Тема 14. Архитектуры вычислительных машин и языки ассемблера, часть 3

14.1 Псевдокоманды

Под псевдокомандами понимается ряд вводимых ассемблером слов, которые могут использоваться синтаксически так же, как и мнемоники машинных команд, хотя командами на самом деле они не являются. Некоторые из команд нам уже известны. Одной из них в частности является *.set*. Эквивалент - *.equ*. Они предназначены для определения констант, и всегда применяются в сочетании с именем, т.е. не поставить после нее имя считается ошибкой. Синтаксис (СЛАЙД 2):

```
.equ symbol, expression
```

Псевдокоманда связывает имя с заданным выражением. Например:

```
.equ four, 4
```

Теперь можно использовать символьное имя, например:

```
movl $four, %ecx
```

Еще один сценарий использования *.equ* / *.set* нами уже демонстрировался (СЛАЙД 3):

```
hello_str:                      /* наша строка                          */
        .string "Hello, goodbye and farewell!\n"

                                /* длина строки                         */
        .set hello_str_length, . - hello_str - 1
```

Директива *.rept* позволяет повторить какую-нибудь команду или псевдокоманду заданное количество раз (закрывающей операторной скобкой является директива *.endr*, у которой нет аргументов). Синтаксис (СЛАЙД 3):

```
.rept count
```

Например, следующий код:

```
.rept   3
.long   0
.endr
```

повторяет 3 раза инструкцию *.long* 0.

Еще две полезные псевдокоманды - *.include* и *.incbin*. Синтаксис (СЛАЙД 3):

```
.include "file"
.incbin "file"[,skip[,count]]
```

Первая псевдокоманда включает в процесс ассемблирования содержимое файла *file*.

Вторая псевдокоманда может использоваться для непосредственного заполнения области памяти содержимым заданного файла. Второй операнд позволяет также пропустить *skip* первых байт файла, а *count* указывает максимальное количество байт для чтения. Важно отметить, что данные никак не выравниваются, и ответственность за это целиком и полностью ложится на программиста.

14.2 Вычисление выражений во время ассемблирования

Ассемблер в некоторых случаях вычисляет встретившиеся ему арифметические выражения непосредственно при ассемблировании. Важно осознавать, что в итоговый машинный код попадают только вычисленные результаты, а не сами действия по их вычислению. Естественно, для вычисления выражений при ассемблировании необходимо, чтобы оно не содержало никаких неизвестных: все, что нужно для вычисления, должно быть известно ассемблеру во время работы.

Выражение, вычисляемое ассемблером, как правило, должно быть целочисленным, т.е. состоять из констант, меток и подвыражений в скобках (СЛАЙД 4). Практически все операции те же, что и в языке C, с сохранением уровней приоритетов и ассоциативностей. Например, имеем такое объявление:

```
.equ num_expression, (6 - 3)*(10 / 2)
```

Очевидно, что после выполнения инструкции:

```
movl $num_expression, %ecx
```

в регистре *%ecx* будет содержаться число 15, которое было вычислено во время ассемблирования.

14.3 Макросредства и макропроцессор

Под макропроцессором (СЛАЙД 5) понимают программное средство, которое получает на вход некоторый текст и, пользуясь указаниями, данными в самом тексте, частично преобразует его, давая на выходе, в свою очередь, текст, но уже не имеющий указаний к преобразованию. В применении к языкам программирования макропроцессор – это преобразователь исходного текста программы, обычно совмещенный с компилятором: результатом работы макропроцессора является текст на языке программирования, который уже потом обрабатывается компилятором в соответствии с правилами языка (СЛАЙД 6, а также конспект соответствующего раздела курса лекций).

Поскольку языки ассемблера обычно по своим выразительным возможностям беднее своих высокоуровневых собратьев, то для компенсации этого разработчики ассемблеров обычно снабжают их мощными макропроцессорами. В частности, ассемблер *gas* содержит в себе алгоритмически полный макропроцессор. Т.е. при желании мы можем заставить его написать за нас почти всю программу ;)

Макросом (СЛАЙД 7) называют некоторое правило, в соответствии с которым фрагмент программы, содержащей определенное слово, должен быть преобразован. Само это слово называется **именем макроса**. Часто вместо «имени макроса» используется слово «макрос», хотя это не совсем верно.

Прежде чем мы сможем воспользоваться макросом, его необходимо определить. Это означает, во-первых, указать макропроцессору, что некий идентификатор теперь считается именем макроса. Следовательно, его появление в тексте программы требует вмешательства макропроцессора. Во-вторых, нужно задать то правило, по которому макропроцессор должен действовать, встретив это имя. Фрагмент программы, определяющий макрос, называют **макроопределением**. Когда макропроцессор встречает в тексте программы имя макроса и параметры (это место в коде называется **вызовом макроса** или **макровызовом**), он заменяет имя макроса и параметры на некий текст,

полученный в соответствии с определением макроса. Такая замена называется **макроподстановкой**, а текст, полученный в результате – **макрорасширением**.

Бывает, что макропроцессор производит преобразование текста программы, не видя ни одного имени макроса, но повинуясь еще более прямым указаниям в виде **макродиректив**. Одну такую макродирективу мы упомянули выше - *.include*, которая приказывает макропроцессору заменить ее саму на содержимое файла, указанного параметром директивы (СЛАЙД 8).

```
.include "abrakadabra.inc"
```

14.4 Однострочные макросы

Для их использования у нас уже есть практически все (СЛАЙД 9). Директивы *.equ* и *.set* рассматривались выше, есть еще один эквивалент, который похож на оператор присваивания в языке С. Например, три следующих однострочные макроопределения идентичны:

```
.equ five, 5
.set five_, 5
_five_ = 5
```

К сожалению, отменить эти макроопределения легким образом не получится, можно лишь их переопределить.

14.5 Условное ассемблирование

Часто при разработке программ возникает потребность в создании различных версий исполняемого файла с использованием одного и того же исходного текста. Скажем, у нас есть отладочная версия и «релиз». Программы почти одинаковые, но в каждой есть специфические потребности, отсутствующие в другой. В такой ситуации, конечно, хочется иметь и поддерживать один исходный текст, иначе появятся две копии одного и того же кода, и придется, например, каждую найденную ошибку исправлять в двух местах. Однако при компиляции отладочной версии нужно исключить из работы фрагменты, предназначенные для «релиза», и наоборот.

Большинство промышленных компилируемых языков программирования поддерживают для подобных случаев специальные конструкции, называемые **директивами условной компиляции**. Они позволяют выбирать, какие фрагменты кода компилировать, а какие игнорировать. Обычно отработку директив условной компиляции возлагают на макропроцессор, если он, конечно, в языке предусмотрен. Используемый нами ассемблер *gas* поддерживает условную компиляцию, или **условное ассемблирование**.

Допустим, мы написали программу, откомпилировали ее и запустили, но она завершается аварийно. Мы не можем понять, в чем причина, но считаем, что крах происходит в одном «подозрительном фрагменте». Чтобы проверить свое предположение, мы хотим непосредственно перед входом в этот фрагмент и сразу после него поставить вывод на экран соответствующих сообщений. Чтобы нам не пришлось много раз стирать эти сообщения и вставлять их снова, воспользуемся директивами условной компиляции. Выглядеть это будет примерно так, как показано на СЛАЙДЕ 10.

Вообще, *.ifdef* – это одна из многочисленных форм директивы *.if*, ее синтаксис простой. Директива *.endif* заканчивает блок кода с условной компиляцией и не имеет операндов. Некоторые формы *.if* приведены на СЛАЙДЕ 11.

Также имеет смысл отметить еще две директивы: *.elseif* и *.else*. Первая выполняется тогда, когда ветка *.if* или предыдущие *.elseif* оказались ложными. Конструкция *.elseif* позволяет избавиться от «распухания» вложенных *.if* - *.else*. Вторая – выполняется в аналогичных ситуациях, однако может быть не более одной такой ветки.

14.6 Многострочные макросы

Вернемся теперь к многострочным макросам; такие макросы генерируют не фрагмент строки, а фрагмент текста, состоящий из нескольких строк. Описание многострочного комментария также состоит в общем случае из нескольких строк, заключенных в конструкции *.macro* и *.endm*. Например, определим макрос *sum*, который записывает в память последовательность чисел (СЛАЙД 12).

Теперь макровызов *sum* 0, 5 выдаст нам следующую серию объявлений:

```
.long   0
.long   1
.long   2
.long   3
.long   4
.long   5
```

Для макроопределений можно использовать следующий синтаксис (СЛАЙД 13):

```
.macro macname
.macro macname macargs ...
```

Начинается макроопределение с имени макроса. Если наше макроопределение требует аргументом, необходимо перечислить их имена после имени макроса, разделяя запятыми или пробелами. Мы можем квалифицировать аргументы макроса, чтобы показать, чтобы при всех макровызовах нужно передавать им аргументы с непустыми значениями, используя конструкцию ‘*:req’*. Также можно указать, что макросу передается переменное число аргументов через конструкцию ‘*:vararg’*. Мы можем задать значение по умолчанию для любого аргумента макроса с помощью конструкции ‘*=deflt’*.

Очевидно, что мы не можем объявить два макроса с одинаковыми именами, пока это не станет объектом директивы *.purgem*.

Следующие макросы являются правильными (СЛАЙД 14).

```
.macro comm /* макрос без аргументов */

.macro plus1 p, p1
.macro plus1 p p1
```

Оба оператора начинают определение макроса *plus1*, который имеет два аргумента, внутри определения макроса *p* и *p1* обозначают значения аргументов.

```
.macro reserve_str p1=0 p2
```

Начинает определение макроса *reserve_str* с двумя аргументами. Первый имеет значение по умолчанию (равное 0), а второй – нет. После завершения определения мы можем вызвать макрос и как *reserve_str A, B* (с *p1* означающим *A* и *p2* означающим *B*), и как *reserve_str* , *B* (\p1 принимает значение по умолчанию, в данном случае 0, *p*2 означает *B*).

```
.macro m p1:req, p2=0, p3:vararg
```

Начинает определение макроса его имя *m*. Макрос принимает, по крайней мере, три аргумента. Первый аргумент должен быть указан всегда, в отличие от второго, который может принимать значение по умолчанию. Третий формальный аргумент будет принимать все оставшиеся фактические параметры во время макровызова.

Кроме того, когда мы вызываем макрос, можно задать значение аргумента как позицией, так и ключевым словом. Например, *sum* 7, 17 эквивалентно *sum to*=17, *from*=7.

В некоторых случаях в зависимости от целевой архитектуры символы,

использующиеся для аргументов макросов, могут представлять не совсем то, чем они являются на самом деле. Например, если символ двоеточия (:) можно использовать в идентификаторах, и на ассемблерах в ряде целевых архитектур этим символом обозначается конец метки, то у кода, заменяющего параметр макроса, не будет сведений об этом, и он не сможет распознать полную конструкцию с двоеточием как идентификатор. Например, следующее макроопределение (СЛАЙД 15):

```
    .macro label l
\l:
    .endm
```

не будет работать так, как ожидается. Вызов '*label kuku*' не создаст в коде метку *kuku*. Он просто в результирующий ассемблерный код вставит метку *\l*. При этом, вероятно, будет получено сообщение об ошибке.

Похожая проблема возможно с символом точки (.). Она нередко встречается при написании кодов операций и идентификаторов. Например, следующий макрос предназначен для создания кода операции из базового имени и длины (СЛАЙД 16):

```
.macro opcode base length
\base.\length
.endm
```

Вызов '*opcode store l*' не создаст инструкцию *store.l*, вместо этого будет получено сообщение об ошибке при ассемблировании конструкции *\base.\length*.

Для решения этой и других подобных проблем существует несколько обходных путей (СЛАЙД 17):

1. Вставка пробелов. Возможно, это самый простой путь.

```
    .macro label l
\l :
    .endm
```

2. Использование конструкции '\()'. Эту строку можно использовать для отделения конца аргумента макроса от последующего текста. Например:

```
.macro opcode base length
\base\().\length
.endm
```

3. Использование альтернативного синтаксиса. В этом режиме в качестве разделителя используется символ амперсанда:

```
.altmacro
.macro label l
l&:
.endm
```

Как было указано выше, директива *.endm* завершает макроопределение. Однако в некоторых случаях требуется досрочное завершение макроса. Здесь уместна директива *.exitm*. СЛАЙД 18.

Так же *gas* хранит число выполненных макросов в псевдо-переменной; мы можем вывести это число при помощи *\@*, но только внутри определения макроса.

Более полный пример «Программа, считывающая строку и отображающая приветствие пользователя» приведен на СЛАЙДАХ 19-21.

Продемонстрирована здесь одна еще не обсуждавшаяся особенность. В частности *.lcomm* используется для определения байтового пространства. Название

переменной следует за ключевым словом *.lcomm*, после чего через запятую указывается величина резервируемого пространства. Синтаксис (СЛАЙД 22):

```
.lcomm varname, size
```

14.7 Встроенный ассемблер

В наше время редкий программист использует на практике язык ассемблера. Языки высокого уровня, в том числе, язык С, поддерживаются почти на всех архитектурах и обеспечивают высокую производительность программ. Для тех немногих случаев, когда требуется встроить в программу ассемблерные инструкции, в *gcc* предусмотрены специальные средства, учитывающие особенности конкретной архитектуры.

Встроенными ассемблерными инструкциями следует пользоваться с осторожностью, т.к. они являются системно-зависимыми. Например, программу с инструкциями x86 не удастся собрать на ARM-системах. В то же время такие инструкции позволяют напрямую обращаться к аппаратуре, вследствие чего код может выполняться немного быстрее.

В программы такие инструкции встраиваются с помощью функции *asm*(). Например, для платформы x86 команда (СЛАЙД 23):

```
asm("fsin" : "=t" (answer) : "0" (angle));
```

является эквивалентом следующему вызову:

```
answer = sin(angle);
```

В отличие от ассемблерных инструкций функция *asm*() позволяет указать входные и выходные операнды, используя синтаксис языка С.

Ассемблерные инструкции используются при написании кода ОС. Например, файл */usr/include/asm/io.h* содержит объявления команд, осуществляющих прямой доступ к портам ввода-вывода. Еще один исходный файл Linux - */usr/src/linux/arch/i386/kernel/process.s*, в котором с помощью инструкции *hlt* реализуется пустой цикл ожидания.

Общий шаблон встроенной ассемблерной инструкции:

```
asm (ассемблерный шаблон
     : выходные операнды                    (не обязательные)
     : входные операнды                     (не обязательные)
     : список «затираемых» регистров        (не обязательный)
);
```

Ассемблерный шаблон состоит из ассемблерных инструкций. Входные операнды являются выражениями на языке С, которые служат в качестве входных параметров для инструкций. Выходные операнды являются выражениями на языке С, в которые будет выполнен вывод ассемблерных инструкций.

```
asm("movl %%cr3, %0\n" :"=r"(cr3val));
A     %eax
b     %ebx
c     %ecx
d     %edx
S     %esi
D     %edi
```

Если операнды находятся в памяти, то любые операции, выполняемые с ними, будут выполняться непосредственно в памяти в отличие от регистровых ограничений, когда сначала значение сохраняется в регистре с целью модификации, а затем записывается обратно в память. Регистровые ограничения обычно используются, если они абсолютно

необходимы для выполнения инструкции или они существенно ускоряют процесс вычислений. Ограничения памяти могут наиболее эффективно использоваться в случаях, когда переменную языка С необходимо изменить внутри конструкции *asm,* и мы не хотим использовать регистр для хранения значения переменной. В следующем примере значение *idtr* сохраняется в переменной памяти *loc* (СЛАЙД 24):

```
asm("sidt %0\n" : :"m"(loc));
```

В некоторых случаях одна переменная может служить в качестве входного и выходного операнда. Для этого в конструкции *asm* используются ограничения совпадения.

```
asm("incl %0" :"=a"(var):"0"(var));
```

В приведенном примере ограничения совпадения регистр %*eax* используется в качестве входного и выходного операнда. Значение переменной *var* считывается в %*eax*, а модифицированное после инкремента значение %eax снова сохраняется в переменной var. Здесь цифра 0 указывает на то же ограничение, что и нулевой выходной операнд, т.е. он указывает, что выходное значение переменой *var* должно быть сохранено только в регистре %*eax*. Данное ограничение может быть использовано в случаях:

- Если входной операнд считывается из переменной, или переменная модифицируется, и полученное значение записывается обратно в ту же самую переменную.
- Если не требуются отдельные экземпляры входных и выходных операндов.

Наиболее существенно при использовании ограничений совпадения то, что при этом более эффективно используются доступные регистры.

14.8 Примеры общего использования встроенного ассемблера

Следующие примеры иллюстрируют использование различных ограничений. Существует слишком много ограничений, чтобы дать примеры для каждого из них, поэтому приведенные примеры относятся к наиболее часто используемым типам ограничений.

Давайте сначала рассмотрим конструкцию *asm* с регистровым ограничением 'r'. Данный пример показывает, как компилятор *gcc* выбирает регистры и как он изменяет значение выходных переменных (СЛАЙД 24).

```
int main(void)
{
   int x = 10, y;

   asm ("movl %1, %%eax;
        "movl %%eax, %0;"
        :"=r"(y)       /*y - выходной операнд*/
        :"r"(x)        /*x - входной операнд*/
        :"%eax");      /*%eax - «затираемый» регистр*/
}
```

В этом примере значение *x* копируется в *y* внутри конструкции *asm*. *x* и *y* переданы в конструкцию *asm*, будучи сохраненными в регистрах. Ассемблерный листинг для данного примера выглядит следующим образом (СЛАЙД 24):

```
pushl %ebp
movl %esp,%ebp
subl $8,%esp
```

```
movl $10,-4(%ebp)
movl -4(%ebp),%edx /*значение x=10 сохранено в регистре %edx*/
#APP    /*начало конструкции asm*/
movl %edx, %eax   /*значение x переслано в регистр %eax*/
movl %eax, %edx   /*значение y помещено в регистр %edx*/

#NO_APP /*конец конструкции asm*/
movl %edx,-8(%ebp) /* y в стеке заменено значением в %edx*/
```

Компилятор *gcc* может выбрать любой регистр в случае использования регистрового ограничения "r". В приведенном примере он выбрал регистр %*edx* для сохранения *x*. После чтения значения *x* в регистре %*edx* компилятор выбрал тот же самый регистр для *y*.

Поскольку переменная *y* указана в секции входных операндов, то модифицированное в регистре %*edx* значение сохраняется в -8(%*ebp*) - местонахождении переменной y в стеке. Если бы переменная *y* была указана в секции входных операндов, то значение переменной *y* в стеке не было бы обновлено, даже если бы оно было обновлено во временном регистровом хранилище переменной *y* (регистр %*edx*). А поскольку регистр %*eax* указан в списке «затираемых» регистров, компилятор *gcc* не использует данный регистр для хранения данных в других местах.

Как входной операнд *x*, так и выходной операнд *y* помещены в один и тот же регистр %*edx*, в предположении, что входные операнды используются до получения выходных значений. Заметим, что при наличии большого количества инструкций это предположение может быть неправильным. Чтобы гарантировать, что входные и выходные операнды помещены в различные регистры, можно указать модификатор ограничения *&*. Ниже приведен тот же пример с добавленным модификатором ограничения (СЛАЙД 25).

```
int main(void)
{
  int x = 10, y;

  asm("movl %1, %%eax;
      "movl %%eax, %0;"
        :"=&r"(y)  /*y - выходной операнд, отметим использование
                         модификатора ограничения &*/
        :"r"(x)    /*x - входной операнд*/
        :"%eax");  /*%eax - «затираемый» регистр*/
}
```

Далее показан ассемблерный листинг для данного примера, из которого ясно следует, что переменные *x* и *y* сохранены в различных регистрах в конструкции *asm*.

```
main:
    pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
    movl $10,-4(%ebp)
    movl -4(%ebp),%ecx  /*x, входной операнд находится в %ecx*/
#APP
    movl %ecx, %eax
    movl %eax, %edx    /* y, выходной операнд находится в %edx*/

#NO_APP
    movl %edx,-8(%ebp)
```

Давайте теперь рассмотрим, как указать конкретные регистры в качестве ограничений для операндов. В следующем примере инструкция *cpuid* берет входной параметр в регистре %*eax* и размещает выходные параметры в четырех регистрах: %*eax*, %*ebx*, %*ecx*, %*edx*. Входной операнд (переменная *op*) передается в конструкцию *asm* в регистре *eax*, как того требует инструкция *cpuid*. Ограничения *a*, *b*, *c* и *d* используются в выходных операндах для соответствующей передачи значений в четыре регистра (СЛАЙД 26).

```
asm ("cpuid"
        : "=a" (_eax),
          "=b" (_ebx),
          "=c" (_ecx),
          "=d" (_edx)
        : "a" (op));
```

Ниже приведен ассемблерный листинг для данной конструкции (предполагается, что переменные *_eax*, *_ebx* и т.д. сохранены в стеке):

```
  movl -20(%ebp),%eax /*сохраняем op в %eax – входной операнд*/
#APP
  cpuid
#NO_APP
   movl  %eax,-4(%ebp)     /*сохраняем  %eax  в  переменную  _eax  –
выходной операнд*/
  movl %ebx,-8(%ebp)   /*сохраняем остальные регистры в */
  movl %ecx,-12(%ebp)  /*соответствующих выходных переменных*/
  movl %edx,-16(%ebp)
```

Функция *strcpy* может быть реализована с помощью ограничений "S" и "D"следующим образом (СЛАЙД 27):

```
asm("cld\n
       rep\n
       movsb"
       : /*входные операнды отсутствуют*/
       :"S"(src), "D"(dst), "c"(count));
```

Указатель на источник данных *src* помещается в регистр %*esi* за счет использования ограничения "S", а указатель на приемник данных *dst* помещается в регистр %*edi* за счет использования ограничения "D". Значение *count* помещается в регистр %*ecx*, как того требует префикс *rep*. Здесь можно увидеть другое ограничение, использующее два регистра %*eax* и %*edx* для объединения двух 32-битных значений в 64-битное значение.

```
#define rdtscll(val) \
     __asm__ __volatile__ ("rdtsc" : "=A" (val))
```

Ассемблерный листинг выглядит следующим образом (если переменная *val* имеет размер 64 бита).

```
#APP
  rdtsc
#NO_APP
  movl %eax,-8(%ebp) /*За счет ограничения A */
  movl %edx,-4(%ebp) /*регистры %eax и %edx служат в качестве
выходных операндов*/
```

Отметим, что значения в паре регистров %*edx*:%*eax* служат в качестве 64-битного выходного операнда

Далее приведен код для системного вызова с четырьмя параметрами (СЛАЙД 28):

```
#define _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
    type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \
    { \
    long __res; \
    __asm__ volatile ("int $0x80" \
            : "=a" (__res) \
            : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
              "d" ((long)(arg3)),"S" ((long)(arg4))); \
    __syscall_return(type,__res); \
    }
```

В данном примере четыре аргумента системного вызова помещены в регистры %*ebx*, %*ecx*, %*edx* и %*esi* за счет использования ограничений *b*, *c*, *d* и *S*. Заметим, что ограничение "=a" используется в выходном операнде, поэтому возвращаемое значение системного вызова, находящееся в регистре %*eax*, помещается в переменную *__res*. За счет использования ограничения совпадения "0" в качестве первого ограничения операнда в секции входных операндов номер системного вызова *__NR_##name* помещается в регистр %*eax* и используется в качестве входного аргумента системного вызова. Тем самым, регистр %*eax* используется как входной, так и выходной регистр. Никаких отдельных регистров для этой цели не используется. Отметим также, что входной аргумент (номер системного вызова) используется до получения выходного операнда (возвращаемое значение системного вызова).

Рассмотрим следующую атомарную операцию декремента:

```
__asm__ __volatile__(
                "lock; decl %0"
                :"=m" (counter)
                :"m" (counter));
```

Ассемблерный листинг будет в этом случае выглядеть примерно следующим образом:

```
#APP
  lock
  decl -24(%ebp) /*переменная counter модифицирована в своей */
                 /* ячейке памяти */
#NO_APP.
```

Мы можем захотеть использовать регистровое ограничение для переменной *counter*. В этом случае значение переменной *counter* должно сначала быть скопировано в регистр, декрементировано, а затем помещено в память. Но тогда не будут иметь смысла блокировка и атомарность, которые ясно показывают необходимость использования ограничения операнда памяти.

Рассмотрим элементарную реализацию процедуры копирования памяти (СЛАЙД 29).

```
  asm ("movl $count, %%ecx;
        up: lodsl;
```

```
stosl;
loop up;"
:                          /*выходные операнды отсутствуют*/
:"S"(src), "D"(dst)        /*входные операнды*/
:"%ecx", "%eax" );         /*список «затираемых» регистров*/
```

Несмотря на то, что инструкция *lodsl* изменяет регистр *%eax*, инструкции *lodsl* и *stosl* используют его неявно. Регистр *%ecx* явно загружает переменную *count*. Однако компилятор GCC не будет знать об этом, пока мы не проинформируем его путем включения регистров *%eax* и *%ecx* в список «затираемых» регистров. До тех пор, пока это не сделано, компилятор *gcc* предполагает, что регистры *%eax* и *%ecx* свободны и может принять решение использовать их для хранения других данных. Отметим здесь, что регистры *%esi* и *%edi* используются конструкцией *asm* и не включены в список «затираемых» регистров, поскольку было объявлено, что конструкция *asm* будет использовать их в списке входных операндов. Последняя строка примера указывает на то, что если регистр используется внутри конструкции *asm* (неявно или явно) и не представлен ни в списке входных операндов, ни в списке выходных операндов, то этот регистр должен быть перечислен в качестве «затираемого» регистра.

На СЛАЙДЕ 30 приведен код поиска старшего значащего бита в 32-разрядном числе с использованием циклов и арифметических сдвигов. На СЛАЙДЕ 31 приведен один вариантов решения этой же задачи с использованием специальной машинной инструкции и ассемблерной вставки. В качестве аргумента командной строки обе получают количество операций поиска.

Сборка и запуск обеих программ достаточно просты. Для сравнения производительности следует включить оптимизацию *O*2.

```
$gcc -O2 bit-pos-loop.c -o bit-pos-loop
$gcc -O2 bit-pos-asm.c -o bit-pos-asm
$time ./bit-pos-loop 250000000
$time ./bit-pos-asm 250000000
```

По понятным причинам результаты могут быть разные, но «ассебмлерная» версия должна выполняться намного быстрее.

Литература и дополнительные источники к Теме 14

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. Использование GNU ассемблера as - http://www.opennet.ru/docs/RUS/gas/gas.html
5. Using as - http://sourceware.org/binutils/docs/as/index.html
6. Ассемблеры для Linux: Сравнение GAS и NASM - http://www.ibm.com/developerworks/ru/library/l-gas-nasm/
7. Ассемблер в Linux для программистов С – http://ru.wikibooks.org/wiki/Ассемблер_в_Linux_для_программистов_С