Теория компиляторов

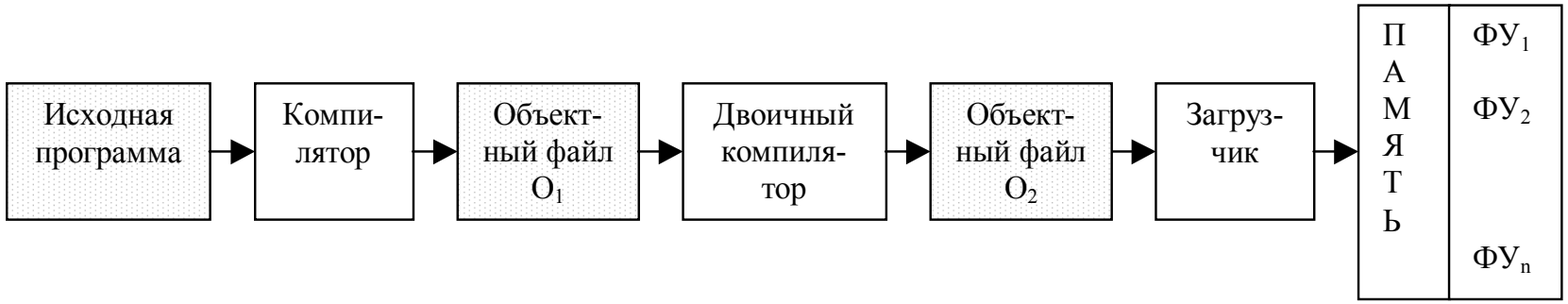
Часть II

Лекция 1.

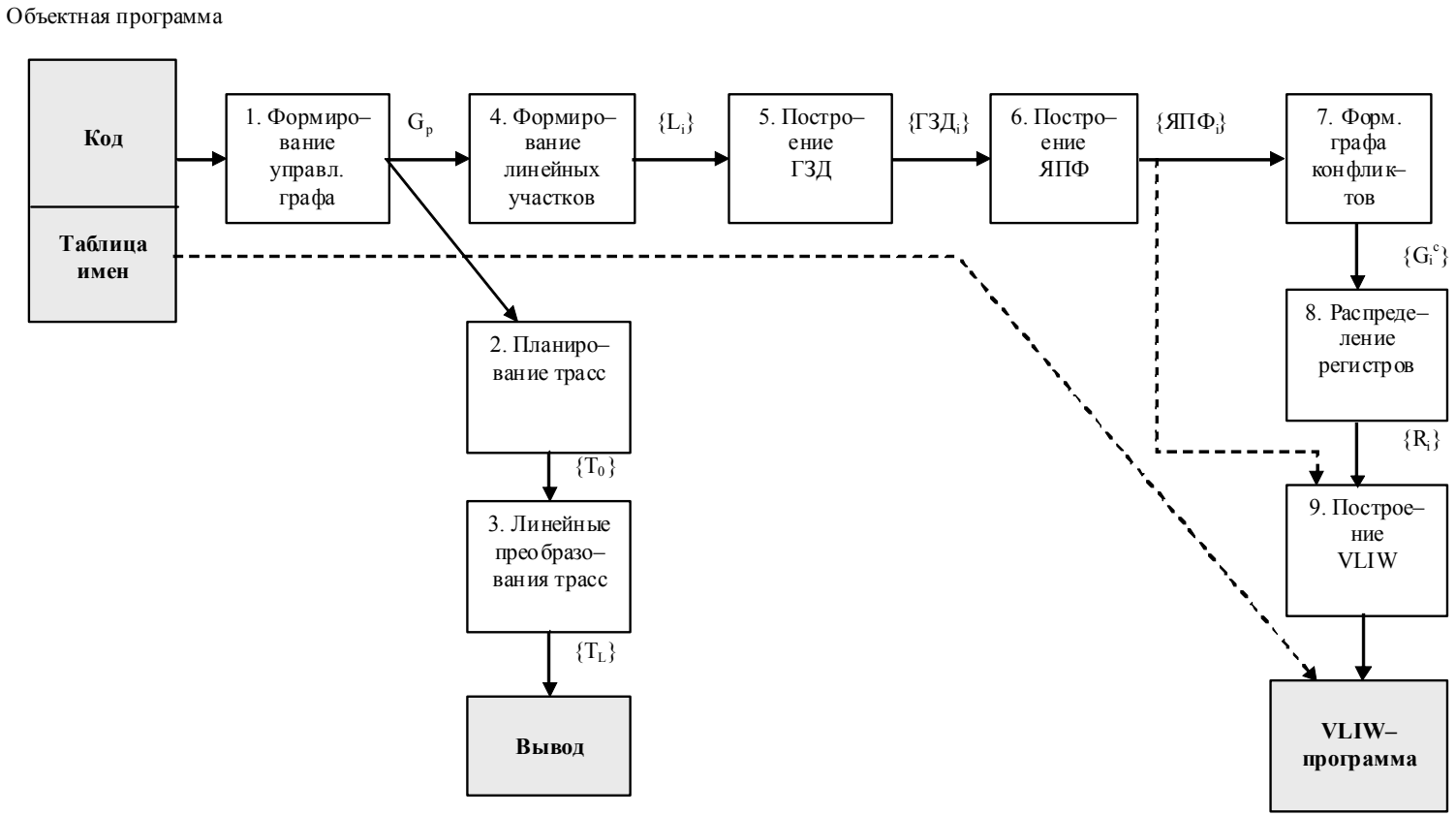
Цели и задачи курса

Создание программного комплекса, включающего в себя:

- Компилятор языка высокого уровня.
- **Двоичный компилятор**.
- Загрузчик.
- Виртуальную машину.

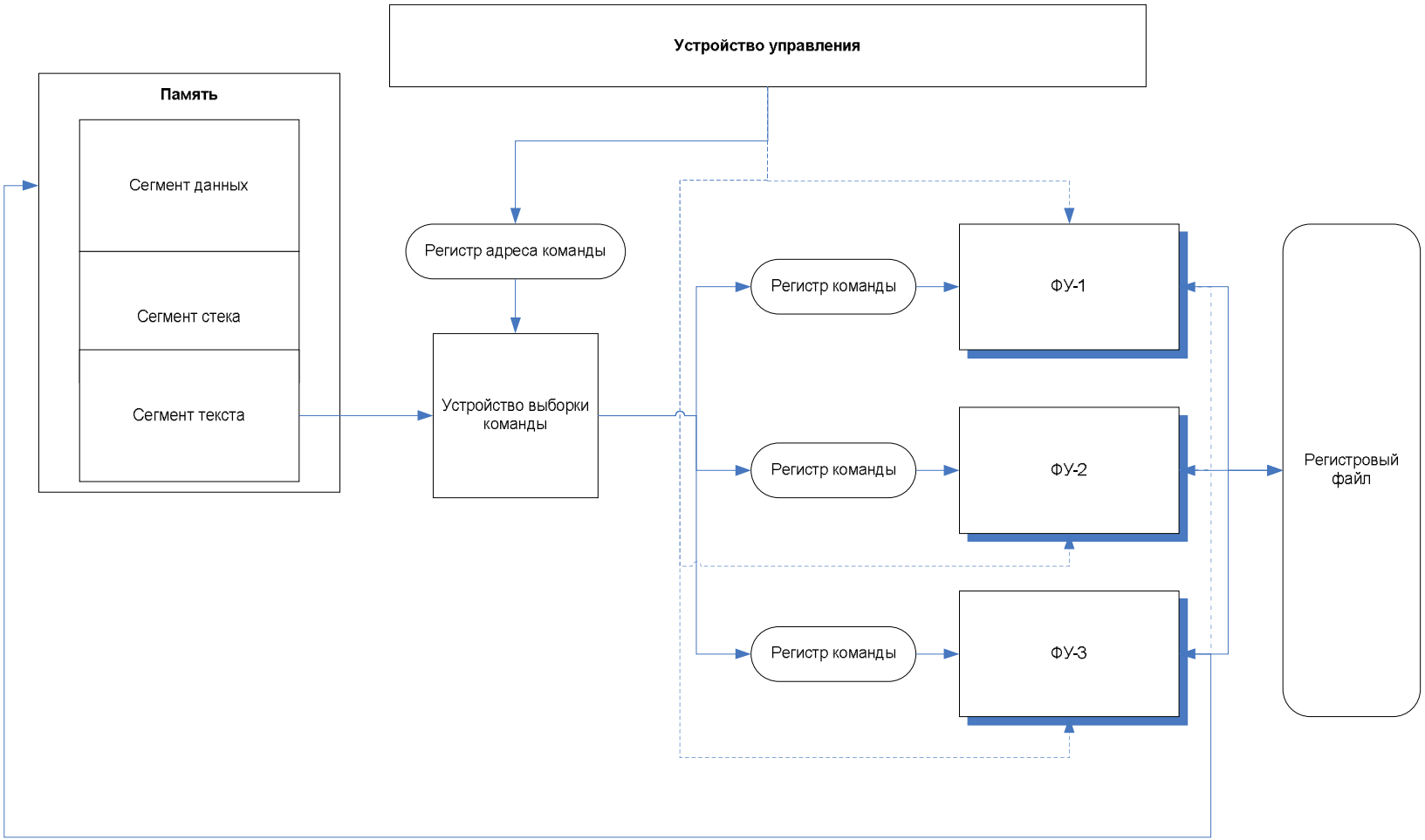


Двоичный компилятор



- Общая структура

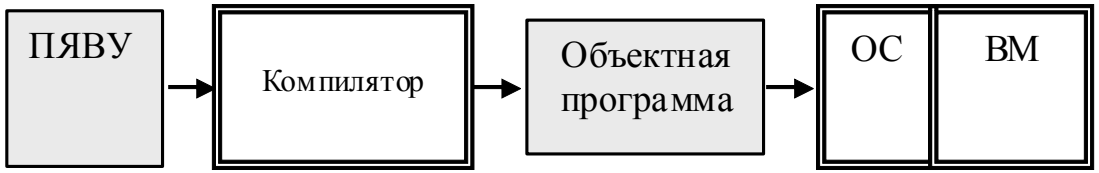
Виртуальная машина



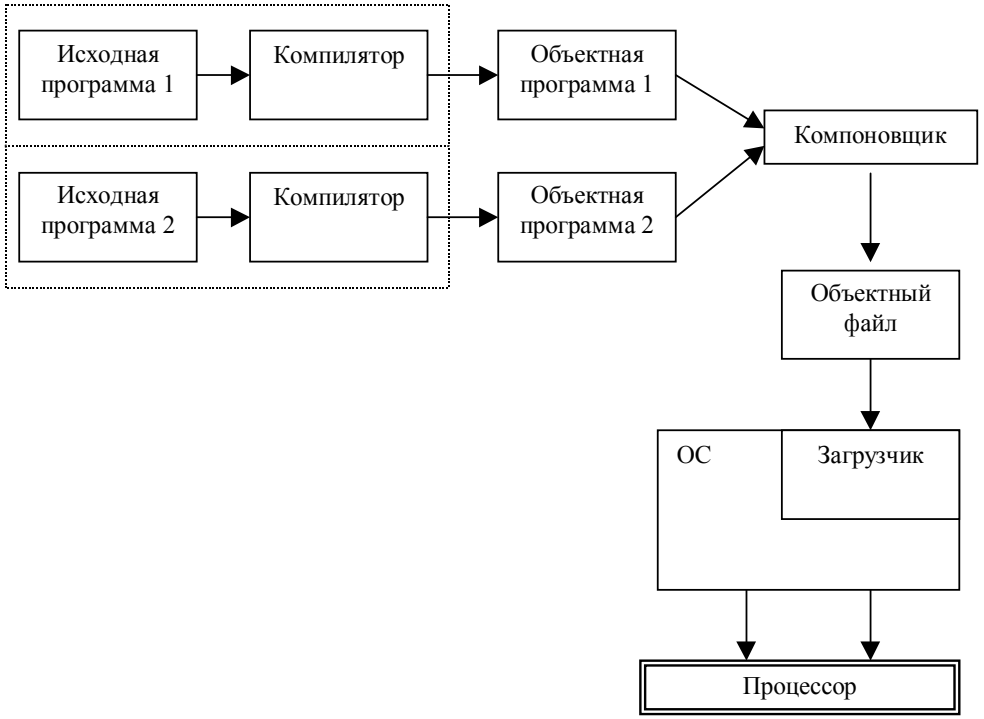
- Общая структура

Общая схема исполнения программы

- Простейшая схема



- «Реальная» схема



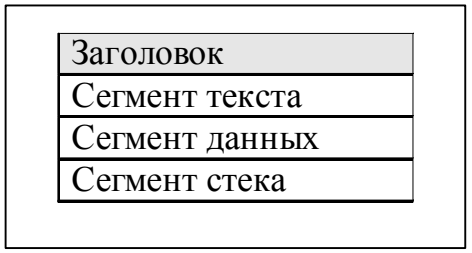
Образ программы в памяти

- C*егмент команд* (сегмент текста).
- *Сегмент данных.* Хранит *проинициализированные переменные*.
- C*егмент BSS* (Block Started by Symbol) – неинициализированные данные, заполняемые перед началом выполнения нулями.
- C*егмент стека*.

```
int global=1;
f()
{
  int i=0;
  static stat=-1; // статическая переменная
  stat++;
  …
}
main()
{
…
}
```

В ходе выполнения необходимо динамическое размещение памяти =>
Специально выделенная область - «*куча*» (*heap*).

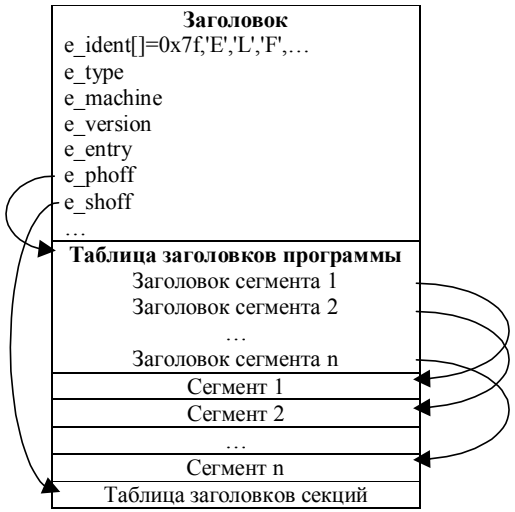
Общий формат объектного файла



Форматы исполняемых файлов. ELF

ELF (Executable and Linking Format) :

- Перемещаемый файл (relocatable file), хранящий инструкции и данные, которые могут быть связаны с другими объектными файлами.
- Разделяемый объектный файл (shared object file), содержащий инструкции и данные и который может быть связан с другими перемещаемыми и разделяемыми объектными файлами, образуя новый объектный файл..
- Исполняемый файл, содержащий полное описание, позволяющее операционной системе создавать образ процесса.

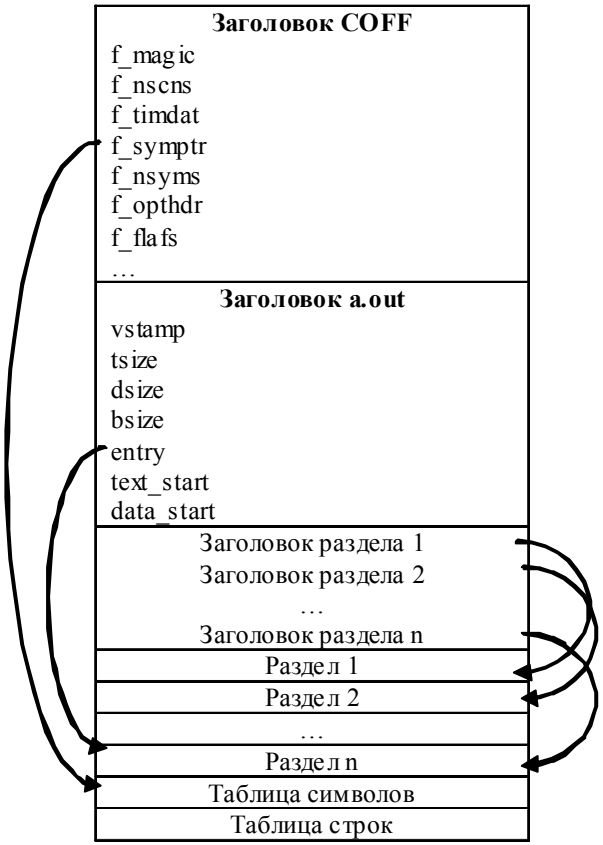


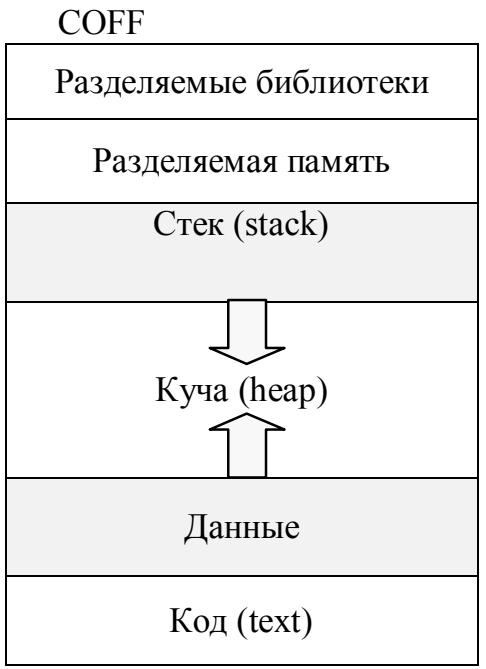
Структура исполняемого файла формата ELF

Виртуальная память процесса

Форматы исполняемых файлов. COFF

- Два основных заголовка - заголовок COFF (Common Object File Format) и стандартный заголовок системы UNIX - a.out.



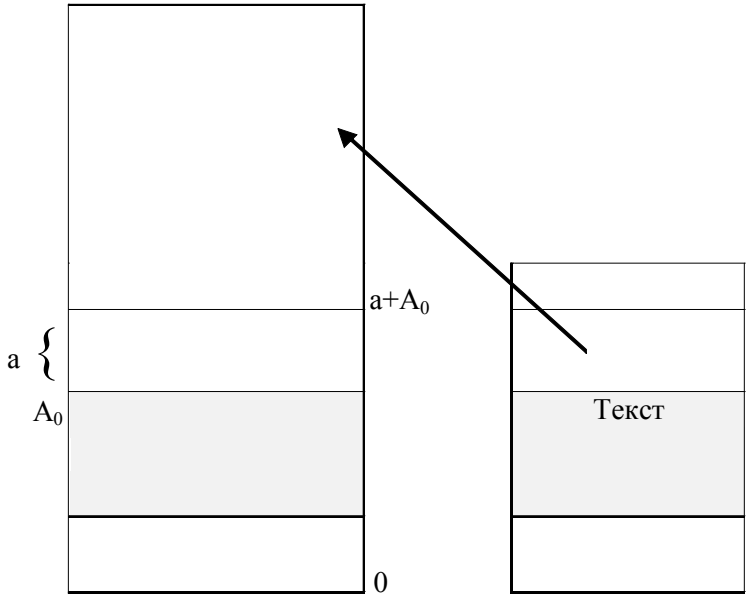


Виртуальная память процесса

ЗАГРУЗЧИКИ

Загрузчик - это программа (или компонента ОС), которая подготавливает объектную программу к выполнению и инициирует ее запуск. Основные функции:

- **Размещение**. Выделение места для программы в оперативной памяти. Это – самая простая операция, реализуемая обычно операционной системой.
- **Перемещение**. Настройка всех величин в программе, зависящих от физических адресов в соответствии с адресами выделенной программе памяти (от относительных *адресов* к *физическим).*
- **Загрузка**. Фактическое размещение структур объектной программы в памяти и инициация ее выполнения.



Типы загрузчиков

- ***«Компиляция - выполнение»***. Это - самый примитивный загрузчик, гибрид интерпретатора и компилятора. Компилятор порождает объектный код, помещая машинные команды и данные сразу в оперативную память.
- **Абсолютные загрузчики**. Компиляторы порождают точный образ размещаемой в памяти программы. Загрузчик просто размещает программу в памяти по адресу, определенному заранее.
- **Динамическая загрузка.** Механизм, позволяющий загружать исполняемый код по мере его необходимости. Механизм DLL (Dynamic Link Library).

Вывод:

- Проблема настройки адресов, динамического связывания (компоновки) и проч. необходимых преобразований объектного кода является весьма трудоемкой.
- Целесообразно переложить решение этих задач на некую отдельную, самостоятельную компоненту - ***компоновщик***.

Компоновщики

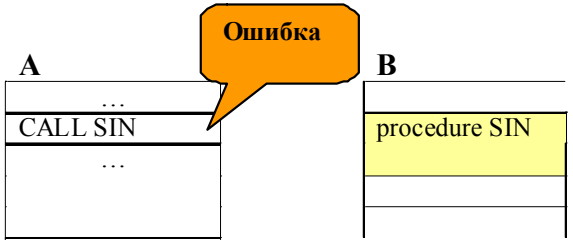
- Два вида адресов:
  - локальные адреса;
  - адреса внешних символов.

Компоновщику необходимо знать:

- Размер каждого модуля.
- Информацию о внешних символах.
- Какие символы из модулей могут быть доступны извне.

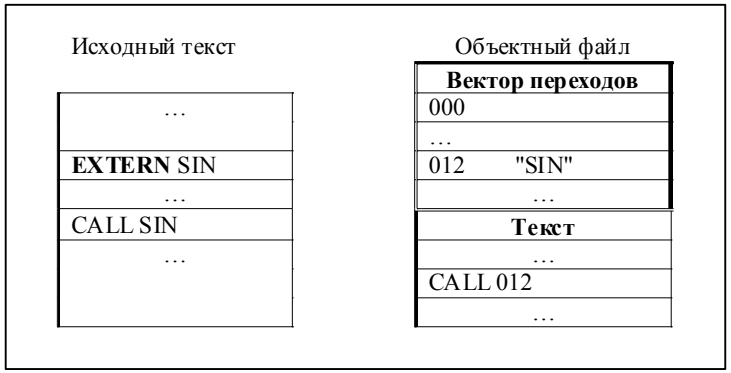
Вызовы подпрограмм

- Для вызова п/п необходимо:
  - хранить адрес точки возврата,
  - сохранить значение регистров, локальных переменных,
  - положить все это в стек и переходить к работе двухступенчатой системы.
- Если п/п определены в одном модуле, то проблем с их вызовами нет, т.к. адреса подпрограмм известны.

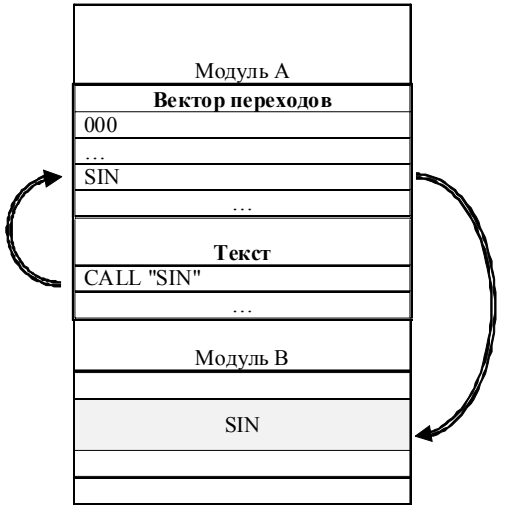


Пример. Два объектных модуля – ***A*** и ***B***.

- Из модуля ***A*** и ***B*** происходит вызов подпрограммы *SIN()*, определенной в модуле ***B***.
- *"CALL SIN"* => выдача сообщения об ошибке (адрес *SIN* в модуле ***A*** не определен).
- Чтобы транслятор не ругался, используется декларация *extern*



- => В объектный файл добавляется т.н. ***вектор переходов*** – специальная область, содержащая имена внешних символов, объявленных как *extern*.
- Компилятор помещает в ВП внешнее имя и заменяет обращение к нему указанием смещения в ВП.



Именно компоновщик может заменить символические имена в ВП виртуальными адресами подпрограмм. Однако эта простая двухступенчатая схема не решает проблемы обращения к внешним переменным.

Некоторые архитектурные принципы. Микропрограммирование

- 1957 г., Н.Я. Матюхин, Лаборатории управляющих машин и систем АН СССР, М. Уилкс, (Манчестерский университет, Великобритания).
- СЭВМ «**Тетива**», 1960 г., система ПВО.

Суть микропрограммного управления:

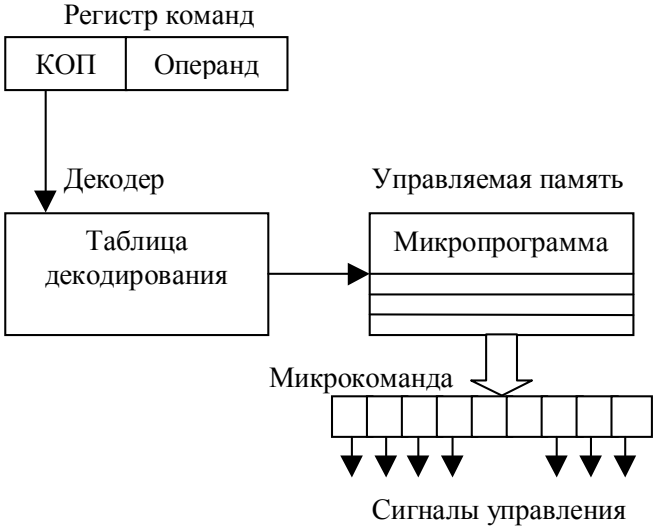
- Каждая микрокоманда становится командой-подпрограммой, а адрес следующей за адресом выполняемой в данный момент микропрограммы помещается в регистр возврата.

Достоинства микропрограммного:

- Упрощение реализации сложных команд.
- Повышение эффективности применения языков высокого уровня (вплоть до непосредственной микропрограммной интерпретации входных программ). Так строятся машины языков высокого уровня (Паскаль–, ЛИСП–, ПРОЛОГ– машины и т.п.).

Структура команды

- Микрокоманда состоит из поля кодов (сегментов или слогов) управляющих сигналов и поля управления переходом.



RSEL – выбор рабочих регистров (определение операндов микрокоманды, описание загрузки аргументов для АЛУ);
ALUF – команда АЛУ;
SPEC – специальный слог (может и не использоваться);
NEXT – поле управления переходом (адрес следующей команды).

Режимы выполнения микропрограмм

- Любая микрокоманда может стать командой–подпрограммой. Для вызова п/п необходим **регистр возврата**. Обычно для этого используется слог SPEC, куда будет записан адрес NEXT команды возврата.
- Для управления микропрограммой необходим **регистр адреса микрокоманды** (РАМК). В него загружается либо содержимое поля NEXT, либо содержимое специального сегмента.

Режимы выполнения микропрограмм:

- **Последовательное** выполнение. Адрес следующей микрокоманды находится в сегменте NEXT текущей команды. Этот адрес записывается в РАМК.
- **Условный** переход (ветвление). При выполнении условия перехода (для этого обычно анализируется состояние АЛУ) в регистр РАМК записывается содержимое поля SPEC, а не NEXT.
- **Вызов** подпрограммы. Адрес следующей команды (адрес начала подпрограммы) берется из SPEC. При этом адрес возврата загружается в специальный регистр (стек) RETURN.
- **Возврат** из подпрограммы.

RSEL	ALUF	BS	F1	F2	LL	LT	NEXT
(7)	(4)	(4)	(4)	(5)	(1)	(1)	(13)

Микрокоманда спецпроцессора «Катана» (39 разрядов):

- RSEL – выбор регистров в регистровый файл
- ALUF – команда АЛУ
- BS – указатель источника информации
- F1, F2 – выполнение специальных команд (хранение адреса следующей команды при условных переходах (ветвлениях), и при вызове п/п (РАМК ← F2)
- NEXT – адрес следующей команды

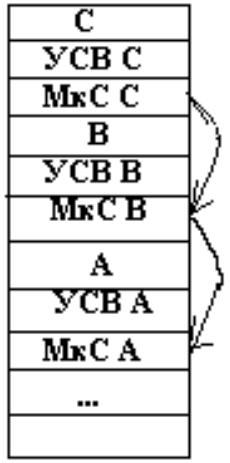
Микропрограммное обеспечение процессора

- Реализация сложных программных конструкций. Воплощение на аппаратном уровне ЯВУ (обычно стековых).
- Конструкции типа goto, *if … else, block, procedure* и т.п. (обычно эти м/п записываются в польской форме и подлежат интерпретации).

Для выполнения инструкций формируется стек процедур, содержащий 3 области:

- рабочая область;
- управляющее слово возврата (УСВ);
- маркер стека (МкС) – указывает область видимости переменных, окружающих данную процедуру.

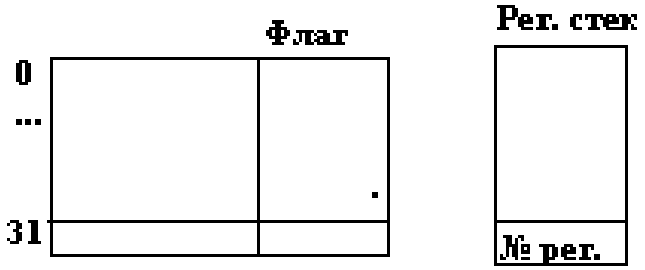
```
program A
        loc x,y
        program B
        ...
        end
program C
        ...
        end
        call B
        ...
end
```



Регистровые файлы и семафоры

Линейный регистровый файл.

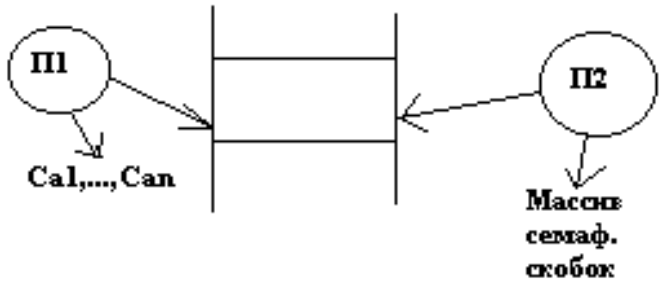
- Каждый регистр снабжается флагом.
- Аппаратура ЦП определяет свободный ресурс по флагу и в специальный регистровый **стек** помещает номер используемого регистра
- Когда стек переполняется, старые данные откачиваются в другое место, а когда пуст – подкачиваются.
- Регистровый стек позволяет отказаться от конкретных номеров регистров стека. => Мы имеем в этом случае *безадресную* организацию команд.



Семафорные скобки

- Конфликты при обращении к областям памяти.
- Программные семафоры (ОС)
- ----------
- Аппаратная реализация семафоров.

  Для получения доступа к области памяти процесс должен обратиться к **массиву семафорных скобок** и если они отсутствуют или открыты, то доступ разрешается.



Байт-код

- **Байт-код** (*byte-code*) - это машинно-независимый код низкого уровня.
- Байт-код занимает промежуточное положение между результирующим объектным (исполняемым) кодом и интерпретируемой программой.
- Для выполнения инструкций байт-кода требуется наличие специальной программы – интерпретатора, называемого также виртуальной машиной.

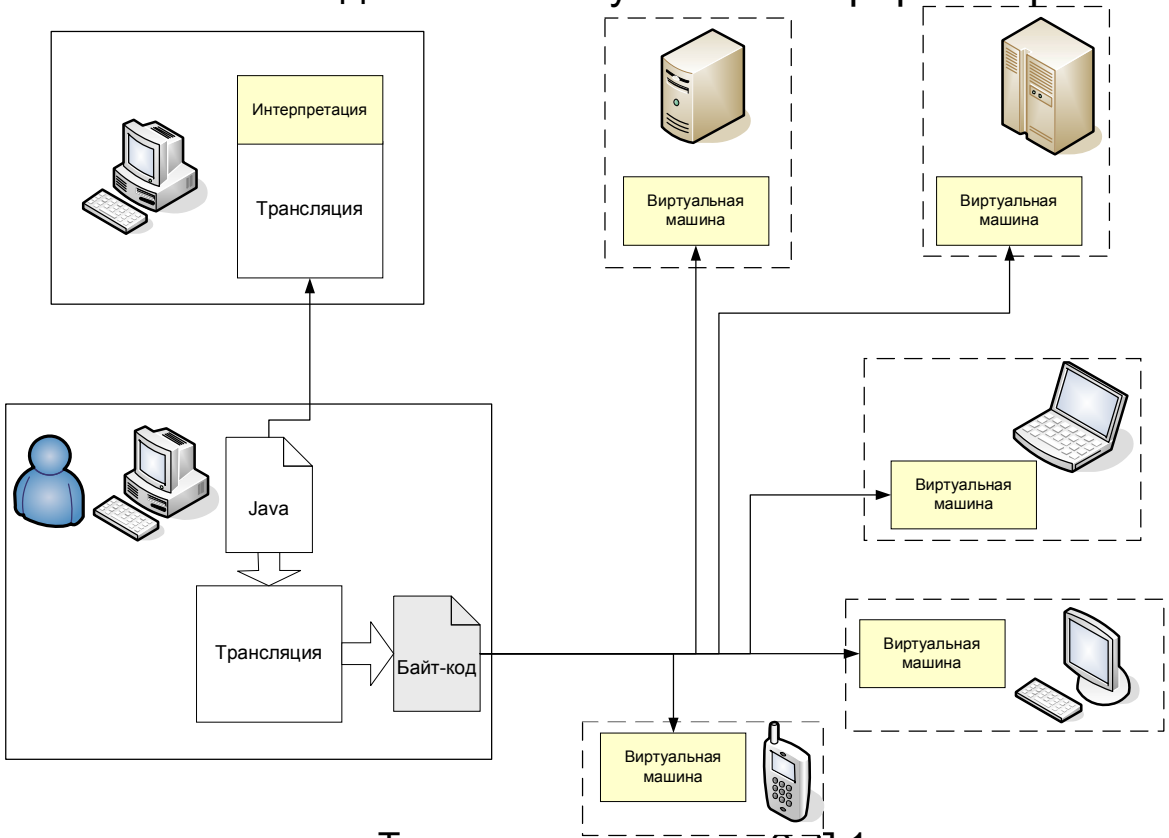
Простота и переносимость

Основное преимущество байт-кода - легкая переносимость.

Большая часть работы по обработке исходного текста программы выполняется транслятором, создающим байт-код, то на ВМ приходится лишь интерпретация этого «почти машинного» кода.

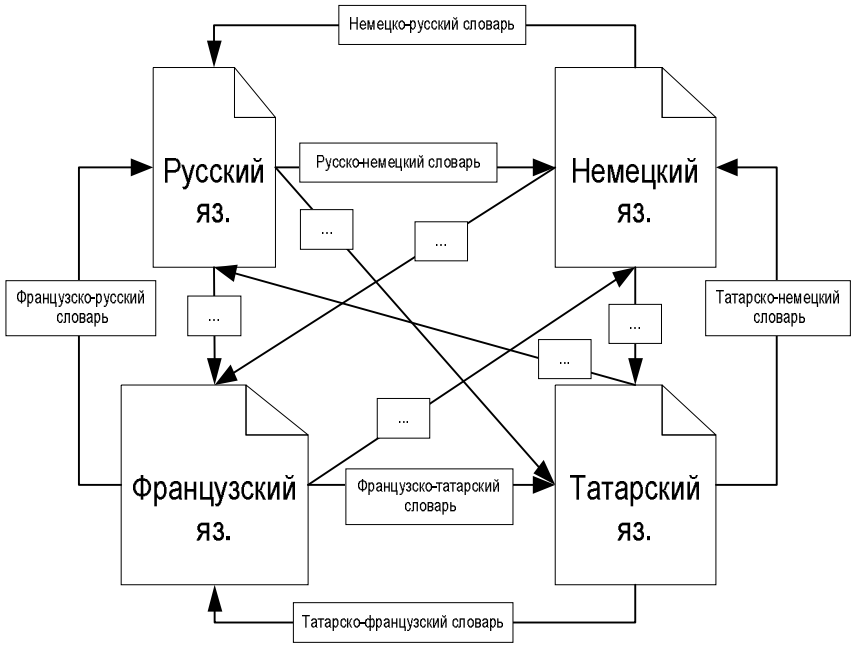
Фактически – это разбиение на отдельные этапы трансляции и интерпретации.

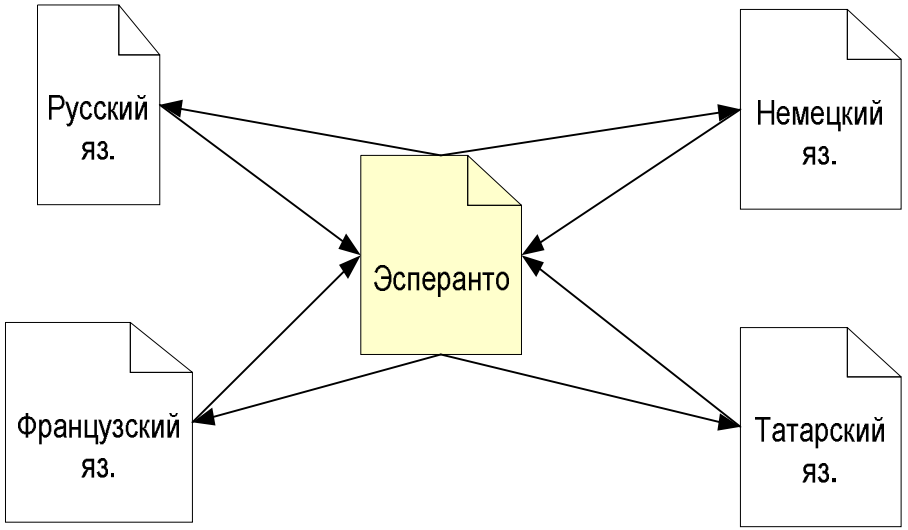
Многие интерпретируемые системы (Perl, PHP, Python и др.) транслируют исходный текст в байт-код и затем запускают интерпретатор байт-кода.



Б-К – язык-посредник

- 4 языка: 12 словарей.
- Если есть язык-посредник, то 8 словарей.
- Если N языков, то без посредника надо иметь N*(N-1) словарей (почти N^2), а с посредником – N*2.



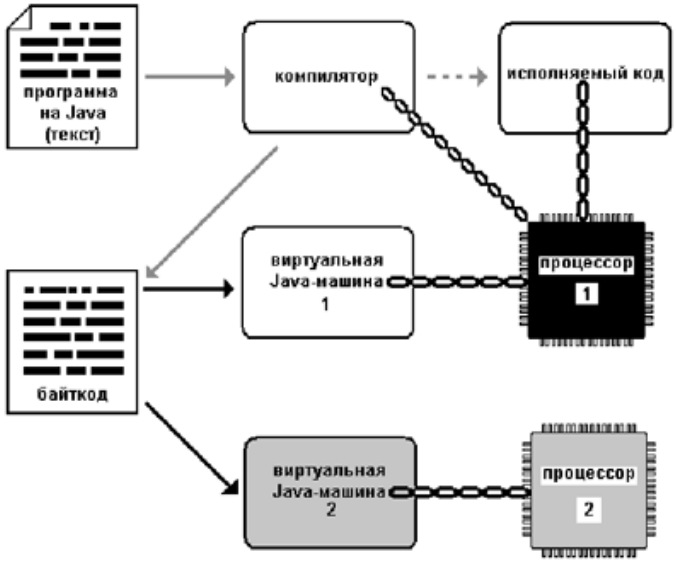


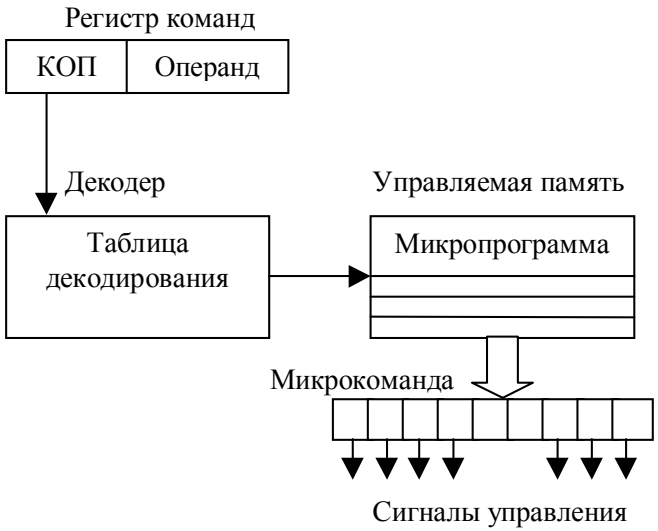
Простота Б-К

- Простота байт-кода позволяет реализовывать его интерпретаторы на уровне микропрограммного обеспечения процессоров. Благодаря этому существуют специальные Java-, Forth- и проч. машины (процессоры).
- Большинство инструкций байт-кода эквивалентны одной или нескольким машинным командам (командам ассемблера).
- Само название «байт-код» определено тем, что длина каждого кода операции — один байт.
- Более того, в байт-коде имеется тенденция реализовывать все команды по возможности в одном байте (при высокой частоте выполнения команд), что позволяет создавать компактный объектный код. (Разумеется, вся команда может занимать более одного байта).

История Б-К

- Концепция аппаратно-независимого исполняемого кода появилась еще в начале 1970-х годов.
- Разработки Н.Вирта по созданию виртуальной машины для языка Паскаль, а также Л.Петера Дойча в связи с созданием Лисп-машины.
- Тогда речь шла больше не о программной переносимости, а об идее персонального компьютера как специализированной машины с аппаратно реализованным языком высокого уровня.





Виртуальная Смолток-машина

Функции виртуальной машины

- Функция интерпретации. Интерпретатор считывает команды языка (байт-код) и выполняет их.
- Управление объектами. Блок управления объектами создает необходимые объекты и передает их интерпретатору, а ставшие ненужными объекты собирает и использует для дальнейшей работы.
- Система базовых операций. В нее входят операции ввода-вывода, управления процессами и другие базовые операции. В системе также регистрируются в качестве элементарных методов (primitive method - примитивы) те операции, которые нельзя реализовать на самом Смолтоке (или их реализация неэффективна) и которые реализуются в виде программ вне системы Смолток-80.

Примеры групп команд виртуальной Смолток-машины

1. Проталкивание в стек переменных экземпляра-получателя. Переменные экземпляра фиксируются для каждого экземпляра, и в каждом объекте для них отводится область памяти. Данная команда проталкивает в стек считанные переменные экземпляра, в частности – получателя.

- Байт-коды 0-15, 128:
- 0-15 [0000iiii] Помещение в стек переменной-экземпляра получателя с номером #iiii.
- 128 [10000000] [jjkkkkkk] Помещение в стек переменной экземпляра получателя, временной переменной, литерала, глобальной переменной, указываемой литералом [jj] с номером #kkkkkk.

2. Проталкивание в стек временной переменной. Временные переменные создаются в момент вызова метода.

- Байт-коды 16-31, 128:
- 16-31 [0001iiii] Помещение в стек временной переменной с номером #iiii.

3. Проталкивание символов в стек. Символ - это селектор сообщения или константа с объектным указателем.

- Байт-коды 32-63, 128:
- 32-63 [001iiiii] Помещение в стек литерала с номером #iiiii.

4. Вызов метода с использованием селектора сообщения, находящегося в области литералов. Команда производит поиск селектора сообщения, начиная со словаря класса получателя. Если поиск успешен, то производится вызов соответствующего метода.

- Байт-коды 131, 132, 134, 208-255.

5. Помещение в стек активного контекста. Команда помещает в поля текущего контекста значения регистров и затем помещает в стек указатель этого контекста.

- Байт-код 137.

6. Команды перехода и условного перехода.

- Байт-коды 144-175:
- 144-151 [10010iii] Переход по адресу iii+1.
- 152-159 [10011iii] Выталкивание из стека, переход по адресу iii+1 при значении false вытолкнутой вершины.
- 160-167 [10100iii] [jjjjjjjj] Переход по адресу (iii-4)*256+jjjjjjjj.

7. Посылка заявки на вычисление. Команда реализует арифметические операции "+" и "-". Если получатель не является целым числом, то выполняются действия, аналогичные обычной посылке заявки.

- Байт-коды 192-207:
- 192-207 [1100iiii] Посылка специальной заявки #iiii.

ПРОБЛЕМЫ АРХИТЕКТУРЫ ФОН НЕЙМАНА

Дж.фон Нейман, отчет 1945 г. для группы, связанной с Муровской электротехнической школой Пенсильванского университета.

ЭНИАК -> ЭДВАК

1. Хранимая программа: запись команд совместно с данными.
2. Линейная память.
3. Последовательное выполнение программы.
4. Отсутствие различий между данными и командами.
5. Отсутствие различий в семантике данных

Гарвардская архитектура
МАРК-1

1. Хранилище инструкций и хранилище данных представляют собой разные физические устройства.
2. Канал инструкций и канал данных также физически разделены.