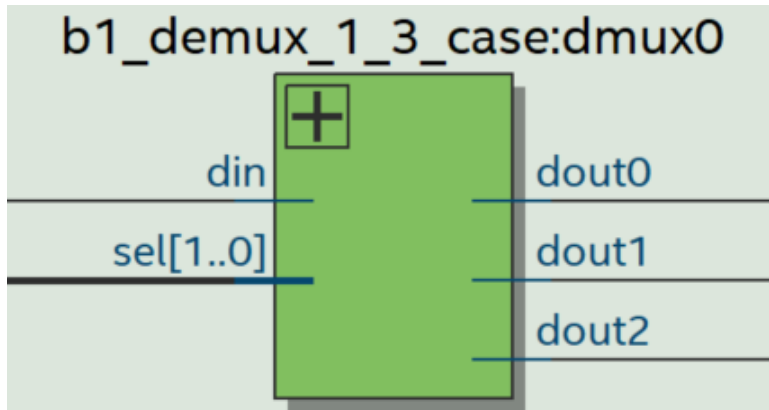
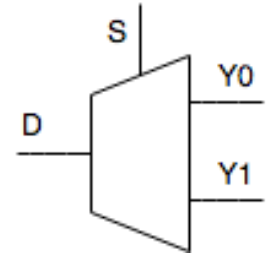
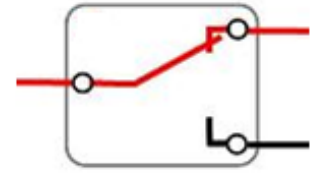
Повторите проверки для One-hot MUX:

- Перейдите в каталог **ddec/lab/14_comb_mux_onehot**
- Запустите проверку работы модуля в режиме симуляции: **make sim**
- Выполните синтез проекта и откройте его: **make synth**
make open
- Откройте синтезированный проект в режиме **Technology Map Viewer**
- Что можно сказать по результатам симуляции и синтеза?
- Подключите отладочную плату и проверьте работу примера: **make load**



Демультимплексор - теория

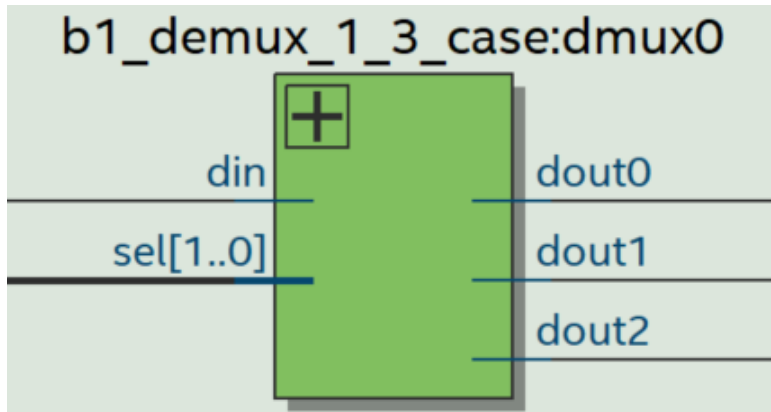
- Является мультиплексором «наоборот»
- Имеет 1 вход и N-выходов
- Установленный адрес определяет выход, на который будут продублированы значения, установленные на единственном входному порту данных
- Перейдите в каталог [dddec/lab/15_comb_demux_1bit](#)



| S | Y0 | Y1 |
|---|----|----|
| 0 | D | 0 |
| 1 | 0 | D |

1-битный демультиплексор – v1

- Использован оператор **case** и реализация «в лоб» - для каждого из возможных вариантов адреса перечислены все значения выходных сигналов
- Написание подобного кода может быть утомительным

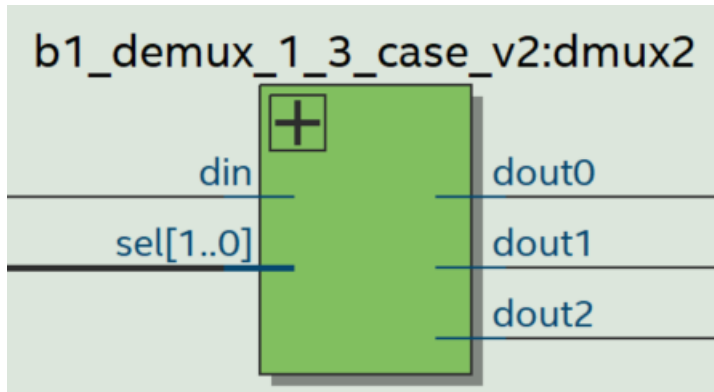


```
module b1_demux_1_3_case
(
    input          din,
    input          [1:0] sel,
    output reg     dout0,
    output reg     dout1,
    output reg     dout2
);

always @(*)
    case (sel)
        2'b00 : begin
                    dout0 = din;
                    dout1 = 0;
                    dout2 = 0;
                end
        2'b01 : begin
                    dout0 = 0;
                    dout1 = din;
                    dout2 = 0;
                end
        2'b10 : begin
```

1-битный демультиплексор – v2

- Оптимизированный вариант с использованием значения по умолчанию, которое задается за пределами блока **case**
- Результаты синтеза – полностью аналогичны предыдущему случаю



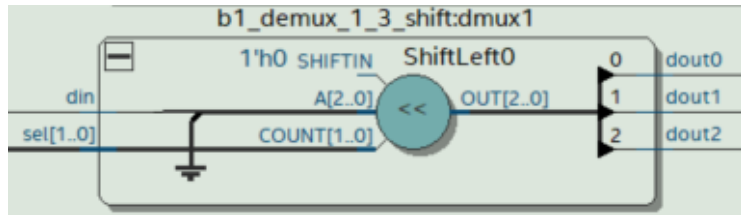
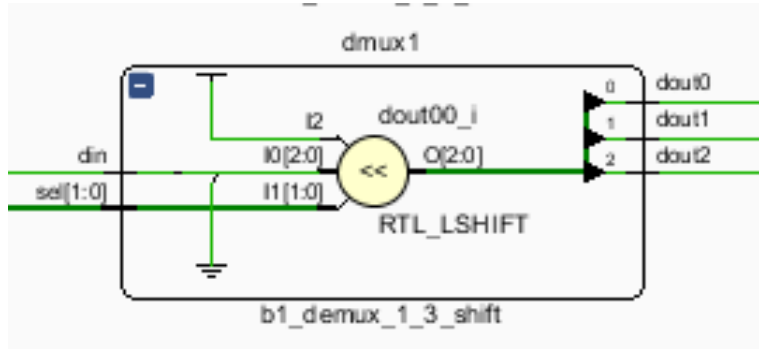
```
module b1_demux_1_3_case_v2
(
    input            din,
    input            [1:0] sel,
    output reg       dout0,
    output reg       dout1,
    output reg       dout2
);

always @(*) begin
    dout0 = 1'b0;
    dout1 = 1'b0;
    dout2 = 1'b0;
    case (sel)
        2'b00: dout0 = din;
        2'b01: dout1 = din;
        2'b10: dout2 = din;
    endcase
end

endmodule
```


1-битный демультиплексор – v3

- С использованием оператора сдвига



```
module b1_demux_1_3_shift
```

```
(
```

```
    input        din,
```

```
    input  [1:0] sel,
```

```
    output        dout0,
```

```
    output        dout1,
```

```
    output        dout2
```

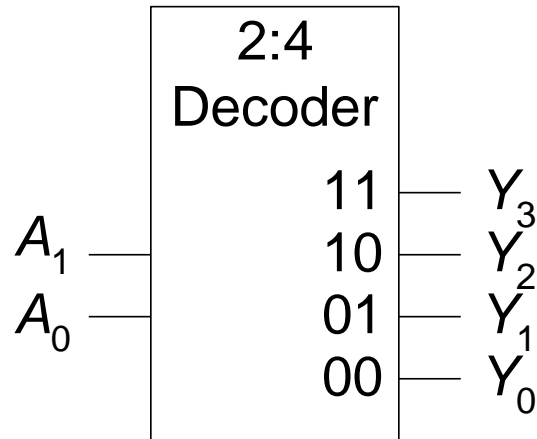
```
);
```

```
    assign { dout2, dout1, dout0 } = din << sel;
```

```
endmodule
```

Дешифратор

- Позволяет выполнить преобразование из двоичного кода в унитарный (one-hot)
- Мы не будем рассматривать примеры реализации дешифратора, т.к. его легко получить из примеров 1-битного демультиплексора – достаточно подать на вход данных единицу

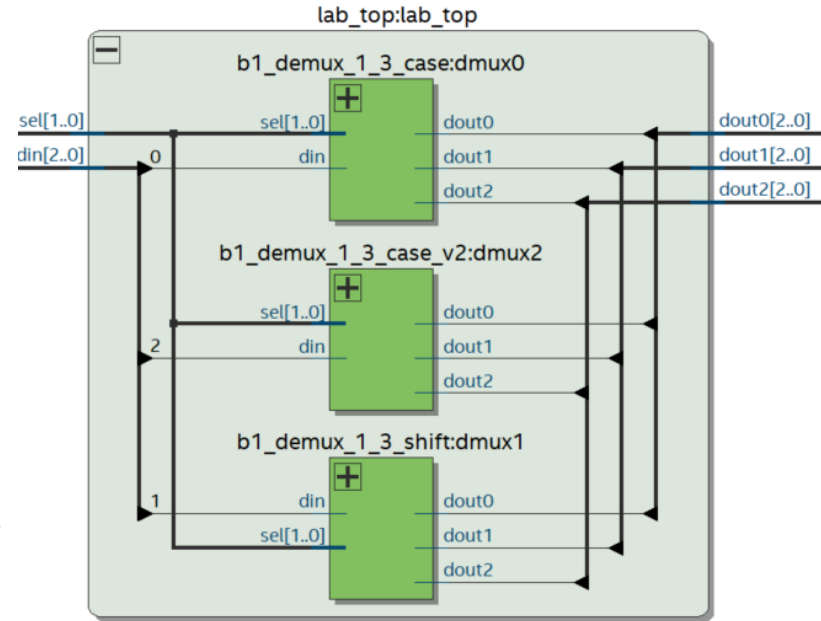


| A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

1-битный демультиплексор – тест и синтез

Повторите проверки для N-портового MUX:

- Перейдите в каталог **ddec/lab/15_comb_demux_1bit**
- Запустите проверку работы модуля в режиме симуляции: **make sim**
- Выполните синтез проекта и откройте его:
make synth
make open
- Откройте синтезированный проект в режиме просмотра **Technology Map Viewer**
- Что можно сказать по результатам симуляции и синтеза?
- Подключите отладочную плату и проверьте работу примера: **make load**



N-битный демультиплексор – v1

- Перейдите в каталог [dddec/lab/16_comb_demux_Nbit](#)
- Реализация практически не отличается от варианта 1-битного демультиплексора
- Фиксированное количество каналов

```
module bN_demux_1_4_case
#(
    parameter    DATA_WIDTH = 2
)
    input        [DATA_WIDTH-1:0] din,
    input        [        1:0] sel,
    output reg   [DATA_WIDTH-1:0] dout0,
    output reg   [DATA_WIDTH-1:0] dout1,
    output reg   [DATA_WIDTH-1:0] dout2,
    output reg   [DATA_WIDTH-1:0] dout3
);

localparam ZEROS = { DATA_WIDTH { 1'b0 } };

always @(*) begin
    dout0 = ZEROS;
    dout1 = ZEROS;
    dout2 = ZEROS;
    dout3 = ZEROS;

    case (sel)
        2'b00: dout0 = din;
        2'b01: dout1 = din;
        2'b10: dout2 = din;
        2'b11: dout3 = din;
    endcase
end
```

N-битный демультиплексор – v2

- Используется синтезируемое подмножество языка **SystemVerilog**
- **SystemVerilog** допускает использование упакованных массивов в списке портов модуля
- Модуль может быть конфигурирован как по ширине данных, так и по количеству портов

```
module bN_demux_1_N
#(
    parameter DATA_WIDTH = 4,
              ADDR_WIDTH = 4,
    parameter OUTPT_SIZE = 2**ADDR_WIDTH
)()
    input      [DATA_WIDTH-1:0] din,
    input      [ADDR_WIDTH-1:0] sel,
    // packed array in a port list is a SystemVerilog feature
    output reg [OUTPT_SIZE-1:0][DATA_WIDTH-1:0] dout
);

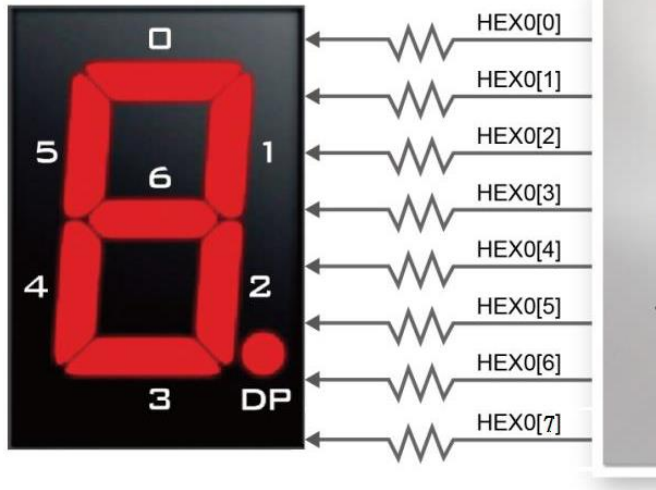
    localparam ZEROS = { (OUTPT_SIZE * DATA_WIDTH) { 1'b0 } };

    always @(*) begin
        dout = ZEROS;
        dout[sel] = din;
    end

endmodule
```

Практический пример – 7seg

- 7-сегментный индикатор
- Используется для вывода цифр, может быть использован для вывода букв
- Вывод инвертирован – при подаче 1 сегмент индикатора **не** светится
- Перейдите в каталог [dddec/lab/17_comb_hex](#)



```
always @*
```

```
case (dig)
```

```
4'h0: hex = 8'b11000000;
```

```
4'h1: hex = 8'b11111001;
```

```
4'h2: hex = 8'b10100100;
```

```
4'h3: hex = 8'b10110000;
```

```
4'h4: hex = 8'b10011001;
```

```
4'h5: hex = 8'b10010010;
```

```
4'h6: hex = 8'b10000010;
```

```
4'h7: hex = 8'b11111000;
```

```
4'h8: hex = 8'b10000000;
```

```
4'h9: hex = 8'b10010000;
```

```
4'ha: hex = 8'b10001000;
```

```
4'hb: hex = 8'b10000011;
```

```
4'hc: hex = 8'b11000110;
```

```
4'hd: hex = 8'b10100001;
```

```
4'he: hex = 8'b10000110;
```

```
4'hf: hex = 8'b10001110;
```

```
endcase
```

Практический пример – Terasic

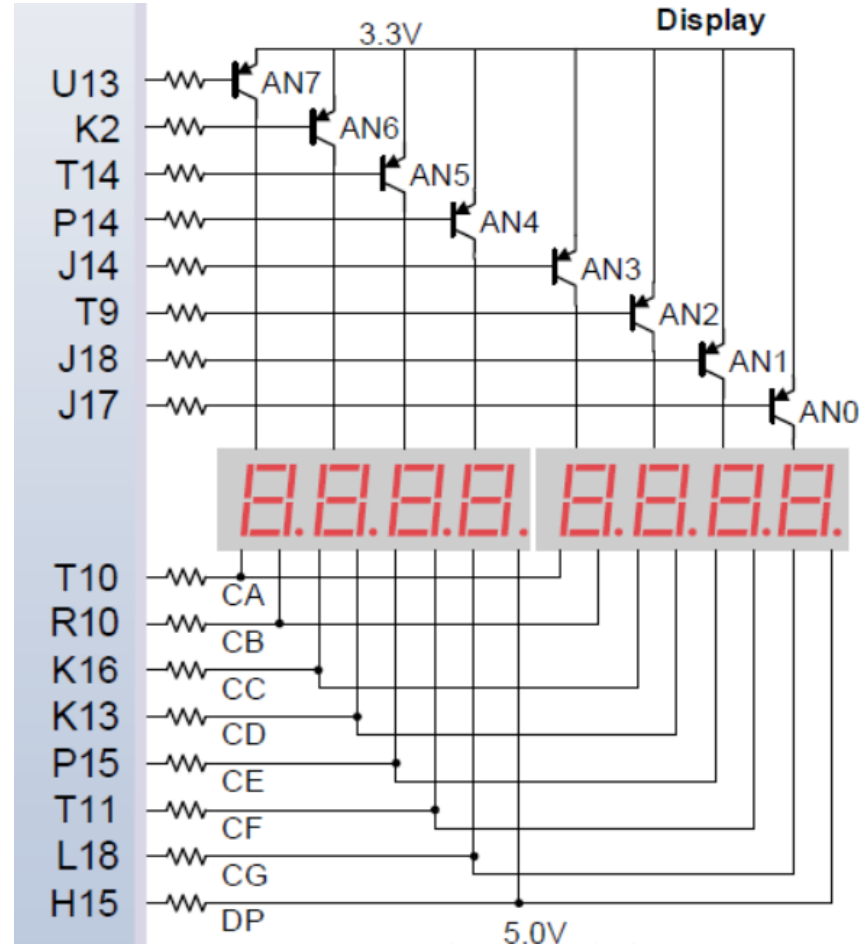
- На отладочных платах Terasic DE10-Lite и Terasic DE0-CV каждый из 7-seg индикаторов подключен к FPGA независимо от другого
- Поэтому для вывода одновременно на все индикаторы достаточно подобного решения

```
module seven_seg_6
(
    input  [23:0] data,
    output [ 7:0] hex0,
    output [ 7:0] hex1,
    output [ 7:0] hex2,
    output [ 7:0] hex3,
    output [ 7:0] hex4,
    output [ 7:0] hex5
);

seven_seg_digit dd0 (.dig (data[4*0 +: 4]), .hex (hex0) );
seven_seg_digit dd1 (.dig (data[4*1 +: 4]), .hex (hex1) );
seven_seg_digit dd2 (.dig (data[4*2 +: 4]), .hex (hex2) );
seven_seg_digit dd3 (.dig (data[4*3 +: 4]), .hex (hex3) );
seven_seg_digit dd4 (.dig (data[4*4 +: 4]), .hex (hex4) );
seven_seg_digit dd5 (.dig (data[4*5 +: 4]), .hex (hex5) );
```

Практический пример – Digilent (1 / 2)

- На отладочной плате Digilent Nexys 4 DDR индикаторы подключены по схеме с общим анодом
- В один момент времени может быть активен только один разряд из всех присутствующих на плате



Практический пример – Digilent (2 / 2)

- Данный модуль вывести информацию **data** на один из разрядов индикатора
- Активный разряд индикатора определяется значением **digit**
- В случае плат Digilent одной только комбинационной логики не достаточно для того, чтобы обеспечить отображение одновременно на всех индикаторах

```
module seven_seg_N_ca
#(
    parameter DIGITS    = 6,
              DT_WIDTH  = 4 * DIGITS,
              DN_WIDTH  = $clog2(DIGITS)
)
    input  [DT_WIDTH-1:0] data,
    input  [DN_WIDTH-1:0] digit,
    output [          7:0] hex,
    output [  DIGITS-1:0] an
);
    wire [DIGITS-1:0][3:0] value = data;
    wire                  [3:0] dig;

    assign an = ~(1 << digit);
    assign dig = value[digit];

    seven_seg_digit dd ( .dig (dig), .hex (hex) );

endmodule
```

Практический пример - синтез

- Перейдите в каталог [ddec/lab/17_comb_hex](https://github.com/ddec/lab/17_comb_hex)
- Выполните синтез для плат Digilent и Terasics и проверьте работу кода на каждой из них:

set BOARD=de10_lite

make synth

make open

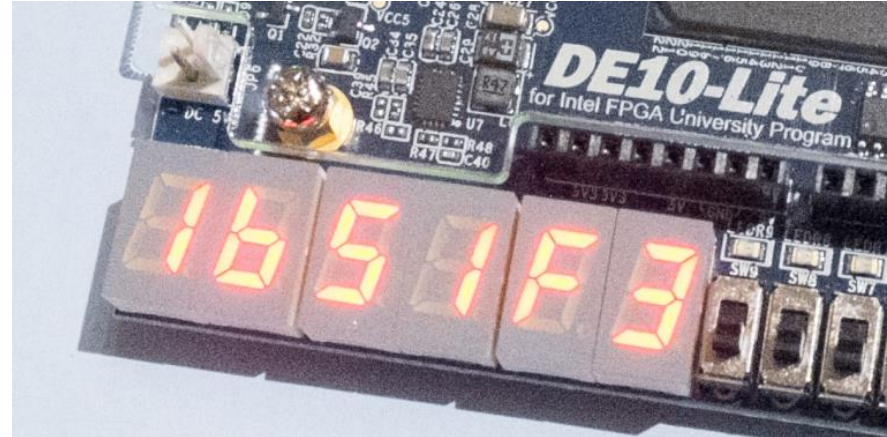
make load

set BOARD=nexys4_ddr

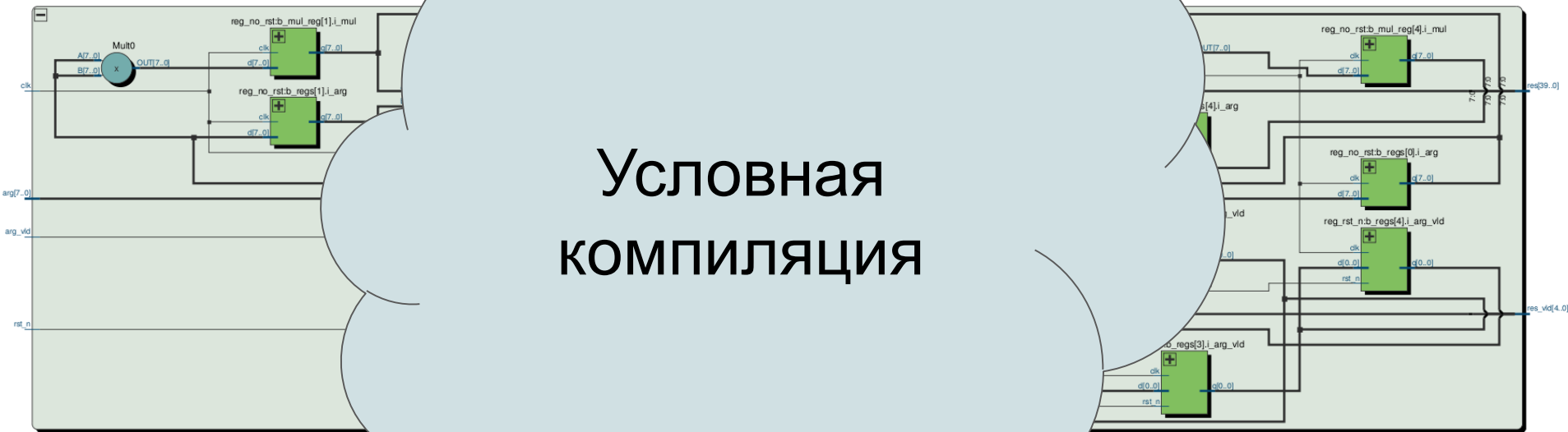
make synth

make open

make load



Условная компиляция



macro

- В проекте может присутствовать несколько модулей, выполняющих одну функцию
- В данном примере используется та реализация модуля, которая объявлена макросом MUX_IMPL
- Пример подобного использования:
ddec/lab/26_seq_synchronizers/rtl/synchronizer.v

```
`define MUX_IMPL bN_mux_N_1

module lab_top;

    `MUX_IMPL mux
    (
        .d    ( input_data ),
        .sel  ( sel        ),
        .y    ( odata4      )
    );

endmodule
```

macro

- Файл `ddec/lab/13_comb_mux_Nbit/rtl/lab_top.sv`
- В зависимости от того, объявлен макрос `ENABLE_SV_CODE` или нет, при синтезе и симуляции будет использоваться тот или иной код
- Макросы в Verilog очень похожи на директивы условной компиляции языка **C**, за одним существенным отличием: в **Verilog** отсутствует аналог директивы **`#if`** — мы можем проверить объявлен ли макрос, но не можем проверить его значение

```
`define ENABLE_SV_CODE

module lab_top;
    ...
`ifdef ENABLE_SV_CODE
    //some code
    ...
`else
    //some other code
    ...
`endif

endmodule
```

parameter & generate

- Файл `ddec/lab/16_comb_demux_Nbit/rtl/b2_demux_1_4_univ.v`
- Использование параметров в связке с `generate` позволяет обеспечить условную компиляцию в зависимости от значения параметра
- Минусы данного решения: в иерархию модуля, отображаемую в отладчике, вносятся изменения

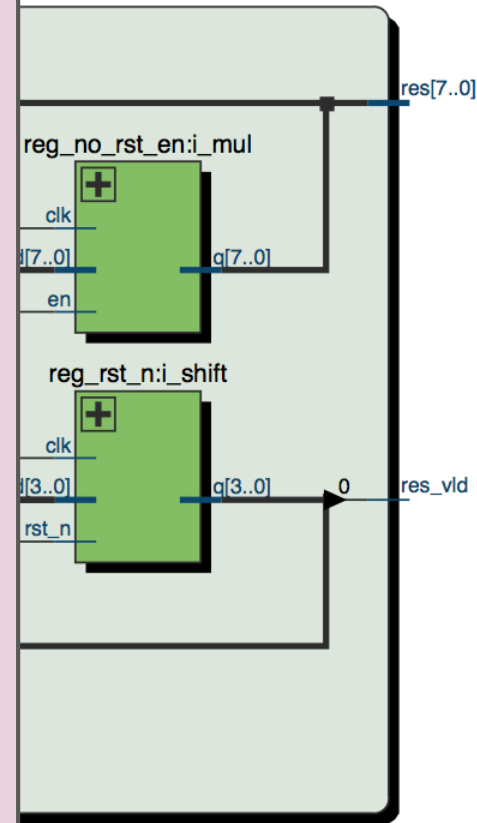
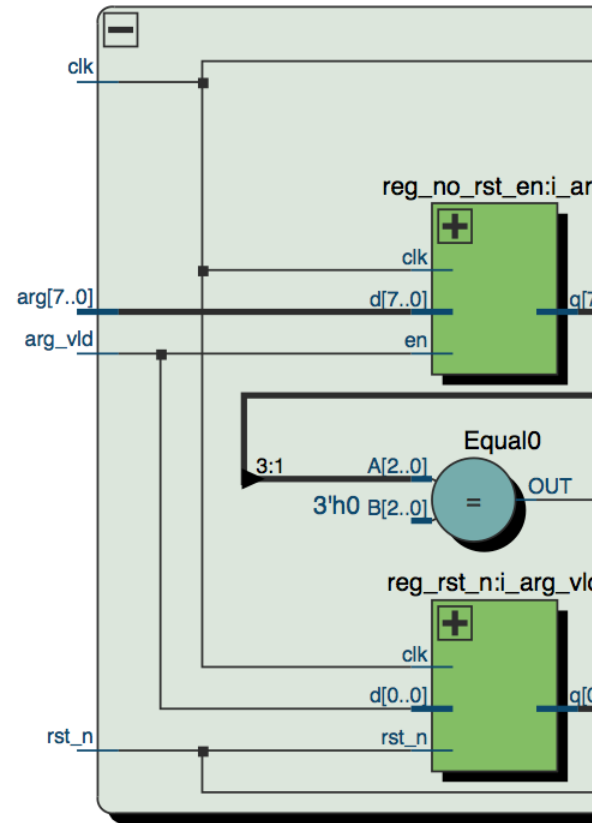
```
module b2_demux_1_4_univ
#(
    parameter MODE = "CASE"
)(
    // module port list
    ...
);

// generate & parameter can be used
// to select one of module implementations
generate
    // the 1st implementation
    if(MODE == "CASE") begin : demux_case
        ...
    end

    // the 2nd implementation
    else if(MODE == "SV") begin : demux_sv
        ...
    end

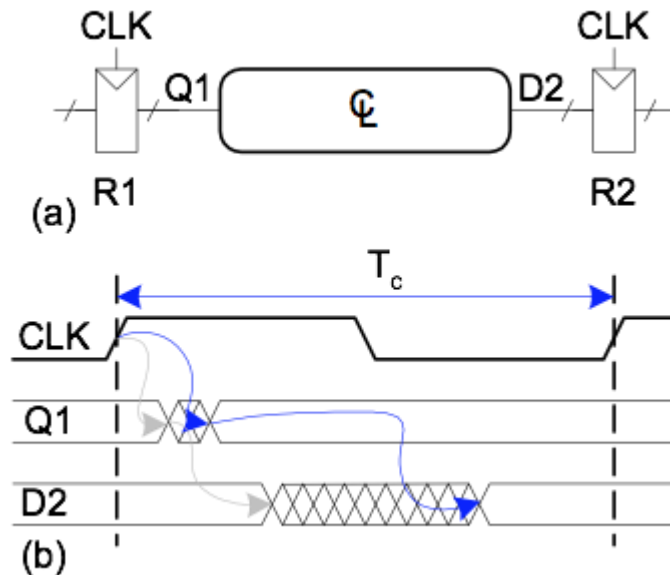
    // or the module stub!
    else begin : nothing
        ...
    end
endgenerate
endmodule
```

Анонс:
Симуляция и синтез
последовательных
схем



Комбинационная логика не работает мгновенно

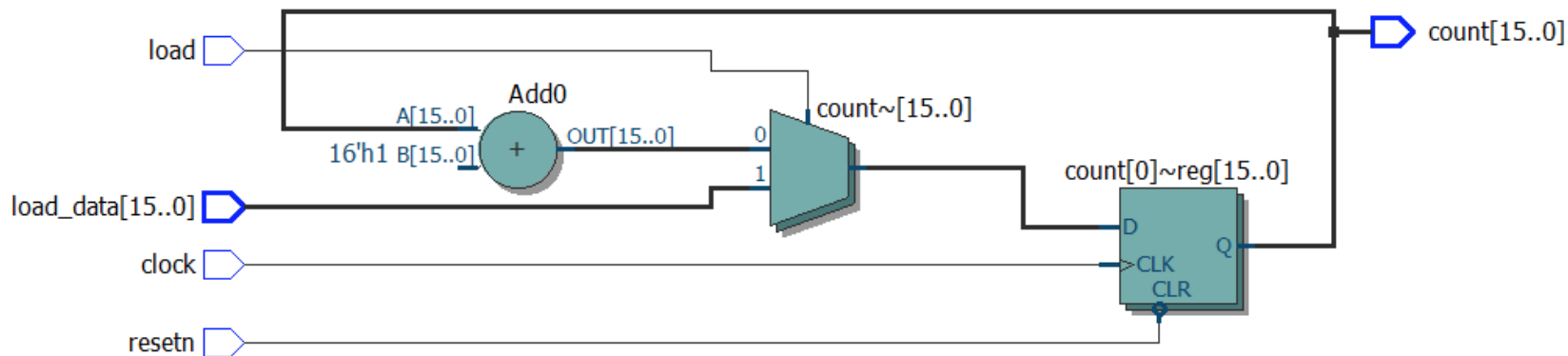
- Для распространения сигнала требуется время
- Перед тем, как сигналы примут окончательное значение на выходах блока могут появляться случайные значения
- Как определить момент, когда сигнал можно использовать?
- Мы можем синхронизировать вычисления с помощью специального тактового сигнала (clock).



The picture is from Digital Design and Computer Architecture
2nd Edition by David Harris and Sarah Harris. Elsevier, 2012

Тактовый сигнал полезен не только для синхронизации выходов схемы

- Схемы, синхронизированные с помощью тактового сигнала, называют **последовательностными**
- Последовательностные схемы также содержат элементы памяти и могут выполнять итерационные вычисления
- Данная тема рассматривается в Lab 2.



Задание

- Используйте 7-сегментные индикаторы для вывода цифр и букв
- При нажатии на одну из кнопок на индикаторе должны отображаться ваши инициалы
- При нажатии на вторую кнопку – дата вашего рождения

