Введение в архитектуру ЭВМ

Юрий Анатольевич Богоявленский, заведующий кафедрой Информатики и математического обеспечения, к.т.н., доцент: ybgv@cs.petrsu.ru, https://cs.petrsu.ru

Веб страница дисциплины

https://cs.petrsu.ru/~chistyak/architecture/

Лекция 1. Введение

План лекции

Мотивация

В работе *Peter J. Denning et al. Computing as a Discipline, Final Report of the ACM Task Force on The Core of Computer Science, PP 62, 1988,* дисциплина Информатика определена как содержащая девять предметных областей (subject area):

Algorithms and Data Structures	Numerical and Symbolic Computation
Architecture	Operating Systems
Artificial Intelligence and Robotics	Programming Languages
Database and Information Retrieval	Software Methodology and Engineering
Human - Computer Communication	

В 1991 г. на основе этого определения в работе *Allen B. Tucker (Editor and Co-chair), Bruce H. Barnes (Co-chair) et al. Computing Curricula. Report of the ACM/IEEE-CS Joint Curriculum Task Force, PP 162, 1991* были разработаны рекомендации по разработке учебных планов (curricula) для ВУЗов, причем, во всех их обновлениях в 2001, 2008, 2013 и 2020 гг. предметная область «Архитектура» сохраняла свою фундаментальную роль.

Причина такого положения в том, что концепции, понятия и навыки этой предметной области являются важнейшими компонентами квалификации профессионала по ИТ, в том числе, разработчика ПО.

Во первых, они являются базовыми элементами профессионального языка, на котором формулируются постановки задач и изложение других разделов Информатики.

Во вторых, они существенно используются при изучении других предметных областей Информатики, и, в частности, дисциплин, читаемых кафедрой ИМО:

“Операционные оболочки“.

“Взаимодействующие параллельные системы“.

“Операционные системы”.

“Системное программирование”.

“Компьютерные сети”.

“Архитектура современных ЭВМ“.

“Формальные языки и методы трансляции“.

И в третьих, существует целый ряд актуальных продуктов, которые можно разрабатывать только на архитектурном уровне с использованием языка ассемблера, либо языка С или других языков системного программирования.

Вот далеко не полный и постоянно расширяющийся перечень таких продуктов:

- начальный загрузчик, BIOS на ПЗУ;
- драйверы устройств и обработчики прерываний;
- ПО встраиваемых систем, микроконтроллеров, IoT-устройств, где критичны компактность и производительность;
- специфические для процессора команды, не реализованные в компиляторе. Например — команда побитового вращения — для алгоритмов шифрования;
- донастройка векторизованных функции для программ на языках более высокого уровня, таких как С, до набора команд SIMD;
- пилотажно-навигационные системы обеспечивающие передачу управляющих сигналов от органов управления в кабине экипажа к исполнительным приводам аэродинамических поверхностей (недопустима сборка мусора, подкачки и т. п.);

• криптографические алгоритмы, которые всегда должны выполняться за строго одинаковое время для выполнения, предотвращая временные атаки.

Разработка подобных продуктов требует освоения студентами навыков разработки на уровне архитектуры, формирования у них культуры архитектурного мышления.

В ИМИТ эта культура формируется дисциплиной "Введение в архитектуру ЭВМ/Архитектура компьютера", которая читается как базовая на первом или втором курсе.

При таком подходе необходимо строить схему первой дисциплины так, чтобы, при минимальном количестве деталей, дать студенту возможность освоить базовые архитектурные принципы.

NBNB. Инструмент — язык ассемблера

Как организована дисциплина

Б*огоявленский, Ю.А. Цифровая среда Института математики и информационных технологий. Дисциплина «Введение в архитектуру ЭВМ» — минималистский подход [Текст] / Ю.А. Богоявленский // Цифровые технологии в образовании, науке, обществе : материалы XII всероссийской научно-практической конференции (4-6 декабря 2018 года). - Петрозаводск, 2018. - С.31-34. - Режим доступа:* https://it2018.petrsu.ru/files/pages/it2018.pdf

Предмет изучения и используемые инструменты

Мы будем изучать архитектура набора команд (Instruction Set Architecture, ISA), которая является частью архитектуры процессора и включает в себя описания:

• команд процессора;

• регистров процессора;

• моделей оперативной памяти и режимов ее адресации;
• обработки прерываний и исключений;
• машинной арифметики;
• архитектурных типов данных.

Мы также рассмотрим такие тесно связанны с ISA вопросы как:

• кодирование символов;
• системные вызовы ядра ОС Linux и системные файлы `stdin/stdout`;
• архитектурный стек и соглашения о связях функций по стандарту `Application Binary Iinterface (ABI)` для `i386;`
• связь между раздельно оттранслированными объектными файлами.

Отметим, что в ISA не входит описание аппаратной микроархитектуры процессора (арифметического устройства , шины связи блоков и т.п.)

Мы будем использовать `ISA IA-32 (Intel Architecture, 32-bit)` которая с одной стороны содержит большинство современных архитектурных механизмов, масштабированных без существенной новизны в современной архитектуре `INTEL® 64`, а с другой - более «компактна» и удобна для обучения.

`IA-32` с 1985 по 2008 г. применялась для разработки операционных систем, компиляторов, офисных пакетов, инструментов разработки, игры, мультимедийных приложений и т.п.. Подчеркнем, что в современной архитектуре `INTEL® 64` сохраняется режим совместимости, который позволяет запускать 32-битные приложения без перекомпиляции.

Архитектура I`NTEL® 64` изучается на третьем курс в дисциплине по выбору “Архитектура современных ЭВМ“, которая. опирается на знания полученные на первом курсе.

Основной рабочий инструмент на лабораторных работах — язык семейства индустриальных ассемблеров `gas (man as, см. TARGET);`

Также будут использоваться:

• среда - `OpenSuSe Linux;`
• процессоры реальных ПЭВМ для решения задач;

- профессиональный синтаксис AT&T, который используется компилятором gcc для записи результата трансляции файла pr.c в файл на языке ассемблера pr.S командой gcc . . .-S pr.c если отменить фазу ассемблирования с помощью ключа – S;
- отладчик kdbg для демонстрации примеров.

Разделы дисциплины

1. Вводная часть.

   Системная среда центрального процессора. Регистры, оперативная память. Язык ассемблера, представление на нем команд, символьные имена, характеризация операндов. Системы счисления, порядки байтов Big/Little endian, машинная арифметика, дополнительный код, Unicode.

2. Простые программы.

   Системные вызовы на примере read/write, их взаимодействие со стандартными системными файлами stdin/stdout. Примеры на ввод символов с клавиатуры и вывод на экран. Директивы определения данных.

3. Режимы адресации и архитектурный стек

   Структура двоичных команд, режимы адресации, перемещение (OFFSET) и смещение (dicplacement). Пример вычисления адресов элементов матрицы, расположенной в оперативной памяти. Стековый доступ к оперативной памяти.

4. Функции

   Параметры функции, ее тело, вызов, возврат. Соглашения о связях по стандарту Application Binary Iinterface (ABI) для i386. Команды call, ret, адрес возврата, коды пролога и эпилога, бесконечная вложенность вызовов, локальные переменные, кадр стека. Пример использования библиотеки glibc.

5. Раздельная трансляция и внешние имена

Виды отладки, объектный файл, связь между раздельно оттранслированными объектными файлами, модулей, внешние и внутренние символьные имена, редактор связей `ld`. Применение соглашений `ABI` для взаимных вызовов функций на языках ассемблер и `C`.

NB. Сложность разделов нарастает в любом разделе используется материал из предыдущих.

Рекомендуется активно задавать вопросы, вести конспект, обращаться за консультациями.

Отчетность

ПМиИ, ИСиТ — зачет, ПрИн — экзамен.

Баллы, бонусы. Зачет/экзамен – без письменной работы.

Новость от 30.06.2026:

https://cs.petrsu.ru/news/2026/bosp/exam.php.ru

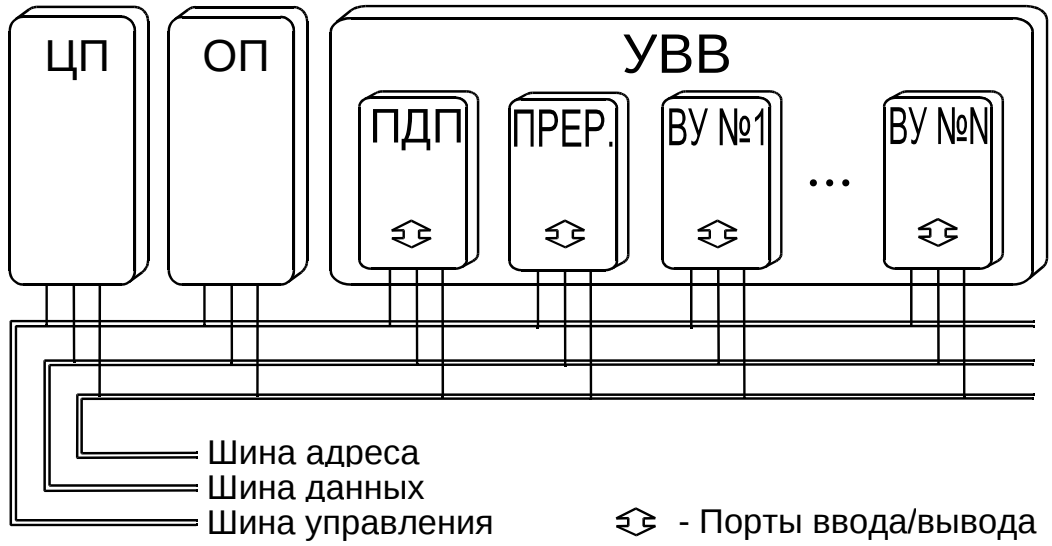
Учебник – оглавления, указатель

Богоявленский Ю. А., Дьяконов М. В., Печников А. А. Центральные процессоры персональных ЭВМ

https://edu.petrsu.ru/files/upload/2124_1427288068.pdf

Архитектура набора команд `IA-32`

Архитектура фон Неймана



Модель архитектура ЭВМ с общей шиной для разработчика ПО.

Джона фон Нейман — математик с мировым именем (при рождении Янош Лайош Нейман, при работе в Берлинском университете — Иоганн фон Нейман). Он является ведущим автором работы:

Burks A. W., Goldstine H. H., Neumann J. Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. — Institute for Advanced Study, Princeton, N. J., July 1946.

Русский перевод «Беркс А., Голдстейн Г., Нейман Дж. Предварительное рассмотрение логической конструкции электронного вычислительного устройства // Кибернетический сборник. М.: Мир, 1964. Вып. 9

В этой работе были сформулированы основные принципы логический структура ЭВМ, названные позже «архитектура фон Неймана» и реализованные в десятках ЭВМ по всему миру.

1. Двоичное кодирование информации
2. Неразличимость команд и данных
3. Адресуемость оперативной памяти
4. Последовательное выполнение команд

Оперативная память. Единицы доступа и адреса.

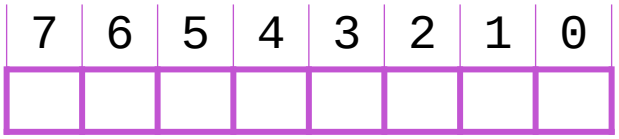
Единица доступа это блок ОП, который может быть прочитан/записан за одну команду ЦП. Байт – минимальная единица доступа. Байты ОП и их последовательные номера – **адреса** реализованы аппаратно.

Физический адрес — это аппаратный номер байта, физическое адресное пространство — вся совокупность байтов ОП.

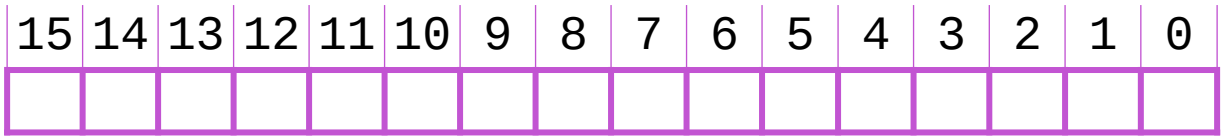
Байты, состоят из физически реализованных двоичных цифр **(Binary digit) – bit – 0/1**. Биты нумеруются СПРАВА НАЛЕВО в порядке степеней разрядов в представлении двоичного числа:

$$Q = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \ldots + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

Байт.



Слово.



Двойное слово.

Базовая среда выполнения IA-32

Материал этого раздела взят из руководства «Intel® 64 and IA-32 Architectures Software Developer’s Manual. Volume 1: Basic Architecture»:

https://kappa.cs.petrsu.ru/~chistyak/architecture/docs/64-ia-32-architectures-software-developer-vol-1-manual.pdf

IA-32 (сокращение от "Архитектура Intel, 32-разрядная"), называемая также i386 - это поддерживающая 32-разрядные вычисления архитектура x86, разработанная Intel и впервые реализованная в процессоре 80386 в 1985 году.

Аббревиатура "IA-32" используется как общее обозначения для всех версий x86, обеспечивающих такую поддержку. IA-32 полноценно поддерживается процессорами с архитектурой Intel®64, операционными системами и системами программирования.

Архитектура IA-32 поддерживает три основных режима работы:

I. Protected mode – защищенный режим с возможностью многозадачного выполнения ПО для процессора Intel 8086;
II. Real-address mode – реализует среду программирования процессора Intel 8086. Процессор работает в этом режиме после включения питания или нажатия кнопки Reset.
III. System management mode – режим управления системой (System management mode).

Режим работы определяет, какие команды и архитектурные функции доступны.

Любой программе или задаче, выполняющейся на процессоре IA-32, предоставляется набор ресурсов для выполнения машинных команд и для хранения кода, данных и информации о состоянии. Эти ресурсы составляют базовую среду выполнения (БСВ) для процессора IA-32.g

Процессоры Intel®64 поддерживают БСВ архитектуры IA-32 и аналогичную среду в режиме IA-32e, которая может выполнять 64-разрядные программы (64-разрядный подрежим) и 32-разрядные программы (подрежим совместимости).

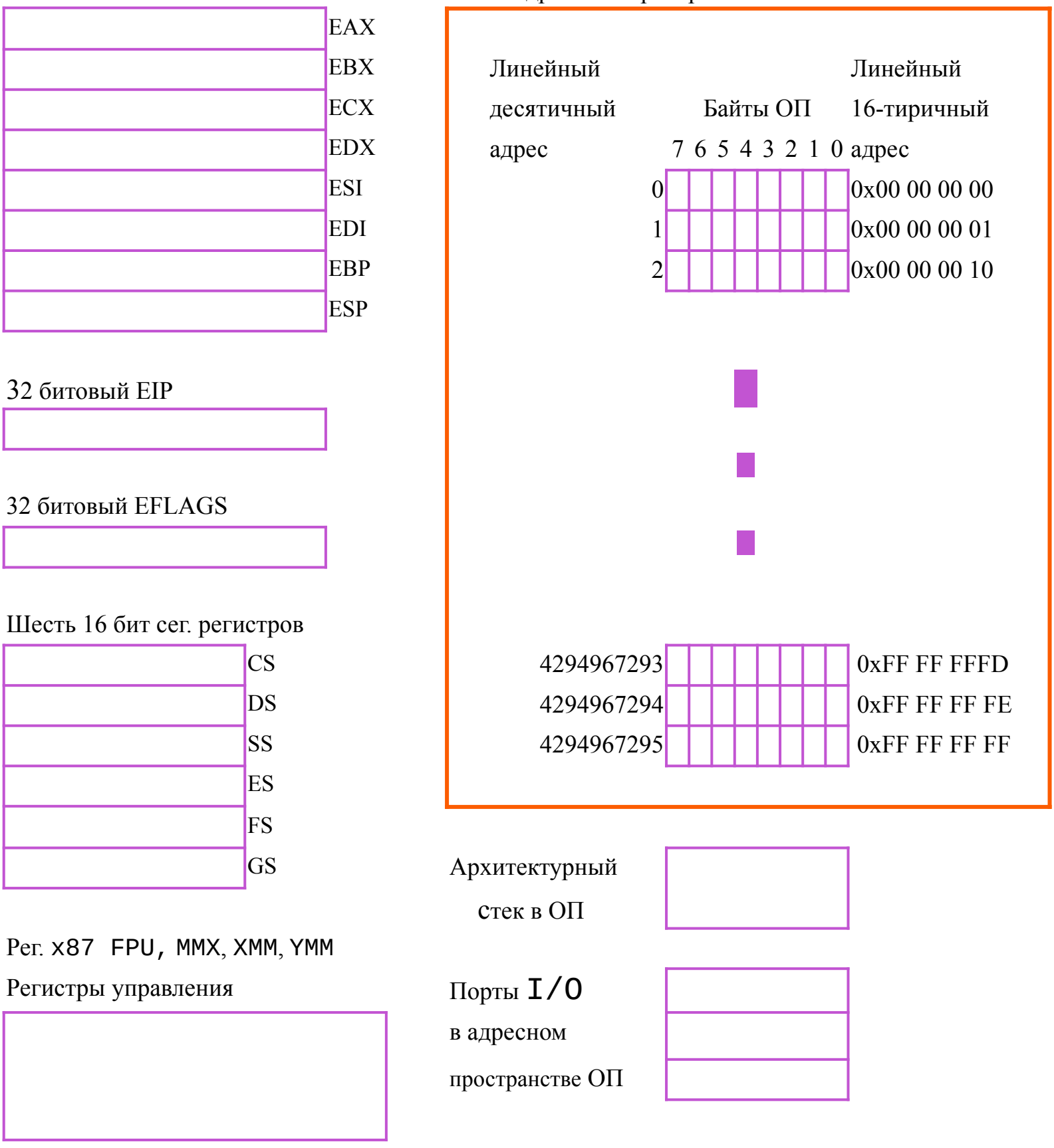


Рис. Базовая среда выполнения в архитектуре IA-32

БСВ используется прикладными программами и операционной системой и содержит следующие элементы.

I. Адресное пространство ОП

- Физическое адресное пространство объемом до 64 Гбайт (2^{36} байт).
  - Модели памяти IA-32
    - При использовании средства управления памятью процессора, программы напрямую не обращаются к физической памяти а используют одну из трех моделей памяти: плоскую (`Flat memory model`), сегментированную (`Segmented memory model`) или реального адреса (`Real-address mode memory model`), которая соответствует модели памяти процессора `8086`.
  - Мы будем работать только в модели плоской памяти — едином непрерывном т. н. линейном адресном пространстве объемом 4 Гб, содержащем код, данные и стеки. Адреса байтов называются линейными адресами и задаются в диапазоне 0 — (2^{32}-1). Т. о. мы НЕ РАССМАТРИВАЕМ использование сегментных регистров. NBNB Линейный адрес — 32 битовое целое число.

II. Базовые регистры выполнения программ

- восемь 32 битовых регистров общего назначения (РОН);
- шесть 16 битовых сегментных регистров;
- 32 битовые регистры EFLAGS и EIP (указатель команд).

составляют среду в которой выполняется команды общего назначения — целочисленная арифметика, управление порядком выполнения, операции с битовыми и байтовыми строками, адресация памяти. Эти регистры (кроме сегментных) буду рассмотрены на следующей лекции.

- В БСВ `IA-32` входят также не рассматриваемые нами регистры `x87 FPU` (операции над числами с плавающей точкой), регистры `MMX`, `XMM`, `YMM` для поддержки операций `SIMD` (`single-instruction, multiple-data`) с различными данными
- Различные регистры управления.

III. Архитектурный стек (ОС выделяет блок в ОП).

IV. Порты ввода/вывода (I/O) в адресном пространстве ОП.

NBNB. Центральный процессор (ЦП) распознает и выполняет команды ISA, хранящиеся в ОП в двоичной системе счисления выполняя основной цикл ЦП (изучить на стр. 29).

До начала 50-х годов программировали в двоичных кодах используя 8 или 16 представление двоичных команд и данных.

Введение в язык ассемблера

Assembly Language – символьное представление команд, адресов и данных.

NBNB. Assembly language may also be called symbolic machine code

Исходный, объектный, исполняемый файлы

Входной файл	Имя программы	Выходные файлы
p.S	Ассемблер as	p.o, p.lst (по ключам)
p.o	Редактор связей ld	pp

Структура программы. Секции

В исходном модуле можно создать три предопределенных и (для исполняемого файла ELF) произвольное число определяемых программистом секций, образы которых будут также сформированы в объектном и исполняемом файлах.

Секции содержат элементы программы определенного класса и могут иметь специфические свойства — например «только для чтения». Предопределены следующие секции:

- данных — содержит определения данных, следующие за директивами `.data;`
- команд процессора — содержит символьные представления команд процессора, следующие за директивами `.text;`
- неинициализированных данных — содержит описания размеров данных, следующие за директивами `.bss`. Это единственная секция, образ которой не формируется в объектном и исполняемом файлах а создается в момент запуска исполняемого файла загрузчиком ОС, что позволяет уменьшить объем внешней памяти, нужный для хранения объектных и исполняемых файлов.

Все строки исходного кода, следующие за каждым вхождением директив, задающих секции, считаются принадлежащими этой секции. Хотя бы одна секция должна присутствовать.

Символьное представление адресов оперативной памяти

В двоичных командах процессора необходимо задавать адреса операндов в ОП и адреса передачи управления. В модели ОП `ISA IA-32` это в общем случае 32b целое число, определенное в конкретной секции . В языке ассемблера эти адреса представляются символьными именами, в частности — метками — символьными именами, заканчивающихся символом `':'.`

При ассемблировании или редактировании связей выполняется распределение ОП для команд и данных для исполняемых и объектных файлов программы и символьным именам, в т.ч. меткам присваиваются значения 32b адресов ОП. Функции символьных имен аналогичны функциям как имен переменных так и меток в ЯВУ . Ниже рассмотрим примеры.

Операторы языка ассемблера gas в синтаксисе AT&T

Оператор это:

- директива языка ассемблера;
- или символьное представление двоичной команды процессора;
- или макрокоманда (будут рассмотрены позже).

Оператору может предшествовать метка.

Директивы языка ассемблера

Директивы — это разнообразные по назначению указания для программы ассемблера которые, например:

- определяют следующие за ними операторы, как принадлежащие одной из секций:
- переводят в двоичную систему числовые литералы;
- формируют двоичные коды строковых и односимвольных литералов;
- выделяют ОП для данных и размещают их двоичные коды в объектном файле.
- управляют файлом листинга.

NBNB. Литералы описаны в лекции 3.

Формат директивы:

```
<метка> .<имя директивы> пробел/таб <список параметров>
```

Метка может отсутствовать, имя директивы начинается с символа точка, параметры даются в описании директивы.

Примеры директив

- `.macro` — начать текст макроопределения;
- `.endm` — завершить текст макроопределения;

- .include "ФАЙЛ" вставка файла с исходным текстом в основной исходный файл после этой директивы;
- `.data` — следующие операторы принадлежат секции данных;
- `n: .long 2345` — определить в ОП 32b константу с меткой `n;`
- `ten: .word 10` — определить в ОП 16b константу с меткой `ten`;
- `.text` — следующие операторы принадлежат секции команд;
- `.global _start` — определить внешнее символьное имя точки входа в исполняемый файл для загрузчика ОС;
- `.end` — последняя строка исходного текста.

См. список директив по адресу:
https://kappa.cs.petrsu.ru/~chistyak/architecture/docs/gas/gas-7.html

Символьное представление двоичных команд процессора

Команда процессора при ассемблировании транслируется в двоичное представление, размещается в объектном файле, а после редактирования связей — в исполняемом.

После запуска файла ЦП выполняет соответствующие двоичным командам операции над операндами в ОП, регистрах или в самой команде (непосредственный операнд — immediate), или операции переходов и управления процессором.

Символьное представление двоичной команды в синтаксисе `AT&T` может иметь три формы:

- метка коп`{b|w|l}` операнд1 (источник), операнд2 (приемник — результат);
- метка коп`{b|w|l}` операнд (и источник и приемник)
- метка коа — операндов нет

Где один из суффиксов кода операции `{b|w|l}` указывает, что длина операндов 8 `(b)`, 16 `(w)` или 32 `(l)` битов.

Здесь метка нужна для передачи управления команде и может отсутствовать.

Опишем место нахождения значений операндов и способ его указания в команде:

- R — в регистре ЦП, символьное имя регистра;
- I — в теле команды (непосредственный операнд), литерал или метка (символьное имя) в ОП, которым предшествует символ «$»:
- P — в порту ввода/вывода;
- M — в ОП, метка (символьное имя) данного или метка команды для перехода в ОП, имеющая значение адреса ОП. Нужно различать три случая:
    - M - операнд источник. ЦП читает по шине значение по адресу ОП, затем выполняет операцию;
    - M — операнд приемник. ЦП выполняет операцию, затем записывает по шине результат по адресу ОП;
    - M — операнд перехода. ЦП читает команду по адресу ОП и передает ей управление.

Важно отличать два случая:

- метке НЕ предшествует символ «$» — формируется операнд M, операция выполняется над данным по адресу в ОП, соответствующему метке;
- метке предшествует символ «$» — формируется операнд I, адрес в ОП, соответствующий метке, помещается в двоичный образ команды и операция выполняется над этим адресом.

Для случая двух операндов их допустимые комбинации даны в таблице:

	операнд2 (приемник — результат)		
операнд1 (источник)	R	I	M
R	Да	Нет	Да
I	Да	Нет	Да
M	Да	Нет	Нет

Отметим, что команды работы с портами вводв/вывода имеют один операнда — адрес порта, второй операнд находится в фиксированном регистре.

Любое данное в ОП задается парой:

Адрес ОП:Значение

Выполнение операций как над данными так и над их адресами в ОП как на уровне команд процессора так и на уровне выражений языка ассемблера в отличие от ЯВУ реализовано естественным и равноправным образом. В языке Си, например, адресам ОП соответствуют указатели.

NBNB. Команды и операнды (стр. 74 книги) заданы в синтаксис Intel: коп операнд1 (приемник — результат), операнд2 (источник), в синтаксисе AT&T – наоборот, не так, как в книге.

Примечание.

Команда умножения imul может иметь три операнда, ее представление в ассемблере дано в описании синтаксиса AT&T.

Синтаксисы AT&T и Intel — основные отличия.

Синтаксис AT&T используется по умолчанию. Директива .intel_syntax переводит as в этот синтаксис, а директива .att_syntax – обратно в AT&T:

- Перед непосредственным операндом нужен символ "$".
- Перед операндами регистра нужен символ "%'.
- Порядок следования операндов источника и приемника противоположен порядку принятому в Intel.
- Код операции должен иметь суффикс указывающий размер обоих (как правило) операндов:
  - b байт, 8 битов;
  - w слово, 16 битов;

- l двойное слово, 32 бита.

Другие отличия синтаксисов представлены в руководстве «Using as. The gnu Assembler (GNU BinutilsVersion 2.45)» и на стр. дисциплины:

http://kappa.cs.petrsu.ru/~chistyak/architecture/docs/gas/gas-8.html#ss8.1

https://cs.petrsu.ru/~chistyak/architecture/docs/as.2.45.pdf

Примеры команд

1. movl $0, %ebx — переслать значение 0 длиной 32b (непосредственный операнд) в регистр %ebx;
2. movl n, %eax — переслать ЗНАЧЕНИЕ длиной 32b из области ОП, распределенной для метки (символьного имени) n в регистр %eax;
3. movl $n, %eax — переслать АДРЕС длиной 32b значения из области ОП (непосредственный операнд), распределенной для символьного имени n в регистр %eax;
4. movl %ebx, length — переслать значение из регистра %ebx диной 32b в область ОП, распределенную для символьного имени length;
5. movw %bx, %dx — переслать значение из регистра %bx диной 16b в регистр %dx;
6. movb %bh, %cl — переслать значение из старшего байта регистра %ebx длиной 8b в младший байт регистра %ecx;
7. incl %ebx — увеличить на 1 значение в регистре;
8. sti — присвоить значение 1 флагу прерываний.

Задача первой лабораторной работы

Исходный файл `task1.c`

```
/*
 * Вычисление количества позиций символов для
 * символьного представления в десятичной системе счисления
 * неотрицательного 32-битового двоичного целого числа
 *
 * Программа содержательно эквивалентна программе task1.S
 *
 * Компиляция: gcc -m32 -ggdb -o task1-exe-c task1.c
 *
 * -m32 генерировать 32 разрядные машинные команды

 * -ggdb - включить в исполняемый файл отладочную информацию

 * -o task1-exe-c - задание имени выходного (исполняемого)
файла

 * task1.c - исходный - входной файл (этот файл)
 *
 * Запуск отладчика: kdbg task1-exe-c
 *
 * Получение asm текста, сгенерированного gcc: gcc -m32 -S -o
task1-c.S task1.c
 *
 * -S - генерировать asm текст, не генерировать объектный и
исп. файлы

 * -o task1-c.S - выходной файл программы на языке ассемблера
 *            в которую gcc транслирует файл task1.c

 * task1.c - исходный - входной файл (этот файл)
*/

int main()
{
                    /*  соотв. конструкция ассемблера в 1.S
и коммент. */
    int n = 2345;    /* n:      .long 2345  число */
    int length = 0;  /* length: .long 0     кол-во симв.
позиций */
```

```
/*  Эти данные РАЗМЕЩАЕМ во встроенных в процессор
    целых 32 разрядных переменных - регистрах с
фиксированными
    именами. ПОЭТОМУ ОПИСЫВАТЬ ИХ НЕ НАДО. Нет аналога в asm
                            */
    int counter;   /* рег. %ebx - счетчик делений */
    int quotient;  /* будет в рег. %eax после деления -
частное */
    int remainder; /* будет в рег. %edx после деления -
остаток */

    quotient = n;  /* movl n, %eax  */
    counter = 0;   /* movl $0, %ebx */

nextdigit:
    remainder = quotient % 10; /* этот и следующий оператор
                                  /* выполняются командой idivl
ten
*/
    quotient = quotient / 10;

    ++counter;                /* incl %ebx */

    if (quotient)
        goto nextdigit;

/* проверка условия в ISA ВСЕГДА выполняется парой команд:

cmp{b|w|l} <операнд1>, <операнд2> - вып. вычитание
<операнд2>-<операнд1>
           - отражает знак и др. свойства результата
                          в битах регистра флагов.

 jcc <операнд> - j[ump on] c[ondition] c[ode] проверяет
    условие в битах регистра флагов и по результату проверки
    выполняет/не выполняет переход  по адресу <операнд>.

 В нашем случае команда jg   nextdigit { jg - jump if greater
- перейти если строго  больше} выполняет переход на метку
nextdigit: если предыдущая команда cmpl $0, %eax установила в
регистре флагов биты, указывающие, что значение в  регистре
%eax,
т.е. частное, СТРОГО  БОЛЬШЕ нуля.
*/
    length = counter; /* movl %ebx, length */
}
```

Исходный файл `task1.S`

```
  /*
   * Вычисление количества позиций символов для
   * символьного представления в десятичной системе счисления
   * неотрицательного 32-битового двоичного целого числа

   * Программа содержательно эквивалентна программе task1.c

   * Ассемблирование:  as -ahlsm=task1.lst --32 -gstabs+ -o
task1.o task1.S
   *
   * -ahlsm ключи полного листинга
   * task1.lst - имя ТЕКСТОВОГО файла листинга, описывающего
   * результат ассемблирования

   * --32 - генерировать 32 разрядные машинные команды

   * -gstabs+ - включать отладочную информацию формата stabs

   * -o task1.o - задание имени выходного ( для команды as -
   * объектного) файла

   * task1.S -  исходный -  входной файл (этот файл)
   *
   * Редактирование связей: ld -melf_i386 -o task1-exe-S
task1.o

   * МОЖНО НЕ ЧИТАТЬ! В нашем курсе используется только
   *                -melf_i386
   *  -m<эмуляция ld> ld может генерировать машинный код
   *     исполняемого файла для нескольких архитектур.
   *     Говорят, что ld "эмулирует" архитектуру.
   *  Поддерживаемые архитектуры - "эмуляции":

   *    elf_x86_64
   *    elf_i386
   *    i386linux
   *    elf_l1om

   * -o task1-exe-S - задание имени выходного
   * (для команды ld -  исполняемого) файла

   * task1.o - объектный файл (входной для редактора связей
ld)
```

```
   * Запуск отладчика: kdbg task1-exe-S
   *
*/

.include "my-macro"  # подключение файла с макроопределениями


.data     # секция данных, распределение памяти
          # соотв. конструкция языка С и коммент.
n:      .long 2345        # int n = 2345; число
length: .long 0           # int length =0; результат
ten:    .long 10          # определяем константу ЯВНО
                          # нет аналога в С

.text # секция команд процессора

.global _start     # точка входа - глобальная метка

_start:
    nop             # пустая операция - no operation

#   нужна, чтобы задать отладчику контр. точку останова на
следующей, первой
#   содержательной команде программы

    movl $0, %ebx   # counter = 0; счетчик делений
    movl n, %eax    # готовим к команде деления idivl
                    # нет аналога в С
nextdigit:
        movl  $0, %edx          # еще готовим, уже в цикле
                                # нет аналога в С

#   Смысл этой подготовки.
#   Примем, что делим 32 битовое число, размещенное в EAX.
#   Т.к. команда idivl интерпретирует значения в паре
регистров EDX:EAX
#   как единое 64 битовое делимое, то перед делением EDX
должен иметь значение
#   0.  После выполнения эта команда
#   помещает в EDX - остаток, в EAX - частное, что
#   НАРУШАЕТ ИНТЕРПРЕТАЦИЮ значений пары EDX:EAX. Перед
следующим делением эту
#   интерпретацию надо восстановить, присвоив EDX значение 0.
```

```
        idivl ten         # делим объединенные регистры
edx:eax на 10
                          # частное в  eax, остаток в edx

        incl %ebx         # ++counter; счетчик делений + 1

#    две следующие команды соответствуют условному оператору
if (quotient) goto
#    nextdigit;

        cmpl $0, %eax     # частное > 0 ?
        jg   nextdigit    # ДА, продолжаем
                          # НЕТ, на след. команду

/*

проверка условия в ISA ВСЕГДА выполняется парой команд:

   cmp{b|w|l} <операнд1>, <операнд2> - вып. вычитание
<операнд2>-<операнд1>
                           - отражает знак и др. свойства
результата
                             в битах регистра флагов.

   jcc <операнд> - j[ump on] c[ondition] c[ode] проверяет
          условие в битах регистра флагов и по результату
проверки
          выполняет/не выполняет переход  по адресу
<операнд>.

 В нашем случае команда jg   nextdigit { jg - jump if greater
- перейти если строго  больше} выполняет переход на метку
nextdigit: если предыдущая команда cmpl $0, %eax установила в
регистре флагов биты, указывающие, что значение в регистре
%eax, т.е. частное, СТРОГО  БОЛЬШЕ нуля.

*/

        movl %ebx, length  # length = counter;

    Finish                 # конец работы,
                           # возврат в ОС
                           # (макро из файла my-macro)

.end   # последняя строка исходного текста
       # as прекращает чтение файла исходного текста
```

Конец исходного файла.

Файл `task1.lst` — шестнадцатиричное представление двоичного содержания секций объектного файла, полученного при ассемблировании файла `task1.S`.

```
GAS LISTING task1.S                     page 1


   1     /*
   2        * Вычисление количества позиций символов для
   3        * символьного представления неотрицательного
   4        * 32-битового целого числа
   5
   6        * Программа содержательно эквивалентна программе task1.c
   7
   8        * Ассемблирование:  as -ahlsm=task1.lst --32 -gstabs+ -o
task1.o task1.S
   9        *
  10        * -ahlsm ключи полного листинга
  11        * task1.lst - имя ТЕКСТОВОГО файла листинга, описывающего
  12        * результат ассемблирования
  13
  14        * --32 - генерировать 32 разрядные машинные команды
  15
  16        * -gstabs+ - ключи генерации отладочной информации для от
              отладчиков
  17
  18        * -o task1.o - задание имени выходного ( для команды as -
  19        * объектного) файла
  20
  21        * -o ключ определения имени выходного ( для команды as -
  22        * объектного) файла
  23
  24        * task1.S -  исходный -  входной файл (этот файл)
  25        *
  26        * Редактирование связей: ld -melf_i386 -o task1-exe-S task1.o
  27
  28        * МОЖНО НЕ ЧИТАТЬ! В нашем курсе используется только
  29        *    -melf_i386
  30        *  -m<эмуляция ld> ld может генерировать машинный код
  31        *     исполняемого файла для нескольких архитектур.
  32        *     Говорят, что ld "эмулирует" архитектуру.
  33        *  Поддерживаемые архитектуры - "эмуляции":
  34
  35        *    elf_x86_64
  36        *    elf_i386
  37        *    i386linux
  38        *    elf_l1om
  39
  40        * -o task1-exe-S - задание имени выходного
```

```
  41        * (для команды ld -  исполняемого) файла
  42
  43        * -o ключ определения имени выходного
  44        * (для команды ld -  исполняемого) файла
  45
  46        * task1.o - объектный файл (входной для редактора связей ld
  47
  48        * Запуск отладчика: kdbg task1-exe-S
  49        *
  50     */
  51
;  52     .include "my-macro"  # подключение файла с макроопределениями
   1     /* Макроопределение завершения работы */
   2
   3     .macro Finish
   4         movl $0, %ebx # first argument: exit code
   5         movl $1, %eax # sys_exit index

GAS LISTING task1.S             page 2


   6         int  $0x80         # kernel interrupt
   7     .endm
   8
  53
  54
  55     .data     # секция данных, распределение памяти
  56               # соотв. конструкция языка C и коммент.
  57 0000 29090000     n:      .long 2345       # int n = 2345; число
  58 0004 00000000     length: .long 0          # int length =0; результат
  59 0008 0A000000     ten:    .long 10         # определяем константу ЯВНО
  60                                             # нет аналога в C
  61
  62     .text # секция команд процесора
  63
  64     .global _start     # точка входа - глобальная метка
  65
  66     _start:
  67 0000 90               nop # пустая операция - no operation
  68
  69     #   нужна, чтобы задать отладчику контр. точку останова
  70     #   содержательной команде программы
  71
  72 0001 BB000000         movl $0, %ebx   # counter = 0; счетчик делений
  72      00
  73 0006 A1000000         movl n, %eax    # готовим к команде деления
idivl
  73      00
  74               # нет аналога в C
  75     nextdigit:
  76 000b BA000000             movl  $0, %edx       # еще готовим, уже в
цикле
  76      00
  77               # нет аналога в C
  78
  79     #   Смысл этой подготовки.
  80     #   Примем, что делим 32 битовое число, размещенное в EAX.
```

```
  81      #   Т.к. команда idivl интерпретирует значения в паре регистров-
слов
  82      #   EDX:EAX как единое 64 битовое делимое, то перед делением EDX
  83      #   должно быть равно 0.  После выполнения деления эта команда
  84      #   помещает в EDX - остаток, в EAX - частное, что
  85      #   НАРУШАЕТ ИНТЕРПРЕТАЦИЮ значений пары EDX:EAX. Перед следующим
  86      #   деление мэту интерпретацию надо восстановить, присвоив EDX 0
  87
  88 0010 F73D0800        idivl ten  # делим объединенные регистры edx:eax
на 10
  88      0000
  89                  # частное в  eax, остаток в edx
  90
  91 0016 43              incl %ebx        # ++counter; счетчик делений + 1
  92
  93      #    две следующие команды соответствуют условному оператору
  94      #    if (quotient) goto nextdigit;
  95
  96 0017 83F800                cmp $0, %eax    # частное > 0 ?
  97 001a 7FEF                  jg   nextdigit   # ДА, продолжаем
  98
  99
 100      /*
 101
 102      проверка условия командами ISA ВСЕГДА выполняется парой команд

GAS LISTING task1.S             page 3


 103
 104        cmp{b|w|l} <операнд1>, <операнд2> - вып. вычитание
 105       - <операнд2>-<операнд1> отражает знак и др. свойства результата
 106         в битах регистра флагов.
 107
 108        jcc <операнд> - j[ump on] c[ondition] c[ode] проверяет
 109        условие в битах регистра флагов и по результату проверки
 110        выполняет/не выполняет переход  по адресу <операнд>
 111
 112       В нашем случае команда jg   nextdigit { jg - jump if greater
 113      - перейти если строго  больше} выполняет переход на метку
 114      nextdigit: если предыдущая команда cmpl $0, %eax установила в
 115      регистре флагов биты, указывающие, что значение в регистре
 116      т.е. частное, СТРОГО  БОЛЬШЕ нуля.
 117
 118      */
 119
 120 001c 891D0400              movl %ebx, length  # length = counter;
                                 НЕТ,сохраняем результат
 120      0000
 121
 122          Finish     # конец работы,
 122 0022 BB000000     >  movl $0,%ebx
 122      00
 122 0027 B8010000     >  movl $1,%eax
 122      00
 122 002c CD80         >  int $0x80
 123        # возврат в ОС
```

```
 124         # (макро из файла my-macro)
 125
 126     .end   # последняя строка исходного текста

GAS LISTING task1.S                page 4


DEFINED SYMBOLS
 task1.S:57     .data:0000000000000000 n
 task1.S:58     .data:0000000000000004 length
 task1.S:59     .data:0000000000000008 ten
 task1.S:66     .text:0000000000000000 _start
 task1.S:75     .text:000000000000000b nextdigit

NO UNDEFINED SYMBOLS
```
Конец файла листинга

Иллюстрация работы команд `idivl` (описание ее на стр. 101).

Делимое — `64`-разрядное число в паре объединенных регистров `EDX:EAX`

До первого выполнения `idivl`:

EDX	EAX
0	2345

После выполнения `idivl`:

EDX	EAX
5	234
остаток	частное

NBNB. Записывая остаток в регистр EDX команда ИСКАЖАЕТ объединенное делимое.

Если перед следующим делением не присвоить регистру `EDX` значение `0`, то при следующем делении в цикле делимое будет иметь значение `5234` и получится остаток `4`, а частное `523`.

Если же присвоить регистру `EDX` значение `0`, то делимое будет иметь значение `234`, остаток — `4`, частное `23`, что нам и нужно.

Выводы

- Профессионалу в сфере IT необходимо владеть архитектурной культурой.
- Материалы разделов дисциплины опираются на предшествующие разделы, не следует откладывать их изучение.
- Изучается `ISA IA-32`, на третьем курсе изучается `ISA Intel® 64`.
- Базовая средв выполнения (БСВ) `ISA IA-32` предоставляет программе ресурсы: регистры, оперативную память, архитектурный стек, порты ввода/вывода.
- Плоская модель ОП `IA-32` имеет размер 4Гб, для задания адресов — номеров байтов в ней требуется 32b целое число
- Исходный файл `p.S` ассемблером `as` ассемблериуется в объектный файл `p.o` из которого редактор связей `ld` получает исполняемый файл `p`. Файлы `p.o` и `p` в нашей среде имеют формат `ELF`.
- Программа может иметь три предопределенные секции, задаваемые директивами: `.data, .text, .bss`.
- 32b адреса оперативной памяти представляются в языке ассемблера метками — символьными именами.
- Оператор языка ассемблера это директива языка ассемблера или команда процессора. Оператору может предшествовать метка.
- Два операнда в ОП задать нельзя.
- Операнд команды, заданный как `<метка>` определяет адрес операнда в ОП по адресу <метка>, а заданный как `$<метка>` определяет сам этот адрес как операнд, который размещается в команде.
- Проверка условия в `ISA` ВСЕГДА выполняется парой команд `cmp` и `jcc`.