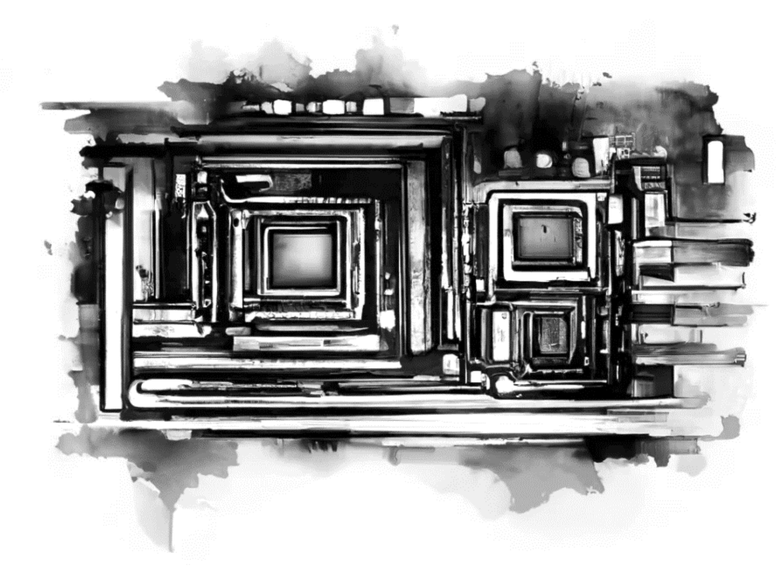
ИТМО

П.С. Скаков, В.Е. Яковлева

АРХИТЕКТУРА ЭВМ: УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ ПО ЛАБОРАТОРНЫМ РАБОТАМ



Санкт-Петербург
2023

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

П.С. Скаков, В.Е. Яковлева

АРХИТЕКТУРА ЭВМ: УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ ПО ЛАБОРАТОРНЫМ РАБОТАМ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

РЕКОМЕНДОВАНО К ИСПОЛЬЗОВАНИЮ В УНИВЕРСИТЕТЕ ИТМО
по направлению подготовки 01.03.02 Прикладная математика и информатика
в качестве Учебно-методическое пособие для реализации основных профессиональных образовательных программ высшего образования бакалавриата

ИТМО

Санкт-Петербург
2023

Скаков П.С., Яковлева В.Е., Архитектура ЭВМ: учебно-методическое пособие по лабораторным работам– СПб: Университет ИТМО, 2023. – 99 с.

Рецензент(ы):
Волынский Максим Александрович, кандидат технических наук, доцент, доцент (квалификационная категория "ординарный доцент") физико-технического мегафакультета, Университета ИТМО.

В пособии приведено описание лабораторных работ по дисциплине «Архитектура ЭВМ», а также методические указания по их выполнению и справочные материалы по рекомендованному инструментарию с примерами кода. Пособие предназначено для студентов бакалавриата, обучающихся по направлению подготовки 01.03.02 «Прикладная математика и информатика».

ИТМО

Университет ИТМО – ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

© Университет ИТМО, 2023
© Скаков П.С., Яковлева В.Е., 2023

Содержание

Введение 6
Инструментальное обеспечение 7
Logisim Evolution 7
Языки программирования..7
Verilog 8
Лабораторная работа №1. Минимизация логических функций......................9
Лабораторная работа №2. Логические схемы на элементах с памятью.........13
Лабораторная работа №3. Описание логических схем на Verilog 17
Лабораторная работа №4. Моделирование работы кэша процессора............20
Справочная информация о Logisim Evolution...31
Интерфейс..31
Создание схем..34
Отладка схем...36
Временная диаграмма 37
Справочная информация о Verilog..39
Инструменты для работы на Verilog 39
Основы Verilog...43
Типы данных...44
Логические значения...44
Сила сигнала 45
Регистровые типы данных 46
Провода (wire) 47
Логические типы данных (logic и bit) 48
Строки...49
Перечисления...49
Векторы...51
Массивы..52
Статические массивы 52
Динамические массивы..53
Другие виды массивов..54
Примитивы..54

Модули 56
Создание экземпляра модуля (инстанцирование) 57
Объявление портов и внутренних переменных 59
Параметры 60
Интерфейс 61
Операторы 61
Присваивание 61
Сравнение 63
Логические и побитовые операторы 65
Сдвиг 65
Тернарный оператор 67
Арифметические операторы 67
Объединение 68
Приоритет исполнения 68
Поведенческое моделирование 69
Поведенческий процедурный оператор initial 70
Поведенческий процедурный оператор always 70
Блочный последовательный оператор begin-end 71
Блочный параллельный оператор fork-join 72
Управление временем 74
Временная задержка (delay) 74
`timescale 76
События изменения уровня сигнала 76
Именованные события (event) 77
Ожидание события (wait) 78
Управляющие конструкции 79
if-else 79
case 79
forever 81
repeat 81
while / do while 82
for 82

foreach....83
function....83
task....84
Системные функции и задачи....85
Пример моделирования схемы....93
Симуляция....96
Модель исполнения....98
Список рекомендуемой литературы....99

Введение

Данное учебное-методическое пособие рекомендуется применять в учебном процессе для контактной работы (лекции, лабораторные работы) и самостоятельной работы студентов (СРС).

Настоящее пособие предназначено для бакалавров, обучающихся по направлению подготовки 01.03.02 «Прикладная математика и информатика» по программам бакалавриата «Компьютерные технологии: Программирование и искусственный интеллект», которая предусматривает освоение дисциплины «Архитектура ЭВМ».

В лабораторном практикуме кратко описаны логические принципы построения цифровых схем вычислительной техники, а также методы описания их на языке описания аппаратуры Verilog HDL.

Обучающимся предлагается самостоятельно спроектировать цифровые схемы как на низком уровне (с использованием транзисторной логики), так и на более высоком уровне – регистровом и уровне логических элементов, а также реализовать и протестировать описанные схемы.

Описание лабораторных работ включает цель работы, задание по работе, описание теоретической и экспериментальной частей работы, сведения по работе с инструментальным программным обеспечением. Общие для ряда работ справочные сведения оформлены отдельным разделом в конце пособия. При выполнении работ рекомендуется ознакомиться с ними.

Работы могут быть выполнены не по порядку, но для лучшего понимания рекомендаций по выполнению и теоретических сведений следует ознакомиться с материалами предыдущих работ.

Контроль знаний предполагает оформление студентом отчёта в соответствии с требованиями, изложенными в Приложении к настоящему пособию, а также его защиту преподавателю. Целью защиты является контроль приобретённых студентом компетенций, предусмотренных направлением подготовки и формируемых в рамках осваиваемой дисциплины (модуля), включая уровень достижения студентом конкретных результатов обучения (умений, навыков). При необходимости, в ходе подготовки к защите отчёта студенту может потребоваться изучение дополнительной литературы, список которой приведён в конце настоящего пособия, а также литературы, рекомендованной преподавателем.

Инструментальное обеспечение

В рамках лабораторного практикума предлагается использовать следующие инструментальные средства:

1. Logisim Evolution для разработки и моделирования логических схем.
2. Icarus Verilog для компиляции и симуляции кода, написанного на языке описания аппаратуры Verilog.
3. IDE для написания кода на одном из разрешённых высокоуровневых языков программирования.

В практикуме предлагается использовать только открытые сервисы и свободно распространяемое ПО. В разделах со справочной информацией приведено подробное описание и полезные сведения по работе с инструментарием.

Logisim Evolution

Logisim представляет собой кроссплатформенный инструмент для разработки и моделирования логических схем с открытым исходным кодом. Благодаря простому интерфейсу Logisim прост в освоении и подходит для начинающих пользователей. Он позволяет создавать схемы, состоящие из различных элементов, которые могут быть соединены между собой проводами.

Начиная с 2011 года разработка Logisim была приостановлена, и вместо него получило развитие его ответвление (fork) – Logisim Evolution.

В связи с этим для выполнения части лабораторных работ будет использоваться именно Logisim Evolution https://github.com/logisim-evolution/logisim-evolution

Для установки требуется скачать инсталляционный файл по ссылке https://github.com/logisim-evolution/logisim-evolution/releases/ и следовать инструкциям по установке.

Описание интерфейса и функционала Logisim Evolution приведено в разделе Справочная информация о Logisim Evolution.

Языки программирования

Часть лабораторной работы №4 заключается в написании кода на языке программирования высокого уровня. На выбор студентам предлагается один из языков из следующего перечня: C, C++, Java, Kotlin, Python.

Студенты не ограничены в выборе среды разработки и исполнения (IDE). Выбранный язык программирования должен быть указан в отчёте к лабораторной работе. Файлы с исходным кодом сдаются студентами на проверку преподавателю вместе с файлом отчёта.

Verilog

Verilog – язык описания аппаратуры (HDL, hardware description language), широко использующийся для моделирования электронных схем. Синтаксис Verilog схож с C, что упрощает его освоение программистами. Verilog имеет препроцессор, очень похожий на препроцессор языка C, основные управляющие конструкции («if», «while», …) также подобны одноимённым конструкциям языка C.

SystemVerilog – является надмножеством Verilog-2005 и позволяет работать на более высоком уровне абстракции, что отвечает сложности современных цифровых систем.

В современных подходах к проектированию аппаратуры проверка модели (верификация) не менее важна, чем её создание и имитационное моделирование (симуляция). Verilog предлагает конструкции, позволяющие отразить инженерный замысел в моделях, программные абстракции, упрощающие разработку тестовых окружений, утверждения, обеспечивающие проверку поведения сложных систем, а также средства измерения функционального покрытия в процессе верификации.

Важнейшим отличием между обычными языками программирования и языками HDL является явное включение концепции времени в языки описания аппаратуры (подробнее в разделе описания симуляции).

Установить Icarus Verilog можно по инструкции с официального сайта https://steveicarus.github.io/iverilog/usage/installation.html в случае установки на Linux и MacOS. Для Windows можно использовать готовые сборки https://bleyer.org/icarus/ .

Для написания кода можно пользоваться любым текстовым редактором.

В рамках выполнения лабораторных работ студентам разрешается использовать любые конструкции языка Verilog (стандарт IEEE-1364), а также конструкции System Verilog, поддерживаемые Icarus 12.0 https://github.com/steveicarus/iverilog .

Лабораторная работа №1. Минимизация логических функций

Цель работы: изучение принципов построение цифровых схем и метода минимизации логических функций (карты Карно).

Инструментарий: работа выполняется с использованием Logisim Evolution.

Порядок выполнения работы:

1. Изучить теоретические основы минимизации логических функций (ЛФ) с использованием карт Карно.
2. Составить таблицу истинности для заданной вектор-функции согласно полученному варианту.
3. Записать ЛФ в СКНФ и СДНФ.
4. Составить схемы для СКНФ и СДНФ в Logisim Evolution в виде отдельных модулей.
5. Провести тестирование созданных в пункте 4 модулей.
6. Минимизировать ЛФ из пункта 2 при помощи таблицы Карно в МКНФ и МДНФ. Нужно построить 2 карты Карно и выделить группы (области/склейки), которые использовались при построении минимизированных функций.
7. Составить схемы для МКНФ и МДНФ в Logisim Evolution в виде отдельных модулей.
8. Провести тестирование созданных в пункте 7 модулей.
9. Составить отчёт по результатам выполнения заданий.

Содержание отчёта

1. Титульный лист.
2. Цель работы.
3. Задания работы.
4. Выданный вариант.
5. Теоретическая часть.
6. Таблица истинности для заданной вектор-функции.
7. Описание построения совершенных форм ЛФ и полученные СКНФ и СДНФ.
8. Логические схемы СКНФ и СДНФ.
9. Карта Карно.
10. Описание построения минимизированных форм ЛФ и полученные МКНФ и МДНФ;
11. Логические схемы МКНФ и МДНФ.

Варианты

Варианты выдаются преподавателем на практическом занятии.

Пример варианта:

Вектор функция в формате $f(x_0, x_1, x_2, x_3) = 1000100100101110$.

Логический базис: НЕ, И, ИЛИ. Элементы логического базиса должны собраны на транзисторах в КМОП-логике.

Теоретические сведения

Логическая функция (ЛФ) – функция, у которой значения переменных (параметров функции) и значение самой функции выражают логическую истинность.

Логическим базисом (ЛБ) называется набор простейших функций, позволяющих реализовать любую другую ЛФ. В качестве логического базиса может выступать {НЕ, И, ИЛИ}, {ИЛИ-НЕ}, {И-НЕ} или другие.

Базовые логические элементы могут быть построены из различных электронных компонентов. Будем рассматривать технологию КМОП, в которой используются полевые транзисторы с изолированным затвором с каналами разной проводимости (n- и p-канальные полевые транзисторы).

Рассмотрим пример (рис. 1) построения простейшего логического элемента НЕ на транзисторах в Logisim. Схема включает PMOS (p1) и NMOS (n1) транзисторы, источник питания (VDD), заземление (VSS) и контакты входа и выхода инвертора (InNOT и OutNOT).

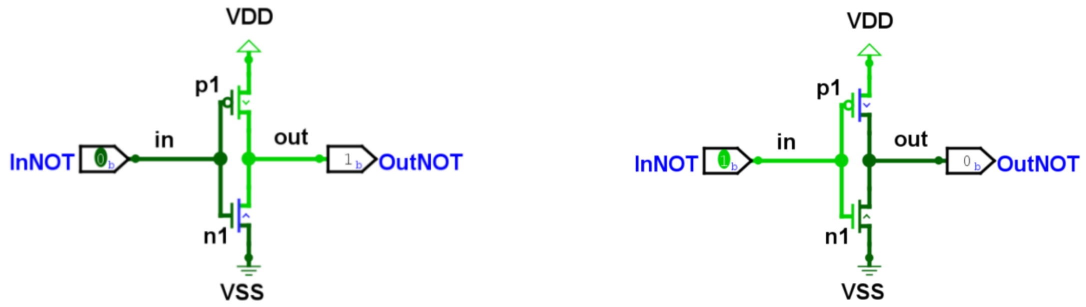


Рис. 1 – Логический элемент НЕ на транзисторах

На рисунках выше показано моделирование работы элемента НЕ. Видно, что в случае подачи на вход 0 открывается p-канальный транзистор p1, а n-канальный транзистор n1 закрыт. Синим на транзисторах обозначается высокоимпедансное состояние. Если подать на вход 1, то открывается нижний транзистор и закрывается верхний. Собранная схема работает корректно, так как, действительно, на выходе получается противоположное входу значение: 1 при 0 на входе и наоборот.

ЛФ задаётся конечным набором значений и может быть представлена в виде *таблицы истинности* или *карты Карно*. Карта Карно может быть построена по таблице истинности и наоборот. Помимо табличного представления ЛФ может быть представлена словесно, математически, схематически, а также на алгоритмическом языке.

На рис. 2 примеры таблицы истинности и карты Карно для логической функции от 4 аргументов, заданной следующей вектор-функцией:

$f(x_0, x_1, x_2, x_3) = 0111\ 0111\ 1000\ 0010$

x_3	x_2	x_1	x_0	f
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**1**
⋮	⋮	⋮	⋮	⋮
1	1	0	1	**0**
1	1	1	0	**1**
1	1	1	1	**0**

Таблица истинности (частично)

f		x_1, x_0			
		00	01	11	10
x_3, x_2	00	**0**	**1**	**1**	**1**
	01	**0**	**1**	**1**	**1**
	11	**0**	**0**	**0**	**1**
	10	**1**	**0**	**0**	**0**

Карта Карно (полностью)

Рис. 2 – Таблица истинности и карта Карно для вектор-функции $f(x_0, x_1, x_2, x_3) = 0111\ 0111\ 1000\ 0010$

Для правильного построения карты Карно должны быть соблюдены следующие требования:

1. В клетках значений (выделено **жирным**) размещаются значения ЛФ для заданного набора аргументов.
2. Число клеток значений в карте совпадает с числом строк в таблице истинности.
3. Порядок размещения ячеек таков, что расстояние Хэмминга аргументов двух соседних ячеек равно 1 (другими словами, в соседних клетках находятся значения, которые различаются ровно одним аргументом функции).

ЛФ может быть записана математически в различных формах:

1. Дизъюнктивная нормальная форма (ДНФ) – дизъюнкция простых конъюнкций.
2. Конъюнктивная нормальная форма (КНФ) – конъюнкция простых дизъюнкций.
3. Алгебраическая нормальная форма (АНФ).

В данной работе на интересуют первые две формы, а точнее их совершенная и минимизированная формы.

Совершенной дизъюнктивной нормальной формой (СДНФ) называется такая ДНФ, у которой в каждую конъюнкцию входят все переменные данного набора, причём в одном и том же порядке. Для её построения достаточно в

таблице истинности функции найти все булевы векторы, на которых её значение равно 1, и для каждого такого вектора построить конъюнкцию. Дизъюнкция этих конъюнкций является СДНФ исходной функции.

Для *совершенной конъюнктивной нормальной формы* (СКНФ) – аналогично.

Полученные совершенные формы (СКНФ и СДНФ) чаще всего могут быть преобразованы в другую форму, содержащую меньшее число переменных и операций с ними по сравнению с исходной – то есть *минимизированы*.

Рассмотрим пример минимизации ЛФ с использованием карты Карно для примера выше. Для построения МДНФ нужно на полученной карте объединить все клетки с 1 в прямоугольники таким образом, чтобы размер прямоугольников равнялся степени двойки по каждому измерению и количество прямоугольников было минимально (следовательно, площадь прямоугольников должна быть как можно больше). Прямоугольники могут выходить с одного края таблицы и продолжаться с обратной стороны, а также пересекаться между собой. После этого для каждого прямоугольника записывается конъюнкция всех аргументов, попавших в этот прямоугольник и не изменяющих своё значение в соседних клетках. Значения входят в конъюнкцию в прямом виде, если их значение в соседних клетках равно 1, иначе – в инверсном. Полученные конъюнкции объединяются по дизъюнкции в искомую ЛФ – МДНФ.

Для МКНФ, аналогично, за тем исключением, что объединяются 0 в прямоугольники дизъюнкции, которые затем объединяются по конъюнкции. На рис. 3 представлено объединение получившихся блоков для заданной картой Карно функцией и получившиеся минимизированные формы. Конъюнкции обозначены произведением (знак умножения опущен), дизъюнкции – сложением.

f		x_1, x_0			
		00	01	11	10
x_3, x_2	00	0	1	1	1
	01	0	1	1	1
	11	0	0	0	1
	10	1	0	0	0

$$f_{МДНФ}(x_3, x_2, x_1, x_0) = \overline{x_3}x_0 + \overline{x_3}x_1 + x_2x_1\overline{x_0} + x_3\overline{x_2x_1x_0}$$

f		x_1, x_0			
		00	01	11	10
x_3, x_2	00	0	1	1	1
	01	0	1	1	1
	11	0	0	0	1
	10	1	0	0	0

$$f_{МКНФ}(x_3, x_2, x_1, x_0) = (x_3 + x_1 + x_0)(\overline{x_2} + x_1 + x_0)(\overline{x_3} + \overline{x_0})(\overline{x_3} + x_2 + \overline{x_1})$$

Рис. 3 – Пример выделения блоков для построения МДНФ и МКНФ и полученные выражения

Лабораторная работа №2. Логические схемы на элементах с памятью

Цель работы: освоение навыка проектирования цифровых схем на элементах с памятью.

Инструментарий: работа выполняется с использованием Logisim Evolution.

Порядок выполнения работы:

1. Изучить теоретические основы работы элементов с памятью.
2. Описать принципы построения схемы 1 согласно варианту.
3. Составить схему 1 в Logisim Evolution в виде отдельного модуля.
4. Провести тестирование модуля из предыдущего пункта.
5. Описать принципы построения схемы 2 согласно варианту.
6. Составить схему 2 в Logisim Evolution в виде отдельного модуля.
7. Провести тестирование модуля из предыдущего пункта.
8. Составить отчёт по результатам выполнения заданий.

Содержание отчёта

1. Титульный лист.
2. Цель работы.
3. Задания работы.
4. Выданный вариант.
5. Теоретическая часть.
6. Описание принципа построения схемы 1.
7. Логические схема 1.
8. Временная диаграмма для схемы 1.
9. Описание принципа построения схемы 2.
10. Логические схема 2.
11. Временная диаграмма для схемы 2.

Варианты

Варианты выдаются преподавателем на практическом занятии.
Пример варианта:
Схема 1: асинхронный вычитающий счётчик; модуль счёта 17.
Схема 2: регистр сдвига; сдвиг влево, битность – 6.

Теоретические сведения

Временные диаграммы представляют собой вид диаграмм, иллюстрирующий изменение сигналов во времени. Временные диаграммы сигналов составляются для всех входов и выходов схемы. Названия сигналов на временной диаграмме должно совпадать с названиями на схеме. На диаграмме могут быть представлены условные обозначения задержек и дополнительная информация, поясняющая работу системы. Примеры временных диаграмм представлены на рис. 4.

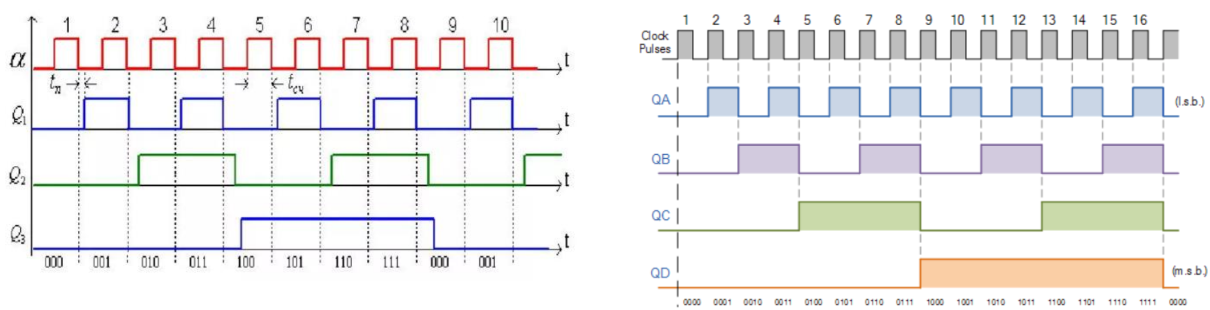


Рис. 4 – Примеры представления временных диаграмм

Счётчиком называют последовательную схему, преобразующую количество импульсов, поступающих на счётный вход, в соответствующий код. Счётчик может иметь дополнительные входы и выходы, например, вход сброса, выход переполнения и т. д.

Важным параметром счётчика является *модуль счёта* – число различных состояний (число импульсов), после которого счётчик возвращается в исходное состояние.

Счётчики выполняются на счётных Т-триггерах.

Счётчики классифицируют по способу счёта на суммирующие, вычитающие и реверсивные; по способу переключения триггеров на асинхронные и синхронные.

В асинхронных (рис. 5) Т-триггеры переключаются последовательно от разряда к разряду – от младших к старшим. В синхронных – одновременное переключение.

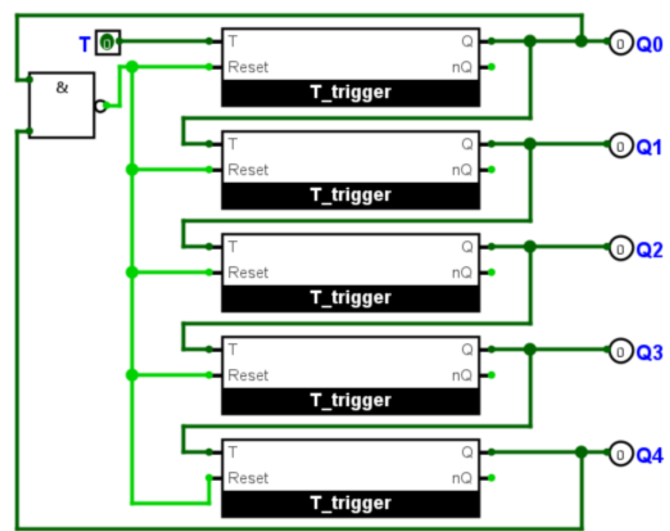


Рис. 5 – Асинхронный суммирующий счётчик по модулю 17

Схема состоит из 5 Т-триггеров, поскольку это минимальное число триггеров, позволяющих хранить максимальное значение 16 (определяется модулем счёта). Логический элемент И слева на схеме добавлен для сброса счётчика в исходное состояние после достижения значения 16.

Временная диаграмма, иллюстрирующая смену состояний триггеров, представлена на рис. 6. На ней видно, что после состояния 16 (двоичный код 10000) произошёл сброс состояния в исходное – 0.

Рис. 6 – Временная диаграмма асинхронного суммирующего счётчика по модулю 17

Регистр сдвига (последовательные регистры) – последовательная схема, состоящая из триггеров, содержимое которых можно сдвигать в определённую сторону с подачей тактовых импульсов. При поступлении сигнала синхронизации, который также называется сигналом или импульсом сдвига, все значения передаются в соседний разряд, а значение со входа данных записывается в освободившийся триггер. В качестве основного элемента используются D-триггеры с динамическим управлением.

Пример регистра сдвига влево с разрядностью 4 и его временная диаграмма представлены на рис. 7 и рис. 8 соответственно. Сдвиг влево означает сдвиг от младших к старшим разрядам.

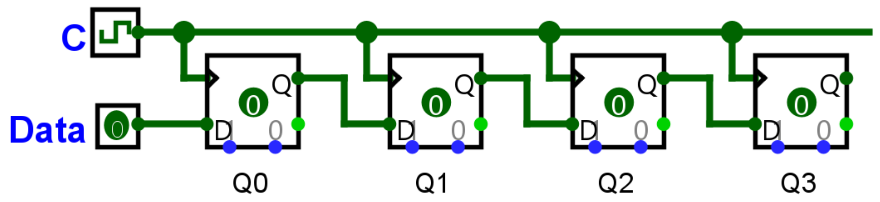


Рис. 7 – Регистр сдвига влево разрядностью 4

Рис. 8 – Временная диаграмма регистра сдвига влево разрядностью 4

Лабораторная работа №3. Описание логических схем на Verilog

Цель работы: освоение навыка описания цифровых схем на Verilog.

Инструментарий: работа выполняется с использованием Icarus Verilog.

Порядок выполнения работы:
1. Изучить основы языка описания аппаратуры Verilog.
2. Описать принципы построения подсхем согласно варианту.
3. Написать код подсхемы на языке Verilog в виде отдельного модуля.
4. Провести тестирование подсхем из предыдущего пункта в отдельном тестовом окружении.
5. Описать принципы построения основной схемы согласно варианту.
6. Написать код основной схемы на языке Verilog в виде отдельного модуля.
7. Провести тестирование схемы из предыдущего пункта.
8. Составить отчёт по результатам выполнения заданий.

Содержание отчёта
1. Титульный лист.
2. Цель работы.
3. Задания работы.
4. Выданный вариант.
5. Теоретическая часть.
6. Описание принципа построения подсхем.
7. Код, описывающий подсхемы.
8. Отчёт о тестировании подсхем с описанием тестового окружения.
9. Описание принципа построения основной схемы.
10. Код, описывающий основную схему.
11. Отчёт о тестировании основной схемы с описанием тестового окружения.

Варианты

Варианты выдаются преподавателем на практическом занятии.

Пример варианта:

Основная схема: $f(x) = \sqrt{x}$*, числа целые беззнаковые 8-битные. Тип округления: к 0.*

Подсхемы: схема взятия корня, компаратор (реализуется на базисе {НЕ, И, ИЛИ}). Все подсхемы должны быть реализованы в виде отдельных модулей на Verilog.

Теоретические сведения

При выполнении операций над числами возникают случаи, когда результат нельзя точно представить, например, при делении или взятии корня. В таком случае необходимо произвести *округление*, для чего может потребоваться вычислить несколько дополнительных разрядов. Выделяют несколько видов округления:

- Округление к $-\infty$ (вниз) – если значение не представимо точно, то кодируется ближайшее представимое, которое меньше этого числа. Обозначается как $\lfloor x \rfloor$ или $floor(x)$.
  Примеры: $1.5 \to 1$, $-1.5 \to -2$.
- Округление к $+\infty$ (вверх) – если значение не представимо точно, то кодируется ближайшее представимое, которое больше этого числа. Обозначается как $\lceil x \rceil$ или $ceil(x)$.
  Примеры: $1.5 \to 2$, $-1.5 \to -1$.
- Округление к 0 – если число положительное – округление проводится по правилу округления вниз, если отрицательное – по правилу округления вверх.
  Примеры: $1.5 \to 1$, $-1.5 \to -1$.
- Округление к ∞ (от 0) – противоположно округлению к 0. Округление происходит к бесконечности того же знака, что и число.
  Примеры: $1.5 \to 2$, $-1.5 \to -2$.
- Округление к ближайшему чётному – округление происходит к ближайшему представимому числу, если таких чисел несколько (то есть значение находится ровно посередине), то выбирается чётное из них (в английском языке известно под названием banker's rounding).
  Примеры: $1.5 \to 2$, $-1.5 \to -2$, $2.5 \to 2$, $-1.4 \to -1$.

Рассмотрим пример *извлечения квадратного корня* из 8-битного беззнакового целого числа с округлением к 0. Очевидно, что целая часть результата уместится в 4 бита.

Одним из самых простых методов, учитывая небольшую битность чисел, может стать *полный перебор* входных и выходных значений: составить 4 булевы функции (по одной для каждого выходного бита), построить их МКНФ и собрать из логических элементов. Построенная схема работает очень быстро. Но из недостатков нельзя не отметить плохую масштабируемость метода: аналогичный подход для 32-битных чисел приведёт к чрезмерно большой схеме. Также есть проблема с проверкой правильности: для этого необходимо перебрать все варианты входных данных.

Для нахождения результата можно пойти и другим путём. Заметим, что вычисление результата в целых числах подразумевает одинаковый ответ для

интервала входных данных. При округлении к 0 в интервал заключены значения от квадрата ответа до числа, предшествующего квадрату ответа, увеличенного на 1, исключительно. То есть каждому значению корня n соответствует полуинтервал $[n^2;\{n+1\}^2)$. Поэтому решение можно свести к *перебору точных квадратов* значений и сравнению с ними входного числа. Когда найдутся два последовательных числа таких, что квадрат меньшего из этих чисел меньше или равен входному числу, а квадрат большего строго больше входного числа, то ответом будет меньшее из найденных чисел. Найти ответ можно, например, линейным сравнением с нижними границами в порядке убывания или верхними в порядке возрастания. К недостаткам такого подхода можно отнести необходимость хранения таблицы значений границ и затраты времени на перебор этих значений.

Существует алгоритм, который позволяет без перебора вычислить значение – извлечение квадратного корня *в столбик*. Блок-схема, описывающая алгоритм, представлена на рис. 9.

Вертикальной чертой обозначения операция конкатенации двух чисел, все числа представлены в двоичной системе счисления. Пример: $x = 101, y = 011, z = x|y|1 = 1010111$.

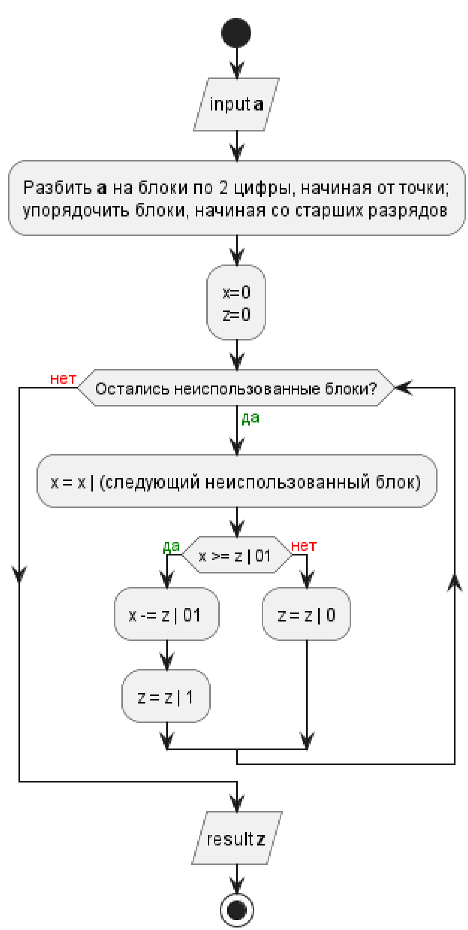


Рис. 9 – Блок-схема алгоритма извлечение квадратного корня в столбик

Лабораторная работа №4. Моделирование работы кэша процессора

Цель работы: изучение работы кэша процессора и развитие навыка проектирования сложных схем на языке описания аппаратуры Verilog.

Инструментарий: работа выполняется с использованием Icarus Verilog, ПО для визуализации временных диаграмм.

Порядок выполнения работы:

1. Изучить теоретические основы работы кэша процессора, системной шины.
2. Вычислить недостающие параметры системы согласно варианту.
3. Решить задачу (раздел Задача) аналитически. Решение может быть представлено в виде кода на языке высокого уровня, эмулирующего работу системы, или вычисления вручную.
4. Провести тестирование представленного аналитического решения.
5. Описать на Verilog модули: кэш, оперативная память, процессор; аргументировать выбранный способ описания системы.
6. Описать на Verilog задачу (раздел Задача) в виде тестового окружения системы.
7. Провести тестирование созданных модулей и системы в целом на примере воспроизведения задачи.
8. Сравнить полученные результаты аналитического решения и моделирования.
9. Составить отчёт по результатам выполнения заданий.

Содержание отчёта

1. Титульный лист.
2. Цель работы.
3. Задания работы.
4. Выданный вариант.
5. Формулировка задачи из условия.
6. Теоретическая часть.
7. Описание вычисления недостающих параметров системы и результат этих вычислений.
8. Аналитическое решение задачи (может быть представлено в виде кода на языке высокого уровня, эмулирующего работу системы, или решения вручную).
9. Моделирование заданной системы на Verilog.
10. Моделирование решения требуемой задачи на Verilog.

11. Сравнение полученных результатов аналитического решения и моделирования.
12. Листинг кода.

Варианты

Варианты выдаются преподавателем на практическом занятии. В качестве варианта выдаются некоторые параметры системы и задача, которую надо сначала решить аналитически на основании параметров системы, а затем промоделировать.

Ниже представлено условие задания для одного из возможных вариантов.

Вариант 1

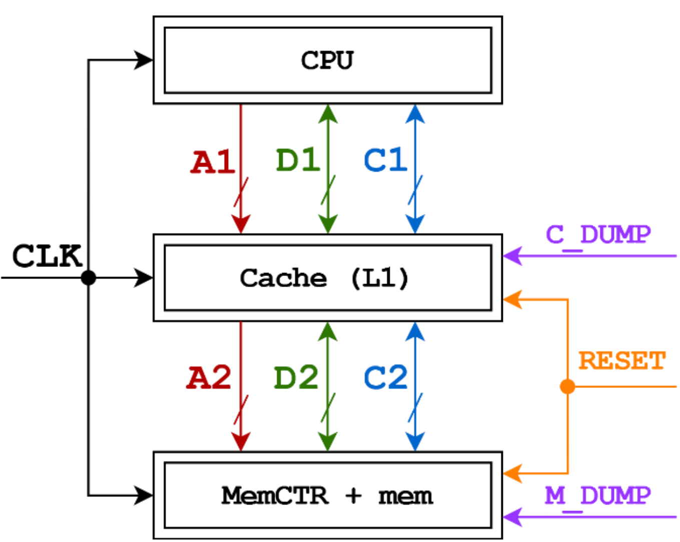


Рис. 10 – Моделируемая система «процессор-кэш-память»

Условные обозначения:

NAME – названия переменных/параметров, которые рекомендуется использовать в работе. Полный список в разделе *Обозначения параметров в коде*.

Сигналы:

- **CLK** – синхронизация всей схемы.
- **RESET** – сброс в начальное состояние.
- ***_DUMP** – сохранение текущего состояния в файл/вывод в консоль для отладки.

Модули:

- CPU – модель процессора для верификации работы кэша
- Cache – одноуровневый кэш

* MemCTR (+ mem) – модель контроллера памяти + модулей памяти для верификации работы кэша

Ниже приведены параметры заданной системы с их условным обозначением. Те параметры, значения которых не указаны явно, должны быть вычислены в пункте *Вычисление недостающих параметров системы*.

Параметры системы и поддерживаемые команды

Команды

Код	Команды CPU → Cache	Ответы CPU ← Cache	Команды Cache → Mem	Ответы Cache ← Mem
0	**C1_NOP**	**C1_NOP**	**C2_NOP**	**C2_NOP**
1	**C1_READ8**			**C2_RESPONSE**
2	**C1_READ16**		**C2_READ_LINE**	
3	**C1_READ32**		**C2_WRITE_LINE**	
4	**C1_INVALIDATE_LINE**			
5	**C1_WRITE8**			
6	**C1_WRITE16**			
7	**C1_WRITE32**	**C1_RESPONSE**		

Число в названиях команд означает количество бит данных, обрабатываемое данной командой. Команды ***_LINE** пишут и читают порциями, равными размеру кэш-линии.

Команда **C1_INVALIDATE_LINE** означает инвалидацию всей кэш-линии, содержащей указанный адрес.

Команды, запрашивающие несколько байт, не могут пересекать кэш-линию.

Команды ***_NOP** – no operation, ***_RESPONSE** – ответ на команду.

Вид и параметры *кэша* представлены в таблице ниже.

Кэш (cache)	
Политика вытеснения	LRU
Ассоциативность	2 – **CACHE_WAY**
Размер тэга адреса	10 бит – **CACHE_TAG_SIZE**
Размер кэша (размер полезных данных)	**CACHE_SIZE**
Размер кэш-линии (размер полезных данных)	16 байта – **CACHE_LINE_SIZE**
Кол-во кэш-линий	64 – **CACHE_LINE_COUNT**
Служебные биты	V (valid), D (dirty)

Память (mem)	
Размер памяти	512 Кбайт – **MEM_SIZE**

Размерность шин

Шина	Обозначение	Размерность
A1, A2	**ADDR1_BUS_SIZE**, **ADDR2_BUS_SIZE**	
D1, D2	**DATA1_BUS_SIZE, DATA2_BUS_SIZE**	16 бит
C1, C2	**CTR1_BUS_SIZE**, **CTR2_BUS_SIZE**	

Время отклика – расстояние в тактах от первого такта команды до первого такта ответа:

- 6 тактов – время, через которое в результате кэш попадания, кэш начинает отвечать.
- 4 такта – время, через которое в результате кэш промаха, кэш посылает запрос к памяти.
- 100 тактов – время ответа контроллера оперативной памяти.

Протокол обмена данными по шине

Команды и ответы на них передаются по шинам за несколько тактов подряд. Но между командой и ответом может быть произвольное количество тактов бездействия.

По шине A1 адрес передаётся за 2 такта: в первый такт tag+set, во второй – offset. По шине A2 передаются адреса без части offset за 1 такт.

По шинам D (D1 и D2) в каждый такт передаётся по 16 бит данных, начиная с младших, little endian (LE).

На линиях команд C (C1 и C2) значение держится всё время передачи команды или ответа.

В начальный момент времени (или после Reset) шиной 1 владеет CPU, а шиной 2 – Cache. После подачи команды и до окончания отправки ответа владение шиной переходит к Cache и MemCTR соответственно. Владение шиной определяет устройство, которое задаёт логические уровни на проводах шины.

Обозначения параметров в коде

В коде *команды* могут указываться как числом с комментарием, так и значением из перечисления (enum):

```
C1_NOP             C1_READ8    C1_READ16     C1_READ32
C1_INVALIDATE_LINE C1_WRITE8   C1_WRITE16    C1_WRITE32
C1_RESPONSE        C2_NOP      C2_READ_LINE  C2_WRITE_LINE
C2_RESPONSE
```

Переменные, параметры, константы:

MEM_SIZE – размер памяти	**CACHE_OFFSET_SIZE** – размер смещения
CACHE_SIZE – размер кэша	**CACHE_TAG_SIZE** – размер тэга адреса
CACHE_LINE_SIZE – размер кэш-линии	**CACHE_SET_SIZE** – размер индекса в наборе кэш-линий
CACHE_LINE_COUNT – кол-во кэш-линий	**CACHE_SETS_COUNT** – кол-во наборов кэш-линий
CACHE_WAY – ассоциативность	**CACHE_ADDR_SIZE** – размер адреса

Детали реализации

Можно пользоваться любыми конструкциями и переменными языка описания Verilog и SystemVerilog.

Инициализация по умолчанию (также по сигналу **RESET**):

- все кэш-линии в состоянии invalid;
- в памяти данные инициализируются по следующему правилу:

```
module test #(parameter _SEED);
  integer SEED = _SEED;
  reg[7:0] a[0:`MEM_SIZE];
  integer i = 0;
  initial begin
    for (i = 0; i < `MEM_SIZE; i += 1) begin
      a[i] = $random(SEED)>>16;
    end

    for (i = 0; i < 100; i += 1) begin
      $display("[%d] %d", i, a[i]);
    end

    $finish;
  end
endmodule
```

Алгоритм инициализации (включая SEED) запрещается менять. Значение SEED задаётся в выданном варианте.

Задача

Имеется следующее определение глобальных переменных и функций:

```
#define M 64
#define N 60
#define K 32
int8 a[M][K];
int16 b[K][N];
int32 c[M][N];

void mmul(void) {
  int8 *pa = a;
```

```
  int32 *pc = c;
  for (int y = 0; y < M; y++) {
    for (int x = 0; x < N; x++) {
      int16 *pb = b;
      int32 s = 0;
      for (int k = 0; k < K; k++) {
        s += pa[k] * pb[x];
        pb += N;
      }
      pc[x] = s;
    }
    pa += K;
    pc += N;
  }
}
```

Сложение, инициализация переменных и переход на новую итерацию цикла, выход из функции занимают 1 такт, умножение – 5 тактов. Обращение к памяти вида pc[x] считается за одну команду.

Массивы последовательно хранятся в памяти, и первый из них начинается с адреса 0.

Все локальные переменные лежат в регистрах процессора.

По моделируемой шине происходит только обмен данными программы (не кодами команд).

Определите процент попаданий (отношение числа попаданий к общему числу обращений) для кэша и общее время (в тактах), затраченное на выполнение этой функции.

Теоретические сведения

Кэш – небольшая быстрая память, расположенная близко к процессору, чаще всего на одном с ним кристалле. Кэш позволяет автоматически (без специальных действий по стороны программиста) ускорить выполнение алгоритмов, обладающих свойством пространственно-временной локальности: обращение к некоторой порции данных означает высокую вероятность обращения к ней же или соседним данным в близкий момент времени. Соответственно, если алгоритм не обладает этим свойством (например, обращается к памяти весьма случайно), то кэш не даст ускорения.

Кэш копирует оперативную память не отдельными байтами, а областями, называемыми кэш-линиями. Если в случае обращения процессора к данным они оказываются в кэше, то это называется *кэш-попаданием*, если же их не оказывается в кэше, то это называется *кэш-промахом*. В случае кэш-попадания запрос процессора в память обрабатывается значительно быстрее (т.к. кэш расположен гораздо ближе к процессору, чем оперативная память), чем в случае кэш-промаха.

При доступе процессора в память сначала производится проверка, хранит ли кэш запрашиваемые данные. Если данные хранятся в кэше, то происходит *кэш-попадание*, и запрос дальше не отправляется. Отношение количества попаданий в кэш к общему количеству запросов к памяти называют процентом попаданий (hit rate) и представляет собой одну из метрик эффективности кэша для выбранного алгоритма или программы.

В случае промаха в кэше выделяется новая запись, в тег которой записывается адрес текущего запроса, а в саму кэш-линию – данные из памяти после их прочтения либо данные для записи в память. Для добавления данных в кэш после кэш-промаха может потребоваться вытеснение ранее записанных данных, определяемое *политикой вытеснения*.

В современных архитектурах кэш делится на несколько *уровней*, чаще всего 3. Пример многоуровневой организации на примере трёхуровневого кэша представлен на рис. 11.

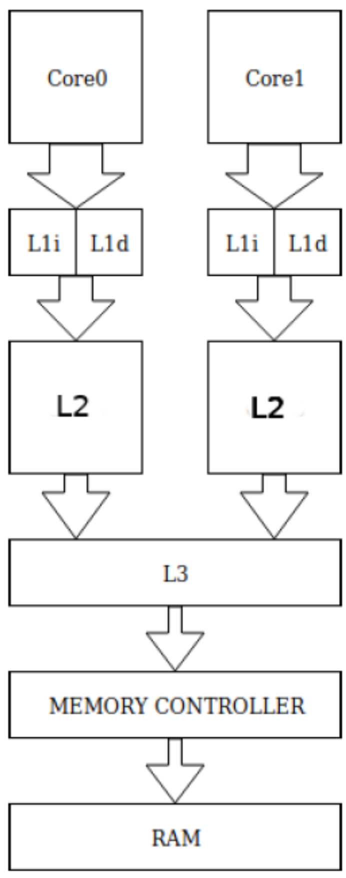


Рис. 11 – Пример трёхуровневой организации кэша

В рамках текущей работы кэш будет одноуровневым L1D кэшем.

Структура записи в кэше

valid	dirty	tag	data
1	1	CACHE_TAG_SIZE	CACHE_LINE_SIZE

Рис. 12 – Типичная структура записи в кэше

Блок данных (кэш-линия, data) содержит непосредственную копию данных из основной памяти.

Для хранения дополнительной информации о состоянии и актуальности данных используют служебные биты. На рисунке выше представлены служебные биты valid и dirty. Если valid установлен в 0, то данная кэш-линия свободна и состояние остальных битов не важно. Флаг dirty означает, что кэш-линия хранит изменённые данные, которые ещё не записаны в память.

Интерпретация адреса кэшем происходит по шаблону, представленному на рис. 13.

tag	set	offset
CACHE_TAG_SIZE	CACHE_SET_SIZE	CACHE_OFFSET_SIZE

Рис. 13 – Паттерн интерпретации адреса кэшем

Адрес памяти разделяется (от старших бит к младшим) на тег (tag), индекс (set) и смещение (offset). Смещение задаётся числом бит, необходимых для хранения смещения внутри кэш-линии и равно $\log_2 \mathbf{CACHE_LINE_SIZE}$. Индекс определяет, в каком месте блока размером ассоциативностью **CACHE_WAY** будет закодированная данная кэш-линия.

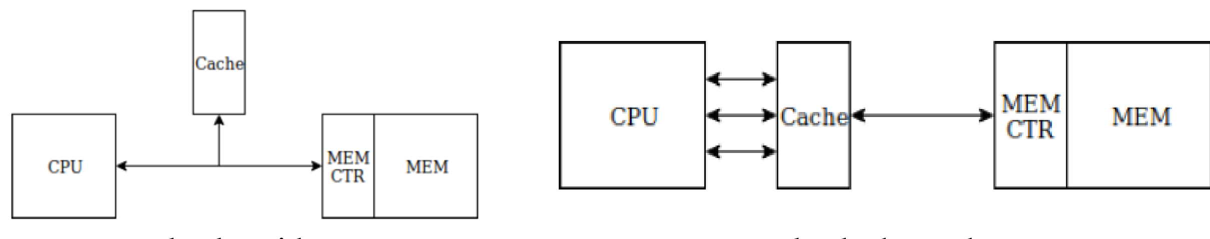
Ассоциативность

В связи с необходимостью обеспечения максимально быстрого поиска данных в кэше обычно некоторая область оперативной памяти может быть закэширована только в некотором подмножестве всех доступных кэш-линий. Количество кэш-линий в доступном наборе называется ассоциативностью. Низкая ассоциативность увеличивает скорость поиска, но уменьшает эффективность использования объёма кэша.

	кэш прямого отображения (direct mapped cache) – наилучшее время поиска и, соответственно, хороший вариант для больших кэшей
	n-канальный множественно-ассоциативный кэш (n-way set associative cache), n обычно принимает значение, равное степени двойки от 2 до 16
	полностью ассоциативный кэш (fully associative cache) – наилучшее использование объёма кэша, самый низкий процент промахов (miss rate), лучший вариант при чрезвычайно высоких затратах при промахе (miss penalty)

Политики чтения

Кэши по архитектуре и связью с другими компонентами системы с точки зрения чтения данных можно разделить на *look-aside* и *look-through* (рис. 14).



look-aside look-through

Рис. 14 – Иллюстрация архитектуры систем при разных политиках чтения

В первом случае (look-aside) процессор (CPU), кэш (Cache) и память (Memory) общаются по общей шине. Когда кэш видит запрос, на который может ответить, то он отвечает по этой шине и уведомляет об этом память, иначе – ничего не делает. Сложность такой архитектуры заключается в организации общения по общей шине, которой чаще всего занимается дополнительный модуль, называемый арбитром (arbiter). Этот модуль также «прослушивает» шину и в случае, если несколько устройств готовы одновременно по шине что-то отправить, решает возникающие конфликты.

При организации второго вида (look-through) вместо одной шины имеются две шины: CPU-Cache и Cache-Memory. Процессор отправляет запрос на чтение в кэш. Если у кэша есть нужная линия, он отвечает. Если нет, то запрашивает линию у основной памяти, а её ответ сохраняет себе. В таком случае не нужно продумывать дополнительный арбитраж, а также можно позволить сделать более широкую шину CPU-Cache, поскольку они находятся на одном кристалле. В случае постоянных кэш-промахов данный вариант работает медленнее, чем look-aside, т.к. происходит дополнительная задержка перед запросом к контроллеру память (MemCTR).

Политики записи

Политика записи чаще всего связана с политикой чтения. При реализации look-through целесообразно применить политику записи *write-back*, заключающуюся в том, что запись данных, поступающих от процессора, осуществляется только в кэше, а в память они попадают при вытеснении. Для этого необходимо хранить служебные биты, рассмотренные ранее.

Если же используется *write-through*, то запись данных происходит одновременно и в кэш, и в память. В таком случае не нужно хранить дополнительные биты, и такое общение проще реализовать в случае, когда имеется одна общая шина (look-aside организация). Из недостатков такого подхода можно выделить то, что очередь запросов к памяти быстро заполняется чаще всего бесполезными запросами, например, на запись данных в одну и ту же ячейку, и при достижении лимита запросов в этой очереди работа сильно замедляется.

Помимо рассмотренных политик также выделяют политики записи при кэш-промахе: *write allocate* (fetch on write) и *write-no-allocate* (write around). При *write allocate* – нужная кэш-линия сначала загружается в кэш, а потом в неё происходит запись. При *write-no-allocate* – данные сразу записывается в память, минуя кэш. Чаще всего применяются комбинации *write-back+write allocate* и *write-through+write-no-allocate.*

Политики вытеснения

В случае, если необходимо выделить кэш-линию, а список свободных кэш-линий, куда может быть записана полученная порция данных, пуст, применяется политика вытеснения:

- LRU (Least-recently used) – вытесняется линия, не использованная дольше всех.
- MRU (Most Recently Used) – вытесняется последняя использованная линия.
- FIFO (First In – First Out) – вытесняется самая давно прочитанная.
- Random – вытесняется случайная линия.

Справочная информация о Logisim Evolution

В этом разделе приведено краткое описание интерфейса Logisim Evolution и пример разработки, моделирования и отладки простой схемы. Описание установки можно посмотреть в начале данного пособия. Далее название Logisim Evolution будет сокращено до Logisim.

Logisim работает с *.circ файлами. В этих файлах проектов Logisim хранятся данные о собранной схеме и её подсхемах. При установке Logisim на компьютеры под ОС Windows можно столкнуться с проблемой, что все файлы, скачиваемые из Интернета, загружаются с расширением .circ. Это можно исправить, удалив в ключе реестра `HKEY_CLASSES_ROOT\MIME\Database\Content Type\application/octet-stream` параметр `Extension`.

Более подробную информацию о работы Logisim и встроенной библиотеке можно найти в документации на официальном сайте: http://www.cburch.com/logisim/docs/2.7/ru/html/guide/index.html.

Интерфейс

Интерфейс Logisim можно условно поделить на 5 частей, представленных на рис. 15.

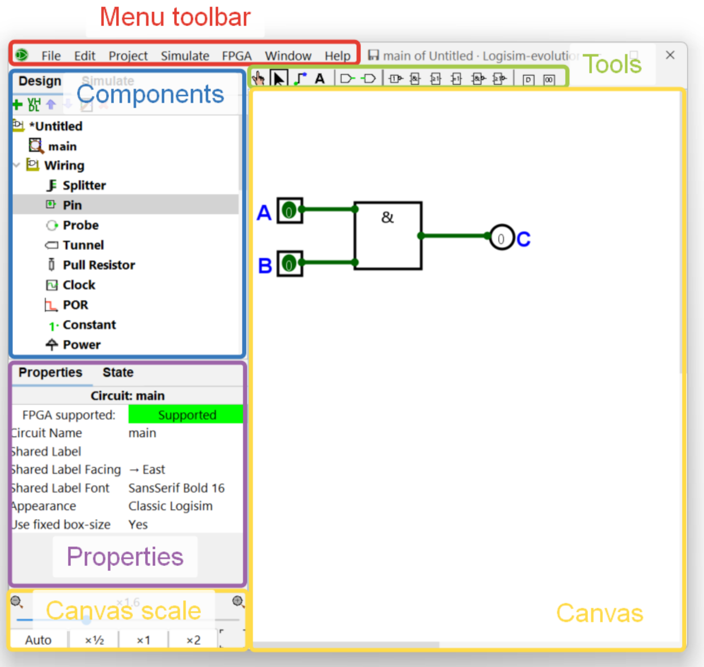


Рис. 15 – Интерфейс Logisim

Menu toolbar содержит разделы для работы с файлом проекта, настройками и запуском симуляции, а также help.

Canvas представляет собой холст (канву), на которой размещаются все компоненты и провода схемы.

Tools содержит инструменты для работы с объектами на канве:

	Нажатие. Позволяет по нажатию на провода проверить их состояние и по нажатию на компоненты посмотреть или сменить их состояние.
	Правка. Даёт возможность выбирать и перемещать объекты на канве, а также взаимодействовать с проводами.
	Добавление провода на канву. Провод может быть как присоединён к входам и/или выходам элементов на канве, так и «висеть в воздухе».
A	Добавление текста на канву.
	Добавление элемента ввода значение (слева) и вывода (справа) значения.
	Добавление логических элементов. Порядок слева направо: НЕ, И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ, И-НЕ, ИЛИ-НЕ, D-триггер, регистр.

Components позволяет выбрать из библиотеки готовых элементов, как базовых логических, так и сложных готовых схем. Все схемы разделены на группы. Встроенные библиотеки элементов Logisim содержат базовые логические элементы, комбинационные элементы (мультиплексоры, декодеры), готовые компоненты, выполняющие арифметические действия (сумматор, вычитатель), компоненты, хранящие данные (триггеры, регистры, и ОЗУ), компоненты для взаимодействия с проводами и пользовательским интерфейсом взаимодействия.

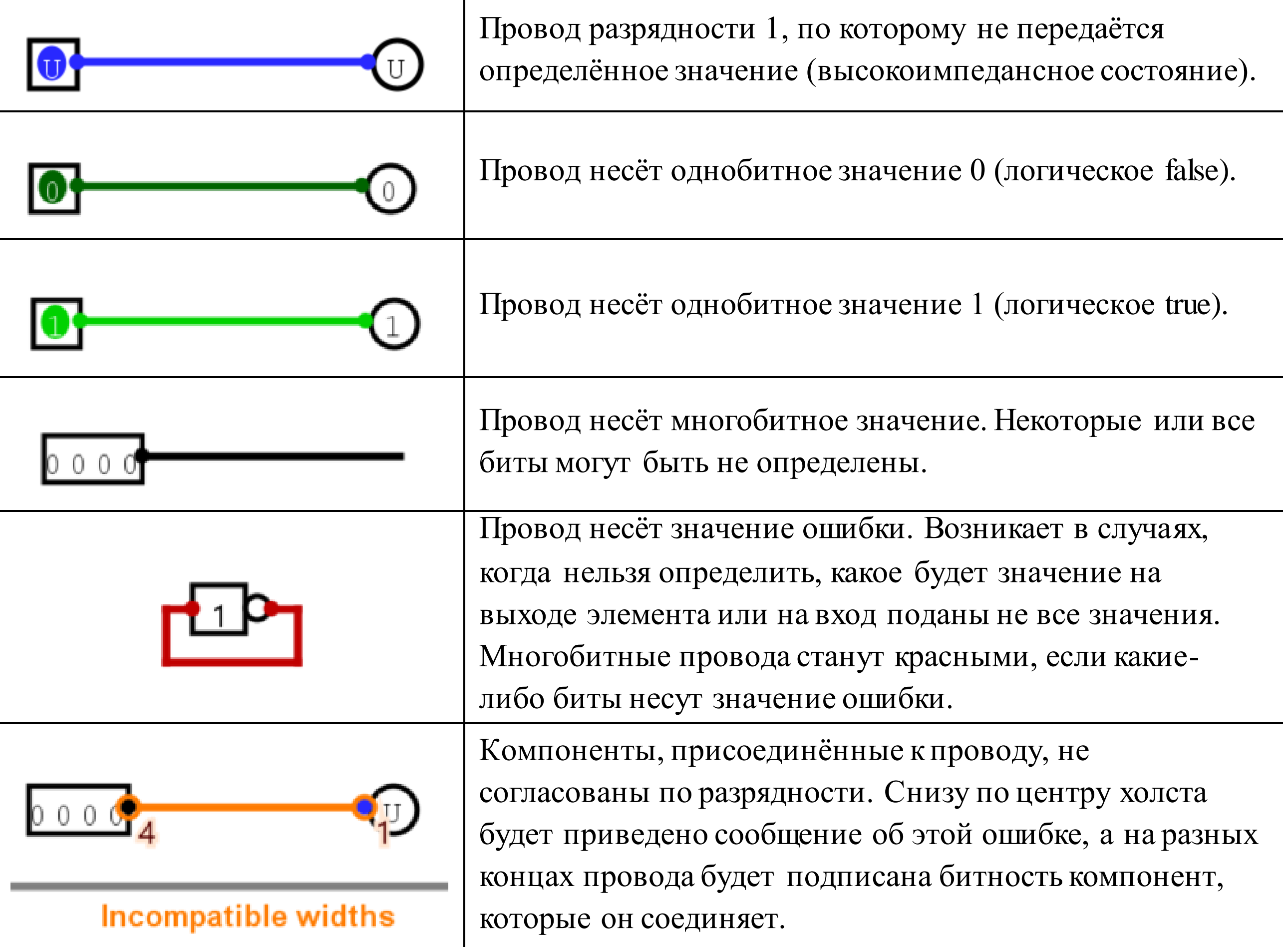
Properties

Раздел, позволяющий настроить выбранный в данный момент компонент на холсте.

Оформление проводов

Провода, размещённые на холсте, могут быть одного из нескольких цветов. Полный список цветов с их интерпретацией приведён ниже.

	Провод ни к чему не подключен, разрядность не известна.
	Соединение двух проводов. Если провода проходят по канве через одно место и не имеют кружка, значит они не соединены.



	Провод разрядности 1, по которому не передаётся определённое значение (высокоимпедансное состояние).
	Провод несёт однобитное значение 0 (логическое false).
	Провод несёт однобитное значение 1 (логическое true).
	Провод несёт многобитное значение. Некоторые или все биты могут быть не определены.
	Провод несёт значение ошибки. Возникает в случаях, когда нельзя определить, какое будет значение на выходе элемента или на вход поданы не все значения. Многобитные провода станут красными, если какие-либо биты несут значение ошибки.
Incompatible widths	Компоненты, присоединённые к проводу, не согласованы по разрядности. Снизу по центру холста будет приведено сообщение об этой ошибке, а на разных концах провода будет подписана битность компонент, которые он соединяет.

Настройки

Во вкладке International (рис. 16) можно выбрать язык локализации и вид логических элементов: IEC (европейский) и ANSI (американский). На всех схемах курса используется вид IEC.

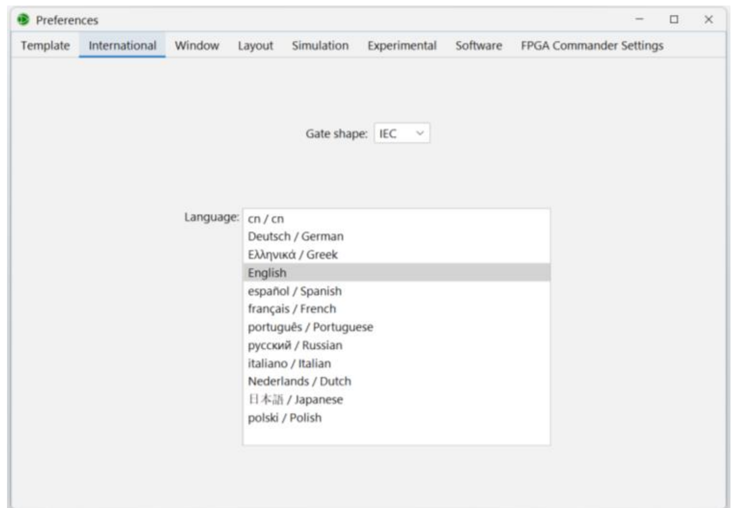


Рис. 16 – Раздел Настройки локализации Logisim

Создание схем

Все схемы создаются из других схем в проекте. Изначально при создании проекта в списке схем отображается единственная схема «main». Добавить дополнительные схемы в проект можно по нажатию на «Добавить схему» в меню Проект.

Рассмотрим пример создания схемы в проекте. Создадим схему mux и соберём на холсте этой схемы мультиплексор 2-в-1. Значком лупы в панели схем отображается отображаемая на холсте схема. Пример такой схемы приведён на рис. 17.

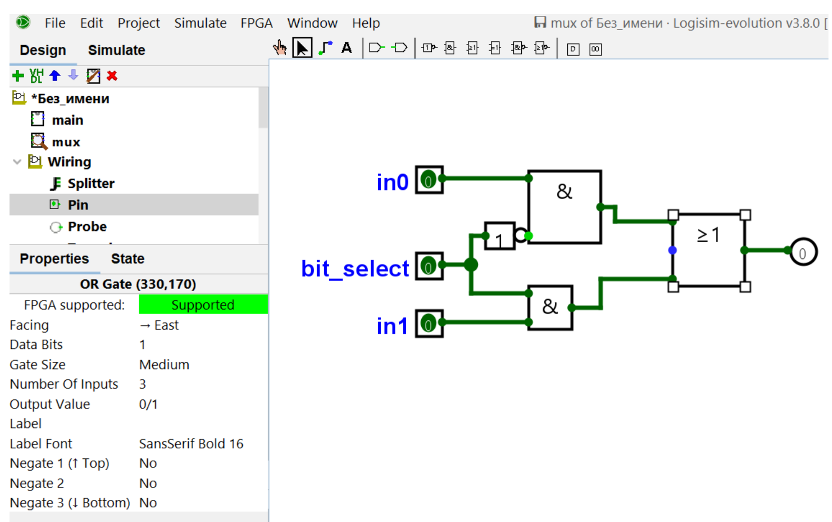


Рис. 17 – Схема мультиплексора 2-в-1

На приведённой схеме размещены элементы с разными свойствами: размером (gate size), числом входов (number of inputs), названием (label), ориентацией (facing). Так, например, логический элемент И (&) приведён в двух размерах, а у элемента логического ИЛИ (>=1, |) задано три входа, хотя по факту используются только два и значение на оставшемся входе неопределенно и не влияет на выходное значение элемента.

Теперь попробуем использовать собранную схему мультиплексора 2-в-1 – *подсхему* в терминах Logisim – для сборки схемы мультиплексора 4-в-1. Для этого добавим новую схему с названием mux4:1 и по нажатию ЛКМ на ранее собранную схему mux из списка схем и перетаскиванию на холст добавим несколько мультиплексоров. Результат сборки представлен ниже. На этой схеме входы обозначены как in_*, а набор из битов выбора задан многобитовым входом bit_select.

Можно заметить, что в стандартном отображении на подсхемах не видно, какой вход за что отвечает. При подключении таких подсхем проводами можно посмотреть названием входа путём наведения на него. Если же изначально при создании подсхемы входы и выходы не были подписаны, то при наведении на

них на подсхеме ничего выведено не будет. Поэтому важно всегда подписывать входы и выходы схемы (рис. 18).

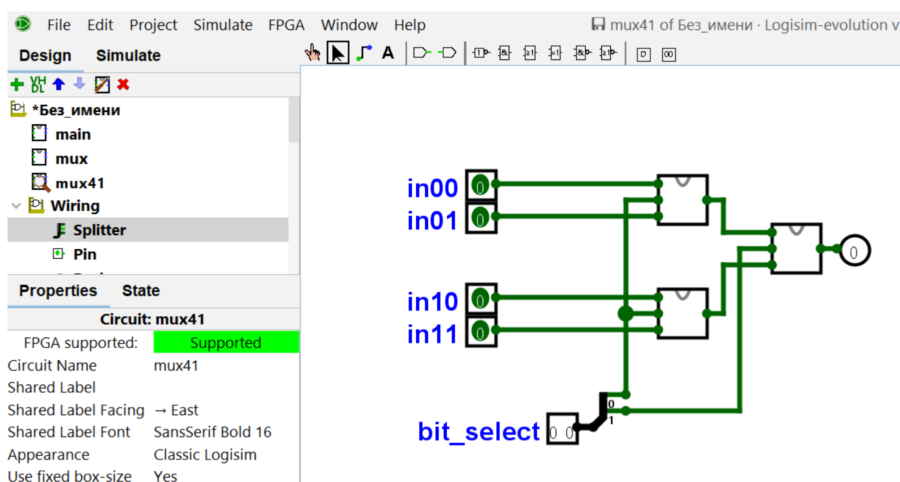
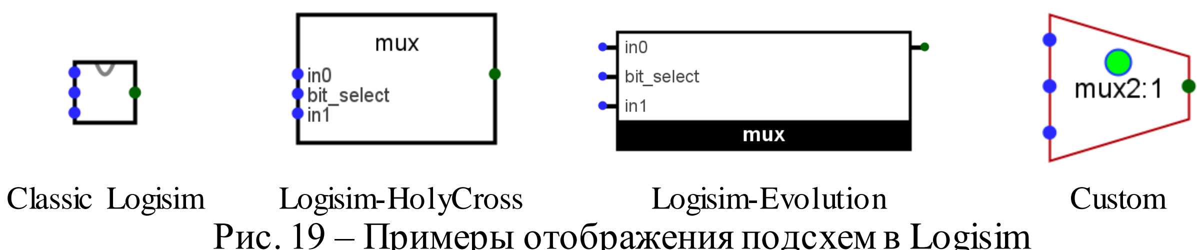


Рис. 18 – Схема мультиплексора 4-в-1 с подписанными входами и выходами

Порядок контактов на западной стороне «коробки» подсхемы соответствует их порядку сверху вниз на чертеже подсхемы. Если бы было несколько контактов на северной или южной стороне «коробки», то их порядок соответствовал бы порядку слева направо в подсхеме.

Вложений подсхем в схемы неограниченно, однако рекурсивное вложение подсхем самих в себя запрещено в Logisim. Также в программе наложено ограничение на тип контактов, которые используются как входы и выходы схем – они могут либо входами, либо выходами; в реальных схемах встречаются входо-выходы.

Поскольку при создании крупных схем с большим количеством разнообразных подсхем проверять, какая «коробка» какой подсхемой является, достаточно трудоёмко и неудобно. По этому поводу в Logisim можно настроить внешний вид схемы. Для этого нужно выбрать «Редактировать внешний вид схемы» из меню Проект, и Logisim переключится с привычного интерфейса редактирования чертежа на интерфейс для рисования внешнего вида схемы. Или же можно использовать один из автоматически генерируемых Logisim вариантов. Примеры отображения подсхемы mux в различных вариантах (appearance) и в созданном вручную (custom) представлен на рис. 19.



Classic Logisim Logisim-HolyCross Logisim-Evolution Custom

Рис. 19 – Примеры отображения подсхем в Logisim

Режим редактирования внешнего вида схемы позволяет по наведению на порты (входы, выходы) просматривать, где он находится на основной схеме, как показано на рис. 20.

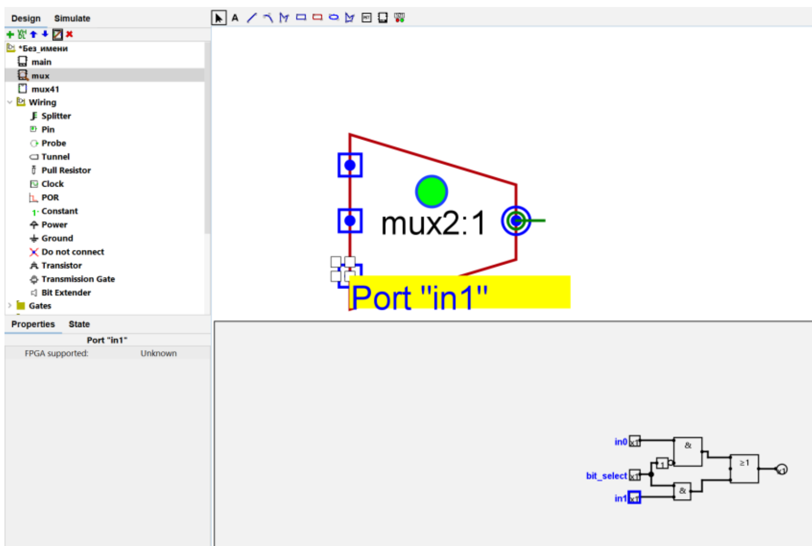


Рис. 20 – Режим редактирования внешнего вида

Отладка схем

После того, как схема собрана, нужно проверить её, чтобы она работала верно. Для включения режима моделирования нужно выбрать в меню Моделирование «Моделирование включено» (Auto-propagate enable).

После этого можно изменять значения на контактах при помощи инструмента Нажатие и проверять значение на выходе. Рассмотрим mux (рис. 21).

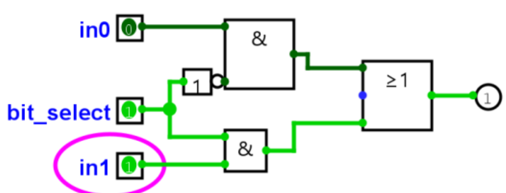
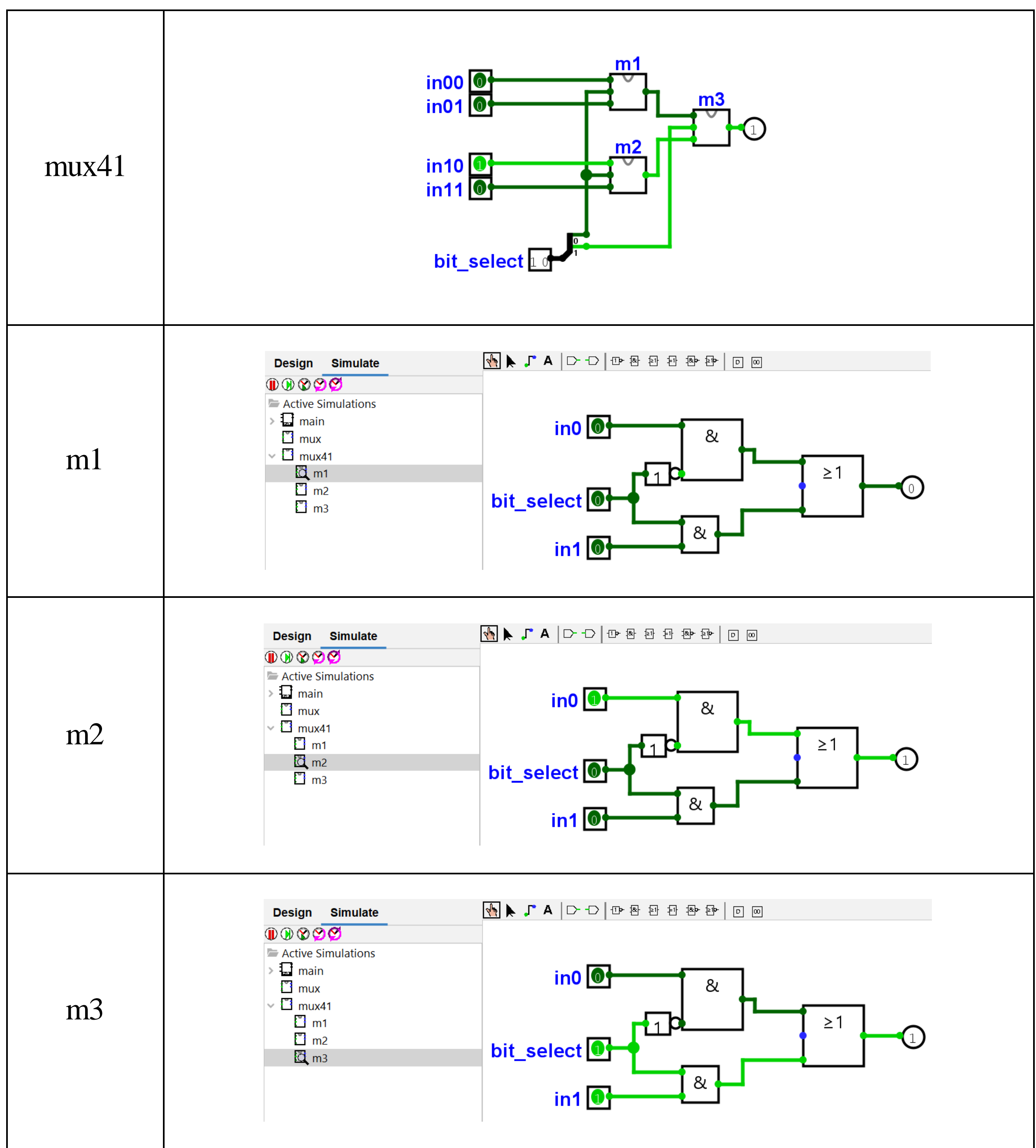


Рис. 21 – Режим отладки на примере мультиплексора 2-в-1

На рисунке выше были переключены входы bit_select и in1 в состояние 1. Текущий нажатый элемент выделяется розовым овалом, и в меню свойств отображаются его свойства.

В случае тестирования схемы, состоящей из пользовательских подсхем, например, mux41, при моделировании можно просматривать как состояние схемы в целом, так и каждой из её подсхем. Ниже приведён пример, когда отображается вид всей схемы и каждой подсхемы.



Временная диаграмма

Дополнительным способом визуализации работы схемы является временная диаграмма (рис. 22). Она отлично подходит для визуализации изменения сигнала во времени. Меню работы с временными диаграммами можно найти в меню Моделирование – Запись в журнал (Simulate – Timing diagram).

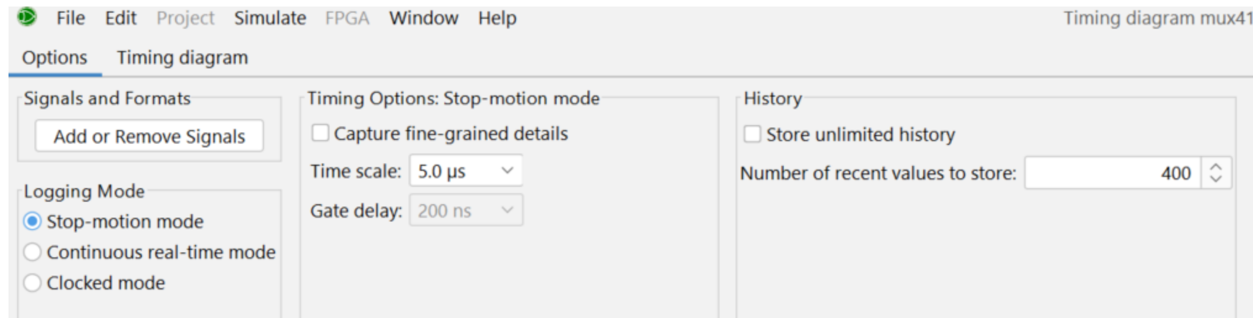


Рис. 22 – Меню временной диаграммы в Logisim

Меню настроек позволяет выбрать режим создания диаграммы: stop-motion mode (запись значений с определённым интервалом), continuous real-time mode (запись с учётом пройденного времени между изменениями сигналов), clocked mode (запись с учётом тактового сигнала, дополнительно требуется указать, какой элемент будет такой сигнал генерировать).

Помимо этого, можно настроить, какие сигналы будут отслеживаться для построения временной диаграммы, чему будет равна 1 секунда реального времени на временной диаграмме и сколько значений на диаграмме будет записано. На рис. 23 приведён пример временной диаграммы для схемы mux41, записанной в режиме stop-motion mode. Во время записи были перебраны все комбинации входов и какие значения получены на выходе схемы, который на диаграмме называется Output (на самой схеме он не имеет метки).

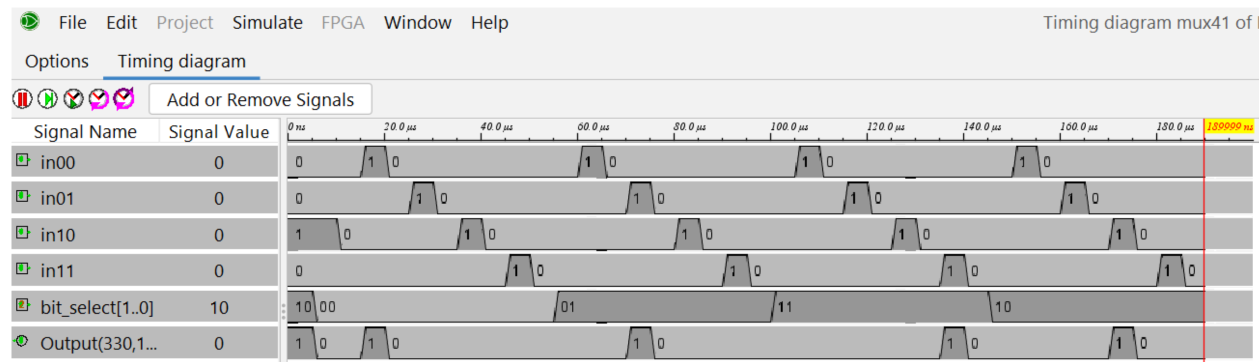


Рис. 23 – Пример временной диаграммы

Справочная информация о Verilog

В этом разделе представлено справочное руководство о большинстве элементов, доступных в Verilog и SystemVerilog. Все подразделы содержат один или несколько примеров и ссылки на другие темы, связанные с текущей.

Синтаксические конструкции, примеры их использования и результат вывода кода будут приведены в каждом подразделе. Большинство примеров кода будут представлены в виде набора оформленных модулей, которые можно при желании скомпилировать и провести симуляцию.

Также дополнительно будут рассмотрены открытые сервисы и онлайн среды для написания кода на Verilog и SystemVerilog.

Инструменты для работы на Verilog

Для написания и отладки схем (скриптов модулей) необходимы:

1. Текстовый редактор для написания кода
2. Компилятор
3. Тестовое окружение и симулятор
4. Просмотрщик файлов с временными диаграммами

Online инструменты

1. EDA Playground (https://www.edaplayground.com/)

На сайте в файлах ***testbench.sv*** и ***design.sv*** пишется код тестового окружения и самой схемы соответственно. Для выполнения заданий предлагается выбирать в «Testbench+Design» SystemVerilog/Verilog и в «Tools & Simulators» бесплатный симулятор Icarus Verilog 0.10.0 11/23/14. На рис. 24 приведён скриншот с настроенными параметрами.

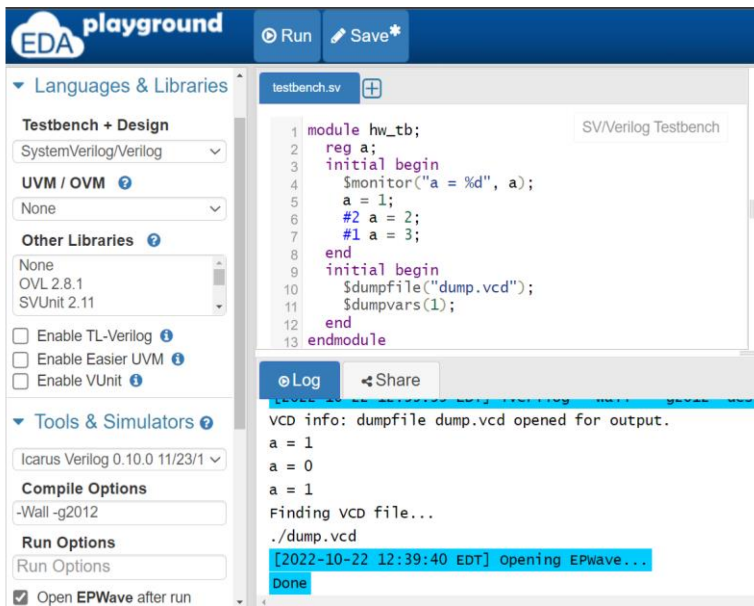


Рис. 24 – EDA Playground

С этим симулятором поддерживается просмотр в браузере временной диаграммы в EPWave. Для этого нужно включить опцию «Open EPWave after run». После окончания симуляции откроется новая вкладка, в которой будет показана временная диаграмма (рис. 25).

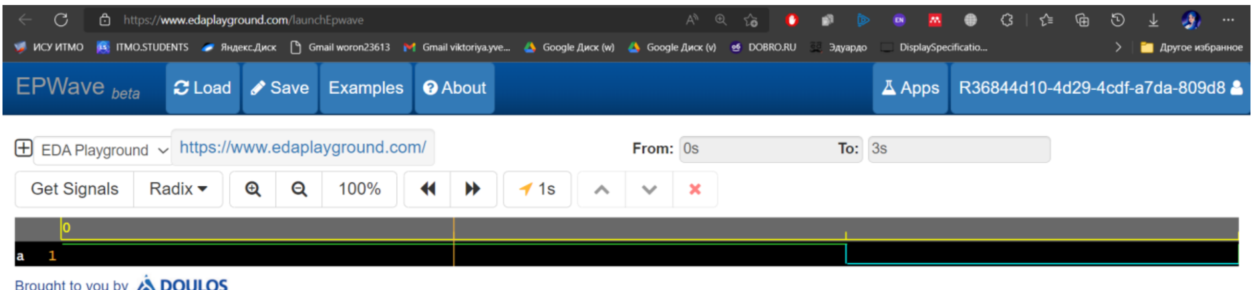


Рис. 25 – Временная диаграмма на EDA Playground

2. [Online Verilog Compiler](https://www.tutorialspoint.com/compile_verilog_online.php) (https://www.tutorialspoint.com/compile_verilog_online.php)

Аналог EDA Playground, в котором весь код пишется в одном файле. Запуск симуляции осуществляется нажатием на «Execute». На панели справа отображается вывод модуля (рис. 26).

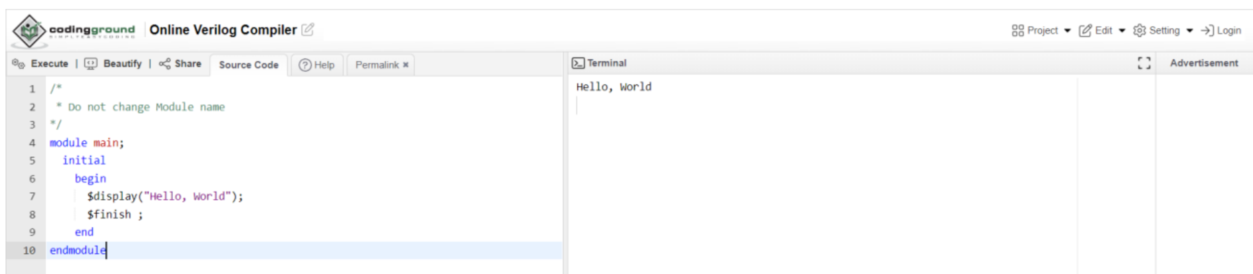


Рис. 26 – Online Verilog Compiler

Используется компилятор Icarus Verilog 10.0.

Из недостатков: данный сервис не позволяет выгружать генерируемые модулем файлы.

3. [Online VERILOG Compiler jdoodle](https://www.jdoodle.com/execute-verilog-online/) (https://www.jdoodle.com/execute-verilog-online/)

Это другой аналог EDA Playground, который представлен на рис. 27. Как и сервис из предыдущего пункта, он не позволяет выгрузить сгенерированные выходные файлы, но поддерживает пользовательский ввод в stdin, в том числе в интерактивном режиме.

В качестве компилятора используется Icarus Verilog. Версия выбирается в разделе «Execute Mode, Version, Inputs & Arguments» из доступных версий: 10.1, 10.2, 10.3 или 11.

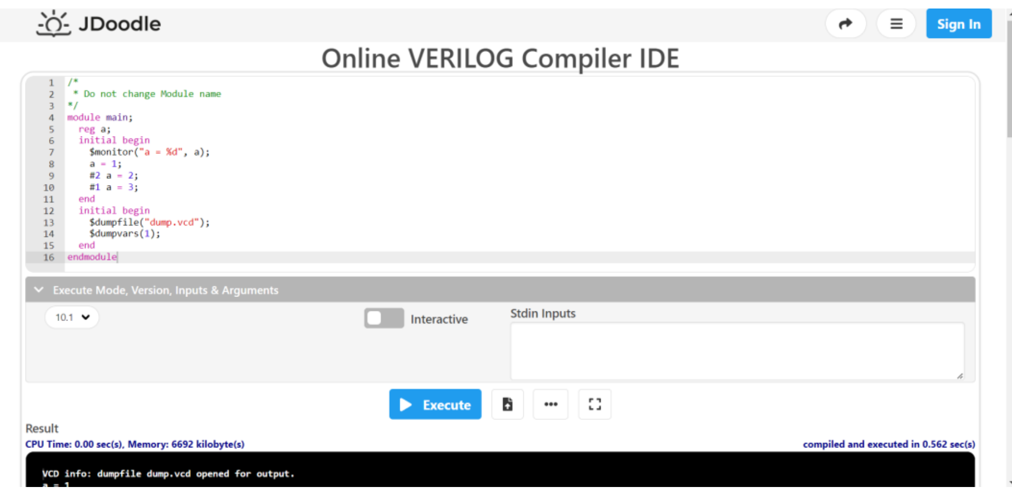


Рис. 27 – Online VERILOG Compiler jdoodle

Offline инструменты

1. Текстовый редактор для написания кода
   Может использоваться любой текстовый редактор. Далее для примера будет использоваться VSCode. Расширение SystemVerilog - Language Support - Visual Studio Marketplace добавит подсветку синтаксиса.

2. Компилятор
   Из свободно распространяемых популярным решением является Icarus Verilog. Для Windows его можно скачать по ссылке: Icarus Verilog for Windows bleyer.org. В остальных случаях рекомендуется ознакомиться со следующей инструкцией: Installation Guide | Icarus Verilog. Из командной строки его можно вызывать как

```
iverilog -help
```

3. Тестовое окружение и симулятор
   Icarus Verilog также является и симулятором, что позволяет запускать скрипты тестового окружения для проверки работоспособности.
   Компиляция и запуск осуществляется командами в терминале:

```
iverilog -g2012 -o <outfilename> <scriptfilename> && vvp <outfilename>
```

   Например:

```
iverilog -g2012 -o testbench.out testbench.sv && vvp testbench.out
```

   В VSCode есть несколько расширений, использующих Icarus Verilog:

   - Verilog HDL - Visual Studio Marketplace. После его установки симуляция запускается через Command Palette (CTRL+SHIFT+P) и выбор “Verilog: Run Verilog HDL Code” на текущем редактируемом файле. Примечание: строка запуска по умолчанию выглядит как

```
iverilog -o <outfilename> <scriptfilename> && vvp <outfilename>
```

Для добавления ключа компиляции `-g2012` нужно вправить расширение, для этого нужно перейти в каталог с расширениями VSCode (**Windows** `%USERPROFILE%\.vscode\extensions`, **Linux** и **MacOS** `~/.vscode/extensions`), найти каталог leafvmaple.verilog-*, в файле out/command.js заменить строку

```
this.COMPILE_COMMANDS = "iverilog -o {fileName}.out {fileName}";
```

на строку

```
this.COMPILE_COMMANDS = "iverilog -g2012 -o {fileName}.out {fileName}";
```

По желанию можно добавить свои ключи компиляции.

- Digital IDE - Visual Studio Marketplace. Компиляция + симуляция запускаются через ПКМ-Simulate или на панели слева при переходе в раздел IDE Tool и нажатии в sim/<testbench_module_name> на значок отладки.

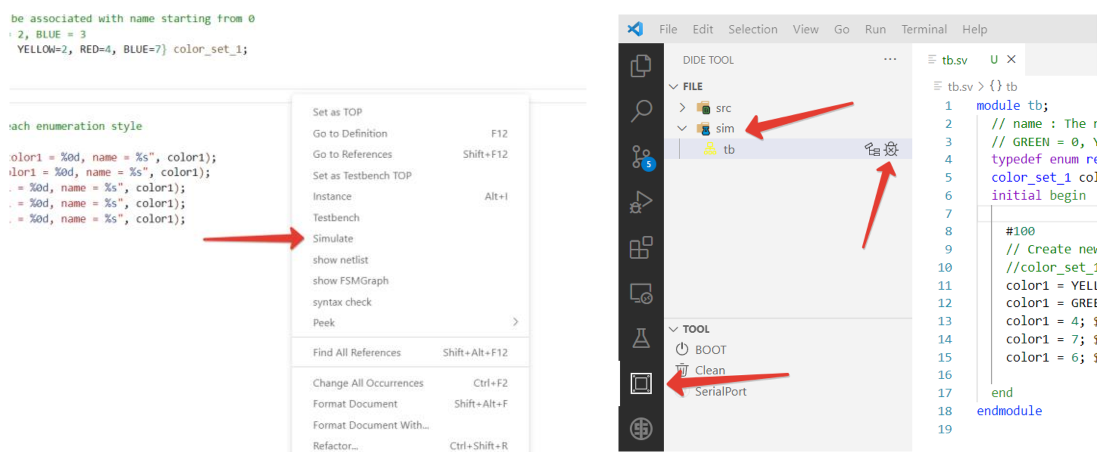


Рис. 28 – Digital IDE

- Verilog Testbench Runner - Visual Studio Marketplace. Предлагает доступ к компиляции и симуляции через отдельную кнопку на панели сверху (см. описание расширения). В настройках расширения можно задать ключи компиляции.

4. Просмотр файлов с временными диаграммами
   В случае использования VSCode можно установить расширение WaveTrace - Visual Studio Marketplace.
   Другим решением является десктопное приложение GTKWave, которое устанавливается сразу вместе с Icarus.

Основы Verilog

Симулятором называется программа, которая моделирует изменение состояния физической системы со временем. Verilog – язык описания аппаратуры (HDL) и симулятор электронной цифровой аппаратуры. Он позволяет моделировать цифровую систему, например соединённые между собой логические элементы, и сообщает, как в системе распространяются значения во времени. Симулятор помогает определить, правильно ли система реализует свою функциональность и обеспечивает ли заданную производительность. Модель проектируемой системы представляется на специальном языке моделирования.

Основные аспекты Verilog, благодаря которым на нем можно описывать цифровую аппаратуру, – ***моделирование времени, функциональности и соединений***. Соединения позволяют связывать элементы схемы ***проводами*** – когда изменяется выход одного элемента, новое значение распространяется по проводам и попадает на входы других. Так физически устроены цифровые схемы, и язык призван это отражать.

Для того, чтобы проверить функциональную корректность модели, симулятор исполняет её вместе с ***тестовым окружением*** – программой, предназначенной для тестирования разрабатываемой системы.

В Verilog иерархия элементов моделируемой системы описывается модулем. ***Модуль*** определяет как интерфейс (входы и выходы) элементов, так и их внутреннюю структуру и поведение.

Базовые логические и электрические элементы встроены в язык Verilog с помощью ***примитивов***. Разработчик также может определять свои собственные пользовательские примитивы.

Проектирование схемы производится созданием ***модулей***, состоящих из ***примитивов*** и других ***модулей***, соединённых с помощью ***проводов***.

Поведение системы описывается с использованием ***поведенческих операторов***, а также непрерывных назначений. Каждая операция выполняется одновременно по отношению ко всем остальным, но инструкции внутри поведенческих операторов выполняются в моменты, определяемые ***временными задержками***.

Поведенческие операторы могут внутри себя использовать ***функции*** и ***задачи***. Существует также ряд ***встроенных системных задач и функций***, позволяющих получать информацию из тестовой среды, выводить текущее состояние в файл или на консоль и приостанавливать или прерывать выполнение поведенческих блоков.

Исходный код Verilog обычно пишется в одном или нескольких текстовых файлах. Затем текстовые файлы передаются компилятору Verilog, который

компилирует их в запускаемый файл. Полученный файл можно запустить в режиме симуляции.

Типы данных

Тип данных определяют наборы значений и операций, которые могут быть выполнены с этими значениями. Типы данных могут использоваться для объявления объектов или пользовательских типов данных. Объект определённого типа данных представляет собой именованную сущность, имеющую значение и связанный с ним тип данных.

	Тип данных	Число состояний	Битность	Знак	Хранимое значение	
SYSTEM	reg	4	>= 1	unsigned	Набор битов	*VERILOG*
	wire	4	>= 1	unsigned	Значение на проводах	
	integer	4	32	signed	Целое число	
	real		64		Число с плавающей точкой	
	time		64	unsigned	Время	
	realtime		64		Аналог real	
	event				Событие	
VERILOG	logic	4	>= 1	unsigned	Аналог reg	
	bit	2	>= 1	unsigned	Набор битов	
	byte	2	8	signed	Целое число	
	shortint	2	16	signed	Целое число	
	int	2	32	signed	Целое число	
	longint	2	64	signed	Целое число	
	shortreal		32		Число с плавающей точкой	

Логические значения

Многие типы данных (с числом состояний 4 в таблице выше) могут принимать одно из четырёх значений. На рис. 29 показано, как эти значения представляются на временных диаграммах моделирования. Большинство симуляторов используют соглашение, где красный означает X, а оранжевый в середине означает высокий импеданс Z.

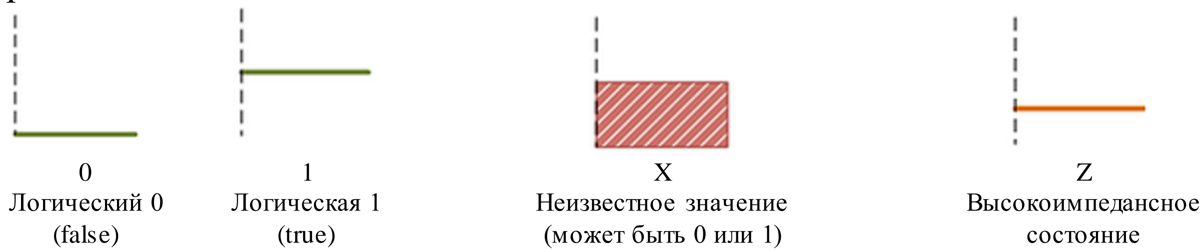


Рис. 29 – Визуализация логических значений на временных диаграммах

X (или x) означает, что значение просто неизвестно в данный момент и может быть либо 0, либо 1.

Провод, который ни к чему не подключен (или подключен через закрытый транзистор), будет иметь высокий импеданс и обозначается Z (или z). Высокоимпедансное состояние – состояние контакта схемы, при котором сопротивление между этим контактом и выводами питания/заземления очень велико.

Сила сигнала

Сила сигнала используется для более точного моделирования электрического соединения проводов. В случае, если несколько проводов объединяются в один, то сила сигнала на выходе определяется на основе силы сигналов на объединяющихся проводах, и результирующая сила сигнала будет определяться согласно приоритету, представленному ниже. Комбинация силы (highz1, highz0) недопустима.

Сила	Определение	Ключевое слово	Является значением по умолчанию для
7	Supply drive	supply0, supply1	Проводов, соединённых с заземлением и питанием соответственно
6	Strong drive	strong0, strong1	Всех остальных сигналов
5	Pull drive	pull0, pull1	Подтягивающих резисторов
4	Large capacitive	large	
3	Weak drive	weak0, weak1	
2	Medium capacitive	medium	
1	Small capacitive	small	
0	High impedance	highz0, highz1	Представления высокоимпедансного состояния

Пример

```
module strength_example1(output out, input
i1, i2);
  or(supply1, strong0) o1(out, i1, i2);
  and(strong1, supply0) a1(out, i1, i2);
endmodule
```

```
module strength_example2(output out, input
i1, i2);
  assign(supply1, pull0) out = (i1 | i2);
  assign(strong1, supply0) out = i1 & i2;
endmodule
```

```
module strength_tb;
  reg i1, i2;
  wire out;

  strength_example1 st(out, i1, i2);
  //strength_example2 st(out, i1, i2);

  initial begin
    $monitor("%0t: i1 = %b, i2 = %b -> out = %b", $time, i1, i2, out);
      i1 = 0; i2 = 0;
```

```
    #1 i1 = 0; i2 = 1;
    #1 i1 = 1; i2 = 0;
    #1 i1 = 1; i2 = 1;
  end
endmodule
```

Результат выполнения

```
Time [0]: i1 = 0, i2 = 0 -> out = 0
Time [1]: i1 = 0, i2 = 1 -> out = x
Time [2]: i1 = 1, i2 = 0 -> out = x
Time [3]: i1 = 1, i2 = 1 -> out = 1
```

Регистровые типы данных

Соответствуют объектам, которые могут хранить значение. К таким типам относятся reg, integer, time и real.

Синтаксис

```
reg signed              // Регистр со знаком (дополнение до 2)
reg unsigned            // Регистр без знака
reg [n-1:0]             // Регистр с разрядностью n
reg [n-1:0] signed      // Регистр с разрядностью n со знаком
reg [n-1:0] unsigned // Регистр с разрядностью n без знака
```

Все переменные объявляются в начале модуля и могут быть использованы только внутри него. Объявления внутри блоков `always` и `initial` не допускаются.

Преобразование вещественного к целому происходит по правилу округления к ближайшему чётному.

Пример

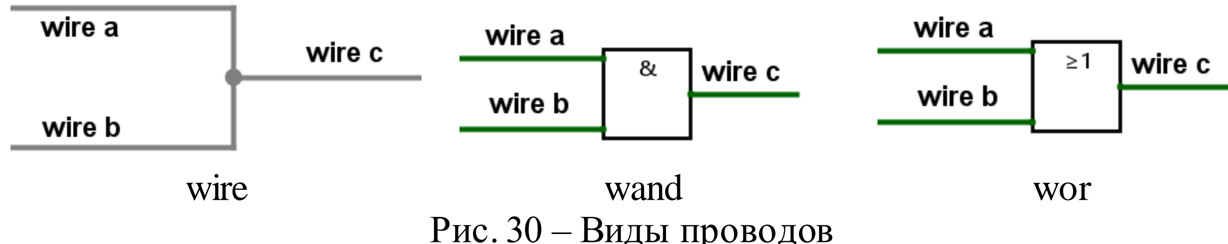
```
module data_types_tb;
  reg a; reg [7:0] b, c;
  reg signed [7:0] d;
  reg [11:0] e; integer i;
  time t; real r; realtime rt;
  initial begin
    a = 1; $display("1bit = %0d", a);
    b = 8'hF2; $display("8bit_dec = %0d", b);
    c = 8'b10110011; $display("8bit_bin = %0d",c);
    d = -8'sd5; $display("8bit (-5) = %0d", d);
    e = 12'd?; $display("Z-state = %0d", e);
    i = 32'hcafe_1234;
    $display("integer = 0x%0h", i);
    t = 112345; $display("time = %t", t);
    t = 112345; $display("time = %0t", t);
    r = 3.8e10; $display("real = %g", r);
    rt = 1.0; $display("realtime = %f", rt);
  end
endmodule
```

Результат выполнения

```
1bit = 1
8bit_dec = 242
8bit_bin = 179
8bit (-5) = -5
Z-state = z

integer = 0xcafe1234
time =                112345
time = 112345
real = 3.8e+10
realtime = 1.000000
```

Провода (wire)

Провода соединяют вместе структурные компоненты модулей. По умолчанию имеют разрядность 1. Наиболее часто используемые провода представлены на рис. 30.



wire wand wor

Рис. 30 – Виды проводов

Если провод имеет несколько источников сигнала (например, выходы двух элементов подключены к одному проводу), то значение результирующего сигнала определяется согласно силе сигнала для `wire` или указанной логической операции для `wand` и `wor`.

Синтаксис

```
wire errl, err2;                 // Два обычный провода
wand err3, err4;                 // Два провода с «монтажным» И
wire [7:0] busl, bus2;           // Две шины (пучок проводов) размером 8 (8бит)
wire vectored[0:7] bus3, bus4; // Две шины (пучок проводов) размером 8 (8бит)
wire (highz1, strong0) [0:7] opbus; // Шина с открытым коллектором
```

По умолчанию на выходе логических элементов и открытых транзисторов, подключенных к питанию или сильному сигналу, будет сильный сигнал.

Пример

```
module st_(output wire out, input wire i1, i2);
  or (supply1, strong0) o1(out, i1, i2);
  and (strong1, supply0) a1(out, i1, i2);
endmodule
module wire_tb;
  reg i1, i2;
  wire out;
  st_ st(out, i1, i2);
  initial begin
    $monitor("Time [%0t]: i1 = %b, i2 = %b -> out = %b", $time, i1, i2, out);
       i1 = 0; i2 = 0;
    #1 i1 = 0; i2 = 1;
  end
endmodule
```

Результат выполнения

```
Time [0]: i1 = 0, i2 = 0 -> out = 0
Time [1]: i1 = 0, i2 = 1 -> out = x
```

Логические типы данных (logic и bit)

Как и `reg`, `logic` хранит значения, находящиеся в одном из 4 состояний. Главным отличием его от регистра является то, что управлять значениями можно как в процедурных блоках, так и через непрерывное присваивание как у проводов. Для `logic` может быть только 1 исходный сигнал (для проводов их может быть несколько).

Пример

```
module logic_tb;
    logic[3:0] data;
    logic x;
    assign x = data[0];
    initial begin
        $display("data=0x%0h x=%0b", data, x);
        data = 4'hB; $display("data=0x%0h x=%0b", data, x);
        #1; $display("data=0x%0h x=%0b", data, x);
    end
endmodule
```

Результат выполнения

```
data=0xx x=x
data=0xb x=x
data=0xb x=1
```

Значения типа `bit` могут находиться в одном из двух состояний: логический 0 или 1. При приведении `logic` (или `reg`) в `bit` значения z и x конвертируются в 0.

Пример

```
module bit_tb;
  bit var_a;
  bit [3:0] var_b;
  logic [3:0] x_val;
  initial begin
    $display("var_a=%0b var_b=0x%0h", var_a, var_b);
    var_a = 1;  var_b = 4'hF;
    $display("var_a=%0b var_b=0x%0h", var_a, var_b);
    var_b = 16'h481a;
    $display("var_b=0x%0h", var_b);
    $display("var_b = %b", var_b);
  end
endmodule
```

Результат выполнения

```
var_a=0 var_b=0x0
var_a=1 var_b=0xf
var_b=0xa
var_b = 1010
```

Строки

Строки представляют собой набор 8-битных символов. Строки могут быть сохранены в регистры. В случае, если строка длиннее, чем регистр, в который её пытаются записать, будет записана конечная часть строки, которая умещается.

Синтаксис

```
"string"
string <var_name> = "string";
```

Пример

```
module string_tb;
  string str0; reg [16*8-1:0] str1, str2;
  initial begin
    str0 = "Hello!"; $display("str0 = |%s|", str0);
    str1 = "How are you"; $display("str1 = |%s|", str1);
    str2 = "How are you doing?"; $display("str2 = |%s|", str2);
  end
endmodule
```

Результат выполнения

```
str0 = |Hello!|
str1 = |     How are you|
str2 = |w are you doing?|
```

Пояснения к примеру

Переменные `str1` и `str2` могут хранить 16 символов (characters). В str1 записывается строка в 11 символов и выравнивается по правому краю. В `str2` же пытаются записать больше символов, чем в её размер, поэтому в итоге записывается только часть символов.

Перечисления

Перечисление определяет набор именованных значений. Имена в перечислениях не могут начинаться с цифры. Можно задать целочисленные значения к именам перечисления на этапе объявления перечисления. Если значение имени на этапе объявления не определено явно, то ему будет назначено значение предыдущего имени в перечислении, увеличенное на 1, а первому имени будет назначен 0.

Синтаксис

`enum {name}`	Следующее числовое значение будет ассоциировано с именем
`enum {name = C}`	Ассоциация константы `C` с именем
`enum {name[N]}`	Генерирует N именованных констант: `name0, name1, ..., nameN-1`
`enum {name[N] = C}`	Первая именованная константа получает значение `C`, а последующие связаны с предыдущими увеличением ассоциированного значения

`enum {name[N:M]}`	Первой константой с именем будет `name[N]`, а последней константой с именем `name[M]`, где `N` и `M` – целые числа
`enum {name[N:M] = C}`	Первая именованная константа `name[N]` получает значение `C`, а последующие связаны с предыдущими увеличением ассоциированного значения до `name[M]`

Примеры

```
module enum_tb;
  // GREEN = 0, YELLOW = 1, RED = 2, BLUE = 3
  typedef enum {GREEN, YELLOW, RED, BLUE} color_set_1;
  // MAGENTA = 2, VIOLET = 3, PURPLE = 8, PINK = 9
  typedef enum {MAGENTA=2, VIOLET, PURPLE=8, PINK} color_set_2;
  // BLACK0 = 0, BLACK1 = 1, BLACK2 = 2, BLACK3 = 3
  typedef enum {BLACK[4]} color_set_3;
  // RED0 = 5, RED1 = 6, RED2 = 7
  typedef enum {RED[3] = 5} color_set_4;
  // YELLOW3 = 0, YELLOW4 = 1, YELLOW5 = 2
  typedef enum {YELLOW[3:5]} color_set_5;
  // WHITE3 = 4, WHITE4 = 5, WHITE5 = 6
  typedef enum {WHITE[3:5] = 4} color_set_6;

  initial begin
    color_set_1 color1;
    color1 = YELLOW; $display("color1 = %0d, name = %s", color1, color1.name());

    color_set_2 color2;
    color2 = PURPLE; $display("color2 = %0d, name = %s", color2, color2.name());

    color_set_3 color3;
    color3 = BLACK3; $display("color3 = %0d, name = %s", color3, color3.name());

    color_set_4 color4;
    color4 = RED1; $display("color4 = %0d, name = %s", color4, color4.name());

    color_set_5 color5;
    color5 = YELLOW3; $display("color5 = %0d, name = %s", color5, color5.name());

    color_set_6 color6;
    color6 = WHITE4; $display("color6 = %0d, name = %s", color6, color6.name());
  end
endmodule
```

Результат выполнения

```
color1 = 1, name = YELLOW
color2 = 8, name = PURPLE
color3 = 3, name = BLACK3
color4 = 6, name = RED1
color5 = 0, name = YELLOW3
color6 = 5, name = WHITE4
```

Векторы

Verilog представляет возможность работы как с отдельными объектами (скалярами), так и с их наборами – векторами.

Объявление `wire` или `reg` без спецификации диапазона считается шириной в 1 бит и является скалярным. Если указан диапазон, то `wire` или `reg` становятся многобитовыми (вектором).

	Скалярные значения	Векторные значения
Провод	wire n1	wire [3:0] n0 n0[3] n0[2] n0[1] n0[0]
	`wire n1;`	`wire [3:0] n0;`
Регистр	reg d1	reg [3:0] d0 3 2 1 0 index
	`reg d1;`	`reg [3:0] d0;`

Диапазон даёт возможность обращаться к отдельным битам в векторе. Диапазон указывается двумя константами компиляции, разделёнными символом `:` в порядке от номера старшего бита к младшему.

Обращение к элементам вектора происходит через оператор `[]`. Помимо отдельных элементов можно адресоваться к непрерывной части элементов.

Синтаксис

```
[<start_bit> +: <width>] // выборка значений в сторону увеличения индекса
[<start_bit> -: <width>] // выборка значений в сторону уменьшения индекса
```

Пример 1 с иллюстрацией

Код									
`reg [7:0] x;`	x								
	Индекс	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
`x[0] = 1;`	x							1	
	Индекс	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
`x[3] = 0;`	x					0		1	
	Индекс	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
`x[8] = 1;`	`// illegal : bit 8 does not exist in addr`								
`x[7:4] = 4'hC;`	x	1	1	0	0	0		1	
	Индекс	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Пример 2

```
module vectorized_tb;
  reg [31:0] data; int i;

  initial begin
```

```
    data = 32'hFACE_CAFE;
    for (i = 0; i < 4; i++) begin
      $display("data[8*%0d +: 8] = 0x%0h", i, data[8*i +: 8]);
    end
    $display("data[7:0]   = 0x%0h", data[7:0]);
    $display("data[15:8]  = 0x%0h", data[15:8]);
    $display("data[23:16] = 0x%0h", data[23:16]);
    $display("data[31:24] = 0x%0h", data[31:24]);
  end
endmodule
```

Результат выполнения

```
// начало
data[8*0 +: 8] = 0xfe
data[8*1 +: 8] = 0xca
data[8*2 +: 8] = 0xce
data[8*3 +: 8] = 0xfa
```

```
// продолжение
data[7:0]   = 0xfe
data[15:8]  = 0xca
data[23:16] = 0xce
data[31:24] = 0xfa
```

Массивы

В SystemVerilog, помимо статических массивов, введённых в Verilog, поддерживаются динамические и ассоциативные массивы, а также очереди. В свою очередь все массивы делятся на ***упакованные*** (***packed***) и ***неупакованные*** (***unpacked***) массивы. Упакованные массивы – это вектора из Verilog. Далее будут рассматриваться неупакованные массивы.

Статические массивы

Статический массив – это массив, размер которого известен во время компиляции. В примере, показанном ниже, объявляется ***статический массив*** шириной 8 бит, ему присваивается некоторое значение, и в цикле выводится содержимое.

Пример

```
module one_dimention_static_array_tb;
  bit [3:0] data;
  initial begin
    data = 8'hA;
    for (int i = 0; i < $size(data); i++) begin
      $display("data[%0d] = %b", i, data[i]);
    end
  end
endmodule
```

Результат выполнения

```
data[0] = 0
data[1] = 1
data[2] = 0
data[3] = 1
```

Динамические массивы

Динамический массив – это неупакованный массив, размер которого может быть установлен или изменён во время выполнения. Размер ***динамического массива*** по умолчанию равен нулю до тех пор, пока он не будет установлен конструктором `new()`, который используется для выделения памяти массиву и, при необходимости, инициализации его элементов.

Синтаксис

```
[data_type] [identifier_name]  [];
bit [7:0] stack []; // Динамический массив 8-битных векторов
string names [];    // Динамический массив данных типа string
```

К динамическим массивам можно применять функции `size()` и `delete()`. Первая возвращает количество элементов в массиве на момент вызова, а вторая – очищает весь массив.

Для добавления нового элемента в существующий массив необходимо создать новый массив нужного размера и скопировать в него старый массив.

Пример

```
module dynamic_array_tb;
  int array[]; int id[];

  initial begin
    array = new [5]; array = '{1, 2, 3, 4, 5};
    id = array; $display("id[0] = %d", id[0]);

    id = new [id.size() + 1] (id);
    id [id.size() - 1] = 6;
    $display("New id[6] = %d", id[6]);

    $display("array.size() = %0d, id.size() = %0d", array.size(), id.size());

    array.delete(); $display("array.size() = %0d", array.size());
  end
endmodule
```

Результат выполнения

```
id[0] =           1
New id[6] =           x
array.size() = 5, id.size() = 6
array.size() = 0
```

Пояснение к примеру

В переменную `array` записывается указатель на первую ячейку области памяти из 5 значений типа `int`, после чего эта память инициализируется значениями `{1, 2, 3, 4, 5}`. Затем копия указателя записывается в

переменную `id`, и выводится первый элемент массива, на который указывает этот указатель – число 1, которым инициализировали `array`. Затем массив расширяется, значения из начального массива `array` копируются в новое место, новый указатель записывается в `id`, после чего в конец массива `id` добавляется значение 6. Т.к. данные были скопированы в другое место, то array и id ссылаются теперь на два совершенно разных массива. Последним шагом показано использование функции освобождения памяти, выделенной динамически.

Другие виды массивов

Помимо статических и динамических массивов, в SystemVerilog поддерживаются *ассоциативные массивы* и *очереди*. На момент написания данного пособия они не поддерживаются в полной мере в Icarus Verilog 12.0 (devel), поэтому кратко обозначим их предназначение и потенциальные возможности без примеров.

Ассоциативный массив реализует таблицу поиска элементов его объявленного типа. Тип данных, который будет использоваться в качестве индекса, служит ключом поиска. К ним могут применяться функции поиска по ключу элемента, извлечение значения по ключу, перемещение по списку в зависимости от значения ключа, получение размера ключа и удаление всего массива.

Очередь представляет собой массив с доступом к элементам вида «первый пришёл – первый вышел» (FIFO, first in, first out), который может иметь переменный размер для хранения элементов одного и того же типа данных. Ими также можно управлять с помощью операторов индексации, конкатенации и нарезки.

Примитивы

Примитивы используются в модулях для описания поведения или структуры модуля. Примитивы могут быть встроенными или пользовательскими. В этом разделе будут рассматриваться только ***встроенные примитивы***, которые разделяют на переключатели, логические элементы, события, а также `parameter` и `specparam`.

Ниже приведено несколько полезных примеров переключателей и логических элементов.

Описание	**Тип**	**Параметры**
Логические элементы	not, and, nand, or, nor, xor, xnor	output, input(s)
Буфер	buf	output(s), input(s)
Полевые транзисторы	nmos, pmos	output (drain), input (source), input (gate)

Коммутатор	cmos	output (drain), input (source), input (n-gate), input (p-gate)

Отдельно рассмотрим транзисторы:

Обозначение	Описание	Схема
nmos	Полевой транзистор с индуцированным n-каналом	gate source drain
pmos	Полевой транзистор с индуцированным p-каналом	gate source drain

cmos	Соответствует следующей схеме:
	`nmos (out, in, n_gate);`
	`pmos (out, in, p_gate);`
		p_gate
		source drain
		n_gate

Синтаксис

```
тип_логического_элемента [ (сила сигнала) ] [ #(задержка) ] [ имя_инстанса ] [
размерность ] ( terminal, terminal, ... );
тип_переключателя [ #(задержка) ] [ имя_инстанса ] [ размерность ] ( terminal,
terminal, ... );
```

Пример

```
module not_(output out, input in);
  supply1 pwr;
  supply0 grd;
  pmos p1(out, pwr, in);
  nmos n1(out, grd, in);
endmodule

module not_tb;
  reg[1:0] in_value;
  wire[1:0] out;
  not_ not_var(.out(out[0]), .in(in_value[0]));
  //not_ not_var(.out(out), .in(out)); // короткое замыкание
  not (out[1], in_value[1]);
  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
    $monitor("Time [%0t]:\tin:%b, out:%b", $time, in_value, out);
    in_value = 0;
    #3 in_value[0] = 1;
    #4 in_value[0] = 0;
    #3 in_value[1] = 1;
```

```
    #4 in_value[1] = 0;
  end
endmodule
```

Результат выполнения

```
Time [0]:   in:00, out:11
Time [3]:   in:01, out:10
Time [7]:   in:00, out:11
Time [10]:  in:10, out:01
Time [14]:  in:00, out:11
```

Модули

Модуль – это основной структурный элемент. Он описывает поведение системы, которая может быть реализована на физическом устройстве. Модуль описывается либо через ***описание поведения***, либо через ***описание структуры*** (например, из каких логических элементов он состоит).

Модули соединяются ***проводами***, благодаря чему они могут взаимодействовать друг с другом, образуя более крупные схемы.

Модуль описывается через ***интерфейс***, который включает в себя ***входные*** и ***выходные сигналы*** (они же называются ***порты***), а также внутреннюю функциональность. Интерфейс может быть описан как при описании модуля, так и отдельно и повторно использоваться при описании модулей.

Порты могут быть ***входом*** (`input`), ***выходом*** (`output`) или ***входо-выходом*** (`inout`) и могут быть представлены ***проводами*** (`wire`) или ***регистрами*** (`reg`). Регистры сохраняют значение между тактами синхронизации. Провода не хранят значение между тактами синхронизации.

Описание модуля начинается с ключевого слова `module`. Описание модуля заканчивается ключевым словом `endmodule` (без `;` в конце). При составлении модуля ***порты*** могут располагаться в любом порядке, подобно аргументам функции в языках программирования.

Синтаксис

```
// Вариант 1: объявление модуля без входных и выходных сигналов
module <название>;

// Вариант 2: объявление модуля с входными и выходными сигналами
module <название>(<список портов для взаимодействия с остальными модулями>);
  <объявления типов для входов и выходов>
  <объявления внутренних переменных>
  <инстанцирование (создание экземпляра модуля) внутренних модулей>
  <объявления цепей и примитивов структурного моделирования>

// Вариант 3: объявление модуля с входными и выходными сигналами с указанием их
типов
module <название>(<список портов для взаимодействия с остальными модулями с
типами портов>);
  <объявления внутренних переменных>
```

```
  <инстанцирование (создание экземпляра модуля) внутренних модулей>
  <объявления цепей и примитивов структурного моделирования>
```

Примеры

```
/* Пример 1: объявление модуля без входных и выходных сигналов */
module my_module_wo_args;

/* Пример 2: объявление модуля с входными и выходными сигналами */
module my_module_with_args(output[3:0] a, output b, input c, d);
    wire[3:0] a;
    wire b;
    reg c, d;
    …

/* Пример 3: объявление модуля с входными и выходными сигналами с указанием их
типов */
module my_module_with_args_and_types(output wire a, output wire b, input reg c,
input reg d);
```

Создание экземпляра модуля (инстанцирование)

Модули могут содержать внутри себя другие модули. Такое использование называется инстанцированием и образует иерархию модулей.

Обозначение `#` при создании экземпляра модуля используется для переопределения значений ***параметров*** в экземпляре модуля. ***Параметры*** должны быть переопределены в том же порядке, в котором они объявлены в модуле. Начиная с Verilog-2001, можно использовать обозначение переопределения именованных параметров.

При создании экземпляра модуля можно указать диапазон, аналогично созданию векторов из `reg` и `wire`, что позволяет кратко описывать подключение векторных значений к скалярным портам инстанцируемых модулей.

Подключение к ***портам*** может быть ***упорядоченным*** или ***именованным*** списком. В упорядоченном списке соединения должны быть в том же порядке, что и список портов в описании модуля. В случае, когда не указан аргумент (две запятые подряд), какие-то порты окажутся неподключенными. В именованном списке имена должны соответствовать портам в модуле.

Для подключения к входным портам могут использоваться произвольные выражения, но выходные порты могут быть подключены только к проводам, частичным выборкам проводов или их объединениям. Входные выражения создают неявные ***непрерывные присваивания***.

Сила сигнала может быть использована только для примитивов.

Синтаксис

```
// вариант объявления экземпляра модуля с упорядоченным списком портов
<имя модуля> <имя инстанса>(<порты>);

// вариант объявления экземпляра модуля с именованным списком портов
<имя модуля> <имя инстанса>(.<имя порта> ( <сигнал> ), .<имя порта> ( <сигнал>
), …);
```

```
// вариант объявления экземпляра модуля с заданием силы выходного сигнала
<имя модуля> (<сила сигнала>) <имя инстанса>(.<имя порта> ( <сигнал> ), .<имя
порта> ( <сигнал> ), …);

// вариант объявления экземпляра модуля с переопределением значений параметров
<имя модуля> #(<список значений параметров>) <имя инстанса>(.<имя порта> (
<сигнал> ), .<имя порта> ( <сигнал> ), …);
```

Примеры

```
/* Пример 1: определение модуля и его инстанцирование с именованным списком
портов */
module logic_gates(input in1, in2, output out1, out2);
    assign out1 = in1 & in2;
    assign out2 = in1 | in2;
endmodule

module logic_gates_tb;
    reg in1, in2;
    wire out1, out2;
    logic_gates lg_inst(.in1(in1), .in2(in2), .out1(out1), .out2(out2));

    initial begin
        $monitor("%d %d %d %d", in1, in2, out1, out2);
        in1 = 1’b0; in2 = 1’b0;
        #10 in1 = 1’b1;
        #10 in2 = 1’b1;
    end
endmodule


/* Пример 2: инстанцирование встроенных логических примитивов с заданием силы
выходного сигнала */
module strength_(output out, input i1, i2);
  or (supply1, strong0) o1(out, i1, i2);
  and (strong1, supply0) a1(out, i1, i2);
endmodule


/* Пример 3: определение модуля, в котором инстанцируется с переопределением
параметра модуля */
module counterm_tb #(parameter _N = 6);
    reg clk, reset, enable;
    wire[_N-1:0] out;
    counterm #(_N) _counterm(out, clk, reset, enable);
endmodule
```

Объявление портов и внутренних переменных

Порты модуля моделируют контакты аппаратных компонентов. При объявлении порта также можно указывать тип и размерность (ANSI-стиль).

Verilog требует, чтобы сигналы, подключенные к входу или выходу модуля, имели два объявления: ***направление*** и ***тип данных портов***. Если размерность внутренних переменных и портов явно не указана, то она будет равна 1. Если явно не указан тип, то будет `wire`. При объявлении внутренних переменных и портов необходимо указывать их имя. Если же имя не указано, то компилятор выдаст ошибку.

Синтаксис

```
/* Вариант 1: объявление порта с размерностью по умолчанию 1 */
<направление> <тип> <имя_порта>

/* Вариант 2: объявление порта с заданной размерностью */
<направление> <тип> <имя_порта>[<размерность>]

/* Вариант 3: объявление портов одинакового направления, размерности и типа */
<направление> <тип> <имя_порта>, <имя_порта>, …

/* Вариант 4: объявления внутренней переменной с размерностью по умолчанию 1 */
<тип> <имя_переменной>;

/* Вариант 5: объявления внутренней переменной с заданной размерностью */
<тип> <имя_переменной>[<размерность>];
```

Примеры

```
/* Пример 1: определение модуля и его инстанцирование с именованным списком
портов */
module logic_gates(input in1, in2, output out1, out2);
    ...
endmodule

module logic_gates_tb;
    logic_gates logic_gates_inst(.in1(in1), .in2(in2), .out1(out1),
.out2(out2));
    ...
endmodule

/* Пример 2: инстанцирование встроенных логических примитивов с заданием силы
выходного сигнала */
module strength_(output out, input i1, i2);
  or (supply1, strong0) o1(out, i1, i2);
  and (strong1, supply0) a1(out, i1, i2);
endmodule

/* Пример 3: определение модуля с переопределением параметра модуля */
module counterm_tb #(parameter _N = 6);
    counterm #(_N) _counterm(out, clk, reset, enable);
    ...
endmodule
```

Параметры

Параметры позволяют легко создать несколько похожих модулей с различной спецификацией. Параметры являются константами времени компиляции. SystemVerilog предоставляет несколько методов для установки значения констант параметров. Каждому параметру может быть присвоено значение по умолчанию при объявлении. Значение параметра созданного модуля, интерфейса или программы может быть переопределено в каждом экземпляре.

Параметры обозначаются ключевым словом `parameter`. Помимо параметров, которые можно специфицировать извне модуля, можно задать локальные параметры (`localparam`), которые задаются внутри модуля и не видны снаружи.

Синтаксис

```
/* Вариант 1: объявление параметризованного модуля */
module module_name #(parameter <parameter_name> = <value>) ...

/* Вариант 2: объявление параметризованного модуля с локальным параметром */
module module_name(ports);
  parameter <parameter_name> = <value>; // или localparam
  ...

/* Вариант 3: инстанцирование параметризованного модуля */
module_name module_var #(<value1>, <value2>, ...) ...

/* Враиант 4: инстанцирование параметризованного модуля именнованным списком
параметров */
module_name module_var #(.<parameter_name1>(<value1>), ...) ...
```

Пример

```
module counterm #(parameter N =
4)(output reg[N-1:0] out, input
clk, reset, enable);
  always @ (posedge clk) begin
    if (enable) begin
      if (reset)
        out = 0;
      else
        out = out + 1;
    end else begin
      out = out;
    end
  end
endmodule
```

```
module counterm_tb;
    parameter _N = 5;
    reg clk, reset, enable;
    wire[_N-1:0] out;
    counterm #(_N) _counterm(out, clk,
reset, enable); // mod 2**5

    initial begin
        $display("Time
[**]:\tclk\trst\ten\tout_bin\tout_dec");
        clk = 0; reset = 1; enable = 1;
        #1 reset = 0;
        #2 reset = 1;
        #2 reset = 0;
        #10 $finish;
    end
```

	``
	    always #1 clk = ~clk;
	
	    always @(posedge clk)
	        $display("Time
	[%0t]:\t%b\t%b\t%b\t%b\t%d",
	            $time, clk, reset, enable,
	out, out);
	endmodule
	``

Результат выполнения

```
Time [**]:  clk rst en  out_bin out_dec
Time [1]:   1   0   1   xxxxx    x
Time [3]:   1   1   1   00000    0
Time [5]:   1   0   1   00000    0
Time [7]:   1   0   1   00010    2
Time [9]:   1   0   1   00010    2
Time [11]:  1   0   1   00100    4
Time [13]:  1   0   1   00100    4
Time [15]:  1   0   1   00110    6
```

Интерфейс

Под ***интерфейсом*** понимается набор портов модулей и параметров функций. Если моделируется несколько модулей с одинаковым интерфейсом, то сам интерфейс можно описать отдельно и использовать его именованное определение в объявлении модулей. Для обозначения интерфейса используются ключевые слова `interface` и `endinterface` по аналогии с модулями.

Пример

```
interface slave_if (input logic clk,
reset);
  reg reset;
  reg enable;
  reg gnt;
endinterface
```

```
module d_slave (slave_if s_if);
  always (s_if.enable & s_if.gnt) begin
  ...
  end
endmodule
```

```
module d_top (input clk, reset);
  slave_if slave_if_inst( .clk(clk), .reset(reset));
  d_slave slave_0(.s_if(slave_if_inst));
  d_slave slave_1(.s_if(slave_if_inst));
endmodule
```

Операторы

Присваивание

В левой части оператора присваивания может быть указано несколько переменных, разделённых запятыми. В правой части может быть указано несколько выражений, разделённых запятыми. Если количество переменных в левой части больше, чем количество выражений в правой части, то последние

переменные не будут изменены. Если число выражений в правой части больше, чем число переменных в левой части, то последние выражения будут проигнорированы.

Синтаксис

```
// Блокирующее присваивание (blocking assignment)
<#задержка> <переменная> = <#задержка> <выражение>
<переменная> = <#задержка> <выражение>
<#задержка> <переменная> = <выражение>
<переменная> = <выражение>

 // Неблокирующее присваивание (non-blocking assignment)
[задержка] <регистр> <= [задержка] RHS
```

Блокирующий оператор присваивания (blocking assignment)

Обозначаются = и выполняются один за другим в процедурном блоке. Однако исполнение в различных процедурных блоках (например, при задании нескольких блоков `initial`) между собой не упорядочено.

Пример

```
module blocking_assignment_tb;
  reg [3:0] data; real r_value;
  integer i_value; time T;

  initial begin
    data = 4'h4;
    $monitor("Time [%0t] data = %0d, r_value = %0f, i_value = %0h",
             T, data, r_value, i_value);
    r_value = 3.14;
    i_value = 4;
    #2 data = 4'h5;
    #3 data = 'd7;
    i_value = 10;
    i_value = 6;
    $finish;
  end
  always #1 T = $time;
endmodule
```

Результат выполнения

```
Time [x] data = 4, r_value = 3.140000, i_value = 4
Time [1] data = 4, r_value = 3.140000, i_value = 4
Time [2] data = 5, r_value = 3.140000, i_value = 4
Time [3] data = 5, r_value = 3.140000, i_value = 4
Time [4] data = 5, r_value = 3.140000, i_value = 4
Time [4] data = 7, r_value = 3.140000, i_value = 6
```

Неблокирующий оператор присваивания (non-blocking operator)

Обозначается `<=` и работает следующим образом: когда выполнение дошло до строки с неблокирующим оператором присваивания, вычисляется правая часть оператора. Затем в конец очереди инструкций этого такта времени добавляется инструкция «присвоить вычисленный результат в переменную слева от оператора», и само присвоение происходит в конце такта.

Пример

```
module non_blocking_race_cond_tb;
  reg [3:0] data = 4'h2;
  reg [3:0] y = 4'h3;

  initial begin
    y <= data;
    $display("1st block: data = %0h, y = %0h", data, y);
  end

  initial begin
    data <= y;
    $display("2nd block: data = %0h, y = %0h", data, y);
  end
endmodule
```

Результат выполнения

```
1st block: data = 2, y = 3
2nd block: data = 2, y = 3
```

Непрерывное присваивание

Проводам требуется ***непрерывное соединение*** с каким-то значением. В Verilog эта концепция реализована с помощью оператора `assign`. Значение может быть константой или выражением.

Пример

`// Вариант назначения значения` `assign <lvalue> = <expr>;`	`// Вариант с временной задержкой` `assign #<delay> <lvalue> = <expr>;`

`module assign1;`
`  reg a, b, c;`
`  wire d;`
`  assign d = a & b | c;`
`endmodule`
	`module assign2;`
	`  reg a, b, c;`
	`  wire d;`
	`  assign #1 d = a & b | c;`
	`endmodule`

Сравнение

Выражение с оператором сравнения даст результат 1, если выражение истинно (***true***), иначе – 0 (***false***). Если любой из операндов равен X или Z, то результатом будет X.

У операторов проверки на равенство результат равен 1, если ***true***, и 0, если ***false***. Если любой из операндов логического равенства (==) или логического

неравенства (!=) равен X или Z, то результатом будет X. Также есть оператор точного равенства (===) или оператор точного неравенства (!==) для проверки без каких-либо преобразований.

Пример

```
// <тип данных> <переменная> <оператор> <выражение>
module comparison_tb;
  reg [7:0] a, b, c, d;

  initial begin
    a = 8'd4; b = 8'd5;
    $display("a = %0d, b = %0d", a, b);
    $display("(a == a) = %b", a == a); // равно
    $display("(a == b) = %b", a == b);
    $display("(a != b) = %b", a != b);  // не равно

    $display("(a < b)  = %b", a < b);   // меньше
    $display("(a <= b) = %b", a <= b);  // меньше или равно
    $display("(a > b)  = %b", a > b);   // больше
    $display("(a >= b) = %b", a >= b);  // больше или равно

    a = 8'dz; b = 8'dx;
    $display("a = %0d, b = %0d", a, b);
    $display("(a == X) = %b", a == 8'bx);
    $display("(a == Z) = %b", a == 8'bz);
    $display("(b == X) = %b", b == 8'bx);
    $display("(b == Z) = %b", b == 8'bz);
    $display("(a === X) = %b", a === 8'bx);
    $display("(a === Z) = %b", a === 8'bz);
    $display("(b === X) = %b", b === 8'bx);
    $display("(b === Z) = %b", b === 8'bz);
  end
endmodule
```

Результат выполнения

```
// начало
a = 4, b = 5
(a == a) = 1
(a == b) = 0
(a != b) = 1
(a < b)  = 1
(a <= b) = 1
(a > b)  = 0
(a >= b) = 0
```

```
// продолжение
a = z, b = x
(a == X) = x
(a == Z) = x
(b == X) = x
(b == Z) = x
(a === X) = 0
(a === Z) = 1
(b === X) = 1
(b === Z) = 0
```

Логические и побитовые операторы

Выделяют логические (`&&`, `||`, `!`) и побитовые (`&`, `|`, `~`, `^`) операторы.

Результатом логического ***И*** (`&&`) является 0 (*false*), когда хотя бы один его операнд 0; 1 (*true*), когда оба операнда 1; иначе X. Аналогично для логического ***ИЛИ*** (`||`). Оператор логического отрицания (`!`) преобразует истинный операнд в 0, а ложный операнд в 1, иначе X.

Побитовые операторы объединяют биты в одном операнде с соответствующим битом в другом операнде для вычисления побитового результата. Унарные побитовые операторы объединяют биты единственного операнда между собой, и возвращается однобитовое значение.

Пример

```
// <тип данных> <переменная> <оператор> <выражение>
module logical_operators_tb;
  reg [7:0] a, b;
  initial begin
    a = 8'd4; b = 8'd5;
    $display("a_bin = %b, b_bin = %b ", a, b);
    $display("~a     = %b", ~a);  // побитовое логическое НЕ (унарный оператор)
    $display("a & b = %b", a & b); // побитовое логическое И (бинарный)
    $display("a | b = %b", a | b); // побитовое логическое ИЛИ (бинарный)
    $display("a ^ b = %b", a ^ b); // побитовое логическое Искл. ИЛИ (бинарный)
    $display("&a = %b", &a);   // горизонтальное логическое И (унарный)
    $display("|a = %b", |a);   // горизонтальное логическое ИЛИ (унарный)
    $display("^a = %b", ^a);   // горизонтальное логическое Искл. ИЛИ (унарный)
    $display("^~a = %b", ^~a);
    $display("1 && 0 = %b", 1 && 0); // логическое И (бинарный)
    $display("1 || 0 = %b", 1 || 0); // логическое ИЛИ (бинарный)
    $display("!1 = %b", !1);         // логическое НЕ (унарный)
  end
endmodule
```

Результат выполнения

```
// начало
a_bin = 00000100, b_bin = 00000101
~a    = 11111011
a & b = 00000100
a | b = 00000101
a ^ b = 00000001
```

```
// продолжение
&a = 0
|a = 1
^a = 1
^~a = 0
1 && 0 = 0
1 || 0 = 1
!1 = 0
```

Сдвиг

Выделяют два вида операторов сдвига:

- Операторы логического сдвига: `<<` и `>>`.
- Операторы арифметического сдвига: `<<<` и `>>>`.

Пример

```
// <переменная> <оператор> <выражение>
module shift_operators_tb;
  reg [3:0] a; int i;

  initial begin
    a = 8'h1; $display("a = 'd%0d ('b%0b)", a, a);
    for (i = 0; i < 8; i +=1 ) begin
      $display("a << %0d = 'b%b", i, a << i); // логический сдвиг влево
    end

    a = 8'h80; $display("a = 'd%0d ('b%0b)", a, a);
    for (i = 0; i < 8; i +=1 ) begin
      $display("a >> %0d = 'b%b", i, a >> i); // логический сдвиг вправо
    end

    a = 8'h1; $display("a >> 1 = 'b%b", a >> 1);
    a = 8'h1; $display("a = 'd%0d ('b%0b)", a, a);
    for (i = 0; i < 8; i +=1 ) begin
      $display("a <<< %0d = 'b%b", i, a <<< i); // арифметический сдвиг влево
    end

    a = 8'h80; $display("a = 'd%0d ('b%0b)", a, a);
    for (i = 0; i < 8; i +=1 ) begin
      $display("a >>> %0d = 'b%b", i, a >>> i); // арифметический сдвиг вправо
    end
  end
endmodule
```

Результат выполнения

```
// логический сдвиг
a = 'd1 ('b1)
a << 0 = 'b00000001
a << 1 = 'b00000010
a << 2 = 'b00000100
a << 3 = 'b00001000
a << 4 = 'b00010000
a << 5 = 'b00100000
a << 6 = 'b01000000
a << 7 = 'b10000000
a = 'd128 ('b10000000)
a >> 0 = 'b10000000
a >> 1 = 'b01000000
a >> 2 = 'b00100000
a >> 3 = 'b00010000
a >> 4 = 'b00001000
a >> 5 = 'b00000100
a >> 6 = 'b00000010
a >> 7 = 'b00000001
a >> 1 = 'b00000000
```

```
// арифметический сдвиг
a = 'd1 ('b1)
a <<< 0 = 'b00000001
a <<< 1 = 'b00000010
a <<< 2 = 'b00000100
a <<< 3 = 'b00001000
a <<< 4 = 'b00010000
a <<< 5 = 'b00100000
a <<< 6 = 'b01000000
a <<< 7 = 'b10000000
a = 'd128 ('b10000000)
a >>> 0 = 'b10000000
a >>> 1 = 'b01000000
a >>> 2 = 'b00100000
a >>> 3 = 'b00010000
a >>> 4 = 'b00001000
a >>> 5 = 'b00000100
a >>> 6 = 'b00000010
a >>> 7 = 'b00000001
```

Тернарный оператор

Оператор `? :`, выполняющий `if-then-else` конструкцию внутри выражения. Если первый аргумент ***true***, то результатом данного оператора будет второе выражение (после `?`); если первый аргумент ***false***, то третье выражение (после `:`); иначе X.

Примеры

```
// <условие> ? < если выражение истинно> : <если выражение ложно>
 module ternary_operator_tb;
  reg [7:0] a, b, c;
  initial begin
    a = 8'd4; b = 8'd5; c = (a == b) ? 8'd1 : 8'd0;
    // d = (a != b) ? 8'd1; // ошибка компиляции
    $display("a = %0d; b = %0d; c = %0d", a, b, c);
  end
endmodule
```

Результат выполнения

```
a = 4; b = 5; c = 0
```

Арифметические операторы

Если второй операнд оператора деления или модуля равен нулю, то результатом будет X.

Если любой из операндов оператора возведения в степень является вещественным, то результат также будет вещественным. Результат будет равен 1, если второй операнд оператора возведения в степень равен 0.

Пример

```
// <переменная/выражение> <оператор> <переменная/выражение>
// <переменная> <оператор>= <переменная/выражение>
 module arithmetic_operators_tb;
    reg [7:0] a, b;
    real c, d;
    initial begin
        a = 8'd4; b = 8'd5;
        $display("a = %0d; b = %0d", a, b);

        $display("a + b = %0d", a + b);     // сложение
        $display("a - b = %0d", a - b);     // вычитание
        $display("a * b = %0d", a * b);     // умножение
        $display("a / b = %0d", a / b);     // деление
        $display("a %% b = %0d", a % b);    // остаток от деления
        $display("a ** b = %0d", a ** b);   // возведение в степень

        $display("a++ = %0d", a++);     // инкремент (постфикс)
        $display("++a = %0d", ++a);     // инкремент (префикс)
        $display("a-- = %0d", a--);     // декремент (постфикс)
        $display("--a = %0d", --a);     // декремент (префикс)

        $display("a * 8'd0 = %0d", a * 8'd0);
        $display("a / 8'd0 = %0d", a / 8'd0);
```

```
        $display("a %% 8'd0 = %0d", a % 8'd0);
        $display("a ** 8'd0 = %0d", a ** 8'd0);

        c = 1.5; d = 3.14;
        $display("c = %0f; d = %0f", c, d);
        $display("2 ** c = %0f", 2 ** c);
        $display("d ** 3 = %0f", d ** 3);
    end
endmodule
```

Результат выполнения

// начало	// продолжение
a = 4; b = 5	a-- = 6
a + b = 9	--a = 4
a - b = 255	a * 8'd0 = 0
a * b = 20	a / 8'd0 = x
a / b = 0	a % 8'd0 = x
a % b = 4	a ** 8'd0 = 1
a ** b = 0	c = 1.500000; d = 3.140000
a++ = 4	2 ** c = 2.828427
++a = 6	d ** 3 = 30.959144

Объединение

Многобитовые провода и переменные могут быть объединены вместе оператором объединения `{ }`, чтобы сформировать больший провод или переменную. Данная операция называется ***конкатенацией***. В конкатенации также разрешено использовать выражения и константы размера в качестве операндов в дополнение к проводам и переменным.

Пример

```
module concatenation_tb;
  reg [7:0] a, b;
  initial begin
    a = 8'd4; b = 8'd5;
    $display("a = %0d; b = %0d", a, b);
    $display("{a, b} = %0d or %0b", {a, b}, {a, b});
    $display("{a, 8'd8, b} = %0d or %0b", {a, 8'd8, b}, {a, 8'd8, b});
  end
endmodule
```

Результат выполнения

```
a = 4; b = 5
{a, b} = 1029 or 10000000101
{a, 8'd8, b} = 264197 or 1000000100000000101
```

Приоритет исполнения

Выражения объединяют операнды с соответствующими операторами. Порядок исполнения этих операторов определяется ***приоритетом исполнения***.

В таблице ниже показаны операторы в порядке убывания приоритета. Операторы с равным приоритетом сгруппированы. Во втором столбце приведено

название на английском для удобства поиска дополнительной информации в случае необходимости.

<table>
<tr><th>Операторы</th><th>Англ. названия</th><th>Вид операторов</th></tr>
<tr><td>[ ]</td><td>bit-select or part-select</td><td></td></tr>
<tr><td>( )</td><td>parenthesis</td><td></td></tr>
<tr><td>!
~
&
|
~&
~|
^
~^ или ^~</td><td>logical negation
negation
reduction AND
reduction OR
reduction NAND
reduction NOR
reduction XOR
reduction XNOR</td><td>logical
bit-wise
reduction
reduction
reduction
reduction
reduction
reduction</td></tr>
<tr><td>+
-</td><td>unary (sign) plus
unary (sign) minus</td><td>arithmetic
arithmetic</td></tr>
<tr><td>{ }</td><td>concatenation</td><td>concatenation</td></tr>
<tr><td>{{ }}</td><td>replication</td><td>replication</td></tr>
<tr><td>*
/
%</td><td>multiply
divide
modulus</td><td>arithmetic
arithmetic
arithmetic</td></tr>
<tr><td>+
-</td><td>binary plus
binary minus</td><td>arithmetic
arithmetic</td></tr>
<tr><td><<
>></td><td>shift left
shift right</td><td>shift
shift</td></tr>
<tr><td>>
>=
<
<=</td><td>greater than
greater than or equal to
less than
less than or equal to</td><td>relational
relational
relational
relational</td></tr>
<tr><td>== и !=</td><td>logical (in)equality</td><td>equality</td></tr>
<tr><td>=== и !==</td><td>case (in)equality</td><td>equality</td></tr>
<tr><td>&</td><td>bit-wise AND</td><td>bit-wise</td></tr>
<tr><td>^
^~ или ~^</td><td>bit-wise XOR
bit-wise XNOR</td><td>bit-wise
bit-wise</td></tr>
<tr><td>|</td><td>bit-wise OR</td><td>bit-wise</td></tr>
<tr><td>&&</td><td>logical AND</td><td>logical</td></tr>
<tr><td>||</td><td>logical OR</td><td>logical</td></tr>
<tr><td>?:</td><td>conditional</td><td>conditional</td></tr>
</table>

Поведенческое моделирование

Поведение модуля описывается с использованием инструкций процедурного блока `initial` и `always` или с использованием ***непрерывных назначений***. Операторы внутри `initial` или `always` группируются вместе в `begin-end` или `fork-join`.

Выполнение инструкций контролируется программными инструкциями, управлением временем, процедурными назначениями и процедурными непрерывными назначениями.

Поведенческий процедурный оператор initial

Процедурный оператор `initial` позволяет инициализировать внутренние переменные модуля. С него начинается исполнение модуля при запуске симулятора. Внутри `initial` можно поставить одно выражение или блок `begin-end`. Содержимое `initial` выполняется один раз, последовательно, начиная с момента времени 0.

Операторов инициализации `initial` может быть несколько. Разные `initial` выполняются в неопределённом порядке, независимо друг от друга.

Синтаксис

```
initial begin
  <код>
end
```

Пример

```
/* Пример 1: несколько операторов
initial */
module multi_initial_tb();
  initial begin
    $display("Hello world0");
  end
  initial begin
    $display("Hello world1");
  end
endmodule
```

```
/* Пример 2: несколько операторов
initial, в которых действия будут
выполнены в разных тактах */
module multi_initial_tb();
  initial begin
    #1 $display("Hello world0");
  end
  initial begin
    $display("Hello world1");
  end
endmodule
```

Пояснение к примеру

В силу «одновременного» исполнения всех блоков `initial` с начала симуляции в первом примере может отработать сначала как первый `display`, так и второй. Во втором примере гарантируется, что сначала будет выведено `"Hello world1"`, а после `"Hello world0"` за счёт установленной временной задержки в первом блоке.

Поведенческий процедурный оператор always

Операторы `always` выполняются непрерывно во время моделирования. Выражение внутри оператора `always` выполняется только при условии того, что выражение, описанное в ***списке чувствительности***, равно 1 (***true***). В списке чувствительности имена входных сигналов разделяются ключевым словом `or` или символом `,` .

Токен `@*` добавляет в список чувствительности все провода и переменные, которые считываются операторами внутри `always`.

Синтаксис

`/* Пример 1: always, реагирующий на все переменные в модуле */`
`always @* begin`
`  <выражения>`
`end`
 	
`/* Пример 2: always выполняется, когда a || b == true */`
`always @(a, b) begin`
`  <выражения>`
`end`
	`/* Пример 3: always выполняется, когда a || b || c == true */`
	`always @(a or b or c) begin`
	`  <выражения>`
	`end`
	
	`/* Пример 4: always, срабатывающий каждый такт */`
	`always #1 begin`
	`  <выражения>`
	`end`

Пример

```
module always_tb;
  reg [3:0] counter;

  initial begin
    counter = 0;
  end

  always #2 begin
    counter = counter + 1;
  end

  always #1 @(counter) begin
      $display("Hello world %d at %0t", counter, $time);
  end

  always #10 begin
    $finish;
  end
endmodule
```

Результат выполнения

```
Hello world  1 at 2
Hello world  2 at 4
Hello world  3 at 6
Hello world  4 at 8
Hello world  5 at 10
```

Блочный последовательный оператор begin-end

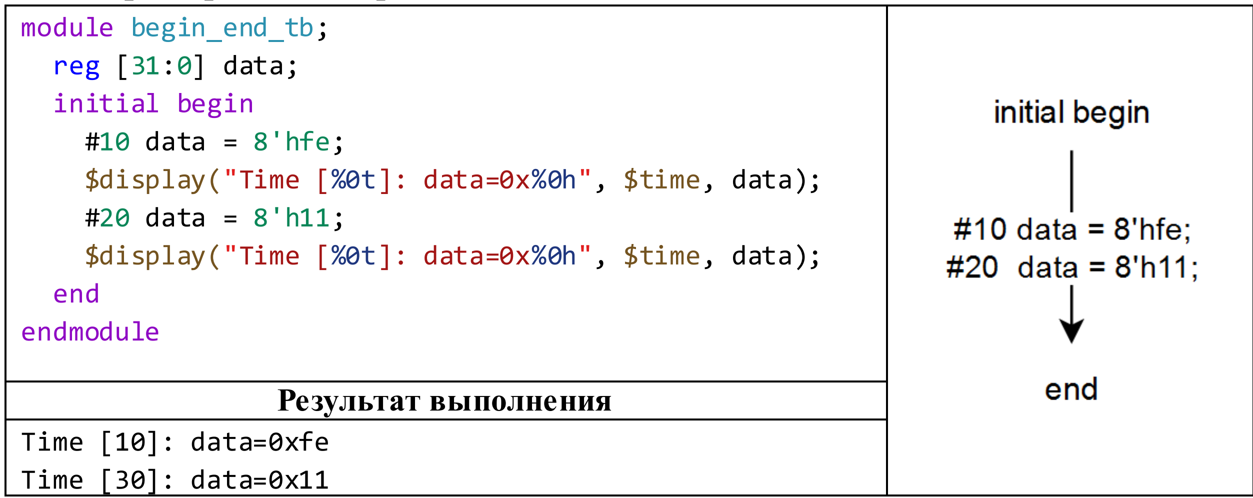
В блочном последовательном операторе `begin-end` происходит последовательное выполнение в порядке перечисления операторов. Значения задержки обрабатываются относительно времени выполнения предыдущего оператора. После выполнения всех инструкций внутри блока управление может быть передано в другое место.

Блок `begin-end` должен содержать по крайней мере один оператор.

Синтаксис

```
begin
  // ...
end
```

Пример с иллюстрацией



```
module begin_end_tb;
  reg [31:0] data;
  initial begin
    #10 data = 8'hfe;
    $display("Time [%0t]: data=0x%0h", $time, data);
    #20 data = 8'h11;
    $display("Time [%0t]: data=0x%0h", $time, data);
  end
endmodule
```

Результат выполнения

```
Time [10]: data=0xfe
Time [30]: data=0x11
```


Блочный параллельный оператор fork-join

Параллельный блок может выполнять инструкции одновременно, и управление задержкой может использоваться для обеспечения временного упорядочения выражений. Порядок инструкций внутри блока `fork-join` не имеет значения, т.к. все действия выполняются параллельно. Упорядочивание инструкций внутри оператора может обеспечиваться указанием временных задержек. Блок `fork-join` завершается, когда все инструкции были завершены.

Существуют варианты `fork-join`, которые позволяют основному потоку продолжать выполнение остальных инструкций в зависимости от того, когда дочерние потоки заканчиваются.

Синтаксис

```
fork
  // Thread 1
  // ...
join
```

Пример

```
// Пример 1
module fork_join_tb1;
  reg [31:0] data;

  initial begin
    $display("Time [%0t]: data=0x%0h\t1", $time, data);
    #10   data = 8'hfe;
    $display("Time [%0t]: data=0x%0h\t2", $time, data);

    fork
      $display("Time [%0t]: data=0x%0h\t3", $time, data);
      #20 data = 8'h11;
      #19 $display("Time [%0t]: data=0x%0h\t4", $time, data);
      #20 $display("Time [%0t]: data=0x%0h\t5", $time, data);
```

```
      #10 data = 8'h00;
      #10 $display("Time [%0t]: data=0x%0h\t6", $time, data);
    join
  end
endmodule


// Пример 2
module fork_join_tb2;
  initial begin

    $display("Time [%0t]: Main Thread: Fork join going to start", $time);

    fork
      // Поток (Thread) 1
      #30 $display("Time [%0t]: Thread1 finished", $time);

      // Поток (Thread) 2
      begin
        #5 $display("Time [%0t]: Thread2 ...", $time);
        #10 $display("Time [%0t]: Thread2 finished", $time);
      end

      // Поток (Thread) 3
      #20 $display("Time [%0t]: Thread3 finished", $time);
    join

    $display("Time [%0t]: Main Thread: Fork join has finished", $time);
  end
endmodule
```

Результат выполнения

<table>
<tr>
<td>

```
// Пример 1
Time [0]:
data=0xxxxxxxxx 1
Time [10]: data=0xfe 2
Time [10]: data=0xfe 3
Time [20]: data=0xfe 6
Time [29]: data=0x0 4
Time [30]: data=0x0 5
```

</td>
<td>

Иллюстрация примера 1

initial begin
#10 data = 8'hfe;
#10 data = 8'h11;
#20 data = 8'h00;
#30 data = 8'haa;
end

</td>
<td>

```
// Пример 2
Time [0]: Main Thread: Fork join
going to start
Time [5]: Thread2 ...
Time [15]: Thread2 finished
Time [20]: Thread3 finished
Time [30]: Thread1 finished
Time [30]: Main Thread: Fork
join has finished
```

</td>
</tr>
</table>

Управление временем

Необходимо для моделирования последовательности событий и конечной скорости распространения сигналов в реальных схемах.

Временная задержка (delay)

Временная задержка представляет собой способ задания симулятору задержки между моментом, когда он встречается с инструкцией, и моментом, когда он фактически её выполняет.

Если выражение задержки вычисляется как ***неизвестное*** или ***высокоимпедансное значение***, оно будет интерпретироваться как нулевая задержка. Если оно принимает отрицательное значение, то оно будет интерпретировано как беззнаковое целочисленное значение, соответствующее значению в форме дополнения до 2, того же размера, что и временная переменная.

Синтаксис

`#<number>`
`#<identifier>`
`#(<min_exp>: <typical_exp>, <max_exp>)`
	`<> = #<number> <>`
	`<> = #<identifier> <>`
	`<> = #(<min_exp>: <typical_exp>, <max_exp>) <>`

Пример

```
`timescale 1ns/1ps

module timescale_tb;
  reg [3:0] a, b;

  initial begin
    {a, b} <= 0; $display("Time [%5t]: a=%0d b=%0d", $realtime, a, b);

    #10; a <= $random; $display("Time [%5t]: a=%0d b=%0d", $realtime, a, b);

    #10 b <= $random; $display("Time [%5t]: a=%0d b=%0d", $realtime, a, b);

    #(a) $display("Time [%5t]: After delay of a=%0d units", $realtime, a);
    #(a+b) $display("Time [%5t]: After delay of (a=%0d + b=%0d =) %0d units",
$realtime, a, b, a+b);
    #((a+b)*10ps) $display("Time [%5t]: After delay of %0d * 10ps", $realtime,
a+b);

    #(b-a) $display("Time [%5t]: Expr evaluates to a negative delay",
$realtime);
    #('h10) $display("Time [%5t]: Delay in hex", $realtime);

    a = 'hX;
```

```
    #(a) $display("Time [%5t]: Delay is 'hX, taken as zero a=%h", $realtime,
a);

    a = 'hZ;
    #(a) $display("Time [%5t]: Delay is in high impedance, taken as zero a=%h",
$realtime, a);

    #1ps $display("Time [%5t]: Delay of 10ps", $realtime);
  end
endmodule
```

Результат выполнения

```
Time [    0]: a=x b=x
Time [10000]: a=0 b=0
Time [20000]: a=4 b=0
Time [24000]: After delay of a=4 units
Time [29000]: After delay of (a=4 + b=1 =) 5 units
Time [29050]: After delay of 5 * 10ps
Time [42050]: Expr evaluates to a negative delay
Time [58050]: Delay in hex
Time [58050]: Delay is 'hX, taken as zero a=x
Time [58050]: Delay is in high impedance, taken as zero a=z
Time [58051]: Delay of 10ps
```

Задержка также может указываться внутри оператора присваивания. В таком случае сначала будет вычислена правая часть оператора, затем произойдёт задержка на указанное время, после чего произойдёт запись вычисленного значения в переменную слева.

Пример

```
module inter_assignment_delays_tb;
  reg  a, b, c, q;
  initial begin
    $monitor("[%0t] a=%0b b=%0b c=%0b
q=%0b", $time, a, b, c, q);
    a <= 0;
    b <= 0;
    c <= 0;
    q <= 0;
    #5 a <= 1; c <= 1;
    #5 q <= a & b | c;
    #20;
  end
endmodule
```

```
module intra_assignment_delays_tb;
  reg  a, b, c, q;
  initial begin
    $monitor("[%0t] a=%0b b=%0b c=%0b
q=%0b", $time, a, b, c, q);
    a <= 0;
    b <= 0;
    c <= 0;
    q <= 0;
    #5 a <= 1; c <= 1;
    q <= #5 a & b | c;
    #20;
  end
endmodule
```

Результат выполнения

```
[0] a=0 b=0 c=0 q=0
[5] a=1 b=0 c=1 q=0
[10] a=1 b=0 c=1 q=1
```

```
[0] a=0 b=0 c=0 q=0
[5] a=1 b=0 c=1 q=0
```

Пояснение примера

Результат вывода в примере справа изменился, потому что теперь задержка внутри оператора присваивания не позволяет симулятору вычислить значение выражения `q = a & b | c`, пока не истечёт задержка.

Таким образом, значение a и c вычисляется равным 1, но ещё не присваивается, когда выполняется следующий неблокирующий оператор. Таким образом, когда вычисляется правая часть переменной q, значения a и c по-прежнему равны 0, и, следовательно, `$monitor` не обнаруживает изменения для отображения выражения.

`timescale

Моделирование зависит от того, как определено время, потому что симулятор должен знать, что означает `#1` с точки зрения времени. Директива компилятора `` `timescale `` определяет единицу времени и точность для следующих за ней модулей.

Примеры

```
// `timescale <time_unit>/<time_precision>
`timescale 1ns/1ps
`timescale 10us/100ns
`timescale 10ns/1ns
```

Значение time_unit задаёт измерение задержек и времени моделирования, в то время как time_precision определяет, как значения задержки округляются перед использованием в моделировании.

Спецификации задержки в проекте не поддаются синтезу, не могут быть преобразованы в аппаратную логику и используются только при моделировании.

Суффиксы единиц времени:

s	секунды	us	микросекунды	ps	пикосекунды
ms	миллисекунды	ns	наносекунды	fs	фемтосекунды

Целые числа в этих спецификациях могут быть только 1, 10 или 100, за которым следует суффикс из таблицы выше.

События изменения уровня сигнала

Изменения значений переменных или проводов могут использоваться в качестве ***события синхронизации*** для запуска выполнения других процедурных инструкций. ***Событие*** также может быть основано на изменении уровня сигнала, например, переходе в 0.

Для обозначения перехода сигнала в 0 или 1 используются следующие ключевые слова:

negedge	переход от 1 к X, Z или 0 и от X или Z к 0
posedge	переход от 0 к X, Z или 1 и от X или Z к 1
edge	эквивалентно `posedge <сигнал> or negedge <сигнал>`

Пример

```
module edge_tb;
  reg a, b, c, q;

  initial begin
    $monitor("Time [%0t]: a=%0b b=%0b c=%0b q=%0b", $time, a, b, c, q);
    a <= 0; b <= 0; c <= 0; q <= 0;
    #5 a = 1; c = 1;
    q = #5 a & b | c;
    #1 q = 0;
    #10;
  end

  always @(posedge a) $display("Time [%0t]: a=%0b [posedge]", $time, a);
  always @(negedge b) $display("Time [%0t]: b=%0b [negedge]", $time, b);
  always #2 begin
    c = @(posedge q) (a & b);
    $display("Time [%0t]: c=%0b [posedge =]", $time, c);
  end
  always #1 begin
    c = @(negedge q) (a | 1);
    $display("Time [%0t]: c=%0b [negedge =]", $time, c);
  end
  always @(edge a) begin
    $display("Time [%0t]: a=%0b [edge]", $time, a);
  end
  always @(posedge a or negedge a) begin
    $display("Time [%0t]: a=%0b [manual edge]", $time, a);
  end
endmodule
```

Результат выполнения

```
//начало
[0] a=0 [edge]
[0] a=0 [manual edge]
[0] b=0 [negedge]
[0] a=0 b=0 c=0 q=0
[5] a=1 [posedge]
[5] a=1 [edge]
```

```
// продолжение
[5] a=1 [manual edge]
[5] a=1 b=0 c=1 q=0
[10] c=0 [posedge =]
[10] a=1 b=0 c=0 q=1
[11] c=1 [negedge =]
[11] a=1 b=0 c=1 q=0
```

Именованные события (event)

В случае, если событие используется несколько раз в модуле, ему можно задать имя. Это позволяет упростить код и уменьшить вероятность ошибки при изменении события.

Для обозначения события используется ключевое слово `event`. Срабатывание события указывается через `->`. Ожидание события описывается как `@()`.

Пример

```
module named_events_tb;
  event e1;

  initial begin
    #10; $display("Triggering an event e1 at %0t", $time);
    ->e1;
  end
  initial begin
    @(e1) $display("Event e1 is triggered at %0t", $time);
  end
endmodule
```

Результат выполнения

```
Triggering an event e1 at 10
Event e1 is triggered at 10
```

Ожидание события (wait)

Конструкция `wait` ожидает, пока указанное условие станет истинным, блокируя выполнение дальнейших выражений этого процедурного блока.

Пример

```
module wait_tb;
  integer count = 0;

  initial begin
    $monitor("Time [%0t]: count = %0d", $time, count);
    repeat (5)
      #2 count++;
  end
  initial begin
    wait(count == 'd3);
    $display("{3} count has reached till %0d at time = %0t", count, $time);
    $finish;
  end
  initial begin
    wait(count == 'd4);
    $display("{4} count has reached till %0d at time = %0t", count, $time);
    $finish;
  end
endmodule
```

Результат выполнения

```
Time [0]: count = 0
Time [2]: count = 1
Time [4]: count = 2
{3} count has reached till 3 at time = 6
Time [6]: count = 3
```

Управляющие конструкции

if-else

Если выражение в `if` принимает значение ***true***, то будет выполнена первая инструкция. Если выражение принимает значение ***false*** и если существует блок `else`, то будет выполнен блок `else`.

Синтаксис

```
if (expression)
  [statement]

if (expression)
  [statement]
else
  [statement]
```

```
if (expression) begin
  [multiple statements]
end else begin
  [multiple statements]
end
```

```
if (expression)
  [statement]
else if (expression)
  [statement]
else
  [statement]
```

Пример

```
module if_else_tb;
  int x = 4;
  initial begin
    if (x == 3)
      $display("x is %0d", x);
    else if (x == 5)
      $display("x is %0d", x);
    else
      $display("x is neither 3 nor 5");
  end
endmodule
```

Результат выполнения
x is neither 3 nor 5

case

Оператор начинается с ключевого слова `case` и заканчивается ключевым словом `endcase`. Выражение в скобках будет вычислено один раз, после чего оно сравнивается со списком альтернатив в том порядке, в котором они записаны, и выполняются операторы, для которых альтернатива соответствует данному выражению.

Если ни один из элементов `case` не соответствует заданному выражению, выполняются инструкции внутри необязательный элемента `default`. Операторы `case` могут быть вложенными.

Синтаксис

```
case (<expression>)
  case_item1 :  <single statement>
  case_item2,
  case_item3 :  <single statement>
  case_item4 :  begin
```

```
                   <multiple statements>
                 end
  default     : <statement>
endcase
```

Пример

```
module mux_(input [2:0]  a, b, c, input [1:0]  sel, output reg [2:0]  out);
  always @(*) begin
    case(sel)
      2'b00   : out = a;
      2'b01   : out = b;
      2'b10   : out = c;
      2'bz    : out = 3'bz;
      2'bx    : out = 3'bx;
      default : out = 0;
    endcase
  end
endmodule

module case_tb;
  reg [2:0] aa, bb, cc, d;
  reg [1:0] sel;

  mux_ mux(aa, bb, cc, sel, d);
  initial begin
    $display("Time\t a \t b \t c \tout\tsel");
    $monitor("[%2t]\t%b\t%b\t%b\t%b\t%b", $time, aa, bb, cc, d, sel);
    #10 bb = 0; cc = 1;
    #10 sel = 0;
    #10 aa = 1;
    #10 sel = 1;
    #10 sel = 2;
    #10 cc = 2'bz;
    #10 sel = 5;
    #10 sel = 2;
    #10 sel = 2'bz;
    #10 sel = 2'bx;
  end
endmodule
```

Результат выполнения

```
Time     a   b   c   out sel
[ 0]    xxx xxx xxx xxx xx
[10]    xxx 000 001 xxx xx
[20]    xxx 000 001 xxx 00
[30]    001 000 001 001 00
[40]    001 000 001 000 01
[50]    001 000 001 001 10
```

```
[60]    001 000 0zz 0zz 10
[70]    001 000 0zz 000 01
[80]    001 000 0zz 0zz 10
[90]    001 000 0zz zzz zz
[100]   001 000 0zz xxx xx
```

forever

Цикл, исполняемый постоянно.

Синтаксис

forever
  [statement]
	forever begin
	  [multiple statements]
	end
		forever #[delay] begin
		  [multiple statements]
		end

Пример

```
module forever_tb;
  initial begin
    forever #1 begin
      $display("This will be printed forever ... %0t", $time);
    end
  end
  initial #5 $finish();
endmodule
```

Результат выполнения

```
This will be printed forever ... 1
This will be printed forever ... 2
This will be printed forever ... 3
This will be printed forever ... 4
This will be printed forever ... 5
```

repeat

Цикл выполняется заданное количество раз. Если значение выражения X или Z, то оно будет интерпретироваться как ложь.

Синтаксис

repeat ([num_of_times]) begin
  [statements]
end
	repeat ([num_of_times]) @([event]) begin
	  [statements]
	end

Пример

```
module repeat_tb;

  initial begin
    repeat(3) begin
      $display("This is a new iteration ...");
    end
  end
endmodule
```

Результат выполнения
This is a new iteration ... This is a new iteration ... This is a new iteration ...

while / do while

Цикл выполняется, пока условие истинно. Если условие с самого начала равно *false*, инструкции вообще не будут выполняться.

Синтаксис

while (expression) begin
  [statements]
end
	do begin
	  [statements]
	end while (expression)

Пример

```
module while_tb;
  integer i = 3;
  initial begin
      while (i > 0) begin
        $display("Iteration#%0d", i);
        i = i - 1;
      end
  end
endmodule
```

Результат выполнения
Iteration#3 Iteration#2 Iteration#1

for

Цикл с определёнными начальным состоянием счётчика цикла, условием выполнения и правилом изменения значения счётчика цикла. Тело цикла выполняется, пока условие истинно. Все параметры счётчика должны быть явно заданы.

Синтаксис

```
for (initial_assignment; condition; modifiyng_variable) begin
  [statements]
end
```

Пример

```
module for_tb;
  integer i = 3;
  initial begin
      for (i = 2; i < 5; i = i + 1) begin
        $display("For loop #%0d", i);
      end
    end
endmodule
```

Результат выполнения
For loop #2
For loop #3
For loop #4

foreach

Цикл перебирает элементы массива и выполняет тело цикла для каждого из них.

Синтаксис

```
foreach ( array ) begin
  [statements]
end
```

Пример

```
module foreach_tb;
   bit [7:0] array [8];

   initial begin
      foreach (array [index]) begin
         array[index] = index << 4 + (4-index);
      end
      foreach (array [index]) begin
         $display("array[%0d] = 0x%0d", index, array[index]);
      end
   end
endmodule
```

Результат выполнения	
// начало	// продолжение
array[0] = 0x0	array[4] = 0x64
array[1] = 0x128	array[5] = 0x40
array[2] = 0x128	array[6] = 0x24
array[3] = 0x96	array[7] = 0x14

function

Функции – подпрограммы, позволяющие описывать общий для нескольких модулей код один раз и затем его повторно использовать. Функции выполняются мгновенно, и в них нельзя устанавливать временные задержки. Из этого следует, что функции не могут запускать внутри себя ***задачи***, т.к. те могут содержать конструкции управления временем моделирования. Функции могут возвращать только одно значение, и в качестве аргументов у них могут быть только входные порты.

Синтаксис

```
function <type> <name> (<args>);
  <statements>
endfunction
```

Пример

```
module function_tb;
  int a, b;
  initial begin
    b = 1; a = sum(3, 6, b);
    $display("a = %0d, b = %0d ", a, b);
    $display("sum(3, 6, b) = %0d", sum(3, 6, b));
    $display("b = %0d", b);
  end

  function bit[7:0] sum (input int x, y, z);
    z++;
    return x+y;
  endfunction
endmodule
```

Результат выполнения

```
a = 9, b = 1
sum(3, 6, b) = 9
b = 1
```

task

Задачи, в отличие от ***функций***, могут возвращать несколько значений и содержать внутри себя конструкции управления временем. Как и у модулей, у задач порты могут быть трёх направлений: входные, выходные и двунаправленные.

Задачи делятся на ***статичные*** и ***автоматические***. По умолчанию все задачи – статичные. В ***статичных*** задачах все переменные, объявленные внутри, являются общими для всех вызовов этой задачи. В ***автоматических*** все переменные являются приватными, то есть личными для каждого запуска задачи.

В примере ниже в задаче создаётся переменная `i`, которая в случае статической задачи, общая для всех четырёх запусков задачи и по факту инициализируется в 0 один раз. В случае автоматической задачи переменная создаётся внутри каждой задачи, и все созданные `i` являются независимыми друг от друга.

Синтаксис

```
// Статичные задачи (static task)
task [name] (input [port_list], inout [port_list], output [port_list]);
  begin
    [statements]
  end
endtask

// Автоматические задачи (automatic task)
task automatic [name] (input [port_list], inout [port_list], output
[port_list]);
```

```
  begin
    [statements]
  end
endtask

// Задача с пустым листом портов
task [name] ();
  begin
    [statements]
  end
endtask
```

Пример

module static_task_tb;	module automatic_task_tb;

<pre>module static_task_tb;
  initial display();
  initial display();
  initial display();
  initial display();
 	
  task display();
    integer i = 0;
    i = i + 1;
    $display("i=%0d", i);
  endtask
endmodule</pre>
	<pre>module automatic_task_tb;
	  initial display();
	  initial display();
	  initial display();
	  initial display();
	
	  task automatic display();
	    integer i = 0;
	    i = i + 1;
	    $display("i=%0d", i);
	  endtask
	endmodule</pre>

Результат выполнения	
i=1 i=2 i=3 i=4	i=1 i=1 i=1 i=1

Системные функции и задачи

В этом разделе рассмотрена только часть встроенных функций.

$display(), $write()

Используются для отображения в консоли симулятора значения переменных, строк или выражений. Строка формата, задаваемая первым аргументом, работает как в языке C.

`$display` всегда переходит на новую строку после печати. Поэтому вызов без параметров – это печать пустой строки. `$write` не переходит на новую строку после печати.

Пример

```
module display_write_tb;
  reg [7:0] a, b;

  initial begin
      a = 8'd4;
      b = 8'd5;
      $write("Vars: ");
```

```
      $display("a = %0d; b = %0d", a, b);
  end
endmodule
```

Результат выполнения

```
Vars: a = 4; b = 5
```

$strobe()

Выводит текст после выполнения всех событий моделирования на текущем временном шаге. Новая строка автоматически добавляется к тексту. Имеет формат, аналогичный формату `$display`.

Пример

```
module strobe_tb;
  reg [7:0] a, b;
  initial begin
    a = 8'h2D; b = 8'h2D;
    #10; b <= a + 1;
    $display("[$display] Time [%0t]: a=0x%0h b=0x%0h", $time, a, b);
    $strobe ("[$strobe]  Time [%0t]: a=0x%0h b=0x%0h", $time, a, b);
    #1;
    $display("[$display] Time [%0t]: a=0x%0h b=0x%0h", $time, a, b);
    $strobe ("[$strobe]  Time [%0t]: a=0x%0h b=0x%0h", $time, a, b);
  end
endmodule
```

Результат выполнения

```
[$display] Time [10]: a=0x2d b=0x2d
[$strobe]  Time [10]: a=0x2d b=0x2e
[$display] Time [11]: a=0x2d b=0x2e
[$strobe]  Time [11]: a=0x2d b=0x2e
```

Стоит отметить, что `$strobe` показывает окончательное обновлённое значение переменной b в момент времени 10, которое равно 0x2E, а `$display` отображает текущее значение своих аргументов внутри такта.

$monitor(), $monitoron(), $monitoroff()

`$monitor` выводит в консоль текст каждый раз, когда изменяется один из аргументов. Новая строка автоматически добавляется к тексту.

Важным отличием от `$display` является то, что тот выводит данные только 1 раз, а `$monitor` – в конце каждого цикла симуляции.

`$monitoron` и `$monitoroff` управляют флагом, который включает и отключает мониторинг соответственно.

Пример

``` `timescale 10 ns / 1 ns module monitor_tb; logic set; ```	``` `timescale 10 ns / 1 ns module monitor_on_off_tb; logic set; ```

<table>
<tr>
<td> parameter p = 1.55;
  initial begin
    $monitor("Time [%3t]:[%4g]: set=%b",
$time, $realtime, set);
  end

  initial begin
    repeat (3) begin
      #p set = 0;
      #p set = 1;
    end
  end
endmodule</td>
<td> parameter p = 1.55;
  initial begin
    $monitor("Time [%3t]:[%4g]: set=%b",
$time, $realtime, set);
    #3 $monitoroff();
    #3 $monitoron();
  end

  initial begin
    repeat (3) begin
      #p set = 0;
      #p set = 1;
    end
  end
endmodule</td>
</tr>
<tr>
<td colspan="2">Результат выполнения</td>
</tr>
<tr>
<td>Time [0]:[0]: set=x
Time [20]:[1.6]: set=0
Time [30]:[3.2]: set=1
Time [50]:[4.8]: set=0
Time [60]:[6.4]: set=1
Time [80]:[8]: set=0
Time [100]:[9.6]: set=1</td>
<td>Time [0]:[0]: set=x
Time [20]:[1.6]: set=0
Time [60]:[6]: set=0
Time [60]:[6.4]: set=1
Time [80]:[8]: set=0
Time [100]:[9.6]: set=1</td>
</tr>
</table>

$dumpfile(), $dumpvars(), $dumpon, $dumpoff

`$dumpfile` используется для сохранения изменений в значениях переменных и проводов в файл, имя которого указано в качестве аргумента. `$dumpvars` отвечает за то, какие именно значения будут записаны в файл. `$dumpon` и `$dumpoff` включает и выключает флаг сохранения, аналогично флагу мониторинга в командах `$monitoron` и `$monitoroff`.

Без указания аргументов в `$dumpvars` будут записываться все значения в моделируемой системе. Если первым аргументом указан 0, за которым следует перечисление названий модулей, то будут записываться все значения этого модуля и всех модулей нижестоящих уровней, инстанцированных в текущем модуле. Если какой-либо модуль не создан этим модулем, то его переменные не будут учитываться. Помимо модуля верхнего уровня, можно указать модули, которые не инстанцированы в этом модуле.

Чтобы записывать только сигналы в определённом модуле, указывается первым аргументом 1 и вторым – название модуля.

Если указать первым аргументом 2, а затем имя модуля, то будут записываться все переменные указанного модуля и модулей, инстанцированных в том на один уровень ниже.

Синтаксис

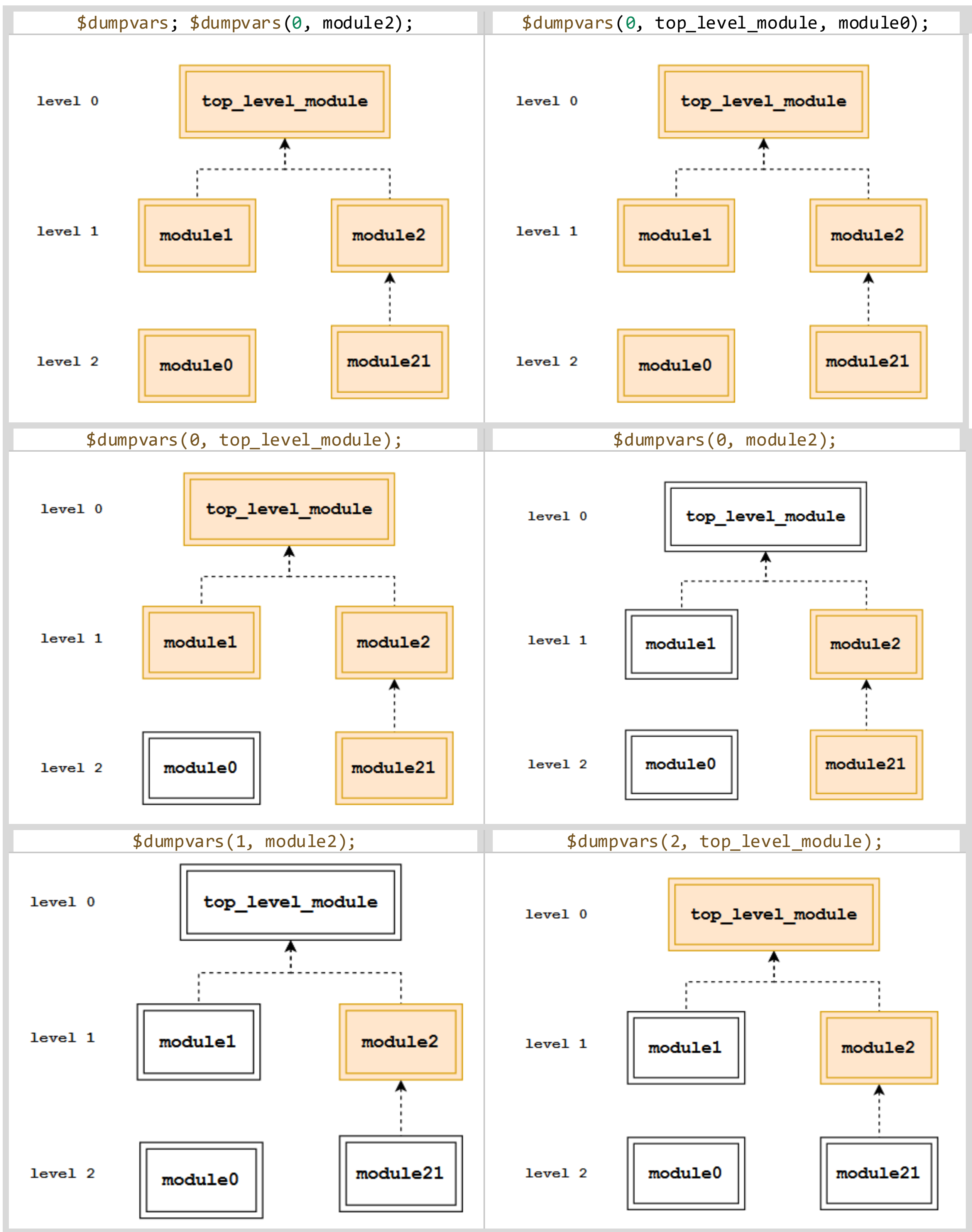
`$dumpfile (filename);`	`$dumpvars;`	`$dumpvars (level, scope);`

Пример

```
module dumpfile_tb;
  reg a, b, c, d;

  initial begin
    repeat (5) begin
      #1; c = b & a;
      d = a | b;
    end
  end

  initial begin
    #2 a = 0; b = 1;
    #2 a = 1'bz; b = 1'bz;
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, dumpfile_tb);
    #10 $finish;
  end
endmodule
```

Временная диаграмма

На примерах ниже стрелками обозначено, что модуль инстанцирован в другом. Выделение жёлтым показывает, значения из каких модулей будут записываться в файл.



$finish(), $stop(), $exit()

Системная команда `$stop` приводит к приостановке моделирования. Команда `$finish` приводит к завершению работы симулятора, а `$exit` – ожидает

завершения всех программных блоков, а затем выполняет неявный вызов `$finish`.

Синтаксис

$finish(code);	$stop(code);	$exit();

$time(), $realtime()

Возвращает текущее время моделирования в виде 64-разрядного целого числа без знака или в виде числа с плавающей точкой двойной точности.

Пример

```
module time_tb;
  initial begin
    $display("time = %0d", $time());
    $display("realtime = %0f", $realtime());
    #10 $display("time = %0d", $time());
    #2 $display("realtime = %0f", $realtime());
  end
endmodule
```

Результат выполнения

```
time = 0
realtime = 0.000000
time = 10
realtime = 12.000000
```

$timeformat(), $printformat()

Определяет формат, используемый спецификатором текстового формата `%t`. Единица измерения – целое число от 0 до -15, определяющее базовое время, которое должно отображаться:

0	1 sec	-4	100 us	-8	10 ns	-12	1 ps
-1	100 ms	-5	10 us	-9	1 ns	-13	100 fs
-2	10 ms	-6	1 us	-10	100 ps	-14	10 fs
-3	1 ms	-7	100 ns	-11	10 ps	-15	1 fs

Пример

```
// $timeformat(единица_измерения, точность, суффикс, мин_ширина_поля);
module timeformat_tb;
  initial begin
    $timeformat(-9, 2, " ns", 18);
    $display("time = %t", $time());
    #2 $display("time = %t", $time());
    $timeformat(-5, 2, " us", 18);
    #1 $display("time = %t", $time());
   end
endmodule
```

Результат выполнения
`time =            0.00 ns`
`time =   2000000000.00 ns`
`time =        300000.00 us`

$random()

Возвращает случайное 32-разрядное целое число со знаком. Аргумент ***seed*** – инициализирующее значение генератора псевдослучайных чисел типа `reg`, `integer` или `time`.

Пример

```
module random_tb;
  reg[31:0] seed1;
  integer seed2;

  initial begin
    seed1 = 8'b10010101; seed2 = 139;
    $display("random1 = %0d", $random(seed1));
    $display("random2 = %0d", $random(seed2));
    $display("random1 = %0d", $random());
    $display("random2 = %0d", $random());
  end
endmodule
```

Результат выполнения
`random1 = -2137191935`
`random2 = -2137882623`
`random1 = 303379748`
`random2 = -1064739199`

Работа с файлами

Verilog поддерживает функционал по работе с файловым вводом и выводом. Файл может быть открыт для чтения или записи с помощью команды `$fopen()`, которая возвращает 32-разрядный целочисленный дескриптор, называемый файловым дескриптором. Этот дескриптор можно использовать для чтения и записи в этот файл до тех пор, пока он не будет закрыт. Дескриптор файла может быть закрыт с помощью `$fclose()`. После закрытия дальнейшее чтение или запись через этот файловый дескриптор не допускается.

По умолчанию файл открывается в режиме записи. Файл также можно открыть в других режимах, указав тип режима. Ниже приведена часть режимов, в которых может быть открыт файл:

- "r" – открыть файл для чтения (файл должен существовать);
- "w" – открыть пустой файл для записи; если файл существует, то его содержимое удаляется;

- "a" – открыть файл для записи в конец (для добавления); файл создаётся, если он не существует

Для открытия в двоичном режиме используются "rb", "wb", "ab" и пр.

Системные команды `$fdisplay()` и `$fwrite()` могут использоваться для записи форматированной строки в файл. Первым их аргументом является дескриптор файла, а затем указываются данные, которые следует сохранить.

Чтобы прочитать файл, он должен быть открыт либо в режиме чтения r, либо в режиме чтения-записи r+. `$fgets()` считывает одну строку из файла.

`$feof()` возвращает значение ***true*** при достижении конца файла. Это может быть использовано в цикле для чтения всего содержимого файла.

`$fscanf()` работает по аналогии с функцией fscanf в языке C, считывая данные из файла, интерпретируя их согласно заданному формату и записывая значения в остальные аргументы.

Пример

```
module file_tb;
  integer fd, idx;
  string  str;

  initial begin
    fd = $fopen ("filename.ext", "w");
    for (int i = 0; i < 4; i++)
      $fdisplay (fd, "Iteration = %0d", i);
    $fclose(fd);

    fd = $fopen ("filename.ext", "r");
    $display("fd = %0d", fd);
    while ($fscanf (fd, "%s = %d", str, idx) == 2) begin
      $display("Line: %s = %0d", str, idx);
    end

    $fclose(fd);
  end
endmodule
```

Результат выполнения

```
fd = -2147483645
Line: Iteration = 0
Line: Iteration = 1
Line: Iteration = 2
Line: Iteration = 3
```

$readmemb(), $readmemh,

Инициализируют массив памяти значениями из файла. Файл должен быть текстовым файлом в кодировке ASCII со значениями, представленными в двоичной (`$readmemb`) или шестнадцатеричной системе счисления (`$readmemh`).

Значения данных должны быть той же ширины, что и массив памяти, и разделяться пробелами.

Пример

```
module readmem_tb();
  reg [7:0] hex_memory[0:15];
  initial begin
    $readmemh("hex.mem", hex_memory);
    $display("%h", hex_memory[0]);
  end
  reg [2:0] bin_memory[0:5];
  initial begin
    $readmemb("bin.mem", bin_memory);
    $display("%b", bin_memory[0]);
  end
endmodule
```

Результат выполнения

```
// если файлы открылись
ab
001
```

```
// если файлы не доступны на чтение
ERROR: test.v:4: $readmemh: Unable to open hex.mem for reading.
xx
ERROR: test.v:9: $readmemb: Unable to open bin.mem for reading.
xxx
```

bin.mem

```
001 101 111 111 101 001
```

hex.mem

```
ab cd ef 01  // this is a comment
ef 22 1e 00
9f ff 13 e6
ce b7 28 8f
```

Пример моделирования схемы

Verilog позволяет создать модель схемы (***design***) и тестовое окружение (***testbench***), в котором можно проверить корректность работы разработанной схемы.

Рассмотрим пример схемы, приведённой на рис. 31.

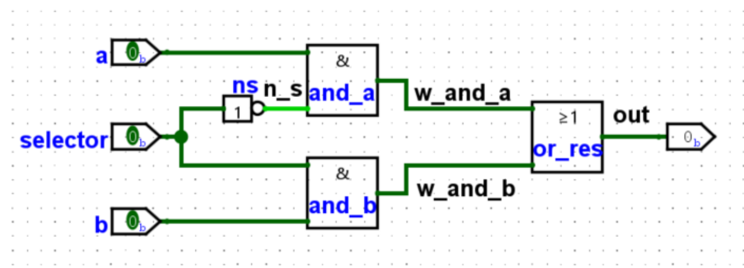


Рис. 31 – Пример схемы

Синим подписаны элементы схемы, черным – провода. Вместо выходного контакта, на котором можно наблюдать итоговое значение, в Verilog используется системная функция `$display`, которая выводит результат в консоль.

Приведённую схему можно описать на языке Verilog следующим образом:

```
module mux(output logic out, input logic a, s, b);
    or #2 or_res(out, w_and_a, w_and_b);
    and #2 and_a(w_and_a, a, n_s),
        and_b(w_and_b, b, s);
    not ns(n_s, s);
endmodule
```

В этом примере были инстанцированы логические элементы `and` и `or`, которые соединили между собой проводами. `#` означает временную задержку исполнения.

Порядок описания компонентов модуля не важен, поэтому следующие объявления модуля эквивалентны предыдущему:

```
module mux_v1(output logic out, input logic a, s, b);
    not ns(n_s, s);
    and #2 and_a(w_and_a, a, n_s),
        and_b(w_and_b, b, s);
    or #2 or_res(out, w_and_a, w_and_b);
endmodule

module mux_v2(output logic out, input logic a, s, b);
    and #2 and_a(w_and_a, a, n_s),
        and_b(w_and_b, b, s);
    not ns(n_s, s);
    or #2 or_res(out, w_and_a, w_and_b);
endmodule
```

Важно отметить, что в модели схемы не описывается, какие значения будут на входах. Это делается в модели тестового окружения или в других модулях, которые будут инстанцировать этот.

Схемы можно описывать на основе структурного или поведенческого подходов:

```
module mux_v3(output logic out, input logic a, s, b);
    assign #4 out = s ? b : a;
endmodule

module mux_v4(output logic out, input logic a, s, b);
    or #4 or_res(out, (a & !s) | (b & s));
endmodule

module mux_v5(output logic out, input logic a, s, b);
  always @(*) begin
    case(s)
      0         : out = #4 a;
      1         : out = #4 b;
```

```
      default  : out = #4 a;
    endcase
  end
endmodule
```

Пример тестового окружения для проверки работоспособности и корректности модуля:

```
module mux_tb;
  string display_format = "[%2t] %d  %d  %d %d";
  event display_states;
  logic a, sel, b, y;
  mux _mux(y, a, sel, b);

  initial begin
    $display("time a sel b y");
         a = 0; b = 0; sel = 0;
    #10; a = 1; b = 0; sel = 1;
    #10; a = 1; b = 1; sel = 1;
    #10; a = 0; b = 1; sel = 1;
    #10; a = 0; b = 1; sel = 0;
    #10; a = 0; b = 0; sel = 0;
    #10; $finish;
  end

  always @(display_states) $display(display_format, $time, a, sel, b, y);
  initial begin
     #3; ->display_states;
     #1; ->display_states;
     #1; ->display_states;
    #10; ->display_states;
    #10; ->display_states;
    #10; ->display_states;
    #10; ->display_states;
    #10; ->display_states;
    #10;
  end
endmodule
```

Результат выполнения

```
time a sel b y
[ 3] 0  0  0 x
[ 4] 0  0  0 0
[ 5] 0  0  0 0
[15] 1  1  0 0
[25] 1  1  1 1
[35] 0  1  1 1
[45] 0  0  1 0
[55] 0  0  0 0
```

Можно заметить, что в первой строке y имеет неопределённое состояние, т.к. присвоение значения в результате исполнения мультиплексора, которое занимает 4 такта, ещё не произошло. Начиная со следующей строки, сигнал

успел распространиться до выхода подсхемы и при выводе на экран выводится корректное значение.

Симуляция

Симулятор отслеживает время и изменяет значения входных сигналов в соответствии с описанным поведением/структурой модели и блока `initial` с задержками.

Время в симуляторе называется виртуальным. Виртуальное время не совпадает с реальным временем. Виртуальное время симулятора может быть ускорено или замедлено, чтобы ускорить или замедлить процесс симуляции. Виртуальное время симулятора может быть установлено в 0, чтобы начать симуляцию заново.

Изменение значения переменной в определённый момент времени называется событием обновления. Симулятор хранит события обновления в упорядоченном по времени их возникновения списке. Когда симулятор достигает момента времени, когда должно произойти событие обновления, он обновляет внутреннее состояние модели, распространяет новое состояние и удаляет событие из списка. Симулятор продолжает обновлять состояние модели, пока в списке есть события обновления.

В момент запуска симулятора обнуляется время, и всем переменным присваивается неопределённое значение x. Затем выбираются события обновления, запланированные на текущий момент времени, и выполняются (обновляется состояние модели и распространяются новые состояния). После этого симулятор исполняет модели и проверяет выходы на изменение. Если выходы изменились, то симулятор запланирует событие обновления в конец списка событий текущего такта. Если выходы не изменились, то симулятор просто переходит к следующему такту. И так далее, пока не будут выполнены все события обновления.

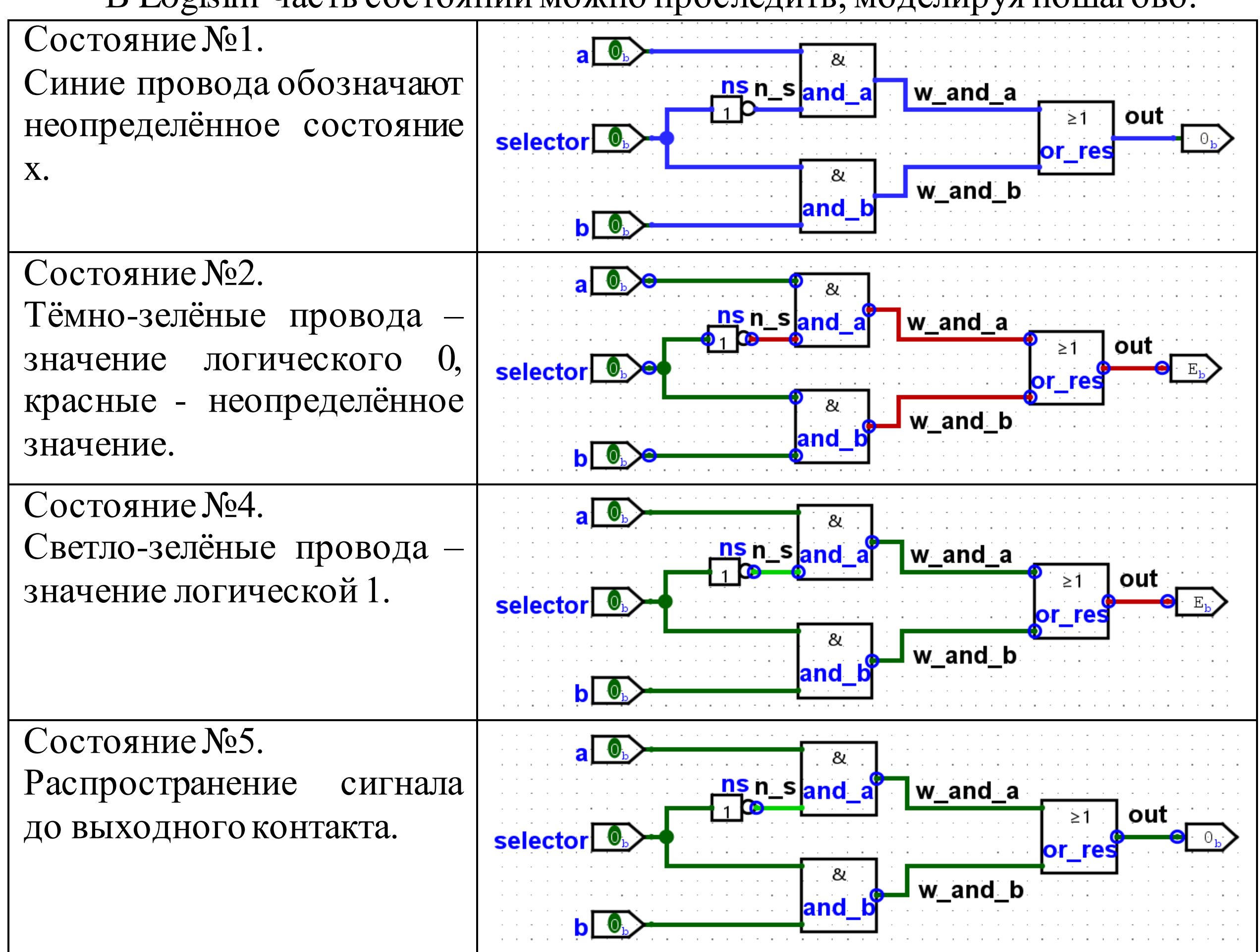
Для примера протестируем схему mux следующим модулем:

```
module mux_tb;
    reg  a, sel, b;
    wire y;
    mux mux1(y, a, sel, b);

    initial begin
        a = 0;
        b = 0;
        sel = 0;
        #12; a = 1;
        #6; $finish;
    end
endmodule
```

В терминах цикла симуляции выполнение вышеуказанного кода описывается следующим образом:

№	Время	Конец цикла симуляции	Список событий в конце цикла симуляции
1	Начало симуляции	-	Время 0: блок initial
2	0	1	Время 0: n_s = 1 Время 2: w_and_a = 0, w_and_b = 0 Время 12: возобновление initial, строка 6
3	0	2	Время 2: w_and_a = 0, w_and_b = 0 Время 12: возобновление initial, строка 6
4	2	1	Время 4: or_res = 0 Время 12: возобновление initial, строка 6
5	4	1	Время 12: возобновление initial, строка 6
6	12	1	Время 14: w_and_a = 1 Время 18: возобновление initial, строка 7
7	14	1	Время 16: or_res = 1 Время 18: возобновление initial, строка 7
8	16	1	Время 18: возобновление initial, строка 7

В Logisim часть состояний можно проследить, моделируя пошагово.

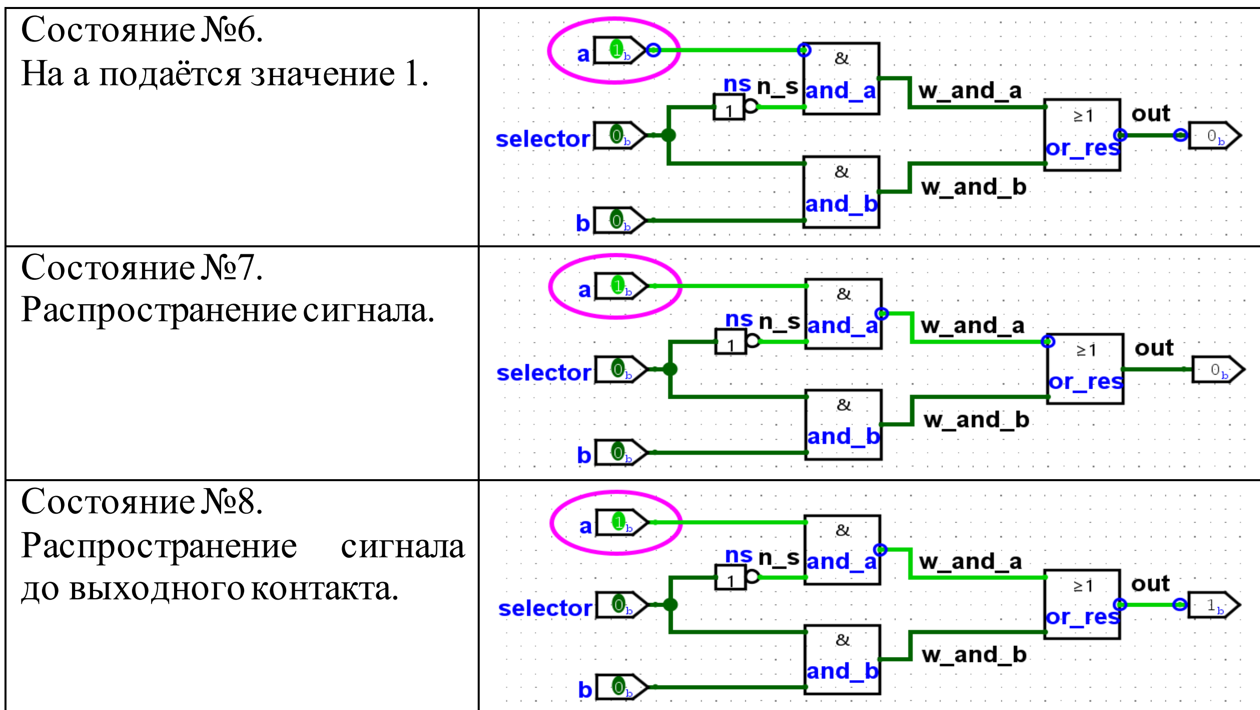


Состояние №1.
Синие провода обозначают неопределённое состояние x.

Состояние №2.
Тёмно-зелёные провода – значение логического 0, красные - неопределённое значение.

Состояние №4.
Светло-зелёные провода – значение логической 1.

Состояние №5.
Распространение сигнала до выходного контакта.



Состояние №6. На a подаётся значение 1.	
Состояние №7. Распространение сигнала.	
Состояние №8. Распространение сигнала до выходного контакта.	

Модель исполнения

Модель исполнения определяет, каким образом вычисляются новые значения на основе старых по времени.

В поведенческой модели исполнения вычисление новых значений осуществляется в соответствии с порядком, определённым в коде модуля. Т.е. операторы выполняются в том порядке, в котором они записаны, пока не встретился #. Когда это происходит, исполнение приостанавливается на указанное время. После этого в список событий добавляется новое событие исполнения. При наступлении соответствующего времени исполнение возобновляется с того места, где оно было приостановлено.

В структурной модели исполнения вычисления происходят при изменении значений на входах логических и электрических элементов. Если значение на выходе изменяется, то в список симуляции добавляется новое событие обновления. По наступлении этого события значение выхода будет обновлено и новое значение будет распространено далее по проводам. Это происходит до тех пор, пока все выходы не получат свои значения.

Симуляция позволяет проверить корректность работы модели как с функциональной точки зрения (выходных значений), так и с точки зрения времени (таймингов).

Список рекомендуемой литературы

1. Таненбаум Э., Остин Т. Архитектура компьютера // 6-е издание. – СПб.: Питер, 2013. – 811 с., илл. – ISBN: 978-5-496-00337-7.
2. Цифровая схемотехника и архитектура компьютера: RISC-V / пер. с англ. В. С. Яценкова, А. Ю. Романова; под ред. А. Ю. Романова. – М.: ДМК Пресс, 2021. – 810 с.
3. Логическое проектирование и верификация систем на SystemVerilog / пер. с анг. А. А. Слинкина, А. С. Камкина, М. М. Чупилко; науч. ред. А. С. Камкин, М. М. Чупилко. - М.: ДМК Пресс, 2019. - 384 с.
4. «IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language». IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009), pp.1-1315, 21 Feb. 2013, doi: 10.1109/IEEESTD.2013.6469140.
5. Руководство пользователя Logisim // Официальный сайт Logisim. URL: http://www.cburch.com/logisim/docs.html (дата обращения: 23.01.2023).

Яковлева Виктория Евгеньевна
Скаков Павел Сергеевич

Архитектура ЭВМ: учебно-методическое пособие по лабораторным работам

Учебно-методическое пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49, литер А