МИР электроники

А.Л. Переверзев,
М.Г. Попов,
А.П. Солодовников

Архитектуры процессорных систем. Практический курс

2-е исправленное издание

ТЕХНОСФЕРА
Москва
2025

УДК 004.2
ББК 32.97
П26

П26 Переверзев А.Л., Попов М.Г., Солодовников А.П.
Архитектуры процессорных систем. Практический курс
2-е исправленное издание
Москва: ТЕХНОСФЕРА, 2025. – 398 с. + 10с. цв. вкл. ISBN 978-5-94836-730-9

Лабораторный практикум ориентирован на разработку процессора с архитектурой RISC-V: читатель может пройти путь от азов разработки цифровых схем до проектирования однотактного процессора в составе микроконтроллера. К разработанной системе читатель сможет написать и скомпилировать программное обеспечение как на языке ассемблера, так и на языках высокого уровня (например, C++).

Освоив материал данной книги, читатель приобретёт практические навыки разработки цифровых схем на языке SystemVerilog, познакомится с основами работы с ПЛИС и системой автоматизированного проектирования Vivado, а также получит опыт программирования и понимание того, как исполняется программа на уровне аппаратуры.

Отличительной особенностью данного курса лабораторных работ является минимальное количество заготовленных компонентов. Благодаря выверенным методическим рекомендациям у читателя есть возможность самому создать процессорную систему, а не собрать её из готовых частей.

УДК 004.2
ББК 32.97

ISBN 978-5-94836-730-9

Содержание

IV. Основы Vivado

V. Выдержки из спецификации RISC-V

Предисловие

Настоящее издание открывает цикл сборников лекций и лабораторных практикумов, созданных в национальном исследовательском университете «Московский институт электронной техники» (НИУ МИЭТ) в рамках деятельности Передовой инженерной школы «Средства проектирования и производства электронной компонентной базы» (далее — ПИШ).

Реализация программы развития ПИШ осуществляется совместно с промышленными партнёрами, интересом которых является целевая и заказная подготовка дефицитных инженерных и управленческих кадров. На базе ПИШ создаются новые научно-образовательные пространства и лаборатории, в том числе для организации практики, стажировок и переподготовки кадров с учётом потребностей отрасли. Разрабатываются и реализуются новые образовательные программы с участием ведущих специалистов высокотехнологичных компаний.

Важной частью образовательных программ ПИШ является практика обучающихся в составе проектных команд, которые привлекаются к научным и инженерным проектам промышленных партнёров. Например, совместно с YADRO в 2024 году была разработана новая магистерская программа «Вычислительные системы и электронная компонентная база», в рамках которой осуществляется подготовка кадров, обладающих компетенциями в области сквозного проектирования сложнофункциональных блоков и систем на кристалле на базе открытой микропроцессорной архитектуры RISC-V.

Проектное обучение в магистратуре ведётся по трём направлениям: RTL-проектирование, функциональная верификация, топологическое проектирование. В рамках обучения в магистратуре каждый обучающийся получает актуальные знания от ведущих экспертов компании YADRO и преподавателей НИУ МИЭТ, а также проходит практику в лаборатории ПИШ «СФ-блоки и библиотеки», где решаются задачи в области расширения системы команд, а также оптимизации вычислительных ядер. При выполнении командного проекта по разработке экспериментальных образцов СФ-блоков и систем-на-кристалле обучающиеся имеют возможность изготовить в формате MPW (Multi Project Wafer) спроектированные микросхемы на отечественной фабрике и провести её экспериментальное исследование.

Кроме того, НИУ МИЭТ участвует в организации Школы синтеза цифровых схем, которая при поддержке YADRO действует по всей России и успешно решает свою задачу быстрого освоения современных подходов к проектированию цифровых микросхем для различных уровней подготовки — от школьников до инженеров. Также НИУ МИЭТ совместно с YADRO ежегодно проводит инженерный хакатон *SoC Design Challenge* для студентов и профессиональных инженеров по тематике проектирования СнК.

Таким образом, на базе ПИШ НИУ МИЭТ реализуется сквозной цикл мероприятий по подготовке кадров в области проектирования сложнофункциональных блоков и систем на кристалле на основе архитектуры RISC-V.

Представленный в книге лабораторный практикум является результатом более чем 20-летней эволюции курса АПС в рамках инженерной и научной

школы кафедры Вычислительной техники, а позже Института микроприборов и систем управления имени Л.Н. Преснухина. Сборник отражает один из трендов развития методологии проектирования вычислительной техники и систем управления: степень интеграции такой аппаратуры неуклонно повышается, а её разработчики заходят в область проектирования электронной компонентной базы.

Лабораторный практикум ориентирован на разработку процессора с архитектурой RISC-V: читатель может пройти путь от азов разработки цифровых схем до проектирования однотактного микроконтроллера с архитектурой RISC-V, а также написания и компиляции программного обеспечения для него. Несмотря на то что проектируемый процессор имеет исключительно академическое назначение, на нём вполне можно запускать простые игры, например «змейку»[1].

Освоив материал данной книги, вы приобретёте практические навыки разработки на языке SystemVerilog, познакомитесь с основами работы с ПЛИС и инструментами проектирования, такими как Vivado, а также получите опыт программирования на языке ассемблер. Таким образом, книга предоставляет уникальную возможность начать с нуля и шаг за шагом освоить основы проектирования процессоров и работы с ПЛИС, что может стать первым шагом к вашей карьере в области цифровой электроники.

Авторы надеются, что книга будет полезна как при самостоятельном изучении, так и при использовании в качестве пособия в университете. Желаем вам успехов в освоении данного материала лабораторного практикума «Архитектуры процессорных систем»!

История курса и разработчики

Дисциплины, связанные с организацией вычислительной техники, читаются в МИЭТ с самого его основания. Курс «Архитектуры процессорных систем» эволюционировал из курса «Микропроцессорные средства и системы» (МПСиС), читаемого на факультете Микроприборов и технической кибернетики сначала д.т.н., профессором Савченко Юрием Васильевичем, а после — директором Института микроприборов и систем управления, д.т.н. Перевертзевым Алексеем Леонидовичем. С 2014 по 2024 годы дисциплина значительно модернизировалась с участием преподавателей, сотрудников и студентов Института МПСУ, в частности существенный вклад внесли преподаватели Попов Михаил Геннадиевич и Солодовников Андрей Павлович.

В 2019—2024 годах была значительно переработана, осовременена и дополнена теоретическая часть курса. Разработаны и полностью обновлены лабораторные работы с переходом на использование архитектуры RISC-V. Все материалы курса, включая видеозаписи лекций[2], были выложены в свободный доступ.

С подготовкой курса и репозитория помогали студенты и сотрудники Института МПСУ и ПИШ НИУ МИЭТ (бывшие и нынешние):

[1] https://github.com/MPSU/APS/tree/master/Labs#обзор-лабораторных-работ

[2] https://www.youtube.com/@digital_machines

Фамилия, имя, отчество	**Вклад в курс**
Барков Евгений Сергеевич	Профессиональные консультации по деталям языка SystemVerilog, спецификации RISC-V и RTL-разработки, тематике синтеза и констрейнов.
Булавин Никита Сергеевич	Отработка материалов, подготовка тестбенчей и модулей верхнего уровня для плат Nexys A7 для лабораторных работ.
Козин Алексей Александрович	Отработка материалов, подготовка обфусцированных модулей для лабораторных работ.
Коршунов Андрей Владимирович	Профессиональные консультации по темам проектирования и синтеза цифровых схем.
Кулешов Владислав Константинович	Вычитка и исправление ошибок в методических материалах, сбор обратной связи от студентов.
Орлов Александр Николаевич	Профессиональные консультации по деталям языка SystemVerilog, спецификации RISC-V и RTL-разработки, примерам программ, иллюстрирующим особенности архитектуры.
Примаков Евгений Владимирович	Профессиональные консультации по деталям языка SystemVerilog, спецификации RISC-V и RTL-разработки и вопросам микроархитектуры.
Протасова Екатерина Андреевна	Подготовка индивидуальных заданий и допусков к лабораторным работам, вычитка и отработка материалов, а также сбор обратной связи от студентов.
Русановский Богдан Витальевич	Перенос лабораторной работы по прерываниям из PDF в Markdown, подготовка иллюстраций.
Рыжкова Дарья Васильевна	Подготовка тестбенчей для лабораторных работ.
Силантьев Александр Михайлович	Профессиональные консультации по деталям языка SystemVerilog, спецификации RISC-V и RTL-разработки, вопросам микроархитектуры, тематике синтеза и констрейнов, особенностям компиляции и профилирования.
Стрелков Даниил Владимирович	Отработка материалов, подготовка тестбенчей для лабораторных работ и иллюстраций структуры курса.
Терновой Николай Эдуардович	Профессиональные консультации по деталям языка SystemVerilog, спецификации RISC-V и RTL-разработки, вычитка материалов, сбор обратной связи от студентов.
Харламов Александр Александрович	Отработка материалов, проектирование вспомогательных модулей для лабораторных работ.
Хисамов Василь Тагирович	Вычитка материалов, сбор обратной связи от студентов.
Чусов Сергей Андреевич	Вычитка материалов, сбор обратной связи от студентов.

Иллюстрации были подготовлены Краснюк Екатериной Александровной.

Как читать эту книгу

Представленный здесь материал является сборником лабораторных работ, выполняемых студентами в НИУ МИЭТ в рамках дисциплины «Архитектуры процессорных систем» (АПС), и в первую очередь рассчитан именно на них, поэтому в тексте будут встречаться фразы вроде: «Допуск к лабораторной работе», «Проконсультироваться с преподавателем», которые имеют смысл только для обучающихся в вузе. Если вы читаете эту книгу для самостоятельного обучения, большую часть подобных фраз можно игнорировать.

В целом книга рассчитана на широкий охват аудитории по уровню их подготовки на момент начала прослушивания дисциплины АПС, поэтому для кого-то некоторые материалы окажутся избыточными, а для кого-то — крайне необходимыми.

Вне зависимости от вашего уровня подготовки, работу с этим курсом рекомендуется начать с прочтения раздела «Введение».

Далее можно приступать к разделу «Лабораторные работы». Перед каждым лабораторным занятием вам **рекомендуется** заранее ознакомиться с методическими материалами, т.к. они очень подробные и их чтение требует некоторого терпения. Время, отведённое на лабораторное занятие, рекомендуется использовать по максимуму: заниматься практической деятельностью, консультироваться с преподавателем, отлаживать разработанные блоки устройства и тому подобное, а для этого лучше прочитать методичку заранее.

Важно отметить, что в начале многих лабораторных работ приведены **дополнительные материалы для подготовки** со ссылками на главы раздела «Базовые конструкции SystemVerilog», которые студент **должен освоить** перед выполнением этой лабораторной работы. Данный раздел ориентирован в первую очередь на студентов, не работавших ранее с Verilog/SystemVerilog, однако, даже если вы работали с этими языками, рекомендуется пролистать данные главы и проверить свои знания в параграфе «Проверь себя».

Лабораторные занятия будут проходить с использованием САПР *Vivado* (и отладочных стендов *Nexys A7*). Это сложный профессиональный инструмент, однако основной информации по взаимодействию с САПР в разделе «Основы Vivado» хватит, чтобы с помощью *Vivado* реализовать весь цикл лабораторных работ.

Традиционно данный лабораторный практикум считают сложным, однако за годы отработки методических материалов в них было внесено множество уточнений и акцентов на действия, которые могут привести к ошибкам, добавлены ссылки на вспомогательные материалы. В данный момент авторы считают, что для успешного выполнения лабораторной работы от обучающегося требуется внимательно прочитать предоставленный ему материал и не бояться задать вопрос преподавателю, если что-то непонятно.

Если вы читаете данную книгу не в рамках курса АПС, вы вольны в выборе как программных средств, так и способов отладки. Репозиторий[1], сопровождающий эту книгу, будет содержать некоторые файлы, специализированные для плат Nexys A7 (так называемые *ограничения*/*constraints*), однако при

[1] https://github.com/MPSU/APS

должном уровне навыков вы с лёгкостью сможете портировать его под свою плату. В случае если вы будете использовать стороннюю плату, обратите внимание на число её переключателей, светодиодов и семисегментных индикаторов. В курсе их необходимо 16, 16 и 8 соответственно, поэтому, возможно, вам потребуется плата расширения.

Эта книга может быть интересна и полезна читателю, не имеющему никакой отладочной платы: проверка работоспособности осуществляется в первую очередь моделированием (на самом деле 90% времени вы будете проверять всё именно посредством моделирования).

В ходе выполнения лабораторных работ вы наверняка столкнётесь как с ошибками, связанными с работой Vivado, так и с ошибками описания на языке SystemVerilog. В первую очередь рекомендуется ознакомиться с текстом ошибки. В случае ошибок, связанных с языком SystemVerilog, чаще всего там содержится вся необходимая информация по её устранению. В случае если текст непонятен, рекомендуется ознакомиться со списком типичных ошибок[1].

Материал этой книги будет пестрить множеством ссылок, которые в электронной версии этой книги, разумеется, будут представлены в виде гиперссылок. Однако, если вы имеете удовольствие читать эту книгу в «аналоговом» формате, для вашего удобства все ссылки будут представлены в виде сносок под соответствующей страницей в текстовом формате.

Большая часть информации, касающаяся архитектуры RISC-V, взята напрямую из спецификации. Поскольку работа над спецификацией всё ещё идёт (хотя базовый набор инструкций rv32i уже заморожен и не изменится), чтобы ссылки на конкретные страницы спецификации имели смысл, они будут даваться на следующие версии двух документов:

- The RISC-V Instruction Set Manual Volume I: Unprivileged ISA — версия документа 20240411[2];
- The RISC-V Instruction Set Manual Volume II: Privileged Architecture — версия документа 20240411[3].

Курс лабораторных работ неразрывно связан с онлайн-репозиторием, расположенным по адресу: https://github.com/MPSU/APS. Этот репозиторий хранит методические материалы, верификационное окружение, готовые модули и файлы ограничений для отладочного стенда Nexys A7.

По всем вопросам / замечаниям / предложениям вы можете связаться с авторами курса через разделы *Issues* и *Discussions* данного репозитория.

Данный курс непрерывно эволюционировал на протяжении нескольких лет до самого своего издания. Авторы допускают, что где-то в тексте могли остаться некоторые недочёты, которые после издания тиража уже «не вырубишь топором». Для того чтобы дать читателям возможность узнать об ошибках, найденных уже после издания, в корне репозитория находится специальный документ *ERRATA.md*.

[1] https://github.com/MPSU/APS/blob/master/Other/FAQ.md

[2] https://github.com/riscv/riscv-isa-manual/releases/download/20240411/unpriv-isa-asciidoc.pdf

[3] https://github.com/riscv/riscv-isa-manual/releases/download/20240411/priv-isa-asciidoc.pdf

Как пользоваться репозиторием

В корне репозитория находятся следующие элементы (символом '/' на конце обозначены папки):

- .github/
- .pic/
- Basic Verilog structures/
- Introduction/
- Labs/
- Lectures/
- Other/
- Vivado Basics/
- .gitmodules
- ERRATA.md
- LICENSE
- README.md

Серым цветом обозначены элементы, которые не потребуются в ходе выполнения лабораторных работ.

В папках Introduction, Basic Verilog structures и Vivado Basics описаны разделы 1, 3 и 4 данной книги. Папка Other среди прочего содержит информацию, формирующую раздел 5 данной книги. Файл ERRATA.md содержит список ошибок, обнаруженных после публикации издания.

Рассмотрим структуру папки Labs:

01. Adder/
02. Arithmetic-logic unit/
03. Register file and memory/
04. Primitive programmable device/
05. Main decoder/
06. Main memory/
07. Datapath/
08. Load-store unit/
09. LSU Integration/
10. Interrupt subsystem/
11. Interrupt integration/
12. Daisy chain/
13. Peripheral units/
14. Programming/
15. Programming device/
16. CoreMark/

Made-up modules/

Readme.md

Здесь находятся методические материалы ко всем 16 лабораторным работам, разложенные по соответствующим им папкам.

Практически в каждой такой папке находится файл формата lab_xx.tb_xxx.sv, это файл с верификационным окружением для данной лабораторной работы. Такой файл необходимо добавлять в Simulation Sources проекта (подробней в разделе Vivado Basics).

Кроме того, в папке лабораторной работы могут находиться файлы xxx_pkg.sv и xxx.mem, содержащие параметры и данные, соответственно, которыми необходимо проинициализировать память устройства. Такие файлы будет необходимо добавлять в Design Sources проекта.

Также в большинстве папок будет находиться папка board files, которая содержит модуль верхнего уровня (если требуется), описание способов взаимодействия с ним и файлы ограничений (constraints) под отладочную плату Nexys A7.

Помимо прочего, в папке Made-up modules/ находятся готовые модули для некоторых лабораторных работ. В случае если по какой-то причине вы не смогли выполнить лабораторную работу, вы можете продолжить работу над курсом, использовав готовый модуль из этой папки.

У репозитория есть зеркало (копия сайта), расположенное по адресу: https://git-chips.miet.ru/MPSU/APS. Структура файлов в зеркале полностью совпадает с исходным репозиторием.

I. Введение

Что вас ждёт в этом разделе

Первая глава посвящена концепции использования **языков описания аппаратуры** (Hardware Description Languages, HDL). Это ключевая тема, поскольку в ходе работы вы будете описывать цифровые схемы в виде кода. У многих на этом этапе возникает ощущение, что они пишут программу. Однако важно осознать фундаментальное различие: описание цифровых схем предписывает каким устройство будет, а программирование предписывает какую последовательность действий нужно выполнить, это принципиально разные вещи. Если при работе над лабораторными заданиями вы будете использовать выражения вроде *«объявляем переменную»* или *«вызываем функцию»*, это станет сигналом того, вы по-прежнему пытаетесь писать программу. Такой взгляд может усложнить процесс освоения материала.

Лабораторный практикум построен таким образом, чтобы вы могли проверить свои разработки с помощью программируемой логической интегральной схемы (ПЛИС). Это устройство позволяет воспроизводить цифровые схемы, описанные вами на языке HDL. Отладочные платы с ПЛИС могут вызвать ассоциацию с аналогичными платами на основе процессоров и микроконтроллеров, например, Arduino. Чтобы этого избежать, **вторая глава** подробно объясняет, что такое ПЛИС и принципы их работы.

Третья глава посвящена классификации цифровой логики с акцентом на элементы последовательностной логики, или другими словами – устройств с памятью. Эти знания необходимы для понимания ключевых аспектов четвёртой главы, а также особенностей цифровой электроники, которые станут основой для выполнения лабораторных работ. Кроме того, **четвёртая глава** проведёт вас по маршруту преобразования текстового описания схемы в её физическое воспроизведение в ПЛИС.

Что такое язык описания аппаратуры (HDL)

На заре появления цифровой электроники цифровые схемы[1] в виде диаграммы на бумаге были маленькими, а их реализация в виде физической аппаратуры — большой. В процессе развития электроники (и её преобразования в микроэлектронику) цифровые схемы на бумаге становились всё больше, а относительный размер их реализации в виде физических микросхем — всё меньше. На *рисунке I.1-1* вы можете увидеть цифровую схему устройства Intel 4004, выпущенного в 1971 году.

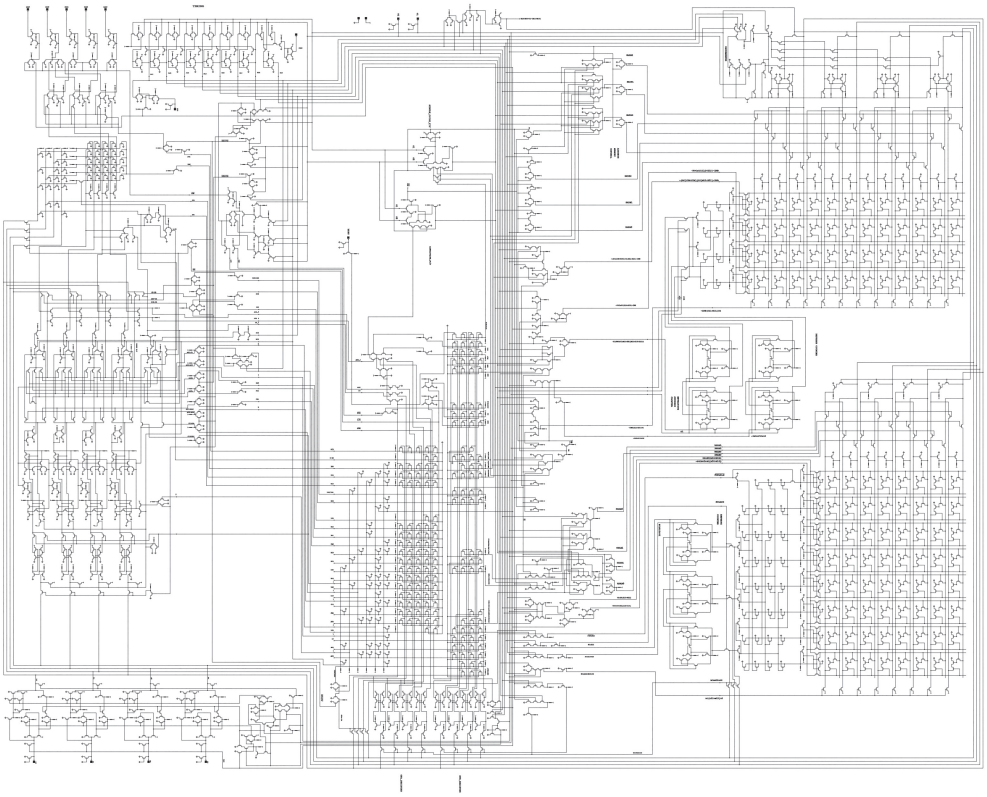


Рисунок I.1-1. *Цифровая схема процессора Intel 4004 на уровне транзисторов [1] (полноразмерный вариант рисунка представлен на вклейке 1)*

Данная микросхема состоит из 2300 транзисторов [2].

[1] Несмотря на то что данная глава пестрит термином «цифровая схема», определение этому термину будет дано только в начале следующей главы. Такое решение было принято ввиду того, что текущая глава проще и приоритетней к прочтению и не требует глубокого понимания данного термина, в то время как его определение более органично вписывается в следующую главу.

За прошедшие полсотни лет сложность цифровых схем выросла колоссально. Современные процессоры для настольных компьютеров состоят из десятков миллиардов транзисторов. *Рисунок I.1-1* при печати в оригинальном размере займёт прямоугольник размером 115 × 140 см с площадью около 1,6 м^2. Предполагая, что площадь печати имеет прямо пропорциональную зависимость от количества транзисторов, получим, что распечатка схемы современного процессора из 23 млрд транзисторов заняла бы площадь в 16 млн м^2, что эквивалентно квадрату со стороной в 4 км (см. *рисунок I.1-2*).

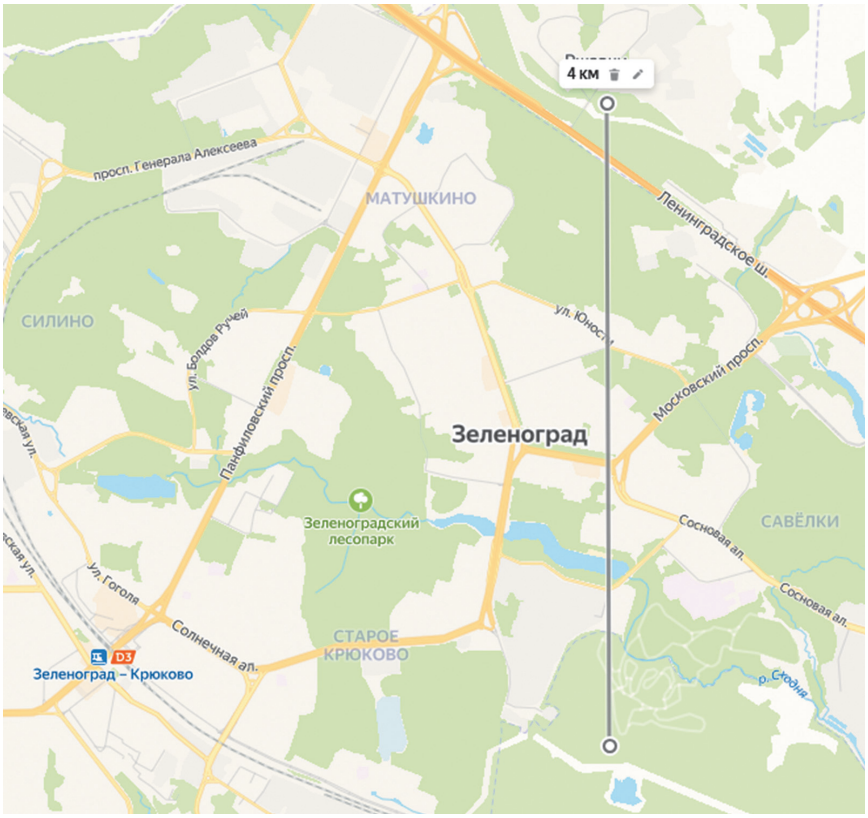


Рисунок I.1-2. *Масштаб размеров, которых могли бы достигать цифровые схемы на уровне транзисторов современных процессоров, если бы они печатались на бумаге*

Как вы можете догадаться, в какой-то момент между 1971 и 2024 годами инженеры перестали разрабатывать цифровые схемы, рисуя их на бумаге.

Разумеется, разрабатывая устройство, не обязательно вырисовывать на схеме каждый транзистор — можно управлять сложностью, переходя с одного уровня абстракции на другой. Например, начинать разработку схемы с уровня функциональных блоков, а затем рисовать схему для каждого отдельного блока.

К примеру, схему Intel 4004 можно представить *рисунком I.1-3.*

Однако, несмотря на это, даже отдельные блоки порой бывают довольно сложны. Возьмем блок аппаратного шифрования по алгоритму AES [3] на *рисунке I.1-4.*

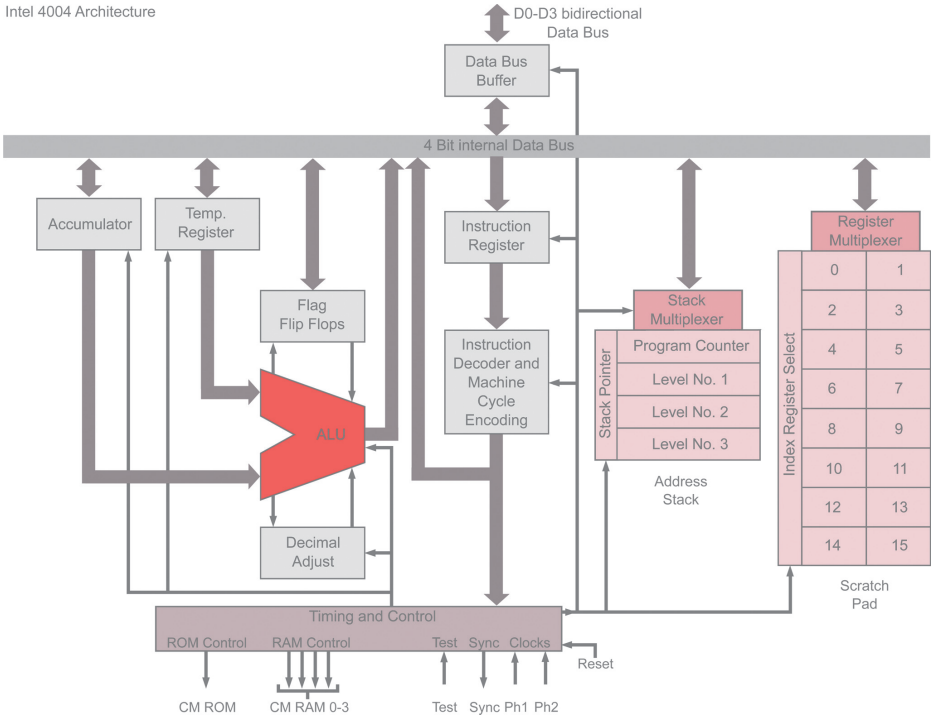


Рисунок I.1-3. *Цифровая схема процессора Intel 4004 на уровне функциональных блоков [2]*

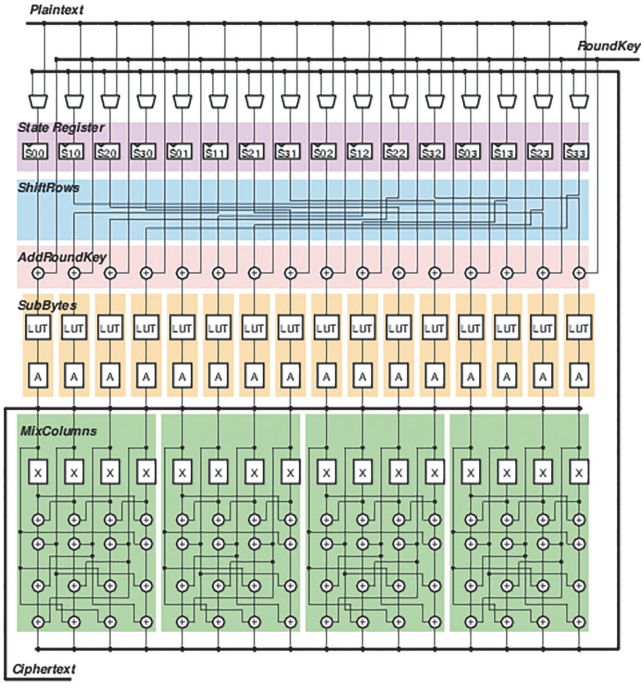


Рисунок I.1-4. *Цифровая схема блока аппаратного шифрования по алгоритму AES [4]*

Заметьте, что даже этот блок не представлен на уровне отдельных транзисторов. Каждая операция Исключающего ИЛИ, умножения, мультиплексирования сигнала и таблицы подстановки — это отдельные блоки, функционал которых ещё надо реализовать. В какой-то момент инженеры поняли, что проще описать цифровую схему в текстовом представлении, нежели в графическом.

Как можно описать цифровую схему текстом? Рассмотрим цифровую схему полусумматора, представленную на *рисунке I.1-5*.

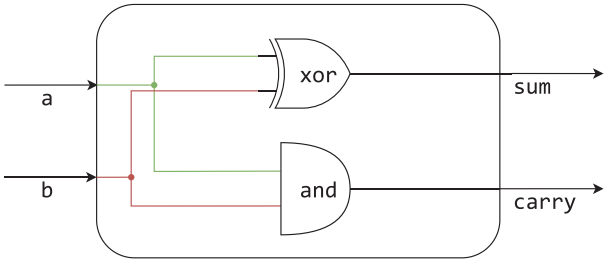


Рисунок I.1-5. *Цифровая схема полусумматора на уровне логических вентилей*

Это **устройство** (*полусумматор*) имеет два **входа** — *a* и *b*, а также два **выхода** — *sum* и *carry*. Выход *sum* является **результатом** логической операции **Исключающее ИЛИ** от операндов *a* и *b*. Выход *carry* является **результатом** логической операции **И** от операндов *a* и *b*.

Данный текст и является тем описанием, по которому можно воссоздать эту цифровую схему. Если стандартизировать описание схемы, то в нём можно будет оставить только слова, выделенные жирным и курсивом. Пример того, как можно было бы описать эту схему по стандарту IEEE 1364-2005[1] (языком описания аппаратуры [Hardware Description Language — HDL] Verilog) представлен в *листинге I.1-1*).

```
module half_sum(      // устройство полусумматор со
  input   a,          // входом a,
  input   b,          // входом b,
  output  sum,        // выходом sum и
  output  carry       // выходом carry.
);
assign sum = a ^ b;   // Где выход sum является результатом
                      // Исключающего ИЛИ от a и b,
assign carry = a & b;// а выход carry является результатом
                      // логического И от a и b.
endmodule
```

Листинг I.1-1. Развёрнутое описание модуля half_sum на языке Verilog

На первый взгляд, такое описание выглядит даже больше, чем записанное естественным языком, однако видимый объём получен только за счёт переноса строк и некоторой избыточности в описании входов и выходов, которая

[1] https://ieeexplore.ieee.org/document/1620780

была добавлена для повышения читаемости. То же самое описание можно было записать и в виде, представленном в *листинге I.1-2*.

```
module half_sum(input a, b, output sum, carry);
  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

Листинг I.1-2. Компактное описание модуля half_sum на языке Verilog

Важно отметить, что код на языке Verilog описывает устройство целиком, одномоментно. Это **описание** *рисунка I.1-5*, а **не построчное выполнение программы**.

С практикой описание схемы в текстовом виде становится намного проще и не требует графического представления. Для описания достаточно только спецификации — формальной записи того, как должно работать устройство. По ней разрабатывается алгоритм, который затем претворяется в описание на HDL.

Занятный факт: ранее было высказано предположение о том, что инженеры перестали разрабатывать устройства, рисуя цифровые схемы в промежуток времени между 1971 и 2024 годами. Так вот, первая конференция, посвящённая языкам описания аппаратуры, состоялась в 1973 году [5, стр. 8]. Таким образом, Intel 4004 можно считать одним из последних цифровых устройств, разработанных без использования языков описания аппаратуры.

Список источников

1. Intel 4004. 50th Anniversary Project. URL: https://www.4004.com/mcs4-masks-schematics-sim.html
2. Wikipedia. Intel 4004. URL: https://en.wikipedia.org/wiki/Intel_4004
3. FIPS 197, Advanced Encryption Standart (AES). URL: https://csrc.nist.gov/files/pubs/fips/197/final/docs/fips-197.pdf
4. F. Kağan Gürkaynak / Side Channel Attack Secure Cryptographic Accelerators. URL: https://iis-people.ee.ethz.ch/~kgf/acacia/acacia_thesis.pdf
5. P. Flake, P. Moorby, S. Golson, A. Salz, S. Davidmann. Verilog HDL and Its Ancestors and Descendants. URL: https://dl.acm.org/doi/pdf/10.1145/3386337

Что такое ПЛИС и как она работает

Параграфы «Цифровые схемы и логические вентили» и «Таблицы подстановки» во многом используют материалы статьи «How Does an FPGA Work?»[1] за авторством Alchitry, Ell C, распространяемой по лицензии CC BY-SA 4.0[1].

История появления ПЛИС

До появления интегральных схем электронные устройства строились на базе вакуумных ламп, которые выполняли функции усиления и переключения. Эти лампы были громоздкими, энергозатратными и недолговечными. Затем их заменили на транзисторы, которые стали основой современных электронных схем. Поначалу транзисторы, как и лампы, использовались в виде отдельных компонентов и схемы собирались из них, как модель из кубиков Lego. В случае ошибки её можно было исправить ручной корректировкой соединений между элементами, подобно исправлению ошибки при сборке модели Lego.

С развитием технологий произошла миниатюризация транзисторов, что позволило разместить их вместе с соединениями на одном кристалле. Так появились интегральные схемы — электронные схемы, выполненные на полупроводниковой подложке и заключённые в неразборный корпус. Этот переход стал революционным шагом в развитии электроники, открыв путь к созданию компактных и производительных устройств.

В большинстве случаев исправить ошибку, допущенную при разработке и изготовлении интегральной схемы, невозможно. С учётом того, что изготовление прототипа интегральной схемы является долгим и затратным мероприятием (от десятков тысяч до миллионов долларов), возникла необходимость в гибком, быстром и дешёвом способе изготовления прототипа и проверки на нём схемы до её изготовления. Так появились **программируемые логические интегральные схемы** (**ПЛИС**). В связи с повсеместным использованием англоязычной литературы имеет смысл дать и англоязычное название этого класса устройств — **programmable logic devices** (**PLD**).

Стоит оговориться, что в данной книге под термином ПЛИС будет подразумеваться конкретный тип программируемых схем: **FPGA** (**field-programmable gate array**, **программируемая пользователем вентильная матрица**, **ППВМ**).

ПЛИС содержит некоторое конечное множество базовых блоков (примитивов), блоки межсоединений примитивов и блоки ввода-вывода. Подав определённый набор воздействий на ПЛИС (**запрограммировав** её), можно настроить примитивы, их межсоединения между собой и блоками ввода-вывода, чтобы получить требуемую цифровую схему. Удобство ПЛИС заключается в том, что в случае обнаружения ошибки на прототипе, исполненном в ПЛИС, вы можете исправить свою цифровую схему и повторно запрограммировать ПЛИС.

[1] https://creativecommons.org/licenses/by-sa/4.0/

Кроме того, эффективно использовать ПЛИС не только как средство относительно дешёвого прототипирования, но и как средство реализации конечного продукта в случае малого тиража (дешевле купить и запрограммировать готовую партию ПЛИС, чем изготовить партию собственных микросхем).

Давайте разберёмся, что же это за устройство и как оно работает, но перед этим необходимо провести ликбез по цифровым схемам и логическим вентилям.

Цифровые схемы и логические вентили

Цифровые схемы

Цифровая схема — это **абстрактная модель** вычислений, которая оперирует двумя дискретными состояниями, обычно обозначаемыми как *0* и *1*. Важно понимать, что эти состояния не привязаны к конкретным физическим величинам, таким как напряжение в электрической цепи. Вместо этого они представляют собой обобщённые логические значения, которые могут быть реализованы на любой технологии, способной различать два дискретных состояния.

Благодаря этой абстракции цифровые схемы могут быть реализованы не только с помощью традиционных электронных компонентов, но и на совершенно иных платформах, например на пневматических системах[1], из картона и шариков[2], красной пыли[3] в игре Майнкрафт или даже из людей подобно тому, как это описано в романе Лю Цысиня «Задача трёх тел» (эффективность подобных схем — это уже другой вопрос). Основная идея заключается в том, что цифровая схема отвязывается от физической реализации, фокусируясь лишь на логике взаимодействия состояний *0* и *1*, что делает её универсальной и независимой от конкретной технологии.

Разумеется, при проектировании эффективных цифровых схем необходимо учитывать технологию, по которой эти схемы будут работать.

В электронике словом «цифровая» описывают схемы, которые абстрагируются от непрерывных (аналоговых) значений напряжений, вместо этого используются только два дискретных значения — 0 и 1. На данном уровне абстракции нас не интересуют конкретные значения напряжений и пороги этих значений, что позволяет нам разрабатывать схему в идеальном мире, где у напряжения может быть всего два значения — 0 и 1. А обеспечением этих условий будут заниматься базовые блоки, из которых мы будем строить цифровые схемы.

Эти базовые блоки называются **логическими вентилями**.

Логические вентили

Существует множество логических вентилей, мы рассмотрим четыре из них: **И**, **ИЛИ**, **Исключающее ИЛИ**, **НЕ**. Каждый из этих элементов принимает на вход **цифровое значение** (см. **цифровая схема**), выполняет определённую

[1] https://habr.com/ru/companies/ruvds/articles/692236/

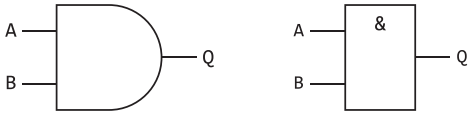
[2] https://habr.com/ru/articles/399391/

[3] https://minecraft.fandom.com/wiki/Tutorials/Redstone_computers

логическую функцию над входами и подаёт на выход результат этой функции в виде **цифрового значения**.

Логические вентили на *рисунках с I.2-1 по I.2-4* иллюстрируются условными графическими обозначениями (**УГО**), взятыми из двух стандартов — **ANSI** и **ГОСТ**. Ввиду частого использования в литературе и документации первого варианта в дальнейшем в книге будет использован он.

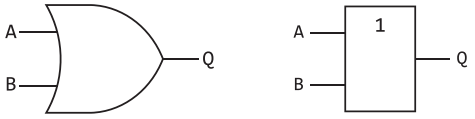
Логический вентиль **И** принимает два входа и выдаёт на выход значение *1* только в том случае, если оба входа равны *1*. Если хотя бы один из входов *0*, то на выходе будет *0*. На схемах логический вентиль **И** отображается следующим образом:



Входы		Выход
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

Рисунок I.2-1**. УГО логического вентиля **И

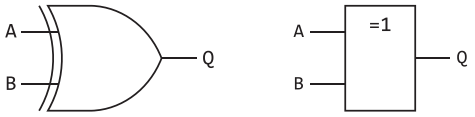
Логический вентиль **ИЛИ** принимает два входа и выдаёт на выход значение *1* в случае, если хотя бы один из входов равен *1*. Если оба входа равны *0*, то на выходе будет *0*. На схемах логический вентиль **ИЛИ** отображается следующим образом:



Входы		Выход
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

Рисунок I.2-2**. УГО логического вентиля **ИЛИ

Логический вентиль **Исключающее ИЛИ** принимает два входа и выдаёт на выход значение *1* в случае, если значения входов не равны между собой (один из них равен *1*, а другой — *0*). Если значения входов равны между собой (оба равны *0* или оба равны *1*), то на выходе будет *0*. На схемах логический вентиль **Исключающее ИЛИ** отображается следующим образом:

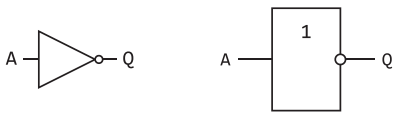


Входы		Выход
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Рисунок I.2-3. УГО логического вентиля **Исключающее ИЛИ

Логический вентиль **НЕ** самый простой. Он принимает один вход и подаёт на выход его инверсию. Если на вход пришло значение *0*, то на выходе будет

1, если на вход пришло значение *1*, то на выходе будет *0*. Он обозначается на схемах следующим образом:



Input	Output
A	Q
0	1
1	0

Рисунок I.2-4**. УГО логического вентиля **НЕ

Также существуют вариации базовых вентилей, такие как **И-НЕ**, **ИЛИ-НЕ**, **Исключающее ИЛИ-НЕ**, отличающиеся от исходных тем, что результат операции инвертирован относительно результата аналогичной операции без **-НЕ**.

Логические вентили строятся из **транзисторов**. **Транзистор** — это элемент, который может пропускать/блокировать ток в зависимости от поданного напряжения на его управляющий вход.

Особенностью современных интегральных схем является то, что они строятся на основе комплементарной (взаимодополняющей) пары транзисторов **P**- и **N**-типа (**комплементарная металл-оксид-полупроводниковая**, **КМОП**-логика). Для данного типа транзисторов оказалось эффективнее реализовать операции **И-НЕ** и **ИЛИ-НЕ**.

С точки зрения построения цифровых схем МОП-транзисторы (**P**- и **N**-типа) можно воспринимать как выключатели, которые замыкают или размыкают связь между двумя выводами. Разница между **P**- и **N**-типами заключается в значении напряжения на управляющем входе, при котором транзистор «открыт» (вход и выход замкнуты) или «закрыт» (связь разорвана). *Рисунок I.2-5* иллюстрирует данное различие.

Вход и выход, между которыми образуется связь, называются «**сток**» (**drain**, **d**) и «**исток**» (**source**, **s**), а управляющий вход — «**затвор**» (**gate**, **g**). Логический вентиль (**logic gate**) и затвор транзистора (**gate**) — это разные сущности!

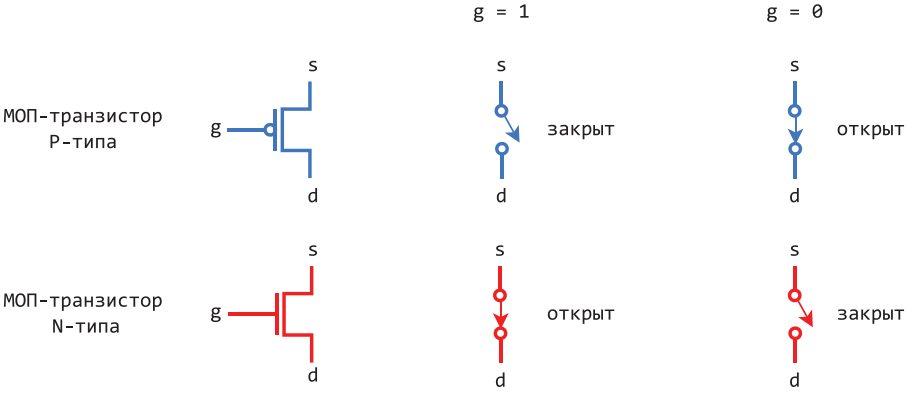


***Рисунок I.2-5**. МОП-транзисторы P- и N-типа*

На *рисунке I.2-6* показан способ построения логических вентилей **И-НЕ**, **ИЛИ-НЕ** по **КМОП**-технологии. Рассмотрим принцип работы вентиля **И-НЕ**.

Подача значения *1* на вход **A** или **B** открывает соответствующий этому входу n-канальный транзистор (обозначен на *рисунке I.2-6* красным цветом) и

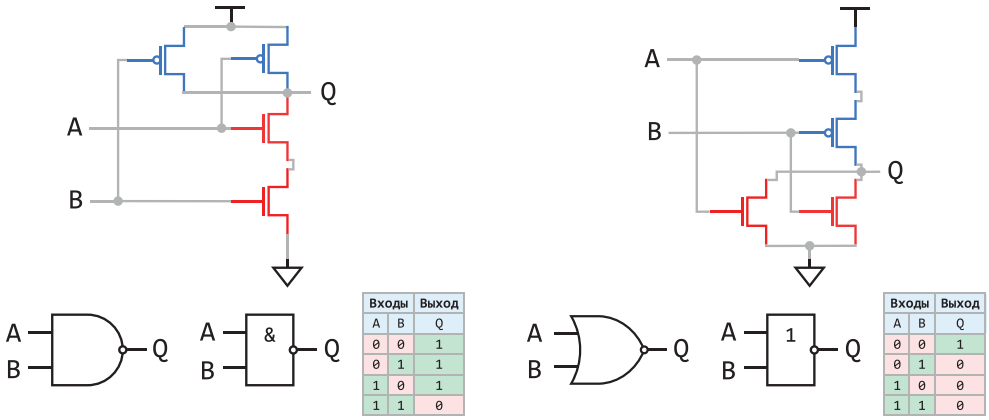


***Рисунок I.2-6**. Схема логических вентилей **И-НЕ**, **ИЛИ-НЕ**, построенных на КМОП-транзисторах*

закрывает дополняющий его (комплементарный ему) p-канальный транзистор (обозначен синим цветом). Подача на оба входа *1* закрывает оба p-канальных транзистора (верхняя часть схемы разомкнута, что для значения на выходе означает, что её будто и нет) и открывает оба n-канальных транзистора. В результате выход замыкается на «землю» (чёрный треугольник внизу схемы), что эквивалентно *0* в контексте цифровых значений.

В случае если хотя бы на одном из входов **A** или **B** будет значение *0*, откроется один из параллельно соединённых p-канальных транзисторов (в то время как соединение с «землёй» будет разорвано) и выход будет подключён к питанию (две перпендикулярные линии вверху схемы), что эквивалентно *1* в контексте цифровых значений.

Как вы видите, напряжение на выход подаётся от **источников постоянного питания** или **земли**, а не от входов вентиля, именно этим и обеспечиваются постоянное обновление напряжения и устойчивость **цифровых схем** к помехам.

Как правило, при необходимости инвертировать вход или выход логического элемента на схеме на нём рисуют кружок вместо добавления логического вентиля **НЕ** в том виде, котором он изображён на *рисунке I.2-4*. К примеру, логический элемент **И-НЕ** обозначают в виде, представленном на *рисунке I.2-6*.

При желании из логического элемента **И-НЕ** можно легко получить логический элемент **И** (как и элемент **ИЛИ** из **ИЛИ-НЕ**). Для этого необходимо поставить на выходе **И-НЕ** инвертор, собираемый из двух МОП-транзисторов по схеме, представленной на *рисунке I.2-7*.

КМОП-логика — далеко не единственный способ построения цифровых элементов, ранее достаточно широко применялись другие варианты построения схем, например только на одном типе транзисторов. Однако наиболее эффективным оказалось использование именно комплементарных пар, на сегодня такой подход для цифровых схем является доминирующим.

Используя одни лишь описанные выше логические вентили, можно построить **любую(!)** цифровую схему.

При описании цифровых схем некоторые цифровые блоки используются настолько часто, что для них ввели отдельные обозначения (**сумматоры**, **умно-**

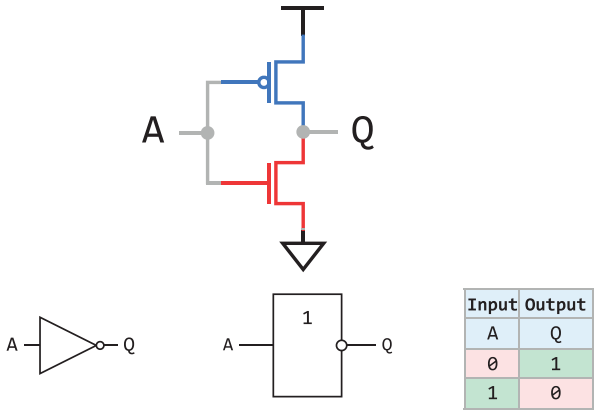


Рисунок I.2-7. *Схема логического вентиля* ***НЕ,*** *построенного на КМОП-транзисторах*

жители, **мультиплексоры** т.п.). Рассмотрим один из фундаментальных строительных блоков в ПЛИС — **мультиплексор**.

Мультиплексоры

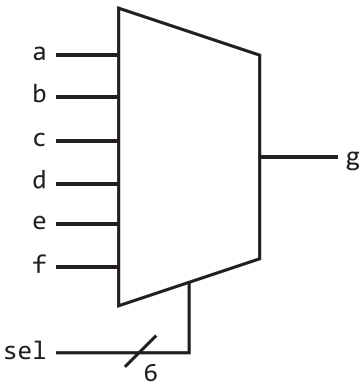


Рисунок I.2-8. *УГО мультиплексора*

Мультиплексор — это устройство, которое в зависимости от значения **управляющего сигнала** подаёт на выход значение одного из входных сигналов.

УГО мультиплексора представлено на *рисунке I.2-8*. Символ / на линии *sel* указывает на то, что данный сигнал является многоразрядным, а число ниже указывает на то, что его разрядность составляет 6 бит. Число входов мультиплексора может быть различным, но выход у него всегда один.

Способ, которым кодируется значение управляющего сигнала, может также различаться. Простейшая цифровая схема мультиплексора получится, если использовать унитарное[1] (**one-hot**) кодирование. При таком кодировании значение **многоразрядного** управляющего сигнала **всегда** содержит **ровно одну** *1*. Информация, которую несёт закодированный таким образом сигнал, содержится в положении этой *1* внутри управляющего сигнала.

Посмотрим, как можно реализовать мультиплексор с управляющим сигналом, использующим one-hot-кодирование, при помощи одних лишь логических вентилей **И**, **ИЛИ** (*рисунок I.2-9*).

Если мы выставим значение управляющего сигнала, равное *000010*, означающее, что только **первый** бит этого сигнала (**счёт ведётся с нуля**) будет равен **единице** (*sel[1] = 1*), то увидим, что на один из входов каждого логиче-

[1] https://ru.wikipedia.org/wiki/Унитарный_код

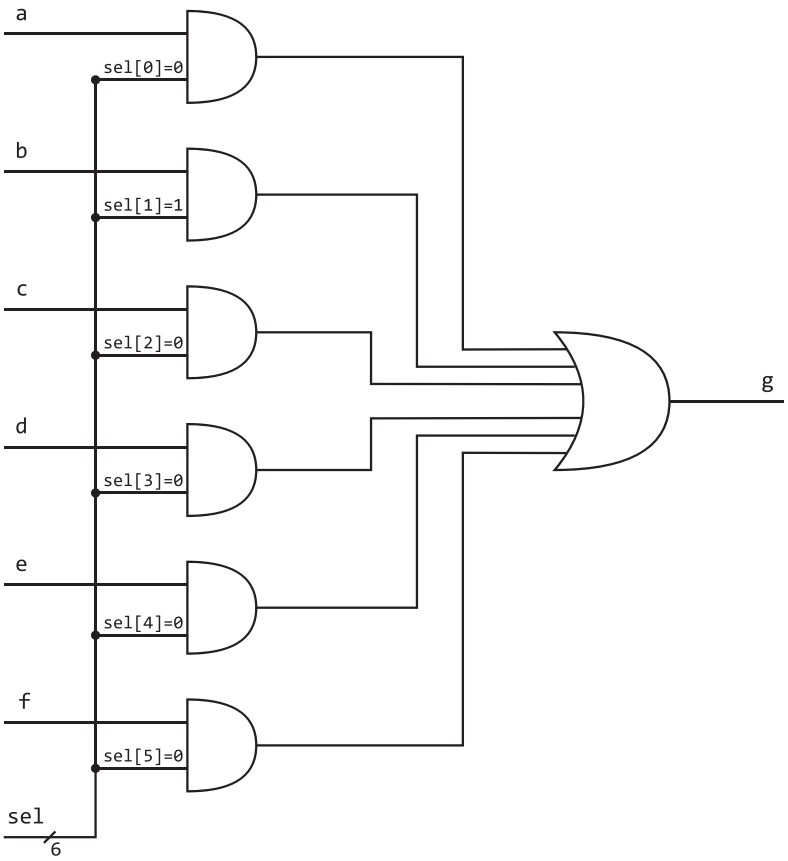


***Рисунок I.2-9**. Реализация мультиплексора, использующего one-hot-кодирование*

ского вентиля **И** будет подано значение *0*. Исключением будет логический вентиль **И** для входа *b*, на вход которого будет подано значение *1*. Это означает, что все логические вентили **И** (кроме первого, на который подаётся вход *b*) будут выдавать на выход *0* (см. стр. 23) вне зависимости от того, что было подано на входы *a*, *c*, *d*, *e* и *f*. Единственным входом, который будет влиять на работу схемы, окажется вход *b*. Когда он равен *1*, на выходе соответствующего логического вентиля **И** окажется значение *1*. Когда он равен *0*, на выходе **И** окажется значение *0*. Иными словами, выход **И** будет повторять значение *b* (*рисунок I.2-10*).

Логический вентиль **ИЛИ** на данной схеме имеет больше двух входов. Подобный вентиль может быть создан в виде каскада логических вентилей **ИЛИ** (*рисунок I.2-11*).

Многовходовой вентиль ИЛИ ведёт себя ровно так же, как двухвходовой: он выдаёт на выход значение *1*, когда хотя бы один из входов равен *1*. В случае если все входы равны *0*, на выход **ИЛИ** пойдёт *0*.

Для нашей схемы мультиплексора гарантируется, что каждый вход **ИЛИ**, кроме одного, будет равняться *0* (поскольку выход каждого **И**, кроме одного, будет равен *0*). Это означает, что выход **многовходового ИЛИ** будет зависеть только от **одного** входа (в случае когда *sel = 000010*, — от входа *b* (*рисунок I.2-12*)).

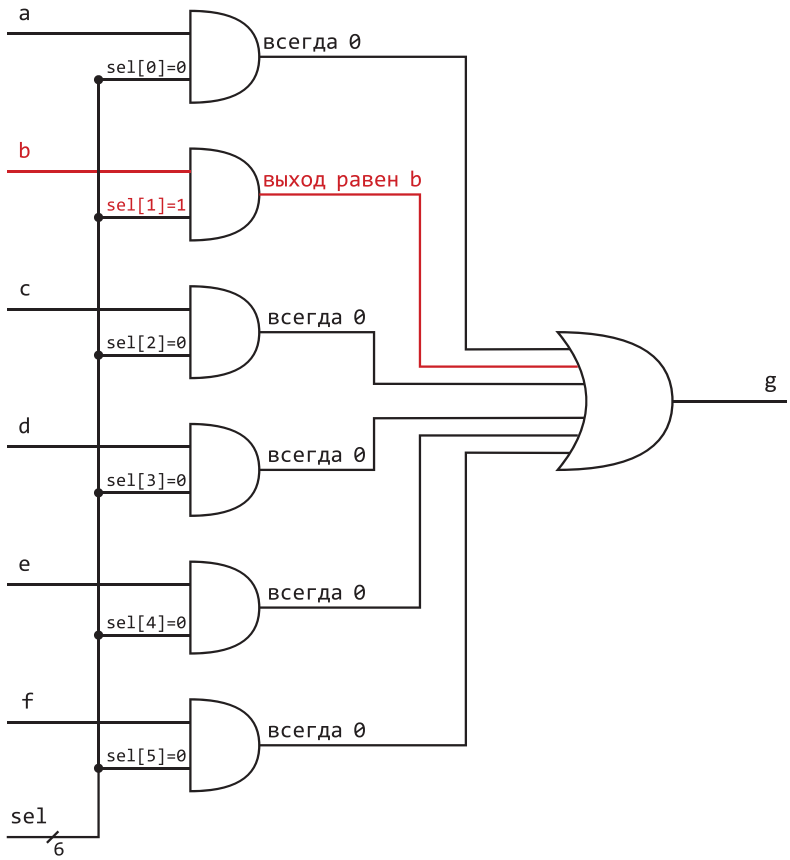


***Рисунок I.2-10**. Реализация мультиплексора, использующего one-hot-кодирование*

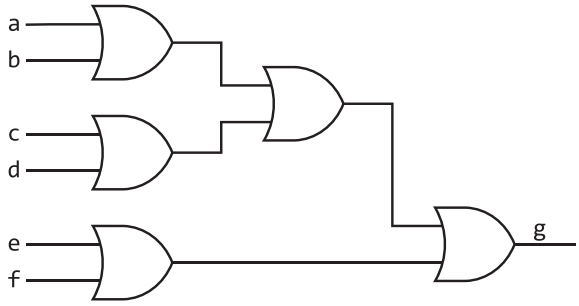


Рисунок I.2-11**. Реализация многоходового логического **ИЛИ

Меняя значение *sel*, мы можем управлять тем, какой из входов мультиплексора будет идти на его выход.

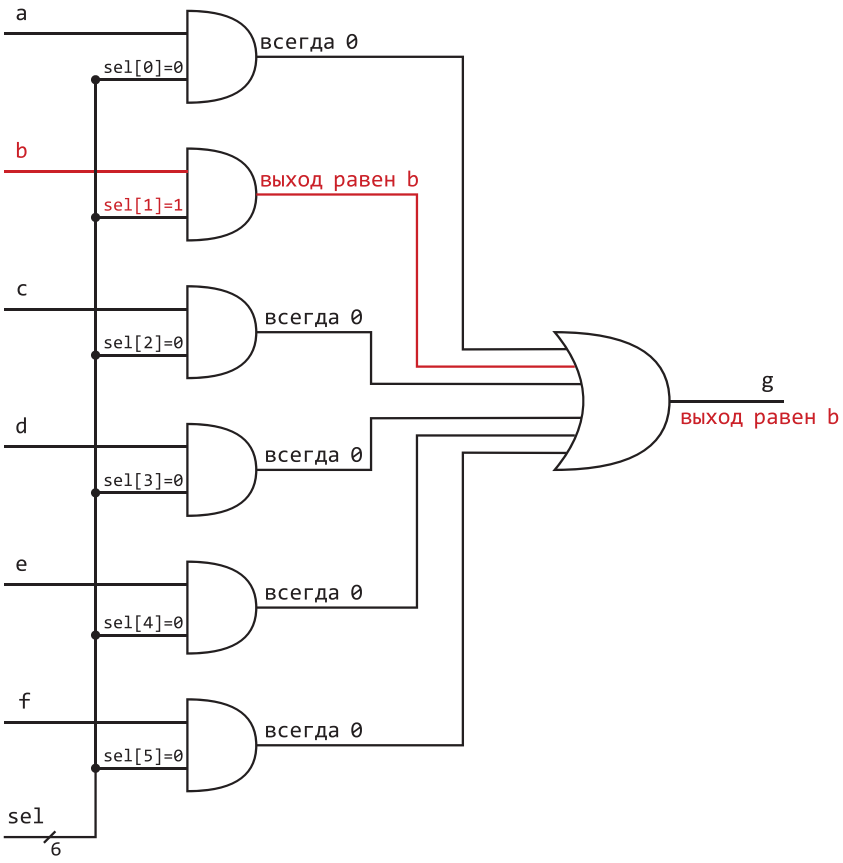


***Рисунок I.2-12**. Реализация мультиплексора, использующего one-hot-кодирование*

Программируемая память

Из транзисторов можно построить не только логические элементы, но и элементы памяти. На *рисунке I.2-13* представлена схема простейшей ячейки статической памяти, состоящей из транзистора и двух инверторов (т.е. сум-

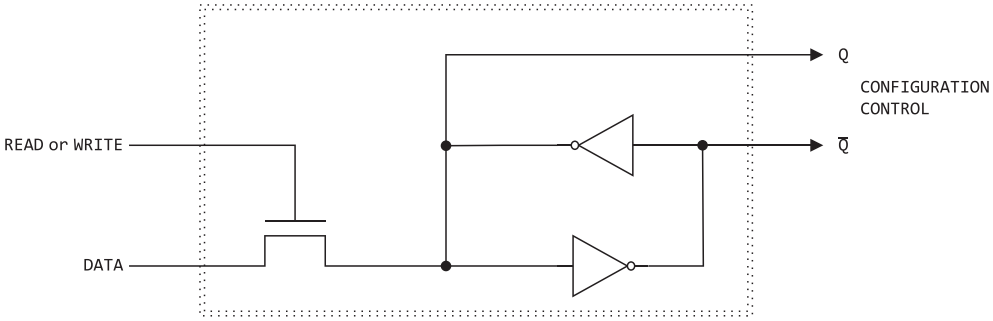


***Рисунок I.2-13.** Программируемая ячейка памяти ПЛИС Xilinx XC2064 [2, стр. 2-63]*

марно состоящей из пяти транзисторов, поэтому она называется **5T** SRAM). Данная ячейка реализует 1 бит программируемой памяти, являвшейся одним из основных компонентов самой первой ПЛИС.

Данная память представляет собой **бистабильную ячейку**, реализованную в виде петли из двух инверторов, в которых «заперто» хранимое значение (подробнее о бистабильных ячейках будет рассказано в следующей главе). Дважды инвертированный сигнал совпадает по значению с исходным, при этом, проходя через каждый из инверторов, сигнал обновляет своё значение напряжения, поддерживая тем самым уровни напряжения логических значений.

Для того чтобы поместить в бистабильную ячейку новое значение, к её входу подключается ещё один транзистор, замыкающий или размыкающий её с напряжением питания/земли.

Таблицы подстановки (Look-Up Tables, LUTs)

Представьте мультиплексор с четырьмя входными сигналами и двухбитным управляющим сигналом (обратите внимание, что теперь этот сигнал использует обычное двоичное кодирование). Но теперь, вместо того чтобы выставлять входные сигналы во «внешний мир», давайте подключим их к программируемой памяти. Это означает, что мы можем «запрограммировать» каждый из входов на какое-то константное значение. Поместим то, что у нас получилось, в отдельный блок, и вот, мы получили двухвходовую **Таблицу подстановки** (**Look-Up Tables**, далее — **LUT**).

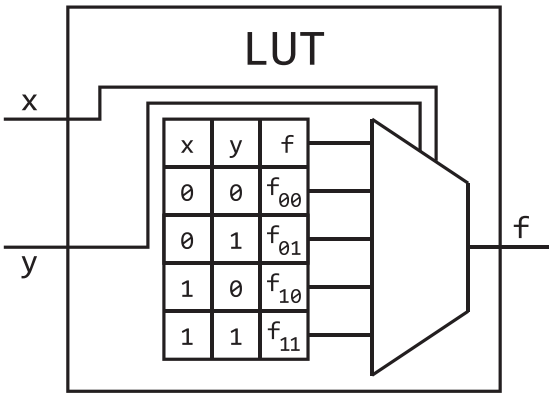


***Рисунок I.2-14**. Реализация таблицы подстановки (Look-Up Table, LUT)*

Эти два входа **LUT** являются битами управляющего сигнала мультиплексора, спрятанного внутри **LUT**. Программируя входы мультиплексора (точнее, программируя память, к которой подключены входы мультиплексора), мы можем реализовать на базе **LUT любую(!)** логическую функцию, принимающую два входа и возвращающую один выход. Для этого необходимо записать в LUT таблицу истинности реализуемой логической функции.

Допустим, мы хотим получить **логическое И**. Для этого нам потребуется записать в память содержимое в соответствии с *таблицей I.2-1.*

Адрес {x, y}	**Значение (f)**
00	0
01	0
10	0
11	1

Таблица I.2-1. *Реализация операции логического И с помощью LUT*

Это простейший пример: обычно **LUT** имеют больше входов, что позволяет им реализовывать более сложную логику.

D-триггеры

Используя неограниченное количество LUT, вы можете построить цифровую схему, реализующую логическую функцию любой сложности. Однако цифровые схемы не ограничиваются реализацией одних только логических функций (цифровые схемы, реализующие логическую функцию, называются **комбинационными**, поскольку выход зависит только от комбинации входов). Например, так не построить цифровую схему, реализующую процессор. Для таких схем нужны элементы памяти. Заметим, что речь идёт не о программируемой памяти, задавая значения которой, мы управляем тем, какие логические функции будут реализовывать LUT. Речь идёт о ячейках памяти, которые будут использоваться логикой самой схемы.

Такой базовой ячейкой памяти является **D-триггер** (**D flip-flop**). Из D-триггеров можно собирать другие ячейки памяти, например **регистры** (а из регистров можно собрать **память с произвольным доступом** (**random access memory**, **RAM**)), **сдвиговые регистры** и т.п.

D-триггер — это цифровой элемент, способный хранить один бит информации (*рисунок I.2-15*). В базовом варианте у этого элемента есть два входа и один выход. Один из входов подаёт значение, которое будет записано в

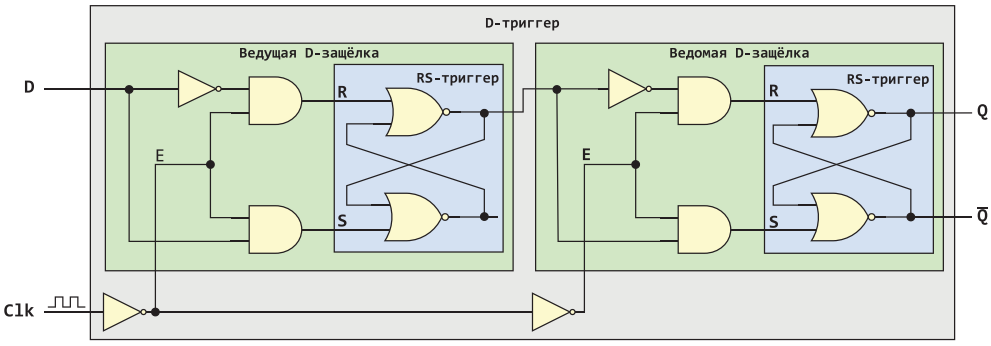


Рисунок I.2-15*. Реализация D-триггера*

D-триггер, второй вход управляет записью (обычно он называется *clk* или *clock* и подключается к тактирующему синхроимпульсу схемы). Когда управляющий сигнал меняет своё значение с *0* на *1* (либо с *1* на *0*, зависит от схемы), в **D-триггер** записывается значение сигнала данных. Обычно, описывая **D-триггер**, говорится, что он строится из двух **триггеров-защёлок** (**D-latch**), которые в свою очередь строятся из **RS-триггеров**. Однако в конечном итоге все эти элементы могут быть построены на базе логических вентилей **И/ИЛИ**, **НЕ**.

Арифметика

Помимо описанных выше блоков (мультиплексоров и построенных на их основе LUT и регистров) выделяется ещё один тип блоков, настолько часто используемый в цифровых схемах, что его заранее размещают в ПЛИС в больших количествах, — это арифметические блоки. Эти блоки используются при сложении, вычитании, сравнении чисел, реализации счётчиков. В разных ПЛИС могут быть предустановлены разные блоки: где-то это может быть 1-битный сумматор, а где-то — блок вычисления ускоренного переноса (*carry-chain*).

Все эти блоки могут быть реализованы через логические вентили, например так можно реализовать сумматор:

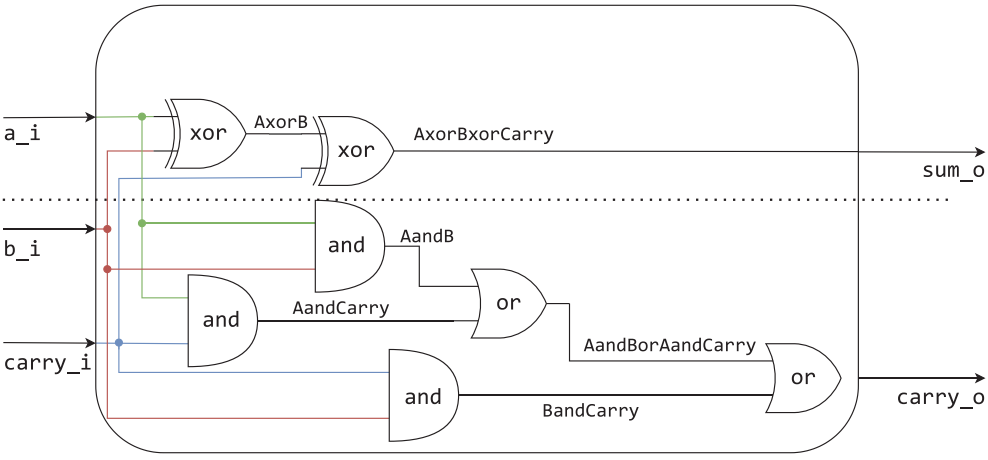


***Рисунок I.2-16**. Реализация полного однобитного сумматора*

Логические блоки

В предыдущих параграфах были рассмотрены отдельные виды цифровых блоков: таблицы подстановок, регистры, арифметические блоки. Для удобства структурирования эти блоки объединены в ПЛИС в виде **логических блоков**. Обычно логические блоки современных ПЛИС состоят из **логических ячеек** (или **логических элементов**), но для простоты повествования мы объединим все эти термины.

Логический блок может содержать одну или несколько **LUT**, **арифметический блок** и один или несколько **D-триггеров**, которые соединены между собой некоторым количеством мультиплексоров. На *рисунке I.2-17* представлена схема того, как может выглядеть **логический блок**.

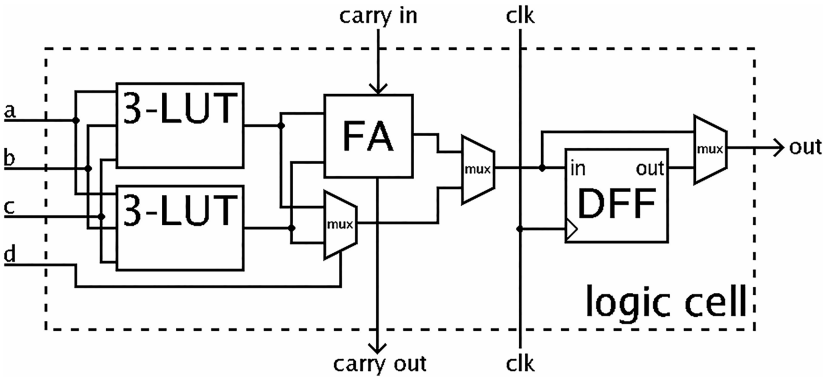


***Рисунок I.2-17**. Схема логического блока [3]*

Логический блок на *рисунке I.2-17* представляет собой цепочку операций: *логическая функция, реализованная через LUT, → арифметическая операция → запись в D-триггер*. Каждый из мультиплексоров определяет то, будет ли пропущен какой-либо из этих этапов. Таким образом, конфигурируя логический блок, можно получить следующие вариации кусочка цифровой схемы.

1. Комбинационная схема (логическая функция, реализованная в LUT).
2. Арифметическая операция.
3. Запись данных в D-триггер.
4. Комбинационная схема с записью результата в D-триггер.
5. Арифметическая операция с записью результата в D-триггер.
6. Комбинационная схема с последующей арифметической операцией.
7. Комбинационная схема с последующей арифметической операцией и записью в D-триггер.

На *рисунке I.2-18* приведён реальный пример использования логического блока в ПЛИС *xc7a100tcsg324-1* при реализации арифметико-логического устройства (АЛУ), подключённого к периферии отладочной платы *Nexys-7*.

На нём вы можете увидеть использование LUT, арифметического блока (ускоренного расчёта переноса) и одного из D-триггеров. D-триггеры, обозначенные серым цветом, не используются.

Располагая большим наборов таких логических блоков и имея возможность межсоединять их нужным вам образом, вы получаете широчайшие возможности по реализации практически любой цифровой схемы (ограничением является только ёмкость ПЛИС, т.е. количество подобных логических блоков, входов-выходов и т.п.).

Помимо логических блоков в ПЛИС есть и другие примитивы: **блочная память**, **блоки умножителей** и т.п.

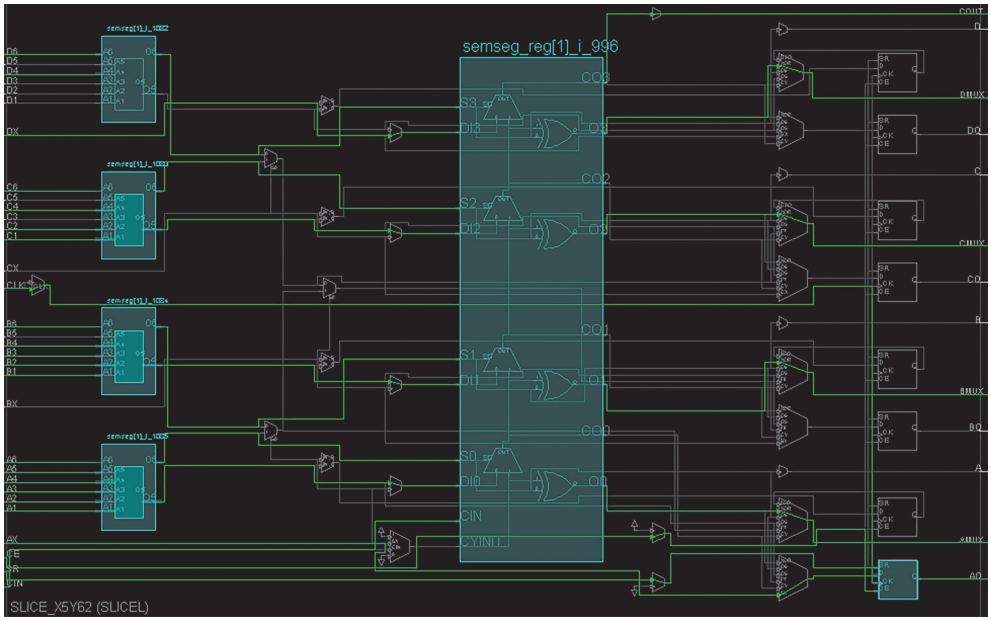


Рисунок I.2-18. *Пример использования логической ячейки*

Сеть межсоединений

Для того чтобы разобраться, как управлять межсоединением логических блоков, рассмотрим *рисунок I.2-19*, входящий в патент на ПЛИС [4].

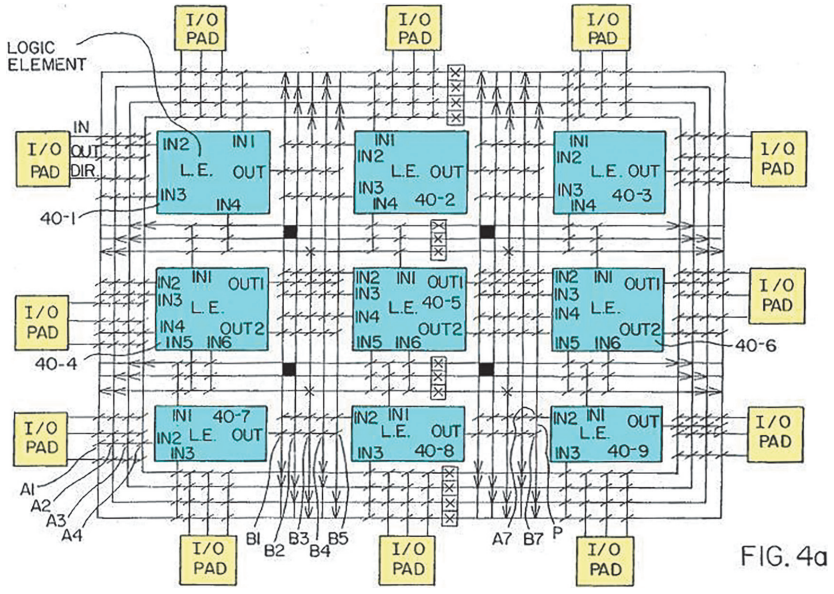


Рисунок I.2-19. *Содержимое ПЛИС в виде межсоединения логических блоков и блоков ввода-вывода [5]*

Синим показано 9 логических блоков, жёлтым — 12 блоков ввода-вывода. Все эти блоки окружены **сетью межсоединений** (interconnect net), представляющей собой решётку из горизонтальных и вертикальных соединительных линий — межсоединений общего назначения (general purpose interconnect) [2, 2—66].

Косыми чертами в местах пересечения линий обозначены **программируемые точки межсоединений** (**programmable interconnect points**, **PIPs**), представляющие собой транзисторы, затвор которых подключён к программируемой памяти.

Управляя значением в подключённой к затвору транзистора памяти, можно управлять тем, что из себя будет представлять транзистор в данной точке — разрыв или цепь. А значит, можно удалять «лишние» участки сети, оставляя только используемые логические блоки, соединённые между собой.

Итоги главы

1. Используя такие элементы, как **транзисторы**, можно собирать **логические вентили**: элементы **И**, **ИЛИ**, **НЕ** и т.п.
2. Используя **логические вентили**, можно создавать схемы, реализующие как **логические функции** (**комбинационные схемы**), так и сложную логику с памятью (**последовательностные схемы**).
3. Из логических вентилей строится и такая важная комбинационная схема, как **мультиплексор** — цифровой блок, который в зависимости от значения управляющего сигнала подаёт на выход один из входных сигналов.
4. Кроме того, подключив вход бистабильной ячейки (представляющей собой петлю из двух инверторов) к транзистору, можно получить 1 бит **программируемой памяти**.
5. Подключив входные сигналы мультиплексора к программируемой памяти, можно получить **таблицу подстановок** (**Look-Up Table**, **LUT**), которая может реализовывать простейшие логические функции. LUT позволяют заменить логические вентили И/ИЛИ/НЕ и удобны тем, что их можно динамически изменять. Логические вентили в свою очередь исполняются на заводе и уже не могут быть изменены после создания.
6. Из логических вентилей также можно собрать базовую ячейку памяти — **D-триггер**, и такую комбинационную схему, как **полный 1-битный сумматор** (или любой другой часто используемый арифметический блок).
7. Объединив LUT, арифметический блок и D-триггер, получаем структуру в ПЛИС, которая называется «**логический блок**».
8. Логический блок (а также другие **примитивы**, такие как **блочная память** или **умножители**) — это множество блоков, которые заранее физически размещаются в кристалле ПЛИС, их количество строго определено конкретной ПЛИС и не может быть изменено.
9. Подключая программируемую память к транзисторам, расположенным в узлах **сети межсоединений**, можно управлять расположением разрывов в сети, а значит, можно оставить только маршрут, по которому сигнал пойдёт туда, куда нам нужно (**трассировать сигнал**).

10. **Конфигурируя примитивы** и **трассируя сигнал** между ними (см. п. 4), можно получить **практически любую цифровую схему** (с учётом ограничения ёмкости ПЛИС).

Список источников

1. Alchitry. Ell C / “How Does an FPGA Work?”. URL: https://learn.sparkfun.com/tutorials/how-does-an-fpga-work/all
2. Xilinx. The Programmable Gate Array Data Book. URL: https://archive.org/details/programmablegate00xili
3. Wikipedia. Field-programmable gate array. URL: https://en.wikipedia.org/wiki/Field-programmable_gate_array
4. Ross H. Freeman / Configurable electrical circuit having configurable logic elements and configurable interconnects / United States Patent: https://patents.google.com/patent/US4870302A
5. Ken Shirriff. Reverse-engineering the first FPGA chip, the XC2064. URL: http://www.righto.com/2020/09/reverse-engineering-first-fpga-chip.html

Последовательностная логика

Классификация цифровой логики

Цифровая логика делится на **комбинационную** и **последовательностную**.

Комбинационная логика (или «логика без памяти») — это цифровая логика, выходы которой зависят только от её входов. Один и тот же набор входных воздействий на эту логику всегда будет давать один и тот же результат. Комбинационную логику можно всегда представить в виде таблицы истинности (или логической функции) всех её выходов от её входов.

В противоположность комбинационной, существует также и **последовательностная логика** (sequential logic), или «логика с памятью» — цифровая логика, выходы которой зависят не только от её входов, но и от её внутреннего состояния.

Простейшим примером комбинационной логики может быть любой логический вентиль, например исключающее ИЛИ (*рисунок I.3-1 (а)*). Эта комбинационная схема всегда будет давать *0*, если оба её входа равны, в противном случае она выдаст *1*.

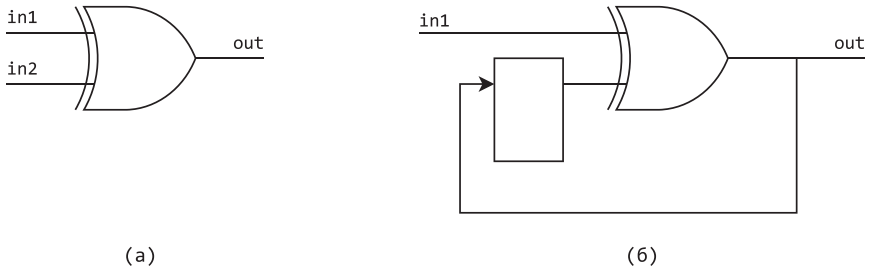


Рисунок I.3-1. *Пример комбинационной (а) и последовательностной (б) схем*

Предположим теперь, что в качестве одного из входов исключающего ИЛИ стоит абстрактная ячейка памяти, которая запоминает предыдущее значение, выданное этим логическим вентилем (*рисунок I.3-1 (б)*). Теперь выходы схемы зависят не только от того, что мы подадим на вход, но и от того, что находится в данной ячейке памяти, а самое главное — теперь, подавая на вход одно и то же воздействие, мы можем получить разные результаты.

Будем исходить из того, что изначально ячейка памяти проинициализирована нулём. Сперва подадим на вход этой схемы *0*. Поскольку оба входа равны *0*, на выход схемы подаётся *0* и значение в ячейке памяти остаётся прежним. Затем подадим на вход *1* — теперь на выход схемы идёт значение *1* и оно же сохраняется в ячейке памяти. После мы снова подаём на вход 0, однако, в отличие от первого раза, на выход схемы пойдёт *1*, т.к. входы исключающего ИЛИ не равны. Выставив на вход *1* ещё раз, мы получим на выходе 0.

Как вы видите, результат последовательностной логики зависит от **последовательности** произведённых входных воздействий, в то время как комбинационная логика зависит от **комбинации** её текущих входных воздействий.

Последовательностная логика делится на **синхронную** и **асинхронную**.

Синхронной логикой называется такая логика, которая обновляет своё состояние (содержимое ячеек памяти) одновременно (**синхронно**) с фронтом тактового сигнала[1]. В свою очередь **асинхронная последовательностная логика** — это логика, которая может обновлять своё состояние **асинхронно** (т.е. без привязки к фронту тактового синхроимпульса). Бывает также и синхронная логика с асинхронными сигналами предустановки/сброса.

Комбинационная логика по своей природе является асинхронной, поэтому в зависимости от контекста под асинхронной логикой может подразумеваться как комбинационная логика, так и последовательностная логика, которая может обновлять значение не по фронту тактового синхроимпульса.

Бистабильные ячейки

Бистабильная ячейка — это элемент статической памяти, способный принимать одно из двух устойчивых состояний, соответствующих цифровым значениям «0» или «1». **Статическая память** — это тип памяти, который сохраняет данные в течении неопределённого времени, пока его питание остаётся включённым, без необходимости регенерации (в отличие от **динамической памяти**, использующей для хранения конденсаторы, требующие для хранения регулярного обновления данных).

Рассмотрим простейшую ячейку статической памяти, представленную на *рисунке I.3-2*, которая способна хранить 1 бит информации.

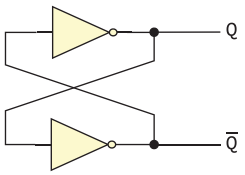


Рисунок I.3-2. *Простейшая ячейка статической памяти*

Данная ячейка представляет собой петлю из двух инверторов, в которых «заперто» хранимое значение. Дважды инвертированный сигнал совпадает по значению с исходным, при этом, проходя через каждый из инверторов, сигнал обновляет своё значение напряжения, поддерживая тем самым уровни напряжения логических значений. Главной проблемой подобной ячейки является то, что она требует дополнительной логики для записи информации.

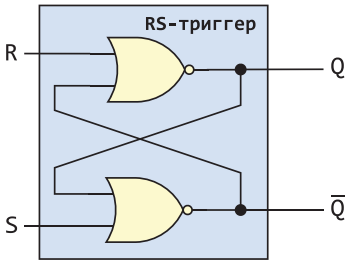
Для того чтобы добавить в эту ячейку возможность записи данных, можно поставить перед инверторами логические элементы ИЛИ (образующие вместе с инверторами элементы ИЛИ-НЕ).

В результате получится **RS-триггер** — бистабильная ячейка, представленная на *рисунке I.3-3*.

RS-триггер

RS-триггер — это бистабильная ячейка, имеющая два управляющих входа — R (reset) и S (set), и два выхода — Q и $\overline{Q}$. $\overline{Q}$ является инверсией Q.

[1] В некоторых источниках синхронной логикой могут называть и ту, что работает по уровню (а не фронту) единого источника тактового синхроимпульса [1, стр. 164].



Входы			Выходы	
S	R	Q_n	Q_{n+1}	$\overline{Q}_{n+1}$
0	0	0	0	1
0	0	1	1	0
0	1	x	0	1
1	0	x	1	0
1	1	x	0	0

Рисунок I.3-3. *Схема и таблица истинности RS-триггера. X означает, что в этой строке результат не зависит от хранимого значения*

RS-триггер, построенный на логических элементах ИЛИ-НЕ, работает следующим образом.

1. Если вход $R = 1$, а $S = 0$, то выход верхнего элемента ИЛИ-НЕ (а значит, и выход Q) равен *0* вне зависимости от второго его входа. Этот выход поступает вместе с входом S на нижний элемент ИЛИ-НЕ, который выдаёт 1 (на выход $\overline{Q}$), поскольку оба его входа равны *0*. Эта единица подаётся на второй вход верхнего элемента ИЛИ-НЕ, и теперь, даже если вход R станет равным *0*, *1* на втором его входе сможет воспроизвести то же самое поведение, запирая внутри триггера стабильное состояние $Q = 0$.
2. Если вход $R = 0$, а $S = 1$, схема работает противоположным образом: поскольку на нижний элемент подаётся *1* с входа S, то выход $\overline{Q}$ равен 0 вне зависимости от второго входа нижнего элемента ИЛИ-НЕ. Этот 0 подаётся на второй вход верхнего элемента ИЛИ-НЕ, и, поскольку оба его входа равны *0*, на выходе этого элемента (на выход Q) подаётся *1*, которая возвращается обратно на вход нижнего элемента ИЛИ-НЕ, запирая внутри триггера стабильное состояние $Q = 1$.
3. Таким образом, если оба входа одновременно равны *0*, RS-триггер хранит своё предыдущее значение.

Проблемой данного триггера является то, что он имеет **запрещённую** комбинацию входов. В случае RS-триггера, построенного на элементах ИЛИ-НЕ, данной комбинацией входов является $R = 1$ и $S = 1$. Даже с точки зрения функционального назначения данная комбинация не имеет смысла: кому потребуется одновременно и сбрасывать RS-триггер в 0, и устанавливать его в 1? Тем не менее вот что произойдёт, если использовать эту комбинацию.

4. Если оба входа одновременно равны *1*, то оба выхода Q и $\overline{Q}$ будут равны *0*, что нарушает логику работы триггера, поскольку выход $\overline{Q}$ должен быть инверсией выхода Q. При этом, если после этого перевести оба входа в *0*, RS-триггер окажется в неустойчивом состоянии (в состоянии гонки), а выходы могут начать неопределённо долго инвертироваться. Пока RS-триггер был в запрещённом состоянии, *выходы* Q *и* $\overline{Q}$, равные *0*, подавались на входы обоих элементов ИЛИ-НЕ, а если после этого **одновременно** перевести *входы* R *и* S в *состояние* 0, то на входах обоих вентилей будут *0*, что побудит их выдать на *выходы* 1, которые пойдут обратно на входы этих вентилей, после чего те подадут на *выход* 0, и так будет продолжаться до тех пор, пока