Тема 12. Архитектуры вычислительных машин и языки ассемблера

Программисту не так уж важно разбираться в тонкостях и особенностях внутренней организации вычислительных машин и систем. Тем не менее мы сделаем короткую «вводную». Обсуждаются вопросы, перечисленные на СЛАЙДЕ 1.

12.1 Архитектура фон Неймана

Принципы организации современных вычислительных систем основан на концепции Джона фон Неймана, согласно которой вычислительная машина содержит устройство управления, арифметико-логическое устройство (АЛУ), память и устройства ввода-вывода (УВВ). В связях, соединяющих устройства, выделены потоки данных, команд и управляющих сигналов. Преобразование данных выполняется последовательно под управлением программы, состоящей из команд. Набор команд составляет машинный язык низкого уровня. Общий вид показан на СЛАЙДЕ 2.

Для организации вычислительной системы предложены следующие принципы:

- двоичное кодирование данных, их разделение на слова фиксированной разрядности;
- память имеет линейно-адресную организацию. Средства для записи, хранения и чтения *n*-разрядного слова называются **ячейкой памяти**. Они нумеруются по порядку от 0 до $N - 1$, где N – количество ячеек. Номер ячейки — это адрес, который в командах является именем программного объекта, хранящегося в ячейке;
- программа состоит из команд, каждая из которых определяет очередной шаг процесса выполнения. Команда состоит из кода операции, адресов операндов и дополнительных служебных кодов;
- команды и данные хранятся в одной памяти, различаются они только способом использования и интерпретацией считанного из памяти слова;
- вычислительный процесс организуется как последовательное выполнение команд в порядке, определенном в программе;
- в процессе работы структура вычислительной системы, набор команд и методы кодирования не изменяются.

АЛУ и устройство управления как правило объединены в один блок, называемый **процессором**. Он читает и выполняет команды, обеспечивает обращение к основной памяти (ОП), инициирует работу УВВ.

Устройство ввода преобразует входные сигналы в сигналы, принятые для представления данных на шине, соединяющей устройство ввода с АЛУ. Команды и данных находятся в ОП, туда же записываются промежуточные результаты. Они поступают в устройство вывода.

Устройство вывода преобразует выходные сигналы в форму, удобную для восприятия человеком или внешней системой.

Выборка команды и ее выполнение циклически повторяются. Фазы цикла: выборка, декодирование, исполнение.

На фазе выборки содержимое ячейки памяти, адресуемое так называемым **счетчиком команд** (*program counter*), выбирается из памяти и размещается в специальный компонент процессора, называемый **регистром команд**. На фазе

декодирования код команды поступает дешифратор команд и преобразуется в последовательность сигналов, обеспечивающих «настройку» АЛУ на выполнение операции, определяемой командой. После этого в устройстве управления формируется адрес следующей команды. Далее идет фаза исполнения команды, после которой происходит возврат к выборке следующей команды.

Архитектура вычислительной машины (системы) — набор типов данных, операций и характеристик каждого отдельно взятого уровня. Архитектура связана с программными аспектами. Аспекты реализации (например, технология, применяемая при реализации памяти) не являются частью архитектуры. Выделяют несколько уровней организации компьютера (**компьютерной архитектуры**), от двух и более (СЛАЙД 3).

Архитектура процессора — количественная составляющая компонентов микроархитектуры вычислительной машины (процессора компьютера). Например, наличие или отсутствие регистра флагов, количество регистров процессора и т. д.

Имеются различные классификации архитектур процессоров как по организации (например, по количеству и скорости выполнения команд: RISC, CISC), так и по назначению (например, специализированные графические, математические или предназначенные для цифровой обработки сигналов).

Программистами архитектура процессора рассматривается с точки зрения совместимости с определенным набором команд (например, процессоры, совместимые с командами Intel x86), их структуры (например, систем адресации или организации регистровой памяти) и способа исполнения (например, счётчик команд).

Архитектура CISC (от англ. *complex instruction set computing* или *complex instruction set computer*) — тип процессорной архитектуры, которая характеризуется, как правило, следующим набором свойств:

- нефиксированное значение длины команды;
- арифметические действия кодируются в одной команде;
- небольшое число регистров, каждый из которых выполняет строго определённую функцию.

Типичными представителями CISC-архитектуры являются процессоры на основе команд x86.

RISC (от англ. *restricted* (*reduced*) *instruction set computer* — «компьютер с сокращённым набором команд») — архитектура процессора, в котором быстродействие увеличивается за счет упрощения команд, чтобы их декодирование было более простым, а время выполнения — меньшим.

Характерные особенности:

- Фиксированная длина машинных инструкций (например, 32 бита) и простой формат команды.
- Специализированные команды для операций с памятью - чтения или записи. Операции вида Read-Modify-Write («прочитать-изменить-записать), обычно отсутствуют. Любые операции «изменить» выполняются только над содержимым регистров (т. н. подход *load-and-store*).
- Большое количество регистров общего назначения (32 и более).
- Отсутствие поддержки операций вида «изменить» над укороченными типами данных - байт, 16-разрядное слово.
- Отсутствие микропрограмм внутри самого процессора.

Типичными представителями RISC-архитектуры являются процессоры на основе команд MIPS, SPARC, Alpha, ARM, POWER и др.

Конечно, в современном мире редко требуется программировать на языке Ассемблера. Большую часть задач можно решить другими средствами, наиболее популярным из которых является язык программирования C (СЛАЙД 4). Однако, ассемблер не заменим с точки зрения изучения вычислительных архитектур.

12.2. Введение программирование на языке ассемблера для архитектуры x86

Мы будем использовать *gas*, однако существуют и альтернативы *fasm*, *nasm* и другие. Ими можно будет пользоваться, но при соблюдении определенных условий. Часть из рассматриваемого далее материала будет использовано при изучении ассемблера для MIPS32, но с определенной долей условности, конечно.

Регистры

Регистр – это небольшой объем очень быстрой памяти, размещённой на процессоре. Он предназначен для хранения результатов промежуточных вычислений, а также некоторой информации для управления работой процессора. Так как регистры размещены непосредственно на 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, а также регистры математического сопроцессора, регистры MMX, SSE, SSE2, SSE3. **Системные** регистры (регистры управления, регистры управления памятью, регистры отладки, машинно-специфичные регистры MSR и другие мы рассматривать не будем.

Регистры общего назначения (РОН, англ. *General Purpose Registers*, сокращённо GPR). Размер – 32 бита (СЛАЙД 5).

- %*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 базового адреса стекового фрейма (*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» (СЛАЙД 6). Например, к младшим 16 битам регистра %*eax* можно обратиться как %*ax*. Регистр %*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*.

К сожалению, обратиться, например, к старшей части регистра %*eax* простым образом не получится. Придется ухищряться разными способами, например, с использованием инструкций циклического сдвига и др.

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 обычно документировано.

Сегментные регистры (СЛАЙД 7):

%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 Linux используется плоская модель памяти (*flat memory model*), в которой все сегменты описаны как использующие все адресное пространство процессора. Они, 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 ОС предоставляет специальные средства (системный вызов *modify_ldt*()), позволяющие описывать нестандартные сегменты и работать с ними. Однако такая потребность возникает редко, поэтому мы вдаваться в подробности не будем.

Регистр флагов *eflags* (и его младшие 16 бит, 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 (СЛАЙД 8).

- 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*) – СЛАЙД 9. Размер – 32 бита. Содержит указатель на следующую команду. Регистр напрямую недоступен, изменяется неявно командами условных и безусловных переходов, вызова и возврата из подпрограмм.

Первая программа на ассемблере приведена на СЛАЙДЕ 10 и позволяет вывести на экран текстовое сообщение.

Команды

Команды ассемблера (СЛАЙД 11) – это те инструкции, которые будет исполнять процессор. По сути, это самый низкий уровень программирования процессора. Каждая команда состоит из операции (что делать?) и операндов (аргументов). Операции мы будем рассматривать отдельно. А операнды у всех операций задаются в одном и том же формате. Операндов может быть от 0 (то есть нет вообще) до 3. В роли операнда могут выступать:

- Конкретное значение, известное на этапе компиляции, – например, числовая константа или символ. Записываются при помощи знака $, например: $0xf1, $10, $*hello_str*. Эти операнды называются **непосредственными**.
- **Регистр**. Перед именем регистра ставится знак %, например: %*eax*, %*bx*, %*cl*.
- **Указатель** на ячейку в памяти.
- **Неявный операнд**. Эти операнды не записываются непосредственно в исходном коде, а подразумеваются. Нет, конечно, компьютер не читает мысли. Просто некоторые команды всегда обращаются к определенным регистрам без явного указания, так как это входит в логику их работы. Такое поведение обычно описывается в документации.

Почти у каждой команды можно определить операнд-источник (из него команда читает данные) и операнд-назначение (в него команда записывает результат). Общий синтаксис команды ассемблера приведен на СЛАЙДЕ 12.

Суффикс *l* в имени команды указывает на то, что ей следует работать с операндами длиной в 4 байта. Все суффиксы:

- *b* (*byte*) – 1 байт,
- *w* (*word*) – 2 байта,
- *l* (*long*) – 4 байта,
- *q* (*quadword*) – 8 байт.

Важной особенностью всех команд является то, что они не могут работать с двумя операндами, находящимися в памяти. Хотя бы один из них следует сначала загрузить в регистр, а затем выполнять необходимую операцию.

Синтаксис указателя на ячейку памяти (СЛАЙД 13):

```
смещение(база, индекс, множитель)
```

Вычисленный адрес будет равен **база** + **индекс** × **множитель** + **смещение**. Множитель может принимать значения 1, 2, 4 или 8. Например:

- (%*ecx*) адрес операнда находится в регистре %*ecx*. Этим способом удобно адресовать отдельные элементы в памяти, например, указатель на строку или указатель на *int*;
- 4(%*ecx*) адрес операнда равен %*ecx* + 4. Удобно адресовать отдельные поля структур. Например, в %ecx адрес некоторой структуры, второй элемент которой находится «на расстоянии» 4 байта от её начала (говорят «по смещению 4 байта»);
- -4(%*ecx*) адрес операнда равен %*ecx* − 4;
- *kuku*(, %ecx, 4) адрес операнда равен *kuku* + %*ecx* × 4, где *kuku* – некоторый адрес. Удобно обращаться к элементам массива. Если *kuku* – указатель на массив, элементы которого имеют размер 4 байта, то мы можем заносить в %ecx номер элемента и таким образом обращаться к самому элементу.

Команды нужно помещать в секцию кода. Для этого перед командами нужно указать директиву *.text*, как мы сделали на СЛАЙДЕ 10.

Данные

Существуют директивы ассемблера, которые размещают в памяти данные, определенные программистом. Аргументы этих директив – список выражений, разделенных запятыми (СЛАЙД 14).

.byte – размещает каждое выражение как 1 байт;

.short – 2 байта;
.long – 4 байта;
.quad – 8 байт.

Например:
```
.byte   0x10, 0xf5, 0x42, 0x55
.long   0xaabbaabb
.short  -123, 456
```

Также существуют директивы для размещения в памяти строковых литералов (СЛАЙД 15):

.ascii "STR" размещает строку STR. Нулевых байтов не добавляет.

.string "STR" размещает строку STR, после которой следует нулевой байт (как в языке *C*).

У директивы *.string* есть синоним *.asciz* (z от англ. *zero* – ноль, указывает на добавление нулевого байта).

Строка-аргумент этих директив может содержать стандартные escape-последовательности, которые мы используем в C, например, \n, \r, \t, \\, \" и т. д.

Данные нужно помещать в секцию данных. Для этого перед данными нужно поместить директиву *.data*. Вот так (СЛАЙД 16):
```
.data
        .string "Hello, goodbye and farewell\n"
        ...
```

Если некоторые данные не предполагается изменять в ходе выполнения программы, их можно поместить в специальную секцию данных только для чтения при помощи директивы *.section .rodata*:
```
.section .rodata
        .string "program version 0.419"
```

Выравнивание задано у каждого фундаментального типа данных (типа данных, которым процессор может оперировать непосредственно). Например, выравнивание *word* – 4 байта. Это значит, что данные типа *word* должны располагаться по адресу, кратному 4 (например, 0x00000100, 0x03284473). Выравнивание рекомендуется, но не требуется: доступ к невыровненным данным может быть медленнее, но принципиальной разницы нет, и ошибки это не вызовет (СЛАЙД 17).

Для соблюдения выравнивания в распоряжении программиста есть директива *.p2align*.
```
.p2align степень_двойки, заполнитель, максимум
```

Директива *.p2align* выравнивает текущий адрес до заданной границы. Граница выравнивания задаётся как степень числа 2: например, если мы указали *.p2align* 3 – следующее значение будет выровнено по 8-байтной границе. Для выравнивания размещается необходимое количество байт-заполнителей со значением *заполнитель*. Если для выравнивания требуется разместить более чем максимум заполнителей, то выравнивание не выполняется.

Второй и третий аргумент являются необязательными.

См. примеры на СЛАЙДЕ 17.

Мы не присвоили имен нашим данным. Как же к ним обращаться? Очень просто: нужно поставить метку. **Метка** – это просто константа, значение которой – адрес (СЛАЙД 18).

```
hello_str:
        .string "Hello, goodbye and farewell!\n"
```

Сама метка, в отличие от данных, места в памяти программы не занимает. Когда компилятор ассемблера встречает в исходном коде метку, он запоминает текущий адрес и читает код дальше. В результате ассемблер помнит все метки и адреса, на которые они указывают. Программист может ссылаться на метки в своём коде. Существует специальная псевдометка, указывающая на текущий адрес. Это метка . (точка).

Значение метки как константы – это всегда адрес. А если нам нужна константа с каким-то другим значением? Тогда мы приходим к более общему понятию «символа». **Символ** – это просто некоторая константа. Причём он может быть определён в одном файле, а использован в других. Просмотреть символы в объектном модуле можно, используя утилиту *nm* (СЛАЙД 19). Например:

```
$ nm hello.o
00000000 d hello_str
0000000e a hello_str_length
00000000 T main
```

В первой колонке выводится значение символа, во второй – его тип, в третьей – имя. Посмотрим на символ *hello_str_length*. Это длина строки. Значение символа чётко определено и равно 0xe, об этом говорит тип *a* – *absolute value*. А вот символ *hello_str* имеет тип *d* – значит, он находится в секции данных (*data*). Символ *main* находится в секции кода (*text section*, тип *T*). А почему *a* представлено строчной буквой, а *T* – прописной? Если тип символа обозначен строчной буквой, значит это локальный символ, который видно только в пределах данного файла. Заглавная буква говорит о том, что символ глобальный и доступен другим модулям. Символ *main* мы сделали глобальным при помощи директивы *.globl main*.

Для создания нового символа используется директива *.set*. Синтаксис (СЛАЙД 20):

```
 .set    символ, выражение
```

Например, определим символ *kookoo* = 42:

```
.set    kookoo, 42
```

Ещё пример из *hello.s*:

```
hello_str:
    .string "Hello, goodbye and farewell!\n"/*наша строка*/
    .set hello_str_length, . - hello_str - 1 /*длина строки*/
```

Сначала определяется символ *hello_str*, который содержит адрес строки. После этого мы определяем символ *hello_str_length*, который, судя по названию, содержит длину строки. Директива *.set* позволяет в качестве значения символа использовать арифметические выражения. Мы из значения текущего адреса (метка «точка») вычитаем адрес начала строки – получаем длину строки в байтах. Потом мы вычитаем ещё единицу, потому что директива *.string* добавляет в конце строки нулевой байт (а на экран мы его выводить не хотим).

Часто требуется просто зарезервировать место в памяти для данных, без инициализации какими-то значениями. Например, у нас есть переменная, значение которой определяется параметрами командной строки. Действительно, вряд ли мы сможем дать ей какое-то осмысленное начальное значение, разве что NIL, 0 и прочие малопонятные имена. Такие данные называются **неинициализированными**, и для них выделена специальная секция под названием *.bss*. В скомпилированной программе эта секция места не занимает. При загрузке программы в память секция неинициализированых данных будет заполнена нулевыми байтами.

Версия 0.91 pre-release от 07.11.2016. Возможны незначительные изменения.

Хорошо, но известные нам директивы размещения данных требуют указания инициализирующего значения. Поэтому для неинициализированных данных используются специальные директивы (СЛАЙД 21):

```
.space  количество_байт
.space  количество_байт, заполнитель
```

Директива *.space* резервирует *количество_байт* байт.

Также эту директиву можно использовать для размещения инициализированных данных, для этого существует параметр заполнитель – этим значением будет инициализирована память.

Например:

```
.bss
long_var_1:                     /* по размеру как .long                */
        .space 4

buffer:                         /* какой-то буфер в 1024 байта         */
        .space 1024

struct:                         /* какая-то структура размером 20 байт */
        .space 20
```

Семейство команд *mov*

Синтаксис:

```
movX   источник, назначение
```

Команда *movX* производит копирование источника в назначение. Рассмотрим пример (СЛАЙДЫ 22 -23).

Целочисленная арифметика

Арифметических команд в нашем распоряжении довольно много (СЛАЙД 24). Синтаксис:

inc операнд
dec операнд
neg операнд
add источник, приёмник
sub источник, приёмник
mul множитель_1
imul множитель_1
div делитель
idiv делитель

Принцип работы:
inc: увеличивает операнд на 1.
dec: уменьшает операнд на 1.
neg: меняет знак операнда (унарный минус).
add: приёмник = приёмник + источник (то есть, увеличивает приёмник на источник).
sub: приёмник = приёмник - источник (то есть, уменьшает приёмник на источник).

Команда *mul* имеет только один операнд. Второй сомножитель задаётся неявно. Он находится в регистре %*eax*, и его размер выбирается в зависимости от суффикса команды (*b*, *w* или *l*). Место размещения результата также зависит от суффикса команды. Нужно отметить, что результат умножения двух разрядных чисел может уместиться только в разрядном регистре результата. В следующей таблице описано, в какие регистры попадает

результат при той или иной разрядности операндов.

Команда	Второй сомножитель	Результат
mulb	*%al*	16 бит: *%ax*
mulw	*%ax*	32 бита: младшая часть в *%ax*, старшая в *%dx*
mull	*%eax*	64 бита: младшая часть в *%eax*, старшая в *%edx*

Операндами команд умножения *imul* являются целые числа со знаком.

Примеры (СЛАЙДЫ 25-26):

```
.text
        movl  $72, %eax
        incl  %eax              /* в %eax число 73                    */
        decl  %eax              /* в %eax число 72                    */

        movl  $48, %eax
        addl  $16, %eax         /* в %eax число 64                    */

        movb  $5, %al
        movb  $5, %bl
        mulb  %bl               /* в регистре %ax произведение
                                   %al × %bl = 25                     */
```

Каков будет результат сложения байтовых значений 252 и 8? Здесь возникает ситуация переполнения, которую желательно перехватывать. См. СЛАЙД 26.

```
        movb  $0,   %ah         /* %ah = 0                            */
        movb  $252, %al         /* %al = 252                          */
        addb  $8,  %al         /* %al = %al + 8
                                   происходит переполнение,
                                   устанавливается флаг cf;
                                   в %al число 4                      */
        jnc   no_carry          /* если переполнения не было, перейти
                                   на метку                           */
        movb  $1,   %ah         /* %ah = 1                            */
no_carry:
        /* %ax = 260 = 0x0104 */
```

Этот код выдаёт правильную сумму в регистре *%ax* с учётом переполнения, если оно произошло. Что произойдет, если поменять числа в строках 2 и 3?

Для реализации условного оператора (*if*) можно воспользоваться командой *cmp*. Ее синтаксис (СЛАЙД 27):

```
cmp   операнд_2, операнд_1
```

Команда *cmp* выполняет вычитание *операнд_1 – операнд_2* и устанавливает флаги. Результат вычитания нигде не запоминается.

Сравнили, установили флаги, – а дальше-то что? А у нас есть целое семейство *jump*-команд, которые передают управление другим командам. Эти команды называются командами условного перехода. Каждой из них поставлено в соответствие условие, которое она проверяет. Синтаксис (СЛАЙД 27):

```
jcc   метка
```

Команды *jcc* не существует, вместо *cc* нужно подставить мнемоническое обозначение условия (СЛАЙД 27).

Мнемоника	Английское слово	Смысл	Тип операндов
e	*equal*	равенство	любые
n	*not*	инверсия условия	любые

g	*greater*	больше	со знаком
l	*less*	меньше	со знаком
a	*above*	больше	без знака
b	*below*	меньше	без знака

Есть также безусловный переход *jmp*.

Таким образом, *je* проверяет равенство операндов команды сравнения, *jl* проверяет условие *операнд_1* < *операнд_2* и так далее. У каждой команды есть противоположная: просто добавляем букву *n*:

je – jne: равно – не равно;

jg – jng: больше – не больше.

Пример кода приведен на СЛАЙДЕ 28.

12.3. Память и адресация в Ассемблере

В языке С после вызова функции *malloc* 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». На отдельный бит можно сослаться адресом байта, содержащего этот бит, и номером этого бита в байте.

Заметим еще, что 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 способом организовывают доступ к произвольным элементам стека.

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 коротком слове, длинном слове или квадрослове? Предположим, что у нас есть число 0x01020304. Его можно записать в виде байтовой последовательности (СЛАЙД 29):

- 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 системах на основе x86.

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

Методы адресации

Пространство памяти предназначено для хранения кодов команд и данных, для доступа к которым имеется богатый выбор методов адресации (около 24). См. СЛАЙД 30. Операнды могут находиться во внутренних регистрах процессора (наиболее удобный и быстрый вариант). Они могут располагаться в системной памяти (самый распространенный вариант). Наконец, они могут находиться в устройствах ввода-вывода (наиболее редкий случай). Определение местоположения операндов производится кодом команды. Причем существуют разные методы, с помощью которых код команды может

определить, откуда брать входной операнд и куда помещать выходной операнд. Эти методы называются **методами адресации**. Эффективность выбранных методов адресации во многом определяет эффективность работы всего процессора в целом.

Прямая или абсолютная адресация

Физический адрес операнда содержится в адресной части команды. Формальное обозначение:

Операнд$_i$ = (A_i),

где A_i – код, содержащийся в i-м адресном поле команды.

См. пример кода на СЛАЙДЕ 31.

Непосредственная адресация

В команде содержится не адрес операнда, а непосредственно сам операнд.

Операнд$_i$= A_i.

Непосредственная адресация позволяет повысить скорость выполнения операции, так как в этом случае вся команда, включая операнд, считывается из памяти одновременно и на время выполнения команды хранится в процессоре в специальном регистре команд. Однако при использовании непосредственной адресации появляется зависимость кодов команд от данных, что требует изменения программы при каждом изменении непосредственного операнда.

Пример кода на СЛАЙДЕ 32.

Косвенная (базовая) адресация

Адресная часть команды указывает на адрес ячейки памяти или регистр, в котором содержится адрес операнда:

Операнд$_i$ = ((A_i))

Применение косвенной адресации операнда из оперативной памяти при хранении его адреса в регистрах существенно сокращает длину поля адреса, одновременно сохраняя возможность использовать полную разрядность регистра для указания физического адреса. Недостаток этого способа – необходимо дополнительное время для чтения адреса операнда. Вместе с тем он существенно повышает гибкость программирования. Изменяя содержимое ячейки памяти или регистра, через которые осуществляется адресация, можно, не меняя команды в программе, обрабатывать операнды, хранящиеся по разным адресам. Косвенная адресация не применяется по отношению к операндам, находящимся в регистрах.

Пример кода на СЛАЙДЕ 33.

Предоставляемые косвенной адресацией возможности могут быть расширены, если в системе команд процессора предусмотреть определенные арифметические и логические операции над ячейкой памяти или регистром, через которые выполняется адресация, например увеличение или уменьшение их значения.

Автоинкрементная и автодекрементная адресация

Иногда, адресация, при которой после каждого обращения по заданному адресу с использованием механизма косвенной адресации, значение адресной ячейки автоматически увеличивается на длину считываемого операнда, называется **автоинкрементной**. Адресация с автоматическим уменьшением значения адресной ячейки называется **автодекрементной**.

Регистровая адресация

Предполагается, что операнд находится во внутреннем регистре процессора.

Версия 0.91 pre-release от 07.11.2016. Возможны незначительные изменения.

Пример кода на СЛАЙДЕ 34.

Относительная адресация

Используется тогда, когда память логически разбивается на блоки, называемые **сегментами**. В этом случае адрес ячейки памяти содержит две составляющих: адрес начала сегмента (**базовый адрес**) и **смещение** адреса операнда в сегменте. Адрес операнда определяется как сумма базового адреса и смещения относительно этой базы (СЛАЙДЫ 35-36):

$$\text{Операнд}_i = (\text{база}_i + \text{смещение}_i)$$

Для задания базового адреса и смещения могут применяться рассмотренные выше способы адресации. Как правило, базовый адрес находится в одном из регистров, а смещение может быть задано в самой команде или регистре.

Два примера:

1. Адресное поле команды состоит из двух частей, в одной указывается номер регистра, хранящего базовое значение адреса (начальный адрес сегмента), а в другой (адресное поле) задается смещение, определяющее положение ячейки относительно начала сегмента. Именно такой способ представления адреса обычно и называют относительной адресацией.
2. Первая часть адресного поля команды также определяет номер базового регистра, а вторая содержит номер регистра, в котором находится смещение. Такой способ адресации чаще всего называют **базово-индексным**.

Основной недостаток относительной адресации – большое время вычисления физического адреса операнда. Существенное преимущество этого способа адресации – возможность создания «перемещаемых» программ (можно размещать в различных частях памяти без изменения команд). То же относится к программам, обрабатывающим по единому алгоритму информацию, расположенную в различных областях ЗУ. В этих случаях достаточно изменить содержимое базового адреса начала команд программы или массива данных, а не модифицировать сами команды. По этой причине относительная адресация облегчает распределение памяти при разработке сложных программ и широко используется при автоматическом распределении памяти в мультипрограммных операционных системах.

Команды манипуляции адресами и стеком

Команда lea

Load Effective Address, синтаксис (СЛАЙД 37):

```
lea  источник, назначение
```

Команда *lea* помещает адрес источника в назначение. Источник должен находиться в памяти (Иначе говоря, не может быть непосредственным значением, т.е. константой или регистром).

Пример кода приведен на СЛАЙДЕ 37.

Для выполнения некоторых арифметических операций так же можно использовать команду *lea*. Она вычисляет адрес своего операнда-источника и помещает этот адрес в операнд-назначение. Она не производит чтение памяти по этому адресу, следовательно, всё равно, что она будет вычислять: адрес или какие-то другие числа.

Вспомним, как формируется адрес операнда:

```
смещение(база, индекс, множитель)
```

Вычисленный адрес = *база* + *индекс* × *множитель* + *смещение*. Для чего это может понадобиться? Мы можем получить команду с двумя операндами-источниками и одним

результатом:

Пример кода приведен на СЛАЙДЕ 38.

При сложении командой *add* результат записывается на место одного из слагаемых. Теперь, возможно, становится более ясным главное преимущество команды *lea* в тех случаях, где её можно применить: она не перезаписывает операнды-источники. Как мы это сможем использовать, зависит только от задачи и фантазии: прибавить константу к регистру и записать в другой регистр, сложить два регистра и записать в третий... Также *lea* можно применять для умножения регистра на 3, 5 и 9, как показано на СЛАЙДЕ 38.

Команды для работы со стеком

Предусмотрено две специальные команды для работы со стеком: *push* (втолкнуть в стек) и pop (вытолкнуть из стека). Синтаксис (СЛАЙД 39):

```
push  источник
pop   назначение
```

Команды *push* и *pop* работают только с операндами размером 4 или 2 байта. Если мы попробуем скомпилировать код со следующей командой:

```
pushb 0x1F
```

GCC выдаст сообщение о неверном суффиксе для команды *push*.

В Linux стек выровнен по *long*. Сама архитектура этого не требует, это только соглашение между программами, но не стоит рассчитывать на то, что другие библиотеки функций или ОС захотят работать с невыровненным стеком. Что всё это значит? Если мы резервируем место в стеке, количество байт должно быть кратно размеру *long*, т.е. 4. Даже если нам нужно всего 2 байта в стеке для *short*, то всё равно придётся резервировать 4 байта, чтобы соблюдать выравнивание.

Пример кода приведен на СЛАЙДЕ 40.

Команда *pushl %esp* должна помещать в стек уменьшенное значение %*esp*, однако в [5] сказано, что в стек помещается такое значение %*esp*, каким оно было до выполнения команды – и она действительно работает именно так.

12.4 Подпрограммы: общие принципы

Подпрограммой называется некоторая обособленная часть программного кода, которая может быть вызвана из главной программы или другой подпрограммы. Под вызовом подпрограммы мы понимаем временную передачу управления подпрограмме с тем, чтобы после завершения работы подпрограммы, она могла вернуть управление в ту точку кода, откуда ее вызвали. Очевидно, что многие из нас встречались с подпрограммами в рамках своих программ или просмотре чужого кода. В языке С (да, и пожалуй, во всех языках, основанных на кодовой базе этого языка) подпрограммы называются **функциями**, в Паскале и некоторых других языках программирования подпрограммы бывают двух видов – **процедуры** и **функции**.

При вызове подпрограммы необходимо запоминать адреса возврата, т.е. адрес машинной команды, следующей за команды вызова подпрограммы, причем сделать это так, чтобы сама вызываемая подпрограмма могла, когда закончит свою работу, воспользоваться этим сохраненным адресом для возврата управления. Кроме того, как правило, подпрограммы получают параметры, влияющие на их работу, и для своей работы они используют локальные переменные. Следовательно, и под параметры, и под локальные переменные необходимо предусмотреть распределение памяти. Самый простой способ решения этого вопроса – выделить каждой подпрограмме свою собственную

область памяти под хранение всей требуемой информации, включая и адрес возврата, и параметры, и локальные переменные. Тогда, прежде всего вызов подпрограммы потребует записать в принадлежащую ей область памяти (в заранее оговоренные места) значения параметров и адрес возврата, а затем передать управление в начало подпрограммы.

Когда-то давным-давно именно так с подпрограммами и поступали. Однако с развитием методов и приемов программирования возникала потребность в рекурсии – таком построении программы, при котором подпрограммы могут прямо или косвенно вызывать сами себя такое количество раз, которое ограничено лишь ресурсами вычислительной машины. Очевидно, что при каждом рекурсивном вызове требуется новый экземпляр области памяти для хранения адреса возврата, параметров и локальных переменных. При этом, чем позже такой экземпляр будет создан, тем раньше соответствующий вызов закончит свою работу. Иначе говоря, рекурсивные вызовы подпрограмм в определенном смысле подчиняется правилу LIFO. И совершенно логично, что из этого вытекает идея использования стека при вызовах подпрограмм.

В современных вычислительных системах перед вызовом подпрограммы в стек помещаются значения параметров вызова, затем производится собственно вызов (передача управления подпрограмме), который совмещается с сохранением в стеке адреса возврата. Когда подпрограмма получает управление, она резервирует в стеке определенное количество памяти для хранения локальных переменных, обычно просто сдвигая адрес вершины вниз на соответствующее количество ячеек. Область стековой памяти, содержащую связанные с одним вызовом значения параметров, адрес возврата и локальные переменные, называют **стековым фреймом**.

Итак, **вызов подпрограммы** – это передача управления по адресу начала подпрограммы с одновременным запоминанием в стеке адреса возврата (адреса команды, непосредственно следующей за командой вызова). Процессоры семейства x86_32 предусматривают для этой цели команду *call*. Она похожа на безусловный переход *jmp* в том смысле, что аргумент команды *call* может быть непосредственным (адрес перехода задан в команде, например, меткой), регистровым (адрес передачи управления находится в регистре) и косвенным (переход осуществляется по адресу, прочитанному из заданного места памяти). См. СЛАЙД 41.

Возврат из подпрограммы производится командой *ret* (программирующие на языке C должны уловить ассоциацию с оператором *return*). В одной из форм эта команда не имеет аргументов. Выполняя команду, процессор извлекает 4 байта с вершины стека и записывает их в регистр %*eip*. В результате управление перейдет на адрес, который находился в памяти на вершине стека.

Рассмотрим простой пример. Предположим, в программе часто приходится заполнять каким-то однобайтовым значением области памяти разного размера. Такое действие вполне логично оформить в виде подпрограммы. Для простоты картины примем соглашение, что адрес нужной области памяти передается через регистр %*edi*, количество однобайтовых ячеек, которые нужно заполнить – через регистр %*ecx*, и само значение-заполнитель ячеек – через регистр %*al*. Код соответствующей подпрограммы может выглядеть так, как показано на СЛАЙДАХ 42 и 43.

В результате вызова 80 байтов памяти, начиная с адреса, заданного меткой *array*, окажутся заполнены кодом символа ‘A’ (десятичное число 65).

12.5 Организация стековых фреймов

Подпрограмма, приведенная выше в качестве примера, фактически не использовала механизм стековых фреймов, сохраняя в стеке только адрес возврата. Этого оказалось достаточно, поскольку программе не требовались локальные переменные, а параметры передавались через регистры. На практике оказывается, что подпрограммы редко бывают такими простыми. В более сложных случаях обычно требуются локальные переменные, поскольку регистров может на все не хватить. Кроме того, передача параметров через

регистры тоже может оказаться неудобна.

Во-первых, регистров может и не хватить, т.к. иногда встречаются функции с большим и даже переменным числом аргументов, как, например, использованная нами функция *printf*.

Во-вторых, подпрограмме могут быть долго нужны значения, переданные через регистры, и это фактически лишит ее возможности использовать под свои внутренние нужды те из регистров, которые были задействованы при передаче параметров.

В-третьих, передача параметров через регистры, а также через какую-либо фиксированную область памяти, напрочь лишает нас возможности использовать рекурсию, что тоже, разумеется, не очень хорошо.

Поэтому обычно параметры в подпрограммы передаются через стек, и в стеке же размещаются локальные переменные (СЛАЙД 44). Выше было показано, что параметры в стеке размещает вызывающая программа. Затем при вызове подпрограммы в стек заносится адрес возврата, а после этого сама вызванная подпрограмма резервирует в стеке место под локальные переменные. Все это в совокупности и образует стековый фрейм. К его содержимому можно обращаться, используя адреса, «привязанные» к адресу, по которому находится адрес возврата. Иначе говоря, ячейку памяти, начиная с которой в стек был занесен адрес возврата, используют в качестве своего рода основания.

Так, если в стек занести три четырехбайтных параметра, а потом вызвать подпрограмму, то адрес возврата будет лежать в памяти по адресу (%*esp*), а параметры окажутся доступными по адресам 4(%*esp*), 8(%*esp*) и 12(%*esp*). Если же разместить в стеке локальные 4-байтные переменные, то они окажутся доступными по адресам -4(%esp), -8(%esp) и т.д. См. СЛАЙД 45.

Использовать для доступа к параметрам регистр %*esp* оказывается не всегда и не слишком удобным, ведь в самой подпрограмме нам тоже может потребоваться стек, как для временного хранения данных, так и для вызова других подпрограмм. Поэтому первым же своим действием (СЛАЙД 46) подпрограмма обычно сохраняет значение регистра %*esp* в каком-то другом регистре. Обычно в качестве такового выступает %*ebp*, который и используется для доступа к параметрам и локальным переменным. В то же время %*esp* продолжает играть свою главную роль указателя стека, изменяясь по мере необходимости. Перед возвратом из подпрограммы его обычно восстанавливают в исходном значении за счет пересылки в него значения из %*ebp*, чтобы оно снова указывал на адрес возврата.

И возникает следующий вопрос: что если другие подпрограммы тоже используют регистр %*ebp* для тех же целей? Ведь в этом случае первый же вызов другой подпрограммы испортит нам, как говорится, всю обедню. Можно, конечно, сохранять %*ebp* в стеке перед вызовом каждой подпрограммы, но поскольку в программе обычно гораздо больше вызовов подпрограмм, чем собственно их самих, то оказывается экономнее следовать простому правилу (СЛАЙД 47): каждая подпрограмма должна сама сохранять старое значение %*ebp* и восстановить его перед возвратом управления. Естественно, для сохранения значения %*ebp* тоже используется стек, причем сохранение выполняется простой командой *push %ebp* сразу после получения управления. Таким образом, старое значение %*ebp* помещается в стек непосредственно после адреса возврата из подпрограммы, и в качестве «точки привязки» в дальнейшем используется именно этот адрес. Для этого следующей командой выполняется *movl %esp, %ebp*. В результате в %*ebp* указывает в стеке на то место в стеке, где находится его сохраненное ранее значение. Если теперь обратиться к памяти по адресу 4(%*ebp*), мы обнаружим там адрес возврата из подпрограммы. Параметры, занесенные в стек перед вызовом подпрограммы, оказываются доступны по адресам 8(%*esp*), 12(%*esp*), 16(%*esp*) и т.д. Память под локальные переменные выделяются путем простого вычитания нужной длины из текущего значения %*esp*. Если под локальные переменные нужно 16 байт, то после сохранения %*ebp* и копирования в него содержимого %*esp*, нужно выполнить команду *subl $16, %esp*. Если, скажем, все наши локальные переменные занимают по 4 байта, они

окажутся доступными по адресам -4(%*ebp*), -8(%*ebp*) и т.д. Структура стекового фрейма с тремя 4-байтными параметрами и четырьмя 4-байтными локальными переменными показана на СЛАЙДЕ 48.

Повторим: в начале своей работы, согласно нашей договоренности, каждая подпрограмма должна выполнить такой код («**пролог**»), а в конце – другой код («**эпилог**»). Оба кода на СЛАЙДЕ 49.

Еще одно важное замечание. При работе в UNIX-подобных системах мы можем не беспокоиться ни о наличии стека, ни об указании его размера. Операционная система создает стек автоматически при запуске любой задачи. Более того, уже во время ее выполнения по необходимости увеличивает размер доступной для стека памяти. По мере того как вершина стека продвигается по виртуальному адресному пространству в сторону уменьшения адресов, операционная система ставит в соответствие виртуальным адресам все новые и новые разделы физической памяти. Именно поэтому на СЛАЙДЕ 48 мы сделали верхний край стека нечетким.

Литература и дополнительные источники к Теме 12

1. Магда, Ю.С. Ассемблер для процессоров Intel Pentium/ Ю.С. Магда. – СПб.: Питер, 2006. – 416 с.
2. Робачевский, А. Операционная система Unix, 2 изд./ А.Робачевский, С.Немнюгин, О.Стесик. – СПб.: БХВ-Петербург, 2010. – 656 с.
3. Столяров, А.В. Программирование на языке ассемблера NASM для ОС UNIX: учеб.пособие. – М.: Макс, 2011. – 188 с. – Доступ: http://www.stolyarov.info/books/asm_unix
4. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 3.5.1.3 Using LEA.
5. Intel® 64 and IA-32 Architectures Software Developer’s Manual, 4.1 Instructions (N-Z), PUSH