Пособие по курсу «Языки системного программирования»

Игорь Жирков

13 сентября 2012 г.

Оглавление

Введение

Зачем нужен этот курс? Основные цели этого курса таковы:

- Показать в общем и целом, как работает процессор с точки зрения системного программиста;
- Научить студентов азам программирования на Ассемблере и C;
- Дать студентам кругозор по части стилей программирования в целом, позволяя им выбирать те инструменты, которые наилучшим образом подходят для решения конкретной задачи;
- Осветить работу компьютера и операционной системы настолько, чтобы у студентов сложилось общее представление о том, как запускаются и исполняются программы.

В данном пособии находится необходимый теоретический минимум по курсу, вопросы для самостоятельной проработки, а также список некоторых источников, полезных для самообразования в данной области.

Глава 1

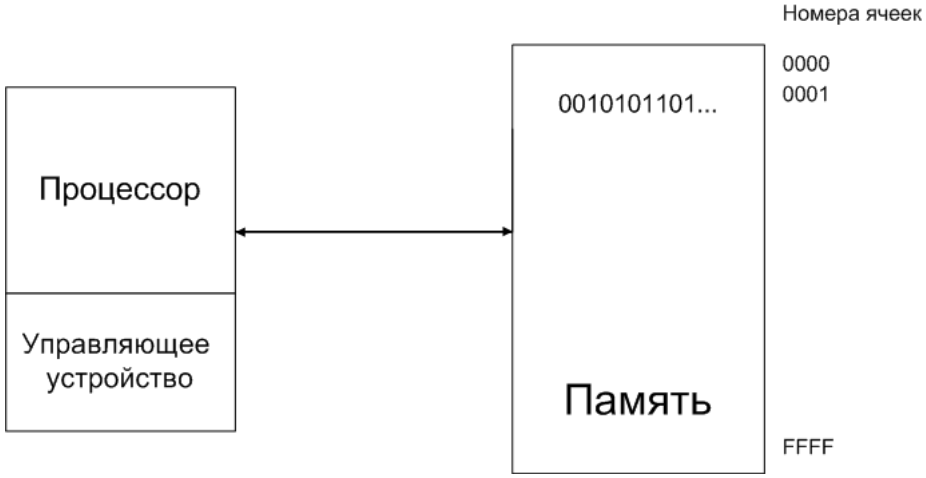
Архитектура компьютера

1.1 Архитектура фон Неймана

Представим, что мы перенеслись на много лет назад, когда компьютеров еще не существовало. Существовала лишь необходимость каким-то образом организовать автоматический вычислительный процесс. Тогда на бумаге существовало множество совершенно различных моделей вычислительных систем. Это и машина Тьюринга, и лямбда-исчисление Алонзо Чёрча, и другие вычислительные модели. Это показывает нам, что организовать набор транзисторов или ламп в различные схемы можно совершенно различными способами, и компьютеры отнюдь не обязаны быть именно такими, какие они есть сейчас. В связи с тем, что ранее электронные компоненты были не очень надёжны, а также со своей относительной простотой в создании и написании для неё программ, прижилась концептуальная модель компьютера, получившая название «Архитектура фон Неймана». Её аппаратные реализации также были достаточно надёжны.

Архитектура компьютера — концептуальная структура вычислительной машины, определяющая то, как информация обрабатывается и преобразуется в данные, а также то, как взаимодействует аппаратура и программное обеспечение.

Взглянем на схему компьютера с такой архитектурой:



На схеме вы видите процессор, который умеет выполнять команды; память, которая хранит данные и команды, и управляющее устройство, которое указывает процессору, какие команды выбирать из памяти для выполнения.

Какие основополагающие принципы такой архитектуры?

Двоичное кодирование В памяти хранятся только нули и единицы.

Однородность памяти Никаким способом нельзя узнать, команда ли лежит в данной ячейке, или данные. Процессор может попытаться выполнить данные, что, скорее всего, вызовет ошибку, так как произвольные данные врядли соответствуют корректно закодированной команде.

Адресуемость памяти Каждая ячейка имеет свой номер. Ячейки нумеруются последовательно: за нулевой ячейкой идёт первая и т.д. Единица адресации в реальных компьютерах — байт, то есть каждые 8 бит имеют свой адрес.

Последовательное программное управление Программа состоит из набора команд, которые выполняются последовательно.

Исключение – команды перехода, которые явно заставляют компьютер изменять порядок команд.

Принцип жесткости архитектуры Все связи и блоки на схеме остаются собой на протяжении работы компьютера. Никаких новых связей не возникает.

Помимо архитектуры фон Неймана существуют другие похожие архитектуры, например, гарвардская, их рассмотрение мы оставим за рамками нашего курса. Сейчас наша задача – проследить, как из фон Неймановской основы вырос типичный персональный компьютер, каковым он был 20 лет назад.

1.1.1 Модификация фон Неймановской модели

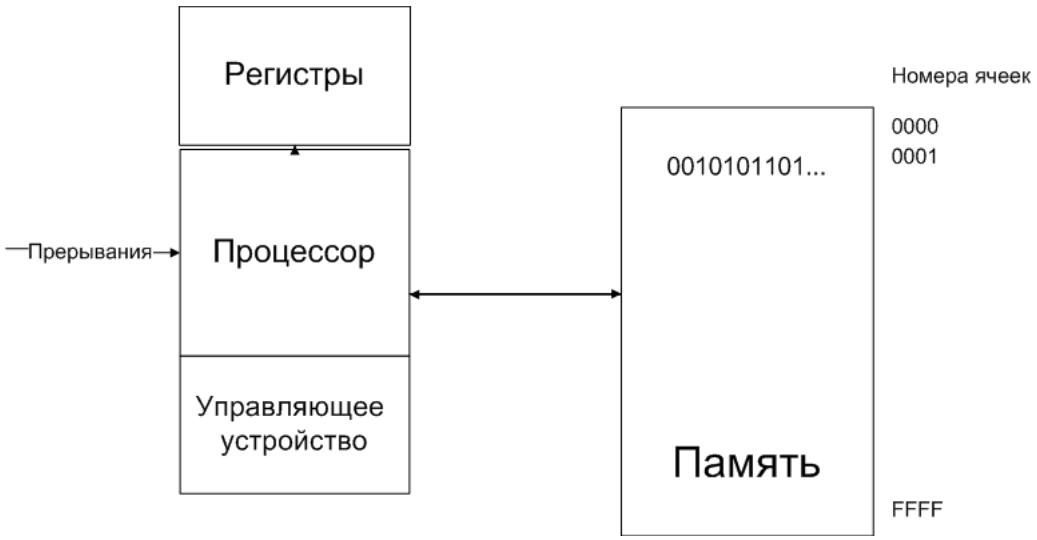
У архитектуры фон Неймана есть несколько серьёзных недостатков. Некоторые из них перечислены ниже:

- Система не интерактивна — никакого взаимодействия человека с системой, кроме прямого редактирования памяти;
- Узкое место системы — канал обмена между памятью и процессором. Для каждой команды необходимо не только обратиться в память, чтобы считать её, но и обращаться туда, чтобы считывать операнды, и, возможно, записывать результат.

Чтобы сгладить эти недостатки, было придумано несколько механизмов.

Прерывания — нарушение исполнения работы программы по специальному сигналу процессору. Процессор останавливает исполнение текущей программы и начинает исполнять программу обработки прерывания;

Регистры — ячейки памяти, расположенные непосредственно на кристалле процессора. Обращение к ним очень быстрое и не использует канал обмена с памятью. Основные операции компьютер осуществляет именно с содержимым регистров.



Разумеется, использование регистров в худшем случае замедляет работу компьютера. Быстрее было бы совершить действие с ячейкой памяти, нежели еще копировать её содержимое в регистр и из регистра. Однако существует объективное свойство компьютерных программ — локальность — которое делает использование регистров оправданным.

Локальность бывает двух видов: пространственная и временная.

Временная локальность заключается в том, что если произошло обращение по некоторому адресу, то следующее обращение по тому же адресу с большой вероятностью произойдет в ближайшее время.

Пространственная локальность заключается в том, что если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам.

В программе, обычно, б'ольшую часть времени мы работаем с очень маленьким набором переменных. Если обращение в память за данными понадобится нам лишь в одном случае из десяти, то в этом случае мы потеряем немного производительности, зато в других — значительно ускоримся. Таким образом, в среднем мы получим прирост производительности.

Это очень распространённая ситуация в инженерной практике. За счёт дополнительных механизмов мы ухудшаем производительность в худшем случае, но поднимаем её в среднем. С похожей ситуацией вы столкнётесь, когда будете изучать кэши и виртуальную память в курсе «Организация ЭВМ и систем».

1.2 Процессор Intel 80386

Мы начнём программировать на языке ассемблера для персонального компьютера, каким он был 20 лет назад. Это оправдано, потому что в дальнейшем процессор изменялся эволюционно, почти всегда сохраняя совместимость с более старыми моделями. Поэтому мы начнём с того, что обзорно изучим процессор Intel 80386 со стороны программиста.

Замечание. *Информация, помещённая в данном разделе, носит лишь обзорный характер. Категорически рекомендуется ознакомиться с самой документацией на*

процессор. Можно использовать документацию на следующую модель процессора – 80386, она доступна онлайн на сайте Массачусетского университета. `http://css.csail.mit.edu/6.858/2012/readings/i386/toc.htm`

1.2.1 Регистры

Некоторые регистры, которые можно использовать в арифметических операциях, называются регистры общего назначения. Их 8 штук, и все они имеют размер 16 бит. Хотя они и универсальны, у них также есть и особое назначение — в некоторых командах они используются неявно, без указания их имени. Из этого проистекают их названия.

- AX — аккумулятор, обычно используется в арифметических действиях;
- BX — база для базовой адресации;
- CX — счетчик цикла;
- DX — данные, например, прерывания DOS часто получают данные именно из DX;
- SI — Source Index, адрес ячейки-отправителя в командах работы со строками (MOVSB, ...);
- DI — Destination Index, адрес ячейки-получателя в командах работы со строками (MOVSB, ...);
- SP — хранит адрес последнего занесённого в стек элемента;
- BP — база для базовой адресации в случае, когда используются стековые фреймы. Хранит адрес начала текущего стекового фрейма (с этой темой вы подробно познакомитесь позднее).

К младшим и старшим 8 битам первых четырёх регистров (AX, BX, CX, DX) можно обращаться отдельно, используя имена AL, AH и т.п. Например, если DX=0103_{16}, то DH = 01_{16}, DL = 03_{16}.

Помимо этого существуют особые 16-битные регистры, с которыми вы также столкнётесь:

- IP — счётчик команд;
- Flags — в своих битах хранит флаги процессора.

  Вопрос 1. *Обратитесь к документации и прочитайте, какие флаги хранит этот регистр и за что они отвечают. Особое внимание уделите управляющим флагам DF, IF, TF.*

- CS, DS, SS, ES — 16-сегментные регистры, хранят кусок адреса начала сегмента кода, данных и стека текущей программы соответственно. Регистр ES используется для служебных целей программистом, например, в командах работы со строками (MOVSB, MOVSW...).

1.2.2 Сегментация

Так как регистры имеют разрядность 16 бит, а любые операции с адресами ячеек памяти задействуют регистры (хотя бы для хранения самого адреса), то количество ячеек, которые мы можем пронумеровать, ограничивается разрядностью регистра. Так, 16-битными регистрами мы можем перенумеровать $2^{16} = 65536$ байт. Однако программистам всегда не хватало памяти, и инженеры придумывали различные механизмы, чтобы расширить адресацию. Начиная с модели Intel 8086 стало возможным нумеровать 2^{20} ячеек памяти благодаря сегментным регистрам.

Теперь вместо того, чтобы указывать в регистре адрес ячейки относительно начала памяти (нулевого адреса), он указывается относительно начала определённого сегмента, то есть другого адреса.

Хотя для компьютера нет разницы, какие нули и единицы в памяти что кодируют, но программистам удобно держать в памяти раздельно код и данные. В любой программе, обычно, есть три области памяти:

- Код программы, машинные команды;
- Область данных;
- Область памяти, зарезервированная под стек.

По этой причине инженеры выделили три сегментных регистра, в которых программисту предлагается хранить адреса начала областей кода, данных и стека соответственно. Адрес начала сегмента формируется мз соответствующего сегментного регистра путём его умножения на 16.

Так, если CS = 0012_{16}, то предполагаемый компьютером адрес начала сегмента кода – 00120_{16}.

Различные команды подразумевают по умолчанию адреса относительно различных сегментов. Так, команда перемещения данных mov использует, конечно, смещение относительно начала сегмента данных, а команда перехода jmp — сегмента кода. Если из самой сути команды неясно, какой сегмент она использует, следует посмотреть раздел документации по соответствующей команде.

Теперь будем различать линейные адреса (20-разрядные) и адреса в форме сегмент:смещение и всегда обращать внимание на сегмент. Так, адрес следующей исполняемой команды лежит не в IP, а в паре CS:IP.

Например, CS=$042F_{16}$, IP=1212_{16}. Линейный адрес следующей исполняемой команды — $042F_{16} * 16_{10} + 1212_{16}$

```
042f0+
 1212=
```

05502 — адрес следующей исполняемой команды.

1.2.3 Аппаратный стек

Вообще, стек это структура данных типа Last In — First Out, которая ведёт себя подобно стопке тарелок. Новый элемент с помощью команды push помещается на вершину стека, а с помощью команды pop достаётся последний помещённый в стек элемент.

В наших компьютерах память линейно адресуема, это значит, что никакой стековой памяти у нас нет. Однако стек эмулируется с помощью машинных команд push,

pop и пары регистров SS:SP. SS:SP хранит адрес вершины стека, а команды push и pop работают по следующим алгоритмам:

- push
  1. SP уменьшается на 2;
  2. По адресу SS:SP кладётся значение (содержимое регистра или число, в зависимости от параметра команды).
- pop
  1. В регистр, указанный, в качестве параметра, кладётся значение, взятое по адресу из SS:SP;
  2. SP увеличивается на 2.

Из этого есть несколько важных выводов:

- Не бывает ситуации, когда "в стеке ничего нет тем более, если "мы туда ничего не положили". При выполнении, команда pop всегда будет следовать вышеуказанному алгоритму и выдавать соответствующий результат;
- Стек растёт вверх (к младшим адресам);
- В стек можно положить или вынуть оттуда только одно слово (2 байта).

1.2.4 Режимы работы процессора

Процессор 80386 может работать в реальном, защищённом, специальном или виртульном режимах. Мы поговорим о них позднее, пока что нам нужно запомнить несколько фактов:

1. Мы работаем в реальном режиме работы. Практически все регистры в нём 16-разрядные.
2. В защищённом режиме работы все регистры общего назначения становятся 32-разрядными, к их имени добавляется префикс E. Например, регистр AX расширился до EAX (32-разрядный), но мы по прежнему можем обращаться к младшей половине EAX как к AX, а также к половинам AX как к AH и AL.

1.2.5 Прерывания

Когда процессор получает прерывание, он останавливает выполнение программы чтобы выполнить обработчик прерывания. Каждое прерывание имеет жёстко фиксированный номер. Нам неважно, как именно процессор получает номер прерывания от контроллера прерываний. Имея номер X, мы обращаемся в память за адресом обработчика прерывания. В начале памяти лежит таблица векторов прерываний, каждый из которых занимает 4 байта (2 байта на адрес начала сегмента, 2 байта на смещение). Всего номеров прерываний 256, поэтому размер таблицы — 1 килобайт.

Алгоритм работы процессора при получении прерывания такой:

- Заносим в стек флаги;

- Сбрасываем флаги прерывания и трассировки;
- Заносим в стек CS и IP;
- Выбираем новые CS и IP из таблицы векторов прерываний.

Если мы не сбросим флаг прерывания, то мы сразу же можем получить другое прерывание, в то время, как нам нужно гарантированно выполнить начало обработчика прерывания и сориентироваться в ситуации. Если мы не сбросим флаг трассировки, то после каждой выполненной команды процессор будет генерировать прерывание с кодом 03.

Так как мы сбрасываем эти флаги, то нам необходимо прежде всего сохранить регистр флагов, иначе мы не сможем восстановить полное состояние программы в момент прерывания.

Прерывания можно вызвать вручную из вашей программы, для этого есть специальная команда int. Это используется, например, для системных вызовов (см. лабораторную работу №1).

В конце обработчика прерываний используется команда iret, которая восстанавливает из вершины стека IP, CS и FLAGS.

Лабораторная работа №1

В данной лабораторной работе вы найдёте примеры простейших программ на MASM, а затем на основе имеющегося материала напишете свою программу. В теоретической части вы встретите вопросы, ответы на которые вы должны найти сами. К защите лабораторной работы вы должны быть готовы ответить на любой из них (разумеется, не ограничиваясь только ими).

Для выполнения работы вам понадобятся по меньшей мере следующие команды:

- mov
- lea
- jmp
- jg/jb/ja/jl
- cmp
- test
- push
- pop
- add
- sub
- call

1.2.6 Hello, world!

По традиции начнём изучение языка ассемблера с программы "Hello, world!".

```
.model small
.stack 128

.data
mess db 'Hello world!$'

.code

main:

mov ax,@data
mov ds,ax
```

```
lea dx,mess
mov ah,09h
int 21h

mov ax,4c00h
int 21h

end main
```

1.2.7 Комментарий

.model small

Выбираем модель памяти small, что означает один сегмент кода и один сегмент данных.

Вопрос 2. *Что такое модель памяти?*

Вопрос 3. *Какие есть еще модели памяти?*

.stack 128

Выделяем 128 байт под стек. Это значит, что нам гарантировано, что если мы положим в стек не более 128 байт, то программа будет корректно работать.

Вопрос 4. *Что произойдёт, если размер стека не указать явно?*

Вопрос 5. *Может ли программа работать без стека?*

```
.data
mess db 'Hello world!$'
```

Объявляем начало сегмента данных и метку mess. Метка в ассемблере – имя для определённого адреса, так что в дальнейшем каждый раз при использовании имени mess вместо него будет подставляться смещение относительно начала сегмента данных.

```
.code

main:
```

Начало сегмента кода и метка main. В ассемблере можно произвольно смешивать описания кода и данных, каждый раз указывая начало соответствующего сегмента. Метка main это такой же адрес, впоследствии она будет нам важна – см. комментарий к последней строчке.

```
mov ax,@data
mov ds,ax
```

Нам необходимо настроить регистр DS, потому что иначе все смещения в сегменте данных будут отсчитываться от неправильного начала сегмента. В начале работы программы в DS лежит мусор. Вместо @data компилятор подставит корректный адрес начала сегмента данных, т.е. число. Например, команда:

```
lea dx,mess
```

использует адрес DS:mess (в формате адреса сегмент:смещение), поэтому для неё критично, чтобы значение DS было корректным. Напрямую переслать значение в DS нельзя, это не предусмотренно процессором, поэтому мы пересылаем его через регистр общего назначения.

Следующая команда загружает в DX смещение mess от начала сегмента данных.

lea dx,mess

Вопрос 6. *В чем разница между командами:*

lea dx,*mess*
mov dx, *mess*

? Что будет в регистре DX после выполнения второй команды?

Вопрос 7. *Что такое Little Endian и Big Endian?*

mov ah,09h
int 21h

С помощью этих двух команд мы перемещаем в регистр AH значение 09h (h — суффикс шестнадцатиричной системы счисления) и вызываем вручную прерывание с номером 21h.

Прерывание с номером 21h закреплено за DOS, а не за внешними устройствами. При его вызове происходит передача управления операционной системе, которая может выполнить одно из четко определённых действий. Каждое из этих действий имеет свой номер. Выбирается то действие, номер которого находится в регистре AH. В данном случае требуемая операция – вывод строки по адресу DS:DX до тех пор, пока не встретится символ доллара.

Замечание. *Вниманию тех, кто помнит про нуль-терминированные строки в языках высокого уровня. Доллар не заменяется на символ с кодом 0, доллар это доллар , символ экономической мощи Америки .*

Аналогичная операция по вызову прерывания DOS'а с требуемыми параметрами происходит в следующих двух строчках.

Директива end означает конец ассемблирования файла. Всё, что идёт после неё, игнорируется. Кроме того, её параметр указывает точку входа в программу – адрес, с которого она должна начать своё выполнение.

end main

Вопрос 8. *Что такое директива и чем она отличается от команды?*

Вопрос 9. *Объясните, что такое системный вызов и чем он отличается от обычной процедуры?*

Как вы видите, прерывания можно вызвать вручную с помощью команды int. Здесь они используются чтобы обратиться к системному вызову операционной системы DOS. Механизм системных вызовов DOS прост:

- В регистр AH заносится номер системного вызова;
- В другие регистры, возможно, заносятся параметры (зависит от системного вызова);

- Выполняется инструкция int 21h, вызывающая прерывание номер 21h, закреплённое за операционной системой DOS;
- Готово, некоторые системные вызовы возвращают в регистрах различные значения.

Вопрос 10. *Объясните, что такое системный вызов и чем он отличается от обычной процедуры?*

Вопрос 11. *В чем отличие инструкций call и int? Что такое call near и call far?*

Мы столкнулись уже с двумя системными вызовами:

- AH = 09h – вывод строки. Параметры: DS:DX = адрес начала строки. Концом строки считается символ доллара, то, что следует за ним, не выводится.

  Вопрос 12. *Попробуйте убрать символ доллара из строки, скомпилировать и запустить программу. Объясните результат.*

  Вопрос 13. *Как можно вывести на экран символ доллара, используя это прерывание с AH=09h? А используя другие прерывания или другие значения AH?*

- AH = 4ch – завершить выполнение программы и передать управление DOS.

  Параметры: AL = код возврата.

Вопрос 14. *Что такое код возврата?*

Рассмотрим также вызовы для вывода и считывания символов из консоли.

- AH = 01h – считывание символа из стандартного ввода, дублируя его в стандартный вывод.

  Возвращает: AL = считанный символ.

- AH = 02h – вывести символ на стандартный вывод.

  Параметры: DL = символ для вывода.

  Возвращает: AL = Последний выведенный символ.

Например, такая программа при запуске дублирует каждый введённый символ.

```
.model small
.code
start:
mov ah, 01h
int 21h
mov dl, al
mov ah, 02h
int 21h
jmp start
end start
```

Вопрос 15. *В каком случае после такого системного вызова AL не будет равен DL? (Подсказка: ищите полное описание системного вызова.)*

1.3 Задание

Ваша задача – написать калькулятор, работающий со стеком. Он должен реагировать на нажатия клавиш пользователем следующим образом:

1. Ввод числа от 0 до 9 — число (не код введённой цифры!) кладётся на вершину стека, счётчик элементов в стеке увеличивается;
2. Ввод символа +/-
   - если в стеке есть как минимум два числа, то последние два числа вынимаются оттуда, и результат их сложения/вычитания помещается в стек. Затем счётчик элементов в стеке изменяется так, чтобы отражать их реальное число. Затем вершина стека выводится на экран;
   - если в стеке слишком мало чисел, то программа выдаёт на экран строчку "error"и игнорирует этот символ.
3. Ввод символа 'p' — вывод на экран числа, находящегося в вершине стека, или ошибка, если стек пуст. Для этого вы можете пользоваться процедурой вывода содержимого регистра AX на экран, которая находится в приложении.
4. Ввод символа 'q' — выход из программы. Желательно при этом очистить стек.

Несколько советов:

- Обратите внимание на то, что для того, чтобы очистить ваш стек, совершенно необязательно вынимать оттуда каждое лежащее там число. Такие решения неразумны и неэффективны, и не будут приниматься!
- В ассемблере можно писать символы в кавычках подразумевая их ASCII-коды.
- Для организации условных используются команды cmp и test, которые совершают вычитание и логическое И операндов соответственно, выставляя соответствующие результату флаги. Сам результат, однако, они не сохраняют. Затем имея флаги, отражающие результат, используется соответствующая команда условного перехода.

  С помощью test можно быстро проверять содержимое регистра на ноль.

1.3.1 Состав работы

- Отчёт
- Листинг
- Таблица трассировки
- Блок-схема алгоритма

1.4 Приложение

Процедура для вывода регистра AX в стандартный вывод. Пример использования — call ax_out.

```
;-------------------------------------
ax_out  proc  uses dx cx
; Outputs AX
;-------------------------------------

.code
push ax

mov ah,02h
mov cx, 12d

mainloop:

        pop dx
        push dx
        shr dx,cl
        and dl, 0Fh
        cmp dl, 10d
        jge add_37h
        add dl, 30h

        jmp print_4

        add_37h:
                add dl, 37h

        print_4:

                int 21h
        sub cx,3

loop mainloop

pop dx
push dx

and dl, 0fh
cmp dl, 10d
jge add_37h_last
add dl, 30h

jmp print_last

add_37h_last:
        add dl, 37h

print_last:
                int 21h

pop ax
ret
```