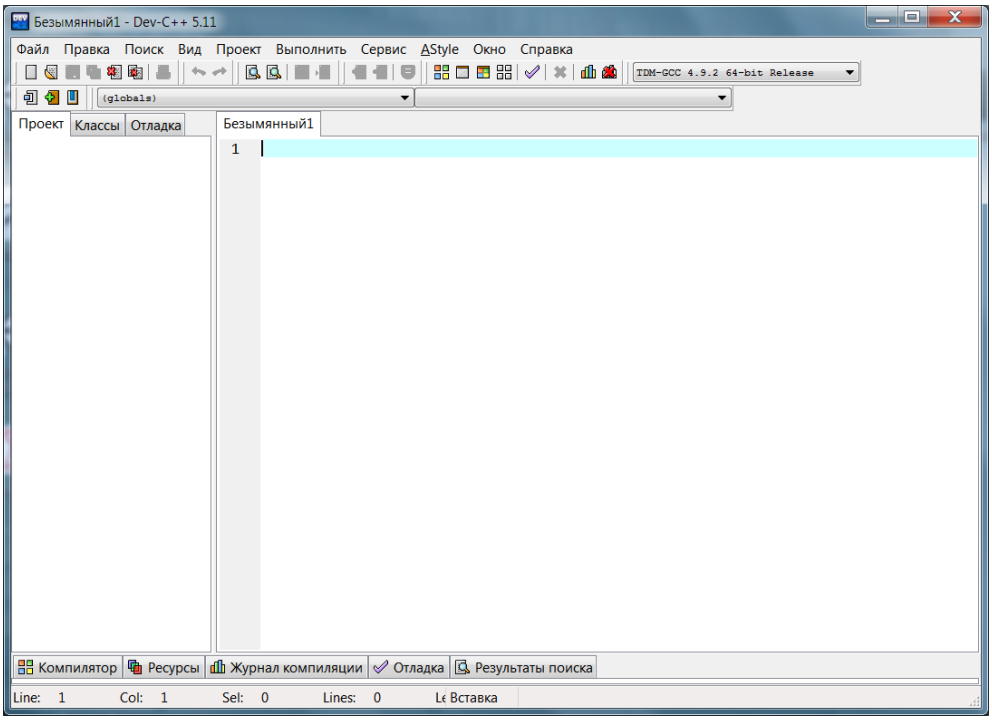
Лабораторная работа №1
Знакомство с языком С.
Ассемблер и ассемблерные вставки.

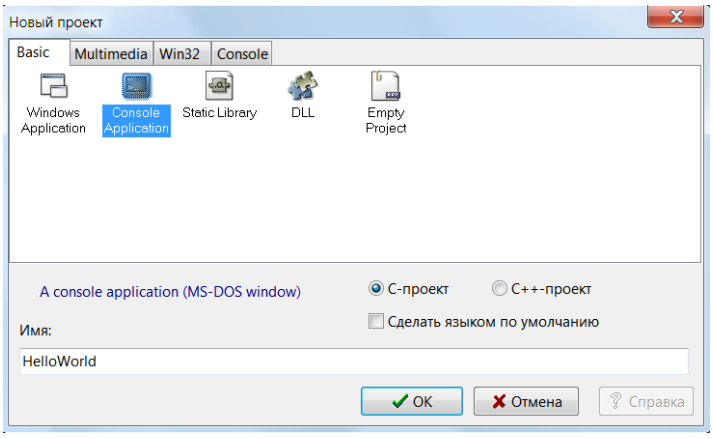
Среда Dev C++

Dev C++ IDE - среда разработки на **C** и **C++**, графическая надстройка над **GCC** компилятором. IDE бесплатна и с открытым исходным кодом.
Последнюю её версию 5.11 можно по адресу:
https://sourceforge.net/projects/orwelldevcpp/

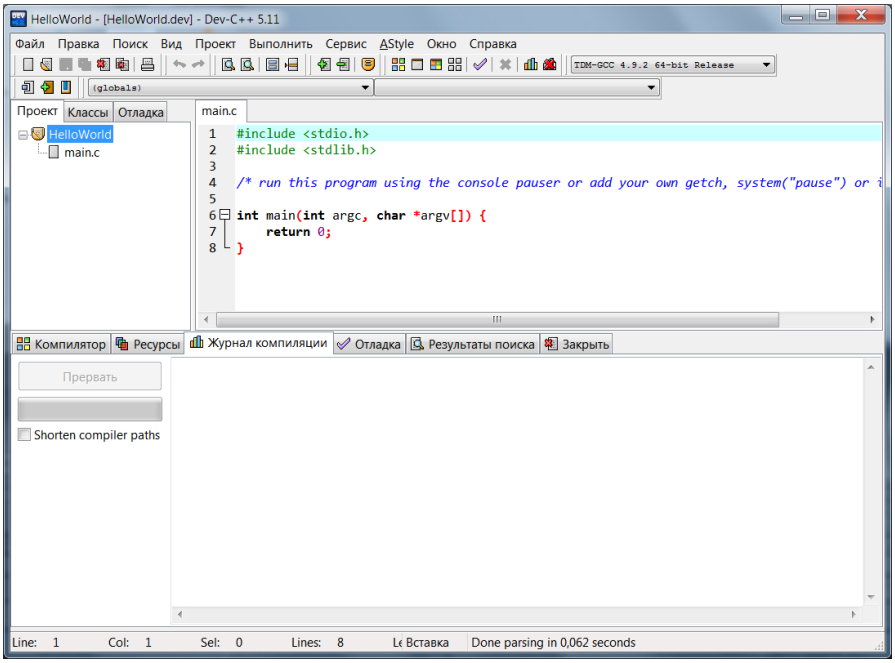
Главное окно программы представлено четырьмя панелями: сверху – главное меню и панель команд, слева - менеджер проектов, по центру – редактор кода, и снизу - информационная панель.



Чтобы создать новый проект на языке **C** необходимо в главном меню выбрать ***Файл -> Создать -> Проект***, и далее выбрать **Console Application**, выбрать в опциях **C-проект**, и далее выбрать ему подходящее имя, например HelloWorld, и нажать на **[OK]**



Далее среда сразу предложит сохранить файл проекта **HelloWorld.dev**, необходимо его сохранить в той папке, в которой предполагается хранить все исходные файлы проекта.

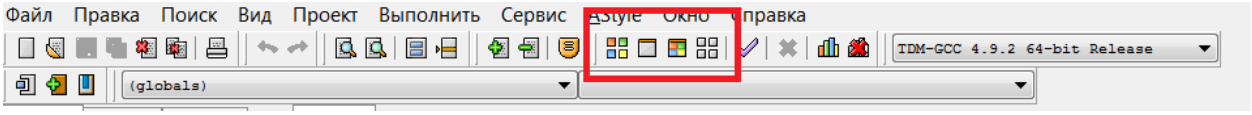


После этого в текстовом окне появится заготовка консольного проекта, а в менеджере проекта появиться файл **main.c**, в котором и находится данный код. Стоит сразу нажать **[Ctrl]+[S]** или ***Файл -> Сохранить,*** и сохранить исходный файл **main.c** в той же папке, где был сохранён **HelloWorld.dev.**

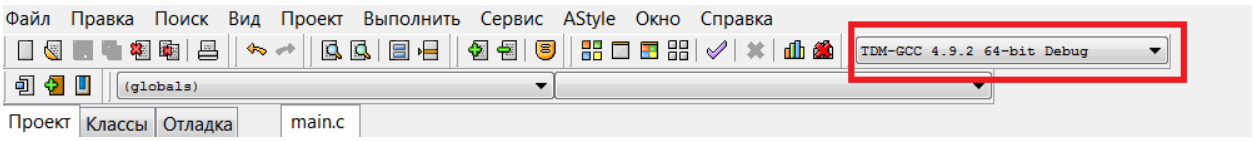
Любую из программ, написанных на **C,** перед запуском необходимо скомпилировать и только затем запустить на выполнение. Для этого в **Dev C++** используется четыре команды:

- **Скомпилировать [F9]** - простая компиляция программного кода. На данном этапе компилятор проверяет написанный код на наличие ошибок и, если все в порядке - переводит код программы в исполняемый файл *.exe.
  Если же имеются ошибки, то работа компилятора прерывается и внизу в информационном окне "Компилятор" выводятся коды ошибок, помогающих их найти и исправить.
- **Выполнить [F10]** - эта команда позволяет многократно запускать программу без повторной компиляции кода.
- **Скомпилировать и выполнить [F11]** - если необходимо сразу посмотреть выполнение программы в консоли после компиляции, то необходимо использовать данную команду.
- **Перестроить всё [F12]** – команда, позволяющая перекомпилировать не только измененные файлы, но и все другие модули, использующиеся в программе.

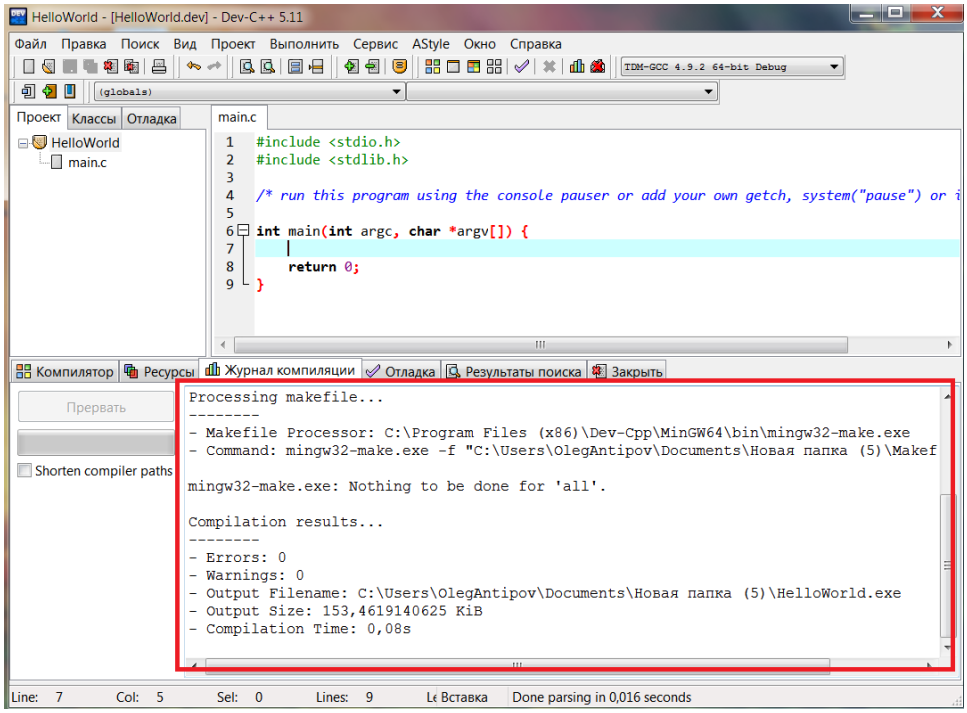
Их также можно найти на панели управления в виде кнопок:



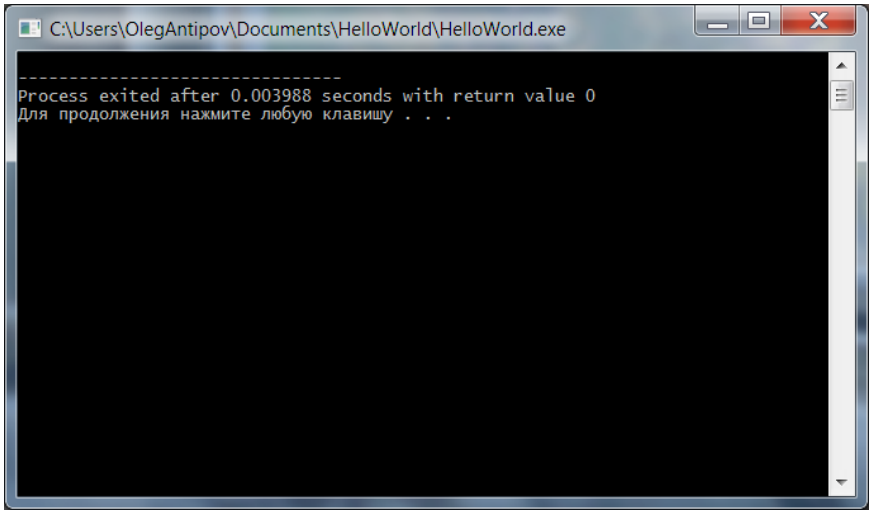
Также на этой панели можно выбрать режим работы компилятора, например, Release \ Debug \ Profile или разрядность 32-bit или 64-bit. При разработке в целях получения дополнительной отладочной информации стоит переключить компилятор в отладочный режим, выбрав Debug вариант:



После нажатия на «Скомпилировать и Выполнить» [F11] в информационном окне появляется лог компиляции, где в том числе выводятся сообщения об ошибке или предупреждениях:

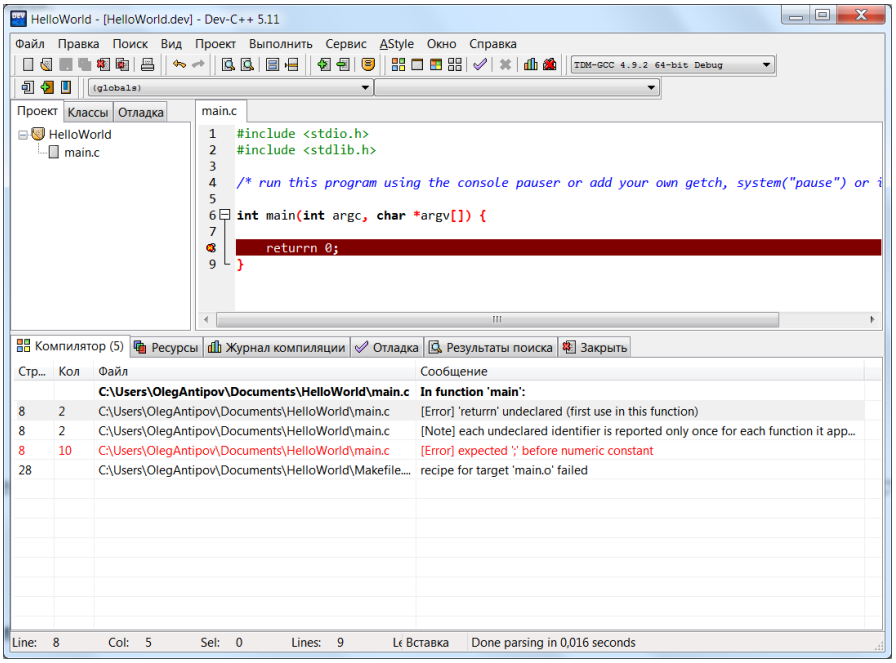


И если ошибок найдено не было, то будет выведена консоль с результатами работы программы:



Естественно никаких результатов работы выведено не будет, т.к. был запущен пустой проект.

Если же при компиляции возникла ошибка – информация о ней будет выведена во вкладке «Компилятор» на информационной панели:

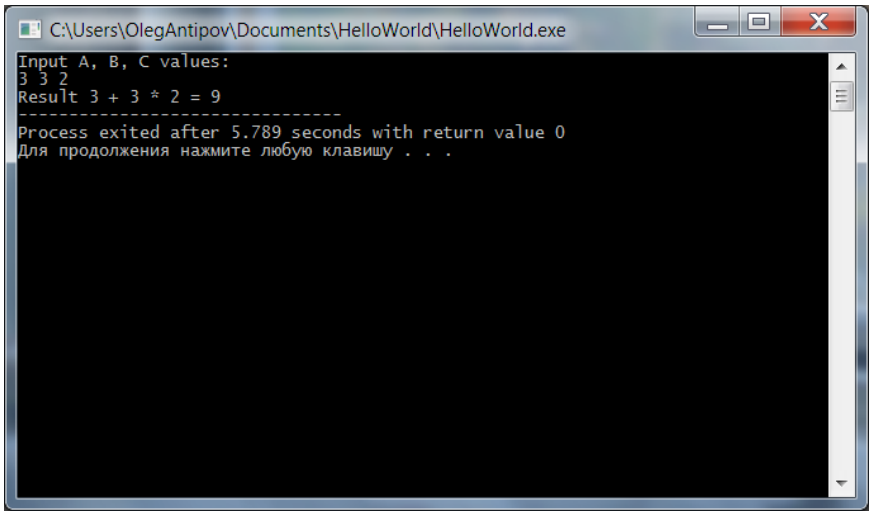


Пример программы

Напишем следующую программу на языке **C**, вычисляющая значение выражения A + B * C:

```c
#include <stdio.h>

int main()
{
    int A, B, C, res; //объявляем переменные

    printf("Input A, B, C values:\n"); //выводим на экран подсказку для ввода
    scanf("%d%d%d", &A, &B, &C); //запрашиваем значения переменных (обязательно указать знак взятия адреса &)

    res = A + B * C; //расчитываем результат операции

    printf("Result %d + %d * %d = %d", A, B, C, res); // выводим на экран полученный результат выражения

    return 0;
}
```

В результате компиляции и выполнения программы получаем в консоли следующее:



Ассемблер

Ассемблер (от англ. assembler — сборщик) — транслятор исходного текста программы, написанной на языке ассемблера, в программу на машинном языке.

Ассемблеры, как правило, специфичны для конкретной архитектуры, операционной системы и варианта синтаксиса языка. Вместе с тем существуют мультиплатформенные или вовсе универсальные (точнее, ограниченно-универсальные, потому что на языке низкого уровня нельзя написать аппаратно-независимые программы) ассемблеры, которые могут работать на разных платформах и операционных системах.

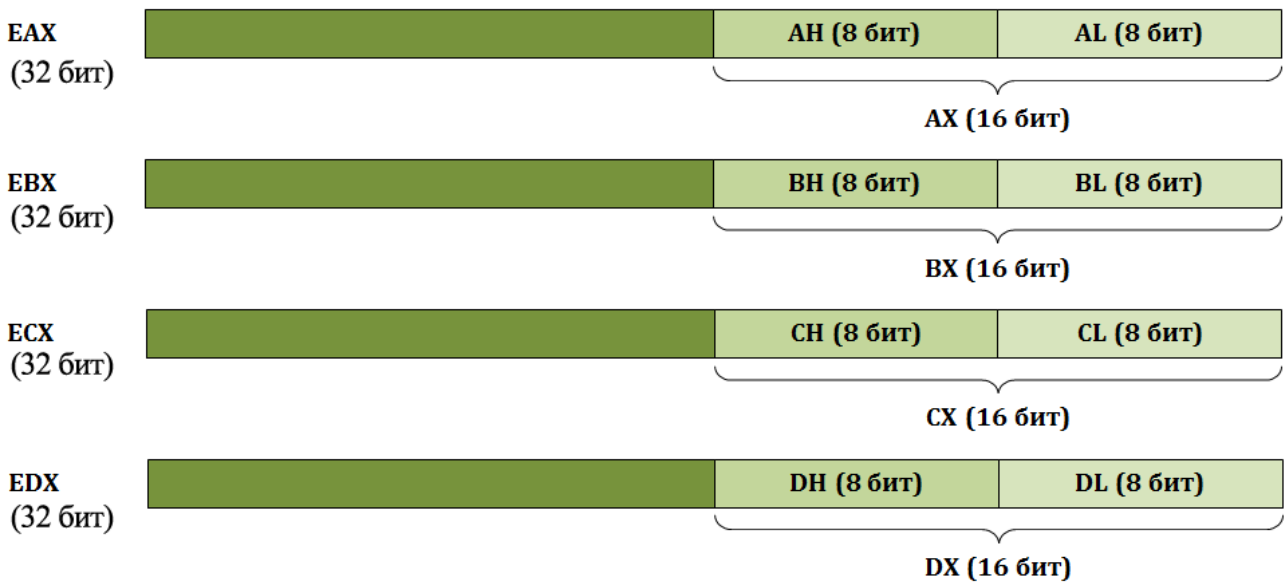
В данной лабораторной работе будут рассмотрены базовые ассемблерные арифметические операции над целыми числами, а также способ взаимодействия встраивания кода ассемблера в программу на **C**.

Регистры процессора

Регистры – это специальные ячейки памяти, расположенные непосредственно в процессоре. Работа с регистрами выполняется намного быстрее, чем с ячейками оперативной памяти, поэтому регистры активно используются как в программах на языке ассемблера, так и компиляторами языков высокого уровня.

Регистры можно разделить на регистры общего назначения, указатель команд, регистр флагов и сегментные регистры.

В данной лабораторной работе будут рассмотрены только 4 32-битных регистра общего назначения: EAX, EBX, ECX, EDX, а также обращение к их частям – 16-битные регистры AX, BX, CX, DX, и 8-битные регистры AH, AL, BH, BL , CH, CL , DH, DL.



Как видно из рисунка, регистры EAX, EBX, ECX и EDX позволяют обращаться как к младшим 16 битам (по именам AX, BX, CX и DX), так и к двум младшим байтам по отдельности (по именам AH/AL, BH/BL, CH/CL и DH/DL).

Названия регистров происходят от их назначения:

- EAX/AX/AH/AL (accumulator register) – аккумулятор;
- EBX/BX/BH/BL (base register) –регистр базы;
- ECX/CX/CH/CL (counter register) – счётчик;
- EDX/DX/DH/DL (data register) – регистр данных;

Несмотря на существующую специализацию, все регистры можно использовать в любых машинных операциях. Однако надо учитывать тот факт, что некоторые команды работают только с определёнными регистрами. Например, команды умножения и деления используют регистры EAX и EDX для хранения исходных данных и результата операции.

Команды ассемблера

Команды языка ассемблера обычно имеют 1 или 2 операнда, или не имеют операндов вообще. Во многих, хотя не во всех, случаях операнды (если их два) должны иметь одинаковый размер.

Важно: синтаксис команд бывает двух версий: AT&T и Intel. Т.к. в среде Dev C++ используется компилятор GCC, использующий синтаксис ассемблера AT&T, то далее нами будет рассматриваться именно этот формат команд.

<команда> <операнд 1 (источник)>, <операнд 2 (приемник)>;

Команда пересылки

С помощью команды пересылки можно записать в регистр значение другого регистра, константу или значение ячейки памяти, а также можно записать в ячейку памяти значение регистра или константу. Команда имеет следующий синтаксис:

mov *<операнд 1>, <операнд 2>*

По команде **mov** значение первого операнда записывается во второй операнд. Операнды должны иметь одинаковый размер. Например:

```
mov %EBX, %EAX;        # EBX -> EAX
```

Команды сложения и вычитания

Команды сложения и вычитания реализуют соответствующие арифметические операции.

add *<операнд 1>, <операнд 2>*

sub *<операнд 1>, <операнд 2>*

Команда ***add*** складывает операнды и записывает их сумму на место второго операнда.

```
add %EBX, %EAX;        # EAX + EBX -> EAX
```

Команда ***sub*** вычитает из второго операнда первый и записывает полученную разность на место второго операнда. Операнды должны иметь одинаковый размер.

```
sub %EBX, %EAX;        # EAX - EBX -> EAX
```

Команды умножения и деления

Сложение и вычитание знаковых и без знаковых чисел производятся по одним и тем же алгоритмам. Поэтому нет отдельных команд сложения и вычитания для знаковых и без знаковых чисел. А вот умножение и деление знаковых и без знаковых чисел производятся по разным алгоритмам, поэтому существуют по две команды умножения и деления. Для без знакового умножения используется команда **mul,** для знакового - **imul**:

mul *<операнд>*

imul *<операнд>*

Операнд, указываемый в команде, – это один из сомножителей. Местонахождение второго сомножителя и результата фиксировано, и в команде явно не указывается. Если операнд команды **mul** имеет размер 1 байт, то второй сомножитель берётся из регистра **AL**, а результат помещается в регистр **AX**. Например:

```
mul %BL;  # AL * BL -> AX
```

Если операнд команды **mul** имеет размер 2 байта, то второй сомножитель берётся из регистра **AX**, а результат помещается в регистровую пару **DX:AX**. Если операнд команды **mul** имеет размер 4 байта, то второй сомножитель берётся из регистра **EAX**, а результат помещается в регистровую пару **EDX:EAX**. Например:

```
mul %EBX;       # EAX * EBX -> EDX:EAX
```

Деление, как и умножение, реализуется двумя командами, предназначенными для знаковых и без знаковых чисел:

div *<операнд>*

idiv *<операнд>*

в командах указывается только один операнд – делитель. Местоположение делимого и результата для команд деления фиксировано.

Если делитель имеет размер 1 байт, то делимое берётся из регистра AX. Если делитель имеет размер 2 байта, то делимое берётся из регистровой пары DX:AX. Если же делитель имеет размер 4 байта, то делимое берётся из регистровой пары EDX:EAX.

Поскольку процессор работает с целыми числами, то в результате деления получается сразу два числа – частное и остаток. Эти два числа также помещаются в определённые регистры. Если делитель имеет размер 1 байт, то частное помещается в регистр AL, а остаток – в регистр AH. Если делитель имеет размер 2 байта, то частное помещается в регистр AX, а остаток – в регистр DX. Если же делитель имеет размер 4 байта, то частное помещается в регистр EAX, а остаток – в регистр EDX. Например:

```
div %EBX; # EDX:EAX / EBX -> EAX (остаток -> EDX)
```

Суффиксы команд

В синтаксисе AT&T для каждой команды добавляется однобуквенный суффикс, указывающий на размер операндов:

b (byte) — операнды размером в 1 байт
w (word)— операнды размером в 1 слово (2 байта)
l (long) — операнды размером в 4 байта

Т.е. чтобы использовать команду сложения для 16-битных регистров, нужно записать следующий префикс команде **mov:**

```
movw %ax, %bx;  # переместить 16-битное значение
                # из регистра ax в регистр bx
```

Как видно выше к команде **mov** добавился суффикс **w** означающий работу с 16-битными операндами. В синтаксисе AT&T все регистры записываются со знаком **%,** и команды разделяются **;** - точкой с запятой. Знак **#** означает комментарий.

Числовые константы

Числовые константы имеют следующую форму записи:

```
movb $0xFF, %al # переместить значение 0xFF (т.е. 255) в регистр AL
movb $255, %dl # переместить значение 255 в регистр DL
```

Т.е. значение числовых констант можно задавать как в 16-ричной так и в 10-й системе счисления, перед этим только необходимо установить знак доллара.

Команды расширения числа с учётом знака

В программах довольно часто возникает задача пересылки меньшего по длине значения в больший по длине регистр. В качестве примера предположим, что нам нужно загрузить 16-разрядное беззнаковое значение (например 123), в 32-разрядный регистр ECX. Самое простое решение этой задачи заключается в том, что вначале нужно обнулить регистр ECX, а затем загрузить 16-разрядное значение в регистр CX.

```
movl $0, %ecx;
movw $123, %cx;  #ecx = 0x0000007B (123)
```

А что делать, если нужно загрузить в регистр ECX отрицательное значение (например -123)? для получения правильного результата нам нужно не обнулять регистр ECX, а загрузить в него значение 0xFFFFFFFF (т.е. -1, т.к. отрицательные числа представляются в памяти компьютера в дополнительном коде), и только затем загрузить в регистр CX нужное отрицательное значение. Код будет выглядеть так:

```
movl $0xFFFFFFFF, %ecx;
movw $-123, %cx;  #ecx = 0xFFFFFF85 (-123)
```

То есть, для решения задачи, вначале нужно проанализировать знак числа и в зависимости от результата загрузить в регистр либо 0, либо -1, что довольно неудобно. Поэтому для выполнения расширения можно использовать готовые команды:

Команда **cbw (convert byte to word)** выполняет преобразование значения, находящегося в регистре **al**, до размера слова, и помещает его в **ax**, при этом свободные старшие биты **ax** заполняются знаковым битом **al (al -> ax)**. Результат выполнения для отрицательного и положительного числа:

```
movb $-123, %al;  #al = 0xFF85
cbw;  #ax = 0xFFFFFF85 (123)

movb $123, %al;  #al = 0x007B
cbw;  #ax = 0x0000007B (123)
```

Команда **cwd (convert word to double)** выполняет преобразование значения, находящегося в регистре **ax**, до размера двойного слова, и помещает его в пару регистров **dx:ax**, при этом свободные биты **dx** заполняются знаковым битом **ax**. **(ax -> dx:ax)**

```
movw $-123, %ax;  #ax = 0xFFFFFF85
cwd;  #dx:ax = 0xFFFFFFFFFFFFFF85 (-123)

movw $123, %ax;  #ax = 0x0000007B
cwd;  #dx:ax = 0x000000000000007B (123)
```

Команда **cwde (convert word to double extended)** выполняет преобразование значения, находящегося в регистре **ax**, до размера двойного слова, и помещает его в регистр **eax**, при этом свободные старшие биты **eax** заполняются знаковым битом **ax. (ax -> eax)**

```
movw $-123, %ax;  #ax = 0xFFFFFF85
cwde;  #eax = 0xFFFFFFFFFFFFFF85 (-123)

movw $123, %ax;  #ax = 0x0000007B
cwde;  #eax = 0x000000000000007B (123)
```

Команда **cdq (convert double to quadruple)** выполняет преобразование значения, находящегося в регистре **eax**, до размера учетверенного слова, и помещает его в пару регистров **edx:eax**, при этом свободные биты **edx** заполняются знаковым битом **eax**. **(eax -> edx:eax)**

```
movl $-123, %eax;  #ax = 0xFFFFFFFFFFFFFF85
cdq;  #edx:eax = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF85 (-123)

movl $123, %eax;  #ax = 0x000000000000007B
cdq;  #edx:eax = 0x00000000000000000000000000000007B (123)
```

Команда **movzx** копирует содержимое исходного операнда в больший по размеру регистр получателя данных. При этом оставшиеся неопределенными биты регистра-получателя (как правило, старшие 16 или 24 бита) сбрасываются в ноль. Эта команда используется только при работе с беззнаковыми целыми числами.

Команда **movsx (move with sign-extend)** копирует содержимое исходного операнда в больший по размеру регистр получателя данных, также как и команда **movzx**. При этом оставшиеся неопределенными биты регистра- получателя (как правило, старшие 16 или 24 бита) заполняются значением знакового бита исходного операнда. Эта команда используется только при работе со знаковыми целыми числами.

```
movw $123, %cx;  #cx = 0x0000007B
movsx %cx, %%edx;  #edx = 0x000000000000007B (123)
```

Примеры арифметических действий

Сложить два числа 15 + 7:

```
movw $7, %ax;  # записываем константу 7 в регистр AX
movw $15, %bx; # записываем константу 15 в регистр BX
addw %bx, %ax; # складываем значения в регистрах AX + BX => AX,
               # результат находится в AX = 22
```

Стоит отметить что ассемблер не учитывает регистр букв, так что можно записывать как в нижнем регистре название команд и регистров **movw** так и в верхнем **MOVW**.

Вычислить 7 – 2 * (- 4):

```
movw $2, %ax;
movw $-4, %bx;
imulw %bx;       # ax * bx => dx:ax (dx можно не учитывать,
                 # т.к. хватает размера регистра ax)
movw $7, %bx;
subw %ax, %bx;  # bx - ax => bx = 15
```

Вычислить (-7) / 3:

```
movw $-7, %ax;
cwd; # обязательно расширяем число ax => dx:ax (учитывая его знак)
movw $3, %bx;
idivw %bx;            # DX:AX / BX => AX = -2 (остаток: DX = 1)
```

Ассемблерные вставки

В программировании ассемблерной вставкой называют возможность компилятора встраивать низкоуровневый код, написанный на ассемблере, в программу, написанную на языке более высокого уровня, например, **C**. Использование ассемблерных вставок может преследовать следующие цели:

- Оптимизация: С этой целью, вручную пишется ассемблерный код, реализующий наиболее критичные в отношении производительности части алгоритма.
- Доступ к специфичным инструкциям процессора: многие процессоры поддерживают специальные инструкции, например, MMX, SSE, 3DNow.
- Системные вызовы: языки программирования высокого уровня редко предоставляют возможность делать непосредственное обращение прикладной программы к ядру операционной системы, для этих целей используется ассемблерный код.

Среда **Dev C++** использует компилятор **GCC,** который имеет следующий синтаксис для вставки ассемблерных инструкций (написанных в формате **AT&T**) непосредственно в код на **C** или **C++:**

```
asm("ассемблерный код");
```

Например:

```
asm("movl %bx, %ax;"); /*перемещает содержимое регистра bx в ax */
```

Инструкции можно группировать в одной команде **asm**, например:

```
int main()
{

    /* Сложить 10 и 20 и сохранить результат в регистре %ax */
    asm ( "movw $10, %ax;"
          "movw $20, %bx;"
          "addw %bx, %ax;"
    );

    return 0;
}
```

Однако если такую программу скомпилировать и запустить – никакого результата выведено не будет, т.к. результат вычисления только сохранился в регистре AX.

Тема управления памятью в ассемблере выходит за рамки данной лабораторной работы, поэтому мы применим т.н. расширенную ассемблерную вставку для передачи значений переменных в регистры процессора, а также считывания обратно полученного значения в обычную переменную.

Расширенная ассемблерная вставка

В расширенной ассемблерной вставке появляется возможность указать операнды для передачи значений непосредственно в регистры, извлекать их из регистров в переменные и указывать список задействованных регистров.

```
asm ( "ассемблерный код"
            : [<выходные операнды>]
            : [<входные операнды>]
            : [<список задействованных регистров>]
);
```

Если нет выходных операндов, но есть входные операнды, необходимо поместить два последовательных двоеточия перед ними.

Важно: расширенная ассемблерная вставка раскрывается в большее количество инструкций за счёт сокрытия доступа к входным и выходным операндам, чем обычная (которая переносится в ассемблерный код 1 к 1), поэтому чтобы компилятор случайно не использовал под хранение операндов регистры, в которых уже хранятся какие-то данные, их следует указать в списке задействованных регистров.

Перепишем пример программы на **C**, написанный выше для вычисления A + B * C через расширенную ассемблерную вставку:

```c
#include <stdio.h>

int main()
{
    int A, B, C, res; //объявляем переменные

    printf("Input A, B, C values:\n"); //выводим на экран подсказку для ввода
    scanf("%d%d%d", &A, &B, &C); //запрашиваем значения переменных (обязательно указать знак взятия адреса &)

    //расчитываем результат операции
    asm("movl %1, %%eax;"        /* B -> EAX */
        "movl %2, %%ebx;"        /* C -> EBX */
        "imull %%ebx;"           /* EAX * EBX -> EAX */
        "movl %3, %%ebx;"        /* A -> EBX */
        "addl %%ebx, %%eax;"     /* EAX + EBX -> EAX */
        "movl %%eax, %0;"        /* EAX -> res*/
        : "=r" (res)
        : "r" (B), "r" (C), "r" (A)
        : "%eax", "%ebx"
        );

    printf("Result %d + %d * %d = %d", A, B, C, res); // выводим на экран полученный результат выражения

    return 0;
}
```

В данном случае работа идёт с переменными типа int (4 байта) поэтому были выбраны 32-битные регистры EAX и EBX а также суффикс **l** для команд. Доступ к операндам в ассемблерном коде осуществляется через знак **%** и указания порядкового номера операнда, в данном случае res - %0, B - %1, C - %2, A - %3. Спецификатор доступа **«r»** говорит компилятору, что сохранить данный операнд можно в любом доступном регистре, кроме тех, что указаны в списке «задействованных». Для выходных операндов должен быть указан спецификатор доступа **«=r»,** сообщающий компилятору что данный операнд в режиме только для записи.

Важно: в расширенной ассемблерной вставке, по сравнению с обычной, при доступе к регистрам пишется два знака процента **%%,** обусловлено это тем, что доступ к операндам осуществляется через одинарный **%.**

В принципе нет никакой необходимости весь ассемблерный код писать в одном блоке **asm,** пример выше можно разбить построчно следующим образом:

```c
asm("movl %0, %%eax;":: "r"(B) : "%eax");          /* B -> EAX */
asm("movl %0, %%ebx;":: "r"(C) : "%eax", "%ebx");  /* C -> EBX */
asm("imull %ebx;");                                /* EAX * EBX -> EAX */
asm("movl %0, %%ebx;":: "r"(A) : "%eax", "%ebx");  /* A -> EBX */
asm("addl %ebx, %eax;");                           /* EAX + EBX -> EAX */
asm("movl %%eax, %0;": "=r" (res) : : "%eax");     /* EAX -> res */
```

Таким образом, компилятор сможет более точно указать в какой ассемблерной инструкции была допущена ошибка, ссылаясь на нужную строку.

Контрольные вопросы:

1) Что такое языки программирования «высокого» и «низкого» уровня?
2) Что такое уровни абстракции?
3) Каково назначение языка С?
4) Какие есть команды для компиляции и выполнения программы в среде Dev C++?

5) Зачем нужна функция main?
6) Что такое ассемблер?
7) Что такое регистры процессора?
8) Что такое команда пересылки и как осуществить сложение и вычитание в ассемблере?
9) Что такое команда пересылки и как производится умножение и деление в ассемблере?
10) Что такое ассемблерная вставка? Для чего она нужна и как ещё добавить?
11) Что такое расширенная ассемблерная вставка? Формат записи?
12) Формат объявления констант и переменных в языке С?
13) Что такое область видимости в языке С? Какие области видимости бывают?
14) Какие есть целочисленные типы данных в языке С?
15) Какие есть вещественные типы данных в языке С?

Задание

В соответствии с вариантом, необходимо рассчитать и вывести на экран выражение и его значение, рассчитанное с помощью ассемблерной вставки. Переменные A, B, C, D, E, F являются целочисленными и вводятся пользователем с клавиатуры.

Варианты выражений:

1. D + (A + F) - E * B * (D + C / D)
2. F / (A * B + C) – (D * C + E) / E
3. B + (A – D) / F - E * D / (C – F)
4. D / (B + C – F) + A * (C / D – E)
5. D * A / (D + A) - F / (B – E) + C
6. D / (C + B + E) + (F – B) * A / C
7. (C + F) / D – (B * E + F) * A - A
8. B * (C * E – D) / C + A * (F / D)
9. (E – A) * C - B * (C – F) - D / E
10. (B - F + E) * A + (A - D / C) * E
11. B * D / (F + D) – (B / E – A) * C
12. (D - E – B) / B * A + F * (C – D)
13. D / C * (F + D) + B * (F – A / E)
14. (E + C) * (F / A – D) * A + B / D
15. C * (A – D) - F / (B + E) + C / A
16. E * (D + F) - C / B - A * (E – C)
17. A * B / (E + F) – (C + D) * D / E
18. (D – B) / (C + A) * D - A + E - F
19. D * (E + A / C) - F * (B / F + A)

20. C / (B - E * C) + A / (F + B * D)
21. A + A / (F – E) – (B * E + D) / C
22. (C + B) / C * F + (E – D) / F * A
23. B + (C - C / F) / A + E * (E – D)
24. B * (E * F + A) - C * (F - D / E)
25. A * (C * F + D) – (D – C) / B - E
26. D * E * (C / B – C) – (E + F) / A
27. E * F / (F – C) + B * D / (A + B)
28. B / (F + D * A) - C * F / (B + E)
29. (F + B) * C - C + E / (B - D – A)
30. F + (F – B) * A * E + D / (A + C)
31. (B + E – A) * E * F - F / (C – D)
32. C + D * F / (F – E) + (A – B) * F