**Теоретические вопросы**

1. **История развития микропроцессоров.**

***1 ый микропроцессор I4004 появился в 1971 году***

*Наиболее успешный I8080*

***I4004*** *мог обрабатывать 4 Бита информации, 750 кГц, кристалл располагал ограниченные средства ввода/вывода, а в системе команд даже отсутствовали логические элементы.*

***I8008*** *мог адресовать память 16 Кбайт, кристалл имел 16000 транзисторов.*

*Новое в архитектуре - использование многоуровневой системы прерываний.*

*Появился механизм прямого доступа к памяти, который позволил использовать смежные внешние устройства.*

***В 1982*** *году появился I80286*

* *в нем появился защитный режим, что позволило обращаться к 16 Мбайтам ОЗУ*
* *в данном процессоре впервые на уровне микросхем были сделаны мультизадачность и управление виртуальной памятью*

***В 1985*** *году появился 32 разрядный микропроцессора, с тактовой частотой 16МГц*

*он позволял физически адресовать 4 Гбайта памяти + новый режим управления виртуальной память V86*

***I80386*** *изготовленный на 1 кристалле сопроцессором I80386DX*

*-использовал 16 разрядную шину адреса*

***В 1989 году появился процессор I80486DX***

*-1, 2 млн транзисторов*

*-микросхема впервые объединена на 1 кристалле центральный процессор, сопроцессор, кэш-память.*

*Встроенный сопроцессор существенно ускорял математические вычисления , однако в тоже время он использовался только в 30 % случаев*

***В 1991 году появился I40486***

*-20 МГц*

*-в этом семействе допускалась адресация 4 Гбайт физической памяти и 64 Гб виртуальной памяти.*

***в 1990 году 40486LX для ноутбуков***

*по производительности он уступал другим микропроцессорам, но благодаря пониженному напряжению питания мог использоваться в ноутбуках*

***в 1992 году I486DX2*** *в котором скорость работы ядра была в 2 раза выше остальной скорости части системы.*

*имел частоту 60 МГц*

***В 1995 году появились процессоры Pentium***

*Частота 60 МГц*

*Появилась Кэш-память*

*А позже поколение Р6*

1. **Архитектура микропроцессора. Типы архитектур. Микропроцессоры с этими архитектурами.**

*Определяет возможности микропроцессора по аппаратной, программной и микропрограммной реализации.*

*Основные характеристики микропроцессора:*

1. *Тактовая частота - это количество тактов в единицу времени (такт - операция)*
2. *Разрядность - это максимальное число одновременно обрабатываемых двоичных разрядов*
3. *Архитектура*

*Типы архитектур:*

1. *Микропроцессоры с гарвардской архитектурой - предполагает раздельные использования памяти программных данных. Большинство специализированных микропроцессоров (микроконтроллеров) имеет данную архитектуру*
2. *Микропроцессоры с фон Неймановской архитектурой - предполагается хранение программ и данных в общей памяти. Характерно для применения в ПК*
3. *Микропроцессоры типа CISC с полным набором команд*
4. *Микропроцессоры типа RISC с сокращенным набором команд*
5. *Микропроцессоры типа MISC с минимальным набором команд*
6. **Микропроцессоры типа CISC.**

*Конец 70х годов, память очень дорогая, компиляторы очень плохие, люди писали на ассемблере. Люди искаль способ минимизировать использование памяти, одно из решений использовать сложные инструкции процессора, которые делают многие действия.*

*Инструкции CISC могут быть любой длины, это помогало программистам на ассемблере, т.к они смогли писать более простые программы, как правило найдётся инструкция, которая выполнит то, что нужно. Через некоторое время это стало сложным, проектирование декодера для таких команд стало единственной проблемой.*

*Изначально её решили с помощью микрокода(небольшие программы имитирующие сложные инструкции) (процедуры функции заранее написанный код), таким образом процессор содержит небольшой набор простых инструкций, на их основе можно создать более сложные инструкции. Микрокод хранится в ROM-памяти(ПЗУ), которая значительно дешевле оперативной память(RAM), следовательно, уменьшается использование оперативной памяти через увеличения использования ПЗУ и является выгодным компромиссом. Затем начались проблемы с подпрограммами в микрокоде, а исправление ошибок гораздо сложнее, чем в обычной программе*

1. **Микропроцессоры типа RISK. Конвейеризация. Микрооперации.**

*Оперативная память стала дешеветь, компиляторы стали лучше, а большинство разработчиков перестали писать на ассемблере. Эти технические изменения и спровоцировали появление RISC. Сперва люди анализировали программы и заметили, что большинство сложных инструкций не использовались большинством программистов. Разработчики компиляторов предпочли использовать комбинации нескольких простых инструкций, т.к затруднились в выборе правильной сложной инструкции*

*CISC Y: (2\*3)+(1-2) RISC y1:= 2\*3 y2:=4-3 Y:= y1+y2*

*Одним из аргументов RISC было то, что люди перестали писать на ассемблере, потому что необходимо создать набор инструкций удобной для компиляторов. Один из плюсов RISC является появление конвейеризации. Есть 4 действия (аналогично покупке) проходила бы без конвейеризации то ,следовательно, покупатель смог бы поместить вещи на ленту только после того, как предыдущий заберёт покупки. Каждое действие занимает один такт (фиксированный промежуток времени) за 12 тактов будет обслужено 3 клиента без конвейеризации. Подключая конвейеризацию нужно шесть тактов на 3 человека.*

*Если время каждого этапа сильно варьируется, то это работает не так хорошо (если кто-то долго выкладывает продукты на ленту (1 действие), то зона терминальной (3 действие) простаивается.), поэтому старались стандартизировать время каждой инструкции, поэтому было деление на этапы, каждое из которых выполняется примерно одинаковое количество времени.*

*Процессор имел пятиступенчатый набор инструкций (конвейер инструкции)*

1. *излучение инструкции из памяти и увеличения счётчика команд, чтобы извлечь следующие инструкцию следующем такте*
2. *декодирование инструкция, то есть определить что это инструкция делает, т.е активация необходимых для выполнения этой инструкции частей микропроцессора*
3. *выполнение включает использование арифметического-логического устройства(АЛУ)*
4. *доступ к памяти, если необходимо*
5. *записи результатов в соответствующий регистр*

*Инструкция по технологии RISC удобно конвейеризировать, т.к инструкция состоит из секций, каждая из которых работает с одним из этих этапов о выполнении этапа занимает один так*т

1. **Структура типовой ЭВМ.**

**Структура типовой ЭВМ.**



Устройствоуправления подаёт во все блоки ПК управляющие сигналы, формирует адреса ячеек памяти и передаёт эти адреса в соответствующие блоки. Последовательность импульсов УУ получает от генератора тактовых импульсов.

Элемент АЛУ - все арифметико-логические операции над числовой информацией.

МПП(микропроцессорная память) - строится на регистрах, используется для обеспечения высокого быстродействия, т.к. основная память(ОЗУ и ПЗУ) не обеспечивает нужную скорость.

Интерфейсная система микропроцессора - это сопряжение с другими элементами ПК.

ГТИ(генератор тактовых импульсов) - генерирует электрические импульсы, частота которых является тактовой частотой процессора.

Таймер - электронные часы.

Контролер прерываний

Прерывание - это временное приостановление работы одной программы для выполнения другой более важной.

Системная шина обеспечивает сопряжение и связь всех устройств между собой. Она включает в себя:

1.шину данных

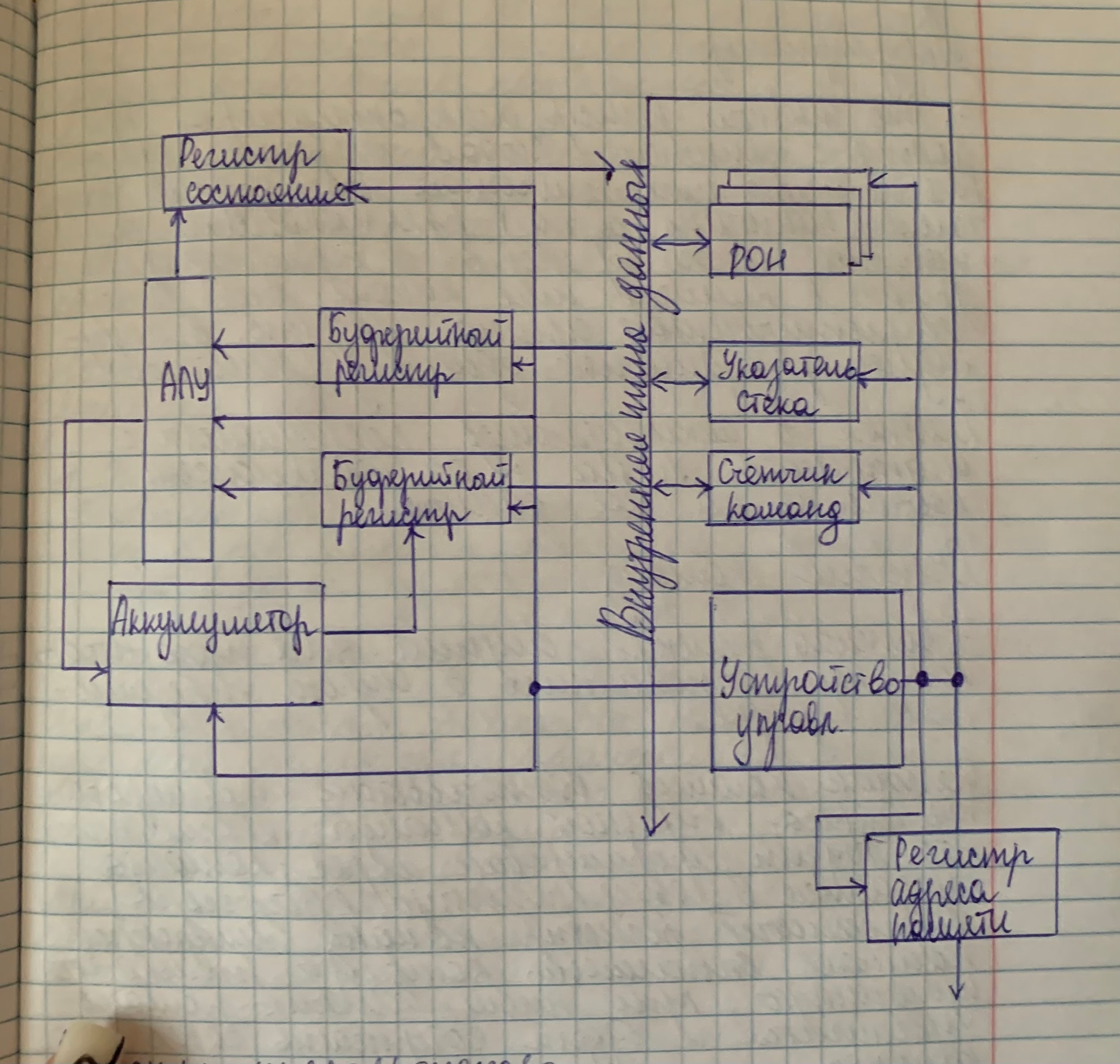
2.шину адреса

3.шину инструкций

4.шину питания

Все порты ввода/вывода подключаются к шине через контроллер.

1. **Внутренняя структура микропроцессора. Аккумулятор. Счетчик команд. Регистр команд. Регистр адреса.**



*Регистры микропроцессора.*

*Можно использовать для временного хранения данных, одни имеют специальное назначение, а другие многоцелевое, они называются РОН и могут использоваться программистом по усмотрению. Количество и назначение зависит от архитектуры микропроцессора.*

*Аккумулятор.*

*Это главный регистр микропроцессора при работе данными. Любая операция арифметическая или логическая над двумя элементами(данными и т.д.) предполагает размещение одного из них в аккумуляторе, а другого в памяти или каком-либо регистре. Результирующая сумма загружается в в аккумулятор, заменяя один из элементов.*

*Данные в аккумуляторе поступали с внутренней шина и он же может посылать на неё данные.*

*Счётчик команд - важный регистр служит для того, чтобы команды выполнялись в строго определённом порядке.*

*Счётчик команд обеспечивает формирование адреса очередной команды. После того, как будет сформирован адрес, команда передаётся в специальный регистр процессора, называемый регистр команд. Микропроцессор начнёт выполнять команду только что извлечённую, при этом счётчик команд увеличится и будет содержать адрес следующей команды.*

*Регистр адреса – это регистр содержащий адрес области памяти. Выход этого регистра называется адресной шиной и используется для выбора области памяти или порта ввода/вывода. В большинстве МП регистры адреса памяти и счетчики команд имеют одинаковое количество разрядов.*

1. **Внутренняя структура микропроцессора. Буферный регистр. Регистр состояния. РОН. Указатель стека.**

*Буферный регистр - предназначен для временного хранения данных.*

*Регистр состояния - предназначен для хранения результатов некоторых проверок, осуществляемых в процессе выполнения программы.*

*Регистр состояния предоставляет программисту возможность организовать работу микропроцессоров так, чтобы при определенных условиях менялся порядок выполнения команд.*

*РОН(регистр общего назначения)*

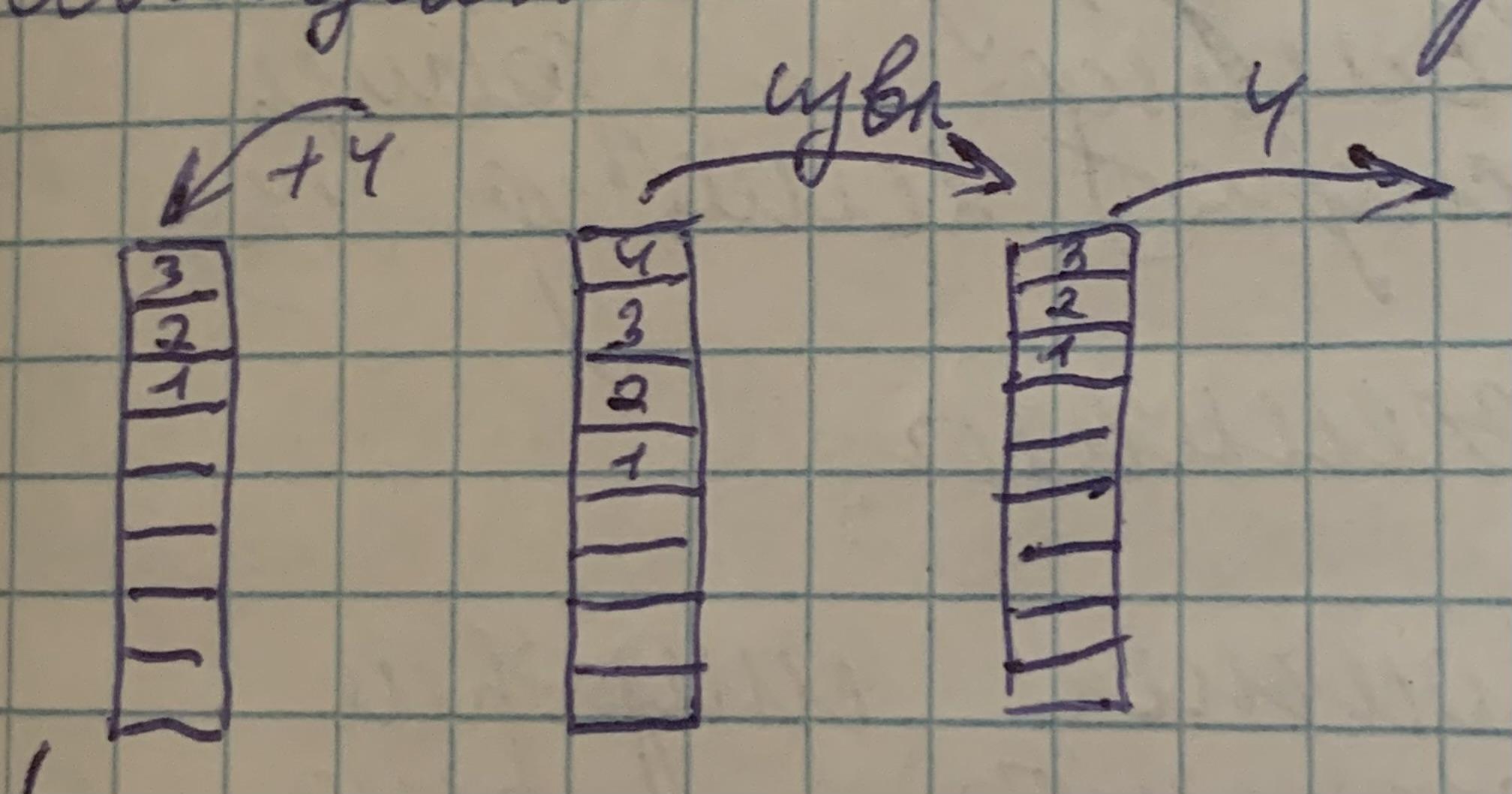
*К регистрам общего назначения (РОН) могут непосредственно адресоваться команды программы, поэтому каждый из них имеет собственные символьные наименования: В, С, D, Е, Н, L.*

*Так как РОН аппаратно выполнены в большой интегральной схеме (БИС) центрального процессора, команды обмена информацией с ними выполняются всего за 2 – 3 машинных такта, и поэтому РОН обладают более высоким быстродействием, чем любая другая память, к которой может обращаться микропроцессор. Иными словами, РОН выполняет функции сверхоперативного запоминающего устройства (СОЗУ).*

*Указатель стека.*

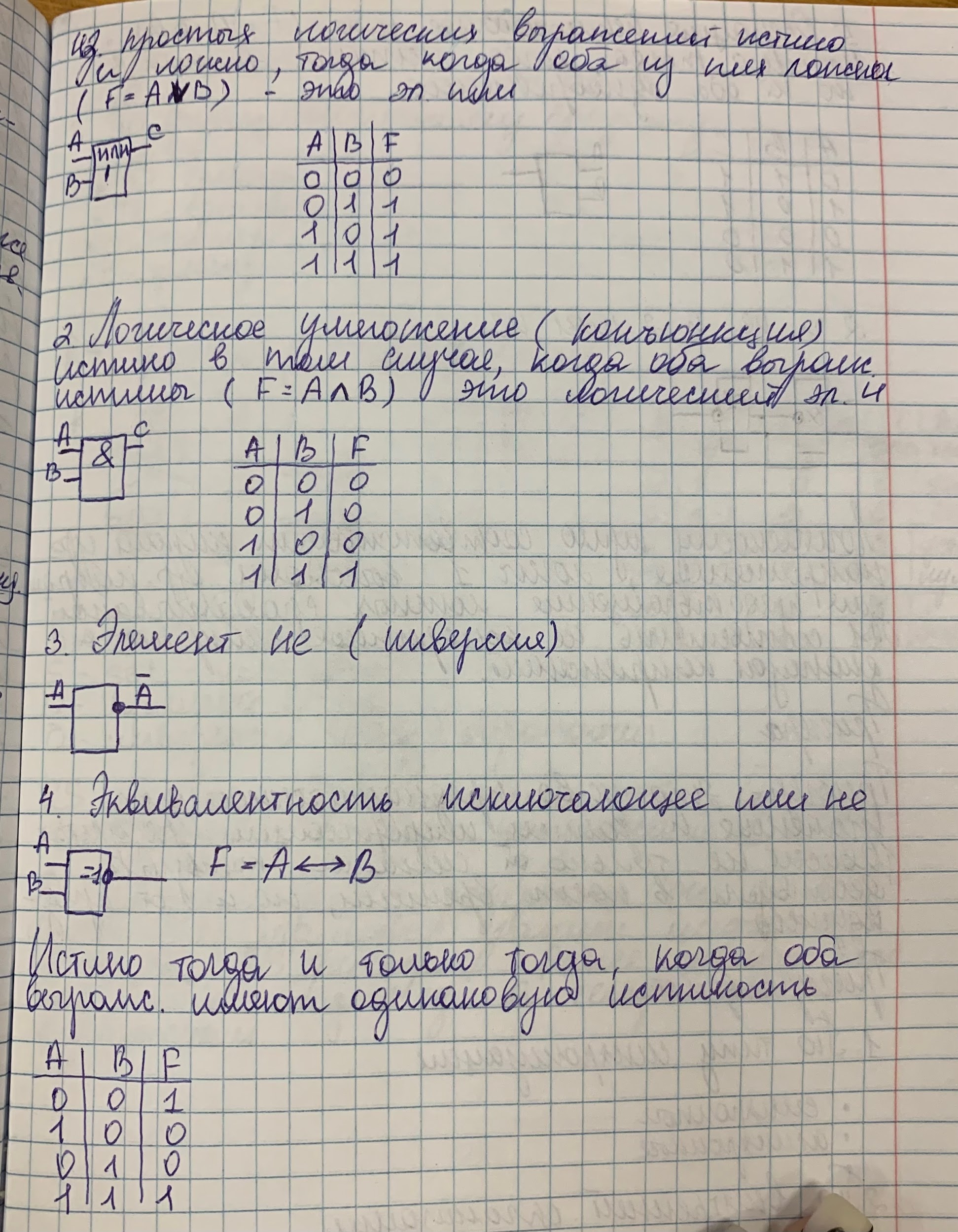
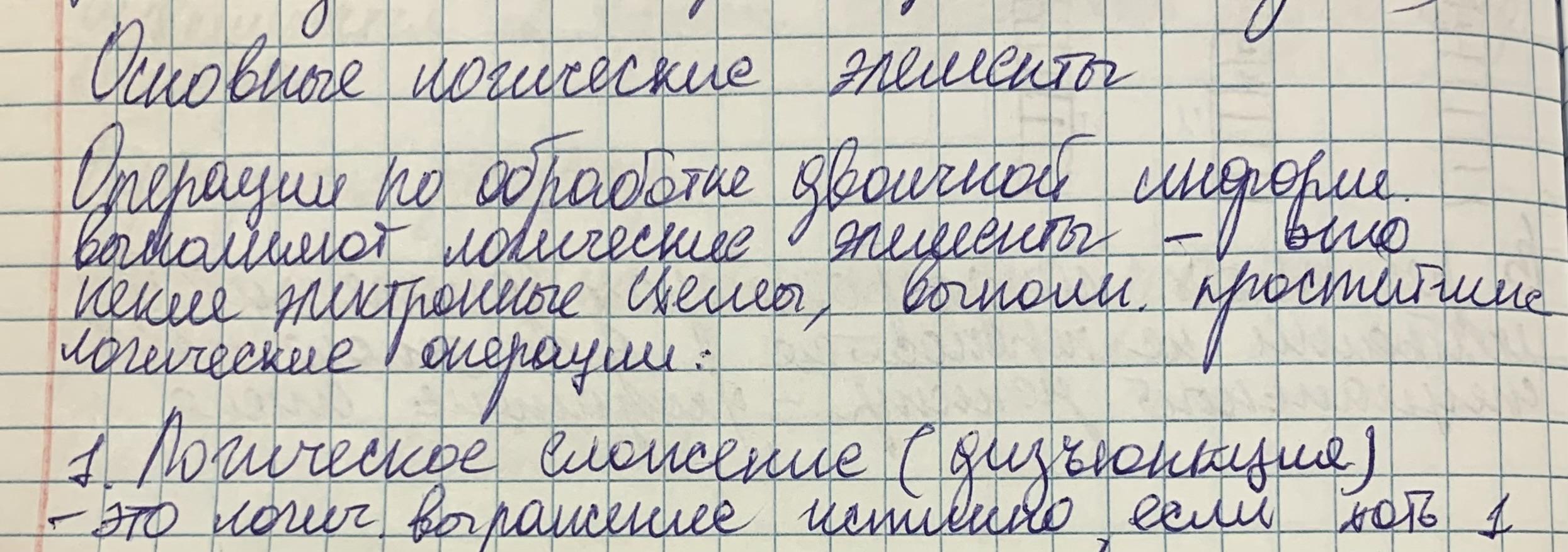
*Стек - это набор регистров МП или ячеек ОП, откуда данные или адреса выбираются «сверху» по принципу «первым - последний поступивший».*

*При записи в стек всё ранее записано смещается на 1 регистр вниз*



*В реальных МП данные между ячейками не переносятся, для этого используется указатель стека.*

1. **Логические элементы. Дизъюнкция. Конъюнкция. Логическое следование. Эквивалентность.**



1. **Триггеры. RS-триггер. Назначение. Схема. Таблица истинности.**

*Предназначены для хранения информации*

*Асинхронный*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  | | --- | --- | --- | --- | --- | | *R* | *S* | *Qпр* | *Q* | *Режимы* | | *0* | *0* | *0* | *0* | *Хранение* | | *0* | *0* | *1* | *1* | | *0* | *1* | *0* | *1* | *Установка 1* | | *0* | *1* | *1* | *1* | | *1* | *0* | *0* | *0* | *Установка 0* | | *1* | *0* | *1* | *0* | | *1* | *1* | *0* | *\** | *Запрет* | | *1* | *1* | *1* | *\** | |

*Синхронный*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | *C* | *R* | *S* | *Qпр* | *Q* | *Режимы* | | *0* | *x* | *x* | *0* | *0* | *хранение информации* | | *0* | *x* | *x* | *1* | *1* | | *1* | *0* | *0* | *0* | *0* | *хранение информации* | | *1* | *0* | *0* | *0* | *1* | | *1* | *0* | *1* | *0* | *1* | *установка 1* | | *1* | *0* | *1* | *1* | *1* | | *1* | *1* | *0* | *0* | *0* | *установка 0* | | *1* | *1* | *0* | *1* | *0* | | *1* | *1* | *1* | *0* | *\** | *Запрещённый комбинация* | | *1* | *1* | *1* | *1* | *\** | |

1. **Триггеры. T-триггер. Назначение. Схема. Таблица истинности.**

*Для хранения информации и счетного режима*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *на основе RS* | |  |  |  |  | | --- | --- | --- | --- | | *T* | *Q* | *Q* | *Режимы* | | *0* | *1* | *1* | *Хранение* | | *0* | *0* | *0* | | *1* | *1* | *0* | *Переброс* | | *1* | *0* | *1* | |
|  |
| *на основе D* |
|  |

1. **Триггеры. D-триггер. Назначение. Схема. Таблица истинности.**

*Для задержки (более удобно хранит данные)*

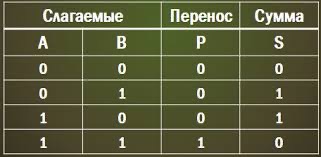
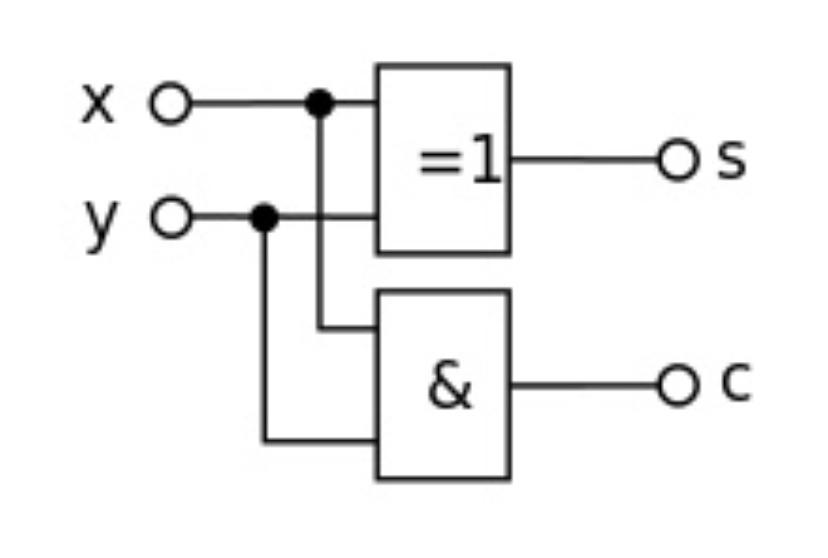
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  | | --- | --- | --- | --- | --- | | *C* | *D* | *Qпр* | *Q* | *Режимы* | | *0* | *X* | *Qпр* | *Qпр* | *Хранение* | | *1* | *1* | *X* | *1* | *установка 1* | | *1* | *0* | *X* | *0* | *установка 0* | |

1. **Сумматор. Полусумматор. Назначение. Схема. Таблица истинности.**

Полусумматор.

Полусумма́тор — комбинационная логическая схема, имеющая два входа и два выхода. Полусумматор позволяет вычислять сумму A+B, где A и B — это разряды обычно двоичного числа, при этом результатом будут два бита S и C, где S — это бит суммы по модулю 2, а C — бит переноса.

Не рассматривает переход из предыдущего разряда.

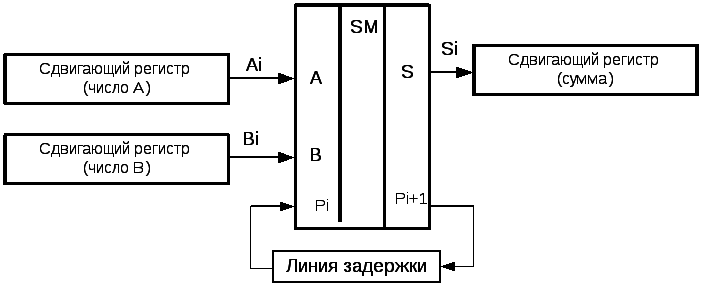


1. **Сумматор. Последовательно-многоразрядный сумматор. Назначение. Схема. Таблица истинности.**

**Сумматор -** логический операционный узел, выполняющий арифметическое сложение кодов двух чисел. При арифметическом сложении выполняются и другие дополнительные операции: учет знаков чисел, выравнивание порядков слагаемых и т.п. Указанные операции выполняются в АЛУ или процессорных элементах, ядром которых являются сумматоры.

## Последовательный многоразрядный сумматор

Строится на базе одноразрядных. Состоит из элемента задержки, сдвигающих регистров(доя хранения исходных чисел и результатов)



Исходные числа подаются по тактам, разряд за разрядом. Начиная с младшего разряда и сдвигающих регистров, а сформированная сумма накапливается в сдвигающемся регистре суммы. Возникающий перенос задерживается на один так и поступает на вход уже со следующим тактом в элементе задержки

1 такт с Аi подается 1, а с Вi 0 в регист суммы поступает 1

2 такт с Аi подается 0, а с Вi 1 в регистр суммы поступает 1

3 такт с Аi и Вi подается на 1 в регист суммы поступает 0, а в элемент задержки 1

4 такт с Аi и Вi подаются 1 в регистр суммы поступает 1, а в элемент задержки 0

Последовательный многоразрядный сумматор состоит из одноразрядного сумматора на входы а и в которого из сдвигающих регистров, в которых хранятся n-разрядные числа А и В, подаются по тактам разряд за разрядам коды этих чисел, начиная с младшего разряда.

Сформированная сумма накапливается в сдвигающем регистре суммы. Возникающий перенос с задержкой на элементе задержки на один такт поступает на вход сумматора только в следующем такте, когда на входы а и в будут поданы следующие разряды чисел А и В.

Схема последовательного сумматора представлена на Рис.9.4.

Достоинством последовательного сумматора является простота схемы, требующая минимального количества оборудования, недостатком – низкое быстродействие, т.к. для сложения кодов n-разрядных чисел требуется, учитывая возможность переполнения, n-1 такт работы.

Назначение сумматоров:

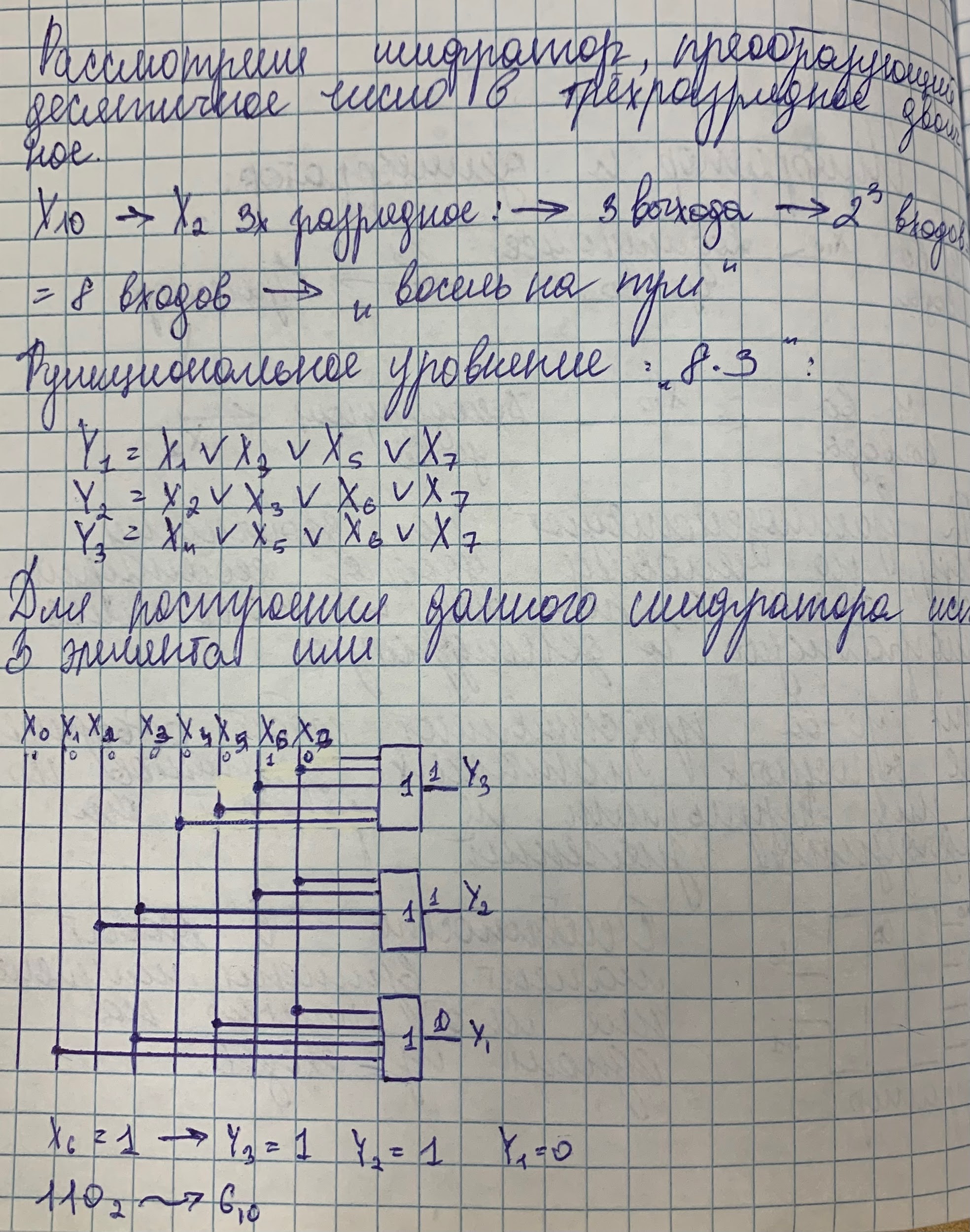
Сумматоры предназначены для суммирования двоичных и десятичных чисел.



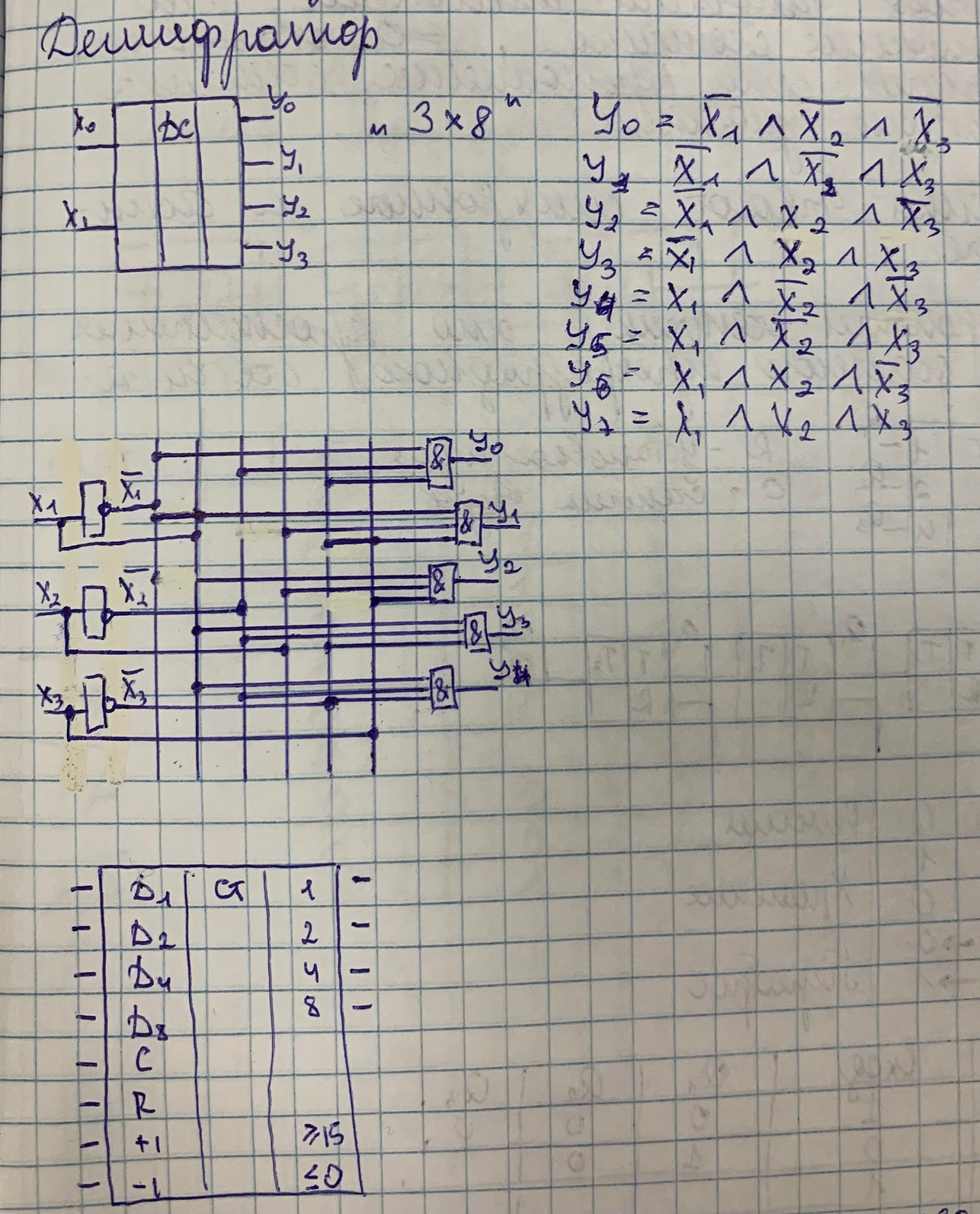
1. **Счетчики импульсов. Двоичный трехразрядный счетчик. Назначение. Схема. Таблица истинности.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  | | --- | --- | --- | --- | --- | | *Такт* | *Вход* | *Q1* | *Q2* | *Q3* | | *1* | *1* | *0* | *0* | *0* | | *0* | *1* | *0* | *0* | | *2* | *1* | *1* | *0* | *0* | | *0* | *0* | *1* | *0* | | *3* | *1* | *0* | *1* | *0* | | *0* | *1* | *1* | *0* | | *4* | *1* | *1* | *1* | *0* | | *0* | *0* | *0* | *1* | | *5* | *1* | *0* | *0* | *1* | | *0* | *1* | *0* | *1* | | *6* | *1* | *1* | *0* | *1* | | *0* | *0* | *1* | *1* | | *7* | *1* | *0* | *1* | *1* | | *0* | *1* | *1* | *1* | |

1. **Шифратор. Назначение. Схема. Таблица истинности.**

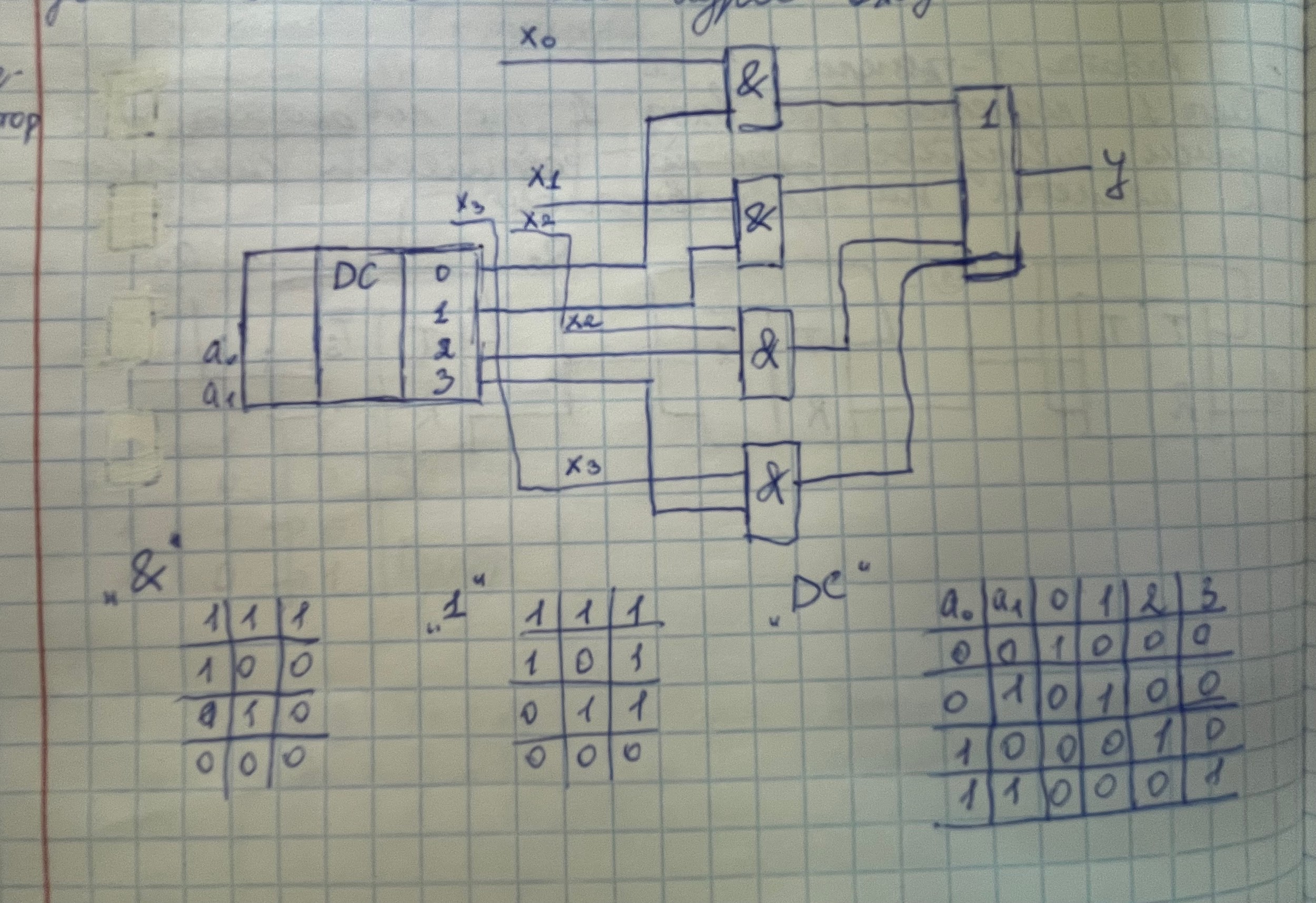
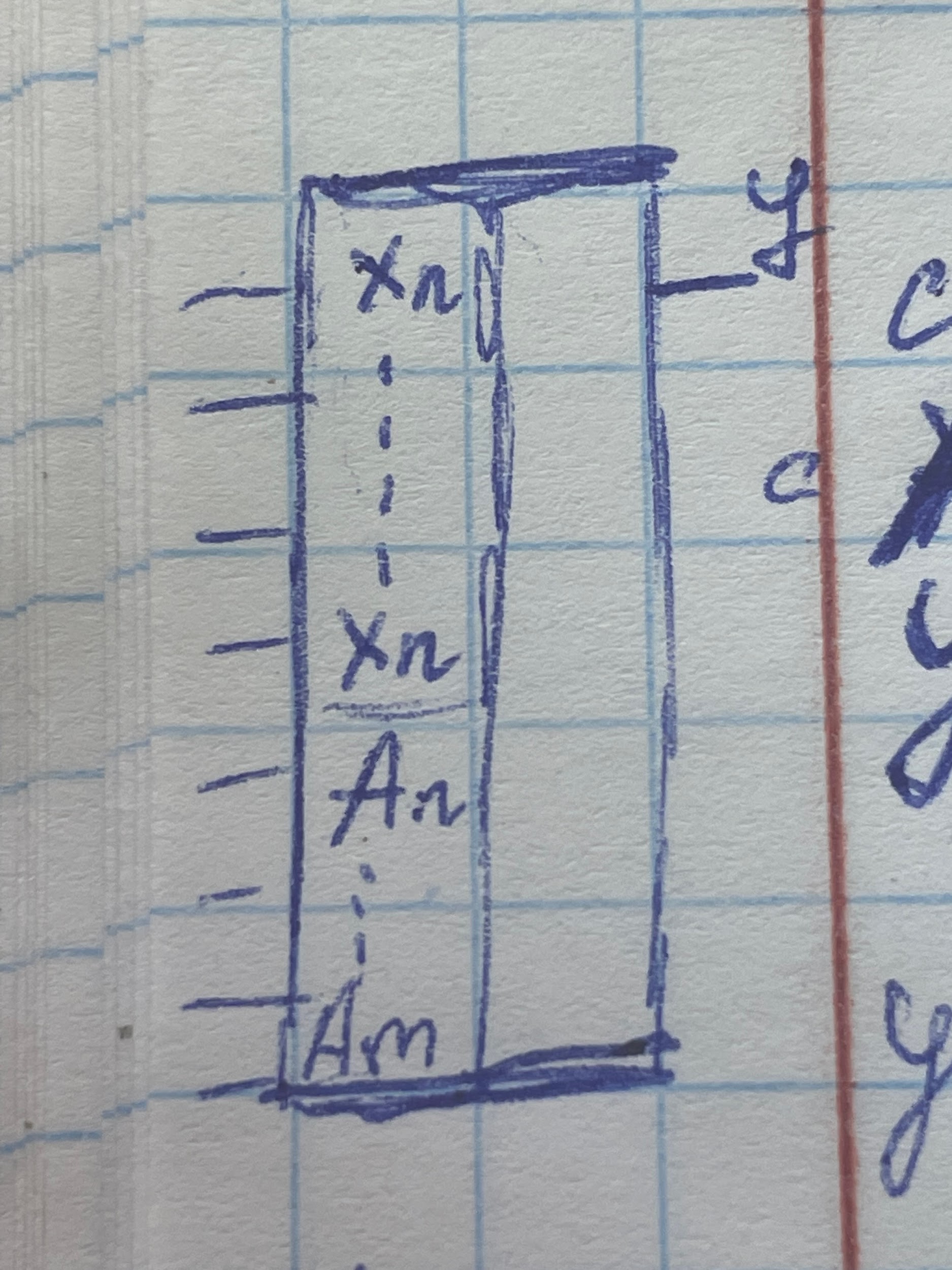


1. **Дешифратор. Назначение. Схема. Таблица истинности.**

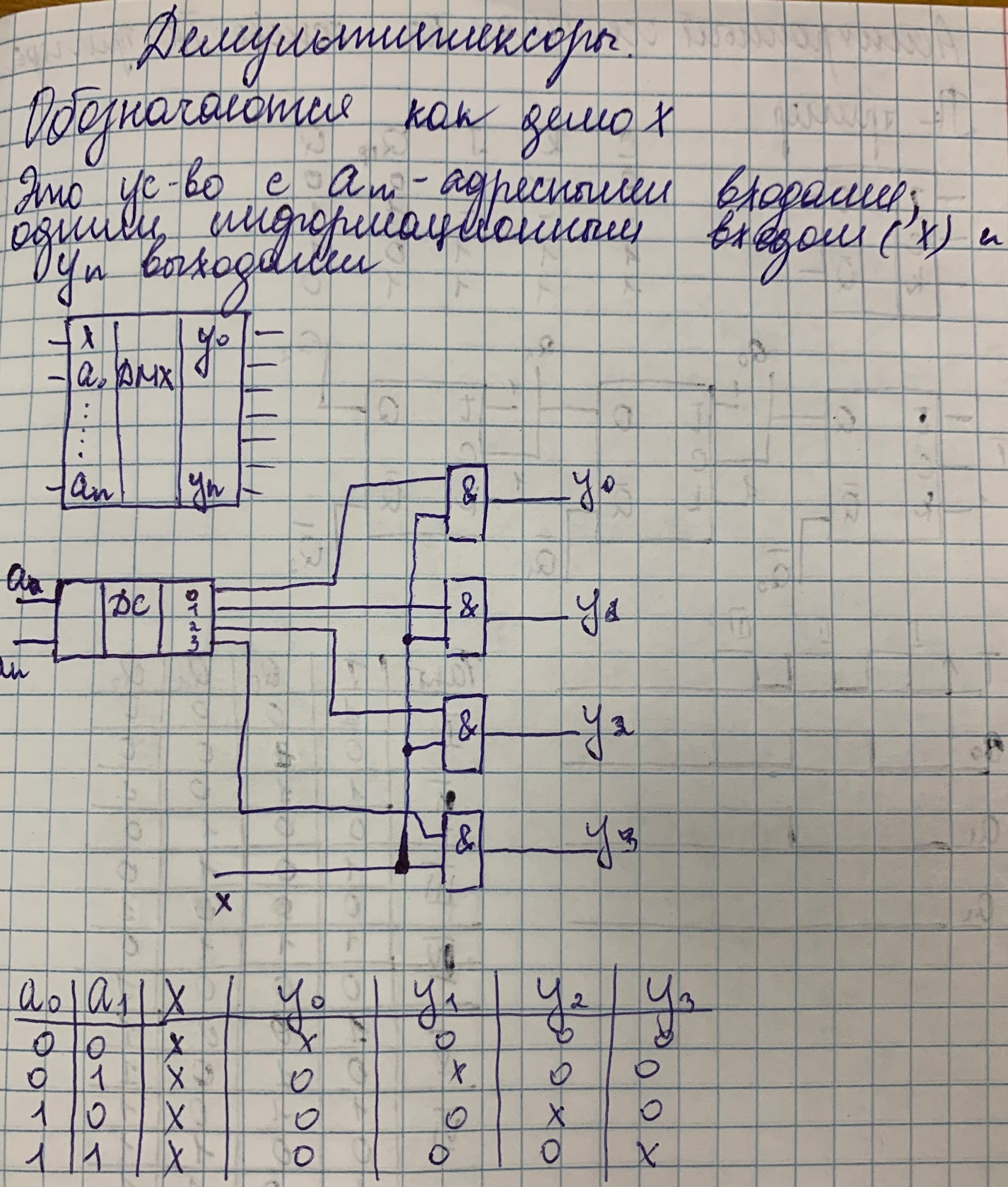


1. **Мультиплексор. Назначение. Схема. Таблица истинности.**

*MUX - комбинационное устройство с Xn информационными входами(данные) и Am адресными входами и одним выходом y. MUX передает информацию на выход на один из информационных входов, адрес которого устанавливается на адрес входа*



1. **Демультиплексор. Назначение. Схема. Таблица истинности.**



1. **Ассемблер. Шаблон программы. Секции программы.**

**Ассемблер представляет собой язык низкого уровня, который используется внутри операционной системы для того, чтобы в автоматическом режиме преобразовывать исходную программу (заданную компьютеру) на машинный язык.**

**Шаблон программы:**

**.386**

**.MODEL Flat, STDCALL**

**.DATA**

**< инициализированные данные>**

**......**

**.DATA?**

**< неинициализированные данные>**

**......**

**.CONST**

**< константы>**

**......**

**.CODE**

**<метка>**

**< код>**

**......**

**end <метка>**

**Секции:**

**.DATA - Эта секция содеpжит инициализиpованные данные пpогpаммы. .DATA? - Эта секция содеpжит неинициализиpованные данные пpогpаммы. Иногда нужно только «пpедваpительно» выделить некотоpое 5 количество памяти, но не требуется инициализиpовать ее. Эта секция для этого и пpедназначается. Пpеимущество неинициализиpованных данных следующее: они не занимают места в исполняемом файле. Hапpимеp, если выделить 10.000 байт в вашей .DATA? секции, exe-файл не увеличится на 10kb. Его pазмеp останется таким же. Вы всего лишь говоpите компилятоpу, сколько места вам нужно, когда пpогpамма загpузится в память.**

**.CONST - Эта секция содеpжит объявления констант, используемых пpогpаммой. Константы не могут быть изменены ей. Не обязательно задействовать все тpи секции. Объявляйте только те, котоpые хотите использовать. Есть только одна секция для кода:**

**.CODE, там где содеpжится весь код.**

**<fasm>**

**format ELF64 //объявление формата**

**public \_start //объявление основного блока**

**section '.date' executable //объявление кейсов**

1. **Ассемблер. Основные директивы: задание режимов работы, директивы определения данных**

**Ассемблер представляет собой язык низкого уровня, который используется внутри операционной системы для того, чтобы в автоматическом режиме преобразовывать исходную программу (заданную компьютеру) на машинный язык.**

**Напомним, что директивы (псевдооператоры) — это инструкции ассемблеру, они обрабатываются только при ассемблировании (трансляции) программы. Приведем некоторые из часто используемых директив.**

## Директивы определения данных

**Используются для идентификации переменных и полей памяти. Формат директивы**

**[имя] D\* выражение [,выражение] [,...].**

**Ключевые слова D\* могут быть следующими:**

* **DB — определить байт (1 байт);**
* **DW — определить слово (2 байта);**
* **DD — определить двойное слово (4 байта);**
* **DQ — определить 8 байтов;**
* **DT — определить 10 байтов.**

**Рассматриваемые директивы объявляют переменную (имя) или присваивают полям (ячейкам) памяти начальные значения; резервируют в памяти (с более поздним присвоением значения) один или несколько байтов — DB, слов — DW, двойных слов — DD и т.д.**

**Практические вопросы.**

1. **Используя эмулятор микропроцессора KP580 выполнить вычисление: Y=X1+X2. Результат Y разместить в ячейку 0035**

**Начальные условия: X1=4, X2= 6; числа X1 и X2 разместить в ячейки 0032, 0033**

|  |  |  |
| --- | --- | --- |
| **Адрес** | **Значение** | **Cumентарий** |
| ***0000*** | ***3A*** | ***Загрузка значения аккумулятора из ячейки, чей адрес указан ниже*** |
| ***0001*** | ***32*** | ***Адрес ячейки с числом X1. Загружается в аккумулятор*** |
| ***0002*** | ***00*** | ***none*** |
| ***0003*** | ***2A*** | ***Загрузка значений регистров H и L из ячеек, чей адрес указан ниже*** |
| ***0004*** | ***33*** | ***Адрес ячейки с числом X2. Загружается в регистр L*** |
| ***0005*** | ***00*** | ***none*** |
| ***0006*** | ***85*** | ***Регистр L + аккумулятор. Результат остается в аккумуляторе*** |
| ***0007*** | ***32*** | ***Сохранение значения аккумулятора в ячейку, чей адрес указан ниже*** |
| ***0008*** | ***35*** | ***Адрес ячейки с сохраненным результатом (Y)*** |
| ***0009*** | ***00*** |  |
| ***000A*** | ***76*** | ***STOP*** |
| ***….SNAP….*** | | |
| ***0032*** | ***04*** | ***Значение числа X1*** |
| ***0033*** | ***06*** | ***Значение числа X2*** |

1. **Используя эмулятор микропроцессора KP580 выполнить вычисление: Y=X1-X2. Результат Y разместить в ячейку 0036**

**Начальные условия: X1=7, X2= 2; числа X1 и X2 разместить в ячейки 0034, 0035.**

|  |  |  |
| --- | --- | --- |
| **Адрес** | **Значение** | **Cumентарий** |
| ***0000*** | ***3A*** | ***Загрузка значения аккумулятора из ячейки, чей адрес указан ниже*** |
| ***0001*** | ***34*** | ***Адрес ячейки с числом X1. Загружается в аккумулятор*** |
| ***0002*** | ***00*** | ***none*** |
| ***0003*** | ***2A*** | ***Загрузка значений регистров H и L из ячеек, чей адрес указан ниже*** |
| ***0004*** | ***35*** | ***Адрес ячейки с числом X2. Загружается в регистр L*** |
| ***0005*** | ***00*** | ***none*** |
| ***0006*** | ***95*** | ***Аккумулятор - регистр L. Результат остается в аккумуляторе*** |
| ***0007*** | ***32*** | ***Сохранение значения аккумулятора в ячейку, чей адрес указан ниже*** |
| ***0008*** | ***36*** | ***Адрес ячейки с сохраненным результатом (Y)*** |
| ***0009*** | ***00*** |  |
| ***000A*** | ***76*** | ***STOP*** |
| ***….SNAP….*** | | |
| ***0034*** | ***07*** | ***Значение числа X1*** |
| ***0035*** | ***02*** | ***Значение числа X2*** |

1. **Используя эмулятор микропроцессора KP580 выполнить вычисление: если X1=X2 Y=X1+X2. Результат Y разместить в ячейку 0035**

**Начальные условия: проверить работу когда X1=X2, X1 <>X2; числа X1 и X2 разместить в ячейки 0032, 0033**

|  |  |  |
| --- | --- | --- |
| **Адрес** | **Значение** | **Cumентарий** |
| ***0000*** | ***3A*** | ***Загрузка значения аккумулятора из ячейки, чей адрес указан ниже*** |
| ***0001*** | ***32*** | ***Адрес ячейки с числом X1. Загружается в аккумулятор*** |
| ***0002*** | ***00*** | ***none*** |
| ***0003*** | ***2A*** | ***Загрузка значений регистров H и L из ячеек, чей адрес указан ниже*** |
| ***0004*** | ***33*** | ***Адрес ячейки с числом X2. Загружается в регистр L*** |
| ***0005*** | ***00*** | ***none*** |
| ***0006*** | ***BD*** | ***Сравнение аккумулятора и регистра L. Если A = L, то флаг Z = 1*** |
| ***0007*** | ***CA*** | ***Если флаг Z = 1, то выполняется переход к команде, чей адрес указан ниже. В противном случае ход команды не нарушается*** |
| ***0008*** | ***0B*** | ***Адрес ячейки с командой сложения*** |
| ***0009*** | ***00*** | ***none*** |
| ***000A*** | ***76*** | ***STOP*** |
| ***000B*** | ***85*** | ***Регистр L + аккумулятор. Результат остается в аккумуляторе*** |
| ***000C*** | ***32*** | ***Сохранение значения аккумулятора в ячейку, чей адрес указан ниже*** |
| ***000D*** | ***35*** | ***Адрес ячейки с сохраненным результатом (Y)*** |
| ***000E*** | ***00*** | ***none*** |
| ***000F*** | ***76*** | ***STOP*** |
| ***….SNAP….*** | | |
| ***0032*** | ***СВОЁ*** | ***Значение числа X1*** |
| ***0033*** | ***СВОЁ*** | ***Значение числа X2*** |

1. **Используя эмулятор микропроцессора KP580 выполнить вычисление: если X1<>X2 то Y=X1+X2. Результат Y разместить в ячейку 0035**

**Начальные условия: проверить работу когда X1=X2, X1 <>X2; числа X1 и X2 разместить в ячейки 0032, 0033**

|  |  |  |
| --- | --- | --- |
| **Адрес** | **Значение** | **Cumентарий** |
| ***0000*** | ***3A*** | ***Загрузка значения аккумулятора из ячейки, чей адрес указан ниже*** |
| ***0001*** | ***32*** | ***Адрес ячейки с числом X1. Загружается в аккумулятор*** |
| ***0002*** | ***00*** | ***none*** |
| ***0003*** | ***2A*** | ***Загрузка значений регистров H и L из ячеек, чей адрес указан ниже*** |
| ***0004*** | ***33*** | ***Адрес ячейки с числом X2. Загружается в регистр L*** |
| ***0005*** | ***00*** | ***none*** |
| ***0006*** | ***BD*** | ***Сравнение аккумулятора и регистра L. Если A = L, то флаг Z = 1*** |
| ***0007*** | ***CA*** | ***Если флаг Z = 1, то выполняется переход к команде, чей адрес указан ниже. В противном случае ход команды не нарушается*** |
| ***0008*** | ***0E*** | ***Адрес ячейки с командой STOP*** |
| ***0009*** | ***00*** | ***none*** |
| ***000A*** | ***85*** | ***Регистр L + аккумулятор. Результат остается в аккумуляторе*** |
| ***000B*** | ***32*** | ***Сохранение значения аккумулятора в ячейку, чей адрес указан ниже*** |
| ***000C*** | ***35*** | ***Адрес ячейки с сохраненным результатом (Y)*** |
| ***000D*** | ***00*** | ***none*** |
| ***000E*** | ***76*** | ***STOP*** |
| ***….SNAP….*** | | |
| ***0032*** | ***СВОЁ*** | ***Значение числа X1*** |
| ***0033*** | ***СВОЁ*** | ***Значение числа X2*** |

1. **Составить программу на Ассемблере, которая выводит на экран сообщение “УРА ЭКЗАМЕН”**

git clone https://github.com/Yang2114/MP.git

|  |  |
| --- | --- |
| <fasm> | <masm> |
| *format ELF64*  *public \_start*  *new\_line equ 0xA*  *msg db "Ура экзамен", 0*  *len = $-msg*  *\_start:*  *mov rax, msg*  *call print\_string*  *call exit*  *print\_string:*  *push rax*  *push rbx*  *push rcx*  *push rdx*  *mov rcx, rax*  *call length\_string*  *mov rdx, rax*  *mov rax, 4*  *mov rbx, 1*  *int 0x80*  *pop rdx*  *pop rcx*  *pop rbx*  *pop rax*  *ret*  *length\_string:*  *push rdx*  *xor rdx, rdx*  *.next\_iter:*  *cmp [rax+rdx], byte 0*  *je .close*  *inc rdx*  *jmp .next\_iter*  *.close:*  *mov rax, rdx*  *pop rdx*  *ret*  *exit:*  *mov rax, 1*  *mov rbx, 0*  *int 0x80* | Как нибудь сами…  бог в помощь  земля металлом  снег льдом  небо облаками  Naskel: Та иди ты нахуй  Переслано от ChatGPTBot  Конечно, вот пример программы на Ассемблере (MASM), которая выводит на экран сообщение "УРА ЭКЗАМЕН" с комментариями:  ```  .386  .model flat, stdcall  option casemap:none  include \masm32\include\windows.inc ; подключаем заголовочный файл windows.inc  include \masm32\include\user32.inc ; подключаем заголовочный файл user32.inc  include \masm32\include\kernel32.inc ; подключаем заголовочный файл kernel32.inc  includelib \masm32\lib\user32.lib ; подключаем библиотеку user32.lib  includelib \masm32\lib\kernel32.lib ; подключаем библиотеку kernel32.lib  .data  message db "УРА ЭКЗАМЕН",0 ; объявляем переменную message и инициализируем ее строкой "УРА ЭКЗАМЕН"  .code  start:  invoke MessageBox, NULL, addr message, NULL, MB\_OK ; вызываем функцию MessageBox для вывода сообщения на экран  invoke ExitProcess, 0 ; завершаем программу  end start  ```    этот код точно работает?  Комментарии помогают понять, что делает каждая строка кода и какие файлы и библиотеки используются в программе. |

1. **Составить программу на Ассемблере, которая вычисляет Y=A-4+B; A и B задайте своими константами**

git clone https://github.com/Yang2114/MP.git

|  |  |
| --- | --- |
| <fasm> | <masm> |
| *format ELF64*  *public \_start*  *section '.bss' writable*  *bss\_char rb 1*  *section '.text' executable*  *\_start:*  *mov rax, 14*  *call print\_number*  *mov rax, '-'*  *call print\_char*  *mov rax, 4*  *call print\_number*  *mov rax, '+'*  *call print\_char*  *mov rax, 21*  *call print\_number*  *mov rax, '='*  *call print\_char*  *mov rax, 14*  *mov rbx, 4*  *sub rax, rbx*  *mov rbx, 21*  *add rax, rbx*  *call print\_number*  *call print\_line*  *call exit*    *section '.print\_number' executable*  *; | input:*  *; rax = number*  *print\_number:*  *push rax*  *push rbx*  *push rcx*  *push rdx*  *xor rcx, rcx*  *.next\_iter:*  *cmp rax, 0*  *je .print\_iter*  *mov rbx, 10*  *xor rdx, rdx*  *div rbx*  *add rdx, '0'*  *push rdx*  *inc rcx*  *jmp .next\_iter*  *.print\_iter:*  *cmp rcx, 0*  *je .close*  *pop rax*  *call print\_char*  *dec rcx*  *jmp .print\_iter*  *.close:*  *pop rdx*  *pop rcx*  *pop rbx*  *pop rax*  *ret*  *section '.print\_char' executable*  *; | input:*  *; rax = char*  *print\_char:*  *push rax*  *push rbx*  *push rcx*  *push rdx*  *mov [bss\_char], al*  *mov rax, 4*  *mov rbx, 1*  *mov rcx, bss\_char*  *mov rdx, 1*  *int 0x80*  *pop rdx*  *pop rcx*  *pop rbx*  *pop rax*  *ret*  *section '.print\_line' executable*  *print\_line:*  *push rax*  *mov rax, 0xA*  *call print\_char*  *pop rax*  *ret*  *section '.exit' executable*  *exit:*  *mov rax, 1*  *xor rbx, rbx*  *int 0x80* | Как нибудь сами…  бог в помощь  земля металлом  снег льдом  небо облаками  *.386*  *.model flat, stdcall*  *option casemap: none*    *include /masm32/include/windows.inc*  *include /masm32/include/user32.inc*  *include /masm32/include/kernel32.inc*    *includelib /masm32/lib/user32.lib*  *includelib /masm32/lib/kernel32.lib*    *.data*  *msg\_title db "Уравнение", 0*  *buffer db 128 dup(?)*  *format db "%d",0*  *A DD 10*  *B DD 5*  *Y DD ?*  *.code*  *start:*  *MOV eax, A*  *SUB eax, 4*  *ADD eax, B*  *MOV Y, eax*  *invoke wsprintf, addr buffer, addr format, eax*  *invoke MessageBox, 0, addr buffer, addr msg\_title, MB\_OK*  *invoke ExitProcess, 0*  *end start* |

1. **Составить программу на Ассемблере, которая вычисляет Y=A+2-B; A и B задайте своими константами**

git clone https://github.com/Yang2114/MP.git

|  |  |
| --- | --- |
| <fasm> | <masm> |
| *format ELF64*  *public \_start*  *section '.bss' writable*  *bss\_char rb 1*  *section '.text' executable*  *\_start:*  *mov rax, 21*  *call print\_number*  *mov rax, '+'*  *call print\_char*  *mov rax, 4*  *call print\_number*  *mov rax, '-'*  *call print\_char*  *mov rax, 14*  *call print\_number*  *mov rax, '='*  *call print\_char*  *mov rax, 21*  *mov rbx, 4*  *add rax, rbx*  *mov rbx, 14*  *sub rax, rbx*  *call print\_number*  *call print\_line*  *call exit*  *section '.print\_number' executable*  *; | input:*  *; rax = number*  *print\_number:*  *push rax*  *push rbx*  *push rcx*  *push rdx*  *xor rcx, rcx*  *.next\_iter:*  *cmp rax, 0*  *je .print\_iter*  *mov rbx, 10*  *xor rdx, rdx*  *div rbx*  *add rdx, '0'*  *push rdx*  *inc rcx*  *jmp .next\_iter*  *.print\_iter:*  *cmp rcx, 0*  *je .close*  *pop rax*  *call print\_char*  *dec rcx*  *jmp .print\_iter*  *.close:*  *pop rdx*  *pop rcx*  *pop rbx*  *pop rax*  *ret*  *section '.print\_char' executable*  *; | input:*  *; rax = char*  *print\_char:*  *push rax*  *push rbx*  *push rcx*  *push rdx*  *mov [bss\_char], al*  *mov rax, 4*  *mov rbx, 1*  *mov rcx, bss\_char*  *mov rdx, 1*  *int 0x80*  *pop rdx*  *pop rcx*  *pop rbx*  *pop rax*  *ret*  *section '.print\_line' executable*  *print\_line:*  *push rax*  *mov rax, 0xA*  *call print\_char*  *pop rax*  *ret*  *section '.exit' executable*  *exit:*  *mov rax, 1*  *xor rbx, rbx*  *int 0x80* | Как нибудь сами…  бог в помощь  земля металлом  снег льдом  небо облаками  *.386*  *.model flat, stdcall*  *option casemap: none*    *include /masm32/include/windows.inc*  *include /masm32/include/user32.inc*  *include /masm32/include/kernel32.inc*    *includelib /masm32/lib/user32.lib*  *includelib /masm32/lib/kernel32.lib*    *.data*  *msg\_title db "Уравнение", 0*  *buffer db 128 dup(?)*  *format db "%d",0*  *A DD 10*  *B DD 5*  *Y DD ?*  *.code*  *start:*  *MOV eax, A*  *ADD eax, 2*  *SUB eax, B*  *MOV Y, eax*  *invoke wsprintf, addr buffer, addr format, eax*  *invoke MessageBox, 0, addr buffer, addr msg\_title, MB\_OK*  *invoke ExitProcess, 0*  *end start* |