

# Цифровая логика

## Оглавление

Цифровая логика .....	1
Введение .....	3
Комбинационная логика .....	4
Логические элементы .....	4
Практическая работа: инвертор.....	5
Демонстрация: инвертор .....	6
Синтаксис TL-Verilog: философия.....	6
Синтаксис TL-Verilog: логические операции .....	7
Синтаксис TL-Verilog: отступы и имена сигналов.....	8
Отступ .....	8
Имена сигналов .....	8
Практическая работа: логические элементы .....	9
Демонстрация: Логические элементы .....	9
Арифметическая логика .....	10
Концепция .....	10
Синтаксис TL-Verilog .....	11
Практическая работа: арифметические операторы .....	12
Демонстрация: арифметические операторы .....	12
Мультиплексор.....	13
Концепция .....	13
Синтаксис TL-Verilog .....	14
Практическая работа: калькулятор .....	15
Демонстрация: калькулятор.....	15
Литералы и конкатенация .....	16
Концепция и синтаксис .....	16
Литералы .....	16
Конкатенация.....	16

Практическая работа: моделирование калькулятора .....	16
Демонстрация: моделирование калькулятора.....	17
Визуальная отладка .....	18
Практическая работа: визуальная отладка .....	18
Демонстрация: визуальная отладка.....	19
Структура файла и маршрут выполнения .....	19
Принципы работы Makerchip.....	19
Видео: структура файла и концепция маршрута запуска инструментов .....	21
Мотивация.....	21
Последовательная логика .....	23
Концепция тактирования и триггеры .....	23
Примеры последовательной логики и сигнал сброса .....	23
Синтаксис TL-Verilog .....	26
Обобщение последовательной логики .....	26
Практическая работа: счетчик .....	27
Демонстрация: счетчик.....	27
Практическая работа: калькулятор с обратной связью .....	28
Демонстрация: калькулятор с обратной связью.....	29
Список источников .....	30

## Введение

В данной главе продемонстрированы основные схемы в Makerchip IDE. Она дает базовое понимание фундаментальных принципов проектирования цифровой логики и способов ее проектирования с использованием TL-Verilog в Makerchip IDE.

В этой главе вы изучите:

- основные логические элементы;
- способы объединения логических элементов в функции комбинационной логики более высокого порядка, включая мультиплексоры и арифметические схемы;
- синтаксис языка TL-Verilog для выражения комбинационных и арифметических логических функций;
- последовательностную логику и способы ее описания на TL-Verilog;
- способы отладки комбинационных схем в Makerchip, включая использование возможностей визуальной отладки, уникальных для платформы Makerchip.

## Комбинационная логика

### Логические элементы

Если вы уже знакомы с логическими элементами, можете пропустить этот раздел.

В цифровых схемах провода (источники сигналов) стабилизируются в одном из двух напряжений: высоком (VDD) или низком (VSS или земля). Таким образом, напряжение может быть интерпретировано как логическое значение, где высокое значение можно рассматриваться как 1/истина/включено, а низкое как 0/ложь/выключено. Это обеспечивает абстракцию для составления логических функций более высокого порядка с предсказуемым поведением.

Логические элементы – это основные базовые блоки для создания логических функций. Таблица ниже описывает основные логические элементы. Их функция определяется таблицами истинности, которые показывают значение выхода X для каждой комбинации входных значений A и B.

Name	NOT	AND	OR	XOR	NAND	NOR	XNOR																																																																																																
Symbol																																																																																																							
Truth Table	<table><tr><th>A</th><th>X</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	X	0	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	0	0	1	0	1	0	0	1	1	1	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	0	0	1	1	1	0	1	1	1	1	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	0	0	1	1	1	0	1	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	1	0	1	1	1	0	1	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	1	0	1	0	1	0	0	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	1	0	1	0	1	0	0	1	1	1
A	X																																																																																																						
0	1																																																																																																						
1	0																																																																																																						
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	1																																																																																																					
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					

Рисунок 1 – Таблица логических элементов

Примечание:

- элементы «И» (AND) и «ИЛИ» (OR) соответствуют их русским значениям;
- маленький кружок (или «пузырек») на выходе некоторых элементов означает инвертированный выход;
- в элементе «XOR», он же «исключающее ИЛИ» и «XNOR», он же «исключающее ИЛИ-НЕ», «исключающее» означает «не оба»;
- логические элементы могут быть соединены в различные комбинации для создания логических функций более высокого порядка, как в приведенной ниже схеме (это схема, известная как «полный сумматор»);
- схемы, состоящие только из логических элементов, называются «комбинационными».

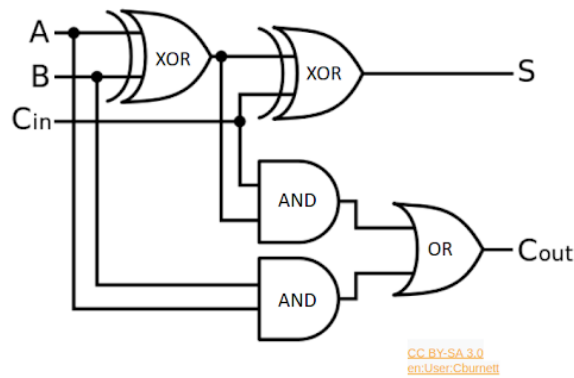


Рисунок 2 – Схема полного сумматора

Для заданного набора входных значений, таких как изображенные ниже, получим определенный набор выходных значений.

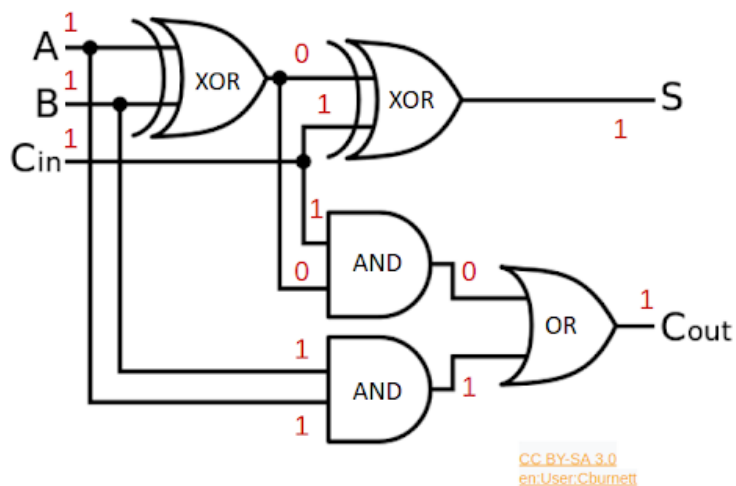


Рисунок 3 – Пример распространения логических значений в комбинационной схеме

### Практическая работа: инвертор

Разработаем код инвертора (элемент «НЕ» (NOT)) в Makerchip:

1. Перезагрузите IDE Makerchip, чтобы начать с шаблона кода по умолчанию. (Вы также можете использовать Ctrl-Z в редакторе, чтобы восстановить шаблон по умолчанию или загрузить шаблон по умолчанию со страницы примеров.)
2. В первой строке исходного файла указана языковая версия TL-Verilog. Если она отличается от «1d», возможно, потребуется изменить языковую версию, чтобы она соответствовала этому курсу. Для этого следует обратиться за инструкциями в репозиторий GitHub [1].
3. Вместо «//...» введите «\$out = ! \$in1;». Обязательно сохраните отступы в 3 пробела, как в других выражениях. Данный код реализует логику инвертора.

4. Скомпилируйте и проведите моделирование (выбрав пункт в меню «Е» редактора или Ctrl-Enter). Если на вкладках появятся красные крестики (вместо зеленых галочек), убедитесь, что вы правильно следовали инструкциям, и попытайтесь исправить ошибки. При необходимости используйте LOG для отладки или видео в разделе демонстрации.

Рассматривая полученные результаты, сделаем несколько ключевых наблюдений:

1. В отличие от Verilog, здесь нет необходимости объявлять сигналы (линии связи) (\$out и \$in1). В TL-Verilog оператор присваивания действует как объявление выходного сигнала или сигналов.
2. Эта схема имеет неподключенный входной сигнал и неподключенный выходной сигнал. Вероятно, в ней также есть неподключенный сигнал \$reset, который был определен в шаблоне. Это приводит к некритическим предупреждениям/ошибкам. О них сообщается в среде разработки, но они не препятствуют моделированию.
  - Обратите внимание на некритические ошибки в LOG и на соответствующие всплывающие окна при наведении курсора мыши в NAV-TLV.
  - Нет необходимости разрабатывать тестбенчи (код для проверки работоспособности) для подачи входных сигналов на преобразователь. Makerchip обеспечивает подачу случайных сигналов для неподключенных входов. По мере развития проекта вы захотите избавиться от неподключенных сигналов и задавать более целенаправленные входные сигналы, но пока ваш код находится в стадии разработки, автоматическая генерация входных сигналов может быть очень удобной.
3. Убедитесь в том, что в окне «WAVEFORM» выход инвертора действительно является обратным его входу.
4. Найдите логическое выражение для вашего инвертора в «DIAGRAM» во всплывающем окне при наведении мыши. Выберите выражение во вкладке «DIAGRAM» и проследите за его выделением на других панелях.

### **Демонстрация: инвертор**

В случае возникновения трудностей, для вас был подготовлен видеоматериал записи действий [2], которые нужно выполнить для выполнения работы.

### **Синтаксис TL-Verilog: философия**

TL-Verilog очень строго определяет синтаксис. Это может вызывать трудности у новичков с их собственными предпочтениями в стиле кодирования. Но эта строгость имеет важное

преимущество. В промышленности с кодом работают много людей из разных команд, и такая строгость обеспечивает определенность. Спецификацию синтаксиса TL-Verilog можно найти в меню «Help» («Помощь») Makerchip, но наиболее важные аспекты будут рассмотрены далее.

### Синтаксис TL-Verilog: логические операции

Логические операции имеют разную нотацию. На следующей схеме показаны некоторые из этих математических обозначений, а также операторы TL-Verilog (которые аналогичны операторам Verilog) для основных логических элементов (вентилей).








Op	Bool Arith	Bool Calc	Verilog	Gate
NOT	$\overline{A}$	$\neg A$	$\sim A$ (or $!A$ )	
AND	$A \cdot B$	$A \wedge B$	$A \& B$ (or $\&\&$ )	
OR	$A + B$	$A \vee B$	$A   B$ (or $  $ )	
XOR	$A \oplus B$	$A \oplus B$	$A \wedge B$	
NAND	$\overline{A \cdot B}$	$\neg (A \wedge B)$	$!(A \& B)$	
NOR	$\overline{A + B}$	$\neg (A \vee B)$	$!(A   B)$	
XNOR	$\overline{A \oplus B}$	$\neg (A \oplus B)$	$!(A \wedge B)$	

Рисунок 4 – Логические операции: графическое обозначение и ключевые слова TL-Verilog для основных логических элементов

Для группировки операций, чтобы сформировать более сложные логические функции, можно использовать круглые скобки. Если выражение растягивается на несколько строк, эти строки должны иметь больший отступ, чем первая строка. Выражения всегда должны заканчиваться точкой с запятой. Всегда ставьте пробел до и после «=». Например:

```
$foo = ( $val1 && $val2) ||
      (! $val1 && ! $val2);
```

## Синтаксис TL-Verilog: отступы и имена сигналов

### Отступ

TL-verilog обладает позиционным синтаксисом. т.е. отступы и пробелы имеют смысл.. Табуляции (которые не имеют определенного поведения) не допускаются. Каждый уровень отступа равен 3 пробелам (редактор Makerchip помогает придерживаться этим правилам).

### Имена сигналов

Если вы будете придерживаться предложенных в этом курсе имен сигналов, у вас никогда не возникнет проблем. Тем, кто захочет немного отклониться от сценария, следует знать, что TL-Verilog накладывает ограничения на имена сигналов. В то время как языки обычно оставляют выбор между camel-case регистром и подчеркиванием на усмотрение кодовых конвенций, TL-Verilog принуждает к такому выбору.

Ограничения наименований служат нескольким целям:

- они обеспечивают определенность;
- они различают типы;
- при обработке TL-Verilog в Verilog автоматически генерируемая логика может использовать имена сигналов Verilog, которые не конфликтуют с именами сигналов, созданными разработчиком.

В частности, имена сигналов TL-Verilog:

- имеют префикс «\$»;
- состоят из лексем, разделенных символами подчеркивания, где каждая лексема представляет собой строку символов нижнего регистра, за которыми следуют ноль или более цифр;
- начинаются как минимум с двух строчных символов.

Так, например, это корректные варианты наименований переменных:

- `$my_sig;`
- `$val1.`

А эти нет:

- `$a;`



- `$Sig` (на самом деле это «сигнал состояния», который в этом курсе использоваться не будет);
- `$val_1`;

В других местах вы можете встретить термин «`pipesignal`» относящийся к сигналам в TL-Verilog. Различие между сигналами Verilog и сигналами «`pipesignals`» TL-Verilog в этом курсе не имеет значения, и мы будем просто использовать сокращенный термин «сигналы».

### Практическая работа: логические элементы

1. Как и в случае с инвертором, попробуйте использовать другие одноэлементные логические выражения.

Если вы новичок в языках описания аппаратуры (HDL), попробуйте закодировать полную схему сумматора, как показано ниже (рисунок 5).

2. ). Сначала попробуйте сделать это с каждым логическим элементом в виде отдельного оператора, а затем попробуйте объединить три элемента, переводящие «`$carry_out`» в один оператор, используя круглые скобки для группировки подвыражений.

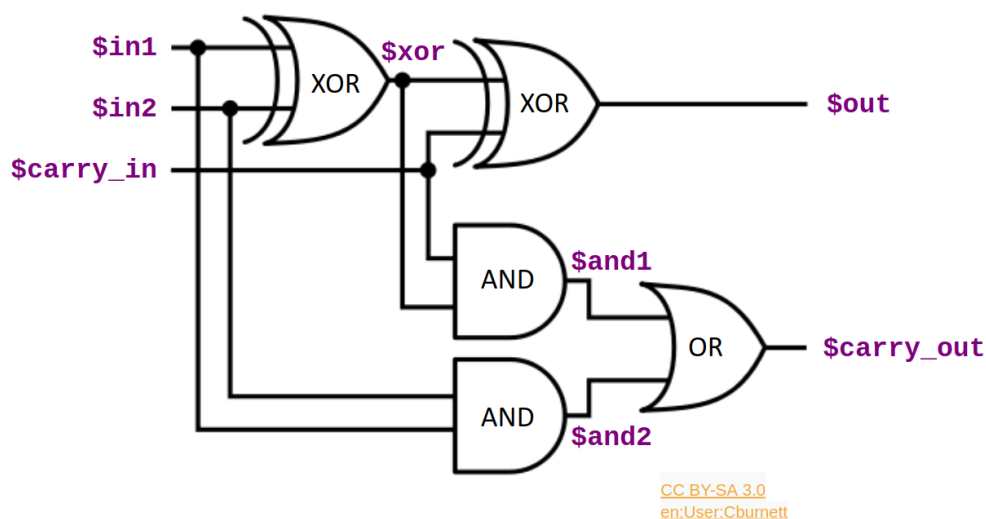


Рисунок 5 – Схема полного сумматора

### Демонстрация: Логические элементы

В случае возникновения трудностей, для вас был подготовлен видеоматериал записи действий [3], которые нужно выполнить для выполнения работы.

# Арифметическая логика

## Концепция

Если у вас есть опыт работы с языками описания аппаратуры (HDL), и вы знакомы с двоичной и шестнадцатеричной системой счисления, вы можете пропустить этот раздел.

В то время как отдельные сигналы (или «биты») хранят одно из двух значений в цифровой схеме, набор из N сигналов (называемых «шиной» или «вектором») позволяет описать до  $2^N$  возможных значений.

Привычный для всех формат представления чисел – в десятичной системе счисления (СС). В десятичной СС мы используем десять цифр (0-9), и когда мы считаем до последней доступной цифры (9), мы возвращаемся к 0 и увеличиваем значение следующего разряда, который равен десяти. Основание десять очень неудобно для цифровой логики. Двоичное основание или любая степень двойки (4, 8, 16) гораздо более естественна. В двоичной или бинарной СС есть цифры 0 и 1. Каждая из них может быть представлена битом. Шестнадцатеричная СС также довольно распространена. В ней используются цифры 0-9 и символы A-F (для чисел от 10 до 15). Одна шестнадцатеричная цифра может быть представлена 4 битами. В таблице ниже показаны значения от нуля до двадцати в десятичной, двоичной и шестнадцатеричной СС.

Decimal (base 10)	Binary (base 2)	Hexadecimal (base 16)
00	00000	00
01	00001	01
02	00010	02
03	00011	03
04	00100	04
05	00101	05
06	00110	06
07	00111	07
08	01000	08
09	01001	09
10	01010	0A
11	01011	0B
12	01100	0C
13	01101	0D
14	01110	0E
15	01111	0F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14

Рисунок 6 – Таблица десятичной, двоичной и шестнадцатеричной СС

Языки HDL поддерживают «векторные» сигналы, которые содержат несколько битов. Векторы могут использоваться для представления двоично-кодированных (знаковых или беззнаковых) целых значений. Например, 5-битный вектор может хранить значение 13 в виде битового значения 01101. HDL поддерживают арифметические операции, такие как сложение, которые работают с этими битовыми векторными значениями.

### Синтаксис TL-Verilog

В TL-Verilog наиболее распространенными типами данных являются битовые переменные (которые вы использовали в предыдущей лабораторной работе) и битовые векторы. Вектор объявляется путем указания его битового диапазона следующим образом:

```
$vect[7:0] = ....;
```

Битовые диапазоны обычно не используются в правой части выражения. Используется вектор с указанием нужного диапазон битов

В Verilog и TL-Verilog арифметические операции, такие как +, -, \*, /, и % (остаток от деления или деление по модулю) могут использоваться с векторами. Без этих операторов схему сумматора пришлось бы строить путем повторения полной схемы сумматора (которая была рассмотрена ранее) для каждой позиции бита в сумматоре. Это привело бы к созданию схемы «каскадного сумматора», изображенной ниже.

$$S = A + B$$

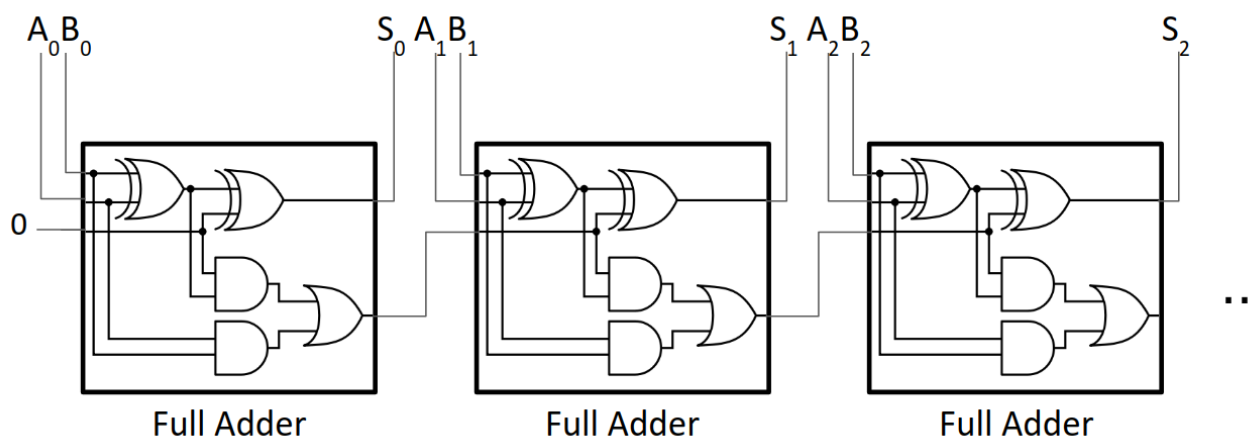


Рисунок 7 – Схема каскадного сумматора (ripple-carry), состоящая из полных сумматоров

Поддерживаются и другие векторные операторы, включая операторы сравнения ==, !=, >, >=, <, <=. Мы рассмотрим только те операторы, которые необходимы для данного курса.

## **Практическая работа: арифметические операторы**

Начните новый проект с шаблона по умолчанию, или удалите код, который был разработан ранее. Составьте несколько арифметических выражений.

Примечание:

Как и в предыдущих примерах, нет необходимости объявлять входные вектора. Таким образом, не будет определена их длина. В этом случае длина векторов будет задана при использовании битовых диапазонов в правой части выражения, следующим образом:

```
«$out[7:0] = $in1[6:0] + $in2[6:0];»
```

Значения векторных сигналов в программе просмотра временных диаграмм представлены в шестнадцатеричном виде. Существует множество онлайн-инструментов преобразования, таких как RapidTables, если нужно узнать десятичный или двоичный эквивалент.

Задание: закодируйте различные арифметические выражения и сравнения, для их усвоения.

## **Демонстрация: арифметические операторы**

В случае возникновения трудностей, для вас был подготовлен видеоматериал записи действий [4], которые нужно выполнить для завершения работы.

# Мультиплексор

## Концепция

Если вы знакомы с мультиплексорами, вы можете пропустить эту часть.

Одной из наиболее важных логических функций является мультиплексор (или MUX), изображенный ниже.

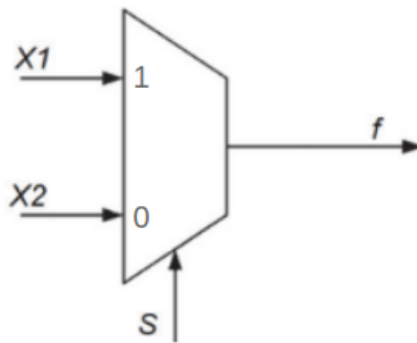


Рисунок 8 – Двухканальный однобитный мультиплексор с однобитным (закодированным) выбором

Мультиплексор осуществляет выбор между двумя или более входами (которые могут быть двоичными значениями, векторами или любыми другими типами данных). Линия выбора определяет вход, который нужно передать на выход. Чаще всего линией выбора является либо двоично-кодированный индекс входа, либо «one-hot» вектор, в котором каждый бит вектора соответствует одному входу (для выбора соответствующего входного значения подается сигнал на один и только один из битов).

MUX, изображенный выше, может быть построен из базовых логических вентилях, как показано ниже. Эту реализацию можно описать как «задать выход  $X1$ , если он выбран (по сигналу  $S == 1$ ) ИЛИ  $X2$  (если  $S == 0$ )».

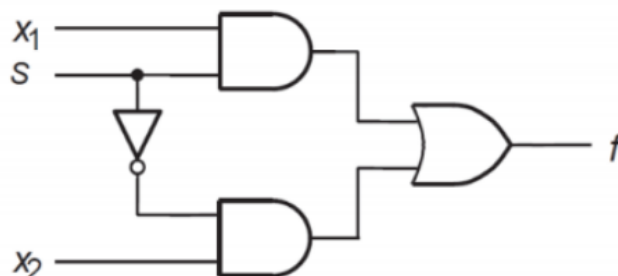


Рисунок 9 – Реализация мультиплексора на логических вентилях

## Синтаксис TL-Verilog

Verilog предоставляет не менее шести способов кодирования мультиплексора, каждый из которых имеет свои преимущества и недостатки. В TL-Verilog предпочтительно использовать тернарный оператор условия (`? :`), мы будем придерживаться такого подхода на протяжении всего курса. В своей простейшей форме тернарный оператор выглядит следующим образом:

```
«$out = $sel ? $in1 : $in0;»
```

Это выражение можно интерпретировать так:

```
«$out соответствует: $in1 если выбран $sel ($sel == 1), иначе $in0 (если $sel == 0)».
```

Тернарный оператор можно объединить в цепочку для реализации мультиплексоров с более чем двумя входными значениями, из которых нужно выбирать. Сами входы могут быть векторами. Для согласованности мы будем использовать очень специфическое форматирование кода, показанное ниже для четырехканального мультиплексора шириной 8 бит с однократным выбором (здесь сигналы `$in0-3` являются 8-битными векторами).

```
$out[7:0] =  
    $sel[3]  
        ? $in3 :  
    $sel[2]  
        ? $in2 :  
    $sel[1]  
        ? $in1 :  
    //default  
    $in0;
```

Это выражение устанавливает приоритет сверху вниз. Так, если выбран «`$sel[3]`», то «`$in3`» будет соединен с выходом независимо от других битов «`$sel`». Ниже показана его буквальная интерпретация, а также его одноэлементное представление (которое неоднозначно в отношении приоритета).

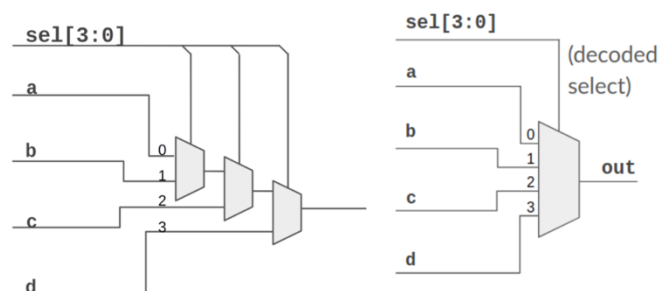


Рисунок 10 – Четырехканальный MUX, представленный как цепочка двухвходовых MUX в виде одного элемента

## Практическая работа: калькулятор

Закодируйте приведенную ниже схему. Эта схема реализует калькулятор, который может выполнять + (сложение), - (вычитание), \* (умножение) или / (деление) над двумя входными значениями. Приведите выражение для каждого из сигналов и обязательно используйте точные имена и кодировки выбора, указанные на входах MUX. Это будет важно в дальнейшем.

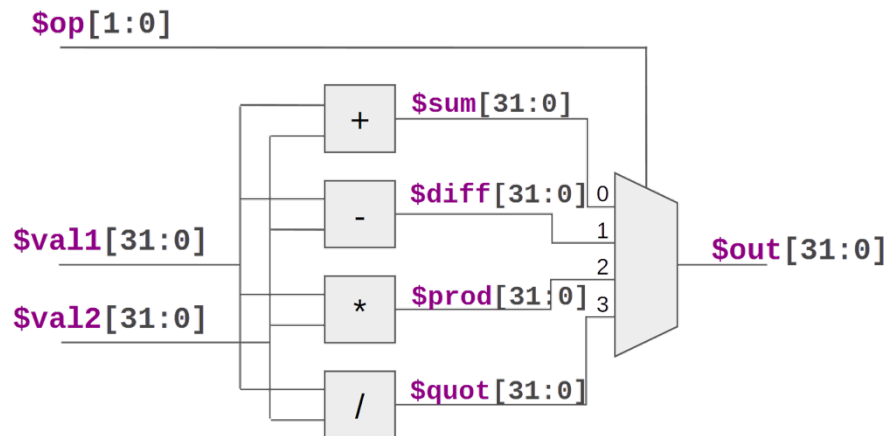


Рисунок 11 – Схема калькулятора

Обратите внимание на то, что эта схема использует кодированный (он же двоичный) выбор, где два бита « $\$op[1:0]$ » выбирают из четырех возможных входов MUX. « $\$op$ » можно декодировать, используя выражения. Например, « $\$op[1:0] == 2$ » для выбора « $\$prod$ ».

1. Нажмите кнопку «Save as New Project» («Сохранить как новый проект») в правом верхнем углу. Это создаст проект, который будет автоматически сохранен, и обновит URL для нового проекта. Сделайте закладку на этот новый URL и дайте ей имя, чтобы вы могли вернуться к своей работе позже (или, возможно, просто положитесь на то, что ваш браузер запомнит URL в своей истории). Текст статуса в правом верхнем углу подтвердит, что ваш файл автоматически сохраняется.
2. Разработайте код, описывающий приведенную на рисунке 11 схему.
3. Убедитесь, что ваш LOG или журнал ошибок содержит только ожидаемые ошибки/предупреждения о неподключенных сигналах.
4. Изучите схему, чтобы убедиться, что она выглядит правильно. (Пока не беспокойтесь об временных диаграммах, мы отладим эту схему в следующих двух практических работах.)

## Демонстрация: калькулятор

В случае возникновения трудностей, для вас был подготовлен видеоматериал [5].

## Литералы и конкатенация

### Концепция и синтаксис

#### Литералы

Если вы знакомы с синтаксисом выражений Verilog, вы можете пропустить этот раздел.

Выражение: «`$foo[7:0] = 6;`» определяет «`$foo`» для хранения постоянного значения 6. В данном случае 6 принудительно разделено на восемь бит присвоением. Часто необходимо явно указать ширину литерала:

```
«$foo[7:0] = 8'd6;»
```

Данное выражение явно присваивает «`$foo`» 8-битное десятичное («d») значение числа 6 («'» – символ одинарной кавычки). Другие способы записать тоже самое:

```
$foo[7:0] = 8'b110;    // 8-битная двоичная шестерка
```

или

```
$foo[7:0] = 8'h6;      // 8-битная шестерка в шестнадцатеричной СС
```

#### Конкатенация

Конкатенация битовых векторов – это объединение двоичных векторов один за другим с образованием вектора большей разрядности. Пример синтаксиса, реализующего операцию конкатенации:

```
$word[15:0] = {$upper_byte, $lower_byte};
```

### Практическая работа: моделирование калькулятора

Схема калькулятора управляется случайными 32-битными входными сигналами. Можно заметить, что вычисления часто теряют значимость или переполняются, то есть правильное значение результата может быть слишком большим ( $> 2^{32}$ ) или слишком маленьким ( $< 0$ ), чтобы выразить его в 32-битном сигналом `$out`.

Чтобы легче визуально проверить поведение симуляции, нужно использовать входные значения такие, чтобы результат не выходил за границы разрешенного диапазона значений. Для этого надо задать случаемыми только младшие биты `$val1` и `$val2`. А чтобы уменьшить вероятность переполнения, нужно, чтобы значение `$val2` было меньше чем `$val1`. Для этого надо:

1. Вернуться к проекту калькулятора.



2. Назначьте `$val1` таким образом, чтобы его старшие 26 бит были нулевыми, а оставшиеся 6 бит – случайными. Для этого нужно использовать литералы и конкатенацию. Для случайных битов используйте новый не назначенный сигнал `$val1_rand[5:0]`.
3. Аналогично назначьте `$val2`, но со случайными значениями только в младших 4 битах.

Шаблон кода по умолчанию, с который использовался вначале, компилируется без строгой проверки размерности используемых векторов. Включите строгую проверку, чтобы убедиться, что все сделано правильно.

4. Ниже строки `«m4_makerchip_module»`, добавьте в отдельную строку следующее выражение: `«/* verilator lint_on WIDTH */»`, и нажмите **Ctrl-Enter** для компиляции и симуляции. Проверьте LOG сообщения и выполните отладку кода при необходимости.
5. Теперь временные диаграммы будет понять легче, и можно визуально проверить работу созданного калькулятора. Убедитесь, что его сохранение произошло успешно (сообщение в верхнем правом углу).

### **Демонстрация: моделирование калькулятора**

В случае возникновения трудностей, для вас был подготовлен видеоматериал записи действий [6], которые нужно выполнить для выполнения этой практической работы.

## Визуальная отладка

### Практическая работа: визуальная отладка

Программы просмотра временных диаграмм являются давно зарекомендовавшим себя стандартным инструментом отладки при разработке схем. Но Makerchip поддерживает и более совершенную методику отладки. Рассмотрим несколько пользовательских визуализаций, чтобы помочь в отладке разработанного калькулятора.

Чтобы включить визуализацию:

1. Чтобы подключить библиотеку визуализации вставьте ниже строки «**m4\_makerchip\_module**» следующий код:  

```
m4_include_lib(['https://raw.githubusercontent.com/stevehoover/LF-Building-a-RISC-V-CPU-Core/main/lib/calc_viz.tlv']).
```

Возможно, потребуется исправить символы одиночных кавычек, заменив их после копирования и вставки.
2. Добавьте в область «**\TLV**» в конец строку кода «**m4+calc\_viz()**». Нажмите **<Ctrl>-Enter**.
3. Теперь в панели VIZ будет отображен калькулятор. Если необходимо, отладьте LOG. В NAV-TLV, строка «**m4\_include\_lib**» должна превратиться в комментарий, а макрос «**m4+calc\_viz()**» должна добавиться в блок «**\viz**» кода.
4. Расположите IDE так, чтобы можно было видеть и VIZ, и WAVEFORM. Воспользуйтесь VIZ, чтобы увидеть операции, выполняемые калькулятором. Обратите внимание, что калькулятор, как и диаграмма сигналов, показывает значения в шестнадцатеричном формате. Соотнесите то, что отображается в VIZ, с тем, что показано на осциллограмме. Если вы заметили неправильное поведение, отладьте код, выделив неисправные элементы схемы и исправив их.
5. Убедитесь, что проект сохранен.

Обычно разработчики создают собственные визуализации по мере разработки схемы, чтобы легче понимать общую картину поведения симулятора. В этом курсе акцент сделан на аппаратной реализации логических функций, и визуализации разработаны заранее. Но если вам интересно узнать, как создавать собственные визуализации, вы можете изучить код визуализации в панели NAV-TLV, где раскрываются макросы.

## Демонстрация: визуальная отладка

В случае возникновения трудностей, для вас был подготовлен видеоматериал записи действий [7], которые нужно выполнить для завершения работы.

## Структура файла и маршрут выполнения

### Принципы работы Makerchip

Данная тема представляет собой дополнительный материал. В ней описывается файловая структура TL-Verilog, используемая в Makerchip, и мотивы ее создания. Здесь описывается использование макропрепроцессора M4 и другие аспекты маршрута запуска инструментов, используемых при компиляции и моделировании. Этот раздел можно пропустить.

Чтобы получить более глубокое представление об этой теме, вы можете либо следовать последовательности инструкций, либо посмотреть видео, в котором эта тема рассматривается более подробно. Для того, чтобы узнать основы, в меню «Справка» («Help») IDE выберите «Справка» («Help») и прочитайте страницу справки.

Исходный файл TL-Verilog сначала обрабатывается макропрепроцессором M4. Именно так была импортирована и потом запущена визуализация калькулятора. Полученный файл TL-Verilog обрабатывается SystemVerilog или Verilog инструментом SandPiper™ компании Redwood EDA. Существует возможность просмотреть, как код обрабатывается в Verilog с помощью SandPiper, открыв «Show Verilog» в меню «Е» редактора.

Verilator это симулятор который выполняет симуляцию путем преобразования Verilog в компилируемое описание на C++ и затем выполнение.. Этот симулятор запускается для получения данных трассировки, которые можно просмотреть в программе отображения временных диаграмм.

На вкладке LOG отображаются результаты работы всех этих инструментов. Вывод M4 и SandPiper отображается синим цветом, а вывод Verilator и его итоговая симуляция – черным.

Функции TL-Verilog используются для определения логики внутри модуля (System)Verilog. Комментарии к коду ниже объясняют части исходного файла TL-Verilog, структурированные для использования в Makerchip.

### Листинг 1 – Исходный файл TL-Verilog

```
\m4_TLV_version 1d --bestsv: tl-x.org
// Данная строка версии определяет:
// макропрепроцессинг на M4 активен;
// версию используемого языка (1d);
```

```

// опциональные аргументы командной строки;
// ссылка на документацию.
\SV
// Данная область содержит SystemVerilog (или Verilog) код.
// SandPiper пропускает данный код напрямую в Verilog файл без
// какой-либо обработки.
m4_makerchip_module
// Данный M4 макрос расширяется до определения Verilog модуля
// с интерфейсом, который требует платформа Makerchip.
// Этот модуль интерфейса обеспечивает коммуникацию между
// Makerchip и проектом.
// Он включает глобальные входные
// тактовый сигнал и сигнал сброса.
// Через этот интерфейс и его выходные «успешные» («passed») и
// и «неуспешные» («failed») сигналы симуляция может быть
// завершена с сообщением "Simulation Passed" или "Simulation
// FAILED", которое отобразится в LOG.
//
// Чтобы увидеть расширение этого макроса, перейдите к панели
// NAV-TLV. Этот макрос также предоставляет случайный вектор,
// который может быть использован в качестве входного
// воздействия, и обеспечивает
// некоторую конфигурацию Verilator'a.
\TLV
// Синтаксис TL-Verilog доступен в данной области для описания
// пользовательского кода. В данном курсе мы всегда будем объявлять
// код в пределах модуля m4_makerchip_module, но можно
// также поместить логику TLV в отдельный модуль с определенным
// интерфейсом.
$reset = *reset;
// В \TLV контексте «*reset» ссылается на сигнал сброса
// (System)Verilog. Здесь он подключен к сигналу сброса
// TL-Verilog $reset pipesignal.
// Модуль с пользовательским кодом
*passed = ...;
*failed = ...;

```

```

        // Примените любой из них для завершения моделирования
        // (до кода ограничения количества циклов в Makerchip)
\SV
    // Возращение к контексту SystemVerilog,
    // чтобы завершить работу модуля.
endmodule

```

### **Видео: структура файла и концепция маршрута запуска инструментов**

Посмотрите данное 9-минутное видео [8], чтобы узнать все подробности того, как работает Makerchip.

### **Мотивация**

Для тех, кто интересуется мотивацией такой структуры файлов, необходимо понять стратегию развития TL-Verilog из Verilog. Конечной целью является создание нового языка моделирования, философски отличного от Verilog во всех отношениях. Это произойдет в течение следующего десятилетия или десятилетий. Пока же сообщество работает над этим постепенно, опираясь на Verilog в качестве рабочей отправной точки, и с языком TL-Verilog как расширением Verilog. Это наложение также обеспечивает последовательный и постепенный путь миграции. И по мере развития инструментов всегда можно вернуться к Verilog.

Также следует отметить тот факт, что TL-Verilog – это реализация TL-X на языке Verilog, расширение языка, определенное для использования поверх любого HDL для расширения его возможностей на уровне транзакций. Таким образом, существует возможность миграции с любого поддерживаемого HDL (на момент подготовки этого материала Verilog является единственным поддерживаемым HDL).

Благодаря использованию синтаксиса TL-Verilog только в определениях модулей, инструменты на базе Verilog, которые используются для сшивания связей между модулями, могут оставаться в неведении об использовании TL-Verilog. Внутри модуля Verilog можно использовать другие формы модульности и иерархии, характерные для TL-Verilog.

Весь код, который разрабатывается в этом курсе, находится внутри области `\TLV`. Теперь вы понимаете, как это связано с инструментами на базе Verilog.

#### **Примечание:**

Весь код, который используется в данном курсе, может быть написан так же и на Verilog, но, используя TL-Verilog, вы будете готовы к дальнейшему изучению проектирования цифровых схем с

использованием концепции транзакций. (Кроме того вкладки Makerchip DIAGRAM работают только с TL-Verilog.)

## Последовательная логика

### Концепция тактирования и триггеры

Если вы знакомы с последовательной логикой и триггерами, вы можете пропустить этот раздел.

В последовательной логике вводится понятие тактового сигнала (рисунок 12).



Рисунок 12 – Временная диаграмма тактового сигнала

Тактовые импульсы проходят через всю схему к триггерам, которые упорядочивают логику. Триггеры бывают разных типов, но самым простым и распространенным является «D-триггер с управлением по переднему фронту сигнала» (в курсе будет рассмотрен именно этот триггер). Они передают значение с входа на выход, но только при приходе переднего фронта тактового сигнала (рисунок Рисунок 13). Триггеры хранят значения выходных сигналов до следующего переднего фронта тактового сигнала.

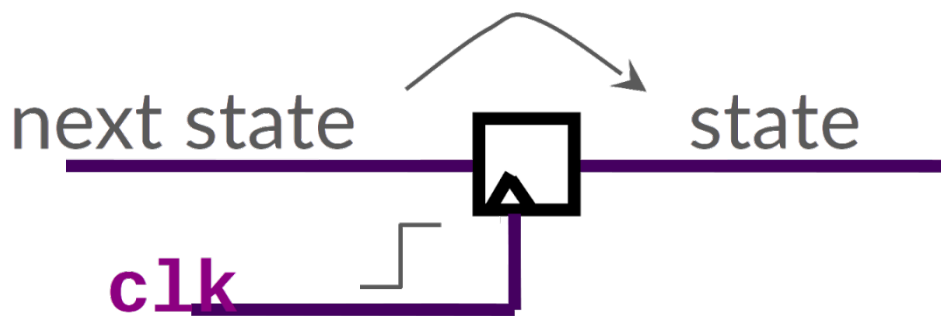
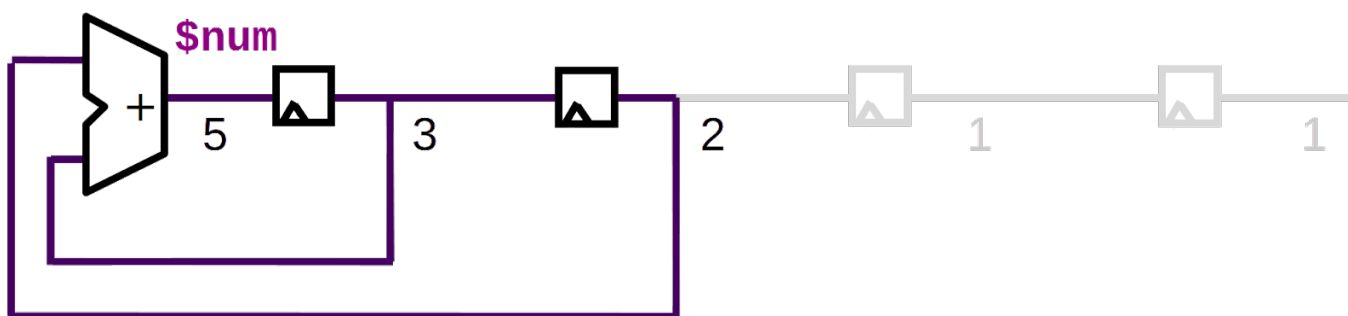


Рисунок 13 – Триггер

Хотя существуют триггеры, которые работают по заднему фронту тактового сигнала, разрабатываемые в данном курсе схемы будут работать только по переднему фронту. Кроме того, существуют триггеры, которые включают в себя логические функции. Поскольку в данном курсе будем использоваться только D-триггеры, в дальнейшем будем называть их просто триггерами.

### Примеры последовательной логики и сигнал сброса

Прежде чем перейти к теоретическим рассуждениям о последовательной логике, изучите схему, которая вычисляет последовательность Фибоначчи. Каждое число в последовательности Фибоначчи является суммой двух предыдущих чисел: 1, 1, 2, 3, 5, 8, 13, ... Эта схема будет вычислять следующее число в последовательности:



С каждым передним фронтом тактового сигнала значения будут распространяться через триггеры, сдвигая один триггер вправо, создавая следующую временную диаграмму:

Данная схема неполная. В ней не хватает входа сброса. Нужен способ инициализировать схему двумя единицами, чтобы начать генерацию последовательности.

В отличие от комбинационных схем, где выходные значения являются исключительно функцией входных значений, последовательные схемы имеют внутреннее состояние. Каждая последовательная схема должна иметь возможность перейти в известное состояние «сброса» («reset»). Поэтому в каждой такой схеме должен быть сигнал «сброса», который отвечает за это. Схема должна быть спроектирована таким образом, чтобы при подаче сигнала сброса схема стабилизировалась в известном состоянии сброса.

Самый простой подход к обеспечению возможности сброса заключается в том, чтобы при подаче сигнала сброса каждый триггер устанавливался в значение сброса. Эта методология используется некоторыми группами разработчиков, но, если важны площадь и энергопотребление, можно сделать иначе.

В рассматриваемой схеме Фибоначчи достаточно сбросить только первый триггер, как это показано на рисунке ниже.



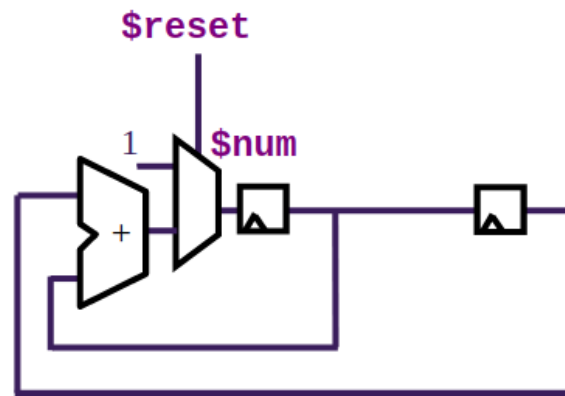


Рисунок 16 – Схема ряда Фибоначчи

Пока подается сигнал `$reset`, в `$num` подается значение 1. Тактовый сигнал продолжает переключаться в течение сброса и значение 1 распространяется через оба триггера, сбрасывая их в состояние 1.

Это обеспечивает первые две 1 в последовательности. После прихода сигнала сброса `$reset`, `$num` принимает значение 2, и схема продолжает генерировать новые значения в последовательности на каждом следующем тактовом сигнале (Рисунок 17).

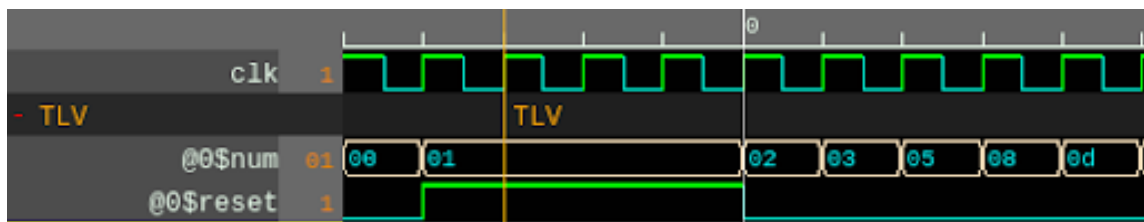


Рисунок 17 – Временная диаграмма схемы ряда Фибоначчи

#### Примечание:

Обычно `$reset` подается в виде обратного сигнала, возможно, с именем `rstn`, то есть сброс происходит, когда `rstn` равен 0, а `rstn` равен 1 во время нормальной работы. Хотя мотивация для этого редко актуальна в современных инструментах логического синтеза, это все равно часто встречается на практике. В примерах будет использоваться сброс по единичному значению сигнала `$reset`.

#### Примечание:

Часто возможность сброса встроена в сам триггер. Здесь рассматривается логика сброса в явном виде. Инструменты логического синтеза могут выбрать для реализации этого поведения стандартный примитив «сброс триггера» («reset flip-flop»).

## Синтаксис TL-Verilog

В TL-Verilog можно сослаться на предыдущую и предпоследнюю версии  $\$num$  как  $>>1\$num$  и  $>>2\$num$ . В отличие от RTL, в TL-проектировании нет необходимости назначать их в явном виде. Они неявно доступны для использования.

Поэтому рассматриваемый пример схемы последовательности Фибоначчи можно задать следующим выражением:

```
 $\$num[31:0] = \$reset ? 1 : (>>1\$num + >>2\$num);$ 
```

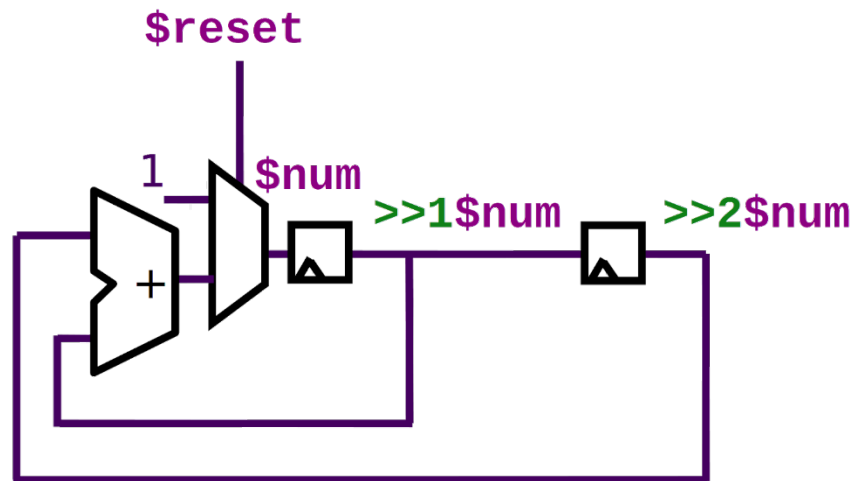


Рисунок 18 – Схема Фибоначчи с поэтапным обращением к сигналам

## Обобщение последовательной логики

Последовательная схема, содержащая триггеры и комбинационную логику, может быть представлена следующим образом:

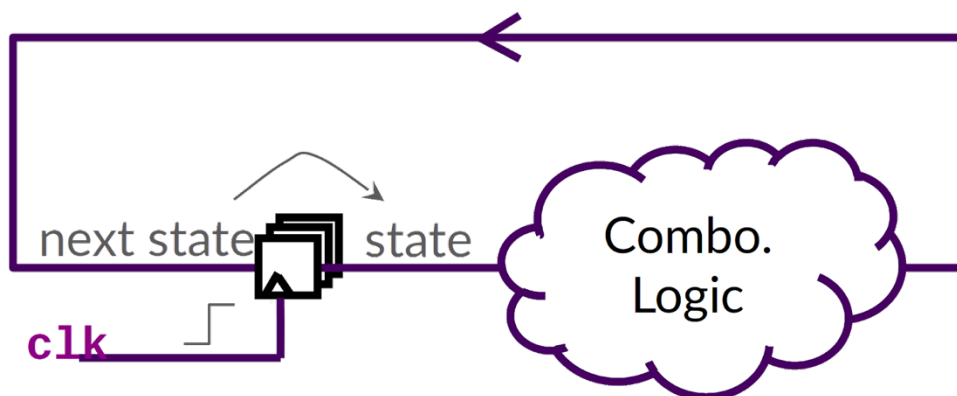


Рисунок 19 – Обобщенный вид последовательной логики

Во время каждого цикла между приходами фронта тактового сигнала комбинационная логика выполняется, а затем приходит тактовый импульс, и следующее состояние становится состоянием схемы, и процесс продолжается.

## Практическая работа: счетчик

По аналогии со схемой последовательности Фибоначчи разработайте 16-разрядный последовательный счетчик, изображенный на рисунке ниже.

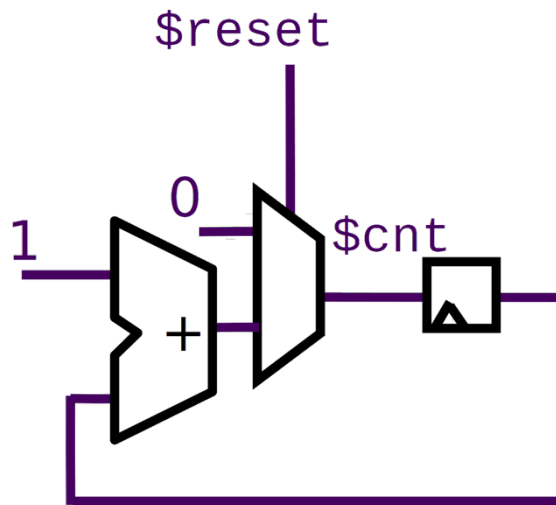


Рисунок 20 – Схема счетчика

\$cnt сбрасывается в ноль (16'b0) и, после сброса, увеличивается на единицу (16'b1) на каждом цикле.

1. Перезагрузите или сбросьте проект в Makerchip, чтобы начать работу с шаблона по умолчанию.
2. Убедитесь, что шаблон по умолчанию выдает сигнал \$reset следующим образом:

```
$reset = *reset;
```

3. Создайте однострочное выражение для этой схемы. Если форма итогового сигнала такая, как показано ниже (Рисунок 21), значит, все сделано правильно. Для справки ниже приведено выражение для схемы Фибоначчи:

```
$num[31:0] = $reset ? 1 : (>>1$num + >>2$num);
```

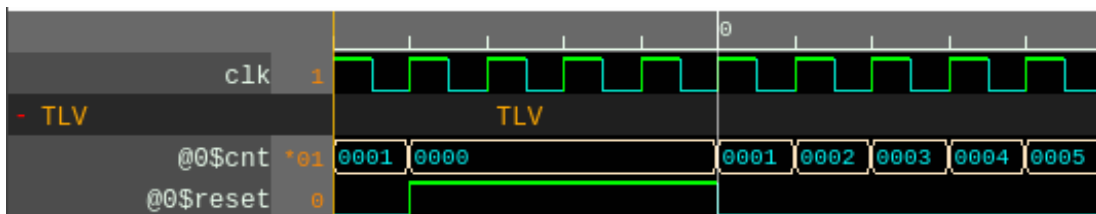


Рисунок 21 – Корректная осциллограмма схемы счетчика

## Демонстрация: счетчик

В случае возникновения трудностей, для вас был подготовлен видеоматериал записи действий [9], которые нужно выполнить для выполнения практической работы.

## Практическая работа: калькулятор с обратной связью

Классический калькулятор отображает результат каждого вычисления. Он сохраняет это значение результата и использует его в качестве первого операнда в следующем вычислении. Если вы введете в калькулятор "+ 3", он прибавит три к предыдущему результату. Давайте доработаем калькулятор, разработанный ранее, чтобы он действовал подобным образом. Каждый цикл будет выполняться новое вычисление, основываясь на предыдущем результате.

Предыдущий результат – это состояние. Какое бы ни было состояние, всегда должен быть `$reset`, который установит это состояние в известное значение. Как и в настоящем калькуляторе, будем сбрасывать значение в ноль.

Для использования по обратной связи результата (`$out`) и сброса его в ноль, нужно реализовать следующую схему:

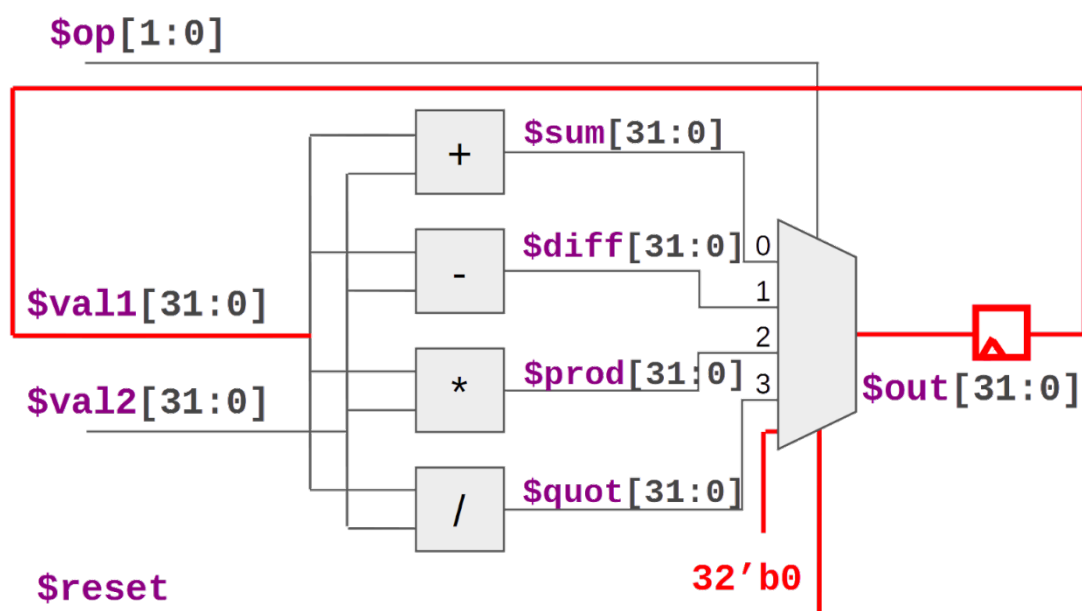


Рисунок 22 – Модификации последовательной работы схемы калькулятора

Выполните следующую последовательность действий:

1. Вернитесь к проекту калькулятора.
2. Присвойте `$val1[31:0]` предыдущему значению сигнала `$out` (изменив его старое присвоение).
3. Добавьте сигнал сброса `$reset` и новый (с наивысшим приоритетом) вход MUX для сброса `$out` в ноль.
4. Визуально убедитесь в правильности работы в VIZ и WAVEFORM. Обратите внимание, что отрицательные значения будут представлены со старшими битами, равными 1 («fff...» в

шестнадцатеричной системе). (Можно отключить вычитание, чтобы предотвратить появление отрицательных значений).

5. Сохраните проект Makerchip.

### **Демонстрация: калькулятор с обратной связью**

В случае возникновения трудностей, для вас был подготовлен видеоматериал записи действий [10], которые нужно выполнить для выполнения этой практической работы.

## Список источников

1. Building a RISC-V CPU Core // GitHub. URL: <https://github.com/stevehoover/LF-Building-a-RISC-V-CPU-Core>
2. Видеоматериал демонстрации выполнения работы «Инвертор». URL: [https://git.miem.hse.ru/mtomarov/RISC-V\\_courses/-/blob/master/RISC-V\\_CPU\\_core\\_building/Chapter\\_2.Digital\\_Logic/Combinational\\_Logic/Demo\\_Inverter.mp4.zip](https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/blob/master/RISC-V_CPU_core_building/Chapter_2.Digital_Logic/Combinational_Logic/Demo_Inverter.mp4.zip)
3. Видеоматериал демонстрации выполнения работы «Логические элементы». URL: [https://git.miem.hse.ru/mtomarov/RISC-V\\_courses/-/blob/master/RISC-V\\_CPU\\_core\\_building/Chapter\\_2.Digital\\_Logic/Combinational\\_Logic/Demo\\_Logic\\_Gates.mp4.zip](https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/blob/master/RISC-V_CPU_core_building/Chapter_2.Digital_Logic/Combinational_Logic/Demo_Logic_Gates.mp4.zip)
4. Видеоматериал демонстрации выполнения работы «Арифметические операторы». URL: [https://git.miem.hse.ru/mtomarov/RISC-V\\_courses/-/blob/master/RISC-V\\_CPU\\_core\\_building/Chapter\\_2.Digital\\_Logic/Arithmetic\\_Logic/Demo\\_Arithmetic\\_Operators.mp4.zip](https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/blob/master/RISC-V_CPU_core_building/Chapter_2.Digital_Logic/Arithmetic_Logic/Demo_Arithmetic_Operators.mp4.zip)
5. Видеоматериал демонстрации выполнения работы «Калькулятор». URL: [https://git.miem.hse.ru/mtomarov/RISC-V\\_courses/-/blob/master/RISC-V\\_CPU\\_core\\_building/Chapter\\_2.Digital\\_Logic/Multiplexers/Demo\\_Calculator.mp4.zip](https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/blob/master/RISC-V_CPU_core_building/Chapter_2.Digital_Logic/Multiplexers/Demo_Calculator.mp4.zip)
6. Видеоматериал демонстрации выполнения работы «Calculator Stimulus». URL: [https://git.miem.hse.ru/mtomarov/RISC-V\\_courses/-/blob/master/RISC-V\\_CPU\\_core\\_building/Chapter\\_2.Digital\\_Logic/Literals\\_%26\\_Concatenation/Demo\\_Calculator\\_Stimulus.mp4.zip](https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/blob/master/RISC-V_CPU_core_building/Chapter_2.Digital_Logic/Literals_%26_Concatenation/Demo_Calculator_Stimulus.mp4.zip)
7. Видеоматериал демонстрации выполнения работы «Визуальная отладка». URL: [https://git.miem.hse.ru/mtomarov/RISC-V\\_courses/-/blob/master/RISC-V\\_CPU\\_core\\_building/Chapter\\_2.Digital\\_Logic/Visual\\_Debug/Demo\\_Visual\\_Debug.mp4.zip](https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/blob/master/RISC-V_CPU_core_building/Chapter_2.Digital_Logic/Visual_Debug/Demo_Visual_Debug.mp4.zip)
8. Видеоматериал «Структура файла и концепция маршрута запуска инструментов». URL: [https://git.miem.hse.ru/mtomarov/RISC-V\\_courses/-/tree/master/RISC-V\\_CPU\\_core\\_building/Chapter\\_2.Digital\\_Logic/File\\_Structure\\_%26\\_Tool\\_Flow](https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/tree/master/RISC-V_CPU_core_building/Chapter_2.Digital_Logic/File_Structure_%26_Tool_Flow)
9. Видеоматериал демонстрации выполнения работы «Счетчик». URL: [https://git.miem.hse.ru/mtomarov/RISC-V\\_courses/-/blob/master/RISC-V\\_CPU\\_core\\_building/Chapter\\_2.Digital\\_Logic/Sequential\\_Logic/Demo\\_Counter.mp4.zip](https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/blob/master/RISC-V_CPU_core_building/Chapter_2.Digital_Logic/Sequential_Logic/Demo_Counter.mp4.zip)
10. Видеоматериал демонстрации выполнения работы «Калькулятор с обратной связью». URL: [https://git.miem.hse.ru/mtomarov/RISC-V\\_courses/-/blob/master/RISC-V\\_CPU\\_core\\_building/Chapter\\_2.Digital\\_Logic/Sequential\\_Logic/Demo\\_Recirculating\\_Calculator.mp4.zip](https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/blob/master/RISC-V_CPU_core_building/Chapter_2.Digital_Logic/Sequential_Logic/Demo_Recirculating_Calculator.mp4.zip)