**Архитектура**

**x86 или IA-32?**

Вы, вероятно, уже слышали такое понятие, как «архитектура x86». Вообще оно довольно размыто, и вот почему. Само название x86 или 80x86 происходит от принципа, по которому Intel давала названия своим процессорам:

* Intel 8086 — 16 битов;
* Intel 80186 — 16 битов;
* Intel 80286 — 16 битов;
* Intel 80386 — 32 бита;
* Intel 80486 — 32 бита.

Этот список можно продолжить. Принцип наименования, где каждому поколению процессоров давалось имя, заканчивающееся на 86, создал термин «x86». Но, если посмотреть внимательнее, можно увидеть, что «процессором x86» можно назвать и древний 16-битный 8086, и новый Core i7. Поэтому 32-битные расширения были названы архитектурой IA-32 (сокращение от Intel Architecture, 32-bit). Конечно же, возможность запуска 16-битных программ осталась, и она успешно (и не очень) используется в 32-битных версиях Windows. Мы будем рассматривать только 32-битный режим.

**Регистры**

Регистр — это небольшой объём очень быстрой памяти, размещённой в процессоре. Он предназначен для хранения результатов промежуточных вычислений, а также некоторой информации для управления работой процессора. Так как регистры размещены непосредственно в процессоре, доступ к данным, хранящимся в них, намного быстрее доступа к данным в оперативной памяти.

Все регистры можно разделить на две группы: пользовательские и системные. Пользовательские регистры используются при написании «обычных» программ. В их число входят *основные программные регистры* (англ. basic program execution registers; все они перечислены ниже), а также регистры математического сопроцессора, регистры MMX, XMM (SSE, SSE2, SSE3). Системные регистры (регистры управления, регистры управления памятью, регистры отладки, машинно-специфичные регистры MSR и другие) здесь не рассматриваются. Более подробно см. [1].

**Регистры общего назначения** (РОН, англ. General Purpose Registers, сокращённо GPR). Размер — 32 бита.

* %eax: Accumulator register — аккумулятор, применяется для хранения результатов промежуточных вычислений.
* %ebx: Base register — базовый регистр, применяется для хранения адреса (указателя) на некоторый объект в памяти.
* %ecx: Counter register — счетчик, его неявно используют некоторые команды для организации циклов (см. loop).
* %edx: Data register — регистр данных, используется для хранения результатов промежуточных вычислений и ввода-вывода.
* %esp: Stack pointer register — указатель стека. Содержит адрес вершины стека.
* %ebp: Base pointer register — указатель базы кадра стека (англ. stack frame). Предназначен для организации произвольного доступа к данным внутри стека.
* %esi: Source index register — индекс источника, в цепочечных операциях содержит указатель на текущий элемент-источник.
* %edi: Destination index register — индекс приёмника, в цепочечных операциях содержит указатель на текущий элемент-приёмник.

Эти регистры можно использовать «по частям». Например, к младшим 16 битам регистра %eax можно обратиться как %ax. А %ax, в свою очередь, содержит две однобайтовых половинки, которые могут использоваться как самостоятельные регистры: старший %ah и младший %al. Аналогично можно обращаться к %ebx/%bx/%bh/%bl, %ecx/%cx/%ch/%cl, %edx/%dx/%dh/%dl, %esi/%si, %edi/%di.

Не следует бояться такого жёсткого закрепления назначения использования регистров. Большая их часть может использоваться для хранения совершенно произвольных данных. Единственный случай, когда нужно учитывать, в какой регистр помещать данные — использование неявно обращающихся к регистрам команд. Такое поведение всегда чётко документировано.

**Сегментные регистры:**

* %cs: Code segment — описывает текущий сегмент кода.
* %ds: Data segment — описывает текущий сегмент данных.
* %ss: Stack segment — описывает текущий сегмент стека.
* %es: Extra segment — дополнительный сегмент, используется неявно в строковых командах как сегмент-получатель.
* %fs: F segment — дополнительный сегментный регистр без специального назначения.
* %gs: G segment — дополнительный сегментный регистр без специального назначения.

В ОС Linux используется плоская модель памяти (flat memory model), в которой все сегменты описаны как использующие всё адресное пространство процессора и, как правило, явно не используются, а все адреса представлены в виде 32-битных смещений. В большинстве случаев программисту можно даже и не задумываться об их существовании, однако операционная система предоставляет специальные средства (системный вызов modify\_ldt()), позволяющие описывать нестандартные сегменты и работать с ними. Однако такая потребность возникает редко, поэтому тут подробно не рассматривается.

**Регистр флагов eflags и его младшие 16 битов, регистр flags.** Содержит информацию о состоянии выполнения программы, о самом микропроцессоре, а также информацию, управляющую работой некоторых команд. Регистр флагов нужно рассматривать как массив битов, за каждым из которых закреплено определённое значение. Регистр флагов напрямую не доступен пользовательским программам; изменение некоторых битов eflags требует привилегий. Ниже перечислены наиболее важные флаги.

* cf: carry flag, флаг переноса:
  + 1 — во время арифметической операции был произведён перенос из старшего бита результата;
  + 0 — переноса не было;
* zf: zero flag, флаг нуля:
  + 1 — результат последней операции нулевой;
  + 0 — результат последней операции ненулевой;
* of: overflow flag, флаг переполнения:
  + 1 — во время арифметической операции произошёл перенос в/из старшего (знакового) бита результата;
  + 0 — переноса не было;
* df: direction flag, флаг направления. Указывает направление просмотра в строковых операциях:
  + 1 — направление «назад», от старших адресов к младшим;
  + 0 — направление «вперёд», от младших адресов к старшим.

Есть команды, которые устанавливают флаги согласно результатам своей работы: в основном это команды, которые что-то вычисляют или сравнивают. Есть команды, которые читают флаги и на основании флагов принимают решения. Есть команды, логика выполнения которых зависит от состояния флагов. В общем, через флаги между командами неявно передаётся дополнительная информация, которая не записывается непосредственно в результат вычислений.

**Указатель команды eip (instruction pointer).** Размер — 32 бита. Содержит указатель на следующую команду. Регистр напрямую недоступен, изменяется неявно командами условных и безусловных переходов, вызова и возврата из подпрограмм.

**Стек**

Мы полагаем, что читатель имеет опыт программирования на Си и знаком со структурами данных типа стек. В микропроцессоре стек работает похожим образом: это область памяти, у которой определена вершина (на неё указывает %esp). Поместить новый элемент можно только на вершину стека, при этом новый элемент становится вершиной. Достать из стека можно только верхний элемент, при этом вершиной становится следующий элемент. У вас наверняка была в детстве игрушка-пирамидка, где нужно было разноцветные кольца надевать на общий стержень. Так вот, эта пирамидка — отличный пример стека. Также можно провести аналогию с составленными стопкой тарелками. На разных архитектурах стек может "расти" как в сторону младших адресов (принцип описан ниже, подходит для x86), так и старших.

Содержимое стека Адреса в памяти

. .

. .

. .

+----------------+ 0x0000F040

| |

+----------------+ 0x0000F044 <-- вершина стека (на неё указывает %esp)

| данные |

+----------------+ 0x0000F048

| данные |

+----------------+ 0x0000F04C

. .

. .

. .

+----------------+ 0x0000FFF8

| данные |

+----------------+ 0x0000FFFC

| данные |

+----------------+ 0x00010000 <-- дно стека

Стек растёт в сторону младших адресов. Это значит, что последний записанный в стек элемент будет расположен по адресу младше остальных элементов стека.

При помещении нового элемента в стек происходит следующее (принцип работы команды push):

* значение %esp уменьшается на размер элемента в байтах (4 или 2);
* новый элемент записывается по адресу, на который указывает %esp.

. .

. .

. .

+----------------+ 0x0000F040 <-- новая вершина стека (%esp)

| новый элемент |

+----------------+ 0x0000F044 <-- старая вершина стека

| данные |

+----------------+ 0x0000F048

. .

. .

. .

+----------------+ 0x0000FFFC

| данные |

+----------------+ 0x00010000 <-- дно стека

При выталкивании элемента из стека эти действия совершаются в обратном порядке(принцип работы команды pop):

* содержимое памяти по адресу, который записан в %esp, записывается в регистр;
* а значение адреса в %esp увеличивается на размер элемента в байтах (4 или 2).

. .

. .

. .

+-----------------+ 0x0000F040 <-- старая вершина стека

| верхний элемент | -------------> записывается в регистр

+-----------------+ 0x0000F044 <-- новая вершина стека (%esp)

| данные |

+-----------------+ 0x0000F048

. .

. .

. .

+-----------------+ 0x0000FFFC

| данные |

+-----------------+ 0x00010000 <-- дно стека

**Память**

В Си после вызова malloc(3) программе выделяется блок памяти, и к нему можно получить доступ при помощи указателя, содержащего адрес этого блока. В ассемблере то же самое: после того, как программе выделили блок памяти, появляется возможность использовать указывающий на неё адрес для всевозможных манипуляций. Наименьший по размеру элемент памяти, на который может указать адрес, — байт. Говорят, что память адресуется побайтово, или гранулярность адресации памяти — один байт. Отдельный бит можно указать как адрес байта, содержащего этот бит, и номер этого бита в байте.

Правда, нужно отметить ещё одну деталь. Программный код расположен в памяти, поэтому получить его адрес также возможно. Стек — это тоже блок памяти, и разработчик может получить указатель на любой элемент стека, находящийся под вершиной. Таким образом организовывают доступ к произвольным элементам стека.

**Порядок байтов. Little-endian и big-endian**

Оперативная память — это массив битовых значений, 0 и 1. Не будем говорить о порядке битов в байте, так как указать адрес отдельного бита невозможно; можно указать только адрес байта, содержащего этот бит. А как в памяти располагаются байты в слове? Предположим, что у нас есть число 0x01020304. Его можно записать в виде байтовой последовательности:

|  |  |  |
| --- | --- | --- |
| начиная со старшего байта: | 0x01 0x02 0x03 0x04 | — big-endian |
| начиная с младшего байта: | 0x04 0x03 0x02 0x01 | — little-endian |

Вот эта байтовая последовательность располагается в оперативной памяти, адрес всего слова в памяти — адрес первого байта последовательности.

Если первым располагается младший байт (запись начинается с «меньшего конца») — такой порядок байт называется little-endian, или «интеловским». Именно он используется в процессорах x86.

Если первым располагается старший байт (запись начинается с «большего конца») — такой порядок байт называется big-endian.

У порядка little-endian есть одно важное достоинство. Посмотрите на запись числа 0x00000033:

0x33 0x00 0x00 0x00

Если прочесть его как двухбайтовое значение, получим 0x0033. Если прочесть как однобайтовое, получим 0x33. При записи этот трюк тоже работает. Конечно же, мы не можем прочитать число 0x11223344 как байт, потому что получим 0x44, что неверно. Поэтому считываемое число должно помещаться в целевой диапазон значений.