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/

Лекция 1. Введение в дисциплину

План лекции

- Мотивация
- Как организована дисциплина
- Изучаемая архитектура и инструменты
- Разделы дисциплины
    1. Вводная часть
    2. Простые программы
    3. Режимы адресация и архитектурный стек
    4. Функции
    5. Раздельная трансляция и внешние имена
- Учебник – оглавления, указатель
- Системная среда центрального процессора
- Оперативная память. Единицы доступа и адреса.
- Архитектура `IA-32`. Базовая среда выполнения
- Введение в язык ассемблера

Знания и умения - “Операционные системы”, “Системное программирование”, “Компьютерные сети”, встраиваемые системы, роботы, Интернет вещей.

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, которые обновлялись в 2001, 2008, 2013, 2020 годах. В этом процессе количество предметных областей выросло до восемнадцати, однако во всех этих документах предметная область «Архитектура» сохранила свою фундаментальную роль в учебных планах.

С практической точки зрения происходит существенное расширение внедрения устройств с ограниченными вычислительными ресурсами (встроенные процессоры, одноплатные ЭВМ, интеллектуальные датчики и т. п.), что требует освоения

студентами навыков разработки на уровне архитектуры, формирования у них культуры архитектурного мышления.

В ИМИТ эта культура формируется дисциплиной "Введение в архитектуру ЭВМ", которая читается как базовая на первом или втором курсе. Дисциплина "Архитектура современных ЭВМ" — читается на третьем курсе, по выбору, опирается на знания полученные ранее, и расширяет навыки разработки на уровне архитектуры за счет углубленного изучение деталей.

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

Как организована дисциплина.

Богоявленский, 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/doc/it2018.pdf

Архитектура и инструменты:

- среда - `OpenSuSe Linux`;
- архитектура - только `IA-32`;
- задачи для процессоров реальных ПЭВМ;

- язык семейства индустриальных ассемблеров gas (man as, см. TARGET);
- 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.
- демонстрации — отладчик kdbg.

Разделы дисциплины.

1. Вводная часть.

   Системная среда центрального процессора, регистры, оперативная память. Язык ассемблера, представление на нем команд, характеризация операндов. Пример программы. Системы счисления, порядки байтов Big/Little endian, машинная арифметика, дополнительный код, Unicode.
2. Простые программы.

   Символьные имена, 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.

Страница курса: http://kappa.cs.petrsu.ru/~chistyak/architecture/

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

2020 г. новость от 09.06.2020 на

https://cs.petrsu.ru/news/archive.php.ru?q=news2020.xml

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

Системная среда центрального процессора, стр. 25

Архитектура фон Неймана

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

Оперативная память. Единицы доступа и адреса.

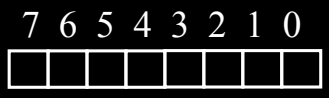
Единица доступа это блок ОП, который может быть обработан за одну команду ЦП чтения или записи. Байт — минимальная единица доступа. Байты ОП и их последовательные номера реализованы аппаратно.

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. Суффикс кода операции – b, пример - movb



Слово. Суффикс кода операции – w, пример - addw

Двойное слово. Суффикс кода операции – l, пример - cmpl

Далее в лекциях мы рассмотрим как задаются адреса слов и двойных слов.

Архитектура IA-32. Базовая среда выполнения

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

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.

Термин "IA-32" также используется как общее обозначения для всех версий x86, обеспечивающих такую поддержку. IA-32 по прежнему 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 — режим реального адреса , реализующий 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.

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).

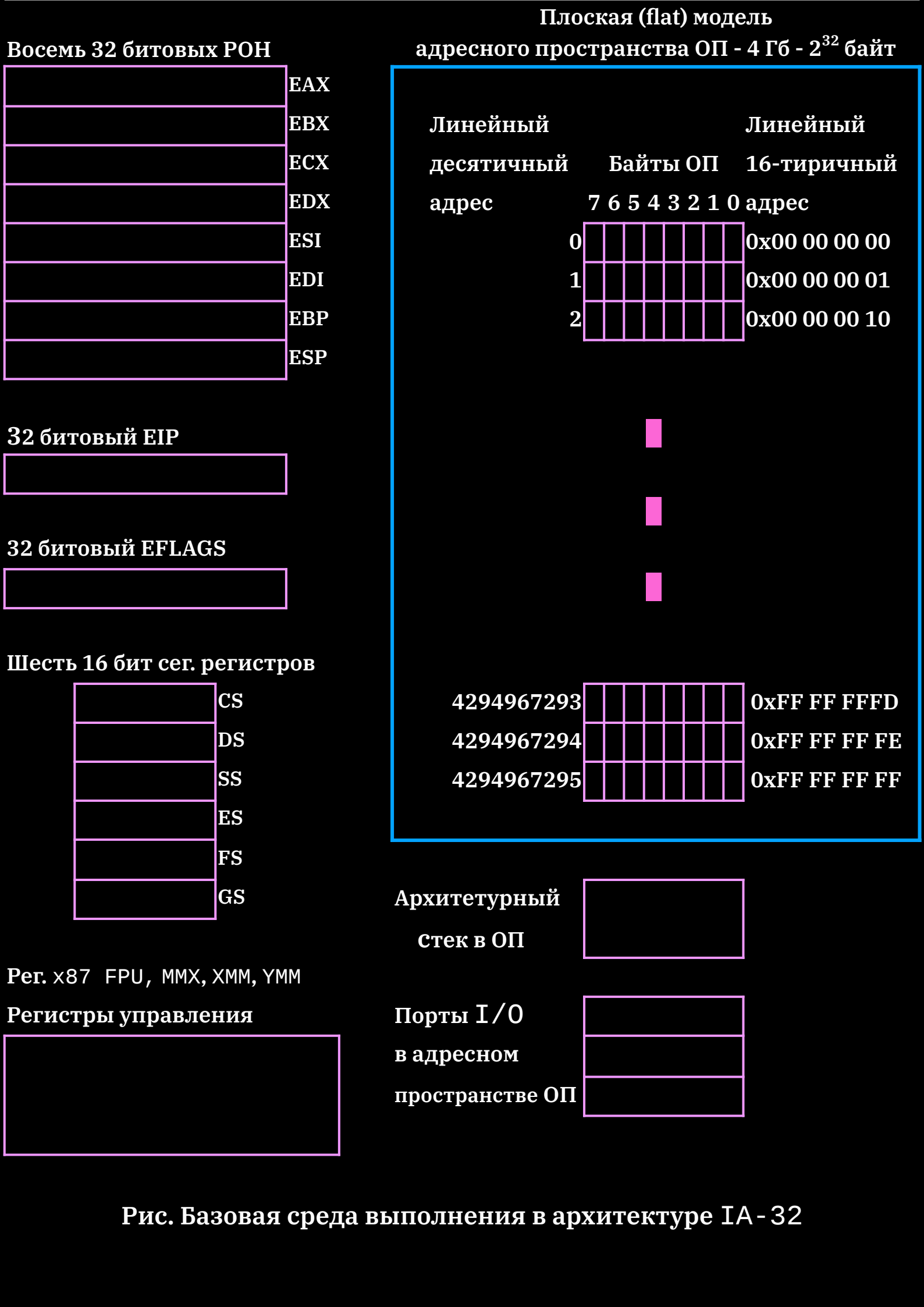


Рис. Базовая среда выполнения в архитектуре IA-32

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)

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.

ЯКЦП – набор двоичных команд, основной цикл ЦП, 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.

Ассемблер – транслятор с языка ассемблера на ЯКЦП.

```
              as         ld
Схема – p.S ---> p.o  ---> p
                      ---> p.lst
```

Команда на языке ассемблера состоит из:

- символьных кодов операций (определены разработчиками систем команд);
- регистровых операндов (выбираются программистом из списка регистров, определенного разработчиками архитектуры);

- непосредственных (заданных в тексте команды) операндов — констант или выражений (определяются программистом);
- символьных имен операндов в ОП (определяются программистом).

Задание команд в языке ассемблера для синтаксиса AT&T.
http://kappa.cs.petrsu.ru/~chistyak/architecture/docs/gas/gas-8.html#ss8.1

Команда на языке ассемблера в синтаксисе AT&T может иметь три формы:

- коп{b|w|l} операнд1 (источник), операнд2 (приемник — результат операции);
- коп{b|w|l} операнд;
- коп.

Операнд можно задать как:

- символьное имя адреса памяти;
  - размер обоих операндов должен соответствовать (если иное явно не оговорено) суффиксу имени операции:
    - 'b' — байт (8-бит);
    - `w' — , слово (16-бит);
    - `l' — двойное слово (32-бит для нашей архитектуры).
- имя регистра с обязательным префиксом — символом `%';
- непосредственный операнд в команде (например константа) с обязательным префиксом — символом `$'.

NBNB. Не определены команды, оба операнда которых являются символьными именами адресов памяти.

Примеры (часть из программы task1.S)

- movl $0, %ebx — переслать значение 0 длиной 32 бита (непосредственный операнд) в регистр ebx;
- movl n, %eax — переслать значение из области ОП, распределенной для символьного имени n в регистр eax;
- movl %ebx, length — переслать значение из регистра ebx диной 32 бита в область ОП, распределенную для символьного имени length;
- movw %bx, %dx — переслать значение из регистра bx диной 16 бит в регистр dx;
- movb %bh, %cl — переслать значение из старшего байта регистра ebx длиной 8 бит в младший байт регистра ecx;
- incl %ebx — увеличить на 1 значение в 32 битовом регистре;
- sti — присвоить значение 1 флагу прерываний.

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.

Видно, что команда обычно имеет 0, 1 или 2 операнда. Команда умножения imul может иметь три операнда. При этом в синтаксисе AT&T ее нужно задавать в виде:

imul i,r1,r2 — r2 = i*r1 или imul i,m,r — r = i*m,

где i — непосредственный операнд, r, r1,r2 — имена регистров, m — символьное имя операнда в памяти.

Рассмотрим исходный файл первой задачи 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)
*/

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 на ЯКЦП 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 */
}
```

Рассмотрим исходный файл первой задачи 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 процесора

.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


/*

проверка условия на ЯКЦП 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; ДА,сохраняем
результат

    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 16 представление результатов ассемблирования, т.е. что получилось на ЯКЦП из 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     #   делениемэту 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 на ЯКЦП 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;
ДА,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).

Объединенное делимое `EDX:EAX`

До первого выполнения `idivl`

EDX	EAX
0	2345

После первого выполнения `idivl`

EDX	EAX
5	234

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.