Семен Игонин

FRE*e*ASM

Научно-популярная книга о устройстве и функционировании микропроцессоров. Предназначена для читателей, которые хотят понять принципы, лежащие в основе работы современных компьютеров. Книга начинается с изучения поведения электрона в электрическом поле, а заканчивается низкоуровневым языком ассемблера. Каких-либо знаний из области физики, математики или информатики от читателя не требуется. Только внимательность и усердие.

Также затрагивается переход от ассемблера к языкам программирования высокого уровня. Бонусом, в процессе создания микропроцессора, читатель кратко ознакомится с основными темами информатики: системы счисления, кодирование чисел, алгебра логики, архитектура компьютера, алгоритмические конструкции.

Книга, представленная электронной версией, распространяется бесплатно.

Рецензенты и редакторы: *вакантная позиция, буду премного благодарен за любую помощь в указании на грамматические, стилистические и логические ошибки. Исправления выделять желтым цветом и присылать на is-info64@ya.ru, отзывы и пожелания туда же.*

Ссылка на .odt версию для редактирования: https://disk.yandex.ru/d/ehZb6LB6eewe1w

Энгельс, 2020

Краткое оглавление

Пролог....5

О чем?....5

Что дальше?....6

Посвящается....7

Раздел 1. Физика....8

Глава 1.1 Основные физические понятия....8

Глава 1.2 Основы физики полупроводников....15

Глава 1.3 Транзистор....29

Глава 1.4 Устройство базисных логических элементов....39

Раздел 2 Логика....46

Глава 2.1 Основные логические элементы....46

Глава 2.2 Исключающее ИЛИ....54

Глава 2.3 Сумматор....57

Глава 2.4 Декодер....68

Глава 2.5 Секвенциальная логика и память....73

Раздел 3. Арифметико-логическое устройство....81

Глава 3.1 Середина....83

Глава 3.2 Конец....90

Глава 3.3 Шина....93

Глава 3.4 Начало....95

Глава 3.5 Пример работы с АЛУ....98

Раздел 4. Микропроцессорная авантюра....103

Глава 4.1 Движущие элементы микропроцессора....104

Глава 4.2 Элементы УИИ.....108
Глава 4.3 Исполнение программы в УИИ.....115
Глава 4.4 Основные элементы устройства управления.....120
Раздел 5. ASSEMBLER МПА.....138
Важное дополнение.....139
Глава 5.1 ASSEMBLER с первого взгляда.....140
Глава 5.2 Набор команд.....143
Глава 5.3 Пример программы.....155
Глава 5.4 Прошивка.....156
Раздел 6. Алгоритмические конструкции.....161
Глава 6.1 Умный дом (следование).....161
Глава 6.2 Холодильник (ветвление).....171
Глава 6.3 Утюг (циклы).....176
Глава 6.4 Высокоуровневое программирование.....185
Раздел 7. Пошаговое руководство.....192
Шаг 1 Закупка комплектующих.....193
Шаг 2 Знакомство.....196
Шаг 3 Сборка экспериментальной установки.....200
Шаг 4 Настройка программной и аппаратной среды.....205
Шаг 5 Трансляция и прошивка программы.....209
Шаг 6 Бегущие огни.....215
Эпилог.....216

Пролог

С развитием технологий и рыночной экономики человечество увязло в мультимедиа индустрии, которая физически воплощена в кремнии. Кремний не просто кусок земной коры. В устройствах из кремния отражается все развитие человеческой цивилизации. От логики Аристотеля, до современных технологий литографии. От элементарной арифметики, до фото котят в ленте новостей. Все лучшее, созданное людьми, воплощено в устройствах электронной промышленности. Это касается не только научных достижений, но и искусства.

Большинство людей не имеют представления о том, как устроен мир, в котором они живут. Имеется ввиду виртуальный мир. Возникает завеса тайны. Незнание приводит к искаженному восприятию технологий, к преклонению перед прогрессом. Поклонение материальным вещам вызывает зависимость и порабощение.

Освободимся от оков виртуального рабства. Рассмотрим виртуальность с позиции реальности.

О чем?

Этот небольшой пролог должен был заставить задуматься, наверное.

На самом деле всё куда проще. В книге разбирается, как работает компьютер. С самого начала. От знания, что + и — притягиваются до функционирования микропроцессора.

Сперва займемся описанием физики явлений, происходящих в микропроцессоре. Далее, соберем простейший логический элемент — базис (основу). Из базисных элементов сконструируем логические схемы, позволяющие совершать арифметические и логические операции, рассмотрим устройство схем, лежащих в основе памяти. После этого, нарисуем собственный микропроцессор, придумаем язык ассемблера и напишем на нем несколько программ.

Целью книги не является изучение какого-нибудь существующего устройства или общее рассмотрение архитектуры микропроцессорной техники. «С нуля» разрабатывается собственный микропроцессор и на его примере возникает представление об основных принципах работы всякого современного компьютера. Постараемся сделать непротиворечиво. Если устройство не противоречит логике, значит логично, что оно будет работать (логично, но не физично :)).

Входным порогом к пониманию работы компьютера является умение складывать числа в пределах десяти. Требуется только внимательность и усердие.

Что дальше?

Для тех, кого книга вдохновит на создание чего-то реального, в конце приведен дополнительный раздел.

При написании первой программы на ассемблере может возникнуть большой психологический барьер, потому что появляется множество вопросов. Какие комплектующие купить, какую программу установить, что где писать, как прошить, как запустить? Поэтому было решено привести подробное пошаговое руководство по созданию гирлянды.

В качестве технической составляющей выбран микроконтроллер серии *AVR* фирмы *Atmel*. Существует множество хороших самоучителей по программированию на данных устройствах. В данной книге рассматриваются только шаги, которые необходимо выполнить, чтобы гирлянда заработала. Дальнейшее знакомство с архитектурами конкретных систем возлагается на читателя.

Посвящается

Несостоявшемуся профильному 9Б классу.

Препятствия отсутствуют, все получится, верьте в себя, а я верю в вас!

Раздел 1. Физика

В данном разделе разбирается устройство базисного логического элемента. Базис – это основа. Базисный логический элемент – логическая основа микропроцессора.

Глава 1.1 Основные физические понятия

Даны общие представления о материальном мире, необходимые для понимания функционирования полупроводниковых структур. Из полупроводниковых структур состоит транзистор. Транзистор – материальная основа базисного логического элемента.

1.1.1 Поле

Пусть справедлив *закон обыденного опыта*: “противоположности притягиваются”.

Доказательство закона: им так интереснее. ч.т.д.

Разделим грубо материю на три вида: положительную (обозначим ее знаком +), отрицательную (знаком -) и нейтральную. Нас будут интересовать частички + и - материи. По-научному они называются **заряд**.

Нарисуем две частички рис. 1.1:

Рисунок 1.1 – Положительная и отрицательная материя рядом.

Из закона обыденного опыта они должны притянуться друг к другу. Попробуем это изобразить на рис. 1.2. Притяжение обозначим стрелкой. Стрелочка указывает в каком направлении заряд притягивается (отталкивается, для зарядов с одинаковым знаком). Количественно притяжение/отталкивание характеризуется **силой Кулона**, которая выражается известной формулой

$$\vec{F} = \frac{q_1 q_2}{r^2},$$

где $q_{1,}q_2$ - величины зарядов, r - расстояние между зарядами, $\vec{F}$ - сила электрического взаимодействия между зарядами - сила Кулона. **Сила** - векторная величина, у нее есть направление. Сила Кулона указывает насколько сильно частицы притягиваются или отталкиваются из-за наличия у них заряда, а стрелка указывает направление, в котором происходит это притяжение или отталкивание. Можно сказать, что сила тащит заряд в указанном стрелкой направлении.

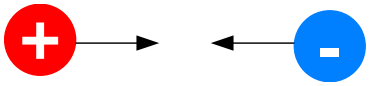


Рисунок 1.2– Положительная и отрицательная материя притягиваются.

Теперь представим несколько зарядов. Пусть некоторые из них закреплены на месте, не могут двигаться рис. 3. Вопрос: куда и с какой силой будет двигаться положительный заряд q_0?

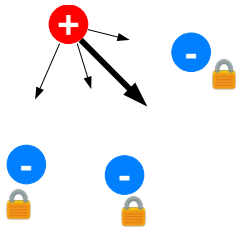


Рисунок 1.3 – Суперпозиция сил для нескольких зарядов.

Правило простое. Необходимо нарисовать силу притяжения заряда q_0 к каждому заряду по-отдельности (как будто остальных зарядов нет). Затем сложить силы по правилам сложения векторов. В итоге получится некоторая средняя сила

(необходимо помнить, что складываются стрелки, а не просто числа, в формуле это не показано)

$$\vec{F}=\frac{q_0 q_1}{r_1^2}+\frac{q_0 q_2}{r_2^2}+\frac{q_0 q_3}{r_3^2}\ .$$

Результирующая сила, действующая на положительный заряд, показана на рисунке жирной стрелкой. Такое сложение называется **суперпозицией.**

Если заменить q_0 на другой заряд, то силу придется пересчитать. А если зарядов несколько сотен? Тогда каждый раз, когда при изменении заряда q_0, придется пересчитывать несколько сотен слагаемых. Люди хитрые, нашли элегантное и простое решение, они вынесли q_0 за скобки

$$\vec{F}=q_0\left(\frac{q_1}{r_1^2}+\frac{q_2}{r_2^2}+\frac{q_3}{r_3^2}\right).$$

Уберем с рисунка положительный заряд. Что останется на его месте? Из формулы, очевидно, что некоторое число E_0 равное $\frac{q_1}{r_1^2}+\frac{q_2}{r_2^2}+\frac{q_3}{r_3^2}$ и стрелка (рис. 1.4).

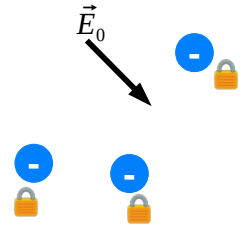


Рисунок 1.4 – Электрическое поле в некоторой точке.

Число и стрелку $\vec{E}_0$ назвали **электрическим полем** в данной точке пространства

$$\vec{E}_0=\frac{q_1}{r_1^2}+\frac{q_2}{r_2^2}+\frac{q_3}{r_3^2}\ .$$

Каждый заряд по отдельности создает в точке свое электрическое поле $\vec{E}=\frac{q}{r^2}$. Поля отдельных зарядов складываются. Опять-таки принцип суперпозиции.

$$\vec{E}=\vec{E}_1+\vec{E}_2+\vec{E}_3=\frac{q_1}{r_1^2}+\frac{q_2}{r_2^2}+\frac{q_3}{r_3^2}\ .$$

Таким образом, пространство вокруг зарядов представляет собой набор векторов (чисел со стрелками). В каждой точке свое поле: свой вектор со своим числовым значением (длиной) и направлением. Теперь сила находится просто. Величина заряда умножается на поле.

Если заряд поместить в некоторую точку, то сила, действующая на заряд, будет равна величине заряда, умноженной на электрическое поле в данной точке $\vec{F}_0=q_0\vec{E}_0$. Вся прелесть в том, что поле можно вычислить один раз. Для некоторой конфигурации зарядов поле не изменяется, пока не изменяется положение зарядов. Тогда легко найти силу, действующую на любой заряд.

Рассмотрим конфигурацию зарядов рис. 1.5, в которой заряды слева и справа закреплены. Данный случай необходимо разобрать, чтобы понять работу транзистора.

Рисунок 1.5 – Поле двух разноименных зарядов.

Вспомним, что + и + отталкиваются, а + и – притягиваются. Тогда сила, действующая на положительный заряд, размещенный посередине между + и -, направлена вправо, а значит и электрическое поле направлено вправо.

А если вместо положительного заряда в центре разместить отрицательный заряд? Тогда поле должно быть направлено влево? Люди договорились при вычислении электрических полей использовать положительный заряд, да еще равный по величине единице. Важные выводы, которые пригодятся:

1. **Поле направлено от + к -**
2. **Положительный заряд перемещается по полю**
3. **Отрицательный заряд перемещается против поля.**

К слову, о поле. Оно бывает не только электрическое. Есть еще гравитационное (аналогия с электрическим, только вместо зарядов – масса). Или, например, температурное поле. Температура - просто числа в каждой точке пространства, без стрелок. Поля такого рода, называют **скалярными**. Электрическое же поле – **векторное**. У него есть направление — направление действия силы, оно же направление перемещения положительного заряда.

Можно взять любую величину и придумать для нее поле. Например, поле освещенности комнаты, поле запахов и т.д. Подумайте об этом. Должно быть интересно.

1.1.2 Принцип неопределенности Гейзенберга.

Общеизвестный факт, что материальный мир состоит из атомов, а те, в свою очередь, из положительного ядра и отрицательных электронов. Ядро очень маленькое, электрон еще меньше. Почему же размер атома в тысячи раз больше ядра, ведь по **закону**, + и – должны притянуться. Почему электрон не падает на ядро? Почему материя в нормальных условиях не стягивается в точку?

Красивый ответ можно вывести из основного принципа квантовой механики - **принципа неопределенности Гейзенберга**

$$\Delta p \Delta x > h \ ,$$

здесь Δp - погрешность (неточность) в определении импульса, Δx - погрешность в определении координаты (местоположения в пространстве), h – очень маленькая величина. Для простоты рассуждений, будем смотреть на импульс, как на скорость

$$\Delta \upsilon \Delta x > h \ .$$

Измерим одновременно положение и скорость автомобиля. Мы умеем достаточно точно определять и скорость, и координату. Однако все-равно неточность измерений будет огромна, по-сравнению с очень маленьким значением **постоянной Планка *h***, равным примерно 10^{-34} (0.00000000000000000000000000000001). Согласитесь не очень много.

Всё-таки представим, что скорость измерена абсолютно точно, т.е. $\Delta\upsilon = 0$. По принципу неопределенности Гейзенберга должно выполняться соотношение $0 \cdot \Delta x > h$. На что необходимо умножить ноль, чтобы получить, хоть какое-нибудь маленькое число *h?* В математическом смысле это невозможно. Но в физике абсолютных нулей не бывает, поэтому можно смело сказать, что $\Delta x = \infty$, и считать неравенство $0 \cdot \infty > h$ справедливо выполняющимся. Подождите, а что значит $\Delta x = \infty$? Мы потеряли автомобиль в пространстве.

Зная точную скорость объекта, мы не можем сказать, где он находится. Зная точное расположение объекта, мы ничего не знаем о его скорости. Это предельные случаи, которые в физике не реализуются. Там нет нуля и нет бесконечности. Их лучше заменить на очень большие и очень малые величины.

В обыденной жизни принцип неопределенности Гейзенберга выполняется незаметно, поскольку скорости и расстояния очень велики. Но в атомах ситуация другая, величина произведения $\Delta\upsilon\Delta x$ уже сравнима с постоянной Планка *h*.

Посмотрим на рис. 1.6a. Здесь предполагается, что скорость известна абсолютно точно, электрон летит к ядру: – притягивается к +. Но тогда ничего не известно о координате. Ничто не мешает электрону лететь из бесконечности. Следовательно, он никогда не долетит и не упадет на ядро.

На рис. 1.6b ситуация противоположная. Предполагается, что электрон все-таки упал на ядро. Но тогда неизвестна скорость. Скорее всего она равна бесконечности и электрон бесконечно быстро от ядра отскочит.

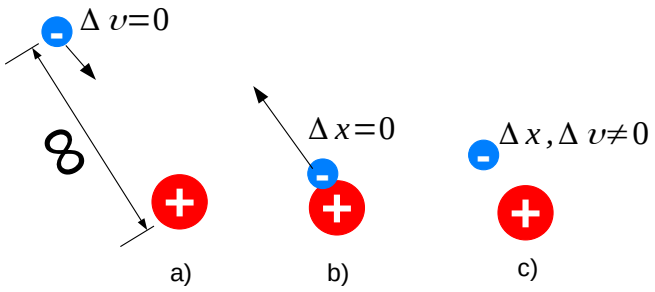


Рисунок 1.6 – Принцип неопределенности Гейзенберга: a) $\Delta\upsilon = 0$; b) $\Delta x = 0$; c) все хорошо, $\Delta\upsilon \neq 0, \Delta x \neq 0$.

Таким образом, на неопределенности $\Delta\upsilon \neq 0, \Delta x \neq 0$ держится мир. Если ноль, то все рушится, если чуть неопределенно, то все хорошо.

Именно поэтому мы и вся мебель не проваливаемся сквозь пол, и далее вместе с полом и всей материей не стягиваемся в небольшую сферу, состоящую из сгустка электронов и ядер, минусов и плюсов.

Глава 1.2 Основы физики полупроводников

Рассматривается классификация материалов в зависимости от величины их проводимости. Приводятся основные сведения о полупроводниках. Производится деление полупроводников на три типа: собственный, донорный и акцепторный. Подробно разбирается устройство и функционирование *p-n* перехода, лежащего в основе работы транзисторов, диодов, лазеров и множества других элементов современной электроники.

1.2.1 Классификация материалов

Электрический ток – это направленное движение заряженных частиц, в частности, электронов. Электроны могут перемещаться в воздухе, в вакууме, но в жизни чаще всего люди обращают внимание не движение электронов в твердых телах, например, в проводах.

В зависимости от того насколько хорошо среда проводит электрический ток материалы делятся на металлы, диэлектрики и полупроводники.

Металлы проводят ток хорошо, т.е. имеют малое сопротивление потоку электронов. Диэлектрики практически не проводят ток, имеют большое сопротивление. **Полупроводники** – плохие металлы и плохие диэлектрики. Однако свойствами полупроводников очень легко управлять.

Добавляя в полупроводник посторонние атомы, или, оказывая внешнее воздействие, например, изменения освещенность или температуру, можно в широких пределах изменять свойства полупроводников. Более того из полупроводников создается специальная структура – **p-n переход**. Этой структурой также можно управлять.

Полупроводники позволяют управлять собой. Это позволяет использовать их для управления вычислениями. Другими словами, микропроцессор состоит из управляемых *p-n* переходов.

1.2.2 Собственный полупроводник

Основным материалом при производстве современных электронных компонентов является кремний. Поэтому рассмотрим внутреннее строение полупроводниковых материалов на примере кремния.

Кремний в таблице периодических элементов Менделеева стоит на 14 позиции. Значит у кремния ядро имеет заряд +14. Вокруг ядра летают 14 электронов (и не падают) рис. 1.7a. На внешнем уровне атома кремния оказывается четыре электрона.

Представим, что мы сидим на внешнем уровне атома кремния и смотрим в область ядра. Мы увидим совокупный положительный заряд, равный +4, а не +14. Электроны, летающие на внутренних уровнях, как бы уменьшают заряд ядра. По-другому, можно сказать, электроны на внутренних уровнях **экранируют** положительный заряд ядра. Тогда схему атома можно упростить рис. 1.7b. Внутренние электроны не участвуют в процессах, которые будут рассматриваться далее. Поэтому будем пользоваться упрощенной схемой атома кремния рис. 1.7b.

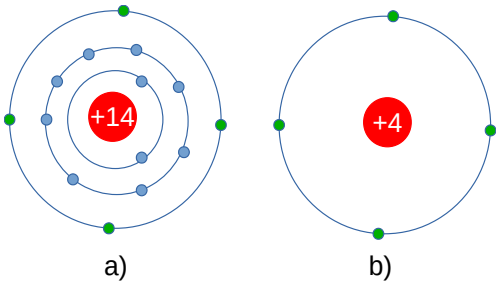


Рисунок 1.7– Строение атома кремния: a – полная схема; b – упрощенная схема.

Как известно из химии, электроны на внешнем уровне называются **валентными**. У кремния четыре валентных электрона. При этом на внешнем уровне есть свободные места ещё для четырех электронов. Поместим рядом с атомом кремния несколько атомов кремния рис. 1.8.

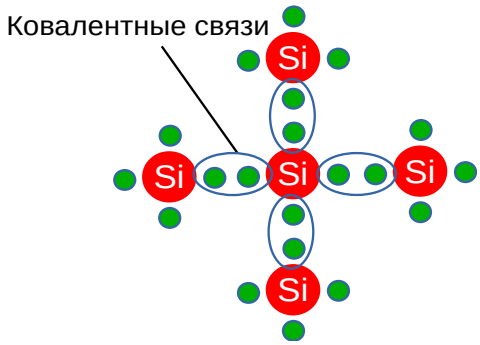


Рисунок 1.8 – Ковалентная связь

Теперь электроны одного атома кремния, могут перейти в другой атом кремния и наоборот. Говорят, что происходит перекрытие электронных орбиталей. Электрон не знает, какому атому кремния он принадлежит. Два электрона принадлежат двум атомам кремния одновременно. Таким образом, атомы “сцепляются”. Такое сцепление называется **ковалентная связь**.

Добавим больше атомов кремния. Если все делать аккуратно, то атомы будут располагаться в строго определенном порядке рис. 1.9. Этот порядок остается постоянным на протяжении всей структуры. Такую структуру с правильным расположением атомов называют **кристаллической**, а сам материал **кристаллом**.

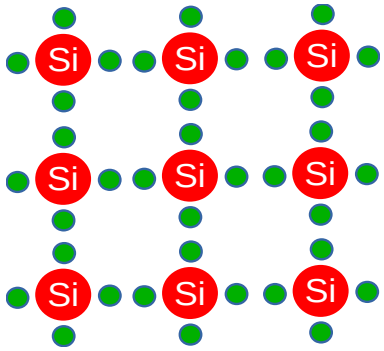


Рисунок 1.9 – Двумерная модель кристалла кремния

На рис. 1.9 приведена упрощенная схема кристалла на плоскости. На самом деле атомы располагаются в пространстве в виде красивой трехмерной кристаллической решетки. В случае кремния решетка относится к типу **алмазной** рис. 1.10.

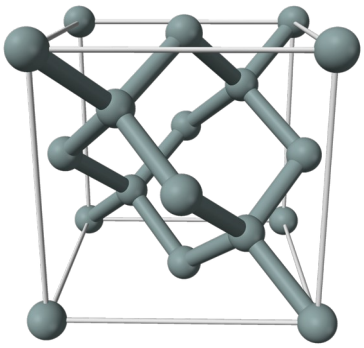


Рисунок 1.10 – Трехмерная модель кристалла кремния.

Следует помнить, что мы живем в трехмерном мире, но рассмотрение все-равно продолжим в пространстве, в котором живут амебы. Так проще.

Исследуем способность только что построенного кристалла проводить электрический ток. Другими словами, оценим проводимость (сопротивление) материала. **Проводимость** показывает, насколько хорошо материал проводит электрический ток. **Сопротивление** – величина обратная проводимости, показывает, насколько плохо материал проводит электрический ток.

Поднесем к одной стороне куска кремния положительный заряд, к противоположной стороне – отрицательный рис. 1.11. Внутри материала возникнет электрическое поле, направленное от плюса к минусу. А появится ли электрический ток?

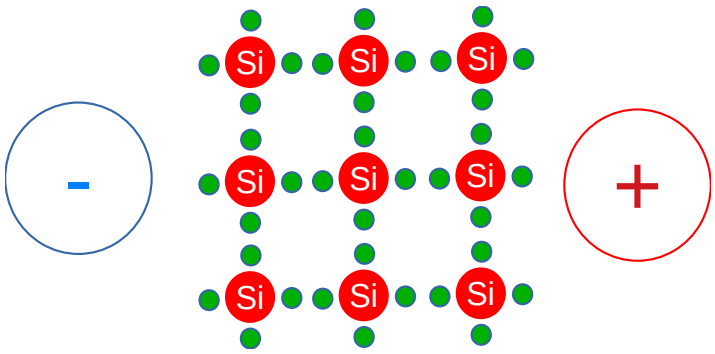


Рисунок 1.11 – Приложение электрического поля к полупроводниковому кристаллу (идеальный случай).

Ток – это движение электронов. Смотрим на рис. 1.11 и видим, что все электроны прочно сидят на своих местах – они сцепляют атомы кремния между собой

посредством ковалентной связи. Внешнее электрическое поле не способно вырвать электроны из связей (хотя очень сильное поле может это сделать, но данный случай нас не интересует). Таким образом, тока нет, проводимость равна нулю, а сопротивление бесконечно.

Вспомним про температуру. Её можно представить как дрожание кристаллической решетки. Атомы немного колеблются. Из-за этих колебаний некоторые электроны отрываются от связей и становятся **свободными** рис. 1.12. Свободный электрон может перемещаться по кристаллу. Также может вернуться обратно в ковалентную связь - занять **вакантное** место. В любом случае, при комнатной температуре (да и при других) в полупроводниках присутствует некоторое количество свободных электронов. Они то и двигаются в электрическом поле, т.е. создают электрический ток.

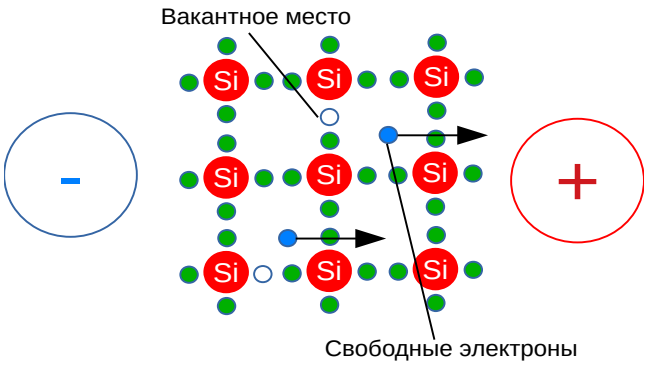


Рисунок 1.12 – Приложение электрического поля к полупроводниковому кристаллу (с учетом температуры)

Полупроводники такого рода называются **собственными**. Проводимость собственных полупроводников мала и создается за счет электронов, оторванных от ковалентных связей из-за температурных колебаний атомов и наличия дефектов кристаллической структуры. В кристаллической структуре собственных полупроводников отсутствуют посторонние атомы.

Собственный полупроводник – плохой диэлектрик, еще более плохой металл. Что толку от этих полупроводников? И правда, толку мало, пока не начнем добавлять в кристаллическую структуру атомы посторонних элементов, например, *Al* или *As*.

1.2.3 Донорный полупроводник

Добавим в кристаллическую структуру полупроводника атом мышьяка *As*. Такие чужие атомы называют **примесными**. Из периодической системы элементов Менделеева, узнаем, что на внешнем уровне мышьяка имеется пять валентных электронов. Четыре валентных электрона мышьяка участвуют в образовании ковалентных связей с атомами кремния. Остается один электрон-неудачник рис. 1.13a. Ему не досталось места ни в одной связи. С атомом мышьяка электрон-неудачник оказывается связан очень слабо и при комнатной температуре с высокой вероятностью отрывается от атома мышьяка и становится свободным рис. 1.13b. Теперь он способен проводить электрический ток. Не такой уж он и неудачник.

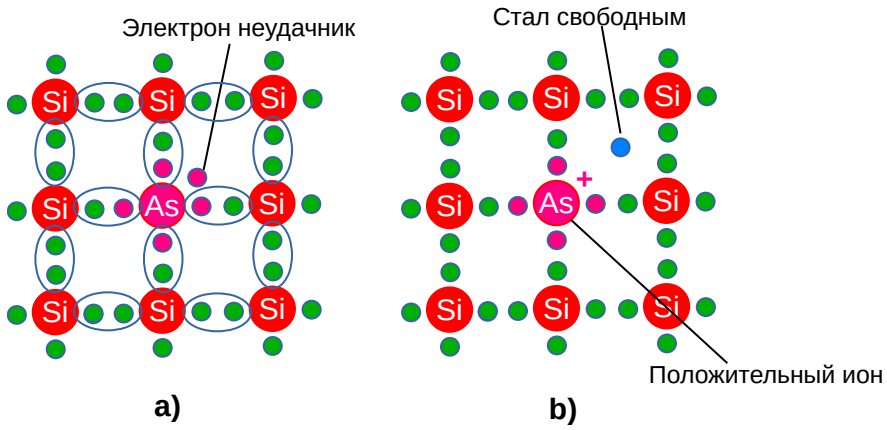


Рисунок 1.13 – Структура донорного полупроводника: a – электрон, который не участвует в образовании связи с кремнием; b – электрон, не участвующий в связи, оторвался, оставив после себя положительно заряженный ион мышьяка.

В целом атомы и кремния, и мышьяка нейтральны. У них одинаковое количество положительных и отрицательных зарядов. Атом мышьяка становится заряженным положительно, после того как электрон покинул его. Действительно, 0 - (-1) = +1. В данном случае, атом мышьяка называют **положительно заряженным ионом примеси**.

Чем больше атомов мышьяка добавлено в структуру, тем больше свободных электронов, тем выше проводимость. Таким образом, изменяя количество (**концентрацию**) атомов мышьяка, можно управлять проводимостью

полупроводника. Если теперь приложить к куску кремния со вставками атомов мышьяка электрическое поле, через структуру потечет электрический ток.

Атомы, которые отдают кристаллической структуре свободные электроны называются **донорной примесью** (аналогия, доноры дают кровь), а сами полупроводники – **донорными полупроводниками**. Процесс добавления примесных атомов в кристаллическую структуру называется – **легированием**.

Донорные полупроводники называют полупроводниками *n*-типа. Символ *n* от англ. *negative* – отрицательный. Ток в полупроводниках *n*-типа осуществляется электронами – *negativ*ными частицами. Все логично!

Итак, изменяя концентрацию донорной примеси, из полупроводника можно сделать металл. Отлично! Почему бы сразу не взять металл вместо того, чтобы осуществлять сложный технологический процесс легирования донорной примесью. Даже звучит непонятно.

1.2.4 Акцепторный полупроводник

Добавим в кристаллическую структуру полупроводника атом алюминия *Al*. Из периодической системы элементов Менделеева узнаем, что на внешнем уровне алюминия имеется только три валентных электрона. Все они участвуют в образовании ковалентных связей с атомами кремния рис. 1.14a. Однако, для образования связи с четвертым атомом кремния, одного электрона не хватает. Т.е. существует свободное для электрона – **вакантное** – место. Оно имеет положительный заряд. Действительно, если присутствие электрона означает отрицательный заряд, то отсутствие электрона – положительный заряд. Это вакантное место может занять электрон из соседней связи рис. 1.14b. В этом случае вакантное (свободное для электрона место) переместится. Атом алюминия в целом был нейтральным. Когда он принял лишний электрон, стал заряжен отрицательно. В данном случае атом алюминия называют **отрицательно заряженным ионом примеси**. Действительно 0 + (-1) = -1.

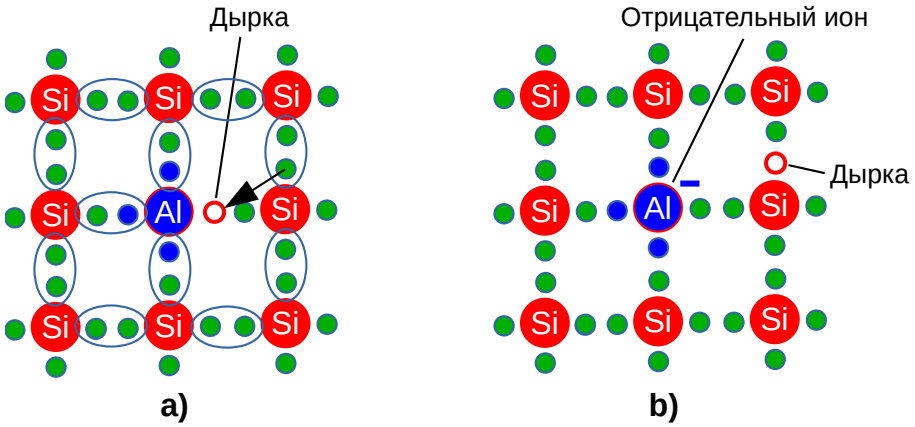


Рисунок 1.14 – Структура акцепторного полупроводника: a – у атома алюминия не хватает электрона, чтобы образовать связь, на его месте находится дырка; b – электрон из соседней связи занял вакантное место, дырка переместилась.

Если приложить электрическое поле, то электроны начнут двигаться против поля, в результате чего вакантное место начнет перемещаться в направлении электрического поля, как самая настоящая положительная частица. Назвали такую частицу **дыркой**. Движение электронов вправо, равносильно движению дырок влево.

Дырка – это положительная **квази**-частица. Приставка квази – означает “ненастоящая”, “как бы-частица”. Если электрон, существующая материальная реальность, то дырки не существует. Это свободное место, которое способен занять электрон. Просто это место ведет себя как частица с положительным зарядом. Поэтому на дырку будем смотреть, как на реально существующую положительную квази-частицу.

Вакантное место может занять и собственный (вырванный температурой из ковалентной связи кремния) свободный электрон. Вероятность этого процесса намного меньше, поскольку самих собственных свободных электронов очень мало. Поэтому учитывать влияние собственных электронов не будем.

Атомы, которые принимают к себе электроны называются **акцепторной примесью** (от англ. *accept* - принимать), а сами полупроводники – **акцепторными полупроводниками**.

Акцепторные полупроводники называют полупроводниками *p*-типа. Символ *p* от англ. *positive* – положительный. Ток в полупроводниках *p*-типа осуществляется дырками – *positiv*ными частицами. Все логично!

Итак, имеется два полупроводника. Хороших полупроводника с высокой проводимостью, как у металлов. С одним только нюансом. В одном полупроводнике ток осуществляется отрицательными частицами – электронами, а в другом положительными – дырками. Уже становиться интереснее. Так и хочется их соединить, чтобы произошла *аннигиляция* материи.

Подытожим. В полупроводниках любого типа проводимость осуществляется электронами. В акцепторных полупроводниках эти электроны не являются свободными. Они перескакивают по вакантным местам. Таким образом происходит перемещение вакантного места, которое эквивалентно перемещению положительной квази-частицы – дырки. Вот и весь секрет. Свободный электрон перемещается по донорному полупроводнику быстро. Прыжки электронов по вакантным местам в акцепторном полупроводнике происходят относительно редко. В связи с этим, скорость дырки ниже скорости электрона. Это следует учитывать при оценке быстродействия устройств.

Справедливости ради, следует отметить, что проводимость в примесных полупроводниках осуществляется не только примесными электронами или дырками (примесная проводимость), но и собственными свободными электронами от атомов кремния (собственная проводимость). Вклад примесной проводимости гораздо выше, поэтому собственную проводимость в дальнейших рассуждениях учитывать не будем.

1.2.5 Контакт акцепторного и донорного полупроводника

Полупроводники *n* и *p* типа на рис. 1.15а изображены по-отдельности. Маленькими кружками обозначены свободные носители заряда. В *n*-типе электроны показаны зелеными кружками, в *p*-типе дырки – красными кружками без заливки. Также на рисунке изображены заряженные ионы примесей. Следует помнить, что эти заряды неподвижны. Они сидят на своих местах – в узлах кристаллической решетки.

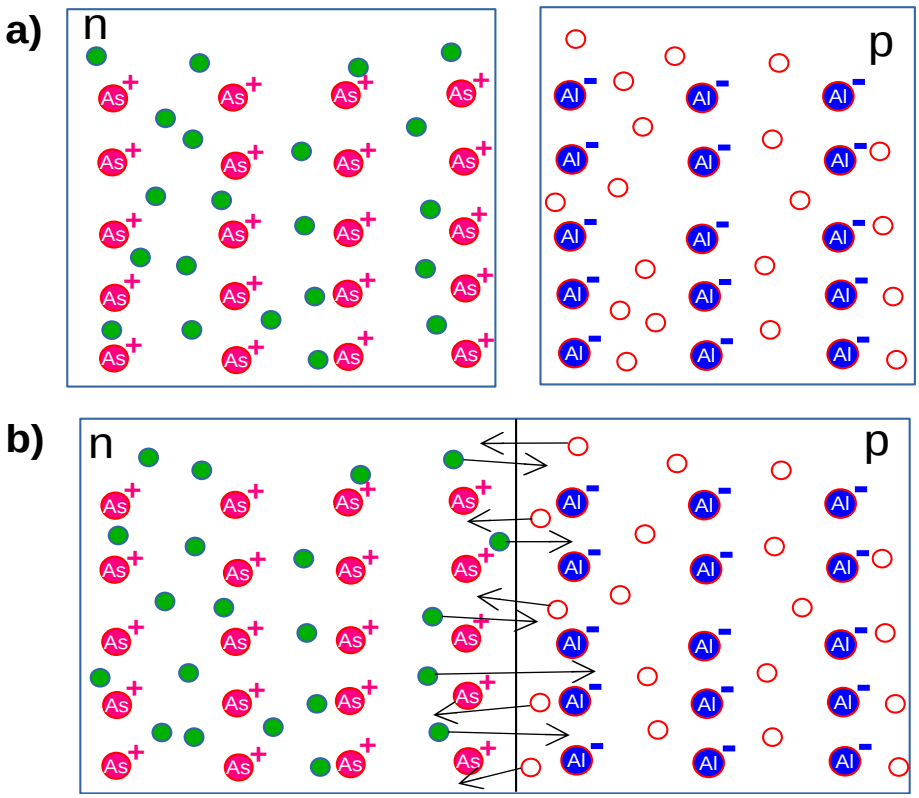


Рисунок 1.15 – Полупроводники n и p-типов: a – до приведения в контакт; b – момент сразу после контакта.

В полупроводнике *n*-типа количество свободных электронов равно количеству ионов примесей, таким образом, заряд *n*-области в среднем равен нулю. Заряд полупроводника *p*-типа также равен нулю. Не показаны на рисунке только атомы кремния, которых гораздо больше, чем примесных атомов.

Расположим вплотную полупроводники *n* и *p* типа рис. 1.15b и понаблюдаем некоторое время за процессами, которые начнут происходить в области контакта. В полупроводнике *n*-типа много свободных электронов, в *p*-типе свободных электронов нет. Электроны из области с высокой концентрацией начнут переходить в область с меньшей концентрацией – из *n*- в *p*-область. Этот процесс называется **диффузией** (см. аналогию распространения запаха от флакона духов по всей комнате), а поток электронов – **диффузионный ток**. Тоже самое происходит с дырками. В итоге мы имеем диффузионный ток электронов из *n*- в *p*-область, и диффузионный поток дырок из *p*- в *n*-область. Данные потоки на рис. 1.15b показаны стрелками.

Посмотрим, что происходит с электронами, ушедшими в *p*-область. В акцепторном полупроводнике много вакантных мест – дырок. Перешедшие электроны с радостью займут эти места. Произойдет уничтожение пары электрон-дырка. Данный процесс носит красивое название **рекомбинация**. Аналогично, дырки, перешедшие в *n*-область, рекомбинируют с примесными электронами этой области.

Вернемся в приконтактную область. После ухода электронов в приконтактной *n*-области остались положительные ионы донорной примеси. После ухода дырок в приконтактной *p*-области остались отрицательные ионы акцепторной примеси. Положительные и отрицательные заряды ионов примесей создают **внутреннее электрическое поле** в области контакта донорного и акцепторного полупроводников рис. 1.16. Поле возникает только в области контакта, в глубине материала ионы примеси по-прежнему скомпенсированы свободными носителями заряда.

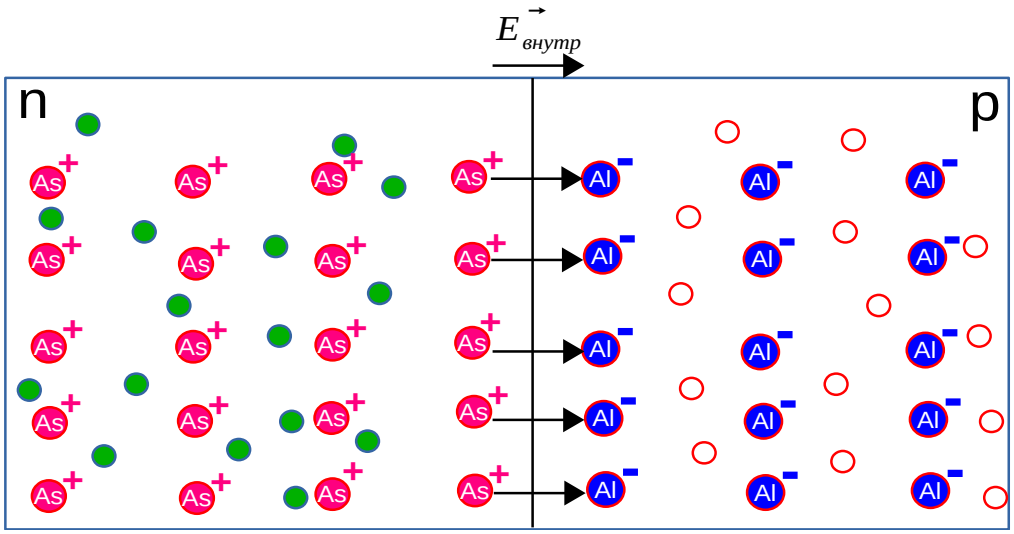


Рисунок 1.16 – Электрическое поле в области контакта полупроводников разного типа

Диффузионный поток пытается выровнять концентрации электронов и дырок во всей структуре, а возникающее электрическое поле стремиться его остановить. Действительно, поместите электрон/дырку в область контакта. Электрическое поле вытолкнет его/её обратно в *n*- / *p*-область. Тем самым поток электронов из *n*- в *p*-область и дырок из *p*- в *n*-область прекратится. Полупроводники между собой не перемешаются и не превратятся в какой-нибудь полупроводник *pn*-типа.

Перемещение заряда под действием электрического поля называется **дрейфовым током**. В нашем случае, дрейфовый ток направлен противоположно диффузионному. Когда оба тока сравниваются наступает **диффузионно-дрейфовое равновесие**. Оно наступает достаточно быстро.

Интересно получается. С одной стороны, электрическое поле создается диффузионным потоком. С другой стороны, возникшее электрическое поле подавляет этот самый диффузионный поток. В итоге, потоки электронов и дырок останавливаются. А область контакта полупроводников оказывается без носителей заряда - **обедненная область**. Отсутствие носителей заряда приводит к уменьшению проводимости и увеличению сопротивления. Получается, что *n*-область – проводящая, *p*-область проводящая, а по середине ОПЗ (**область пространственного заряда**) – область, обедненная носителями заряда, т.е. область с большим сопротивлением рис. 1.17.

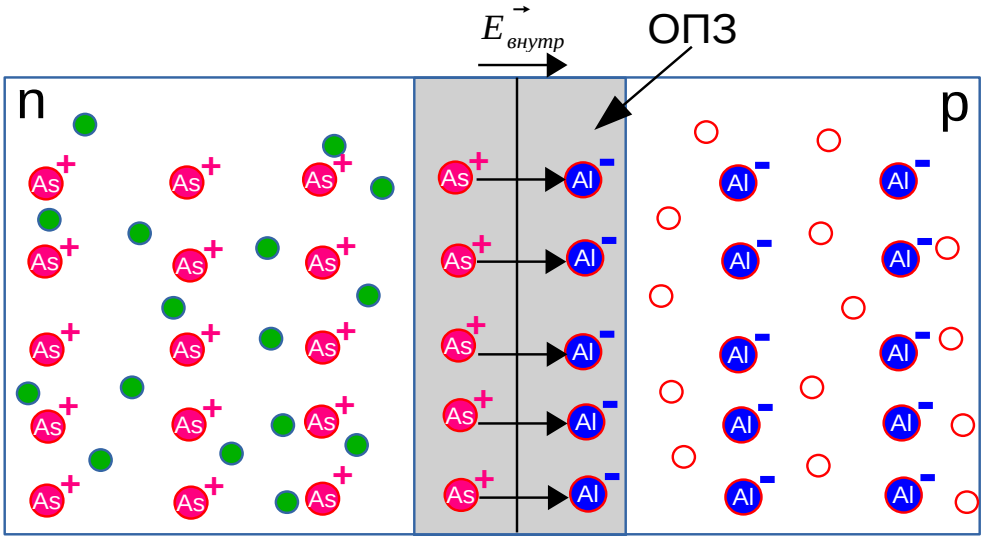


Рисунок 1.17 – Область пространственного заряда – область с высоким сопротивлением.

1.2.6 Прямое смещение p-n перехода

Давайте уже приложим к *p-n* переходу внешнее электрическое поле. На *positive* полупроводник подадим плюс, на *negative* полупроводник минус рис. 1.18. Когда поле прикладывается таким образом, говорят, что *p-n* **переход смещен в прямом направлении**.

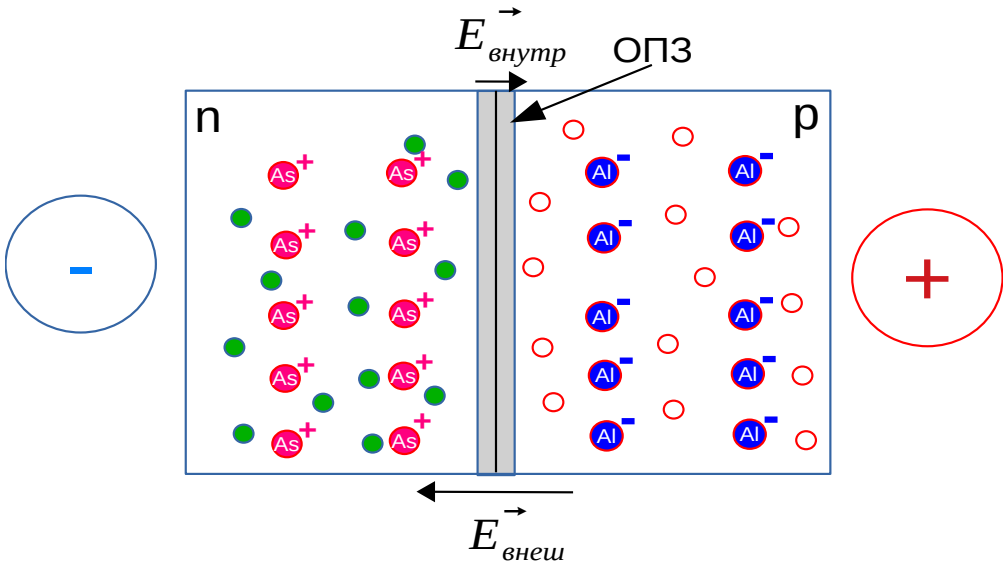


Рисунок 1.18 – Прямое смещение p-n перехода.

Внешнее поле направленно противоположно внутреннему. В результате суперпозиции полей внутреннее поле уменьшается (его гасит внешнее поле), вследствие чего область с высоким сопротивлением **ОПЗ становится уже** и сопротивление *p-n* перехода уменьшается, проводимость возрастает. В итоге, **при прямом смещении *p-n* переход становится проводящим**.

1.2.7 Обратное смещение p-n перехода

Теперь приложим поле в обратном направлении. На *positive* полупроводник подадим минус, на *negative* полупроводник плюс рис. 1.19. Когда поле прикладывается таким образом, говорят, что *p-n* **переход смещен в обратном направлении**.

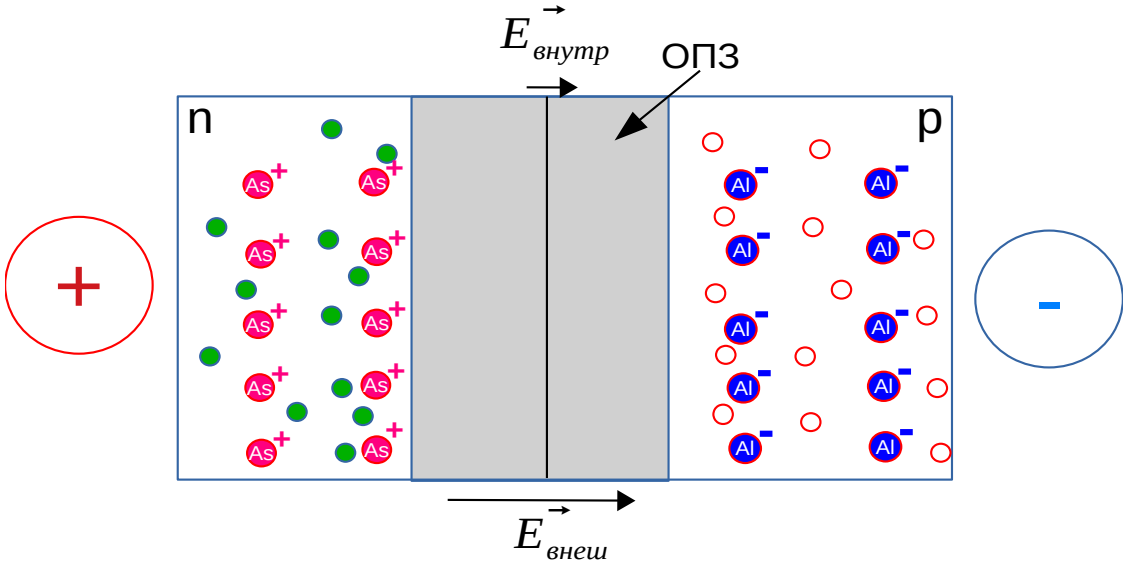


Рисунок 1.19 – Обратное смещение p-n перехода.

Внешнее поле направленно в туже сторону, что и внутреннее. В результате суперпозиции полей внутреннее поле увеличивается. Вследствие этого область с высоким сопротивлением **ОПЗ становится шире** и сопротивление *p-n* перехода увеличивается ещё сильнее, проводимость остается равной близкой к нулю. В итоге, **при обратном смещении *p-n* переход становится непроводящим**.

Итак. При контакте полупроводников *n*- и *p*-типа возникает область с высоким сопротивлением – область пространственного заряда (им. ввиду нескомпенсированный заряд ионов примеси). Изменяя величину и направление электрического поля, можно управляться шириной ОПЗ, а, следовательно, и сопротивлением всей структуры.

Ура! Теперь всё готово для понимания устройства и принципа работы транзистора.

Глава 1.3 Транзистор

Как говорилось ранее, транзистор – основной элемент современных микропроцессоров. Транзистор – это триод – имеет три вывода: входной, выходной и управляющий. Управляющий вывод используется для управления током от входа к выходу.

Рассматривается два типа транзистора: полевой и биполярный. Даются основные сведения об электрических цепях. Демонстрируется работа транзистора в ключевом режиме.

1.3.1 Полевой транзистор

Первым был изобретен биполярный транзистор. Официальная дата 23 декабря 1947 года. Данное изобретение определило облик современной электроники.

Начнем с более простого для объяснения типа транзистора – полевого. Тем более, они в основном и используются в современных микропроцессорах.

Понять принцип работы полевого транзистора легко, поскольку мы уже имеем представление о поведении *p-n* перехода при прямом и обратном смещении. Возьмем параллелепипед из *p-Si* (кремний, легированный акцепторной примесью). На противоположных гранях сделаем карманы *n+*, легированные донорной примесью рис. 1.30. Индекс + означает сильное легирование, а, следовательно, и высокую проводимость легированных участков. Чем выше проводимость материала, тем меньше глубина проникновения электрического поля в материал.

Таким образом, получили p-n+ переход. Мы уже знаем, что в области контакта возникает ОПЗ с большим сопротивлением.

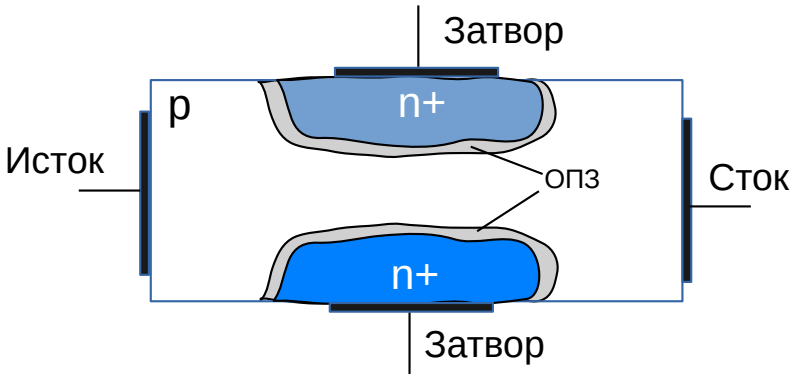


Рисунок 1.20 – Структура полевого транзистора.

На левую и правую торцевые грани нанесем металлические контакты. Один контакт назовем **истоком**, другой **стоком**. Область между истоком и стоком называется каналом. К карманам n+ также подведем контакт - **затвор**. Затвор – это управляющий контакт.

При приложении электрических полей, как показано на рис. 1.21a, ток дырок потечет по каналу от истока к стоку. Сопротивление транзистора, в данном случае мало. Говорят, что транзистор **открыт**.

При поднесении к затвору положительного заряда, *p-n+* переход сместится в обратном направлении. Внешнее электрическое поле будет сонаправлено с внутренним электрическим полем *p-n* перехода, вследствие чего область ОПЗ расширится рис. 1.21b. Проводящий канал при этом сузится (т.к. ОПЗ – ток не проводит). В итоге сопротивление транзистора повысится, а ток дырок уменьшится. Аналогия из жизни: через узкую трубу протечет меньше воды, чем через широкую при одном и том же давлении. В качестве давления выступает электрическое поле от истока к стоку.

При дальнейшем увеличении смещения на затворе ОПЗ полностью перекрывает проводящий канал рис. 1.21c. Сопротивление транзистора станет очень большим. Ток дырок от истока к стоку прекратится. В этом случае говорят, что транзистор **закрыт** или **заперт**.

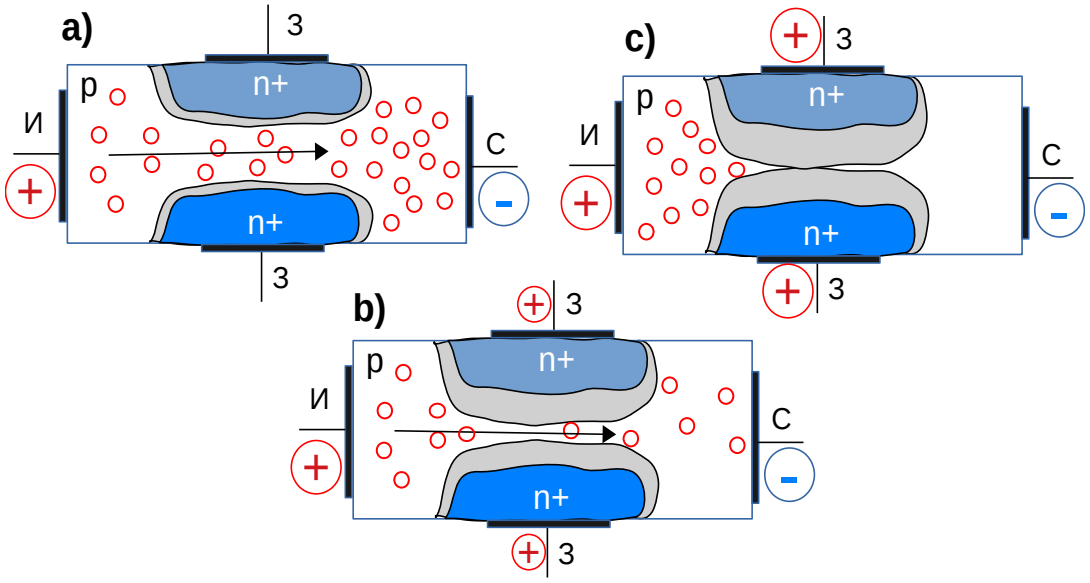


Рисунок 1.21 – Управление полевым транзистором: a – без смещения на затворе, транзистор открыт; b – небольшое смещение на затворе, транзистор начинает закрываться; c – сильное смещение на затворе, транзистор закрыт.

Если транзистор может находиться только в одном из двух состояний – открытом / закрытом, говорят, что транзистор работает в **ключевом режиме**. В качестве ключа выступает поле на затворе. Когда поле большое – транзистор закрыт, когда поле нулевое – открыт рис. 1.22.

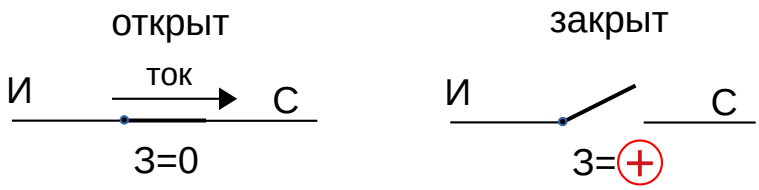


Рисунок 1.22 – Ключевой режим работы транзистора.

Следует отметить, что существует множество разновидностей полевых транзисторов. Здесь можно остановиться надолго, поэтому пойдем дальше

1.3.2 Зажигание лампочки

Разберем схему включения лампочки, заодно вспомним основные понятия электричества – напряжение, ток, сопротивление.

Рассмотрим провод с каким-нибудь элементом, например, лампочкой рис. 1.23. К левому концу провода приложим положительный потенциал, к правому – отрицательный потенциал. При этом электроны потекут справа налево. **Потенциал** – это энергетическая характеристика электрического поля, обозначается буквой φ, подробно рассматривать данное определение не будем.

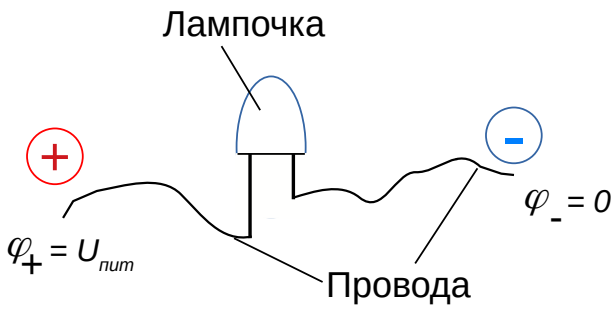


Рисунок 1.23 – Лампа накаливания и потенциалы

Интуитивно понятно, чем сильнее различаются + и – на концах провода, тем сильнее будет ток. Ток зависит от разности + и –. Эту разность потенциалов называют **напряжением**. Для удобства систему отсчета выбирают так, что отрицательный потенциал равен нулю. Тогда напряжение *Uпит* будет равно положительному потенциалу $Uпит = \varphi^{+❑} + \varphi^{-❑} = \varphi^{+❑} - 0 = \varphi^{+❑}$.

Если приложить разность потенциалов к проводу на рис. 1.23 лампочка не загорится, максимум моргнет. Электроны один раз переместятся от 0 к +, и всё. Чтобы лампочка горела постоянно, электроны необходимо вернуть обратно к нулю. Т.е чтобы в схеме тёк ток, **схема должна быть замкнутой**. Обычно цепь замыкается батарейкой (источником питания) рис. 1.24а. Задача батарейки перенести электроны с + на 0 (электроны стремятся, наоборот, от 0 к +), т.е. против действия поля. С данной задачей справляется какая-нибудь сторонняя – не электрическая сила, например, возникающая в результате химических реакций, в щелочных батарейках.

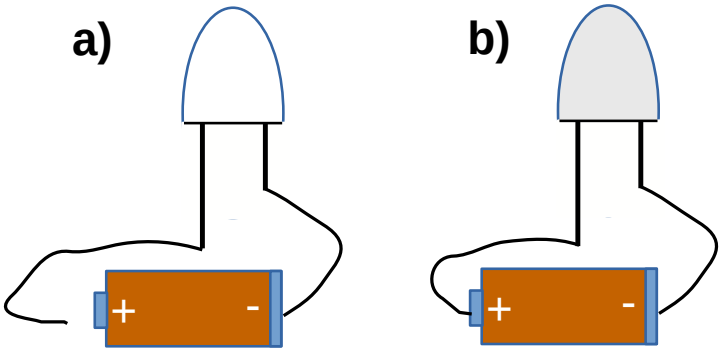


Рисунок 1.24 – Источник питания, подключенный к лампочке: а – цепь разомкнута; б – цепь замкнута

Представим, что светится *p-n* переход. Такая лампочка называется **светодиодом**. Замкнем цепь рис. 1.24b... Разберемся, почему лампочка перегорела.

Напряжение можно представить как силача, который проталкивает электроны от положительного потенциала к нулевому (в схемах он называется **общим проводом** или **землей**). Если электроны на своем пути не встречают никаких препятствий (сопротивлений), то даже слабый силач проталкивает их по проводам и через светодиод с неимоверной силой. Электроны начинают все сильнее ударять по атомам кристаллической структуры кремния. В результате чего колебания атомов, т.е. температура, сильно увеличиваются. Кристаллическая структура кремния разрушается, расплавляется. Этот занимательный эффект называется **тепловой пробой**. К сожалению, необратим.

Лампочка будет долго и счастливо гореть, если удастся угомонить пыл силача. Необходимо оказать ему сопротивление. В электронике существует специальный элемент для этих целей – **резистор**.

Часто резистор называют сопротивлением, что некорректно. Сопротивление – физическая величина. Оно есть и у лампочки, и у проводов, просто очень маленькое. Чтобы светодиод не перегорал, необходимо что-то побольше, например, резистор.

Если резистор используется с целью ограничить ток, то называют его **токоограничивающим резистором**. Резистор можно подключить как перед лампочкой, так и после, главное последовательно рис. 1.25, т.к. ток одинаков во всех элементах, соединенных последовательно.

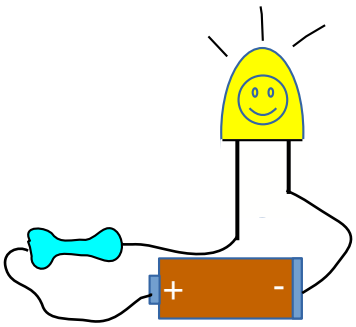


Рисунок 1.25 – Последовательное включение токоограничивающего резистора.

Далее и всегда будем считать, что направление тока противоположно реальному потоку электронов. Так сложилось исторически, за направление тока выбрано направление перемещения положительных частиц. Хотя реально перемещаются отрицательные электроны. Следует смириться и принять, что ток течет от плюса к нулю, также как падает снег с небес на землю январским утром. За тем исключением, что снег реально падает сверху.

Электрическая схема устройства, собранного на рис. 1.25, показана на рис. 1.26.

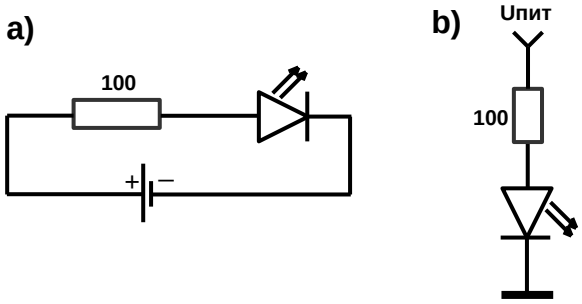


Рисунок 1.26 – Электрическая схема, устройства, изображенного на рис. 1.25.

На схеме резисторы обозначаются прямоугольниками. **Номинал** (сопротивление) резистора обычно записывается без единиц измерения Ом. Если же номинал большой, например, 120 кОм, то пишут 120К, для 5 МОм будет использовано обозначение 5М. В общем, слово Ом опускают, остаются только буквы от приставок.

Треугольником со стрелками наружу обозначается **светодиод**. Стрелки символизируют исходящий свет (у фотодиода, наоборот, под действием света он генерирует напряжение). У светодиода два вывода (потому он и *ДИод*). Эти выводы полярны. Т.е. плюс ножка подключается к плюсу, а минус к минусу (нулю, земле,

общему проводу). Визуально следует запомнить, что ток через диод течет вдоль стрелки (вдоль треугольника), тогда ошибиться при добавлении диода на схему будет проблематично.

Земля (общий провод) обозначается перевернутой буквой **Т**, она подсоединяется к минусу источника питания рис. 1.26b. Оба рисунка отображают одну и туже реальность. Будем пользоваться обозначениями, представленными на рис. 1.27b.

Схему необходимо собирать так, чтобы, направившись от плюса по любым разветвлениям, в итоге прийти к общему проводу, земле, перевернутой букве **Т**, минусу, короче, к нулю. Иначе цепь не будет замкнутой и ток не потечет. Точнее потечет, но не везде, а только по замкнутой её части.

1.3.3 Биполярный транзистор в качестве ключа

Устройство биполярного транзистора отличается от полевого транзистора. В основе работы также лежит *p-n* переход рис. 1.27a. Точнее уже два *p-n* перехода, включенных встречно друг другу. Углубляться в физику процессов не будем.

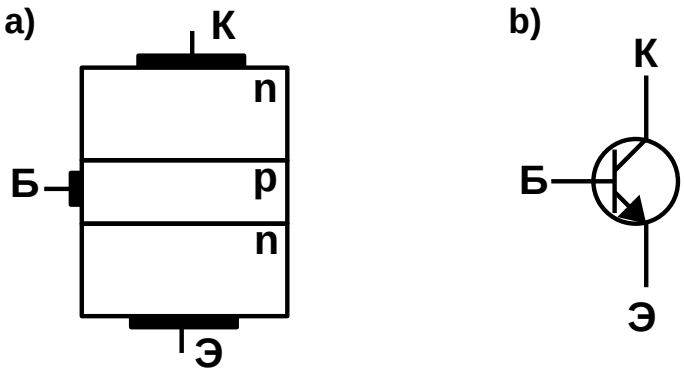


Рисунок 1.27 – Схематическое изображение структуры npn транзистора (a) и его обозначение на электрических схемах (b).

На схемах биполярный транзистор изображается, как показано на рис. 1.27b. Как и у всех триодов, биполярный транзистор имеет три вывода. **Эмиттер** – поставщик носителей заряда (электронов или дырок), **коллектор** – собиратель (коллекционер) носителей заряда, **база** – управляющий вывод. Регулируя напряжение на базе,

управляют сопротивлением транзистора, и, как следствие, током от эмиттера к коллектору. Сравни с истоком, стоком, затвором у полевого транзистора.

Рассмотрим подробно работу схемы рис. 1.28, в которой транзистор выполняет роль ключа – пропускает, либо не пропускает через себя ток.

Напряжение питания *Uпит* подается на коллектор. Оно стремится протащить поток носителей заряда к земле через эмиттер (реально электроны текут в противоположную сторону от эмиттера к коллектору). Токоограничивающий резистор *r* необходим, чтобы транзистор не перегорел в открытом состоянии, когда его сопротивление мало.

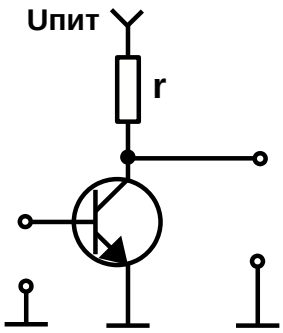


Рисунок 1.28 – Схема включения транзистора в качестве ключа.

Разберем случай, когда напряжение на базе относительно земли равно нулю (соединим базу с землей) рис. 1.29a. При этом сопротивление транзистора оказывается очень большим, гораздо больше, чем сопротивление токоограничивающего резистора. Вместо рис. 1.29a можно составить эквивалентную схему рис. 1.29b.

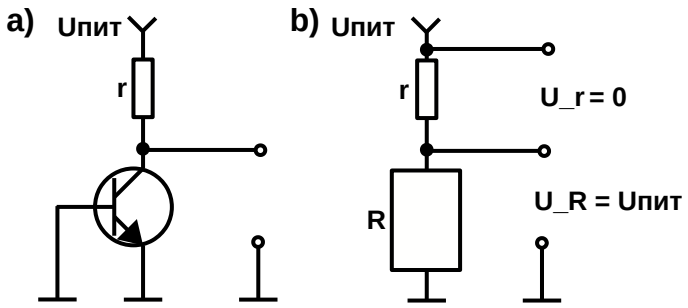


Рисунок 1.29 – Работа транзистора в ключевом режиме, на базу подано низкое (нулевое) напряжение: a – электрическая схема; b – эквивалентная электрическая схема.

Вспомним, что делает напряжение. Оно пытается протолкнуть носители заряда через все элементы схемы к земле. На рис. 1.29b имеется два резистора и провода. Сопротивление проводов мало, поэтому напряжение без особого труда проталкивает вдоль них носители заряда. Сопротивление резисторов значительно больше, чем у проводов. Напряжение тратит основные силы, чтобы протащить носители заряда именно через резисторы.

Сопротивление резистора *R* гораздо больше, чем сопротивление токоограничивающего резистора *r*, поскольку транзистор закрыт и не пропускает ток. Соответственно, напряжение тратит больше всего усилий, на проталкивание зарядов через резистор *R*. Говорят, что напряжение падает на резисторе *R*. Напряжение на транзисторе *U_R* намного больше, чем напряжение *U_r*. Если *R* стремится к бесконечности, то все напряжение падает на *R*, т. е. *U_R = Uпит*.

Рассмотрим другой случай. Напряжение на базе относительно земли равно напряжению питания *Uпит* рис. 1.30a. При этом сопротивление транзистора оказывается очень маленьким, гораздо меньше, чем сопротивление токоограничивающего резистора (*r* << *R*). Вместо рис. 1.30a можно составить эквивалентную схему рис. 1.30b. Теперь напряжение падает на резисторе *r*: *U_r = Uпит*, *U_R = 0.*

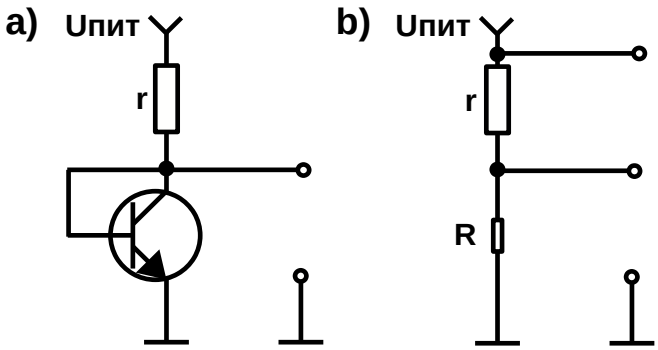


Рисунок 1.30– Работа транзистора в ключевом режиме, на базу подано высокое напряжение.

Делаем следующие важные выводы:

1. Если на базе напряжение равно нулю, то напряжение на транзисторе равно напряжению питания, транзистор закрыт, тока от коллектора к эмиттеру нет.

2. Если на базе напряжение равно напряжению питания, то напряжение на транзисторе равно нулю, транзистор открыт, от коллектора к эмиттеру идет значительный ток, который ограничивается резистором *r*.

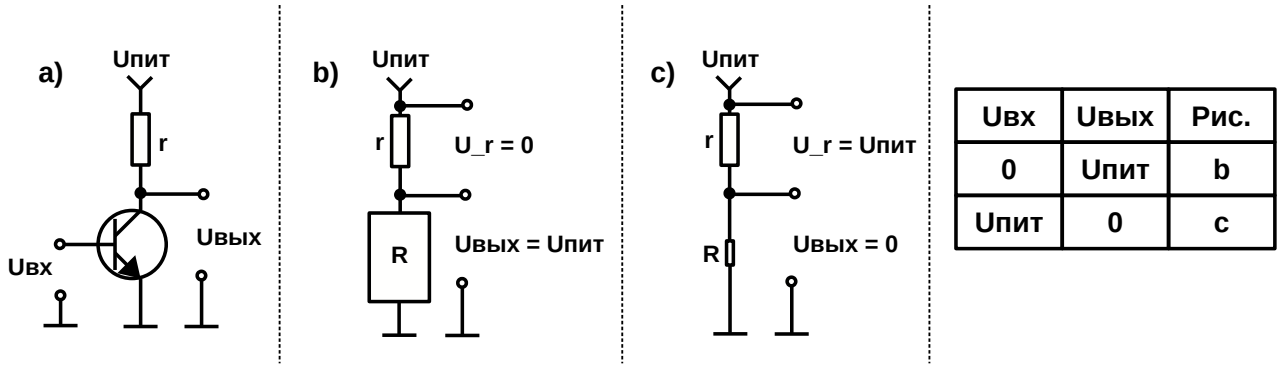
Глава 1.4 Устройство базисных логических элементов

Рассматриваются электрические схемы основных логических элементов – *НЕ*, *И-НЕ*, *ИЛИ - НЕ*. Дается представление о транзисторно-транзисторной логике (ТТЛ). Преодолевается грань между физикой и логикой.

1.4.1 НЕ

Самый простой элемент называется **инвертор, отрицание** или **НЕ**. Инвертор изменяет низкое напряжение на высокое, а высокое на низкое.

У инвертора имеется два вывода: вход и выход. На выходе появляется значение, противоположное значению, поданному на вход. Транзистор в режиме ключа, на самом деле, выполняет функцию инвертора. Входным значением является напряжение на базе, а выходным – напряжение на транзисторе рис. 1.31. На рис. 1.31b – показана эквивалентная схема инвертора, при подаче на вход нулевого напряжения. В данном случае транзистор закрыт и имеет высокое сопротивление. Нам известно, что напряжение падает в основном на более высоком сопротивлении. Соответственно, выходное напряжение будет равно напряжению питания. На рис. 1.31c, ситуация противоположная.

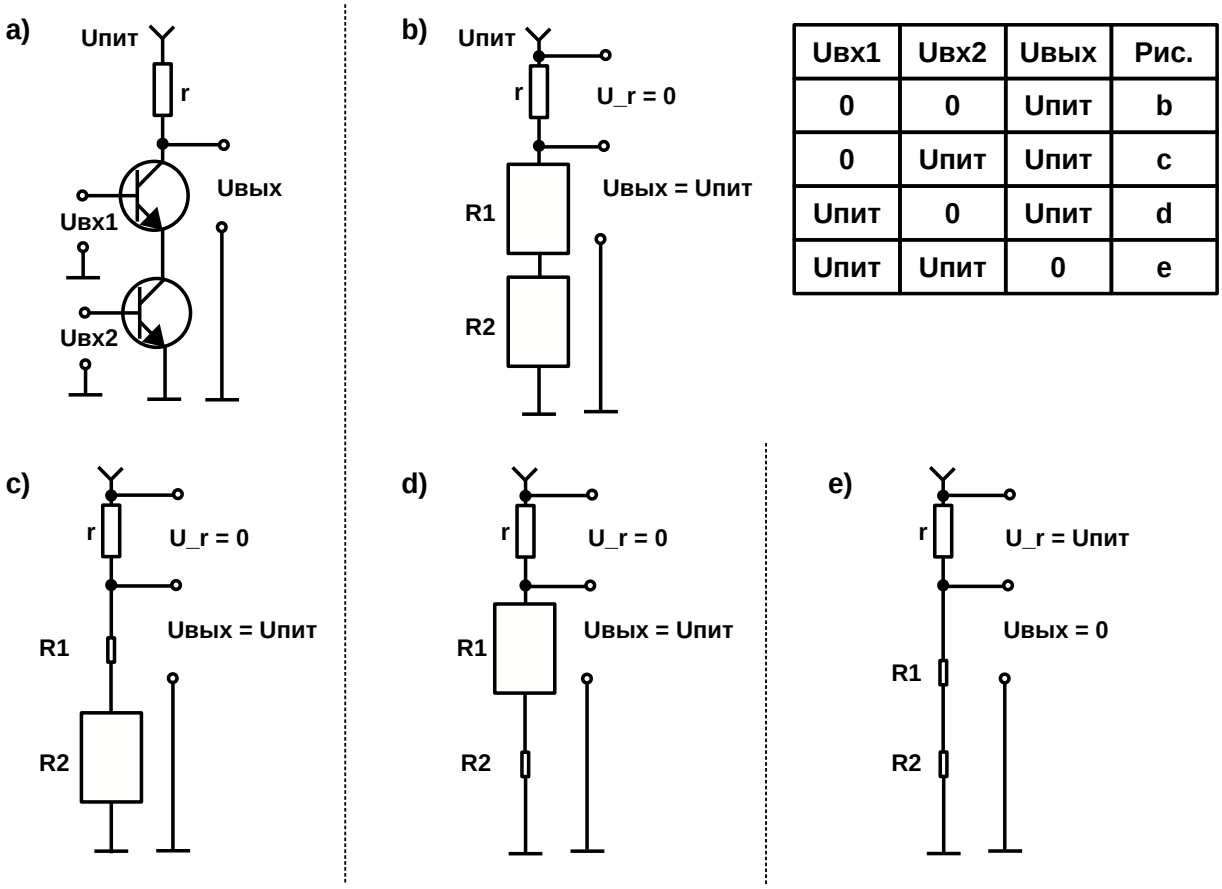


Uвх	Uвых	Рис.
0	Uпит	b
Uпит	0	c

Рисунок 1.31 – Схема инвертора: a – электрическая схема, b – эквивалентная схема, при подаче на вход нулевого напряжения, c – эквивалентная схема, при подаче на вход высокого напряжения.

1.4.2 И-НЕ

У элемента *И-НЕ* два входа и один выход. На выходе будет нулевое напряжение в единственном случае – когда на оба входа подано высокое напряжение.



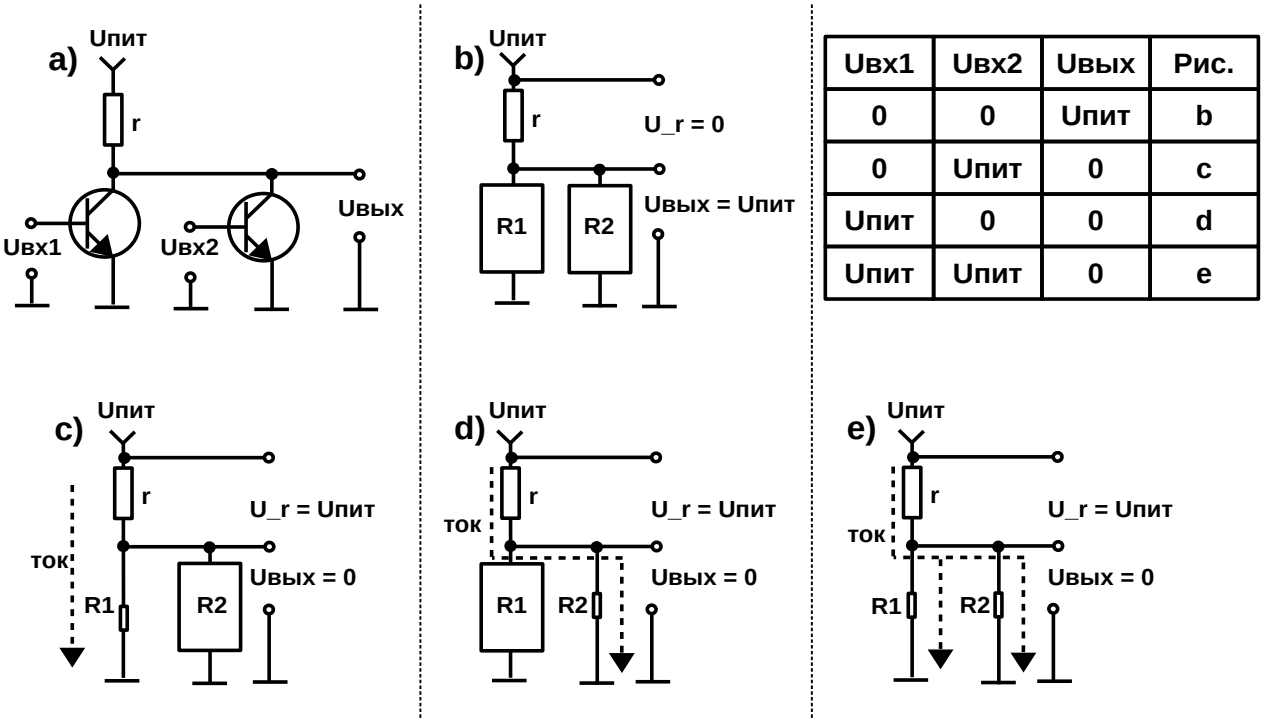
Uвх1	Uвх2	Uвых	Рис.
0	0	Uпит	b
0	Uпит	Uпит	c
Uпит	0	Uпит	d
Uпит	Uпит	0	e

Рисунок 1.32 – Схема элемента И-НЕ: a – электрическая схема; b – эквивалентная схема при Uвх1 = Uвх2 = 0; c – эквивалентная схема при Uвх1 = Uпит , Uвх2 = 0; d – эквивалентная схема при Uвх1 = 0 ,Uвх2 = Uпит; e – эквивалентная схема при Uвх1 = Uвх2 = Uпит.

Схема элемента *И-НЕ* показана на рис. 1.32a. Она представляет собой два транзистора, соединенных последовательно. Если хотя бы один транзистор закрыт, то *Uвых = Uпит* рис. 1.32b, c, d. Только когда оба транзистора открыты, все напряжение упадет на токоограничивающем резисторе и напряжение на выходе станет равным нулю, рис. 1.32 e.

1.4.3 ИЛИ-НЕ

У элемента *ИЛИ-НЕ* также два входа и один выход. На выходе будет высокое напряжение в единственном случае – когда на оба входа подано низкое напряжение.



Uвх1	Uвх2	Uвых	Рис.
0	0	Uпит	b
0	Uпит	0	c
Uпит	0	0	d
Uпит	Uпит	0	e

Рисунок 1.33 – Схема элемента ИЛИ-НЕ: a – электрическая схема; b – эквивалентная схема при Uвх1 = Uвх2 = 0; c – эквивалентная схема при Uвх1 = Uпит , Uвх2 = 0; d – эквивалентная схема при Uвх1 = 0, Uвх2 = Uпит; e – эквивалентная схема при Uвх1 = Uвх2 = Uпит.

Схема элемента *ИЛИ-НЕ* показана на рис. 1.33a. Она представляет собой два транзистора, соединенных параллельно. На рис. 1.33c, d, e показаны случаи, когда открыт хотя бы один транзистор. Ток устремится через открытый транзистор и сопротивление участка цепи с транзисторами, будет низким. Все напряжение упадет на токоограничивающем резисторе, поэтому *Uвых = 0*. Только когда оба транзистора закрыты, сопротивление участка цепи с транзисторами, будет очень большими и напряжение упадет на выходе *Uвых = Uпит* рис. 1.33b.

1.4.4 Физика переходит в логику

На микропроцессор можно смотреть с двух сторон: физической и логической. Физическую сторону мы разобрали – управляемые *p-n* переходы, с летающими электронами, резисторы с падающими напряжениями и т.д. Но пока это просто кусок земли. Осмысленность нашему устройству придает логика. Следуя законам логики, можно заставить микропроцессор выполнять какие-то полезные действия. Например, вычисления, управление чем-либо, загрузка котенка через интернет и вывод его на экраны.

Наверняка вы слышали, что компьютер умеет работать только с нулями и единицами. Логика же работает с высказываниями, принимающими значения **ИСТИНА** либо **ЛОЖЬ**. Обозначим слово **ИСТИНА** цифрой **1**, а слово **ЛОЖЬ** цифрой **0**. Получается, что компьютер логическое устройство. Он умеет работать только с «ложностями» и «истинностями», с нулями и единицами.

Как же перейти от физических напряжений к логическим нулям и единицами? Известно, что слово обладает великой силой. Люди договорились, что низкое напряжение соответствует логическому нулю, а высокое напряжение – логической единице.

Что считать за высокий и низкий уровень? Все зависит от договоренности. Самая распространенная договоренность получила название **уровни ТТЛ** (уровни **транзисторно-транзисторной логики**) рис. 1.34. Обратите внимание, что логической единице и нулю соответствует не конкретное напряжение, а диапазон. Это сделано с целью обеспечить надежность работы схемы, т.к. напряжение может немного отклоняться в большую или меньшую сторону.

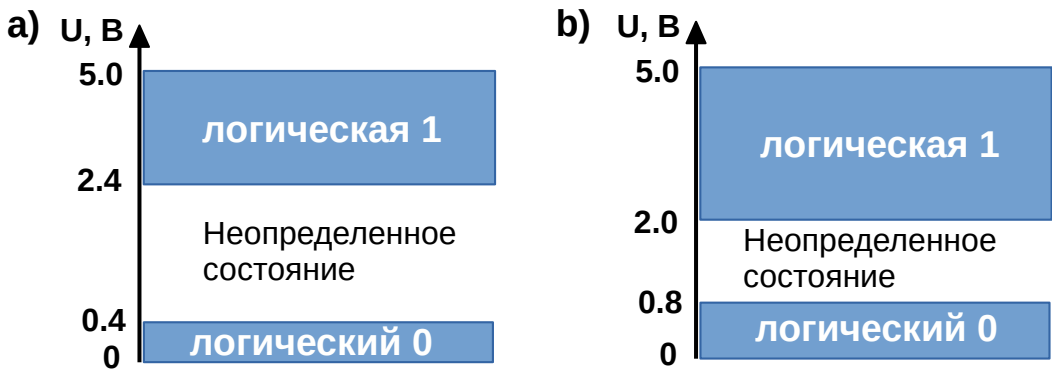


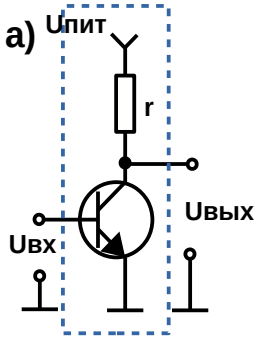
Рисунок 1.34 – Уровни логических сигналов ТТЛ: a – на выходе; b – на входе.

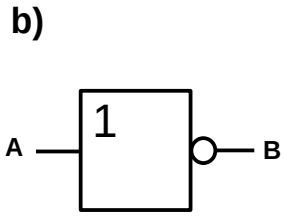
В ТТЛ логической единице соответствует высокое напряжение +5 Вольт, либо чуть меньше. Логическому нулю соответствует низкое напряжение 0 Вольт или немного большее. Обратите внимание, что диапазоны для выходных (рис. 1.34a) и входных (рис. 1.34b) значений немного различаются.

В электрических схемах ТТЛ на входах и выходах не могут возникать напряжения, соответствующие неопределенным состояниям. Если такое происходит, то схема уже не выполняет свое логическое назначение.

На рис. 1.35, 1.36, 1.37 показан переход от физических электрических схем к логическим схемам. Здесь мы воспользовались **концепцией черного ящика**.

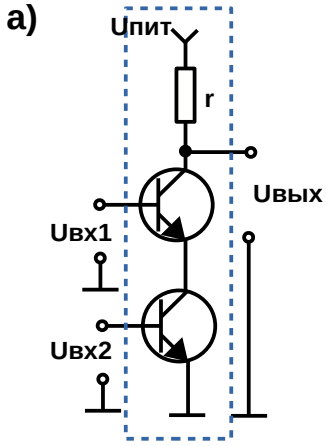
Концепция черного ящика заключается в следующем: сложная внутренняя реализация какого-либо устройства (выделана на рисунках пунктиром) скрывается от глаз рис. 1.35b, 1.36b, 1.37b в “черный ящик”. В дальнейшем частенько придется прятать сложные схемы в “черные ящики”, пока не получится в качестве черного ящика микропроцессор. Более подробно логическая суть данных элементов рассматривается далее.

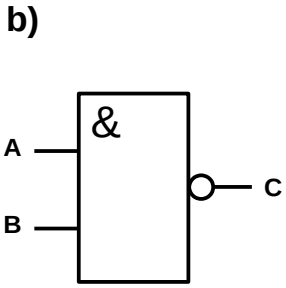




A	B
0	1
1	0

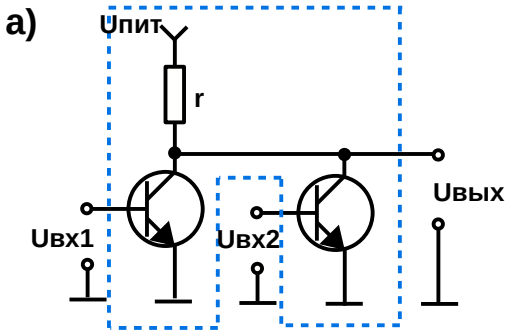
Рисунок 1.35 – Элемент НЕ (инвертор): a – электрическая схема; b – логическая схема; c – таблица истинности.

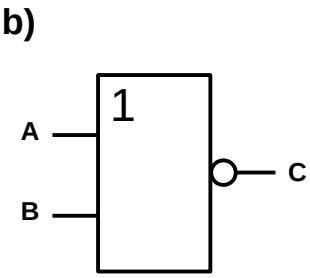




A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Рисунок 1.36 – Элемент И-НЕ: a – электрическая схема; b – логическая схема; c – таблица истинности.





A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Рисунок 1.37 – Элемент ИЛИ-НЕ: a – электрическая схема; b – логическая схема; c – таблица истинности.

Отметим, что также широко распространена договоренность LVTTL (low voltage TTL) – ТТЛ с меньшим уровнем высокого напряжения, равным 3 Вольта. Низкие напряжения позволяют уменьшить ток, потребляемый устройством, и экономнее расходовать заряд источника питания, что, к сожалению, актуально в современном мире…

Элементы *ИЛИ-НЕ, И-НЕ* являются базисными. Из данных элементов можно построить любую логическую схему, в т.ч. и микропроцессор. Иными словами, приобретя несколько камазов с элементами *ИЛИ-НЕ* или *И-НЕ*, не составит особого труда собрать из них микропроцессор, чем и начнем заниматься в следующем разделе.

Раздел 2 Логика

Рассматриваются основы алгебры логики. Из базисных элементов собираются схемы основных логических операторов *И, ИЛИ, НЕ*. Далее конструируется сумматор. Разбирается устройство некоторых логических схем и памяти.

Глава 2.1 Основные логические элементы

Прослеживается аналогия обыкновенной алгебры (точнее арифметики) с алгеброй логики. Из базисных логических элементов конструируются элементы, реализующие основные операции алгебры логики.

2.1.1 Операторная форма записи

Любая алгебра создана для выполнения операций над объектами. В результате выполнения операций по определенным правилам получаются новые объекты.

Например, школьная арифметика. Здесь алгебра имеет дело с числами. Производя операции сложения, умножения и деления над числами по строго определенным правилам, формируются новые числа. Арифметика необходима для вычисления количественных показателей некоторых процессов или явлений, например, зарплаты.

Со временем люди формализовали арифметику, т.е. стали записывать выражения с использованием специальных знаков. **Формализация** от слова форма. Наши предки придали арифметике ту удобную форму, которую многие так не любят в школе.

Рассмотрим выражение **5 + 7**. В этой записи символом плюс обозначается сложение. Символ **+** называется **оператором**. Оператор в том смысле, что он совершает операцию по правилам сложения. Числа **5** и **7** называют **операндами**. Таким образом, о**ператор совершает операцию над операндами**.

Попробуйте немного абстрагироваться от привычного восприятия **5 + 7**, и посмотреть на это выражение как на оператор, который воздействует на два операнда. Другой пример, **A * B** – тоже операторная форма записи. В приведенных выражениях операторы **+** и ***** действуют на два операнда. Операторы, действующие на два операнда, называются **бинарными операторами**. На три операнда – **тернарными**. И самое интересное, если оператор действует на один операнд, то он **унарный**. Примером унарного оператора служит **инкремент**. В Си подобных языках инкремент обозначается знаком ++. Например, запись 5++ или ++5 даст шесть. Инкремент увеличивает значение на единицу.

2.1.2 Основные операторы алгебры логики

Алгебра логики имеет дело с логическими высказываниями. Она позволяет составлять из высказываний целые выражения, делать логические умозаключения. Появляется возможность на листочке формально записать некоторую логическую мысль, также как **5 + 7**. Не правда ли завораживает? Алгебра логики настолько проста, что обрабатывать такие формальные записи может даже электронная вычислительная машина (ЭВМ). На самом деле, ЭВМ — чисто логическое устройство. Арифметические операции в компьютерах вычисляются логическими операторами.

В качестве операндов в алгебре логики выступают логические высказывания. **Логическое высказывание** – утверждение, которое может принимать только два значения: **ИСТИНА** либо **ЛОЖЬ**.

Мда.. так себе определение. Например, утверждение, что в данный момент вы читаете книгу **ИСТИНА**, а что вы сейчас читаете книгу на английском языке **ЛОЖЬ**, хм.. хотя кто знает :).

С операндами разобрались. Что насчет операторов? Какие операции можно совершить над **ИСТИНОЙ** и **ЛОЖЬЮ**? Их всего три: дизъюнкция, конъюнкция, инверсия. Не смотря на красивые названия, они выполняют достаточно простые операции над операндами. А результатом также является опять таки логическое высказывание **ЛОЖЬ** или **ИСТИНА**.

Поскольку количество значений, которые могут принимать операнды равно всего лишь двум, то для всех операторов (и любых выражений) в алгебре логике можно составить таблицу истинности. В **таблице истинности** перечисляются все возможные комбинации операндов и соответствующий им результат операции (значение выражения). В случае бинарных операторов, в таблице истинности присутствует всего лишь четыре строчки. Отметим, что в обычной арифметике, строчек будет бесконечно много.

Обозначим **ИСТИНУ** цифрой **1**, а **ЛОЖЬ** цифрой **0**. Приступим.

2.1.3 Конъюнкция

Логическая операция, в результате выполнения которой **1** возникает в единственном случае, когда оба операнда **1**. Оператор конъюнкция обозначается символом $\square\wedge\square$. Иногда используются *&, AND, И*.

$0\wedge 0=0 \qquad 0\wedge 1=0 \qquad 1\wedge 0=0 \qquad 1\wedge 1=1$

Данные выражения можно свести в таблицу истинности

A	B	$A\wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

По своей логической сути, конъюнкция сродни союзу *И*. В том смысле, что **ИСТИНА** получается, когда первый операнд **ИСТИНА** *И* второй операнд **ИСТИНА**. Заставим алгебру логики сделать полезный вывод на простом примере.

Представим, что вы ребенок и составили с родителями договор. Вам дадут карманные деньги в том случае, если вы показываете отличные успехи в учебе и спорте. Пусть высказывание **A** означает, являетесь ли вы ударником, а высказывание **B** занимаетесь ли вы спортом.

A = 1 – вы хорошо учились в этом месяце,
A = 0 – плохо учились в этом месяце,
B = 1 – регулярно тренируетесь в спортивной секции,
B = 0 – пропускаете тренировки.
C = 1 – вам дадут карманные деньги
C = 0 – карманных денег вам не видать.

Допустим, что родители следуют принципу конъюнкции. Чтобы понять, давать ли деньги, они будут использовать следующее логическое выражение $C=A\wedge B$. Т.е. деньги вы получите в единственном случае, когда хорошо учитесь и ответственно подходите к занятиям спортом. Достаточно строго, но справедливо. Таким образом, алгебра логики позволяет выражать мысли в виде формул.

2.1.4 Дизъюнкция

Логическая операция, в результате выполнения которой **1** возникает в случае, когда хотя бы один операнд равен **1**. Оператор дизъюнкция обозначается перевернутым символом $\blacksquare\vee\blacksquare$. Иногда используются *OR*, *ИЛИ*, в Си-подобных языках символ вертикальная черта - |.

$0\vee0=0 \qquad 0\vee1=0 \qquad 1\vee0=0 \qquad 1\vee1=1$

Данные выражения можно свести в таблицу истинности

A	B	$A\vee B$
0	0	0
0	1	1
1	0	1
1	1	1

По своей логической сути, дизъюнкция сродни союзу *ИЛИ*. В том смысле, что **ИСТИНА** получается, когда первый операнд **ИСТИНА** *ИЛИ* второй операнд **ИСТИНА**.

Вернемся к примеру, с карманными деньгами. Допустим, что родители теперь следуют принципу дизъюнкции. Чтобы понять, давать ли деньги, они будут использовать следующее логическое выражение $C=A \vee B$. Т.е. деньги вы получите во всех случаях, кроме единственного – когда вы истинный лентяй, плохо учитесь и не занимаетесь спортом. Очевидно, люди, которые следуют принципам дизъюнкции, более лояльны.

Сделаем важные выводы, которые пригодятся при конструировании арифметико-логического устройства:

- выражение, состоящее из одних только конъюнкций, будет **ЛОЖНО**, если *хотя бы один* операнд **ЛОЖЬ**.
- выражение, состоящее из одних только дизъюнкций, будет **ИСТИННО**, если *хотя бы один* операнд **ИСТИНА**.

2.1.5 Инверсия

Логическая операция, в результате выполнения которой операнд изменяется на противоположный. Оператор инверсии обозначается перевернутым символом $\neg$❑ . Иногда используются *NOT, НЕ*, в Си-подобных языках символ *!*. Но самое удобное обозначение это штрих над операндом.

$\bar{0}=1 \quad \bar{1}=0$

Данные выражения можно свести в таблицу истинности

A	$\bar{A}$
0	1
1	0

Часто инверсию называют **отрицание**. По своей логической сути, она сродни частице *НЕ*. В том смысле, всё что стоит после этой частицы отрицается.

Вернемся к примеру, с карманными деньгами. Допустим, что в семье возникли небольшие разногласия по-поводу воспитания. Мама считает, что давать деньги необходимо, следуя принципу дизъюнкции, а папа, который дает *cash*, за строгое

воспитание конъюнкцией. Решение принимает мама, а папа делает все с точностью наоборот. Получим нечто следующее $C_{папа} = \neg C_{мама}$. Хороший пример с точки зрения логики, но плохой со стороны психологии семейной жизни, необходимо искать компромиссы.

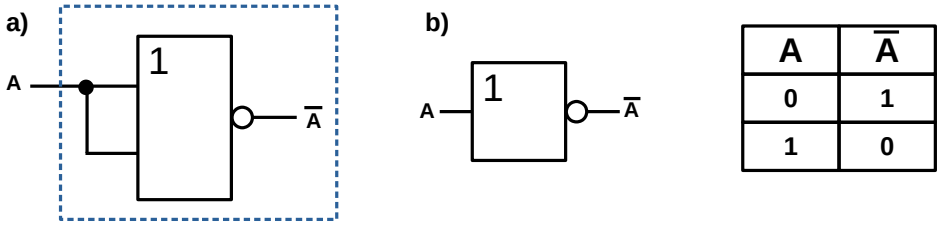
2.1.6 Схемы основных логических элементов

Перейдем к элементам, которые позволяют реализовать вышеуказанные операции в железе.

Из железа было разобрано только два элемента *И-НЕ* и *ИЛИ-НЕ*, они выполняют совсем иные операции, нежели основные логические операторы. Но все-таки являются базисными, т.е. из них можно собрать любую логическую схему. Так давайте же возьмем за основу *ИЛИ-НЕ* и соберем из него все остальное.

НЕ

Необходимо взять *ИЛИ-НЕ*, на первый вход будем подавать операнд, второй вход соединим с первым рис. 2.1. В результате операции *ИЛИ* операнда самим с собой получится тоже самое значение операнда. А на выходе *ИЛИ-НЕ* стоит инверсия. Т.е. когда на входе **0**, на выходе **1**. Когда на вход приходит **1**, на выходе **0**. Все, как и должно быть у инверсии. Разумеется, проще было взять один транзистор, который уже выполняет необходимую операцию, но это будет не честно. Мы же хотим сделать все из одного базисного элемента, в том числе и более простые элемент, чем сам базис.

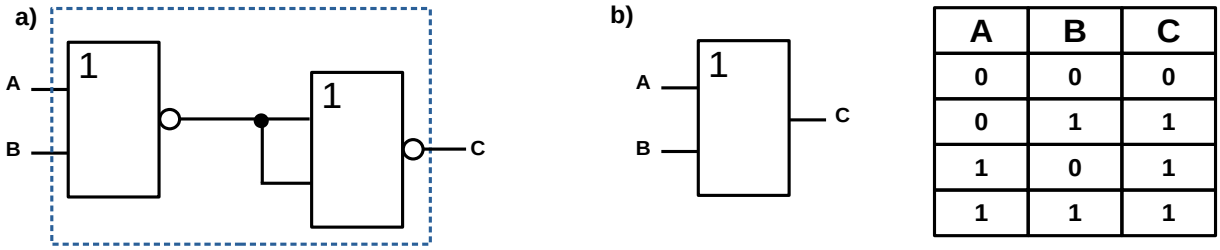


A	$\overline{A}$
0	1
1	0

Рисунок 2.1 – Элемент НЕ: a – внутреннее устройство, b – обозначение на схемах.

ИЛИ

Очевидно, элемент *ИЛИ* изготовить проще пареной репы. Необходимо на выход базисного элемента *ИЛИ-НЕ* поставить инвертор рис. 2.2.



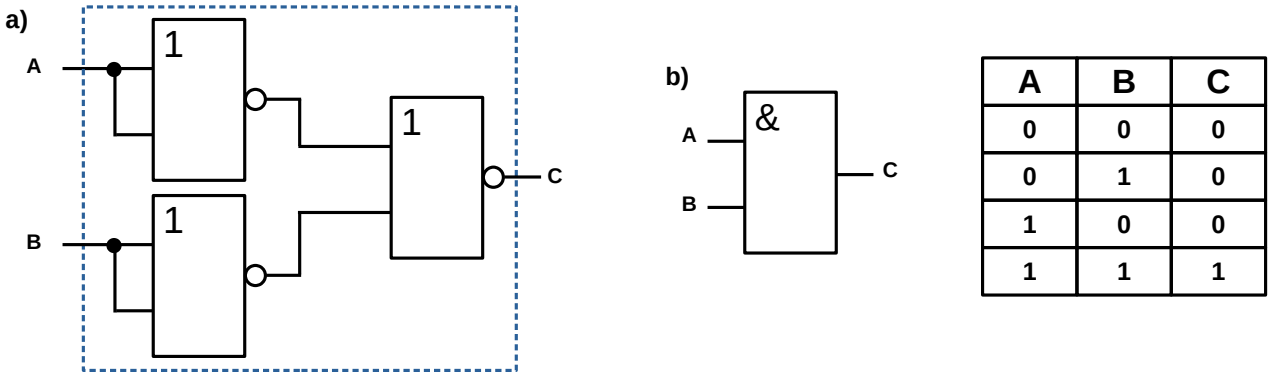
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Рисунок 2.2 – Элемент ИЛИ: a – внутреннее устройство, b – обозначение на схемах.

На схемах элемент *ИЛИ* обозначается также как *ИЛИ-НЕ*, только без кружочка на выходе. К слову кружочек – означает инверсию. *ИЛИ-НЕ*, это элемент *ИЛИ* с инверсией на выходе В данном случае говорят, что **элемент** *ИЛИ* **с инвертированным выходом**.

И

С элементом *И* все не так просто. Необходимо использовать три элемента *ИЛИ-НЕ* рис. 2.3. По факту входные элементы *ИЛИ-НЕ* представляют собой инверсии. Третий элемент *ИЛИ-НЕ* получает инвертированные входные значения.



A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Рисунок 2.3 – Элемент И: a – внутреннее устройство, b – обозначение на схемах.

На схемах элемент *И* обозначается также как *И-НЕ*, только без кружочка на выходе. Символ *&* переводится как *AND*.

Проанализируем схему рис. 2.3. На входе и выходе каждого элемента будем записывать логические значения **0** или **1**.

Подадим на оба входа **0**, и протолкнем их сквозь нашу схему, на выходе получим **0** рис. 2.4a. Значение **0** на выходе получится для любой комбинации на входе рис. 2.4a,b,c, за исключением случая, когда на оба входа подана **1** рис. 2.4d

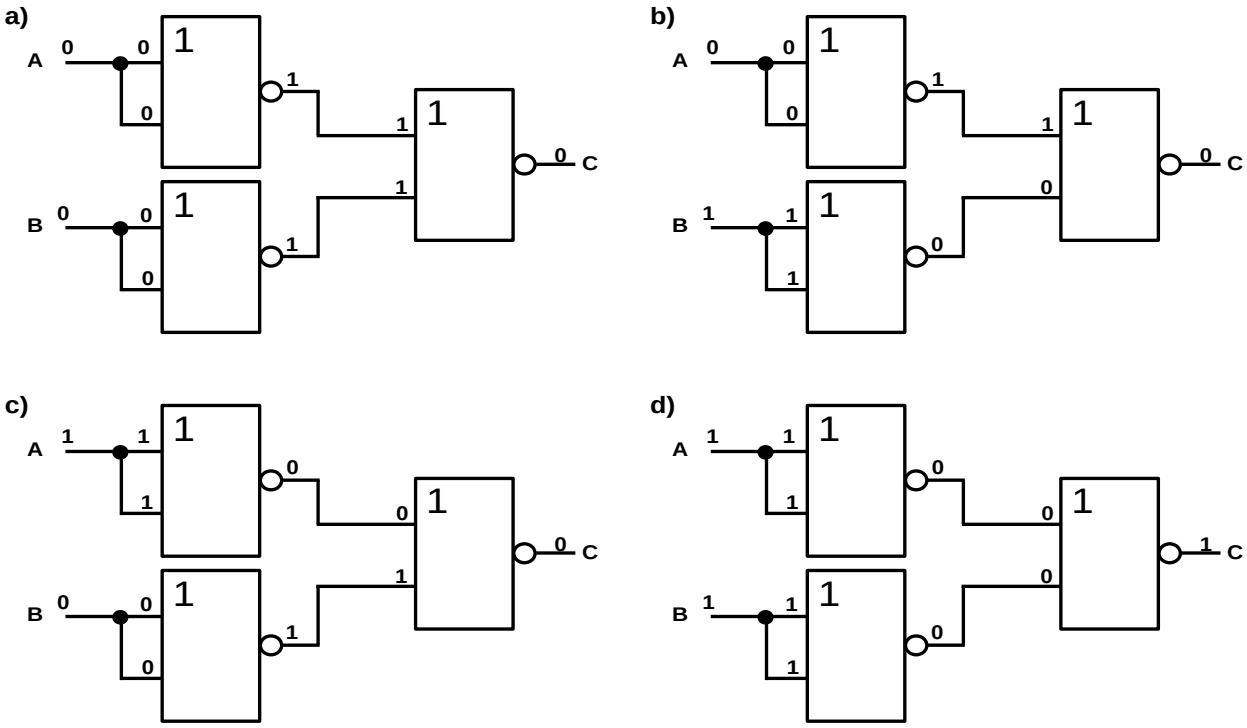


Рисунок 2.4 – Анализ работы схемы при различных входных значениях

Глава 2.2 Исключающее ИЛИ.

Рассмотрим еще один интересный элемент eXclusiveOR (*XOR – исключающее ИЛИ*) рис. 2.5. Проанализируем схему, представленную на рисунке. Будем поочередно подавать все возможные комбинации сигналов на входы **A**, **B** и наблюдать, как изменяется значение на выходе **C**.

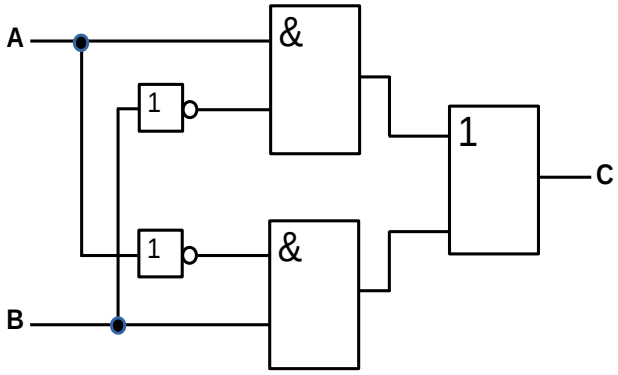


Рисунок 2.5 — Внутреннее устройство и схематичное изображение элемента XOR.

Направим на входы **A** и **B** логический **0**. Сигналы начнут перемещаться по проводам рис. 2.6a. После элемента инверсия, ноль изменится на единицу. На верхнюю конъюнкцию от входа **A** поступит ноль, а от входа **B** единица рис. 2.6b.

Та же ситуация и с нижней конъюнкцией. Конъюнкция **ИСТИННА**, когда оба операнда истинны. В итоге на выходную дизъюнкцию поступит два **00**. Результатом дизъюнкции является **ЛОЖЬ**, когда оба операнда **ЛОЖЬ**. В итоге **C = ЛОЖЬ** рис. 2.6c.

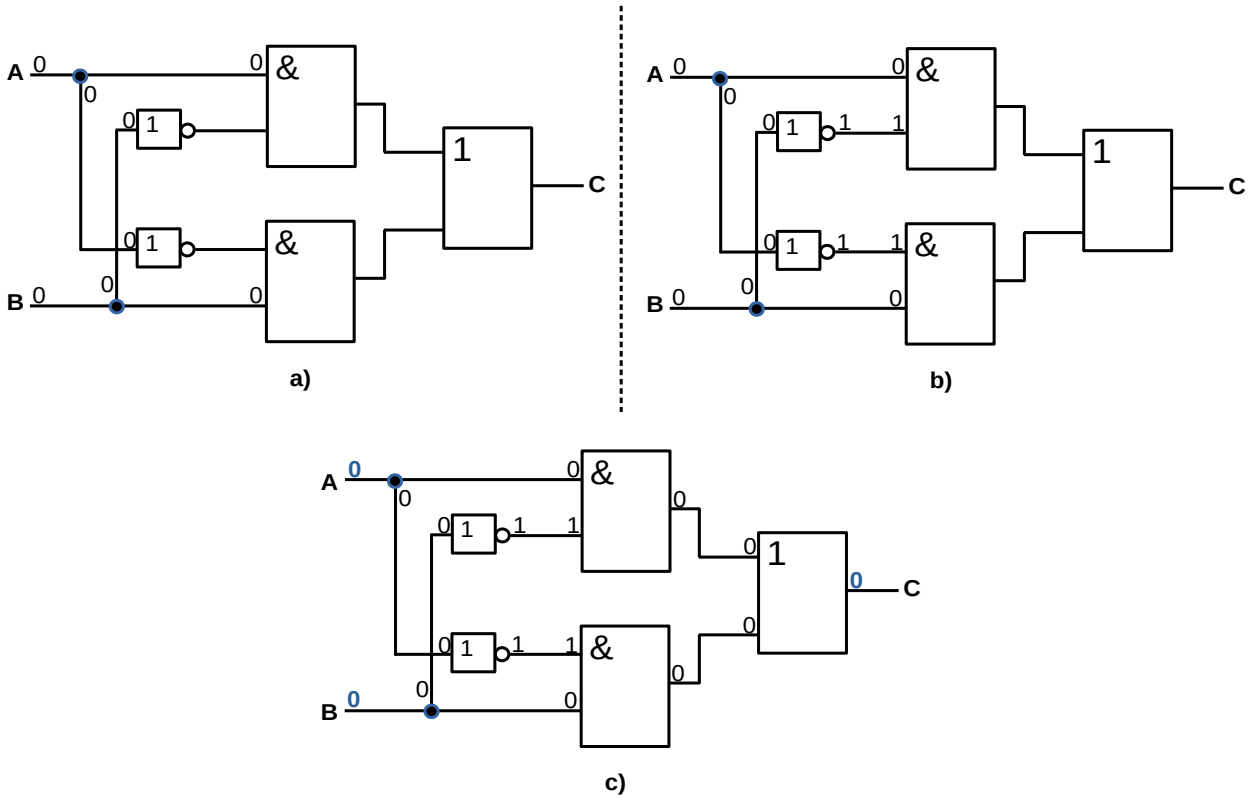


Рисунок 2.6 — Анализ схемы при подаче на входы eXclusiveOR логических нулей

Исследование схемы для оставшихся трех комбинаций входных значений представлено на рис. 2.7. Комбинации операндов обозначены различными цветами. Для последней комбинации **1 1** работа схемы приведет к появлению **0** на выходе **C**. Так же на рисунке представлено схематичное изображение элемента *исключающее ИЛИ* (как ИЛИ только со знаком равно) и таблица истинности для *XOR*. Результат операции *XOR* **ИСТИНА**, когда операнды отличаются друг от друга - «**истина — если разные**».

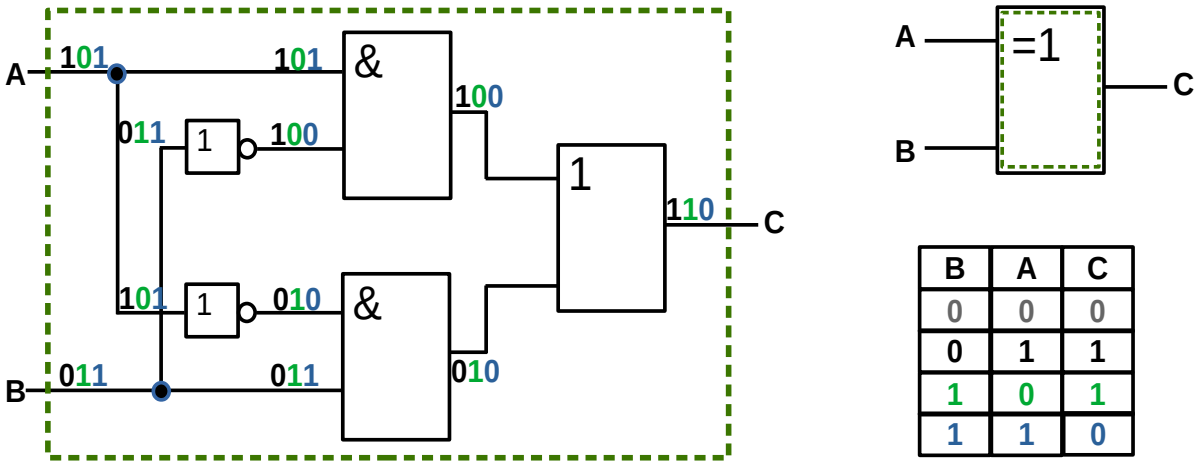
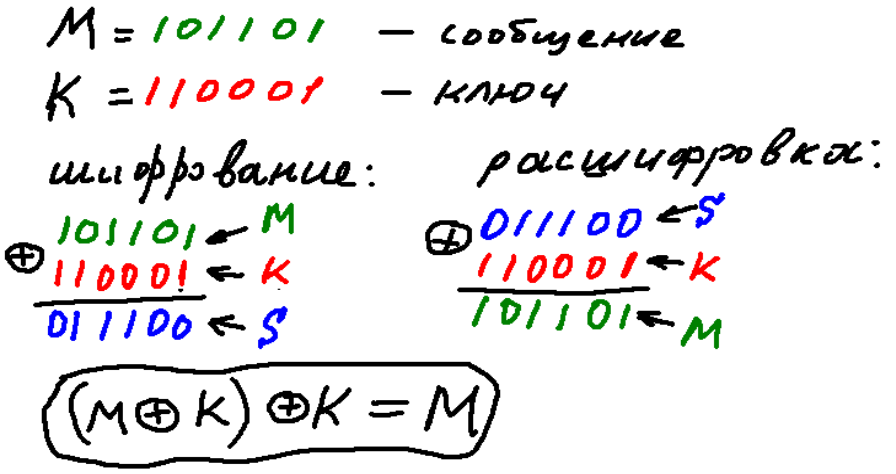


Рисунок 2.7 — Продолжение исследования работы eXclusiveOR, схематичное изображение элемента и таблица истинности.

Аккуратнее, интересный факт. Элемент *XOR* позволяет осуществить простейшее шифрование закрытым ключом. Ключ должен быть закрытым (т.е. не известен третьим лицам), поскольку, расшифровка происходит с использованием того же самого ключа **K**. Формулы шифрования и расшифровки элементарные: **M** *XOR* **K = S и S** *XOR* **K = M**, соответственно.



Пояснение — Шифрование с использованием XOR.

Глава 2.3 Сумматор.

Проектируется схема сумматора – устройства, позволяющего осуществлять арифметическую операцию сложение. Сперва освежаются в памяти сакральные знания о системах счисления. После этого полусумматор и сумматор собираются с легкостью.

2.3.1 Системы счисления

Представим себе далекое прошлое. Ещё дальше.. Ещё….. Живем в пещерах. Ночь, повар спит. Часть племени вернулась из дальнего похода. Как сообщить повару расход на завтрак?

Расход – количество человек, которое будет завтракать.

Будить повара нельзя – будет злой и чего глядишь дубиной замахнется.

Первое, что приходит на ум. положить кучу камней по количеству жителей. Хорошо, составим кучу из семьсот пятьдесят двух камней. Повар проснется в четыре утра и до обеда будет перебирать камушки. И племя останется без завтрака и повару не поздоровится.

Следует сделать так, чтобы и люди были сыты и повар цел. Как первобытным существам, должна прийти идея, заранее договориться с поваром, что камушек – это один человек, палочка это десять камушков, а листочек – десять палочек или сто камней рис. 2.8а.

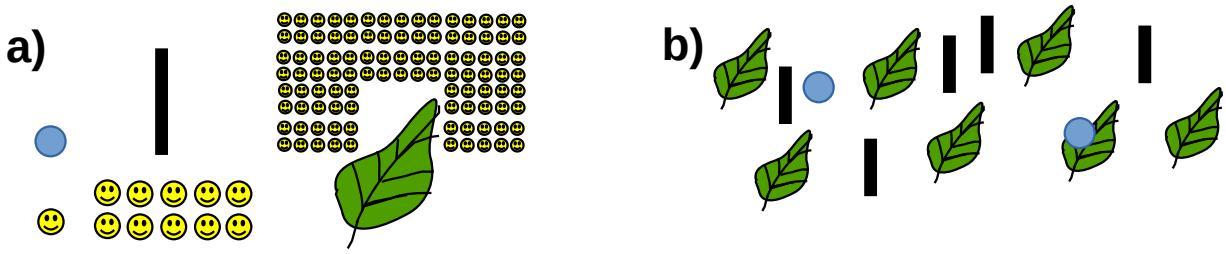


Рисунок 2.8 — Алфавит первобытной системы счисления (a) и семьсот пятьдесят два человека, записанные с использованием первобытной системы счисления (b).

Теперь семьсот пятьдесят два человека будут выглядеть как семь листочков, пять палочек и два камушка рис. 2.8b. Количество человек не изменится, если поменять предметы местами.

Таким образом, были придуманы знаки для обозначения количества человек. Полный набор обозначений, для каких-то объектов, называется **алфавитом**. Правила подсчета количества человек по листочкам, палочкам и камушкам называются **системой счисления**. Знаки алфавита называют **цифрами**. У нас три цифры – камушек, палочка, листочек.

Можно придумать и другие обозначения для цифр, да и другие правила. Поэтому дадим более научные определения.

Система счисления – правила записи чисел и операций над ними.

Алфавит системы счисления– символы (цифры), используемые для записи чисел в данной системе счисления.

Основание системы счисления (**мощность** алфавита) – количество цифр в ее алфавите.

Разберем подробнее первую часть определения *системы счисления*. Какие правила записи чисел бывают? На самом деле широкое распространение получили два правила.

Непозиционные системы счисления

Порядок (позиция) цифры в записи числа не важен. Если переставить цифры местами число не изменится. Наша первобытная система счисления является непозиционной. Другой пример, римская система счисления. Но стоп. Числа IX и XI разные! Мы были слишком грубы, давая определение непозиционным системам счисления. Более корректно говорить, в них **вес цифры не зависит от позиции**. Тогда все хорошо. В обоих числах — девять и одиннадцать — у цифры **I** вес равен единице, а у **X** – десяти. Просто римляне (или кто-то) ввели дополнительное правило, что если меньшая цифра стоит слева, то она вычитается из большей, а если справа – то прибавляется к ней.

Позиционные системы счисления.

Вес цифры зависит от позиции в числе. Давайте уже определимся с понятием **вес цифры**, для этого взвесим число на весах. Вес цифры – та величина, которую она вносит в общую массу числа.

Люди пользуются **позиционными системами счисления**. Числа 25 и 52 разные. В первом числе вес двойки равен двадцати, а во втором двум.

Коль перешли к записи чисел арабскими цифрами, давайте узнаем, почему они выглядят именно так. Не могли же люди просто придумать закорючки. Должна быть какая-то логика в их обозначениях. Это логика очень простая. Цифры хранят количество углов, содержащихся в их записи рис. 2.9. Обратите внимание на ноль. У него нет углов. Хотя на самом деле есть и их бесконечно много.

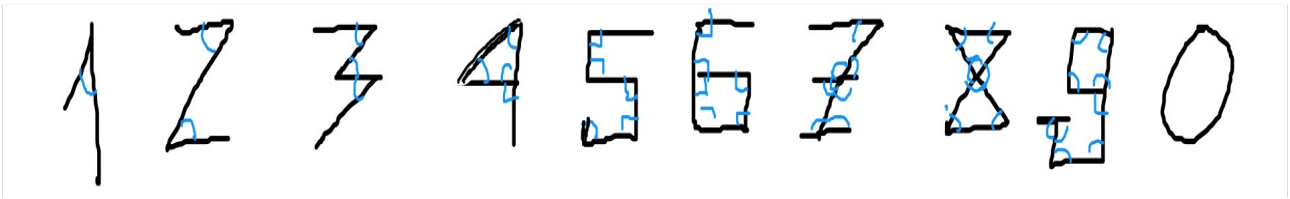


Рисунок 2.9 – Арабские цифры.

В привычном мире используется десять цифр. Таким образом, система счисления имеет основание десять. Иногда говорят, **десятичная система счисления**. Аккуратнее, сейчас знания о числах начнут расширяться. Введем одно хитрое правило, касающееся записи чисел.

Хитрое правило: значение числа в системе счисления с основанием *N* вычисляется суммированием весов всех цифр. Вес рассчитывается как некоторый *коэффициент* b_i, умноженный на числовое значение цифры. *Коэффициент* b_0 самой правой цифры всегда равен единице. *Коэффициент* следующей цифры в записи числа, больше *коэффициента* предыдущей цифры в *N* раз.

Коэффициенты b_i для каждой позиции цифры составляют **базис системы счисления**. Базис для каждой системы счисления является неизменным набором коэффициентов. Это основа. Подставив к базису цифры, можно получить различные числа. Согласно нашему правилу, число 1752_{10}, разворачивается, как показано на рис. 2.10.

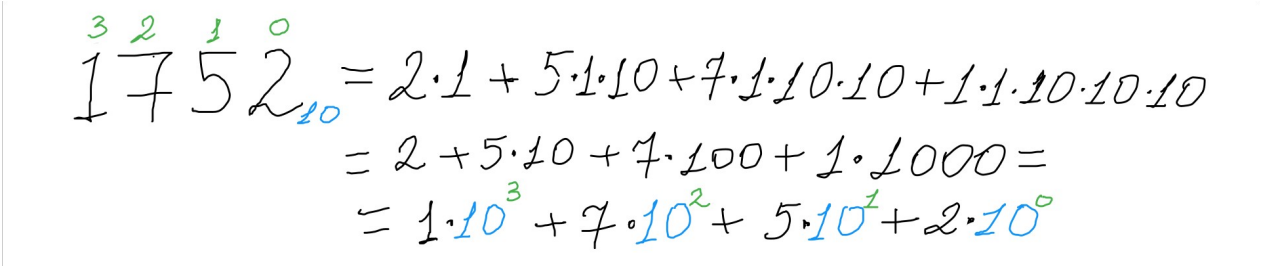


Рисунок 2.10 – Развернутая форма числа.

Нижний индекс **10** – говорит о том, что число записано в десятичной системе счисления. Зеленые числа **0, 1, 2, 3** над цифрами, указывают на позицию цифры в числе, которая называются **разряд**. Разряды нумеруются справа налево, начиная с нуля. Обратите внимание, что в развернутой форме числа, каждая цифра умножается на основание системы счисления, которое возводится в степень, численно равную разряду цифры. Для удобства основание на рис. 2.10 изображено **голубым** цветом, а разряды – **зеленым**. Голубым и зеленым цветом выделен базис десятичной системы счисления, точнее небольшая его часть.

2.3.2 Как образуются числа

Посмотрим, как образуются числа в десятичной системе счисления. Алфавит состоит из десяти цифр: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**.

Набор | | | | | | | | записывается одной цифрой 9.

Добавим ещё одну палочку | | | | | | | | |. Получилось число равное основанию системы счисления – число десять. Для записи больших чисел придумали понятие разряда. Правый (нулевой) разряд указывает сколько в числе присутствует единиц. Следующий (первый) разряд, сколько раз присутствует *N* палочек, где *N* – основание системы счисления. Число 10_{10} означает **1** десяток и **0** единиц.

Таким образом, в восьмеричной системе счисления (*N* = *8*) ряд чисел будет выглядеть следующим образом, **0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21… 77, 100, 101, …** Число 10_8 означает **1** восьмерка и **0** единиц. Запись 123_8 говорит о том, что в числе встречается **1** раз шестьдесят четыре, **2** раза восемь и **3** единицы, всего восемьдесят три палочки.

В двоичной системе счисления последовательность подряд идущих чисел выглядит достаточно просто **0, 1, 10, 11, 100, 101, 110, 111, 1000...**

2.3.3 Сложение

Вспомним как осуществляется сложение в привычной десятичной системе счисления рис. 2.11.

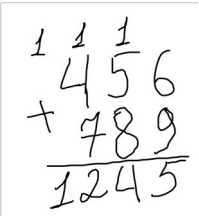


Рисунок 2.11 – Сложение столбиком в десятичной системе счисления.

Сперва складываются младшие разряды (цифры справа). Если в результате сложения происходит переполнение разряда (получается число больше или равное основанию *N = 10*), то отнимаем от него основание *N = 10* и записываем количество единиц. А оставшийся десяток (*N=10*) переносим в следующий разряд. Обратите внимание, что второй разряд хранит количество десятков в числе, поэтому добавляем ко второму разряду *1*, имеется ввиду один десяток.

Сложение в системах счисления с другим основанием происходит аналогичным образом.

Рассмотрим **сложение в двоичной системе счисления** (*N = 2*) рис. 2.12.

Рисунок 2.12 – Сложение в двоичной системе счисления

Складываем единицы справа (нулевой разряд). Получаем две единицы – в двоичной системе счисления цифра *2* отсутствует, произошло переполнение разряда. Отнимаем от двух основание (*N=2*), получается **0**, записываем ноль единиц, и переносим одну двойку, которую только что отняли, в следующий разряд, который хранит количество двоек, присутствующих в числе.

Складываем цифры во втором разряде с учетом переноса. Получается три двойки (**1+1+1**). Опять переполнение. Отнимаем от трех двоек две двойки (*N = 2*), получаем одну двойку, которую и записываем в виде **1** в перовом разряде. Отнятые две двойки, т.е. одну четверку переносим в следующий разряд, который хранит количество четверок в числе.

В каждом операнде во втором разряде ноль четверок, поэтому просто записываем одну четверку, перенесенную из младшего разряда.

Самый левый разряд хранит количество восьмерок. При сложении получается две восьмерки, которые нельзя записать одной цифрой, пишем ноль, а одну шестнадцатерку (т.е. две восьмерки) переносим в следующий разряд.

Таким образом, получается число $\mathbf{10110_2 = 16 + 4 + 2 = 22_{10}}$**.** Нижний индекс указывает на основание системы счисления, в которой записано число.

2.3.4 Полусумматор

Составим устройство, способное складывать два одноразрядных двоичных числа.

Сложение в двоичной системе осуществляется легко, не смотря на всю громоздкость описания, представленного выше. Его можно даже описать словами.

- Если нет единиц, то при сложении будет ноль.
- Если одна единица, то результатом будет единица.
- Если две единицы, то результат **10**.

0 + 0 = 0 0 + 1 = 1 1 + 0 = 1 1 + 1 = 10

Перепишем в виде таблицы: $\mathbf{A_0}$ и $\mathbf{B_0}$ – исходные слагаемые, $\mathbf{C_1C_0}$ – результат, т.е. $\mathbf{A_0 + B_0 = C_1C_0}$.

A_0	B_0	C_1	C_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Напоминает какие-то таблицы?

Столбцы $\mathbf{A_0B_0C_0}$ – представляют собой таблицу истинности для *XOR* (исключающее *ИЛИ*). Столбцы $\mathbf{A_0B_0C_1}$ – таблицу истинности для *AND* (конъюнкция).

Логические элементы *XOR* и *AND* были собраны несколько страниц тому назад. Остается только нарисовать схему полусумматора на рис. 2.13.

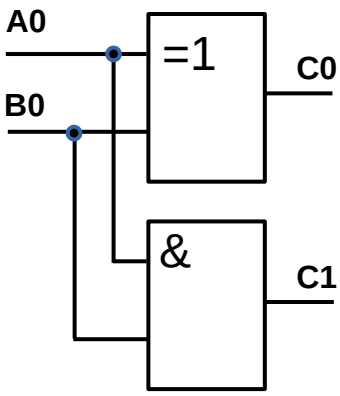


Рисунок 2.13 – Схема полусумматора.

Итак, полусумматор имеет два входа и два выхода. На вход подаются складываемые двоичные цифры A_0 и B_0, на выходе появляется результат сложения C_1C_0.

Полусумматор бессилен, если появляется перенос из младшего разряда. Зато отлично подходит для сложения самого младшего разряда двоичных чисел — цифр числа, стоящих справа.

2.3.5 Полный сумматор

Как же из полусумматора сделать сумматор? Правильно мыслите… Если взять два полусумматора, то возникнет сумматор! Только остается их соединить, а для этого необходимо четко представлять, что с чем складывать.

Пусть первый сумматор, как и прежде складывает две цифры **Ai** и **Bi** из некоторого *i* разряда чисел. К получившемуся значению необходимо прибавить перенос **Cin**, который мог возникнуть из предыдущего разряда рис. 2.14.

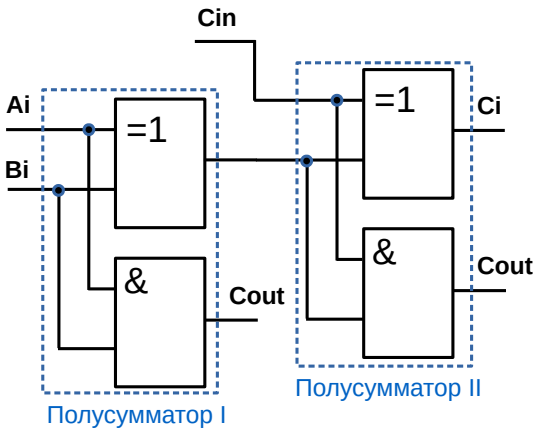


Рисунок 2.14 – Начало построения сумматора.

Полусумматоры соединить удалось, но теперь имеется два выхода **Cout**, а у сумматора должен быть один выход переноса в следующий разряд.

Перенос на полусумматоре I может случиться в результате сложения двух единиц (**Ai** = **1**, **Bi** = **1**). В данном случае, на выходе полусумматора I появится ноль, следовательно в результате полусумматора II перенос будет отсутствовать. Единица на выходе переноса **Cout** полусумматора II, возникает только в случае, если от полусумматора I пришла **1**, но тогда на переносе **Cout** полусумматора I может быть только **0**.

Таким образом, **1** не может появиться на выходах **Cout** обоих полусумматоров одновременно. Либо на одном, либо на другом, либо вообще не появится. Единица *ИЛИ* на первом *ИЛИ* на втором выводе — это *дизъюнкция*.

Соединим выводы **Cout** элементом *ИЛИ*. На рис. 2.15a изображена схема полного сумматора, а на рис. 2.15b его условное обозначение на схемах (вспоминаем концепцию черного ящика, он показан на рисунке зеленым пунктиром).

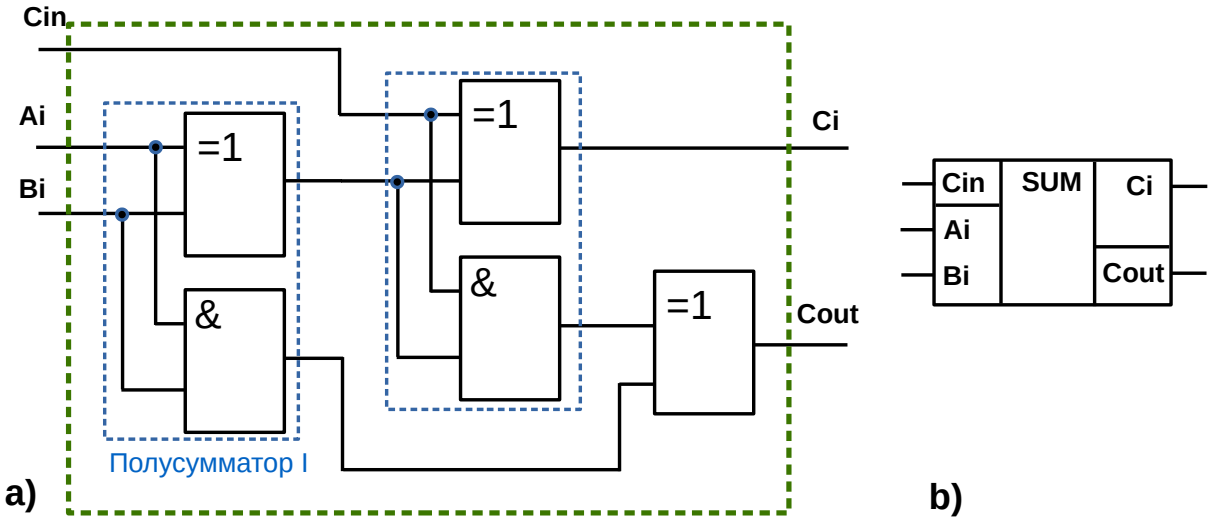


Рисунок 2.15 – Схема полного сумматора (a) и его условное обозначение (b).

Устройство на рис. 2.15 позволяет полноценно сложить любой разряд двух чисел, с учетом переноса из младшего разряда и выводом переноса в следующий разряд. Как видим, сумматор, осуществляющий *арифметические операции*, полностью собран из логических элементов, и *функционирует согласно алгебре логики*.

Осталось собрать устройство, складывающее числа целиком. Возьмем горсть сумматоров по количеству разрядов в складываемых числах. Выход переноса **Cout** сумматора, складывающего младший разряд, соединим с входом переноса **Cin** сумматора, складывающего более старший разряд. Схема 4-х разрядного сумматора, показана на рис. 2.16а. $A_3A_2A_1A_0 + B_3B_2B_1B_0 = C_4C_3C_2C_1C_0$. Обратите внимание, что выход переноса последнего сумматора есть старший разряд **C4** результирующего числа.

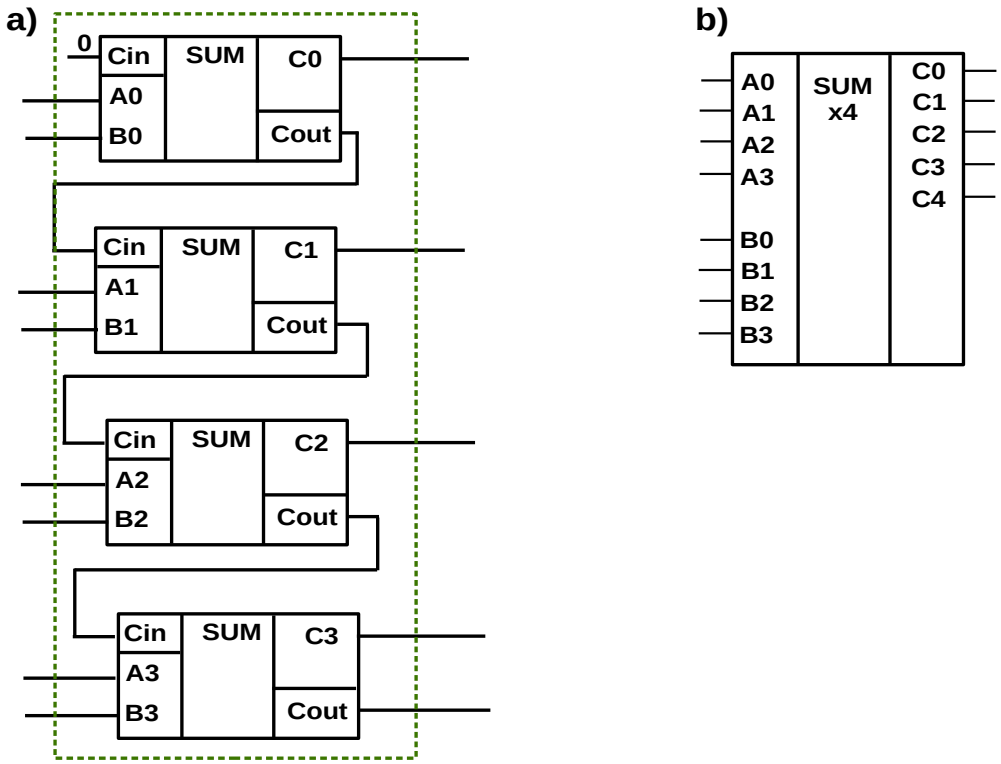


Рисунок 2.16 – Схема четырех разрядного сумматора (a) и условное обозначение (b).

Как ни странно, на данный момент мы рассмотрели всё, что умеет делать микропроцессор. Несколько логических операций и сложение. Остальные операции реализуются через сложение, хитрые сдвиги и магию математики.

Глава 2.4 Декодер

Все рассмотренные ранее схемы относятся к элементам **комбинационной логики**. Состояние выходов устройств *комбинационной логики* однозначно определяются сигналами на входах. Определенной *комбинации* значений на входе, соответствует определенная *комбинация* значений на выходе. Поэтому устройства и называют комбинационным. Рассмотрим ещё одно устройство комбинационной логики.

Микропроцессор, помимо сложения (*сумматор*), будет уметь производить логические операции (*дизъюнкцию*, *конъюнкцию*, *XOR*). Какую конкретно операцию выполнит микропроцессор определяется командой программы. Команда представляет собой двоичный код.

Таким образом, необходимо спроектировать *устройство*, на вход которого будет поступать код команды. В зависимости от кода команды у *устройства* будет активироваться один из выходов. Выходы *устройства* будут «выбирать» результат какой из 4-х операций, появится на выходе микропроцессора. Данное *устройство* — называется **дешифратор** (*англ. Decoder*).

Разберемся сколько входов и выходов имеется у дешифратора.

Количество операций, между которыми микропроцессору предстоит выбирать, равно четырем. Значит следует спроектировать дешифратор с четырьмя выходами. Далее, необходимо придумать четыре различных двоичных кода, для активации каждого из выходов. Активация выхода заключается в следующем — на активном выходе возникает **1**, на остальных выходах **0**:

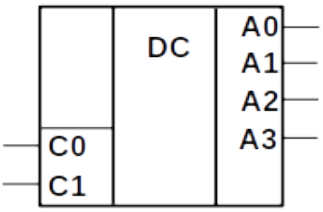
Код **00** – активируется первый выход (**1000**)

Код **01** – активируется второй выход (**0100**)

Код **10** – активируется третий выход (**0010**)

Код **11** – активируется четвертый выход (**0001**)

Для управления дешифратором достаточно двух ножек на входе. Черный ящик и таблица истинности дешифратора показаны на рис. 2.17.



C0	C0	A3	A2	A1	A0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Рисунок 2.17 — Черный ящик и таблица истинности дешифратора.

Данная схема называется дешифратор *2* на *4*. Два входа управляют активацией одного из четырех выходов. Внимание вопрос! Что находится в черном ящике?

Обратите взор на выходные значения, и вспомните смысл основных элементов алгебры логики (конъюнкции, дизъюнкции и инверсии). В каждом из выходных значений дешифратора **1** появляется только в одном случае. Какой логический элемент дает **1** в единственном случае. Верно! Конъюнкция. У нее на выходе **1** в том случае, когда на обоих входах **1**. Просто расположим на всех выходах конъюнкцию рис. 2.18.

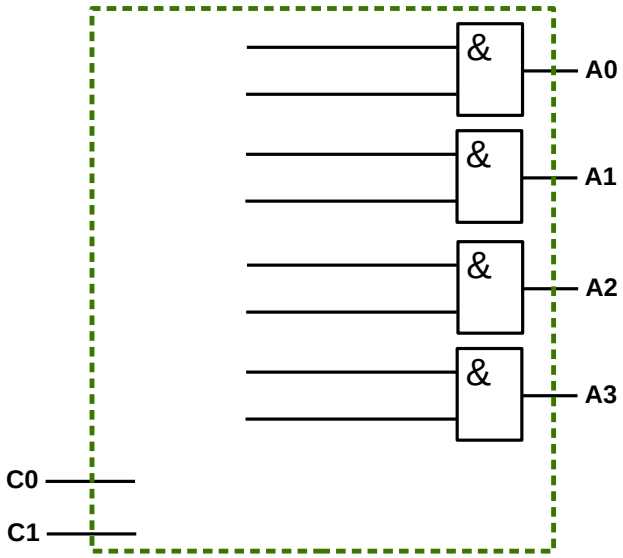


Рисунок 2.18 — Конъюнкции на выходах дешифратора.

Теперь необходимо получить **1** для первого выхода дешифратора **A0**, в то время как на обоих его входах (**C0** = **C1** = **0**) находятся **0** (см. таблицу истинности дешифратора). Но конъюнкция даёт **1**, только при поступлении на оба входа **1**. Что же делать? Правильно! Инвертировать оба входа рис. 2.19! Сейчас, как никогда важно вспомнить, что соединение линий (проводов) обозначается точкой (маленьким черным кружочком).

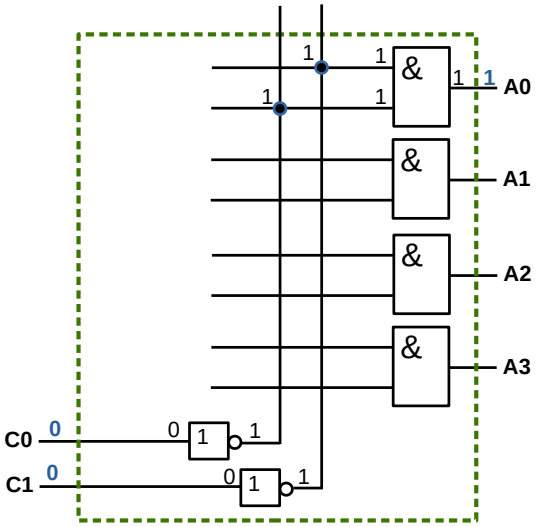


Рисунок 2.19 — Подключение первого выхода дешифратора.

Обратите внимание, на выходе значение **1** появится в единственном случае, когда на входах дешифратора **00**. Если хотя бы на один вход подать **1**, то инверсия даст **0**, а конъюнкция с **0** всегда **0**.

Аналогично, чтобы получить **1** для следующего выхода дешифратора, отправим на вторую конъюнкцию две **1**. Для этого подадим сигнал **C0** без инверсии (для второй команды он равен **1**), а сигнал со входа **C1** перед подачей на вторую конъюнкцию подвергнем отрицанию рис. 2.20 .

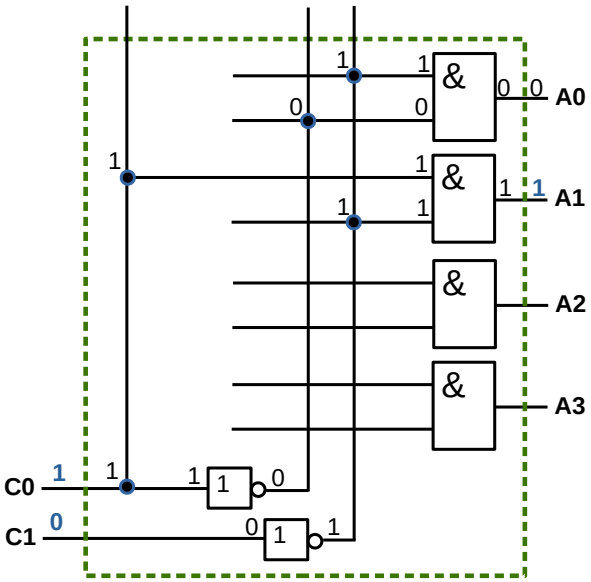


Рисунок 2.20 — Подключение второго выхода дешифратора.

Соединим оставшиеся конъюнкции и получим готовый дешифратор рис. 2.21.

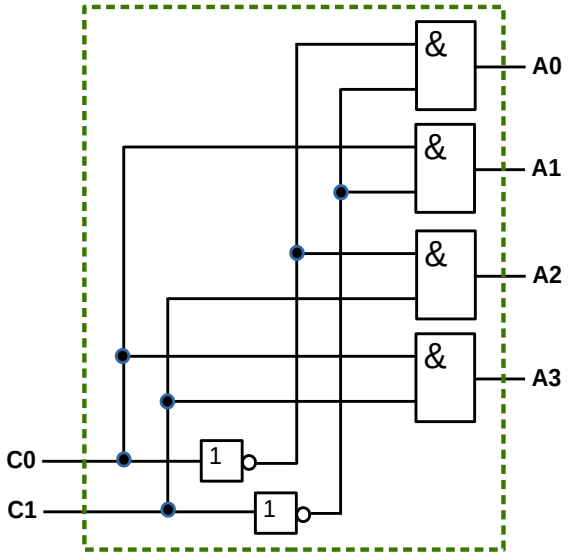


Рисунок 2.21 — Схема дешифратора.

Принцип построения дешифратора запомнить очень легко. От входов вдоль всей схемы отводятся вертикальные линии без инверсии и с инверсией. Каждая выходная конъюнкция соединяется с необходимыми линиями от входа, таким образом, чтобы конъюнкция давала единицу для соответствующего ей кода на входе в соответствии с таблицей истинности. Такое вот сложное предложение получилось.

Итак, созданный дешифратор имеет два входа и четыре выхода. Какой из четырех выходов активируется, зависит от входных значений.

Запомним и ненадолго забудем о комбинационных элементах и перейдем к *последовательностной* логике. Прочитайте это длинное слово ещё раз. Оно написано без ошибок и опечаток.

Глава 2.5 Секвенциальная логика и память

В последовательностной логике, в отличие от комбинационной, состояния на выходах не определяются состояниями на входах однозначно. Они зависят от текущего состояния на входах и *предыдущих* состояний системы. Изменение состояний происходит во времени, т.е. значения на выходах зависят от состояния схемы в предыдущие моменты времени. А это и есть **память**. Схема памяти хранит значения, поданные на ее входы, в предыдущие моменты времени. **Секвенциальная логика** - не менее сложновыговариваемое слово для названия последовательностных элементов. На самом деле это английское слово *sequence*, которое переводится как *последовательность*.

2.5.1 RS – триггер

Проанализируем работу схемы, изображенной на рис. 2.22.

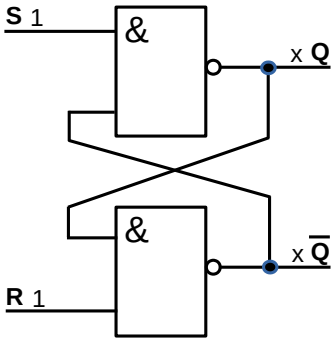


Рисунок 2.22 — Схема RS-триггера.

Принципиальное отличие данной схемы, от рассмотренных ранее, заключается в том, что выходы соединены со входами.

Примем за начальное состояние следующие значения на выводах схемы (рис. 2.22): $S=1$, $R=1$ $Q=X$, $\overline{Q}=X$, где X – обозначает любое состояние (**0/1**). Проанализируем работу схемы при подаче различных сигналов на S и R . Для этого вспомним, что выход элемента *И-НЕ* равен **0** в единственном случае, когда на вход поданы две **11**.

Переключим вход S в **0** рис. 2.23a. Верхний *И-НЕ* перейдет в состояние $Q=1$, т.к. на вход было подано $S=0$, $\bar{Q}=X$. Нижний И-НЕ перейдет в состояние $\overline{Q}=0$, поскольку на вход поступят две единицы $Q=1$, $R=1$. Получим состояние, изображенное на рис. 2.23a.

Вернем вход S в **1**. Состояние на выходах не изменится: $Q=1$, $\overline{Q}=0$ рис. 2.23b. Действительно, на входах верхнего *И-НЕ* будет $S=1$, $\overline{Q}=0$, а на входах нижнего $Q=1$, $R=1$. Вход S изменился, а значения на выходах остались прежними. Значения на выходе уже не определяются значениями на входе. Они зависят от предыдущего состояния схемы. Независимо от того, что хранилось на выводах $Q=X$ $\overline{Q}=X$, при поступлении $S=0$ схема переходит в состояние $Q=1$, $\overline{Q}=0$. Данные выходные значения не изменяются (*сохраняются*), при возвращении S в исходное состояние $S=1$.

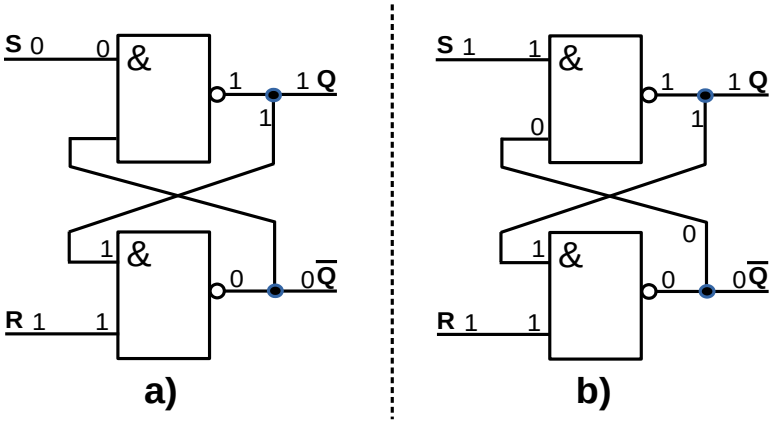


Рисунок 2.23 — Запись и хранение единицы: a — сброс выхода **Q** *в единицу при подаче на* **S** *нуля; b — хранение единицы на выходе* **Q** *при возвращении входа* **S** *в единицу.*

Рассмотренная схема называется *RS*-триггер и позволяет хранить один бит информации.

Вход S – используется для сохранения **1** (от англ. *Set* – установить).

Вход R – предназначен для сохранения **0** (от англ. *Reset* – сброс).

Выводы Q и $\overline{Q}$ называются неинвертирующий и инвертирующий выход, соответственно. Схема спроектирована так, что значения на данных выводах всегда противоположны.

Итак, существует три режима работы *RS*–триггера:

- $S=1 \quad R=1$ - **режим хранения**, значения на выводах Q и $\overline{Q}$ не изменяются.
- $S=0 \quad R=1$ - **режим записи единицы**, при этом $Q=1 \quad \overline{Q}=0$
- $S=1 \quad R=0$ - **режим записи нуля**, при этом $Q=0 \quad \overline{Q}=1$
- $S=0 \quad R=0$ - режим непредсказуемой работы

И окончательно! Чтобы сохранить единицу, необходимо «установить» триггер, для этого достаточно подать кратковременный нулевой импульс на вход **Set**. Чтобы сохранить ноль, необходимо сбросить триггер, для этого достаточно подать кратковременный нулевой импульс на вход **Reset**.

Анализ работы схемы при подаче нулевого импульса на вход R читателю предлагается произвести самостоятельно.

2.5.2 D-триггер

RS-триггер — простейшее устройство для хранения одного бита информации со сложной логикой работы. Чтобы сохранить единицу необходимо подать импульс на один вход, а для сохранения нуля на другой. Спрашивается, кто собственно будет сидеть и перенаправлять эти импульсы? Оказывается, если присоединить к *RS*-триггеру хитрым образом, как показано на рис. 2.24, два элемента *И-НЕ*, то получится достаточно удобная в работе схема. Она называется *D*-триггер.

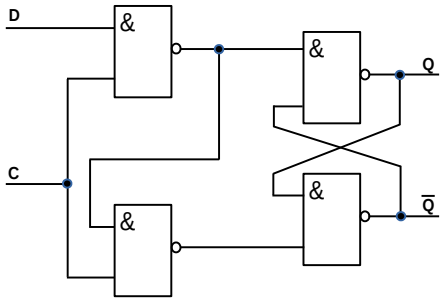


Рисунок 2.24 — Устройство D-триггера.

У *D*-триггера два входа и два выхода. Вход **D** – является информационным, вход **C** – управляющий. На вход **D** – подается бит, (**0** или **1**), который планируется сохранить, т.е. *информация*.

При **C = 0**, *D*-триггер *хранит* информацию. Ноль, поступивший от **C** на верхний и нижний *И-НЕ*, приведет к появлению на входах *RS*-триггера единиц. А в этом случае, *RS*-триггер *хранит* информацию рис. 2.25.

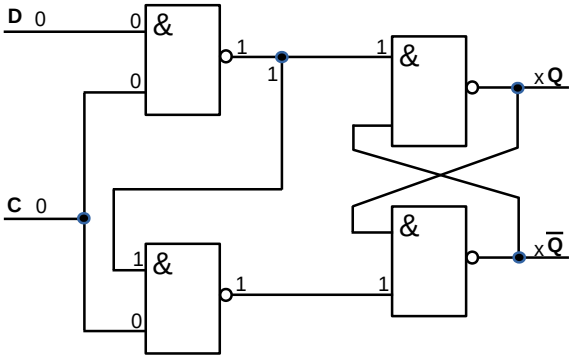


Рисунок 2.25 — D-триггер в режиме хранения.

При **C = 1**, *D*-триггер работает в режиме *прозрачности*. Если в этот момент **D = 0**, то **S = 1**, а **R = 0** и триггер сбрасывается в **0** рис. 2.26a. Если же **D = 1**, то **S = 0**, **R = 1** и триггер устанавливается в **1** рис. 2.26b. Таким образом, $Q=D$, информация со входа поступает на выход, это и есть *прозрачность*.

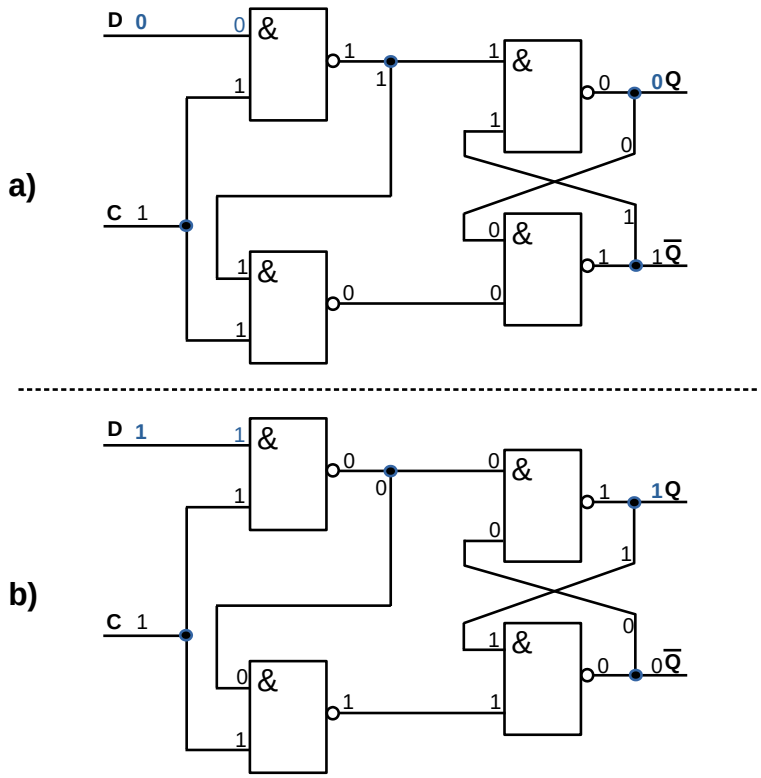
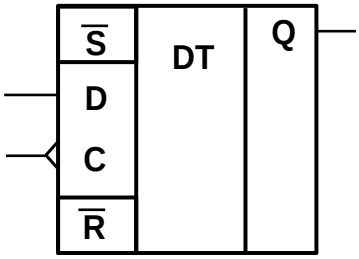


Рисунок 2.26 — D-триггер в режиме прозрачности: a — доказательство прозрачности для нуля; b — доказательство прозрачности для единицы.

Условное обозначение и таблица истинности *D*-триггера представлены на рис. 2.27



C	D	Q	Режим
0	0	x	хранение
0	1	x	
1	0	0	прозрачность
1	1	1	

Рисунок 2.27 — Условное обозначение и таблица истинности D-триггера.

Итак, при подаче на **C** единицы триггер становится прозрачным, информация с входа **D** беспрепятственно проходит на выход **Q**. Другими словами, при спаде напряжения с **1** до **0**, происходит *защелкивание* (запись) информации, хранящейся в данный момент на выходе **Q**. *D*-триггер часто называют «*защелкой*».

На рис. 2.28 изображены, так называемые, **временные диаграммы**. Они показывают, как изменяются напряжения на входах и выходах схемы с течением времени в процессе работы схемы (в данном случае, записи).

Изначально **C = 0** — триггер хранит информацию (1), например, **Q = 1**. Далее, на вход **D** подается бит, который хотим записать (2), например, **D = 0**. После этого, на управляющий выход подается высокий уровень **C = 1** (возникает фронт импульса — изменение с нуля до единицы) и триггер становится прозрачным (**3**). Информация со входа **D** поступает на выход **C** (**4**). В момент спада (5) управляющего импульса, бит защелкивается в триггере. Дальнейшее изменение сигнала **D** (6) не приводит к изменениям на выходе **Q** (6). Осуществляется **процесс хранения информации**.

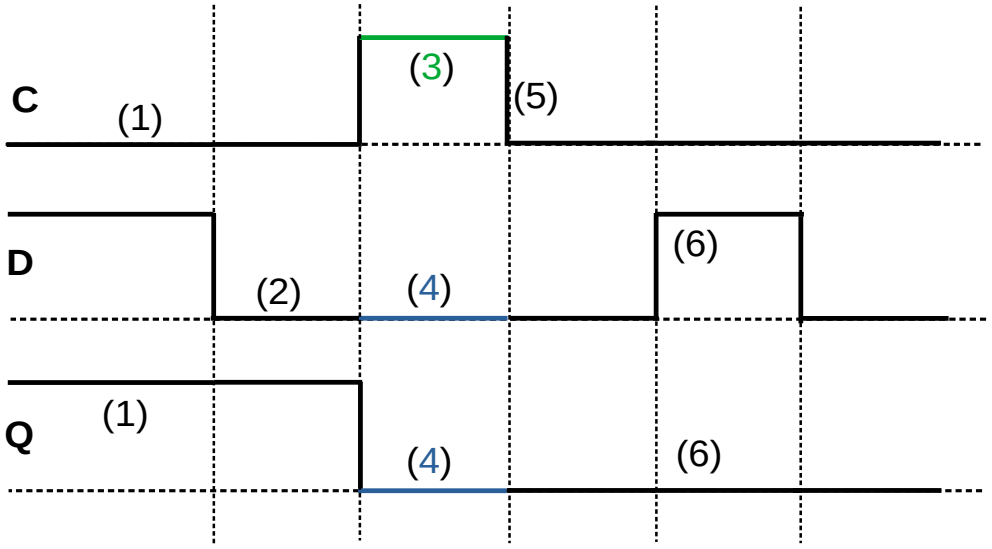


Рисунок 2.28 — Временные диаграммы, отображающие процессы записи и хранения.

Обратите внимание как выглядят фронт импульса (*front*) — переход из низкого в высокое состояние — и спад импульса (*fall*) — переход из высокого в низкое состояние рис. 2.29.

Рисунок 2.29 — Пояснение к понятию фронт и спад.

Далее будем предполагать, что триггер *срабатывает по фронту*. Т.е. значения сохраняются в момент изменения сигнала **C** с **0** до **1**. Срабатывание входа по фронту будем обозначать треугольником, направленным вправо, рис. 2.29. В приведенном выше примере, триггер срабатывал по спаду.

2.5.3 Параллельный регистр

Используем несколько *D*-триггеров для хранения нескольких бит информации. Для этого объединим входы **C** между собой. Чтобы можно было быстро сбросить все биты в ноль, объединим все входы **R.** Выведем все необходимые ножки из черного ящика рис. 2.30.

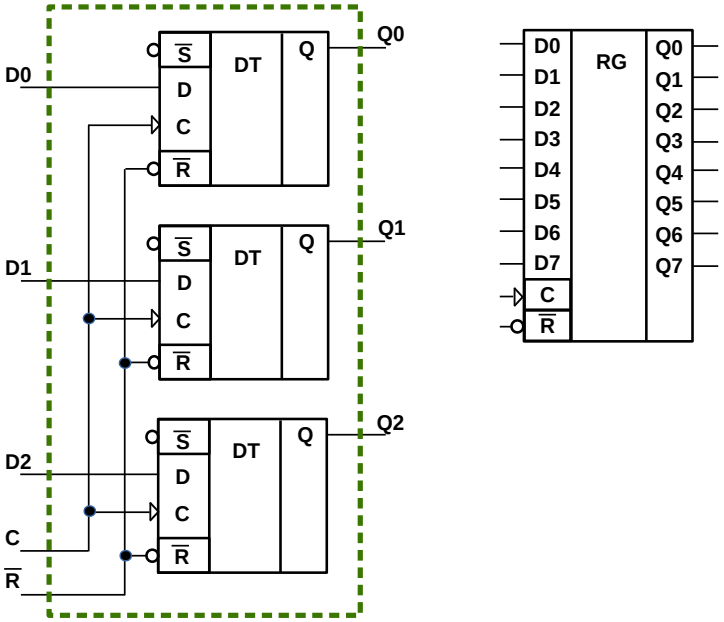


Рисунок 2.30 — Устройство и логическая схема параллельного регистра.

Данная схема называется параллельным регистром. Обратите внимание на треугольник на входе **C**, направленный вправо. Он означает, что регистр срабатывает (записывает) по фронту импульса. Теперь разберемся с кружочком у **R** на схеме.

Кружочек, насколько известно, означает инверсию. Получается, что вход **R** – инвертированный. Что это значит? В логике принято за активный сигнал считать **1**. Но *RS*-триггеры, из которых в конечном счете состоит регистр, сбрасываются при подаче на **R** нуля **0**. Активным, в данном случае, является нулевой уровень, он приводит к действиям. Кружочек говорит о том, что вход **R** начинает что-то делать (становится активным), при подаче на него **0**, а не **1**. Чтобы сбросить все разряды регистра в **0**, на **R** необходимо подать **0**.

Чтобы записать информацию в параллельный регистр необходимо:

1. выставить на его информационные входы **D7..D0** значения (биты), которые хотим сохранить,
2. подать на управляющий вход **C** короткий импульс с высоким уровнем **1,**
3. на входе **R** постоянно должен поддерживаться высокий уровень **1.**

Чтобы стереть информации, необходимо установить **R** в **0**.

Теперь все готово, для сборки арифметико-логического устройства.

Раздел 3. Арифметико-логическое устройство

Арифметико-логическое устройство (*ALU*) — то, что называют сердце микропроцессора. Оно выполняет арифметические и логические операции. Эти операции позволяют обрабатывать данные (информацию, в компьютерной форме в виде **`0`** и **`1`**). Обработка данных — основное назначение микропроцессоров.

Далее не будем придерживаться какой-то строгой терминологии или схемотехники. Рассмотренные устройства могут отличаться от устройств с таким же названием, описанных в других книгах, и даже выполнять немного другие функции. Если устройство не противоречит логике, то логично, оно будет работать. А как принято называть, это уже дело науки.

ALU получится как на рис. 3.1. Разберем подробно назначение всех выводов *ALU*.

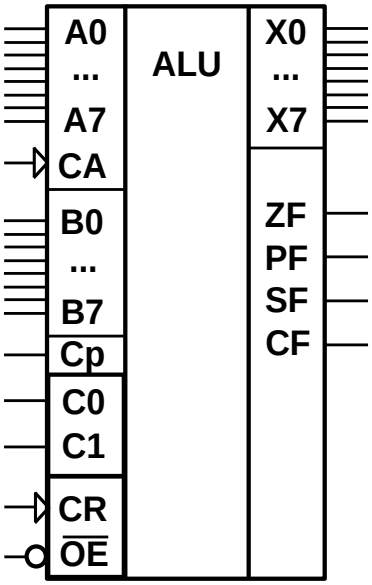


Рисунок 3.1 — Схематичное изображение ALU.

Входы

A7..A0 и **B7..B0** — операнды, двоичные числа над которыми будет совершена операция.

Cp — вход переноса единицы из младшего разряда. Необходим, при сложении чисел, занимающих более 8 двоичных разрядов (>255).

C1..C0 (*от англ. Command*) — код операции.

OE (*OutputEnable*) служит для подключения выходов *ALU* к шине данных. О шинах чуть позже.

CA, CR — тактовые сигналы, инициирующие запись во внутренние регистры *ALU.*

Выходы

X7..X0 — результат выполнения *ALU* очередной операции над операндами.

ZF, PF, CF, SF — флаги, изменяются после выполнения операции и указывают на дополнительные результаты операции.

Флаг **CF** (от англ. *Carry Flag*) устанавливается в **1**, если при выполнении операции возникло переполнение разрядов (произошел перенос в разряд **X8**, которого в представленном *ALU* не существует). Флаги очень важны при написании программы. Например, осуществив вычитание чисел **A + (-B)**, и, проверив значение флагов **ZF** и **SF**, можно сделать вывод какое из чисел больше.

Для простоты разработаем 2-х разрядное *ALU*. Повышение разрядности не приводит к улучшению понимания принципа работы схемы. Конструировать начнем с середины.

Глава 3.1 Середина

Середина *ALU* включает хорошо знакомые *сумматор, конъюнкцию, XOR* и *дешифратор* рис. 3.2. Да, именно, *XOR*, а не инверсия. *XOR* более интересная операция, чем *отрицание.* Тем более, программным способом не составит труда на элементе *XOR* выполнить *инверсию*.

ALU выполняет сразу все четыре операции одновременно. Но постойте! *ALU* имеет только один выход **`X0..X1`**! Вот тут и пригодился дешифратор, который, в соответствии с поступившей на него командой, позволяет направить результат нужной операции на выход. Процесс направления данных с одного из многих входов на единственный выход называется **мультиплексированием**.

Давайте осуществим это *мультиплексирование*. Поступим следующим образом. Уничтожим результаты всех операций, кроме того, которому суждено выбраться от *ALU* на выход. Судьей выступает активный выход дешифратора, на котором располагается **`1`**. На всех остальных выходах дешифратора **`0`**.

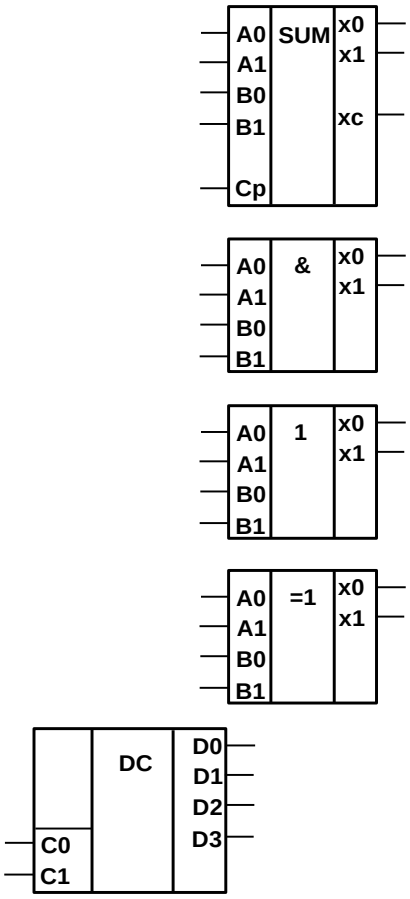


Рисунок 3.2 — Середина ALU — сумматор, дизъюнкция, конъюнкция, XOR и дешифратор

Направим выход каждой операции и соответствующий выход дешифратора на вход конъюнкции. Если выход дешифратора будет активным, то конъюнкция с **`1`** не изменит результата, в противном случае, умножение на **`0`** даст **`0`**.

Главное не запутаться. Разрядов много, целых два.

Договоримся, что выходы дешифратора выполняют следующие функции:

`D0` — пропускает сумматор,

`D1` — пропускает конъюнкцию,

`D2` — пропускает дизъюнкцию,

`D3` — пропускает *XOR*.

Аккуратно осуществим конъюнкцию каждого разряда на выходе сумматора с выходом дешифратора **D0** рис. 3.3, активирующим выходы сумматора.

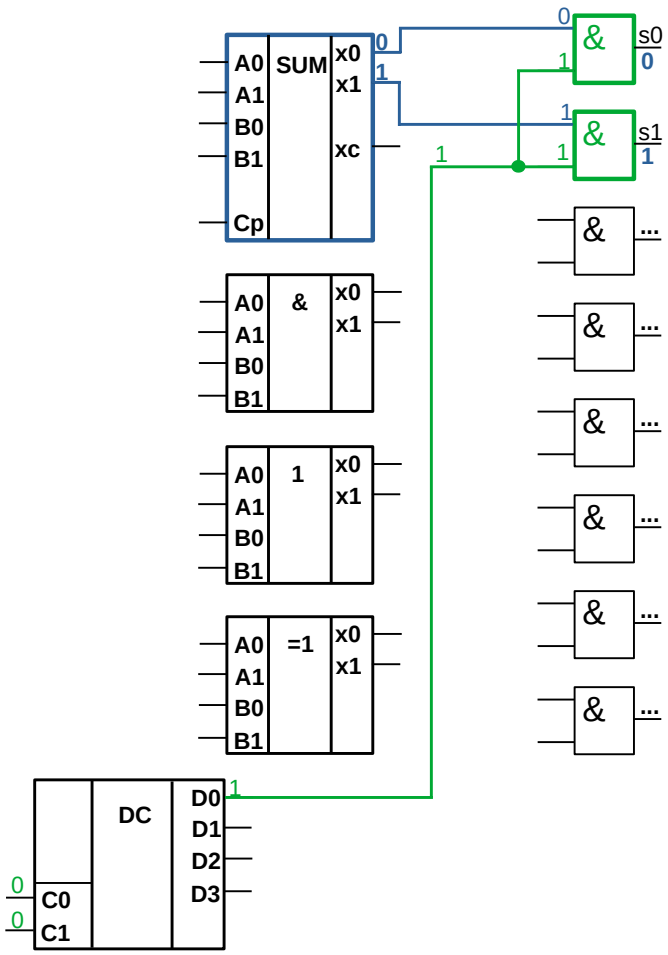


Рисунок 3.3 — Конъюнкция выходов **x0..x1** *сумматора с выходом* **D0** *дешифратора.*

При подаче на вход **C1..C0** дешифратора кода **00**, активируется выход **D0** и на выводах **s1..s0** появляется результат сумматора **x1..x0**. При поступлении кода, отличного от **00**, вывод **D0** переводится в нулевое состояние, конъюнкция с **0** уничтожает результат сумматора. На выводах **s1..s0** наблюдается **0**.

Аналогично соединим конъюнкцию с выходом **D1** дешифратора рис. 3.4. Смотрим на рисунки поочередно, чтобы не испугаться и не запутаться.

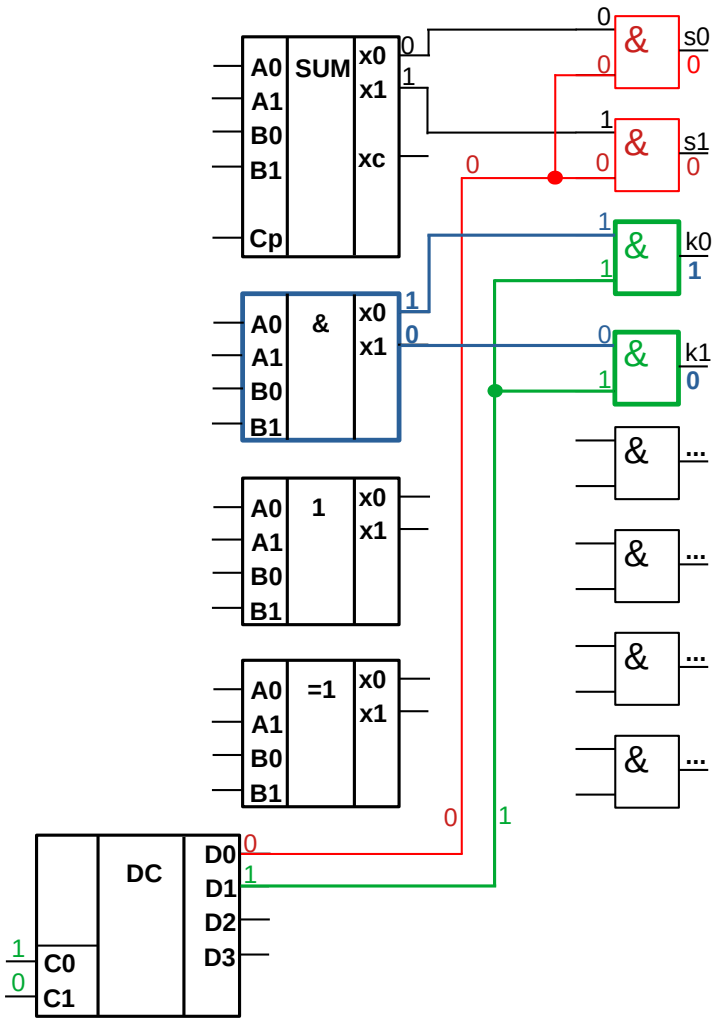


Рисунок 3.4 — Конъюнкция выходов **x0..x1** *сумматора с выходом* **D0** *дешифратора и конъюнкция выходов* **x0..x1** *поразрядной конъюнкции с выходом* **D1** *дешифратора.*

Соединим дизъюнкцию и *XOR* с выходами дешифратора **D2** и **D3** рис. 3.5.

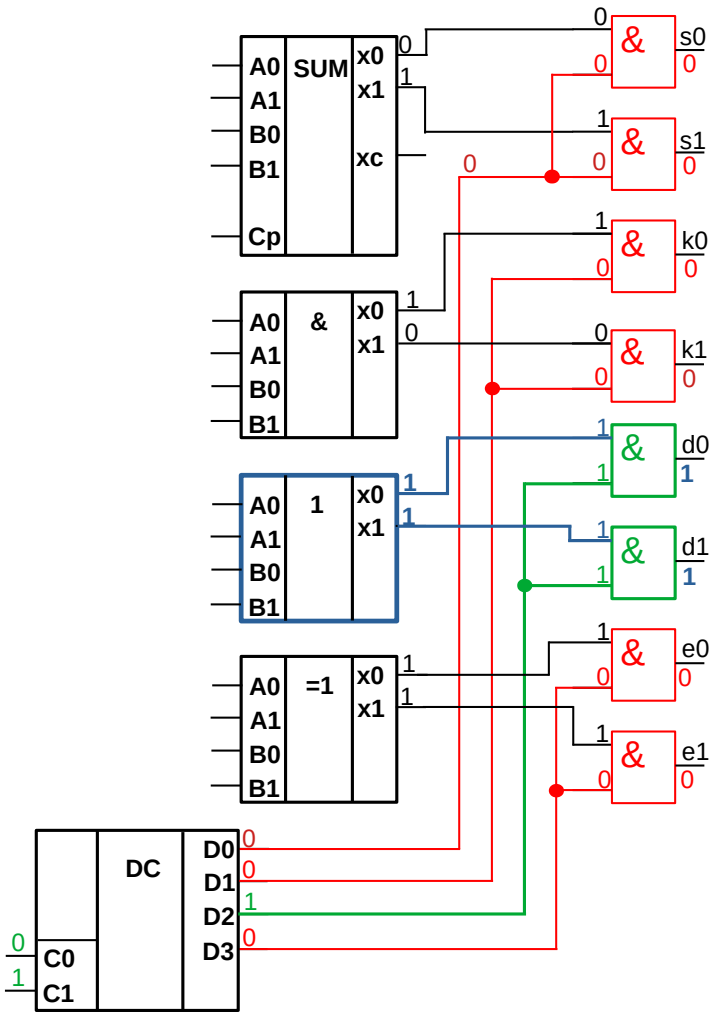


Рисунок 3.5 — Конъюнкция разрядов результатов операций с активирующими их выходами дешифратора.

Обратите внимание, только на одном из выходов (**d1..d0**) находятся искомые данные, которым удастся выбраться из *ALU*. Остальные выходы сброшены в **0**.

Из четырех выходов **s1..s0, k1..k0, d1..d0, e1..e0** следует сделать один, назовем его **x1..x0**. Данное объединение позволит произвести дизъюнкция, т.к. сложение актуального выхода с **0** не изменит результата. Смело просуммируем младшие разряды **s0 | k0 | d0 | x0** рис. 3.6a и старшие разряды **s1 | k1 | d1 | x1** рис. 3.6b результатов.

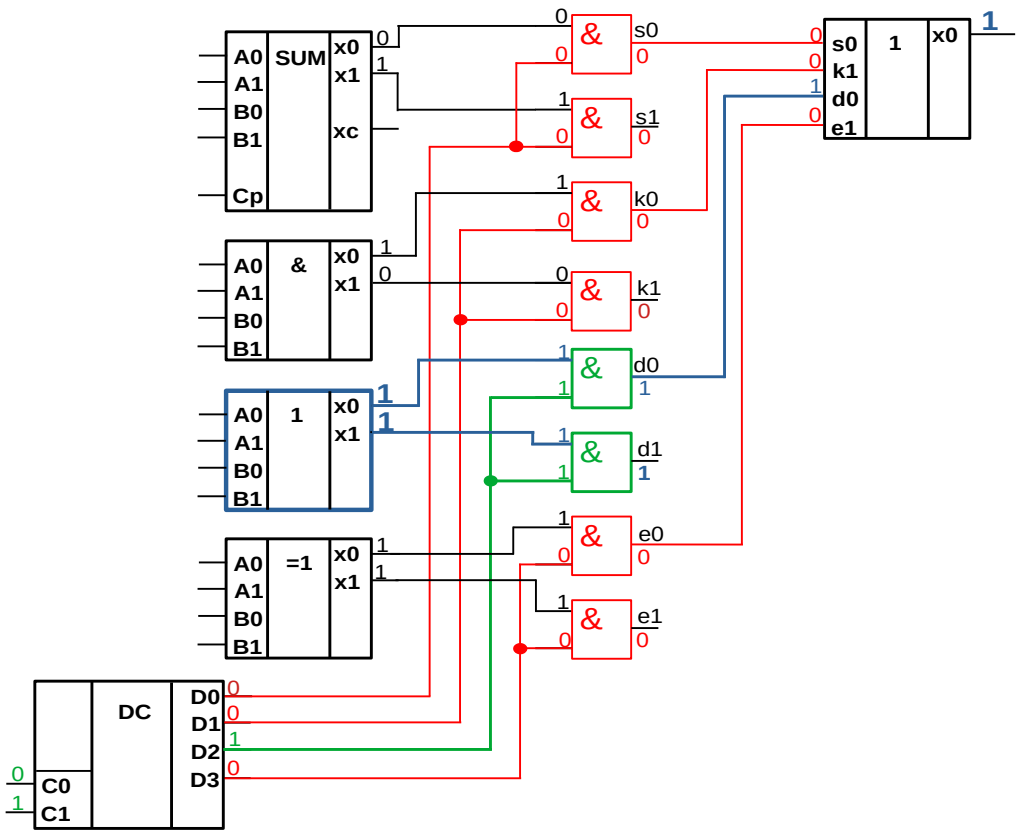


Рисунок 3.6а — Дизъюнкция младших разрядов, отфильтрованных дешифратором

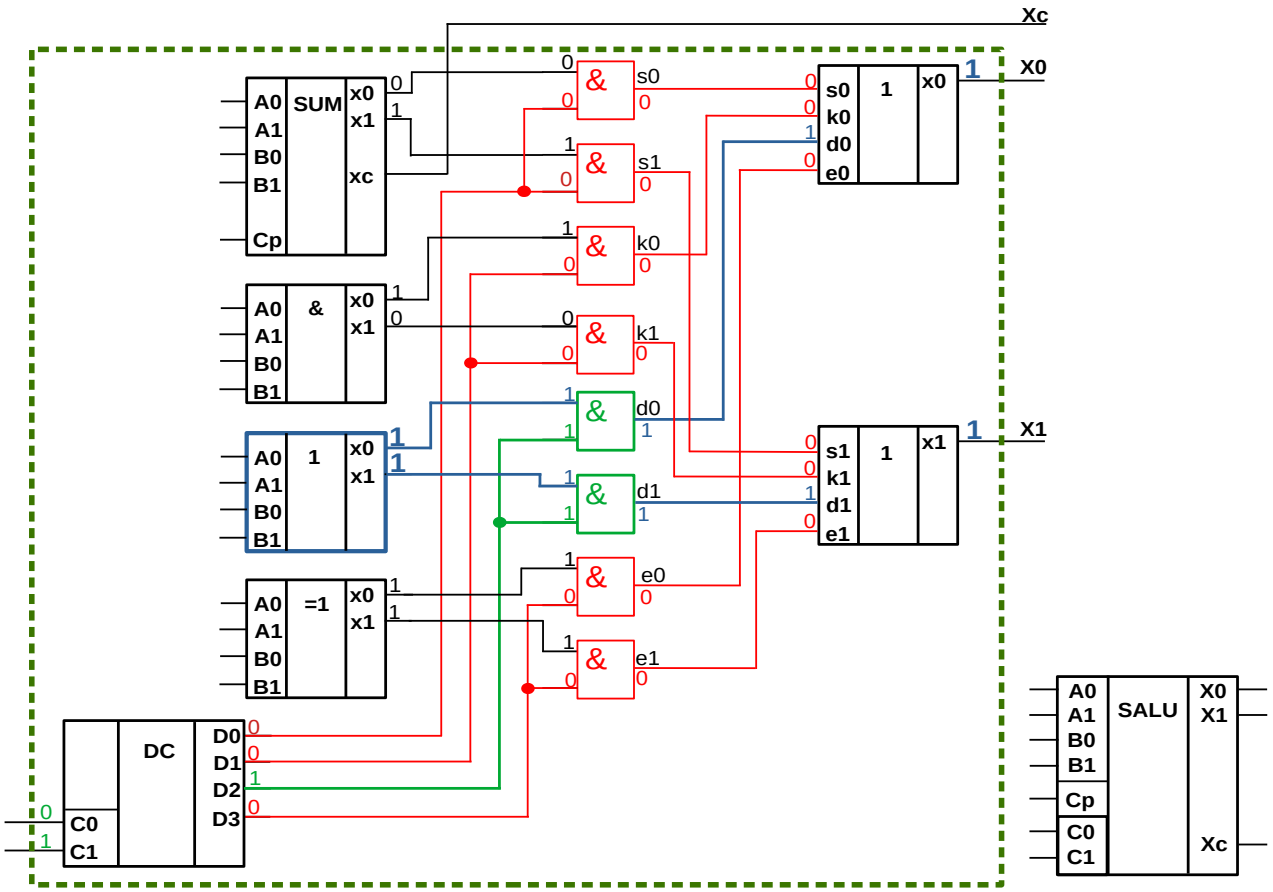


Рисунок 3.6b — Дизъюнкция результатов четырех операций, отфильтрованных дешифратором, и схематичное изображение «середины» ALU — SALU.

Заключим содержимое клочка бумаги в черный ящик зеленого цвета. Назовем ящик Середина *ALU* (сокращенно *SALU* рис. 3.6b). Разумеется, что название Середина, а, тем более, *SALU*, вымышлены с целью уместить изображение дальнейших схем на листе формата А4. Ни в коем случае не произносите их в светском обществе.

Глава 3.2 Конец

После середины логично посмотреть, что находится в конце *ALU*. В конце располагаются результат и флаги. Если быть точнее, то регистр, хранящий результат операции и регистр флагов, хранящий дополнительные признаки, возникшие после выполнения SALU очередной операции. Откуда подать данные на регистр с результатом понятно. Разберемся с флагами.

Флаг **ZF** *(Zero Flag)* — признак того, что после выполнения очередной операции результатом является **0** (все разряды **X1..X0** результата равны **0**). Другими словами, **ZF** = **1**, если результат операции, выполненной *SALU*, получился нулевой. Применим к разрядам **X1..X0** дизъюнкцию (сложим их между собой). Дизъюнкция равна нулю в единственном случае, когда все разряды равны нулю **X1 = X0 = 0**. Отлично, то что нужно. Осталось инвертировать **0** на **1**. Сперва применили *ИЛИ*, потом *НЕ*. А это и есть элемент *ИЛИ-НЕ* (в алгебре логики он называется *стрелка Пирса*) рис. 3.7.

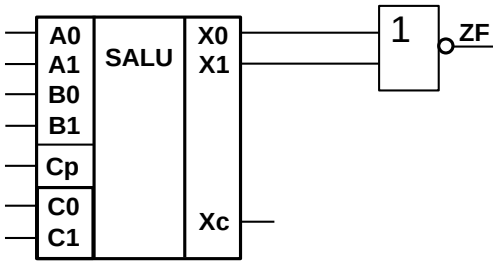


Рисунок 3.7 — Конструирование флага **ZF** *(zero flag).*

Флаг **PF** *(Parity Flag)* — признак того, что после выполнения очередной операции результат содержит четное количество единиц. Четное количество единиц приводит к поднятию флага **PF = 1**. Если быть более точным, то **PF = 1**, в случае, когда *сумма по модулю два* всех битов результата равна нулю. Известно, что элемент *XOR* (его иногда так и называют — *сумма по модулю 2*) отвечает требованиям флага **PF = 1** , за тем лишь исключением, что *XOR* равен нулю, а не единице (**1** *XOR* **1 = 0)**. Инвертируем выход *XOR*, получим *XOR-NOT*. Обратите внимание на кружочек на выходе *XOR*, обозначающий инверсию рис. 3.8.

Сумма по модулю 2 — это операция сложения и вычисление остатка от деления этой суммы на два $A \oplus B = (A + B) \, mod \, 2$. В данной формуле знак плюс в кружочке является оператором *сложения по модулю 2*, а слово *mod* — оператор взятие остатка от деления, в нашем случае на 2. Результат *сложения по модулю 2* равен нулю, если количество единиц четное, либо единиц нет вообще (0 mod 2 = 0).

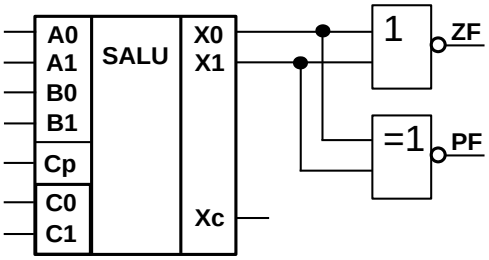


Рисунок 3.8 — Конструирование флага **PF** *(parity flag).*

Флаг **SF** *(Sign Flag)* — знак числа. Обычно старший бит (самый левый) в записи двоичного кода числа отводится под хранение знака.

С использованием 8-ми бит, по-прежнему возможно закодировать $\mathbf{2^8 = 256}$ различных чисел от - **128** до **+ 127**. Введение отрицательных чисел уменьшает модуль максимального числа в два раза, потому что половина чисел становится отрицательными.

Таким образом, флаг **SF** — это просто старший бит результат. Единица в старшем бите есть признак отрицательного числа, инверсии здесь не нужны. Выведем **X1** в качестве флага **SF**. При этом **SF = 1**, если результат очередной операции получился отрицательным.

Отметим, что *ALU* не умеет вычитать. Вычитание реализуется программным способом через сложение. Опустим подробное рассмотрение операций над отрицательными числами. Зато у вас появляется уникальная возможность разобраться с ними самостоятельно. Вас ожидает простая магия двоичной математики.

Флаг **CF** *(Carry Flag)* — переполнение старшего разряда или флаг переноса. Например, при сложении $11_2 + 1_2 = 100_2$ происходит перенос в следующий разряд **X2**, который не предусмотрен конструкцией *ALU*. Возникшая единица на выходе **Xc** сумматора без инверсии, прямиком направляется в флаг **CF**.

Подключение всех флагов к SALU показано на рис. 3.9.

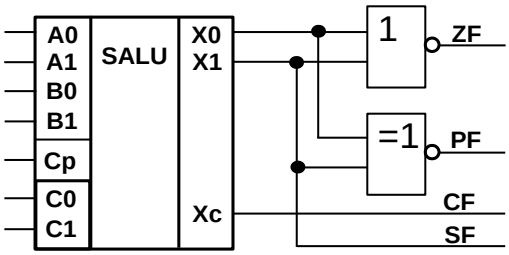


Рисунок 3.9 — SALU с подключенными флагами.

Приправим все регистром флагов и регистром для хранения результата, завершив проектирование конца *ALU* рис. 3.10. Обратите внимание на вход **OE** у регистра результата **RGR**, об этом позже. Назовем, полученную схему *SALUK* (*ALU* с серединой и концом)

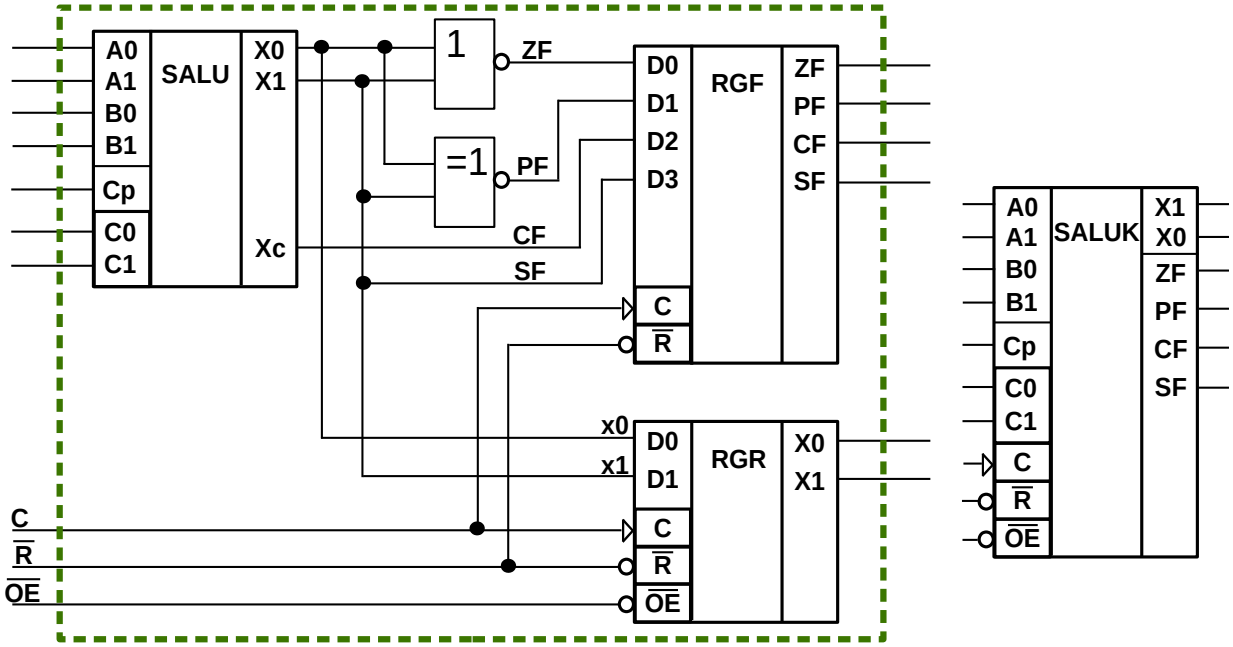


Рисунок 3.10 — Схема ALU без начала, но с серединой и концом (SALUK).

Глава 3.3 Шина

Переместимся в начало и поразмышляем, каким образом возможно подать на вход *ALU* операнды — числа, над которыми *ALU* произведет одну из операций. Сразу становится непонятно, откуда эти числа взять? Скорее всего они хранятся в памяти. Пусть память представляет собой схему, скрывающую множество регистров, каждый из которых хранит некоторое число. Порядковый номер регистра назовем **адресом**. Подача соответствующего адреса (номера регистра) на вход памяти **`A1..A0`**, приводит к возникновению на выходе памяти **`D7..D0`** двоичного кода числа, хранящегося в соответствующем регистре рис. 3.11.

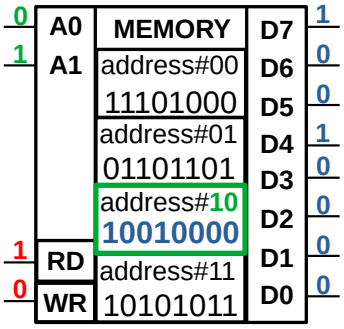


Рисунок 3.11 — Пример извлечения числа из памяти.

Ставится вопрос, с каким из входов *ALU* необходимо соединить выход памяти, **`A`** или **`B`**? Если соединить со входом **`A`**, то где взять второй операнд для входа **`B`**. Использовать ещё одну схему памяти не вариант. Как тогда организовать обмен данными между этими двумя памятями. А еще у *ALU* есть выход, значение которого тоже неплохо было бы сохранить в туже самую память. Человечество нашло гениальное решение — соединило выход памяти со всеми входами и выходами всех устройств внутри микропроцессора, в том числе со выводами *ALU* **`A1..A0, B1..B0`** и **`X1..X0`**. Назвали такое соединение шиной.

Шина — набор проводов, сгруппированных логически, по которым перемещается некоторая информация (**0** и **1**). Набор проводов, по которым перемещаются данные, называется **шиной данных** (*ШД*). Для перемещения внутри микропроцессора 8-ми разрядных чисел необходимо 8 проводов.

Таким образом, восьми разрядная шина данных представляет собой 8 проводов (по одному для каждого разряда), к которым подсоединяются соответствующие разряды всех устройств, оперирующих с данными. В нашем упрощенном примере достаточно двух проводов — двухрязрядной шины данных.

Шина на схемах обозначается полужирной линией, иногда на соединительной линии размещается наклонная черта с числом, обозначающим разрядность шины. Пример схемы подключения памяти к двум регистрам через шину данных *ШД* представлен на рис. 3.12. Один провод скрывает сразу восемь проводов.

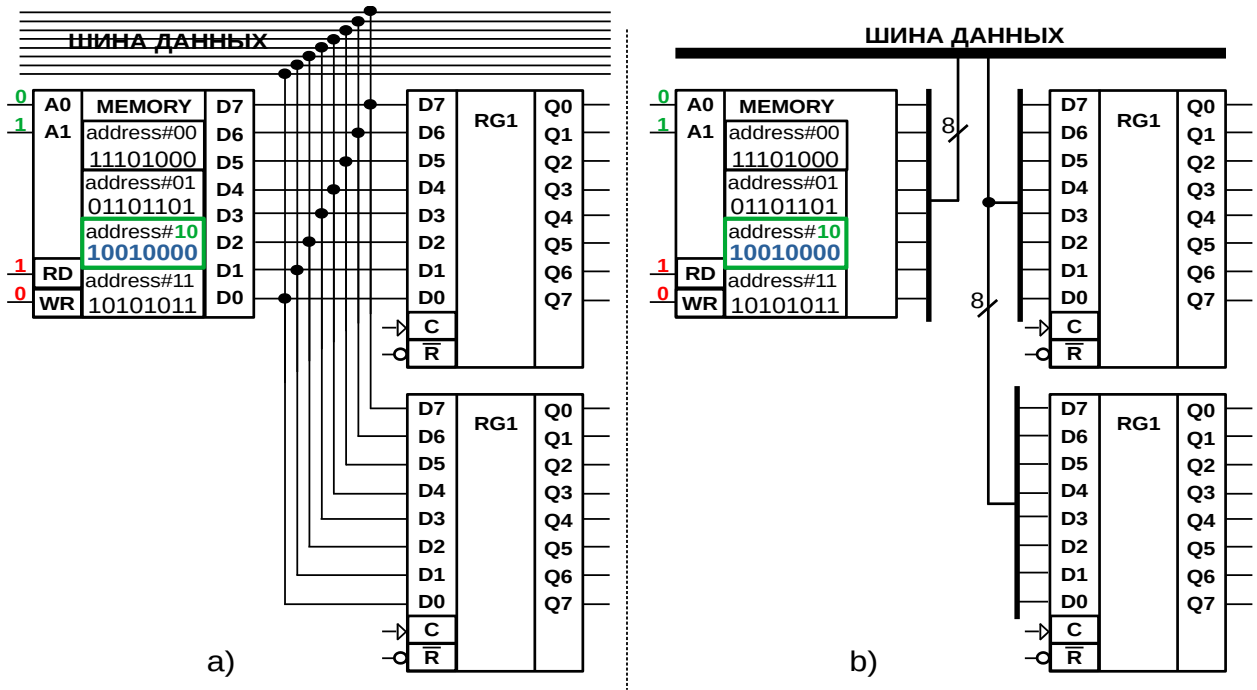


Рисунок 3.12 — Шина на схемах: a — реальные соединения в шине, b — схематичное изображение шины.

Глава 3.4 Начало

Подключим к *SALUK* рис. 3.10 память. Для этого соединим входы и выход *ALU* через шину (два провода) с выходом (он же вход) данных памяти рис. 3.13. Обратите внимание, что выводы **A0, B0, X0, D0** электрически соединены между собой. Тоже самое касается выводов **A1, B1, X1, D1**.

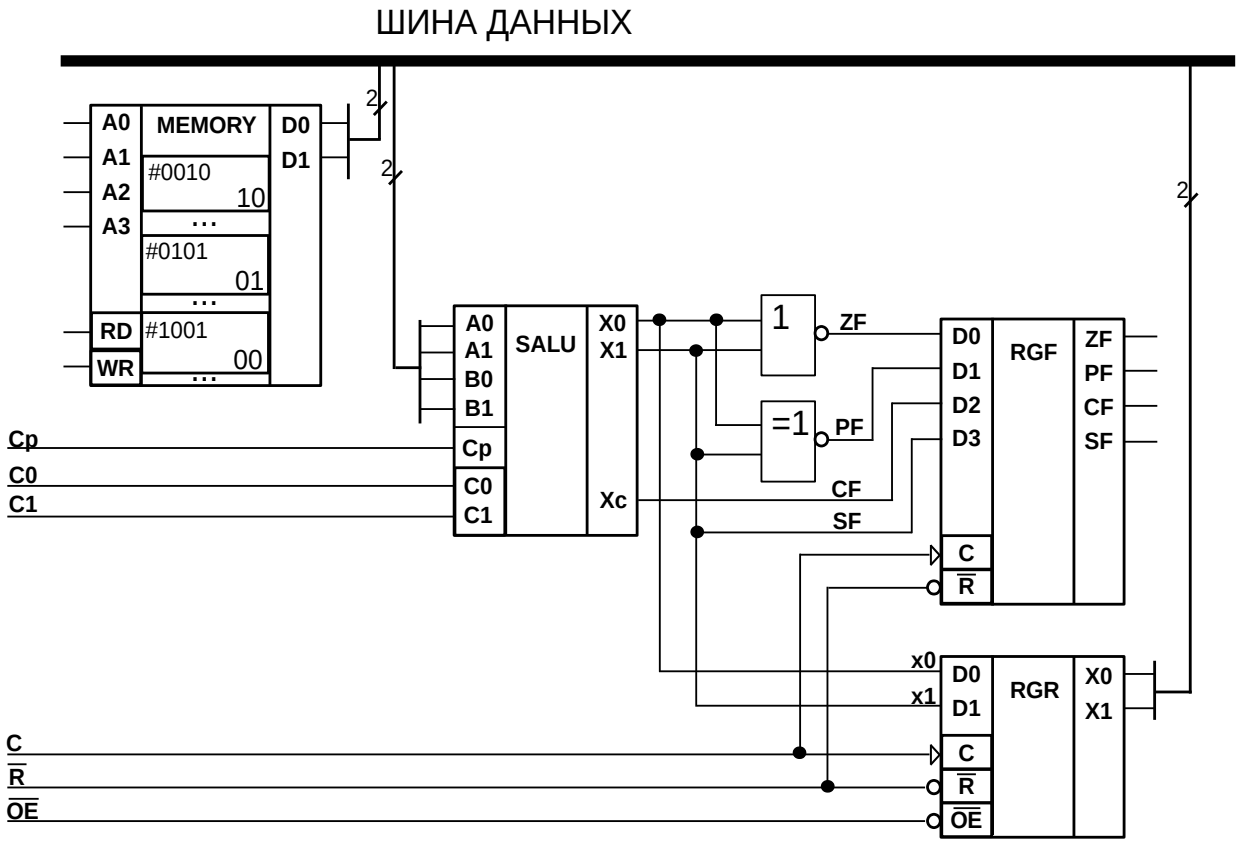


Рисунок 3.13 — подключение SALUK к памяти напрямую.

Теперь одно и тоже значение из памяти поступает входы **A1..A0** и **B1..B0** одновременно.

Допустим необходимо сложить два различных числа. Как быть? Давайте придержим один из операндов в регистре. Подключим между памятью и входом **A1..A0** регистр *RGA* рис. 3.14. Одну проблему решили.

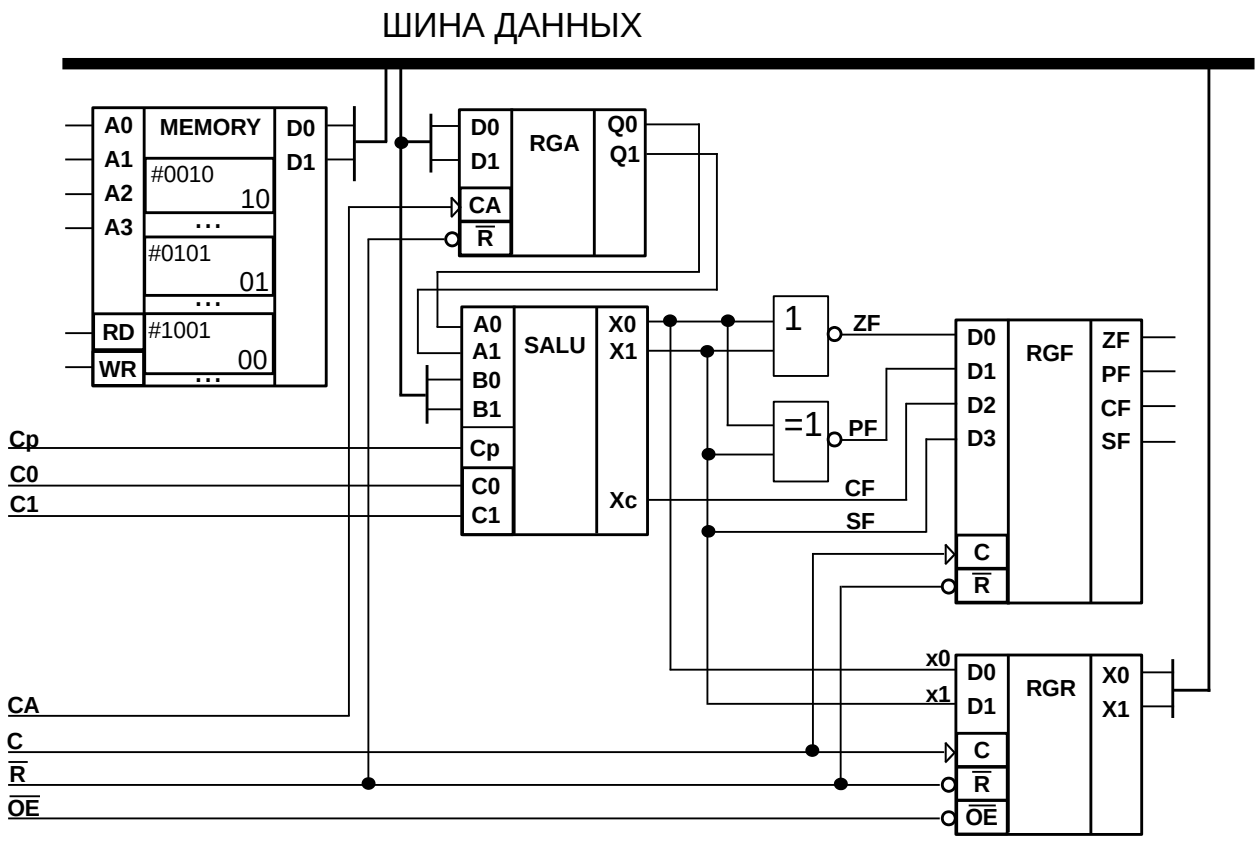


Рисунок 3.14 — Подключение SALUK к памяти через регистр RGA.

Output Enable

Теперь к другой проблеме. На данный момент к шине данных подключено два устройства: выходы памяти и регистр результата операции *RGR*. Получается, на входы **A1..A0** и вход **B1..B0** одновременно поступает значение из памяти и из регистра *RGR*? Разумеет нет!

На шину данных **выдает информацию** всегда **только одно устройство**. Либо память, либо *RGR*. Осуществить такую работу с шиной позволяет управляющий вход $\overline{OE}$ (*Output Enable* -разрешение выхода).

При подаче на $\overline{OE}$ логической **1** выходы устройства полностью отключаются от шины рис. 3.15a.

При подаче на $\overline{OE}$ логического **0** выходы устройства подключаются к шине рис. 3.15b (инвертированный вход активируется при **0**).

Рассматривать электрическую схему *OutputEnable* не будем, а с логикой работы все понятно.

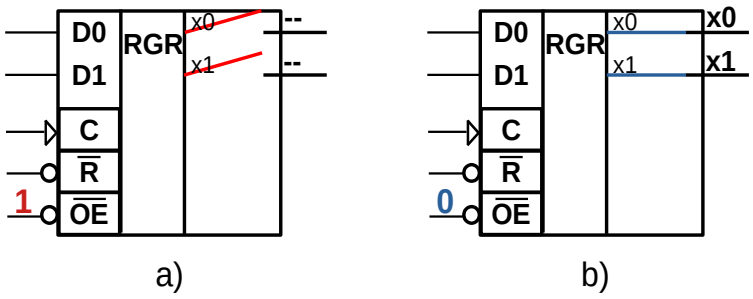


Рисунок 3.15 — Концепция Output Enable: a - $\overline{OE}=1$ - выходы отключены от шины; b - $\overline{OE}=0$ - выходы подключены к шине.

Кстати, на рис. 3.14 изображена схема *SALUK* с началом (регистром *RGA*), т. е. полноценное *ALU*.

В завершении раздела, рассмотрим важный пример.

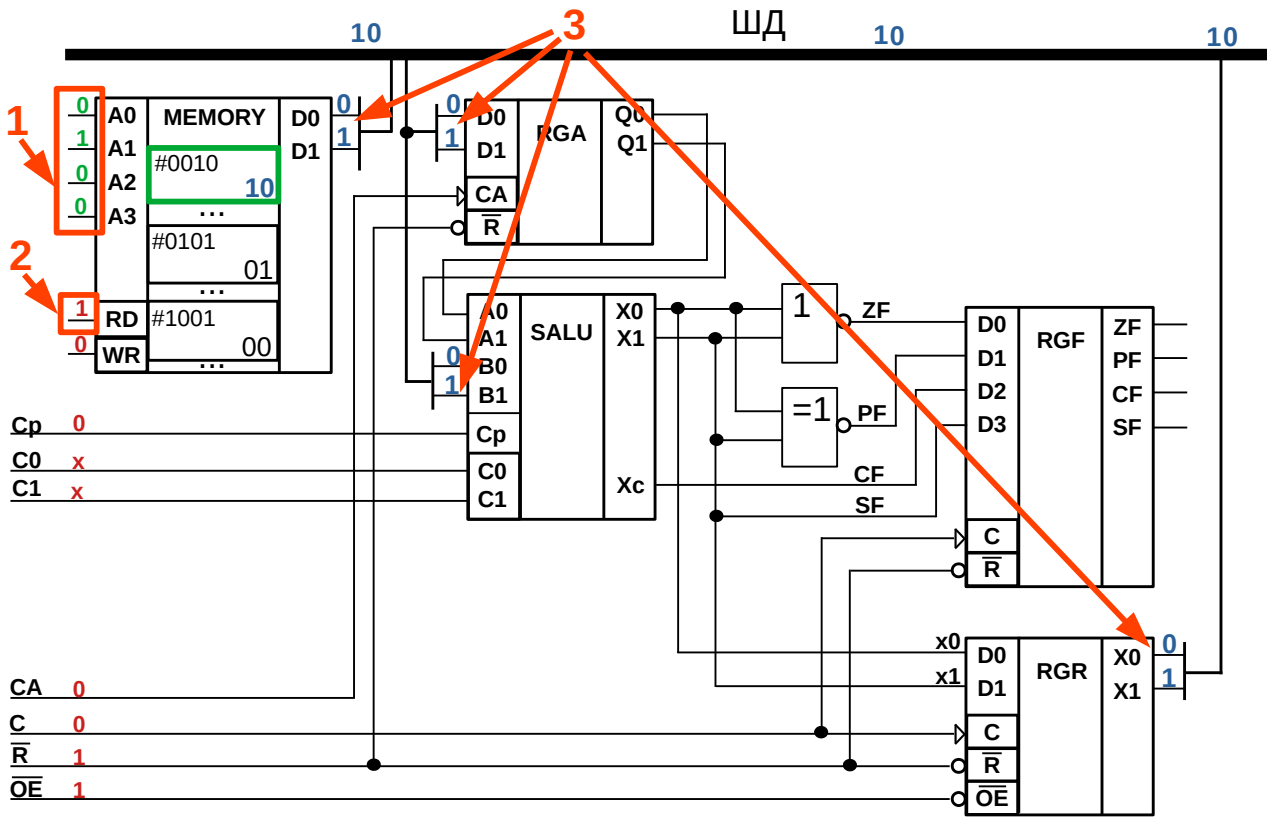
Глава 3.5 Пример работы с АЛУ

Допустим, что в памяти 16 ячеек, т. е. необходима 4-х разрядная шина адреса, *ALU* по-прежнему двухразрядное, шина данных тоже.

*Задача следующая: сложить числа из ячеек памяти с адресами 2 (**0010₂**) и 5 (**0101₂**), результат записать в память по адресу 9 (**1001₂**). В ячейке **0010** хранится число **10₂**, в ячейке **0101** хранится число **1₂**.*

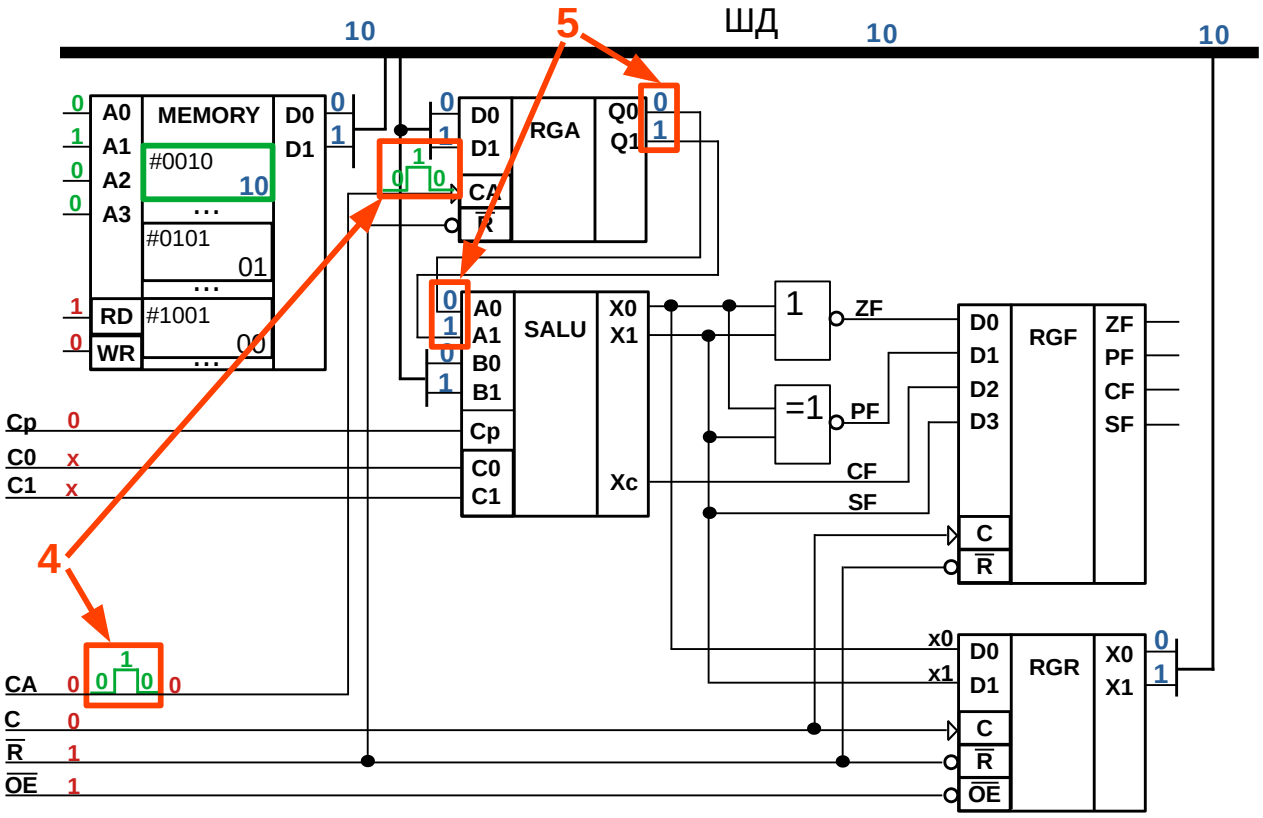
1. Извлечение первого операнда из памяти на ШД

Запишем в *RGA* число, хранящееся по адресу **0010**. Для этого выставим на шину адреса **0010** [1]. После чего переведем память в режим чтения, установив через шину управления сигнал **RD** в единицу [2]. Теперь на *ШД* окажется значение, хранящееся в ячейке по адресу **0010** – число **10**. Оно поступит на входы всех устройств, в том числе и на вход *RGA* [3].



2. Запись первого операнда в RGA

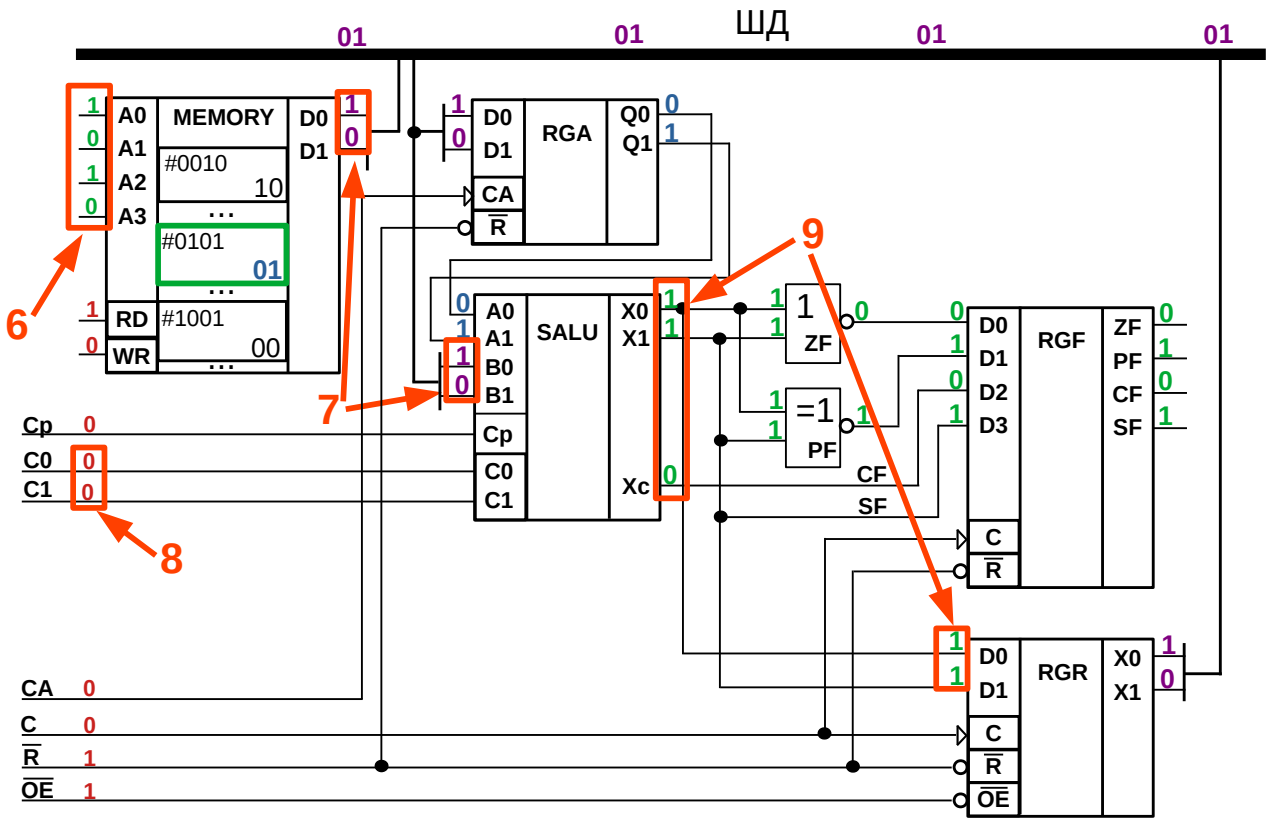
Подадим импульс [4] на вход **C** регистра *RGA*. В регистре сохранится число **10** [5], которое поступит на вход **A1..A0** *ALU* [5].



3. Выставление второго операнда и выполнение операции

3.1 Выставим на шину адреса **0101** [6]. Теперь на входе **B1..B0** *ALU* будет «висеть» число **1** [7], в *RGA* по-прежнему хранится число **10**, которое поступает на вход **A1..A0**. Нам таки удалось выставить два операнда на входе *SALU*, которое осуществит необходимую операцию.

3.2 Подадим команду сложения на входы **C1..C0** [8] дешифратора *ALU*. В этот момент *ALU* совершит вычисления, на выходе **x1..x0** появится результат сложения **11** [9].

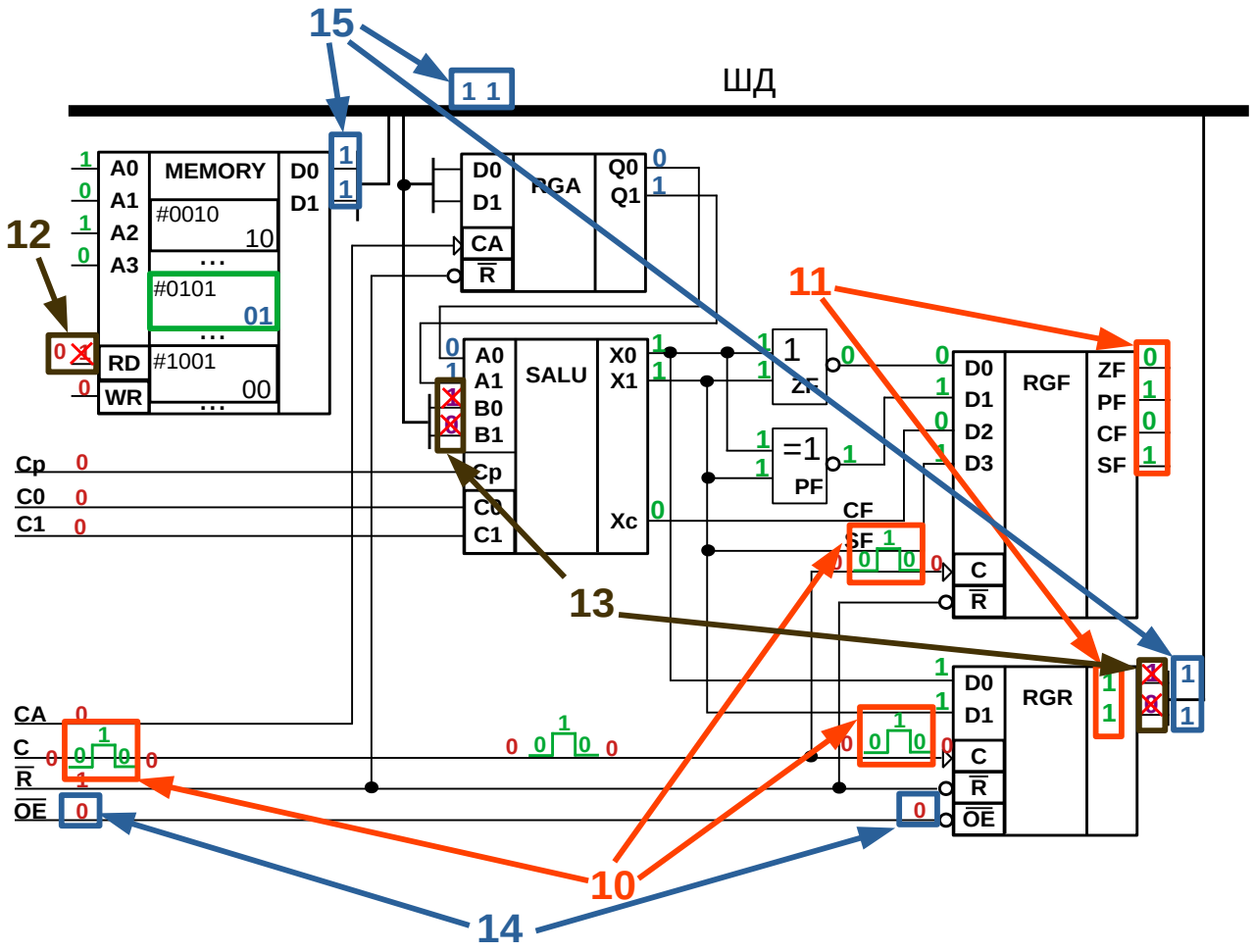


4. Выдача результата на шину данных

4.1 Подадим импульс на вход **C** регистров *RGR* и *RGF* [10] для сохранения результата и флагов. На выходах регистра *RGF* появятся значения флагов [11].

4.2 Отключим память от *ШД*, установив входу **RD** значение **0** [12]. Теперь шина данных «освободилась» от влияния на нее памяти, операнд **01** исчез с шины данных [13], но это не страшно, поскольку результат уже сохранен в *RGR*.

4.3 Направим данные из *RGR* на *ШД*, установив **OE** в значение **0** [14] (до этого момента, на данном входе поддерживался высокий уровень сигнала). Результат **11** оказался на *ШД* и поступил всюду, в том числе и на вход памяти [15].

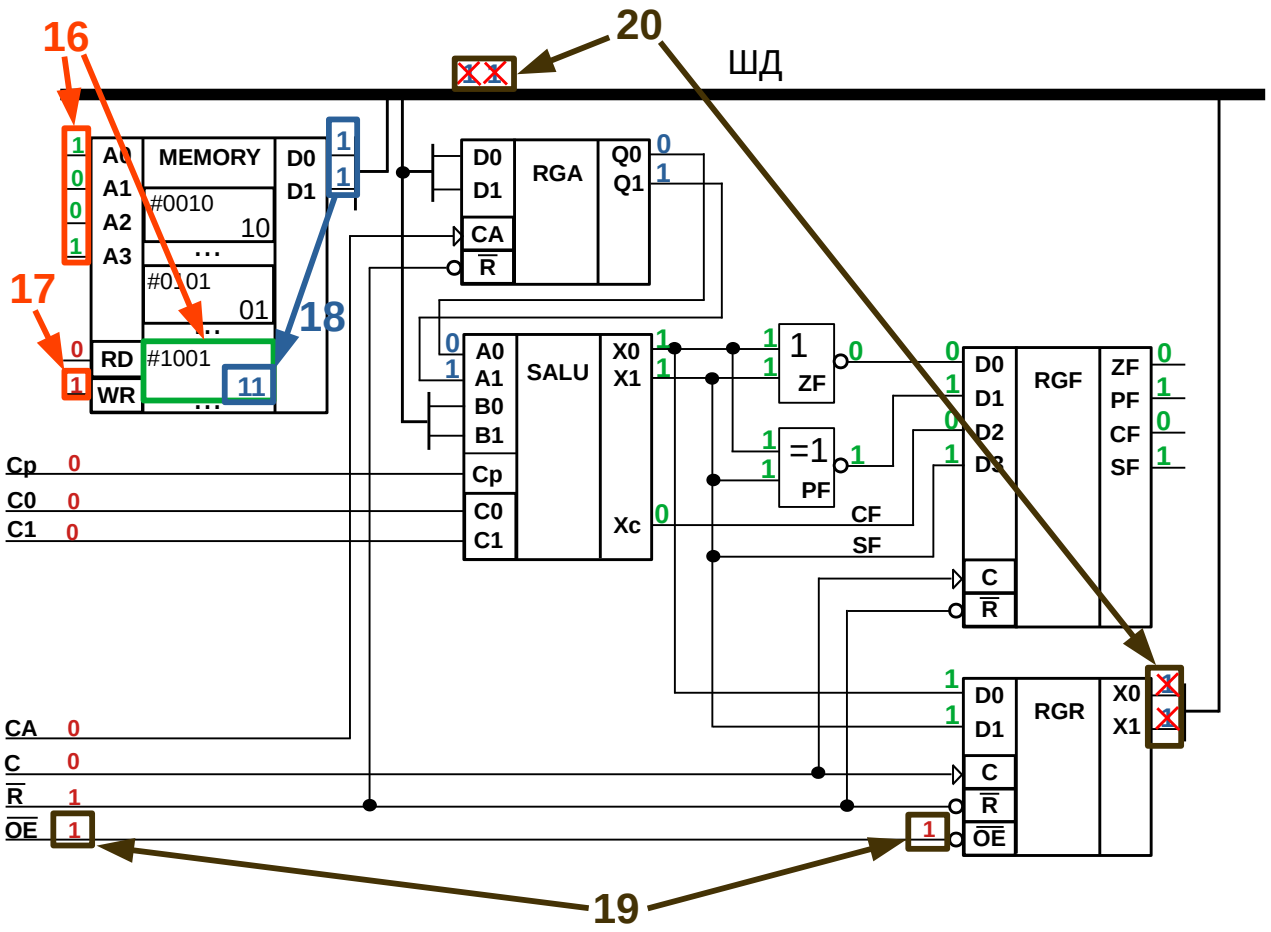


5. Сохранение в память результата

5.1 Выставим адрес **`1001`** [16].

5.2 Подадим высокий уровень на вход **`WR`** памяти [17], тем самым сохраним значение **`11`** в ячейке с адресом **`1001`** [18].

5.3 Установим **`OE = 1`** [19], тем самым отключим *RGR* от *ШД* [20].



Рассмотрение работы *ALU* завершено.

Раздел 4. Микропроцессорная авантюра

Наконец-таки добрались, до самого сокровенного. До кульминации нашей маленькой микропроцессорной авантюры.

Применим метод *дедукции*, чтобы понять принцип работы микропроцессора. Дедукция означает, что сперва будет рассмотрена наиболее общая схема, в которой отбрасываются нюансы и частности. Далее из общей схемы будем вычленять и анализировать все более мелкие и мелкие блоки. В итоге должно придти озарение. К слову, при конструировании *ALU* применялся обратный метод — *индукции* — переход от частностей к общей схеме.

Чтобы не спутать разрабатываемую систему с другими, придумаем условный шифр **МПА (МикроПроцессорная Авантюра).**

МПА устройство сугубо логическое. Предполагается, что все элементы срабатывают мгновенно, нет никаких задержек. В физическом воплощении высока вероятность непредсказуемой работы МПА.

Глава 4.1 Движущие элементы микропроцессора

4.1.1 Счетчики

Движущими элементами микропроцессора являются **счетчики**. Счетчики инициируют последовательную смену команд. Команды меняются, значит программа выполняется.

Счетчик, представляет собой сдвиговый регистр, у которого инвертированный выход $\overline{Q}$ триггера, соединен с его входом **D**. Суть такого включения в том, что каждый раз с приходом импульса на вход **C**, триггер переключается в противоположное состояние (с **0** на **1** и наоборот). Выход **Q** подключается к входу **C** следующего триггера. Устройство счетчика показано на рис. 4.1.

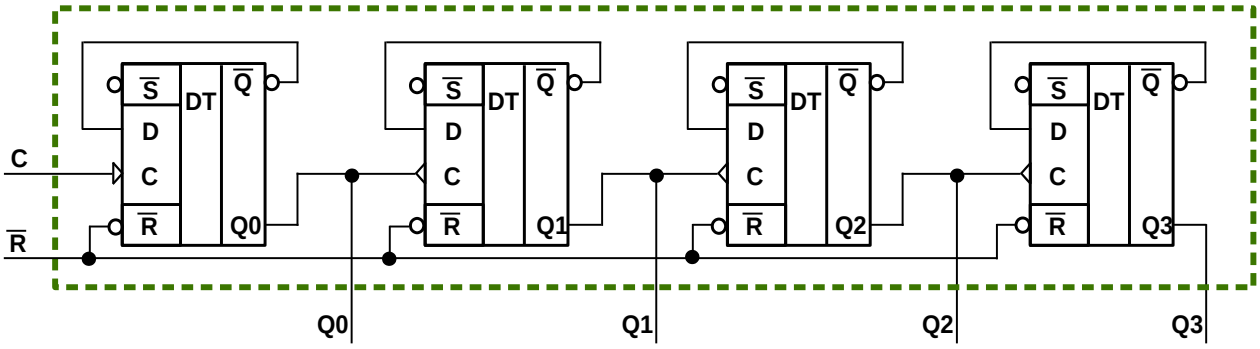


Рисунок 4.1 — Устройство счетчика.

Если требуется начинать счет с определенного значения, существует возможность записи начального значения через входы **D3..D0**, как в обычный параллельный регистр. Для этого добавим специальный вход **WR** (на рис. 4.1 не показан). Легко сообразить, что при **WR = 1** биты **D3..D0** должны инвертироваться и поступать на входы **S** и **R**. Таким образом, осуществляется параллельная запись сразу во все триггеры. При **WR = 0**, на все **RS** должны поступать **1**, схема выполняет счет импульсов.

Рассмотрим, как счетчику удается осуществить свое предназначение — считать.

Предположим, изначально во всех триггерах хранится **0**. Обратите внимание, первый триггер срабатывает по фронту импульса (сигнал изменяется с **0** на **1**), остальные по спаду импульса.

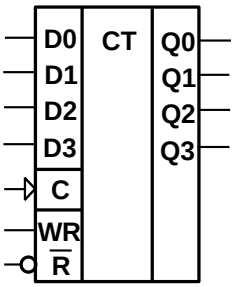
Первый импульс: в первом триггере по фронту импульса защелкивается **1**, которая уже давно подошла к входу триггера от инвертированного выхода**. Q3Q2Q1Q0 = 0001** — счетчик досчитал до одного.

Второй импульс: первый триггер изменяет свое выходное состояние на **0**, при этом спад импульса на выходе **Q0** приводит к переключению второго триггера в состояние **1**. **Q3Q2Q1Q0 = 0010 –** счетчик досчитал до двух.

Третий импульс: первый триггер изменяет свое выходное состояние на **1. Q3Q2Q1Q0 = 0011** — счетчик досчитал до трех.

Четвертый импульс: первый триггер изменяет свое выходное состояние на **0**, что приводит с изменению состояния второго триггера на **0** по спаду импульса **Q0**, возникает спад на выходе **Q1** для третьего триггера, который изменяет состояние на **1**. **Q3Q2Q1Q0 = 0100 -** счетчик досчитал до четырех.

Пятый импульс: первый триггер изменяет свое выходное состояние на **1**, остальные не изменяются **Q3Q2Q1Q0 = 0101**. Счетчик досчитал до пяти. И т. д. Схематичное изображение счетчика и таблица истинности в режиме счета, показаны на рис. 4.2.



C	Q3	Q2	Q1	Q0	Десятичный эквивалент
Не было импульсов	0	0	0	0	0
Один ипульс	0	0	0	1	1
Два импульса	0	0	1	0	5
Три импульса	0	0	1	1	3
Четыре импульса	0	1	0	0	4
Пять импульсов	0	1	0	1	5
...	..	..	..	..	...
Пятнадцать импульсов	1	1	1	1	15

Рисунок 4.2 — Схематичное изображение счетчика и его таблица истинности

Выход **Q3Q2Q1Q0** есть двоичное число, равное количеству фронтов импульсов, поступивших на вход **C**. Устройство оказалось таким простым, потому что счет в двоичной арифметике реализуется изменением значения в каждом разряде с **0** на **1** и наоборот. *D*-триггер, у которого $\overline{Q}$ соединен с *D*, выполняет тоже самое действие.

4.1.2 Память энергозависимая

Память **RAM** расшифровывается как *Random Access Memory* (память с произвольным доступом). По-русски, Оперативное Запоминающее Устройство (**ОЗУ**). В предыдущем *ALU*-разделе, *RAM* называлась просто — *MEMORY*. Память *RAM* энергозависимая. При отключении питания, информация исчезает. Существует несколько разновидностей энергозависимой памяти.

1. *DRAM (DynamicRAM)* — каждая ячейка памяти состоит из транзистора и конденсатора. Физическим свойством, осуществляющим хранение бита информации, является *заряд конденсатора*. Конденсаторы имеют свойство разряжаться. С целью сохранения информации ячейки *DRAM* необходимо постоянно обновлять, подзаряжать - *регенерировать*. *Регенерация* осуществляется примерно каждые 50 мс. В связи с постоянной *регенерацией* память *DRAM* работает относительно медленно, зато она дешевая и может иметь большие объемы (т.к. размер ячейки очень маленький, состоит всего из 2-х элементов). *DRAM* отлично подходит в качестве оперативной памяти персонального компьютера ПК, которая располагается в разъеме материнской платы, и которой так любят хвастаться дети.

2. *SRAM (StaticRAM)* — в основе данной памяти лежат знакомые триггеры. *SRAM* не требует регенерации, поэтому работает быстрее *DRAM*. Однако, из-за сложной конструкции ячейки (вспомните сколько элементов в триггерах) большие объемы изготовить труднее и дороже. *SRAM* используется в качестве внутренней памяти микропроцессора. Обычно на основе *SRAM* строятся регистры и кэш-память.

КЭШ-память используется для временного хранения данных и команд, с целью уменьшить количество обращений к менее быстрой *DRAM*. КЭШ память вместе с *конвейером* — это всё тонкости архитектуры современных микропроцессоров, позволяющие увеличить их производительность.

Итак, *SRAM* — быстрая, дорогая, маленькая из нее состоит память микропроцессора, хранит текущие команды и данные, которые обрабатываются, будут обрабатываться или были обработаны в ближайшее время.

DRAM — медленная, большая, дешевая. Это оперативка (в современных компьютерах ~8 Гб). Используется для хранения данных запущенных программ, переменных, массивов и всего того, с чем имеет дело высокоуровневое программирование (PASCAL, C++, …), а также другой информации, обрабатываемой в текущем процессе.

Программа при запуске с жесткого диска копируется в *DRAM*, т.к. доступ к данным на жестком диске осуществляется относительно бесконечно долго. Микропроцессор старается больше общаться с быстрой *SRAM*, в крайнем случае с *DRAM*, а не тормозным жестким диском.

4.1.3 Память энергонезависимая

ROM — *Read Only Memory* или Постоянное Запоминающее Устройство (ПЗУ). Энергонезависимая память. Используется для длительного хранения данных и программ после отключения питания. Например, магнитные и твердотельные жесткие диски, FLASH, CompactDisk, магнитофонная лента. Записи в тетради — тоже энергонезависимая память.

Read Only Memory, в строгом смысле слова, «память только для чтения», т.е. память, в которую нельзя ничего записать. Предполагается, что запись возможна только один раз.

Изменить данные можно в *EPROM* (*Erasable Programmable ROM*), *EEPROM* (*Electrically Erasable Programmable ROM*). *FLASH*, кстати, относится к *EEPROM*.

Видов много, будем же смотреть на всякую *ROM* как на ПЗУ — устройство, запоминающее и хранящее данные после отключения питания.

Глава 4.2 Элементы УИИ

Проанализируем схему микропроцессорной авантюры (*МПА*), представленную на рис. 4.3. Данную часть схемы условимся называть Устройством ИсполненИя (*УИИ*).

Из схемы *УИИ* делаем вывод, что *МПА* получается четырех разрядной. За одну команду она способна обработать не более 4-х бит информации. Максимальное число, которое может получиться в результате вычислений МПА составляет $\mathbf{2^4 - 1 = 15}$. Шина адресе *ША* тоже четырех разрядная, т. е. максимально возможное количество ячеек в памяти — 16. Не так уж и много, всего лишь 8 байт. Но, для понимания сути МПА, более чем достаточно. При увеличении разрядности, увеличивается площадь бумаги, отводимая для размещения рисунка.

Из всего многообразия, представленного на рис. 4.3, наиболее знакомыми являются три элемента: *ALU*, *RAM* и *ШД*. Из новенького три регистра (*PC*, *SP*, *RGX*), *ROM* и порт *PRT*.

Уясним назначение и принцип действия каждого элемента *УИИ* в отдельности, затем рассмотрим в системе.

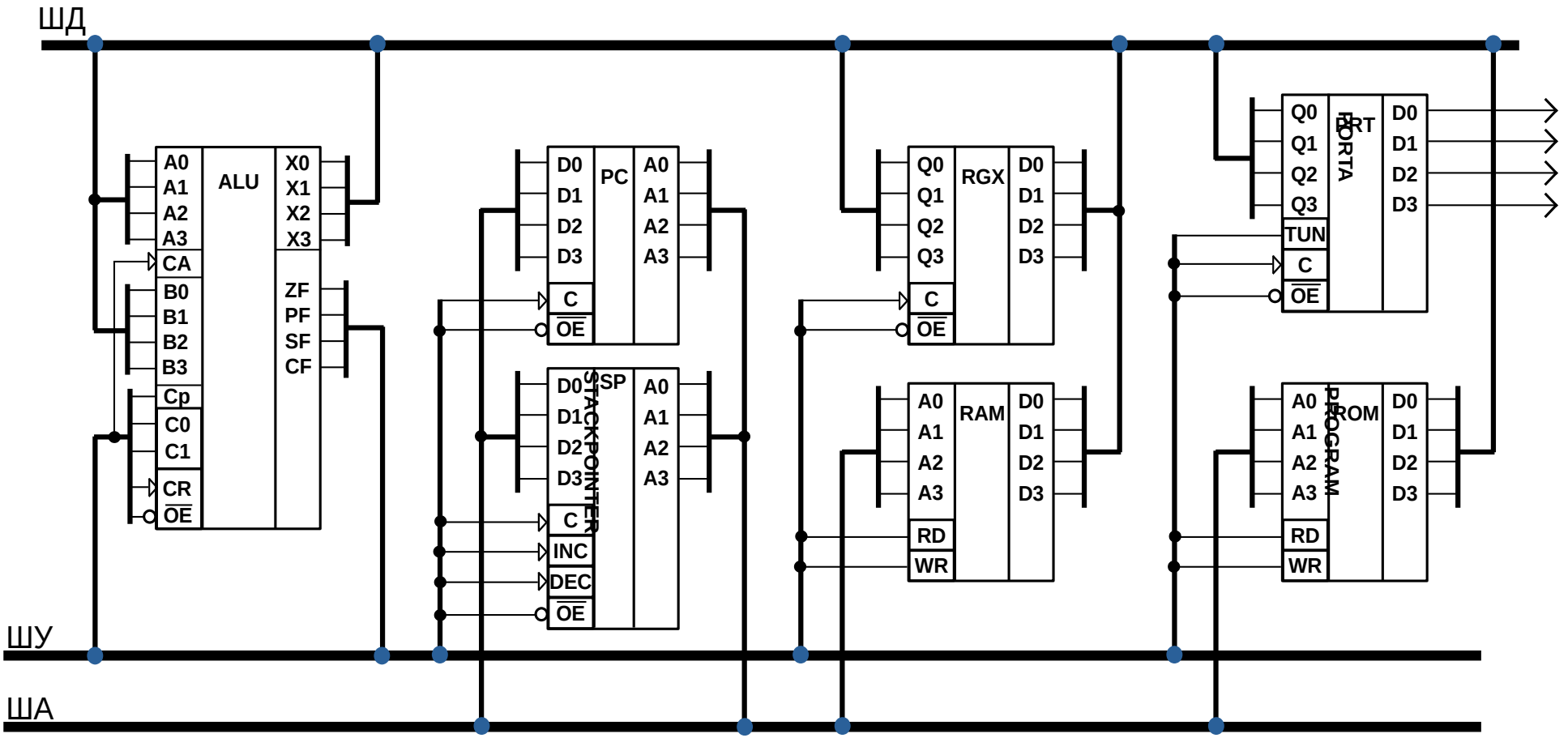


Рисунок 4.3 Общая схема устройства исполнения микропроцессорной авантюры УИИ МПА.

4.2.1 RAM

Внутренняя оперативная память *МПА* в *SRAM* исполнении. Предназначена для записи и хранения промежуточных результатов вычислений, возникающих в ходе выполнения программы. Возможны следующие режимы работы:

- **`WR = 1`** — память находится в режиме записи, данные с *ШД* сохраняются в ячейку с адресом, поступившем по *ША* на входы **`A4..A0`**.
- **`RD = 1`** — память находится в режиме чтения, данные из ячейки с адресом **`A4..A0`** поступают на *ШД*, соответственно направляются на все элементы *МПА*, соединенные с шиной данных.
- **`WR = RD = 0`** — память отключена от *ШД*, не оказывает на шину никакого воздействия.
- **`WR = RD = 1`** — недопустимое состояние.

4.2.2 ROM

Постоянное запоминающее устройство. Хранит программу, написанную программистом.

Программа — последовательность команд.

Команда — число, которое микропроцессор в состоянии понять и выполнить соответствующие действия.

Другими словами, **программа** — последовательность чисел, которые микропроцессор способен «понять» и выполнить.

4.2.3 RGX

Самый важный и единственный универсальный регистр в архитектуре *МПА*. Универсальный в том смысле, что у регистра нет особого назначения. *RGX* промежуточный регистр-островок, через который в *МПА* данные перебрасываются из *ALU* в *RAM* или *PRT*.

4.2.4 PRT

Порт — схема, осуществляющая взаимодействие *МПА* с устройствами внешнего мира.

Лампочки, механизмы, приводящие в движение космический летательный аппарат и т. д., все это подключается к порту. Причем *PRT* умеет работать и на ввод, например, обрабатывать нажатие кнопок или поступающие на радиолокатора сигналы. Что именно пользователь подключит к *МПА* зависит от фантазии и количества доступных у порта ножек.

*И*так, порт — **устройство ввода/вывода**. Схема *PRT* скрывает в себе три регистра. Один для вывода (*OUT*), другой для ввода (*IN*), третий для настройки (*TUNE*).

При **`TUN = 1`** — порт находится в режиме настройки. Настройка регистра осуществляется записью по *ШД* в регистр *TUNE* четырех-разрядного *настроечного слова*. Каждый разряд *настроечного слова* определяет в какую сторону работает соответствующий разряд *PRT*. Единица означает настройку соответствующего разряда порта на ВЫВОД, а ноль на ВВОД. Таким образом, возможно осуществить одновременную настройку порта на ввод и вывод. Часть ножек на ввод, остальные на вывод. Подробно рассматривать внутреннее устройства порта не будем.

4.2.5 Stack Pointer

SP (Stack Pointer) — **указатель стека**.

По определению, **стек** — структура данных (в оперативной *RAM*) по типу «стопка тарелок». Значение, попавшее в стек последним, в первую очередь его и покинет. Указатель стека *SP* хранит *адрес* ячейки *RAM*, поэтому выход регистра соединен с *ША*, а не *ШД*.

Данные, помещаемые в стек, записываются по адресу, хранимому в *SP*, после чего значение адреса в *SP* уменьшается на единицу. Таким образом, *SP* указывает на ячейку стека (область *RAM*), в которую будет записана следующая порция данных (число).

При извлечении из стека значение в *SP* увеличивается на единицу. Данные извлекаются из *RAM* по адресу, хранимому в *SP*.

Импульс на входе **`INC`** приводит к увеличению на единицу (*increment*), импульс **`DEC`** приводит к уменьшению на единицу (*декремент*). С *инкрементом* все понятно — обычный режим работы счетчика. Объяснение *декремента* опустим. Представим, что счетчик умеет считать и в обратном направлении, чтобы не усложнять схему.

Подробнее работу со стеком рассмотрим при реализации в машинных кодах вызова подпрограммы.

4.2.6 Programm Counter

PC (Programm Counter) — программный счетчик.

Самый интересный регистр. Напомним, что программа хранится в ячейках постоянной памяти *ROM*. Команды располагаются в *ROM* последовательно друг за другом, в смежных (соседних) ячейках.

Программный счетчик хранит адрес следующей команды, которую устройство исполнения *УИИ* планирует выполнить. После извлечения из *ROM* по адресу, указанному в *PC*, очередной команды, значение программного счетчика увеличивается на единицу (*инкремент*).

Некоторые команды способны переводить выполнение программы к команде, располагающейся на несколько ячеек раннее или далее от адреса текущей команды. В данном случае, выполнение команды приводит к перезаписи значения *PC*, на адрес ячейки с необходимой командой. Таким образом, становится возможным осуществление ветвлений и циклов. **Следование, ветвление и цикл, позволяют реализовать любой алгоритм.**

Итак, **программный счетчик** указывает на адрес команды, которая будет выполнена **следующей**. Обычно отсчет начинается с нуля.

4.2.7 Шина управления

Работа *УИИ* регулируется устройством управления *УУ*, которое рассматривается в следующей главе.

Все сигналы, возникающие на *шине управления ШУ*, генерируются устройством управления *УУ*. Эти сигналы поступают на служебные входы устройств (**`C, OE, RD, WR`** и **`т.д.`**).

УУ позволяет выполнить команду *МПА*, выставляя управляющие сигналы в определенной последовательности. Обратите внимание, что флаги *ALU* соединены с *ШУ*, они являются входными значениями для устройства управления *УУ*. Опираясь на значения флагов, *УУ* генерирует необходимые команды, реализуя ветвления и циклы.

Пока достаточно просто, знать, что устройство *УУ* существует и выставляет необходимые сигналы на шину управления *ШУ*.

4.2.8 Такты

Такт представляет собой минимальную единицу исполнения команды. Изменение значений на любых из выводов, представляет собой такт. Изменение состояния системы и есть такт. Смена таких состояний приводит к выполнению команды.

Микропроцессоры работают по тактам. Наверняка, известно понятие тактовая частота — количество тактов в секунду. Так вот — это не количество команд. А именно тактов. Для выполнения каждой команды требуется различное количество тактов. В *МПА* каждая команда будет выполняться за 8 тактов. Такты необходимы, чтобы синхронизировать работу всей системы и исключить непредсказуемое поведение схемы, связанное с задержками протекания физических процессов во времени. Поэтому слишком сильно увеличивать тактовую частоту не получится.

Глава 4.3 Исполнение программы в УИИ

Рассмотрим основные этапы выполнения программы устройством исполнения, начиная с подачи питания и заканчивая отключением питания.

4.3.1 Инструкции

Первым делом программу необходимо написать. Собственно ставится вопрос, что писать? Двоичные коды! Любая команда для *МПА* представляет собой 8-битное число. Команды программы будем называть *инструкциями*.

oper7 oper6 oper5 oper4 arg3 arg2 arg1 arg0

Младшие 4-ре разряда **arg3..arg0** представляют собой **данные**: некоторое число или адрес.

Старшие 4-ре бита **oper7..oper4** определяют **код операции** (**КОП**) — действие, которое будет произведено над данными. В качестве КОП может выступать совершение логической или арифметической операции, запись или чтение в *RAM*, *PRT* или *RGX* и некоторые другие действия. Максимальное количество инструкций *МПА* определяется количеством битов, отводимых под КОП, и составляет $\mathbf{2^4 = 16}$.

Набор таких двоичных чисел (инструкций) записывается в ROM.

Обозначения чисел в различных системах счисления

Когда имеют дело с микропроцессорами, адреса и данные часто записываются в 16 - ой системе счисления. Будем пользоваться следующими обозначениями:

16-ая система **#FE** либо **FEh** (#, h — *Hexadecimal* - шестнадцатеричная)

10-ая система **254d** (d — *Decimal* — десятичная)

2-ая система **11111110b** (b — *Binary* — двоичная)

Перевод числа из 2-й системы счисления в 16-ю осуществляется группировкой справа налево (именно справа!) цифр двоичного числа в тетрады. Тетрада — это блок из **4-х** цифр. Каждая тетрада записывается одной 16-ой цифрой. В нашем примере, две тетрады: **1111b = #F** и **1110b = #E**. Получается **11111110b = #FE**. Перевод из 16-й в 2-ю производится аналогично.

4.3.2 Включение МПА

Представим, что у каждого устройства есть вход **RESET**, который сбрасывает все его выходы в **0**. При подаче питания, происходит **RESET**. Программный счетчик *PC* хранит значение **0000b**. Начинается работа *МПА*. Для удобства представим ещё раз схему *УИИ* рис. 4.4 .

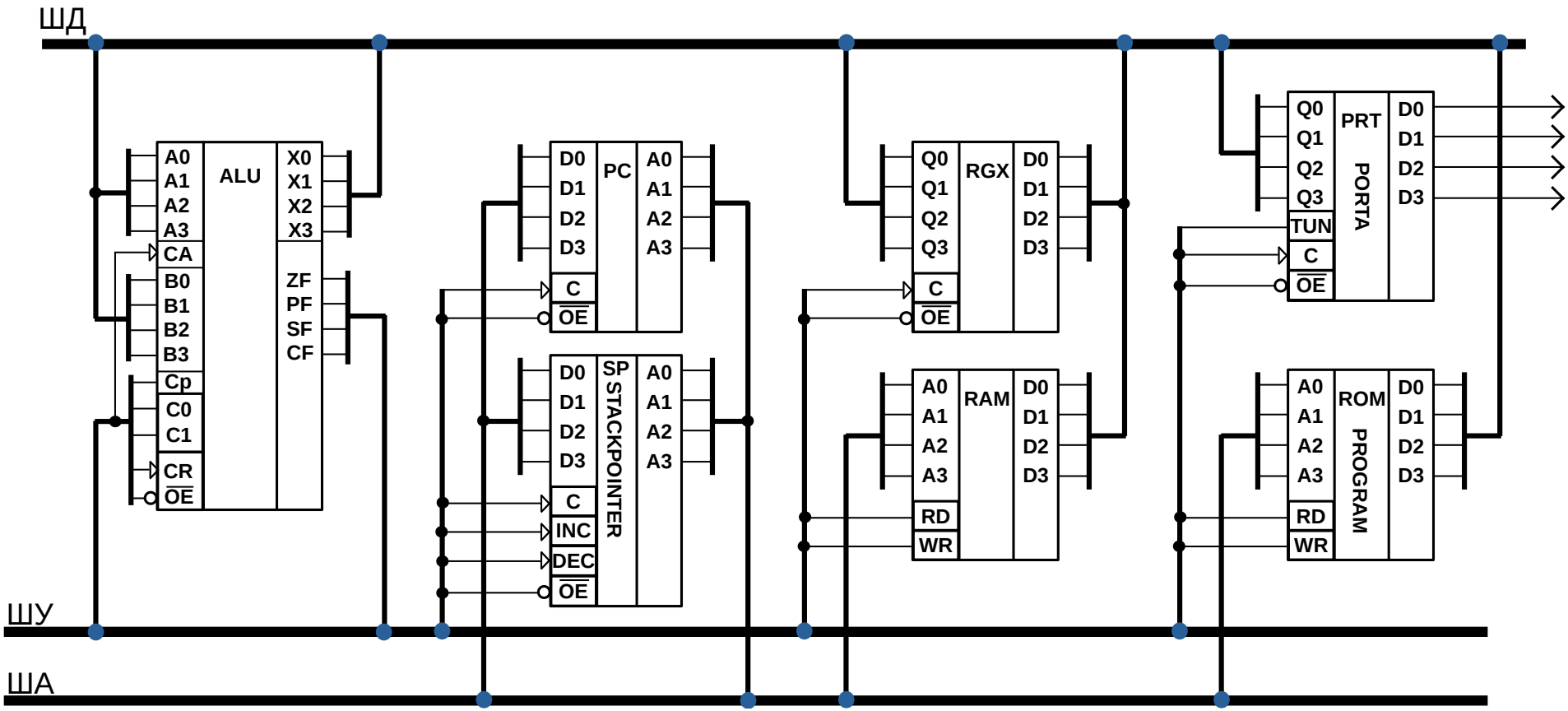


Рисунок 4.4 — УИИ МПА — устройство исполнения микропроцессорной авантюры.

4.3.3 Инструкция FETCH

Первой командой, которая выполняется любым микропроцессором, является извлечение инструкции из памяти *ROM* и ее запись в *УУ* для дальнейшей обработки. Микропрограмму, осуществляющую данные действия, называют **инструкция FETCH**. Инструкция FETCH выполняется за 8-мь тактов.

Такт 1: *УУ* выдает на **`PC[OE] = 0`**, тем самым **`0000b`** оказывается на *ША*.

Такт 2: *УУ* выдает на **`ROM[RD] = 1`** (при этом **`PC[OE] = 0`** остается), команда, хранящаяся по адресу **`0000b`** поступает на *ШД*.

Такт 3: *УУ* сохраняет первую часть команды в своем внутреннем регистре команд. Поскольку данные 4-х разрядные, то из *ROM* сперва извлекаются и сохраняются младшие биты команды (аргумент).

Такт 4: *УУ* выдает **`PC[C] = 1`** (**`PC[OE] = 0`**), фронт провоцирует срабатывание счетчика **`0 + 1 = 1`**, тем самым **`0001b`** оказывается на *ША*. *ROM* по-прежнему работает на чтение **`RD[ROM] = 1`** и на *ШД* оказывается вторая часть команды — код операции КОП. Как много всего произошло за один такт!

Такт 5: *УУ* сохраняет вторую часть команды в своем внутреннем регистре команд. Теперь команда извлечена полностью. Также сбрасывается импульс на счетчике **`PC[C] = 0`**, счетчик снова готов поддаться фронту следующего импульса.

Такт 6: *УУ* выдает на **`PC[OE] = 1`** и **`RD = 0`**, тем самым освобождает *ША* и *ШД* для работы с другими устройствами. Одновременно с этим **`PC[C] = 1`**, фронт провоцирует срабатывание счетчика **`0001b + 1b = 0010b`**. Теперь счетчик хранит адрес ячейки *ROM*, хранящей следующую команду.

Такт 7: УУ выдает на **`PC[C] = 0`**, счетчик снова готов считать.

Такт 8: Все затихает… Только текущая команда потрескивает внутри устройства управления и ждет своего исполнения.

4.3.4 Выполнение программы

Таким же образом как было с командой FETCH, *УУ* генерирует с каждым новым тактом необходимые сигналы для выполнения извлеченной инструкции.

После исполнения первой инструкции программы, происходит выполнение команды FETCH - извлекается аргумент и КОП второй инструкции, хранящейся в *ROM* по адресам **#2..#3**, а программный счетчик *PC* устанавливается в значение **#4**.

Происходит выполнение второй команды.

Далее снова FETCH и т. д.

До каких пор? Что произойдет, когда счетчик дойдет до максимального значения *PC* **= #F = 1111b**. Увеличьте это значение на единицу, получите ответ: **1111b + 1b = 10000b = #0**. Команды программы в микропроцессорах крутятся как барабан в револьвере рис. 4.5.

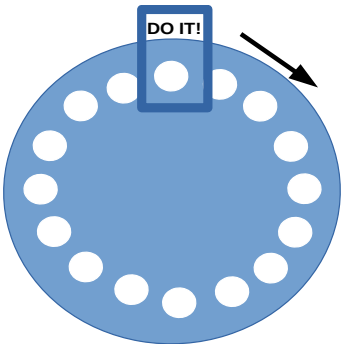


Рисунок 4.5 — Схематичное изображение выполнения программы в микропроцессоре, белые кружочки — инструкции программы.

На практике обычно реализуется бесконечный цикл — барабан прокручивается на несколько команд и возвращается в исходное положение.

Познакомимся теперь с устройством управления *УУ*, чтобы понять

Глава 4.4 Основные элементы устройства управления

С устройством управления *УУ* рис. 4.6 поступим также как с общей схемой *УИИ*. Сперва разберем функции отдельных элементов, а затем их работу в системе.

Первое, что бросается в глаза — шины. Это продолжение тех самых шин, изображенных на схеме *УИИ*. На шину можно смотреть, как на широкую дорогу, связывающую все устройства. По дороге перемещаются данные, адреса и управляющие сигналы.

4.4.1 Регистр команд

Микропрограмма FETCH извлекает из *ROM* очередную инструкцию программы. Данная инструкция записывается в **регистр команд** *RGC*.

На самом деле 8-битный *RGC* состоит из двух 4-х разрядных регистров *LOW* и *HIGH*. При поступлении фронта импульса на вход **CL** в регистр *LOW*(**L3..L0**) заносятся четыре младших бита команды (аргумент). При поступлении фронта импульса на вход **CH** в регистр *HIGH*(**H7..H4**) защелкиваются четыре старших бита — КОП. Вход **RST** сбрасывает все 8 битов в ноль.

Аргумент команды представляет собой либо данные, либо адрес, поэтому младшие разряды *RGC* **K3..K0** (аргумент) соединены с регистром данных *DAA* и регистром адреса *ADA*. Выходы **K7..K4** (КОП) направляются на старшие разряды памяти микрокоманд *ROM-MP*. КОП определяет адрес в *ROM-MP* начала микропрограммы, которая предназначена для выполнения текущей инструкции.

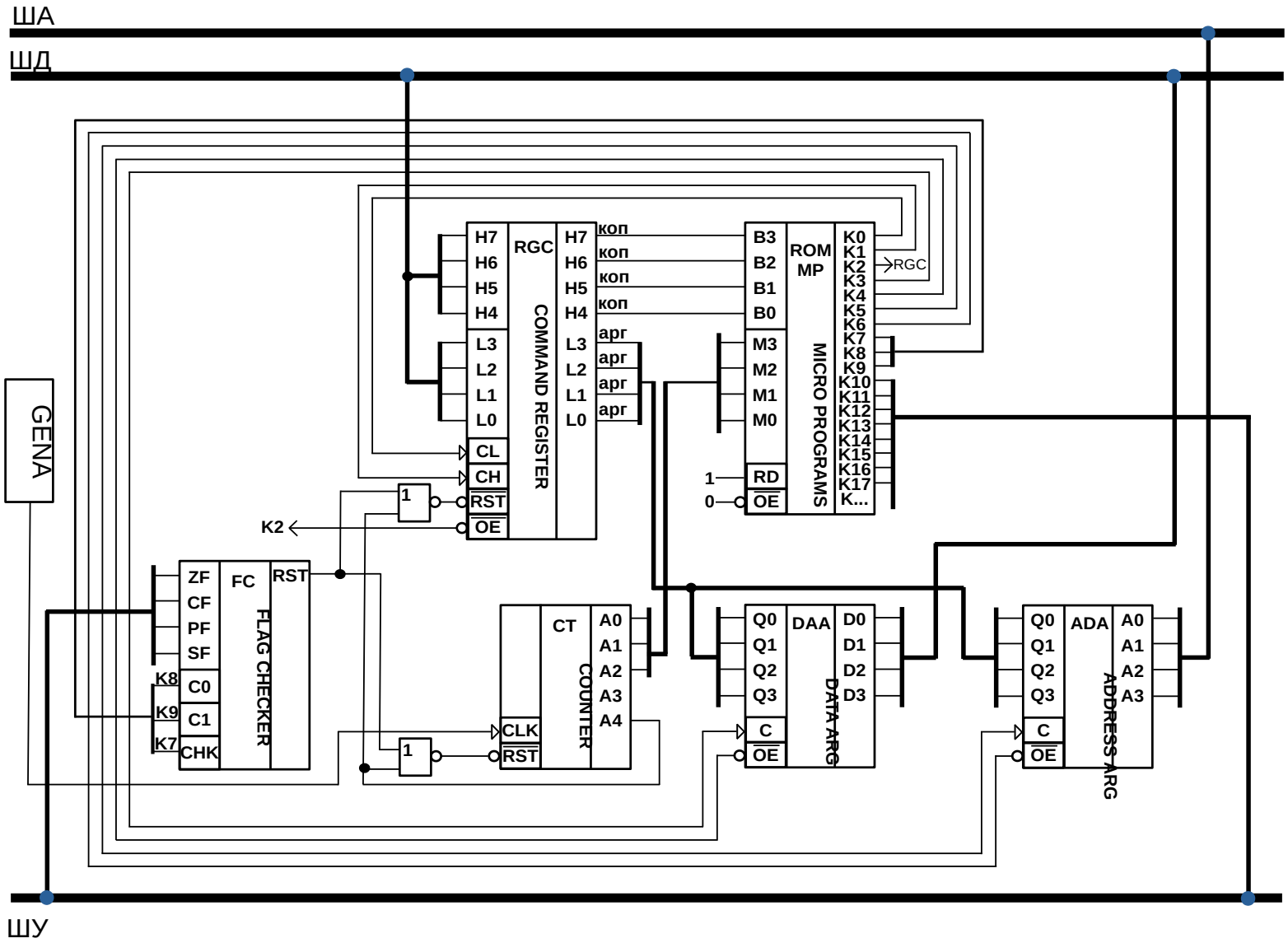


Рисунок 4.6 — Общая схема устройства управления УУ.

4.4.2 Счетчик и тактовый генератор

Счетчик *CT* синхронизирует работу всей системы. *CT* отсчитывает шестнадцать импульсов, поступающих от высокостабильного тактового генератора.

В армии, во времена пребывания на полигоне, источником электричества служил бензовый генератор. Называли его в народе GENA. В *МПА GENA* дает не просто электричество, а электрические прямоугольные импульсы с одинаковым периодом (временем следования, частотой).

Выходы счетчика **CT[A2.A0.A1]** соединены с младшими адресными входами памяти микрокоманд. Счетчик перебирает 8 микрокоманд микропрограммы FETCH, значения при этом изменяются в диапазоне **00000b..00111b**.

После восьмого импульса счетчик равен **01000b**. Три младших разряда счетчика снова отсчитывают 8 импульсов. За эти 8 тактов *МПА* выполнит микрокоманды извлеченной инструкции, хранящейся в *RGC*.

После шестнадцатого импульса счетчик равен **10000b** (**CT[A4] = 1**). К данному моменту выполнение инструкции завершено. Необходимо вновь вызвать FETCH, которая хранится в начале памяти. Соединим выход счетчика **CT[A4]** через инвертор с входом сброса счетчика **CT[RST]** и входом сброса регистра команд **RGC[RST]**. Таким образом, после 16-го такта на вход *ROM-MP* поступит адрес первой микрокоманды FETCH **0000**000b.

4.4.3 Регистр данных и адреса

Если аргумент инструкции является данными, то он сохраняется в регистре данных *DAA* (*DAtaArgument*). Выходы *DAA* подключены к шине данных *ШД*.

В регистре адреса *ADA* (*ADdressArgument)* сохраняется аргумент, если он является адресом. Выходы *ADA*, очевидно, подключены к шине адреса *ША*.

Первым программистом в мире была Ада Лавлейс, в честь которой назван язык программирования ADA.

Импульсы записи **C** и сигнал **OE** поступают от *ROM-MP*.

4.4.4 Обработчик флагов

Флаги дают микропроцессору возможность делать выбор. Обработчик флагов *FC* (*FlagChecker*) позволяет реализовать этот выбор в схеме.

FlagChecker — воплощение фантазии автора. В других системах он может быть реализован по-другому. Вообще, в реальных схемах многое по-другому. Напоминаю, что МПА логически непротиворечивое устройство, способное выполнять команды. Оно не обязано быть таким, как все. Целью служит не сделать правильно, а сделать, чтобы работало.

FC единственный элемент устройства управления *УУ*, на который данные (флаги) поступают по шине управления *ШУ*. Обычно, наоборот, сигналы от *УУ* направляются по *ШУ* на *УИИ*. Получается, что *ALU* управляет устройством управления. Действия, которые выполнит *УУ*, зависят от значения флагов.

Задача *FC* завершить дальнейшее исполнение команды, если соответствующий флаг равен нулю. Обработчик флагов запускается сигналом **CHK**. Выбор флага осуществляется через входы **C1..C0** (двух битов достаточно, чтобы перебрать четыре флага **00, 01, 10, 11**). Сигнал сброса поступает на выход **RST**, который соединен с входом сброса регистра команд **RGC[RST]** и счетчика **CT[RST]**. При необходимости, *FC* просто уничтожает команду из *RGC* и сбрасывает счетчик. Начинается извлечение следующей инструкции — выполнение FETCH. Обратите внимание, **C1..C0**, **CHK** опять-таки соединены с выходами *ROM-MP*.

4.4.5 Память микрокоманд

Добрались до загадочного *ROM-MP* — памяти микрокоманд.

Микропроцессор обладает некоторым набор инструкций, который он в состоянии выполнить. Служебные сигналы, поступающие на все устройства *МПА*, исходят от устройства управления *УУ*. Точнее, все управляющие сигналы появляются на выводах *ROM-MP* (в т.ч. «красные нолики или единички», которые фигурировали при подробном рассмотрении работы схемы *ALU*).

Память микрокоманд *ROM-MP* хранит управляющие последовательности (микропрограмму) для каждой инструкции, имеющейся в наборе у *МПА*.

Чтобы выполнить инструкцию программы, извлеченную из *ROM* в регистр команд *RGC*, *УУ* требуется совершить несколько микрокоманд (выполнить микропрограмму), по перемещению данных, переключению регистров, изменению программного счетчика и т.д.

Получается своеобразный фрактал. Если издалека смотреть на *УИИ*, то различимы инструкции программы в памяти *ROM*, программный счетчик *PC*, регистр *RGA* и регистр *RGR*. Если посмотреть на *УУ*, то опять видим память, но уже микропрограмм *ROM-MP*, опять счетчик *CT*, два регистра *DAA* и *ADA*. Подобно тому, как рисунок внутри зеленого листочка повторяет очертание дерева, структура устройства управления *УУ* повторяет структуру устройства исполнения *УИИ*.

Схема памяти микропрограмм *ROM-MP*, как и всякой памяти, включает адресные входы и выходы с данными. Младшие адресные разряды подключены к счетчику *CT*, а старшие к коду операции *RGC*.

4.4.6 Организация хранения микропрограмм в ROM-MP

Микропрограмма — это последовательность восьми микрокоманд, осуществляющих выполнение какой-либо инструкции из набора *МПА*. Каждая микрокоманда представляет достаточно длинное двоичное число. Ведь *ROM-MP* соединена с управляющими входами всех устройств *МПА*. Получилось насчитать порядка 34 таких входов. Соответственно микрокоманда — это 34-х разрядное двоичное число, каждая ячейка памяти *ROM-MP* состоит из 34 битов.

Обратим взор на странное соединение адресных входов, часть которых подключена к *счетчику*, а часть к *коду операции*. Такое подключение позволяет разбить память на блоки по 8 ячеек. Каждый блок хранит микропрограмму для определенного кода операции КОП (инструкции) рис. 4.7.

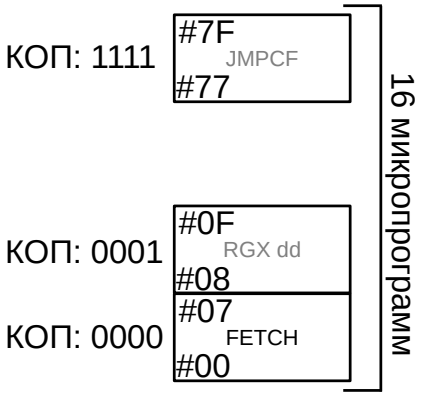


Рисунок 4.7 — Разделение памяти ROM-MP на блоки микропрограмм, в каждом блоке по 8-мь микрокоманд.

Счетчик, считая до 8-ми, последовательно перебирает 8 ячеек памяти. В какой именно области памяти производится перебор, определяется *кодом операции* инструкции. КОП указывает адрес первой микрокоманды, с которой начинается выполнение соответствующей инструкции. Начиная с этого адреса, счетчик перебирает 8 микрокоманд, тем самым, выполняется микропрограмма.

На рис. 4.8 изображена память *ROM-MP*. В правом столбце указаны микрокоманды. В левом столбце указаны адреса ячеек памяти. Зеленым цветом обозначены разряды адреса, на которые поступает КОП, синим — разряды адреса, на которые

поступает значение от счетчика. Если КОП = **0000**b, то счетчиком перебираются 8-мь микрокоманд FETCH. Если поступит инструкция сложения с кодом операции КОП = **1010**, счетчик переберёт 8-мь микрокоманд ADD, выполняющих операцию сложения.

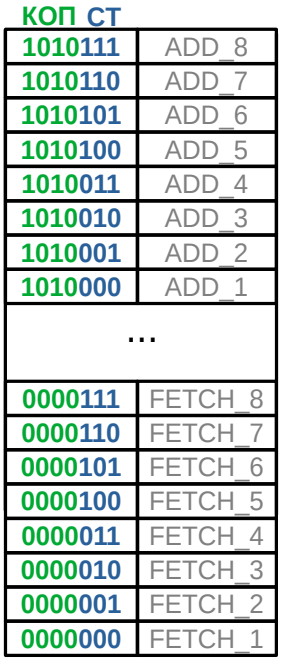


Рисунок 4.8 — Влияние кода операции КОП на выбор микропрограммы

В процессе выполнения текущей инструкции, может произойти запись какого-либо адреса в *PC*. Например, можно вернуться на несколько команд назад или перепрыгнуть вперед. Таким образом организуется ветвление и цикл.

4.4.7 FETCH

Разберем ещё более подробно, что происходит при включении **МПА**.

На старшие разряды *ROM-MP* от *RGC* поступает **0000b**, на младшие разряды от счетчика, также **000b**. С адреса **0000000b** хранятся 8 микрокоманд микропрограммы FETCH.

Разберем 8-микрокоманд, которые каждый такт сменяют друг друга на выходах **K33..K00**, заставляя *МПА* извлекать команду из *ROM УИИ* и заносить её в *RGC УУ*. Распишем побитно микрокоманды *ROM-MP* для тех самых тактов, которые рассматривались при анализе работы *УИИ*.

В заголовке таблицы указывается адрес, поступающий на ROM-MP, состоящий из КОП текущей инструкции программы и значения счетчика.

Последние изменения на шине управления *ШУ* будем изображать красным цветом. Остальные биты, принимающие непосредственное участие в выполнении команды, зеленым.

Такт #1 Выставление адреса инструкции на ША

`RGC[OE] = 1` — на время извлечения инструкции из ROM отключаем выходы регистра команд. В противном случае, КОП извлеченной команды поступит на входы *ROM-MP*, *МПА* начнет выполнять инструкцию преждевременно.

`PC[OE] = 0` — к *ША* подключается программный счетчик. Значение **`0000b`** оказывается на *ША*. Выходы остальных устройств отключены от шин (**`OE = 1`**).

ROM-MP[B3..M0]	0	0	0	0	0	0	0

K33	K32	K31	K30	K29	K28	K27	K26	K25	K24	K23	K22	K21	K20	K19	K18	K17	K16	K15	K14	K13	K12	K11	K10	K9	K8	K7	K6	K5	K4	K3	K2	K1	K0
PF	CF	ZF	PRT [OE]	PRT [C]	PRT [TUN]	ROM [WR]	ROM [RD]	RAM [WR]	RAM [RD]	SP [DEC]	SP [INC]	SP [OE]	SP [C]	RGX [OE]	RGX [C]	ALU [OE]	ALU [CR]	ALU [CA]	ALU [Cp]	ALU [C1]	ALU [C0]	PC [OE]	PC [C]	FC [C1]	FC [C0]	FC [CHK]	ADA [OE]	ADA [C]	DAA [OE]	DAA [C]	RGC[OE]	RGC [CH]	RGC [CL]
0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	**0**	0	0	0	0	1	0	1	0	**1**	0	0

PC				ША				ШД				RGC							
0	0	0	0	**0**	**0**	**0**	**0**	0	0	0	0	0	0	0	0	0	0	0	0

Такт #2 Выставление младших битов инструкции (аргумента) на ШД

`ROM[RD] = 1 -` аргумент команды (например, **`1101`**), хранящийся по адресу **`0000b`** поступает на *ШД*. При этом **`PC[OE] = 0`** остается.

ROM-MP[B3..M0]	0	0	0	0	0	0	1

K33	K32	K31	K30	K29	K28	K27	K26	K25	K24	K23	K22	K21	K20	K19	K18	K17	K16	K15	K14	K13	K12	K11	K10	K9	K8	K7	K6	K5	K4	K3	K2	K1	K0
PF	CF	ZF	PRT [OE]	PRT [C]	PRT [TUN]	ROM [WR]	ROM [RD]	RAM [WR]	RAM [RD]	SP [DEC]	SP [INC]	SP [OE]	SP [C]	RGX [OE]	RGX [C]	ALU [OE]	ALU [CR]	ALU [CA]	ALU [Cp]	ALU [C1]	ALU [C0]	PC [OE]	PC [C]	FC [C1]	FC [C0]	FC [CHK]	ADA [OE]	ADA [C]	DAA [OE]	DAA [C]	RGC[OE]	RGC [CH]	RGC [CL]
0	0	0	1	0	0	0	**1**	0	0	0	0	1	0	1	0	1	0	0	0	0	0	**0**	0	0	0	0	1	0	1	0	**1**	0	0

PC				ША				ШД				RGC							
0	0	0	0	0	0	0	0	**1**	**1**	**0**	**1**	0	0	0	0	0	0	0	0

Такт #3 Запись младших битов инструкции (аргумента) в RGC

`RGC[CL] = 1 -` возникающий фронт импульса, инициирует запись младших битов в *RGC*. Поскольку данные 4-х разрядные, спервa из *ROM* извлекаются и сохраняются младшие биты команды, которые соответствуют аргументу.

ROM-MP[B3..M0]	0	0	0	0	0	1	0

K33	K32	K31	K30	K29	K28	K27	K26	K25	K24	K23	K22	K21	K20	K19	K18	K17	K16	K15	K14	K13	K12	K11	K10	K9	K8	K7	K6	K5	K4	K3	K2	K1	K0
PF	CF	ZF	PRT [OE]	PRT [C]	PRT [TUN]	ROM [WR]	ROM [RD]	RAM [WR]	RAM [RD]	SP [DEC]	SP [INC]	SP [OE]	SP [C]	RGX [OE]	RGX [C]	ALU [OE]	ALU [CR]	ALU [CA]	ALU [Cp]	ALU [C1]	ALU [C0]	PC [OE]	PC [C]	FC [C1]	FC [C0]	FC [CHK]	ADA [OE]	ADA [C]	DAA [OE]	DAA [C]	RGC[OE]	RGC [CH]	RGC [CL]
0	0	0	1	0	0	0	**1**	0	0	0	0	1	0	1	0	1	0	0	0	0	0	**0**	0	0	0	0	1	0	1	0	**1**	0	**1**

PC				ША				ШД				RGC							
0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	**1**	**1**	**0**	**1**

Такт #4 Увеличение PC. Выставление старших битов инструкции (КОП) на ШД.

`PC[C] = 1` — фронт провоцирует срабатывание счетчика, значение *PC* увеличивается на единицу **`0000b + 0001b = 0001b`**. Адрес **`0001b`** поступает на *ША*. При этом **`PC[OE] = 0`** остается.

`ROM[RD] = 1` — память программ *ROM*, по-прежнему, работает на чтение, на *ШД* поступает вторая часть команды — код операции, например, **`0001b`**.

`RGC[CL] = 0` — сброс импульса записи, регистр команд *RGC* готов к следующему сигналу записи

ROM-MP[B3..M0]	0	0	0	0	0	1	1

K33	K32	K31	K30	K29	K28	K27	K26	K25	K24	K23	K22	K21	K20	K19	K18	K17	K16	K15	K14	K13	K12	K11	K10	K9	K8	K7	K6	K5	K4	K3	K2	K1	K0
PF	CF	ZF	PRT [OE]	PRT [C]	PRT [TUN]	ROM [WR]	ROM [RD]	RAM [WR]	RAM [RD]	SP [DEC]	SP [INC]	SP [OE]	SP [C]	RGX [OE]	RGX [C]	ALU [OE]	ALU [CR]	ALU [CA]	ALU [Cp]	ALU [C1]	ALU [C0]	PC [OE]	PC [C]	FC [C1]	FC [C0]	FC [CHK]	ADA [OE]	ADA [C]	DAA [OE]	DAA [C]	RGC[OE]	RGC [CH]	RGC [CL]
0	0	0	1	0	0	0	**1**	0	0	0	0	1	0	1	0	1	0	0	0	0	0	**0**	**1**	0	0	0	1	0	1	0	**1**	0	**0**

PC				ША				ШД				RGC							
0	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	0	0	0	0	1	1	0	1

Как много всего произошло за один такт!

Такт #5 Запись старших битов инструкции (КОП) в RGC

`RGC[CH] = 1 –` возникающий фронт импульса, инициирует запись старших битов с *ШД* в *RGC*. Теперь команда извлечена полностью.

`PC[C] = 0` — счетчик снова готов поддаться фронту импульса.

ROM-MP[B3..M0]	0	0	0	0	1	0	0	

K33	K32	K31	K30	K29	K28	K27	K26	K25	K24	K23	K22	K21	K20	K19	K18	K17	K16	K15	K14	K13	K12	K11	K10	K9	K8	K7	K6	K5	K4	K3	K2	K1	K0
PF	CF	ZF	PRT [OE]	PRT [C]	PRT [TUN]	ROM [WR]	ROM [RD]	RAM [WR]	RAM [RD]	SP [DEC]	SP [INC]	SP [OE]	SP [C]	RGX [OE]	RGX [C]	ALU [OE]	ALU [CR]	ALU [CA]	ALU [Cp]	ALU [C1]	ALU [C0]	PC [OE]	PC [C]	FC [C1]	FC [C0]	FC [CHK]	ADA [OE]	ADA [C]	DAA [OE]	DAA [C]	RGC[OE]	RGC [CH]	RGC [CL]
0	0	0	1	0	0	0	**1**	0	0	0	0	1	0	1	0	1	0	0	0	0	0	**0**	**0**	0	0	0	1	0	1	0	**1**	**1**	**0**

PC				ША				ШД				RGC							
0	0	0	1	0	0	0	**1**	0	0	0	1	**0**	**0**	**0**	**1**	1	1	0	1

Такт #6 Отключение PC от ША. Отключение ROM от ШД. Увеличение PC.

PC[OE] = 1 и **ROM[RD] = 0** – *ША* и *ШД* освобождаются для работы с другими устройствами.

PC[C] = 1 - фронт провоцирует срабатывание счетчика **0001b + 0001b = 0010b**. В программном счетчике находится адрес ячейки *ROM*, хранящей следующую команду *PC* **= 0010b**.

RGC[CH] = 0 – сброс импульса записи, регистр команд *RGC* готов к следующему сигналу записи.

ROM-MP[B3..M0]	0	0	0	0	1	0	1

K33	K32	K31	K30	K29	K28	K27	K26	K25	K24	K23	K22	K21	K20	K19	K18	K17	K16	K15	K14	K13	K12	K11	K10	K9	K8	K7	K6	K5	K4	K3	K2	K1	K0
PF	CF	ZF	PRT [OE]	PRT [C]	PRT [TUN]	ROM [WR]	ROM [RD]	RAM [WR]	RAM [RD]	SP [DEC]	SP [INC]	SP [OE]	SP [C]	RGX [OE]	RGX [C]	ALU [OE]	ALU [CR]	ALU [CA]	ALU [Cp]	ALU [C1]	ALU [C0]	PC [OE]	PC [C]	FC [C1]	FC [C0]	FC [CHK]	ADA [OE]	ADA [C]	DAA [OE]	DAA [C]	RGC[OE]	RGC [CH]	RGC [CL]
0	0	0	1	0	0	0	**0**	0	0	0	0	1	0	1	0	1	0	0	0	0	0	**1**	**1**	0	0	0	1	0	1	0	**1**	**0**	**0**

PC				ША				ШД				RGC							
0	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	1	1	0	1

Такт #7

`PC[C] = 0` — сброс импульса записи, программный счетчик вновь готов считать.

ROM-MP[B3..M0]	0	0	0	0	1	1	0

K33	K32	K31	K30	K29	K28	K27	K26	K25	K24	K23	K22	K21	K20	K19	K18	K17	K16	K15	K14	K13	K12	K11	K10	K9	K8	K7	K6	K5	K4	K3	K2	K1	K0
PF	CF	ZF	PRT [OE]	PRT [C]	PRT [TUN]	ROM [WR]	ROM [RD]	RAM [WR]	RAM [RD]	SP [DEC]	SP [INC]	SP [OE]	SP [C]	RGX [OE]	RGX [C]	ALU [OE]	ALU [CR]	ALU [CA]	ALU [Cp]	ALU [C1]	ALU [C0]	PC [OE]	PC [C]	FC [C1]	FC [C0]	FC [CHK]	ADA [OE]	ADA [C]	DAA [OE]	DAA [C]	RGC[OE]	RGC [CH]	RGC [CL]
0	0	0	1	0	0	0	**0**	0	0	0	0	1	0	1	0	1	0	0	0	0	0	**1**	**0**	0	0	0	1	0	1	0	**1**	**0**	**0**

PC				ША				ШД				RGC							
0	0	1	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	1	1	0	1

Такт #8

RGC[OE] = 0 - КОП = **0001b** из *RGC* поступает на входы *ROM-MP*. Значение счетчика в данный момент составляет *CT* **= 0111b**. Таким образом, на адресных входах *ROM-MP* оказывается значение **0001111b**.

ROM-MP[B3..M0]	0	0	0	0	1	1	1

K33	K32	K31	K30	K29	K28	K27	K26	K25	K24	K23	K22	K21	K20	K19	K18	K17	K16	K15	K14	K13	K12	K11	K10	K9	K8	K7	K6	K5	K4	K3	K2	K1	K0
PF	CF	ZF	PRT [OE]	PRT [C]	PRT [TUN]	ROM [WR]	ROM [RD]	RAM [WR]	RAM [RD]	SP [DEC]	SP [INC]	SP [OE]	SP [C]	RGX [OE]	RGX [C]	ALU [OE]	ALU [CR]	ALU [CA]	ALU [Cp]	ALU [C1]	ALU [C0]	PC [OE]	PC [C]	FC [C1]	FC [C0]	FC [CHK]	ADA [OE]	ADA [C]	DAA [OE]	DAA [C]	RGC[OE]	RGC [CH]	RGC [CL]
0	0	0	1	0	0	0	**0**	0	0	0	0	1	0	1	0	1	0	0	0	0	0	**1**	**0**	0	0	0	1	0	1	0	**0**	**0**	**0**

PC				ША				ШД				RGC							
0	0	1	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	1	1	0	1

Данный адрес соответствует *последней* микрокоманде микропрограммы, выполняющей инструкцию с КОП = **0001b**. Договоримся, что *последняя* микрокоманда каждой микропрограммы не выполняет никаких действий.

ROM-MP[B3..M0]	0	0	0	1	1	1	1

K33	K32	K31	K30	K29	K28	K27	K26	K25	K24	K23	K22	K21	K20	K19	K18	K17	K16	K15	K14	K13	K12	K11	K10	K9	K8	K7	K6	K5	K4	K3	K2	K1	K0
PF	CF	ZF	PRT [OE]	PRT [C]	PRT [TUN]	ROM [WR]	ROM [RD]	RAM [WR]	RAM [RD]	SP [DEC]	SP [INC]	SP [OE]	SP [C]	RGX [OE]	RGX [C]	ALU [OE]	ALU [CR]	ALU [CA]	ALU [Cp]	ALU [C1]	ALU [C0]	PC [OE]	PC [C]	FC [C1]	FC [C0]	FC [CHK]	ADA [OE]	ADA [C]	DAA [OE]	DAA [C]	RGC[OE]	RGC [CH]	RGC [CL]
0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0	**0**	0	0

PC				ША				ШД				RGC							
0	0	1	0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**	1	1	0	1

Здесь никаких действий не произошло.

4.4.8 Выполнение извлеченной команды

Пусть инструкция с кодом операции **0001b**, которая была извлечена в *RGC*, осуществляет запись аргумента в регистр *RGX*. Параграф в книге сменился, а счетчик продолжает считать такты.

Такт #9 Запись аргумента в регистр данных DAA или регистр адреса ADA

ROM-MP[B3..M0] = 0001000b. Значение счетчика равно **01000b**. Разряд счетчика **CT[A3] = 1** не соединен ни с одним из устройств. Его наличие позволяет осуществить 8 тактов для выполнения микрокоманд инструкции из *RGC*, перед тем как осуществить сброс *RGC* (при **CT[A4] = 1**). На вход *ROM-MP* поступает значение **0001000b**.

DAA[C] = 1 - первая микрокоманда инструкции **0001ddddb,** фронт импульса осуществляет запись младших битов **RGC[L4..L0]** в регистр данных *DAA*.

В других микропроцессорных системах инструкции бывают однобайтовыми, двухбайтовыми и т.д. Инструкция перед выполнением пропускается через специальное устройство — дешифратор команд. Дешифратор команд вычленяет из инструкции КОП, исходя из КОП понимает сколько аргументов имеется у команды. Заносит аргументы в регистр данных или регистр адреса. Наша МПА сильно упрощена. Функции дешифратора команд выполняется первой микрокомандой инструкции. Да и то не всегда. Большинство инструкций МПА не имеют аргументов вовсе.

Такт #10 Выставление аргумента на ШД (либо ША)

ROM-MP[B3..M0] = 0001001b.

DAA[OE] = 0 – данные из регистра *DAA* выставляются на *ШД*.

DAA[C] = 0 – сброс импульса записи.

Такт #11 Запись данных в RGX

ROM-MP[B3..M0] = 0001010b.

RGX[C]=1 - данные, поступившие из регистра данных *DAA* на *ШД,* заносятся в универсальный регистр *RGX:*.

Такт #12 Отключаем DAA от ШД

ROM-MP[B3..M0] = 0001011b.

DAA[OE] = 1 – отключение *DAA* от шины данных**.**

RGX[C] = 0 – сбрас импульса записи**.**

Такт #13 Not Operation

ROM-MP[B3..M0] = 0001100b. NOP — not operation (ничего не происходит)

Такт #14 Not Operation

ROM-MP[B3..M0] = 0001101b. NOP

Такт #15 Not Operation

ROM-MP[B3..M0] = 0001110b. NOP

Такт #16 Not Operation

ROM-MP[B3..M0] = 0001111b. **Последняя** микрокоманда **в МПА всегда** NOP.

Такт #17

Значение счетчика равно *CT* = **10000b**. Старший разряд **CT[A4] = 1** через инвертор подсоединен к входу сброса *RGC* и *CT*.

RGC[RST] = 0 и **CT[RST] = 0** — счетчик и регистр команд сбрасываются в ноль.

На входах *MP-ROM* оказывается адрес первой микрокоманды инструкции FETCH **ROM-MP[B3..M0] = 0000000b**. Происходит извлечение из *ROM* следующей инструкции, адрес которой давно ждет в программном счетчике *PC* = **0010b**.

Рассмотренные выше 16 тактов повторяются.

Таким образом, работает МПА.

Раздел 5. ASSEMBLER МПА

Пройден довольно длинный и интересный путь. На основе закона притяжения противоположностей было рассмотрено строение полупроводниковых материалов и структура *p-n* перехода, на базе *p-n* перехода создан транзистор. Из транзисторов был сконструирован базисный логический элемент.

На основе базисного логического элемента, были собраны другие основные логические операторы: конъюнкция, дизъюнкция, инверсия, *XOR*. Из этих элементов был сконструирован сумматор — устройство, осуществляющее основную операцию, выполняемую микропроцессором.

Также были рассмотрены элементы секвенциальной (последовательностной) логики — триггеры. Триггер — элемент, позволяющий хранить минимально возможное количество информации — 1 бит. Триггеры являются составной частью регистров — ячеек памяти, способных хранить несколько бит (многоразрядное двоичное число).

Из сумматора, элементов, реализующих логические операции, и двух регистров было сконструировано арифметико-логическое устройство. *ALU* — сердце компьютера, выполняет все вычисления, которые могут быть произведены микропроцессором. Назначение одного из регистров *RGA* — хранение первого операнда, в другой регистр *RGR* записывается результат работы *ALU*.

Устройство исполнения *УИИ* возникло присоединением к *ALU* памяти программ *ROM*, программного счетчика *PC* и оперативной памяти *RAM*. Соединение осуществляется посредство шины данных и шины адреса. *УИИ* позволяет выбирать и исполнять инструкции из *ROM*. Управляющие входы всех устройств *УИИ* соединены с устройством управления *УУ* через шину управления. *УУ* содержит память микропрограмм с микрокомандами, которые по шине управления поступают

на управляющие входы всех устройств *УИИ*. Микрокоманды представляют собой управляющие сигналы. *УУ* управляет работой *УИИ*.

Таким образом, была сконструирована микропроцессорная система, которая получила название микропроцессорная авантюра *МПА*.

В итоге было исследовано, что происходит с *МПА* при включении (подаче питания). Подробно рассмотрено выполнение команды FETCH, извлекающей очередную инструкцию из *ROM*. После выполнения извлеченной инструкции снова выполняется команда FETCH, которая извлекает следующую инструкцию. Таким образом, последовательно выполняются все инструкции программы в режиме револьвера, разумеется, если программа не содержит инструкцию перескока к какой-либо команде.

К данному моменту были разобраны самые низкие уровни *МПА*: физический и логический. Как происходит выполнение инструкций, в какую сторону и из каких мест летят **0** и **1** было подробно рассмотрено. Теперь осуществим переход на более высокий уровень абстракции. От схем с летающими **0** и **1**, перейдем к языку ASSEMBLER. Сперва придумаем для *МПА* набор команд, которые она способна выполнить. Далее сконструируем несколько практических устройств.

Важное дополнение

Разобравшись и абстрагировавшись от схемы *МПА*, мы вправе увеличить разрядность всех устройств и шин до 8-ми бит. Схема останется той же, просто количество входов и выходов у каждого устройства увеличится в два раза.

Таким образом, количество ячеек в памяти увеличивается с $2^4 = 16$ до $2^8 = 256$. Адреса ячеек имеют значения в диапазоне **#00..#FF**. Каждая ячейка способна хранить 8-битное число от **0** до **255**.

Инструкция теперь занимает два байта. Старший байт (8 бит) инструкции хранит код операции. Вместо 16 становится возможным закодировать 256 инструкций. Младший байт (8 бит) инструкции хранит аргумент.

Глава 5.1 ASSEMBLER с первого взгляда

Посмотрим, какое впечатление ASSEMBLER произведет на читателя с первого взгляда.

Вспомним, что программа состоит из последовательности команд, называемых *инструкциями*. Каждая инструкция выглядит следующим образом.

коп7 коп6 коп5 коп4 коп3 коп2 коп1 коп0 арг7 арг6 арг5 арг4 арг3 арг2 арг1 арг0

Младшие 8-мь разрядов **арг** представляют собой данные, старшие 8-мь битов **коп** определяют код операции. Код операции — это действие, которое будет произведено над данными.

Примерно, программа на языке ASSEMBLER выглядит следующим образом:

0000100000000001000000000000010000011000000000100000000000000100100000000000011. Мда.. тяжеловато, однако, программировать на ASSEMBLER, заметит читатель. Что ж, запишем инструкции в столбик:

```
00001000 00000001

00000000 00000010

00001100 00000001

00000000 00001001

00000000 00000011
```

Уже лучше, но в глазах по-прежнему возникает рябь.

Хочется чего-то большего, например, букв. Хорошо, хотели буквами, получите:

```
RGX  #08
SETA
RGX  #0C
ADD
GETR
```

Час от часу не легче. Давайте разбираться.

Каждая строка в наборе **`0`** и **`1`**, записанных в столбик выше, представляет собой отдельную инструкцию, хранящуюся в памяти *ROM*. Поскольку ячейка *ROM* хранит только 8-ми битное число, каждая инструкция занимает две ячейки памяти. Т.е. программа занимает не 5 ячеек, как могло показаться на первый взгляд, а десять, начиная с адреса **`#00`**:

#00 00001000 аргумент 1-й инструкции (число #08)

#01 00000001 КОП 1-й инструкции

#02 00000000 аргумент 2-й инструкции

#03 00000011 КОП 2-й инструкции

#04 00001100 аргумент 3-й инструкции (число #0C)

#05 00000001 КОП 3-й инструкции

#06 00000000 аргумент 4-й инструкции

#07 00001000 КОП 4-й инструкции

#08 00000000 аргумент 5-й инструкции

#09 00000010 КОП 5-й инструкции

С целью облегчить написание программы, люди придумали вместо двоичных чисел записывать буквы, они как бы дали инструкциям человеческие имена — *мнемоники*. **Мнемоника** — символьное обозначение инструкции. Например, первая инструкция **`0000100000000001`** записывается мнемоникой `RGX #08`.

Между *двоичным* и «*мнемоничным*» кодом разница отсутствует, но присутствует однозначное соответствие. Это одно и тоже, в разных обозначениях. Замена **`0`** и **`1`** на *мнемоники*, не облегчает процесс программирования. Сокращается количество ошибок, связанных с рябью в глазах, при записи и чтении программистом (человеком) инструкций **`0000100000000001`**.

Чтобы программировать на ассемблере, необходимо думать, как машина. Машина думает достаточно просто, следовательно, программировать на ассемблере легко. В этом основное отличие низкоуровневого от высокоуровневого программирования — простота и настройка человеческого ума на своеобразное мышление машины.

Глава 5.2 Набор команд

Придумаем инструкции для *МПА*, из которых будут составляться программы. Поскольку для кода операции отводится **8** бит, то возможно наличие 2^8 **= 256** инструкций. Инструкция с КОП **#00** зарезервирована под команду FETCH. Остается **255**, вполне достаточно, ещё и останется.

Ради интереса попробуем обойтись минимально возможным количеством инструкций. По крайней мере, чем тупее машина, тем легче ее отпрограммировать. Будем придерживаться следующей *идеологии*: *все операции перемещения данных осуществляются через универсальный регистр* RGX. *Идеология* позволит сэкономить на количестве инструкций в наборе *МПА*, однако, программа получится более длинной.

Приступим к составлению полного набора команд, группируя инструкции по назначению. Байты инструкций будем записывать в 16-ном виде. Код операции и аргумент будут составлять по две цифры. В записи инструкции **#A5F3** кодом операции является **#A5 = 10100101b**, а аргументом **#F3 = 11110011b**.

5.2.1 Инструкции записи данных

В *идеологии* данные могут быть записаны непосредственно только в универсальный регистр RGX. Поэтому в данной группе будет только одна инструкция. Шестнадцатеричный код инструкции выглядит следующим образом **#01dd**, где **01** — код операции, **dd** — данные (число), которые будут помещены в универсальный регистр *RGX*. Придумаем для команды **#01dd** мнемонику, например, RGX #dd. Если захочется писать в коде двоичные числа, нет проблем, просто ставьте символ **b** после двоичного числа RGX ddddddddb.

5.2.2 Инструкции копирования данных

Самая многочисленная группа команд. Копирование будет происходить либо *ИЗ* либо *В* универсальный регистр *RGX*. Вспомним, в какие элементы *МПА* можно записать данные. *И*х всего три: *ALU, RAM, PRT*.

У *ALU* имеется два регистра. Для регистра *RGA*, предназначенного хранить первый операнд, определим инструкцию копирования числа из *RGX* в *RGA* **#0200** с мнемоникой SETA. Также определим инструкцию копирования из регистра результата *RGR* в универсальный регистр *RGX*, пусть это будет **#0300** с мнемоникой GETR.

Обратите внимание, что у SETA и GETR отсутствует аргумент. Чего не скажешь о инструкциях для работы с оперативной памятью *RAM*. В качестве аргумента выступает адрес ячейки, с которой происходит запись или чтение. Пусть код **#04aa** с мнемоникой WR #aa осуществляет запись данных из *RGX* в ячейку *RAM* с адресом **#aa**. В то время как инструкция, **#05aa** с мнемоникой RD #aa осуществляет чтение данных из ячейки *RAM* с адресом **#aa** в универсальный регистр *RGX*.

В соответствии с идеологией, запись числа **#E0** в ячейку RAM с адресом **#5D** осуществится вызовом двух инструкций. Сперва значение заносится в универсальный регистр RGX #E0, затем оно копируется по необходимому адресу (**#5D**) в оперативную память WR #5D.

Инструкции записи в порт **#0600** (PRTOUT) и чтения **#0700** (PRTIN) из порта будут без аргументов. В данных командах значение копируется из универсального регистра *RGX* в порт *PRT* и наоборот.

5.2.3 Инструкция настройки порта

Для настройки порта сперва необходимо записать настроечное слово в универсальный регистр *RGX*. Напоминаем, что *настроечное слово* представляет собой двоичное число. Каждый разряд двоичного числа определяет в какую сторону будет работать соответствующий вывод порта. Значение **1** в разряде означает настройку на вывод, **0** — на ввод.

Рассмотрим слово **#F0 = 11110000b**. Младшие 4-ре разряда порта настраиваются на ввод, к ним можно подключить кнопки, например. Старшие 4-ре разряды настраиваются на вывод, к ним можно подключить лампочки, дисплей, мотор и любое другое устройство вывода. Команда **#0800** с мнемоникой TUNE производит копирование *настроечного слова* из *RGX* в регистр настройки порта.

5.2.4 Инструкции АЛО

Пусть инструкция **#0900** с мнемоникой ADD осуществляет сложение чисел, хранящихся в данный момент в регистрах *RGA* и *RGX*, результат сложения помещается в регистр арифметико-логического устройства *RGR*. Поразрядная конъюнкция **#0A00** AND, дизъюнкция **#0B00** OR и исключающее *ИЛИ* **#0C00** XOR выполняются аналогично.

5.2.5 Инструкции переходов

Позволяют перескочить с одного места программы в другое. Реализуется такое поведение записью в программный счетчик адреса инструкции, к которой необходимо перескочить. Переход бывает безусловный и условный.

В первом случае перескок происходит всегда. Для **безусловного перехода**, осуществляющему перескок по адресу **#aa**, определим инструкцию **#0D00** с мнемоникой JMP #aa. Адрес **#aa** — это адрес ячейки *ROM*, хранящей команду, к которой происходит переход. Команда JMP осуществляет запись **#aa** в программный счетчик *PC*.

Ни рис. 5.1 представлены ячейки *ROM*, в которые записаны некоторые инструкции, и значение, хранимое в программном счетчике в начале выполнения текущей инструкции (до команды FETCH). Команда `JMP #12`, записанная в ячейке с адресом **#08**, изменяет значение в программном счетчике. После **#08** инструкции выполнится инструкция, хранящаяся по адресу **#12**.

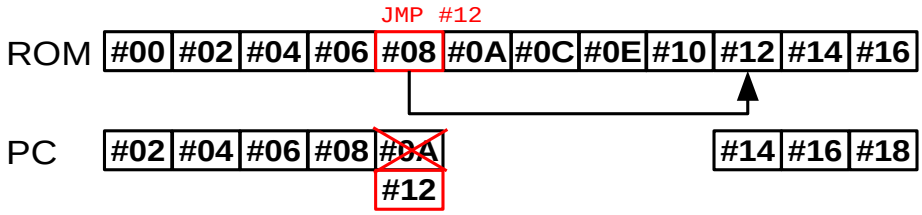


Рисунок 5.1 — Схема безусловного перехода

К другому виду «перескока» относятся команды **условного перехода.** Они осуществляют перескок по указанному адресу, только в том случае, если выполнено некоторое условие. Если условие не выполнено, перескока не возникает, выполнение переходит к следующей по порядку инструкции.

Условный переход, происходящий при выполнении условия **ZF = 1,** определим инструкцией **#0E00** с мнемоникой `JMPZF #aa`. Если **ZF = 1,** команда JMPZF изменит значение программного счетчика (рис. 5.2) и программа перепрыгнет через инструкции **#0A..#10**. Выполнение указанных инструкций (выделены зеленым цветом) произойдет только в случае, когда **ZF = 0**.

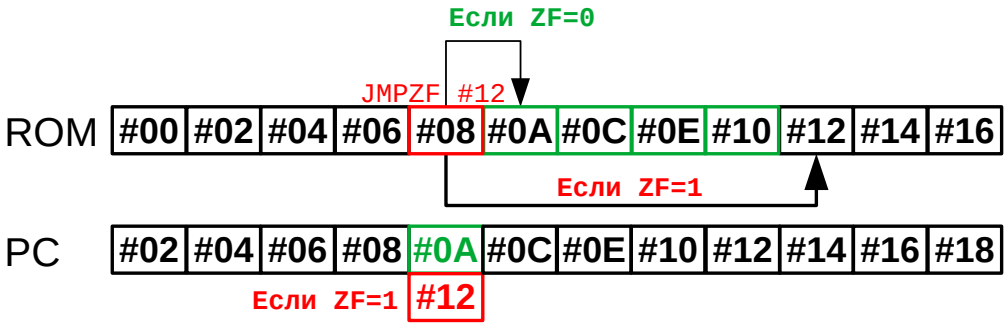


Рисунок 5.2 - условный переход

Зададим аналогичные инструкции условных переходов:

#0F00 JMPCF #aa если в результате вычислений *ALU* возникло переполнение,

#1000 JMPPF #aa если в результате вычислений *ALU* количество единиц оказалось четным или все нули

5.2.6 Инструкции работы со стеком

Напомним, что стеком — называется структура данных по типу «стопка тарелок», либо LIFO (Last In First Out), что переводится как «последний зашел, первый ушел». Физически стек располагается в оперативной памяти *RAM*. Программист сам решает в какой области *RAM* располагается область стека. Чтобы задать эту область, он записывает в регистр *указатель стека SP* адрес ячейки, являющейся *вершиной* (началом) *стека*. Обычно за вершину стека принимают адрес последней ячейки *RAM*. В таком случае, вероятность, что стек доберется до области *RAM*, в которой могут храниться важные данные и уничтожит их, снижается.

Определим инструкцию **#11aa** с мнемоникой SP #aa, записывающую в указатель стека *SP* адрес вершины стека **#aa**.

Обычно стек используется при реализации вызова подпрограммы. Представим, что в ходе выполнения программы в регистре *RGX* хранится *важное значение*. Вызывается подпрограмма, которая активно использует *RGX* в своих целях, в итоге *важное значение* перезаписывается другим числом. Чтобы после вызова подпрограммы *важное значение* в регистре не изменилось следует перед вызовом подпрограммы сохранить *важное значение* в оперативной памяти *RAM*, а после вызова восстановить из *RAM*. Собственно эту функцию и выполняет *стек*.

Добавим две инструкции. Первая из них **#1200** с мнемоникой PUSH заносит данные из *RGX* по адресу, указанному в *SP*, после чего уменьшает значение в *SP* на единицу. Вторая инструкция **#1300** с мнемоникой POP увеличивает *SP* на единицу и копирует значение по адресу *SP* из *RAM* в *RGX*.

Польза стека особо ощутима, когда в микропроцессоре имеется несколько универсальных регистров. В начале подпрограммы все регистры, которые используются подпрограммой, заталкиваются PUSH в стек. После значения извлекаются POP из стека обратно в регистры рис. 5.3

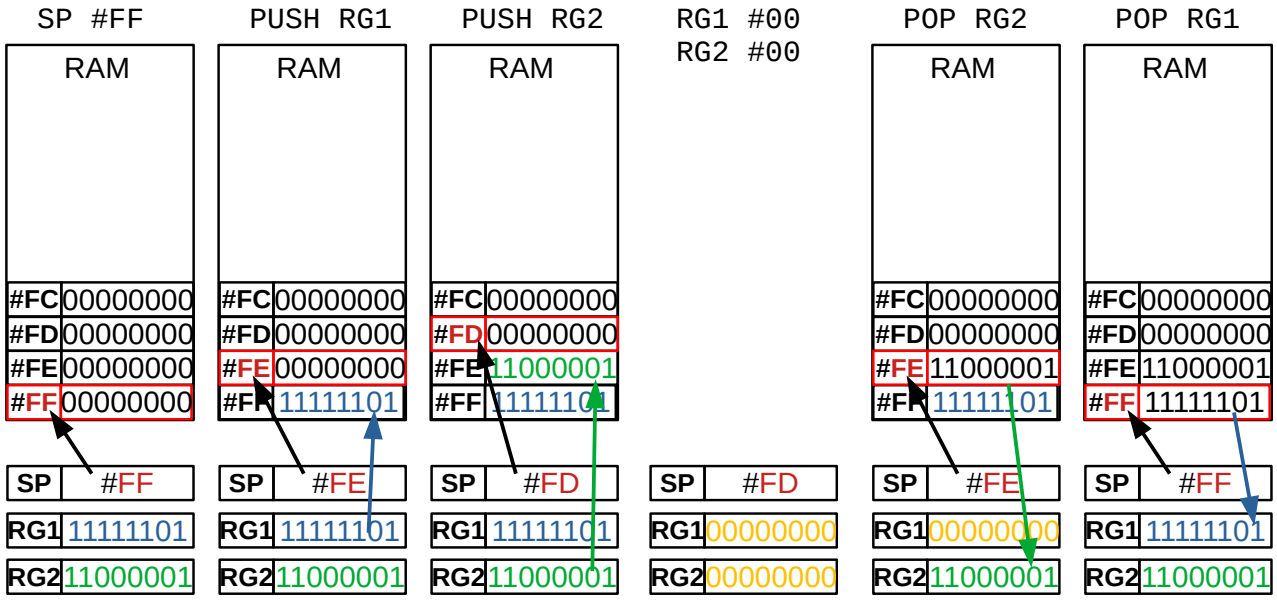


Рисунок 5.3 — Принцип работы стека.

Обратите внимание, что извлечение регистров происходит строго в обратном порядке их заталкиванию.

```
PUSH RG1
PUSH RG2
PUSH RG3
;  начало подпрограммы
….
;  конец подпрограммы
POP RG3
POP RG2
POP GR1
```

5.2.7 Инструкции вызова подпрограммы

Подпрограмма представляет собой набор инструкций, выполняющих определенное действие. Чтобы выполнить подпрограмму, необходимо в программе произвести ее вызов (позвать подпрограмму на помощь). Предполагается что подпрограмма может вызываться несколько раз из разных мест программы.

Для вызова подпрограммы, начинающейся с адреса **#aa**, определим инструкцию **#14aa** с мнемоникой CALL #aa. Инструкция CALL выполняет два действия:

1. Копирует значение программного счетчика *PC* в стек (*RAM{#SP}* << *CP*). По данному адресу будет осуществлен возврат в программу после выполнения подпрограммы. Уменьшает на единицу указатель стека. Собственно, *SP* всегда уменьшается на **1**, после помещения в него данных.

2. Заносит в программный счетчик адрес первой инструкции подпрограммы (*PC* **<< #aa**).

После завершения подпрограммы управление должно вернуться в тоже самое место программы, откуда подпрограмма была вызвана. Для возврата в программу определим инструкцию **#1500** с простой мнемоникой RET.

Инструкция возврата из подпрограммы RET увеличивает *SP* на **1** (*SP* всегда увеличивается на **1**, перед извлечением данных). Копирует значение из стека в программный счетчик (*PC* << *RAM{#SP}*). Напомним, что в стеке как раз хранится адрес инструкции основной программы, которая следует после инструкции CALL.

Разберемся подробнее как осуществляется вызов, выполнение и возврат из подпрограммы. На схеме рис. 5.4 изображено несколько столбцов. В каждом столбце показано состояние всех регистров и оперативной памяти на очередном шаге выполнения программы. Шаги перечислены в 1-й строке. Инструкции и значения, имеющие отношение к основной программе изображены синим цветом, к подпрограмме — зеленым.

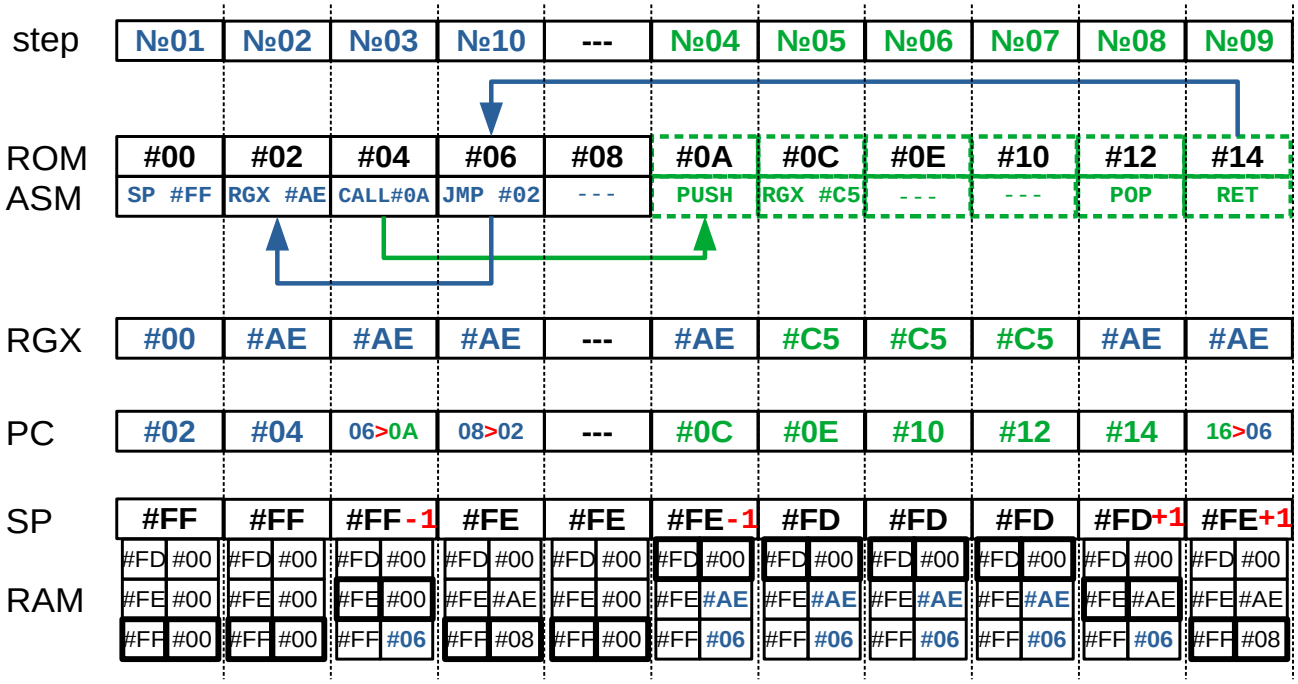


Рисунок 5.4 — Выполнение программы с вызовом подпрограммы

Рассмотрим внимательно каждый шаг, их всего-то десять:

№ 01. В *ROM* по адресу **#00..01** хранится инструкция SP #FF, которая заносит в *SP* значение вершины стека **#FF**. Ячейка на которую указывает *SP* выделена для наглядности более жирным прямоугольником. Программный счетчик указывает на следующую инструкцию **#02**.

№ 02. RGX #AE — значение **#AE** заносится в *RGX*, *PC* указывает на следующую инструкцию **#04**

№ 03. `CALL #0A` — вызов подпрограммы по адресу **#0A**. После команды FETCH значение программного счетчика указывает на адрес *ROM*, по которому хранится следующая инструкция *PC* = **#06**. Данное значение заносится в стек (*RAM*) по адресу **#FF**, указатель стека *SP* уменьшается на единицу *SP* = **#FE**. Аргумент команды `CALL #0A` записывается в программный счетчик *PC* = **#0A**.

№ 04. Обратите внимание, в схеме *шаг № 04* располагается правее, происходит выполнение инструкции PUSH по адресу **ROM(#0A)**. Инструкция PUSH заносит в стек по адресу **#FE** значение *RGX* = **#AE**, после этого значение указателя стека *SP* уменьшается на **1**. Программный счетчик увеличивается и указывает на адрес следующей инструкции подпрограммы **#0C**.

№ 05. `RGX #C5` — в регистр *RGX* заносится **#C5**. Важное для программы значение, подпрограмма перезаписала. Программный счетчик указывает на следующую инструкцию подпрограммы *PC* = **#0E**.

№ 06. Некоторая инструкция подпрограммы. Программный счетчик указывает на следующую инструкцию подпрограммы *PC* = **#10**.

№ 07. Некоторая инструкция подпрограммы. Программный счетчик указывает на следующую инструкцию подпрограммы *PC* = **#12**.

№ 08. Команда POP. Значение стека увеличивается на **1**, по получившемуся адресу **#FE** данные (**#AE**) из стека заносятся в регистр *RGX* = **#AE**. Таким образом, восстанавливается значение в регистре *RGX*, которое находилось в нем до вызова подпрограммы. Обратите внимание, что после извлечения значения доступ к нему в стеке теряется, т.к. *SP* указывает на ячейку куда необходимо поместить значение. Перед извлечением *SP* увеличивается на **1**. Программный счетчик указывает на следующую инструкцию подпрограммы *PC* = **#14**.

№ 09. Инструкция RET. Программный счетчик по-прежнему указывает на следующую инструкцию *PC* = **#16**. Значение стека увеличивается на **1**, по получившемуся адресу **#FF** данные (**#06**) из стека заменяют значение в программном счетчике *PC* = **#06**. Теперь счетчик указывает на адрес инструкции основной программы, которая следует после CALL. Следовательно, на следующем шаге произойдет возврат из

подпрограммы в основную программу. Стек в данный момент пуст, *SP* = **#FF** указывает на его вершину.

№ 10. FETCH извлекает инструкцию `JMP #02` по адресу **#06..#07** и увеличивает значение программного счетчика *PC* = **#08**. Однако, инструкция JMP заменяет значение в программном счетчике на **#02.** Происходит *безусловный* переход к *шагу № 02* и все шаги повторяются.

Замечание о PUSHах и POPах

Подпрограмма является кодом, который многократно используется в различных программах. **Подпрограмма должна оставлять данные программы без изменений**!

Другими словами, после вызова подпрограммы регистры должны хранить те же самые значения, что и до вызова. Для этого все регистры, которые используются в подпрограмме, сперва заталкиваются в стек командой PUSH. В конце подпрограммы регистрам восстанавливаются их значения из стека командой POP. При этом извлекать регистры необходимо строго в обратном порядке, иначе значения в них перепутаются местами.

Обратите внимание, что количество PUSHов и POPов должно совпадать! Если POPов будет меньше, то RET осуществит возврат не по тому адресу. Возврат произойдет по адресу со значением, содержащемся в регистре, который программист забыл извлечь из стека, перед вызовом RET.

5.2.8 Инструкция FETCH

Инструкция извлечения команды уже была рассмотрена ни один раз, более чем подробно. Единственное, следует отметить, что FETCH выполняется автоматически без участия программиста. FETCH не является командой программы, она как раз извлекает команды из *ROM*. Т.е. в коде писать FETCH бессмысленно. Код не может извлекать сам себя неведома куда. Это уже сбой в матрице называется. Имеется интересный одноименный фильм на эту тему. Единственное, у FETCH существует код

инструкции **#0000**, поэтому код операции первой команды `RGX #dd` имел значение **#01**, а не **#00**.

5.2.9 Полный набор команд

В таблице представлен полный набор инструкций МПА с кратким описанием.

КОП	Мнемоника	Аргумент	Описание
Инструкции записи данных			
#01	`RGX #dd`	данные	Запись числа в *RGX*
Инструкции копирования данных			
#02	`SETA`	--	Копирование из *RGX* в *RGA*
#03	`GETR`	--	Копирование из *RGR* в *RGX*
#04	`WR #aa`	адрес	Копирование из *RAM* в *RGX*
#04	`RD #aa`	адрес	Копирование из *RGX* в *RAM*
#06	`PRTOUT`	--	Копирование из *RGX* в *PRT*
#07	`PRTIN`	--	Копирование из *PRT* в *RGX*
Инструкции настройки порта			
#08	`PRTUNE`	--	Копирование из *RGX* в регистр настройки *TUNE* порта *PRT*
Инструкции АЛО			
#09	`ADD`	--	Сложение *RGA* с *RGX*
#0A	`AND`	--	Конъюнкция *RGA* с *RGX*
#0B	`OR`	--	Дизъюнкия *RGA* с *RGX*
#0C	`XOR`	--	Исключающее *ИЛИ RGA* с *RGX*
Инструкции переходов			
#0D	`JMP #aa`	адрес	Безусловный переход к команде программы, хранящейся в ROM по адресу **#aa**
#0E	`JMPZF #aa`	адрес	Переход по адресу **#a**a, если ZF = 1
#0F	`JMPCF #aa`	адрес	Переход по адресу **#a**a, если CF = 1
#10	`JMPPF #aa`	адрес	Переход по адресу **#aa**, если PF = 1
Инструкции работы со стеком			
#11	`SP #aa`	--	Запись в регистр указателя стека SP адреса вершины стека #aa

#12	PUSH	--	Запись в стек значения из RGX
#13	POP	--	Извлечение значения из стека в RGX
Инструкции вызова подпрограммы			
#14	CALL #aa	адрес	Вызов подпрограммы по адресу #aa
#15	RET	--	Возврат из подпрограммы, в то место, откуда она была вызвана.
#00	--	--	FETCH

Всего получилось **#15 = 1 * 16^1 + 5 *16^0 = 21d**. Двадцать одна инструкция. Остальные **256 – 21 = 235** кодов операций отправляются в резерв.

Как видно из таблицы, данные в качестве аргумента имеет единственная инструкция RGX #dd, записывающая данные **#dd** в регистр *RGX*. Все остальные команды, которые обрабатывают или пересылают данные работают с этим регистром.

Можно добавить инструкции сложения *RGA* непосредственно с числом, минуя его запись в *RGX* и другие аналогичные команды. В других микропроцессорных системах, для хранения КОП отводится 7 байт, соответственно, имеется **128** инструкций.

Глава 5.3 Пример программы

Поймем, что делает пример кода, представленный в начале раздела. Добавим комментарии к каждой строчке кода. Комментарии в ASSEMBLERe начинаются со знака точка с запятой «;». Не путайте с аналогичным обозначением в языке **C**, в котором символ «;» означает конец выражения.

```
RGX #08     ; запись в регистр RGX первого операнда (числа)
SETA        ; копирование первого операнда из RGX в RGA ALU
RGX #0C     ; запись второго операнда в RGX
ADD         ; сложение двух операндов RGA (#08) и RGX (#0C)
GETR        ; перемещение результата сложения ALU
            ; из регистра результата RGR в универсальный регистр RGX
```

Программа, представленная выше, выполняет сложение двух чисел **#08 + #0C**. Результат сложения **#14** помещается в универсальный регистр *RGX*. Ничего сложного, согласитесь?

Заметили, как плавно осуществился переход от электрона к ASSEMBLER рис. 5.5

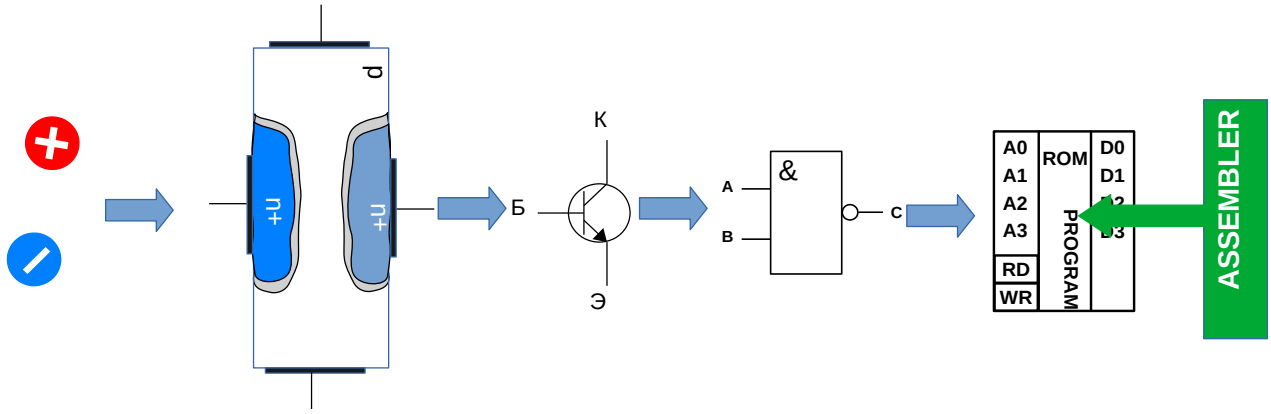


Рисунок 5.5— Переход от электрона к ассемблеру.

Глава 5.4 Прошивка

Рассмотрим каким образом программа, написанная на ASSEMBLER, из текстового файла оказывается в *ROM МПА*, но сперва поймем, чем отличается микроконтроллер от микропроцессора.

5.4.1 Микроконтроллеры и микропроцессоры

На самом деле, *МПА* является не просто микропроцессором. Был спроектирован целый *микроконтроллер*. Основное отличие микроконтроллера от микропроцессора в том, что *микроконтроллер* содержит постоянную память *ROM*. В *ROM* заносится программа, которая выполняется микроконтроллером по кругу (вспоминаем револьвер), до тех пор, пока на него подается напряжение питания.

Микроконтроллер самодостаточное устройство, содержащее внутри себя микропроцессор, который выполняет программу, и память, которая хранит программу. Помимо постоянной памяти *ROM* микроконтроллеры обременены различным устройствами типа оперативной памяти *RAM*, портами для общения с внешним миром, аналогово-цифровыми и цифро-аналоговыми преобразователями (АЦП и ЦАП), таймерами и т. д. Микроконтроллер — маленький компьютер, целый мир в одной микросхеме.

Микропроцессор — физическое логическое устройство вычисления, составная часть микроконтроллера, персонального компьютера (ПК) и множества других устройств.

В персональной ЭВМ (ПК, компьютере) в качестве *ROM* используется жесткий диск, в качестве *RAM* — отдельные платы, которые также можно заменить. Микропроцессор типичного компьютера более сложное и дорогое устройство (~10 000 т.р.), чем типичный микроконтроллер (~ 150 р.). Но микропроцессор не работает сам по себе, необходима внешняя память с программой, которую он может выполнить. По факту, системный блок компьютера — большой, мощный мегаконтроллер.

Итак, *МПА* — это микроконтроллер. Далее будем иметь дело только с микроконтроллерами.

5.4.2 Трансляция и прошивка

Пусть имеется текстовый файл с последовательностью мнемоник, представляющих собой программу на ASSEMBLER рис. 5.6.

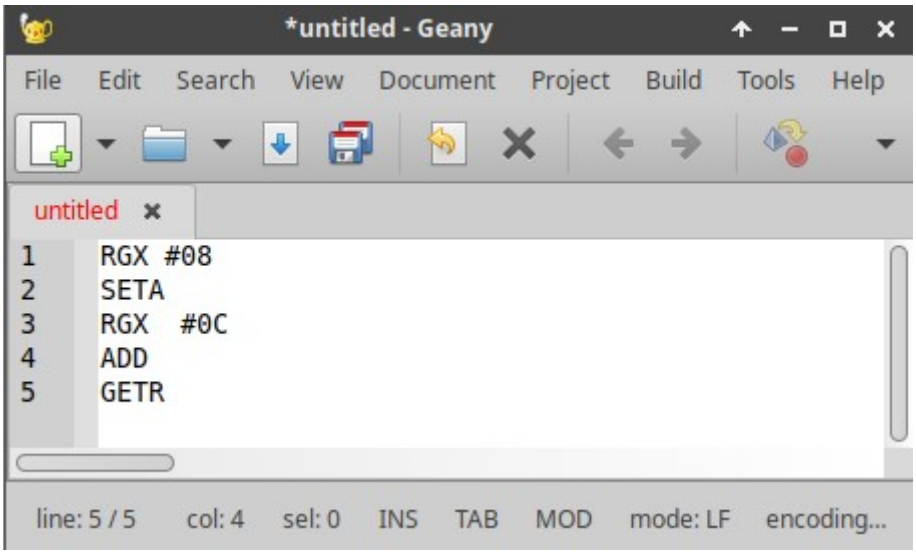


Рисунок 5.6 — Программа на ASSEMBLER в текстовом редакторе.

Требуется программу из текстового редактора записать в ROM микроконтроллера.

Процесс записи программы в *ROM* называется **прошивкой** (иногда, *программированием*). *Прошить* в *МПА* текстовый файл вряд ли удастся, поскольку *ROM* хранит двоичные коды команд, а не символы мнемоник. На помощь приходит *трансляция*.

Трансляция - процесс замены мнемоник двоичным кодом.

Транслятор — программа, осуществляющая трансляцию.

Прогнав через транслятор текстовый файл с программой *.asm*, получим файл с расширением *.hex*, хранящий двоичные коды программы рис. 5.7.

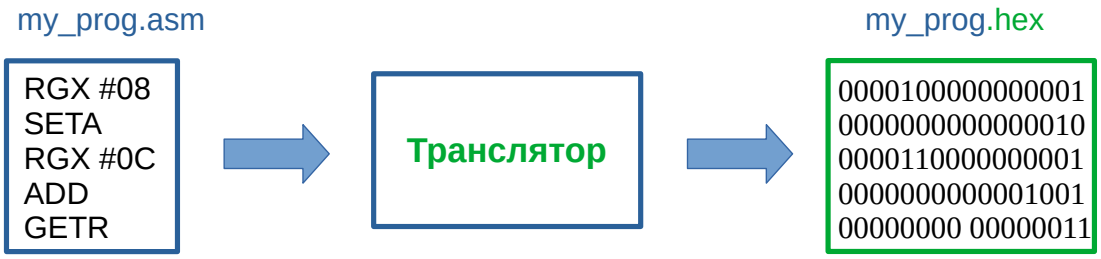


Рисунок 5.7 — Процесс трансляции текстового кода ASSEMBLER в двоичный код.

Через специальное устройство — **программатор**, содержимое файла *.hex прошивается* (записывается) в *ROM* программ микроконтроллера.

К компьютеру программатор подключается обычно через интерфейс USB. К микроконтроллеру же существует два способа подключения программатора: через параллельный и последовательный интерфейс. Слово интерфейс подразумевает способ обмена данными.

Параллельный интерфейс

При параллельном интерфейсе каждый бит *порции данных* передается по отдельному проводу. Все биты *порции данных* передаются одновременно. В качестве *порции данных* обычно выступает *байт.*

МПА предусматривает передачу данных непосредственно на шину данных микроконтроллера. Поэтому программатор соединяется с шиной данных (через выводы порта **`P7..P0`**) и шиной адреса по параллельному интерфейсу.

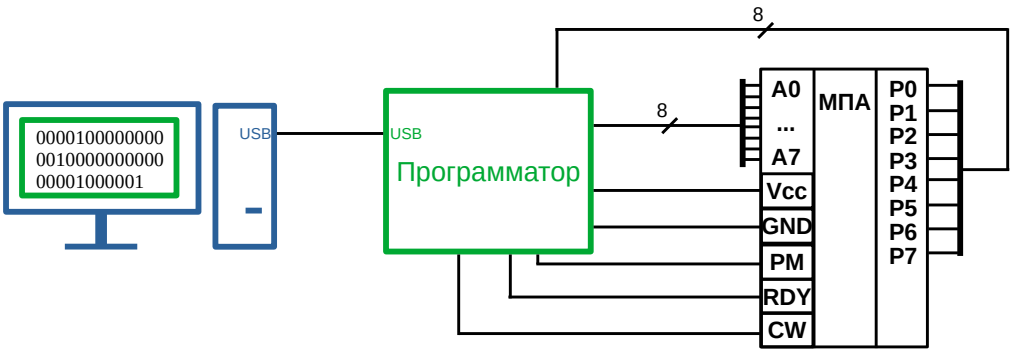


Рисунок 5.8 — Подключение программатора к МПА через параллельный интерфейс.

Перед началом прошивки необходимо подать на вход **PM** (**PROGRAMMING MODE**) **1**, чтобы *МПА* подготовилась к процессу. Например, настроила все выходы порта **PRT** на ввод. Запитаться (**Vcc** и **GND**) напряжением 5 В также можно от программатора. Общение между *МПА* и программатором осуществляется через два вывода **CW** (**CLOCK WRITE**) и **RDY** (**READY**).

Импульс на входе **CW** записывает очередную порцию данных **D7..D0** в указанную ячейку **A7..A0** *ROM МПА*. По выходу **RDY** микроконтроллер сообщает программатору, что байт успешно записан и *МПА* готова к обработке очередной порции данных.

Программатор отправляет данные на *ШД МПА*, немного ждет пока данные надежно «установятся» на *ШД*, пройдут все физические переходные процессы, подает импульс записи **CW**. При этом *МПА* устанавливает **RDY** в **0**. Далее, программатор ждет пока **RDY** станет равным **1**, после чего отправляет на *ШД* очередной байт.

Сперва записываются младшие 8 байт первой инструкции (число **#08**), далее настает очередь кода операции. Таким образом, байт за байтом, инструкция за инструкцией, программа размещается в памяти *ROM*.

Рассмотренный протокол передачи данных по параллельному интерфейсу вымышлен автором, но не теряет от этого своей теоретической значимости.

Последовательный интерфейс

Широко используется в современных устройствах. Например, USB (*Universal Serial Bus*) — универсальный последовательный интерфейс. USB содержит всего четыре контакта. Два из них это питание и земля, (поэтому флешки не имеют аккумуляторов) ещё два — информационные. Программатор к компьютеру подключается через последовательный интерфейс рис. 5.9.

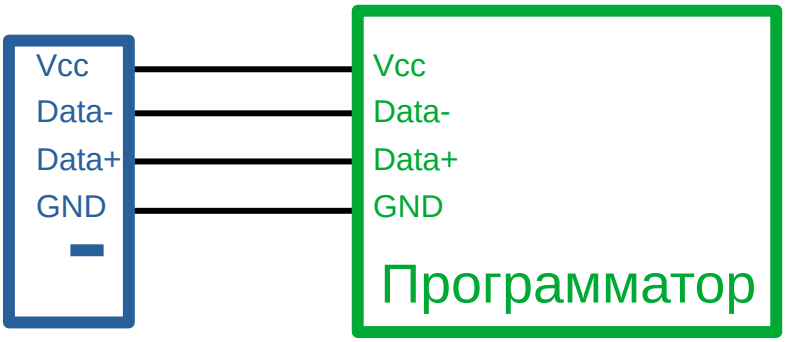


Рисунок 5.9 — Подключение программатора к компьютеру через последовательный интерфейс USB.

Каждый байт в последовательном интерфейсе передается по-битно. Сперва отправляется нулевой бит, затем первый бит, второй бит и т. д (либо наоборот).

Как ни странно, но последовательный интерфейс имеет более высокое быстродействие, проще решается проблема синхронизации. В параллельном интерфейсе, прежде чем отсылать данные на *ШД*, необходимо выждать некоторое время, чтобы завершились все переходные физические процессы и данные надежно выставились на выводах шины. В последовательном интерфейсе данные идут в виде электрических импульсов единым потоком. Проще отследить их корректность и последовательность.

Побитный способ передачи очень практичен. Ведь одно устройство может иметь 16-разрядную *ШД*, другое 8-разрядную, а третье 32-х разрядную. Тогда возникает ряд сложностей при соединении устройств по параллельному интерфейсу — острая нехватка ножек у микросхем.

Этих сведений об интерфейсах пока достаточно.

Раздел 6. Алгоритмические конструкции

Через призму ASSEMBLER рассматриваются три основные алгоритмические конструкции. С использованием данных конструкций реализуется любой мыслимый человеческим умом алгоритм. Изучение каждой конструкции подкреплено практическим примером с подробным разбором кода. Пришло время программировать на ASSEMBLER!

Глава 6.1 Умный дом (следование)

На базе 8-разрядного микроконтроллера *МПА* создадим систему, которая поддерживает уровень освещенности помещения постоянным. Назовем данную систему «умный дом».

6.1.1 Схема подключения

Модель «умного дома» изображена на рис.6.1. Поскольку «лезть в розетку 220В» отсутствует должная квалификация, вместо лампы освещения в схеме используется обычный светодиод, который вполне безопасно подключается к микроконтроллеру.

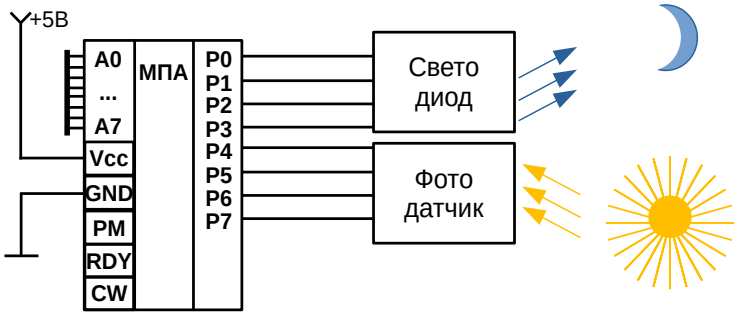


Рисунок 6.1 — Схема умного дома.

!**Никогда не подключайте напряжение сети 220В** к микроконтроллеру. Между микроконтроллером и сетью 220В используются специальные устройства, преобразующие низкое выходное напряжение микроконтроллера в высоковольтное и наоборот. Получайте соответствующее образование и квалификацию, чтобы работать с напряжениями, представляющими угрозу жизни.

На схеме представлен фотодатчик и регулируемый светодиод. Двоичное число на выходах фотодатчика пропорционально освещенности в помещении. Двоичное число на входах светодиода регулирует его яркость.

6.1.2 Описание алгоритма словами

Пусть фотодатчик настроен таким образом, что на выходе возникает максимальное значение *PHD* = **1111b**, если освещенность в помещении на достаточном уровне. В данном случае светодиод зажигать не требуется, на него должно поступать значение *LED* = **0000b**.

Если же с фотодатчика поступает значение *PHD* = **0000b**, то светодиод следует включить на полную яркость *LED* = **1111b.** Задача свелась к следующим действиям: пока микроконтроллер включен подавать значение с выхода фотодатчика на вход светодиода, предварительно инвертировав все биты. Блок-схема будущей программы представлена на рис 6.2.

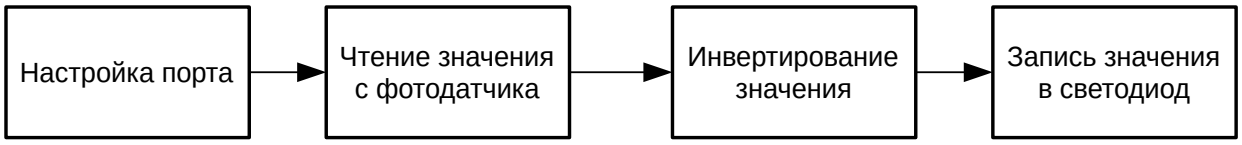


Рисунок 6.2 — Алгоритмическая конструкция следование.

Данная алгоритмическая конструкция называется *следование*. Инструкции выполняются последовательно одна за другой. Нет никаких перескоков. \

После выполнения всех инструкций программы устройство управления *УУ* будет извлекать пустые инструкции (по факту FETCH), пока не достигнет конца *ROM*. Затем выполнение начинается с самого начала, с настройки портов (см. револьвер).

6.1.3 Настройка выводов порта

Первая часть программы настраивает порты. Из схемы рис. 6.1 видим, что к младшим разрядам порта подключен фотодадчик, следовательно их необходимо настроить на ВВОД, для этого младшие биты настроечного слова следует установить в **0**. К старшим разрядам порта подключен светодиод, следовательно их необходимо настроить на ВЫВОД. Получается настроечное слово **11110000b = #F0**, которое через *RGX* заносится в регистр настройки порта.

```
; настройка порта
     RGX #F0    ; запись настроечного слова в RGX
     PRTUNE     ; запись в регистр настройки порта TUNE из RGX
; Чтение значения с порта
     PRIN       ; данные с фотодатчика заносятся в RGX
```

6.1.4 Инвертирование значения

Далее необходимо инвертировать все биты, считанные с фотодатчика (четыре младшие бита). Все бы хорошо, но в *МПА* нет команды *ИНВЕРСИЯ*, зато есть *XOR*…

Чтобы инвертировать число, осуществим операцию *XOR* каждого бита числа с единицей. Ведь *XOR* по определению, меняет **0** на **1** и **1** на **0**:

0 XOR 1 = 1 **и** **1 XOR 1 = 0**.

Например, **0100 XOR 1111 = 1011**. Код инверсии выглядит следующим образом:

```
     SETA       ; копирование значения фотодатчика из RGX в RGA
     RGX #0F    ; запись единиц в RGX
                ; (#0F – 00001111b инвертируем только младшие биты)
     XOR        ; RGA (значение с фотодиода) XOR RGX (#0F)
     GETR       ; запись результата инверсии из RGR в RGX
```

6.1.5 Сдвиг 4-х младших битов в старшие разряды

Все хорошо, однако, 4-ре бита, которые необходимо выдать на светодиод, располагаются в младших разрядах *RGX*, а светодиод подключен к старшим разрядам порта.

Например, с порта было считано число **00001001b**. Инверсия дает **00000110b.** Тогда на порт необходимо выдать значение **01100000b**. Что же делать? Правильно! Сдвинуть цифры инвертированного значения с фотодотчика, хранящиеся в *RGX*, на четыре позиции влево, тогда интересующее значение окажется в старших разрядах.

Но как осуществить сдвиг? Команда сдвига не предусмотрена в системе команд *МПА*. Зато умножение на основание системы счисления сдвигает цифры на одну позицию влево. Например, в десятичной системе счисления умножение на **10d** сдвигает цифры влево **32d * 10d = 320d.** Аналогично, в двоичной системе счисления умножение на **2d** сдвигает цифры влево **1101b * 2d = 11010b**.

Но *МПА* не умеет умножать. Ничего не умеет этот *МПА*…. Только складывать! Заменим умножение сложением. Сложим значение *RGX* само с собой четыре раза.

```
SETA      ; копирование в RGA значения из RGX
ADD       ; сложение RGA + RGX
GETR      ; копирование результата из RGR в RGX
SETA      ; и ещё три раза, тоже самое.
ADD
GETR
SETA
ADD
GETR
SETA
ADD
GETR
```

6.1.6 Запись значения в порт

Выдадим готовое значение через старшие разряды порта на светодиод.

```
PRTOUT    ; запись старших битов из RGX в старшие биты порта
```

6.1.7 Полная версия программы

Листинг 1. «Умный дом»

```
   ;    ****************************************
   ;    ***умный дом (постоянная освещенность)***
   ;    ****************************************
   ; Настройка порта
#00     RGX  #F0  ; запись настроечного слова в RGX
#02     PRTUNE    ; запись в регистр настройки порта из RGX
   ; Чтение значения фотодиода из порта
#04     PRIN           ; данные с фотодиода заносятся в RGX
   ; Инверсия значения, считанного с порта
#06     SETA      ; копирование из RGX в регистр ALU RGA
#08     RGX  #0F  ; запись единиц в RGX для инверсии через XOR
#0A     XOR       ; RGA (значение с фотодиода) XOR RGX (#0F)
#0C     GETR      ; запись результата инверсии из RGR в RGX
   ; Сдвиг значения из младших в старшие разряды
#0E     SETA      ; копирование в RGA значения из RGX
#10     ADD       ; сложение RGA + RGX
#12     GETR      ; копирование результата из RGR в RGX
#14     SETA      ; и ещё три раза, тоже самое.
#16     ADD
#18     GETR
#1A     SETA
#1C     ADD
#1E     GETR
#20     SETA
#22     ADD
```

```
#24     GETR
   ; Выдача значения на светодиод через старшие разряды порта
#26     PRTOUT    ; запись старших битов из RGX в порт.
```

В левом столбце указаны адреса ячеек памяти *ROM*, в которые записываются инструкции программы. Каждая ячейка памяти хранит 1 байт (8 бит). Каждая инструкция *МПА* занимает 2 байта. В первом байте хранится аргумент, во втором код операции. Поэтому адреса инструкций увеличиваются на два. Вся программа занимает **#28** ячеек памяти, т.к. отсчет отсчет ведется с нуля + ячейка с адресом **#27**, хранящая КОП последней инструкции. Итого программа занимает **#28 = 2 * 16^1 + 8 * 16^0 = 40** байтов. Что уместится в *МПА*. Напомним, что объем *ROM* 256 байт.

Особое внимание уделите на оформление кода ASSEMBLER.

Программа состоит из 4-х столбцов:

1. метки (о них чуть позже)

2. название инструкции

3. аргумент инструкции

4. ; комментарии (обратите внимание на пробел, после точки с запятой)

6.1.8 Подпрограмма

Длинный блок кода, осуществляющий сдвиг вправо на 4-ре разряда, наверняка пригодится еще ни раз, при написании других программ. Выглядит несуразно и «захламляет» код программы. Не плохо было бы вынести его в отдельную подпрограмму листинг 2.

Листинг 2. «Умный дом» с вызовом подпрограммы и бесконечным циклом

```
#00     SP   #FF
#02     RGX  #F0  ; запись настроечного слова в RGX
#04     PRTUNE    ; запись настроечного слова в порт
#06     PRIN      ; данные с фотодиода заносятся в RGX
#08     SETA      ; копирование из RGX в регистр ALU RGA
#0A     RGX  #0F  ; запись единиц в RGX для инверсии через XOR
#0C     XOR       ; RGA (значение с фотодиода) XOR RGX (#0F)
#0E     GETR      ; запись результата инверсии из RGR в RGX
#10     CALL #16
#12     PRTOUT    ; запись старших битов из RGX в порт
#14     JMP  #06
   ; Подпрограмма
#16     SETA      ; копирование в RGA значения из RGX
#18     ADD       ; сложение RGA + RGX
#1A     GETR      ; копирование результата из RGR в RGA
#1C     SETA      ; и ещё три раза, тоже самое.
#1E     ADD
#20     GETR
#22     SETA
#24     ADD
#26     GETR
#27     SETA
#28     ADD
#29     GETR
#30     RET
```

Для осуществления вызова подпрограммы пригодится *стек*. Чтобы использовать стек, в начале программы необходимо указать адрес вершины (начала) стека. Зададим последнюю ячейку памяти *RAM* в качестве вершины стека инструкцией SP #FF.

Инструкции, которые следует вынести в подпрограмму располагаются по адресам **#0E..#24** (см. предыдущий листинг 1). Вместо них разместим одну команду CALL. Команды подпрограммы переместим в конец, и добавим инструкцию возврата RET. Таким образом, будет создана подпрограмма, начинающаяся с адреса **#16**, и, заканчивающаяся адресом **#31**. В качестве аргумента CALL укажем адрес первой инструкции подпрограммы CALL #16.

Инструкция CALL #16 совершает два действия:

1. заносит текущее содержимое программного счетчика **#12** в стек **STACK(#FF) = #12**, значение *SP* уменьшается на единицу *SP* = **#FE**. Указатель стека указывает на следующую ячейку, куда будут размещены данные, например, инструкцией PUSH.

2. в программный счетчик заносится адрес **#16:** *PC* = **#16**.

Следующая после CALL инструкция FETCH начинает извлекать команды подпрограммы по адресу **#16**.

При вызове команды RET указатель стека *SP* увеличивается на единицу *SP* = **#FF**, и значение из указанной ячейка стека **STACK(#FF) = #12** записывается в программный счетчик *PC* = **#12**. Следующая после CALL инструкция FETCH извлекает команду основной программы по адресу **#12**, следующую сразу после CALL.

Команда JMP #06 изменяет значение программного счетчика *PC* = **#06**. Осуществляется переход к инструкции считывания данных с порта (настраивать стек и порт второй раз нет необходимости). Таким образом, реализуется бесконечный цикл.

Если не указать JMP, то после команды PRTOUT продолжится выполнение инструкций, относящихся к подпрограмме. Это не логично, потому что основная программа совершила все необходимые действия, и пора бы начать с начала.

Замечание о PUSHax и POPax

Команды PUSH и POP не требуются. После возврата, в RGX должно остаться измененное значение.

6.1.9 Метки

Указывать адреса инструкций в командах CALL и JMP неудобно. При добавлении в код программы новых команд адреса смещаются и велика возможность запутаться. Люди придумали вместо адресов записывать слова - *метки*. Транслятор при трансляции кода заменяет *метки* актуальными адресами инструкций. **Метка** представляет собой слово без пробелов, заканчивающееся двоеточием. Обратите внимание, что метка не является инструкцией. *Метка* просто дает имя адресу в том месте программы, где она располагается. Проверьте, пожалуйста, что адреса инструкций в листинге 2 и листинге 3 совпадают. *Метки* ничего не изменяют, также как и *мнемоники*. Это всего лишь обозначение.

В имени метки LEFT_RGX_4 присутствует название регистра. Таким образом, напоминаем сами себе, что подпрограмма изменяет значение в регистре *RGX*.

Листинг 3 «Умный дом» с вызовом подпрограммы, бесконечным циклом, с использованием меток

```
#00     SP   #FF  ; указываем вершину стека
#02     RGX  #F0  ; запись настроечного слова в RGX
#04     PRTUNE    ; запись настроечного слова в порт
   START:
#06     PRIN      ; данные с фотодиода заносятся в RGX
#08     SETA      ; копирование из RGX в регистр ALU RGA
#0A     RGX  #0F  ; запись единиц в RGX для инверсии через XOR
#0C     XOR       ; RGA (значение с фотодиода) XOR RGX (#0F)
#0E     GETR      ; запись результата инверсии из RGR в RGX
```

```
#10     CALL LEFT_RGX_4
#12     PRTOUT    ; запись старших битов из RGX в порт.
#14     JMP  START
   ; Подпрограмма
   LEFT_RGX_4:
   ; PUSHы при необходимости
#16     SETA      ; копирование в RGA значения из RGX
#18     ADD       ; сложение RGA + RGX
#1A     GETR      ; копирование результата из RGR в RGA
#1C     SETA      ; и ещё три раза, тоже самое.
#1E     ADD
#20     GETR
#22     SETA
#24     ADD
#26     GETR
#27     SETA
#28     ADD
#29     GETR
   ; POPы в порядке, обратном PUSHам
#30     RET
```

Глава 6.2 Холодильник (ветвление)

Напишем простенькую программу для холодильника: «Если дверца открыта — зажечь лампочку».

6.2.1 Схема подключения

Схема подключения дверцы и лампочки показана на рис. 6.3. Дверца подключена к самому старшему разряду порта, а лампочка к самому младшему.

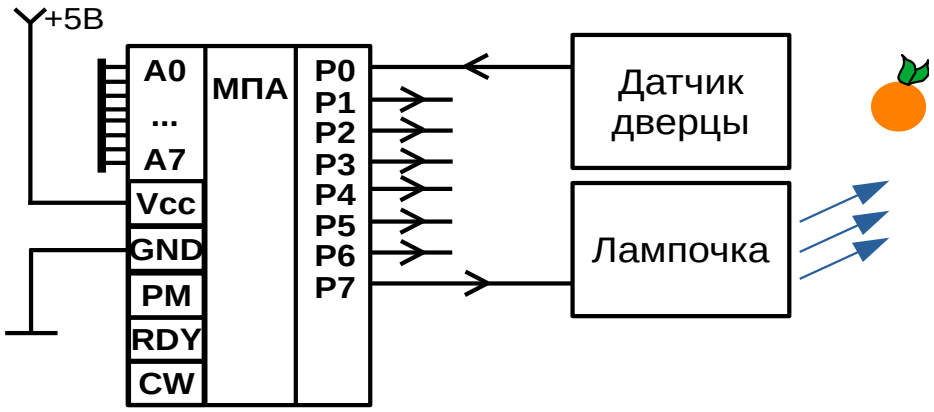


Рисунок 6.3 — Схема подключения дверцы и лампочки к МПА.

6.2.2 Описание программы словами

Действия, совершаемые программой, описываются следующим образом: «Если дверца открыта — зажечь лампочку, иначе потушить лампочку».

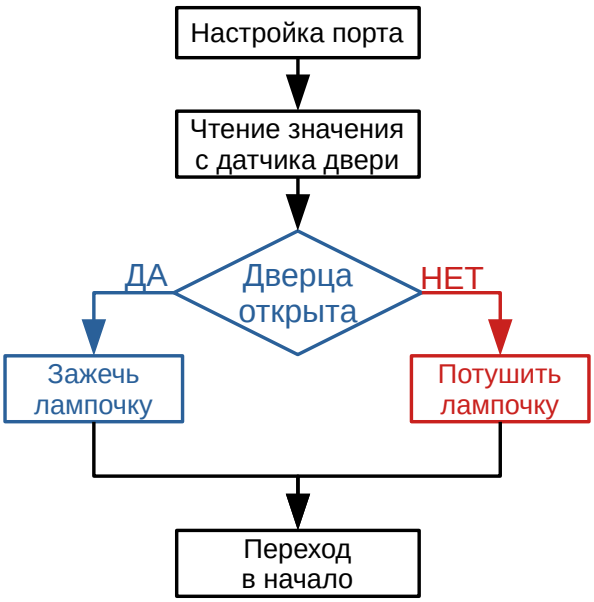


Рисунок 6.4 — Алгоритмическая конструкция ветвление

Данная алгоритмическая конструкция называется ветвление — некоторая последовательность команд (`зажечь лампочку`) выполняется, только при выполнении условия (`дверца открыта`), в противном случае выполняется другая последовательность команд (`потушить лампочку`).

6.2.3 Настройка порта

Как и в предыдущей программе, начнем с настройки стека и портов. Дверца подключена к самому старшему разряду порта, а лампочка к самому младшему. Настроим все выводы порта на ВЫВОД, кроме самого старшего. Настроечное слово будет иметь вид **`01111111b = #7F`**.

```
SP    #FF   ; указываем вершину стека
RGX   #7F   ; запись настроечного слова в RGX
PRTUNE      ; запись настроечного слова из RGX в порт (TUNE)
```

6.2.4 Получение состояния дверцы

Датчик открывания дверцы выдает **1**, если дверца открыта, иначе **0**. Интересующий бит, записывается в старший разряд порта. Необходимо проверить, равен ли старший разряд **?** порта единице **?0000000b**. Для этого осуществим сдвиг числа влево на одну позицию.

```
PRTIN     ; чтение порта в RGX
SETA      ; копирование в RGA значения из RGX
ADD       ; сложение RGA + RGX
```

Если старший разряд был равен единице, то возникнет переполнение разрядов.

Появляется возможность проверить флаг переноса **CF**.

6.2.5 Проверка дверцы (ветвление)

Если **CF = 1** — зажечь лампочку, иначе потушить.

```
      JMPCF LIGHT

      RGX, #00  ; потушить лампочку

      PRTOUT

      JMP END
LIGHT:
      RGX, #01  ; зажечь лампочку

      PRTOUT
END:
```

Очень хитрый код рис. 6.5 получился.

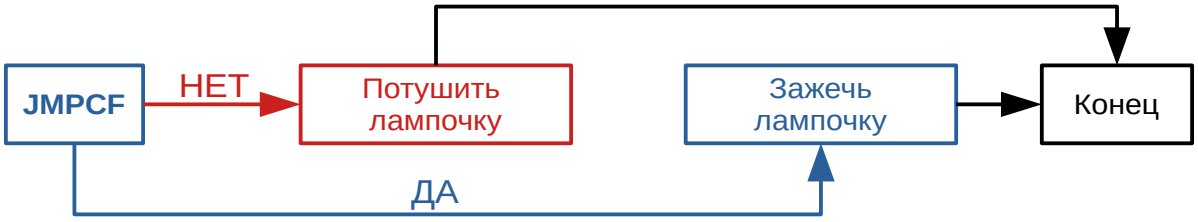


Рисунок 6.5 — Реализация алгоритмической конструкции ветвление.

Если дверца открыта, то JMPCF перепрыгивает через команды, которые выключают лампочку, и выполнение программы переносится к инструкциям включения лампочки по адресу LIGHT.

Если же дверца закрыта, то после JMPCF программа продолжает выполнятся, заставляя лампочку гаснуть. Команда JMP END перепрыгивает инструкции зажигания лампочки, выполнение переходит к сразу к инструкции с адресом END.

Чтобы проверить открыта ли дверца, пришлось сдвигать число влево. Чтобы сдвинуть число влево, пришлось сложить его с самим с собой. Чтобы зажечь или потушить лампочку пришлось выполнить акробатический этюд, состоящий из прыжков через последовательность команд в определенном порядке. Примерно это подразумевалось в фразе «программист на ассемблере мыслит как машина».

Вы обратили внимание, что стек не пригодился, но вершина была все-таки записана в SP? На всякий случай, в начале программы лучше всегда настраивать указатель стека, если вдруг стек потребуется, чтобы было.

Листинг 4 «Дверца холодильника»

```
   ; Настройка стека и порта
#00     SP   #FF  ; указываем вершину стека
#02     RGX  #7F  ; запись настроечного слова в RGX
#04     PRTUNE    ; запись настроечного слова
   ; Чтение датчика двери и сдвиг с целью дальнейшей проверки
   START:
#06     PRTIN     ; чтение порта в RGX
#08     SETA      ; копирование в RGA значения из RGX
#0A     ADD       ; сложение RGA + RGX
   ; Ветвление, в ходе которого лампочка зажигается и тушится
#0C     JMPCF LIGHT ; переход к адресу LIGHT если CF = 1
#0E     RGX, #00  ; значение, которое потушит лампочку
#10     PRTOUT    ; тушим лампочку
#12     JMP END
   LIGHT:
#14     RGX, #01  ; значение, которое зажжет лампочку
#16     PRTOUT    ; зажигаем лампочку
   END:
#18     JMP START ; переход к началу программы
```

Вряд ли в холодильнике рассмотренная элементарная функция реализована микроконтроллером. Но если проявить смекалку, и подключить предыдущую программу, можно заставить лампочку не зажигаться при открывании дверцы днем.

Глава 6.3 Утюг (циклы)

Запрограммируем более совершенное устройство — *утюг*. Пусть у утюга имеется датчик наклона, показывающий находится ли утюг в горизонтальном (рабочем) положении. Также утюг оснащен устройством парового удара.

Решим следующую задачу: «*Паровой удар утюга должен осуществляться каждые 10 секунд, если утюг находится в горизонтальном состоянии*». Схема подключения датчика наклона и устройства парового удара представлена на рис. 6.6.

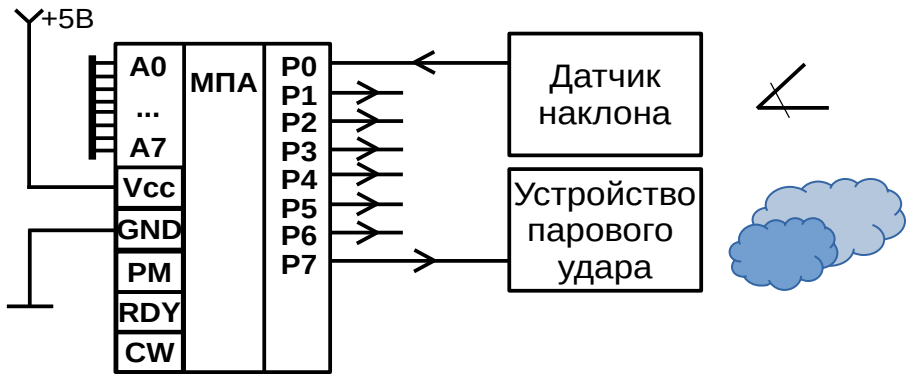


Рисунок 6.6 — Схема подключения датчика наклона и устройства парового удара к МПА

6.3.1 Описание программы словами

Действия, совершаемые программой, описываются следующим алгоритмом рис. 6.7:

1. Ждем, когда утюг окажется в горизонтальном положении

2. Делаем паровой удар

3. Ждем 10 секунд

4. Переход к п.1.

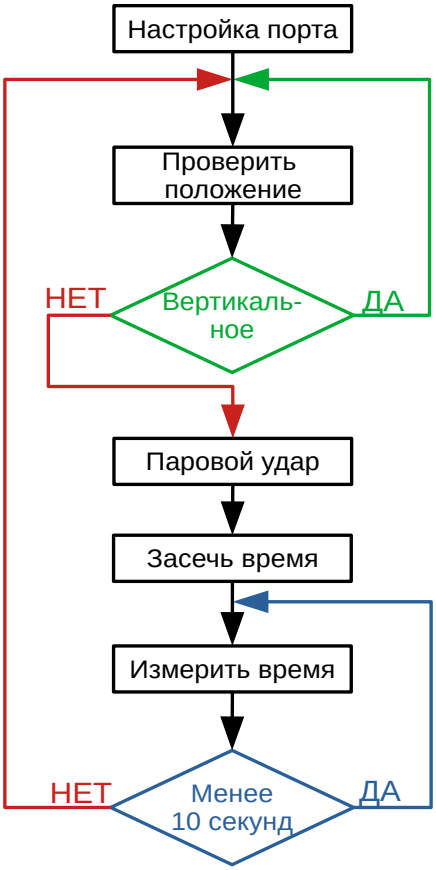


Рисунок 6.7 — Блок-схема программы «паровой удар утюга»

Первый и третий пункт представляют собой алгоритмические конструкции — *циклы*. **Циклы** позволяют выполнить последовательность команд, пока истинно некоторое условие. Повторяющаяся в цикле последовательность команд называется **телом цикла**. Другими словами, *тело цикла выполняется пока условие истинно*.

В первом пункте телом является команда проверить (узнать) положение, а условием вертикальное ли положение утюга. В третьем пункте, телом является команда измерить время, а условием прошло менее 10 секунд.

6.3.2 Настройка портов

Датчик наклона подключен к самому старшему разряду порта, а устройство парового удара к самому младшему. Настроим все выводы порта на ВЫВОД, кроме самого старшего, *настроечное слово* будет иметь вид **01111111b = #7F**. Все как для холодильника.

```
SP   #FF  ; указываем вершину стека
RGX  #7F  ; запись настроечного слова в RGX
PRTUNE    ; запись настроечного слова в порт
RGX  #00  ; отключение парового удара
PRTOUT    ; отключение парового удара
```

Напишем первую часть программы, которая ожидает момента, когда утюг окажется в горизонтальном положении, в данном случае датчик наклона выдаст **1**.

Необходимо проверить, равен ли старший разряд порта единице. Как и в случае с холодильником, используем прием со сдвигом порта влево. Будем проверять флаг **CF**, который установится в **1**, если в старшем разряде порта хранится **1**, т.е. утюг в горизонтальном положении. Алгоритм звучит следующим образом: «проверять наклон утюга пока **CF** равен **0** (утюг в вертикальном положении)».

```
CHK_ANGLE:
     PRTIN          ; чтение порта в RGX
     SETA           ; копирование в RGA значения из RGX
     ADD            ; сложение RGA + RGX
     JMPCF END_CHK  ; утюг в рабочем положении завершаем цикл
     JMP CHK_ANGLE
END_CHK:
```

Поскольку в наборе инструкций МПА отсутствует проверка флага на равенство нулю, пришлось применить маленькую хитрость. Был организован бесконечный цикл `CHK_ANGLE: … JMP CHK_ANGLE`. Внутри этого цикла считывается и проверяется значение старшего бита порта. Когда оно становится равным единице команда `JMPCF END_CHK` «выпрыгивает» из цикла рис. 6.8.

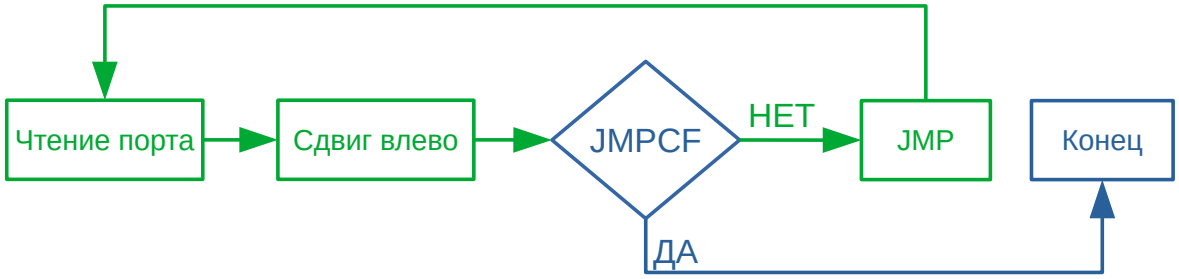


Рисунок 6.8 — Реализация алгоритмической конструкции цикл.

6.3.3 Включение парового удара

Без комментариев

```
RGX #01
PRTOUT
RGX #00
PRTOUT
```

6.3.4 Ожидание...

Третья часть самая интересная. Требуется подождать 10 секунд… Единственное устройство в *МПА*, связанное с таким видом бытия как время, является генератор импульсов *GENA*, о котором было несправедливо упомянуто лишь единожды. Восстановим справедливость.

Немного элементарной арифметики. Пусть *GENA* генерирует импульсы с частотой **8000** кГц (кило герц). Это означает, что за секунду возникает **8000** тактов. На выполнение каждой команды в *МПА* отводится **8** тактов. Тогда за одну секунду выполнится **8000 / 8 = 1000** команд. За **10** секунд выполнится **10 000** команд. Таким образом, чтобы отсчитать **10** секунд, следует заставить *МПА* выполнить около **10 000** любых команд.

CCL_256

Реализуем цикл CCL_256, который считает от **0** до **256**. Для этого напишем программу, которая прибавляет к регистру значение **1**, пока не произойдет переполнение разряда. Значение, хранящееся в регистре *RGX,* будем называть СЧЕТЧИК_256.

```
      RGX #00
CCL_256:
      SETA
      RGX  #01
      ADD
      GETR
      JMPCF     END_256
      JMP  CCL_256
END_256
```

Как видно из кода за *итерацию* (одно повторение тела цикла) выполняется **6** команд, написанных программистом (между CCL_256 и END_256), и **6** раз инструкция FETCH. Итого, за итерацию выполняется **12** команд. Всего итераций **256**, именно столько раз необходимо прибавить **1**, чтобы возникло переполнение. Умножим **256 * 12 = 3072**. Выполнив **3** таких цикла, получим порядка **3072 * 3 = 9216**. Немного изменим задание, пусть время ожидания составляет **9,216** с вместо **10** с.

Так бывает, иногда техническое задание корректируется.

CCL_3

Реализуем цикл `CCL_3`, который считает от **0** до **3**х. Значение, хранящееся в регистре *RGX,* будем называть СЧЕТЧИК_3.

Первая проблема в том, что универсальный регистр один, а хранить требуется одновременно СЧЕТЧИК_256 и СЧЕТЧИК_3. Придется задействовать оперативную память *RAM* для хранения СЧЕТЧИК_3.

Вторая проблема, как проверить, что значение счетчика равно **3**-м? Нормальный способ — вычесть из счетчика число **3**, и проверить флаг **ZF**. Если он равен **1**, то значение счетчика равно **3**. Но вот беда, нет операции вычитания. Тогда прибавим к СЧЕТЧИК_3 число **253d** и проверим опять-таки флаг переполнения разряда **CF**. Если значение СЧЕТЧИК_3 равнялось трем, то

253d + 3d =256d = 1 0000 0000b.

Немного отвлечемся и переведем **253d** в шестнадцатеричную систему счисления. Используем способ попроще. Известно, что **255d = #FF**. Тогда **253d = 255d – 2d = #FF - #02 = #FD**.

Вернемся к циклам. Во время исполнения цикла CCL_256 значение СЧЕТЧИК_3 будем хранить в ячейке *RAM* с адресом **#00**. После завершения цикла CCL_256 будем извлекать, увеличивать и проверять значение СЧЕТЧИК_3 и помещать измененное значение обратно в *RAM*.

```
     RGX #00
     WR #00    ; сохраняем значение счетчика_6 в первой ячейке RAM
CCL_3:
 ; CCL_256 отсчитываем 3072 мс
; увеличение СЧЕТЧИК_3 в памяти RAM на единицу
     RGX  #01
     SETA      ; заносим 1 в RGA, ее будем прибавлять к СЧЕТЧИК_3
     RD   #00  ; извлекаем значение СЧЕТЧИК_3 из RAM в RGX
     ADD       ; увеличиваем на единицу СЧЕТЧИК_3 + RGA (+1)
     GETR      ; сохраняем увеличенное значение в RGX
     WR   #00  ; записываем СЧЕТЧИК_3 из RGX в первую ячейку RAM
; проверка на равенство СЧЕТЧИК_3 трем
     SETA      ; заносим СЧЕТЧИК_3 в RGA
     RGX  #FD  ; прибавляем 253 к RGA
     JMPCF     END_3  ; если СЧЕТЧИК_3 = 3 (при этом CF = 1),
выходим из цикла
     JMP  CCL_3
END_3
```

Листинг 5 «Паровой удар утюга через 10 с в горизонтальном положении»

```
; Настройка стека и порта

     SP   #FF  ; указываем вершину стека
     RGX  #7F  ; запись настроечного слова в RGX
     PRTUNE    ; запись настроечного слова в порт

; PART_I ждем пока утюг окажется в горизонтальном положении

CHK_ANGLE:
     PRTIN     ; чтение порта в RGX
     SETA      ; копирование в RGA значения из RGX
     ADD       ; сложение RGA + RGX (сдвиг влево)
     JMPCF END_CHK  ; если утюг в рабочем положении завершаем цикл
     JMP CHK_ANGLE  ; иначе заново проверяем значение из порта
END_CHK:

; PART_II делаем паровой удар

     RGX #01
     PRTOUT
     RGX #00
     PRTOUT

; PART_III ждем 9 секунд

RGX #00
     WR #00    ; сохраняем начальное значение СЧЕТЧИК_3 в RAM
CCL_3:
```

```
; НАЧАЛО ЦИКЛА CCL_256
     RGX #00
CCL_256:
     SETA
     RGX  #01
     ADD
     GETR
     JMPCF     END_256
     JMP  CCL_256
END_256
; КОНЕЦ ЦИКЛА CCL_256

; увеличение СЧЕТЧИК_3 в памяти RAM на единицу
     RGX  #01
     SETA      ; заносим 1 в RGA, ее будем прибавлять к СЧЕТЧИК_3
     RD   #00  ; извлекаем значение СЧЕТЧИК_3 из RAM в RGX
     ADD       ; увеличиваем на единицу
     GETR      ; сохраняем увеличенное значение в RGX
     WR   #00  ; копируем СЧЕТЧИК_3 из RGX в первую ячейку RAM
; проверка на равенство счетчика 3-м
     SETA      ; заносим СЧЕТЧИК_3 в RGA
     RGX  #FD  ; прибавляем 253d к RGA
     JMPCF     END_3 ; если досчитали до 3х выходим из цикла
     JMP  CCL_3 ; иначе идем считать от 0 до 256 ещё разок
END_3:
JMP CHECK_ANGLE     ; прошло около 9 секунд, вновь проверяем угол
```

На этом примере заканчивается изучение архитектуры разработанного микроконтроллера *МПА* и составленного для него языка программирования низкого уровня ASSEMBLER МПА.

Таким образом, был пройден путь от электрона до ассемблера. Теперь посмотрим на переход от ассемблера к языкам программирования высокого уровня.

Глава 6.4 Высокоуровневое программирование

Заставить микропроцессорную систему выполнять *задачуX* можно на разных уровнях.

Физический уровень — самая маленькая фигура. Здесь понадобится паяльник. Ничто не мешает собрать из транзисторов устройство, выполняющее *задачуX.* Разве что конечное время жизни Вселенной (если *задачаX* сложная) .

Логический уровень накрывает собой «сложную» физику. Собрать схему из логических элементов для выполнения *задачиX* уже вполне реально. Некоторые простые задачи решаются таким способом.

Низкоуровневое программирование рассматривает микропроцессор как единое целое. Представлено языком ASSEMBLER. Низкоуровневое программирование позволяет решать повседневные *задачиX.* Однако, не все имеют способность думать как машина, чтобы писать на ASSEMBLER. Низкоуровневое программирование плохо подходит для реализации больших программных систем.

Выскоуровневое программирование скрывает все непонятные дебри электрических, логических схем и двоичных кодов инструкций. Чтобы программировать на высоком уровне, необходимо лишь иметь способность думать как человек, и выполнять некоторые правила языка, называемые **синтаксисом**. Разумеется, программирование называется *высокоуровневым* не в том плане, что оно лучше других.

Уровень прикладных программ включает то, чем пользуемся в обычной жизни (офисные приложения, графические редакторы и многие другие программы).

Уровень Instagram. Сюда лезть не будем.

6.4.1 Простой пример высокоуровневого кода

Запишем программу сложения двух чисел на языке программирования высокого уровня, например, **C++**:

```
/*
     Сложение на языке высокого уровня C++
*/
int a; // объявление переменных
int b;
int c;
a = 5; // присваивание значений (инициализация)
b = 12;
c = a + b; // сложение и присваивание, две операции в одной строке
```

Чтобы *МПА* смогла выполнить данный код, его необходимо перевести в низкоуровневый код ASSEMBLER *МПА*.

6.4.2. Компиляция

Компиляция – процесс перевода высокоуровневого кода в низкоуровневый.

Компилятор – программа, которая разбирает высокоуровневый текст символ за символом и генерирует низкоуровневые инструкции ASSEMBLER.

Чтобы компилятор «понял», что от него хочет человек, программист при написании кода обязан придерживаться строго определенным правилам — **синтаксису**.

Переменная — это ячейка *RAM* (оперативной памяти), для хранения некоторых значений. Что именно хранится, определяется **типом**. Тип `int` (от англ. *integer*) — означает целое число. Понятие *тип* исключительно высокоуровневое и абстрактное, в ячейках *RAM* хранятся просто двоичные числа. Понятие *тип* придумано для удобства программирования.

С **именем** переменной дело обстоит также, как с метками. При компиляции имена переменных заменяются адресами ячеек *RAM.* В процессе выполнения программы в данных ячейках сохраняются значения соответствующих переменных.

С «высоты высокого уровня» на объявление переменной можно смотреть как на создание именованного контейнера для хранения значений определенного типа. Запись `int a;` создает контейнер для хранения целых чисел, к которому можно обращаться по имени `a`. Хотя на самом деле, просто выделяется ячейка *RAM* для хранения числа.

Попробуем думать, как компилятор, и скомпилируем программу, представленную выше.

1. `int a;` Я компилятор и анализирую первую строку кода. Здесь объявляется переменная типа `int` с именем `a.` Выделю для нее ячейку *RAM* с адресом **`#00`**. Если далее в коде встречу имя `a`, то буду обращаться по адресу **`#00`**.

Количество ячеек, которое будет выделено под хранение переменной определяется ее типом и компилятором. Зачастую для типа `int` выделяется 4 байта. Но в нашем случае пусть выделяется одна ячейка.

2. `int b;` Я компилятор и анализирую вторую строку кода. Здесь объявляется переменная типа `int` с именем `b` Выделю для нее ячейку *RAM* с адресом **`#01`**. Если далее в коде встречу имя `b`, то буду обращаться по адресу **`#01`**.

3. `int c;` Я компилятор и анализирую третью строку кода. Здесь объявляется переменная типа `int` с именем `c.` Выделю для нее ячейку *RAM* с адресом **`#02`**. Если далее в коде встречу имя `c`, то буду обращаться по адресу **`#02`**.

4. `a = 5;` Я компилятор и анализирую четвертую строку кода. Имеется оператор присваивания `=`. Значение `5`, стоящее справа от `=`, необходимо поместить в ячейку, стоящую слева **`(#00)`**. Генерирую код:

```
RGX  #05
WR   #00
```

5. `b = 12;` Я компилятор и анализирую пятую строку кода. Имеется оператор присваивания `=`. Значение `12`, стоящее справа от `=`, необходимо поместить в ячейку, стоящую слева **`(#01)`**. Генерирую код:

```
RGX  #0C
WR   #01
```

6. `c = a + b;` Я компилятор и анализирую шестую строку кода. Здесь имеется много всего. Ты меня совсем не любишь. Сперва выполним операцию сложения. Требуется сложить значения из ячеек a и b, т. е. **`#00`** и **`#01`**

```
RD   #00
SETA
RD   #01
ADD
```

А потом еще результат положить в переменную `c`. Ой! Т.е. в ячейку **`#02`**

```
GETR
WR #02
```

7. Компиляция выполнена успешно, держи код:

```
RGX  #05
WR   #00
RGX  #0C
WR   #01
RD   #00
SETA
RD   #01
ADD
GETR
WR   #02
```

Таким образом, после компиляции получается *.hex* файл с двоичным кодом, который можно прошить в микроконтроллер.

6.4.3 Интерпретация

Когда имеем дело с персональным компьютером (ПК), то зачастую сталкиваемся с другим методом выполнения программного кода - **интерпретацией**. В отличие от компиляции, файл с двоичным кодом не генерируется. При интерпретации программа компилируется и выполняется *«построчно (на лету)»*. Представьте, что в разобранном выше примере, команды, сгенерированные на каждом шаге, сразу отправляются на выполнение. На самом деле это удобно, поскольку всегда можно изменить исходный высокоуровневый код программы.

При *интерпретировании* программа работает чуть медленнее, поскольку интерпретатору необходимо время разобраться, что написано в очередной строчке, сгенерировать и отправить на выполнение низкоуровневый код. В компилированной программе, двоичные коды команд уже готовы и ждут своей очереди в памяти.

При нынешнем развитии технологий интерпретаторы работают довольно быстро, последнее время им отдается большее предпочтение. Ведь они удобнее, всегда можно изменить код, сделать его гибким. Программа вообще может работать, не имея всего программного кода целиком, просто будет выполняться имеющаяся часть исходного текста программы.

Плохим примером хорошего интерпретируемого языка программирования высокого уровня является **C#**. Говорят, что программа на языке **C#** компилируется. Это факт. Только вот код **C#** переводится не в машинный ASSEMBLER, а в язык промежуточного уровня **IL** (*Intermediate Language*). А уже код на языке **IL** интерпретируется виртуальной машиной **CLR** (*Common Language Runtime*) в машинный ASSEMBLER.

Программист пишет не для реального микропроцессора, а для виртуальной машины. Имеет место абстрагирование от аппаратной составляющей. Что тоже удобно, т.к. у всех компьютеры разные. Для выполнения таких программ, на компьютере пользователя должна быть установлена виртуальная машина (.NET Framework). К слову код на любом языке можно скомпилировать в **IL**, был бы компилятор. **JVM** (Java Virtual Machine) — другой пример интерпретатора, уже для языка **Java**.

6.4.4 Сравнение кода

Просто сравните, как выглядит код на языке программирования высокого уровня (слева) и низкого уровня (справа)

```
int a, b, c;
a = 5;
b = 12;
c = a + b;
```

```
RGX  #05
WR   #00
RGX  #0C
WR   #01
RD   #00
SETA
RD   #01
ADD
GETR
WR   #02
```

Спрашивается, зачем тогда необходимо изучать и писать на ASSEMBLER?

Бывают задачи, выполнение которых критично по времени. Например, управление полетом крылатой ракеты, система аварийного отключения и т.д. Каким бы хорошим не был компилятор, человек способен написать код на ASSEMBLER более эффективно. Это связано с понятием *универсальность*. Чем более система универсальная, тем более она «*тормознутая»*.

Компилятор, можно сказать, система универсальная. Ведь он одинаково универсально компилирует программу передачи котенка через интернет и программу передачи в цель крылатой ракеты.

```
/*
     Просто посмотрите на код для утюга, на языке высокого уровня
*/

#include "mpa.h"

int main()
{
     PortTune(127); // запись в порт настроечного слова #7F
     bool isHorizontal = false; // логический тип (true/false)
     while(1)
     {
          isHorizontal = ReadPin(7); // чтение старшего бита порта
          if(isHorizontal == true) // если горизонтальное полож.
          {
               //Включаем паровой удар
               WritePin(0, true); // запись младшего бита порта
               WritePin(0, false); // запись младшего бита порта
          }
          else
          {
               // переходим к следующей итерации (в начало цикла)
               continue; // завершение текущей итерации
          }

          // Отсчитываем задержку около 9 секунд
          for(int i = 0; i < 3; i++) // внешний цикл
          {
               for(int j = 0; j < 255; j++) // внутренний цикл
               {
               }
          }
     }
}
```

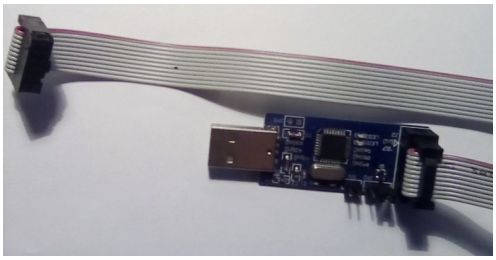
Раздел 7. Пошаговое руководство

Приводится подробная инструкция по конструированию гирлянды на базе микроконтроллера *ATmega328P* фирмы *Atmel*. Сперва решается вопрос приобретения комплектующих, далее подробно разбирается электрическая схема подключения микроконтроллера и светодиодов. Приводятся необходимые сведения о макетных платах. Производится подготовка персонального компьютера, заключающаяся в установке драйвера для программатора, транслятора для ассемблера и программы для прошивки двоичного кода в микроконтроллер. В заключении подробно разбирается программный код и последовательность действий, которые необходимо совершить, чтобы прошить программу в микроконтроллер и светодиод заморгал.

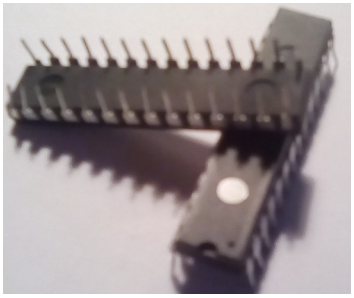
Шаг 1 Закупка комплектующих

Начнем проектирование гирлянды с закупки комплектующих:

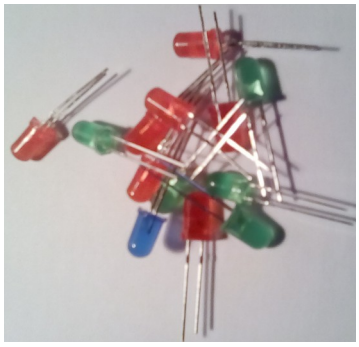
1. **Программатор USBasp.**



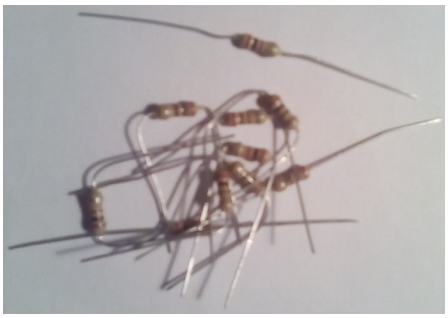
2. **Микроконтроллер ATMega328P** — 2 шт. (один запасной). Обязательно обращайте внимание на тип корпуса. Он должен быть PDIP. Данная разновидность корпуса предназначена для размещения на макетной плате, ножки у микросхемы длинные и направленны вниз.



3. **Светодиоды** — горсть, цвета по вкусу. Они будут выступать в качестве лампочек гирлянды.



4. **Резисторы** — сопротивление 200 Ом — горсть по количеству светодиодов, можно побольше (мощность 0.125 или 0.25 Вт). Будут использоваться в качестве токоограничивающих резисторов, чтобы светодиоды светили долго и счастливо.



5. **Беспаячная макетная плата** — 1 шт. Специальная плата с отверстиями на которой можно разместить микроконтроллер и светодиоды, и соединить их **перемычками для макетной платы**.

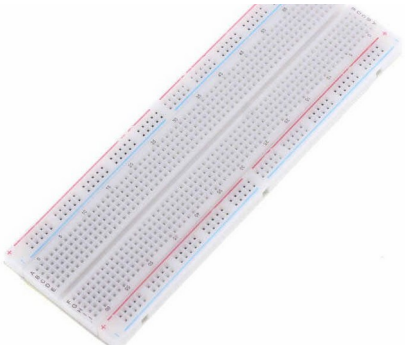


рисунок с сайта http://top-shop-magazin.ru

6. Набор **перемычек для макетной платы** — 1 шт.

рисунок с сайта https://voltiq.ru/

7. Набор **гибких перемычек** - десяти перемычек будет достаточно, но купите больше, пригодятся. Обратите внимание на обоих концах перемычки присутствует штырь (вилка). Кабели такого типа называют *Male-Male* (вилка-вилка).

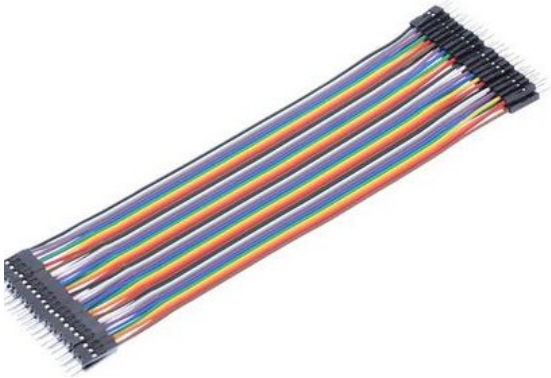


рисунок с сайта http://top-shop-magazin.ru

8. **Мультиметр** — прибор для измерения напряжений и сопротивлений.

рисунок с сайта https://m.elektro.ru/

Шаг 2 Знакомство

Для начала познакомимся поближе с микроконтроллером, изучим назначение его выводов. Далее разберемся с программатором и установим для него драйвер.

Шаг 2.1 Знакомство с ATmega328P

Микроконтроллер обладает следующими интересными характеристиками:

- Память программ (FLASH) — 32 Кбайт
- Оперативная память (SRAM) — 1 Кбайт (1024 байта)
- Память данных (EEPROM) 2 Кбайта (2048 байт) использоваться не будет
- Количество выводов на схеме 32
- Количество портов 3 — *PORTB* (8 выводов), *PORTD* (8 выводов), *PORTC* (7 выводов).

Более подробную информацию о любой микросхеме можно найти в техническом описании (*datasheet*, даташит) в интернете, например по ссылке

https://pdf1.alldatasheet.com/datasheet-pdf/view/392284/ATMEL/ATMEGA328-PU.html

Рассмотрим расположение выводов и их назначение (распиновку) рис. 7.1.

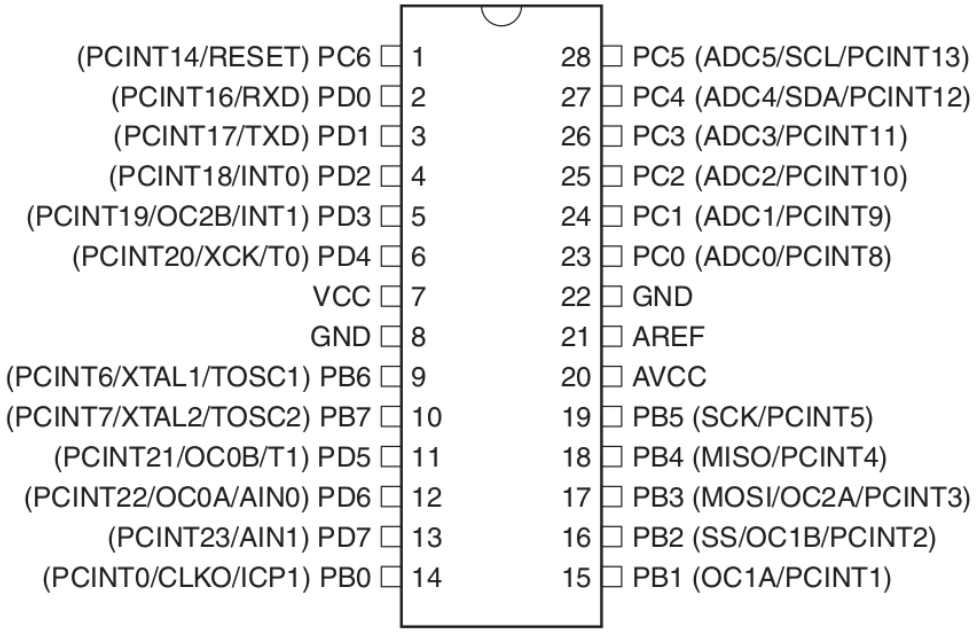


Рисунок 7.1 — Распиновка ATmega328P

Для удобства все ножки на рисунке пронумерованы, но на самой микросхеме надписи отсутствует. Вместо надписей имеется маленький кружочек рядом с ножкой под номером один. Все остальные выводы *ATmega328P* отсчитываются относительно первой ножки против часовой стрелки.

Найдем на микросхеме интересующие нас выводы.

- *Ножка 7* — **`Vcc`** — положительное напряжение питания +5 В.
- *Ножка 8*, ножка 22 — **`GND`** — общий провод (земля)
- *Ножки 2, 3, 4, 5, 6, 11, 12, 13* — восемь выводов порта *D*, к которым будут подключаться светодиоды.
- *Ножка 14* — вывод **`PB0`** порта *B*, к данному входу подключим кнопку включения гирлянды.
- *Ножки 17, 18, 19, 1* — **`MOSI`**, **`MISO`**, **`SCK`**, **`RESET`** — выводы, через которые подключается программатор *USBasp* (по последовательному интерфейсу).

Шаг 2.2 Знакомство с программатором USBasp

Программатор представляет собой плату с разъемом ISP. В разъем установлен ISP кабель. Если кабеля нет в комплекте, можно его прикупить, либо использовать перемычки *Male-Female. Female* — означает розетка (отверстие), *Male* — означает вилка (штырь). Распиновка *USBasp* представлена на рис. 7.2

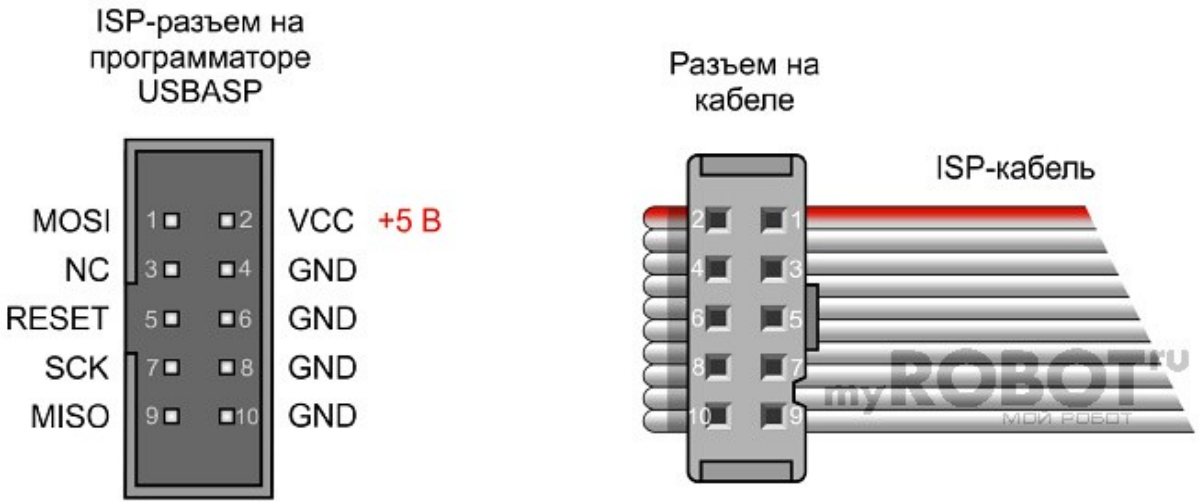


Рисунок 7.2 — Распиновка выводов программатора.
Картинка взята с сайта https://myrobot.ru/stepbystep/mc_programmer.php, там же можно посмотреть подробную информацию о USBasp.

В качестве тренировки попробуйте проверить назначение выводов с использованием мультиметра. Задача — найти вывод **Vcc**. Расположение остальных выводов легко определить относительно **Vcc** (они подписаны на плате).

1. Подсоедините черный измерительный щуп к входу **COM** мультиметра (**COM** от англ. *common* — общий) — это земля. Красный щуп подключите к входу **V** мультиметра (не перепутайте с входом **ADC**!!). Такое подключение позволяет измерить напряжение.

Не используйте мультиметр для измерения напряжения в розетке и где-бы то ни было еще, если не обладаете необходимой квалификацией!!!

2. Переведите мультиметр в режим измерения постоянного напряжения с пределом **20 Вольт**, для этого установите переключатель в положение **DCV 20**.

3. Подсоедините гибкие перемычки к тем выводам штекера на конце шлейфа *USBasp*, которые на ваш взгляд могут являться **Vcc** и **GND**.

4. Подключите программатор к USB порту компьютера.

5. Соедините красный щуп с **Vcc** программатора, а черный с **GND** программатора.

6. Если на индикаторе мультиметра появилось значение 5В, значит выводы определены верно. Если напряжение отрицательное -5В, значит, то что вы считали за **Vcc** на самом деле **GND**. А **GND** на самом деле **Vcc**. Если на индикаторе ноль или другое значение, проверьте все соединения и перейдите к п. 3.

Подсоедините гибкие перемычки к программатору и приклейте к ним (скотчем) маленькие клочки бумаги с подписями **Vcc**, **GND**. Аналогично установите в штекер программатора и промаркируйте перемычки **MOSI**, **MISO**, **SCK**, **RST**. Вывод NC not connected, судя по названию, не используется.

При программировании перемычки от программатора подключаются к соответствующим выводам *ATmega328P*.

Шаг 2.3 Установка драйвера USBasp

Установка драйверов для *USBasp* дело не хитрое. Набираем в гугле **USBasp Windows 10** (ваша версия ОС) скачиваем и устанавливаем.

Установку и настройку для Linux рассматривать не будем. Скорее всего там проблема с драйвером отсутствует, возможно установка драйвера даже не потребуется.

Шаг 3 Сборка экспериментальной установки

Составим электрическую схему гирлянды и соберем ее на макетной плате.

Шаг 3.1 Разработка электрической схемы

Схема подключения программатора и светодиода представлена на рис. 7.3.

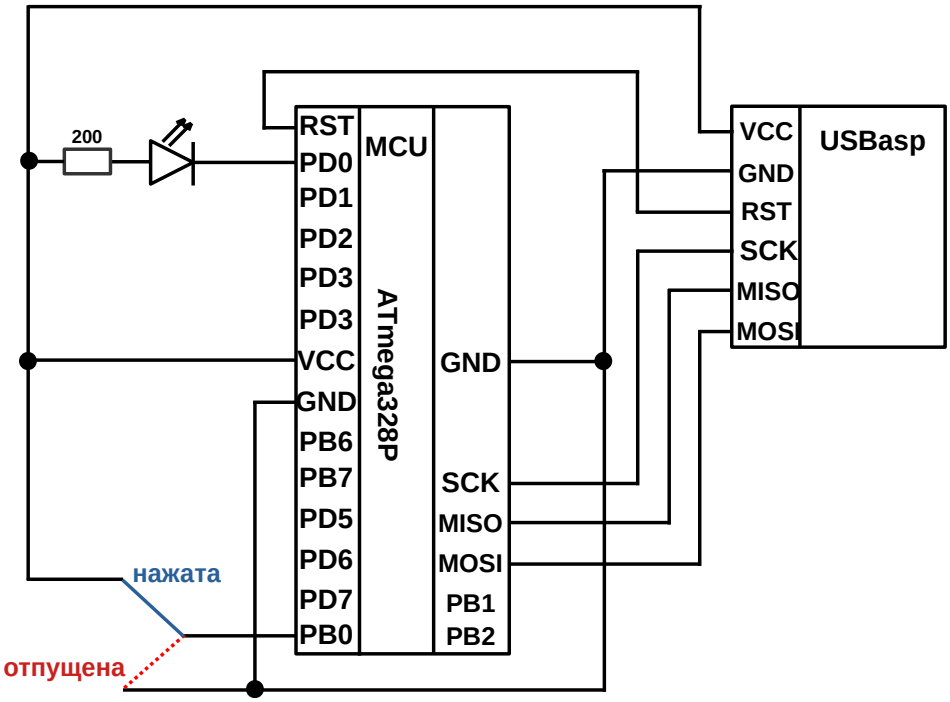


Рисунок 7.3 — Электрическая схема гирлянды и подключение программатора.

Немного проанализируем схему. Питание и земля на *ATmega328P* поступают от программатора. Обратите внимание, что земля от программатора соединена с двумя земляными выходами **`GND`** микроконтроллера. Выводы программатора **`RST`**, **`SCK`**, **`MISO`**, **`MOSI`** соединены с соответствующими выводами *ATmega328P*.

Светодиод подключен через токоограничивающий резистор одним выводом к питанию, другим к входу **`PD0`** порта *D*. При **`PD0 = 0`** напряжение на светодиоде U = 5В - 0В = 5В и светодиод горит. В противном случае, если **`PD = 1`**, то

напряжение на светодиоде U = 5В - 5В = 0В и светодиод гаснет. Получается некая инверсия 0 — горит, 1 — гаснет.

Кнопка в нашем случае будет представлена перемычкой, которая подключается от вывода **PB0** либо к **Vcc** (кнопка нажата), либо к **GND** (кнопка отпущена). Контакты в «воздухе» лучше не оставлять, поскольку там «летает напряжение». Если напряжение на выводе должно быть нулевым, соединяйте его с землей **GND**.

Шаг 3.2 Основы монтажа на макетной плате

Для удобства сборки прототипов и моделирования различных устройств используются беспаячные макетные платы. На них можно разместить интересующие элементы и, используя провода-перемычки, осуществить между ними электрический контакт.

Макетная плата представляет собой прямоугольную область с множеством отверстий - **разъемов**. Плата делится вдоль на два сектора. Иногда вдоль и поперек на четыре сектора (зависит от платы) рис. 7.4.

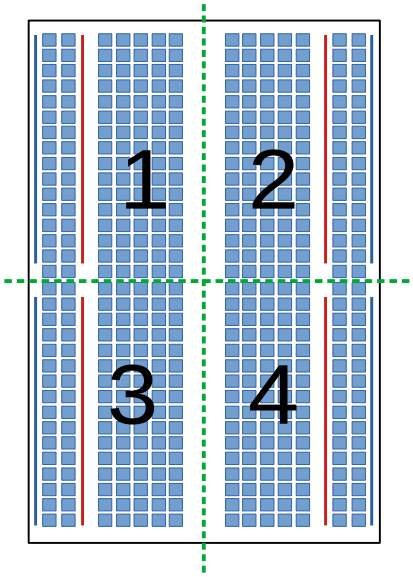


Рисунок 7.4 — Сектора на макетной плате.

1. **Шина питания.** Длинные ряды по бокам — это шины питания и земли. В них все разъемы соединены между собой. Если подать напряжение 5 В через один разъем,

то 5 В появится на всех остальных разъемах шины вдоль красной полосы вплоть до разрыва рис. 7.5.

Обратите внимание, что левая и правая шины не соединены между собой (они находятся в разных секторах). Часто присутствует разрыв шины и посередине платы. При необходимости подать питание на все четыре сектора можно перемычками соединить вертикальные разрывы, а левую шину с правой. С землей дело обстоит точно также.

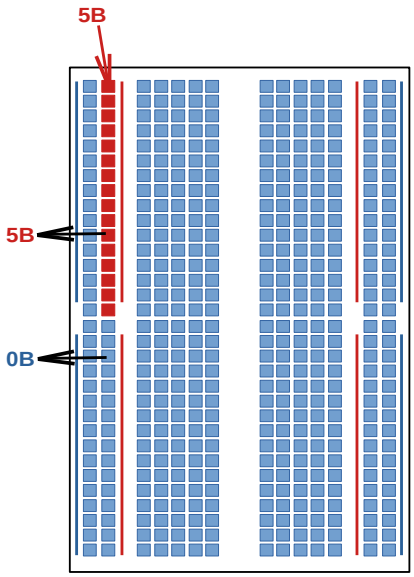


Рисунок 7.5 — Шина питания на макетной плате

2. **Основная часть**. Здесь все куда интереснее. В каждом секторе имеется пять столбцов и много строк. Разъемы в каждой строке соединены между собой рис. 7.6a. Разъемы в соседних строках изолированы. Поэтому все элементы необходимо располагать «вертикально» рис. 7.6a. В противном случае, вы просто соедините контакты элемента между собой, цепь не будет замкнута, ток не пойдет, произойдет короткое замыкание (КЗ, КОЗА).

Поскольку пять разъемов в каждой строке соединены между собой, возможно соединить два элемента перемычкой рис. 7.6b.

Микроконтроллер устанавливается по-середине. Таким образом, выводы левой и правой стороны *ATmega328P* размещаются в разных секторах, не связанных между собой рис. 7.6c.

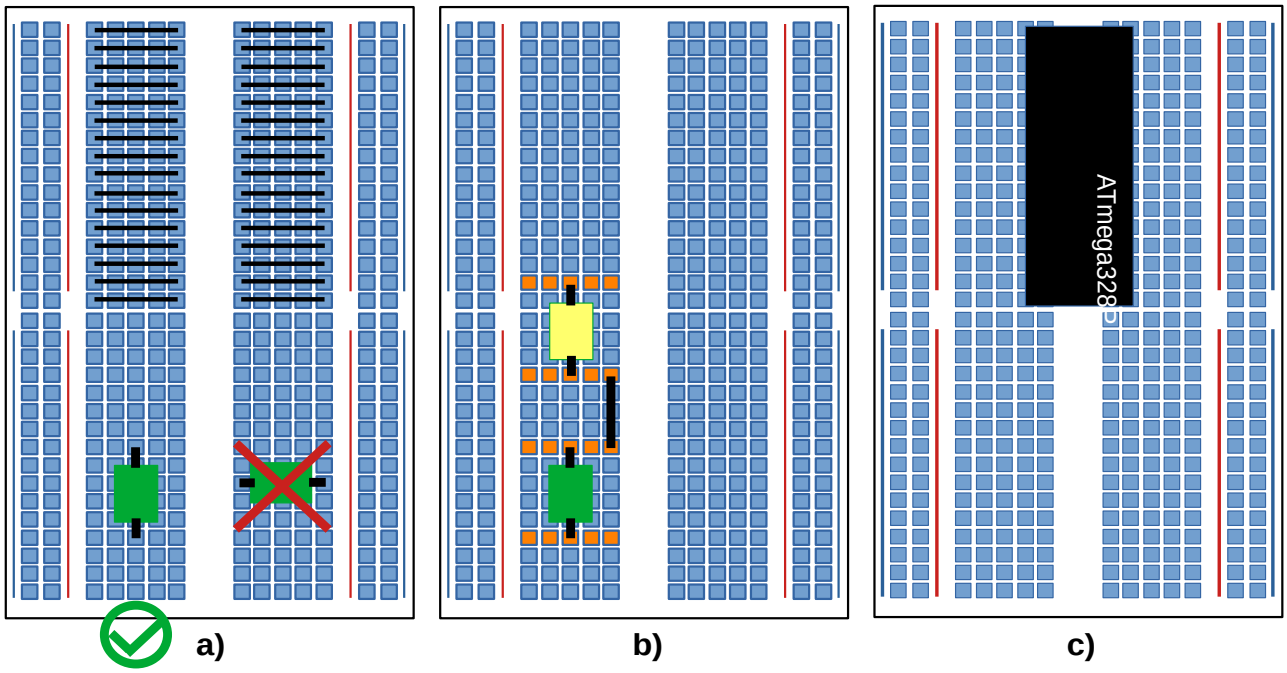


Рисунок 7.6 — Расположение элементов на плате: a — правильное и ошибочное расположение элементов на плате (разъемы в строках соединены между собой); b — соединение элементов друг с другом перемычкой; c — размещение микроконтроллера, левые и правые выводы расположены в разных секторах.

Шаг 3.3 Сборка схемы на макетной плате

Соберите схему (рис. 7.3) на макетной плате. Обратите внимание, что светодиод длинной ножкой подключается к **Vcc**, а короткой к микроконтроллеру. Это важно, иначе он не загорится.

Проверьте несколько раз, особенно подключение питания **Vcc** и земли **GND**. Небольшая подсказка. Перемычки везде использовать необязательно. Например, резистор можно подключить одной ногой к шине питания, а другой непосредственно к длинной ножке светодиода. В итоге получится нечто, показанное на рис. 7.7.

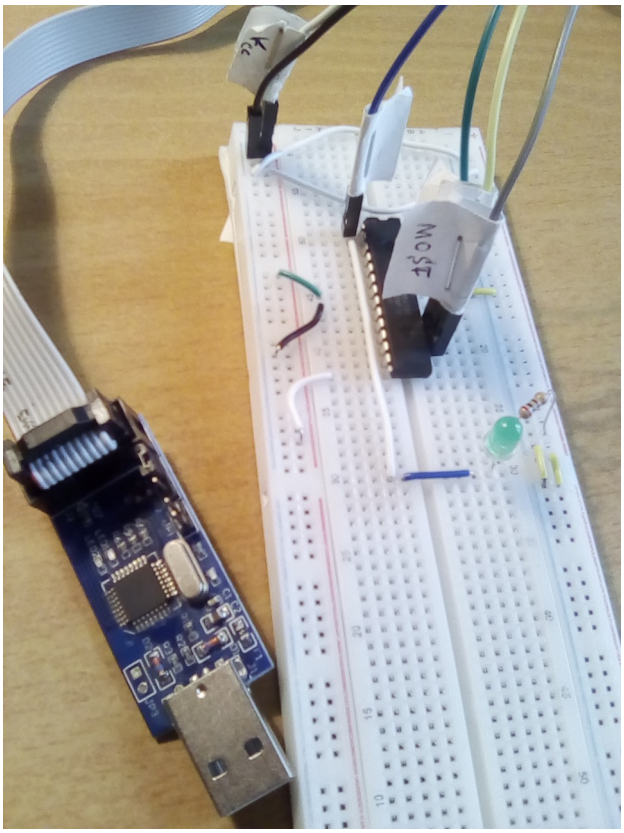


Рисунок 7.7 — Фотография схемы, собранной на макетной плате.

Шаг 4 Настройка программной и аппаратной среды

Найдем и скачаем необходимое программное обеспечение и настроим его “под себя”. Всего-то необходимо заполучить две программы **avrasm2** и **avrdude**. Далее запрограммируем FUSE-биты Но для начала создадим папку **asm** в корне диска **C:/asm**, которая будет является средой для наших экспериментов с микроконтроллером.

Шаг 4.1 Установка транслятора avrasm2

Транслятор необходим, чтобы переделать код, написанный словами на языке ассемблера (файл **.asm**), в двоичный код, который будет прошиваться в микроконтроллер.

К сожалению, фирма разработчик не предоставляет этот транслятор отдельно (хотя на неофициальных сайтах его все же можно найти). **Avrasm2** включен в состав **Atmel Studio**, которую можно скачать с официального сайта Microchip https://www.microchip.com/mplab/avr-support/atmel-studio-7. Программа очень большая, нам потребуется только одна папка **avrassembler**. Она располагается в директории, в которую вы установите **Atmel Studio** (скорее всего **C:/ProgramFiles(x86)/Atmel/..**). Поищите внимательнее и обязательно найдете. Скопируйте папку целиком в директорию **C:/asm**

Проверьте, что в папке **include** имеется файл описания **m328pdef.inc**.

Создайте файл **translate.cmd** со следующим скриптом (I — это буква ай заглавная):

```
C:/asm/avrassembler/avrasm2.exe -I C:/asm/avrassembler/include -fI C:/asm/my_prog.asm
```

и сохраните файл в рабочую директорию **C:/asm/translate.cmd**

Данный скрипт запускает транслятор, передает ему в качестве параметра каталог, в котором содержатся файлы описания и файл с исходным кодом программы **my_prog.asm**, который подвергнется трансляции.

Шаг 4.2 Установка программы прошивки avrdude

Здесь просто переходим по ссылке https://yadi.sk/d/lol-f_MgsbGOEw и извлекаем содержимое в каталог **C:/asm**. (либо используем http://google.com и поисковый запрос **скачать avrdude**).

Создайте файл **write.cmd** со скриптом:

```
C:/asm/avrdude/avrdude.exe -p m328p -c usbasp -e -U flash:w:my_prog.hex
```

и сохраните файл в рабочую директорию **C:/asm/write.cmd**

Данный скрипт запускает процесс прошивки микросхемы **m328p** через программатор *usbasp*, запись осуществляется во *FLASH* память (память программ), прошивается файл **my_prog.hex**.

Таким образом, к файлам **avrasm2.exe** и **avrdude.exe** должны получиться следующие пути:

C:/asm/avrassembler/avrasm2.exe **C:/asm/avrdude/avrdude.exe**

А к файлам **translate.cmd** и **write.cmd:**

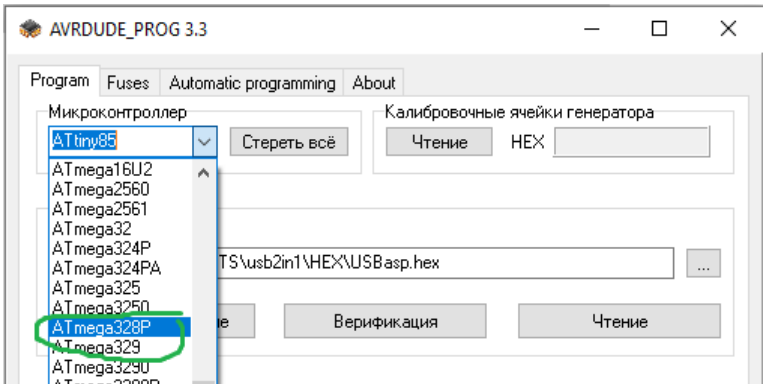
C:/asm/translate.cmd **C:/asm/write.cmd**

Шаг 4.3 Программирование FUSE-битов

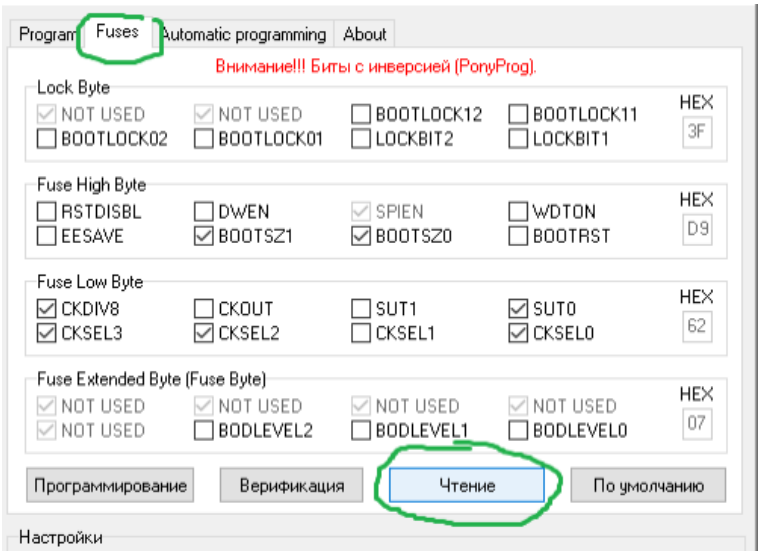
FUSE-биты — это настроечные биты, определяющие режим работы микроконтроллера. Их необходимо установить один раз, перед тем как начать многочисленные прошивки и эксперименты с *ATmega328P*.

1. Подключаем программатор *USBasp* к USB-порту компьютера (с *ATmega328P* программатор уже соединен через последовательный интерфейс).

2. Открываем программу **C:/asm/avrdude/AVRDUDEPROG.exe** и выбираем из списка микроконтроллер **ATmega328P**.



3. Переходим на вкладку **FUSES** и жмем чтение



4. Устанавливаем **FUSE** следующим образом:

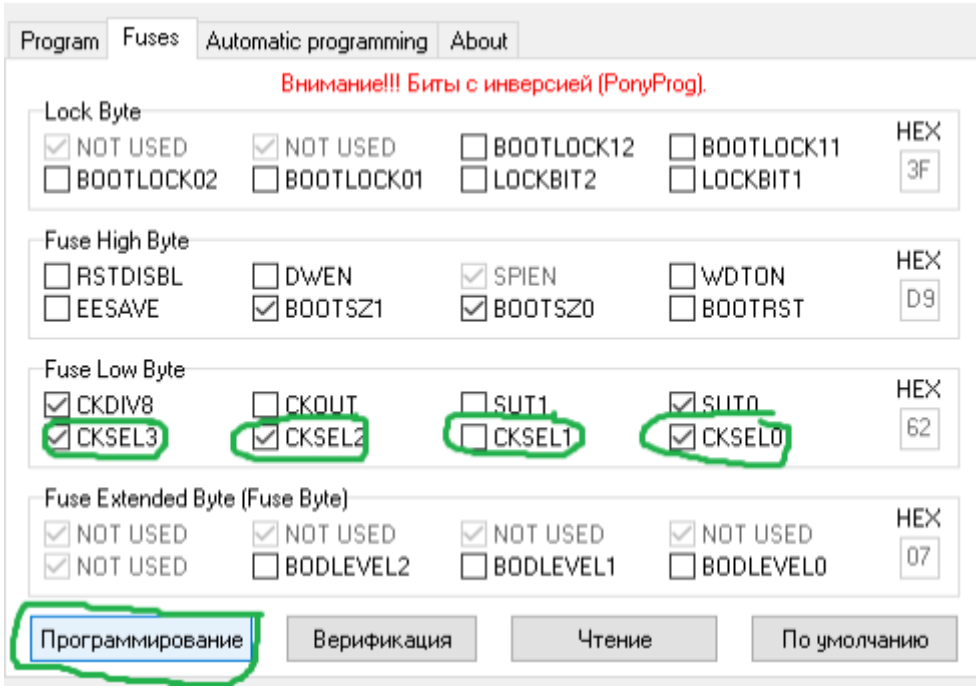
CKSEL3 — есть галочка

CKSEL2 — есть галочка

CKSEL1 — галочка отсутствует

CKSEL0 — есть галочка

и жмем **Программирование**.



Больше FUSE не трогаем, пока не возьмем другую микросхему.

Шаг 5 Трансляция и прошивка программы

Сперва рассмотрим код программы, которую будем прошивать. Далее осуществим его трансляцию и прошивку в *ATmega328P*.

Шаг 5.1 Написание кода

Хотим написать программу, реализующую следующий функционал: при нажатии на кнопку (**PB0** соединен с **Vcc**) светодиод мигает, если кнопка отжата (**PB0** соединен с **GND**) — светодиод не горит.

Опустим подробный разбор алгоритма и особенности *ATmega328P*. Просто распишем назначение каждой из команд, представленной в листинге. Далее, шестнадцатеричные числа начинаются с символа $, а не #, как было прежде.

Создаем файл C:/asm/my_prog.asm и печатаем в нем следующий код (для удобства написания помимо Блокнота можно использовать Geany или Notepad++, только не используйте Word или WordPad, это должны быть простые редакторы).

```
; ********************************************
; **************Мигающий светодиод**************
; ********************************************

.include "m328pdef.inc" ; файл с описанием констант

.list     ; включение листинга кода
.cseg     ; запись в память программ
.org 0    ; установка текущего адреса на ноль

; ------------- Инициализация стека -------------

     ldi  R16, HIGH(RAMEND)   ; запись в регистр R16 старшего байта адреса конца оперативки
     out  SPH, R16            ;  установка  вершины  стека  (старший байт адреса)
     ldi  R16, LOW(RAMEND)    ; запись в регистр R16 младшего байта адреса конца оперативки
     out  SPL, R16            ;  установка  вершины  стека  (младший байт адреса)
```

```
; ------------- Настройка портов -------------
     ldi R16, $00   ; запись в регистр R16 значения #00
     out DDRB, R16  ; порт B на ввод (к нему подключена кнопка)
     ldi R16, $FF
     out DDRD, R16  ; порт D на вывод (к нему подключен светодиод)
     out PORTB, R16 ; чтобы работала кнопка
     out PORTD, R16 ; записываем $FF в порт D (тушим лампочку)

; ------------- Основной цикл программы -------------
main:
     sbi  PORTD, 0  ; тушим светодиод
     in   R16, PINB ; читаем содержимое порта B
     sbrs R16, 0    ; пропуск следующей команды, если PB0 = 1
     rjmp main      ; если ноль, переходим в начало

; ------------- PB0 = 1 => светодиод мигает -------------
     cbi PORTD, 0   ; PD0 = 0 => светодиод горит
     rcall wait     ; Вызов подпрограммы задержки
     sbi PORTD, 0   ; PD0 = 1 => светодиод гаснет
     rcall wait     ; Вызов подпрограммы задержки
     rjmp main      ; переход в начало

; ------------- Подпрограмма задержки -------------
wait:
     PUSH R17
     PUSH R18
     PUSH R19
     ldi  R17, $03 ; помещаем константу задержки для внешнего цикла
в R17
     loop_ext:
     dec  R17       ; уменьшение R17 на единицу
     breq end_wait  ; если dec R17 привел к нулю переход к end_wait
     ldi  R18, $FF  ;  помещаем  константу  задержки  для  среднего
цикла в R18
loop_mid:
     dec  R18
     breq loop_ext
     ldi  R19, $FF
loop_int:
     dec  R19
     brne loop_int
     rjmp loop_mid
end_wait:
     POP  R19
     POP  R18
     POP  R17
     ret
```

`.include "m328pdef.inc"` — подключение файла с описанием констант для микросхемы *ATmega328P*;

`.list` — включение листинга (сообщения об ошибках в процессе трансляции);

`.cseg` — выбор сегмента память (`.cseg` означает *FLASH* — память программ);

`.org` — указатель текущего адреса (текущее положение в сегменте памяти `.cseg`). Указываем **`0`**, чтобы программа располагалась с самого начала памяти (с ячейки с адресом #**`0000`**).

Представленные 4-ре команды не являются командами *ATmega328P*, это так называемые инструкции препроцессора — управляющие слова для транслятора. Анализируя их, транслятор понимает, как ему транслировать код.

Следующие инструкции уже входят в набор команд *ATmega328P*.

- `ldi Rrr, $dd` — команда записи числа `$dd` в регистр `Rrr;`
- `out xxx, Rrr` — запись значения из регистра `Rrr` в регистр порта `xxx;`
- `in Rrr, PINx` - чтение из порта `x` в регистр `Rrr;`
- `sbi PORTx, d` — установка вывода `d` порта `PORTx` в единицу (*Set Bit Immediately*);
- `cbi PORTx, d` — сброс вывода d порта `PORTx` в ноль (*Clear Bit Immediately*);
- `sbrs Rrr, x` – пропуск следующей команды, если разряд `x` регистра `Rrr` равен **`1`**;
- `dec Rrr` — уменьшение на **`1`** значения в регистре **`Rrr`** (декремент);
- `breq label` — переход к метке `label`, если `Z = 1;`
- `brne label` — переход к метке `label`, если `Z = 0.`

Шаг 5.2 Трансляция

Каждый раз перед трансляцией сохраняйте файл **`my_prog.asm`**.

Открываем консоль, для этого на клавиатуре жмем сочетание клавиш **`Win+R`** (либо **`ПУСК - Выполнить`**), вводим `cmd` и жмем **`ENTER`**.

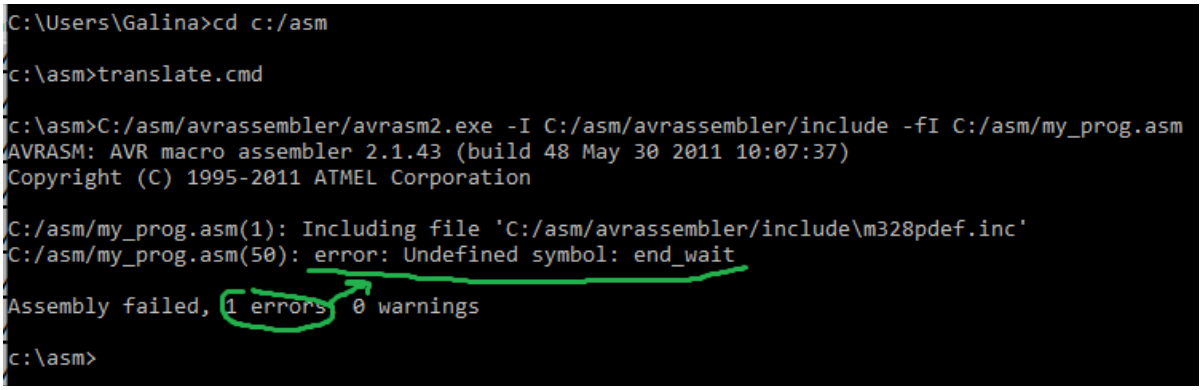
1. Переходим в рабочий каталог, для этого введите

```
cd C:/asm
```

и нажмите **`ENTER`** (обратите внимание, что все буквы латинские).

2. Выполните команду `translate.cmd.`

Появится примерно следующий вывод.



Оказывается в коде имеется *1 ошибка*: **`Undefined symbol: end_wait`** (у вас будут свои ошибки :)). Для исправления ошибки, добавим забытую метку `end_wait:` в код **`my_prosg.asm.`** Помним о необходимости после изменений сохранять файл **`my_prog.asm`**.

Нажатие клавиш **вверх** и **вниз** позволяет перебирать команды, которые вы вводили ранее в консоль. Понажимайте **вверх**-**вниз** и выберите `translate.cmd`.

Транслятор выдаст следующую информацию, если трансляция прошла успешно:

```
c:\asm>translate.cmd

c:\asm>C:/asm/avrassembler/avrasm2.exe -I C:/asm/avrassembler/include -fI C:/asm/my_prog.asm
AVRASM: AVR macro assembler 2.1.43 (build 48 May 30 2011 10:07:37)
Copyright (C) 1995-2011 ATMEL Corporation

C:/asm/my_prog.asm(1): Including file 'C:/asm/avrassembler/include\m328pdef.inc'

ATmega328P memory use summary [bytes]:
Segment   Begin    End      Code   Data   Used    Size   Use%
---------------------------------------------------------------
[.cseg] 0x000000 0x000048     72      0     72   32768   0.2%
[.dseg] 0x000100 0x000100      0      0      0    2048   0.0%
[.eseg] 0x000000 0x000000      0      0      0    1024   0.0%

Assembly complete, 0 errors, 0 warnings
```

Проверьте, что появился файл прошивки с двоичным кодом **C:/asm/my_prog.hex**.

Шаг 5.3 Прошивка ATmega328P

Если трансляция прошла успешно, выполните команду `write.cmd`.

```
c:\asm>C:/asm/avrdude/avrdude.exe -p m328p -c usbasp -e -U flash:w:my_prog.hex

avrdude.exe: warning: cannot set sck period. please check for usbasp firmware update.
avrdude.exe: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% -0.00s

avrdude.exe: Device signature = 0x1e950f
avrdude.exe: erasing chip
avrdude.exe: warning: cannot set sck period. please check for usbasp firmware update.
avrdude.exe: reading input file "my_prog.hex"
avrdude.exe: input file my_prog.hex auto detected as Intel Hex
avrdude.exe: writing flash (74 bytes):

Writing | ################################################## | 100% 0.08s

avrdude.exe: 74 bytes of flash written
avrdude.exe: verifying flash memory against my_prog.hex:
avrdude.exe: load data flash data from input file my_prog.hex:
avrdude.exe: input file my_prog.hex auto detected as Intel Hex
avrdude.exe: input file my_prog.hex contains 74 bytes
avrdude.exe: reading on-chip flash data:

Reading | ################################################## | 100% 0.08s

avrdude.exe: verifying ...
avrdude.exe: 74 bytes of flash verified

avrdude.exe: safemode: Fuses OK (E:07, H:D9, L:62)

avrdude.exe done.  Thank you.
```

Теперь светодиод мигает.

Если не мигает проверьте схему. Включена ли «кнопка». Доходит ли питание до кнопки и до светодиода. Подключен ли светодиод к **Vcc** своей длинной ножкой, а не короткой. Для измерения напряжения используйте мультиметр. Черный щуп подсоедините к **GND**, а красный к контакту, на котором хотите измерить напряжение, не забудьте перед этим установить переключатель мультиметра в положение **VDC 20**. Проверьте наконец уже код, может имеются опечатки.

При возникновении непонятных ошибок в консоли посетите сайт http://google.com/ :) или воспользуйтесь помощью друга, который имел опыт программирования на ассемблере.

А где гирлянда? Гирлянда представляет собой множество светодиодов. Просто подключите светодиоды через токоограничивающие резисторы к оставшимся семи выводам **PD7..PD1**.

Шаг 6 Бегущие огни

Если удалось подключить все 8-мь светодиодов и они замигали, попробуйте усовершенствовать программу и сделать «бегущие огни». Огонек очень красиво будет перемещаться от начала линии светодиодов к ее концу. Для этого в порт необходимо выдавать через задержку следующие значения:

00000001b,
00000010b,
00000100b,
…
10000000b

Знакомая операция? Да, это сдвиг влево. К счастью в наборе команд *ATmega328P* имеется соответствующая команда:

`LSL Rd` — арифметический сдвиг влево значения в регистре `Rd`.

Кстати, всего регистров тридцать две штуки R0..R31, но лучше использовать регистры с номером 16 и выше. Регистры R0..R15 не поддерживают команды обмена информацией с регистрами ввода-вывода.

Потребуется еще одна из команд условного перехода на ваш выбор:

- `BRCS label` – осуществляет переход к метке `label` если был перенос;

  либо

- `BRCC label` – осуществляет переход к метке `label` если не было переноса.

Из литературы советую начать с книги *Белов А.В. «Микроконтроллеры AVR: от азов программирования до создания практических устройств»*. Приведенный пример «гирлянды» был взят отчасти именно из этой книги.

Эпилог

Таким образом, была предпринята попытка показать путь от электрона до программирования. Что теперь с этим делать, решать вам.

В любом случае, мы стали на уровень выше в понимании процессов, которые так прочно и еле заметно производят подмену истинных ценностей.

Виноваты не микропроцессоры... Нет. Это всего лишь инструмент, который можно применить как во вред, так и во благо. И если вы выберите путь программиста, хочется верить, что этот программист через «высокие» низкие технологии сделает мир чуточку лучше.

Продуктивного программирования!

Оглавление

Пролог....5
О чем?....5
Что дальше?....6
Посвящается....7
Раздел 1. Физика....8
Глава 1.1 Основные физические понятия....8
1.1.1 Поле....8
1.1.2 Принцип неопределенности Гейзенберга....12
Глава 1.2 Основы физики полупроводников....15
1.2.1 Классификация материалов....15
1.2.2 Собственный полупроводник....16
1.2.3 Донорный полупроводник....20
1.2.4 Акцепторный полупроводник....21
1.2.5 Контакт акцепторного и донорного полупроводника....23
1.2.6 Прямое смещение p-n перехода....27
1.2.7 Обратное смещение p-n перехода....28
Глава 1.3 Транзистор....29
1.3.1 Полевой транзистор....29
1.3.2 Зажигание лампочки....32
1.3.3 Биполярный транзистор в качестве ключа....35
Глава 1.4 Устройство базисных логических элементов....39
1.4.1 НЕ....39
1.4.2 И-НЕ....40

1.4.3 ИЛИ-НЕ....41
1.4.4 Физика переходит в логику....42
Раздел 2 Логика....46
Глава 2.1 Основные логические элементы....46
2.1.1 Операторная форма записи....46
2.1.2 Основные операторы алгебры логики....47
2.1.3 Конъюнкция....48
2.1.4 Дизъюнкция....49
2.1.5 Инверсия....50
2.1.6 Схемы основных логических элементов....51
Глава 2.2 Исключающее ИЛИ....54
Глава 2.3 Сумматор....57
2.3.1 Системы счисления....57
2.3.2 Как образуются числа....61
2.3.3 Сложение....61
2.3.4 Полусумматор....63
2.3.5 Полный сумматор....65
Глава 2.4 Декодер....68
Глава 2.5 Секвенциальная логика и память....73
2.5.1 RS – триггер....73
2.5.2 D-триггер....76
2.5.3 Параллельный регистр....79
Раздел 3. Арифметико-логическое устройство....81
Глава 3.1 Середина....83
Глава 3.2 Конец....90

Глава 3.3 Шина.....93
Глава 3.4 Начало.....95
Глава 3.5 Пример работы с АЛУ.....98
Раздел 4. Микропроцессорная авантюра.....103
Глава 4.1 Движущие элементы микропроцессора.....104
4.1.1 Счетчики.....104
4.1.2 Память энергозависимая.....106
4.1.3 Память энергонезависимая.....107
Глава 4.2 Элементы УИИ.....108
4.2.1 RAM.....110
4.2.2 ROM.....110
4.2.3 RGX.....111
4.2.4 PRT.....111
4.2.5 Stack Pointer.....112
4.2.6 Programm Counter.....113
4.2.7 Шина управления.....114
4.2.8 Такты.....114
Глава 4.3 Исполнение программы в УИИ.....115
4.3.1 Инструкции.....115
4.3.2 Включение МПА.....116
4.3.3 Инструкция FETCH.....118
4.3.4 Выполнение программы.....119
Глава 4.4 Основные элементы устройства управления.....120
4.4.1 Регистр команд.....120
4.4.2 Счетчик и тактовый генератор.....122

4.4.3 Регистр данных и адреса....123
4.4.4 Обработчик флагов....123
4.4.5 Память микрокоманд....124
4.4.6 Организация хранения микропрограмм в ROM-MP....125
4.4.7 FETCH....127
4.4.8 Выполнение извлеченной команды....136
Раздел 5. ASSEMBLER МПА....138
Важное дополнение....139
Глава 5.1 ASSEMBLER с первого взгляда....140
Глава 5.2 Набор команд....143
5.2.1 Инструкции записи данных....143
5.2.2 Инструкции копирования данных....144
5.2.3 Инструкция настройки порта....145
5.2.4 Инструкции АЛО....145
5.2.5 Инструкции переходов....145
5.2.6 Инструкции работы со стеком....147
5.2.7 Инструкции вызова подпрограммы....149
5.2.8 Инструкция FETCH....152
Глава 5.3 Пример программы....155
Глава 5.4 Прошивка....156
5.4.1 Микроконтроллеры и микропроцессоры....156
5.4.2 Трансляция и прошивка....157
Раздел 6. Алгоритмические конструкции....161
Глава 6.1 Умный дом (следование)....161
6.1.1 Схема подключения....161

6.1.2 Описание алгоритма словами....162
6.1.3 Настройка выводов порта....163
6.1.4 Инвертирование значения....163
6.1.5 Сдвиг 4-х младших битов в старшие разряды....164
6.1.6 Запись значения в порт....165
6.1.7 Полная версия программы....165
6.1.8 Подпрограмма....167
6.1.9 Метки....169
Глава 6.2 Холодильник (ветвление)....171
6.2.1 Схема подключения....171
6.2.2 Описание программы словами....172
6.2.3 Настройка порта....172
6.2.4 Получение состояния дверцы....173
6.2.5 Проверка дверцы (ветвление)....173
Глава 6.3 Утюг (циклы)....176
6.3.1 Описание программы словами....176
6.3.2 Настройка портов....178
6.3.3 Включение парового удара....179
6.3.4 Ожидание....179
Глава 6.4 Высокоуровневое программирование....185
6.4.1 Простой пример высокоуровневого кода....186
6.4.2. Компиляция....186
6.4.3 Интерпретация....189
6.4.4 Сравнение кода....190
Раздел 7. Пошаговое руководство....192

Шаг 1 Закупка комплектующих....193
Шаг 2 Знакомство....196
Шаг 2.1 Знакомство с ATmega328P....196
Шаг 2.2 Знакомство с программатором USBasp....198
Шаг 2.3 Установка драйвера USBasp....199
Шаг 3 Сборка экспериментальной установки....200
Шаг 3.1 Разработка электрической схемы....200
Шаг 3.2 Основы монтажа на макетной плате....201
Шаг 3.3 Сборка схемы на макетной плате....204
Шаг 4 Настройка программной и аппаратной среды....205
Шаг 4.1 Установка транслятора avrasm2....205
Шаг 4.2 Установка программы прошивки avrdude....206
Шаг 4.3 Программирование FUSE-битов....207
Шаг 5 Трансляция и прошивка программы....209
Шаг 5.1 Написание кода....209
Шаг 5.2 Трансляция....212
Шаг 5.3 Прошивка ATmega328P....213
Шаг 6 Бегущие огни....215
Эпилог....216