****

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ**

**УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ**

(**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**)» (**МАИ)**

Кафедра «Технология испытаний и эксплуатации»

**Курсовая работа**

Дисциплина «Архитектура процессоров и ЭВМ»

**Тема: Разработка эмулятора вычислительной машины Intel 8086 с ограниченным набором команд**

Листов 48

Отметка о защите Задание выполнил  
\_\_\_\_\_\_\_ / Шерышев А. Е. студент группы Т12О–309Б–19

«\_\_» Мая 2022 г. \_\_\_\_\_\_\_\_ / Атрашков Н. А.

«30» Мая 2022 г.

2022

Оглавление

[Задание 3](#_Toc104558776)

[Введение 4](#_Toc104558777)

[Раздел I. Описание процессора Intel 8086 5](#_Toc104558778)

[1.1. Архитектура IBM PC (8086) 5](#_Toc104558779)

[1.2. Архитектура процессора Intel 8086 7](#_Toc104558780)

[1.3. Команды процессора Intel 8086 14](#_Toc104558781)

[Раздел II. Среды разработки и требования к аппаратно-программному комплексу 20](#_Toc104558782)

[2.1. Требования к аппаратно-программному комплексу 20](#_Toc104558783)

[2.2. Требования к среде разработки 20](#_Toc104558784)

[2.3. Выбор аппаратно-программного комплекса 20](#_Toc104558785)

[2.4. Выбор средств разработки 20](#_Toc104558786)

[2.5. Анализ команд 21](#_Toc104558787)

[Раздел III. Модель процессора и описание алгоритма эмулятора 22](#_Toc104558788)

[3.1. Описание модели работы процессора 22](#_Toc104558789)

[3.2. Алгоритм эмулятора 24](#_Toc104558790)

[3.3. Описание интерфейса эмулятора 31](#_Toc104558791)

[Раздел IV. Разработка программы эмулятора и его верификация 32](#_Toc104558792)

[4.1. Разработка программы 32](#_Toc104558793)

[4.2. Тестовые команды 33](#_Toc104558794)

[4.3. Верификация 34](#_Toc104558795)

[Заключение 35](#_Toc104558796)

[Литература 36](#_Toc104558797)

[Другие источники 37](#_Toc104558798)

[Приложение А. Ограниченный набор команд для эмуляции 38](#_Toc104558799)

[Приложение Б. Текст программы 39](#_Toc104558800)

# **Задание**

Разработать эмулятор Intel 8086 на языке высокого уровня. Набор команд приведен в приложении А. Дата получения задания: 25.02.2022. Дата сдачи задания: 30.05.2022.

# **Введение**

Микропроцессор 8086 фирмы INTEL стал основой персональных ЭВМ фирмы IBM и большого числа совместимых компьютеров других фирм. Изучение микропроцессора 8086 требуется для понимания функционирования микропроцессоров фирмы INTEL и им подобных.

В данной курсовой работе на основе изученного материала нужно разработать эмулятор Intel 8086 с ограниченном набором команд на языке высокого уровня.

# **Раздел I. Описание процессора Intel 8086**

## **1.1. Архитектура IBM PC (8086)**

Спрос персональных компьютеров к концу 1970-х годов привело к снижению спроса на большие ЭВМ и мини ЭВМ. В 1979 году фирма IBM решила попробовать свои силы на рынке персональных компьютеров.

Архитектура ПК от IBM несет в себе 3 главных принципа:

* конструкция устройства должна предусматривать возможность расширения возможностей системы;
* изменения внутри системы не должны требовать лицензионных соглашений или затрат;
* пользователь самостоятельно может изменять базовые возможности компьютерной системы.

Открытая архитектура компьютера подразумевает наличие системы, которую пользователь может самостоятельно изменить или настроить. Он может самостоятельно заменить компоненты системы на более мощные и современные, в отличие от закрытых систем, где такое действие нельзя выполнить.

Персональные компьютеры IBM PC состоят из трех блоков:

* системного блока;
* клавиатуры, позволяющей вводить символы в компьютер;
* монитора (или дисплея) — для изображения текстовой и графической информации.

В системном блоке располагаются все основные узлы компьютера:

* электронные схемы, управляющие работой компьютера (микропроцессор, оперативная память, контроллеры устройства и т. д.);
* блок питания, преобразующий электропитание сети в постоянный ток низкого напряжения, подаваемый на электронные схемы компьютера;
* накопители (или дисководы) для гибких магнитных дисков, используемые для чтения и записи на гибкие магнитные диски (дискеты);
* накопитель на жестком магнитном диске, предназначенный для чтения и записи на несъемный жесткий магнитный диск (винчестер).

Основы архитектуры IBM для ПК:

* присутствует центральный процессор Intel и/или совместимые с ним процессоры других производителей;
* присутствует BIOS;
* регламентируется процедура стартового запуска системы;
* есть механизм собственного конфигурирования системы;
* присутствует реестр системы, где хранятся сведения о конфигурации устройства;
* блочная организация памяти в устройстве, к которой организован прямой доступ;
* наличие нормативов, которые описывают конструкцию компьютера, режимы работы, протоколы по обмену данными и др.

Открытая архитектура для ПК регламентирует не только аппаратную часть устройства, то есть тот факт, что можно заменять внутренние модули, но и программную часть.

Главная идея, которую несет в себе архитектура для ПК от IBM — участвовать в производстве одного компьютера может множество разных компаний, которые производят компоненты для компьютеров.

## **1.2. Архитектура процессора Intel 8086**

Первым представителем семейства Intel x86, или, согласно официальной классификации фирмы Intel (Integrated Electronics, США), семейства процессоров IA (Intel Architecture), является микропроцессор 8086, разработанный к 1978 году. Программы, написанные для него, выполняются на всех последующих процессорах семейства, включая все современные модели.

Микропроцессор 8086 ориентирован на выполнение команд параллельно с их выборкой и может быть условно разделен на две части, работающие асинхронно:

* устройство сопряжения с внешними шинами (УС)

Устройство сопряжения обеспечивает формирование 20-разрядного физического адреса памяти, выборку команд и операндов из памяти, организацию очередности команд и запоминание результатов выполнения команд в памяти. В состав УС входит шесть 8-разрядных регистров очереди команд, четыре 16-разрядных сегментных регистра, 16- разрядный регистр обмена и 16-разрядный сумматор адреса, интерфейс с внешними шинами.

Регистры очереди команд организованы по принципу FIFO - «первым пришел - первым вышел». УС готово выполнить цикл выборки 16-разрядного слова из памяти всякий раз, когда в очереди освобождаются, по меньшей мере, два байта, а УО извлекает из очереди команды по мере их выполнения. При выполнении команд передачи управления, например условных и безусловных переходов, очередь очищается УС и начинает заполняться заново.

* устройство обработки (УО).

Устройство обработки предназначено для выполнения операций по обработке данных и состоит из устройства микропрограммного управления (УМУ), 16-разрядного АЛУ, восьми 16-разрядных регистров общего назначения и регистра признаков. Команды из очереди, сформированной УС, поступают в УМУ, где декодируются и выполняются в 16-разрядном АЛУ согласно процедурам, записанным в памяти микропрограмм. Последовательное выполнение команд обеспечивается секвенсором команд, часть которого (регистр счетчика команд IP) изображена в составе УС, т.к. именно УС записывает в IP смещение следующей команды, т.е. положение новой команды относительно начала сегмента команд. УО обменивается данными с УС через внутреннюю 16-разрядную шину и регистр обмена.

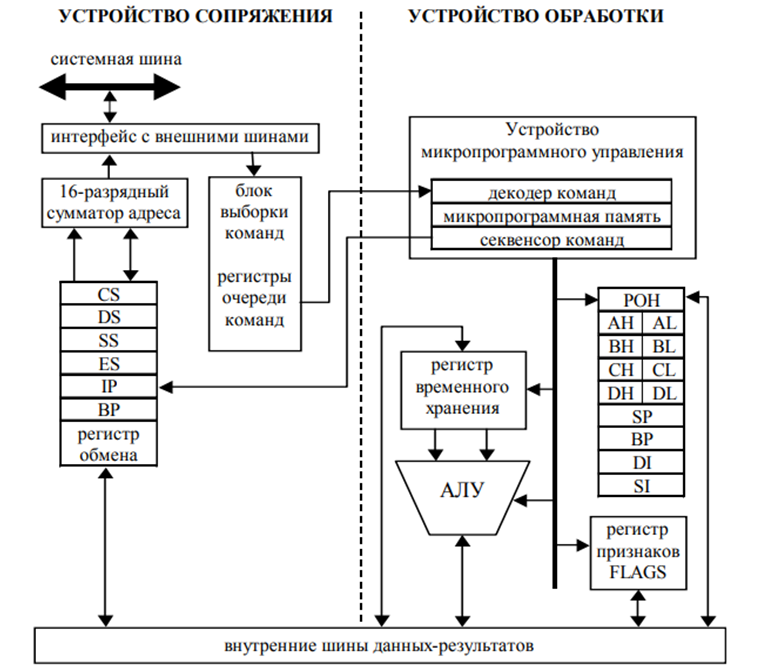


Рисунок 1.2.1 — Микроархитектура процессора 8086

Основные характеристики микропроцессора 8086

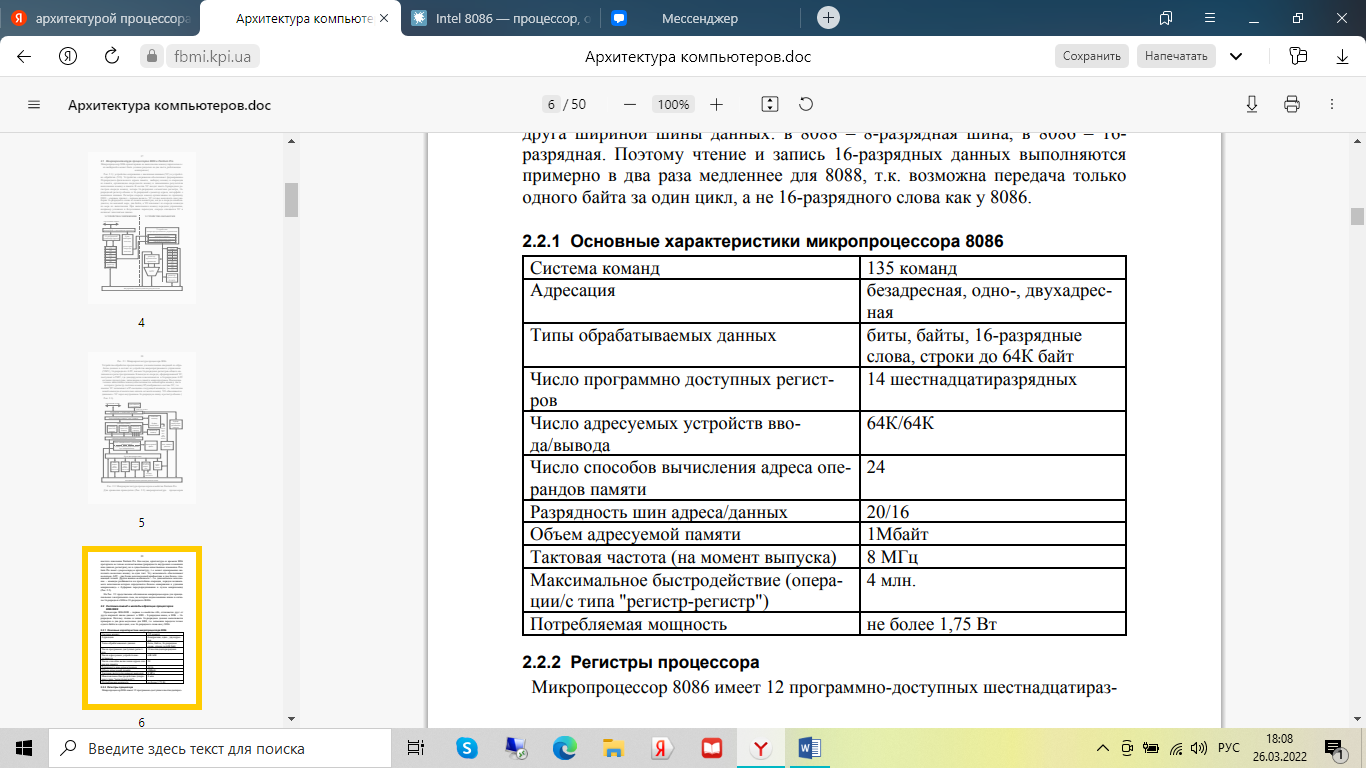


Рисунок 1.2.2 — Основные характеристики микропроцессора 8086

Процессор интересует нас, прежде всего, как набор регистров. Регистр — быстродействующее запоминающее устройство, реализованное на электронных компонентах.

Все регистры имеют размер слова (16 разрядов), за каждым из них закреплено определенное имя. По назначению и способу использования регистры можно разбить на следующие группы:

* регистры общего назначения (AX, BX, CX, DX, SI, DI, BP, SP);
* сегментные регистры (CS, DS, SS, ES);
* указатель команд (IP);
* регистр флагов (Flags).

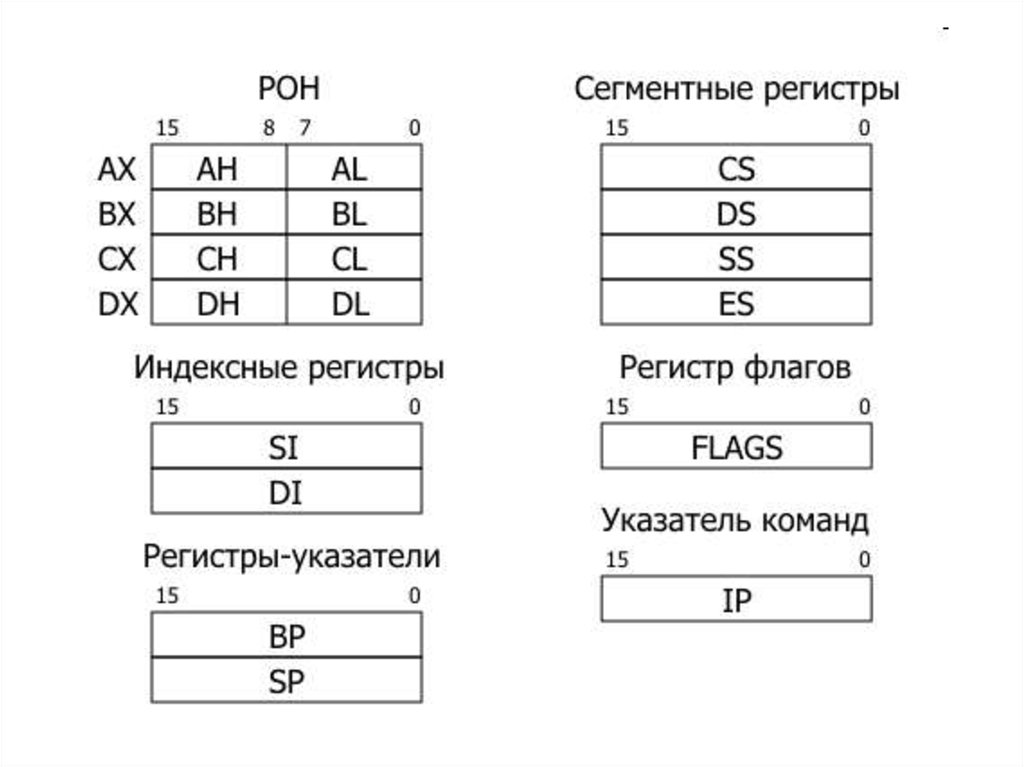


Рисунок 1.2.3 — Регистры микропроцессора 8086

(X - от слова eXtended, расширенный: в процессоре 8080 были байтовые регистры A, B, C и D, но затем их расширили до размера слова)

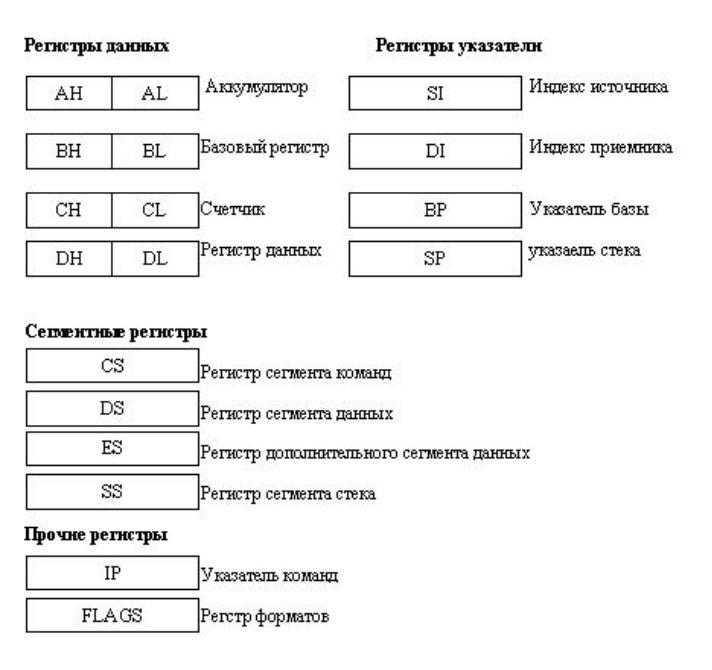


Рисунок 1.2.4 — Расшифровка Регистров.

Регистры общего назначения

Особенностью регистров общего назначения является то, что их можно использовать в любых арифметических, логических и т.п. машинных операциях. В то же время каждый из этих регистров имеет определенную специализацию: некоторые команды требуют, чтобы их операнд или операнды обязательно находились в определенных регистрах. Например, команда деления требует, чтобы первый операнд (делимое) находился в регистре AX или регистрах AX и DX (в зависимости от размера операнда), а команды управления циклом используют регистр CX в качестве счетчика цикла.

Регистры AX, BX, CX и DX устроены так, что возможен независимый доступ к их старшей и младшей половинам. Обозначают эти половины буквами H (high - выше, старший) и L (low - ниже, младший) и первой буквой из названия регистра: AH и AL - в AX, BH и BL - в BX и т.д.

Отметим, что на части делятся только регистры AX, BX, CX и DX. Запись же в другие регистры и чтение из них осуществляются только целиком.

Сегментные регистры

Ни в каких арифметических, логических и т.п. операциях эти регистры не могут участвовать. Эти регистры используются для сегментирования адресов, которое является разновидностью модификации адресов и которое используется для сокращения размера команд.

Так как существует всего четыре сегментных регистра, то одновременно можно работать с четырьмя сегментами памяти. Если же надо работать с большим числом сегментов, то сохраняем содержимое сегментных регистров в обычных регистрах или в памяти, а затем записываем в них адреса нужных сегментов. Позже, если надо, можно восстановить прежние значения сегментных регистров.

Указатель команд

В регистре IP всегда находится адрес команды, которая должна быть выполнена следующей. Более точно, в IP находится адрес этой команды, отсчитанный от начала сегмента команд, на начало которого указывает регистр CS. Поэтому абсолютный адрес этой команды определяется парой регистров CS и IP. Изменение любого из этих регистров есть ничто иное, как переход. Следовательно, содержимое регистра IP (как и CS) можно менять только командами перехода.

Регистр флагов

Флаг — это бит, принимающий значение 1 ("флаг установлен") или значение 0 ("флаг сброшен"). В i8086 используется 9 флагов, собранных в один 16-разрядный регистр, называемый регистром флагов (Flags). Некоторые разряды регистра не заняты.

Флаги делятся на две группы:

* флаги условий (они автоматически меняются при выполнении команд и фиксируют те или иные свойства их результата (например, равен ли он нулю)):

1. CF (carry flag) — флаг переноса. Наиболее полезен в арифметических операциях над числами без знака; например, если при сложении беззнаковых чисел получилась слишком большая сумма - с единицей переноса, которой нет места в ячейке, тогда флаг CF принимает значение 1, а если сумма "укладывается" в размер ячейки, то значением CF будет 0.
2. OF (overflow flag) — флаг переполнения. Полезен в арифметических операциях над числами со знаком; например, если при сложении или вычитании знаковых чисел получился результат, по модулю превосходящий допустимую величину (произошло переполнение мантиссы), тогда флаг OF получает значение 1, а если переполнения мантиссы не было - значение 0.
3. ZF (zero flag) — флаг нуля. Устанавливается в 1, если результат команды оказался нулевым.
4. SF (sign flag) — флаг знака. Устанавливается в 1, если в операции над знаковыми числами получился отрицательный результат.
5. PF (parity flag) — флаг четности. Равен 1 , если в 8 младших битах результата очередной команды содержится четное количество двоичных единиц.
6. AF (auxiliary carry flag) — флаг дополнительного переноса. Фиксирует особенности выполнения операций над двоично-десятичными числами.

* флаги состояний (их меняет программа; состояние этих флагов оказывает влияние на дальнейшее поведение процессора):

1. DF (direction flag) — флаг направления. Устанавливает направление просмотра строк в строковых командах: при DF=0 строки просматриваются "вперед" (от начала к концу), при DF=1 - в обратном направлении.
2. IF (interrupt flag) — флаг прерываний. При IF=0 процессор перестает реагировать на поступающие к нему прерывания, а при IF=1 блокировка прерываний снимается.
3. TF (trace flag) — флаг трассировки. При TF=1 после выполнения каждой команды процессор делает прерывание, чем можно воспользоваться при отладке программы.

## **1.3. Команды процессора Intel 8086**

Систему команд процессора 8086 образуют 113 базовых команд, многие из которых допускают использование разнообразных режимов адресации.

По функциональному назначению выделяют следующие группы команд:

- команды передачи данных;

- команды арифметических операций;

- команды логических операций и сдвигов;

- команды передачи управления;

- цепочечные команды;

- команды управления микропроцессором.

**Команды передачи данных**

Команды передачи данных разделяют на 4 подгруппы:

* общие команды передачи данных:

В эту подгруппу входят команды, осуществляющие передачу регистр-регистр, регистр-память и память-регистр. Наиболее мощной среди них является команда MOV.

* стековые команды:

Каждая команда занесения данных в стек PUSH имеет соответствующую ей команду извлечения из стека POP. Для адресации вершины стека используется стековый указатель SP. Все стековые команды манипулируют только словами и сопровождаются автоматической модификацией указателя стека: при включении в стек производится декремент, а при извлечении из стека - инкремент SP.

* команды ввода-вывода:

Команда ввода IN и команда вывода OUT допускают работу как с байтами, так и со словами. Команда IN загружает данные из заданного порта в аккумулятор, а команда OUT выполняет передачу из аккумулятора в порт. Для портов ввода-вывода в диапазоне 00-FF можно использовать прямую укороченную адресацию, а остальные порты в диапазоне 100-FFFF можно адресовать только косвенно через регистр DX.

* команды передачи цепочек байт или слов:

Команды передачи данных не модифицируют состояния флажков. Исключение составляют команды POPF и SAHF, прямо воздействующие на регистр флажков.

**Команды арифметических операций**

Процессор 8086 имеет широкий набор команд, реализующих арифметические операции, что позволяет применять его в сложных системах обработки данных.

Арифметические операции выполняются над целыми числами четырех типов: беззнаковыми двоичными, знаковыми двоичными, упакованными десятичными и неупакованными десятичными.  Длина чисел может быть 8 или 16 бит.

Диапазоны чисел: беззнаковое 8-битное: 0 - 255, беззнаковое 16-битное: 0 - 65535, знаковое 8-битное: -128 - +127, знаковое 16-битное: -32768 - +32767.

Команды арифметических операций влияют на состояние флажков OF, SF, ZF, AF, PF и CF.

Одни и те же команды сложения и вычитания могут использоваться для операций как над беззнаковыми, так и знаковыми числами. Контроль над типами чисел должен выполнять сам программист.

* команды сложения;
* команды вычитания;
* команда сравнения;
* команды деления;
* команды преобразования;
* команды десятичной арифметики.

**Команды логических операций и команды сдвигов**

* команды логических операций:

Логические операции представлены командами NOT (инверсия), AND (конъюнкция), OR (дизъюнкция), XOR (исключающее ИЛИ) и командой TEST, которая выполняет конъюнкцию операндов, но не изменяет их значений. Все логические операции являются поразрядными, т.е. выполняются независимо для всех бит операндов.

Бинарные команды AND, OR, XOR и TEST воздействуют на флажки OF, SF, ZF, PF и CF. Унарная операция NOT не влияет на состояние флажков.

* команды сдвигов:

Команды сдвигов подразделяются на команды простых сдвигов и команды циклических сдвигов. Циклические сдвиги влияют только на флаги OF и CF, а обычные изменяют пять флажков: OF, SF, ZF, PF и

CF. Команды сдвигов могут работать как с байтами, так и со словами.

Команды ROL и ROR реализуют простой циклический сдвиг влево и вправо соответственно, помещая значение из выдвигаемого бита в освобождающийся бит.

Команды RCL и RCR называются командами циклического сдвига влево и вправо через перенос, так как флажок CF расширяет сдвигаемый операнд на один бит. Таким образом, значение из CF загружается в освобождающийся бит, а выдвигаемый бит помещается в CF.

Команды SHL и SHR реализуют логический сдвиг влево и вправо соответственно. Для логического сдвига характерно, что в освобождающийся бит загружается нуль, а выдвигаемый бит теряется.

Команды SAL и SAR предназначены для арифметического сдвига влево и вправо. Арифметический сдвиг вправо отличается от логического сдвига тем, что знаковый бит не сдвигается, а дублируется в соседнем правом бите, сохраняя тем самым знак числа. Арифметический сдвиг влево эквивалентен логическому, поэтому мнемоники SAL и SHL обозначают одну и ту же машинную команду.

Команды арифметического сдвига по существу реализуют умножение и деление чисел без знака на степень числа 2.

Поле операнда команд сдвига имеет вид mem/reg,count. Опреанд count определяет число сдвигов и может быть указан как константа 1 или как регистр CL. В первом случае выполняется сдвиг на один бит, а во втором число сдвигов определяется содержимым регистра CL, которое воспринимается как беззнаковое число.

**Команды передачи управления**

Сегментная организация программной памяти определяет две основные разновидности команд передачи управления. Передача управления в пределах текущего сегмента кода называется внутрисегментной - при этом модифицируется только регистр IP и адрес перехода может быть представлен одним словом. Такая передача управления называется ближней (тип NEAR), а ее вариант с сокращённым диапазоном адресов переходов - короткой. Передача управления за пределы текущего сегмента кода называется межсегментной или дальней (тип FAR) - при этом необходимо модифицировать содержимое регистров IP и CS и адрес перехода представляется двумя словами (сегмент: смещение).

Команды передачи управления процессора 8086 подразделяются на команды безусловных переходов, условных переходов, вызовов, возвратов, управления циклами и команды прерываний.

Команды передачи управления не изменяет состояние регистра флагов (за исключением команды IRET)

* команды безусловных переходов:

Команды безусловного перехода имеют общую мнемонику JMP. Команда короткого безусловного перехода содержит во втором байте смещение, которое интерпретируется как знаковое целое. Диапазон значений байта смещения составляет -128 - +127. Если смещение положительное, осуществляется переход вперед, а если отрицательное - переход назад.

Команда ближнего безусловного перехода может либо непосредственно содержать 16-битное смещение, либо косвенный адрес 16-битного смещения. Диапазон смещения составляет -32768 - +32767 байт относительно адреса команды, находящейся после команды JMP.

* команды условных переходов:

В системе команд процессора 8086 имеется 19 двухбайтных команд условных переходов. При выполнении этих команд анализируется некоторое условие, закодированное текущими состояниями флагов, и если оно выполняется, то осуществляется переход, а если нет, то выполняется следующая по порядку команда.

Все условные переходы являются короткими. Некоторые команды для удобства программирования могут иметь несколько различных мнемонических обозначения.

* команды вызова подпрограмм:

Команда вызова подпрограммы CALL передает управление с автоматическим сохранением адреса возврата в стеке. В поле операнда этой команды находится метка первой команды вызываемой подпрограммы.

При переходе к подпрограмме необходимо временно запомнить адрес команды, находящейся после команды CALL. Этот адрес называется адресом возврата. После того, как подпрограмма закончит свои действия, завершающая ее команда возврата RET передает управление по запомненному адресу возврата. Адрес возврата запоминается в стеке.

* команды возврата из подпрограммы:

Каждая подпрограмма должна содержать минимум одну команду возврата RET, которая возвращает управление вызывающей программе. Такая передача управления осуществляется путем извлечения из стека адреса возврата, включенного в него командой вызова подпрограммы.

Команды внутрисегментного и межсегментного возврата имеют одну и ту же мнемонику RET. Тип команды, соответствующей этой мнемонике, определяется ассемблером автоматически.

* команды управления циклами:

Три команды управления циклами применяются для организации программных циклов. В них предусматривается использование регистра CX в качестве счетчика цикла.

В поле операнда команд управления циклами находится метка первой команды цикла (8-битовое смещение). Диапазон переходов этих команд составляет -128 - +127 байт от следующей команды.

Команда LOOP производит декремент регистра CX и, если содержимое CX не равно нулю, происходит переход к началу цикла. В противном случае выполняется следующая по порядку команда.

* команды прерываний:

В процессоре 8086 имеются 3 команды, относящиеся к прерываниям.

Команда программного прерывания INT вызывает программу обработки, определяемую типом прерывания.

**Цепочные команды**

Под цепочкой понимается последовательность любых контекстно связанных байт или слов, находящихся в смежных ячейках памяти.

В системе команд процессора 8086 имеется 5 команд, предназначенных для обработки одного элемента цепочки. Цепочечной команде может предшествовать специальный однобайтный префикс повторения REP, который вызывает повторение действия команды над следующими элементами цепочки. Повторение рассчитано на максимальную длину цепочек 64 Кбайт и выполняется значительно быстрее цикла LOOP.

Цепочечные команды могут иметь операнд-источник, операндполучатель или и то и другое одновременно. Подразумевается, что цепочка-источник по умолчанию находится в текущем сегменте данных, но допускается префикс замены сегмента. Цепочка-получатель может находиться только в текущем дополнительном сегменте. При этом содержимое регистра SI всегда считается смещением текущего элемента цепочки-источника, а содержимое регистра DI - смещением текущего элемента цепочки-получателя.

Необходимые в команде индексные и сегментные регистры должны быть инициализированы до ее выполнения.

При выполнении цепочечной команды содержимое регистров SI и DI автоматически изменяется, чтобы адресовать следующие элементы цепочек. Флаг направления DF определяет автоинкремент (DF=0) или автодекремент индексных регистров.

Если команде предшествует префикс повторения, то после каждого ее выполнения производится декремент регистра-счетчика CX, поэтому его необходимо предварительно инициализировать на требуемое число повторений.  Когда содержимое CX достигает нуля, управление передается следующей команде.

**Команды управления микропроцессором**

Команды данной группы обеспечивают программное управление различными функциями процессора. Они делятся на две подгруппы: команды установки флагов и команды синхронизации.

# **Раздел II. Среды разработки и требования к аппаратно-программному комплексу**

## **2.1. Требования к аппаратно-программному комплексу**

Для функционирования программы необходимо наличие следующих технических средств:

* процессор Intel® Core™ i5 inside™;
* не менее 512 Мб оперативной памяти;
* монитор SVGA с минимальным разрешением 800х600 пикселей;
* манипулятор типа «мышь» и клавиатура.

## **2.2. Требования к среде разработки**

Для разработки программы необходимо:

* Выбрать язык высокого уровня, который подходит для реализации эмулятора.
* операционная система MS Windows 7 или выше.

## **2.3. Выбор аппаратно-программного комплекса**

Для функционирования программы были выбраны следующие технические средства:

* процессор Intel(R) Core(TM) i7-9700K CPU 3.60 GHz;
* 16 ГБ оперативной памяти;
* монитор Samsung C24F390 с разрешением 1920х1080 пикселей;
* манипулятор типа «мышь» и клавиатура.

## **2.4. Выбор средств разработки**

Для разработки программы был выбран следующий инструментарий:

* CLion — платформа выбрана для использования языка С++.
* операционная система Windows 10 Pro.

## **2.5. Анализ команд**

Ниже представлен анализ команд эмулятора

Таблица 1 – Анализ команд

|  |  |  |
| --- | --- | --- |
| Команды | Операнды | Описание |
| PUSH | REG  SREG  memory immediate | Записывает 16-битовое значение в стек. |
| POP | REG  SREG  memory | Получает 16-битовое значение из стека. |
| MOV | REG, memory  memory, REG  REG, REG  memory, immediate  REG, immediate  SREG, memory  memory, SREG  REG, SREG  SREG, REG | Копирует operand2 в operand1. |
| ADD | REG, memory  memory, REG  REG, REG  memory, immediate  REG, immediate | Сложение. |
| SUB | REG, memory  memory, REG  REG, REG  memory, immediate  REG, immediate | Вычитание. |
| INC | REG  memory | Инкремент. |
| AND | REG, memory  memory, REG  REG, REG  memory, immediate  REG, immediate | Логическое И между всеми битами двух операндов. Результат записывается в 1-й операнд. |
| OR | REG, memory  memory, REG  REG, REG  memory, immediate  REG, immediate | Логическое ИЛИ между всеми битами двух операндов. Результат записывается в 1-й операнд. |
| NOT | REG  memory | Инвертируем каждый бит операнда. |

# **Раздел III. Модель процессора и описание алгоритма эмулятора**

## **3.1. Описание модели работы процессора**

Процессор интересует нас, прежде всего, как набор регистров. Регистр – быстродействующее запоминающее устройство, реализованное на электронных компонентах.

Все регистры имеют размер слова (16 разрядов), за каждым из них закреплено определенное имя. По назначению и способу использования регистры можно разбить на следующие группы:

* регистры общего назначения (AX, BX, CX, DX, SI, DI, BP, SP);
* сегментные регистры (CS, DS, SS, ES);
* указатель команд (IP);
* регистр флагов (Flags).

Программа состоит из машинных команд. Программа загружается в оперативную память компьютера. Затем программа начинает выполняться, то есть процессор выполняет машинные команды в той последовательности, в какой они записаны в программе.

Для того чтобы процессор знал, какую команду нужно выполнять в определённый момент, существует счётчик команд – специальный регистр, в котором хранится адрес команды, которая должна быть выполнена после выполнения текущей команды. То есть при запуске программы в этом регистре хранится адрес первой команды.

В процессоре Intel 8086 в качестве счётчика команд используется регистр IP.

Цикл выполнения команды – это последовательность действий, которая совершается процессором при выполнении одной машинной команды. При выполнении каждой машинной команды процессор должен выполнить как минимум три действия: выборку, декодирование и выполнение.

Выборка команды. Блок управления извлекает команду из памяти (из очереди команд), копирует её во внутреннюю память процессора и увеличивает значение счётчика команд на длину этой команды (разные команды могут иметь разный размер).

Декодирование команды. Блок управления определяет тип выполняемой команды, пересылает указанные в ней операнды в АЛУ и генерирует электрические сигналы управления АЛУ, которые соответствуют типу выполняемой операции.

Выборка операндов. Если в команде используется операнд, расположенный в оперативной памяти, то блок управления начинает операцию по его выборке из памяти.

Выполнение команды. АЛУ выполняет указанную в команде операцию, сохраняет полученный результат в заданном месте и обновляет состояние флагов, по значению которых программа может судить о результате выполнения команды.

Запись результата в память. Если результат выполнения команды должен быть сохранён в памяти, блок управления начинает операцию сохранения данных в памяти.

Суммируем полученные знания и составим цикл выполнения команды:

* Выбрать из очереди команд команду, на которую указывает счётчик команд.
* Определить адрес следующей команды в очереди команд и записать адрес следующей команды в счётчик команд.
* Декодировать команду.
* Если в команде есть операнды, находящиеся в памяти, то выбрать операнды.
* Выполнить команду и установить флаги.
* Записать результат в память (по необходимости).
* Начать выполнение следующей команды с п.1.

## **3.2. Алгоритм эмулятора**

**PUSH**

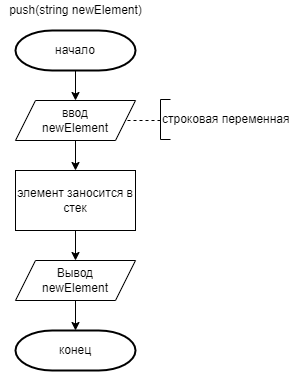


Рисунок 3.2.1 — Блок схема алгоритма команды PUSH.

**POP**

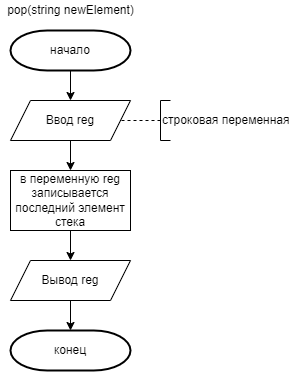


Рисунок 3.2.2 — Блок схема алгоритма команды POP.

**NOT**  


Рисунок 3.2.3 — Блок схема алгоритма команды NOT.

**MOV**

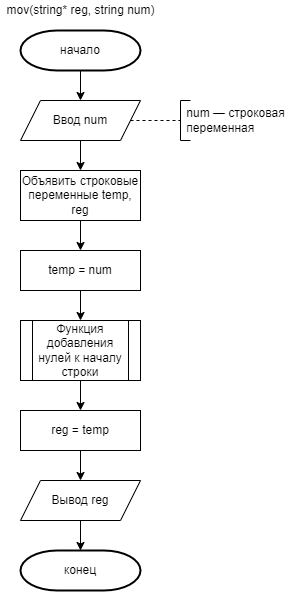


Рисунок 3.2.4 — Блок схема алгоритма команды MOV.

**ADD**

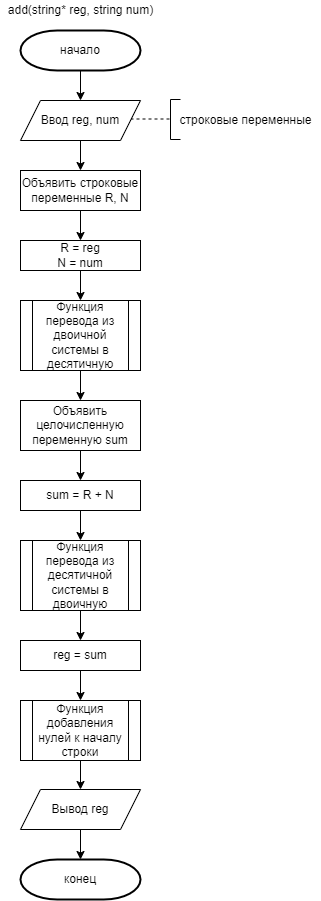


Рисунок 3.2.5 — Блок схема алгоритма команды ADD.

**SUB**

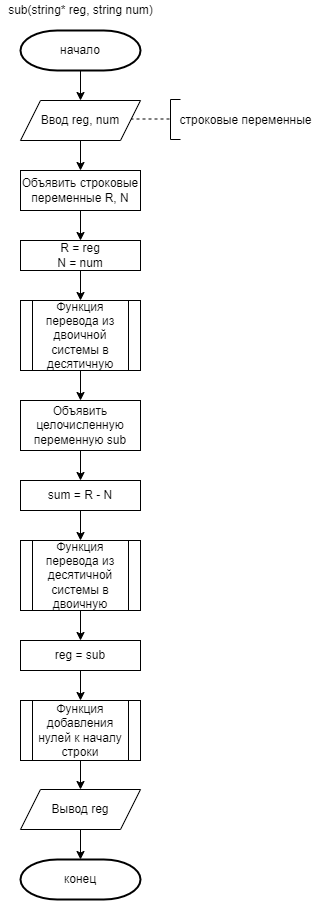


Рисунок 3.2.6 — Блок схема алгоритма команды SUB.

**INC**

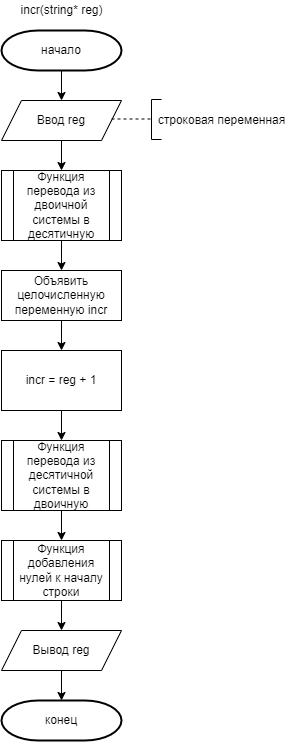


Рисунок 3.2.7 — Блок схема алгоритма команды INC.

**AND**

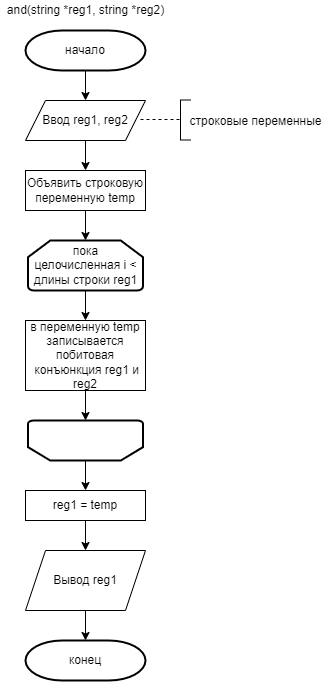


Рисунок 3.2.8 — Блок схема алгоритма команды AND.

**OR**

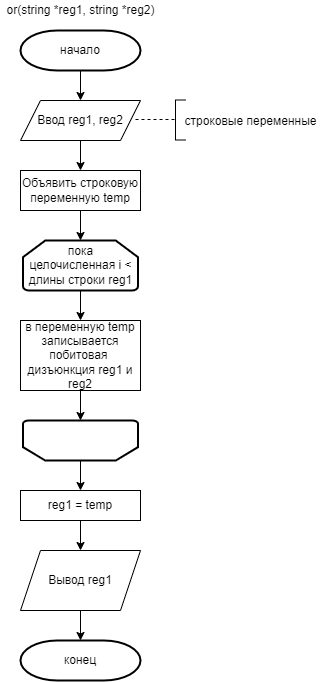


Рисунок 3.2.9 — Блок схема алгоритма команды OR.

## **3.3. Описание интерфейса эмулятора**

Программа должна выполнять ограниченный набор команд эмулятора Intel 8086 на языке высокого уровня. Программа написана на языке C++ и реализована в среде разработки CLion.

Ниже представлены функции, которые должна выполнять программа:

* Помещение операнда в стек;
* Извлечение операнда из стека;
* Копирование одного операнда в другой;
* Сложение операндов;
* Вычитание операндов;
* Инкремент;
* Логическое И;
* Логическое ИЛИ;
* Логическое отрицание.

Входными данными для программы являются: текстовый файл.

Выходными данными для программы являются: значения регистров после выполнения функций.

# **Раздел IV. Разработка программы эмулятора и его верификация**

## **4.1. Разработка программы**

Для решения поставленной задачи необходимо выполнить следующие действия:

1) организовать класс, содержащий в себе поля:

* + число, регистр AX (аккумулятор);
  + число, регистр ВX (регистр базы);
  + число, регистр CX (счётчик);
  + число, регистр DX (регистр данных);
  + стек;

2) организовать вывод текущего состояния регистров на экран;

3) организовать функции реализующие следующие команды:

* + Помещение операнда в стек;
  + Извлечение операнда из стека;
  + Копирование одного операнда в другой;
  + Сложение операндов;
  + Вычитание операндов;
  + Инкремент;
  + Логическое И;
  + Логическое ИЛИ;
  + Логическое отрицание.

## **4.2. Тестовые команды**

В таблице 2 представлена система тестов, использующая стратегию серого ящика для проверки работоспособности программы.

Таблица 2 — Система тестов

|  |  |  |  |
| --- | --- | --- | --- |
| Назначение теста | Входные данные | Фактические результаты | Эталон |
| проверка команды MOV | MOV AL, 3 | AL = 00000011 | AL = 00000011 |
| проверка команды ADD | ADD AL, 1 | AL: 00000100 | AL: 00000100 |
| проверка команды SUB | SUB AL, 4 | AL: 00000000 | AL: 00000000 |
| проверка команды INC | INC AL | AL: 00000001 | AL: 00000001 |
| проверка команды OR | OR AL, BL | AL: 00000000 | AL: 00000000 |
| проверка команды AND | AND AL, BL | AL: 00000000 | AL: 00000000 |
| проверка команды NOT | NOT AL | AL: 11111111 | AL: 11111111 |
| проверка команды PUSH | PUSH AX | в стек занесено значение AX | в стек занесено значение AX |
| проверка команды POP | POP AX | в регистр AX занесено значение из стека | в регистр AX занесено значение из стека |

Проведенное тестирование показывает, что программа работает верно.

## **4.3. Верификация**

Результат работы программы:

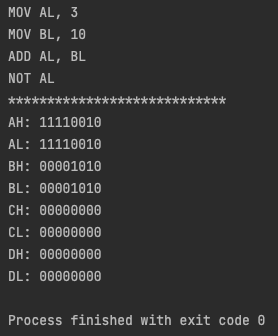
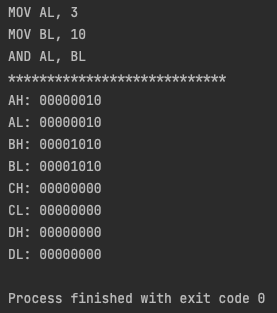
****

Рисунок 4.3.1 – результат работы программы

# **Заключение**

В данной курсовой работе были рассмотрены вопросы создания имитации определенного набора команд для машины IBM PC на базе процессора Intel 80186. Были разработаны алгоритмы и код для данного набора команд, что позволило утверждать о возможности создания подмножества команд для процессора Intel 80186, позволяющего эмулировать данный вычислительный процессор. Также были проведены проверки созданного кода благодаря эмулятору семейства процессоров Intel 86 на вычислительной машине с операционной системе Windows 10 Pro. Метод проверки – серый ящик.

# **Литература**

1. Поляков А.В. Ассемблер для чайников, 2012 г. — 52с
2. Подбельский В. В. «Курс программирования на языке C», 2018г.—384с
3. Молчанова С. И. «Методические указания», 2020г. — 53с

# **Другие источники**

1. <https://www.emanual.ru/download/1255.html>
2. <https://rykovodstvo.ru/exspl/62966/index.html>
3. <http://old.tpt.tom.ru/umk/informat/uchebnik/arhi.htm>
4. <https://cs.istu.ru/index.php?project=asm&page=_theory_pacm_programmodel_>

# **Приложение А. Ограниченный набор команд для эмуляции**

В таблице 3 представлен ограниченный набор команд для эмуляции и их описание.

Таблица 3 — набор команд для эмуляции и их описание.

|  |  |
| --- | --- |
| Команда | Описание |
| MOV | Копирует operand2 в operand1.  Алгоритм: operand1 = operand2 |
| ADD | Сложение.  Алгоритм: operand1 = operand1 + operand2 |
| SUB | Вычитание.  Алгоритм: operand1 = operand1 – operand2 |
| NOT | Инвертирует каждый бит операнда. |
| AND | Логическое И между всеми битами двух операндов. Результат записывается в 1-й операнд. |
| INC | Инкремент.  Алгоритм: operand = operand + 1 |
| OR | Логическое ИЛИ между всеми битами двух операндов. Результат записывается в 1-й операнд. |
| PUSH | Помещение операнда в стек. |
| POP | Извлечение операнда из стека. |

# **Приложение Б. Текст программы**

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#include "math.h"

#include <stack>

#include "unistd.h"

#include <sstream>

using namespace std;

// класс стек

class Stack {

const int MAX = 16;

string\* stack = new string [MAX];

int top = -1;

int size = 0;

// IsEmpty

// если пустой то true

bool IsEmpty() {

if (this->top != 0) {

return 0;

}

else {

return 1;

}

}

// IsFull

// если полный то true

bool IsFull() {

if (this->top == this->MAX) {

return 1;

}

else {

return 0;

}

};

// PUSH

public :

void push(string newElement) {

if (this->IsFull() == true) {

cout << ("Стек заполнен") << endl;

}

else {

this->top++;

this->stack[this->top] = newElement;

}

this->size++;

};

// POP

string pop() {

if (this->IsEmpty() == true) {

cout << ("Стек пустой") << endl;

return "0";

}

else {

string temp = this->stack[this->top];

this->top--;

return temp;

}

this->size--;

};

// PEEK

void peek() {

cout << "Стек:" << endl;

for (int i = 0; i < size; i++) {

cout << this->stack[i] << endl;

}

};

};

// класс регистров

class Register {

public :

// AL

string lR = "00000000";

string\* lowReg = &lR;

// AH

string hR = "00000000";

string\* highReg = &lR;

// AX

string xR = lR + hR;

string \* xReg = &xR;

// если меняется xReg, то true

bool X = false;

};

//intel8086, 16bit

class RMA8086 {

// регистры

public:

Register AX;

Register BX;

Register CX;

Register DX;

Stack stack;

// если двоичное, то false, иначе true

bool d = false;

// PUSH – занести в стек

// 9

void push (string\* newElement) {

this->stack.push(\*newElement);

};

// POP – вытащить из стека

// 10

void pop (string \*reg) {

\*reg = this->stack.pop();

};

string\* checkreg(string xReg) {

string \*p = &xReg;

this-> d = false;

if (xReg == "AX") {

this->AX.X = true;

string temp = \*(this->AX.highReg) + \*(this->AX.lowReg);

\*(this->AX.xReg) = temp;

return this->AX.xReg;

} else

if (xReg == "BX") {

this->BX.X = true;

string temp = \*(this->BX.highReg) + \*(this->BX.lowReg);

\*(this->BX.xReg) = temp;

return this->BX.xReg;

} else

if (xReg == "CX") {

this->CX.X = true;

string temp = \*(this->CX.highReg) + \*(this->CX.lowReg);

\*(this->CX.xReg) = temp;

return this->CX.xReg;

} else

if (xReg == "DX") {

this->DX.X = true;

string temp = \*(this->DX.highReg) + \*(this->DX.lowReg);

\*(this->DX.xReg) = temp;

return this->DX.xReg;

} else

if (xReg == "AL"){

return this->AX.lowReg;

} else

if (xReg == "BL"){

return this->BX.lowReg;

} else

if (xReg == "CL"){

return this->CX.lowReg;

} else

if (xReg == "DL"){

return this->DX.lowReg;

} else

if (xReg == "AH"){

return this->AX.highReg;

} else

if (xReg == "BH"){

return this->BX.highReg;

} else

if (xReg == "CH"){

return this->CX.highReg;

} else

if (xReg == "DH"){

return this->DX.highReg;

} else {

this-> d = true;

return p;

}

}

// перевод 10->2

string fromDecToBin(unsigned int decNum) {

string binNum;

while (decNum > 0) {

int r = decNum % 2;

binNum += to\_string(r);

decNum /= 2;

}

reverse(binNum.begin(), binNum.end());

return binNum;

}

// перевод 2->10

string fromBinToDec(string binNum) {

int decNum = 0;

for (int i = binNum.length() - 1; i >= 0; i--) {

if (binNum[binNum.length() - i - 1] == '1') {

decNum += pow(2.0, 1.0 \* i);

}

}

return to\_string(decNum);

}

// разделение

void split (Register AX) {

if (this->AX.X == true) {

string temp = "";

string temp1 = \*(AX.xReg);

for (int i = 0; i < 8; i++) {

temp += temp1[i];

}

\*(AX.highReg) = temp;

temp = "";

for (int i = 8; i < 16; i++) {

temp += temp1[i];

}

\*(AX.lowReg) = temp;

}

this->AX.X = false;

}

// добавление нулей

void addNULL (string\* reg) {

int length = reg->length();

string temp = \*reg;

\*reg = "";

if (length != 8 and stoi(fromBinToDec(temp)) <= 255) {

int n = 8 - length;

for (int i = 0; i < n; i ++) {

\*reg += "0";

}

\*reg += temp;

}

else {

if (length != 16 and stoi(fromBinToDec(temp)) > 255) {

int n = 16 - length;

for (int i = 0; i < n; i ++) {

\*reg += "0";

}

\*reg += temp;

} else \*reg = temp;

}

};

// mov

void mov(string\* reg, string\* num) {

string\* N = num;

if (this-> d == true) \*N = fromDecToBin(stoi(\*num));

addNULL(N);

\*reg = \*N;

}

// add

void add(string\* reg, string\* num) {

string R = fromBinToDec(\*reg);

string N = \*num;

if (this-> d == false) N = fromBinToDec(\*num);

int sum = stoi(R) + stoi(N);

if (sum < 65535){

\*reg = fromDecToBin(sum);

addNULL(reg);

} else {

string temp = fromDecToBin(sum);

string temp1 = "";

for (int i = 0; i < 16; i++) {

temp1[i] = temp[i];

}

\*reg = temp1;

}

}

// sub

void sub (string\* reg, string\* num) {

string R = fromBinToDec(\*reg);

string N = \*num;

if (this-> d == false) N = fromBinToDec(\*num);

int sub = stoi(R) - stoi(N);

\*reg = fromDecToBin(sub);

}

// inc

void inc (string \*reg) {

int decr = stoi(fromBinToDec(\*reg)) + 1;

\*reg = fromDecToBin(decr);

}

// NOT

void NOT (string \*reg) {

// двоичное число

string temp = \*reg;

int n = temp.length();

// инверсия

for (int i = 0; i < n; i++) {

if (temp[i] == '1') {

temp[i] = '0';

}

else {

temp[i] = '1';

};

};

\*reg = temp;

};

// OR

void OR (string \*reg1, string \*reg2) {

// двоичное число

string \*temp;

// ИЛИ

for (int i = 0; i < temp->length(); i++) {

\*(temp +i) = to\_string(stoi(\*(reg1+i)) + stoi(\*(reg2+i)));

};

reg1 = temp;

};

// AND

void AND (string \*reg1, string \*reg2) {

// двоичное число

string temp = "";

string r1 = \*reg1;

string r2 = \*reg2;

// И

for (int i = 0; i < reg1->length(); i++) {

//int andd = stoi(r1[i]) \* stoi(r2[i]);

string str = to\_string((r1[i] - '0') \* (r2[i]-'0'));

temp += str[0] ;

};

\*reg1 = temp;

};

vector<std::string> split(string stringToBeSplitted, char delimeter)

{

std::vector<std::string> splittedString;

int startIndex = 0;

int endIndex = 0;

while( (endIndex = stringToBeSplitted.find(delimeter, startIndex)) < stringToBeSplitted.size() )

{

std::string val = stringToBeSplitted.substr(startIndex, endIndex - startIndex);

splittedString.push\_back(val);

startIndex = endIndex + 1;

}

if(startIndex < stringToBeSplitted.size())

{

std::string val = stringToBeSplitted.substr(startIndex);

splittedString.push\_back(val);

}

return splittedString;

}

string showReg(string \*reg) {

string temp = \*reg;

return temp;

}

void checkReg(RMA8086 emu) {

emu.split(emu.AX);

emu.split(emu.BX);

emu.split(emu.CX);

emu.split(emu.DX);

}

int main() {

string path;

string str;

vector <string> arr;

vector <string> elems;

path = «путь до файла";

ifstream f(path);

while (f) {

getline(f,str);

arr.push\_back(str);

}

f.close();

int n = arr.size() - 1;

for (int i = 0; i < n; i++) {

cout << arr.at(i) << endl;

}

//int n = arr.size();

int m = 6;

RMA8086 emu = RMA8086();

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

for (int i = 0; i < n; i++) {

arr.at(i).erase(std::remove(arr.at(i).begin(), arr.at(i).end(), ','), arr.at(i).end());

std::vector<std::string> spl = split(arr.at(i), ' ');

if (spl.at(0) == "PUSH") {

emu.push(emu.checkreg(spl.at(1)));

}

else if (spl[0] == "POP") {

emu.push(emu.checkreg(spl.at(1)));

checkReg(emu);

}

else if (spl[0] == "MOV") {

emu.mov(emu.checkreg(spl.at(1)), emu.checkreg(spl.at(2)));

checkReg(emu);

}

else if (spl[0] == "ADD") {

emu.add(emu.checkreg(spl.at(1)), emu.checkreg(spl.at(2)));

checkReg(emu);

}

else if (spl[0] == "SUB") {

emu.sub(emu.checkreg(spl.at(1)), emu.checkreg(spl.at(2)));

checkReg(emu);

}

else if (spl[0] == "DEC") {

emu.dec(emu.checkreg(spl.at(1)));

checkReg(emu);

}

else if (spl[0] == "INC") {

emu.inc(emu.checkreg(spl.at(1)));

checkReg(emu);

}

else if (spl[0] == "XCHG") {

emu.xchg(emu.checkreg(spl.at(1)), emu.checkreg(spl.at(2)));

checkReg(emu);

}

else if (spl[0] == "NOT") {

emu.NOT(emu.checkreg(spl.at(1)));

checkReg(emu);

}

else if (spl[0] == "OR") {

emu.OR(emu.checkreg(spl.at(1)), emu.checkreg(spl.at(2)));

checkReg(emu);

}

else if (spl[0] == "AND") {

emu.AND(emu.checkreg(spl.at(1)), emu.checkreg(spl.at(2)));

checkReg(emu);

}

else if (spl[0] == "NOP") {

emu.NOP();

}

}

cout << "AH: " << showReg(emu.AX.highReg) << endl;

cout << "AL: " << showReg(emu.AX.lowReg) << endl;

cout << "BH: " << showReg(emu.BX.highReg) << endl;

cout << "BL: " << showReg(emu.BX.lowReg) << endl;

cout << "CH: " << showReg(emu.CX.highReg) << endl;

cout << "CL: " << showReg(emu.CX.lowReg) << endl;

cout << "DH: " << showReg(emu.DX.highReg) << endl;

cout << "DL: " << showReg(emu.DX.lowReg) << endl;

return 0;

}